Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Theorie der hybriden Systeme

# 3D-Visualisierung von OpenStreetMap GIS-Daten in WebGL
# 3D Visualization of OpenStreetMap GIS Data in WebGL

Bachelorarbeit
Informatik

November 2020

| | |
|---|---|
| Vorgelegt von<br>Presented by | Philipp Schulz<br>Am Südhang 4<br>57548 Kirchen<br>Matrikelnummer: 370838<br>philipp.gerhard.schulz@rwth-aachen.de |
| Erstprüfer<br>First examiner | Prof. Dr. rer. nat. Erika Ábrahám<br>Lehr- und Forschungsgebiet: Theorie der hybriden Systeme<br>RWTH Aachen University |
| Zweitprüfer<br>Second examiner | Prof. Dr. rer. nat. Thomas Noll<br>Lehr- und Forschungsgebiet: Software Modellierung und Verifikation<br>RWTH Aachen University |
| Externer Betreuer<br>External supervisor | Dr. rer. nat. Pascal Richter<br>Steinbuch Centre for Computing<br>Karlsruhe Institute of Technology |

## Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Aachen, im November 2020

Philipp Gerhard Schulz

# Contents

# 1 Introduction

The main motivation for this work is to make it possible to visualize the planning of wind farms near residential areas or other regions. Wind turbines are environment friendly energy producers, because they generate electrical energy exclusively from wind energy. For this reason they are preferred over fossil energy sources in relation to sustainability. Looking at the annual yields about 747 wind turbines of type Nordex N149 would be needed to cover the energy produced by the hard coal power plant in Hamburg-Moorburg[1], which is enough for 3.29 million housholds assuming one consumes 3 500kwh per year[2]. These results lead to build more wind farms in order to reduce the usage of e.g. fossil energy sources. But there is some criticism about it. Occasionally local communities like Böllen in Baden-Würtemberg e.g. complain about wind farms near them as they produce continuous noises, cast shadows and cause loss of property values[3]. This slows down the process of building up wind farms, which is also shown in the article given above. Not only would this application counteract these problems, but also reduce costs in planning wind farms and give a better imagination of how one would fit in a specific location. Each individual citizen could walk through a virtual presentation of her/his residence to eventually decide whether she/he is satisfied with the given wind farm plan or not. This would make planning those farms more comfortable.

## 1.1 Related Work

There exist a lot of other software implementations which create 3D visualizations of cities and other regions. But some are not free, others do not support visualizing scenes in a web browser and others do not have representations of any arbitrary region available. One for example is virtualcitySYSTEMS[4]. They create 3D city models and offer urban planning as commercial solutions. For the models they use CityGML[5], which is an open data model for the storage and exchange of virtual 3D city models using XML as format. It allows for defining detailed buildings including their interiors, which on one hand is superfluous for this work and on the other hand not available in every arbitrary region. But the results they got are good and realistic, because they make use of tilted satellite images to paint their buildings with.

Another software is OSM2World[6], which is implemented using Java. With it one can create scenes of arbitrary regions selected to view. The software already supports a lot of different objects to display from the OSM database. Lastly developers of OSM2World added support for dynamic road lanes including direction arrows[7]. Export

[1]https://www.entega.de/blog/windkraftanlage-leistung/
[2]https://www.ndr.de/nachrichten/info/Watt-Das-leisten-die-Anlagen-im-Vergleich,watt250.html
[3]https://www.zeit.de/2020/08/windenergie-erneuerbare-energien-klimaschutz-klimapolitik
[4]https://vc.systems/
[5]http://www.citygml.org/
[6]http://osm2world.org/
[7]https://raw.githubusercontent.com/tordanik/OSM2World/master/doc/changes.txt

options are also supported to formats like OBJ for example, which could be used directly to display it in the browser via an according file loader. But the performance for rendering those OBJ data is not as good as the app created in this work, which is compared and shown in Section 6.4.

**3D Visualization** As the application should run on different devices like smartphones, tablets, laptops and other computers, WebGL[8] is choosen to represent real world objects in a virtual 3D scene in an internet browser. With this, only one application needs to be developed in order to be used by all devices which support a modern web browser. To prevent additional workload in creating such scenes we will use three.js[9] to create them. It is a JavaScript library which wraps the functionalities of WebGL and extends it by providing helper functions, predefined primitive objects and more to get started faster. This will help to generate 3D representations of buildings for example. It also supports virtual reality technology, which is important to point out because it gives users a better experience compared to the problems explained above[10].

Another alternative to three.js is babylon.js[11], which is also a powerful JavaScript 3D engine with focus on game development. But we will not make any usage of those features as the complexity of the application is quite small regarding to other 3D software.

**Geographic Information System (GIS)** As the geographic information system we will use OpenStreetMap (OSM) which is free and open source. It is similar to map services like Google Maps[12], Apple Maps[13] or Mapbox [14]. There is no specific company maintaining the data of OSM. Data is being maintained and collected by a community of volunteers. Anyone can register an OSM account to start right away adding information to the OSM world. How to maintain these information is explained in chapter 6 of this work. Mapping Parties or Mapping Weekends are organized to meet up a group of people who select an area to collect information about constructions, roads, forests and more. Afterwards they all come together in a café or at someone's home and fill in the collected data in OSM.

How to extract this information to create a 3D visualization of a selected location will be explained in Section 3 of this work.

## 1.2 Processing Server

The server is used to process GIS data and convert it to 3D objects to then make it available to the browser client to display. When requesting a scene the server expects

[8]www.wikipedia.org/WebGL
[9]https://threejs.org/
[10]https://threejs.org/docs/#manual/en/introduction/How-to-create-VR-content
[11]https://www.babylonjs.com/
[12]maps.google.com
[13]maps.apple.com
[14]mapbox.org

a boundary box of latitude/longitude coordinates which the scene should cover. The current flow to create a scene is shown in Figure 1 in abstract form.

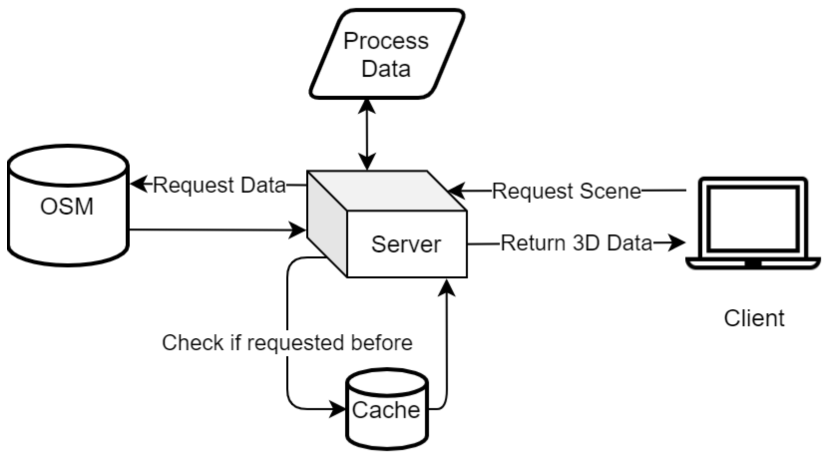


Figure 1: Interactions between client, server and OSM. First the client requests a scene. Then the server checks whether the requested scene was already processed before. If yes, return it from the cache. Otherwise request needed data from OSM, process it, save it in cache and return it to the client.

A cache is also used to prevent unnecessary requests to the OSM database of scenes which were already requested before. This cache holds the 3D information of a scene, which means that there is no reprocessing needed, too.

For the server Deno[15] is used which is modern JavaScript runtime. In contrast to Node.js[16], which is also a popular JavaScript runtime, it is possible to use JavaScript features which are only available in a web browser. This is important for this work, because the `GLTFExporter` of three.js (which is used to export scenes to a file to transfer it to the client) depends on the `Blob` and `FileReader` classes which are generally implemented in the browsers, but not in Node.js. Currently third party packages of those classes are needed to be installed in Node.js.

## 1.3 Outline of this Work

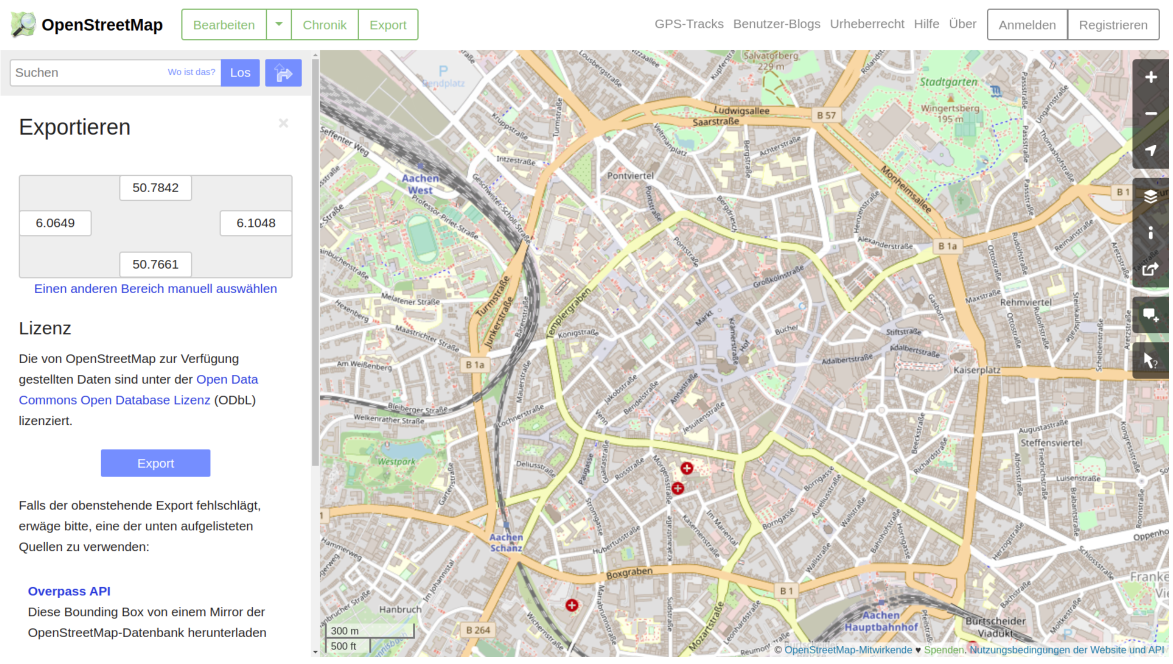
First the Overpass API of OpenStreetMap will be presented in short as the source of geographic information and how choosen data can be gathered. Then in Section 3 the creation of buildings, roads and trees with help of these data is explained. As these objects are placed via latitude/longitude coordinates in OSM a conversion of those into a cartesian coordinate system with help of the Mercator Projection is described in Section 4. Afterwards in Seciton 5 shadows are added which are quite important, because shadow casting is a topic not to neglect as stated above. Then in Section 6 techniques to optimize performance will be shown and different results on different

[15]https://deno.land/
[16]https://nodejs.org/en/

devices will be compared. Last but not least an introduction of how data in OSM can be maintained and what to watch out for when collecting data to get realistic buildings in the 3D visualization.

# 2 Geographical Information

Geographical information is the main source of this work, because they contain details about roads, buildings, rivers, bus stops and many more, which is important to give a realistic visualization. But these are mostly available as 2D information meaning you can view them on a flat map. This makes it hard to construct 3D data out of it in the first place. OpenStreetMap helps us out by providing additional information like heights, forms, colors etc. of objects. In this work only a subset of those objects will be presented as it would go beyond the scope otherwise. One way to get these information is to visit openstreetmap.org and use the export button placed at the top bar. This takes you to the view shown in Figure 2.



50.7842
6.0649
6.1048
50.7661
Einen anderen Bereich manuell auswählen

Lizenz

Die von OpenStreetMap zur Verfügung gestellten Daten sind unter der Open Data Commons Open Database Lizenz (ODbL) lizenziert.

Falls der obenstehende Export fehlschlägt, erwäge bitte, eine der unten aufgelisteten Quellen zu verwenden:

Diese Bounding Box von einem Mirror der OpenStreetMap-Datenbank herunterladen

© OpenStreetMap-Mitwirkende ♥ Spenden. Nutzungsbedingungen der Website und API

Figure 2: View of OpenStreetMap Export webpage for location Aachen-Zentrum.

OpenStreetMap will collect all data available in their databases of the area which is viewed in the map on the right side of the page. This can take a while to process depending on how much information is available in this area. After it finished you will get an OSM file which is formatted as XML and holds the geographical information. With this file we could already start building 3D objects for our visualization. But because we are using Deno as our server, which uses JavaScript as programming language, this would imply to convert these map data to a readable format for JavaScript first before being able to work with it in Deno. Deno packages exist, which would do this work for us, but fortunately OpenStreetMap offers another way where there are no extra steps needed to convert the requested data.

## 2.1 Overpass API

The Overpass API is another interface used to query map data from the OpenStreetMap databases. In contrast to the main API used above, which is optimized for editing data, this one is optimized for consuming only [1]. It offers various output formats such as GeoJSON, JSON, GPX and more. The one we work with is JSON as we can directly use it for our server without extra dependencies needed to be available.

One more advantage is the query language provided by the Overpass API which can be used to fetch data based on specific criteria [3]. Not only can we select data we want, but we can also filter data we do not want which makes our requests faster and the responses cleaner. An example of a query which selects buildings only is shown in Listing 1.

```
// Set output format
[out:json][timeout:25];
// collect results
(
    way["building"]({{bbox}});
    relation["building"]({{bbox}});
);

// Print results
out body;
>;
out skel qt;
```

Listing 1: An example query which queries buildings in Overpass QL.

I will not go into any detail here about making requests to the Overpass API as it is a topic in itself. More information can be found in their user's manual guide [2].

## 2.2 Content Description

Now that we know where to get geographical information from we can go on analyzing these to know how to use them to extract information we need for the 3D visualization. We will work with a minimal example of a response from the Overpass API in section 2.3 to explain the structure.

In general the OSM format consists of three data primitives called *node*, *way* and *relation*. The relationships between them are hierarchically ordered and are interpreted as references to their relationship partner.

### 2.2.1 OSM Node

A node is an object which has an id and a position, which it represents, described in the WGS-84 coordinate system[17] used by OpenStreetMap. It is used to represent

[17]https://en.wikipedia.org/wiki/World_Geodetic_System

single point objects such as bus stops, benches, wind turbines and more. A collection of node ids is referenced by a *way* object which is described in the next subsection.

Additional information are *tags*, *timestamp*, *version*, *changeset*, *user* and *uid*. The only one relevant of these for this work is *tags*. The rest is mainly used to track when and who created or edited a node.

Tags consist of a list of key-value pairs which gives the node more specific information. For example a bus stop usually has a name which results in a key-value pair with key *name* and value *Aachen Bushof* to be listed under tags. These tags will be used later in this work to get the height of a building and more details.

An example of such a node is given in Listing 2.

```
{
  "type": "node",
  "id": 371567077,
  "lat": 50.7768890,
  "lon": 6.0906311,
  "tags": {
    "bus": "yes",
    "highway": "bus_stop",
    "local_ref": "H12",
    "name": "Aachen Bushof",
    "network": "AVV",
    "operator": "ASEAG",
    "public_transport": "platform",
    "ref:IFOPT": "DE:Q:66467110",
    "shelter": "yes",
    "wheelchair": "yes"
  }
}
```

Listing 2: Node example in JSON format.

### 2.2.2 OSM Way

A way is an object which has an id and a list of ordered node ids. Like the node object it also has a *tags* attribute to describe the way more specifically. There are two types of ways: *open* and *closed* ways. An open way is similiar to a path whereas a closed way is similiar to an area. Open ways are used to represent roads or rivers while closed ways represent lakes, simple buildings or fields.

To decide whether a way is open or closed we have to check if the first node id is the same as the last node id in the node ids list.

### 2.2.3 OSM Relation

A relation is an object which has an id and list of members. It has a *tags* attribute also and the same additional attributes like a node. Each member represents a reference to another object. Most of the time it is a reference to a way. Its function is to describe more complex buildings which consist of many single parts for example.

It is hard to represent a cathedral with only one way because it could have multiple spires at different positions. A solution would be to describe each spire with a way and then list them in a relation's members list to tell that they belong together. Another usage is to define buildings which have an inner courtyard for example. In thise cases relations are marked as *multipolygon*, which is used and explained in Section 3.4

## 2.3 Query Example

The base JSON structure is shown in Listing 3.

```
{
    "version": 0.6,
    "generator": "Overpass API 0.7.56.1002 b121d216",
    "osm3s": {
        "timestamp_osm_base": "2020-03-21T13:13:03Z",
        "copyright": "The data included in this ...
    },
    "elements": [
        ...
    ]
}
```

Listing 3: Base response from the Overpass API.

It has some informational attributes about the API used which I will not go into. The most important attribute is the *elements* attribute. This one holds all nodes, ways and relations which we get from the specified query. With the query given in Listing 1 we could get the following response shown in Listing 4 depending on the viewed bounding box.

```
{
    "version": 0.6,
    "generator": "Overpass API 0.7.56.1002 b121d216",
    "osm3s": {
        "timestamp_osm_base": "2020-03-21T14:40:02Z",
        "copyright": "The data included in this document ...
    },
    "elements": [
        {
            "type": "way",
            "id": 217460658,
            "nodes": [
                2267232606,
                2267232735,
                2267232737,
                2267232755,
                2267232606
            ],
            "tags": {
                "building": "yes"
            }
        },
        {
            "type": "node",
            "id": 2267232606,
            "lat": 51.1458531,
            "lon": 5.7479002
        },
        {
            "type": "node",
            "id": 2267232735,
            "lat": 51.1457897,
            "lon": 5.7478806
        },
        {
            "type": "node",
            "id": 2267232737,
            "lat": 51.1457734,
            "lon": 5.7480154
        },
        {
            "type": "node",
            "id": 2267232755,
            "lat": 51.1458368,
                "lon": 5.7480349
            }
        ]
}
```

Listing 4: A JSON response from the Overpass API.

The *elements* list in this response holds five items. Each item has a type attribute which defines whether it is a node, way or relation. So we have one way and four nodes. Additionally the nodes list of this way has five node ids referenced although we only have four nodes available. But the first node id is the same as the last node id and therefore we have a closed way here which results in a valid constellation. And the tags of the way give us more information about the object itself and tells us with the key *building* and belonging value *yes* that this way represents a building. So the four unique nodes of the way are describing the corners of a building. And by the attributes *lat* and *lon*, as abbreviation for latitude and longitude, in each node we also know its location.

# 3 3D Object Construction

Now that we know how to get and read data given by the Overpass API we can go on constructing 3D objects out of it. We choose the Aachen Cathedral as our working object to construct as it includes a variety of forms. three.js offers many classes to build objects with. We will use some of them to reduce additional work in creating buildings. Yet there are special forms like roofs which need manual work to create.

First the creation of buildings will be explained followed by roofs and roads. Additionally the implementation of trees and forests will be introduced.

In the following points are an alias for vectors in $\mathbb{R}^2$ or $\mathbb{R}^3$.

## 3.1 Buildings

As described in Section 2.2.2 buildings in OpenStreetMap can be described as closed OSM ways. This means we have a list of positions given which represent the outline of a building. In OpenStreetMap these positions are described in the WGS-84 coordinate system, which in short are latitude and longitude coordinates to represent positions on earth.

The idea is to use these points to form a footprint and extrude it to a specific height. This height will be the height of the building. We use the Shape class from three.js[18] with which we can define this footprint. It has two methods `moveTo` and `lineTo` we will be making use of:

- `moveTo` allows us to set an initial point to start our shape at.
- `lineTo` then enables us to define more points while lines are being connected between the last added point and the new one.

For example let $S = (1, 1), P_1 = (2, 2), P_2 = (3, 1)$ and $P_3 = (2, 0.5)$ where $S$ is the starting point and $P_i$ is a point to connect to for all $i \in \{1, 2, 3\}$. Calling `moveTo(S)` will move the internal cursor of the shape to point $S$. Further calls `lineTo(`$P_1$`)`, `lineTo(`$P_2$`)` and `lineTo(`$P_3$`)` in this order will result in the shape shown in Figure 3.

[18]https://threejs.org/docs/#api/en/extras/core/Shape

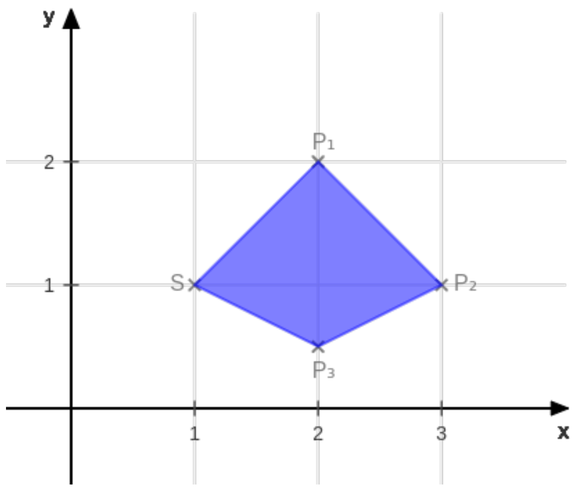


Figure 3: Result of shape with given points.

The shape will automatically connect a line from the last added point to the first one to close it.

So for a building in OSM the list of nodes describing the outline of the building is used to create a shape. As these outlines express a closed way corresponding to section 2.2.2 the last element in the list is skipped to keep unique points. At this point the Aachen Cathedral base is already creatable shown in Figure 4.

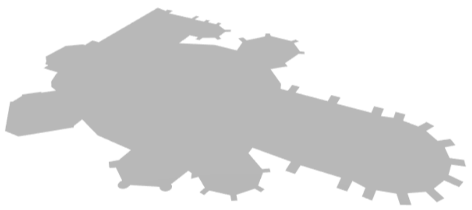


Figure 4: Aachen Cathedral base footprint.

The next step now is to extrude this shape to its height given in the OSM tags. From three.js the `ExtrudeGeometry`[19] class is used which expects a shape or an array of shapes to be passed. It offers many options to define how the geometry shall be created. The important one is `depth`. It sets how deep the shape is to be extruded. In this case we set it to the height of the building.

[19]https://threejs.org/docs/#api/en/geometries/ExtrudeGeometry

In OSM the tag with name *height* is commonly used to specify the height in meters of a building[20]. Sometimes the height tag is not given and the person who added or edited the building in OSM has defined the levels only. If this is the case it can be found under the tag name *building:levels*. We use 4 meters for each level to approximate the height for a building. So if there is no height specified but the levels with value 4 we assume the building to be around 16 meters high. If even this tag is not available, we fall back to a building height of 10 meters.

Another relevant option to care about is `bevelEnabled`. It adds beveling to the geometry which is neither effective nor necessary for the current case and therefore disabled.

Passing the shape of the Aachen Cathedral from Figure 4 to a new `ExtrudeGeomtry` instance with options according to the description given above now results in some depth in our scene shown in Figure 5.

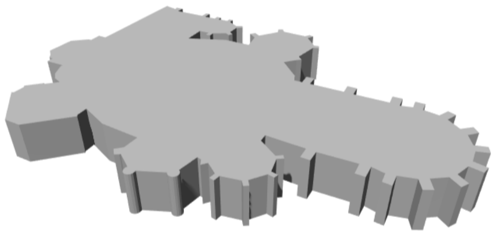


Figure 5: Aachen Cathedral base extruded with height 7 (fall back value).

### 3.1.1 Raised Building Parts

At the moment we only worked with one part or in other words OSM way of the Aachen Cathedral. But as already discussed in section 2.2.3 a complex building needs more ways to describe parts of it separatly. Looking at the Aachen Cathedral it has a mid tower and a front one which are of different shape. The mid one is rounded and the latter is squared. Therefore, we query for more OSM ways to get the full cathedral. After adding these ways belonging to the Aachen Cathedral which are available at the moment we get a result shown in Figure 6.

[20]https://wiki.openstreetmap.org/wiki/Key:building#Additional_Attributes

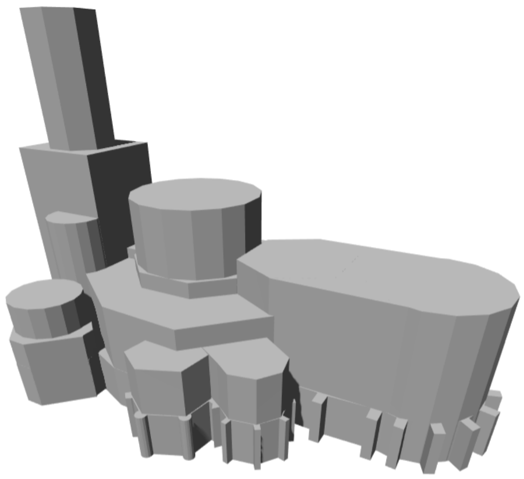


Figure 6: Aachen Cathedral with all OSM ways belonging to it.

As already explained above the mid tower is rounded and the front one is squared which is visible in our new 3D version of the cathedral.

At the moment we assume every building or its part to lie on the ground by using its given height. But this is not the case everywhere. For example the Aachen Cathedral has a little bridge connecting the front and mid tower. It is marked in red in our current model in Figure 7.

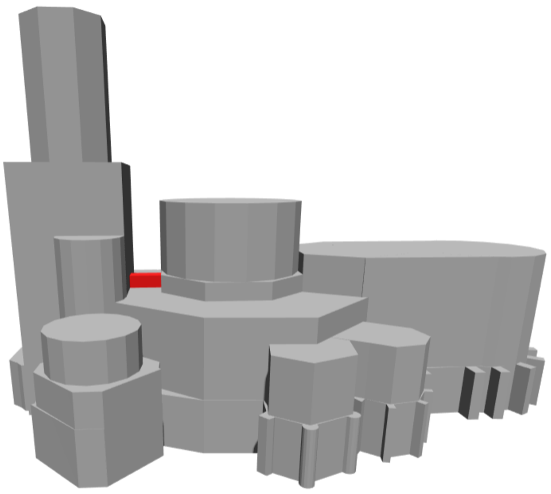


Figure 7: Bridge marked in red with falsely calculated facade in relation to its height.

Currently the height for this bridge set in OSM is 27 meters. But looking at real pictures of the Aachen Cathedral the bridge has no wall underneath it like in our current model. That is because we are missing one important property relevant for these cases called *min_height*[21] in OSM. This tag describes at which height a part of

[21]https://wiki.openstreetmap.org/wiki/Simple_3D_buildings#Height_and_levels

a building is placed at. Depending on that value the real height of this part results in subtracting this value from the value given in *height*. For example if a building has $height = 30$ and $min\_height = 25$ then this means that the building begins at height 25 and ends at height 30 and thus has a real height of 5 meters. An illustration with the sideview of the Super C in Aachen in an abstract form as an example is given in Figure 8.

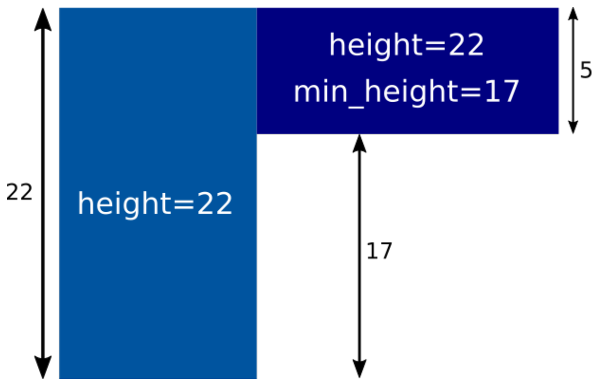
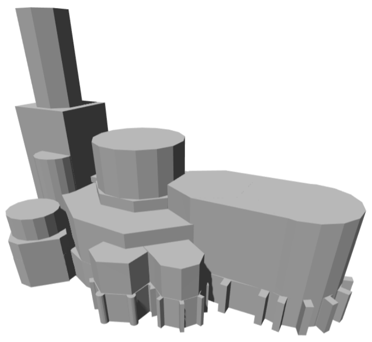
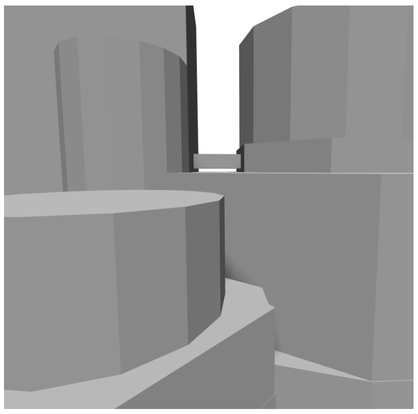


Figure 8: Illustration of *min_height* property in OSM.

After applying this tag the 3D model of the Aachen Cathedral now looks like shown in Figure 9 (b).



(a) Overview of Aachen Cathedral.



(b) Close up view of the bridge. Note the gap underneath it.

Figure 9: Aachen Cathedral after applying *min_height* property.

Now the bridge part does not show any wall underneath it which is the result we expected. Additionally the mid tower base is raised, too, leaving blank space to the ground. This is intended by the OSM commmunity as will be comprehensible in the next section about adding roofs.

## 3.2 Roofs

At the moment we covered many important properties to get an abstract 3D object of a building. But roofs are still missing. As described in the introduction of this section some roof types need to be created manually instead of using a predefined geometry from the three.js library. For these we create the geometry by ourselves. There are many different roofs supported in OSM which are not covered all in this work. The complete list can be found in their wiki[22]. Four new relevant tags named *roof:shape*, *roof:height*, *roof:orientation* and *roof:direction* will be introduced in this section. The *roof:shape* is the important one because it defines whether a building has a roof or not and if so what shape it represents. In this context a roof shape with value `flat` is the same as there is no *roof:shape* tag defined. We will begin with the flat roof shape, then the dome, the pyramidal, the gabled and the skillion one.

### 3.2.1 Flat

The flat roof is the simplest one because there is nothing to do to achieve it as a building part itself is flat on top in our case. The Aachen Cathedral itself has no visible flat roof available in OSM so there is nothing special to show anything apart of Figure 9.

### 3.2.2 Dome

For the dome roof we will be making use of the SphereGeometry[23] from three.js. As it also has a height we have to look for the *roof:height* tag. When this tag is present the real height of a building is once more recalculated depending on that value. Same as for *min_height* the value of *roof:height* gets subtracted from the value of *height*. For example if a building has $height = 30, min\_height = 10$ and $roof{:}height = 5$ (assuming tag *roof:shape* is present) then the real height, respectively height of the building facade, is $30 - 10 - 5 = 15$ meters beginning at 10 meters above the ground.

Not only we can pass a radius to the SphereGeometry constructor, but also whether we want a full sphere or just a half one for example, which represents a dome we can use for a roof. The parameter to be changed is `phiLength` which expects an angle in radians to define the size of the horizontal sweep. This defaults to $2 * \pi$ which is a full sweep and therefore a full sphere. In this case we set it to $\pi$ to get a half one.

Now that we have a half sphere representing our dome we need to define its size. The size is defined by scaling the roof along the up axis until it reaches the height which results the roof in being a half ellipsoid instead of a half sphere. Adding this roof type to our application we get a result shown in Figure 10. As visible the cathedral has exactly two dome roofs at the moment. I have colored these to have some contrast to the building facades and will do this for the other roof types, too.

[22]https://wiki.openstreetmap.org/wiki/Simple_3D_buildings#Roof_shape
[23]https://threejs.org/docs/#api/en/geometries/SphereGeometry

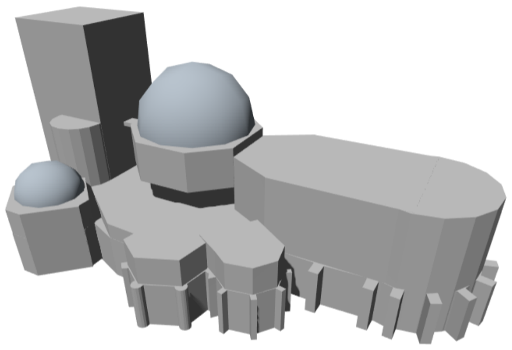


Figure 10: Aachen Cathedral after adding support for dome roof type.

### 3.2.3 Pyramidal

For this roof type we could go analogical to the dome roof and use the ConeGeometry from three.js to create a pyramidal roof. But because this geometry is less complex than a half sphere and we want to align the roof base with the building top (so the complete area of this top is covered) we will create our own geometry. Latter strengthened by the property of the ConeGeometry base being equilateral, which does not hold for every building obviously. For this approach to work accurately the building area must be convex, thus connecting two of any corner points of the building must not clip any edge. But we accept buildings where this case does not hold as the probability of those existing is nonzero, which comes from the fact that no data in OSM is checked against validity before being published. In the following the algorithm for creating this roof type based on a building base will be explained.

Given a set $\mathcal{V}$ of points $P_i \in \{\begin{pmatrix} x \\ y \\ 0 \end{pmatrix} \mid x, y \in \mathbb{R}\}$ and finite $n \in \mathbb{N}$ with $1 \leq i \leq n$ where the third component of a vector represents the height in the local coordinate system of a roof, which in this case is 0 for all points and represents the roof's base. Furthermore, $P_1, P_2, ..., P_n$ are mapped such that the first two components of $P_i$ are equal to the coordinates of the i-th node given in the nodes list of a OSM way described in section 2.2.2. This list has $n + 1$ elements, because we consider closed ways in this scenario. Now we define the center point $P_c \in \mathbb{R}^3$ of all points $P_i$ which represents the top of our pyramidal roof after height being applied. To calculate $P_c$ we take the average of the positions of all points and apply the roof height $h_r \geq 0$ (extracted from tag *roof:height*) to it, which results in the following equation:

$$P_c = \frac{1}{n} \sum_{i=1}^{n} P_i + \begin{pmatrix} 0 \\ 0 \\ h_r \end{pmatrix}$$

With this setup we now can create our geometry. For this we need to define polygons as triangles connecting 3 points each from which exactly one is always $P_c$. All together our pyramidal geometry then has $n$ polygons. After adding support for this roof type the visualization of the Aachen Cathedral looks as shown in Figure 11.

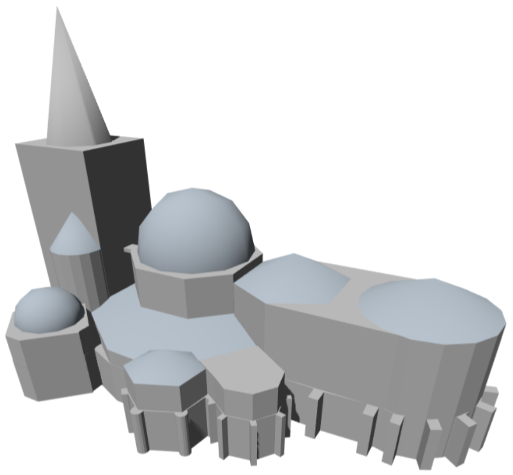


Figure 11: Aachen Cathedral after adding support for pyramidal roof type.

As mentioned in section 3.1.1 the mid tower of the dom was floating in the air. Now this is fixed as the underlying building part has a roof of type pyramidal defined which gives the illusion of it holding the tower part.

Another thing to notice is the front tower being squared, but the pyramidal roof not matching its base which is contradictory to the algorithm described above for creating these roofs. An illustration is given in Figure 12 where the building base is marked in blue and the roof base is marked in red.

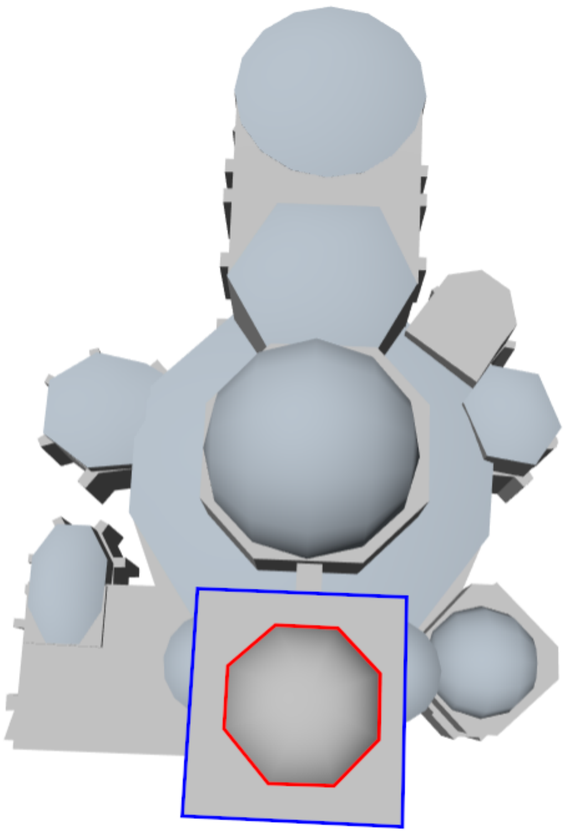


Figure 12: Bird's eye view of Aachen Cathedral marking tower bases.

But this is intended and not an error by the algorithm. The trick is to define buildings whose bases differ from their roof ones by adding another building part with a facade height of 0. This can be achieved by the following equation with corresponding OSM tags as described in section 3.2.2:

$$height_{facade} = height - min\_height - roof{:}height = 0 \qquad \text{with } roof{:}height\ > 0$$

The values currently set for the tower are $height = 74$, $min\_height = 49$ and $roof{:}height = 25$ which results in

$$height_{facade} = 74 - 49 - 25 = 0$$

being true and therefore having a roof without a building.

### 3.2.4 Gabled

The gabled roof type is another object to be constructed manually as three.js does not offer a predefined geometry of this form. It is similiar to a triangular prism. Its base consists of four points in quadrilateral formation and two additional points lying above opposite edges which form a triangle each that connect to make the roof. This implies that there are two options of how the roof can be oriented, because in this case, where the base has exactly four edges, only two can be pairwise selected to build the triangles with. This is where the tag *roof:orientation* comes into play which has two possible values called `along` and `across`. If this tag is not defined then the value `along` will be assumed. `along` means the prism is aligned with the long side of the building. Otherwise it is aligned with the short side. A visual illustration is given in Figure 13.

(a) roof:orientation = along (b) roof:orientation = across

Figure 13: Gabled roof with different orientation on same building

The four corners of the roof are being taken from a building base to construct the roof. But not each building, which has a gabled roof assigned in OSM, has exactly four corners. Consider the building way from OSM given in Figure 14 below.

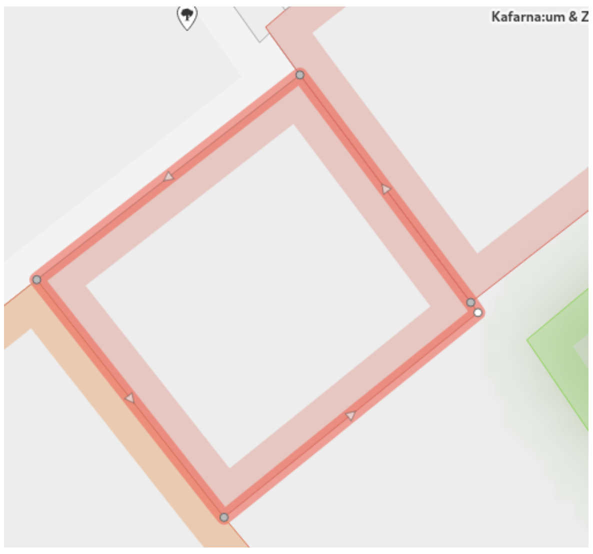


Figure 14: Building (id=83266236) with 5 corners and *roof:shape* set to gabled.

It shows one of many scenarios where two or more buildings are sharing one or more points, because these buildings are directly next to each other. On the right there is one point next to the corner of the marked building which is used to represent a corner for the other building next to it. Because we cannot create a simple prism, which's base consists of more than four corners under the fact that we need two edges lying on the opposite, the approach used in this work is to calculate the smallest bounding box consisting of four corners which covers the whole building base. This bounding box then represents the base of the gabled roof. An intution could be to calculate the minimum and maximum values of x and y coordinates each of every point and use these to represent the corners of the bounding box. But this leads to inaccurate results. Take the building from Figure 14 for example. The result using this approach would cover all corners, but the corners of the resulting roof would not match with the building ones. See Figure 15 for illustration.

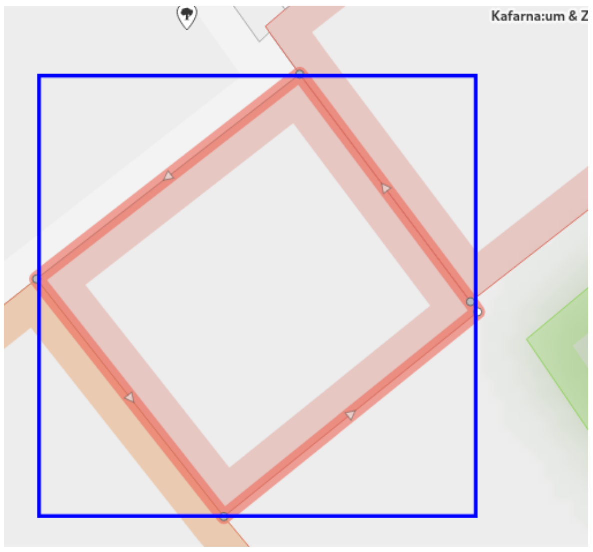


Figure 15: Bounding box in blue with building from above.

Calculating the ratio of the roof base area to the building base area in this example results in a 97.4% surplus of the roof's base area, which is almost twice as big as the actual base area.

To counteract this we take the same approach but apply it to a different set of points given. This set results from aligning the building's longest side with the x-axis and then doing the calculation of the bounding box for this rotated building. This way we make sure that shorter sides are missing in the roof's base area and the longest side is always taken into account resulting in a greater area coverage. For this we iterate through pairwise neighbored points and calculate the distance between those. Mathematically this means finding two neighbored points $P_i$ and $P_{i+1}$ from a set of $n \in \mathbb{N}$ points and $i \in [1, n]$ with $P_{n+1} = P_1$ where $A = P_{i+1} - P_i$ and

$$||A||_2 >= ||P_{j+1} - P_j|| \qquad \text{for all } j \neq i.$$

At next the signed angle $\alpha$ between vector $A$ and $B = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ is calculated with which a rotation matrix $R$ is created from by using the `makeRotationZ` function of the `Matrix4` class from three.js. This matrix has the following entries:

$$R = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So for any vectors $v, w \in \mathbb{R}^3$ with $w = R \cdot v$ $w$ is the vector which results from rotating $v$ by the angle $\alpha$. As in this case the building is rotated by angle $\alpha$ seen from the x-axis in its current state there must be vectors $Q_i$ which were rotated by using the rotation matrix $R$ to result in vectors $P_i$. Changing this equation as follows gives the resulting points $Q_i$ of the building whose longest side is aligned with the x-axis:

$$\begin{aligned} & R \cdot Q_i = P_i \\ \iff & R^{-1} R \cdot Q_i = R^{-1} \cdot P_i \\ \iff & Q_i = R^{-1} \cdot P_i \end{aligned}$$

Now applying the bounding box approach on the points $Q_i$ results in four new points $B_1, B_2, B_3, B_4 \in \mathbb{R}^3$, which represent the corners of the bounding box. These have to be rotated by left multiplying $R$ with $B_k$ for $k \in [1, 4]$ so they align with the original building position. In this case, according to the ratio explained above, the surplus reduces to 2.0%.

The same analysis was applied for 782 different buildings in Aachen where at first 85.4% of the roofs' area were superfluous and after fixing the rotation of the buildings it were reduced to 13.6%.

Now the Aachen Cathedral looks as shown in Figure 16.

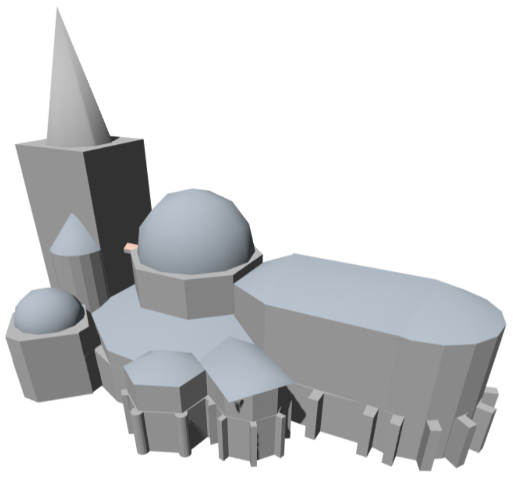


Figure 16: Aachen Cathedral after adding support for gabled roof type.

On the right one gabled roof has been added which connects to two pyramidal roofs next to it to create a bigger related roof.

### 3.2.5 Skillion

The skillion roof type is the last to be supported for now. It is basically a flat roof but tilted to a given direction. That is where the *roof:direction* tag plays a role. It holds either a cardinal value like `N` (north), `S` (south), `SW` (south west) or `ESE` (east south east) or an angle in degree relative to north clockwise[24]. For example an angle of 90 degree is the same as the value `E` meaning it is facing to east direction. Assuming we have already applied the bounding box approach explained above for a building with a skillion roof we get exactly four points. In this case tilting the roof results in four possibilities of pairwise neighbored points to be raised to the given roof height whereas the other two keep a height of zero. Let $P_1, P_2, P_3, P_4 \in \mathbb{R}^3$ be the points of a building base with skillion roof arranged as given in Figure 17.

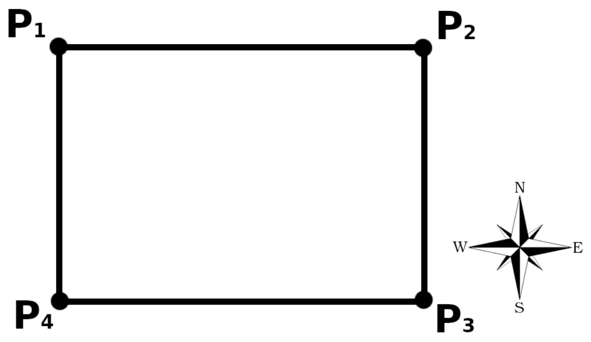


Figure 17: Building base with points $P_1, P_2, P_3, P_4$.

Then if the value given in tag *roof:direction* is an angle in degrees we take it as it is and pass it to the function $d : [0, 360] \to [1, 4]^2$ which maps the angle to a pair of indices $(i, j) \in [1, 4]^2$ where $P_i$ and $P_j$ will be modified such that they have the height given in tag *roof:height*:

$$d(a) = \begin{cases} (1,2) & \text{if } a > 315° \text{ or } a < 45° \\ (2,3) & \text{if } a >= 45° \text{ or } a < 135° \\ (3,4) & \text{if } a >= 135° \text{ or } a < 225° \\ (4,1) & \text{otherwise.} \end{cases}$$

If for example the value of *roof:direction* is 60, then we get $d(60) = (2, 3)$ and therefore points $P_2$ and $P_3$ will be raised to the specified height such that the slope is approximatly oriented to this direction.

Otherwise if the value given in tag *roof:direction* is an direction given as `N`, `SW`, etc. (these values can be looked up on the wiki page[16]) we map it to an angle accordingly and pass it to the function $d$ given above to retrieve the indices of points to raise. For example if the value is `W` it will be mapped to an angle of 270°. Furthermore, if it is `SSE`, `SSW` or `S` all three get mapped to 180°.

[24]`https://wiki.openstreetmap.org/wiki/Key:roof:direction`

As the Aachen Cathedral has no skillion roof assigned in OSM another example with two neighbored buildings, which have a skillion roof, is used and shown in Figure 18.

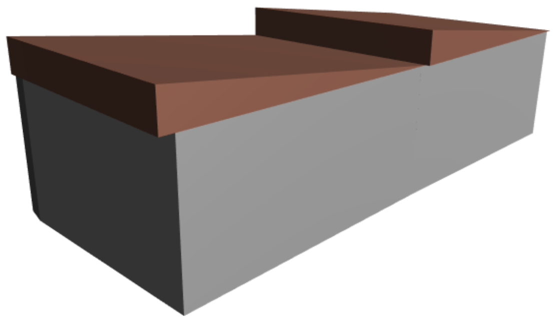


Figure 18: Two buildings with skillion roof type.

## 3.3 Roads

Roads in OSM are mostly represented as open OSM ways which all together give a big road network. One case where roads are put in as closed ways are roundabouts. To identify roads or pathes in OSM a lookup for the tag name *highway* with value `yes` is to be done on way elements. There are many different highway types available which are listed on their corresponding wiki page[25]. We will not cover all of them in this work, but those which are present the most. We will first start with modelling roads in three.js, applying textures to them, discussing different types and their different look and show the results.

### 3.3.1 Construction

Roads in general follow a path which can be straight or curved. three.js offers a class called `SplineCurve`, which uses the `Catmull-Rom Spline` technique to build a curve[26]. Figure 19 shows such a spline curve in red constructed from a set of four points marked in blue:

It also shows that these curves generate smooth transitions between given points. We will take this curve as an example to create a road of. Let the points from left to right be defined as

$$\begin{aligned} P_1 &= (0,0,0) \\ P_2 &= (1,0,0) \\ P_3 &= (2,1,0) \end{aligned}$$

[25]https://wiki.openstreetmap.org/wiki/Key:highway
[26]https://threejs.org/docs/#api/en/extras/curves/SplineCurve

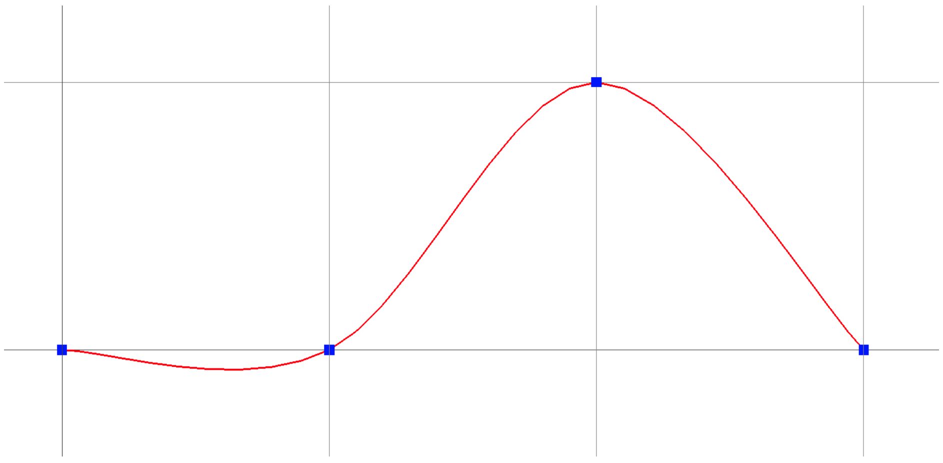


Figure 19: A spline curve constructed from four points.

$$P_4 = (3, 0, 0)$$

where the third component is the value for depth or height respectively. Every curve class in three.js has two functions

`getPointAt getTangentAt`

which both get a value between 0 and 1 passed to get a point or vector respectively from a position on the curve interpolated according to the arc length. For example let the arc length of a curve $C$ be 10 units. Then `getPointAt(0)` returns the position at the beginning of $C$ whereas `getPointAt(1)` returns the position at the end of $C$. `getPointAt(0.5)` returns the position on the curve which is exactly $0.5 * 10 = 5$ units in arc length away from the beginning of $C$. Analogical `getTangentAt` returns the tangent vector at those positions. We use these functions to generate vertices for our road mesh.

For that let $u \in [0, 1]$, $v_u \in \mathbb{R}^3$ be the position at $u$ as a vector returned from `get-PointAt(u)` and $t_u \in \mathbb{R}^3$ the tangent at $u$ as a vector returned from `getTangentAt(u)` with $||t_u||_2 = 1$. Furthermore, let $w \in \mathbb{R}$ be the width of the road to be constructed. To define the outlines of the road we need to create points which follow the curve parallel to its layout with a distance $\frac{w}{2}$. Figure 20 shows an example with six different values for $u$.

To define those points we take the up vector $(0, 0, 1)^T$ and calculate the cross product:

$$t_u \times (0, 0, 1)^T = p_u \in \mathbb{R}^3$$

where $p_u$ is the vector perpendicular to $t_u$ with the third component being 0 as well due to the cross product. Next we set the length of $p_u$ to be $\frac{w}{2}$. Because $||p_u||_2 = 1$ holds we only need to multiply every component of $p_u$ by $\frac{w}{2}$ and therefore get:

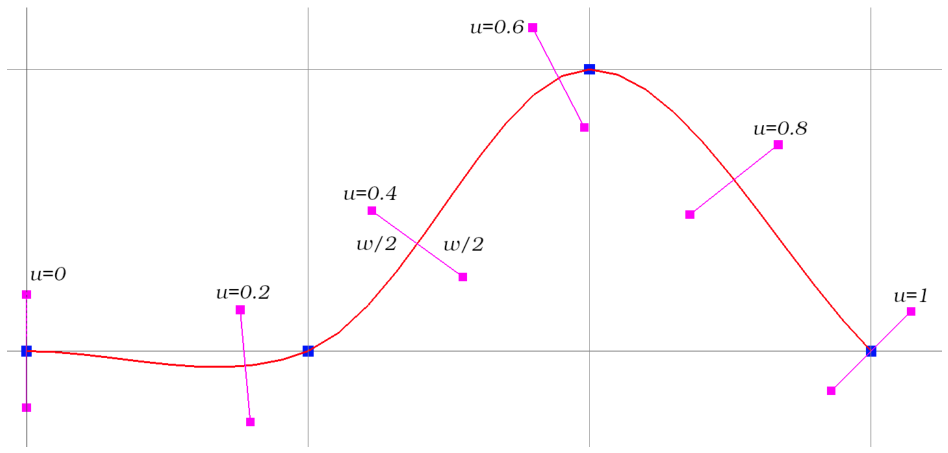


Figure 20: Spline curve with outline points for six different values of $u$.

$$p_{u,w} = \frac{w}{2} \cdot p_u$$

Now we can define two outline positions $v_{u,1}$ and $v_{u,2}$ which both have a distance of $\frac{w}{2}$ from the curve and $v_u$ respectively:

$$v_{u,1} = v_u + p_{u,w}$$
$$v_{u,2} = v_u - p_{u,w}$$

With this we can now create vertices for our road at any value $u$. Those vertices are marked as pink squared points in Figure 20 for example. They are used to define 3D objects, which are also called meshes. This is done by defining faces in form of triangles which connect three vertices each to fill up space between them. We will not go into any detail here as this is a topic in itself. A quick search on the internet gives many results of explanations and instructions.

But before we create such a mesh we need to define how detailed it should be, because theoretically there are infinte values for $u \in [0, 1]$ in $\mathbb{R}$, for which we cannot generate vertices of as it would blow the computation for rendering the road. Therefore, we define a grade value $d \in \mathbb{N}$ which tells how many segments the curve or road respectively should consist of. For example the grade of the curve in Figure 20 is $d = 5$ as the curve is split up into five segments. Increasing $d$ results in more segments and therefore more vertices. Taking this curve to create a road from with the approach given above we get the result shown in Figure 21.

What stands out here is the fourth segment being stretched due to the big angle between the tangents of both positions the segment lies between in. Increasing $d$ counteracts this problem, but at the same time increases the complexity of the geometry. Figure 22 shows the same road once with $d = 15$ and $d = 50$.

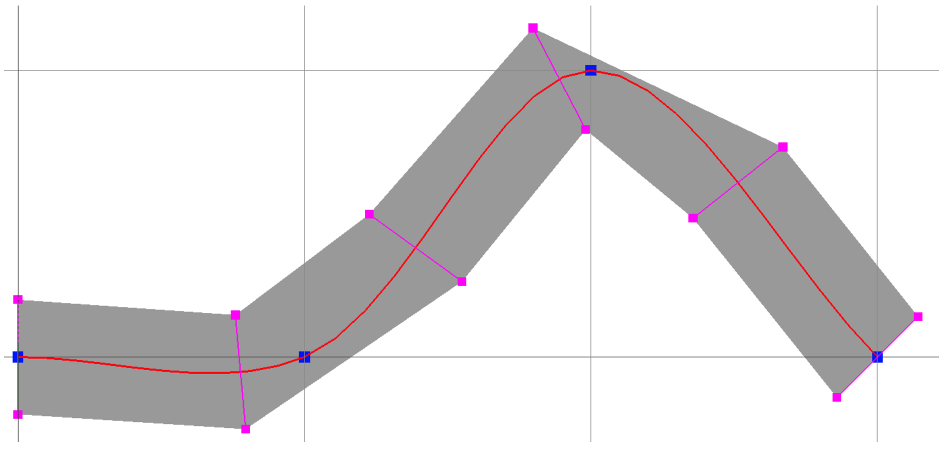
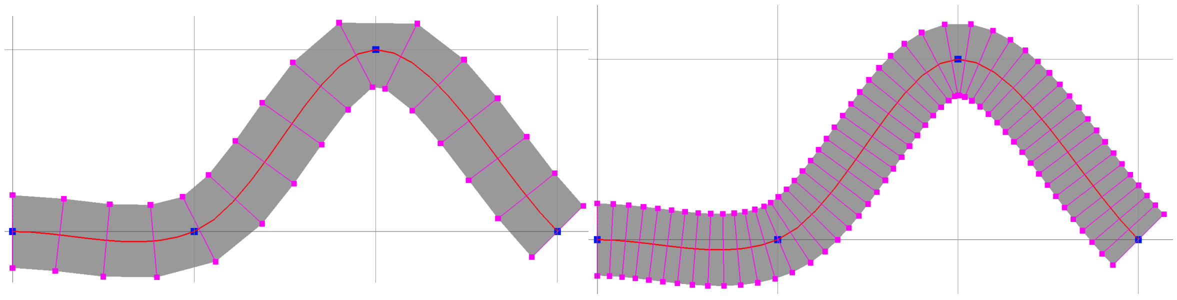


Figure 21: Road in grey created from curve of Figure 20 with $d = 5$.

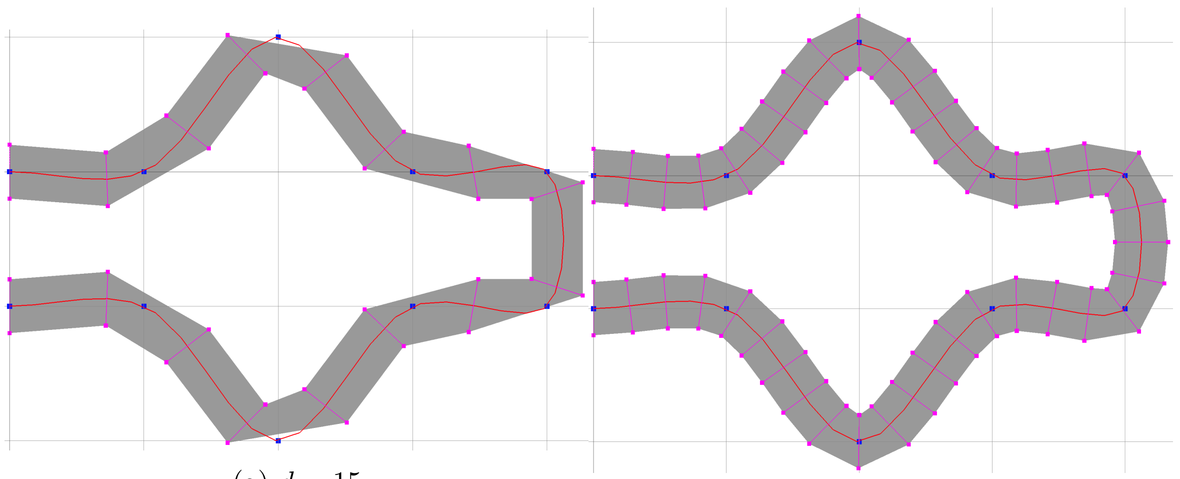


(a) $d = 15$

(b) $d = 50$

Figure 22: Road of length 3.9 with different values for $d$.

The more vertices are used the smoother the road. The value $d = 15$ for this road would be sufficient for the 3D visualization. But we cannot use this value for every road, because every road differs in length from others. Figure 23 shows a longer road with same $d = 15$ compared with $d = 40$.


(a) $d = 15$

(b) $d = 40$

Figure 23: Road of length 10.9 with different values for $d$.

As Figure 23 (a) shows a value of $d = 15$ would not be sufficient for a smooth road whereas for the road in Figure 22 (a) it would be. So we need to make sure that regardless of the length of a road every road is smooth enough to look acceptable. Saying that every arc length of one unit should be split up into five segments gives adequate results. For $d$ this means we first calculate the arc length $\ell$ of a curve and multiply it by 5:

$$d = \lfloor \ell \cdot 5 \rfloor$$

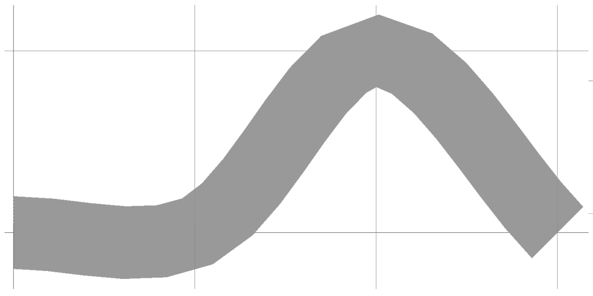
Then for the curve in Figure 19 the value

$$d = \lfloor 3.9 \cdot 5 \rfloor = \lfloor 19.5 \rfloor = 19$$

is calculated while for the other curve we get

$$d = \lfloor 10.9 \cdot 5 \rfloor = \lfloor 54.5 \rfloor = 54$$

The resulting road meshes are shown in Figure 24.


(a) $d = 19$

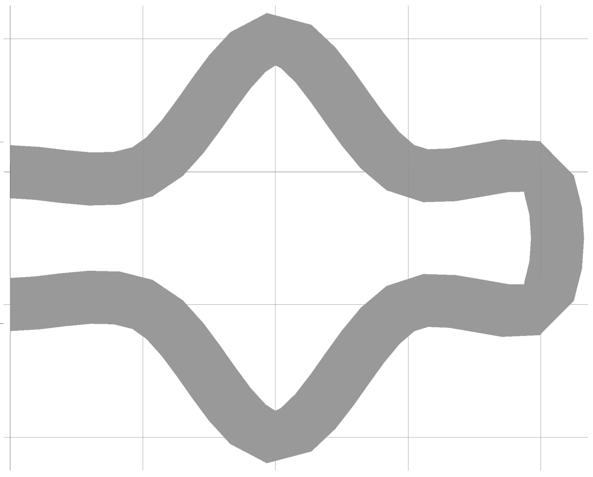

(b) $d = 54$

Figure 24: Road meshes of different length.

As mentioned above the complexity of the geometry can get quite high, which is an important topic in computer graphics. A common measurement for performance is the amount of faces a computer needs to render in a 3D scene. As our scenes can get quite large and should run smoothly on mobile devices we also want to reduce this complexity as much as possible. At the moment the number of faces for the road in Figure 24 (a) is 38 and for (b) it is 108, which in some cases can be unnecessarily high. For example consider a straight road of length 4 in our scene shown in Figure 25.

(a) Solid road mesh.

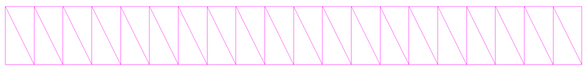

(b) Underlying faces of the road mesh.

Figure 25: Straight road mesh once solid and wired.

This road has 40 faces which all need to be rendered. But the geometry could be reduced to just two faces due to the simple rectangular form it has. This would result in what Figure 26 represents.

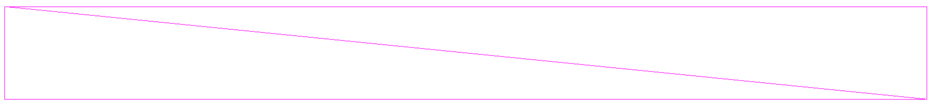


Figure 26: Simplified road mesh.

A simple approach is to scan the curve for how bent it is at certain positions and if there is a lot of bending we split the road more often, otherwise we split it less often or even not at all if it is straight. The amount of scans $n_{scans} \in \mathbb{N}$ to do depends on the arc length of the curve. We choose it to be 200 times the length, which results in

$$n_{scans} = \lfloor \ell * 200 \rfloor$$

The idea is to collect all $u$ where the road needs to be split up. The ones which are definitely needed are $u = 0$ and $u = 1$ as they represent the start and end of the road.

For every iteration the algorithm checks whether the difference of the angles to the x-axis of the current tangent vector $t_u$ at $u \in (0, 1)$ and the previous one $t_v$ at $v \in [0, u)$ exceeds a given threshold $T$. Previous here means where the road was last split, because if we would take the tangent vector from the previous iteration and check their angles a curve with small bending less than the threshold would be considered a straight road. The threshold used in the following results is $T = 0.31$.

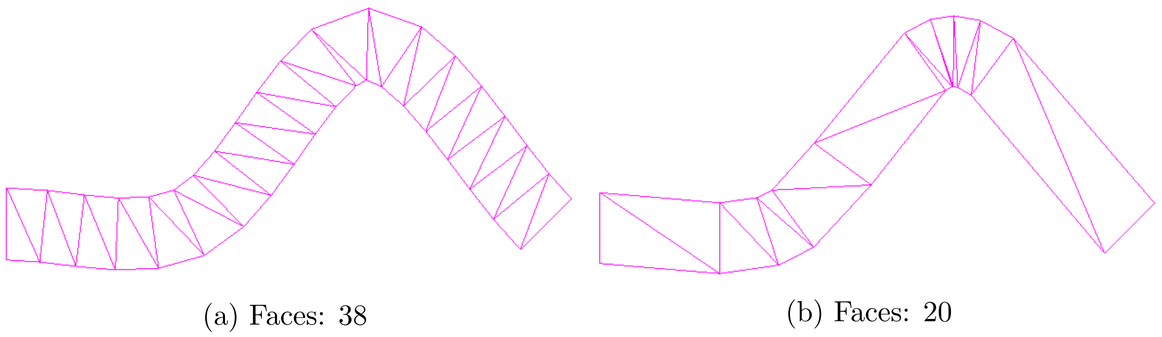

(a) Faces: 38 (b) Faces: 20

Figure 27: Road mesh before and after the algorithm is applied.

For the road in Figure 27 the amount of faces is reduced by almost half of the previous one. Comparing the solid version of the roads there is almost no difference.

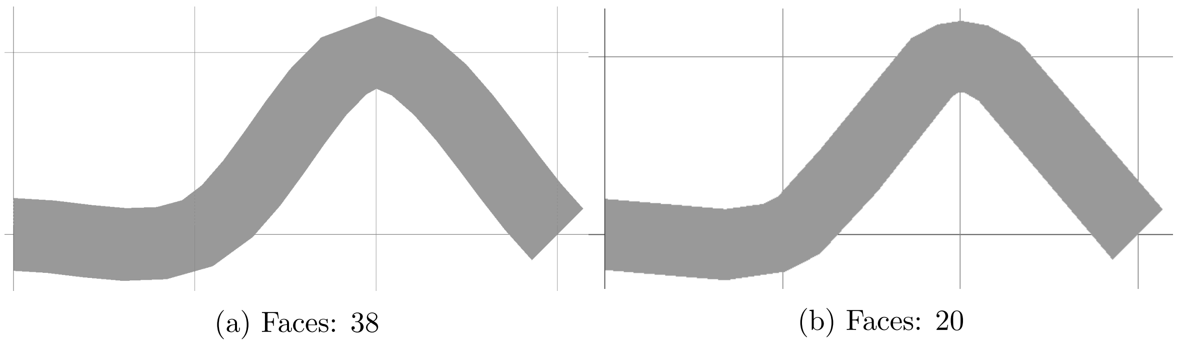
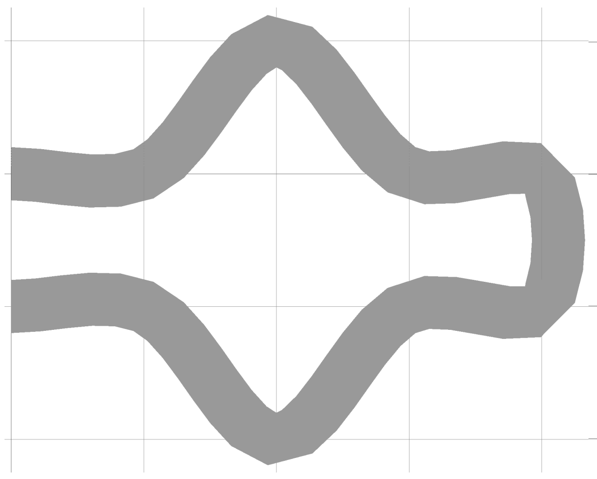

(a) Faces: 38 (b) Faces: 20

Figure 28: Solid road mesh before and after the algorithm is applied.

Curves can even be smoother after applying the algorithm as they are split up more than before. Figure 29 shows the same comparison with the longer road.


(a) Faces: 108

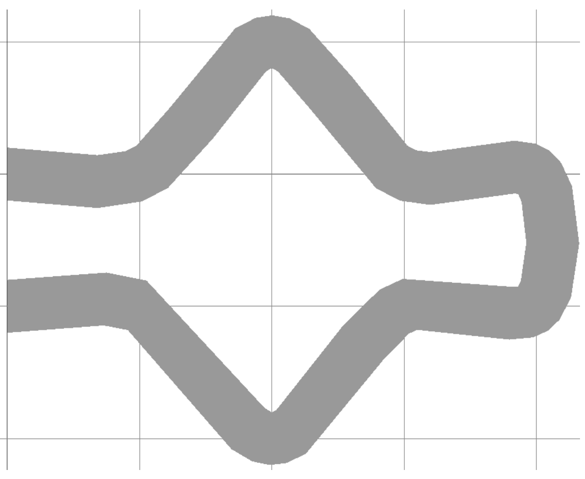

(b) Faces: 70

Figure 29: Long road mesh before and after the algorithm is applied.

At the moment roads look bare due to the simple grey color. The next subsection will explain how to apply a road looking texture to our mesh for realistic style using the UV mapping technique.

### 3.3.2 Texturing

UV mapping is a common technique used to map 2D images onto 3D objects. Usually this is done by assigning each vertex of a mesh a two dimensional coordinate which references a location on an image or texture respectively. We will not go into any detail here on how this technique works and assume that there is basic knowledge given.

Texturing curved objects can be tricky, especially when the form is dynamically created like the roads. Therefore, UV coordinates need to be generated dynamically, too. But there is a simple way to achieve this. The underlying curve of a road will be used to determine distances between adjacent splits, which will be used to define how much of the texture should be applied to the corresponding faces. The texture used is shown in Figure 30.

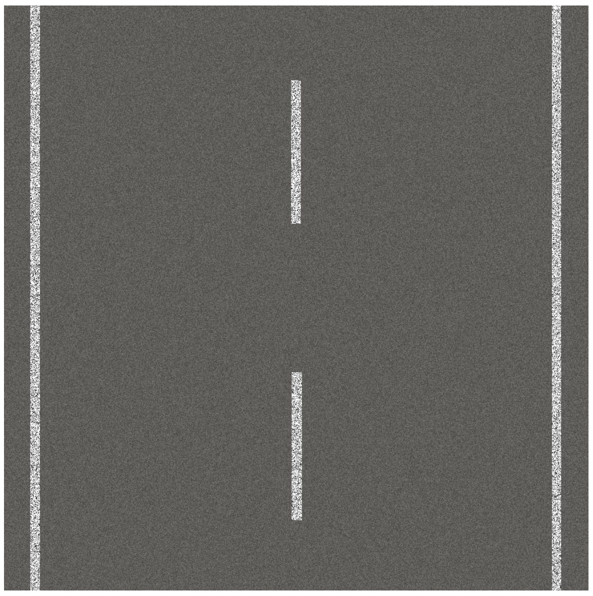


Figure 30: Simple road texture.

This texture is seamless in one way meaning that we can append it to itself on the top or bottom without creating a visible cut at the transition. This is important, because it makes a road look more natural. Furthermore, the width and length of this road segment represented by the texture is assumed to be five meters. So five meters in arc length of the curve should match one time the whole texture. This also means for shorter segments of a road to only match a part of the texture which in turn means that the next segment should start to be textured where the previous segment stopped to be textured. For this an accumulator variable is used to sum up arc lengths of the road segments. In the end the roads created above look like presented in Figure 31 after applying the texture.

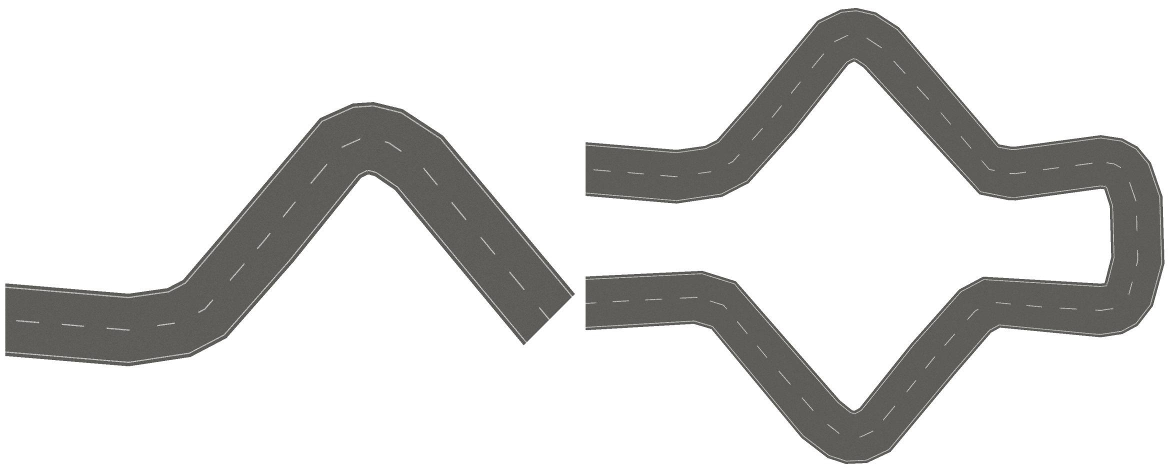


Figure 31: Roads with texture from above applied.

In the next subsection different types of roads with higher meaning are presented.

### 3.3.3 Types

As already mentioned above there are many different types of highways in OSM available. They range from streets to small paths. The ones that will be presented here are streets. There are currently seven different types available of those (`motorway`, `trunk`, `primary`, `secondary`, `tertiary`, `unclassified` and `residential`). According to the wiki page and example images given some of the types can be grouped together by their look. In this work `trunk`, `primary`, `secondary` and `tertiary` are grouped together to have the same texture given in Figure 30. Furthermore, `unclassified` and `residential` are grouped to have the texture shown in Figure 32 (a). The only one left is `motorway` which is assigned to the texture shown in Figure 32 (b) which represents a typically German Autobahn with two lanes.

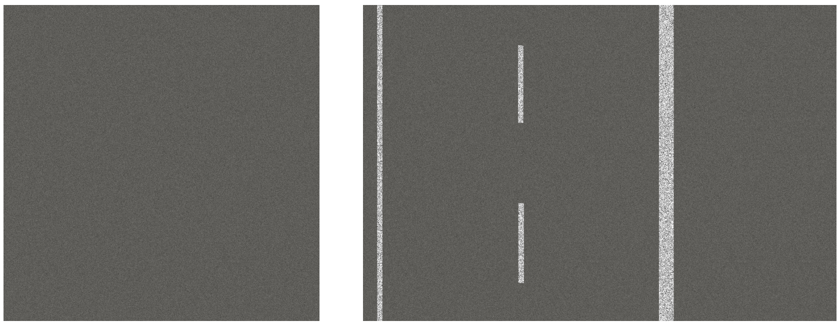

(a) Blank road texture. (b) Road texture for 2 lane motorways.

Figure 32: Road textures for different OSM highway types.

Adding these road types more variety is given to the road network. Figure 33 shows the Autobahn A3 in Cologne Rath.

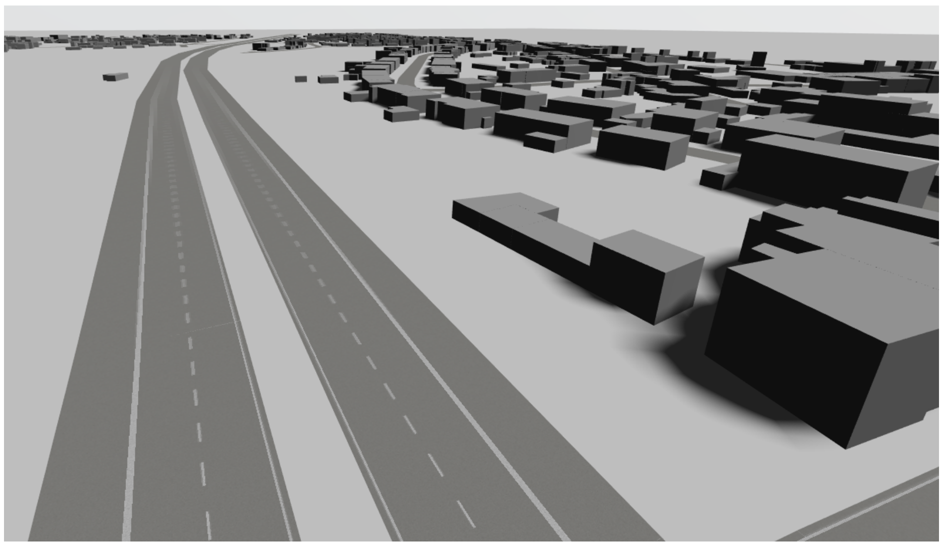


Figure 33: Highway A3 in Cologne Rath.

There are still a lot of things regarding roads missing like intersections or elevations. But these are topics which are complex and hard to implement. Therefore, [8] is referred to give motivation for creating smooth intersections at crosses.

## 3.4 Trees

To give the scene a more natural look trees and forests are added. The following approach can be applied analogical to other natural incidences like water for example. But also farm fields and other landuse types can be derived from it. The idea is to define a polygon from data given in OSM and randomly generate trees inside this polygon. The algorithm used is called PNPOLY by W. Randolph Franklin [11], which tests whether a point lies within a polygon or not. It works by taking a semi-infinte ray and switch between *outside* and *inside* whenever the ray crosses an edge of a polygon. With this technique it also supports holes inside a polygon, which is important in this work's use case. All together a list of positions is collected to define where trees should be placed.

### 3.4.1 Types

For trees there are different ways to add them to the OSM database. One is to place a single tree by defining an OSM node including its location. To identify this node as a tree the tag `natural` with value `tree` should be set. This type should not be used to define forests, but where trees occur particularly like in front of a building for example. In this case there is obviously no need to define a polygon. Only the position is collected and added to the list.

Another way to define where trees are is an area like a forest for example. Those are typically defined using OSM relations of type `multipolygon`[27] and hold a tag named `landuse` with value `forest`[28]. Other landuse types like `farmland` for example can be found on the wiki page[29]. In contrast to a polygon a multipolygon can consist of multiple polygons. With it holes in a polygon can be defined by setting another polygon inside it. OSM distinguishes between `outer` and `inner` polygons by setting an attribute called `role` with one of those values to a member of the relation accordingly. Figure 34 shows an example of a multipolygon which defines a polygon with a hole in it.

[27]https://wiki.openstreetmap.org/wiki/Relation:multipolygon
[28]https://wiki.openstreetmap.org/wiki/Tag:landuse%3Dforest
[29]https://wiki.openstreetmap.org/wiki/Key:landuse

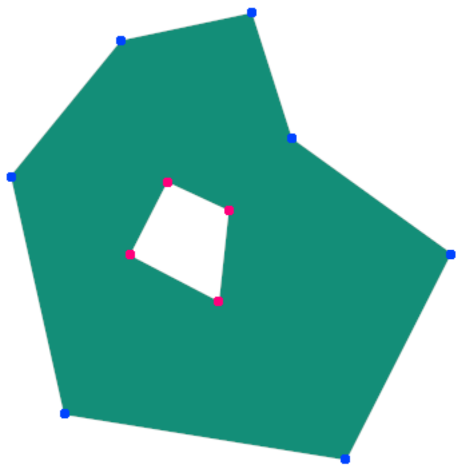


Figure 34: Example of a multipolygon. Outer polygon defined by blue points. Inner polygon defined by red points.

To decide where trees are to be placed in a multipolygon area an iteration of positions is needed which cover this area. So in every iteration the current position is checked against whether it lies within the polygon or not. To save up iterations we make use of the bounds element of a relation given in the Overpass API query response. It defines the boundary box of all elements assigned to the relation. The loop of iterations then starts at the lowest value according to the $x/y$ values (in this case from top left). Every step the current position's $x$ and $y$ values are increased by an offset $o$ (in the following $o = 10$) depending on the current iteration. Taking the multipolygon of Figure 34, the loop results in evenly distributed positions shown in the Figure 35 (a) in red. Placing trees at those positions would result in an unnatural look of the forest, because at certain viewing angles straight empty lines of spaces would be visible. To make it look more chaotic and therefore more natural we add a random values $r_x, r_y$ (in the following $r_x, r_y \in [-5, 5]$) to $x$ and $y$ respectively in every iteration and check the updated position against being contained in the multipolygon. $r$ is reset to a new random value in the same interval per iteration. The result of this is shown in Figure 35 (b).

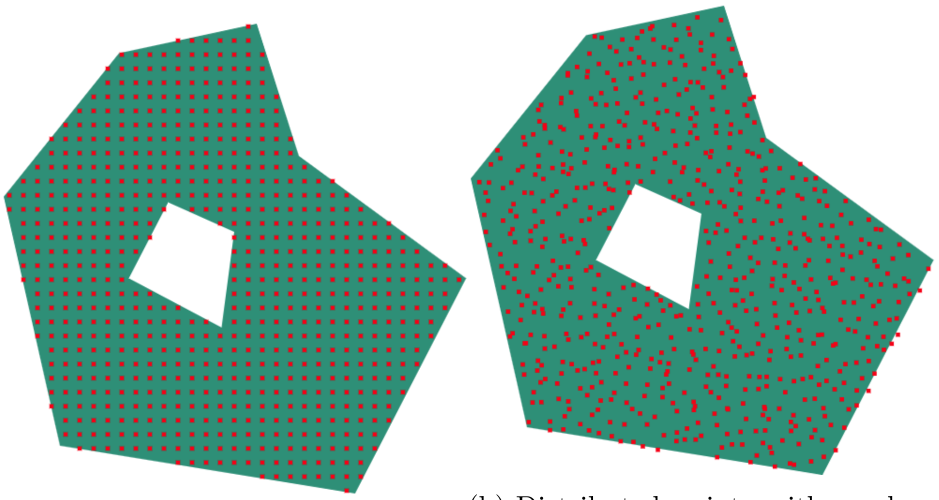

(a) Evenly distributed points. (b) Distributed points with random offset.

Figure 35: Comparison of distributed points in the example multipolygon.

The last method to define tree areas are OSM closed ways. They also hold a tag named `landuse` with value `forest`. The only difference to a relation, regarding to this usage, is that there is no hole in the area to be expected as a closed way defines only one polygon. Therefore, the same algorithm as used for an OSM relation can be used to generate positions for trees to be placed at.

### 3.4.2 Constructing

Tree models can be very complex because of the amount of branches and leaves a tree can have. Therefore, there exist many different techniques to display trees in a 3D scene. For example one of them is to model the tree trunk and branches in 3D and then add leaves by appending alpha textured plane meshes to the branches. This can reveal very realistic looking models. But those are of high geometry complexity and are mostly applicable in animated movies or similiar use cases. Another way would be to reduce the complexity of trees to primitive models like OSM2World does in its current state. But this can lead to unrealistic results. Another less complex but still detailed technique is to take a side viewed texture of a tree and apply it to simple mesh like a plane for example. This way we can store a detailed image of a tree while keeping the complexity of the model low. This is also important as a lot of trees can be rendered inside a forestial area. Such a texture is given in Figure 36.

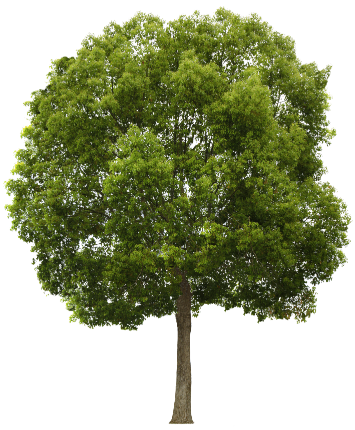
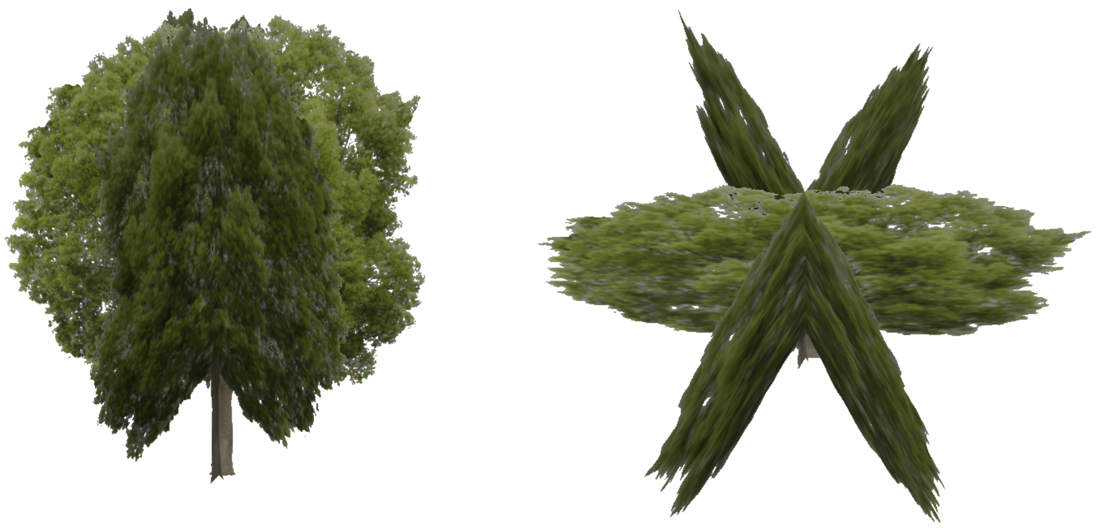


Figure 36: Texture of a tree viewed from the side.

Taken from https://www.cleanpng.com/png-tree-png-66102/

There are now two ways to use this texture to display trees with. One is to create a mesh with multiple planes intersection in their centers.



(a) Tree model viewed from front. (b) Tree model viewed from top.

Figure 37: Three planes intersection at their centers with the tree texture applied.

As the texture is transparent around the tree crown the borders of the planes are not visible, which is intended as it would destroy the look of the tree model.

Using this tree model the Westpark in Aachen looks like shown in Figure 38.

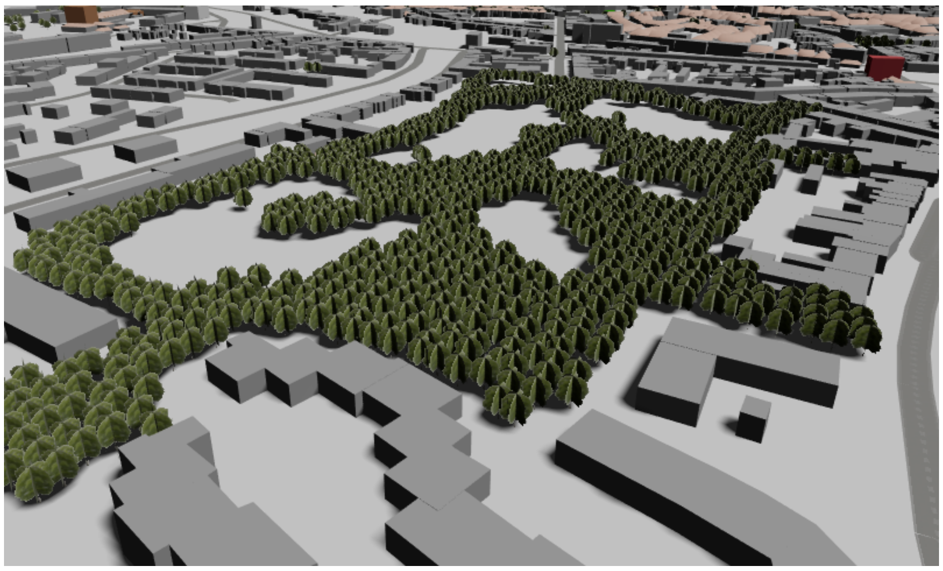


Figure 38: Westpark in Aachen with tree model given above.

Depending on the current sun position some planes of the trees are dark while neighboring planes are bright, which gives an inconsistent look of the tree. This is because the same plane facing or not facing the sun appears on both sides of the tree's center.

The other way is to only use one single plane mesh with the tree texture applied and let it always face the camera. Such planes are also called billboards as they always rotate to the camera independent of its location[30]. With this always the exact same view of a tree is visible to the camera. Not only the amount of faces for this model is reduced (by exactly four faces in this case), but also the result looks better and more natural as shown in Figure 39.

[30]http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/billboards/

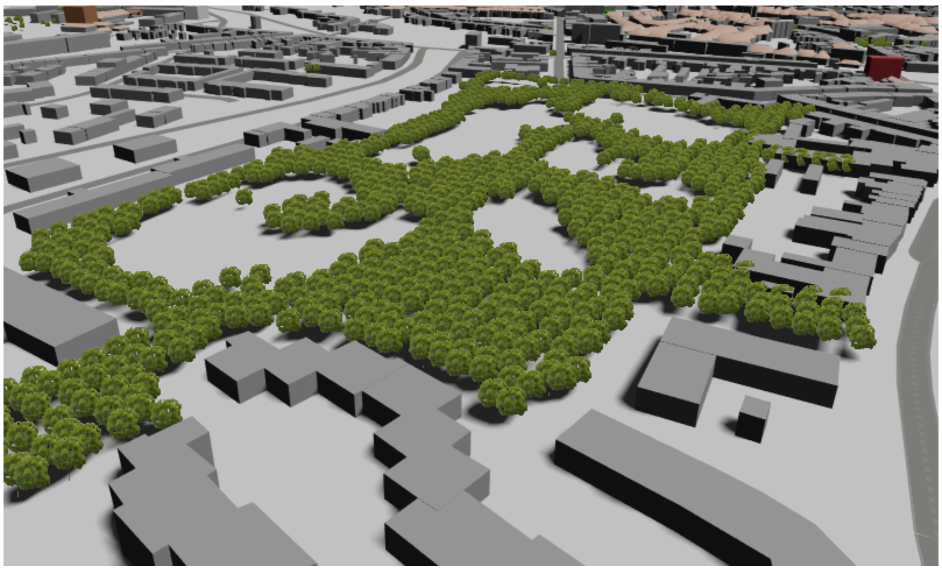


Figure 39: Westpark in Aachen with tree billboards.

# 4 Mercator Projection

The Mercator Projection is a cylindrical map projection created by cartographer Gerardus Mercator in the 16th century and today it is used by atlases and navigation charts to represent a flat visualization of earth's surface. Back then this projection provided a simple way to travel along big distances, which was important for marine navigators, by drawing a straight line on a map from the starting point to the destination point and following a course along that line[9]. This property comes from the fact that this projection maps all lines with constant bearing to straight lines (also called *rhumbs*)[9]. Instead of using rhumb lines one could use a *great circle route* to navigate which would make shorter ways over long distances on spherical objects like earth. But using this way navigators needed to adjust their courses by continuously changing their bearings while travelling a great circle route.

OpenStreetMap mainly uses a pseudo mercator projection meaning that earth is modelized as a perfect sphere, because it gives a fast approximation of the projection[31]. So far we did not discuss how we interpret the position of every node in OpenStreetMap. As already mentioned before every node element holds information about its latitude and longitude coordinate according to the WGS-84 coordinate system. This system is used to describe how locations on an ellipsoid, in this case earth, can be identified by points of two parameters latitude $\varphi$ and longitude $\vartheta$ both in degrees, where $\varphi$ ranges from $-90°$ to $90°$ and $\vartheta$ ranges from $-180°$ to $180°$.

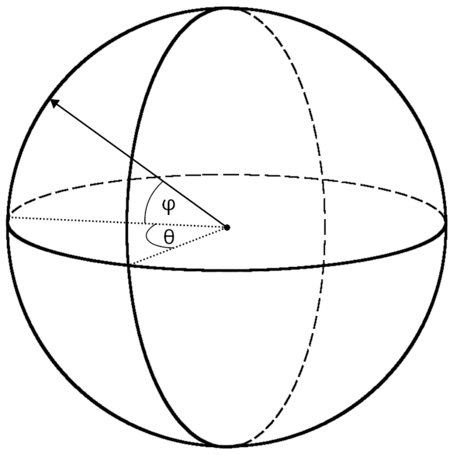


Figure 40: Example of a point on a sphere located at $(\varphi, \vartheta) = (40°, -90°)$

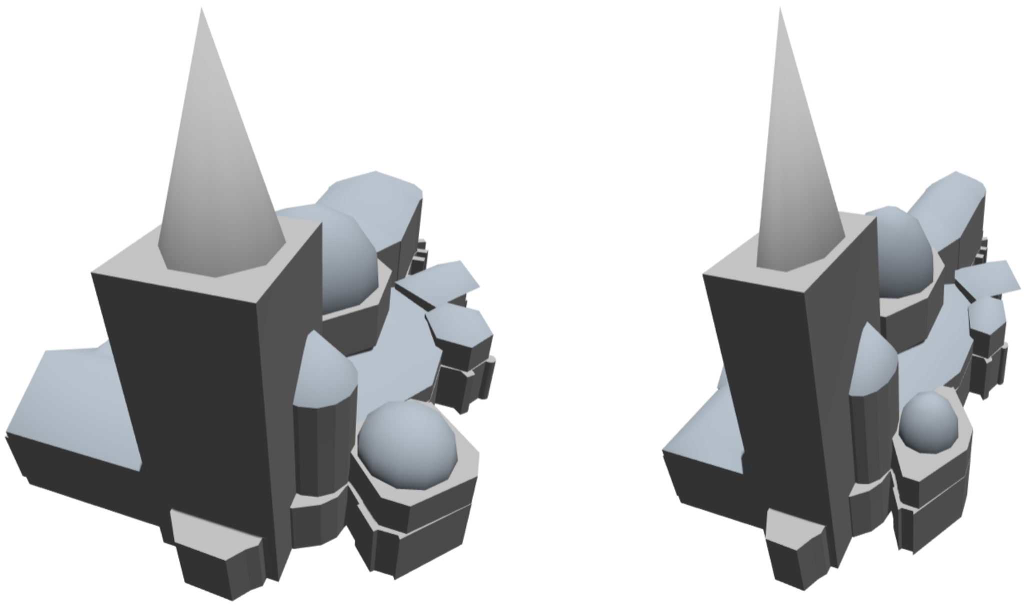
Figure 40 illustrates how a point is located on a sphere where the origin (0°,0°) is at the front cross. As points in the 3D scene will have coordinates in a cartesian

[31]https://wiki.openstreetmap.org/wiki/Mercator

coordinate system, we need to find a way to map coordinates described as points on a sphere to points in a cartesian coordinate system.

## 4.1 Mapping

A scene in three.js is a three dimensional cartesian coordinate system with three axes $x, y$ and $z$ (where $x$ and $z$ are for length and depth while $y$ is for height). Because as input we get coordinates which represent locations on a sphere we need to map those coordinates to cartesian ones. If not we would run into biased results. Here latitude values will be mapped to $y$ values and longitude ones to $x$ values. Take the following scenario as an example: Assuming earth - as a perfect sphere - has a radius of 6,371 km, then, under the condition of $\varphi = 0°$ and variable $\vartheta$, which means every location $(\varphi, \vartheta)$ is on the equator, a difference of $\Delta\vartheta = 1°$ results in an arc length of around 111.19 km. Now when letting $\varphi$ and thus the latitude coordinate converge to 90°, which means the location is exactly at north pole, the arc length would converge nonlinear to 0 km under the same condition that $\Delta\vartheta = 1°$, because all meridians meet at both north and south pole. Therefore, we can not just use the coordinates given in the OSM nodes for our cartesian coordinate system. Figure 41 illustrates what would happen with the cathedral used in section 3 when not taking this behaviour into account.



(a) Real (b) Biased

Figure 41: Comparison of the cathedral's real size and biased size viewed from the exact same location.

The cathedral on the right is stretched, because of the behavior described above. If we would place the cathedral on a sphere with size of earth its dimensions would be correct. But because we are placing objects into an absolutely flat world (the three.js scene), we need to fix this by using the Mercator Projection. As mentioned above the projection is cylindrical meaning that we convert earth's spherical surface to a flat squared map, where the top edge belongs to north pole and the bottom edge belongs to south pole. The squared form comes from the idea of mapping every component of a latitude/longitude coordinate to an interval $[-180, 180]$ in a cartesian coordinate system. With this every longitude value is mapped to its own value. For example: $\vartheta = 100°$ will be mapped to the value 100. For the latitude one a more complex formula is needed:

$$f_{lat}(\varphi) = \ln(\tan((\frac{\varphi}{90°} + 1) \cdot \frac{\pi}{4})) \cdot \frac{180°}{\pi}$$

where $\varphi$ is the latitude value in degree. A proof of this formula can be found in [4].

Now we are able to map WGS-84 coordinates into cartesian coordinates. Last but not least we need to correct the size of distances to be realistic in meters by dividing every mapped value by 360 and multiplying it by earth's circumference of around 40 075 016.69 meters. As the value of the circumference is quite big, which can result to big values for vertices of the geometry, and floating-point numbers used by most computers are not one hundred percent accurate, we do one last step before the 3D scene is finished. This is not really problematic, but counteracting it prevents possible artifacts in the 3D visualization.

## 4.2 Centering the Scene

The technical standard IEEE 754 for floating-point arithmetic is used by many computers. Deno - which uses JavaScript under the hood - also represents floats with this technique. As stated above those numbers are not absolutely accurate which comes from the fact that those numbers are stored either in 4 or 8 bytes depending on the desired precision (in case of JavaScript it is 8 bytes also known as IEEE 764 Double Precision). By mean of this there are only limited possible different values to represent floating-point numbers. As there are infinite decimal numbers in reality this implies that there exist gaps between two neighboring numbers regarding to IEEE 754, which is in fact true[10]. Those gaps grow in increasing numerical values, which is shown in [10].

To decrease numerical values of vertices in geometry used in our scene we move all of it together to the center of the cartesian coordinate system. To determine how much the scene has to be moved an origin has to be calculated. This is done by finding the minimum and maximum value for the longitude coordinate. Note that we cannot do the same for the latitude as with its value the strength of the distortion changes. Let $lon_{min}$ be the minimum longitude value and $lon_{max}$ the maximum value. Then the value $\Delta_{lon}$ to adjust the coordinates can be calculated as:

$$\Delta_{lon} = \frac{lon_{min} + lon_{max}}{2}$$

For example, if we have $lon_{min} = 10$ and $lon_{max} = 20$ then $\Delta_{lon} = \frac{10+20}{2} = 15$ and every longitude value is recalculated by subtracting $\Delta_{lon}$ from it.

Furthermore, with centering the scene it is more convenient to develope and debug the application as the origin is right in the middle of it.

# 5 Shadows

Shadows are important when planning wind farms, because they can be a problem for residents as shown in the introduction of this work. Therefore, implementing those is important for a 3D visualization. Fortunately three.js already offers a way to easily add shadows in a scene. For this the property `shadowMap.enabled` in a `WebGLRenderer` instance[32] needs to be set and a `DirectionalLight` with a shadow object[33] needs to be added. This is already done in the figures given in Section 3. three.js uses the shadow mapping technique and offers three different filters for it out of the box. Those filters are used to blur the resulting shadows in the scene for better look. The filter used in this work is the Variance Shadow Map (VSM) algorithm. Choosing this one over other filters like Percentage-Closer Filtering (PCF) for example comes from the shadow results being more realistic and smoother, but with minimal additional memory storage and computation [5]. Figure 42 shows the result when enabling shadows for the scene using the VSM filter.

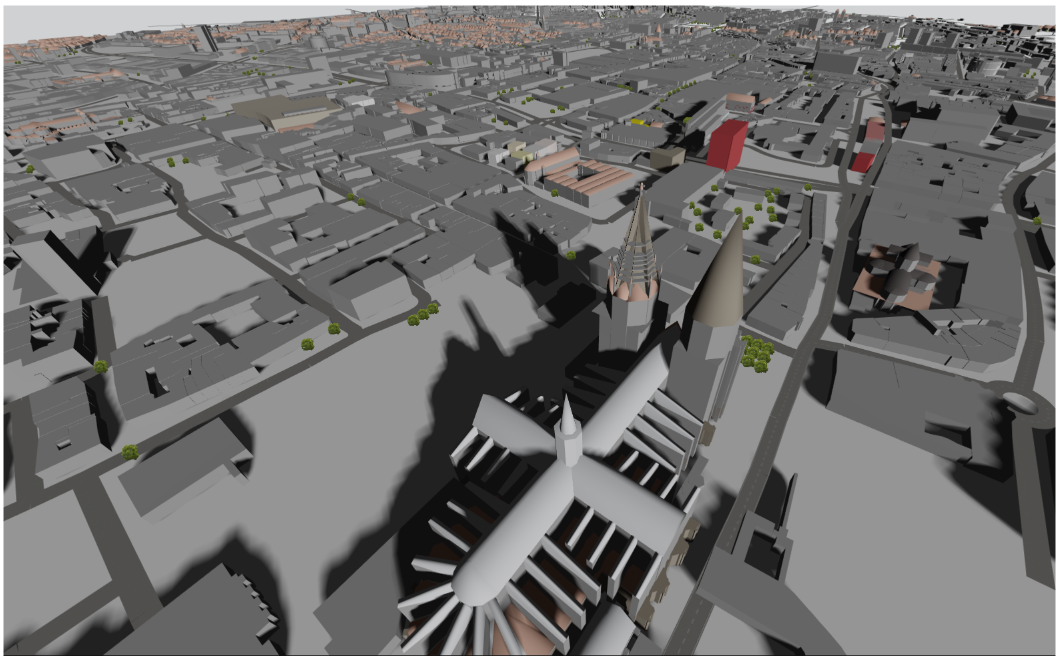


Figure 42: Shadow casting of Cologne Cathedral.

One thing to consider at the moment is that the distance of the directional light is fixed and can only be rotated to simulate the sun position in the sky. This can lead to objects far away from the origin (in this case $> 2\,500$ units) not to cast shadows, because the "view of the light" does not cover regions beyond. The problem is that the greater the allowed distance is set, the worse the quality of the shadows get without increasing the dimensions of the shadow map.

[32]https://threejs.org/docs/#api/en/renderers/WebGLRenderer.shadowMap
[33]https://threejs.org/docs/#api/en/lights/DirectionalLight.shadow

# 6 Performance Optimization

Performance optimization is fundamental for the application to run on slower devices. three.js already offers some techniques out of the box to increase performance. One of them is called Frustum Culling. This means instead of rendering all objects at the time, only objects which are within the camera view are rendered. But for complex scenes this optimization alone is not enough as performance tests done in this section reveal. Those tests will be run on the following three devices to compare the results:

- **Smartphone** iPhone X (iOS 14.0.1)
  - Browser: Safari
- **Laptop** Acer Aspire VN7-571G-52EP (Windows 10 (20H2))
  - CPU: Intel Core i5-5200U 2.20 GHz (2 cores)
  - RAM: 8 GB
  - Graphics Card: GeForce 940M (2048 MB dedicated memory)
  - Resolution: 1920 x 1080 pixels
  - Browser: Google Chrome
- **PC** Desktop (Manjaro Linux (Mikah 20.1.2))
  - CPU: Intel Core i5-3570 3.40 GHz (4 cores)
  - RAM: 12 GB
  - Graphics Card: GeForce GTX 660 Ti (2048 MB dedicated memory)
  - Resolution: 1920 x 1080 pixels
  - Browser: Google Chrome

The testing scene is Aachen Center with an area of around $4 \cdot 3 = 12$ square kilometers, which currently includes 14,245 building and 1,156 road meshes.

First performance tests with the default settings[34] of three.js are run. Afterwards another technique offered by three.js called Level of Detail (LOD) is applied and tested. Last but not least a geometry merging approach is tested and compared to the other two results. All optimizations are evaluated by frames per seconds (FPS) the device can perform. The more FPS the faster and therefore smoother the application. Additionally, the memory usage is compared, too, where less usage is better. stats.js[35], which is developed by the same author as three.js, is used to measure those two performance indicators, where latter is only available on the laptop and PC device, because Safari on iOS currently does not support the `Performance.memory` API[36]. At the end of this section loading times of a scene are also presented depending on the used technique.

[34] general settings like shadows or lights are enabled
[35] https://github.com/mrdoob/stats.js/
[36] https://developer.mozilla.org/en-US/docs/Web/API/Performance/memory

## 6.1 Rendering Tests

In the following the optimization of buildings only is considered. Exclusively for the buildings 353 844 faces were rendered at once. Furthermore, a value of 60 FPS is considered to be ”at least 60 FPS”, because browser generally limit the amount frames according to the refresh rate of the monitor, which in this case was 60 Hz for all three devices. The results of the following subsections will be packed into one diagram at the end for comparison. In this case the tests are run once with shadows enabled and disabled to give performance results between a variable setting.

### 6.1.1 Default

As already stated above three.js already offers and enables Frustum Culling by default for every mesh added to the scene. But it turned out that this is not sufficient for the scene to render smoothly on the devices as the results in Table 1 shows.

| Device | FPS (Shadows off) | FPS (Shadows on) |
|---|---|---|
| PC | 15 | 10 |
| Laptop | 11 | 7 |
| Smartphone | - | - |

Table 1: FPS of the scene with default settings once with shadows enabled and disabled for all three devices. The smartphone did not load the scene at all due to crashes.

### 6.1.2 LOD

LOD is a technique to define multiple versions of the same object with different levels of detail to then switch between those depending on the distance from the camera to the object’s location. The aim is to reduce complexity of objects far away, because details are not recognizable anymore. The same can be applied for buildings. For this we define a simple box, which covers a whole building part. All together when the distance between the camera location and the building location is less than 1 500 units, the original geometry of the building is rendered, otherwise the corresponding box is rendered. At first an optimization compared to the results above was expected, because less faces needed to be rendered, but the results are a bit worse, as Table 2 shows.

| Device | FPS (Shadows off) | FPS (Shadows on) |
|---|---|---|
| PC | 7 | 5 |
| Laptop | 7 | 5 |
| Smartphone | - | - |

Table 2: FPS of the scene with LODs once with shadows enabled and disabled for all three devices. The smartphone did not load the scene at all due to crashes. Furthermore, the best case scenario was given, meaning that all buildings were rendered as their lower level boxes.

This implies that the bad performance does not lie in the complexity of the geometries. The additional loss probably comes from checking the distances for every object in every frame.

### 6.1.3 Merged

Another reason why 3D applications run slowly is the amount of draw calls per frame [6]. In three.js one draw call is done per mesh. So for the scene there are 15 401 draw calls done regarding to the buildings and roads. The next step now is to merge the geometry of those meshes into one single mesh. That means all buildings and all roads of the same type are merged into one mesh each. With this, flexibility regarding the transformation of single objects is lost. But this is not needed in this case, because buildings and roads are static, and therefore merging is no problem. After applying this approch the performance is increased by at least 300% in average[37] compared to the default settings as Table 3 shows. The amount of meshes is reduced to five (1 for buildings and 1 for each road type).

| Device | FPS (Shadows off) | FPS (Shadows on) |
|---|---|---|
| PC | 60+ | 60+ |
| Laptop | 31 | 25 |
| Smartphone | 60+ | 40 |

Table 3: FPS of the scene with merged geometries once with shadows enabled and disabled for all three devices.

All results are available in Figure 43 for direct comparison.

Something to note here is that the scene (representation of Aachen Center) has a lot of buildings compared to a village where the planning of a wind farm is more common. Therefore, even better performance is to be expected for smaller towns.

[37]The results of the smartphone are excluded, because they are only available for the merged approach.

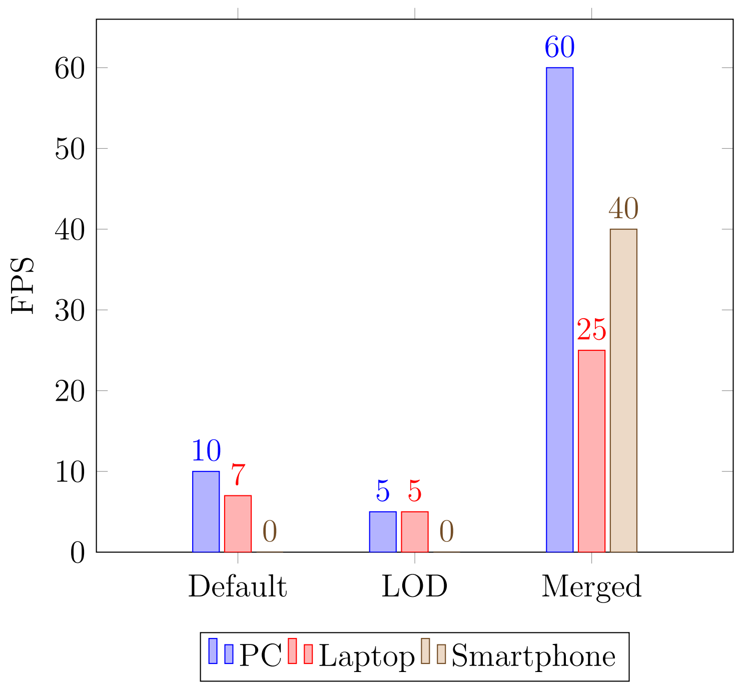


Figure 43: Rendering performance. As the results show the best of the shown methods is the merge approach. As explained above this comes from the reduction of draw calls to the graphics card, because complexity reduction did not increase the performance as shown in the LOD approach.

## 6.2 Memory Usage

Not only FPS is increased, but also memory usage is minimal with the merged approach. The default settings used around 109 MB of RAM. The LOD technique used around 220 MB of RAM, which is comprehensible because twice the amount of objects need to be held in memory. Merging everything together, the used memory could be reduced to just 49 MB, which results from holding just 5 instead of 15,401 mesh objects in memory. Regarding mobile devices battery usage is therefore also reduced, where exact values are not measured here.

## 6.3 Loading Times

Loading times are also an important factor, because they influence the flow of an application. Here the fastest solution is the merged approach again, which results from the same reason that there are only 5 meshes to be processed and loaded to the graphics card. The following loading times are measured from the beginning of the script until the first rendering on the client. Both the server and client were running on the same machine, which is why the loading time for fetching the scene data from the server is very low. So depending on the internet connection, time is to be added to the values given in Table 4.

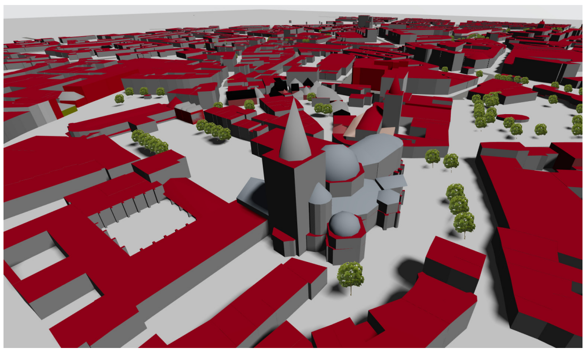
| Device | Default | LOD | Merged |
|---|---|---|---|
| PC | 17.10 | 19.48 | 0.44 |
| Laptop | 39.48 | 40.30 | 1.05 |
| Smartphone | - | - | 1.17 |

Table 4: Loading times in seconds of the application on client side measured from the beginning of the script until the first rendered frame. This was done 3 times for every constellation.

## 6.4 Comparison to OSM2World

As already stated in the introduction the performance of OSM2World is a little worse compared to the application in this work regarding to FPS measurement. This is shown by a smaller test scene (2 km $\times$ 1 km = 2 km$^2$) of Aachen Center run on the PC. While OSM2World reached around 45 FPS, the application of this work reached 60+ FPS. To note here is that OSM2World currently supports some more objects like bus stops, fences, benches etc. which are 3D representated in their visualization.

Furthermore, OSM2World in its current version (0.2.0) has errors in some of their representation of building parts. For example Aachen Cathedral is not complete as Figure 44 shows.


(a) Our version.

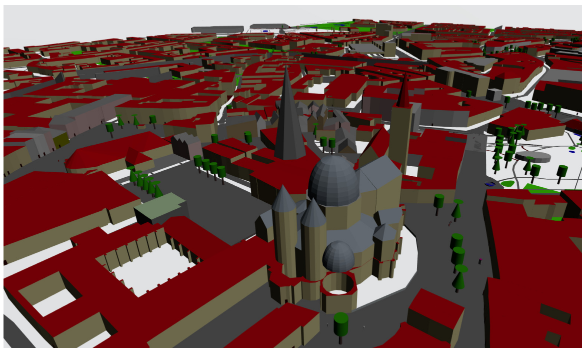

(b) OSM2World version.

Figure 44: Comparison of Aachen Cathedral between our application and OSM2World. Additionally, missing roofs were colored in red to give a better comparison to OSM2World. Visibly the front tower is missing on the right as well as another building part at the bottom.

# 7 Maintaining OSM Data

Maintaining OpenStreetMap data is one of the most important tasks to keep the service up to date and consistent. Moreover it helps to make a 3D visualization as realistic as possible. As already stated in the introduction of this work, data in OSM is maintained by a community of volunteers who collect and add information to the system in their free time. Related to this work this possibility is important, because by choosing an area to plan a wind farm in there could be too little information (e.g. buildings and their details) available to give a realistic 3D representation of. Current observations show that there is more data available in cities than in little towns, where a wind farm is rather to be planned as there is more space in most cases. Therefore, extra work should be done by visiting the choosen area and collecting missing details to add them to the OSM databases before creating a 3D visualization. In the following, we will go through small steps needed to collect relevant data, organize it and publish those information in OSM.

## 7.1 Collecting Information

Results of instances of scenes shall be as precise as possible. This means silhouettes of constructions for example should not differ too much from their real representation, because wrong dimensions lead to wrong shadow castings. Therefore, it is important to keep some properties in mind to collect for. Furthermore, locations of those constructions are important. But these can be gathered from satellite pictures OSM provides while adding data in their editor. These include roads, forests and more as well.

In this case constructions are the most important objects to collect for. Especially their dimensions like form, size, roof and color. One way is to take pictures of buildings and other constructions. If taking pictures is not appropriate directly taking down notes about an object is another option. To not forget associations of those objects with their location, one strategy can be to go street by street, where every building on the right side is noted one by one. This means a street needs to be walked two times at least. Additionally, the starting point of a street should be written down to prevent confusing the direction.

Heights should be estimated, because there is not always the opportunity to measure a buildings facade height exactly with suitable tools (for legal reasons).

## 7.2 Publishing

To be able to publish data to OSM an account needs to be created, where no more than a username, email and password has to be given. After verifying the email, you can go right into managing data in OSM. Navigating to the location chosen, the taken notes can be added one by one. OSM offers an own wiki page for contributing map data to their databases.

# 8 Conclusion and Future Work

In conclusion this work shows that OpenStreetMap is able to offer enough geographic information to make a 3D visualization of a region possible. Here the level of detail depends on how many tags are specified on a given object. Furthermore, the correctness of those values is important to avoid misleading results. Therefore, preparation and maintenance of those data is advantageous. But there are also some objects missing to be supported by the application. These include roof types, where 6 of 12 in total are currently implemented in the application (5 where shown in this work), vegetation types, benches, bus stops and many more. Additionally, the performance is a big issue in this work. At the beginning 14 245 buildings were rendered at 15 FPS on the PC, which is a poor performance considering the human eye can only see smooth transitions from 24 FPS up. With optimizations presented in Section 6 the performance is increased by at least 300% in average. 60+ FPS were reached on the PC as well as on the smartphone. But not only FPS were increased, memory usage and loading times were also decreased with the same approach. On one hand there is still the possibility to fine tune the scene to get a little bit more performance by removing unseen faces of objects. But on the other hand this would lead to more processing time to calculate and find those cases, which is not exceeded by the value of the outcoming improvement and therefore unnecessary to implement at the moment.

At the moment every scene, regardless of the given coordinates, lies in an absolutely flat world, which is not realistic because earth's surface is not perfectly flat. To implement elevation in our application a data source is needed. NASA for example offers land elevation data with a precision of 30 meters collected on their Shuttle Radar Topography Mission[38] (SRTM) in 2000. Since 2015 this information is available globally and can be downloaded freely under condition of having a NASA Earthdata account[39]. The data comes in pixels where each pixel represents an area of $1 \times 1$ arcsecond, which translates into approximately $30 \times 30$ meters.

Applying elevation data comes with some problems to be solved. Firstly buildings must not be tilted by their underlying ground bevel. But without tilting, buildings at a sloped location will partly float above the ground. To prevent this buildings could be extended downwards so that neither the actual building is clipping the ground nor the extended building is floating above it. Secondly roads need to be adjusted, too. This can be more complex, because in some situations the mesh itself needs to wrap around edges with different slopes to prevent clipping them. Therefore, checking against those and changing the geometry of a road mesh is necessary. Altogether [7] is referred here for integrating SRTM data into 3D rendering together with OpenStreetMap data.

Another subject to discuss is the look of objects in the world. Currently for example buildings do not have any structure and just look blank. Applying textures to them would make them look more realistic, which is what OSM2World does, too. Fortunaly OpenStreetMap offers a public collection of textures in their offical texture library[40].

[38] https://www2.jpl.nasa.gov/srtm/

[39] https://dwtkns.com/srtm30m/

[40] https://wiki.openstreetmap.org/wiki/Texture_Library

They can be used freely and do have dimensions given, which describe how big they are in the real world in meters. One challenge here will be to apply them to the single existing buildings mesh generated in Section 6. There is one option available to do: collect all textures representing building facades and roofs and pack them together into one single texture. At the same time the position (i.e. UV coordinates) and dimensions for each packed texture must be saved and be available to the application. This is necessary, because then a specific facade or roof can be selected to be displayed for a building without cutting a shown window for example.

## References

[1] Overpass API, . URL `https://wiki.openstreetmap.org/wiki/Overpass_API`.

[2] Overpass API User's Manual, . URL `https://dev.overpass-api.de/overpass-doc/en/`.

[3] Overpass QL, . URL `https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL`.

[4] Daniel Daners. The mercator and stereographic projections, and many in between. *The American Mathematical Monthly*, 119(3):199–210, 2012. doi: 10.4169/amer.math.monthly.119.03.199. URL `https://www.tandfonline.com/doi/abs/10.4169/amer.math.monthly.119.03.199`.

[5] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, page 161–165, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 159593295X. doi: 10.1145/1111411.1111440. URL `https://doi.org/10.1145/1111411.1111440`.

[6] Sébastien Hillaire. Improving performance by reducing calls to the driver. In Patrick Cozzi and Christophe Riccio, editors, *OpenGL Insights*, pages 353–364. CRC Press, 2012.

[7] Tobias Knerr. Merging Elevation Raster Data and OpenStreetMap Vectors for 3D Rendering. May 2013.

[8] M. C. Lin, J. Sewall, and D. Wilkie. Transforming gis data into functional road models for large-scale traffic simulation. *IEEE Transactions on Visualization & Computer Graphics*, 18(06):890–901, jun 2012. ISSN 1941-0506. doi: 10.1109/TVCG.2011.116.

[9] Mark Monmonier. *Rhumb lines and map wars: a social history of the Mercator projection.* University of Chicago Press, United States, 2004. ISBN 9780226534329. Includes bibliographical references (pages 207-229) and index.

[10] Michael L. Overton. Floating Point Representation and the IEEE Standard. pages 7–21, 1997.

[11] W. Randolph Franklin (WRF). PNPOLY - Point Inclusion in Polygon Test. URL `https://wrf.ecse.rpi.edu//Research/Short_Notes/pnpoly.html`.