[1]

howest hogeschool

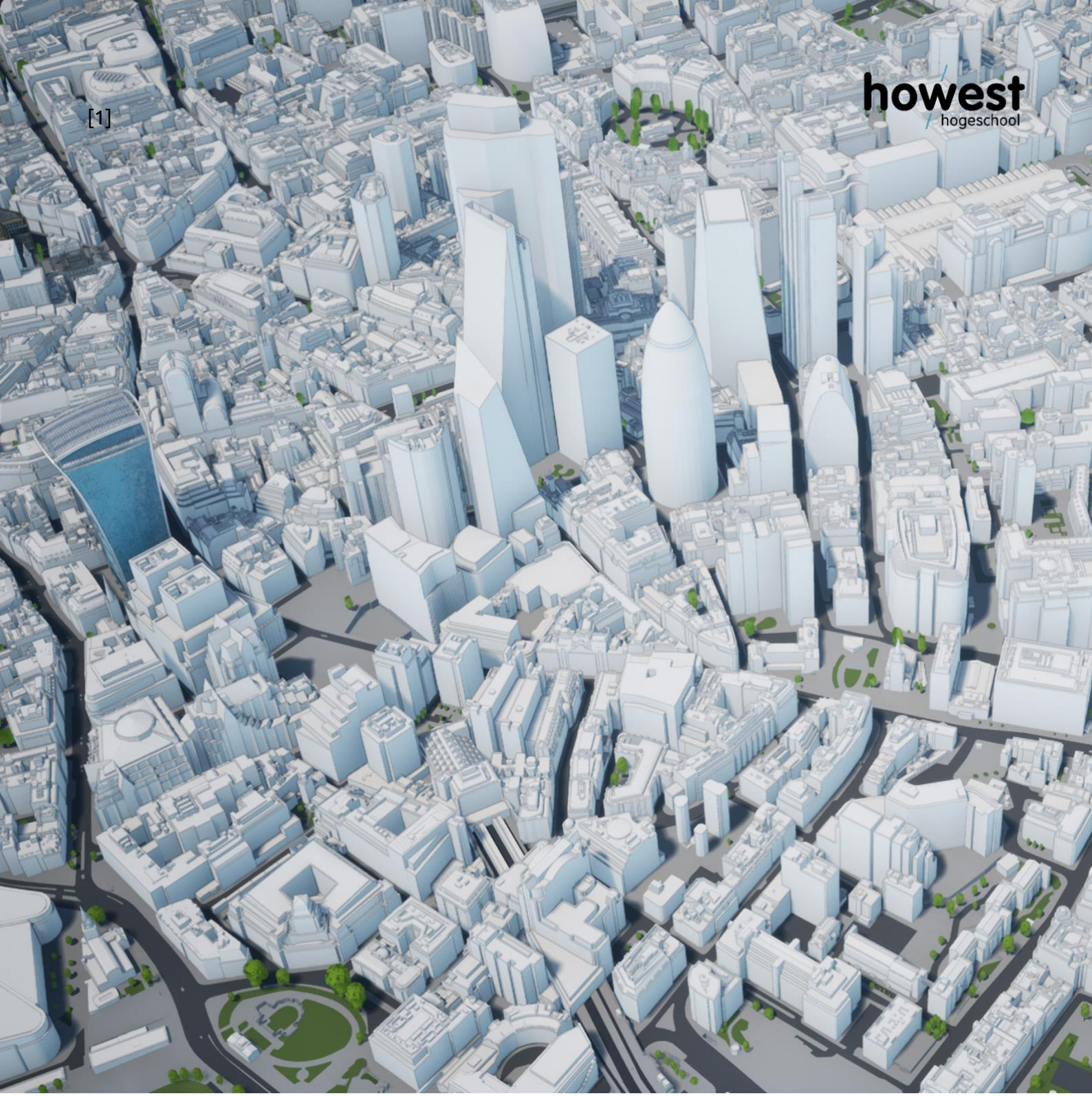

Figure 1 Aerial view of a 3D model of London

How can you use three.js to display an interactive digital map of 3D buildings that visualizes a path of a tour?

Internal promotor: Simon Coudeville
External promotor: Azat Omarov

Research performed by

Andrei Lavrenov

To receive a Bachelor's Degree in

multimedia & Creative Technologies

Howest | 2021-2022

Preface

As a final assignment of the Media & Creative Technology course at Howest, a university of applied sciences in Kortrijk, I am expected to write a bachelor's thesis. This is my last assignment that serves as the final proof of competency.

This thesis is a direct follow-up on the Research Project module from the fifth semester and is based on the following concrete question:

> *"How can you use threejs to display an interactive digital map of 3D buildings that visualizes a path of a tour?"*

In this thesis, I will first review and analyze the research and technical demo from the Research Project module. This will include looking at its development and performance, alongside the user interface, user experience and practical applications.

That will be followed by a reflection, alongside suggestions by various people from the industry. Based on this feedback I will issue an advisory for those who want to tackle a similar challenge.

This thesis wouldn't have been possible without the following people, so I would like to take the time to thank them for their help:

- Sergey Andreev – Full Stack Developer at Redpencil
- Azat Omarov – Tech Lead Frontend at Crystal Spring
- Emile Goeminne – Junior Full Stack Developer at Baldwin
- Daniil Voloshin – Junior Financial Analyst at KPMG
- Thiemo Seys – Artificial Intelligence Engineer at EEVE

I would also like to thank Crystal Spring, the company I was interning at while writing this thesis for their understanding and allowing me time-off to write this thesis.

30th of May, 2022

Andrei Lavrenov

Abstract

3D is still rare sight on the web, and when done well nearly always catches the customer's attention. The goal of this thesis was researching the feasibility and value of using three.js to generate a 3D city environment. More specifically, it answers the following question:

> *"How can you use Three.js to display an interactive digital map of 3D buildings that visualizes a path of a tour?"*

The theoretical section of the research will look at how Three.js and its underlying technology works and how to use it. Aside from that, it will also investigate digital mapping, where to find digital maps, and the various techniques and formats involved.

The practical section implements the knowledge from the theory to create a demo project that allows the user to generate a 3D city environment and then create and animate a path through it. It consists of a core npm package that handles generating the city and path, and two demo projects that implement it.

It is important to mention that my demo is a proof-of-concept that shows the potential of this technology rather than a fully-fledged usable product. With this demo, I have succeeded in generating a 3D city from OpenStreetMaps data and animating a path through it, but the aesthetics leave a lot to be desired and there are many issues and potential improvements that need to be addressed before it can be considered a finished product.

The final decision is divisive – it is indeed possible to use three.js to render a 3D city and animate a path throughout it, but I advise against using 3D for this purpose, and animating on a 2D map instead.

Table of Contents

Annotations

Abbreviations

ABBREVIATION	EXPANDED
3JS	Three.js
API	Application Programming Interface
DOM	Document Object Model
FPS	Frames Per Second
HZ	Hertz
MCT	Media & Creative Technologies
POLY, POLIS	Polygon(s)
UI	User Interface
UX	User Experience

Glossary

TERM	DEFINITION
Document Object Model	A standard API in the browser for code to interact with the webpage
Frames per second	A unit for measuring performance of a 3D application. How many "frames" of motion the GPU is able to render each second. For reference, movies are usually shot at 24FPS, and the 7th generation of consoles usually targeted 30FPS. The average display can refresh at 60FPS, with some performance-oriented models being able to double or triple that.
Mesh	A 3D Object made by combining multiple polygons together
Photogrammetry	A technique for creating 3D object from an array of photographs taken of it
Polygon	A shape made by connecting multiple vertices together. In 3D graphics, the most common shape is a triangle, because it guarantees that the polygon will be planar, which will make math related to it easier and faster.
Rasterization	A type of sampling used in real-time 3D applications to map scene geometry to pixels on a display. This only maps the shapes, and shaders can be used to decide the color of each pixel.
Raycasting	Technique that involves casting (sending) a theoretical ray from and to a specific point to determine what it intersects. Commonly used in a variety of 3D software for interaction or rendering purposes.
Refresh rate	The number of times per second that a display can change all its pixels, usually expressed in hertz
Rendering	The process of generating a digital image from a 2D or 3D model.
Shader, Shading	A specialized computer program used to shade a scene before it is Rendered to the screen. Shaders are used to depict depth perception and color in 3D models, as well as create special effects such as outlines around objects.
Vertex, Vertices	A data structure that describes the location, position, and orientation of a specific point in space

1 Introduction

1.1 Inspiration

The idea for this research and subsequently thesis originated from a Codrops development blog article "Animated Map Path for Interactive Storytelling" where Lucas Bebber used the canvas to animate a photographer's travel log as a way to connect the story being told with the path itself. [2]

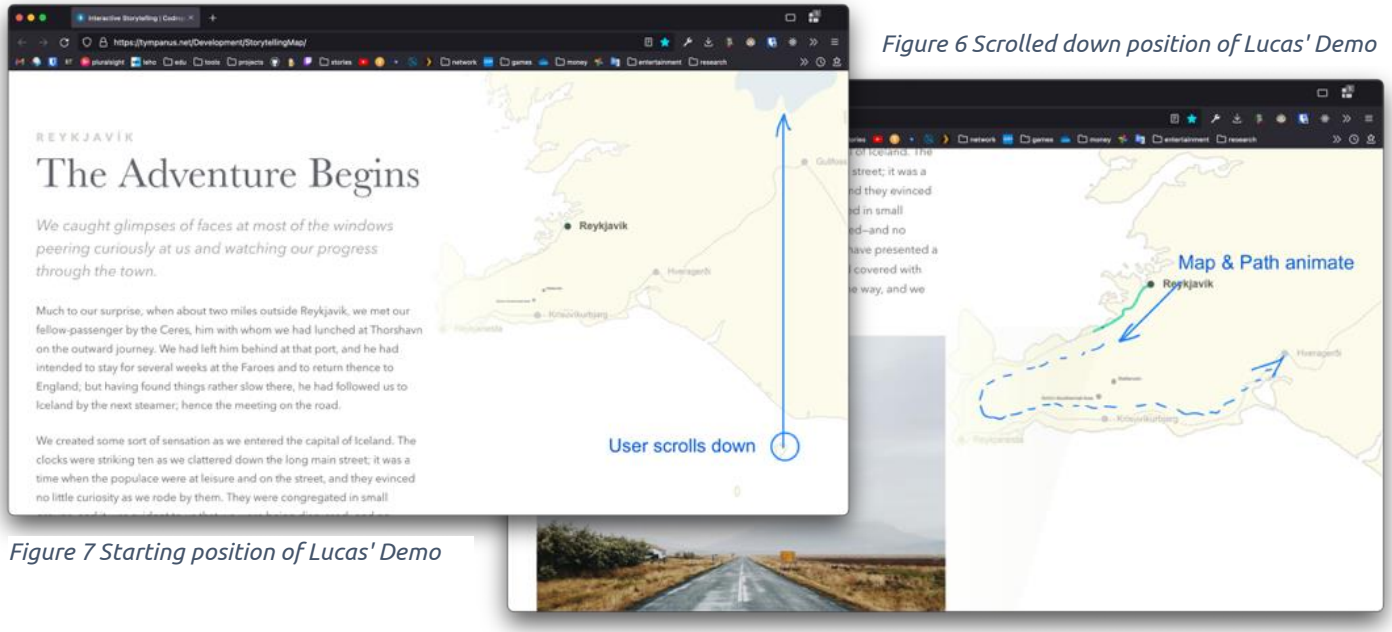

Figure 7 Starting position of Lucas' Demo

Figure 6 Scrolled down position of Lucas' Demo

As I was developing a website for a tour guide at the time, I started wondering whether it was possible to implement a similar effect in 3D within a city environment to animate tours.

Despite the use of 3D in movies, special effects, video games and other media, its presence on the web is still a rare treat, mostly constrained to websites where 3D is already the focus, like Sketchfab or individual web developers' portfolios. [3]

A decade ago, it was unimaginable as the combination of lacking hardware and the inability of browsers to use the GPU for hardware accelerated rendering was the biggest limitation and people had to download external plugins to display 3D content.

That changed in 2011 with the introduction of WebGL. This is a JavaScript API that allows a page to interact with the GPU, opening new possibilities for the web.

However, using WebGL directly is difficult and requires a lot of math and graphics programming knowledge. Because of that, there are a plethora of WebGL frameworks to ease development, with three.js being the most popular one. [4]

1.2 Goals

The primary goal of this thesis is to test whether three.js can be used effectively to display and animate a 3D city environment, and to check whether it is a good idea in the first place.

Over its course, it will also attempt to answer a variety of sub questions such as:

- Where can you find open source 2D or 3D map data?
- How can you use this data in 3JS and is it possible to generate 3D buildings based on 2D data?
- Is it possible to coordinate the experience based on the capabilities of the user's device?
- How to generate a path between two points that would use the roads?

1.3 MVP

The research will be performed by creating a proof-of-concept project. It will consist of an npm package you could import, that will process GeoJSON data and generate a 3D city environment. It should have the feature to allow the web developer to draw a path through the city, and then animate this path for the user.

Figure 8 starting state showing the page on initial load

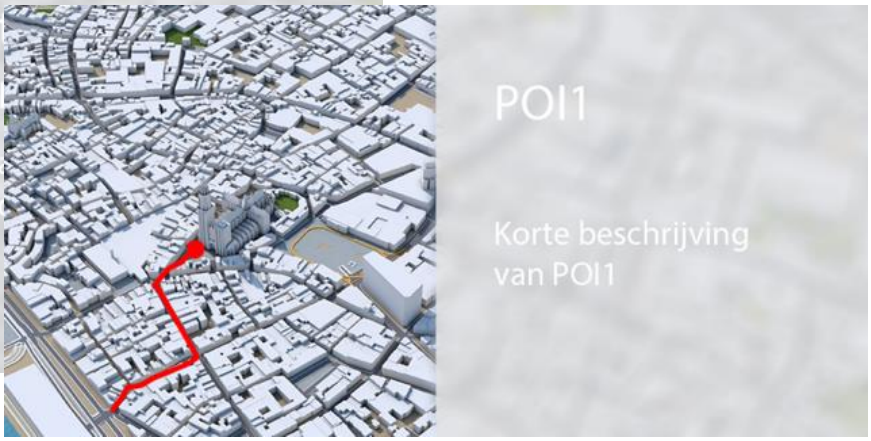

Figure 9 State when the user has scrolled down, showing a path through the city.

Above you can see a mockup in photoshop of the proposed project. This thesis will touch topics such as where to get the data to generate a city, performance on desktop and mobile, whether it adds value and other related questions.

2 Research

2.1 WebGL

What & How

The Web Graphics Library is a JavaScript API that allows for low level graphics programming, making hardware accelerated interactive 3D and 2D graphics possible within any compatible web browser without the use of additional plug-ins.

The API is based on OpenGL ES and can be used in the HTML5 <canvas> elements. It's important to mention that WebGL is not a 3D API. It's a Rasterization API. All it can do it draw points, lines, and triangles. It's up to the programmer to write the math to make them look 3D, which is where libraries like three.js come in.

It is also important to mention that while WebGL's API is very similar to OpenGL, it is not OpenGL, and does not and cannot access the GPU directly. The WebGL commands stay in the browser, and the browser uses various techniques to translate them to something the GPU can understand. [5]

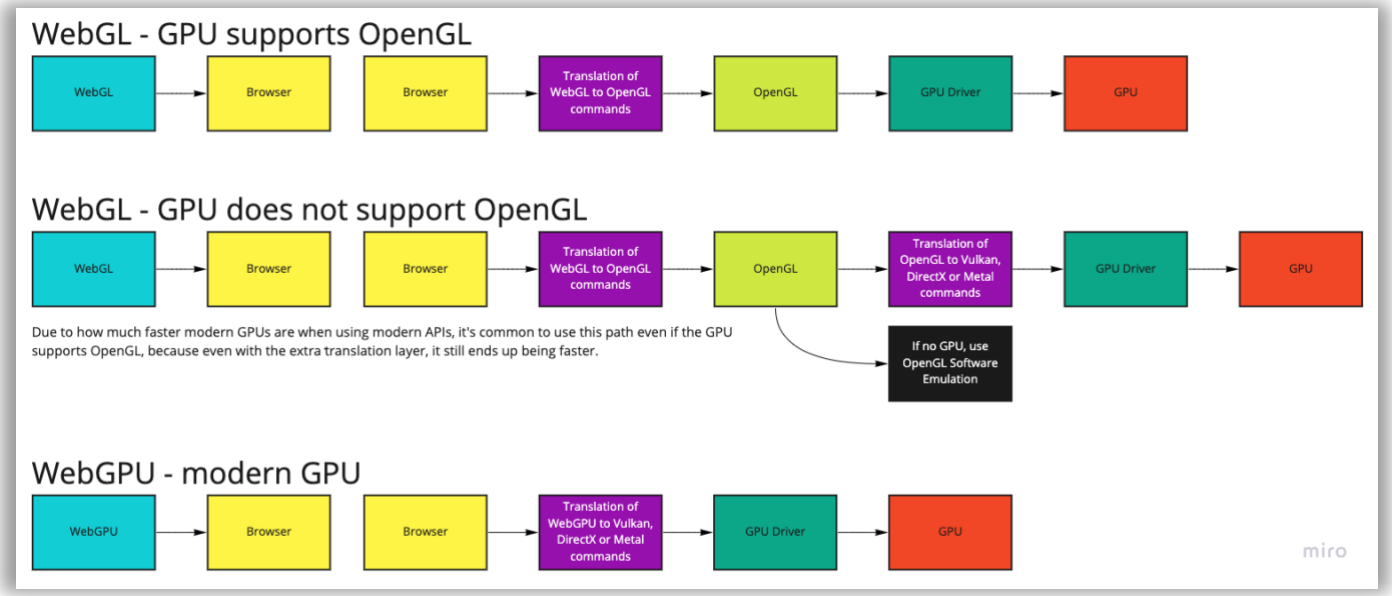

Figure 10 Simplified WebGL *& WebGPU pipelines*

One such technique is Google's Angle project. It's described as an almost native graphics layer engine, and is currently used by both Chrome and Firefox to render WebGL on Windows. In fact, Chrome even uses it for all graphics rendering, including Canvas2D and not only WebGL.

It translates OpenGL ES API calls to one of the hardware-supported APIs available on the user's platform and currently provides translation from OpenGL ES 2.0, 3.0 and 3.1 to Vulkan, desktop OpenGL, OpenGL ES, Direct3D 9, and Direct3D 11.

It is being actively developed by Google, with future plans to add the ability to translate Open GL ES 3.2, and supporing more operating systems like Chrome OS and Fuchsia. [6]

An interesting point is that due to the old paradigms of the OpenGL API, this extra translation step is often preffered over running OpenGL directly even if it's supported, as even with that extra step, the modern APIs will be faster. [7]

WebGPU is a new upcoming standard that is poised to replace WebGL as the way to use the GPU on the web. Its API implements modern paradigms to better correspond with how current GPUs work. More about it in the "Future Development" section.

Current state

As of May 2022, WebGL 2.0 is widely supported by every relevant major browser, and thus can be used without compatibility concerns and is a safe bet for any 3D project that is starting in 2022. [8]

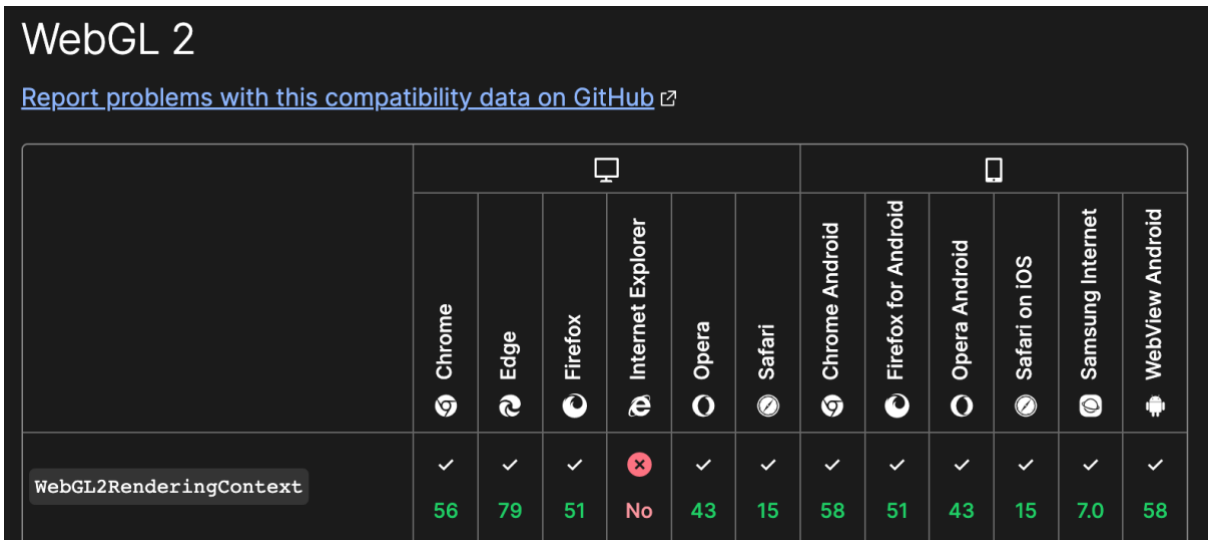

Figure 11 Top of WebGL 2.0's compatibility table [8]

However, as page load time becomes increasingly important, and with the average user not willing to wait more than a couple of seconds before they bounce, making sure your 3D application loads and starts up quickly is a must. [9]

Alongside loading time, and especially on mobile, performance and power consumption remain a concern for mobile devices and users, so testing and optimizing on a variety of hardware moves from being an extravagance to something essential.

Libraries

Because WebGL is a low-level API and is complicated and often tedious to learn and implement, a variety of libraries have sprung up that serve as an abstraction layer on top of it, simplifying the development. A non exhaustive list includes entries such as Babylon.js, Pixi.js, and many others, with Three.js being the most popular of them all if you go by the amount of Github Stars.

Future Development

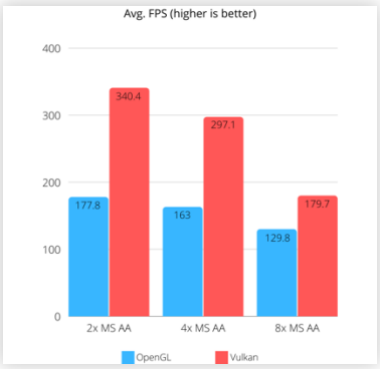

Figure 12 Graph showing Vulkan achieving higher FPS than OpenGL on the same hardware [11]

Despite OpenGL still being widely supported, its development has mostly stalled and it's been in maintenance, with the last major update happening in 2017, and Vulkan being pushed as its alternative. [10]

This is because the architecture of modern GPUs has changed massively, and they require a paradigm shift in the structure and design of the APIs used to access them.

While you do not have to worry about WebGL going away any time soon, it's also being succeeded by a new API called WebGPU that follows those modern paradigms. It is currently in the proposal phase at the W3C forum, but early performance testing already shows massive improvements. [11]

WebGPU is currently available behind a flag in Canary and Firefox and the Three.js library has been working on added support for it as well. [12] As it is currently still in the early development phase, it is not recommended for production projects, but it is something to keep an eye on. [13]

2.2 Three.js

Three.js is a JavaScript library that serves as an abstraction layer on top of WebGL to simplify its use. It was initially developed by Ricardo Cabello for personal use to serve as an alternative to Flash. Later with the public release, it became a way to remove one's dependency on Flash and the OS that supported it. [14]

It was chosen due to its popularity, as that meant a large amount of documentation and resources.

As mentioned in 2.1, WebGL is a Rasterization API. This means that to draw something as simple as a flat polygon requires over 100 lines of code, and going to 3D will only exacerbate that problem, with a rotating cube will require writing your own Quaternion & Matrix multiplication functions. [5] [15] [16]

Three.js abstracts it away into a structure of JavaScript objects, the 3 primaries being the *renderer, camera*, and *scene*.

The Scene object is a tree-like structure that acts as the core of 3js. It contains all the things you want to render in your application, like the geometry, materials, textures, etc... [17]

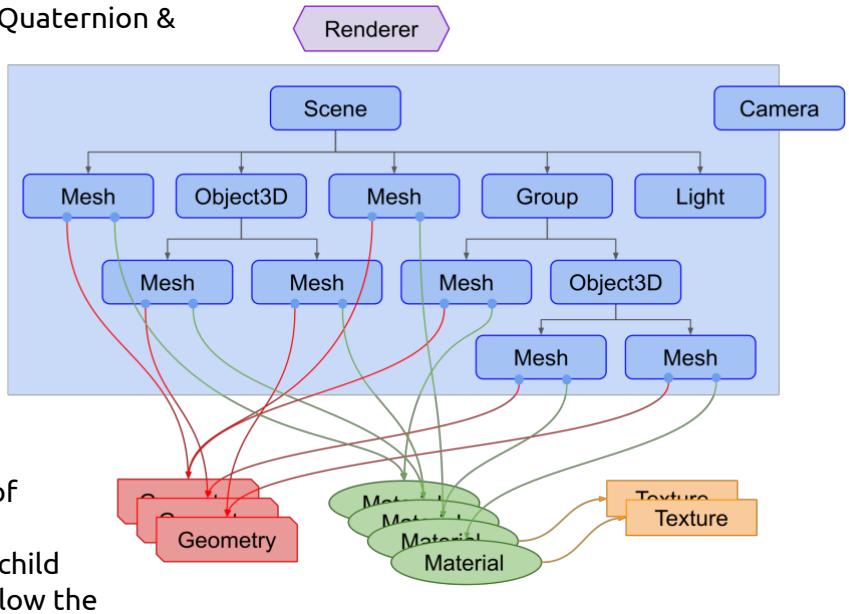

Figure 13 Diagram showing the structure of a simple three.js app [5]

Its tree-like structure serves as a hierarchy of nodes, with each node having its own local coordinate system. This means you can link child objects to a parent, and the children will follow the parent. An example of this would be an automobile in a simple game, where the wheel-objects are attached to the car body and will move together with it. [18]

The Camera object represents the user's view of the scene. There are two types – perspective and orthographic. While the latter can used to draw 3D objects without perspective warping, it's mostly in 3js to draw 2D things.
The former is the more commonly used camera, that shows, as its name suggests, perspective. That means items in the distance will appear smaller than those close by. It defines a frustum - a solid pyramid shape with a cut tip - with four variables, and everything inside it will be drawn by the renderer. The four variables are fov, near, far and aspect.

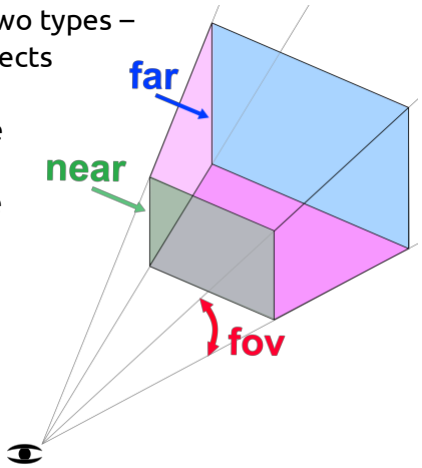

Figure 14 Camera frustum [5]

Fov is short for field of view and in three.js it represents the vertical angle of the camera. Note that while most angles in three.js are in radians, the camera uses degrees.
Near and Far represent the space in front of the camera that will be rendered. Anything before that range or after that range will be clipped (not drawn).
Aspect defines the size of the Near and Far planes. It can be set manually to match common video ratios like 16:9 or be responsive and scale with the browser window. [19]

Finally, the Renderer is responsible for taking the data from the Scene and Camera objects and rendering it out to the canvas. The one used for 3D Canvas scenes uses WebGL 2.0, but there is also a legacy WebGL 1.0 Renderer, alongside a CSS 2D and CSS 3D renderers, for when you don't want to use the canvas. [17] [20]

2.3 Interactivity

When creating regular web content with HTML, CSS & JS, there are common ways to let the user interact with your webpage. HTML Buttons and links, CSS pseudo classes and JavaScript event listeners; the list goes on.

Working with 3D is different. Your mouse moves on a 2D plane, so it's necessary to find a way to translate the XY coordinates. The solution for this is to cast a ray from the mouse's position into the 3D scene from the perspective of the camera and see what it intersects with.

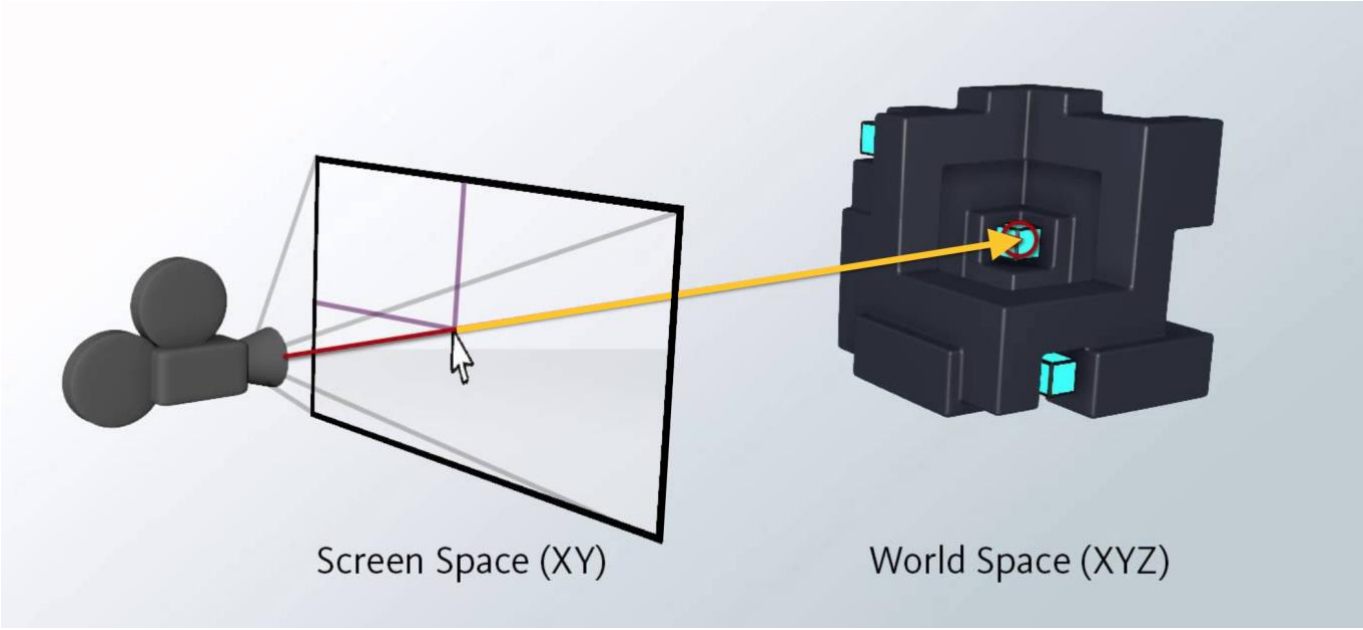

Figure 15 A diagram visualizing a ray cast from a mouse position in screen space intersecting an object in world space [21]

This technique is known as Raycasting, and Three.js provides a built-in function for it. It returns an array of objects in the order that they have been intersected by that ray. [21] [22]

Three.js also provides a variety of ways to control the Camera's position and orientation in the scene. The non-exhaustive list includes ArcballControls, FlyControls, OrbitControls and TrackballControls.

FirstPersonControls and PointerLockControls are commonly used for game-like environments, where instead of moving the camera around an object or scene, you move a character in the scene.

To controls objects inside a scene, Three.js has DragControls and TransformControls. [23]

An interesting point is that because Three.js runs fully in the browser, you can use JavaScript Event Listeners to link the 3D to the HTML and perform specific actions when the user clicks on an html button or scrolls to a specific point. [24]

2.4 Performance optimization

With mobile devices being the most common way to access websites, power consumption becomes a concern. A possible solution would be implementing Three.js' Rendering on Demand functionality, to only render when the user is actively scrolling on the page, and simply keep displaying the last rendered frame when he stops. [25] [26]

Optimizing 3D rendering performance is an art onto itself, and not really in the scope of this thesis, however a few tips include: attempt to reuse objects as creating new ones in JavaScript can be expensive. Avoid doing too much work during the render loop. Always use BufferGeometry instead of regular Geometry. Keep your scene centered around the origin to reduce floating-point errors at large coordinates. Where possible, merge multiple objects into a single mesh to reduce drawcalls. A more comprehensive list of tips and tricks can be found in the citations. [27] [28]

2.5 Map data

Digital mapping has surged in popularity in the early 2000, and continued ever since, coinciding with the rise of portable computers and smartphones. And with that, it brought many ways to digitally store and access them.

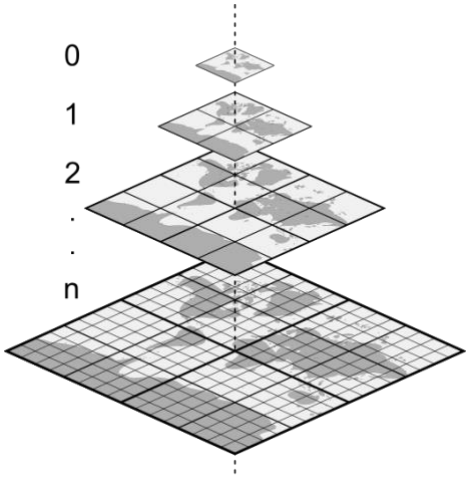

Figure 16 A diagram showing multiple levels of map tiles [29]

A common way to access map data are tile servers. They split the world up into square tiles and are usually accessible using a serverexample.com/{z}/{x}/{y}.png scheme. With this example, Z is the zoom level of the map, with X & Y being the tile numbers. So, a higher Z would result in more details of a smaller region.

There is a distinction to be made between Raster and Vector tiles. The former provides you with an image, usually in PNG format with a resolution of 256x256, while the latter will be a vector, which is a type of image build by mathematical formulas, which can be scaled up infinitely. An additional benefit of vector tiles is that on average, they are about 20-50% smaller than the corresponding raster tile, resulting in faster loading on the web. There are countless tile servers available on the internet, but a few popular ones include ones by OpenStreetMaps, Google Maps, Here Maps and Bing Maps. [29] [30] [31]

Due to the rising popularity of 3D maps, a variety of companies and projects have sprung up that focus on delivering the necessary data to generate them. They include but are not limited to AccuCities, Cesium and OSM Buildings and 3D Buildings, and come in a variety of formats, from vector data in GeoJSON format that describes a building's outline to individual 3D models of buildings and whole cities. Some even provide ready-to-go web integrations you can drop onto your website. [32] [33] [34]

It is also often possible to download map data of a specific area from various other sources, like government websites or community projects. A good example of the latter is Overpass Turbo, that queries the OpenStreetMaps database directly, and allows you to specify what data you want to be included. [35]

2.6 Generating a 3D city

There are many different ways to recreate a real-world environment in 3D.

There is photogrammetry, which uses tilted photography from multiple angles to create a textured 3D shape of the object. This technology plays a major part in Microsoft's 2020 Flight Simulator to recreate the entire Earth. This option was discarded for web due to the very high computational and size cost but could theoretically be implemented by using raster tiles to generate the 3D models, and then importing those models to 3js.

While you could use vector tiles that include a building's outline, and simply extrude it, it would not look good without additional metadata like the individual height of buildings, their roof type, etc…

A possible solution would be providers like Cesium and 3DBuildings, however they either have their own rendering engine and only provide a complete solution or are too expensive. In addition, the performance might suffer when using three.js, because of the relatively high level of detail they provide, especially on mobile devices.

A better option would be using OpenStreetMaps' "Simple 3D Buildings" format. It uses GeoJSON to describe the volume of a building using its outlines and provides the additional metadata necessary to generate it, like the amount of floors and rooftype. In addition, data of a specific area can easily be downloaded in GeoJSON format from Overpass Turbo. [36] [37]

3 Technical research

As discussed in chapter 1, the goal is to create a minimal proof of concept that will generate a simplified 3D City environment from a dataset and animate a path through it. This will be realized in multiple steps described below, starting from the basics of three.js, to parsing the OpenStreetMaps data to create the necessary shapes alongside performance optimization. The theoretical and technical research was carried out in parallel with the technical research continuously being adjusted with gained knowledge.

3.1 Contract

This research was based on a contract that included the desired output and success criteria. The former describes the execution of the technical research:

"A page that shows a map of a city with generated 3D buildings that, when the user scrolls down will animate the path of a tour, with the camera following it as it animates.
I will also make a separate page that will allow an administrator to create new paths.
The application should use OpenStreetMaps data and should remain performant."

while the latter is a summary of the requirements for a minimum viable product:

- *Generate a 3D city from available data*
- *Performance must remain at 30fps or higher*
- *Animate a path on scroll, and have the camera follow it*
- *As the user scrolls by the Points of Interest, show information about them*

[38]

3.2 Choices

To work with node and npm packages I chose to use Vite.js. Vite is a build tool that aims to provide a faster and leaner development experience for modern web projects. It consists of two major parts:

- A dev server that provides rich feature enhancements over native ES modules, for example extremely fast Hot Module Replacement (HMR).
- A build command that bundles your code with Rollup, pre-configured to output highly optimized static assets for production.

Vite was chosen due to my pre-existing familiarity with it from using it on some Vue.js projects in the past, where it has proved to be fast and reliable. [39]

Given its popularity, three.js was chosen as the preferred framework for this project, as that popularity gives rise to a large amount of documentation and tutorials.

3.3 Development

Setup

Because I wanted the ability to integrate this research project into my own website later, I decided to develop it as an NPM package. This involved making two separate projects – the primary one for the actual 3D application, and one for a test website that would implement it.

In the project folder, the `npm init` command was used to create a `package.json` file, that would contain the package's metadata such as its name, version, dependencies, entry point and other information. `Index.js` will serve as the entry point to the package, with the `src` folder containing the code and `sample` folder containing default assets.

The plan was to split the functionality into individual JavaScript modules for better clarity and maintainability, which ended up being initialize, globals, city, animate and path.

Initialization

Starting with the initialize module, it, for a lack of a better term, initializing three.js. The `initialize()` function creates and configures the `Scene, Camera, Light and Renderer` objects, selects the canvas element from the `DOM` and then attaches the `renderer` to it.

Additionally, it is used to enable or disable debug information like an FPS counter and Axis visualizer.

Later, the module was used to initialize `MapControls` and create the animation loop, but more on that in the Interactivity section.

This npm package is called and will be referred to as **Storymap**.

Demo Project - Admin

To test whether the newly created npm package was working, `npm create vite@latest` was used to initialize a separate demo project that would serve as the consumer of the package.
To install the local package in this project, `npm link` was used to create a symlink to storymap in the `node_modules` of the demo.

This project will be used to create and test the "administration" panel, where the web developer could create paths for new or updated tours.

After creating a `canvas` element in the `index.html` file and adding it to storymap's configuration it's possible to start testing by typing npm run dev in the terminal and going to `localhost:3000` in the browser, where we see an empty three.js scene.

Because the storymap package is symlinked, the changes we make to it will automatically propagate to the demo project, so it is possible to keep the demo project running, and continuously write code and test changes, which results in an efficient developer process.

Finding and importing data

I initially planned on using OSM Buildings to get the data, however I found their documentation to be outdated, and I decided to switch to Overpass Turbo so I could start development sooner and figure out how to use OSMB later. OSMB uses GeoJSON, and Overpass Turbo allows exporting its data in that format.

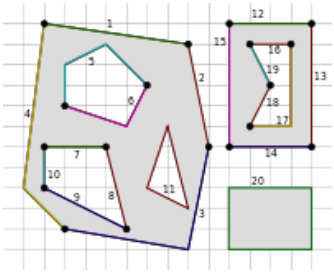

Figure 17 Outline of an object in GeoJSON [60]

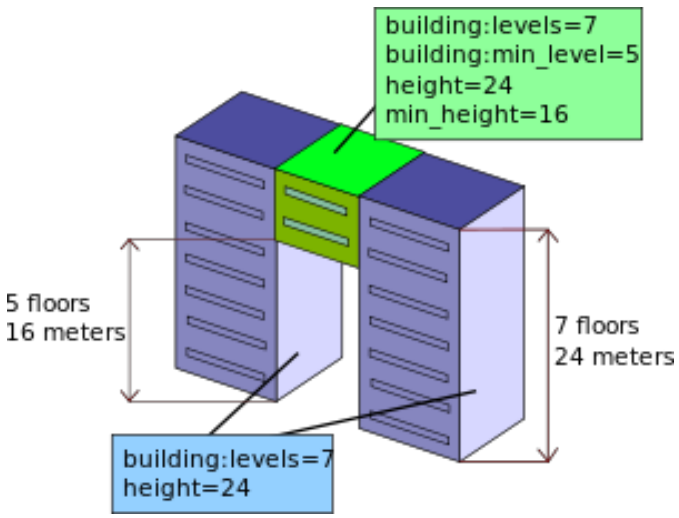

Figure 18 Example model of a building and its metadata in 'Simple 3D Building'-format [59]

GeoJSON files contain various elements of the map like roads, parks, buildings represented by an array of various coordinates and metadata.

Buildings have outlines, holes, section, height or levels, and many other attributes.
JSON is a good choice as it's prevalence on the web means native support for parsing it.

Figuring out how to use the OverPass API took time, as the documentation doesn't make it clear how to structure its query language, and using it remains challenging.

Rendering

The generation of buildings was split out into the city module. It parses the GeoJson file looking for buildings. The buildings are represented as an array of latitude and longitude coordinates of a polygon that forms the outline of the building, eventually more polygons that represents holes in that shape, like an inside garden or overhang.

The first big challenge was caused by JavaScript itself, alongside the coordinate system in Three.js. The scene's center is represented by 0,0,0 and the further away you move from it, the more inaccuracy and instability you will have, caused, among other things, by JavaScript's poor floating-point precision. The GeoJSON coordinates are in global latitude and longitude coordinates which are large floating-point numbers with a very small differences, which exacerbates the precision issue. [40]

Because of this, it was neccesary to normalize the global coordinates to local space. Taking the center of the area, a 3rd party library called geolib was used to calculate the distance from the center to each point, resulting in coordinates normalized to an origin, with a much larger standard deviation.

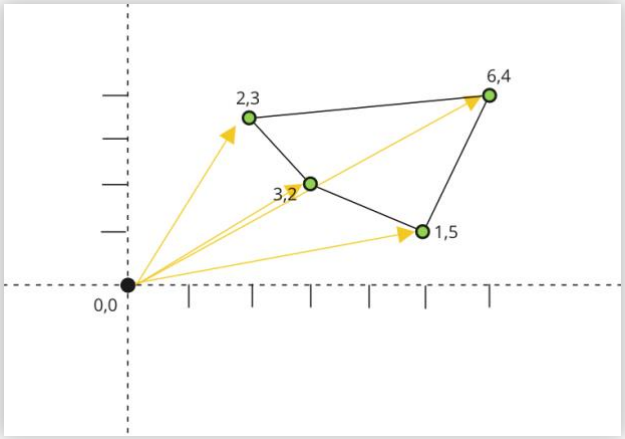

Figure 19 Grid showing coordinates relative to an origin.

With this normalized data, I could then create a Three.js Shape and Geometry. Using Three.js' built-in function, I was able to easily "punch out" the holes from the outline. This Geometry, together with a Material was then used to create a Mesh, and subsequently added to the scene. Due to three.js's XYZ orientation, I also had to rotate it 90 and 180 degrees in the X and Z axes respectively.

Using this approach, I now had a very flat first version of my city. To make it 3D I used the levels attribute which represents the number of floors multiplied with an arbitrary to extrude those shapes.

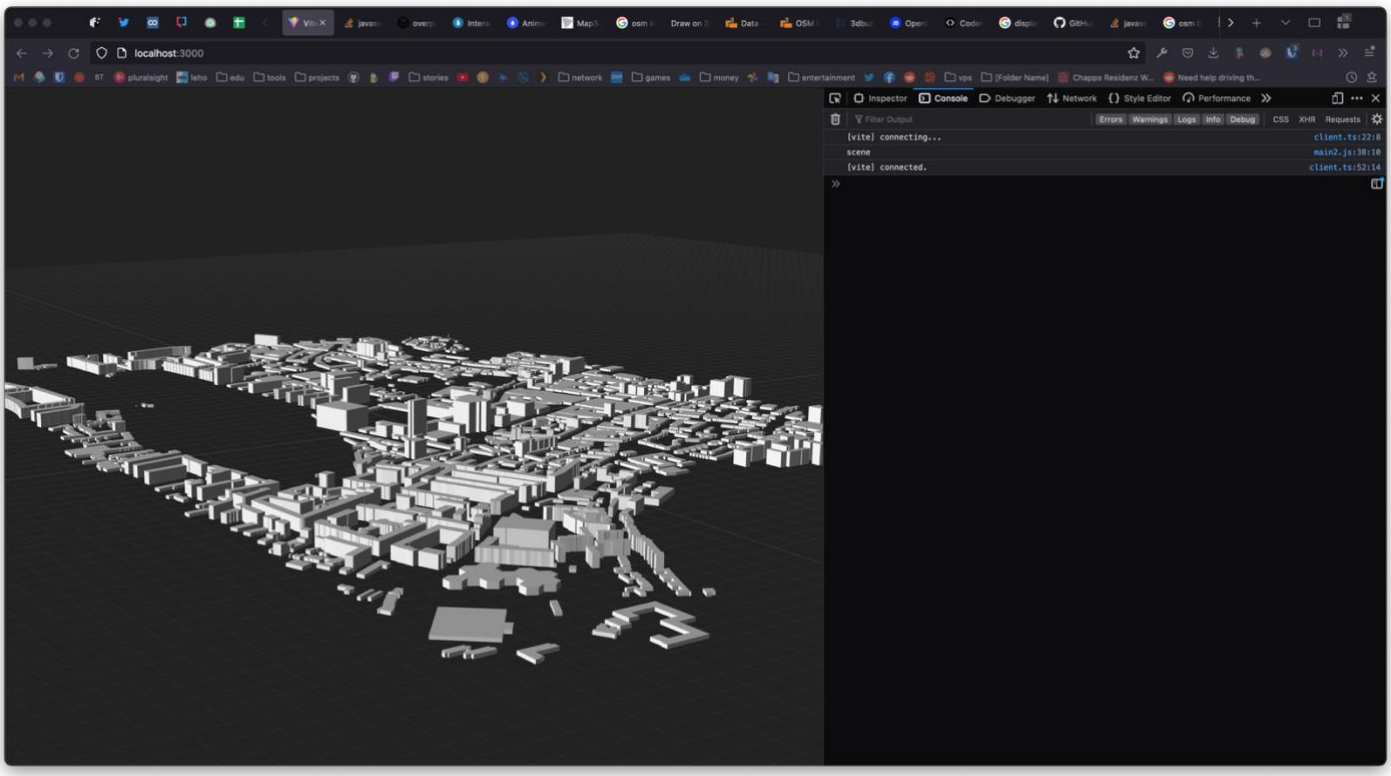

Figure 20 A screenshot of a city being rendered

Performance

Now that the city was generated an issue popped up – the performance was terrible, with the fps going down to single digits. The solution was to merge all the buildings into one big mesh, instead of spawning individual objects for each building. This approach causes a side effect that all buildings will have the same material, but it was worth it due to the massively improved performance. This fix was applied to all geometry that was generated later like roads, green areas, and waterways.

Interactivity - Admin

The web developer needs to be able to load and browse the whole area and set waypoints for the tour path. This can be achieved by using 3js' Map Controls.

It's a subset of 3js' Orbit Controls that works in a similar way to Google Maps and other popular digital mapping software, allowing the camera to orbit around the center of the page using the right click, and drag the camera around using the left click. They were imported and initialized in the initialize module. To then make them work, they need to update every frame, so they were added to the animation loop. [41]

For creating the waypoints for the path, you need to have roads to select, and a way to translate 2D mouse coordinates into 3D space. For the latter, a Raycaster can be used to cast a ray from the mouse position towards the 3D scene and see what intersects it. From there, you can filter it to only be able to select roads. [42] [43]

For the former, a similar technique was used for the buildings. Using Overpass Turbo instead of OSM Buildings turned out to blessing in disguise, as OSM's API only gives GeoJSON of buildings, so I would've needed to find and implement an additional API to get the road data. The Overpass API query was adapted to include roads, their coordinates were normalized, and they were generated and added to the scene in the city module. Because the city looked drab, OverPass was also used to get data for green areas and waterways.

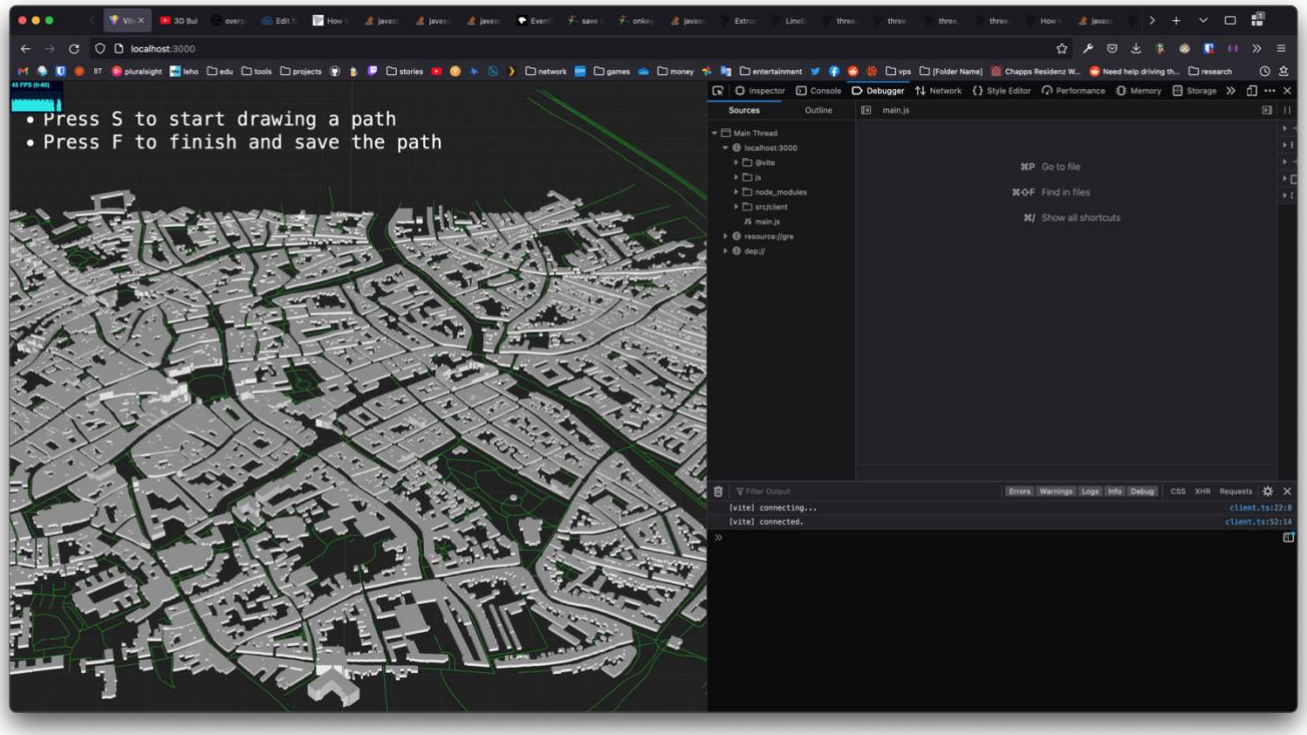

Figure 21 Screenshot showing the application also having roads

A nice feature to have would be the ability to select specific buildings or points of interest, and have the application automatically calculate the shortest path for you, like how Google Maps and other mapping software works (instead of having to manually draw the path). A possible way to calculate the shortest path would be implementing Dijkstra's algorithm into the application. However this task goes too far beyond the scope of my research and would require significant extra time [44]

Path

The path module was used to implement the Raycaster and other necessary controls to draw and export the path.

Three.js has a built-in `Raycaster`, so it was easy to implement it. It is linked to an `OnMouseMove` event listener to continuously cast rays, to show whether the user has a road selected or not.

Left mouse click is used to set a waypoint, with various keyboard buttons responsible for Saving, Resetting, Undo and Redo functionality. The path exports into a JSON array.

Once the developer has started creating a path, I need to make it follow the mouse. Because of this, until clicked, the last point of the path is linked to the mouse position, also using the `OnMouseMove` event listener.

Figure 22 Screenshot of the application, with a path being created

Demo Project - Client

To test the client-side implementation, I created a new project and configured it in a similar manner as the admin project, using vite to bootstrap the project and using npm link to install the storymap package.

The one change was that this time, instead of taking up the whole page, the scene took half, with the other half being used to show information about the Points of Interest.

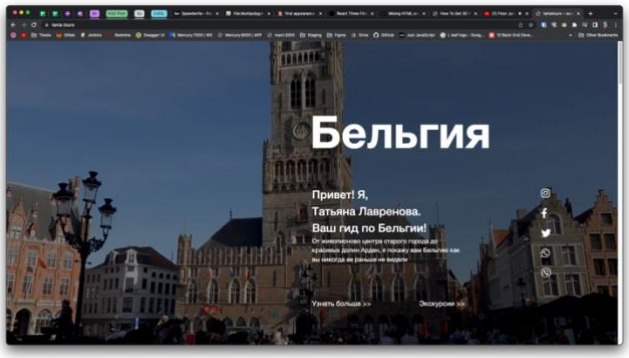

Figure 23 Homepage of tania.tours, the website I was working on at the time.

Interactivity & Animation – Client

For the client, the only control they should have over the interaction is scrolling in the browser. They shouldn't be able to interact with the three.js scene directly, like how the users can't interact with the canvas in Lucas' demo. [2] [24]

In short, when the client scrolls down in the browser, the path should appear and animate out from the starting position and finish once the user scrolls to the bottom with the camera following along. This was one of the most difficult parts of the project

To summarize, the path is drawn as one Line object with multiple points. As the user scrolls down, the last point is continually updated to make it appear to animate. To get the coordinates of this point, a `getScrollPercentage()` function is attached to the scroll event listener and used to calculate the browser scroll position. This percentage is then used to continuously manipulate which pair of points is selected as the start and end point, with the `lerpVectors` function used to interpolate between them and calculate what the current last point of the path should be.

Let's make it clear with an example. Let's take a path with 11 individual points. The first one is drawn from the start, so you must divide the total page height in percent by 10.

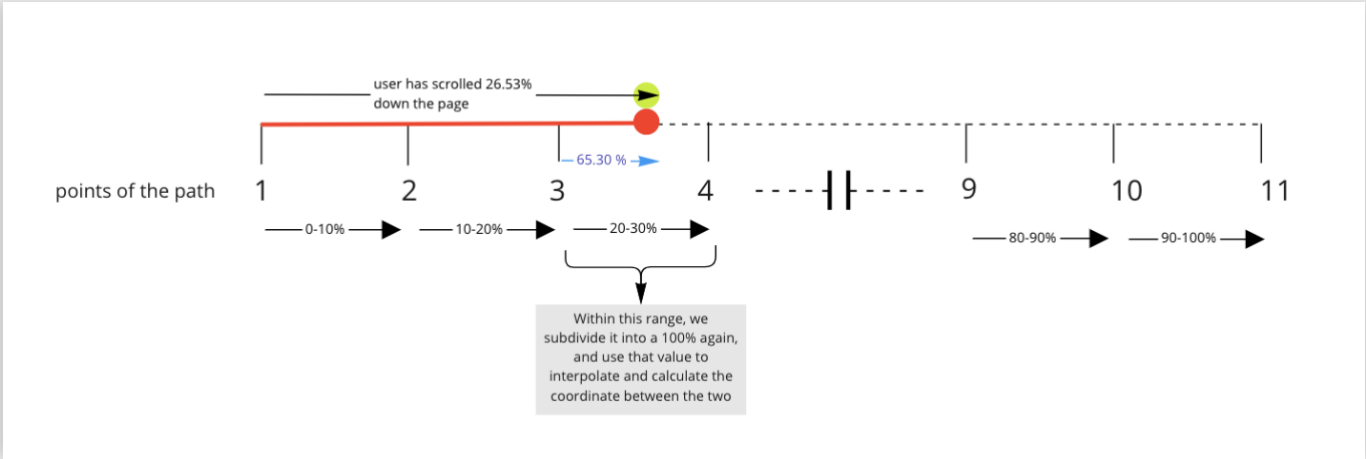

Figure 24 Diagram showing how the path is calculated

1. This means that for every 10% we scroll, we pass a point. The global scroll percentage determines between which pair of points we need to interpolate.
2. If the user has scrolled to 26.53 percent, we need to, using interpolation, calculate the coordinates of the point at 65.3% between the 3rd and 4th points.
3. This coordinate is used to update the last position of the Path. This happens every time the user scrolls, giving the illusion of the path shrinking or growing.
4. The camera's position was offset to behind the to the side of the path, with the last path position acting as the target, and it was moved together with the path.

Final Result

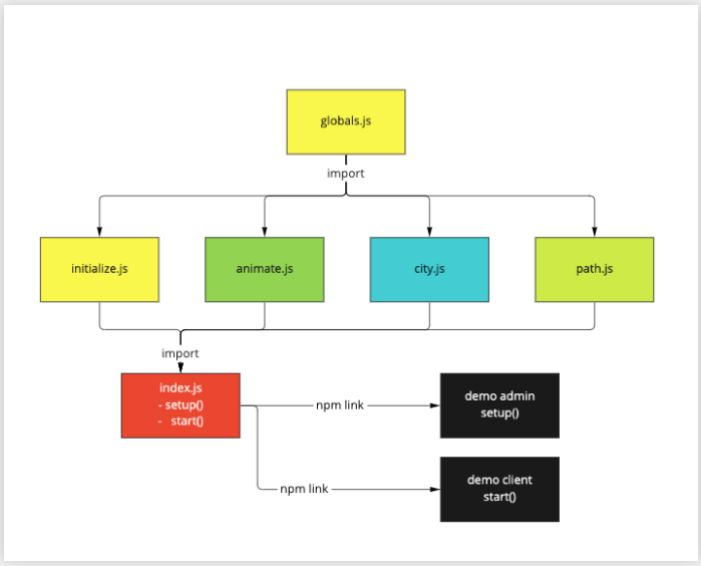

Figure 25 Structure of the project

The final result of the research consists of 3 distinct projects: the storymap npm package that is responsible for generating and parsing the map data, an admin project that allows for the creation of paths on the map, and a client project that animates the created path through the city.

It proves that it is indeed possible to use Three.js to create a 3D City environment and animate a path through it.

3.4 Demos

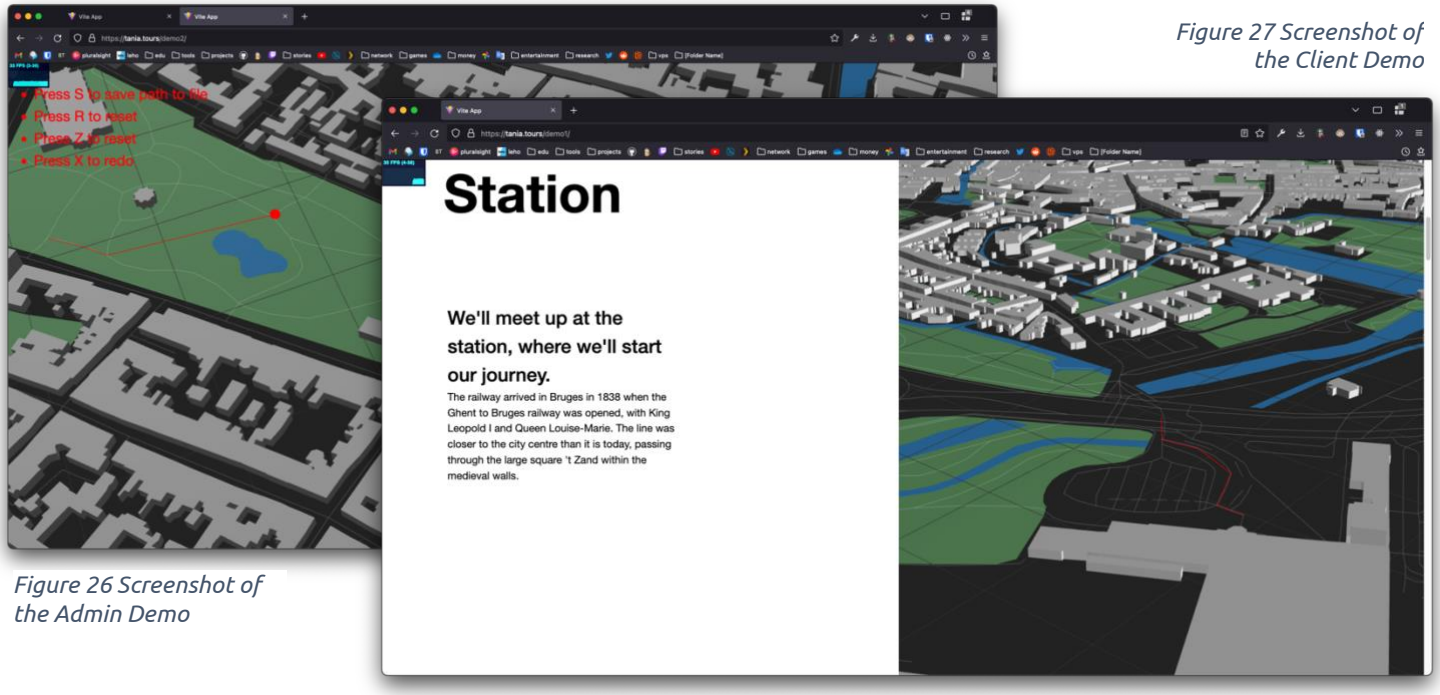

Figure 26 Screenshot of the Admin Demo

Figure 27 Screenshot of the Client Demo

Demos are available online at:

- https://tania.tours/demo/client
- https://tania.tours/demo/admin

The source code is available on Github at https://github.com/storytellingmap

4 Reflection

This section is meant to illustrate my thoughts and analysis of the achieved result and research, including conversations with a variety of experts for a fresh look on the project, including their feedback, suggestions, and advice.

4.1 Proof of concept

While the proof of concept checks all the boxes, I was disappointed in the final result. I hoped the experience would be more impressive than what it is now. I wanted to create a "wow" effect, something like this scene below from Indiana Jones, but unfortunately that is not the case. The effect only works well on a macro scale.

Figure 28 Scene from Indiana Jones, showing a path animating across a 2D world map [64]

it would be interesting to to develop and optimize the project further and add additional features, like optimizing and improve performance further. However it lays beyond agreed scope and allowed time frame for the project.

A lot of time was spent figuring out and learning three.js itself. As it's still a 3D library, it uses a lot of math I was not familiar with, like matrix multiplication and Quaternions, and familiarizing myself with it all took a toll.

The feedback during the presentation also showed that. The path was hard to see, and the camera movement was jerky, resulting in a poor user experience which is understandable in PoC but would need to be addressed in a final product. The administration page was sufficient for PoC but needs to be further updated before it would be usable in production. I expected this PoC to be closer to a usable production version, but it has shown limited value caused by low performance.

Ultimately, it checks the boxes, but not the spirit.

4.2 Problems

Aesthetics

By choosing to go for a simplified extruded look, I shot myself in the foot. I now understand why nearly all 3D City demos use cities with a lot of skyscrapers. They fit the style well.

Older Belgian cities are flat with few height variations, which looks boring. Another issue that without dedicated models, buildings like churches and cathedrals look like big skyscrapers and look very out of place.

The general aesthetics are also not up to par. The city can look empty and boring. Instead of using Overpass Turbo to get the green and water areas, a better solution would've been using Raster Tiles as a ground texture, with roads, waterways, parks and everything in between.

Performance

Even though the fps generally remains above 30, the experience doesn't feel smooth. Even the simple 3D buildings have issues on older hardware, and you would need to add more detail for it to look

pretty. I would've liked to get a stable 60 fps and I'm sure with enough optimization it would be possible.

The downside of using Overpass Turbo rears it's ugly head in loading performance. Because everything is put into a single .json file, loading and generating the city on the web can take multiple tens of seconds. It works okay locally, but ideally this should be fully reworked.

UI & UX

User experience has been a problem from both sides – end users and web developer. The feedback from a would-be administrator was that manually placing points on the road was a pain point, and they would like a more Google Maps -like experience with waypoints, so they could just select the specific building or points of interest and have the route be auto-generated.

From the end-user perspective, the current camera motion is jerky and not an enjoyable experience. They would also like a better link/integration between the information about the point of interest, and it being highlighted on the map. The experience currently also required the user to use their phone in horizontal mode, the feedback was to integrate the POI information into the three.js scene so they can keep their phone vertical.

Firefox was used during development, and the application sometimes breaks in other browsers, which requires debugging. In addition, the CSS does not properly account for a vertical mobile layout, necessitating the user to turn their devices horizontally.

4.3 Improvements

As mentioned in 4.2, being able to highlight and select individual buildings or locations would make it much clearer for the end user what the actual POI is.

Implementing a "shortest path" algorithm like Dijkstra would ease the burned on the administrator for creating new paths. This would combine well with highlighting buildings. If implemented, showing a list of created points with the ability to re-order and delete them would be nice as well. Automatically rotating the camera to keep the path un-occluded would also be a nice feature.

Moving away from Overpass Turbo to tile based GeoJson APIs would make the loading much faster.

Adding "vanity" objects like trees could make the city look a lot more lively, but you would need to take performance into account. In addition, replacing the flat ground texture with a raster tile would also immediately add a lot of information and variety to the city at little performance cost.

While three.js has basic frustum culling, implementing occlusion culling might results in performance gains.

4.4 Usability

Because I decided to develop it from the start as an npm package, it's actually relatively easy to integrate it into an existing application. However it would require first to implement the improvements mentioned in 4.2 to reach a state that would be suitable for public use.

4.5 Suggestions for further research

As the effect seems to work better on a macro scale, I suggest exploring generating a 3D earth and in general pivoting the project for "larger" stories, like Lucas Better's original demo or Figure 7. Maybe even abandoning 3D entirely and doing it in 2D.

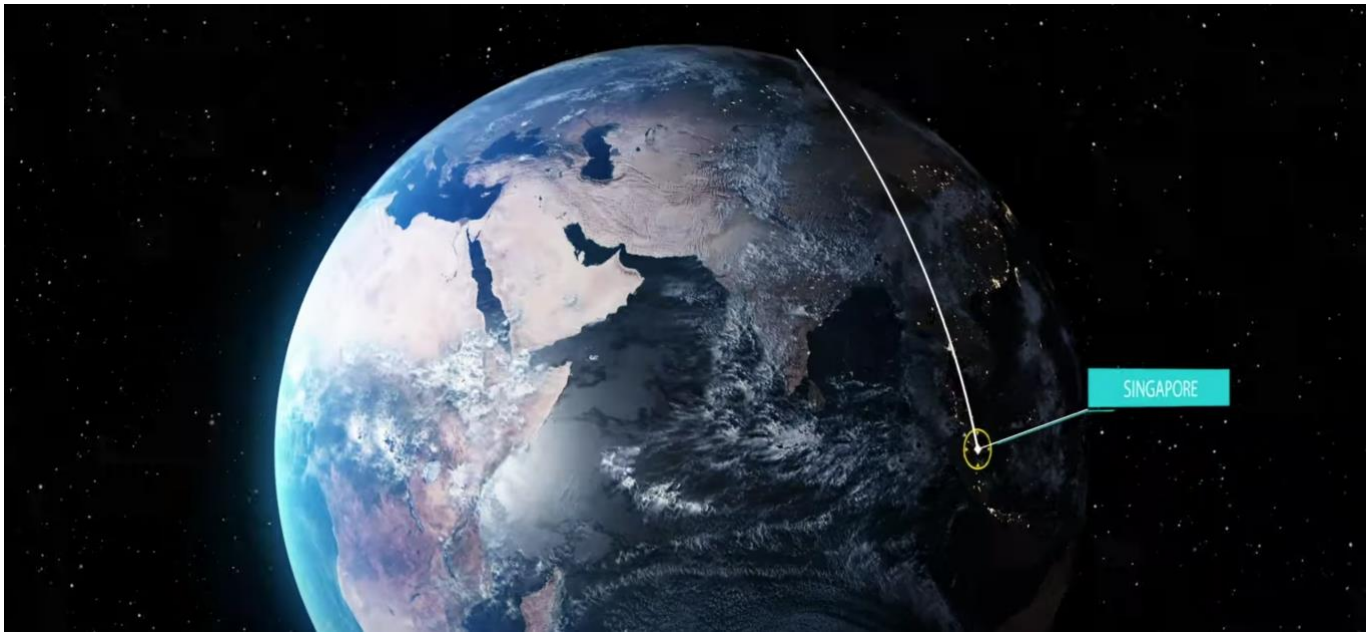

Figure 29 a 3D earth with a path going from Singapore to New York [45]

Another option is using the tech itself to show more than just paths. You could adapt it for data visualization such as traffic, population, etc.

While it's still early days in WebGPU's development, early testing shows a significant performance gain, so it's definitely worth looking into. Once it's released it's very likely Three.js will adapt it as an optional Renderer.

A possible solution for flat cities looking uninteresting would've been getting the roof-type from Overpass Turbo and generating custom roof shapes instead of the flat top that they are now, but that's something left for future research and development.

4.6 External feedback

Sergey Andreev thinks that given the package is open source, publishing it to an npm repository even in its current state should help development, as no-one wants to deal with npm link to just view a demo. He also thinks the path is poorly visible and suggest adding a floating arrow at the end of it. Another suggestion is drawing it on top with transparency when it's being occluded by buildings. When using the client-facing experience, he's also not sure oh where on the map the described buildings are located and suggest those buildings should change color as you get closer and further away from them.

As far as the code itself goes, he thinks it's a promising visualization library. The code is – while being relatively complex – very readable and well structured. However, if this project is to grow any larger or require rapid iteration, things will need to improve even further.
Dealing with 3D, even with the aid of an API or library is a historically complex problem, and the larger the code base gets, the harder it will be to manage. He recommends using TypeScript to alleviate some of the issues that come inherently with JavaScript.
The API could use some work, as it stands now implementing this library into a project is a very involved matter. He suggests thinking to think about what kind of work it would take to implement the things mentioned in the first section and making the API more flexible to aid those changes.

Azat Omarov suggest looking into the react-three/fiber library, which is a React renderer for three.js Given the growing complexity, it might ease the maintainability of the code and make it easier to add or update the various features and will likely make testing easier.

Emile Goeminne felt that with the knowledge he had on Three.js and WebGL the thesis has the right fundamentals and explains it correctly. He adds that he would've liked to see a higher focus on load speed from the start and agrees that implementing a tile-based API is the likely solution.

Thiemo Seys mentions that the demo breaks in certain browsers, which is a problem that needs fixing. He also suggests that maybe a regular website is not the right place for it, and that this project would be more suitable for something like a museum display, showing the history and events in a specific area. He finds that the city in its current iteration feels lifeless and would require more polish to look good. He agrees with Sergey's advice that buildings need to be highlighted.

Daniil Voloshin follows Thiemo Seys in his advice, suggesting that I implement topology for height variation and use additional data available from OpenStreetMaps like trees and statues to spruce up the city, pointing to Apple Maps as a good example.

From the financial standpoint, Daniil notes that such a project may be feasible and of interest to mid- and large-size companies in the tourism sector (for example, museums, tour agencies, etc.). Such firms have sufficient resources to implement all of the refinements that are necessary to transform this product from an MVP into an enjoyable experience.

Figure 30 Screenshot from the Apple Maps website, showing a 3D map with hills and other details [63]

5 Advice

This section aims to provide advice to people who are planning on working on a similar project. It includes a list of recommendations and ideas for people who want to develop an application that uses the technologies and libraries learned in this thesis

5.1 Is 3D necessary?

Creating a 3D website is no small undertaking. It requires a lot of uncommon – especially for web developers – skills, and it will be difficult to find a good team. Implementing a custom solution will bring a significant time investment with a corresponding cost.

You need to be very sure that your problem requires a 3D. For example, most online configurators would be better served with using pre-rendered images and combining them to form the product, instead of rendering it live in 3D.

Thus, I strongly advise making sure to explore all 2D avenues first, as they would be much easier to implement, and be available to a larger audience due to the lower performance requirement [28] [46]

5.2 General 3D Advice

If a 3D solution ends up being required, the first piece of advice is to check whether you cannot adapt and existing solution to your needs.

Many problems, like simply displaying a model or an area, have already been solved and are available as embeddable solution from the open-source community as well as a variety of companies. Emile pointed to Google's ModelViewer web component as one such solution. [47] In the case of mapping, there are companies like 3D Buildings, Cesium and 3D Mapper already offer extensive solution that you can embed on your website through code snippets, with a high amount of customizability.

You need to pay attention to the various ways your customers will and won't interact with your website. With the ever-rising popularity of mobile devices, the majority of your users might not have a precision pointing device or an easily accessible keyboard, so you need to take that into account form the very start.

The rise of mobile also means that power consumption is important. You don't want to drain your user's battery. This means not rendering continuously in a loop, but only when necessary. In my example, as the city itself is static, it is possible to stop rendering once the user stops scrolling and keep displaying the last frame until the user interacts with the page again.

Ideally you would go further and have a variable level of complexity that could be automatically adjusted depending on factors like the user device's performance or battery level. With the latter, you could limit the app to 30fps instead of 60fps to save battery.

5.3 Solving the problems

Good aesthetics are key to making the user experience enjoyable. So if you plan on building a 3D city, make a lot of detailed mockups to hammer out all the necessary details. The API, color palette, the level of detail, shadows, etc…

Aesthetics

To possible way to make flat cities look interesting is increasing the level of detail. OpenStreetMaps has a large amount of additional data I have not implemented in this project, but adding unique roof-types to the buildings will help vary the skyline, as well as miscellaneous objects to the city such as trees. Parking spot information could be used to procedurally place cars onto the roads.

Do not underestimate the importance of aesthetics. Having worked together with Daniil to change the color palette of my demo already massively improved its appearance, making it look much livelier.

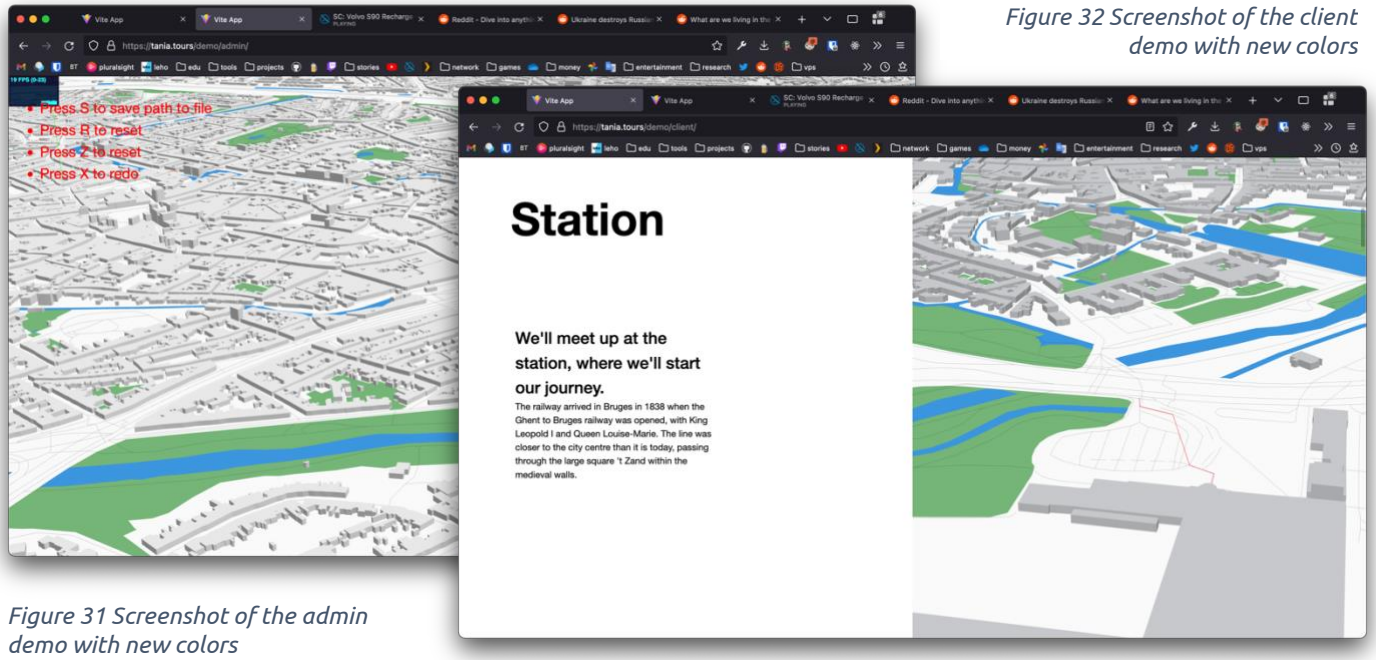

Figure 31 Screenshot of the admin demo with new colors

Figure 32 Screenshot of the client demo with new colors

Performance

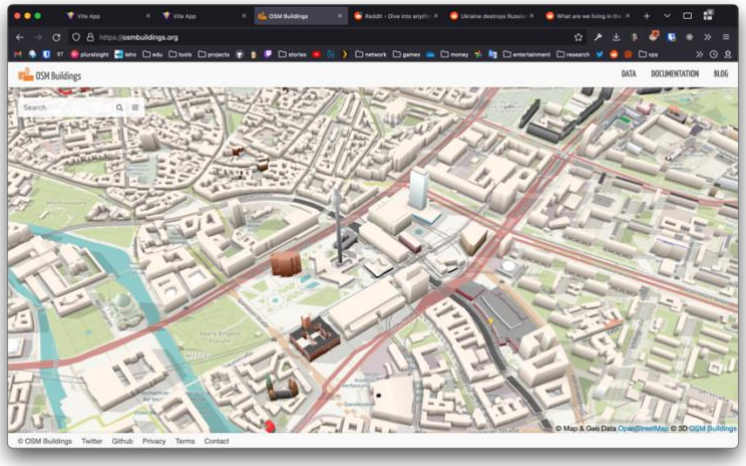

Figure 33 Screenshot of OSM Buildings, showing a combination of raster and vector maps alongside 3D [65]

To improve performance further and improve aesthetics, you could avoid rendering roads, waterways or green areas in favor of using a raster tile as a texture to go under the 3D buildings. The vector road network could still be present in memory to allow creating a path, but it would not need to be rendered, and thus won't affect performance.

Continuously test performance on less powerful hardware. A City is a very complex environment, and every change you make can have a big impact on performance. I agree with Azat and using the React Renderer for Three.js instead of using it natively. As Sergey said, 3D is complicated by its very nature, and you should take every advance to make your source code well-structured and maintainable. Using the React renderer will also allow to write more automatic unit tests. [48]

Use a good Tile based API from the start. Ideally make use of multiple APIs for redundancy. In addition, a high expiration times, so the mapping data is cached by the user, making the next visit better. This will not cause issues, as large changes in the real world take time.

Do not forget to test if the app works in a variety of browsers. Just because the spec says compatible, doesn't mean that it will work the same between different implementations, as seen by my demo breaking in some versions of Chrome.

UX

To make the camera motion smooth, you could generate a Bezier curve based on the path, with a maximum allowed angle-size, and use it for the camera's path, which would make its motion much smoother. In addition, you could use the Raycaster to figure out when the path is occluded and create a function to automatically rotate the camera, so it always remains visible.

To let an administrator, select individual buildings, you could create a bounding box of each and keep it in memory, without rendering it – thus having a minimal impact on performance.

To better link the information about the points of interest to the 3D map, a similar method can be used to detach individual buildings from the large mesh to color them in. Alternatively, it might be possible to write a shader that will only color specific buildings.

To display the information itself, you could use the "Aligning HTML elements to 3D" tutorial from the Three.js' documentation, or handle displaying everything in 3D. You could do a combination of the two, rendering HTML content inside the 3D scene. [49] [50] [51]

6 Conclusion

This thesis was created to critically evaluate my research project that answered the question "How can you use Three.js to display an interactive map of 3D Buildings that visualize a path of a tour". The aim of the project was to create an npm library based on Three.js that would process map data and generate a 3D city environment and then allow the user to create & animate a path through it.

This document explores the Three.js library, an abstraction layer above WebGL, the technology that enables 3D in the browser, and looks at the inner workings, compatibility, and feature set. It investigates the state of digital mapping, the data providers and formats involved, alongside the variety of ways 3D is currently used in that industry.

This paper describes how a demo project was created that implements the functionality necessary to meet its goals and goes in-depth about setup, interaction, and animation in Three.js.

While the ultimate goal was to amaze potential visitors, the final result lacks the "wow" factor I hoped for. Achieving quality aesthetics, performance, and user experience was difficult, as 3D requires a very different mindset compared to regular web content.

The demo does not exceed the boundaries of a proof of concept, though looking at similar projects from established companies like OSM Buildings and Cesium certainly proves there is potential to make a performant and attractive product, given enough time and a large enough budget.

In the reflection and advice sections, we discuss the problems and potential solutions in detail, like mixing raster, vector, and 3D maps together, and the types of companies that would be interested in developing this technology further.

This thesis also serves as a warning to businesses who are evaluating 3D as a possible solution, as it adequately describes the amount of effort that will be required to achieve a custom solution, and suggests investigating both existing tools alongside 2D alternatives.

Despite this, I consider Three.js to be a fantastic tool, and I recommend everyone to take the time to try it. Finally, I can with confidence say that yes – it is possible to use three.js to create a 3D city environment and animate a path through it.

To summarize the answer in one sentence:

> *"It's possible and in this thesis I described how. However, I do not recommend it.".*

7 Bibliography

[1] AccuCities Ltd., "A 3D model of London, captured by aerial imagery in 2019," 1 9 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Digital_heritage&oldid=1071289954#/media/File:2019_City_of_London_3D_model.jpg. [Accessed 18 05 2022].

[2] L. Bebber, "Animated Map Path for Interactive Storytelling," Tympanus, Codrops, 16 12 2016. [Online]. Available: https://tympanus.net/codrops/2015/12/16/animated-map-path-for-interactive-storytelling/. [Accessed 18 05 2022].

[3] M. Potenziani, M. Callieri, M. Dellepianel and R. Scopigno, "Publishing and Consuming 3D Content on the Web: A Survey," Research Gate, 2018.

[4] OpenBase Inc., "10 Best JavaScript WebGL Libraries," [Online]. Available: https://openbase.com/categories/js/best-javascript-webgl-libraries. [Accessed 28 05 2022].

[5] G. Tavares, "WebGL Fundamentals," 08 05 2022. [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-fundamentals.html. [Accessed 19 05 2022].

[6] Google, "ANGLE - Almost Native Graphics Layer Engine," Google, 20 05 2022. [Online]. Available: https://github.com/google/angle. [Accessed 20 05 2022].

[7] A. Start, Interviewee, *Discussion in an online community.* [Interview]. 19 05 2022.

[8] MDN contributors, "WebGL: 2D and 3D graphics for the web," Mozilla, 27 04 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. [Accessed 18 05 2022].

[9] BigCommerce Pty. Ltd., "What is page load time and why is it important?," [Online]. Available: https://www.bigcommerce.com/ecommerce-answers/what-page-load-time-and-why-it-important/. [Accessed 28 05 2022].

[10] Khronos Group, "OpenGL," 31 07 2017. [Online]. Available: https://www.khronos.org/opengl/. [Accessed 28 05 2022].

[11] E. Manor, "The story of WebGPU — The successor to WebGL," Medium, 01 04 2021. [Online]. Available: https://eytanmanor.medium.com/the-story-of-webgpu-the-successor-to-webgl-bf5f74bc036a. [Accessed 20 05 2022].

[12] Three.js, "WebGPU examples," Three.js, [Online]. Available: https://threejs.org/examples/?q=webgpu. [Accessed 28 05 2022].

[13] W3C contributors, "WebGPU W3C Working Draft," W3C, 17 05 2022. [Online]. Available: https://www.w3.org/TR/webgpu/. [Accessed 19 05 2022].

[14] R. Cabello, "Three.js Whitepaper," 22 05 2012. [Online]. Available: https://github.com/mrdoob/three.js/issues/1960. [Accessed 21 05 2022].

[15] G. Tavares, "WebGL 3D Ortographic," [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-3d-orthographic.html. [Accessed 21 05 2022].

[16] G. Tavares, "WebGL 3D Perpective," [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-3d-perspective.html. [Accessed 21 05 2022].

[17] Three.js Contributors, "Fundamentals," Three.js, [Online]. Available: https://threejs.org/manual/#en/fundamentals. [Accessed 18 5 2022].

[18] Three.js Contributors, "Scenegraph," Three.js, [Online]. Available: https://threejs.org/manual/#en/scenegraph. [Accessed 22 05 2022].

[19] Three.js Contributors, "Cameras," Three.js, [Online]. Available: https://threejs.org/manual/#en/cameras. [Accessed 22 05 2022].

[20] Three.js Contributors, "List of Renderers in the Docs," Three.js, [Online]. Available: https://threejs.org/docs/?q=renderer. [Accessed 22 05 2022].

[21] path9263, "How To Get 3D Position Of The Mouse Cursor," godotengine.org, 06 04 2018. [Online]. Available: https://godotengine.org/qa/25922/how-to-get-3d-position-of-the-mouse-cursor. [Accessed 29 05 2022].

[22] K. Day, "Interactivity in Three.js," Medium, 06 04 2020. [Online]. Available: https://medium.com/@joshdaycalgary/interactivity-in-three-js-3c0ece39f424. [Accessed 29 05 2022].

[23] Three.js Contributors, "List of Controls," Three.js, [Online]. Available: https://threejs.org/docs/?q=controls. [Accessed 28 05 2022].

[24] S. Bradley, "Animate on Scroll," Three.js Tutorials, [Online]. Available: https://sbcode.net/threejs/animate-on-scroll/. [Accessed 22 05 2022].

[25] E. Enge, "Mobile vs. Desktop Usage in 2020," Perficient, 21 03 2021. [Online]. Available: https://www.perficient.com/insights/research-hub/mobile-vs-desktop-usage#:~:text=Mobile%20devices%20drove%2061%25%20of,increase%20from%2063.3%25%20in%202019.. [Accessed 29 05 2022].

[26] Three.js Contributors, "Rendering on demand," Three.js, [Online]. Available: https://threejs.org/manual/?q=render#en/rendering-on-demand. [Accessed 22 05 2022].

[27] discoverthreejs, "Tips & Tricks," [Online]. Available: https://discoverthreejs.com/tips-and-tricks/. [Accessed 29 05 2022].

[28] S. Szymczak, "IS HET MOGELIJK OM EEN INTERACTIEVE 3D MUSEUM TE MAKEN IN EEN BROWSER MET BEHULP VAN THREE.JS," Howest, Kortrijk, 2021.

[29] Flexberry, "layer Type tile," ember-flexberry-gis, [Online]. Available: https://flexberry.github.io/en/efg_tile.html. [Accessed 29 05 2022].

[30] OSM contributors, "Tiles," OpenStreetMaps, [Online]. Available: https://wiki.openstreetmap.org/wiki/Tiles. [Accessed 22 05 2022].

[31] M. Elias, "Raster vs Vector Map Tiles: What Is the Difference Between the Two Data Types?," maptiler, 01 2022. [Online]. Available: https://documentation.maptiler.com/hc/en-us/articles/4411234458385-Raster-vs-Vector-Map-Tiles-What-Is-the-Difference-Between-the-Two-Data-Types-. [Accessed 22 05 2022].

[32] OSM Buildings, "Data," [Online]. Available: https://osmbuildings.org/data/. [Accessed 22 05 2022].

[33] 3D Buildings, "3D Buildings Maps," ONEGEO GmbH, [Online]. Available: https://3dbuildings.com/maps. [Accessed 22 05 2022].

[34] GeoJson, "GeoJson," GeoJson, [Online]. Available: https://geojson.org/. [Accessed 22 05 2022].

[35] M. Raifer, "Overpass Turbo," [Online]. Available: https://overpass-turbo.eu/. [Accessed 22 05 2022].

[36] OSM contributors, "Simple 3D Buildings," OpenStreetMaps, [Online]. Available: https://wiki.openstreetmap.org/wiki/Simple_3D_Buildings. [Accessed 22 05 2022].

[37] OSM contributors, "Key:building," [Online]. Available: https://wiki.openstreetmap.org/wiki/Key:building. [Accessed 22 05 2022].

[38] A. Lavrenov, "CONTRACTPLAN RESEARCHPROJECT & BACHELORPROEF," Howest, Kortrijk, 2022.

[39] E. You, "Overview," Vite.js, [Online]. Available: https://vitejs.dev/guide/#overview. [Accessed 29 05 2022].

[40] B. Rinaldi, "What Every JavaScript Developer Should Know About Floating Points," modernweb.com, [Online]. Available: https://modernweb.com/what-every-javascript-developer-should-know-about-floating-points/. [Accessed 23 05 2022].

[41] Three.js Contributors, "Map Controls Demo," Three.js, [Online]. Available: https://threejs.org/examples/misc_controls_map.html. [Accessed 22 05 2022].

[42] Three.js Contributors, "Three.js Interactive Lines Demo," Three.js, [Online]. Available: https://threejs.org/examples/#webgl_interactive_lines. [Accessed 22 05 2022].

[43] Three.js Contributors, "Raycaster," Three.js, [Online]. Available: https://threejs.org/docs/#api/en/core/Raycaster. [Accessed 22 05 2022].

[44] AlgoDaily, "An Illustrated Guide to Dijkstra's Algorithm," AlgoDaily, [Online]. Available: https://algodaily.com/lessons/an-illustrated-guide-to-dijkstras-algorithm/javascript. [Accessed 22 05 2022].

[45] B. L. Video, "Create 3D PATH ANIMATIONS Around the World," Youtube, 27 08 2019. [Online]. Available: https://www.youtube.com/watch?v=LQXdT-odNcU. [Accessed 30 05 2022].

[46] N. Maas, "IS HET MOGELIJK OM EEN ONLINE CONFIGURATOR TE BOUWEN MET BEHULP VAN THREE.JS," Howest, Kortrijk, 2020.

[47] Google, "ModelViewer," Google. [Online]. Available: https://github.com/google/model-viewer. [Accessed 30 05 2022].

[48] React Three Fiber, "Testing," [Online]. Available: https://docs.pmnd.rs/react-three-fiber/tutorials/testing. [Accessed 30 05 2022].

[49] Three.js Contributors, "Align HTML elements to 3D," Three.js, [Online]. Available: https://threejs.org/manual/?q=html#en/align-html-elements-to-3d. [Accessed 18 5 2022].

[50] P. Henschel, "Mixing HTML and WebGL with Occlusion," [Online]. Available: https://codesandbox.io/s/mixing-html-and-webgl-w-occlusion-9keg6. [Accessed 30 05 2022].

[51] Three.js Contributors, "Creating Text," [Online]. Available: https://threejs.org/docs/?q=text#manual/en/introduction/Creating-text. [Accessed 30 05 2022].

[52] A. Lavrenov, Artist, *WebGL & WebGPU pipelines.* [Art]. 2022.

[53] C. W. François Beaufort, "Access modern GPU features with WebGPU," 23 03 2022. [Online]. Available: https://web.dev/gpu/. [Accessed 20 05 2022].

[54] G. Tavares, "WebGL Rasterization vs 3D libraries," [Online]. Available: https://webglfundamentals.org/webgl/lessons/webgl-2d-vs-3d-library.html. [Accessed 21 05 2022].

[55] N. Desaulniers, "Raw WebGL," HTML5DevConf & IoTaconf, 14 06 2014. [Online]. Available: https://www.youtube.com/watch?v=H4c8t6myAWU&t=1019s. [Accessed 20 05 2022].

[56] OSM Buildings, "Twitter | OSM Buildings," Twitter, 25 04 2016. [Online]. Available: https://twitter.com/osmbuildings/status/724554813715931136?lang=en. [Accessed 23 05 2022].

[57] Three.js Contributors, "Documentation," Three.js, [Online]. Available: https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene. [Accessed 18 5 2022].

[58] R. Miletitch and Q. Houben, "WebGL Graphic Pipeline," Open Tech School Brussels, [Online]. Available: https://opentechschool-brussels.github.io/intro-to-webGL-and-shaders/log1_graphic-pipeline. [Accessed 20 05 2022].

[59] OpenStreetMaps, "Minlevel.svg," 10 02 2011. [Online]. Available: https://wiki.openstreetmap.org/wiki/File:Minlevel.svg. [Accessed 30 05 2022].

[60] OSM Contributors, "Multipolygon Illustration 6," 13 11 2008. [Online]. Available: https://wiki.openstreetmap.org/wiki/File:Multipolygon_Illustration_6.svg. [Accessed 30 05 2022].

[61] A. Lavrenov, *Storymap User Manual,* Kortrijk: Howest, 2022.

[62] A. Lavrenov, *Storymap Installation Manual,* Kortrijk: Howest, 2022.

[63] Apple, "Apple Maps," Apple, [Online]. Available: https://www.apple.com/maps/. [Accessed 30 05 2022].

[64] R. Lider, "Showing journey on the map," stackexchange, 19 07 2015. [Online]. Available: https://movies.stackexchange.com/questions/36850/showing-journey-on-the-map/41762#41762. [Accessed 30 05 2022].

[65] OSM Buildings, "OSM Buildings," OSM Buildings, [Online]. Available: https://osmbuildings.org/. [Accessed 30 05 2022].

8 Attachments

8.1 Installation Manual

New Project

To add the storymaps package to a new project:

1. clone the storymap repository
2. use npm i to install dependencies

```
npm i
```

3. use "npm link" inside the storymap repository to create a local global verison of it

```
# .../storymap
npm link
```

4. create a new project with vite.

```
npm init vite@latest
```

5. Install the storymap package using npm link:

```
npm link storymap
```

6. Edit your html to contain:

```
<div id="shortcuts" class="shortcuts">
        <ul>
                <li>
                        Press <span class="keyboardButton">S</span> to save
path to file
                </li>
                <li>Press <span class="keyboardButton">R</span> to
reset</li>
                <li>Press <span class="keyboardButton">Z</span> to
reset</li>
                <li>Press <span class="keyboardButton">X</span> to
redo</li>
        </ul>
</div>
```

```
<span id="scrollProgress"></span>

<div id="container"></div>
```

7. Edit your main.js

```
import * as storymaps from "storymap";
storymaps.setup();
```

8. Now run the project using vite

```
npm run dev
```

Demos

To use the demo, follow these steps:

1. clone the storymap repository
2. use npm i to install dependencies

```
npm i
```

3. use "npm link" inside the storymap repository to create a local global verison of it

```
# .../storymap
npm link
```

4. clone any of the demo repositories.
5. use "npm i" to install dependencies in the demo repository

```
# .../demo-client  or  .../demo-admin
npm i
```

6. use "npm link storymap" inside the demo repository to symlink and "install" the storymap package.

```
npm link storymap
```

7. use "npm run dev" to start a local server.

```
npm run dev
```

Storymaps

This package generates a 3D city using geojson data from OpenStreetMaps, generated by overpass-turbo.
The package comes with a sample dataset of Bruges, however you will need to get your own data if you want another city.

DEMO

- setup https://tania.tours/demo/client
- start https://tania.tours/demo/admin

Getting GeoJSON data

Get the geojson data from overpass-turbo;

Select the area you want to display on the website, and copy and run the query below. Click on Export and save as geojson.

```
[out:json]
[bbox:{{bbox}}]
[timeout:30];

(
        way["building"]({{bbox}});
        relation["building"]["type"="multipolygon"]({{bbox}});

        way["highway"]({{bbox}});
        relation["highway"]["type"="polygon"]({{bbox}});

        way["natural"="water"]({{bbox}});
        way["water"="lake"]({{bbox}});
        way["natural"="coastline"]({{bbox}});
        way["waterway"="riverbank"]({{bbox}});

        way["leisure"="park"]({{bbox}});
        way["leisure"="garden"]({{bbox}});
);

out;
>;
out qt;
```

You also need the latitude and longitude coordinates of the center of the selected area.
You can use Google Maps to get this data.

Configuration

after installing this package following the steps in INSTALLATION.md, you can start configuring it in main.js
before you run the setup() or start() functions, create a configuration.

```
import * as storymaps from "storymap";

storymaps.global.config = {
        debug: true,
        data: "/node_modules/storymap/sample/bruges.geojson",
        path: "/node_modules/storymap/sample/path.json",
        container: "container",
        citycenter: [3.227183, 51.209651],
        color_background: 0x222222,
        color_buildings: 0xfafafa,
        grid: { primary: 0x555555, secondary: 0x333333 },
        color_ground: 0x00ff00,
        opacity_ground: 0.25,
};

storymaps.setup();
```

This will overwrite the default configuration values.

- data: points to the geojson data you can get from overpass.turbo.eu
- path: points to a path you can create in .setup() mode.
- container: the ID of the div in which your canvas will sit.
- citycenter: the latitude/longitude of the centerpoint of your geojson data.

ANIMATION

To exit "create a path mode", use the .start() function.

```
// storymaps.setup()
storymaps.start();
```

This will create an on-scroll animation for the camera that follows the path you created on scroll.

8.3 Report Guest speaker

On the 20th of January 2022, I took part in a session about "Bridging the gap with explainable AI" presented by Matthias Feys, a guest speaker from ML6.

ML6 an European AI consultancy agency with offices in Belgium, Netherlands, Germany & Switzerland. They employ over 90 experts and are the largest growing AI company in Europe.

They work with many clients from different domains, which gives them a unique insight into a variety of AI applications. A few examples include Healthcare, Ecommerce, Manufacturing, Finance and Communication with specific examples including UZ Leuven, Google and ASML.

AI models are complex, and it is often difficult or even impossible to explain to a customer why the model has predicted or chosen a specific outcome. Explainability allows for a better understanding of the result, alongside avoiding yes/no answers, and making the model answer in probabilities.

It also allows to shine a light on which inputs had a significantly affected the result. This allows the client to see what is most impactful, resulting in return on investment and adoption of the tech.

He then continues to talk about three main scenarios in which Explainable AI can drive adoption:

1. AI is weaker – explainability is about improving the AI

To avoid humans blaming the model for defects, it's important to properly visualize and define what exactly the model is doing and is responsible for.
In addition, to gain trust you need to allow humans to correct the model in case they notice an error, which will in turn help the model learn and get better over time.
Finally, implementing a backup plan. In the case of Balta's optical rug folding, it was a simpler model that could be taught to process one specific rug, instead of the global model that would be applied to all rugs.

2. AI is on par – explainability is about building trust in the AI

When the AI performs roughly equally to the human, the best way to build trust is by giving the user a better understanding of how it works. Without that trust it will be tempting to use your own intuition instead.
In addition, knowing how the AI works can help the human use it to their advantage.
In the case of March, the AI was used to not simply suggest companies to call, but the reasons it matched a company to a building were used to argument and convince the company to use it.

3. AI is stronger – explainability is about explaining a complicated concept and informing humans.

Stronger than human performance is great, but with additional insight could result in tangentially related improvements.
In the case of Otary, the AI was used to predict the produced power, and it predicted a loss once planned turbines were added to the model. The human assumption that the planned turbines were placed too close together, but once the explainable model was build it became clear that it was the turbine turning speed.

Matthias then talked about 5 generic design patters that help make a project more explainable:

1. Problem reframing: changing and ideally simplifying the original question so it becomes easier to explain
2. Interpretable models: choosing models that are more explainable from the get go
3. Feature attribution: testing which inputs are most or least significant.
4. Transparency & transferability: taking into account that major changes in data might make the model innacurate
5. Intuitive visualizations: think about how you're going to visualize the result from the start

The talk concluded by saying that explainability is not just a single feature, but a whole pipeline that starts with asking the right questions and ending at the user interface and experience, with the goal of increasing trust in the AI.

Despite me being from the Web & App track, I found this talk to be valuable, as those 5 tips can also be applied to many problems in Web Development, from getting to the bottom of what the client actually wants, figuring out the most important features and then getting to the final presentation and experience.