UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

Master Thesis
of
Tobias Knerr

Merging Elevation Raster Data and OpenStreetMap Vectors for 3D Rendering

Supervisor:
Prof. Dr. Franz J. Brandenburg
Chair for Computer Science
Department of Informatics and Mathematics
University of Passau

May 2013

Abstract

The free OpenStreetMap database contains valuable data for 3D rendering, but elevation needs to be extracted from an external source such as NASA's SRTM raster data. Among several possible algorithms for approximating a terrain surface, natural neighbor interpolation and especially piecewise approximation using least squares yield good experimental results. A proposed framework of connectors and constraints may serve to further refine the terrain surface by inserting information from OpenStreetMap's vector data. Linear programming can be applied to the problem of performing this refinement. A prototype implementation based on the open-source 3D renderer OSM2World produces examples for advantages and disadvantages of that approach.

Contents

1. Introduction

1.1. Motivation

Virtual models of real-world landscapes have a wide range of potential applications, including automotive navigation systems, tourism, the communication of planned construction projects to the public, simulations and computer games. With sufficient computing power available on a wide range of devices, customers of software incorporating geographic data have come to expect a high quality of visual presentation. This calls for a widely available, global data source for 3D landscape models.

OpenStreetMap may be an answer to this. Inspired by the Open Source movement and wikis, most prominently Wikipedia, the volunteer project provides freely licensed, crowdsourced geographic data to the public. It has a large international community of more than one million registered users.[1] And although it should be noted that only 38 % of registered users in 2011 had actually contributed edits to the project's database, with an even smaller number contributing on a regular basis – a thorough analysis of this aspect of the community is available in [NZ12] – the growth continues at a remarkable pace. As a result, the project's basic road network now approaches a level of completeness comparable with that of commercial datasets in some countries, as confirmed for Germany by [NZZ11].

It seems like a worthwhile endeavor to explore the use of this vast dataset for 3D rendering. In addition to the numerous use cases in consumer-facing applications, attractive 3D visualizations would also be a source of motivation for the project's contributors, and may even be integrated into editing software.

1.2. Previous work

When advertising products, the term "3D" is used in a very broad sense, even for fully two-dimensional maps which are rendered with a perspective projection. This style of visualization has already been widely implemented in automotive navigation systems.

Actual three-dimensional modelling of terrain, man-made structures, or both, is a more challenging task. Nevertheless, a large number of approaches and products exist.

[1] `http://www.openstreetmap.org/stats/data_stats.html`

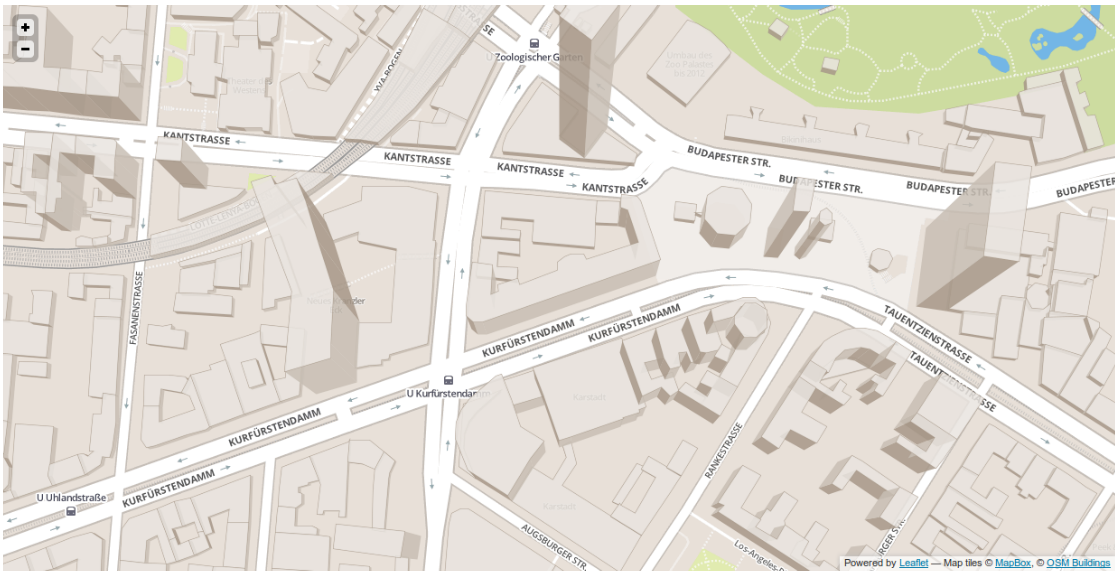


Figure 1.1.: Building visualization by OSMBuildings

1.2.1. Three-dimensional rendering of features

Contemporary applications go beyond a small number of landmarks and may use 3D building models, as well as models for other features, on a large scale. This is available even in browser-based maps today, using JavaScript and technologies such as WebGL or Canvas. A prominent example is Google Maps, which presents an abstract rendering of buildings with outlines and faces shaded in gray. Similar visualizations have been created from OpenStreetMap data by WikiMiniAtlas[2] – an interactive map used by Wikipedia – and the OSMBuildings library.[3]

The most comprehensive evaluation of OpenStreetMap tags related to 3D buildings so far has been implemented by Kendzi3D, which is available as a plugin for the JOSM offline editing software.[4] The tool has also pioneered the use of 3D visualization in providing direct feedback to OpenStreetMap contributors.

[2]http://meta.wikimedia.org/wiki/WikiMiniAtlas
[3]http://osmbuildings.org/
[4]http://wiki.openstreetmap.org/wiki/JOSM/Plugins/Kendzi3D

1.2.2. Three-dimensional terrain rendering

Three-dimensional terrain does not necessarily imply the rendering of three-dimensional buildings and other features. Sometimes a two-dimensional map is simply textured on top of heightfield terrain. Maperitive[5] is an example for a widely used OpenStreetMap rendering tool which offers this feature, though by no means the only one – many other programs offer similar visualizations.

1.2.3. Integration of terrain and three-dimensional feature models

A range of popular three dimensional globe applications for various platforms is offered by Google under the Google Earth brand.[6] At the core of the underlying service is satellite and aerial imagery textured onto a terrain height field. As pointed out by [BN08], the terrain shape often does not match the features depicted in the imagery (see e.g. figure 1.3). Furthermore, features interacting with the terrain (like tunnels) cannot be represented by this approach alone.

To augment their virtual globe, Google are using 3D models of buildings and other structures that are placed on top of the textured terrain. Originally, these were manually created by users of the service in 3D modelling tools – such as the SketchUp modeller then owned by Google –, but Google are now gradually adding meshes auto-generated with photogrammetry. These provide a more detailed and realistic appearance and also improve terrain surfaces, but occasionally suffer from visible deformities at a close range. They also lack semantic information, which is often present in manually created vector data and could serve different purposes: Some attributes, such as materials, can improve the rendering itself by applying advanced effects like reflections or animations. Others serve as a bridge between rendering and other applications, e.g. by linking the routing network with exact locations of roads or even lanes in the 3D landscape, or connecting points of interest with the correct doors of a building model.

To address the quality issues observed with textured terrain surfaces such as those traditionally used in GoogleEarth, [BN08] suggests an alternative approach for merging terrain and features. By representing not only the appearance of features, but also their effects on elevation as textures, they are able to make full use of the capabilities of shader programs executed on graphics hardware. Like the alternative approaches, they augment the visualization with mesh models for certain feature types.

3D rendering of terrain and OpenStreetMap data in particular has been previously studied by researchers at the Department of Geography at the University of Heidelberg. [OSN^{+}09] describes the integration of ground-level features such as roads into a Triangulated Irregular Network (TIN), as an alternative to textured elevation models. This enables a further improvement of terrain quality using known properties of linear fea-

[5]`http://maperitive.net/`
[6]`http://earth.google.com`

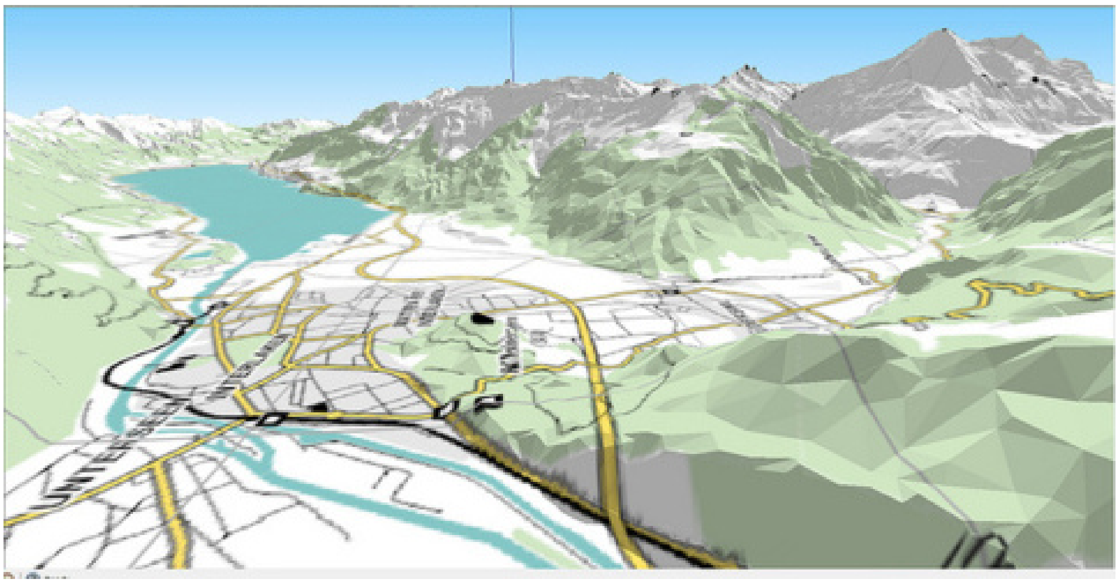


Figure 1.2.: 3D terrain in Maperitive, sample image taken from maperitive.net

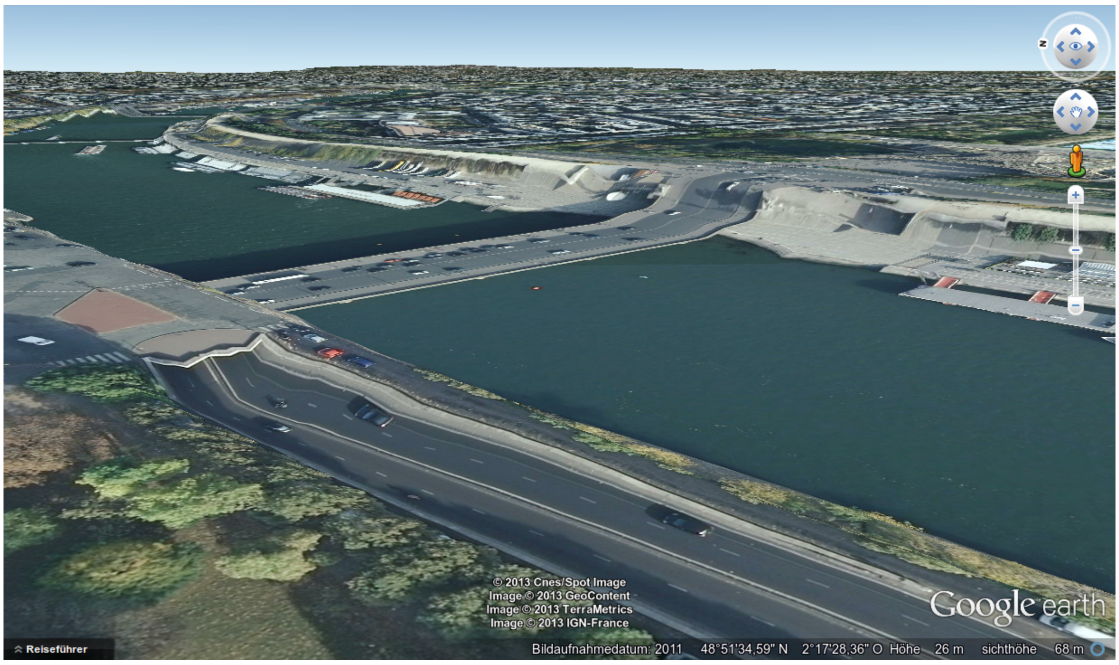

© 2013 Cnes/Spot Image
Image © 2013 GeoContent
Image © 2013 TerraMetrics
Image © 2013 IGN-France
Google earth
Reiseführer
Bildaufnahmedatum: 2011 48°51'34.59" N 2°17'28.36" O Höhe 26 m sichthöhe 68 m

Figure 1.3.: Discrepancies between terrain surface and textured features in Google Earth.

tures, for example making sure that road surfaces are flat. To calculate the effect of such properties on the TIN, [SLNZ09] proposes spring-based optimization – a technique inspired by simulating the mechanical properties of springs. Based on their technology, the department is hosting OSM-3D[7]. This worldwide service features simple feature meshes derived from OpenStreetMap data, and the visualization can be further improved with building models uploaded through the OpenBuildingModels service.[UZ12]

1.3. Data sources

1.3.1. OpenStreetMap

OpenStreetMap serves as a major data source for the software developed as part of this thesis – features such as roads, buildings or landcover information are based solely on OpenStreetMap data. However, OpenStreetMap is less useful as a source for elevation data. Not even 2.5 % of the nodes in the OpenStreetMap database carry an elevation attribute in addition to the required latitude and longitude values.[8] As shown by the Taginfo service, an unusually large percentage of these nodes also carry tags indicating that they have been created through imports, rather than manual contributions. In addition, the correlation with `name` suggests that mostly named objects such as peaks and points of interest have elevation information available. With these limitations, OSM data alone is not sufficient to provide a terrain model for 3D rendering.

OpenStreetMap, however, also provides data regarding relative elevation of features. Examples of attributes in that category include `layer` (which is used on 70 % of bridges[9] and tunnels[10]), `incline`, and others. This data is not as easily available elsewhere and can serve to improve 3D models beyond basic terrain shapes. As popular OpenStreetMap editing programs already contain the necessary tools to conveniently edit these attributes, future widespread coverage is more likely than with absolute elevation.

The OpenStreetMap data model is explored in detail in chapter 2.

1.3.2. SRTM

A widely-used source of surface elevation data is the SRTM (Shuttle Radar Topography Mission) dataset, based on radar measurements from a NASA Space Shuttle. It contains data from 60° N to 56° S, at a resolution of 1 arc-second. Full resolution raw data has only been published for the territory of the United States of America, but data with 3 arc-second resolution is available for most of Earth's land surface.

[7] `http://www.osm-3d.org/`
[8] `http://taginfo.openstreetmap.de/keys/ele`
[9] `http://taginfo.openstreetmap.de/keys/bridge`
[10] `http://taginfo.openstreetmap.de/keys/tunnel`

NASA themselves have published several versions of the dataset. Version 1 refers to the original raw data, whereas version 2 has been improved by clipping data to coastlines and fixing single pixel errors, with additional recalculations of the three-arc second data in version 2.1.

Even the latest version published by NASA still contains significant voids. Several interpolation methods have been employed to fill them, as described in [RNJ07]. Additional postprocessed SRTM datasets have been made available by various providers, including the CGIAR[11]. However, these often impose additional legal restrictions when compared with the original Public Domain dataset and may not be compatible with OpenStreetMap's license according to the OpenStreetMap wiki[12].

Due to the process used to obtain SRTM data, the elevation values may represent any reflective surface rather than terrain elevation. Therefore, buildings and vegetation affect the measured values. Although this thesis ignores the difference, it will likely be necessary to take it into account for highest quality results in the future. Previous efforts include OpenDEM[13], which uses OpenStreetMap landcover data for SRTM postprocessing.

SRTM data and OpenStreetMap have been used together elsewhere, and there has been some work to make SRTM data available in the .osm data format used by many OpenStreetMap tools and editor programs. So far, these efforts have focused on generating contour lines as commonly used on two-dimensional maps (e.g. Contour lines in OSM format[14]) or altitude differences along routes (e.g. srtm2wayinfo[15]). Neither fits the purposes of this thesis, though, which instead directly uses the 2.1 SRTM data files as published by NASA.

[11]`http://srtm.csi.cgiar.org/`
[12]`http://wiki.openstreetmap.org/wiki/SRTM`
[13]`http://opendem.info/`
[14]`http://geoweb.hft-stuttgart.de/opendtm.html`
[15]`http://wiki.openstreetmap.org/wiki/Srtm2wayinfo`

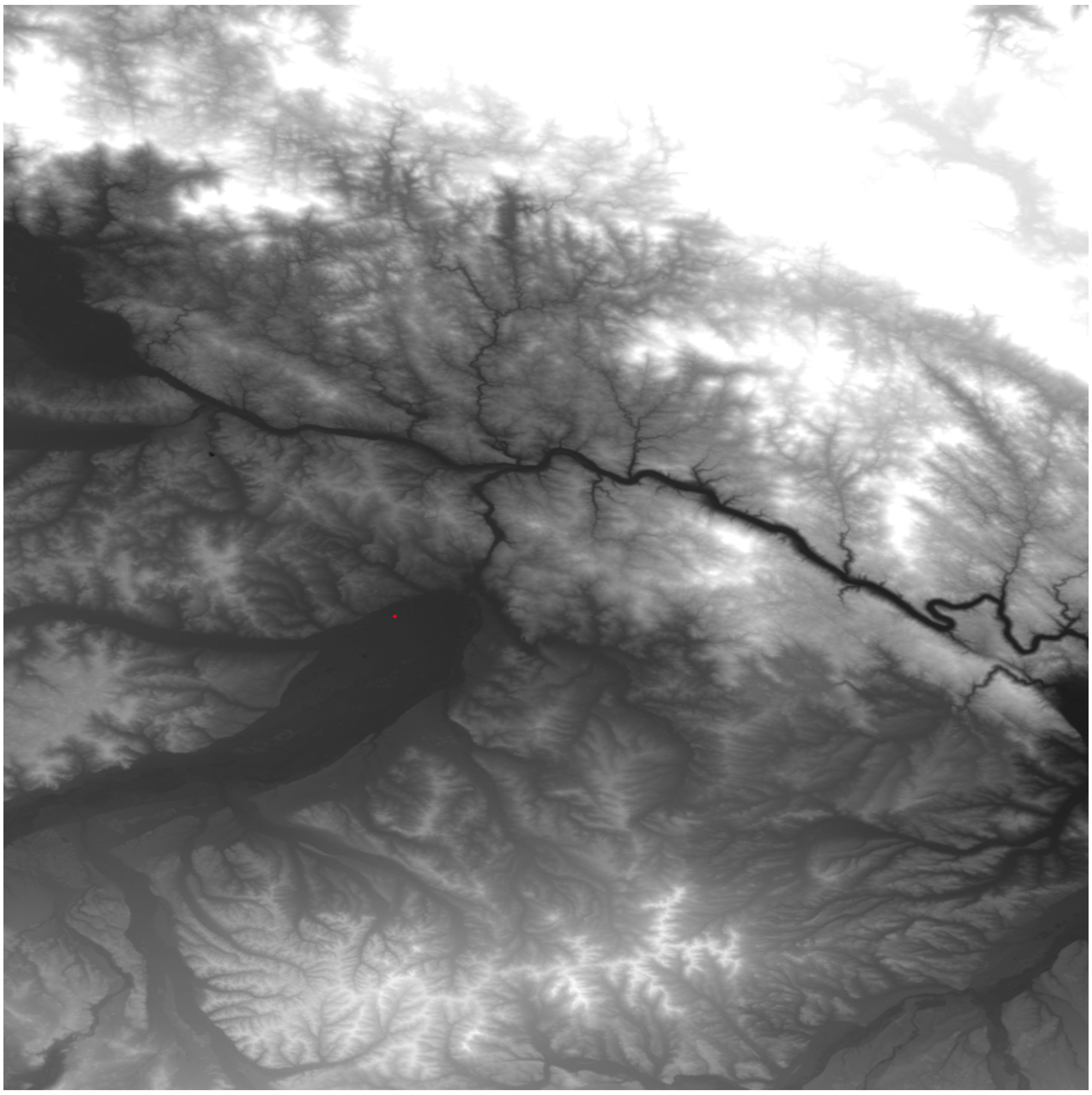


Figure 1.4.: Elevation values in tile N48E013 of SRTM version 2.1. Voids are shown in red.

2. Preparation of OpenStreetMap data

If OpenStreetMap data is to be used as a data source for 3D rendering, significant preprocessing of the originally two-dimensional dataset is necessary. As a first step, we will create three-dimensional models from OpenStreetMap data. Chapter 3 is dedicated to creating a terrain surface from SRTM data or similar sources. In chapter 4, we will finally integrate the models from this chapter with the terrain.

2.1. OpenStreetMap data model

2.1.1. Primitives

As expected for a geographic database, all OpenStreetMap data is connected to locations on the globe. The most basic geographic primitive is the **node** – a point on the globe with latitude and longitude. Isolated nodes may be used to represent small features such as trees or street lights, or even to temporarily represent larger features where the exact extent has not been surveyed yet. Because nodes are the only primitive that directly carries coordinates, all other primitives are built on top of nodes in some manner.

An ordered list of nodes defines a **way**. This primitive can represent any linear feature, such as a stream or a railway. Ways inherently have a direction due to their definition as ordered lists. The direction is relevant for some features, including e.g. oneway roads and cliffs. Nodes may appear in a way multiple times. The most common case of this, where the first and last node are identical, is often called a *closed way* in OpenStreetMap documentation and applications.

Relations are the most complex primitive. They reference an ordered sequence of members, which may be nodes, ways or other relations. An Unicode string, the role, is associated with each member in a relation. Members and roles may appear in a relation multiple times.

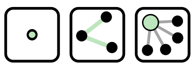


Figure 2.1.: Node, way, and relation. These icons from the OpenStreetMap wiki are widely used in the project's documentation.

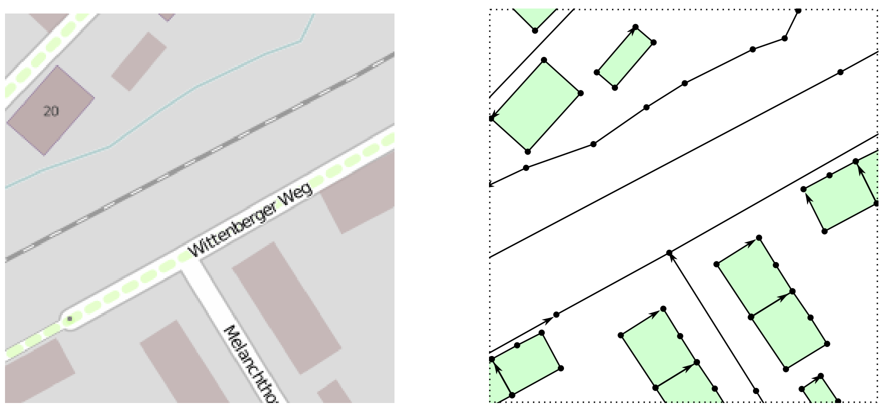


Figure 2.2.: 2D bitmap rendering from openstreetmap.org (left) and schematic data representation. Several nodes and ways are visible, some of the ways are closed and represent areas.

2.1.2. Tags

All the primitives described above can carry attributes, called **tags**. Each tag consists of two Unicode strings with up to 255 characters each, the key and the value. The `key = value` format, a common notation for tags, is used in this thesis.

Tags determine what type of feature is represented by a primitive, and can provide all sorts of additional information about the feature. The OpenStreetMap API is unaware of the meaning of keys, values and roles, thus leaving it to the project's community to define the semantics of the data model.

2.1.3. Areas

Areas are not actually a primitive of their own, even though the introduction of an area primitive is being considered for the next revision of the OpenStreetMap API. Currently, areas are still modeled using one of two approaches.

Most commonly, closed ways are used to model an area. Whether a given closed way represents an area or a ring-shaped linear feature depends on its tags. The most straightforward distinction is achieved by explicitly tagging a way as `area = yes` or `area = no`, but many other tags also implicitly turn a closed way into an area.

The second, more powerful approach for modelling areas is the multipolygon relation. Such a relation may have multiple ways as its members which can be assembled into closed rings. Ways with the `outer` role form the outline of the polygon(s), ways with the `inner` role form holes. In the standard case, tags for the area are added to the

relation #18			
members		tags	
role	member	key	value
outer	way #39	type	multipolygon
inner	way #51	landuse	forest
outer	way #97		
outer	way #42		

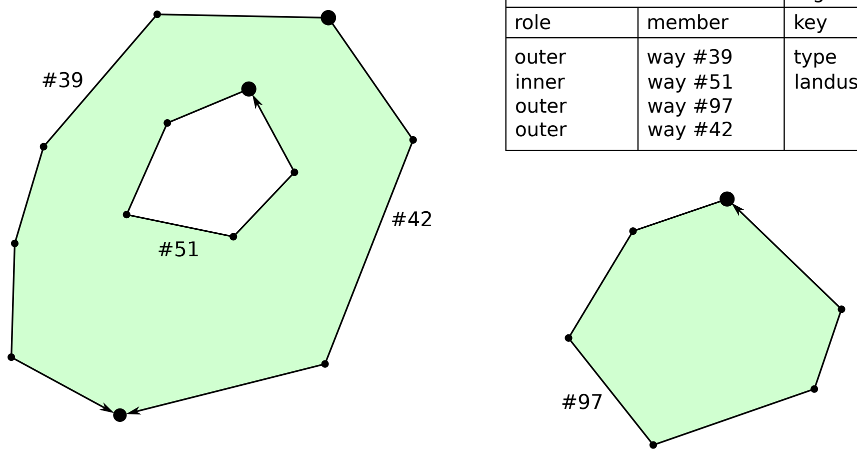


Figure 2.3.: An example for a forest area modeled as a multipolygon relation with two outer rings and one hole. Note that neither way directions nor the order of members affect the area's appearance.

relation in addition to a `type = multipolygon` tag, while the ways may represent features themselves.

Multipolygons need to be preprocessed. For the purpose of the implementation of this thesis, they are treated as multiple independent polygons with identical tags, each possibly containing a number of holes. The implemented algorithm is based on the one outlined in the OpenStreetMap wiki at *Relation:multipolygon/Algorithm*.[1]

2.2. Relevant tags and relations

The lack of API limitations on tags is the foundation for the community's creativity, but makes it necessary to decide on a subset of the huge number of possible tags that may be useful for a given task.

Commonly used tags are documented in the OpenStreetMap wiki,[2] which can be edited by any user, just like the OpenStreetMap database itself. Statistical tools such as Taginfo[3] show the popularity and regional distribution of a tag.

This section provides an overview of tags and relations that are relevant for 3D rendering.

[1]`http://wiki.openstreetmap.org/wiki/Relation:multipolygon/Algorithm`
[2]`http://wiki.openstreetmap.org`
[3]`http://taginfo.openstreetmap.org`

2.3. Universal properties

`ele`	Absolute elevation. As explained in 1.3.1, this key is not common enough to replace SRTM. The project's wiki defines that values should use the WGS84 standard, but manually added values do not always follow this definition.
`incline`	Incline of a way, given either as a percentage (negative values indicate an incline opposite to the way direction) or just roughly as `up`/`down`. If the incline varies along a way, the maximum incline is used, so the average incline may be lower.
`layer`	Relative vertical ordering as an integer between -5 and 5, defaulting to 0. If two features overlap, then the feature with the higher layer is above the other.
`direction`	The direction a feature is facing. This allows nodes and areas, which do not have an inherent direction as ways do, to represent directed features. Values can be given as clockwise angles in degrees relative to north, or as a rough cardinal direction such as `NW`.

Measurements

Values refering to distances are assumed to be in meters by default, but other units can be used explicitly as part of the value.

`height`	Height of a feature.
`length`	Length of a feature.
`width`	Width of a feature. Commonly, but not exclusively, used on ways.

Materials and surfaces

To choose the texture of a feature in 3D rendering, the following tags are often helpful:

`material`	Material a feature is made of.
`surface`	Surface material of a feature. Commonly, but not exclusively, used on roads.

2.3.1. Buildings

Buildings are a prominent feature in many 3D visualizations. Since the availability of high quality aerial imagery for tracing, building coverage in OpenStreetMap has increased a lot, although the level of detail varies greatly between different regions. The tags described here are documented on various wiki pages.[4,5,6]

Basic tags

`building`	Required key to mark a feature as a building, usually applied to areas. The value may simply be set to `yes`, but can alternatively be used to define a building type. Such values, e.g. `garages` or `hut`, can be represented by applying different textures to alter the appearance of the building.
`building:part`	Can be used on additional areas inside the building. It models a section of the building that has different tags than the building as a whole, such as a different height, color, or its own name.

Building relation

It is possible to group all parts and other elements of a building by adding them to a relation with the `type = building` tag. In situations where buildings overlap vertically, this is necessary to avoid ambiguity, and “Simple 3D Buildings” recommends it for all buildings with building parts.

However, this is currently used only for a minority of buildings. To evaluate the others, it is still necessary to check whether a building part is geometrically contained within a building’s outline polygon.

[4]`http://wiki.openstreetmap.org/wiki/Key:building`
[5]`http://wiki.openstreetmap.org/wiki/Simple_3D_Buildings`
[6]`http://wiki.openstreetmap.org/wiki/User:Aschilli/ProposedRoofLines`

Roof shape

`roof:shape`	Defines the shape of the building for simple roofs. `building:roof:shape` is a common synonym.
`roof:orientation=along`	A building where the roof ridge is parallel to the longest wall, the default case.
`roof:orientation=across`	A building where the roof ridge is orthogonal to the longest wall.
`roof:angle`	Angle of the main roof face in degrees.

For more complex roofs, it can be necessary to explicitly draw the roof's geometry:

`roof:ridge=yes`	An additional way within the building outline defining a ridge of the building's roof.
`roof:edge=yes`	An additional way within the building outline defining an edge of the building's roof.
`roof:apex=yes`	An additional node within the building outline describing a peak point of the building's roof.

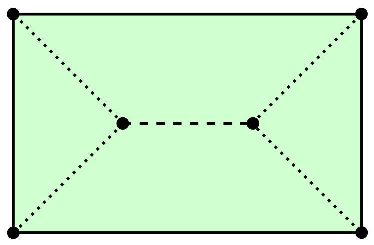


Figure 2.4.: A hipped roof modeled using `roof:ridge` (dashed) and `roof:edge` (dotted) ways, serving as a simple example for explicit roof geometry. The result should be similar to `roof:shape=hipped`, but more complex shapes may have no corresponding `roof:shape` value.

Heights and levels

`height`	The height of the building or building part, not counting underground levels. Usually in meters, but a different unit can be given in the value.
`min_height`	For building parts above the ground, this indicates the distance from the ground.
`building:levels`	The number of levels between the ground and the roof. For texturing, this allows to display an appropriate number of rows of windows. When there is no exact height available, this can also be used to calculate a height estimate.
`building:min_levels`	Like `min_height`, this applies to building parts above the ground.
`roof:height`	The height of the roof itself, not counting the height of the levels below.
`roof:levels`	The number of levels in the roof itself, not counting the height of the levels below.
`levels`	Used on an individual component of the building, such as the entrance nodes, this defines which building level it is on. The lowest ground level is usually treated as level 0.

Materials

`building:material`	The outer material of the building facade, including values such as `plaster`, `brick` or `glass`. This can usually be translated directly to a texture for the building's walls.
`building:colour`	The color of the building facade. In addition to words like `red`, it is also possible to use an arbitrary RGB value in hexadecimal notation (e.g. `#FF30A4`). The texture for the building's walls can be modified based on this key's value.
`roof:material`	Like `building:material`, but for the roof.
`roof:colour`	Like `building:colour`, but for the roof.

Entrances

`building = entrance`	Marks a node as a building entrance. May be omitted if an `entrance` key is present.
`entrance`	Marks a node as a building entrance. The key's values can be used to distinguish between entrance categories, such as main entrances or emergency exits.

Tags not relevant for 3D rendering

There are many other tags commonly found on buildings, such as house numbers (`addr:housenumber`). However, these have no or only marginal use for 3D rendering.

Likewise, we do not use shops and other points of interest commonly found in or at buildings for rendering, although certain practical applications of 3D landscapes would still display icons or other markers for these in addition to the scenery itself.

Possible future developments

The OpenStreetMap community is continually experimenting with new tags for a more detailed description of buildings. For example, the OSM-4D[7] draft suggests further extensions of the tag catalog, allowing for more flexible roof shapes, dormers, and attributes for other feature categories. These concepts are not frequently used in the database yet, but may easily become an essential ingredient of 3D building rendering if they are more widely adopted by the community.

[7] `http://wiki.openstreetmap.org/wiki/OSM-4D/Roof_table`

2.3.2. Transport networks

Many features – most importantly roads and railways – are represented as networks of ways in OpenStreetMap. A common property of this feature category is the convention to that shared nodes between ways are used to model same-level junctions. A challenge for 3D rendering is to turn the nodes and ways into polygons.

Basic tags

`highway`	Required key to mark a way as a road, track or path, usually applied to areas. The available values can be used to choose default values for surface texture and width to visually represent the large difference between e.g. `motorway` and `footway`. Not all tags using the `highway` key represent roads, however – the key also includes related features such as street lights and motorway rest areas, so care should be taken to treat these correctly.
`highway = steps`	One possible value for the `highway` key, steps require special handling in rendering.
`railway`	Required key to mark a way as a railway, usually applied to areas. As with `highway`, various values exist, and some are intended for railway infrastructure rather than actual rails.

Bridges and tunnels

`bridge = yes`	Indicates that a way runs across a bridge. Values other than `no` may indicate particular bridge variants.
`tunnel = yes`	Indicates that a way runs through a tunnel. Values other than `no` may indicate particular tunnel variants. In particular, `tunnel = building_passage` represents a way running though a building instead of through the ground, which is to be taken into account for building rendering.

Ways on top of bridges or in tunnels do not share nodes with ways below/above. This allows distinguishing them from same-level junctions.

Area and node features

A closed way that carries both the `highway` key and `area = yes` is commonly used for town squares or plazas. More generally it represents a component of the road network where traffic (often, but not necessarily pedestrian) moves freely and is not confined to lanes or directions of travel.

Nodes of `highway` ways are usually untagged, but in some cases they may carry tags and represent a distinct feature themselves:

`highway = crossing`	Road crossing for pedestrians.
`railway = crossing`	Railway crossing for pedestrians.
`railway = level_crossing`	Level crossing between road and railway.
`barrier`	A barrier blocking traffic. Values distinguish between types such as bollards or gates.
`traffic_calming`	An obstruction to slow down traffic. Values distinguish between types such as islands or speed bumps.

Lanes

`lanes`	Number of full-width lanes for motorized traffic. Useful for a better estimate of the road's width, and for drawing lane divider lines.
`sidewalk = left/right/both`	Indicates the presence of one or two sidewalks along the road.
`cycleway = lane`	A cycle lane is present. It is possible to define the location as e.g. `cycleway:left = lane`.
`cycleway = track`	A cycleway is separated from the road's carriageway.

Tags not relevant for 3D rendering

For many use cases of OpenStreetMap data, street names (`name`) and numbers (`ref`), as well as various traffic rules, access limitations and turn restrictions are crucial. We can easily ignore them for 3D rendering, though. One exception from this is the legal height limit for vehicles using the road, `maxheight`, which can be used to estimate a minimum amount of free space above a way.

Possible future developments

Some tags which would be very valuable for 3D rendering are either not common enough yet or still controversially discussed in the OpenStreetMap community.

The `bridge` key does not allow to distinguish between two bridges next to each other and a single bridge carrying multiple ways. Therefore, it has been suggested to create relations with a `type = bridge` tag, containing all ways that are running over the same bridge as well as the outline of the bridge itself. The counterpart for tunnels would be `type = tunnel`. Alternatively, the bridge outline may be drawn as an area and tagged as `man_made = bridge`. When one of these approaches is established, it should be taken into account for high-quality rendering of bridge and tunnel structures.

For adding properties to individual lanes, Lanes[8] tagging is slowly becoming more widespread. This relatively new concept is based on value lists. For example, lane surfaces may be defined as `surface:lanes = asphalt|asphalt|cobblestone`.

Furthermore, possible future trends include individually mapped traffic signs using nodes with the `traffic_sign` key, and mapping the exact extent of even an irregularly shaped road as an area with `area:highway` key in addition to the `highway` way. That information could likewise be incorporated into 3D rendering. It remains to be seen if the OpenStreetMap community adopts these ideas.

2.3.3. Water

Many of the the conventions for roads and railways – such as bridges, tunnels, and sharing nodes to indicate connections – apply to the water network as well. One important difference, however, is the convention that the way's direction is chosen to match flow direction. As flow direction is related to incline, this is an interesting property for 3D scenes. Some other concepts are also unique to waterways:

`waterway`	Required key to mark a way as a waterway. Values distinguish between types.
`waterway = riverbank`	An area covering the full extent of a waterway. This is drawn in addition to a way, which should be filtered from rendering if a riverbank area is also present.
`natural = water`	Required tag to mark an area as a body of water. Not used for oceans.
`natural = coastline`	Separation between land and sea. Water is on the right side of these ways.

[8] `http://wiki.openstreetmap.org/wiki/Lanes`

2.3.4. Miscellaneous objects

An almost innumerable variety of other features exist in the OpenStreetMap database. Most of them can be processed in a very similar manner: By applying a texture to an area or placing a simple model at the position of a node. Therefore, this list only includes those tags which are particularly prominent, common or have unusual properties for 3D rendering.

`natural = tree`	A node for a single tree.

`landuse = forest`
`natural = wood`
	Forest areas, managed or natural. Randomly distributed trees produce a decent visualization, although much more complicated forest rendering techniques do exist.

`barrier = wall/fence/hedge`	The most common linear barriers. These are used on ways rather than nodes.

`natural = cliff`
`barrier = retaining_wall`
	A way representing a cliff or wall with terrain elevation difference, which is often not reflected in the coarse elevation raster data. The top is on the left of the way.

2.4. Generation of meshes for 3D world objects

Distinct real-world features which rise above the ground are usually represented with meshes composed of primitives (such as triangles, and possibly more complex faces) which graphics hardware can render efficiently. For the conversion of buildings and other features to graphics primitives and rendering, existing code from OSM2World[9] – an open source community project started by the author of this thesis – can be used, as it already supports many of the tags listed in 2.2. However, for the purpose of integration into the terrain surface (see chapter 4), it was necessary to modify the data structures representing these features.

2.4.1. Ground meshes

OpenStreetMap contains polygons with attributes describing the ground surface texture. These are also converted to primitives for rendering. Where parts of the polygons are overlapped by linear features and dominant area features on the ground (such as roads and water bodies), it is necessary to subtract the area covered by these features first. The remaining surface is then triangulated.

[9] `http://osm2world.org`

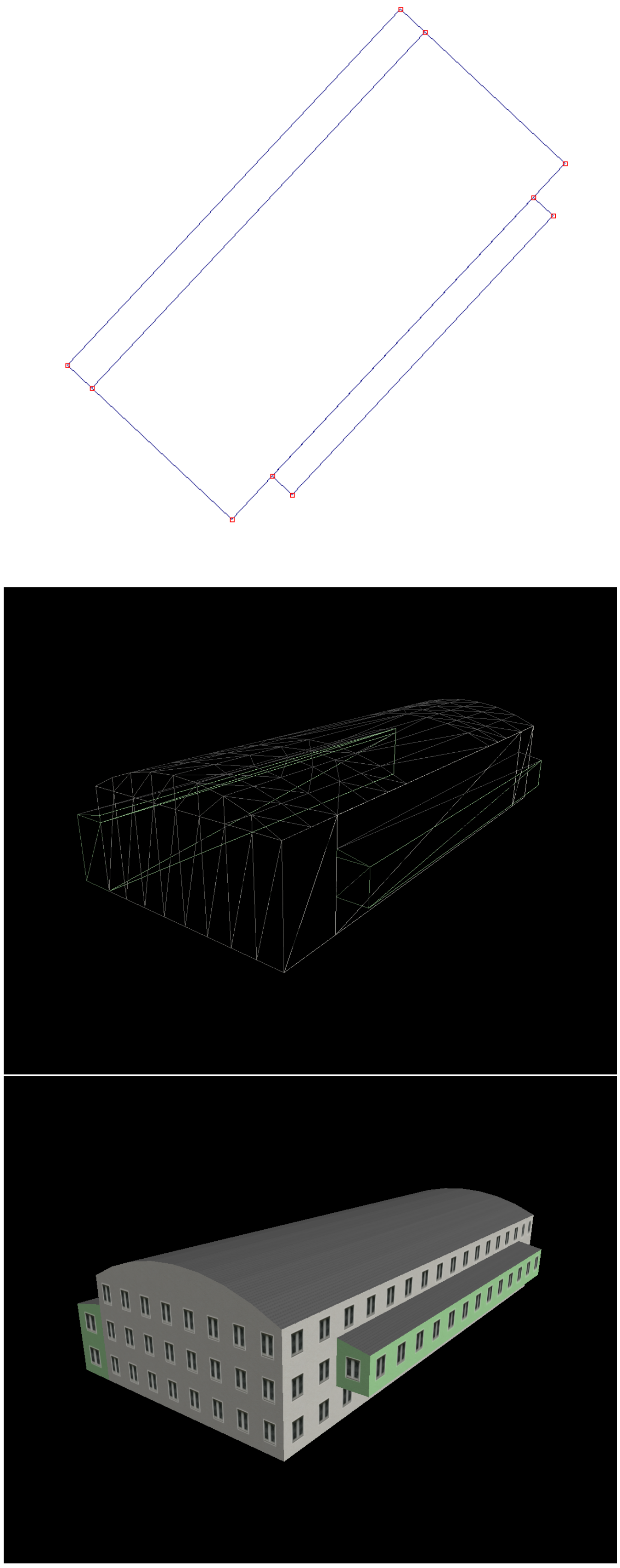


Figure 2.5.: Building mesh in OSM2World. Top to bottom: raw OpenStreetMap ways, wireframe mesh, textured model

For subtraction, we use the JTS Topology Suite[10] library. This library also implements triangulation. It turned out less suitable for our triangulation requirements, however, because it produces Conforming Delaunay Triangulations and has to insert additional points into polygon outlines to achieve this goal. As the approach used for finalizing the 3D scene in chapter 4 only works well with points known in advance, these additional points would not be properly connected to neighboring features. Therefore, poly2tri[11] was chosen for ground surface triangulation instead.

To avoid unnecessary special case handling, empty terrain – i.e. terrain with no explicitly mapped landcover – is treated the same as other ground areas. This is achieved by covering the entire data boundary with a grid of rectangular patches of ground, and subtracting other features, including other ground areas, from those initial polygons as described above.

The default terrain texture is an externally defined constant in this thesis, but may offer room for future improvement. It could, for example, vary depending on the geography and climate of the region in question. For this purpose, it may even be an option to pick the default terrain texture for a region based on low-resolution aerial imagery.

2.4.2. Coastlines

As hinted at above, seas are not mapped as areas in OpenStreetMap. Instead, the coastline is made up of a large number of ways with a `natural = coastline` tag. The ways share end nodes, but are otherwise independent. Evaluating coastlines is only reliably possible because of the convention that the water is always on the right side of the way.

Preferably, we want to represent parts of the sea as water areas, again to avoid special cases. This requires preprocessing of the ways into closed rings, which is similar to multipolygon preprocessing. As any OpenStreetMap data extract covering less than the entire planet will tend to contain incomplete coastline rings, however, we needed to develop a somewhat different algorithm for this task.

[10] `http://tsusiatsoftware.net/jts/main.html`
[11] `http://code.google.com/p/poly2tri/`

3. Terrain approximation

A next step towards the creation of a 3D scene is the approximation of terrain elevations from a sparse set of measurements, such as those provided by SRTM data (see 1.3.2). This can be performed independently from the preprocessing of OpenStreetMap data described in chapter 2.

3.1. Basic definitions

Our goal is the approximation of terrain elevation in three-dimensional space $\mathbb{R}^3$. The x and z values of a point $(x, y, z) \in \mathbb{R}^3$ shall represent the point's location on a plane (the result of transforming its latitude and longitude using a suitable map projection). y shall represent the point's elevation.

We make several assumptions about the terrain:

- The x and z values are bounded by a rectangle, i.e. there exist intervals $X, Z \subset \mathbb{R}$ such that $(x, z) \in X \times Z$ holds for all points (x, y, z) of the terrain.
- For each (x, z), the terrain has exactly one elevation value, so the terrain can be modelled as a function $X \times Z \to R$.

While the first assumption is easily met in practical applications by working with data for a bounded region, the second means that some shapes found in real-world terrains cannot be directly represented. However, reconstructing such shapes from SRTM data is not feasible and the assumption greatly simplifies creating and working with the terrain.

As mentioned before, we use elevation values from SRTM data as the input for our algorithms. Although these values are arranged in a grid, the algorithms presented here should not rely on this structure for correctness. This will allow taking into account additional sources of elevation, including elevations crowdsourced by the OpenStreetMap project, in the future.

Not relying on a grid also lessens the impact of small SRTM voids. Larger voids are most commonly found in areas with low density of human construction, particularly deserts and mountaineous regions, and sophisticated void filling is not the focus of this thesis.

Thus, we can simply define our input as a number of sites with known elevation – a finite set of points in $X \times \mathbb{R} \times Z$.

3.2. Overview of approximation techniques

Although we could only experiment with a limited number of solutions, many possible approaches exist. Therefore, we will start with a short overview of some available algorithms. Afterwards, we will look in detail at (and implement) two different solutions: least squares approximation and natural neighbor interpolation.

Linear interpolation within a triangulation

A common and straightforward approach, described e.g. in [dB00], starts with a triangulation of the set of points with known elevation. Any triangulation algorithm for point sets can be used, but the Delaunay Triangulation (which we describe in a different context in section 3.4.1) often produces good results. Within each triangle, terrain elevation is linearly interpolated based only on the known elevation values of the triangle's vertices. Thus, this algorithm amounts to a piecewise linear interpolation.

This simple solution appears attractive at first, considering that rendering requires triangulation of the ground in some manner anyway. However, when we recall the low resolution of SRTM data, it is clear that the resulting terrain would appear too coarse for any close-up visualization (see figure 3.3). For that reason, we prefer a detailled triangulation with a lot more vertices than there are SRTM data points, like the one previously described in section 2.4.1.

It deserves to be pointed out, however, that we ultimately render our terrain using triangle primitives. As these triangles are perfectly flat, this algorithm will implicitly come into play at a rendering stage – just with a much finer triangle network.

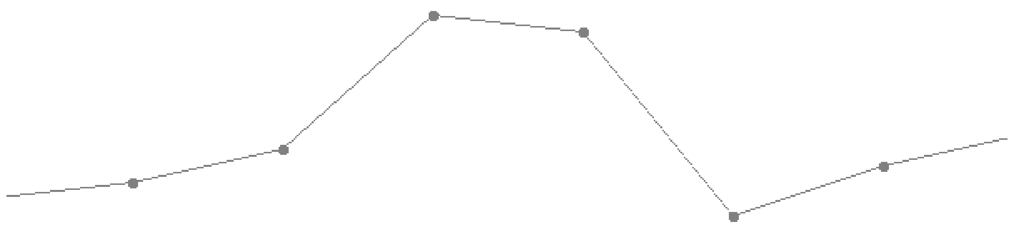


Figure 3.1.: Two-dimensional cross section through a terrain created with linear interpolation. Dots mark points with known elevation, the input for the approximation.

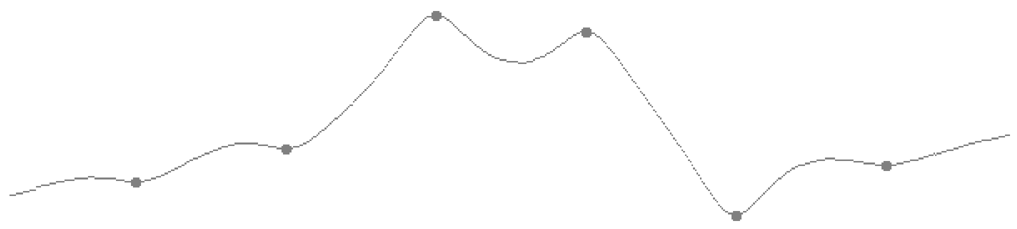


Figure 3.2.: Two-dimensional cross section through a terrain created with inverse distance weighting. Dots mark points with known elevation, the input for the approximation.

Inverse distance weighting

Inverse distance weighting (IDW) is a very common approach in Geographical Information Systems, often presented in introductory material such as [Lon05]. It is based on the idea that elevation values of points close to each other tend to be similar. Therefore, terrain elevation is approximated as a weighted average of known elevations. The weights are based on the inverse distance to the point whose elevation is approximated, most frequently the inverse squares of distances are used.

Usually, not all points with a known elevation will be taken into account, but only those within a neighborhood defined, for example, by a cutoff radius or a maximum number of points.

When used to approximate terrain, IDW often produces very visible minima and maxima at the points with known elevation, appearing as peaks and pits (see figures 3.2 and 3.4). This effect is undesirable in rendering. We will, however, use a similar calculation in section 3.3.2 as a component of another algorithm's implementation. Like IDW, we will use inverse squares of distances as weights for a weighted average within a neighborhood, but instead of averaging the known elevation values themselves, we will average function values from multiple local approximations.

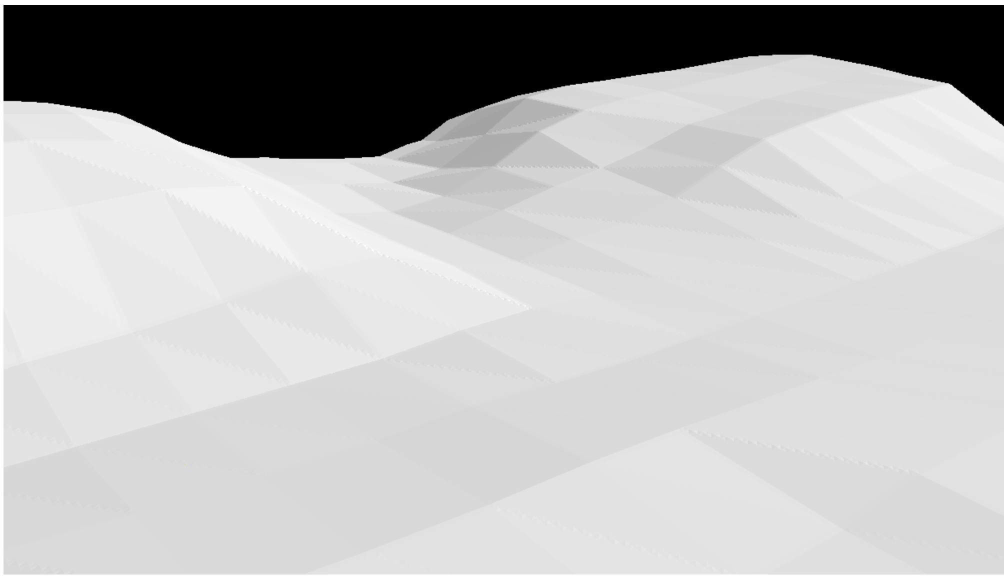


Figure 3.3.: Terrain approximation using flat triangles from a relatively close viewing distance. Compare with figure 3.12

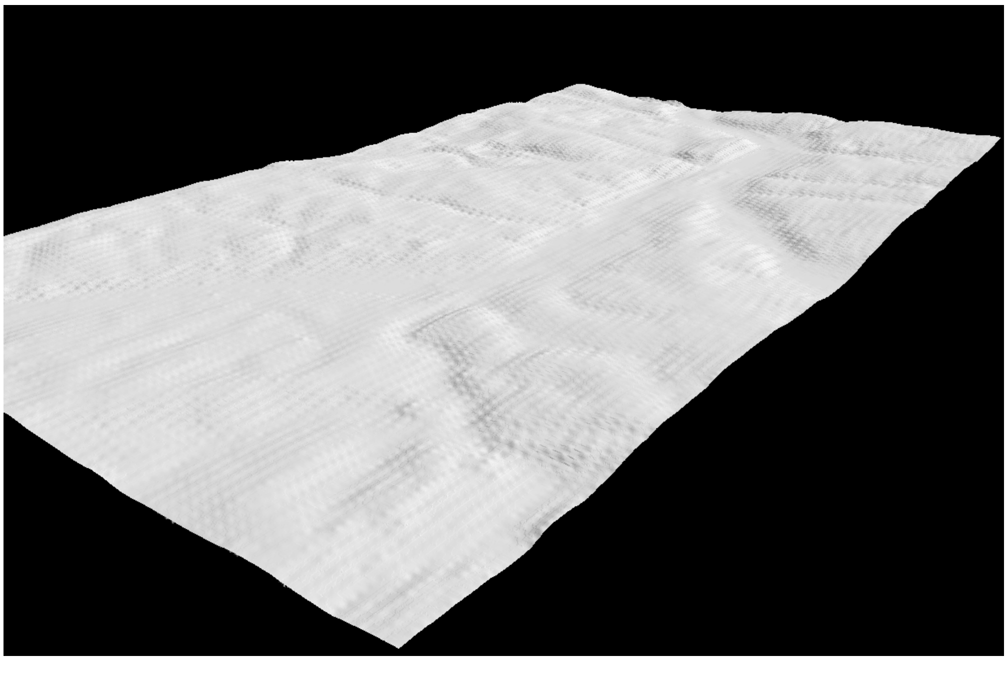


Figure 3.4.: Terrain approximation using IDW. Visible extrema appear at the points with known elevation. Compare with figure 3.10.

3.3. Approximation using least squares

3.3.1. Least squares

Least squares approximation is a popular method for finding a curve approximating a set of data points. As least squares approximation is a generic technique applicable to different classes of functions, we need to first choose a class of functions used for the approximation. This thesis uses quadratic polynomials

$$f((x,z)) = a_0 + a_1 \cdot x + a_2 \cdot z + a_3 \cdot x^2 + a_4 \cdot xz + a_5 \cdot z^2$$

of two variables x and z, representing the coordinates of a point in the plane per the conventions described above. These polynomials are easily evaluated and can be intuitively expected to describe small terrain patches sufficiently well.

Given $k+1 > 6$ sites of known elevation, $p_i = (x_i, y_i, z_i)$ with $i = 0, 1, \ldots k$, we can find values for a_0 through a_5 which produce small distances $\|y_i - f((x_i, z_i))\|$ between the polynomial and the known elevation values. To do so, we need to solve the following overdetermined linear system:

$$\begin{array}{cccccccc}
a_0 + & a_1 \cdot x_0 + & a_2 \cdot z_0 + & a_3 \cdot x_0^2 + & a_4 \cdot x_0 z_0 + & a_5 \cdot z_0^2 & = & y_0 \\
 & \vdots & & & \vdots & & & \vdots \\
a_0 + & a_1 \cdot x_k + & a_2 \cdot z_k + & a_3 \cdot x_k^2 + & a_4 \cdot x_k z_k + & a_5 \cdot z_k^2 & = & y_k
\end{array}$$

Equivalently, this can be written as a matrix equation:

$$\begin{bmatrix} 1 & x_0 & z_0 & x_0^2 & x_0 z_0 & z_0^2 \\ & \vdots & & & \vdots & \\ 1 & x_k & z_k & x_k^2 & x_k z_k & z_k^2 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ \vdots \\ a_5 \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_k \end{bmatrix}$$

The $(a_0, \ldots, a_5)$ obtained as the solution are optimal, in the sense that the sum of squared distances

$$\sum_{i=0}^{k} \|y_i - f((x_i, z_i))\|^2$$

is minimal (hence “least squares”). This is explained, along with a short description of the technique, in [Far02].

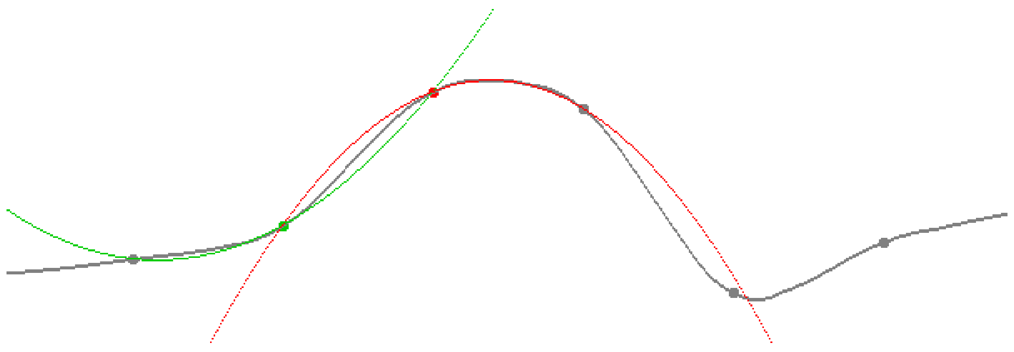


Figure 3.5.: Two-dimensional cross section through a terrain (gray) created with least squares approximation. Dots mark sites with known elevation, the input for the approximation. Additionally, the approximated polynomials at two of sites are shown.

With our prototype, the matrix equations are delegated to the open source Apache Commons Math library[1] which implements a QR decomposition solver.

3.3.2. Implementation

Describing the terrain of a large region with a simple equation is not realistically possible. Therefore, we only use least squares to determine local approximations for small terrain patches. More precisely, we approximate a curve at each site of known elevation, based on the elevation value and position of that site itself and the k nearest sites. The implementation uses $k = 8$, which usually (in the absence of voids or other anomalies) means that the approximation uses the site itself and its direct neighbors. “Nearest” sites are calculated using only x and z dimensions.

To determine the elevation of the approximated terrain at any two-dimensional point p, the set $S_{(p,l)}$ of the nearest l sites (for some predefined value l) is determined. The values $f_s(p)$ of the functions associated with each $s \in S_{(p,l)}$ are calculated and combined using a weighted average, with the inverse distance to the point as the weight:

$$ele(p) = \frac{\sum_{s \in S_{(p,l)}} w_{(s,p)} \cdot f_s(p)}{\sum_{s \in S_{(p,l)}} w_{(s,p)}}$$

where the weight $w_{(s,p)}$ is defined as with inverse distance weighting (section 3.2), using a sensible cutoff distance. The function ele finally represents our terrain.

Finding sites close to a given point reasonably quickly is a necessary component for this solution. With our implementation, all sites are inserted into a regular grid of

[1] `http://commons.apache.org/proper/commons-math/`

rectangular cells. To find the sites closest to a point, sites are retrieved from the cell containing the point, and candidates for the result are managed in a priority queue ordered by proximity. This priority queue contains the best candidates identified so far (up to the number of sites we want in the result). The search is extended to cells at an increasingly larger distance around the one containing the point until we have enough results and even the candidate with the largest distance in the priority queue is closer to the given point than any point from the still unchecked cells could possibly be.

3.3.3. Overall complexity

Solving the matrix equation for one site and approximating the polynomial are both possible in constant time, thus the time complexity of doing so for n points with either known or unknown elevation is $\mathcal{O}(n)$, i.e. this part of the algorithm alone would scale linearly.

However, the algorithm's complexity is dominated by the necessary determination of the k nearest sites for each site and point. As we allow arbitrarily distributed sites as our input, this task is equivalent to a generalization of the all-nearest-neighbors problem, which is solved by algorithms such as one described by [Vai89] running in $\mathcal{O}(k \cdot n \cdot \log n)$ time – which is also the lower bound.

Our implementation described above has a higher worst case time complexity than that, though. If most or even all sites end up in the same cell of our rectangular grid, we end up calculating pairwise distances of all n coordinate pairs. This results in a complexity of $\mathcal{O}(n^2)$. (Inserting into and deleting from the priority queue can be neglected for this consideration because it has a fixed maximum length.) However, we expect that sites tend to be more evenly distributed in practice.

3.4. Natural neighbor interpolation

Natural neighbor interpolation is an interpolation technique where the elevation of a point on the terrain surface is determined by the elevation of nearby sites, the "natural neighbors" of the point. It was described in 1981 by [Sib81]. The implementation used in this thesis is based on the approach suggested by [LG04].

At a site, the terrain's elevation will always be equal to the site's known elevation value. This property is desirable especially if crowdsourced elevation data is created by contributors who may expect their input to have a direct effect.

3.4.1. Voronoi diagram

Voronoi diagrams are the foundation for the definition of natural neighbors. Such a diagram can be constructed for any set of sites S from the infinite set of all points P in the plane. For each $s \in S$, its *Voronoi cell* $C_{s,S}$ is defined as the set of all points that are at least as close to s as to any other point, i.e. $C_{s,S} = \{p \in P \mid \forall q \in P : \mathrm{dist}(s,p) \leq \mathrm{dist}(s,q)\}$. The common boundary of two Voronoi cells (if it contains more than one point) is called a *Voronoi edge*, and the endpoints of Voronoi edges are called *Voronoi vertices*. Voronoi edges and vertices together form the Voronoi diagram.

While other variants and higher-dimensional Voronoi diagrams have been studied, only two-dimensional diagrams based on Euclidean distance are used in this thesis.

Basic properties

Voronoi cells must be convex as an intersection of half-spaces. However, the outermost Voronoi cells have an infinite area. Often it is more convenient to introduce a bounding curve at some distance around all sites and vertices of the Voronoi diagram. Adding that bound and removing the (infinite) parts of the Voronoi edges outside the bound results in a planar and connected embedded graph, the *finite* Voronoi diagram.

According to the Euler formula, the relationship between the numbers of vertices (v), edges (e) and faces (f) in a planar connected graph is given by

$$v - e + f = 2$$

This equation applies to finite Voronoi diagrams. As there is a Voronoi cell for each site in S, the number of faces is equal to the cardinality of S plus the single face outside the bound, i.e. $f = \mathrm{card}(S) + 1$. Therefore, we know that

$$e = v + \mathrm{card}(S) - 1$$

Furthermore, each vertex in a Voronoi diagram is connected to at least three edges, so we know that $e \geq 3v/2$, or $v \leq 2e/3$. Substituting v in the previous equation, we arrive at

$$e \leq 2e/3 + \mathrm{card}(S) - 1$$

which can be transformed to

$$e \leq 3\,\mathrm{card}(S) - 3$$

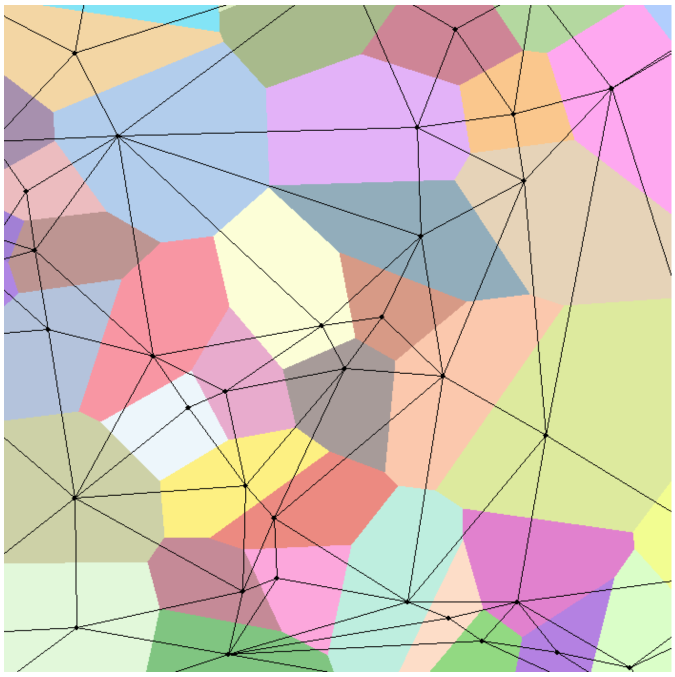


Figure 3.6.: Section of a Voronoi diagram with Delaunay triangulation.

Each of these edges separates two faces. Thus, the average number of edges around each face is $2 \cdot (3\ \mathrm{card}(S) - 3) / (\mathrm{card}(S) + 1) < 6$ – there are, on average, less than 6 edges in a Voronoi cell's boundary. This well-known argument, described e.g. in [AK00], will be useful for discussing algorithmic complexity later.

Delaunay triangulation

From each Voronoi diagram, it is possible to build one Delaunay triangulation (and vice versa) – a partitioning of the space into triangles where each triangle's circumcircle contains no sites except the triangle's three vertices. This will make it possible for us to calculate a Delaunay triangulation first, and use it to fully or partially derive the Voronoi cells as needed (section 3.4.3). To extract the diagram from the triangulation, a vertex of a Voronoi cell is placed at the centre of each triangle's circumcircle, and a Voronoi cell edge is created for each Delaunay triangle edge.

3.4.2. Natural neighbors

Inserting an additional site into an existing Voronoi diagram will decrease the area of nearby sites' Voronoi cells. This observation leads to the concept of natural neighbors: The natural neighbors of a point $p \in P \setminus S$ are those sites in S whose cells would become

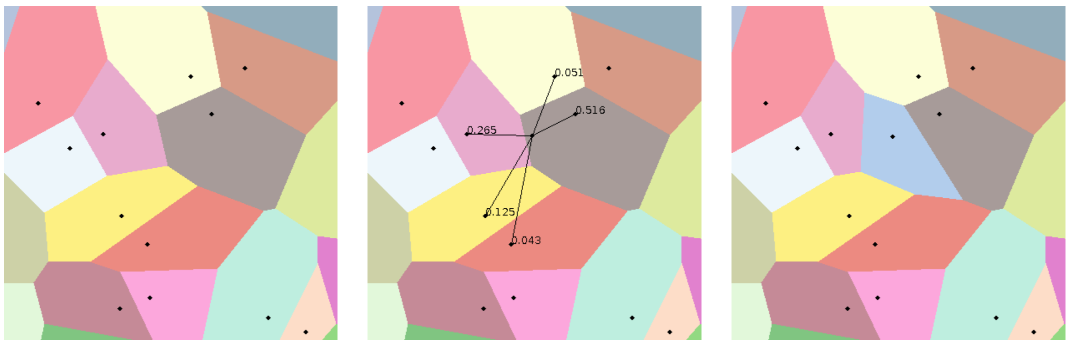


Figure 3.7.: Left to right: Voronoi cells, relative weights of a point's natural neighbors, Voronoi cells after insertion of that point.

smaller by the insertion of p into the Voronoi diagram of the sites S.

Furthermore, this concept allows a relative weighting of a point's neighbors. To achieve this, calculate the area loss of a site $s \in S$:

$$\mathrm{loss}_s = \mathrm{area}(C_{s,S}) - \mathrm{area}(C_{s,S\cup\{p\}})$$

The relative strength of the neighborship of s to p is then determined as its share of the sum of area lost by all existing sites in the diagram, i.e. the quotient

$$\mathrm{loss}_s / \sum_{t \in S} \mathrm{loss}_t$$

.

The sum of these weights will be 1. Thus, an interpolation between values – such as elevation – associated with the sites can be achieved by assigning to each point a weighted average of its neighbors' values.

3.4.3. Construction of the Voronoi diagram

Natural neighbor interpolation as defined above requires the sites' Voronoi diagram, so we need to look into the construction of Voronoi diagrams. Luckily, this problem has been extensively studied in computational geometry. We will briefly summarize some relevant knowledge and algorithms, primarily based on a detailled overview available in [AK00] and the algorithms suggested by [LG04].

Lower bound for complexity

A lower bound of $\Omega(n \log n)$ time for constructing the Voronoi diagram or Delaunay triangulation of a set of n points can be proven via reduction: If n real numbers

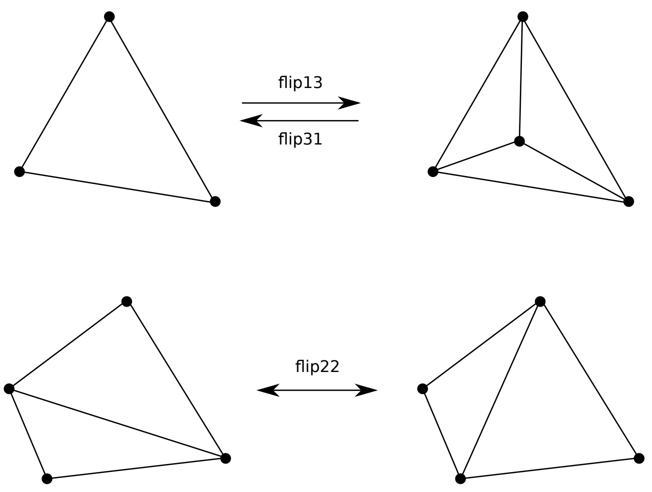


Figure 3.8.: The flips used in a two-dimensional Delaunay triangulation.

$r_1, \ldots, r_n \in \mathbb{R}$ are given, sorting them is known to be to require $\Theta(n \log n)$ time. These numbers can be used to create a set of n points $S = \{(r_i, r_i^2) | i \in [1, n]\}$ in the plane. Creating S takes no more than linear time. By constructing the Voronoi diagram of S, we are able to obtain the set's convex hull. Walking around the convex hull of S, we will encounter all points from S sorted by their first coordinate value, r_i, giving us the correct ordering of the numbers $r_1, \ldots, r_n$. Thus, creating a Voronoi diagram of n points must be at least as complex as sorting n real numbers.

Incremental construction of the Delaunay triangulation

As mentioned before, we first construct the Delaunay triangulation instead of directly calculating Voronoi diagrams. One of many possibilities to construct a Delaunay triangulation is incremental insertion. The main benefit choosing this approach is the ability to use the same algorithm for both the initial insertion of the sites with known elevation and the subsequent temporary insertion of probe points which will be necessary to determine natural neighbors.

We perform incremental insertion using so-called flips, i.e. local modifications to the Delaunay triangulation as shown in figure 3.8. To insert a point into the Delaunay triangulation, we first find the triangle containing it. The triangle is then split into three separate triangles at the new point, an operation called a flip13. This modification of the triangulation can violate the property that no circumcircle in a Delaunay triangle may contain more than three points. We restore it by performing flip22 operations on affected neighbors of previously modified triangles, wherein a quadrilateral composed of two triangles is switched to the unique alternative triangulation.

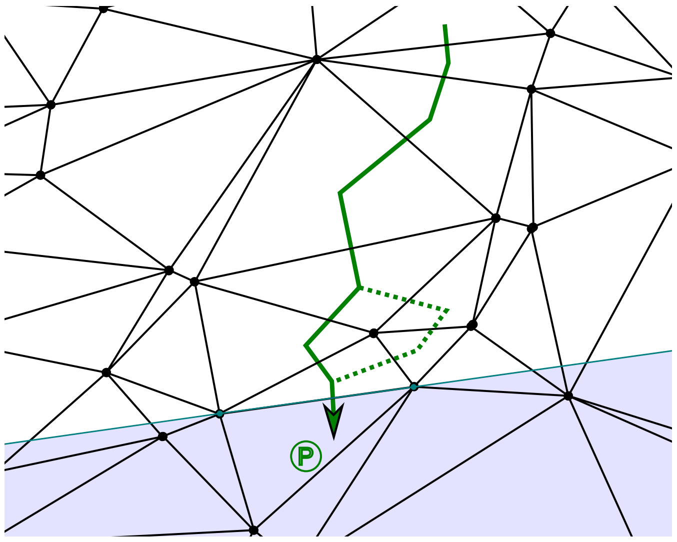


Figure 3.9.: Visibility walk in a Delaunay triangulation towards the triangle containing a point P, along with another valid visibility walk. The last successful half-space test is illustrated with a shaded background.

Both flip13 and flip22 can be inverted, which is a requirement for using flips in the temporary insertion of probe points.

Even in the worst case, we can state that incremental insertion will build the Delaunay triangulation for a set of n points in $\mathcal{O}(n^2)$ time: During the insertion of each of the n points, we first have to find the triangle containing it, and $\mathcal{O}(n)$ time is sufficient for simply testing all triangles. Afterwards, we have to flip pre-existing triangles affected by the insertion, of which there cannot be more than n.

It is not actually necessary to flip all triangles, though. The triangles that participate in flips are connected to the newly inserted point, and their number is limited by the degree of that point in the new triangulation. Thus, we apply the knowledge that the number of edges of the corresponding Voronoi cell, and therefore the point's degree, is on average no higher than 6 (see section 3.4.1). Depending on the order in which the points are inserted, it may still be higher – but with a randomized insertion order, we can expect $\mathcal{O}(1)$ time for a single insertion.

Additionally, the search for the triangles affected by the insertion could be improved to $\mathcal{O}(\log n)$ expected time, which would yield $\mathcal{O}(n \log n)$ expected time for incremental insertion. However, our implementation does not use such an approach, but rather a technique known as visibility walks.

3.4.4. Visibility walks

For fast incremental insertion into a Delaunay triangulation, it is crucial to quickly find the existing triangle containing the point to be inserted. As with the least squares approach, we first used a speedup grid for this purpose. However, this does not work well because triangles span a very large number of grid cells, especially at the beginning of the construction of the triangulation. The resulting excessive number of updates to the grid cells slows down the algorithm.

We achieved better experimental results using *visibility walks* through the triangulation as described by [DPT01]. Given a target point within the triangulation, the algorithm works as follows:

- Choose any triangle as the current triangle
- While the current triangle does not contain the target point:
  - For all three edges of the triangle:
    - Extend the edge to an infinite line
    - Test whether the target point lies in the half-space beyond that line
    - If this is the case: Proceed with the neighbor triangle sharing that edge with the current triangle
  - If none of the neighbors was chosen, the current triangle contains the point.

See figure 3.9 for an illustration. In Delaunay triangulations, a visibility walk is guaranteed to terminate at the target triangle.

3.4.5. Implementation

As explained before, we implemented natural neighbor interpolation based on the Delaunay triangulation. Each triangle is represented as an object, with references to the three sites at its corners and to its three neighbors. There is no global collection of triangles. Instead, triangles are accessed via walking for the insertion of points. Iterating over all triangles is not necessary for the interpolation algorithm itself; where doing so is desired e.g. for debugging, the triangles are enumerated using a depth-first search of the graph structure defined by the neighborship links.

To avoid burdening the later calculations with the special cases occuring at the border of the Delaunay triangulation resp. Voronoi diagram, the triangulation is initialized with two triangles which completely enclose all sites and points. Doing so is easily possible because of our initial assumption that there is a known bound for x and z coordinates.

Subsequently, all sites with known elevation are permanently inserted into the triangulation. At that point, the data structure is ready to be used for interpolation: Each point with unknown elevation is temporarily inserted into the Delaunay triangulation. The necessary flip operations are stored on a stack. After insertion, the Voronoi cell areas of the point's neighbors are calculated. The insertion is then undone by performing the inverse operation of the stored flips in reverse order, and the neighbors' Voronoi cell areas are calculated again. The area differences are then used to determine the weight of each neighbor and to ultimately calculate the interpolated elevation of the point.

3.5. Results

Both implemented algorithms produce visually plausible approximated terrains, see figure 3.10. The average difference between the results for a test area (SRTM tile N48E013) is 0.76 m. Figure 3.11 shows how these differences are distributed.

A closer look reveals some differences, though: Natural neighbor interpolation sticks exactly to the sites with known elevation; least squares approximation offers a very smooth surface (figure 3.12). But despite the algorithms' differences, the numerical results suggest that these will not be particularly prominent in most places once various concealing 3D models have been placed on top of the terrain. Therefore, performance is an important motivation for choosing one algorithm over the other.

In our tests, the algorithm based on least squares approximation creates a terrain with 30 m sample distance for a square degree within 02:45 minutes. Our natural neighbor interpolation needs 05:50 minutes for the same task.[2]

It should be noted that our prototype implementations are by no means perfect. If the ability to handle arbitrarily distributed measurements was discarded or restricted, exploiting the grid structure of SRTM data could yield significant performance gains; in particular, doing so would make it faster to find the neighbors of each data point.

Other possible optimizations depend on the use case. If the algorithms were used to refresh 3D output, e.g. on a server, in regular intervals, then SRTM data would not have to be refreshed every time. Therefore, storing intermediary results of the algorithms – the Voronoi diagram or the polynomials respectively – and loading them instead of the unprocessed SRTM data could become an option. Finally, parallel execution would also be possible for major parts of both algorithms.

[2]on an Intel Core 2 Duo CPU @ 2.66 GHz with 6 GB primary memory, running the OpenJDK 64-Bit Server VM 1.7.0

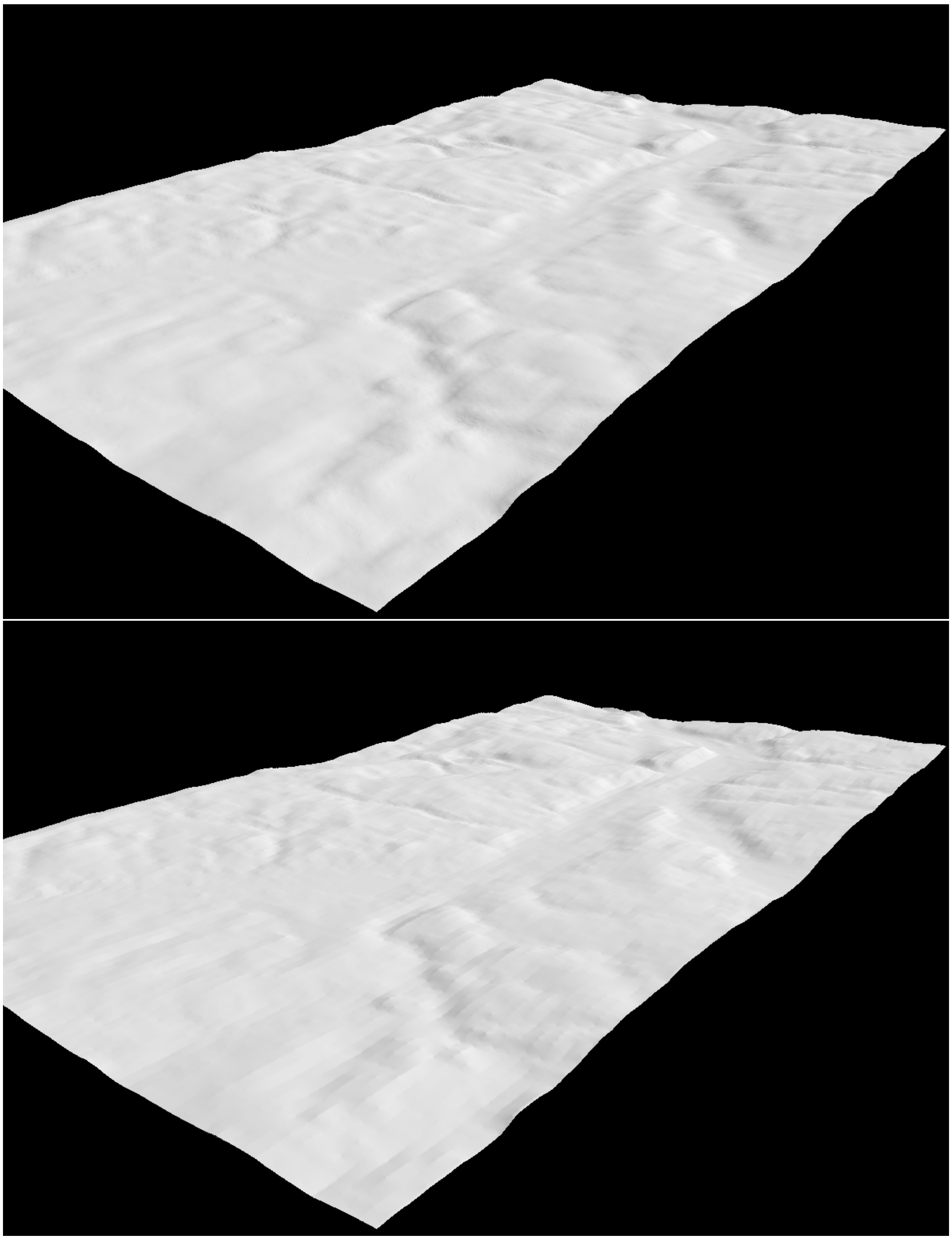


Figure 3.10.: Approximated terrain for an area around Passau, sampled in a 6 m grid. Results shown for least squares approximation (top) and natural neighbor interpolation (bottom).

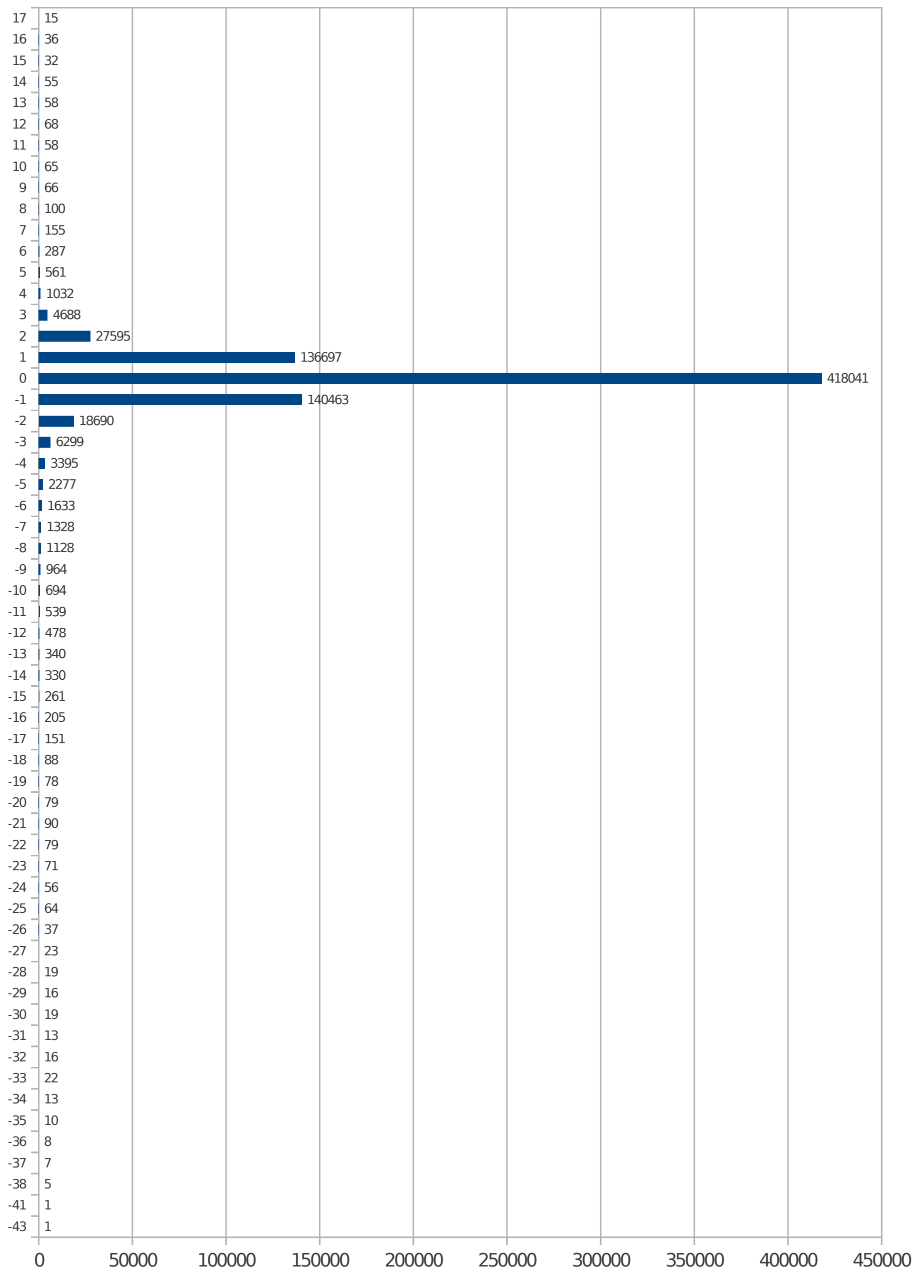


Figure 3.11.: Distribution of elevation differences between least squares approximation and natural neighbor interpolation, rounded to integers. Values in meters. Positive values indicate that the terrain sample is higher with least squares approximation.

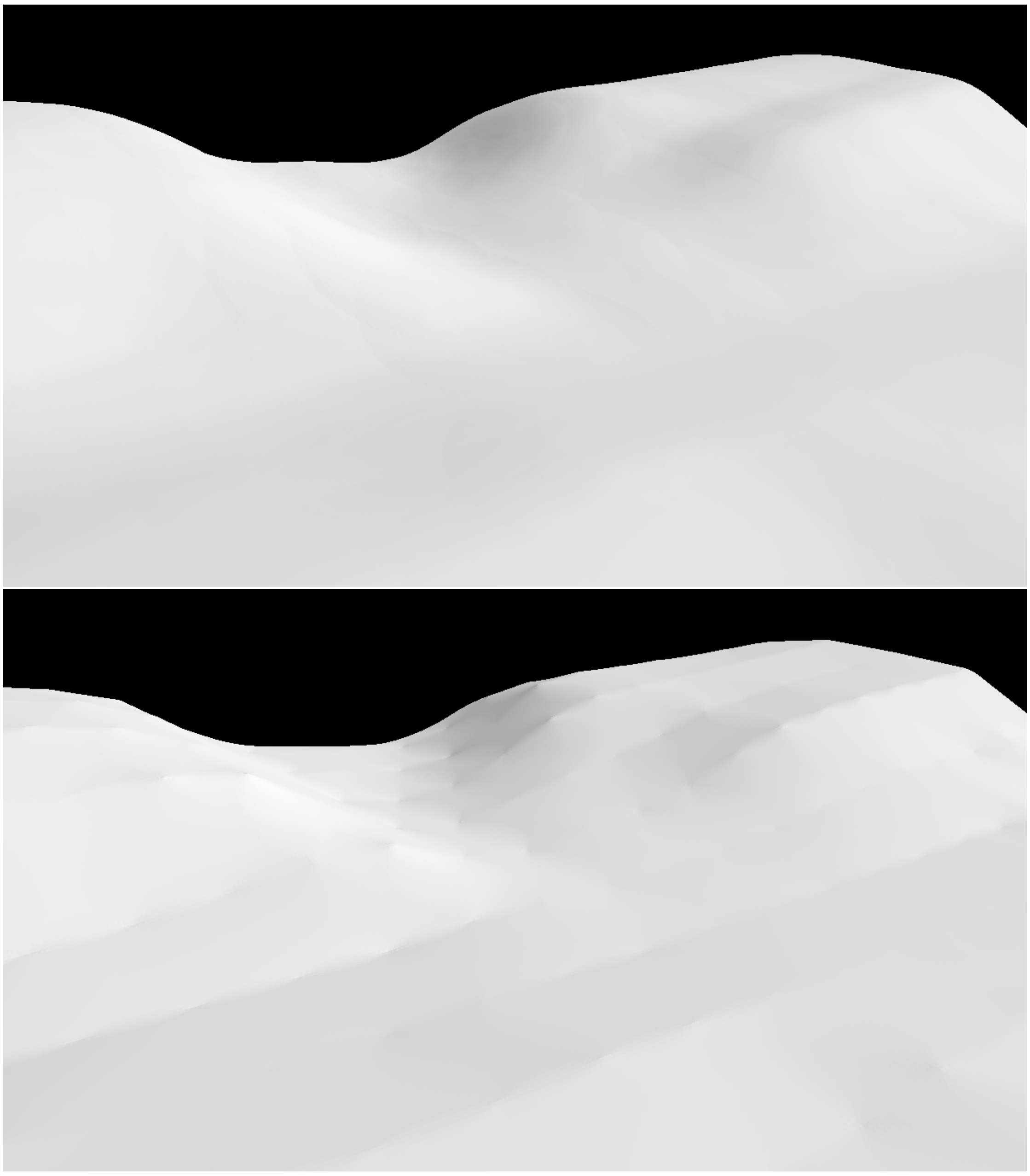


Figure 3.12.: Close look at terrain approximation results, sampled in a high-resolution 3 m grid. Results shown for least squares approximation (top) and natural neighbor interpolation (bottom).

4. Integration of terrain and models

After converting the OpenStreetMap data to 3D models (chapter 2) and approximating a terrain surface (chapter 3), we need to combine the results of the two processes into a three-dimensional scene. During this final step, we will also improve elevations beyond the original terrain elevation data by evaluating more information from OpenStreetMap.

4.1. Defining connectors

Our first goal is to make sure that the models are properly connected with each other and with the terrain. To achieve this, we introduce *connector* nodes. Each model may define one or several such connectors.

A connector initially has only two-dimensional coordinates, and enough information to decide whether two connectors with identical two-dimensional coordinates defined by different objects should be joined with each other: a flag distinguishing between connectors on the ground and those above/below it, plus optionally a reference to an OpenStreetMap node. Connectors which are joined will end up with the same three-dimensional coordinates after the assignment of elevation values.

4.1.1. Connector examples

The following examples illustrate how different model types make use of connectors.

Node-based features

The most straightforward situation occurs with models derived from a single node, such as trees or street lamps. In this case, a single connector is placed at the base of the model. The connector is joined with the surface below – most commonly, this will be a terrain surface.

Connection with terrain is also the only variant currently implemented for these models, although support for placing them e.g. on top of bridges or roofs should be added in the future to properly represent these rarer cases.

Linear features with width

Roads, railways and other linear features with a horizontal width define connectors at the center and the left and right side of their end caps. They can thus be joined to other parts of the road network. In figure 4.1, this is illustrated with a junction area. For features on the ground, the connectors are also joined to the terrain.

Terrain surfaces

Terrain surfaces define a connector for each vertex in their outline polygons. Within the outline, we insert connectors wherever another object defines a connector with a flag indicating that it is on the ground. This ensures that these other objects are properly connected to the terrain surface.

To obtain reasonably smooth surfaces, we insert a grid pattern of additional connectors into the area.

The outline polygons and the connectors from the sources mentioned above are then used as the input for the surface area's triangulation. Therefore, each triangle's vertices will have corresponding connectors.

Tunnel entrances

Entrances into tunnels require special treatment because they create a hole in the terrain surface. We achieve this by creating a ring of connectors around the entrance. All these connectors are connected with the terrain, but only the lower row is connected to the road.

Buildings

Buildings define a connector for each of their outline nodes. They are connected to the terrain at the building's ground level. However, buildings in sloped terrains may be constructed in different styles: The ground may either be flattened to achieve a constant elevation, or it may remain sloped so some levels are partially underground. As it is usually not possible to distinguish these cases in OpenStreetMap, we have chosen to default to the latter style for the current implementation.

One feature of buildings which allows to extract additional elevation data, however, are entrances. They are not necessarily at ground level. Instead, we make that decision depending on the `highway` ways connected to them.

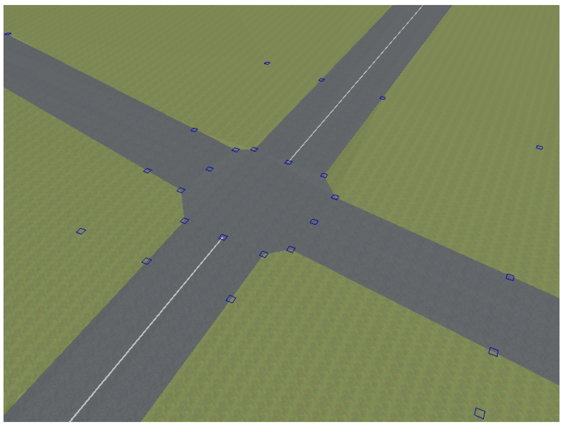


Figure 4.1.: Elevation connectors (blue) joining road sections with a road junction and the surrounding terrain surface areas.

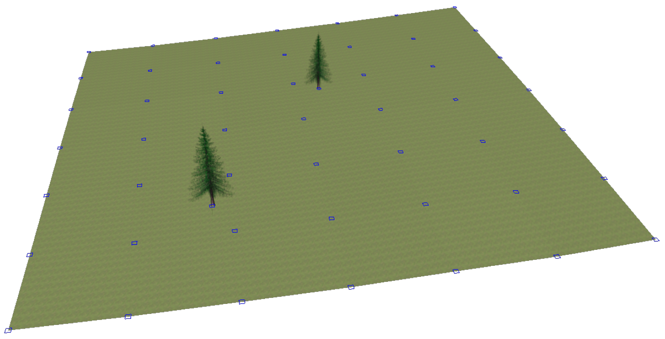


Figure 4.2.: Rectangular terrain surface patch. In addition to a regular grid of connectors, two connectors created to accommodate trees on the ground are visible.

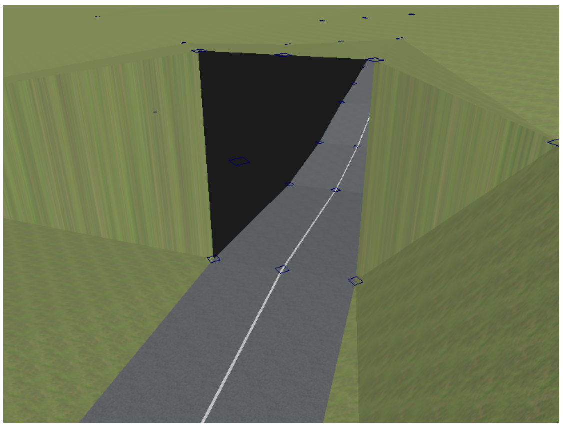


Figure 4.3.: Elevation connectors around a tunnel entrance.

4.1.2. Direct elevation assignment

By directly assigning the approximated terrain elevation from chapter 3 to each connector, we already obtain a result that looks decent from a distance. Basic goals of connectors have been achieved – for example, node-based features such as trees will always be connected to the ground, and roads are seamlessly inserted into the terrain surface.

However, a closer look reveals several remaining limitations. To name a few: Bridges and tunnels are still at ground level, steps and roads with an incline do not keep these properties after elevation assignment, and roads and waterways at steep slopes have unrealistic inclines orthogonal to their direction.

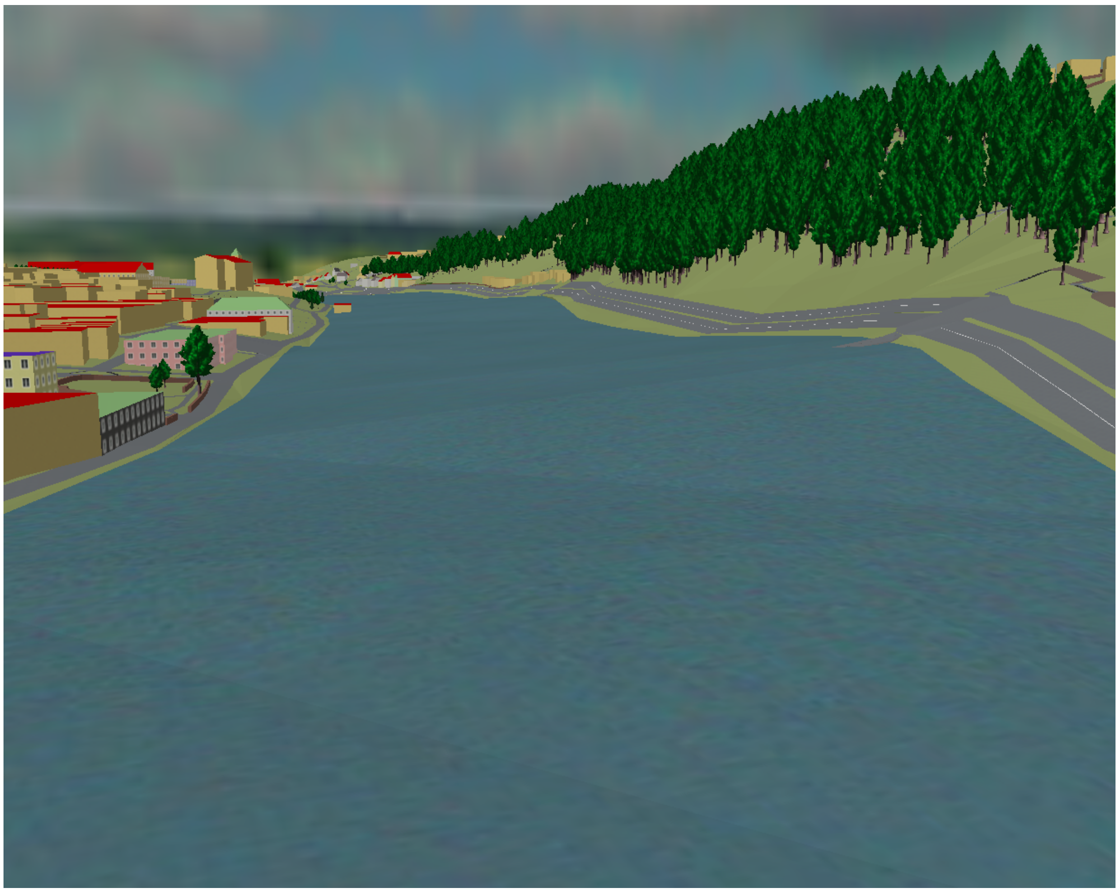


Figure 4.4.: Rendering based on direct assignment of approximated terrain elevation values, highlighting several of the flaws pointed out above. As expected, bridges follow the ground slope and are thus submerged in water.

4.2. Defining constraints

To further improve the resulting scene and address the observed limitations of direct elevation assignment, we introduce another concept, *constraints*. These are relationships between connectors' elevations, such as a minimum distance. A lot of information from OpenStreetMap can be expressed using such relationships. This section suggests a set of basic constraints and explains their potential uses.

Same elevation

It often makes sense to require that a group of connectors shares the same elevation.

With our implementation, this is used for the triples of connectors at each node of a road, railway or other linear feature with a width. While this is still just an approximation to reality, it is preferable to roads whose shape is entirely determined by the underlying terrain surface.

Another use case are outlines of water bodies. We also use it preliminarily for the outlines of waterways that are mapped as areas rather than just linear ways, even though this fails to take their natural incline into account.

Exact elevation difference

In some cases, we want to maintain an exact elevation difference between two connectors. For example, this is the case with tunnels if the height of the tunnel is known. Likewise, we can enforce an exact elevation difference between two entrances at different building levels.

Minimum/maximum elevation difference

Below bridges and above tunnels, we need to enforce a minimum distance to ground level features and the terrain surface itself. It is sensible to assume a default minimum distance, but we could improve it e.g. when the way below a bridge has a `maxheight` tag. Constraints of this type are also helpful to model cliffs and retaining walls.

Sometimes there will not be a single other connector to serve as the counterpart in a constraint. In these situations, we can require a distance between a connector and the line segment between two other connectors instead. This is applied e.g. to keep trees at a distance from a road through the tunnel below.

As a special case, minimum or maximum elevation differences can be used to model inclines. Unlike the other examples, the target value for the elevation difference needs to take the distance between two connectors into account when modeling an incline. Incline constraints are straightforward for ways with explicit `incline` tags, but we are also using a default maximum incline of 35% to prevent particularly extreme elevation differences. Waterways are prevented from flowing uphill.

4.3. Enforcing constraints

4.3.1. Linear programs

To enforce our constraints, we use the *linear program* model, which is more commonly used in fields such as microeconomics or operations research. This overview of the concept is based on [MS08].

Generally speaking, a linear program (LP) is an optimization problem, formulated as a number of linear constraints and a linear objective function. More precisely, an LP consists of

- n **variables** $x_1, \dots, x_n \in \mathbb{R}$, combined as a vector $x = (x_1, \dots, x_n)$
- m **constraints** $a_i \cdot x \bowtie_i b_i$, where $i \in 1..m$, $a_i \in \mathbb{R}^n$, $b_i \in \mathbb{R}$ and $\bowtie_i \in \{\leq, \geq, =\}$
- the **objective function** $f : \mathbb{R}^n \to \mathbb{R}$ with $f(x) = c_1 \cdot x_1 + \cdots + c_n \cdot x_n$

When solving a linear program, the goal is to find an x that maximizes the objective function as much as possible without violating the constraints.

Many variations are equivalent to the basic definition above. For example, minimizing instead of maximizing the objective function f is possible by multiplying each coefficient c_i by -1. That definition itself is a variant of a more restricted form, which allows only non-negative values for x_i.

LPs can be solved in polynomial time. Nevertheless, many solvers instead implement algorithms with up to exponential worst case execution time, but good real-world performance for typically encountered problems.

We will skip discussion and implementation of algorithms for solving linear programs and rely on existing implementations instead. After all, one of the primary advantages of expressing a problem using LP is the availability of a wide range of solver libraries. In our software, we use lp_solve,[1] accessed through the Java ILP[2] wrapper.

[1] `http://lpsolve.sourceforge.net/5.5/`
[2] `http://javailp.sourceforge.net/`

4.3.2. Modeling the problem

To be able to apply an LP solver to our problem, we need to describe it using variables and constraints of a linear program, and define an objective function.

Variables

We create a variable for the elevation of each connector as defined in section 4.1. For multiple joined connectors, though, we use only one variable – their elevation is supposed to be identical.

Constraints

The constraints described in section 4.2 can be translated into LP constraints. As most of the coefficients c_i are 0 for each of the constraints, it serves readability to avoid vector notation here.

It should be noted that incorrect OpenStreetMap data or invalid assumptions on our part can lead to contradictory constraints (an obvious example for the former would be a way with non-zero incline that contains a node more than once), and thus an unsolvable LP. If this occurs, we fall back to direct elevation assignment as described in section 4.1.2.

Same elevation To make sure that two connectors associated with the variables x_1 and x_2 are at the same height, we simply require:

$$1 \cdot x_1 + (-1) \cdot x_2 = 0$$

Based on the equation as given here, we could even merge the variables entirely. However, due to the changes we will introduce later to define our objective function, this is not generally possible for connectors at different positions.

Exact elevation difference Likewise, to require an elevation difference of exactly b between two connectors associated with the variables x_1 and x_2 (with x_1 being the upper connector), we require:

$$1 \cdot x_1 + (-1) \cdot x_2 = b$$

Minimum/maximum elevation difference For a minimum difference (maximum follows easily), we alter the operation in the previous constraint and require:

$$1 \cdot x_1 + (-1) \cdot x_2 \geq b$$

In some cases, the distance is relative to a line segment between two connectors associated with variables x_2 and x_3, rather than the second connector from before. We calculate the horizontal distances $d_{1,2}$ and $d_{1,3}$ between the connector associated with x_1 and the two ends of the line segment, and require

$$1 \cdot x_1 + \frac{-d_{1,3}}{d_{1,2} + d_{1,3}} \cdot x_2 + \frac{-d_{1,2}}{d_{1,2} + d_{1,3}} \cdot x_3 \geq b$$

Objective function

The objective function does not directly follow from our previous decisions and calls for some creativity and experimentation. For our implementation, we chose a relatively simple objective: We try to minimize the sum of differences from the previously approximated terrain elevation – i.e. we want to stay as close to the terrain obtained in chapter 3 as the constraints allow.

It is not straightforward to construct this objective function because absolute values involving variables cannot appear in a linear program. However, it is possible to work around that restriction: If we want $|x_i - ele_i|$ to appear in the objective function, we create two non-negative variables $x_{i,pos}$ and $x_{i,neg}$. The idea is to substitute x_i with $ele_i + x_{i,pos} - x_{i,neg}$. For $x_i \geq ele_i$, we expect $x_{i,pos} = x_i - ele_i$ and $x_{i,neg} = 0$. For $x_i < ele_i$, we want $x_{i,neg} = ele_i - x_i$ and $x_{i,pos} = 0$.

In the objective function, our substitution turns $|x_i - ele_i|$ into $|x_{i,pos} - x_{i,neg}|$. Because at most one of our new variables can be 0 at the same time, we can instead write $x_{i,pos} + x_{i,neg}$. Having this term appear in the objective function also makes sure that the solver will not assign a non-zero value to more than one of these variables and violate our definition – such a result would not be minimal.

The constraints also have to be modified. We apply our substitution to every constraint

$$\dots + c \cdot x_i \bowtie_i b$$

and obtain:

$$\dots + c \cdot (ele_i + x_{i,pos} - x_{i,neg}) \bowtie_i b - c \cdot ele_i$$

$$\dots + c \cdot x_{i,pos} + (-c) \cdot x_{i,neg} \bowtie_i b - c \cdot ele_i$$

After calculating the expression on the right side, this is again a valid LP constraint.

4.3.3. Results

The LP described in the previous sections succeeds at some tasks, such as simple tunnels (figure 4.6) and cliffs (figure 4.7). It also tames the triangulated surface of rivers and roads, and raises bridges above the water level (figure 4.5).

Various minor visible glitches are caused by existing OSM2World code and can be fixed by improving the underlying codebase: Working with raw OpenStreetMap data rather than the final models sometimes causes the intersection detection code to miss areas overlapping roads or other ways with a width. Triangulation artifacts also fall into that category.

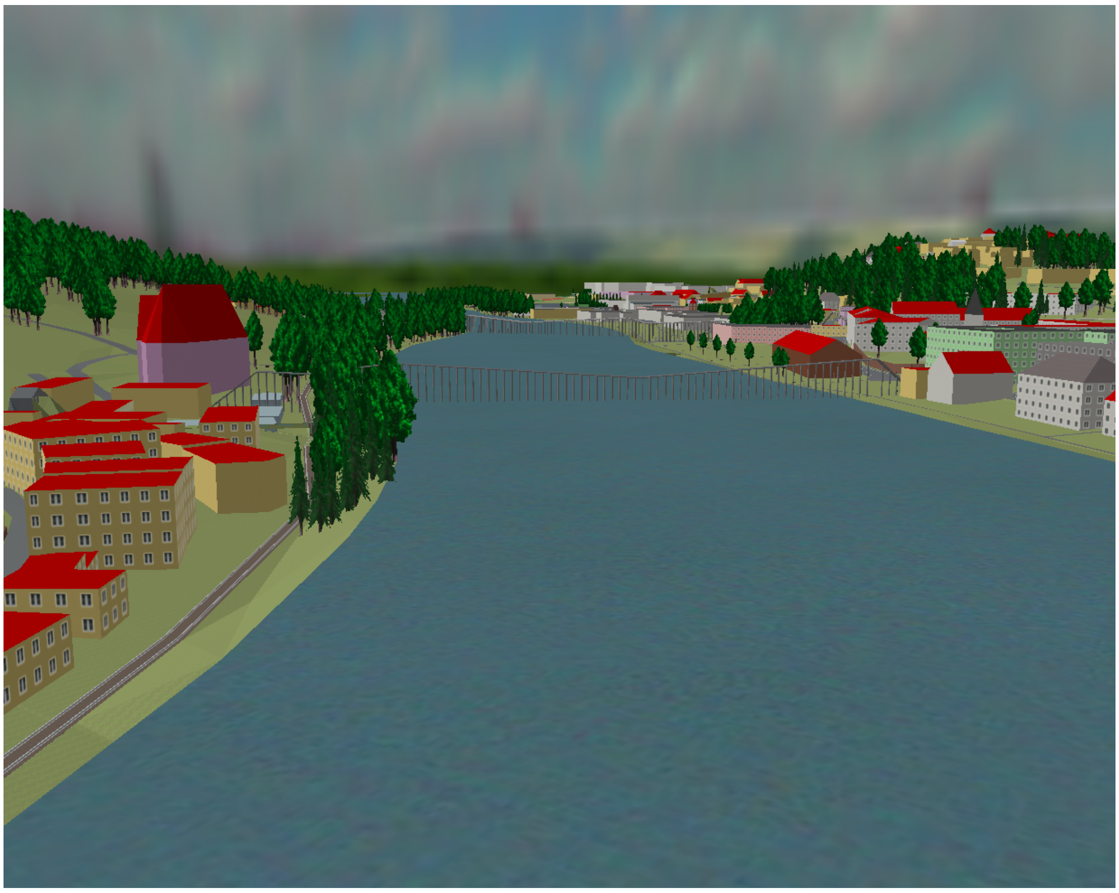


Figure 4.5.: Data from the OpenStreetMap database with constraints enforced using our LP, showing flat water surfaces, bridges and other details.

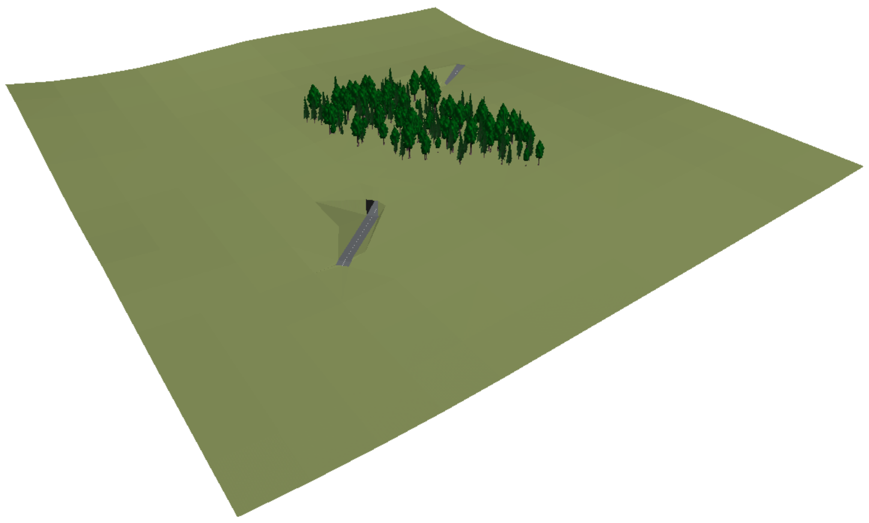


Figure 4.6.: An artificial tunnel test case, with constraints being used to keep the tunnel entrances open and stop the trees from sinking into the tunnel.

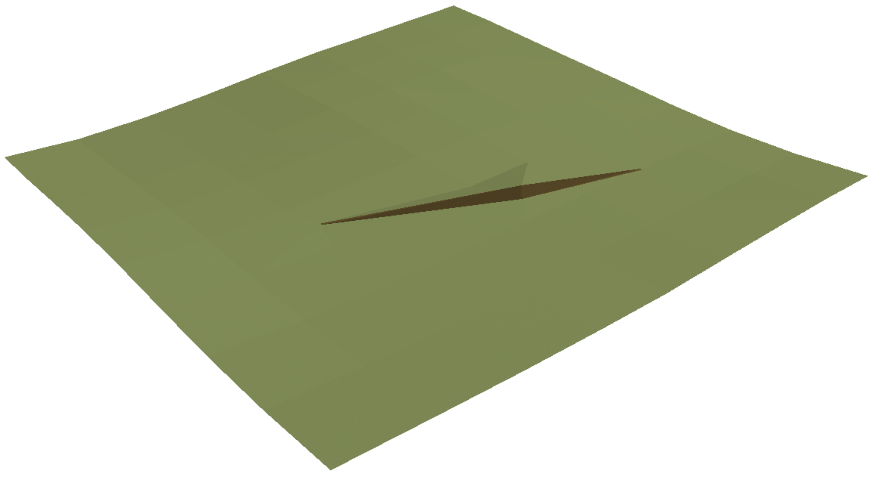


Figure 4.7.: A small cliff raised from the ground by minimum vertical distance constraints.

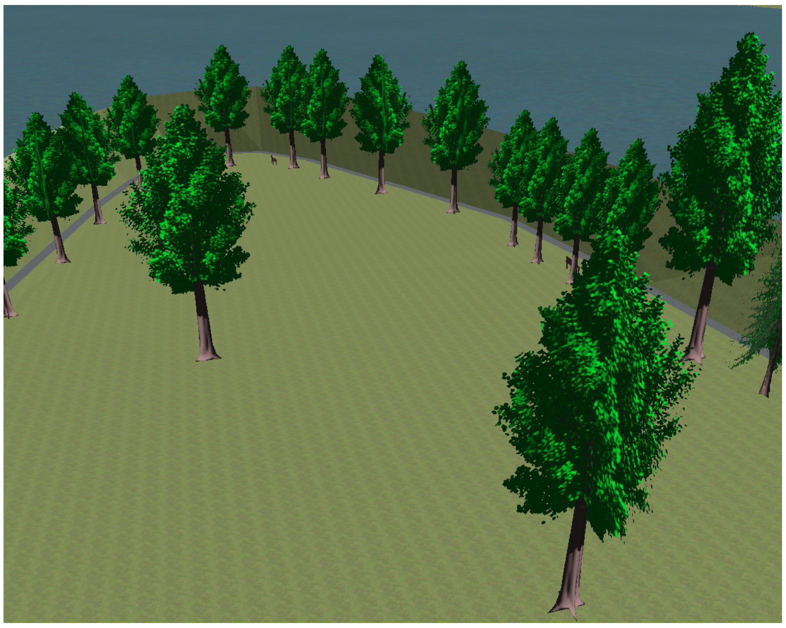


Figure 4.8.: None of the constraints prevent land from being at a lower elevation than a nearby water surface.

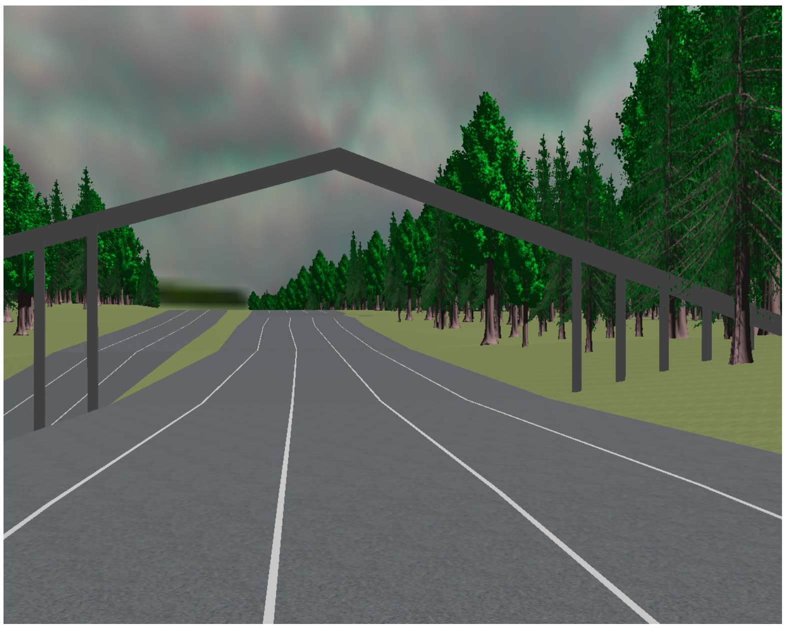


Figure 4.9.: Bridges carrying roads or railways often have unrealistic shapes.

However, other flaws are caused by the LP itself. For example, we would want water surfaces to be below the surrounding terrain (figure 4.8), but this is not being enforced by our basic set of constraints. Furthermore, bridges and tunnels often have sharp peaks at their highest connector (figure 4.9). The root cause of this is the low number of nodes in bridges – bridge ways are usually straight in 2D, so OpenStreetMap contributors will not insert additional nodes for smoothness as would be common practice with curved ways. Artificially increasing the node density could solve this.

Perhaps the most important drawback of our LP, though, is its simplistic objective function. While it is generally desirable to stay somewhat close to the approximated terrain, it neglects other relevant factors. In particular, avoiding abrupt incline transitions and steep inclines (beyond the hard constraints which can only prevent the most cases) would be desirable qualities.

Finally, a general issue with using an LP at all is its sensitivity to contradicting constraints. Due to the crowdsourced nature of OpenStreetMap, this may prove to be a major flaw of the approach. Falling back to an LP with relaxed constraints or the approximated terrain are both problematic strategies because they affect the entire scene. A robust algorithm where data errors have only local effects would be preferable.

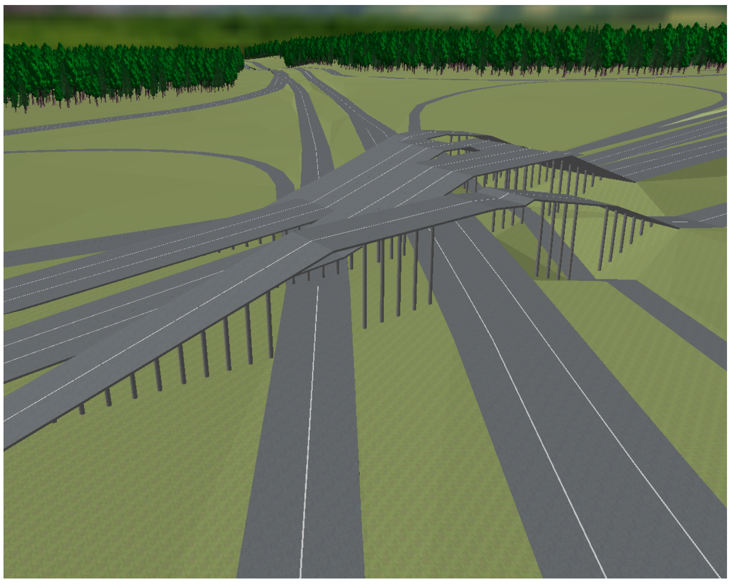


Figure 4.10.: Bridges at a motorway junction, exposing the shortcomings of our objective function. There is no incentive to smooth the bridge surface, and ground level roads are kept close to the approximated terrain at all costs.

5. Conclusion

As a first step (chapter 2), we were able to produce 3D models of various real-world features from OpenStreetMap data. We expect that the OpenStreetMap community continues to tackle unsolved problems in the data model and expand the coverage of existing tagging. If that happens, the project has great potential as a data source for 3D rendering and will easily meet and even exceed the needs of a wide range of popular application categories.

Our experimentation with terrain approximation in chapter 3 confirm that the vast toolbox of established algorithms available to developers is well equipped to deal with the task of creating good-looking terrains. Assuming some performance improvements obligatory for production use, even relatively straightforward approaches such as our elevation connector framework allow the use of OpenStreetMap data together with terrain approximated from external measurements. Thus, we expect that the project's community will be able to make good use of the ideas presented in this thesis.

As it turns out, the most challenging task is that of working smaller-scale detail based on OpenStreetMap vector data into the terrain's elevation. Our overview of constraints in section 4.2 shows and categorizes much of the information that could be obtained from the open database, and the subsequent experiments produce encouraging results for certain isolated examples which underline the desirability of leveraging that information for 3D rendering. However, we cannot recommend our solution based on linear programming yet, due to the problems described in that section. Consistently obtaining results with reasonable performance and quality remains an open challenge.

A. Installing and using the prototype

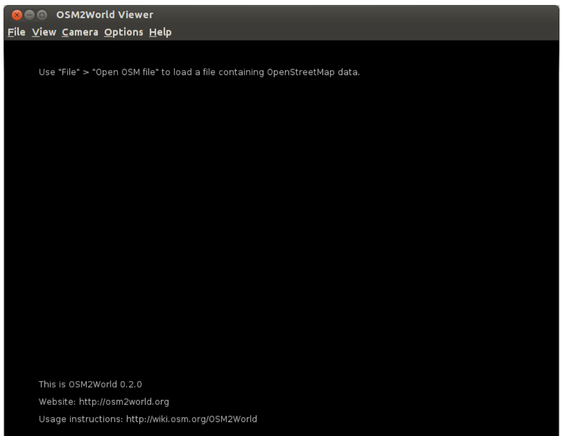
For this thesis, we developed a prototype implementation based on OSM2World. This appendix describes the necessary steps to install and run the prototype.

A.1. System requirements

Running the prototype requires a Java Runtime Environment in version 1.6 or higher. We use OpenGL through JOGL,[1] so the system should have graphics hardware and drivers capable of running modern OpenGL programs.

It is expected that the JVM can allocate 2 GB of memory for the prototype. Otherwise, it is necessary to modify the startup script or pass appropriate parameters.

If you want to experiment with the linear program, make sure to install the native dependencies for lp_solve.[2] If the *LP* option is not enabled, the rest of the prototype will run fine without lp_solve, though.



```
Use "File" > "Open OSM file" to load a file containing OpenStreetMap data.

This is OSM2World 0.2.0
Website: http://osm2world.org
Usage instructions: http://wiki.osm.org/OSM2World
```

Figure A.1.: Program window of the prototype's viewer application.

[1]`https://jogamp.org/jogl/`
[2]`http://lpsolve.sourceforge.net/5.5/Java/README.html`

A.2. Installation and program start

An archive containing the software is available on the CD-R bundled with the printed version, or from the author's website.[3] To install the software, it is sufficient to unpack the archive.

Start the prototype by executing the startup script appropriate for the operating system (`osm2world.sh` on Linux and Mac, `osm2world-windows-*.bat` on Windows).

A.3. Loading OpenStreetMap data

OpenStreetMap data must be stored as a file in .osm,[4] .osm.gz, .osm.bz2, or .osm.pbf[5] format to be used as input for the prototype. Some example files are included, but you can download more data from the OpenStreetMap database using the project's website or editor software such as JOSM.[6] For the prototype to function properly, .osm files must contain one valid `<bounds>` element as documented in the format's specification (many sources for OpenStreetMap data will create this automatically).

To open a file, use the *File* menu, or drag & drop the file into the program window. After changing configuration options, you will usually have to reload the file.

The prototype needs access to SRTM data for the geographical area covered by the OpenStreetMap input file. These need to be placed in the `srtm` subdirectory of the program's working directory. Again, a number of example files are included, and more can be obtained from NASA's SRTM download page.[7]

A.4. Navigation

When the prototype has finished the conversion, an interactive 3D rendering of the output will appear. You can navigate through the scene using your mouse or keyboard. Note that it is possible to reset the camera to its initial position from the *Camera* menu.

- **Left mouse button**: drag the mouse to move the camera position
- **Right mouse button**: drag the mouse to rotate the camera
- **Mouse wheel**: move the camera closer to the scene or away from it

[3]`http://tobias-knerr.de/publications/thesis/`
[4]`http://wiki.openstreetmap.org/wiki/OSM_XML`
[5]`http://wiki.openstreetmap.org/wiki/PBF_Format`
[6]`https://josm.openstreetmap.de/`
[7]`http://dds.cr.usgs.gov/srtm/`

- **W/A/S/D**: move the camera position
- **Arrow keys**: rotate the camera
- **Page up/down**: move the camera up/down

A.5. Configuration options

The following settings are available in the *Options* menu:

- **TerrainInterpolator**: offers a choice between the terrain approximation algorithms presented in detail in chapter 3. The option *Zero* will generate completely flat terrain.
- **EleConstraintEnforcer**: enables or disables the linear program introduced in section 4.3. With the *None* option, direct elevation assignment as described in section 4.1.2 is used, disregarding constraints.

A lot more options are available from the command line and configuration files. These are pre-existing features from the OSM2World codebase, though, and not directly related to this thesis. Refer to OSM2World's documentation[8] for them.

A.6. Views

The *View* menu offers a variety of different ways to look at the program's results. The following views illustrate concepts from this thesis:

- **EleConnectorDebugView**: shows the elevation connectors introduced in 4.1
- **EleConstraintDebugView**: shows some of the constraints introduced in 4.2
- ***InterpolatorDebugView**: these four views show empty shaded terrain created by one of the terrain approximation algorithms

[8] `http://wiki.openstreetmap.org/wiki/OSM2World`

B. Bibliography

[AK00] F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and G. Urrutia, editors, *Handbook of Computational Geometry, Chapter V*, pages 201–290. Elsevier Science Publishing, 2000. [SFB Report F003-092, TU Graz, Austria, 1996].

[BN08] Eric Bruneton and Fabrice Neyret. Real-time rendering and editing of vector-based terrains, 2008.

[dB00] M. de Berg. *Computational geometry: algorithms and applications.* Springer, 3rd edition, 2000.

[DPT01] Olivier Devillers, Sylvain Pion, and Monique Teillaud. Walking in a triangulation. Technical Report RR-4120, INRIA, 2001.

[Far02] G.E. Farin. *Curves and Surfaces for CAGD: A Practical Guide.* The Morgan Kaufmann Series in Computer Graphics. Elsevier Science, 2002.

[LG04] Hugo Ledoux and Christopher Gold. An efficient natural neighbour interpolation algorithm for geoscientific modelling. In *Proc. 11th Int. Symp. Spatial Data Handling*, pages 23–25, 2004.

[Lon05] P. Longley. *Geographic Information Systems and Science.* Wiley, 2nd edition, 2005.

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008.

[NZ12] Pascal Neis and Alexander Zipf. Analyzing the contributor activity of a volunteered geographic information project — the case of OpenStreetMap. *ISPRS International Journal of Geo-Information*, 1(2):146–165, 2012.

[NZZ11] Pascal Neis, Dennis Zielstra, and Alexander Zipf. The street network evolution of crowdsourced maps: OpenStreetMap in Germany 2007–2011. *Future Internet*, 4(1):1–21, 2011.

[OSN$^+$09] Martin Over, Arne Schilling, Steffen Neubauer, Sandra Lanig, and Alexander Zipf. Virtuelle 3D Stadt- und Landschaftsmodelle auf Basis freier Geodaten. 2009.

[RNJ07] H. I. Reuter, A. Nelson, and A. Jarvis. An evaluation of void-filling interpolation methods for SRTM data. *Int. J. Geogr. Inf. Sci.*, 21(9):983–1008, January 2007.

[Sib81] Robin Sibson. A brief description of natural neighbour interpolation. *Interpreting multivariate data*, 1981.

[SLNZ09] Arne Schilling, Sandra Lanig, Pascal Neis, and Alexander Zipf. Integrating terrain surface and street network for 3D routing. *3D Geo-Information Sciences*, pages 109–126, 2009.

[UZ12] Matthias Uden and Alexander Zipf. OpenBuildingModels – towards a platform for crowdsourcing virtual 3D cities. *7th 3D GeoInfo Conference. Quebec City, QC, Canada*, 2012.

[Vai89] Pravin M. Vaidya. An O(n log n) algorithm for the all-nearest-neighbors problem. *Discrete & Computational Geometry*, 4(1):101–115, 1989.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich diese Masterarbeit selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle wörtlich oder sinngemäß übernommenen Ausführungen wurden als solche gekennzeichnet. Weiterhin erkläre ich, dass ich diese Arbeit in gleicher oder ähnlicher Form nicht bereits einer anderen Prüfungsbehörde vorgelegt habe.

Passau, den 2. Mai 2013

(Tobias Knerr)