



UNIVERSITÀ DEGLI STUDI ROMA TRE

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Informatica

Tesi Di Laurea

Strumenti per la navigazione di mappe geografiche nella vista prospettica

Laureando

Cristiano Tofani

Matricola 445202

Relatore

Prof. Maurizio Patrignani

Anno Accademico 2014/2015

Introduzione

Molto spesso utilizziamo le mappe o ci imbattiamo nell’annoso problema di voler visualizzare un’area ben definita di una mappa su un browser web, il problema che ci troviamo davanti è che l’area visualizzata è ridotta o non corretta.

Sono diversi i servizi di visualizzazione di mappe presenti in rete, ma in nessuno di essi è possibile visualizzare la mappa con una vista prospettica. Più in particolare in nessuno è possibile visualizzare la mappa *stradale* con una prospettiva di 45 gradi; Google Maps permette di utilizzare le sue mappe *satellitari* con un’inclinazione di 45 gradi, andando oltre la semplice visualizzazione della mappa, mostrando la struttura tridimensionale della terra e degli edifici, questo, però, risulta spesso fuorviante o, in alcuni casi, anche troppo pesante per il proprio computer, che fatica a caricare la pagina o semplicemente a renderizzare correttamente la mappa.

Proprio per questo lo scopo del tirocinio è stato quello di creare un sistema che al pari dei sistemi di visualizzazione già presenti nel web, permettesse di interagire con la mappa, navigando tra le diverse aree geografiche e permettendo anche di effettuare le azioni di *zoom in* o *zoom out* per aumentare o diminuire i dettagli delle informazioni visualizzate o semplicemente la grandezza dell’area visualizzata.

In breve verrà ora riassunto il contenuto dei capitoli che seguiranno:

- **Capitolo 1:** in questo capitolo verranno analizzati e descritti i sistemi di visualizzazione di mappe più comuni e conosciuti da sviluppatori e utenti comuni, facendo anche riferimento alle banche dati, se necessario, da cui questi servizi ottengono le mappe che poi saranno mostrate sul browser web;
- **Capitolo 2:** in questo capitolo sono descritte le tecnologie e le API di riferimento che sono state utilizzate per la realizzazione del progetto;

- **Capitolo 3:** questo è la sezione in cui vengono spiegati gli obbiettivi e i requisiti del progetto;
- **Capitolo 4:** è il capitolo centrale del progetto, in cui vengono descritte nel dettaglio tutte le componenti: il server sviluppato in node.js, il client sviluppato utilizzando la libreria Three.js e gli algoritmi o i modelli matematici che hanno permesso la realizzazione del progetto;
- **Capitolo 5:** in questa parte sarà possibile vedere le immagini e il funzionamento, per quanto possibile, del progetto;
- **Conclusioni:** in questa parte vengono infine descritti gli scopi raggiunti e quali potrebbero essere gli sviluppi futuri del progetto.

Indice

Introduzione	ii
Indice	iv
Elenco delle figure	vi
1 Servizi di visualizzazione di carte geografiche	1
1.1 OpenStreetMap	1
1.2 Bing Maps	3
1.3 Google Maps	5
2 Tecnologie utilizzate	7
2.1 Node.js, un framework javascript server-side	7
2.2 WebGL: Web Graphics Library	9
2.2.1 Three.js, la libreria javascript semplificata di grafica 3D	11
2.3 Google Maps API	15
3 Requisiti per un sistema di visualizzazione di mappe in 3D	19
3.1 Specifica del problema e modello realizzativo del progetto	19
4 Progetto del sistema di visualizzazione	22
4.1 Modellazione delle informazioni	22
4.1.1 Il calcolo dello zoom adatto	23
4.1.2 Calcolo dei centri successivi per lo spostamento	25
4.2 Il processamento dei dati e la richiesta della mappa	28

4.3	Proiezione nella vista prospettica, il client Three.js	31
4.3.1	Il sistema di movimento	34
4.3.2	Il sistema di zoom	37
5	Verifiche funzionali	39
5.1	La form di richiesta	39
5.2	La proiezione su un piano prospettico	40
	Conclusioni e sviluppi futuri	45
	Bibliografia	47

Elenco delle figure

1.1	Records di modifica della mappa ad opera di utenti	3
1.2	Esempi di mappa stradale (a sinistra) e mappa aerea (a destra)	4
1.3	Esempio di visualizzazione Bird's-eye	4
1.4	Applicazione di Google Traffic sulla mappa di Roma	6
2.1	Risultato della scena precedentemente creata	15
2.2	Esempio di mappa statica di Roma	18
3.1	Esempio di mappa con angolo di visualizzazione editato nel CSS	20
3.2	Esempio della funzione tilt sul polo della Vasca Navale	21
4.1	Rappresentazione della distorsione delle aree geografiche	27
5.1	Struttura della form per la richiesta dell'area iniziale	39
5.2	Inserimento delle coordinate di Parigi e Madrid	40
5.3	Proiezione dell'area tra Parigi (in alto a destra) e Madrid (in basso a sinistra)	41
5.4	Proiezione dell'area tra Roma (in alto a sinistra) e Napoli (in basso a destra)	41
5.5	Questa è la situazione dopo una serie di spostamenti	42
5.6	La scritta dei termini e condizioni di utilizzo e il logo di Google stampati sulla mappa	42
5.7	Risultato dell'azione di zoom in sulla mappa	43
5.8	Risultato dell'azione di zoom out sulla mappa	43
5.9	Nuovo esempio di spostamento	44

Capitolo 1

Servizi di visualizzazione di carte geografiche

Per *servizio di visualizzazione di carte geografiche* si intende un sistema, disponibile in rete, che permette l'utilizzo e la consultazione di mappe geografiche. Nel corso di questo capitolo verrà illustrato il funzionamento e le caratteristiche principali dei maggiori competitor del settore.

1.1 OpenStreetMap

OpenStreetMap (OSM) è molto più di un semplice servizio di visualizzazione di mappe, volendo citare proprio la comunità che ha creato e gestisce il progetto: "OpenStreetMap is a map of the world, created by people like you and free to use under an open license" (*OpenStreetMap è una mappa del mondo, creata da persone come te e utilizzata sotto licenza libera*).[Fou15]

OSM, come si può capire dalla citazione, è un progetto sviluppato e realizzato da una comunità di persone. Fondato nel 2004 da Steve Coast con lo scopo di creare una mappa libera del mondo, nel 2006 ebbe inizio il processo per la trasformazione della società in fondazione non a scopo di lucro, da cui anche la rinominazione in **OpenStreetMaps Foundation**.

La "*mappa*" negli anni ha subito diverse modifiche e soprattutto è cresciuta grazie

alle donazioni non solo di utenti privati, ma soprattutto di enti e aziende quali ad esempio *Automotive Navigation Data*, che nel 2007 donò il database stradale completo dei Paesi Bassi e quello delle principali strade di India e Cina.

La particolarità e l'unicità di questo progetto è sicuramente la sua licenza. Il database è pubblicato secondo la *Open Database License*, la licenza che permette di condividere e adattare il database e creare opere a partire dallo stesso. Le uniche condizioni da rispettare secondo la *ODbL* sono:

- attribuire l'appartenenza del database ad ogni suo utilizzo e ad ogni utilizzo di una banca dati derivata;
- se viene pubblicato il database con una modifica rispetto all'originale o si creano nuovi database a partire da esso è obbligatorio distribuire il nuovo database secondo la licenza *ODbL*;
- anche se il database venisse redistribuito in una nuova versione che ne restringa la libertà d'utilizzo, deve rimanere disponibile una versione aperta del database priva delle restrizioni.

Allo stesso modo del database, anche i dati inseriti e caricati dagli utenti devono rispettare una licenza compatibile con la Creative Commons e gli stessi contributori devono essere registrati al progetto.

Le modifiche operate sulla mappa vengono registrate in un elenco visibile sul sito web della fondazione, come visibile nella figura 1.1.

I dati di *OpenStreetMap* sono utilizzati inoltre da uno dei principali provider di questo settore, ovvero **Mapbox**. L'utilizzo di quest'ultimo lo abbiamo inconsciamente visto diverse volte navigando su internet, su siti web come *Foursquare*, *Pinterest*, *Evernote*, *Financial Times* e *National Geographic*. [Map15]

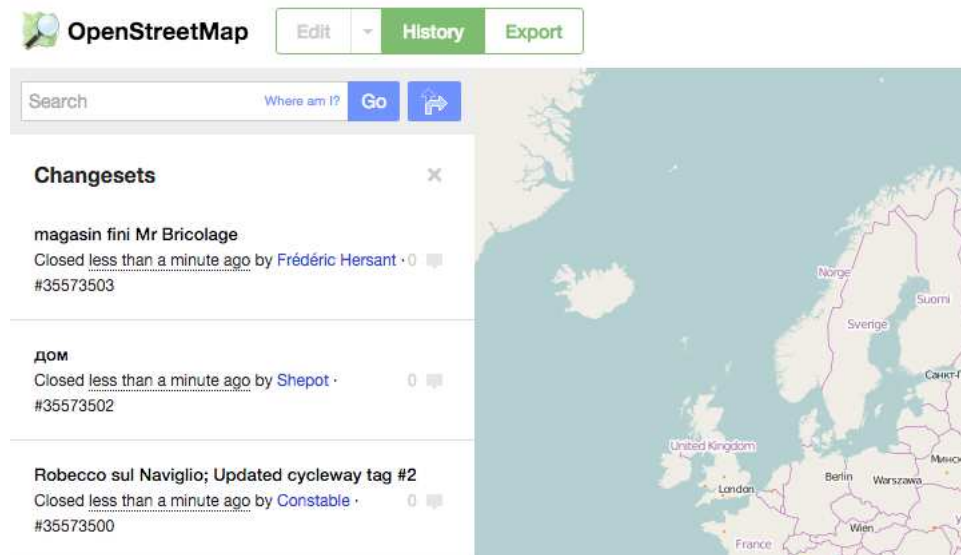


Figura 1.1: Records di modifica della mappa ad opera di utenti

1.2 Bing Maps

Dopo aver visto un servizio *open source*¹ come *OSM*, è importante vedere anche il competitor forse più "demonizzato" dal mondo delle licenze libere. Infatti in questa sezione, descriveremo il funzionamento e le caratteristiche di **Bing Maps**, il sistema di *web mapping*² creato e sviluppato dalla Microsoft.

In questo servizio a differenza di *OpenStreetMap* le mappe non sono curate da una comunità di developer ma raccolte ed elaborate in diversi modi. Anche Bing Maps come *Mapbox* attinge ai database di OSM per le mappe stradali, che rappresentano la visualizzazione di default, ovvero la mappa che possiamo vedere appena accediamo alla pagina web del portale di Microsoft. È necessario però sottolineare, che a differenza dei due servizi sopra citati - OSM e Mapbox -, quello di Microsoft offre altre possibilità di visualizzazione delle cartine, consentendone anche una vista aerea, una cosiddetta "*bird's-eye view*" (vista a volo d'uccello, che ne permette una visualizzazione prospettica di terreni e edifici), una veduta stradale che consente di vedere la strada a 360 gradi e infine troviamo la possibilità di accedere a mappe 3D in cui a prendere forma, oltre

¹Open Source[Wik15d]

²Con "Web Mapping" si intende il processo di utilizzo o offerta di mappe sul World Wide Web

ai terreni e alle montagne, ci sono anche palazzi e monumenti. Di seguito vengono illustrati alcuni esempi di visualizzazione delle mappe di Bing:

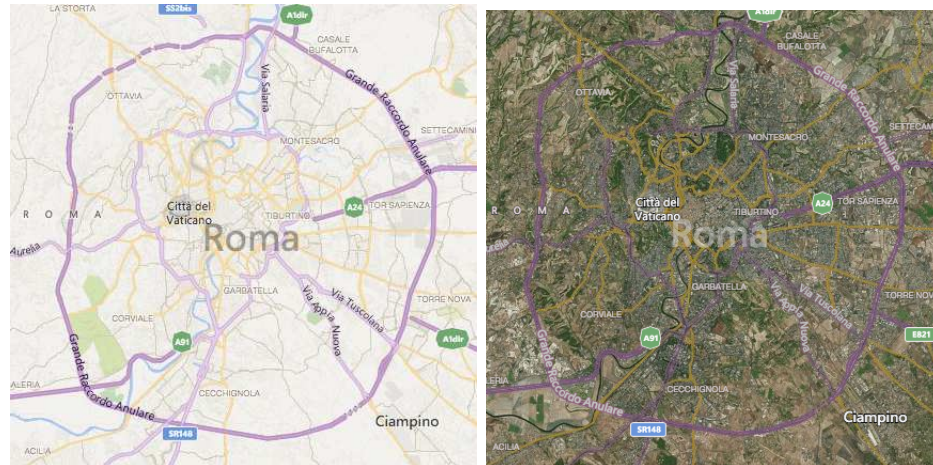


Figura 1.2: Esempi di mappa stradale (a sinistra) e mappa aerea (a destra)



Figura 1.3: Esempio di visualizzazione Bird's-eye

Una delle peculiarità probabilmente più interessanti di Bing Maps, comunque, è la presenza al suo interno di differenti "Map Apps" delle vere e proprie applicazioni interne al servizio che ne sfruttano funzionalità e informazioni e ne arricchiscono così anche

l'esperienza d'uso da parte dell'utente finale. Per fornire un esempio di questo utilizzo, si può far riferimento all'edizione del 2010 del *Tour de France* che utilizzava la mappa Bing per mostrare risultati e tracciati delle differenti tappe della nota competizione ciclistica.[Bin10]

1.3 Google Maps

Dopo aver trattato di *OpenStreetMap* e *Bing Maps*, era inevitabile parlare di quello che è, molto probabilmente, il sistema di visualizzazione cartografico più utilizzato nel mondo. Infatti in questo paragrafo, verrà descritto e analizzato *Google Maps*, che è il servizio di mappe offerto e creato da *Google Inc.*. Nato e reso disponibile nel 2005, a differenza dei due precedenti servizi non era stato ideato e pensato come servizio online, ma doveva essere un programma scritto in *C++*³ e ideato dal cofondatore di *Google* Lars Rasmussen e suo fratello Jens Eilstrup Rasmussen, successivamente il progetto venne convertito e rilasciato online come tutt'oggi lo conosciamo.

Le funzionalità di visualizzazione sono molto simili ai due servizi prima descritti. Infatti anche *Google Maps* come *Bing Maps*, offre agli utenti la possibilità di utilizzare e consultare mappe con vista satellitare, stradale o ibrida⁴. Negli anni *Google Maps* ha esteso molto le sue funzionalità e le sue caratteristiche. Nel tempo, infatti la società di Mountain View⁵ ha unito molti dei suoi progetti o prodotto *ad hoc* servizi aggiuntivi, per aggiungerli al proprio servizio di mappe. Di seguito verranno elencati i principali servizi aggiuntivi offerti da *Google Maps*:

- **Google Traffic:** è un servizio disponibile in oltre di 50 paesi, che permette all'utente di conoscere la situazione del traffico in tempo reale;
- **Google Transit:** è il sistema che permette all'utente la pianificazione di un itinerario utilizzando esclusivamente i mezzi pubblici partito nel 2007, ad oggi serve centinaia di città in tutto il mondo;

³Il C++ è un linguaggio di programmazione orientato agli oggetti[Wik15b]

⁴La visualizzazione ibrida permette all'utente di vedere il terreno con vista satellitare avendo anche il dettaglio delle strade

⁵La sede principale di *Google Inc.* si trova a Mountain View in California

- **Google Places:** attraverso questo servizio si può segnare sulla mappa una qualsiasi attività presente sul territorio (negozi, ristoranti, studi, ambulatori). I luoghi segnalati, verranno memorizzati e sarà disponibile una scheda dell'attività sotto la barra di ricerca ogni volta che selezioneremo il luogo sulla mappa;
- **Google Street View:** è probabilmente la caratteristica più conosciuta del sistema Maps. Il suo funzionamento permette di poter percorrere le strade di numerose città in tutto il mondo, immedesimandosi nella macchina di google che periodicamente percorre le strade per aggiornare le immagini.

Ho introdotto questo sistema di visualizzazione descrivendo come fosse, probabilmente, il più utilizzato in tutto il mondo. È importante, infatti, citare alcune delle applicazioni o società che utilizzano le API⁶ di *Google Maps*. Infatti tra di esse spiccano i nomi di *Whatsapp*, *Airbnb*, *Runtastic* o il *New York Times*[Inc15a]

Entriamo, dunque, nel mezzo del mio progetto, perchè per realizzare il mio sistema di visualizzazione di mappe ho utilizzato proprio le API fornite da *Google*, che avrò modo di illustrare maggiormente nel capitolo successivo.

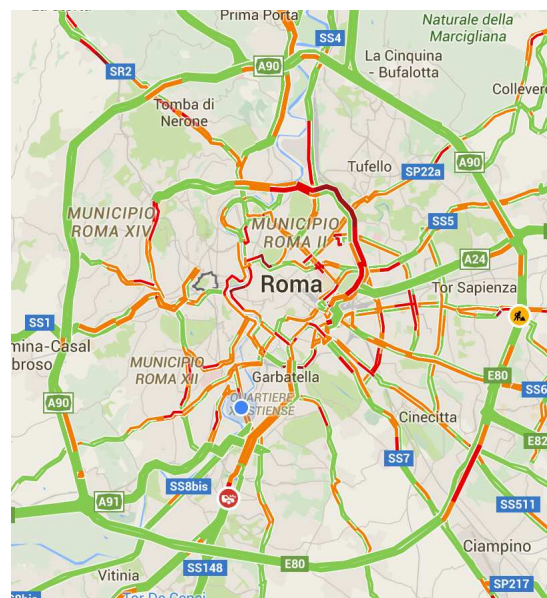


Figura 1.4: Applicazione di Google Traffic sulla mappa di Roma

⁶Application Programming Interface[Wik15a]

Capitolo 2

Tecnologie utilizzate

2.1 Node.js, un framework javascript server-side

Per creare e strutturare una web app è necessario, molto spesso, seguire paradigmi e framework differenti tra di loro. Non è, in alcuni casi, sufficiente conoscere un linguaggio unico per creare la nostra applicazione. Se si pensa alla struttura di un tradizionale sito web, occorre dunque conoscere svariati linguaggi o metalinguaggi di programmazione come Html, Javascript e CSS per il solo sviluppo *front end*¹ e di uno tra quelli più tradizionali come PHP, Pearl, Java e molti altri, rivolti invece allo sviluppo *back end*². Molto spesso è, inoltre, necessario conoscere anche linguaggi come PL/SQL che consentono la gestione e gli accessi del database.

Node.js ha la sua praticità proprio in questo, ovvero la possibilità di unire in una sola figura i due profili prima descritti di sviluppatore *back end* e *front end*, perchè le conoscenze di un web designer esperto di *Javascript* possono essere applicate senza problemi al framework. Il vero e proprio punto di forza di *Node.js* rimane la sua scalabilità, ovvero la possibilità di poterlo applicare a qualunque tipo di progetto, da uno più grande e strutturato fino ad uno più piccolo e semplice.

Analizzando la struttura di questo framework, possiamo vedere come alla sua base si trova il motore **Javascript V8** di **Google**. V8 è un motore di interpretazione,

¹Si intende con sviluppo front end la parte di programmazione rivolta al client, ovvero la componente grafica del sito web

²Si intende con sviluppo back end la programmazione della componente logica del sito web o della applicazione web, ovvero lo sviluppo rivolto al lato server

sviluppato da *Google*, originariamente per poter leggere e compilare codici Javascript all'interno del proprio browser proprietario³, però grazie alle sue altissime performance si è permesso anche di eseguirlo stand alone sui server. Il *Javascript V8 engine* ha nella diretta compilazione in linguaggio macchina di Javascript la sua maggiore caratteristica, ed è proprio questa possibilità che lo rende altamente performante. Un'altra importante proprietà è la portabilità di V8, che unito al suo stato *open source*, rende V8, ed indirettamente *Node.js*, stabile, sicuro ed eseguibile su qualsiasi sistema operativo.

Un'ulteriore importante caratteristica di questo framework, è la sua semplicità. Con poche righe è possibile creare la struttura base del server. Questo è un piccolo esempio di creazione del server in Node.js:

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

Un'altra caratteristica altrettanto importante è il suo carattere asincrono. La programmazione in *Javascript* permette, rispetto ad altri linguaggi tradizionali, la possibilità di essere affrontata in maniera asincrona.

Prima di vedere nel dettaglio cosa sia una programmazione asincrona, è bene analizzare come un tradizionale server **PHP** affronta e gestisce le richieste. In genere, in una web app, ci sono uno o più server in ascolto su una porta specifica in attesa di una richiesta. Nel momento in cui questa arriva viene attivato un nuovo processo sul server o un thread che fa partire l'elaborazione dell'interrogazione. Questo processo molto spesso deve contattare un altro server di elaborazione esterna, un database o magari il file system stesso. Solamente a elaborazione conclusa, il thread verrà segnalato come disponibile e allora sarà possibile gestire nuove richieste. Se pensiamo alle milioni di richieste che provengono dai siti web e dalle applicazioni di tutto il mondo, è facile capire come non sia in tutti i casi raccomandabile e utile utilizzare un processo come questo, nonostante sia il più lineare e semplice. Proprio per questo motivo, un approccio

³Google Chrome

asincrono aiuta a gestire molto più efficientemente, le richieste che sopraggiungono al server. In questo contesto si colloca anche *Node.js*. Il server *Node.js*, grazie alla sua struttura *event-driven*, supera la struttura *multi thread* messa in evidenza prima con il riferimento al server PHP, portando l'esecuzione di più azioni su un singolo thread. Si ha dunque lo svolgimento contemporaneo di più azioni, a differenza dei sistemi sincroni dove un'operazione viene eseguita solamente alla conclusione della precedente. Di seguito vengono mostrate i due tipi di chiamata, nel primo caso sincrona, nel secondo caso asincrona.

```
\\chiamata sincrona
function getData(url)
var a = getData("www.ricevidato.it");
alert(a);

\\chiamata asincrona
function getData(url, callback){...}
getData("www.ricevidato.it", function(result){
    alert(result);
});
```

2.2 WebGL: Web Graphics Library

Per il client del mio progetto ho utilizzato una libreria Javascript dedicata alla grafica tridimensionale e bidimensionale, o meglio una API da essa derivata di cui, però, parlerò in seguito nel corso di questo paragrafo. Analizziamo ora *WebGL* che come si può evincere dal nome, **Web Graphics Library**, è una libreria grafica studiata proprio per il web. Uno dei punti di forza indiscutibili di questa *interface* è sicuramente la possibilità di essere utilizzato in differenti contesti. *WebGL* è una libreria basata su OpenGL ES 2.0[Gro15], di cui ne utilizza il linguaggio di programmazione per la gestione dello *shader*⁴, il GLSL.

⁴Il termine *shader* indica uno strumento della computer grafica 3D che generalmente è utilizzato per determinare l'aspetto finale della superficie di un oggetto

Essendo una API studiata per il web, garantisce una completa interazione con gli elementi della *canvas* di HTML5 e questo la rende anche completamente compatibile con tutte le interfacce del DOM⁵. È proprio per questo motivo che può essere utilizzata con tutti i linguaggi di programmazione compatibili con il DOM stesso: Javascript, Java, Objective C. *WebGL* è inoltre una libreria gratuita e *open source*. Infatti è gestita e curata dal Khronos Group, un consorzio di sviluppatori di grafica 3D di cui fanno parte anche i provider dei maggiori browser web quali Mozilla (Firefox), Google (Chrome), Apple (Safari), Opera (Opera) e le principali aziende produttrici di hardware video dedicato, ovvero AMD e NVIDIA. Questo fatto la rende sicura e molto efficace.

La programmazione *WebGL*, non segue lo schema tradizionale di programmazione web. Ogni progetto ha bisogno di una parte scritta in Javascript e una in GLSL. In pratica, con il codice JS, gli sviluppatori definiscono oggetti, immagini e colori presenti sulla scena, mentre attraverso il GLSL ci si occupa dei programmi *shader* che rendono possibile l'esecuzione dei colori, immagini e vettori sulla GPU. Questo è possibile perché il GLSL è un linguaggio basato sul linguaggio C e proprio per questo è un linguaggio di basso livello in grado di accedere direttamente alle informazioni della GPU, in modo tale da attivarne all'occorrenza anche l'accelerazione grafica.

Una delle caratteristiche però negative di WebGL è l'assenza di funzioni proprie per azioni abbastanza comuni nella computer grafica, ad esempio la riduzione in scala o la rotazione di oggetti. È necessario, inoltre, aggiungere un altro importante dettaglio, quello della definizione e costruzione di oggetti come cubi, sfere ed altre forme geometriche. Nella programmazione grafica base, quindi senza far riferimento ad altri tipi di librerie, per definire ad esempio una sfera occorre far riferimento ad una matrice di coordinate, ognuna delle quali definisce un vertice di un triangolo, utilizzando poi la trigonometria per calcolare i singoli punti.

Proprio per superare questi e altri problemi nello sviluppo di un progetto di grafica 3D, gli sviluppatori fanno uso di librerie appositamente semplificate per l'esecuzione di calcoli più complessi o per operazioni che potrebbero risultare ripetitive. È il caso, per esempio di **Three.js**.

⁵Document Object Model[Con15]

2.2.1 Three.js, la libreria javascript semplificata di grafica 3D

Abbiamo analizzato prima il funzionamento di *WebGL*, vedendo come fosse molto complicato, o meglio articolato, affrontare la programmazione grafica 3D senza l'utilizzo di alcuna libreria di supporto. Proprio in questo contesto si colloca *Three.js*.

Questa è una libreria basata su *WebGL*, utilizzata in almeno la metà dei progetti web di grafica 3D. Come la stessa API di riferimento, anche *Three.js* è una libreria gratuita ed open source. Originariamente venne messa in rete dal suo creatore, Ricardo Cabello che nell'aprile del 2010 pubblicò il progetto su GitHub⁶. Cabello, conosciuto in rete come Mr.doob (il suo username su GitHub), scrisse inizialmente il progetto in ActionScript[Wik14] ma alla fine convertì il tutto in Javascript, questo perchè Javascript permette di non compilare il codice prima di ogni sua esecuzione e soprattutto la sua completa indipendenza dalla piattaforma da cui lo si sta utilizzando. Negli anni sono aumentati sempre di più i contributori al progetto iniziale al punto da essere ormai più di 390 contributori alla repository online. Il compito e merito di Cabello è stato, oltre allo sviluppo iniziale dei renderer principali, anche quello di unire tra di loro le modifiche fatte al progetto dai vari contributori.

Dopo aver descritto brevemente la storia e l'importanza di questa libreria, è opportuno vederne la sua struttura e funzionamento.

La caratteristica, probabilmente, che ne segna l'assoluta praticità è sicuramente il fatto che non richiede alcun tipo di installazione, infatti una volta scaricato il progetto da GitHub sarà sufficiente includere il file *three.min.js* all'interno della nostra pagina html come da esempio:

```
<script src="directory/three.min.js"></script>
```

La struttura di uno *script* che fa uso della libreria è abbastanza standard, sarà infatti necessario dichiarare le variabili, che rappresenteranno gli oggetti presenti sulla nostra canvas⁷. In questa parte andremo, quindi, ad analizzare le componenti fondamentali di una scena 3D. Le prime tre variabili da dichiarare sono senza dubbio **camera**, **scene** e

⁶GitHub è un servizio web che utilizza il sistema di controllo di versione Git per lo sviluppo di progetti software

⁷Canvas è una estensione dell'HTML standard che permette il rendering dinamico di immagini bitmap gestibili attraverso un linguaggio di scripting[Wik15c]

renderer, questi tre elementi rappresentano l'anatomia base di un progetto di grafica 3D e definiscono rispettivamente:

- la camera, ovvero il punto di vista da cui l'utente sta guardando il nostro disegno e da cui controllerà se richiesto i movimenti e le azioni da eseguire;
- la scena sulla quale andremo a "disegnare" e aggiungere forme geometriche o gli elementi di cui abbiamo bisogno;
- il renderer rappresenta invece la componente che effettivamente "*disegna*" la nostra scena, il suo compito è quello di far materializzare le componenti sulla canvas html;

Ovviamente, oltre queste tre variabili sarà necessario all'occorrenza dichiarare la geometria dell'oggetto o degli oggetti da utilizzare, il/i materiale/i per gli oggetti, e l'oggetto in se.

Una volta dichiarate tutte le variabili di cui abbiamo bisogno il nostro script dovrebbe avere una struttura all'incirca simile alla seguente:

```
var camera, scene, renderer;
\\altre variabili necessarie
init();
animate();

function init() {
    scene = new THREE.Scene();
    camera = ...
    \\resto del codice
}

function animate(){
    requestAnimationFrame( animate );
    render();
}
```

```
function render(){
    \\azioni da eseguire
    renderer.render(scene, camera);
}
```

Le tre funzioni, sono necessarie rispettivamente per inizializzare il nostro ambiente grafico, aggiornamento e animazione degli oggetti sulla scena e disegno vero e proprio sulla scena. Questa è, ovviamente, una struttura base del nostro progetto, ogni progetto è differente, potrebbe quindi aver bisogno di altre funzioni.

Abbiamo dunque visto la definizione di una scena con il semplice comando **new THREE.Scene()**, la cosa non è altrettanto semplice per la camera poichè sarà necessario definire il tipo di camera da utilizzare e le sue specifiche. All'interno della libreria sono comunque disponibili due tipi di camere: *OrtographicCamera*, una camera che genera una visione assonometrica, e *PerspectiveCamera*, una camera che utilizza una proiezione prospettica, la più tradizionale ed usata anche nel mio progetto.

Abbiamo parlato prima della presenza anche di oggetti, geometrie e materiali. La presenza dei tre all'interno di un progetto è abbastanza correlata. Questo perchè ogni oggetto è definito a partire da una geometria e un materiale che lo plasmeranno. Vediamo dunque l'esempio di costruzione di un semplice cubo:

```
geometry = new THREE.BoxGeometry(200,200,200);
material = new THREE.MeshBasicMaterial({color: 0xHEXCOLOR, ...});
mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```

In questo esempio *geometry* rappresenta la struttura 3D del nostro cubo, di dimensioni 200 pixels per lato, *material* è, appunto, il nostro materiale che ha come attributi il colore espresso in codice esadecimale, che sarà il colore di base del nostro cubo, *mesh*, infine, è il cubo vero e proprio, costruito a partire da *geometry* e *material*. Infine ho aggiunto anche l'oggetto alla nostra scena utilizzando il metodo *add* definito in *THREE.Scene*. Alla fine il nostro script sul file *Html*, una volta definiti forma e materiali utilizzati, dovrebbe essere più o meno questo:

```
<script>

    var camera, scene, renderer,
        geometry, material, mesh;

    init();
    animate();

    function init() {
        scene = new THREE.Scene();
        camera = new THREE.PerspectiveCamera(75,
            window.innerWidth/window.innerHeight,
            1, 10000);

        camera.position.z = 1000;
        geometry = new THREE.BoxGeometry(200, 200, 200);
        material = new THREE.MeshBasicMaterial({ color: 0xff0000,
            wireframe: true });
        mesh = new THREE.Mesh(geometry, material);
        scene.add(mesh);
        renderer = new THREE.WebGLRenderer();
        renderer.setSize(window.innerWidth, window.innerHeight);
        document.body.appendChild(renderer.domElement);
    }

    function animate() {
        requestAnimationFrame(animate);
        render();
    }

    function render() {
        renderer.render(scene, camera);
    }
</script>
```

Il risultato di questo codice può essere visto nella figura 2.1

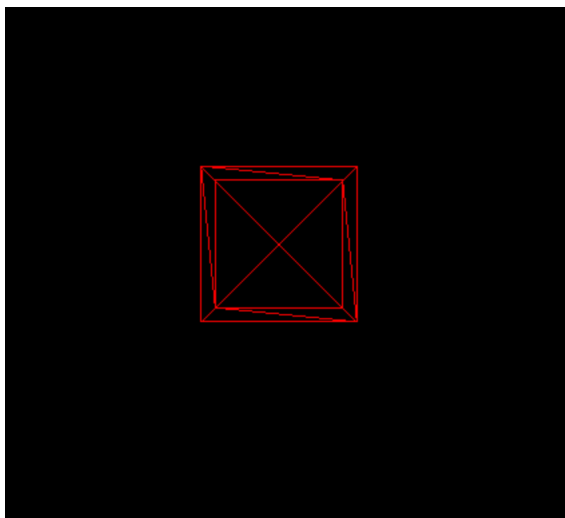


Figura 2.1: Risultato della scena precedentemente creata

2.3 Google Maps API

Nel primo capitolo di questa tesi ho descritto i più utilizzati sistemi di visualizzazione e navigazione di mappe geografiche disponibili sul web. In particolare, avevo evidenziato come *Google Maps* fosse, probabilmente, il sistema più conosciuto ed utilizzato nel mondo. Sicuramente ciò è dovuto anche alla larga scala di diffusione di smartphones, siano essi con sistema operativo Android di Google o iOS di Apple. All'inizio dello studio del contesto intorno al quale ruotava il mio progetto di tirocinio, una delle prime fasi è stata comprendere quale libreria o API sarebbe stata la prescelta per l'acquisizione delle mappe. Le alternative proposte erano principalmente due: **Leaflet.js** e **Google Maps API**. Per descriverle entrambe è necessario sottolineare che la prima è una libreria javascript basata sulle mappe di *OpenStreetMap*, mentre la seconda come si può evincere dal nome è la libreria di riferimento per gli sviluppatori che intendano utilizzare le mappe di Google. La decisione finale, ha portato all'utilizzo delle primitive offerte da Google.

Le API di Google Maps, sono delle librerie scritte in javascript arrivate alla terza versione, che permettono di interagire, adattare o aggiungere informazioni alla mappa stessa da visualizzare su di un qualsiasi sito web. Per utilizzare le API all'interno di

un progetto, è necessario iscriversi gratuitamente alla portale che Google ha destinato agli sviluppatori⁸. Una volta registratosi, lo sviluppatore che intende utilizzare un determinato servizio deve far richiesta di una *API Key*, che permetterà a Google di tracciare l'utilizzo delle mappe all'interno di un sito web. Questo è legato al fatto che l'utilizzo è sì gratuito, ma fino ad un certo limite di richieste giornaliere, il piano standard consente la richiesta di 25000 mappe giornaliere per un massimo di 90 giorni consecutivi, in eccesso Google chiede il pagamento di 0,50 dollari ogni altre 1000 richieste. Queste librerie sono disponibili per differenti target. Accedendo alla documentazione[Inc15b], è possibile vedere che la stessa è divisa in quattro principali sezioni: **Android**, **iOS**, **Web** e **Web Services**. Le prime due sezioni riguardano lo sviluppo di applicazioni per i dispositivi mobili con rispettivamente il sistema della stessa casa di Mountain View e quello di Apple. Le altre due sezioni invece riguardano le altrettante componenti di un sito web, perchè con *Web* si intendono tutte quelle operazioni atte a mostrare la mappa sul client, ovvero il nostro browser web. Con *Web Services* si vuole intendere tutte le restanti operazioni di elaborazione che ci servono nel nostro backend, come riportato sulla documentazione ufficiale potremo dunque: accedere al *Geocode*, ovvero tutte quelle operazioni di conversione tra coordinate geografiche e indirizzi veri e propri; utilizzare servizi di *Geolocalizzazione*, i servizi di posizionamento fruibili da un cellulare con sensore GPS o tramite i nodi WiFi; l'altitudine di ogni posizione della terra, Google ha infatti all'interno dei propri database le informazioni di altitudine di ogni zona della terra (informazioni visibili anche sul portale maps nella vista satellitare prospettica). Quelle elencate sono solo alcune delle principali funzioni disponibili per i Web Services.

Per importare le API all'interno del nostro progetto sarà sufficiente aggiungere un tag "script" all'interno del file HTML, come nell'esempio che segue.

```
<script src="https://maps.googleapis.com/maps/api/js?key=API_KEY
      &callback=initMap" async defer></script>
```

Nell'esempio richiamiamo una funzione, `initMap`, che serve ad inizializzare la nostra mappa. La funzione, prima citata può essere scritta nel modo seguente:

```
var map;
function initMap() {
```

⁸developers.google.com

```
map = new google.maps.Map(document.getElementById('map'), {  
    center: {lat: -34.397, lng: 150.644},  
    zoom: 8  
});  
}
```

Nella funzione viene creata una mappa con centro nel punto di latitudine -34.397 e longitudine 150.644 e zoom pari a 8. La definizione dello zoom la vedremo più avanti, in quanto è stato uno degli argomenti di studio iniziali del mio progetto.

La natura del mio progetto non era però probabilmente adatta per l'impiego di una mappa già interattiva come quella offerta da Google all'interno del suo servizio. Dovendo lavorare su di un piano tridimensionale, disegnato con Three.js, non era completamente funzionale far interagire tra loro due sistemi già interattivi. Per questo motivo ho deciso di utilizzare un altro servizio più semplificato messo a disposizione da Google. Sto parlando delle Google Static Maps. Le mappe statiche offerte da google sono delle immagini definite in base alla richiesta da noi avanzata. Infatti non c'è bisogno di alcun codice definito o particolare per richiedere la mappa. Questa è disponibile grazie ad un url di richiesta, una vera e propria query, a cui vengono concatenati i parametri necessari: centro, che può essere definito con coordinate polari o con un indirizzo testuale; la grandezza dell'immagine, definita come *LARGHEZZA*x*ALTEZZA* e infine il livello di zoom desiderato. All'url è possibile attaccare anche altri parametri nel caso volessimo modificare lo stile della mappa, i colori, se volessimo aggiungere dei fermaposto sulla nostra tile o se volessimo disegnare un tracciato su di essa. Un esempi di url di richiesta può essere questo:

```
https://maps.googleapis.com/maps/api/staticmap?center=Roma,IT  
&zoom=11&size=640x640
```

In questo url troveremo una mappa centrata in Roma di zoom 11 e grandezza 640x640 pixels, che è la grandezza massima che Google consente di utilizzare nelle API gratuite. Il risultato dell'url può essere visto nella figura 2.2

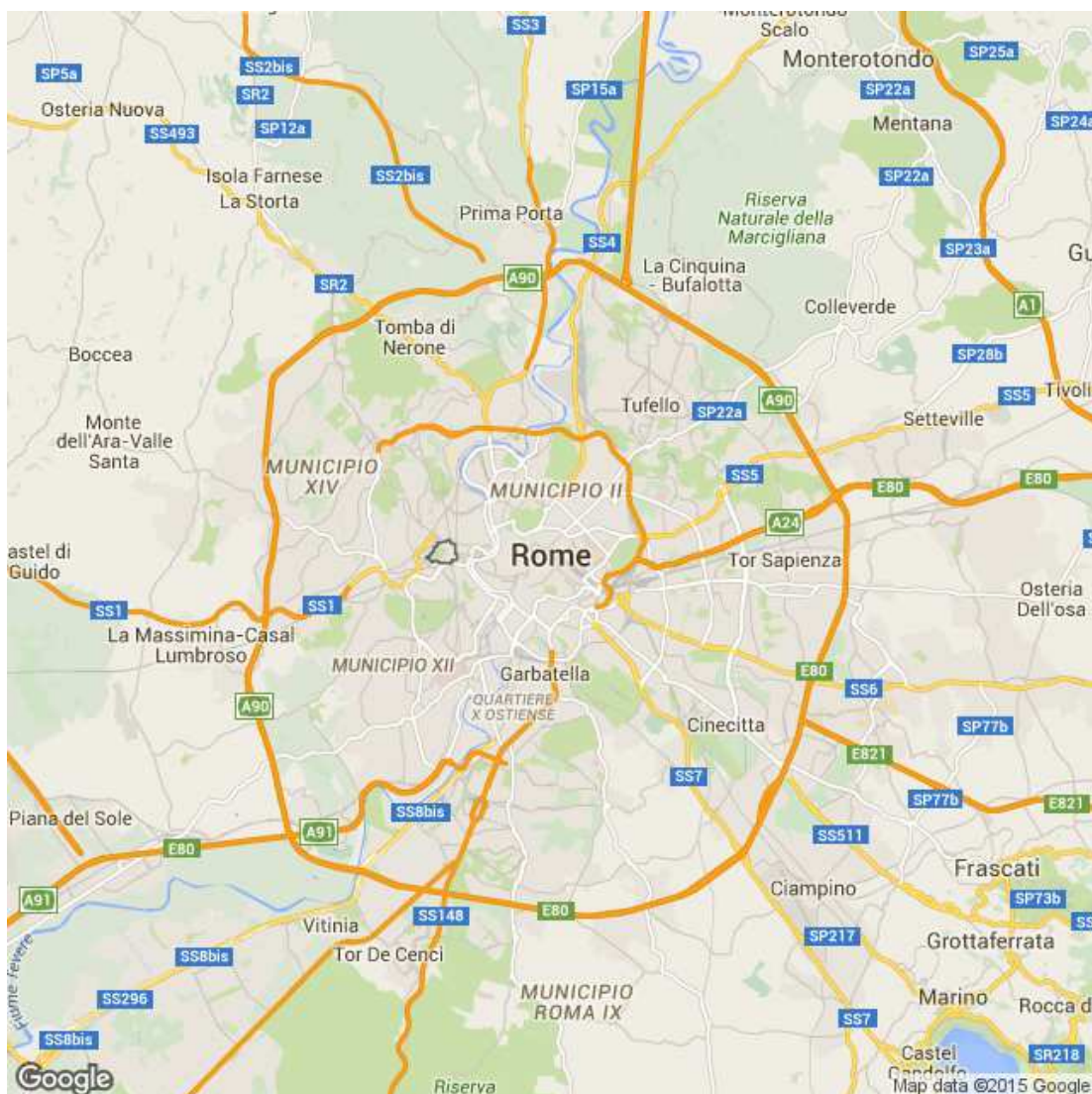


Figura 2.2: Esempio di mappa statica di Roma

Capitolo 3

Requisiti per un sistema di visualizzazione di mappe in 3D

3.1 Specifica del problema e modello realizzativo del progetto

Nel momento in cui mi trovo a scrivere e parlare di questo mio progetto, milioni di utenti stanno visualizzando in giro per il mondo una mappa, che sia essa su Google, Bing o proveniente dal database di OSM. Molti sviluppatori, allo stesso tempo, stanno strutturando al meglio un qualsiasi generico sistema che, utilizzando le stesse mappe, permetta di collocare su di esse oggetti, informazioni e molti altri dati.

Uno dei problemi forse maggiori è la poca flessibilità offerta da questi servizi. Infatti, sfruttando ad esempio alcune funzioni presenti in CSS 3, è possibile prendere la stessa mappa interattiva di Google, inserirla all'interno del body della pagina HTML, ed ottenere una vista semi-prospettica. Il risultato però non appare altrettanto nitido e pulito come accade con una canvas tridimensionale. La mappa, infatti, in questo contesto verrebbe schiacciata in altezza più che stampata lungo un ipotetico terzo asse di riferimento. Proprio per questo motivo, creare un sistema che mostri le mappe del mondo in una vista prospettica, permette la consultazione delle stesse e, ad esempio, un loro utilizzo più attivo, rispetto ad una semplice visualizzazione. Come si può vedere nella figura 3.1, modificando in CSS il parametro "transform" con attributo "rotateX(50deg)"

il risultato è di gran lunga lontano dalla sua idea di visualizzazione.

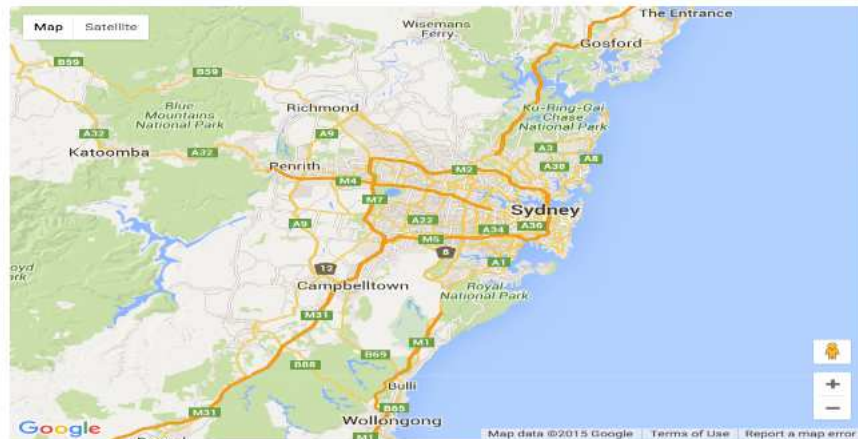


Figura 3.1: Esempio di mappa con angolo di visualizzazione editato nel CSS

Non essendo, dunque, CSS 3 la soluzione migliore da applicare, si è deciso di utilizzare Three.js per la realizzazione di questo sistema. Attraverso l'utilizzo di questa libreria è stato possibile oltre alla renderizzazione della mappa su un piano prospettico, anche la realizzazione di un sistema che consenta all'utente una completa interazione con esso. Sarà possibile dunque, utilizzando la tastiera, navigare letteralmente sulle mappe, potendo muoversi e zoomare su di esse.

Utilizzando sempre *Google Maps* come esempio, la stessa società di Mountain View permette all'interno delle sue mappe di effettuare un "tilt" sul layer, ovvero ruotare rispetto all'asse x la visualizzazione della mappa. La struttura, però, di questa funzione è molto complessa. Essendo disponibile solo per le mappe satellitari, oltre a rappresentare le mappe e le strade permette di visualizzare anche le diverse altitudini dei terreni e una rappresentazione tridimensionale degli edifici come è possibile vedere nella figura 3.2.



Figura 3.2: Esempio della funzione tilt sul polo della Vasca Navale

Uno dei punti di forza del progetto, la cui architettura completa analizzerò nel corso del prossimo capitolo, è sicuramente il fatto che le mappe non vengono scaricate e/o collezionate in alcun database, ma esse stesse vengono richieste direttamente dalla sorgente (Google Maps) e caricate sul nostro client, in un diretto streaming del flusso di dati. Il fatto di non dover memorizzare tutte le mappe, oltre ad essere un vantaggio economico, di risparmio dal punto di vista delle componenti effettivamente necessarie, permette anche di mantenere sempre aggiornate le stesse informazioni non dovendo periodicamente controllare la presenza di modifiche sui dati presenti nello stesso database.

Questo progetto potrebbe, infine, essere la base di partenza per molti altri casi di studio. Molto spesso, ad esempio, si vorrebbe disegnare la topografia di una rete conoscendo le coordinate o gli indirizzi delle macchine che ne fanno parte. Grazie a questo sistema di visualizzazione, si avrebbe una veduta d'insieme maggiore rispetto a quella che probabilmente avremmo posizionando dei semplici "pin" all'interno di una mappa di Google.

Capitolo 4

Progetto del sistema di visualizzazione

La struttura alla base del mio tirocinio è facile da comprendere, ma allo stesso tempo non lo è stata la realizzazione di tutte le operazioni che si trovano dietro ad esso, in particolare il calcolo dello zoom ideale per rappresentare una determinata area geografica e il sistema alla base del movimento e caricamento delle tile aggiornate in base alle direzioni di spostamento. Analizzando la struttura del progetto, troviamo una pagina HTML contenente una form con i dati di partenza per ottenere la mappa dell'area prescelta, un server che elabori le coordinate dei punti inseriti e ne ricavi lo zoom corretto e infine si occupi dello streaming dell'immagine corretta e infine l'interfaccia 3D, sviluppata utilizzando la libreria *Three.js* con cui l'utente può interagire per ottenere le mappe in seguito allo spostamento in direzione nord, sud, est e ovest.

4.1 Modellazione delle informazioni

La prima sfida affrontata nel corso dello sviluppo del seguente progetto è stata sicuramente il modo in cui un utente volesse interfacciarsi con il sistema. Pensando a come, spesso, vengono consultate le mappe nei vari sistemi online, risulta macchinoso trovare, a partire da un centro ipotetico, il modo di visualizzare una determinata area. Proprio per questo motivo, si è preferito affrontare la problema dando la possibilità all'utente

di inserire le coordinate dei vertici della zona d'interesse. Risulta a questo punto immediato il calcolo del centro. Infatti una volta inseriti i due punti il centro avrà come coordinate il punto medio delle due coordinate di partenza. Considerando dunque due punti a e b di coordinate $(lat_a, long_a)$ e $(lat_b, long_b)$ il centro corrispondente sarà:

$$\left(\frac{lat_a+lat_b}{2}, \frac{long_a+long_b}{2}\right)$$

Il passo in questo è stato abbastanza semplice, perchè essendo il centro di un sistema di riferimento bidimensionale in punto medio tra due punti è la media delle coordinate dei due punti. Affrontando però la questione dello zoom e dello spostamento, la soluzione non è stata altrettanto diretta e semplice, tanto che le questioni saranno affrontate in due paragrafi distinti.

4.1.1 Il calcolo dello zoom adatto

Il calcolo dello zoom è stato il primo problema affrontato nel corso dell'implementazione e della progettazione del progetto. Lo zoom nelle mappe di Google è un valore discreto che varia da 0 a 21 e per comprenderne il funzionamento è stato necessario capire cosa venisse rappresentato ad ogni livello. Nella proiezione adottata da Google al livello 0 l'intero globo viene raffigurato in un'immagine di 256x256 pixels di larghezza e altezza. Questo ha rappresentato un valore di riferimento per lo studio della funzione che calcoli lo zoom di partenza. Andando, infatti, ad analizzare sperimentalmente i successivi livelli di zoom ho visto che ponendo lo zoom a 1, 256 pixels erano diventati troppo pochi per rappresentare l'intero globo. A questo grado di zoom infatti sono necessari esattamente 512 pixels di larghezza e altezza per rappresentare l'intera immagine terrestre, esattamente il doppio. L'ipotesi, dunque, a questo punto formulata vuole che lo zoom non è altro che una progressione geometrica di ragione 2. L'ipotesi è stata successivamente verificata analizzando anche il terzo grado di zoom, nonostante non fosse possibile mostrare un'immagine di 1024 pixels per lato in quanto le API di Google limitano l'utilizzo delle mappe statiche a dimensioni di 640 pixel per lato. È, dunque, possibile riassumere lo zoom di Google con la seguente espressione:

$$256 \times 2^{zoom}$$

Ad ogni livello, dunque, corrispondono un certo numero di pixel capaci di contenere la rappresentazione dell'intero globo, come è possibile vedere in tabella:

Livello	Pixels
0	256
1	512
2	1024
...	...
z	256^z
...	...
21	536.870.912

E' risultato a questo punto quasi immediato calcolare lo zoom necessario a rappresentare una determinata area geografica. Infatti, in 256 pixel sono contenuti tutti i 360 gradi della circonferenza terrestre, considerando che la nostra immagine sarà di 640 pixels la proporzione è già costruita:

$$\frac{640}{\Delta} = \frac{256}{360}$$

Il Δ della nostra funzione sarà in questo caso la variazione di **longitudine** tra i due punti richiesti dall'utente. Una variante della stessa proporzione, andrà utilizzata per la latitudine, dove i gradi da rappresentare non sono 360, bensì 180, o meglio per rispettare la proiezione utilizzata da Google 170,1022, perchè i poli in questo tipo di rappresentazione vengono considerati come due punti approssimati all'infinito non saremo quindi in grado di vedere interamente tutti i 180 gradi di latitudine che vanno dal polo nord al polo sud, bensì ne verranno raffigurati 85,0511 rispettivamente a nord e a sud dell'equatore.

A questo punto, una volta ottenuto il risultato del rapporto, è opportuno calcolare lo zoom, che come abbiamo visto può essere rappresentato come una potenza in base due. Dunque per passare dalla nostra proporzione al livello di zoom richiesto occorrerà eseguire il logaritmo in base due del risultato della proporzione:

$$\log_2\left(\frac{640}{\Delta} \times \frac{360}{256}\right)$$

Una volta ottenuto lo zoom necessario, a questo punto sarà sufficiente concatenare i valori del centro e dello zoom all'url di richiesta della mappa, fornito dalle API di Google¹ per poter caricare la mappa richiesta.

4.1.2 Calcolo dei centri successivi per lo spostamento

Se per lo zoom, i ragionamenti e i calcoli sono stati immediati, o quanto meno molto lineari, la stessa cosa non si può dire del calcolo dei nuovi centri per aggiornare la mia mappa nello spostamento lungo la longitudine e, soprattutto, la latitudine.

Perchè dover calcolare un nuovo centro? All'apparenza può risultare abbastanza inutile o eccessivo doversi calcolare un nuovo centro ad ogni spostamento. Essendo centro e zoom i due attributi necessari per caricare una mappa, questa rappresenterà solamente un certo spazio angolare della terra in relazione anche alla dimensione dell'immagine che viene richiesta, sia per quanto riguarda la latitudine che per quanto riguarda la longitudine. Sarà, dunque, necessario una volta raggiunto un certo margine della mappa, caricare la mappa successiva e di conseguenza occorrerà ricavarsi il centro per richiederla. Il problema principale, però è legato alla proiezione che Google ha deciso di utilizzare per rappresentare la terra in piano. Come si sa, la terra non è piana ma una sfera leggermente schiacciata ai due poli. Proprio per questo motivo, non è possibile direttamente rappresentare la terra su di una immagine piana, senza aver fatto le dovute proiezioni. Diversi sono stati i teorici che hanno elaborato delle proiezioni cartografiche del pianeta, ma in particolare il nostro studio ha riguardato la **la proiezione cilindrica centrografica modificata di Mercatore**, creata dal matematico e astronomo Gerhard Kremer e utilizzata da Google nella variante della *Web Mercator Projection*.

All'inizio, inconsapevolmente si erano trattate latitudine e longitudine in maniera indistinta, è stata dunque elaborata una semplice proporzione tra gradi e pixel, derivata da quella per il calcolo dello zoom, al fine di trovare il centro della mappa successiva, sia essa a nord, sud, est oppure ovest. Per il movimento longitudinale, dunque verso oriente e occidente, non sono emersi problemi. Riprendendo la proporzione precedente sappiamo che se a 360 gradi corrispondono a 256 pixels moltiplicati per la potenza

¹Vedi Capitolo 4, Paragrafo 2.3

in base 2 dello zoom, allora in 640 pixel ci saranno una certa quantità di gradi. Se sommiamo questa quantità alla longitudine del nostro centro, ne ricaveremo quella del centro successivo a oriente. Ricostruendo la formula si ha che:

$$\left(\frac{640 \times 360}{256 \times 2^{zoom}}\right) + longitude_c$$

dove $longitude_c$ è la longitudine del centro da cui ci stiamo muovendo.

In egual misura, se ci stiamo muovendo verso occidente, sottraendo la stessa quantità alla longitudine del centro di partenza otterremo il nostro centro ad occidente:

$$longitude_c - \left(\frac{640 \times 360}{256 \times 2^{zoom}}\right)$$

Inizialmente, l'idea è stata applicata anche alla latitudine, muovendosi verso nord e verso sud. Ovviamente in questo caso la proporzione era stata modificata considerando i 170,1022 gradi complessivi dei paralleli terrestri. La funzione verrebbe così riprodotta:

$$latitudine_c \pm \left(\frac{640 \times 360}{256 \times 2^{zoom}}\right)$$

I risultati inizialmente ottenuti non sono stati però quelli sperati. Il centro calcolato risultava sempre spostato, creando duplicazioni di alcune zone geografiche o a volte, addirittura, il completo taglio di altre. Dopo diversi fallimentari tentativi di approssimare il più possibile il calcolo, utilizzando, o teorizzando in alcuni casi, anche delle funzioni che in base alla latitudine approssimassero il punto in cui attaccare le mappe presenti sulla canvas. La risposta è stata trovata, alla fine, studiando la stessa proiezione di Mercatore.

Per la proiezione, un grado di latitudine all'equatore non vale quanto un grado di latitudine ai poli, questo è dovuto principalmente al diverso grado di distorsione presente in distinte aree geografiche della terra. Una rappresentazione di quanto appena detto può essere visto nell'immagine 4.1

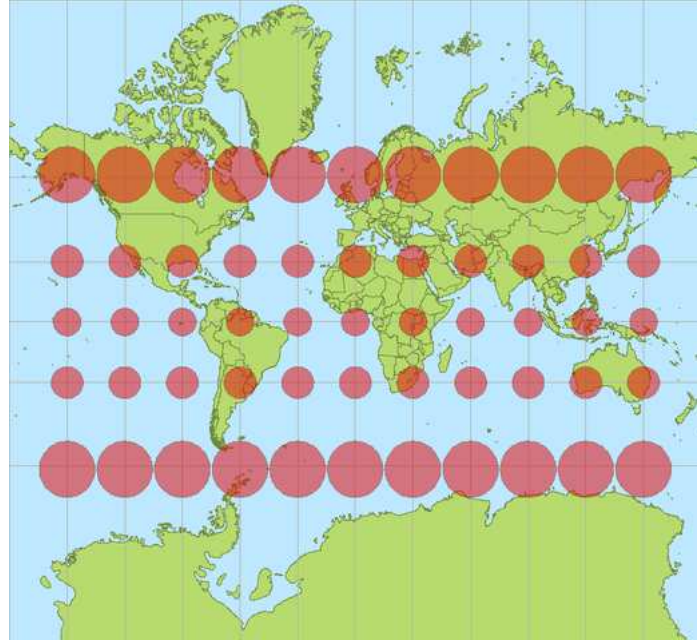


Figura 4.1: Rappresentazione della distorsione delle aree geografiche

Seguendo questa proiezione, per effettuare i calcoli è stato necessario effettuare diverse conversioni. In primo luogo quella tra gradi decimali e radianti seguendo una semplice proporzione:

$$\frac{deg}{180} = \frac{rad}{\pi}$$

A questo punto è stato necessario applicare le conversioni forniteci dallo stesso Mercatore, per comprendere quale fosse il centro successivo a nord e a sud.

Per prima cosa, è necessario passare dall'espressione in gradi a quella in radianti e successivamente occorrerà passare dal sistema di riferimento geoassiale al sistema in pixel utilizzando la seguente formula di conversione:

$$\frac{1}{2} \ln \left(\frac{1 + \sin(lat)}{1 - \sin(lat)} \right)$$

dove lat è la latitudine del nostro centro di partenza espressa in radianti. L'espressione riportata ha lo scopo di proiettare la latitudine dal sistema di riferimento geoassiale in un nuovo piano cartesiano. Bisogna dunque ragionare come se posizionassimo l'asse delle ordinate del nostro nuovo sistema lungo il meridiano zero o meridiano di Greenwich.

Una volta ottenuto questo valore, per ottenere la conversione del nostro punto sarà necessario aggiungerci la metà della dimensione del globo terrestre in pixel, secondo quella che è la dimensione di riferimento della proiezione, ovvero 256 pixels. Questo avviene perchè dovendo il nostro punto essere proiettato rispetto a quello che è il sistema standard, con la terra contenuta in una immagine quadrata di lato 256 pixels. La posizione del centro nel piano geoassiale, con coordinate (0,0), sarà sicuramente nella posizione $(\frac{256}{2}, \frac{256}{2})$, dovremo dunque tener conto di questo valore nella nostra trasformazione. L'espressione riassunta è la seguente:

$$\frac{256}{2} + \frac{1}{2} \ln\left(\frac{1+\sin(lat)}{1-\sin(lat)}\right)$$

Ottenuta questa trasformazione, per ottenere la posizione del nuovo centro occorre considerare altri due valori caratterizzanti per ogni tipo di mappa, il livello di zoom e la dimensione in pixels. Infatti una volta ottenuto il valore di proiezione con la formula data sarà sufficiente sommare o sottrarre la metà dell'altezza della mappa fratto la potenza in base due dello zoom, per ottenere rispettivamente il centro a sud e a nord della nuova mappa, come riportato nella seguente formula:

$$lat_{px} \pm \frac{\frac{h}{2}}{2^{zoom}}$$

con lat_{px} il valore della latitudine convertito in pixels e h l'altezza della nostra figura.

Una volta ottenuto questo valore, sarà necessario effettuare una controinversione che riporterà il riferimento nelle coordinate terrestri, così da essere concatenate poi all'url di richiesta. La formula inversa della conversione è la seguente:

$$2 \tan^{-1}\left(e^{\frac{y-128}{\frac{256}{2^{\pi}}}}\right) - \frac{\pi}{2}$$

Una volta convertito il valore da radianti in gradi sarà possibile utilizzare il risultato nella nostro sistema di movimento.

4.2 Il processamento dei dati e la richiesta della mappa

Avendo capito la proiezione e aver ricavato tutti i dati necessari dalla mappa, a questo punto sarà possibile creare il nostro sistema. Il servizio da me creato all'avvio mostra

una semplice interfaccia grafica con una form che permette all'utente di inserire i dati dell'area geografica inizialmente richiesta:

```
<form action="/map" method="post">
  North<br>
  Lat : <input type="text" name="n_lat"><br>
  Lon: <input type="text" name="n_long"><br>
  <br>
  South<br>
  Lat : <input type="text" name="s_lat"><br>
  Lon: <input type="text" name="s_long"><br>
  <button type="submit">Submit</button>
</form>
```

Come si può evincere, l'utente può inserire i valori di latitudine e longitudine di due punti, uno a nord e l'altro a sud che rappresenteranno gli angoli della mappa visualizzata. Non viene specificato se Nord-est, Nord-ovest o Sud-est, Sud-ovest in quanto non cambia ai fini del calcolo delle distanze angolari tra i due punti. Come si può notare dagli attributi del tag *form* nella porzione di codice sopra riportata il tasto "submit" della form, evoca un metodo post nel server, che si attiverà ed inizierà il processamento dei dati.

In particolare il metodo, richiamato dal percorso `/map`, prende i dati dell'input, ne estrapola lo zoom, la dimensione esatta dell'area in pixel e le coordinate del centro e le invia sotto forma di pacchetto json al client. Nella porzione di codice possiamo vedere l'implementazione del metodo.

```
router.post("/map", function (req, res){
  var n_lat = parseFloat(req.body.n_lat);
  var n_long = parseFloat(req.body.n_long);
  var s_lat = parseFloat(req.body.s_lat);
  var s_long = parseFloat(req.body.s_long);

  var zoom = Math.min(zoom_lat(Math.abs(n_lat- s_lat)),
```

```
        zoom_lon(Math.abs(n_long - s_long)))+1;
    var c_lat = (n_lat+s_lat)/2;
    var c_lon = (n_long+s_long)/2;
    var h = height(Math.abs(n_lat - s_lat),zoom);
    var w = width(Math.abs(n_long - s_long), zoom);
    var a = req.body;
    a.zoom = zoom;
    a.c_lat = c_lat;
    a.c_lon = c_lon;
    a.height = h;
    a.width = w;
    res.render("map", a);
});
```

Come è possibile notare vengono richiamati all'interno del metodo altri algoritmi, oltre a quello dello zoom, calcolato per la latitudine e per la longitudine e di cui poi viene ritornato il valore minimo, troviamo anche i metodi che calcolano le dimensioni in pixel della nostra area geografica. Infatti, anche se la mappa da noi richiesta è una mappa di dimensioni 640x640, la scelta è di mostrare solamente una porzione di essa, quella i cui margini sono definiti dai punti richiesti dall'utente.

Di seguito possiamo vedere il codice dei due metodi:

```
var height = function(d,z){
    var altezza = (256*Math.pow(2,z)*d)/latitude;
    return altezza*0.7;
};

var width = function(d, z){
    var larghezza = (256*Math.pow(2,z)*d)/longitude;
    return larghezza;
}
```

Come si può notare l'altezza è stata adattata moltiplicandola per un fattore di 0.7, questo è dovuto allo stesso problema di cui si è trattato nel paragrafo 4.1.2. Infatti, questo valore moltiplicativo consente di approssimare il più possibile il risultato senza tagli anomali, che essi siano troppo grandi o troppo piccoli, per quanto riguarda invece la larghezza, come avviene per il calcolo della longitudine essa non ha mostrato problemi risultando invece in linea con la proporzione utilizzata. I due valori di "Latitude" e "Longitude" sono rispettivamente 170.1022 e 360, ovvero i gradi complessivi della latitudine e longitudine terrestri.

Una volta dunque ottenuti tutti questi parametri, è possibile creare il flusso streaming della mappa. Per la realizzazione di questo metodo mi sono affidato alla funzione "pipe" presente in *Node.js*. La funzione è stata implementata nel seguente modo:

```
router.get('/image/:c_lat/:c_lon/:zoom', function(req, res) {  
  var url = 'https://maps.googleapis.com/maps/api/staticmap?size='+  
    '640x640&key=YOUR_API_KEY&center=';  
  
  request(url+req.params.c_lat+", "+  
    req.params.c_lon+"&zoom="+req.params.zoom).pipe(res);  
});
```

Attraverso questo metodo avviene dunque il caricamento della mappa che rappresenterà la texture degli oggetti sulla nostra scena 3D, che andremo ora ad analizzare nel dettaglio.

4.3 Proiezione nella vista prospettica, il client Three.js

Ho analizzato precedentemente come impostare la scena Three.js e, naturalmente, anche il progetto di questo "navigatore" seguirà lo stesso schema. Pur mantenendo la stessa struttura, il mio script avrà, come è normale, al suo interno diverse variabili in più oltre a quelle che abbiamo visto nel paragrafo introduttivo a questa tecnologia². Evitando di trattare nuovamente di tutte le caratteristiche ambientali da dichiarare e come di-

²Vedi paragrafo 2.2.1

chiararle, la concentrazione sarà rivolta direttamente alle componenti importanti del progetto.

Il primo elemento di cui mi trovo a parlare è il piano su cui posizionare la mappa come texture. Andando per gradi, analizziamo la struttura del codice:

```
var geometry = new THREE.BoxGeometry(640,640,0.1);
var plane;
var c_lon, c_lat;
var z = parseFloat("<%=zoom%>")
...
function init(){
  c_lon = parseFloat("<%=c_lon%>");
  c_lat = parseFloat("<%=c_lat%>");

  texture = THREE.ImageUtils.loadTexture("/image/"+c_lat+"/"+
    c_lon +"/"+z, {}, function () {});
  texture.minFilter = THREE.LinearFilter;
  material = new THREE.MeshBasicMaterial({ map: texture,
    color: 0xffffffff});
  plane = new THREE.Mesh(geometry, material);
  plane.rotation.x=-Math.Pi/2
  scene.add(plane);
  ...
}
```

La prima cosa che si analizza da questa porzione di codice è l'aver dichiarato una variabile `geometry` che definisce la struttura di semplice scatola piatta (una altezza di 0.1) e di forma quadrata con lato 640. Questa sarà la geometria che ricorrerà per ogni piano che verrà creato o inizializzato durante lo spostamento, azione che successivamente analizzerò nel dettaglio. Un'altra cosa che si nota è che i due centri una volta dichiarati, vengono concatenati nel path di richiesta dell'immagine, che richiama il metodo di streaming presente nell'architettura del codice di back end. Infatti, il metodo presente nella package `THREE.ImageUtils`, `loadTexture`, viene utilizzato per caricare la texture

da una fonte locale. In questo caso la fonte non è locale ma remota, il tutto è semplicemente superato dallo streaming del flusso di dati, che permette al client di caricare e renderizzare sul piano la mappa. La texture viene caricata come elemento del materiale attraverso l'attributo "map". Infine prima di essere aggiunto alla scena, il piano viene ruotato di 90 gradi per permettere la sua perfetta visualizzazione, in una prospettiva di 45 gradi.

Mancano ancora, però, alcuni dettagli di cui avevo parlato precedentemente, ovvero i due valori di "height" e "width". Come detto precedentemente, la mappa che abbiamo trovato, non sarà grande precisamente quanto l'area richiesta, bensì più grande. Proprio per questo motivo avevamo introdotto i valori di altezza e larghezza. Three.js, non ha una funzione che permetta il taglio di oggetti o di mostrare solamente una parte di essi. Dal momento in cui avevo bisogno di realizzare una struttura simile, ho iniziato alle diverse soluzioni da approcciare per la realizzazione del problema che si era creato. Come dicevo nella descrizione di questa libreria, una delle caratteristiche principali è sicuramente la sua licenza libera e open source. Proprio per questo motivo moltissimi sviluppatori nel web hanno implementato, e stanno implementando, librerie complementari al framework di Mr.doob. Una di queste è la libreria **csg.js**, completamente gratuita e disponibile su Github tra le repositories del suo ideatore Evan Wallace[Wal15]. Questa libreria permette di costruire oggetti compositi. Infatti lo stesso nome è l'acronimo di Constructive Solid Geometry. Grazie a questa libreria ho creato un nuovo piano posizionato al di sopra del piano della mappa, e che oscura tutto il contenuto della mappa al di fuori dell'area d'interesse richiesta dall'utente. Un'applicazione della libreria può essere visto in questo esempio:

```
var aux = new THREE.BoxGeometry("<%=width%>",
"<%=height%>", 0.1);
var auxMesh = new THREE.Mesh(aux);
var auxBSP = new ThreeBSP(auxMesh);

var logical = new THREE.BoxGeometry(20000, 20000, 0.1);
var logicalMesh = new THREE.Mesh(logical);
var logicalBSP = new ThreeBSP(logicalMesh);
```

```
var newPlaneBSP = logicalBSP.subtract(auxBSP);  
var newMaterial = new THREE.MeshBasicMaterial(  
  {color: 0xffffffff});  
auxPlane = newPlaneBSP.toMesh(newMaterial);  
auxPlane.position.y = 0.6;  
auxPlane.rotation.x = -Math.PI/2;  
  
scene.add(auxPlane);
```

Gli oggetti vengono prima inizializzati come "mesh" per essere introdotti nella scena Three.js, successivamente convertiti in BSP, come richiesto dalla libreria di supporto e infine in seguito all'operazione di combinazione, in particolare la sottrazione dei due piani per poter ottenere un piano forato delle dimensioni richieste, si ha l'ultima conversione in mesh per essere aggiunto alla scena. Il piano risulterà ruotato di 90 gradi e rialzato su y di 0.6 pixel per evitare contatti tra i piani che ne compromettano la visualizzazione durante gli spostamenti.

Una volta impostata la scena, non ci resta che parlare nel dettaglio delle due azioni che l'utente potrà effettuare sulla mappa.

4.3.1 Il sistema di movimento

Affinchè il progetto risultasse interattivo e funzionale, occorreva rendere dinamica l'interazione con l'utente, volendo ricreare quelle che sono di fatto le dinamiche base offerte dai diversi sistemi di navigazione su mappa presenti nel web. Uno di questi è senza dubbio la possibilità di spostarsi sulla stessa mappa, una volta caricata la tile di riferimento. Per questo scopo, si è deciso di implementare un sistema di movimento che necessita l'uso dei soli tasti freccia della tastiera.

Three.js non possiede al suo interno delle funzioni che catturino l'utilizzo da parte dell'utente della tastiera, è stato dunque necessario far riferimento ad una nuova libreria di supporto, anch'essa sviluppata a partire dalla libreria base estendendo ulteriormente le funzionalità della stessa. Mi sto riferendo alla libreria **THREE.js**, una estensione di Three.js, rivolta principalmente allo sviluppo di videogames. Grazie a questo framework

è stato possibile catturare lo stato della tastiera. Per utilizzarla, a tal scopo, è stato sufficiente dichiarare all'inizio del nostro script una variabile *keyboard* in questo modo:

```
var keyboard = new THREE.KeyboardState();
```

La funzione *KeyboardState()* permette di monitorare in tempo reale lo stato della tastiera, nel momento in cui premeremo un tasto esso attiverà un evento, principalmente booleano che ci permetterà di interagire con la mappa.

I tasti da noi monitorati per il progetto sono 6:

- Tasti freccia (su, giù, destra e sinistra) per lo spostamento lungo gli assi geografici;
- Tasti W e S per lo zoom in e zoom out (di cui parlerò nel prossimo paragrafo).

Analizzando nel dettaglio la struttura del sistema di movimento per prima cosa è necessario definire una nuova funzione alla struttura del nostro script: la funzione *update()*. Questa funzione verrà richiamata subito dopo il metodo *render()*, presente nella funzione *animate()* del nostro progetto *Three.js*.

Non volendo mostrare dettagliatamente il movimento per ogni lato, prenderò come riferimento il modello dello spostamento a nord rispetto alla nostra area di partenza. Il problema che mi sono inizialmente posto è stato quello del voler ottenere un movimento che fosse il più fluido possibile, rappresentando la maggiore area che sia possibile. Soprattutto, per rendere il movimento fluido, occorre caricare le nuove mappe, prima che la nostra area visualizzata ecceda la dimensione dell'immagine di partenza, e che soprattutto questa operazione venga applicata ciclicamente alla fine di ogni tile. Per questo motivo ho creato un ciclo *while* che prenda come riferimento la posizione del nostro oggetto *plane* lungo l'asse su cui si sta spostando, dunque l'asse X o Z. La posizione del piano verrà confrontata con le dimensioni della nostra area di visualizzazione, i parametri *width* e *height* che avevamo definito precedentemente lato server, che ci aiuteranno a ricavare la posizione di controllo. Infatti avendo un quadrato di 640 pixel per lato come immagine, sappiamo che la dimensione dell'area nascosta della nostra mappa precaricata sarà:

$$\frac{640-h}{2}$$

A questo punto la condizione del nostro ciclo while è presto costruita:

```
while(plane.position.z > ((640 - h)/2))
```

Per visualizzare l'area più grande disponibile, si è deciso di non raffigurare solamente il piano sequenzialmente attaccato, ma bensì 3 nuovi piani, che rappresenteranno nel nostro caso, lo spostamento a nord, le aree a nord, nord est e nord ovest. Per evitare comunque la sovrapposizione di troppe immagini che potrebbe causare un rallentamento nella fluidità del movimento, oltre che un glitch grafico nella rappresentazione, si aggiungeranno alla scena solamente i piani a nord est e nord ovest, poichè le due metà adiacenti degli stessi compongono la stessa area rappresentata dall'immagine a nord. A questo punto come nello scorrimento di una lista vengono aggiornati i riferimenti ponendo il piano di partenza come piano inferiore e il nuovo piano non aggiunto come piano di riferimento per proseguire lo spostamento, in questo modo sarà possibile al verificarsi nuovamente della condizione del ciclo while caricare i nuovi piani successivi. Una volta aggiunti i piani e aggiornati i riferimenti, i vecchi piani ormai lontani dal centro della nostra visualizzazione vengono eliminati dalla scena per alleggerire il carico grafico del progetto.

Si può vedere un esempio dell'algoritmo implementato, omettendo la definizione dei nuovi piani poichè uguale a quella fatta per l'oggetto di partenza:

```
function update(){
var moveDistance = 1;
if ( keyboard.pressed("up") ){
    newClat = getNorth(c_lat, z, 640);
    cEs = eastCenter(c_lon);
    cWe = westCenter(c_lon);
    plane.position.z += moveDistance;
    while(plane.position.z > ((640 - h)/2)){
        c_lat=newC_lat;
        scene.remove(supPlane, supE, supW);
        scene.add(plane);
        ...
    }
}
```

```
        scene.add(supW, supE);
        scene.remove(infPlane, infE, infW);
        infPlane=plane;
        plane=supPlane;
        infE=supE;
        infW=supW;
        supE=null;
        supW=null;
        supPlane=null;
    }
```

Come visto nel paragrafo sulla modellazione delle informazioni, al fine di calcolare le nuove coordinate geoassiali, vengono calcolati i nuovi punti di riferimento, nel nostro caso sono a nord, nord est e nord ovest, e proprio per questo le variabili newClat, cEs e cWe rappresentano rispettivamente: la latitudine dei centri a nord, la longitudine del centro nord est e quella del punto a nord ovest.

Al pari del movimento a nord, la stessa tipologia di algoritmo è stato implementato a sud, a ovest e a est, cambiando naturalmente i riferimenti in base al verso dello spostamento.

Le mappe verranno caricate come effettuato nella creazione della scena, richiamando il server per lo streaming dell'immagine.

4.3.2 Il sistema di zoom

Al pari del sistema di movimento anche per lo zoom va fatta una menzione speciale. Infatti anche in questo caso, il ragionamento è molto simile, seppur riferendosi in questo caso ad uno spostamento sull'asse delle Y. Lo spostamento in questo caso è solamente utilizzato per ricreare un effetto di avvicinamento o allontanamento dall'area in cui ci troviamo. Come introdotto precedentemente, i tasti utilizzati in questo caso sono W ed S, rispettivamente per lo zoom in e zoom out. L'effetto dello zoom, cambierà ovviamente l'area visualizzata e aumenterà o diminuirà i dettagli su di essa visibili.

Anche in questo caso come per il movimento si effettua un controllo su un valore prefissato muovendo i piani di un valore pari a 0.2 lungo l'asse Y, fino al raggiungimento

del valore di 7. In quel momento, verrà caricato un nuovo piano, denominato zoomP, con la mappa centrata nel punto del piano di riferimento corrente, e uno zoom aumentato o diminuito di una unità. A questo punto avviene l'aggiornamento del piano di riferimento e l'eliminazione di tutti i piani intorno ad esso esistenti, in questo modo sarà possibile iniziare nuovamente lo scorrimento della mappa con un dettaglio maggiore, oppure sfruttare il ciclo while per aumentare il dettaglio.

L'implementazione in codice di quanto appena descritto è la seguente, omettendo anche in questo caso la definizione del nuovo piano essendo esso identico ai casi precedenti:

```
if(keyboard.pressed("W")){
    if(z<21){
        plane.position.y+=0.2;
        auxPlane.position.y+=0.2;
    }
    while(plane.position.y>7 && z<=20) {
        z += 1;
        ...

        scene.remove(plane);
        scene.add(zoomP);
        plane = zoomP;
        zoomP = null;
    }
}
```

In questo caso viene aggiunto un controllo ulteriore per verificare il valore dello zoom: se siamo già al valore massimo o minimo dello stesso non sarà possibile aumentare o diminuire il suo valore, memorizzato nella variabile z.

Capitolo 5

Verifiche funzionali

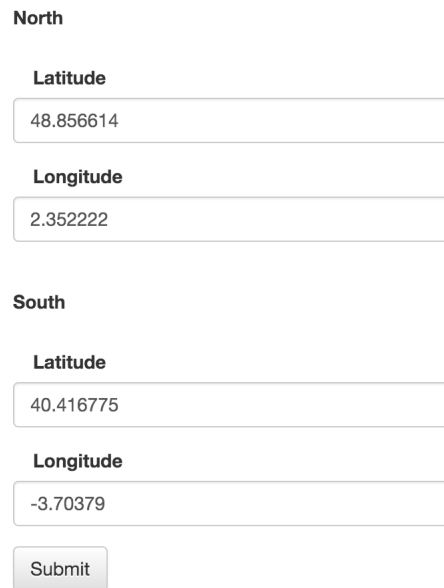
5.1 La form di richiesta

L'interfaccia utente è basilare, la form di richiesta permette all'utente di inserire i vertici dell'area di partenza. Nel corso dello sviluppo del mio progetto ho utilizzato come punti di riferimento le coordinate di Parigi e Madrid.

The form is structured as follows:

- North**
 - Latitude**:
 - Longitude**:
- South**
 - Latitude**:
 - Longitude**:
- Submit**:

Figura 5.1: Struttura della form per la richiesta dell'area iniziale



The form is divided into two sections: 'North' and 'South'. Each section has a 'Latitude' label followed by a text input field, and a 'Longitude' label followed by a text input field. The 'North' section contains the values '48.856614' and '2.352222'. The 'South' section contains the values '40.416775' and '-3.70379'. A 'Submit' button is located at the bottom of the 'South' section.

Section	Latitude	Longitude
North	48.856614	2.352222
South	40.416775	-3.70379

Figura 5.2: Inserimento delle coordinate di Parigi e Madrid

5.2 La proiezione su un piano prospettico

Una volta inserite le coordinate dei due punti e inviate le informazioni al server cliccando il tasto "Submit", l'utente sarà reindirizzato sulla pagina contenente la nostra canvas 3D. A questo punto la mappa verrà caricata sulla scena e visualizzata in chiave prospettica.



Figura 5.3: Proiezione dell'area tra Parigi (in alto a destra) e Madrid (in basso a sinistra)

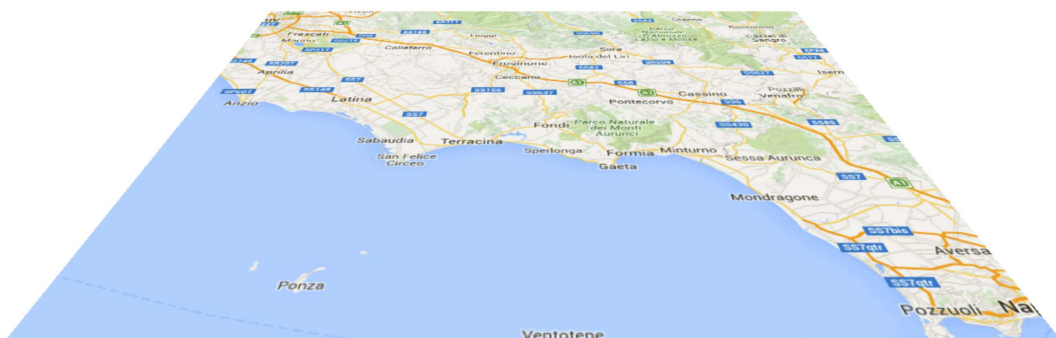


Figura 5.4: Proiezione dell'area tra Roma (in alto a sinistra) e Napoli (in basso a destra)

Una volta caricata la cartina dell'area desiderata sarà possibile iniziare ad interagire con l'ambiente creato e dunque iniziare la navigazione della mappa.



Figura 5.5: Questa è la situazione dopo una serie di spostamenti

L'utente non si accorgerà del caricamento di una nuova tile, se non per la presenza del logo *Google* e la dicitura dei Termini e condizioni di utilizzo, presenti in basso a ogni tile.

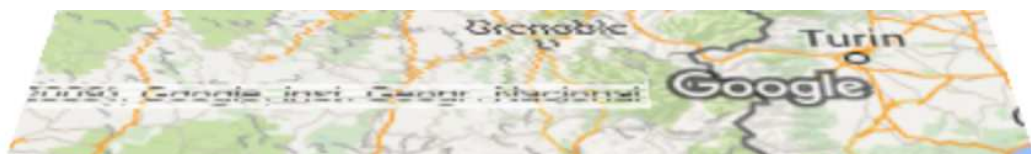


Figura 5.6: La scritta dei termini e condizioni di utilizzo e il logo di Google stampati sulla mappa

Nel momento in cui si volesse effettuare lo zoom, basterà premere il tasto W per aumentare il dettaglio o S per diminuirlo.



Figura 5.7: Risultato dell'azione di zoom in sulla mappa



Figura 5.8: Risultato dell'azione di zoom out sulla mappa

Una volta ottenuto il risultato dello zoom sarà possibile riprendere con la navigazione e spostarsi lungo tutta la terra.



Figura 5.9: Nuovo esempio di spostamento

Conclusioni e sviluppi futuri

L'obiettivo del tirocinio era quello di creare un sistema di visualizzazione di mappe geografiche le cui caratteristiche fossero molto simili a quelle di un navigatore. Proprio come i navigatori satellitari, infatti, il sistema da me creato permette di visualizzare una mappa su di un piano la cui vista ha un'inclinazione di circa 45 gradi. Il secondo scopo era, inoltre, quello di creare un sistema che consentisse una semplice interazione con l'utente per le azioni di spostamento e zoom.

Il sistema creato non genera eccessivi sovraccarichi dal un punto di vista grafico, essendo presenti sulla scena al più 4 o 5 piani durante gli spostamenti, per mostrare correttamente la maggiore area geografica possibile, evitando soprattutto di lasciare dei punti vuoti nello scorrimento.

La struttura del progetto, infine, non possedendo alcuna componente per il salvataggio delle mappe consente di avere sempre le mappe più aggiornate evitando di dover aggiornare manualmente un eventuale archivio delle stesse. Oltre il fatto che sarebbe stato necessario memorizzare un quantitativo di tile molto alto, essendo questo sempre maggiore in relazione anche al livello di zoom delle tile per ogni livello di zoom avremmo il doppio delle tile rispetto a quello precedente.

Diversi possono essere gli sviluppi futuri di questo progetto, che andrò ad elencare di seguito:

- Aggiunta di un database delle coordinate geografiche, per consentire all'utente di scegliere da un elenco di posizioni più comuni, ad esempio città o luoghi noti, i vertici dell'area da visualizzare;
- Consentire all'utente durante la navigazione di passare dalla visualizzazione della

mappa stradale a quella satellitare attraverso un semplice tasto presente sulla scena;

- Utilizzo del progetto come base per lo sviluppo di un sistema di visualizzazione di oggetti o grafi su aree geografiche, come può essere il caso, ad esempio, quello della visualizzazione della topografia di una rete locale.

Bibliografia

- [Bin10] Bing. New bing map app: 2010 tour de france, 2010.
- [Con15] World Wide Web Consortium. Document object model, 2015.
- [Fou15] OpenStreetMap Foundation. Openstreetmap, 2015.
- [Gro15] Khronos Group. Opengl es - the standard for embedded accelerated 3d graphics, 2015.
- [Inc15a] Google Inc. Google maps apis, 2015.
- [Inc15b] Google Inc. Google maps apis | google developers, 2015.
- [Map15] Mapbox. Showcase | mapbox, 2015.
- [Wal15] Evan Wallace. csg.js, 2015.
- [Wik14] Wikipedia. Actionscript — wikipedia, l'enciclopedia libera, 2014. [Online; in data 27-novembre-2015].
- [Wik15a] Wikipedia. Application programming interface — wikipedia, l'enciclopedia libera, 2015. [Online; in data 26-novembre-2015].
- [Wik15b] Wikipedia. C++ — wikipedia, l'enciclopedia libera, 2015. [Online; in data 26-novembre-2015].
- [Wik15c] Wikipedia. Canvas (elemento html) — wikipedia, l'enciclopedia libera, 2015. [Online; in data 28-novembre-2015].

[Wik15d] Wikipedia. Open source — wikipedia, l'enciclopedia libera, 2015. [Online; in data 25-novembre-2015].