Institute for Formal Methods of Computer Science
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelor Thesis

OpenStreetMap Risk Maps

Patrick Lindemann

Course of Study:	Computer Science
Examiner:	Prof. Dr. Stefan Funke
Supervisor:	M.Sc. Tobias Rupp
Commenced:	May 3, 2021
Completed:	November 3, 2021

Abstract

Military strategy games such as the popular turn-based board RISK® by Hasbro offer the opportunity to create playable maps that use real cartographical material, due to the fact that players enjoy to play such games on boards that come as close as possible to real maps. Using the freely available and continuously maintained geodata of the OpenStreetMap project, we have developed a MapMaker tool that mostly automatizes the map generation process. We use various algorithms to compress and assemble administrative boundaries, which are extracted from the geographic source data, and collect additional metadata such as the adjacency of territories to create a fully playable map. Last but not least, we use the online strategy game Warzone for the testing and evaluation of our work with real players.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

The main focus of this work is to create playable maps for strategy games, such as the popular board game RISK® by Hasbro, by using real geographic map data. The following sections are intended to introduce the reader to *OpenStreetMap*, which provides the map data and *Warzone*, an online strategy game that we use to test and evaluate our work.

1.1 Motivation

RISK® is a turn-based strategy board game of world conquest [Gor13] for two up to six players. The most popular version of the game is played on a board that portrays the world map, which is divided into 42 territories that are grouped into 6 continents [Bro63].

At the start of the game, the territories are distributed by cards from the deck evenly among the players. Afterwards, players receive varying amounts of army figures, which they can assign to territories that they control. These armies can be used to attack adjacent territories controlled by an opponent engaging such in a battle by dice rolls, whereas the result of the attack attempt is determined by the dice roll result of the attacker compared to the one of the defendant. Controlling all territories of a continent grants the player a *bonus* in the form of additional troops in each turn. The goal of the standard mode of the game is to defeat and eliminate all opponents by conquering and occupying every territory on the board, achieving the full world domination.

The main focus of this project is to create boards that can be used for play in strategy games like RISK® by using political boundaries (specifically country borders and their administrative subdivisions) of real geographic data. Although it is possible to create such maps manually, this process is very time-consuming and error-prone, especially when working with large map sections that contain many boundaries. We will automatize this process by providing a *MapMaker* tool that assembles the geometries of geographic data extracts and converts them into board maps with territories and bonuses.

In order to make our generated maps playable, we calculate additional *metadata* for the extracted boundaries, namely:

- The center point of territories to act as positions for labels,

- A neighborship graph that contains the adjacencies of territories, with the purpose that armies can be moved between them, and
- A hierarchy that groups territories together to form bonuses which grant the player additional armies.

The following two sections will present the OpenStreetMap, which provides our work with geographic data and Warzone, an online spin-off version of RISK® that we use to test and evaluate the generated maps.

OpenStreetMap

Founded in 2004, the collaborative OpenStreetMap Project[1] aims to create a free geographic database of the world [Pro21b]. The map data is contributed and actively maintained by a community of volunteers and released with an open-content license [Pro21a]. In contrast to other map services, OpenStreetMap does not only provide the maps, but also the underlying geodata that can be used and further processed as desired. As of 2021, OpenStreetMap counts more than 8 million users [Pro21c].

Figure 1.1: The OpenStreetMap logo
Source: https://www.openstreetmap.de/

In the context of our work, we rely on geodata obtained from the OpenStreetMap project. In general, we are interested in the administrative boundaries for countries and their subdivisions, so that we can replicate the real geometries in our playable maps. In order to understand how this data is composed and further processed, we will elaborate the core concepts of the OpenStreetMap objects further in the following chapter.

Warzone

Similar to RISK®, Warzone[2] is a turn-based strategy game that can be played online between 2 and 40 players, in teams or individually [War21]. Each game takes place on a specified map,

[1] https://www.openstreetmap.org/
[2] https://www.warzone.com/

which is a collection of *territories*. Each territory is filled with a color that indicates by which player it is controlled, and a number that indicates how many armies are currently present in this territory. A player or team wins the game by conquering all territories on the entire map, with the result that none of the opponent players has any territories left [Wik21k].

Figure 1.2: The Warzone logo
Source: https://www.warzone.com/

Each turn consists of three phases [Wik21k]:

1. *Army deployment*: at each turn, the player receives a number of armies which can be distributed over controlled territories.
2. *Attack and Transfer*: after the armies have been deployed, the player can move armies between same player-controlled territories that are adjacent. The type of move is determined by the status of the target territory:
    a) *Attack*: if the target territory belongs to an enemy (or is neutral), it will be attacked. Attacks can succeed or fail, what will be determined by the number of armies attacking and defending the territory. Each army that attacks has a 60% chance to defeat one defending army [Wik21j]. If all armies of the defending territory are defeated, the respective territory is conquered and occupied by the attacker.
    b) *Transfer*: if the target territory is friendly (i.e. it belongs to the player or a teammate), the armies may be transferred. Transfers always succeed.
3. *Confirmation*: after preparing the aforementioned moves, they are reviewed and confirmed by the player before submitting them to the Warzone server. Afterwards, the orders will be executed one-by one in the specified order for each player.

In our work, we create maps that are customized to work with Warzone, so that the created maps can be play-tested by real users. We chose Warzone as the testing ground for our results because of its simplicity regarding the map creation process and for its active community of players and map creators, who could benefit directly from a tool that partially automatizes the map making process. In addition, Warzone also defines quality requirements for maps [Wik21l], which we will use to evaluate our results against.

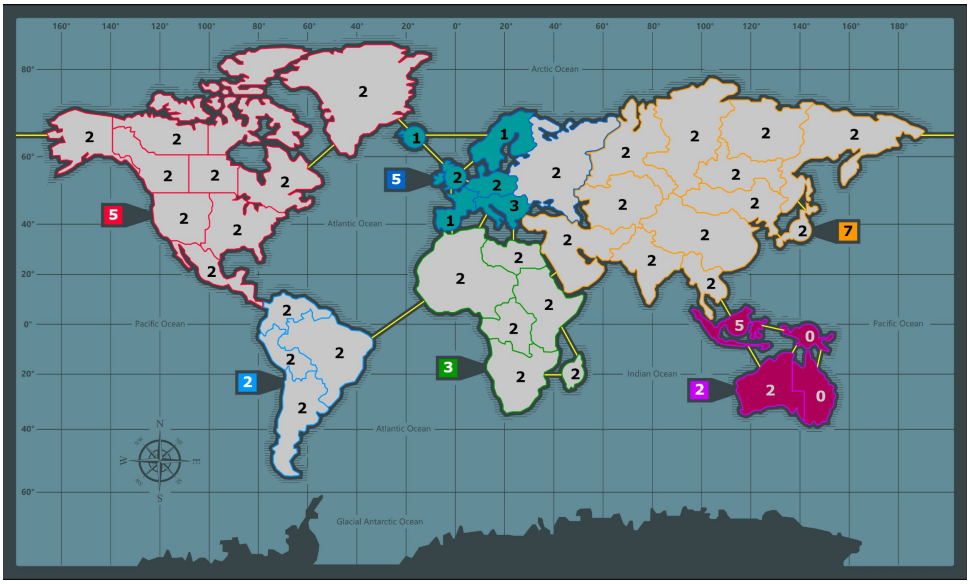


Figure 1.3: An ongoing game on the world map in Warzone

1.2 Related Work

There exist multiple free tools that allow users to extract selected administrative boundaries and their subdivisions from OpenStreetMap. For instance, OSMBoundaries[3] provides an online interface through which a registered OpenStreetMap user can select boundaries for one or multiple countries and convert them to GeoJSON data.

For Warzone, the community developed a the tool collection *Warlight Tools*[4] which provides algorithms to calculate metadata information such as center points and adjacencies of specified *.svg* map files. This tool also uses API endpoints provided by Warzone [Wik21m] to automatize the metadata upload process. However, the implemented algorithms operate directly on the geometries, with the result of low performances for larger maps, as this was reported by users in [War11].

Our work intends to combine both of these parts into a single pipeline to create an MapMaker application that is easy to use and that requires minimal knowledge for users. In addition, we implement efficient algorithms to achieve speedups for the map creation and metadata calculation process, especially for larger maps. At last, our tool uses the the Warzone API to further automatize the upload process.

[3]https://osm-boundaries.com/Map
[4]https://github.com/MatmaRex/Warlight-Tools

2 Foundations

In this Chapter, we will introduce the basic concepts and terminology for the data representation of OpenStreetMap, which we already introduced shortly in Chapter 1. Thereupon, we will dedicate a short section to map projections and transformations that will be applied to the extracted geographic locations. At last, we will define geometric constructs and consistency criteria for those in order to use them for our map-making algorithms in Chapter 3.

2.1 OpenStreetMap Data

OpenStreetMap (OSM) uses three types of basic elements to model data of the real world [Wik21c]:

1. *Nodes* define points in space,
2. *Ways* define linear features, and
3. *Relations* are used to explain how elements work together.

In addition, *Tags* which are key-value-pairs can be added to elements to describe their meaning and to provide additional meta information. As of November 2021, OpenStreetMap contains over 7.2 billion nodes, 811 million ways and 9.3 million relations [Pro21c].

OpenStreetMap data mainly comes in the form of XML formatted *.osm* files [Wik21b]. The planet file *Planet.osm* contains the data of the whole project and is updated on a weekly basis. However, due to its massive size of more than 100 gigabytes, mappers work with so-called *Extracts* of the map, which only cover a specific part of the planet and are usually much smaller in size. Moreover, OSM files are usually interchanged as compressed *.pbf* files using the *Protocolbuffer Binary Format* [Wik21f] to further reduce the file sizes and improve reading and writing times.

The planet file and smaller extracts of the OpenStreetMap project are freely available and may be downloaded from official sources[1] or from third-party providers such as Geofabrik[2].

[1] https://planet.openstreetmap.org/
[2] https://download.geofabrik.de/

Nodes

A node represents an arbitrary point on the earth's surface, which is defined by longitude and latitude[Wik21e]. Most nodes are used to define the path (or shape) polylines, but they can also be extended with an arbitrary amount of tags to represent standalone point features, for example trees, bridges, stores and more.

```
<node id="1674998233" lat="48.7790552" lon="9.1785426">
   <tag k="name" v="Cafe Königsbau"/>
   <tag k="amenity" v="cafe"/>
   <tag k="opening_hours" v="Mo-Sa 09:00-19:00; Su,PH 11:00-19:00"/>
</node>
```

List of Listings 2.1: A café in Stuttgart represented by a node.

Ways

A way is an ordered list of up to 2,000 nodes that represent continuous linear features. [Wik21i] Like nodes, they can contain tags with additional metadata information to specify standalone features such as roads, rivers or walls. On the other hand, ways are also used to define the outer (and inner) boundaries of relations and areas. Ways that start and end at the same node are referred to as *closed ways*.

```
<way id="25490102">
   <nd ref="27168602"/>
   <nd ref="27168603"/>
   <nd ref="886217198"/>
   <nd ref="27168604"/>
         ⋮
   <nd ref="27168607"/>
   <tag k="name" v="Universitätsstraße"/>
   <tag k="highway" v="tertiary"/>
   <tag k="lanes" v="2"/>
   <tag k="maxspeed" v="70"/>
</way>
```

List of Listings 2.2: A way with multiple node references that represents a street.

Relations

A relation is a multi-purpose data structure that defines a relationship between two or more object members such as nodes, ways and other relations [Wik21c]. They usually include a `type`

value which indicates how the relation is to be interpreted. Established types include routes (such as metro lines or numbered highways), waterways and multipolygon geometries such as buildings [Wik21h]. Each member can optionally define a `role` value within the relation.

```
<relation id="9396889">
   <member type="way" ref="406220223"/>
   <member type="node" ref="8529770817" role="stop"/>
   <member type="way" ref="89155651"/>
   <member type="node" ref="129999865" role="stop"/>
      ⋮
   <member type="way" ref="9837998"/>
   <member type="node" ref="129999865" role="stop"/>
   <tag k="type" v="route"/>
   <tag k="name" v="23: Ruhbank (Fernsehturm) -> Straßenbahnmuseum"/>
   <tag k="from" v="Ruhbank (Fernsehturm)"/>
   <tag k="to" v="Straßenbahnmuseum"/>
   <tag k="route" v="tram"/>
</relation>
```

List of Listings 2.3: A metro line represented by a relation with multiple node and way members.

Areas

In contrast to the previously mentioned node, way and relation objects, areas are not native to OpenStreetMap. However, they are almost treated like real OSM objects [Top]. Since areas have no distinct representation in OSM data, they can be defined in one or both of the following two ways according to [Wik21a]:

1. *Explicit*: relations and closed ways that contain the tag `area=true` are explicitly declared as an area. In addition, relations with the type `multipolygon` are always considered to be areas.
2. *Implicit*: there are several tags that imply that a relation or closed way is an area. For example, a closed way with the tag `landuse=forest` is interpreted as a tree-covered area by default rather than a row of trees. However, there can be exceptions, which is why the tag `area=false` can be used to prevent implicit declarations of areas.

```
<relation id="2793104" version="83" timestamp="2021-03-22T20:18:32Z">
   <member type="way" ref="333901568" role="outer"/>
   <member type="way" ref="49931697" role="outer"/>
   <member type="way" ref="49931741" role="outer"/>
         ⋮
   <member type="way" ref="70547034" role="outer"/>
   <member type="way" ref="70547255" role="inner"/>
```

```
    <member type="way" ref="70547831" role="inner"/>
    <member type="node" ref="1674026139" role="admin_centre"/>
    <tag k="name" v="Stuttgart"/>>
    <tag k="type" v="boundary"/>
    <tag k="boundary" v="administrative"/>
    <tag k="admin_level" v="6"/>
</relation>
```

List of Listings 2.4: A boundary area for the administrative borders of the city of Stuttgart

Multipolygon areas are closed ways or relations that include the tag `type=multipolygon`. In case of relations, these contain multiple way members with an assigned `role` each (see listing 2.4). These roles can have two different types: `outer` and `inner`. These members must not necessarily be closed ways that represent a ring each; they can also be partial lines that agglomerate the rings in a specific order (Fig. 2.1). On the one hand, this has the advantage that partial borders between areas which are same do not need to be specified separately, on the other hand, the member ways need to be assembled with specific algorithms (which will be introduced in Section 3.3) in order to retrieve the actual area geometries.

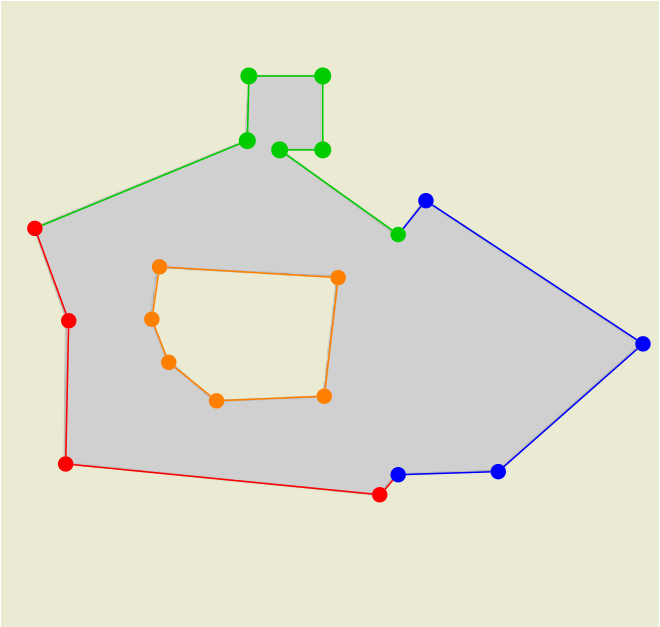


Figure 2.1: A multipolygon area with one inner and three outer way members.

Boundaries

In our map making process, we work with administrative boundaries exclusively, which are areas, territories or jurisdictions recognised by governments or other organisations for

administrative purposes only [Wik21d]. In OpenStreetMap, these are specified as areas that contain the `boundary=administrative` tag.

Administrative boundaries can have subdivisions at different depths, which are referred to as *administrative levels* or simply *levels*, that are specified with the `admin_level=*` tag. These levels are integers between 2 and 11, whereas the level 2 marks country borders, while the levels 3 to 11 delimitate sub-national boundaries, which are country-specific. Table 2.1 shows the different mappings for the levels 3 to 11 for Germany and the United States.

Level	United States	Germany
3	-	-
4	State	Bundesland
5	New York City	Regierungsbezirk
6	State County	Landkreis
7	Civil Township	Amt
8	City/Town/Village	Stadt/Gemeinde
9	Ward	Stadtbezirk
10	Neighborhood	Stadtteil
11	-	Stadtviertel

Table 2.1: Administrative levels and their mapping for the United States and Germany

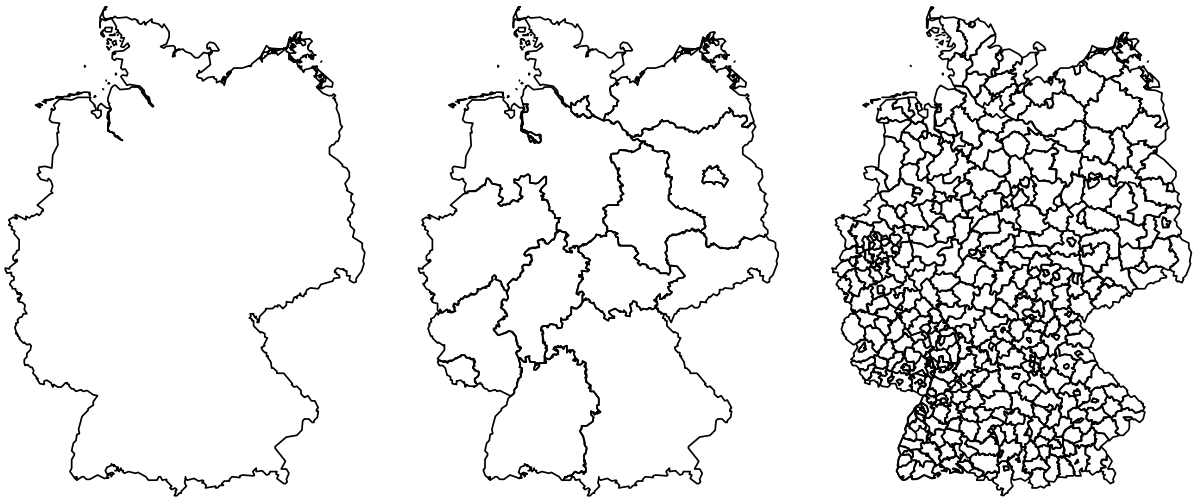


Figure 2.2: Administrative boundaries of Germany with the levels 2, 4 and 6

2.2 Projections and Transformations

As the earth is spherical, locations on its surface cannot be represented by a planar coordinate system such as the Euclidean space. For this purpose, *geographic coordinate systems* (GCS) [D0015], which we also refer to as *spherical coordinate systems*, are used to describe *geographic locations* on the spherical surfaces with *longitude* and *latitude* coordinates.

By using a GCS, a sphere's surface is divided into a grid by equidistant lines. The vertical lines (North/South) are called *meridians*, the horizontal lines (East/West) are called *parallels*. The longitude λ of a geographic location denotes the angle East or West and the latitude ϕ the angle North or South to a fixed meridian and parallel respectively. For the earth's coordinate system, the central meridian was chosen internationally as the meridian that passes through Greenwich in England, which is also called *prime meridian*, and the central parallel as the equator.

Formally, geographic locations are two-dimensional vectors $\begin{pmatrix}\lambda\\ \phi\end{pmatrix}$ with $\lambda \in [-180°, 180°]$ and $\phi \in [-90°, 90°]$. In the following, we assume that these locations were converted into radians, such that $\lambda \in [-\pi, \pi]$ and $\phi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$.

For our work, we want to represent points on a planar, Euclidean surface to perform calculations and to display the resulting geometries. To accomplish this, we use so-called *map projections*.

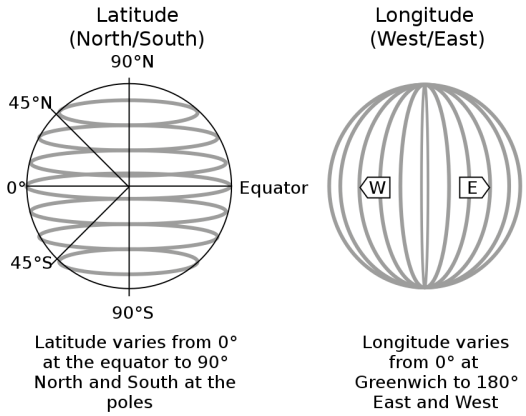


Figure 2.3: The graphical depiction of latitude and longitude [Com05]

2.2.1 Map Projections

In cartographic terms, a *map projection* is a systematic representation of all or a part of a surface's round body on a plane in order to create a map [Sny72]. We define a *map projection function* as a function ρ with

$$\rho : [-\pi, \pi] \times \left[-\frac{\pi}{2}, \frac{\pi}{2} \right] \to \mathbb{R}^2 \tag{2.1}$$

that maps cylindrical locations $(\lambda, \phi)^\top$ given in radians to locations $(x, y)^\top$ on a two-dimensional plane.

Map projections can have one or multiple characteristics. We define three of the most important properties according to [Sny72]:

- *Equal-area*: the projection retains the surface area of shapes relative to the scale. However, the angles and scale of most parts of the map must be distorted to achieve this property.
- *Conformal*: the projection shows the relative local angles for approximately every point correctly. Depending on the map size, large areas must be shown as distorted in shape, while small features stay shaped essentially correctly.
- *True-to-scale*: the projection retains the scale of one or more lines throughout the map. If a projection shows the true scale between one or two points and any other point of the map, it is called *equidistant*.

Some of these properties are mutually exclusive - for example, no projection can both be equal-area and conformal [Sny72]. Therefore, every projection contains some kind of error and must be chosen by the mapper depending on the purpose of the map.

Mercator Projection

For our maps, we chose the *Mercator Projection*, which was presented by Gerardus Mercator in 1569 and is the most famous projection to this day [Sny72]. The Mercator map projection function is defined as

$$\rho_M : [-\pi, \pi] \times \left[-\frac{\pi}{2}, \frac{\pi}{2} \right] \to \mathbb{R}^2, \quad \begin{pmatrix} \lambda \\ \phi \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.2}$$

with

$$x = R(\lambda - \lambda_0) \tag{2.3}$$

$$y = R \ln \tan \left(\frac{\pi}{4} + \frac{\phi}{2} \right) \tag{2.4}$$

where $\lambda_0 \in [-\pi, \pi]$ denotes the central median for the projection and R the radius of the sphere relative to the desired scale of the projected map. By default, λ_0 is chosen as 0, which

denotes the prime meridian, but it can be set to the longitude of an arbitrary meridian to shift the projected locations along the x-axis.

The Mercator projection is a conformal projection. It is important to note that the value of y moves towards infinity as ϕ approaches $\pm\frac{\pi}{2}$. This leads to greater distortions of the size of areas that are further away from the equator, which is the reason why Greenland or the Antarctica for example appear many times larger in size than they actually are (Fig. 2.4). We use the Mercator projection with the sphere radius $R = 1$ and apply our own transformations, which will be introduced in the next section.

2.2.2 Transformations

Usually, we work with map extracts that cover smaller sections of the earth's surface. Yet, the scale of projected maps is independent from the bounding box of the map extract. In this section, we define functions that fit an arbitrarily sized map extract to a specified width w and height h after the projection has been applied.

First, we define the normalization functions n_x and n_y for each dimension:

$$n_x : [x_{min}, x_{max}] \to [0, 1], \quad x \mapsto \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.5}$$

$$n_y : [y_{min}, y_{max}] \to [0, 1], \quad y \mapsto \frac{y - y_{min}}{y_{max} - y_{min}} \tag{2.6}$$

The minimum and maximum coordinates $x_{min}, x_{max}, y_{min}$ and y_{max} are retrieved by calculating the bounding box of the map extract. With these normalization functions, we ensure that all locations on the map are fitted to the edges of the bounding box and scaled relatively to the unit interval $[0, 1]$. To resize the normalized locations to a specified map width w and height h, we apply the scaling transformation

$$s : [0, 1]^2 \to [0, w] \times [0, h], \quad \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} w & 1 \\ 1 & h \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.7}$$

The combined function f fits and resizes the projected locations:

$$f : \mathbb{R}^2 \to [0, w] \times [0, h], \quad \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} w \cdot n_x(x) \\ h \cdot n_y(y) \end{pmatrix} \tag{2.8}$$

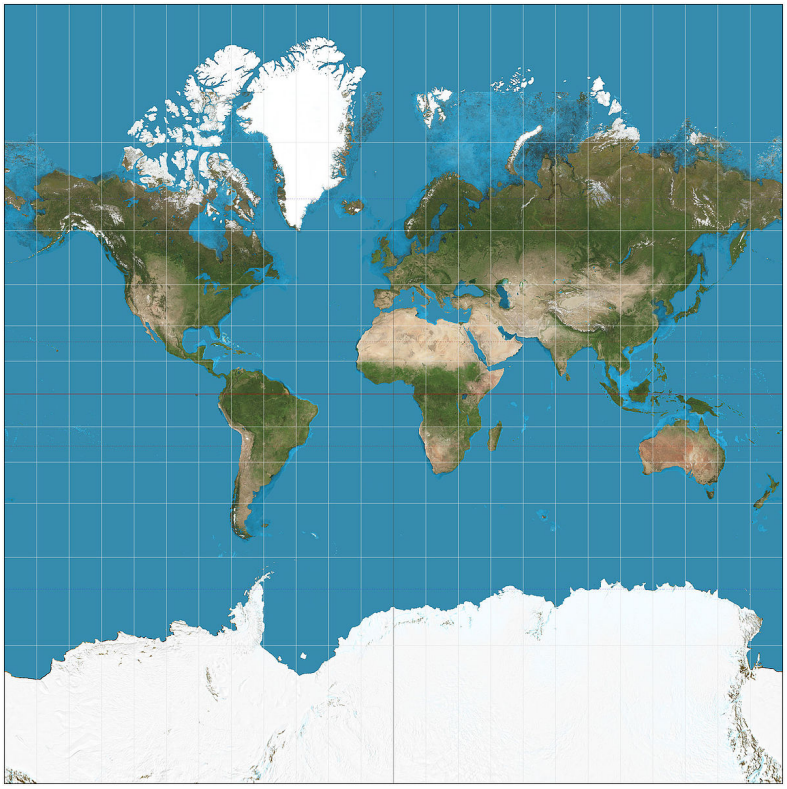


Figure 2.4: The world map created by applying the Mercator projection [Str11]

2.3 Geometry

The algorithms we use in our calculations in Chapter 3 do mainly work with the geometry of the extracted OSM objects. In this section, we introduce a terminology for geometric entities and define criteria to differentiate whether we consider a geometry to be valid or not. For simplicity, we assume that all locations are within the first quarter of the Euclidean space, which can be achieved by applying the map projections and transformations described in Section 2.2 to the extracted areas.

Primitive Objects

A *point* p is a two-dimensional vector (x, y) in the real vector space $\mathbb{R}^2$.

A *segment* s is a pair of two points (p_1, p_2) that represents a finite part of a line between p_1 and p_2.

A *polyline* l is an ordered, finite sequence $(p_1, \ldots, p_n)$ of n points, where at least 2 points are mutually different. The points of the line are called *vertices*, the consecutive pairs (p_i, p_{i+1}) are referred to as *edges*. The edges form visual segments that connect the vertices of the polyline.

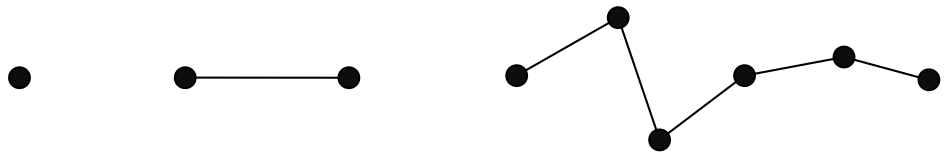
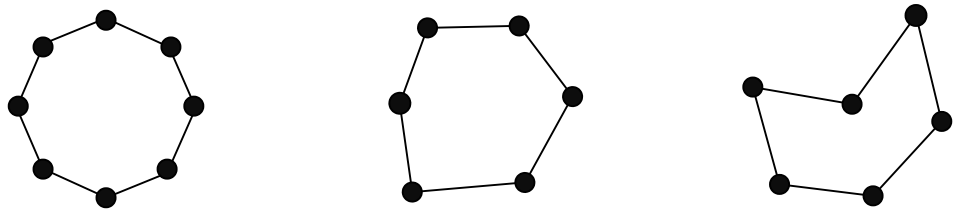


Figure 2.5: The geometric primitives: A point, segment and polyline



(a) A regular polygon **(b)** A convex polygon **(c)** A concave polygon

Figure 2.6: Examples for valid polygon geometries

Polygons

We define a *polygon* $\mathcal{P}$ of size n as a closed polyline with $n+1$ points $(p_1, \ldots, p_n, p_{n+1})$, such that $p_1 = p_{n+1}$. Further, the edges (p_i, p_{i+1}) with $1 \leq i < n+1$ are called *sides* of the polygon.

In our work, we are focusing just on polygons with simple geometries. A polygon $\mathcal{P}$ is *simple* if and only if it fulfills the following three consistency properties:

P-1) $\mathcal{P}$ has at least 3 sides.

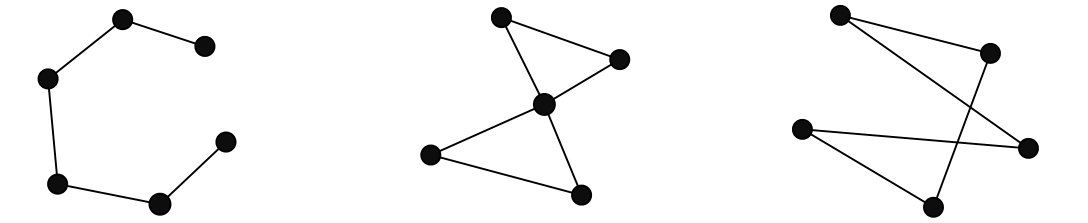
P-2) All points $p_i, p_j \in \mathcal{P}$ with $1 \leq i, j < n+1,\ i \neq j$ are mutually different.

P-3) No two edges $(p_i, p_{i+1}), (p_j, p_{j+1})$ of $\mathcal{P}$ with for $1 \leq i, j < n+1,\ i \neq j$ are intersecting.

We refer to non-simple polygons as *complex* polygons. Further, a simple polygon $\mathcal{P}$ is *convex* if every segment joining two points $(p_i, p_j) \in \mathcal{P}$ with $i \neq j$ lies completely inside of it. If a simple polygon is not convex, it is called *concave*. A simple polygon is *regular* if all edges have the same length (*equilateral*) and all corner have angles (*equiangular*).

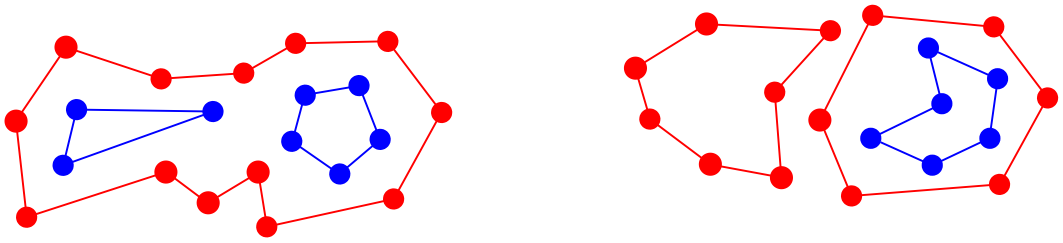
Figures

Multipolygon areas that were presented in Section 2.1 can have 1 to N outer rings and 0 to M inner rings. To properly map these ring structures, which essentially are sets of simple



(a) Unclosed polyline **(b)** Violation of property P-2 **(c)** Violation of property P-3

Figure 2.7: Examples for invalid polygon geometries



(a) A family of one figure that contains two holes **(b)** A family of two figures where one contains a hole

Figure 2.8: Valid family geometries

polygons that are topologically sorted, to geometric entities, we introduce the terms for composite polygons:

We define a *figure* $F = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$ as a tuple of one outer polygon $\mathcal{P}_0$ and n inner polygons $\mathcal{P}_1, \ldots, \mathcal{P}_n$ (also referred to as *holes*), such that all of the following conditions hold:

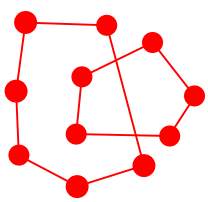
F-1) Every polygon $\mathcal{P} \in F$ is a simple polygon.

F-2) No two Polygons $\mathcal{P}_i, \mathcal{P}_j \in F$ with $i \neq j$ share a common point or intersect.

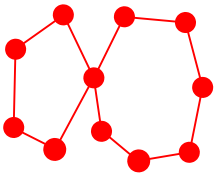
F-3) Every inner polygon $\mathcal{P}_i$ is inside of the outer Polygon $\mathcal{P}_0$.

F-4) No inner polygon $\mathcal{P}_i$ is inside of another inner polygon $\mathcal{P}_j$ with $i \neq j$.

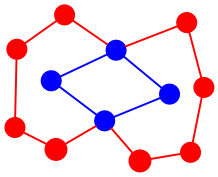
A *family* $\mathcal{F} = \{F_1, \ldots, F_m\}$ is a set of m pairwise different figures, such that no two Figures F_i, F_j with $i \neq j$ share a common point or intersect.



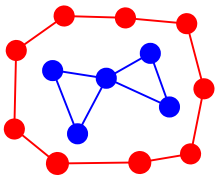
(a) Two figures intersect



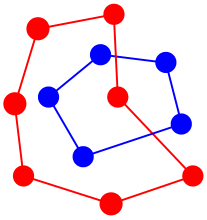
(b) Two figures share a common point



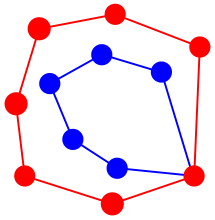
(c) Violation of propterty F-2



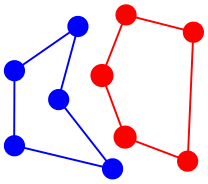
(d) Violation of property F-1



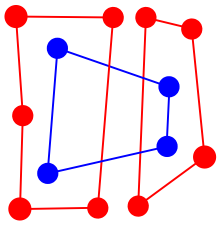
(e) Violation of property F-2



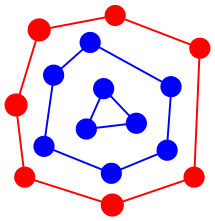
(f) Violaiton of property F-2



(g) Violation of property F-3



(h) violation of property F-3



(i) Violation of property F-4

Figure 2.9: Invalid figure and family geometries (outer polygons are red, inner polygons are blue)

3 Risk Map Generation

After we set the thematic framework in Chapter 1 and introduced basis of data that we work with in Chapter 2, we describe the components of the final MapMaker solution and discuss the general concepts and algorithms they implement.

For the sake of simplicity, we do not explicitly address individual component of the pipeline, but rather discuss the general concepts and algorithms they implement.

Not all of the algorithms we describe in this chapter work with geometries. For example, the neighborship algorithm which we present in Section 3.4 inspects the assembled areas and their ways to find adjacencies rather than inspecting the points of the individual geometries.

3.1 Map Requirements

In Chapter 1 we introduced the basic map structure for Warzone. In order to evaluate the quality of our generated maps, we define the following *quality criteria* in accordance to the map requirements of Warzone[Wik21l]:

Q-1 Each territory has exactly one outer ring.

Q-2 Territories can contain holes.

Q-3 No territory overlaps with other territories.

Q-4 Territories must be large enough to fit a two-digit army number.

Q-5 Connections between territories are visible.

3.2 Compression

As we stated in the Section 2.1, OpenStreetMap extracts may consist of billions of nodes and millions of ways and relations. For example, the comparably small extract of Baden-Württemberg in terms of its surface area contains 3,040 boundary relations, which consist of 650,904 nodes in total. Especially for larger map extracts, this level of detailing is not necessary; in addition, working with huge numbers of nodes substantially impacts the performance of all other steps in the pipeline and clutters the map outputs visually. Therefore, we need to

compress the detailed sections of boundaries while keeping at the same time their general shape.

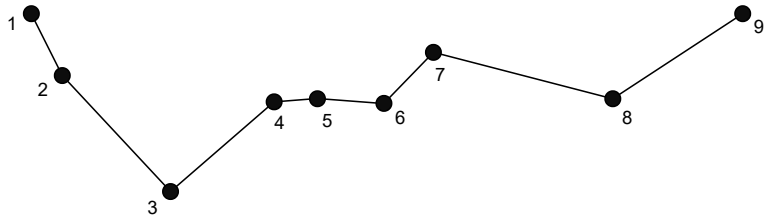
3.2.1 Ramer-Douglas-Peucker Algorithm

A well-known algorithm for the compression of geometric lines is the Ramer-Douglas-Peucker Algorithm [DP73], which was proposed by *David H. Douglas* and *Thomas Peucker* in 1973 and builds on an iterative algorithm for polynomial approximation of curves by *Urs Ramer*[Ram72]. The original algorithm works according to the *Divide-And-Conquer* principle using recursion.

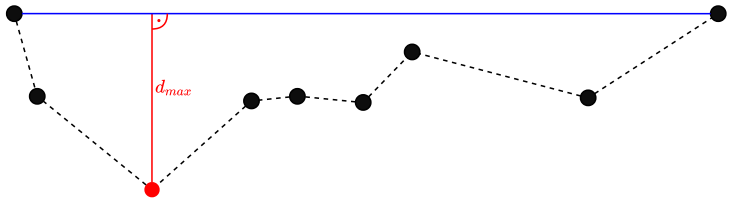
Let $l = (p_1, \ldots, p_n)$ be a polyline and let $\varepsilon > 0$ be a specified distance threshold, which can be interpreted as the level of compression applied to the sequence. The original algorithm works as follows:

1. Choose the points (p_1, p_n) as the compressed polyline.
2. Compute the perpendicular distance to the segment (p_1, p_n) for each point $p_2, \ldots, p_{n-1}$ and save the point p_i with the maximum distance d_{max}.
3. If d_{max} is greater than ε, the left polyline $(p_1, \ldots, p_i)$ and the right polyline $(p_i, \ldots, p_n)$ are compressed recursively. The resulting polylines are merged together, yielding the compressed polyline $(p_1, \ldots, p_i, \ldots, p_n)$.
4. If d_{max} is less or equal to ε, return the polyline (p_1, p_n), which implicitly removes the points $p_2, \ldots, p_{n-1}$ from the line.

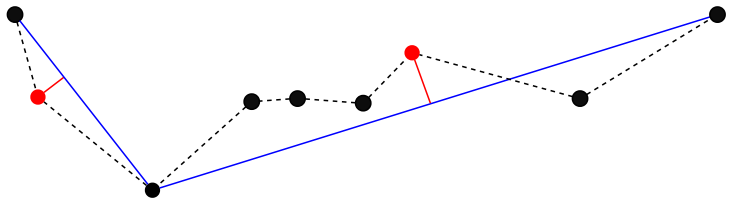
The end of the recursion is reached if the polyline input consists of two points or if the maximum distance d_{max} falls below the threshold ε, in which case the line will be compressed. The algorithm will always terminate for valid polylines, as each recursive call splits the line such that each side has a maximum of $n - 1$ points (which is the case for $p_i = p_2$ or $p_i = p_{n-1}$).



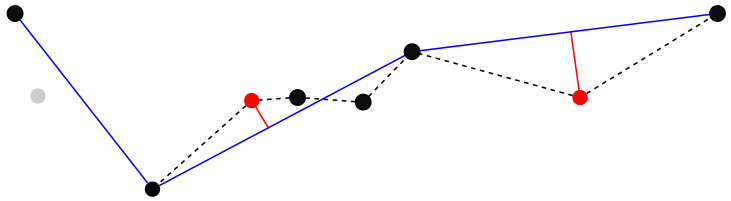
The uncompressed polyline



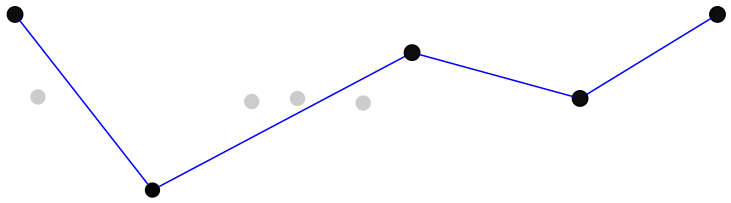
Identification of point 3 with the maximum distance to the line segment (1, 9)



Recursive call for the left (1, 2, 3) and right (3, 4, 5, 6, 7, 8, 9) polyline



Removal of point 2 due to its distance to the line segment (1, 3)



Removal of the points 4, 5 and 6 yields the compressed polyline

Figure 3.1: Ramer-Douglas-Peucker Compression after [Com20]

Algorithm 1: Ramer-Douglas-Peucker Algorithm

Input : A polyline $l = (p_1, \ldots, p_n)$ and a distance threshold $\varepsilon > 0$
Output: The compressed polyline l'

```
i := 0;
d_max := 0;
for k := 2, ..., n − 1 do
    d := perpendicularDist(p_k, (p_0, p_n));
    if d < d_max then
        i := k;
        d_max := d;
    end
end
if d_max > ε then
    (l_1, ..., l_s) := ramerDouglasPeucker((p_1, ..., p_i), ε);
    (r_1, ..., r_t) := ramerDouglasPeucker((p_i, ..., p_n), ε);
    return (l_1, ..., l_s, r_1, ..., r_t);
else
    ▷ Remove all points between p_1 and p_n
    return (p_1, p_n);
end
```

Complexity Analysis

Let $l = (p_1, \ldots, p_n)$ be a polyline with $n > 2$ points. In the worst-case scenario, the point p_i with the maximum distance $d_{max} < \varepsilon$ to (p_0, p_n) is either the leftmost point p_2 or the rightmost point p_{n-1}. Without loss of generality, we assume that $p_i = p_{n-1}$. In this case, the polyline is split into the left part $(p_1, \ldots, p_{n-1})$ and the right part (p_{n-1}, p_n), which reduces the length of the left polyline by 1.

If the described worst-case division of the line continues throughout every step of the recursion, the algorithm will require $O(n)$ subdivision steps. As each step iterates over the subdivided line once, this results in a time complexity in $O(n^2)$. Figure 3.3 shows a polyline for which the worst-case scenario would apply.

The best-case and average-case time complexities for the Ramer-Douglas-Peucker algorithm are in $O(n \log n)$, the latter being derived from proofs [Hoa62] for other divide-and-conquer algorithms such as *Quicksort*.

3.2.2 Compressing Areas

In our implementation, we apply compression to the polyline geometry of the extracted ways before assembling the areas. This is done because a way can be referenced by multiple areas (i.e. when two areas are adjacent). Consequently, if we would apply the compression after the area assembly, the polylines of such shared ways would be compressed twice.

However, this approach comes with a caveat: If a node which is referenced by multiple ways is removed, a hole can appear between said ways. This occurs due to the following two properties:

1. The compression result of a polyline is dependent on the chosen first and last point.
2. Two different ways w_i, w_j can share one or more nodes.

We introduce a simple fix for this problem: before the compression algorithm is performed on the way geometries, we create a node reference dictionary that maps each node n to a set of ways $W = \{w_i, \ldots, w_j\}$ which contain reference to it. Vice versa, we define the *cardinality* of a node n as the number of ways in W. Finally, if the cardinality of n is greater than 1, we simply ignore it during the compression procedure and persist it in the resulting polyline l'. The modified compression algorithm 2 implements this idea together with an iterative version of the Ramer-Douglas-Peucker algorithm.

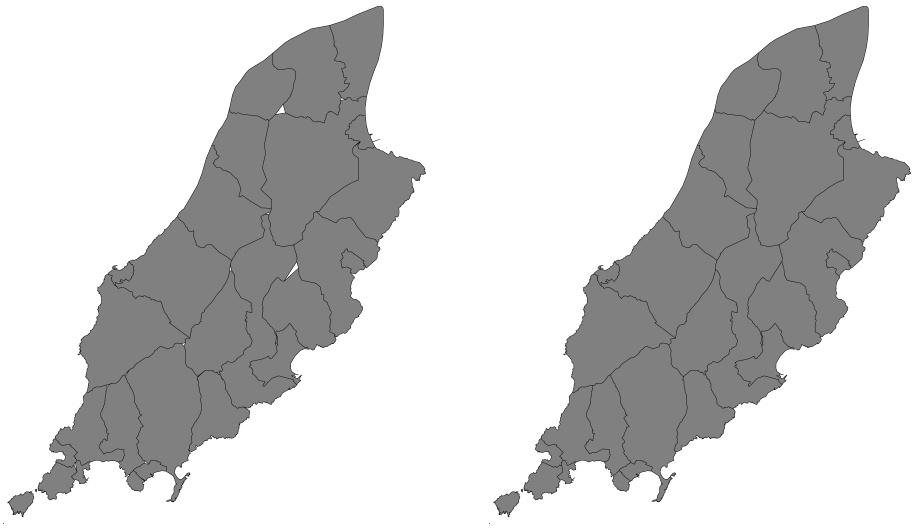


Figure 3.2: A comparison between the unmodified Ramer-Douglas-Peucker compression (left) and the modified version (right).

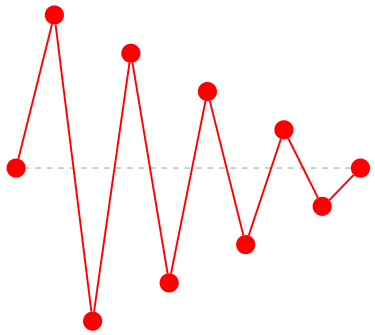


Figure 3.3: A polyline that has the worst-case runtime for the compression algorithm

Algorithm 2: Modified Ramer-Douglas-Peucker Algorithm

Input : A polyline $l = (p_1, \ldots, p_n)$ and a distance threshold $\varepsilon > 0$
Output: The compressed polyline l'

$stack := \{\}$;
$removed := \{\}$;
$stack$.push$((0, n))$;
while $stack \neq \emptyset$ **do**
    $(start, end) := stack$.pop$()$;
    $i := 0$;
    $d_{max} := 0$;
    for $k := start + 1, \ldots, end - 1$ **do**
        $d :=$ perpendicularDist$(p_k, (p_{start}, p_{end}))$;
        if $d < d_{max}$ **then**
            $i := k$;
            $d_{max} := d$;
        end
    end
end
if $d_{max} > \varepsilon$ **then**
    $stack$.push$((start, i))$;
    $stack$.push$((i, end))$;
else
    for $k := start + 1, \ldots, end - 1$ **do**
        if cardinality$(p_k) < 3$ **then**
            $removed$.add(p_k);
        end
    end
end
return $l \setminus removed$;

3.3 Assembly

The geometry of areas is not inherently declared in OpenStreetMap data, but must rather be assembled from the referenced way members (please refer to Section 2.1). This section presents the algorithms necessary in order to assemble and group these geometries and to validate them using the criteria that we defined in Section 2.3.

3.3.1 Line Intersection Tests

A crucial part of the assembly stage is the discovery of intersections between line segments. A naive approach to this problem would be to test each pair of segments for intersections:

Algorithm 3: Brute-Force Intersection Algorithm

```
Input  : A set of segments S = {s_1, ..., s_n}
Output : True if any two segments s_i, s_j intersect
for i := 1, ..., n − 1 do
    for j := i, ..., n do
        if intersect(s_i, s_j) then
            return true;
        end
    end
end
return false;
```

In the worst case, no two-line segments s_i, s_j intersect in such a way that every pair is tested for once before the algorithm can terminate. Since the time complexity for intersection tests for two-line segments is constant [LaM99], we can conclude that the time complexity of this brute-force algorithm is in $O(n^2)$.

A better approach for finding intersections between line segments may be the so-called *plane sweep* algorithms: these can be visualized with a conceptual, often vertical, *sweep line* that is moved across the plane until all of its objects have been visited. These kinds of algorithms are especially useful to solve problems of algorithmic geometry when they are combined with efficient data structures like self-balancing binary trees.

3.3.2 Shamos-Hoey Algorithm

Michael I. Shamos and *Dan Hoey* achieved a breakthrough in computational complexity of geometric algorithms [BP00] when they introduced the *Shamos-Hoey Algorithm* [SH76] in 1976. This algorithm applies the sweep-line concept to a set of segments that lie on a plane to find the leftmost intersection point.

Let $S = \{s_1, ..., s_n\}$ be a set of n segments. We describe the algorithm according to its original implementation:

1. Initialize an empty self-balancing binary tree (e.g. an *AVL-Tree* or *Red-Black-Tree*) to manage the segments. We refer to this three as the sweep line (SL).
2. Retrieve the points of all segments and sort them in a lexicographic order by their x and y coordinates, such that p_1 is leftmost and p_{2n} is rightmost.
3. Iterate over each point p in the sorted collection from left to right and retrieve its segment s.
   3.1. If p is the left endpoint of s, add s to the sweep line and retrieve the segments a, b above and below of s. Then, test the two pairs (a, s) and (s, b) for intersections.
   3.2. If p is the right endpoint of s, the segment already exists in the sweep line. Retrieve the segments a, b above and below of s and test if the pair (a, b) does intersect or not. If not, no other intersection with s is possible, thus s is removed from the sweep line.

If an intersection is found during the process, the algorithm stops immediately and returns the intersection point, which is the leftmost intersection point [SH76]. If not, the algorithm terminates after it reaches the rightmost point. The resulting pseudo-code is denoted in Algorithm 4, which is a slightly changed version that adapts an implementation of [Sun21].

An edge case we need to consider in the application of this algorithm is that the input may contain duplicate segments; whether such are detected as intersections or not is dependent on the implementation of the tree data structure and the intersect function. While some algorithms don't consider duplicate segments as intersections, other algorithms like the assemblers do so. Nevertheless, this problem can easily be fixed by checking the sorted list of segments for duplicates before applying the line intersection algorithm.

Complexity Analysis

The time complexity for sorting the points for an input of n Segments is in $O(n \log n)$. The subsequent line sweep loops over the sorted collection of points once and tests their segments for intersections, which is done in constant time (Section 3.3.1). This results in a linear complexity for the loop and a combined time complexity in $O(n \log n)$ as well as a space complexity in $O(n)$ for the Shamos-Hoey algorithm, which is a significant speedup compared to the brute-force approach.

Algorithm 4: Shamos-Hoey Algorithm

Input : A set of segments $S = \{s_1, \ldots, s_n\}$
Output: The leftmost intersection point i or $null$ if no two segments s_i, s_j intersect

```
SL := {};
P := getPoints(S).sortXYOrder();
for p ∈ P do
    s := getSegment(p);
    if isLeft(p, s) then
        SL.add(s);
        a = SL.getInorderPredecessor(s);
        b = SL.getInorderSucessor(s);
        if i := intersect(a, s) then
            return i;
        end
        if i := intersect(s, b) then
            return i;
        end
    else
        a = SL.getInorderPredecessor(s);
        b = SL.getInorderSucessor(s);
        if i := intersect(a, b) then
            return i;
        end
        SL.remove(s);
    end
end
return null;
```

3.3.3 Assembling Geometries

As we discussed in section 2.1, areas are either closed ways or relations with outer and inner way members. In this section, we describe an algorithm that creates polygon geometries by assembling ways according to [Wik21g]. This algorithm, in its essence, can be visualized as a game of *Dominoes* [1].

Let $W = \{w_1, \ldots, w_n\}$ be a set of unordered ways. We initialize the polygon $\mathcal{P}$ with the points of the first way w_1. Next, we remove w_1 from W and try to complete $\mathcal{P}$ by using *back-tracking*:

[1] https://en.wikipedia.org/wiki/Dominoes

1. *Initial check*: if $\mathcal{P}$ is already complete, i.e. closed and not self-intersecting, the assembly procedure is finished. The line intersection tests are performed on the edges of $\mathcal{P}$ using the Shamos-Hoey algorithm.
2. *Domino step*: if $\mathcal{P}$ is not complete, find all *candidates* w that could continue the polygon, i.e. $\mathsf{last}(\mathcal{P}) = \mathsf{first}(w)$ or $\mathsf{last}(\mathcal{P}) = \mathsf{last}(w)$. In the latter case, reverse the nodes of w to maintain the correct order of points.
3. *Recursive step*: add the points of the current candidate w to $\mathcal{P}$ and test if $\mathcal{P}$ can be completed with the remaining ways $W \setminus \{w\}$ recursively. If so, return the resulting polygon $\mathcal{P}'$, else proceed with the next candidate.
4. *Back-tracking step*: If no candidates are left, the current set of ways W can't complete the polygon.

Finally, if there are any ways left after $\mathcal{P}'$ was completed, the tail recursion starts a new polygon and repeats the process with the remaining ways W'. If the assembly was unsuccessful, the current way w_1 is considered as invalid and thus is removed from W before starting the next iteration. Therefore, complex polygons, which we consider as invalid, are filtered out implicitly.

We apply this algorithm to the closed ways and multipolygon relations that were extracted from OpenStreetmap to retrieve the actual area geometries. If an area is declared by a closed way w, we apply the assembler with $W = \{w\}$, otherwise the inner and outer way members of the relation are assembled separately and combined into families using the *ring grouping algorithm* described in [Wik21g]. At last, if we assemble territories, we split the created families such that each contained figure becomes its own territory, as territories can only have exactly one outer ring (refer to Section 3.1). If we assemble bonuses, then the geometries stay the same.

3.4 Neighborships

After compressing and assembling the territories, we want to determine neighborships between them.

Let $\mathcal{A}_L$ be the set of all areas with the level L. We consider two areas $a_i, a_j \in \mathcal{A}_L$ to be *adjacent*, if any of their rings share a common node. We formalize this statement by defining the relation $R_L^N \subseteq \mathcal{A}_L \times \mathcal{A}_L$ as the *neighbor relation* over all boundaries with the level L. It is apparent that this relation is an equivalence relation, since it is reflexive, transitive and symmetric.

We use a graph structure to model the neighbor relations. A graph G is a pair (V, E), where V is a set of vertices and E a set of edges. We define $G_L = (V, E)$ with $V = \mathcal{A}_L$ and $E = R_L^N \setminus \{(a, a) \mid a \in \mathcal{A}_L\}$ without self-directed edges as the *neighbor graph* over $\mathcal{A}_L$.

By persisting the way references of each area during the assembly stage, we can determine the neighbors for each area by creating a dictionary that maps each way w to a set of areas $A = \{a_j, \ldots, a_k\}$ that reference it. If a way is referenced by more than one area, the areas $a_j, \ldots, a_k$ are considered to be neighbors. This approach clearly is more efficient than iterating and comparing the individual nodes of each area (or the points of their geometries). Figure 3.4 shows the resulting neighbor graph for a map of the Canary Islands.

Algorithm 5: Neighbor Graph Creation

Input : A set $\mathcal{A}_L$ of all areas with the level L
Output: The undirected neighbor graph $G_L = (V, E)$

```
▷ Build the way reference dictionary
```
$dict = \{\}$;
for $a \in A$ **do**
  for $w \in$ getWays(a) **do**
    $dict$.addReference(w, a);
  end
end

```
▷ Create the neighbor graph
```
for $a_i \in A$ **do**
  V.insert(i);
  for $w \in$ getWays(a_i) **do**
    for $a_j \in dict$.addReference(w) **do**
      E.insert(i, j);
    end
  end
end

3.5 Filters

Map extracts can contain tiny areas that are too small in size to create a territory (cf. map quality requirements in Section 3.1). Consequently, we want to remove such areas from our map by filtering them, which is the focus of this section.

3.5.1 Surface Area

In order to determine the size of an area, we calculate the *surface area* of its geometry:

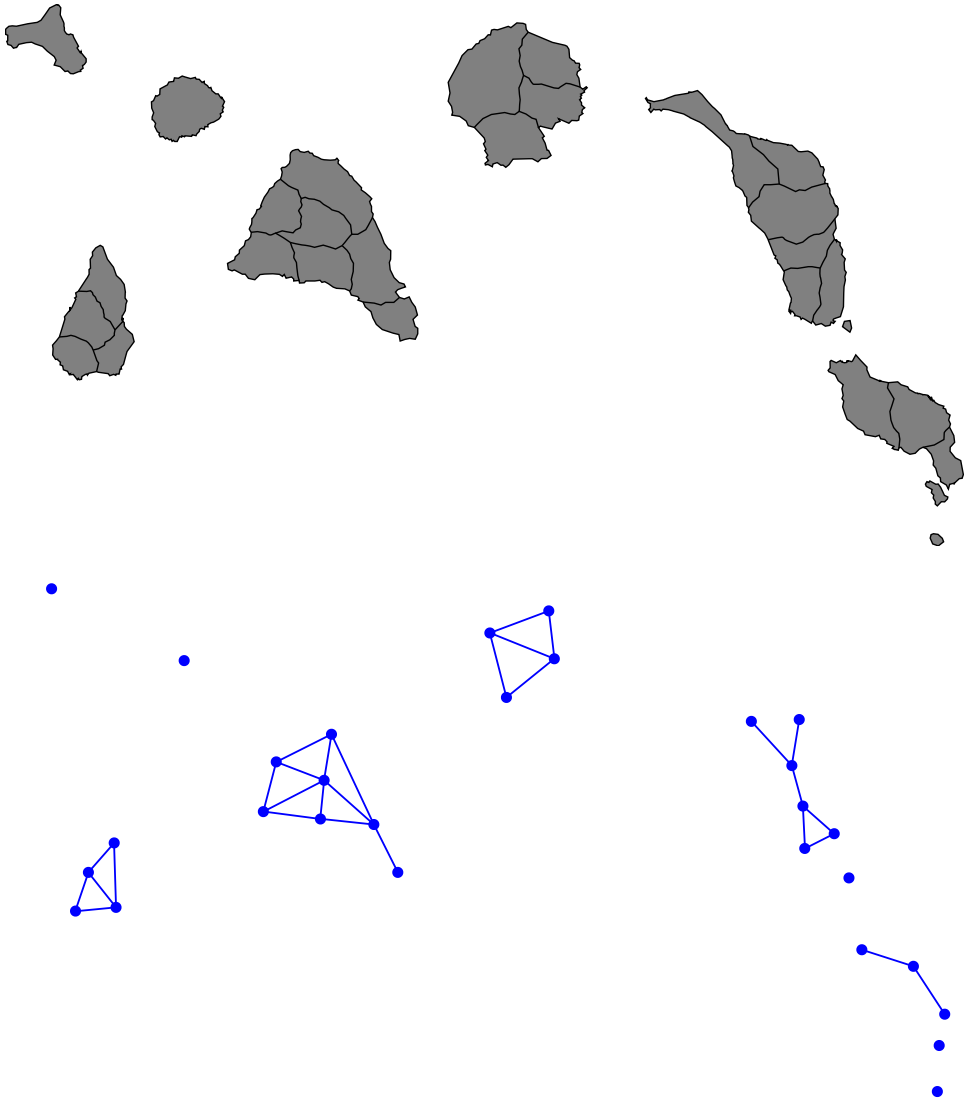


Figure 3.4: A map of the Canary Islands and its neighbor graph

Let $\mathcal{P} = (p_1, \ldots, p_n, p_{n+1})$ be a simple polygon with $n + 1$ points and $p_i = (x_i, y_i)$. Then, the (signed) surface area $A_\mathcal{P}$ of the polygon $\mathcal{P}$ can be calculated with the so-called *shoelace formula* [Bra86]:

$$A_\mathcal{P} = \frac{1}{2} \sum_{i=1}^{n} (x_i y_{i+1} - x_{i+1} y_i) \tag{3.1}$$

To calculate the surface area of a figure $F = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n)$, we simply subtract the sum of the inner surface areas from the outer surface area:

$$A_F = A_{\mathcal{P}_0} - \sum_{i=1}^{n} A_{\mathcal{P}_i} \tag{3.2}$$

3.5.2 Filtering Areas

Now we can move on to the actual filter algorithm: at first, we calculate the total surface area as the sum of the surface areas of each individual figure. Afterwards, we can calculate the relative size of each figure as the quotient of its surface area and the total surface area. We introduce a relative size threshold $\varepsilon \in [0, 1]$ and consider figures to be too small, if their relative size is less than ε. For example, defining $\varepsilon = 0.5$ would remove all figures that are smaller than 50% of the total map area.

Applying this algorithm too all areas comes with one caveat: if an area which is surrounded by other areas is removed due to its size, holes will appear throughout the map. We can correct this problem by changing the filter algorithm to operate the map's islands instead. We can find these islands by finding the *connected components* of the neighbor graph G_t.

3.5.3 Filtering Islands

Let $G = (V, E)$ be an undirected neighbor graph. A *connected component* $C = (V', E')$ is a sub graph of G, such that every vertex $v_i \in V'$ has a path to every other vertex $v_j \in V'$. We simplify this definition and define *components* as the set V'. In order to find all islands of the map, we retrieve the components for our neighbor graph G by applying a depth-first search, which is denoted in algorithm .

After we found the components, we apply the filter described in Algorithm 7 to remove all islands that have a smaller relative size than the specified ε from the map, without creating holes.

3.6 Center Points

One of simplest way to determine the center point of polygons is by taking the average of each point of the polygon. Let $\mathcal{P} = (p_1, \ldots, p_n, p_{n+1})$ be a polygon with $n + 1$ points and $p_i = (x_i, y_i)$. We calculate the center point m by extending the formula for the arithmetic mean to the two-dimensional vector space [Bau19]:

$$m = \frac{1}{n} \sum_{i=1}^{n} (x_i, y_i) = \frac{1}{n} \Big(\sum_{i=1}^{n} x_i, \sum_{i=1}^{n} y_i \Big) \tag{3.3}$$

Algorithm 6: Component Depth-First-Search

Input : An undirected neighor graph $G = (V, E)$
Output: A set of components C

```
C := {};
unvisited := V;
stack := {};
while unvisited ≠ ∅ do
    c := {};
    stack.push(unvisited.pop());
    while stack ≠ ∅ do
        v := stack.pop();
        c.add(v);
        unvisited.remove(v);
        for v_adj ∈ adjacents(E, v_i) do
            if v_adj ∈ unvisited then
                stack.push(v_adj);
            end
        end
    end
    components.add(component);
end
```

Algorithm 7: Component Filter

Input : A set of components C and a threshold $\varepsilon > 0$
Output: The filtered set of components C'

```
totalSurfaceArea := 0;
for c ∈ C do
    totalSurfaceArea := totalSurfaceArea + surfaceArea(c);
end
for c ∈ C do
    relativeArea := surfaceArea(c) / totalSurfaceArea;
    if relativeArea < ε then
        C.remove(c);
    end
end
return C;
```

This formula calculates the center point of a polygon in linear time, which is optimal. However, for polygons that are not regular, the calculated center can differ substantially from its visual center (cf. Fig. 3.5a).

Geometric Center

A better approach for finding the center of a polygon is by calculating its center of mass, which is also referred to as its *centroid*. The general idea is to triangulate the polygon along its sides and to determine its center as the sum of the triangle centers weighed by their respective surface area.

Let $\mathcal{P} = (p_1, \ldots, p_n, p_{n+1})$ be a simple polygon with $n+1$ points. Then, the centroid $c = (x_c, y_c)$ is calculated with the following formulae [Bou97]:

$$x_c = \frac{1}{6A} \sum_{i=1}^{n} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{3.4}$$

$$y_c = \frac{1}{6A} \sum_{i=1}^{n} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i) \tag{3.5}$$

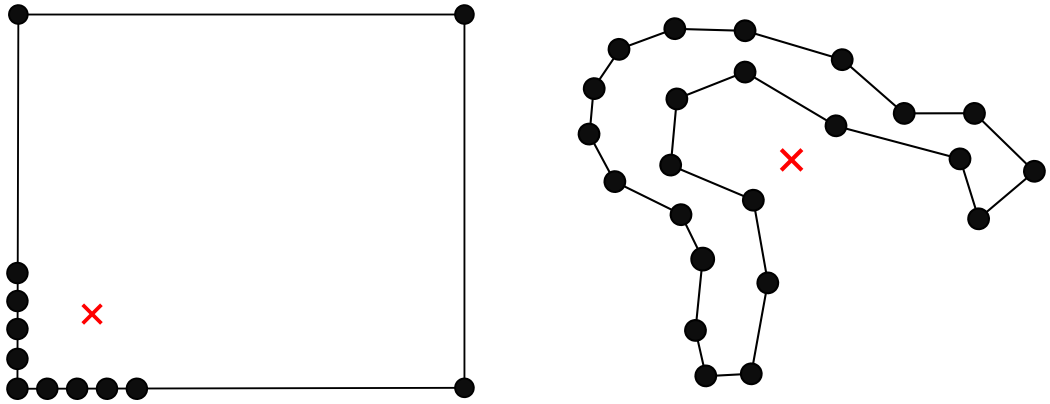
A denotes the polygon's (signed) surface area (cf. Section 3.5.2). For figures $F = (\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_k)$ with one outer polygon $\mathcal{P}_0$ and k inner polygons $\mathcal{P}_1, \ldots, \mathcal{P}_k$, we calculate the centroid m_F as follows [Bau19]:

$$m_F = \frac{\mathcal{P}_0 \cdot A_{\mathcal{P}_0} - \sum_{i=1}^{k} \mathcal{P}_i \cdot A_{\mathcal{P}_i}}{A_{\mathcal{P}_0} - \sum_{i=1}^{k} A_{\mathcal{P}_i}} \tag{3.6}$$

At last, the centroid for a family $\mathcal{F} = (F_1, \ldots, F_m)$ is calculated in the same way as the sum of the weighted figure centroids.

The geometric center formula calculates the centroid of a polygon in linear time and finds a better solution for non-regular polygons. However, for concave polygons, this algorithm does not guarantee that the calculated center actually lies inside of the polygons, which can be seen in Figure 3.5b.

As the counter-examples have shown, the two presented formulaes do not find the optimal center point for arbitrary simple polygons. In fact, there exist several alternatives that may find better center points for our boundaries. However, we ultimately chose to remain with the centroid algorithm due to its linear time complexity and acceptable solutions for most polygons.



(a) A polygon for which the centerpoint by average sum is non-ideal

(b) A concave polygon for which its centroid lies outside

Figure 3.5: Counter-example polygons for average and geometric centers [Bau19]

Figure 3.6: A generated map for the Isle of Man with territory level 8 and bonus level 6

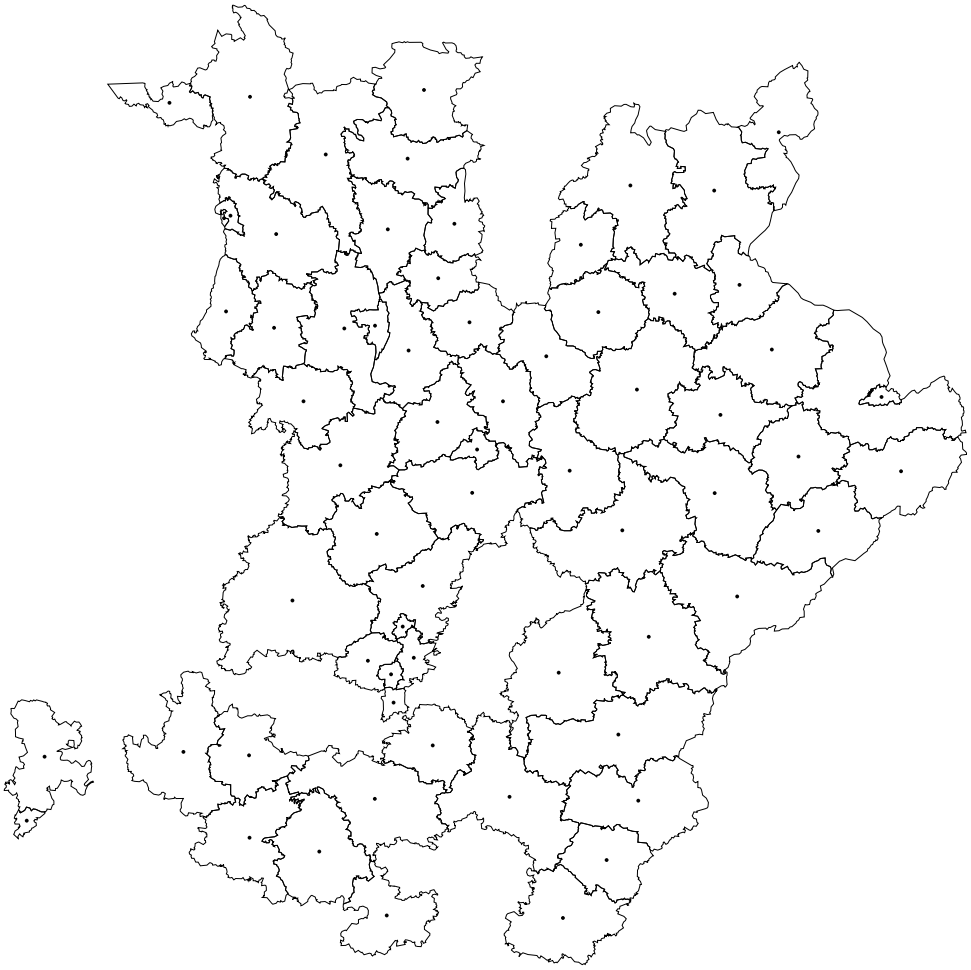


Figure 3.7: Bayern with territory level 8

4 Conclusion

This section concludes our work. As a main purpose thereof, we have taken up the challenge to elaborate a tool that mostly automatizes the map generation process using real cartographical material provided by the freely available geodata of the OpenStreetMap project to be used as a layer for the referenced game, aiming concomitantly a highest possible level of automation as well as a reasonable compression of detailed maps with a high resolution that were generating huge data processing requirements and implicitly significant performance losses. This tool, the MapMaker, resolved in addition the aggregation of adjacent territories for bonuses, boundary and map viewing optimizations, the digital connectivity as well as other targets. In our work, we embedded well-known, helpful projections, such as the Mercator projection, and also integrated algorithms (Shamos-Hoey algorithm, Ramer-Douglas-Peucker compression algorithm) in order to achieve the aforementioned goals. Last but not least, our work has aimed and hopingly succeeded to provide a most user-friendly tool that may be used without any relevant prior knowledge by its users, which also significantly accelerates the computational and resource-intensive process of map creation and metadata computation and automatizes also the uploading process.

We hope that the ideas and proposed solutions included in this paper will generate an impulse for the further use of actual maps and geodata for development and improvement of other popular strategy games, therefore we provide the source code for the MapMaker in a public repository[1] to encourage the further development and maintenance by the community and interested users.

[1] https://github.com/PatrickLindemann/warzone-osm-mapmaker

Bibliography

[Bau19] L. Baur. "Points of Interest - a search for adequate map labelings." University of Stuttgart, 2019. DOI: 10.18419/opus-10620. URL: http://dx.doi.org/10.18419/opus-10620 (cit. on pp. 36, 38, 39).

[Bou97] P. Bourke. *Calculating the area and centroid of a polygon*. 1997. URL: http://paulbourke.net/geometry/polygonmesh/ (cit. on p. 38).

[BP00] J.-D. Boissonnat, F. P. Preparata. "Robust plane sweep for intersecting segments." In: *SIAM Journal on Computing* 29.5 (2000), pp. 1401–1421 (cit. on p. 30).

[Bra86] B. Braden. "The Surveyor's Area Formula." In: *The College Mathematics Journal* 17.4 (1986), pp. 326–337. DOI: 10.1080/07468342.1986.11972974. URL: https://doi.org/10.1080/07468342.1986.11972974 (cit. on p. 35).

[Bro63] P. Brothers. *Risk vs Diplomacy*. 1963. URL: https://www.hasbro.com/common/instruct/Risk1963.PDF (cit. on p. 8).

[Com05] W. Commons. *A simple figure explaining latitude and longitude*. File: $\texttt{FedStats}_{L}at_{l}ong.svg$. 2005. URL: https://de.wikipedia.org/wiki/Datei:FedStats%5C_Lat%5C_long.svg (cit. on p. 17).

[Com20] W. Commons. *File:Douglas Peucker.png — Wikimedia Commons, the free media repository*. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Douglas%5C_Peucker.png&oldid=486144100 (cit. on p. 26).

[D0015] D00659. *A guide to coordinate systems in Great Britain*. Version 2.3. 2015 (cit. on p. 17).

[DP73] D. H. Douglas, T. K. Peucker. "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature." In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10 (1973), pp. 112–122 (cit. on p. 25).

[Gor13] D. Gordon. *Risk vs Diplomacy*. 2013. URL: https://www.cardboardrepublic.com/classics/risk-vs-diplomacy (cit. on p. 8).

[Hoa62] C. A. R. Hoare. "Quicksort." In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10. eprint: https://academic.oup.com/comjnl/article-pdf/5/1/10/1111445/050010.pdf. URL: https://doi.org/10.1093/comjnl/5.1.10 (cit. on p. 27).

[LaM99] A. LaMothe. *Tricks of the Windows Game Programming Gurus*. USA: Sams, 1999. ISBN: 0672313618 (cit. on p. 30).

[Pro21a] O. Project. *About — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/wiki/About_OpenStreetMap (cit. on p. 9).

[Pro21b] O. Project. *OpenStreetMap Deutschland: Die freie Wiki-Weltkarte*. 2021. URL: https://www.openstreetmap.de/ (cit. on p. 9).

[Pro21c] O. Project. *Statistics — OpenStreetMap*. 2021. URL: https://www.openstreetmap.org/stats/data_stats.html (cit. on pp. 9, 12).

[Ram72] U. Ramer. "An iterative procedure for the polygonal approximation of plane curves." In: *Computer Graphics and Image Processing* 1.3 (1972), pp. 244–256. ISSN: 0146-664X. DOI: https://doi.org/10.1016/S0146-664X(72)80017-0. URL: https://www.sciencedirect.com/science/article/pii/S0146664X72800170 (cit. on p. 25).

[SH76] M. I. Shamos, D. Hoey. "Geometric intersection problems." In: *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*. 1976, pp. 208–215. DOI: 10.1109/SFCS.1976.16 (cit. on pp. 30, 31).

[Sny72] J. P. Snyder. *Map projections: A working manual*. 1395. U.S. Government Printing Office, 1972. DOI: https://doi.org/10.3133/pp1395. URL: https://pubs.er.usgs.gov/publication/pp1395 (cit. on pp. 17, 18).

[Str11] D. R. Strebe. *Mercator projection Square*. File: $\texttt{Mercator}_{p}projection_{S}quare.jpg$. 2011. URL: https://commons.wikimedia.org/wiki/File:Mercator%5C_projection%5C_Square.JPG#filelinks (cit. on p. 20).

[Sun21] D. Sunday. *Practical Geometry Algorithms: with C++ Code*. Amazon KDP, 2021. ISBN: 9798749449730 (cit. on p. 31).

[Top] J. Topf. *Libosmium Manual*. URL: https://osmcode.org/libosmium/manual.html (cit. on p. 14).

[War11] Warzone. *Generate your connections automatically, too! - Warzone Forums*. 2011. URL: https://www.warzone.com/Forum/2668-generate-connections-automatically- (cit. on p. 11).

[War21] Warzone. *About - Warzone - Better than Hasbro's RISK® game - Play Online Free*. 2021. URL: https://www.warzone.com/About (cit. on p. 9).

[Wik21a] O. Wiki. *Area — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Area&oldid=2194500 (cit. on p. 14).

[Wik21b] O. Wiki. *Downloading data — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Downloading_data&oldid=2183449 (cit. on p. 12).

[Wik21c] O. Wiki. *Elements — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Elements&oldid=2157322 (cit. on pp. 12, 13).

[Wik21d] O. Wiki. *Key:Boundary — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Key:boundary&oldid=2091920 (cit. on p. 16).

[Wik21e] O. Wiki. *Node — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Node&oldid=2141656 (cit. on p. 13).

[Wik21f] O. Wiki. *PBF Format — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=PBF_Format&oldid=2183349 (cit. on p. 12).

[Wik21g] O. Wiki. *Relation:multipolygon/Algorithm — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Relation:multipolygon/Algorithm&oldid=2138462 (cit. on pp. 32, 33).

[Wik21h] O. Wiki. *Types of relation — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Types_of_relation&oldid=2152781 (cit. on p. 14).

[Wik21i] O. Wiki. *Way — OpenStreetMap Wiki*. 2021. URL: https://wiki.openstreetmap.org/w/index.php?title=Way&oldid=2173770 (cit. on p. 13).

[Wik21j] W. Wiki. *Combat Basics — Warzone Wiki*. 2021. URL: https://www.warzone.com/wiki/Combat_Basics (cit. on p. 10).

[Wik21k] W. Wiki. *Gameplay Basics — Warzone Wiki*. 2021. URL: https://www.warzone.com/wiki/Gameplay_Basics (cit. on p. 10).

[Wik21l] W. Wiki. *Map Requirements — Warzone Wiki*. 2021. URL: https://www.warzone.com/wiki/Map_requirements (cit. on pp. 10, 24).

[Wik21m] W. Wiki. *Set Map Details API — Warzone Wiki*. 2021. URL: https://www.warzone.com/wiki/Set_map_details_API (cit. on p. 11).

All links were last followed on November 3, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature