Die approbierte Originalversion dieser Diplom-/Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Visualization of Computer-Generated 3D Cities using GIS Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics and Visual Computing

by

BSc. Luca Maestri
Registration Number 0926939

to the Faculty of Informatics
at the TU Wien

Advisor: Ao.Univ.Prof. Mag. Dr. Horst Eidenberger

Vienna, 22nd September, 2017

Luca Maestri
Horst Eidenberger

Declaration of Authorship

BSc. Luca Maestri
Stoebergasse 17/23, 1050 Wien

I hereby declare that I have written this Master Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 22nd September, 2017

Luca Maestri

Acknowledgements

I would first like to thank my thesis advisor Prof. Dr. Horst Eidenberger, without whom this work would not have been possible. Prof. Eidenberger always endeavored to steer me in the right direction over the past years, by lending me books, giving me feedback and spurring me to constantly improve my work. I would also like to extend my appreciation to my colleagues and friends Juri Berlanda and Tobias Froihofer, with all of whom I spent countless hours in the laboratory while working on a common project. Their continuous encouragement and valuable insight during that time helped me to come up with the idea of this work. Many thanks also go to my dear friend Zeno Casellato who helped me to further refine my writing during multiple everlasting feedback sessions. Finally I would like to express my gratitude towards my family. Without their support and teachings this whole undertaking would not have been possible in the first place.

Abstract

The constant performance increase of algorithms and hardware over the last decades enabled new ways to collect data, which would have been unthinkable just a few years ago. This is especially true in the sector of geodesy. The release of gps-tracking smartphones enabled users from all over the world to easily collect and upload georeferenced data. Additionally governments also started to make their georeferenced data available to everyone. Through these phenomenons countless databases containing georeferenced information appeared on the Internet. By accessing these databases numerous new applications can be implemented. This thesis focuses on the creation of three-dimensional models that can be easily integrated in a virtual reality environment. The practical part of this thesis consists of four steps. The first step is the data acquisition.As mentioned before nowadays there are various eligible data sources for such a project, however in this work all the data is fetched from a public database of the Austrian government. This database has been chosen because it already contains all required buildings' footprints and heights. In the second step the acquired data is analysed and pre-processed using Matlab. By using the filters implemented in Matlab artefacts resulting from the noise contained in the data can be removed. In the third step a suite capable of combining the data-sets is presented. Quantum GIS offers a complete open source suite capable of combining, displaying, processing and exporting georeferenced data. This tool contains solutions for all the problems proposed during this step of the project. The final step is the implementation of the web-application, which creates the three-dimensional models by importing the files generated during the previous step. This web-application has been implemented using WebGL so that most of the calculations are done on the client's graphics card. The three-dimensional models have been compared with the models offered by the Austrian government for the sake of showing that the presented framework is capable of producing similar models at a lower performance cost in a virtual reality environment. Practically, the presented framework has been implemented and its results have been tested during the course of another project, in which the city models were used in order to create a skydiving experience over the city of Vienna in a virtual reality environment. Over the course of the mentioned project the models were found satisfactory by the users.

Kurzfassung

Die ständige Leistungssteigerung von Algorithmen und Hardware in den letzten Jahrzehnten ermöglichte neue Wege, Daten zu sammeln, die vor wenigen Jahren undenkbar gewesen wären. Dies gilt insbesondere im Bereich der Geodäsie. Die Freigabe von GPS-Tracking-Smartphones ermöglichte es Benutzern aus der ganzen Welt, georeferenzierte Daten zu sammeln und hochzuladen. Darüber hinaus begannen die Regierungen, ihre georeferenzierten Daten öffentlich zugänglich zu machen. Durch diese Entwicklungen erschienen vielfältige Datenbanken mit georeferenzierten Informationen im Internet. Durch den Zugriff auf diese Datenbanken können zahlreiche neue Applikationen implementiert werden. Diese Masterarbeit konzentriert sich auf die Erzeugung von dreidimensionalen Modellen, die sich leicht in beliebige Game-Engines integrieren lassen. Der praktische Teil dieser Arbeit besteht aus vier Schritten. Der erste Schritt beinhaltet die Datenerfassung. Wie bereits erwähnt, gibt es für ein solches Projekt verschiedene Datenquellen, aber in dieser Arbeit werden alle Daten aus einer öffentlichen Datenbank der österreichischen Regierung abgerufen. Diese Datenbank wurde gewählt, weil sie bereits alle Grundrisse und Höhen der Gebäude Wiens enthält. Im zweiten Schritt werden die erfassten Daten mit Matlab analysiert und vorverarbeitet. Durch die Verwendung von Filtern, die in Matlab implementiert sind, können Artefakte eliminiert werden, die sich aus dem in den Daten enthaltenen Rauschen ergeben. Im dritten Schritt wird eine Applikation vorgestellt, die die Datensätze kombinieren kann. Quantum GIS bietet eine komplette Open-Source-Applikation, die in der Lage ist, georeferenzierte Daten zu kombinieren, anzuzeigen, zu verarbeiten und zu exportieren. Dieses Werkzeug enthält Lösungen für alle Probleme, die während dieser Projektphase beschrieben wurden. Der letzte Schritt ist die Implementierung der Web-Applikation, die die dreidimensionalen Modelle durch den Import der im vorherigen Schritt erzeugten Dateien erzeugt. Diese Web-Applikation wurde mit WebGL implementiert, so dass die meisten Berechnungen auf der Grafikkarte des Clients durchgeführt werden. Die dreidimensionalen Modelle wurden mit den Modellen der österreichischen Regierung verglichen, um zu zeigen, dass das präsentierte Framework in der Lage ist, ähnliche Modelle kostengünstig in einer beliebigen Game-Engine für Virtual Reality zu produzieren. Tatsächlich wurden die Ergebnisse dieser Arbeit im Rahmen eines weiteren Projektes getestet. Dabei wurden die Stadtmodelle eingesetzt, um ein Skydiving-Erlebnis über die Stadt Wien in virtueller Realität zu schaffen. Die Benutzer dieses Projekts fanden die vorgestellten dreidimensionalen Modelle der Stadt sehr zufriedenstellend.

Contents

Abstract **vii**

Kurzfassung **ix**

Contents **xi**

1 Introduction **1**
1.1 Context 1
1.2 Problem 2
1.3 Roadmap 3

2 Background **7**
2.1 Geographic Information Systems 7
2.2 Digital Elevation Model 20
2.3 Geometry Modelling 25
2.4 Polygon clipping 28
2.5 Extrusion 32
2.6 Web Graphics Library 34

3 Methodology **37**
3.1 Problem Description 37
3.2 System Requirements 38
3.3 Available Data 38

4 Results **51**
4.1 City Web Extruder 51
4.2 Wall Texturing 53
4.3 Game Engine Integration 55

5 Conclusion **57**

List of Figures **59**

List of Tables **60**

Acronyms **61**

Bibliography **63**

CHAPTER 1

Introduction

Over the past decades the evolution of sensing technologies, as well as automation- and database-systems created the possibility to easily accumulate and store huge amounts of geographical information. "Geographical information" refers to any kind of data associated with a geographic location on the earth's surface [Law13]. However this information is often not easy to use, because it requires knowledge from different disciplines in order to access, interpret and elaborate it. The sole data acquisition process can already apply knowledge from the fields photogrammetry, computer vision and geodesy [WL99]. Additionally the acquired data is stored in a database, either local or remote [Wes10], which requires knowledge of IT systems. Depending on the complexity of the systems using georeferenced data, knowledge of even more disciplines may be needed, making it a non-trivial task.

In my work I elaborated a framework that eases the access to the required data for the creation of 3D city models, which are optimized for the integration into game engines. This is achieved by automating most processes and leaving the user the only problem of choosing the buildings he or she wants to export into different 3D file formats of his choosing as seen in Figure 1.1.

1.1 Context

As mentioned before working with georeferenced information is a task covered by many disciplines. This work addresses the problem from the perspective of media-informatics and results in a framework which creates 3D models that can easily be integrated into any game engine. During interviews with urban planners, architects and civil engineers we realized that the 3D rendering of highly detailed city patches is a common task in order to create 3D simulations for their projects. [ZSP] presents a framework to create 3D city models using georeferenced data and discusses further applications of the resulting models. For example in Figure 1.2 the created 3D model is used to simulate efficient

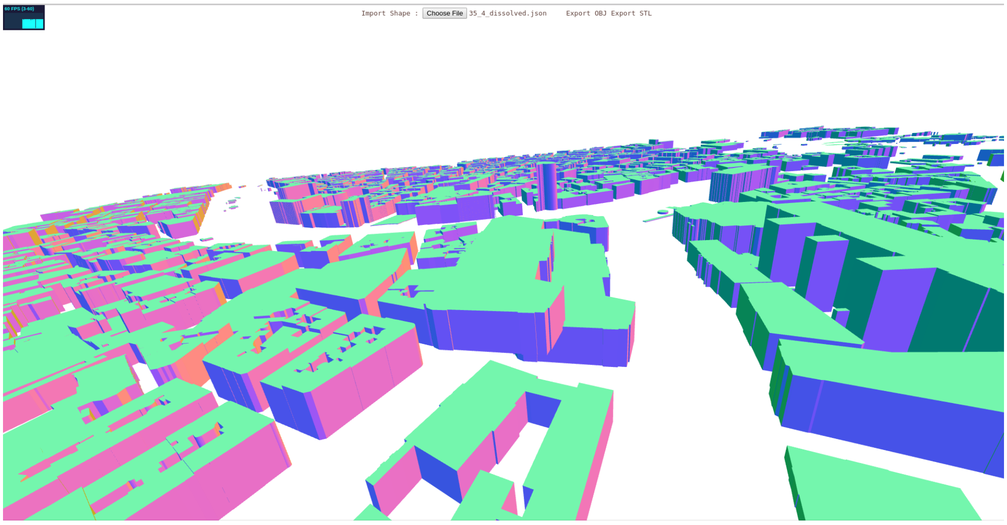


Figure 1.1: A patch of Vienna rendered in the browser using the presented framework.

lighting in an open space. Other applications mentioned in [ZSP] are flooding simulations or visualizations of how planned buildings would integrate in a specific area of the real world. However most of the researched solutions were either licensed, or resulted into models that were not adequate for our virtual reality context which will be described in Section 1.2. Hence this work will explain in detail which problems were encountered, how they were solved and presents the results in form of a framework which can be used to easily create 3D models of any chosen part of Vienna.

Figure 1.2: 3D model used for a light simulation as seen in [ZSP].

1.2 Problem

During a previous project [EM15] the idea was proposed to make use of the most recent advances in virtual reality in order to build a simulator with the specific task of letting people experience the thrill of the free fall. The most evident activity associated with this concept was a skydiving experience over Vienna. In collaboration with experts which created the hardware of the simulator, our task was focused on creating a credible and immersive 3D environment. The users would enjoy the virtual world through a head mounted display (HMI), allowing them the freedom of looking, moving around as well as interacting with the virtual world. Initially there were two suggestions we took in account to create the desired experience. The first proposal that came up was to use 360 degree videos of a skydiving experience to create the virtual reality environment. However we decided to go with the second proposal, which was the use of geo-referenced data for the creation of a 3D city model. We choose the second proposal over the first because we realized that a versatile framework for the creation of the virtual environment could be used to create many different flight experiences over Vienna, thus the additional effort would pay off in the long run.

To overcome the posed challenge we had to get ourselves immersed in different, to us mostly unknown disciplines. Therefore I started to analyse openly acquirable georeferenced data from different sources. During this research I rapidly realized that getting a hold of and combining the data from all the sources that were found would result in exactly the models we were looking for. Hence a big part of this work is focused on the theories connecting all the data sources, which are the coordinate systems geo-referencing the data and their projections. Those topics will be covered in Section 2.1, giving the reader a first overview of the utilized data structures. This thesis conveys most of the information necessary to understand georeferenced systems. Additionally I apply the presented theories and present the resulting framework used to model 3D environments of Vienna.

1.3 Roadmap

Large parts of this work will cover the theory behind geographic information systems (GIS) and the data they encapsulate. The leitmotif of GIS are coordinates, coordinate systems and projections which I am going to describe in detail in the next chapter. The mentioned topics are important because the georeferenced data, as the name already implies, are addressed by coordinates. I will first present the researched theories then continue with the description of my framework. As the reader progresses trough the next chapter of this work different concepts from the fields of geodesy and photogrammetry will be explained. Through this I want to clarify that coordinates describing the world's surface are not as trivial as could be assumed. In Section 2.1.2 I will show that longitude and latitude values are nearly meaningless if not combined with coordinate and projection systems. This becomes especially important when the coordinates are used for complex calculations. Thereby not only the results which will be presented in Chapter 4, but also the necessity of simplification and automation trough the presented framework should become easily comprehensible to the reader.

After covering the basics of geodesy and photogrammetry in Section 2.1, the reader will eventually reach Section 2.4, in which the basics of vector graphics and simple boolean operations with polygons will be discussed. These theories are important in order to create the building footprints as seen in Figure 1.3. In this second part of the work I will start to pivot away from geodesy and move towards 3D modelling (Section 2.3). In this section I will discuss the reasons for the rejection of existing frameworks. Thereby I will clarify how the existing models and the frameworks I analysed neglected the creation of clean footprints which later resulted in inconsistent geometry. This turned out to be problematic when using these models in game engines causing various glitches in the final visualization. Applying these rather simple concepts, that will be discussed in the second chapter, not only avoids the glitches, but also drastically reduces the resulting models' file size.

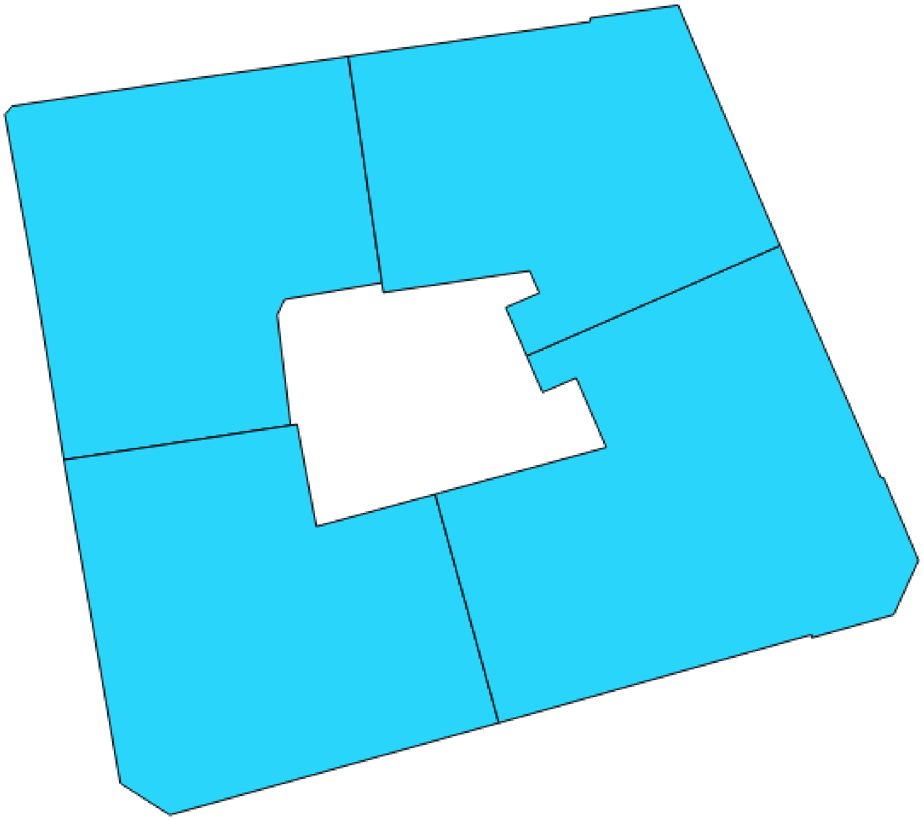


Figure 1.3: A simple buildings footprint created using GIS data.

The third and last part of this work will go over 3D geometry, polygon extrusion, texturing and Web Graphics Library (WebGL) which covers the last part of the frameworks pipeline. I decided to use WebGL because it makes the framework platform-independent and easily accessible as a web application [Cab]. In addition to increasing flexibility WebGL is a relatively new technology which enables browsers to use the client-sided graphics card's power to perform calculations. These are the main advantages motivating the use of WebGL.

Summarizing this work will give an overview of different disciplines needed to understand GIS. Understanding the described basics is necessary in order to accumulate, access and work with the data which is used by the framework. The presented framework enables the user to automatically create 3D models of buildings of the user's choosing. Finally the reader will learn that the framework is only making use of of a small part of the data's potential, and that there is much space for improvement and further development.

CHAPTER 2

Background

In this chapter I will start with a short historical overview of Geographic Information Systems (GIS). Section 2.1 will help the reader get a better grasp of GIS and clarify its state of the art. After this introduction from Section 2.1.2 to Section 2.1.8 I will discuss the basics of photogrammetry. Thenceforth I will explain the theories of the applied solutions for the presented framework from Section 2.2 to Section 2.6. Hence by reading this chapter the reader will understand the concepts applied for the implementation of the presented framework.

2.1 Geographic Information Systems

GIS refers to any kind of information that is bound to geo-spatial coordinates, such as longitude and latitude. The first documented approach of overlaying maps with additional subject information was done by John Snow, an English physician, in 1855. During his attempts to find the cause for the cholera outbreak of 1854 in London, Snow came up with the idea of marking the single cases on a map as seen in Figure 2.1. His idea expanded Charles Picquet's work. In 1834 Charles Picquet created gray shaded maps of Paris, to document the deaths by cholera during the outbreak of 1832. Opposed to Picquets earlier work John Snow's overlays were not only used for documentation. His overlays established the connection between the outbreaks and the bad hygiene of the water supplies in the city. This kind of map overlay was never seen before, as it was used to visualize information and analyse clustered data. Since then geographic information has come a long way.

2.1.1 GIS Roadmap

Nowadays we have many examples and different approaches to geographic information. Simple examples in our daily lives are addresses, buildings, heights, demographic information, landmark types, or borders. It is clear that the usage of georeferenced data has almost no limits and that the term can be applied to various systems.

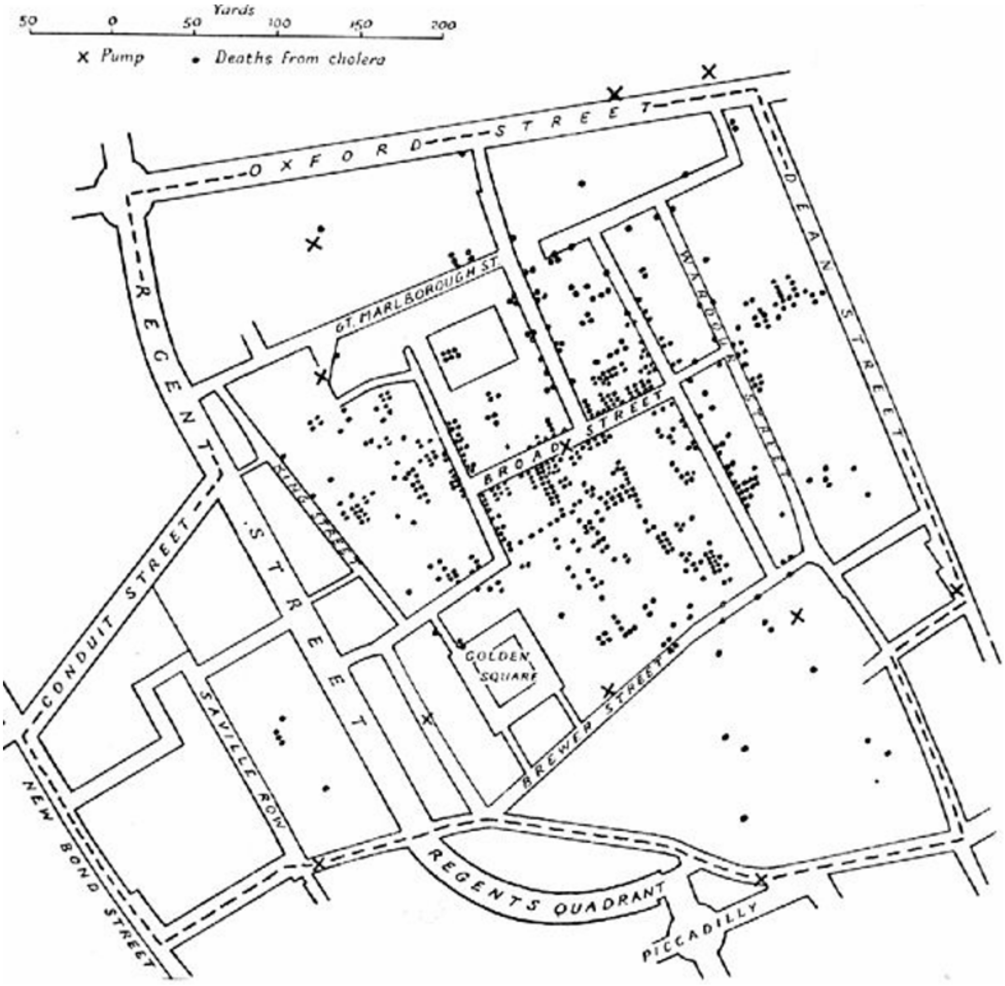


Figure 2.1: John Snow's map overlay.

The father of geographic information systems as we know them today was Roger Tomlinson, an English architect and geographer. As he was tasked by the Canadian government to develop a methodology to evaluate a huge number of maps he came up with different approaches. Eventually Tomlinson recognized that the data to process was bound to grow even more over time. So he decided to look at a computer-based approach which significantly reduced the time and costs of the data analysis [Tom74]. Over the following years his idea evolved into the Canadian geographical information system, the first of its kind.

Today the subject information bound to a specific location can often result in many millions of data points. These vast possibilities originate from the development of newer, more flexible and cheaper sensor systems, such as the Global Position Systems (GPS) which nowadays are incorporated in most smartphones. This constant evolution over the last decades made the acquisition of georeferenced data easier for firms, but most importantly accessible to the public. An example of such a user-created result using just the smartphone is seen in Figure 2.2. This change resulted in a drastic growth of GIS information. During the second half of the 20th century experts, complex programs as well as huge computers were needed to create even simple map overlays. Google was one of the first to make complex GIS data accessible to everyone through simple interfaces using Google Maps and Google Earth in 2005 [Wes10]. Collaborative projects like Open Street Map (OSM), which was started 2004 by Steve Coast, are mostly supported by open communities, accessible for anyone [HW08]. Still, the access being free does not mean that the data is easy to access. The storage and provision of this huge amount of information is mostly achieved by using dedicated databases [Law13]. Without the strategic use of IT systems and the definition of standards it would be almost impossible to get a hold of the data and to use it. For example institutions like the Open Geospatial Consortium (OGC) or the Environmental Systems Research Institute (ESRI) helped developing GIS by defining such standards.

After this short historical overview over GIS the reader should have noted why and how it came to such a huge amount of information stored on the Internet. Furthermore I mentioned OGC and ESRI which are important references because of their defined standards and state of the art tools they present to work with GIS data (e.g. ESRI's ArchGIS or its open source alternative QuantumGIS).

2.1.2 Projections

In this subsection I am going to describe the basics of geodesy and photogrammetry. The first and arguably most important step to use georeferenced data is to understand that the world is not flat. Nowadays this is not a daring statement anymore, you may think, but most of the media used to visualize the world's surface in fact are flat. Notice how printed world maps represent a spherical object on a flat surface. Such representations are achieved through projections. Thereby the necessity of such projections for GIS should slowly become apparent. Note that knowing the coordinates without knowing their projection is useless. This becomes especially important when using coordinates for calculations (e.g. distance between two points) [Sny87].

In our context projections are mostly subdivided into cylindrical, conical and azimuthal projections. The cylindrical projections produce a visualization of the world's surface on a rectangular plane. This is the result of unwrapping the cylinder on which the surface has been projected as we can see in Figure 2.3. This kind of projection is the most seen on world maps. However multiple projection systems are necessary because each one of them deforms the Earth's surface when seen on the flat medium. In the case of the cylindrical projection we get the biggest deformation over the poles. Opposed to this example a polar aligned conic projection would result in a less stretched visualization of the poles as seen in Figure 2.5.

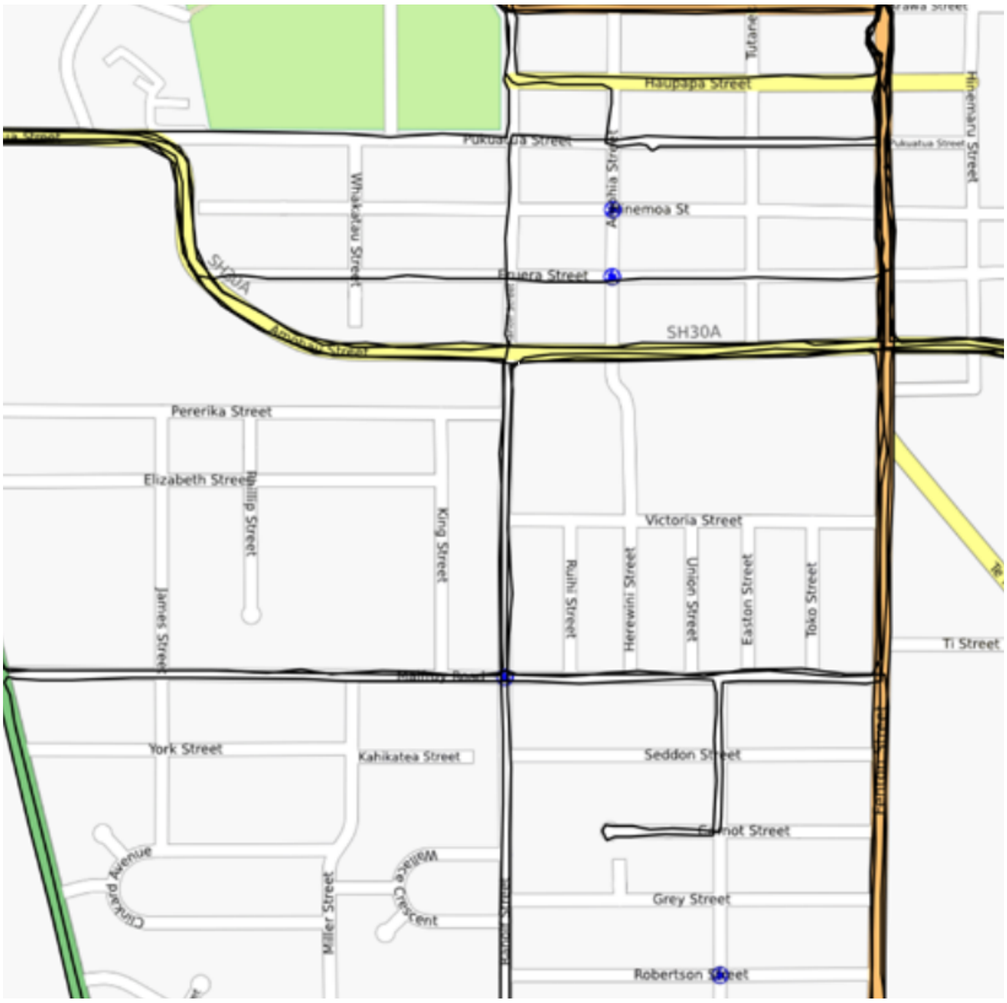


Figure 2.2: GPS data overlaid on a simple map.

Such a projection is mostly used to visualize wide and flat landmarks (e.g. Russia). The third notable projection is the azimuthal projection. Azimuth is a term that comes from the astronomy and it defines the horizontal angle of a position in a spherical coordinate system. The equidistant polar aligned azimuthal projection's biggest advantage is that all points on the result are represented at the right azimuth angle. Thus its name azimuthal projection. It is easy to see how reducing a three-dimensional visualization to two dimensions by using projections will yield deformed results. Note that it is impossible to use a unique projection for the whole world's surface without any drawbacks [Sny87].

In order to minimize deformation errors the World Transverse Mercator (UTM), classifies the world's surface into sixty different sectors as seen in Figure 2.4. This makes working with georeferenced data difficult. In order to create undeformed maps each sector has a specific projection system assigned to it, which best suits the projected area. To know

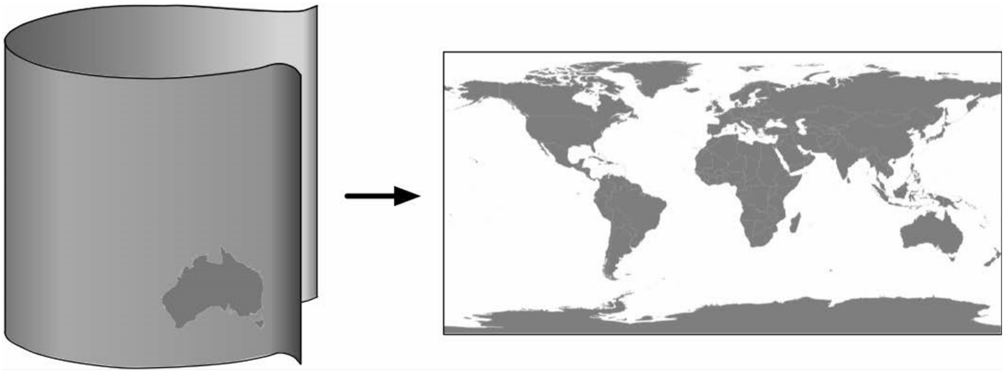


Figure 2.3: Unwrapping a cylindrical projection [Law13].

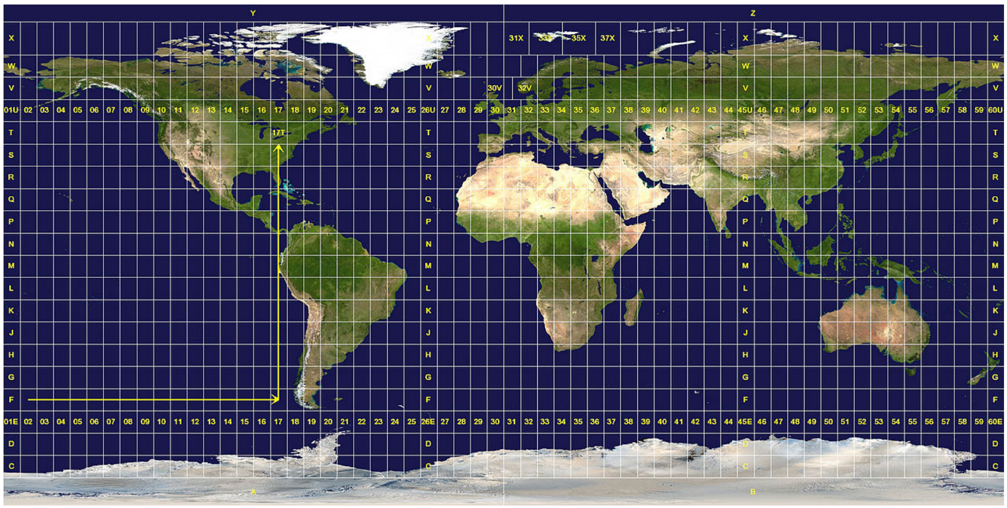


Figure 2.4: The UTM raster from [Dom].

which projection has been applied to the coordinates is important when working with projected coordinates. If we take Figure 2.6 we would use unprojected coordinates to address a point on the sphere's surface, but to address a point on the map (representing the sphere's surface after the projection) we would have to use projected coordinates. Considering that the projection deforms the Earth's surface, the distances between points obviously are also affected by this deformation. Therefore in order to evaluate georeferenced data with calculations, such as distance between two points on a map, the right projection system has to be known.

Opposed to projected coordinates we have unprojected coordinates. The obvious approach to locate a point on the world's surface would be the spherical coordinate system. In contrast to projected coordinates we can use the same unprojected coordinate system for the whole world.

In the last Section 2.1.2 projection systems were explained as well as why georeferenced

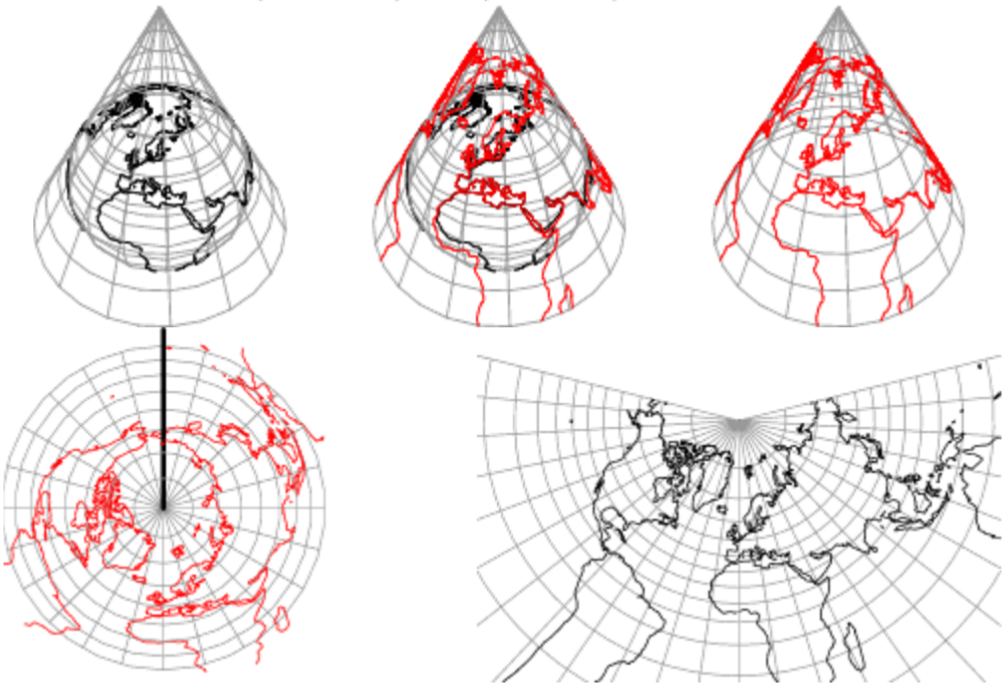


Figure 2.5: Unwrapping a conic projection [Sny87].

data makes use of many different projections and why it is important to know the projection system when working with projected coordinates. Now we may think that working with unprojected coordinates would solve all problems as long as we do not have to visualize our results on a flat surface. In Section 2.1.3 I will show that even unprojected coordinates are not easy to handle. This comes from the fact that the world is an oblate sphere, rather than a perfect sphere. This makes comparing coordinates complicated, even for unprojected coordinates. The coordinate system has to be mathematically adapted to best describe the Earth's shape. The world's surface has also to be taken into account, because just describing the shape neglects the differences in elevation given by the different terrains. Think about how an ellipsoid describing the world's shape can not take the elevation difference between the highest point of the Mount Everest (7.5 km over sea level) and the lowest point of the Mariana Trench into account(11 km under sea level).

By using different mathematical models and reference points, multiple coordinate systems have come to light, over the years. These different models are called datums. The World Geodetic System (WGS), describes the current standard datum, addressed as WGS84. The WGS84 datum is based on a projection system describing the world's shape (oblate spheroid) as seen in Figure 2.7, a surface describing the geoid and a set of fixed reference points. Its most notable use is for GPS data [Law13].

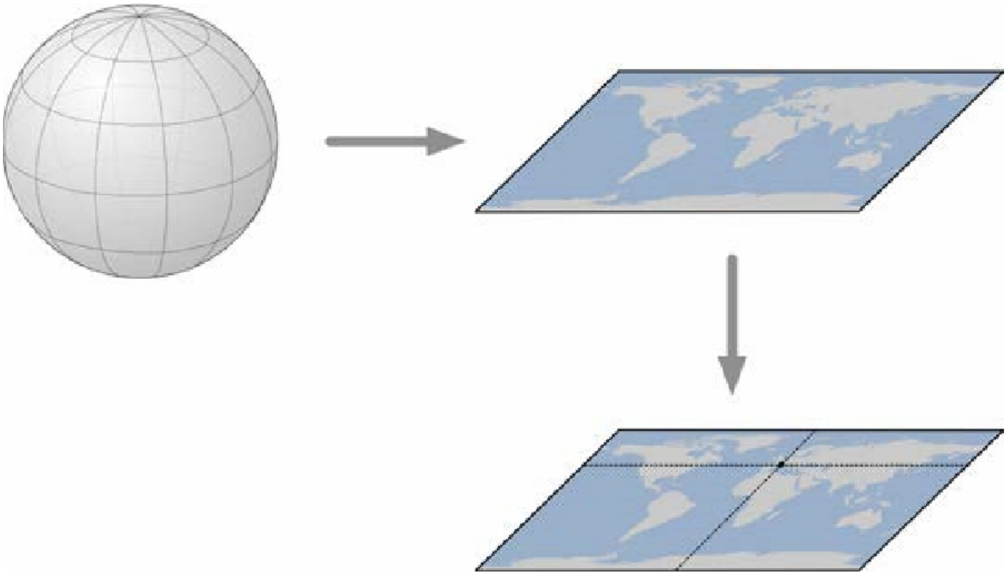


Figure 2.6: From unprojected to projected coordinates [Law13].

2.1.3 Data acquisition

At this point we should understand how the georeferenced data is addressed using coordinates, but we only presented a small overview of the lowest layer of a GIS. The coordinates alone would be meaningless for this work if not associated with additional information. As mentioned before a georeferenced data point can be associated with various information like country outlines or landmark elevation. Populating a data point with information is not as laborious a task as it used to be.

Given the advancements in sensor technology over the last decades most of the work is done remotely. Remote sensing defines the process of acquiring data of an object without having to physically interact with the object itself. Remote sensing has a long history in the context of geographical data acquisition before becoming what we know today. The oldest example of remote sensing in our context is aerial photography. Gaspard-Félix Tournachon, also known as Nadar, took the first documented aerial photo from a hot air balloon over Paris in 1858. However it was not until the Cold War that aerial photography started to flourish. During that time the Americans started to take photos of Russian landmarks for military purposes using the U2 plane, which flew out of range of the anti-aircraft weapons. Opposed to the American U2 the Russians flew even higher launching the Sputnik 1 into orbit [Law13]. Obviously the higher the sensor flies the bigger the areas it can cover.

Taking this into account we can see that the start of the space race had a big impact upon the evolution of remote sensing for geographical purposes. Nowadays depending on the needed resolution data can either be collected with planes or with satellites.

The remote sensing method used for the data I utilized in this project is called Airborne

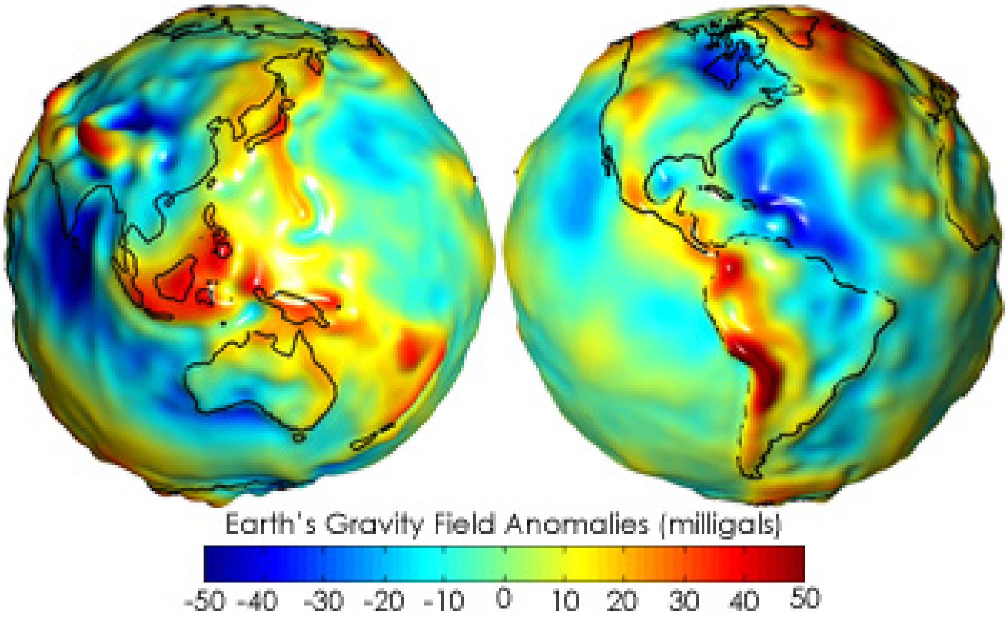


Figure 2.7: Geoid describing the Earth's surface from [NASa].

Laser Scan (ALS). In Figure 2.8 a simple sketch describing how ALS basically works can be seen. On an urban scale the ALS achieves a great area coverage by flying over landmarks in a height between 450 and 500 meters and by using high precision laser scanning to create georeferenced point clouds. In this case the concentration of the used data equals fifteen to twenty points per squared meter.

2.1.4 Raster data

If we interpret the point cloud as raster data we could visualize it as an image. To achieve this we would have to map each height value on the raster to a gray value, resulting in a gray scale image. ALS is more complex than it seems because in addition to the projection and coordinate system transformation, the raw data from the laser scan has to be adjusted to the perspective of the camera and the orientation of the plane. Luckily this first technical preparation, which was the adjustment of the camera and the plane orientation, was already done during the production of the ALS. Raster data that contains elevation information about landmarks is called a Digital Elevation Model (DEM) which is further described in Section 2.2. An example for the visualization of a DEM can be seen in Figure 2.9.

2.1.5 Vector data

Opposed to raster data which is often visualized as an image vector data is represented by geographical coordinates. These coordinates are used to describe positions, outlines, or shapes of features on the Earth's surface. To see the limitations of raster data compared

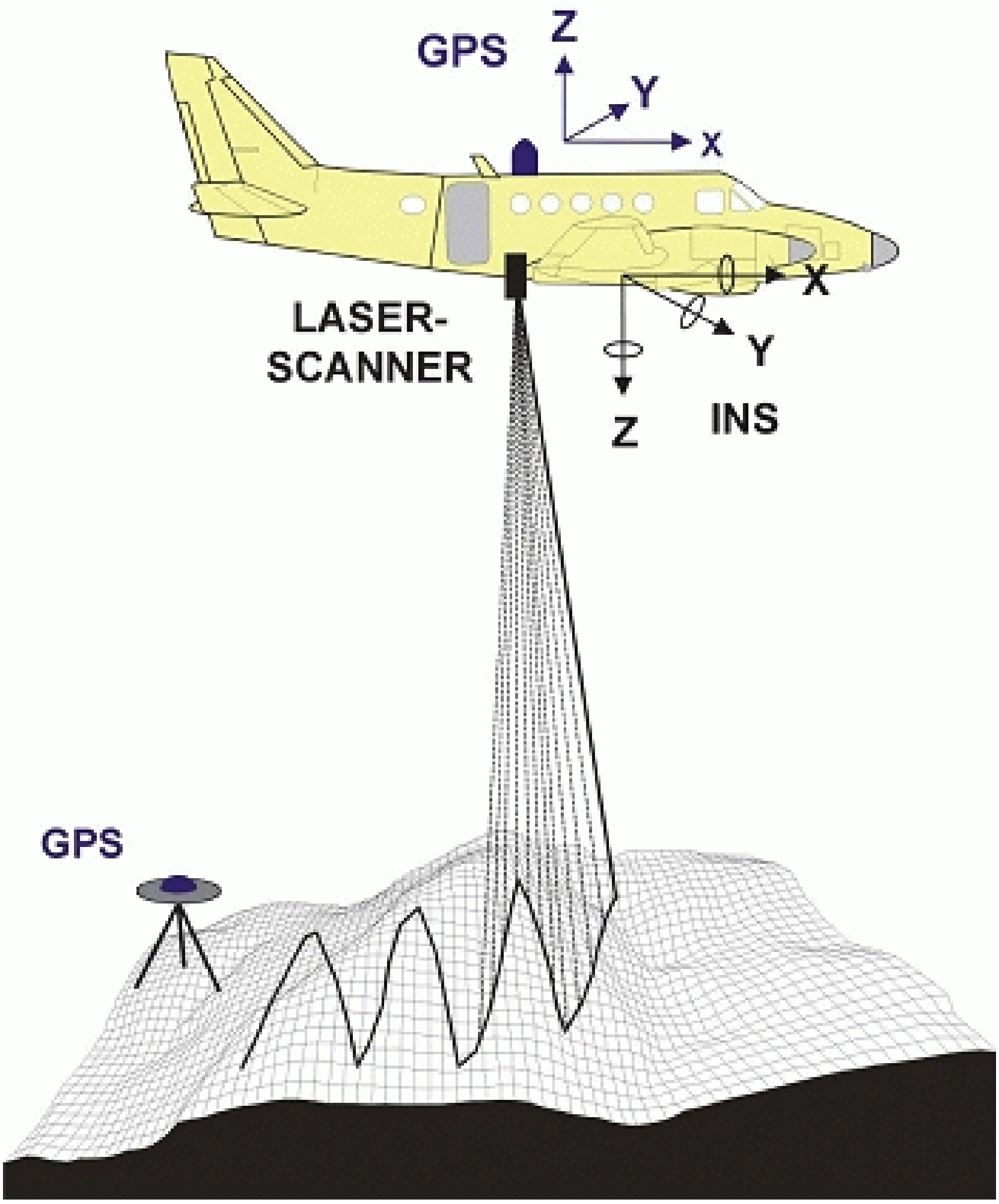


Figure 2.8: Simple visualization of Airborne Laser Scan (ALS) data acqusition [Mag16].

to vector data we could imagine a building layout. If we would like to create a 3D model of a specific building using a layout from a raster image the task would be non-trivial if not impossible. In this sense it is easy to see how a georeferenced vector representation is much more exploitable than a raster image to analyse topographical features in detail. Vector data is more costly in terms of acquisition, because it can only be acquired manually or by complex algorithms which are often based on the raster data. All vector data is either represented by points, lines or polygons. Points form the base for lines,

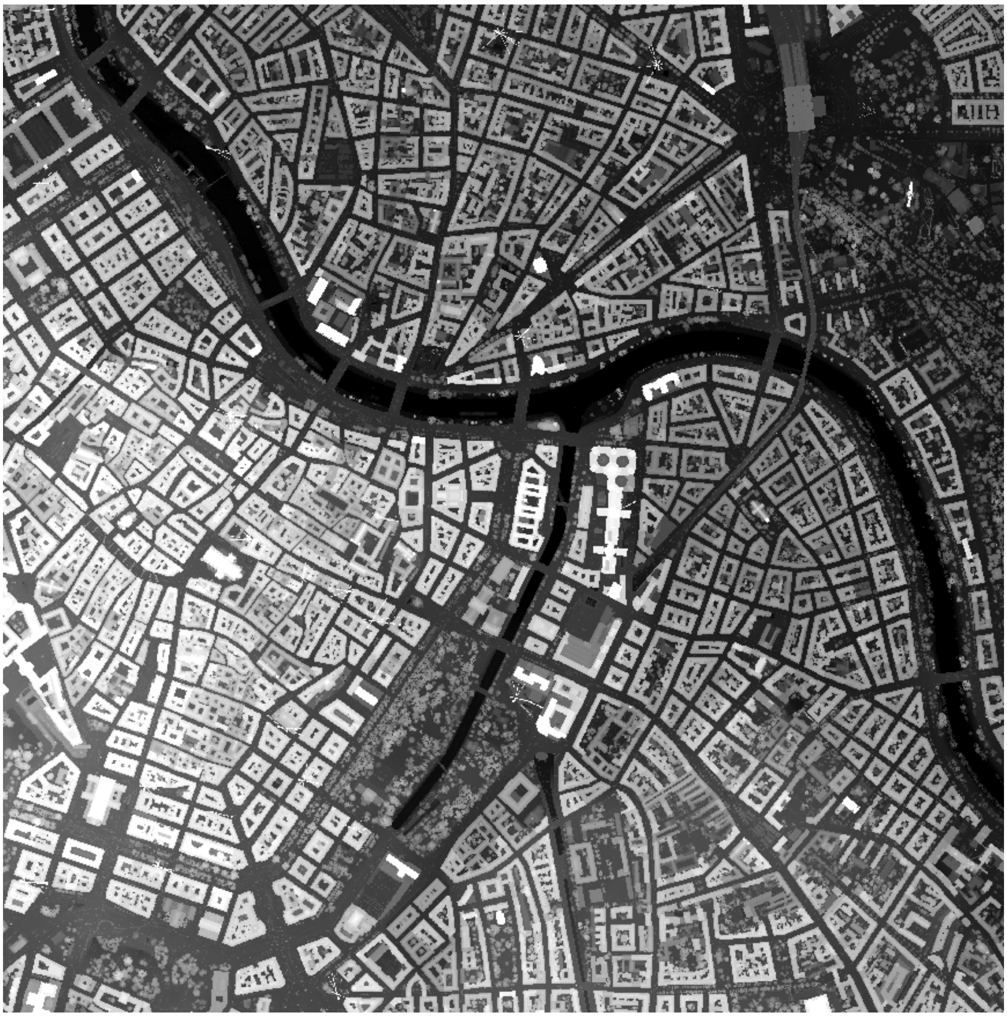


Figure 2.9: DEM of Vienna visualized as a grayscale image.

which form the base for polygons. As simple as this concept may seems it comes with many problems bound to polygon operations, which is going to be described in detail in Section 2.4 of the thesis. A *point* is represented by a projected or unprojected coordinate tuple, while a *line* is the result of a string of points. A line that ends on its starting point is called a *linear ring*. The linear ring could be misinterpreted as a polygon, but a polygon can be described as a set of linear rings. The first linear ring always describes the outer bound of the polygon (exterior ring) while all other following linear rings (interior rings), if any are defined, describe holes in the polygon. Simple examples of points, lines and rings can be seen in Figure 2.10.

To store this vector data on computers different standards have emerged over the years. All standards have in common that they convey information about the datum, projection system and units in order to enable the work with the coordinates [Law13]. An example

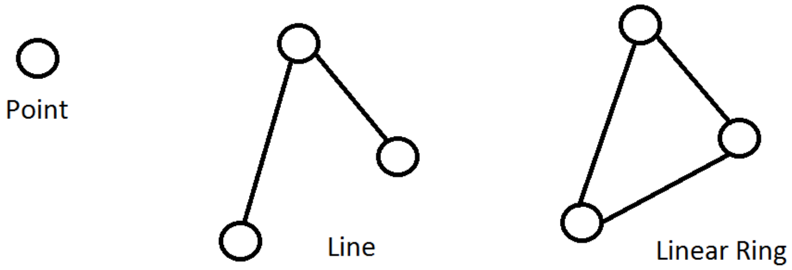


Figure 2.10: The basic structures used for all geometries in GIS.

of a popular georeferenced vector data format is the shapefile format (SHP). Developed by the Environmental Systems Research Institute (ESRI), shapefile is an open standard used to exchange and store georeferenced information in the form of attributes associated to a specific point, line or polygon [Ins98]. The Shapefile standard defines a list of files, each describing a specific information as seen in Table 2.1. The heart of the whole format of course is the .shp file, which contains all the geometric information, the .dbf file containing all the attributes, associated with a specific point, line or polygon defined in the .shp file and finally a .shx file defining an index cursor for faster lookup in the tables. Also notable is the .prj file which stores the type of the projection system. There are many other files described in the shapefile standard but the ones we mentioned are the most important and relevant for this work.

The shapefile standard has to cover various aspects of the georeferenced data it stores. Formats like shapefile are called *macro formats*. Simpler tasks or information exchange tend to become cumbersome when working with macro formats. To exchange information between formats more convenient smaller standards are used called *micro formats*. The micro formats I mostly worked with are called Georeferenced JavaScript Object Notation (GEOJSON) and Geography Markup Language (GML). As the names suggest they are based on the JavaScript Object Notation (JSON) and the Extensible Markup Language (XML). In contrast to macro formats the micro formats are not used to store the information, but solely to ease the information exchange between different systems, for example when transfer speed is essential.

2.1.6 OpenStreetMap

OpenStreetMap (OSM) was one of the first projects based on open communities working together to collect all kinds of georeferenced data [HW08]. As such it defined its own XML-based data format to make its contents exchangeable between users. Similar to shapefile all geometries are described using nodes, ways and areas. The smallest unit in the OSM standard is called node and it describes a single point. Next in size are the paths which are represented by a string of connected nodes. Finally we have areas which are closed paths which can be interpreted as polygons. Closed paths means that the

File	Description
.shp	Contains the geometry itself.
.shx	Contains a positional index for fast forward and backward lookup.
.dbf	Database schema in dBase format.
.prj	Describes the projection and coordinate system.
.sbn / .sbx	Contains a spatial index for the features.
.fbn / .fbx	Contains a spatial index for the read-only features.
.ain / .aih	Contains an attribute index of the active fields.
.ixs	Contains a geocoding index for data-sets.
.mxs	Contains a geocoding index for data-sets in the ODB format.
.atx	Contains an attribute index for the .dbf file.
.shp.xml	Contains geospatial metadata.
.cpg	Contains information to specify the code page for .dbf
.qix	Contains an alternative spatial index.

Table 2.1: List of files of the SHP standard [Ins98].

starting node has the same coordinates as the ending node, thus closing the path. All the coordinates in the OSM standard follow the WGS84 specification.

To access this information there are two possibilities. The first one is to load the data over the OSM API. This gives access to the most up to date data, but restricts the number and size of requests as described in [Law13]. The second approach is to download prepared .osm dumps from specialized portals and in order to work locally on the downloaded data, which is discussed in [Wes10] and [Law13]. Depending on the objective we can choose between both, but for bigger applications the second suggestion is recommended. To work locally on the downloaded data it can be loaded into a local spatial database such as PostGIS for easier access. PostGIS is a optional extension based on PostgreSQL, which adds location awareness and the possibility to run spatial queries on the database [C$^+$15]. Being an open source project many community-based tools can be found online further easing standard tasks. For example loading OpenStreetMap data into a PostGIS database can be easily achieved using tools like *osm2pgsql* [Ope], which loads the downloaded .osm files into a specified database.

2.1.7 Government databases

Nowadays OSM is not the only source for free georeferenced data. For example we used data acquired by the Austrian government. This data was made accessible in the course of the Open Government Data initiative. It can be downloaded using the government's official web application as seen in Figure 2.11 or using their Web Map Service (WMS) which is a standard defined by the OGC. WMS are services especially defined to make GIS data easily loadable as they make use of servers capable of converting data in specific image file formats or even Scalable Vector Format (SVG) to export vector data from the GIS.

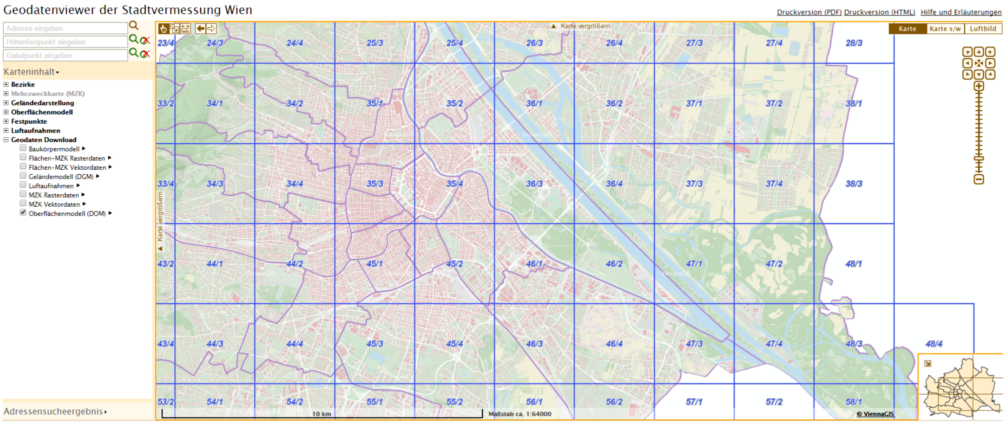


Figure 2.11: A screenshot taken from the online tool to download GIS data for Vienna [Mag16].

There are countless other possibilities to get georeferenced data depending on the quality and task that have to be addressed. Nearly every platform defines its own standards for the information interchange. Even though this formats are based on the same concepts, there can be some variations between some conventions (like naming if we compare SHP and OSM formats). This creates problems when trying to work with data from different sources and has to be taken into account when comparing different datasets with each other.

2.1.8 Challenges of georeferenced data

Even after having accessed the data, the huge amount of information contained in it makes grasping specific contents difficult. Just to make an example the compressed dump of the planet.osm, which is a dump containing the whole OSM database information, has a size of 50 gigabytes. Once uncompressed into an XML file it reaches a size of 666 gigabytes. Even if loaded into a database to ease the information extraction the queries needed to access a specific set of information (e.g. the boundaries of buildings) are not easy to formulate, because fundamental knowledge of the tagging conventions for the data-sets are needed. This mostly applies to the vector data. When considering the raster data, the complexity is severely reduced. Most of the time raster data is stored as georeferenced images. Even though the data has already been processed it is still affected by noise and shadowing as seen in Figure 2.12. Hence, signal processing and filtering playes a big role for this work to improve the DEM's in order to extract realistic heights for the 3D models.

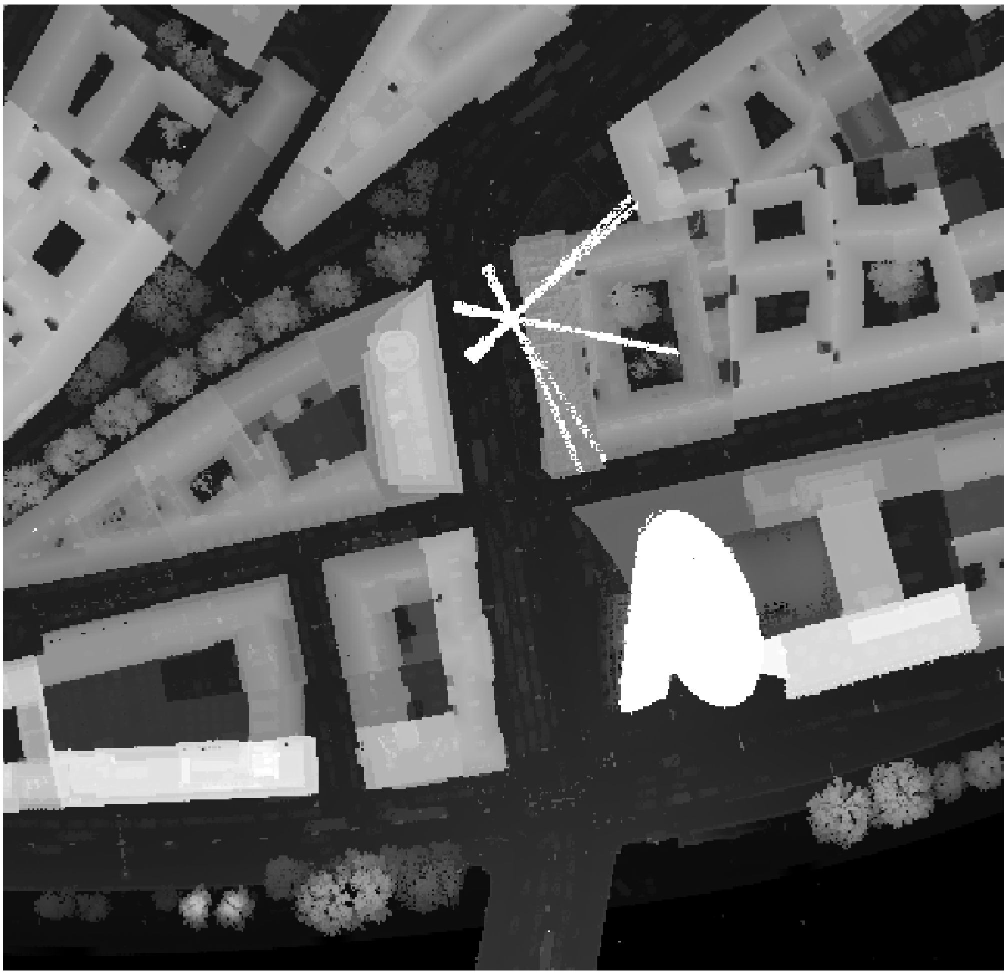


Figure 2.12: Imperfections of DSM's showing a helicopters rotor in the final result.

2.2 Digital Elevation Model

Digital Elevation Models are created using the remote scanning technologies described in Section 2.1.4. A DEM is raster data containing three-dimensional information of the Earth's terrain [TBB01]. I am going to discuss the theories behind DEM and how it can be used to model the terrain's roughness and to calculate the building's heights. At this point it should be clear why it was important to understand remote sensing technology and how ALS works in order to understand how a DEM is created. Here I mention DEM's because I used them to calculate the buildings height when I could not find that information already in OSM as discussed in [Wes10].

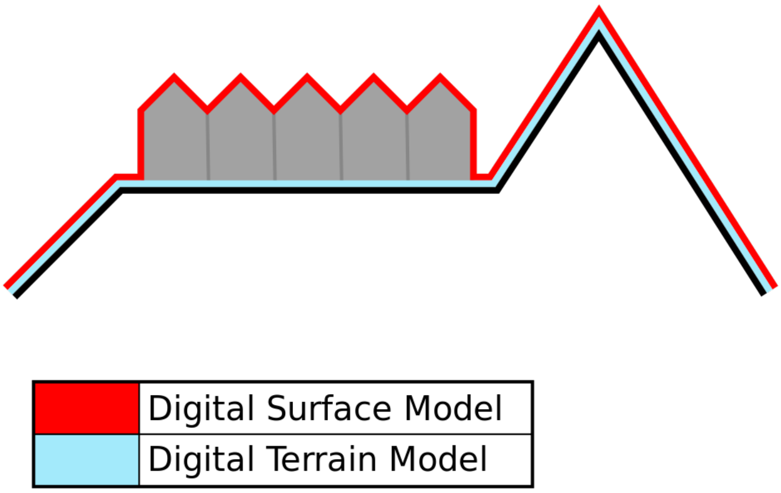


Figure 2.13: Difference between Digital Terrain Model (DTM) and Digital Surface Model (DSM) [Mar].

Even though in most scientific literature there is no uniform usage of the following terms I would like to differentiate between DSM and DTM, both of which are specialized DEM. This terminology has been chosen following the documentation of the government's data used while developing the presented framework [Mag16]. By combining the information contained in the DTM with the information contained in the DSM, as seen in Figure 2.13, it is finally possible to calculate the height of the buildings.

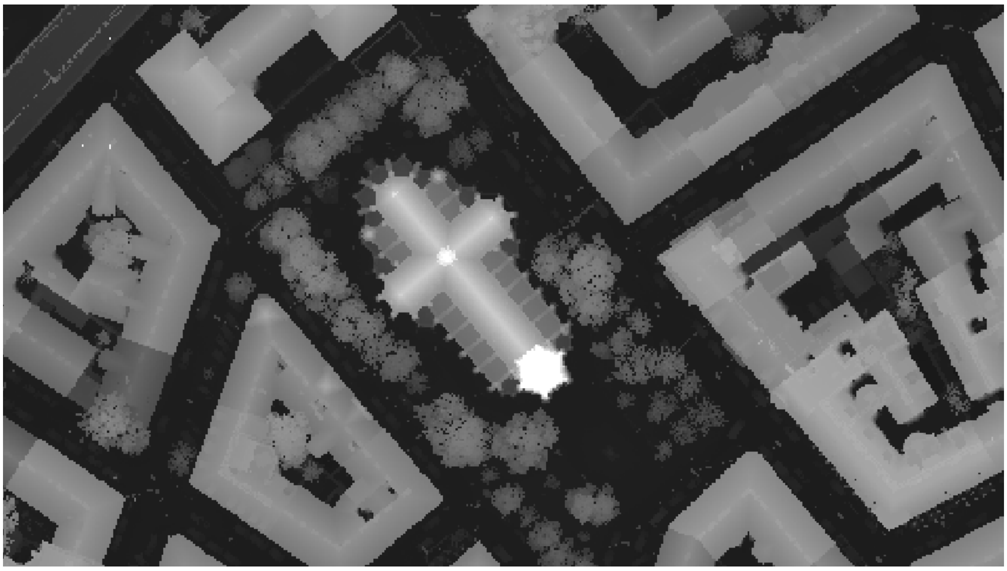


Figure 2.14: Example of DSM from the St. Othmar church in Vienna.

Previously I already mentioned that DEM visualizes raster data. This is true for both DTM and DSM. Opposed to the DTM which contains the terrain's height, as seen in Figure 2.14, I also mentioned the DSM which contains the surface height as seen in Figure 2.14. By looking at the figure it should become obvious that subtracting the heights contained in the DTM from the heights contained in the DSM results in a raster image containing zeros on ground level and the building's height values at any other element of the resulting raster. Working with raster data can be done with many different tools. The tool of my choosing was Quantum Geographic Information System (QGIS).

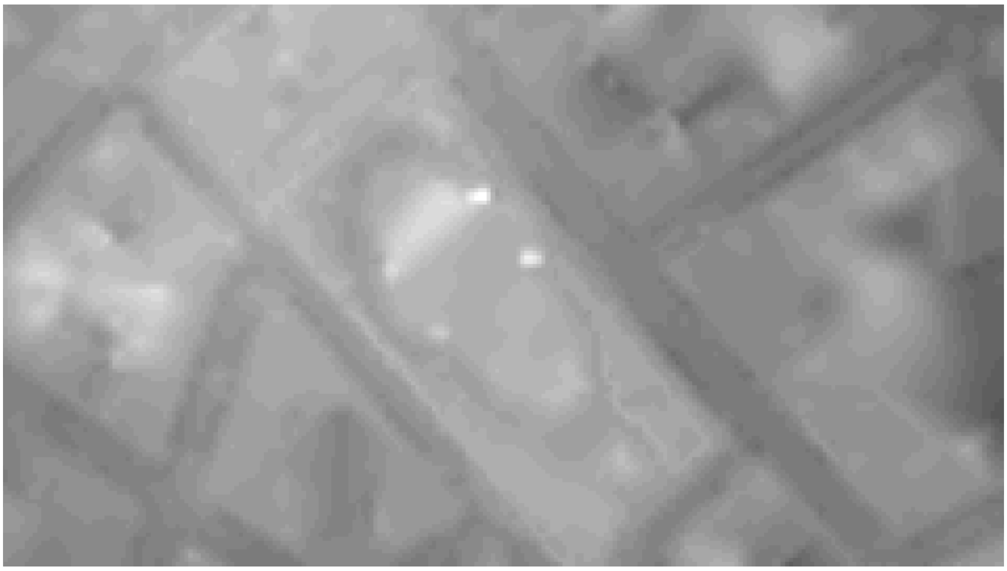


Figure 2.15: Example of DTM of same area seen in Figure 2.14.

QGIS is an open-source GIS available for free under the GPL license, which is the main reason I opted for it, in order to work on the data of the framework. In addition to its availability QGIS offers many state of the art algorithms to process georeferenced data natively. The most notable features for this section of my work are its ability to calculate zone statistics and simple operations with raster images (e.g. the subtraction of two raster images as was mentioned in the previous paragraph). The zone statistic makes it possible to calculate the median over regions of the raster image defined by polygons in the vector layer. Calculating the zone statistics is helpful for assigning a specific height value to each building, because in most cases the buildings roofs are not flat which results in different height values for one single building. This process could be improved by adding algorithms to calculate roof models and merge them with the building in order to create even more impressive city models. However in this work I focused on creating only the basis of the framework, although it may be improved in the future.

My first attempts to create a high-quality city model consisted into directly calculating a surface by interpolating each value contained in the DSM, as is done for raised relief maps such as the example showed in Figure 2.17. As can be seen in Figure 2.16 the resulting

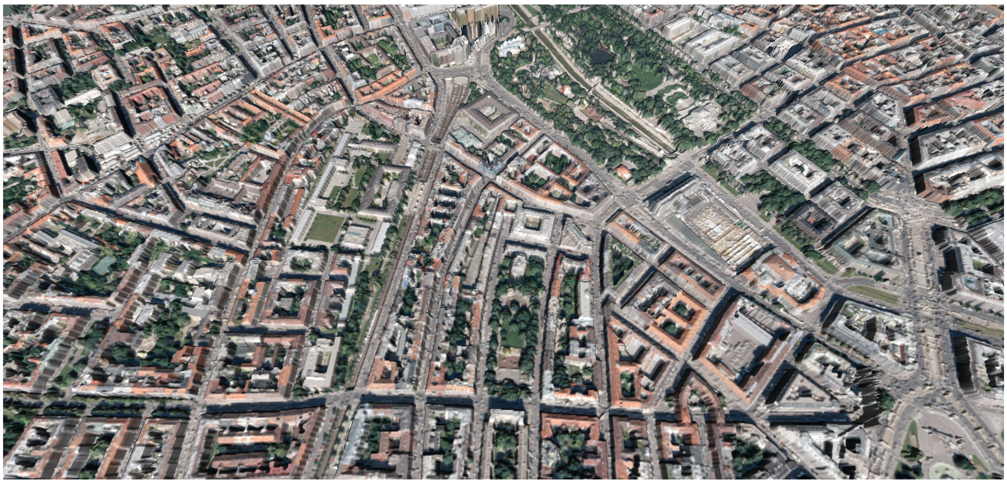


Figure 2.16: Textured surface created by interpolating raster data of DEM.

model was already a decent 3D surface model. However as good as the resolution may be a surface model was not the result I was looking for. The models turned out to be imprecise when looking on at them from the sides. However for the first version of the virtual reality environment used in [EM15] we implemented the surfaces created this way.

I already mentioned before that a DEM representing raster data can be visualized as a gray scale image, and at this point of my thesis I would like to go more into the details of how this representation is created. This task may sound easy but there are some thoughts behind the mapping that I would like to describe in order to make the reader note some different causes of errors when working with gray scale images as well as the parallels to the theories of projected coordinates explained in Section 2.1.2.

First of all gray scale images can have different bit depths depending on the data type used to represent each pixel in the image. Considering the images bit depth means that the height values of the observed DEM have to be normalized in order for them to be comparable. To visualize the same point in a 8-bit gray scale image the data point has to be mapped using an unsigned integer which is restricted to values between 0 and 255. The mapping is carried out by taking the max and min values of the section into account. Obviously the greater the terrain's elevation range the more inaccurate the mapping becomes. Depending on the task it may make sense to increase the bit depth of the gray scale image, in order to increase the precision of the DEM.

The point here is that just taking the pixel values gray scale image as the height values will result in incorrect heights. To calculate the right height the minimum, maximum and bit depth of the gray scale image have to be taken into consideration. I think it is important to understand the parallels between the projection discussed in this section and the coordinate projection discussed in Section 2.1.2. Here I would like the reader to notice that this process is a projection too and as such can be better comprehended after reading Section 2.1.2.

GEOMORPHOLOGICAL MODEL OF THE
CARRARA MARBLE BASINS
(Apuan Alps, Tuscany, Italy)
PLASTICO GEOMORFOLOGICO DEI BACINI MARMIFERI
DI CARRARA (Alpi Apuane, Toscana)

Figure 2.17: Three-dimensional surface which can be created using a DSM. [BRBM10]

One of the challenges when working with DEM, which I already mentioned in Section 2.1.8, were firstly the noisy results of ALS and secondly the artefacts that can occur, as shown in Figure 2.12. A common method used to remove noise from images is to apply a median filter. As described in [TM98] the median filter is a filter with edge preserving qualities. Therefore using the median filter removes salt and pepper noise from the DEM without compromising the building's edge information contained in it. To counter the second problem, which were the artefacts compromising the result, the median value of an area is calculated as explained in [Wes10]. Previously I mentioned the median value when discussing the features of QGIS. Basically calculating the median over an area defined by the building's footprint, as seen in Figure 2.19, avoids outliers (e.g. the artefacts shown in Figure 2.12) in the area. Eventually this process could be improved in order to calculate three-dimensional models of the roofs by using alternative edge preserving filters presented in [TM98].

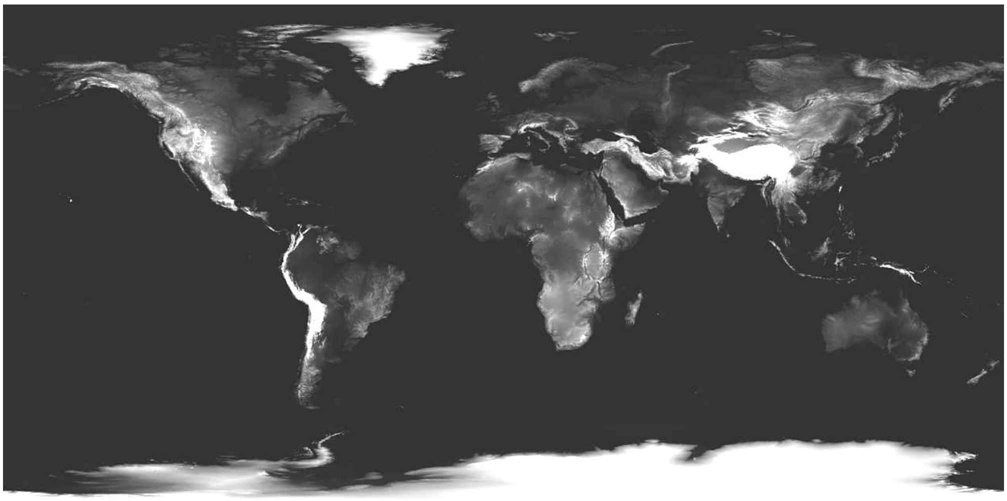


Figure 2.18: Gray scale image of the world’s DEM. [NASb]

2.3 Geometry Modelling

Until now the process of extracting valuable GIS data from different sources was described. From here on this work moves from the fields of geodesy and photogrammetry towards the fields of computer graphics, 3D web development and game design thus building a valuable bridge between these areas.

The information presented until now was necessary to understand the possibility to access georeferenced data and to create more than maps and map-overlays with it. This idea becomes more and more present in the current research and has been already pursued by [ZSP] [DH07] [RV01]. However my thesis is mainly motivated by specific requirements that originated during another project [EM15]. The 3D models created using GIS data acquired and tested at that time, were not satisfying these requirements. Being created by an automated process the tested models did not always present a clean geometry. This may or may not be a problem depending on the context the model is used in or the tools that are used to visualize it. However once I imported the researched models into Blender, Meshlab and Unity, in order to test them, the resulting scene was plagued by the visualization issues and glitches. An example of such glitches can be seen in Figure 2.20.

These issues were mostly caused by two circumstances. The first one was that the scene I had to visualize for [EM15] was vast (Vienna’s area approximately amounts to 400 kms). Jumping towards the city from the sky means that most of the city is in the skydivers field of view for most of the time. As already discussed in [PS12] creating vast game worlds already presents many problems for third person viewed games. It turned out that these problems become even worse for vast virtual reality environments. This leads to the second circumstance aggravating the situation; the use of virtual reality by the means of a head mounted display (HMI). Using the HMI implies that the skydiver perceives the

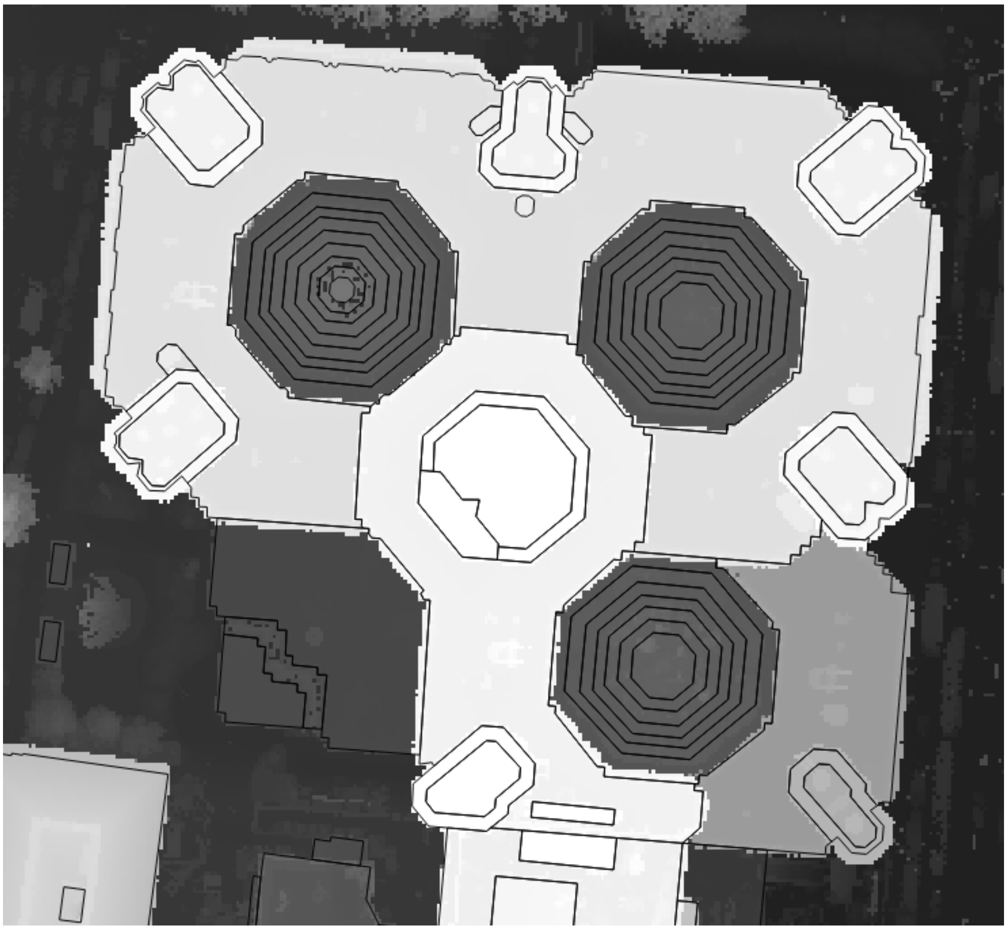


Figure 2.19: QGIS overlay of the DSM and the building footprints.

3D scenery in the first person. To ensure a good first person experience the viewer has to be able to see and interact with very close objects (e.g. the avatars hands touching its own face). Hence the scene had to render a very far horizon, while also rendering the camera's proximity.

In terms of computer graphics the near clipping plane (closest perceived objects) of the scene was close to zero (the center of the camera/eye), while the far clipping plane (farthest perceived objects) had to be moved beyond Vienna's borders. Figure 2.21 visualizes the concept of clipping planes. These circumstances created a well known issue named *z-fighting* [VF13]. The letter "z" of the name stands for the z-axis, which in the context of computer graphics always describes the depth or distance. This information is important because while visualizing a 3D scene the color of each pixel on the screen is decided by checking the distance of an object to the camera. Figure 2.21 visualizes how the occlusion is calculated by checking the object's distance from the camera.

In other words this concept just implements a basic occlusion. Near objects occlude

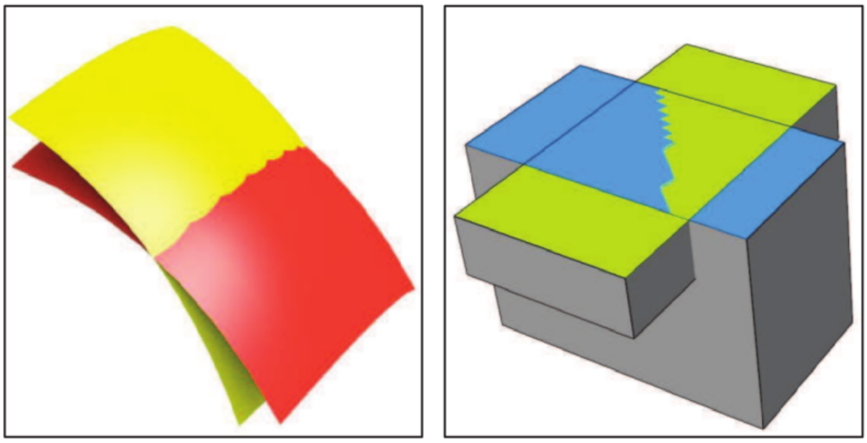


Figure 2.20: Showing z-fighting in *Blender 2.5* (left) and *Google SketchUp 8* (right) from [VF13].

objects behind them and so the color value of the nearest object in the field of view has to be written at the right pixel of the screen. So while looking in the distance the horizon is occluded by nearer objects. As natural as this may sounds, in a digital scenery the distance between two points has to be described by a countable amount of values. Increasing the distance while keeping the size of the set describing it, results in a loss of precision. Moving the near plane closer to the viewer has an even worse impact on the precision loss. This is caused by the perspective divide which naturally causes the z-buffer to be less precise in the distance. This phenomenon implemented trough perspective divide can be observed while moving and looking towards the horizon. Closer objects will appear to be moving by faster opposed to the mountains or trees in the distance. Thus rendering a scene for virtual reality can result in a significant precision loss of the z-buffer.

For [EM15] that loss of precision meant that the game engine could not keep the surfaces of two close objects. This numeric error results in the planes overwriting each other randomly over time, which is perceived as flickering of the object's surface as seen in Figure 2.20.

To oppose z-fighting various complex approaches are discussed in [VF13]. However in most cases, depending on the given constraints, different simpler approaches can be adopted instead. One of them is to increase the precision of the depth buffer. However this approach can not be used when there is no option to access the depth buffer's precision. This is often the case when working with game engines, which abstract the access to the graphics card, masking many options. The arguably most common solution to z-fighting would be to move the clipping planes closer together in order to decrease the distance

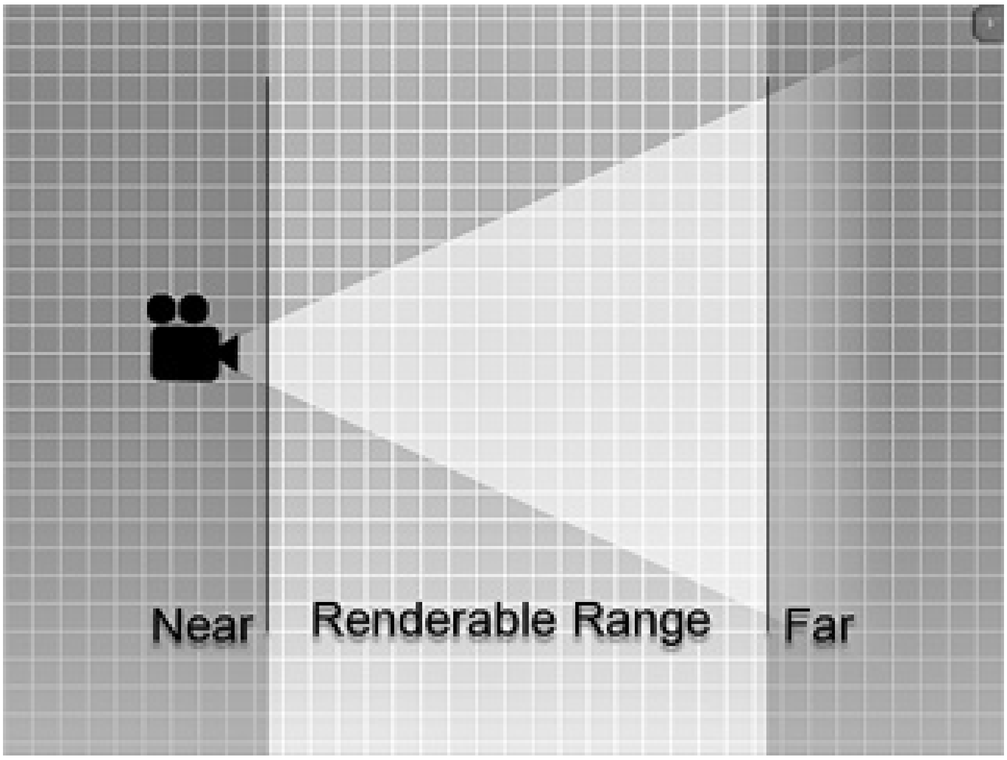


Figure 2.21: Visualization of the view frustum between clipping planes [Rea16].

that the depth buffer has to cover. As mentioned before, though, the requirements of the virtual reality environment used in [EM15] would not allow these commonly adopted solutions. Adapting the level of detail of the buildings is also a commonly used approach, however z-fighting persisted even for the lowest found level of detail for the models. Given the time constraint and available data it was decided to create, even simpler models from scratch, purpose-built to avoid z-fighting for easy integration into game engines, and further procedural improvement.

2.4 Polygon clipping

In this section the parallels between polygons and the vector data used in a GIS will be drawn. As already described in Section 2.1.5 vector data stores two-dimensional polygon information, mainly to describe streets, borders and areas, e.g. using markup language. Obviously in a city, given the amount of streets, buildings and different areas (e.g. parks, public transport, pedestrian areas, etc.), the density of vector data increases. Here I focus on the buildings' footprints which reduces the problem's complexity. By extracting the buildings' footprints the clipping can be applied on simple polygons as already seen in Figure 1.3. However as simple as the polygons may appear after this first filtering there are still many special cases left, as seen in Figure 2.19, which have to be simplified using

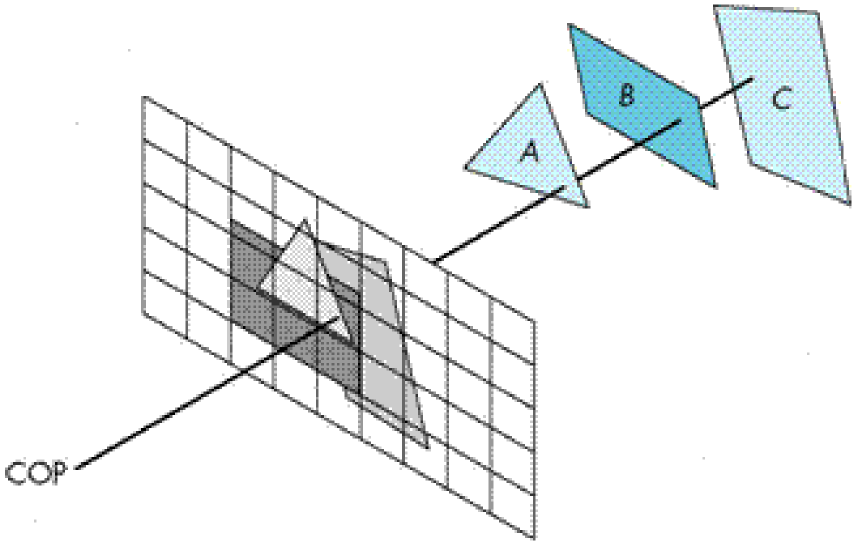


Figure 2.22: Depth of the closest object to each pixel is stored in the z-buffer [LKR].

polygon operations. This figure shows how the buildings' details are defined by further inner polygons. In [RF00] different algorithms are presented to solve problems given by polygons like the one that can be seen in Figure 2.23. By using the union operator on Figure 2.23 the authors of [RF00] were able to create the polygon seen in Figure 2.24.

Basically clipping is defined as an operation between a subject polygon S and a clip polygon C. When mentioning clipping this work mainly refers to the four basic boolean operations which are Union, Intersection, Exclusive-Or and Difference. To clip two polygons most times the first step consists in resolving which nodes of S are inside and which are outside of C. Hence the supporting pillar of all the discussed operations is the definition of the inside and outside of a polygon. This may be easy to define for simple closed polygons but for self-intersecting polygons there is no intuitive solution. To overcome this problem the *winding number* [GH98] can be used. The winding number counts how many times a ray with origin at a specific point P completes a full turn around P when following the contours of a polygon counterclockwise. This results in a odd winding number when A is inside the polygon, in the winding number being 1 when A is inside a simple closed polygon and in a even number when A is outside the polygon as can be seen in Figure 2.25.

For these clipping operations *Clipper* [Joh] can be used which is an open source library implementing a numerically robust algorithm for clipping, of both lines and polygons. The algorithm implemented by the Clipper library is called *Vatti Algorithm* [Vat92]. Opposed to other clipping algorithms, like Sutherland and Hodgeman's algorithm [SH74] which

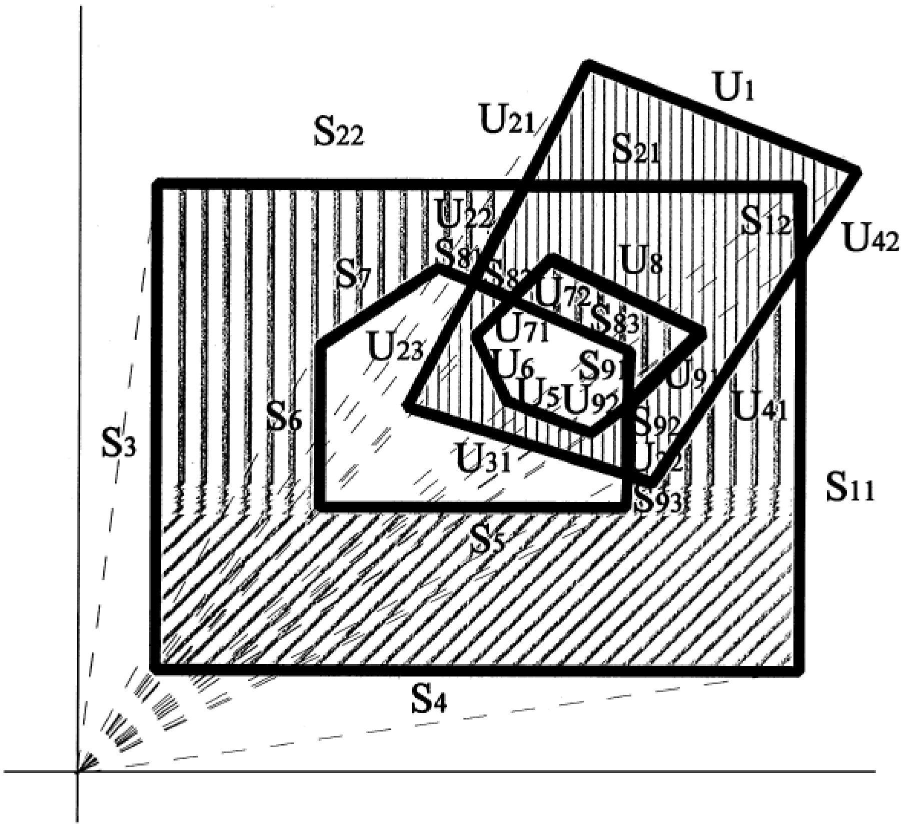


Figure 2.23: Example showing difficulties of overlapping polygons [RF00].

is limited to convex clip polygons and that of Liang and Barsky [LB83] which require the clip polygon being rectangular, the Vatti Algorithm does not have any constraints on the polygons it can be applied to. This applies to self-intersecting polygons and polygons with holes [Joh], both of which can be found in the vector data extracted from a geospatial database. The library's operations are mainly based on the concepts presented in [Ago05], [Vat92] and [Joh]. Union has already been briefly shown in Figure 2.24 as an introductory example of polygon clipping with boolean operations. In the following list we will have a closer look at the definitions of different boolean operations.

1. Intersection : The *Intersection* between two polygons results in a (multi-)polygon covering the overlappings of the two polygons.

2. Exclusive-Or: The *Exclusive-Or* between two polygons results in a (multi-)polygon covering exclusively the areas that were not overlapping.

3. Difference: The *Difference* between, as the name implies, removes the overlapping areas (if there are any) from the first polygon.

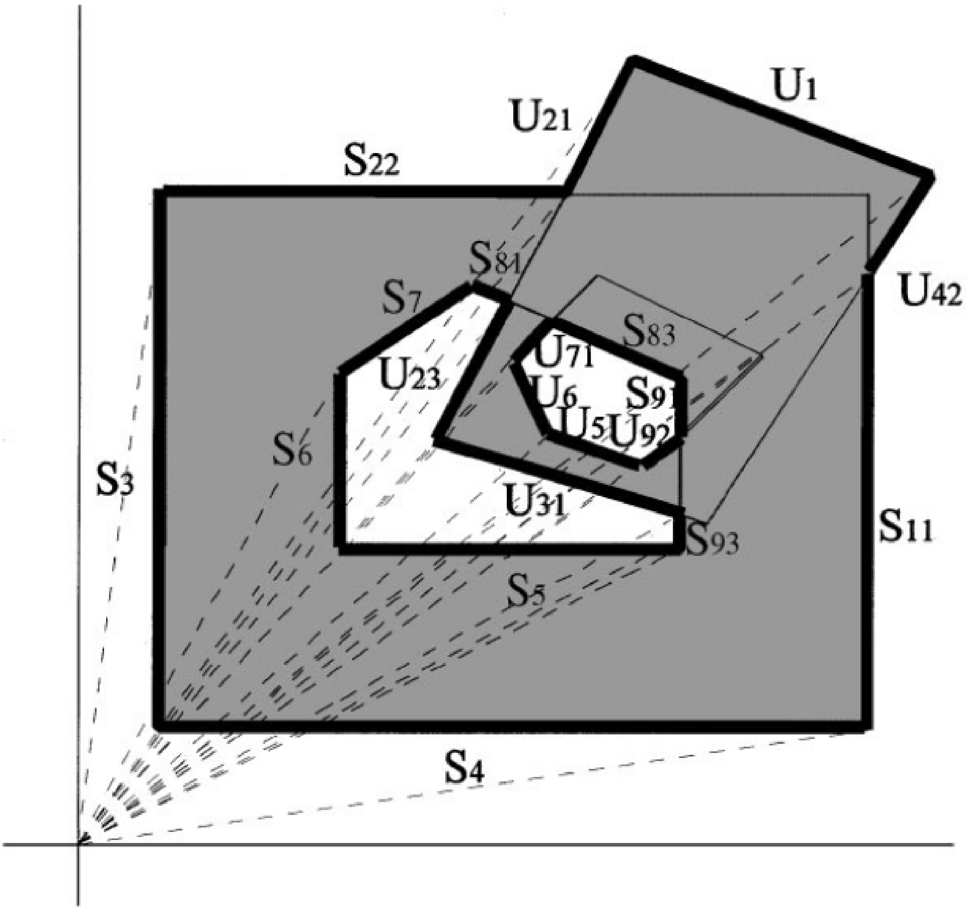


Figure 2.24: Simplified overlapping polygons using union from [RF00].

4. Union: The *Union* between the subject polygon and and clip polygon results in a (multi-)polygon covering the areas of both polygons.

These operations can be utilized in the simplification process of the georeferenced vector data as they can be applied to convey multiple overlapping polygons into one much simpler and compact polygon. In this section I listed concepts that can be applied in order to simplify the vector data on a two-dimensional coplanar space. However the sought results are 3D models. Therefore in the next section I will present applied solutions to transform polygons into 3D models and further elaborate the necessity of polygon clipping in order to achieve clean topology during the extrusion of the footprints.

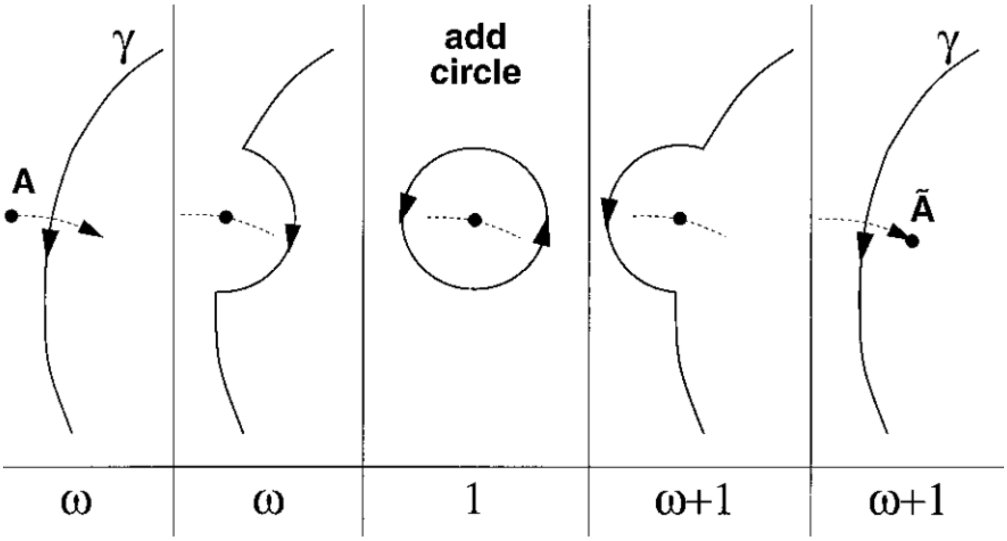


Figure 2.25: Illustration of the winding number (ω) as shown in [GH98].

2.5 Extrusion

[LM11] mentions the idea of achieving a sound topology of the polygons in the two dimensional coplanar space before moving on and creating 3D models, also adding that this step is often omitted in GIS frameworks. The authors discuss extrusion as a mean of transforming the polygons into prisms and address multiple problems that arise with this seemingly simple idea. The main problem can be seen in Figure 2.26. Extruding a coplanar polygon along its orthogonal axes results in a right prism. A right prism has equal base and top and the sidewalls connecting base and top are rectangles. By extruding the footprint of a building by its height the resulting prism can be used as a basic three-dimensional representation of a building.

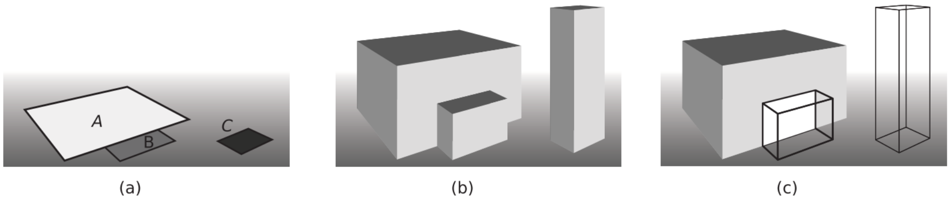


Figure 2.26: The polygons in (a) represent a building's footprint which is extruded in (b) and (c) shows the overlapping of two common sidewalls [LM11].

However to keep a clean topology further steps have to be considered during extrusion as can be seen in Figure 2.26. The problem shown in the image is that the extrusion of adjacent polygons results in walls which will overlap, thus leading to issues like z-fighting. To counter this problem three key requirements are mentioned in [LM11] defining a consistent topology for the polyhedrons, being:

1. No interior of two prisms intersect;

2. The set of faces is topologically consistent;

3. Each polyhedron is formed by a closed bounding surface, naming it *watertight*.

The first and third requirements should be self-explanatory, but the second requirement has to be further explained. Here the faces of the polyhedrons are declared as topologically consistent, referring to the definition of topological consistency for polygons, also defined by the authors of [LM11]. There the topological consistency of polygons is described by the following constraints, M being a set of two dimensional objects:

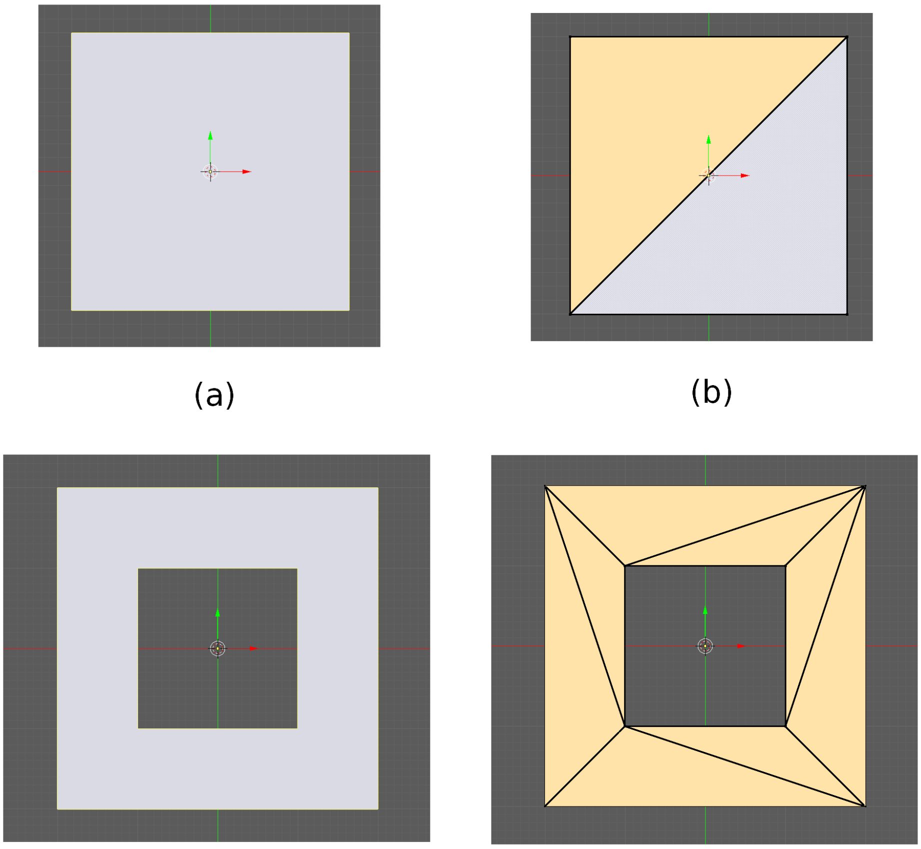
1. Every line segment in M is formed by two points also in M;

2. The intersection of two line segments L1 and L2, denoted $L1 \cap L2$, is either empty or is a point in M;

3. The intersection of a polygon P with another object in M called O, $P \cap O$, is empty.

Obviously the first set of constraints addresses objects in the three-dimensional space, while the second set of constraints addresses objects in the two-dimensional space. In order to adhere to the defined constraints the authors created their own algorithm for polygon extrusion. At this point the structures used to represent and store the data become important in order to explain the algorithm in a further section.

Commonly vector data is stored using a Node-Edge-Face (NEF) structure that describes the boundaries of the model. The boundary representation is widely used for solid modeling [Cha14], making it ideal for representation of the models that have to be integrated into a game engine. In this case however the structures have to be adapted from NEF in order to ease the extraction of relevant information for the extrusion algorithm. In [LM11] the authors mention alternative structures empoyed in relation to GIS data for specific use in a three-dimensional context. However they decide to move from NEF to the Constraint Delaunay Triangulation (CDT). The CDT can adapt the triangularization in order to fit a set of passed polygons, as seen in Figure 2.27. This happens by relaxing the Delaunay condition, which tends to avoid sliver triangles in the resulting structure. The CDT has been chosen because it naturally stores holes without the need of additions to the structure. By comparison NEF would require additional information to be added to the structure, in order to describe holes as can be seen in Figure 2.27. By comparing (c) and (d) in Figure 2.27 this advantage of CDT should become obvious. The set of triangles in (d) is aligned in a way that cuts out the hole, but the NEF structure as seen in (c) just stores the boundary of a bigger and a smaller polygon. In this case the smaller polygon is marked as a hole of the bigger polygon. The CDT offers another advantage over NEF, which is the natural integration into a game engine. This is because the well-known Graphics APIs OpenGL [Mica] and DirectX [Micb] both use triangles as face primitives, therefore the triangularization would be necessary anyway for the integration of the models into a game engine. Thus



(a)

(b)

(c)

(d)

Figure 2.27: Boundary representation of different polygons comparing Node-Edge-Face structure (left) and triangulated structure (right).

the conversion of the NEF polygons into a CDT structure is also the first step of the extrusion algorithm presented by the authors of [LM11].

2.6 Web Graphics Library

With the improvement of the internet connections' bandwidth and the development of new APIs for browsers over the last years a new generation of web-applications has appeared. Mwalongo et al. already mentioned and motivated the use of web applications as a tool for interactive visualizations in [MKK$^+$14]. The problem with remote visualizations until a few years ago, was the missing possibility to run the rendering on the client side.

The Web Graphics Library (WebGL) is an API solving exactly this problem. Trough

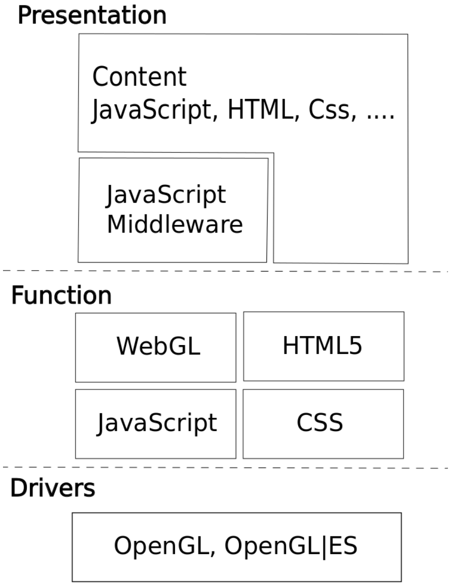


Figure 2.28: Structure of a WebGL application from [FWLL11].

the WebGL standard it is now possible to implement remote applications capable of running the rendering of the sent data on the clients graphic cards as seen in Figure 2.28. Furthermore web applications can be accessed by any compatible browser, making them cross-platform accessible and profit from the web's ubiquity. WebGL is a standard defined in 2011 by the Khronos group and is supported natively in most modern browsers, opposed to alternative solutions which mostly require additional plug-ins to be installed [Mar11]. Freely mixing HTML and 3D without concerns for latency, due to the client side rendering poses a considerable advantage which is still under-utilized in GIS applications.

To further ease the application of WebGL a multitude of libraries can be found with a quick search on the web. These libraries offer different levels of abstraction for WebGL, hence enabling multiple approaches and different solutions depending on the task at hand. One example of such libraries is *three.js* which is very easy to set up and use, but can render complex scenes nonetheless [Cab]. Milner et al. already presented the possibility to improve GIS applications specifically using *three.js* in [MWE14]. The authors show the potential WebGL offers for GIS implementations by implementing basic geographic calculations, based on actual 3D visualizations in the browser, as can be seen in Figure 2.29.

This chapter started with a historical overview of GIS, discussing its evolution over the past decades. By discussing its origins I wanted to explain how and why GIS grew into databases containing a humongous amount of free data. The reader should have noticed that accessing and processing of geo-referenced data is a complex task, thus recognizing the motivation for my work and the projects which have been mentioned in Section 2.1.2,

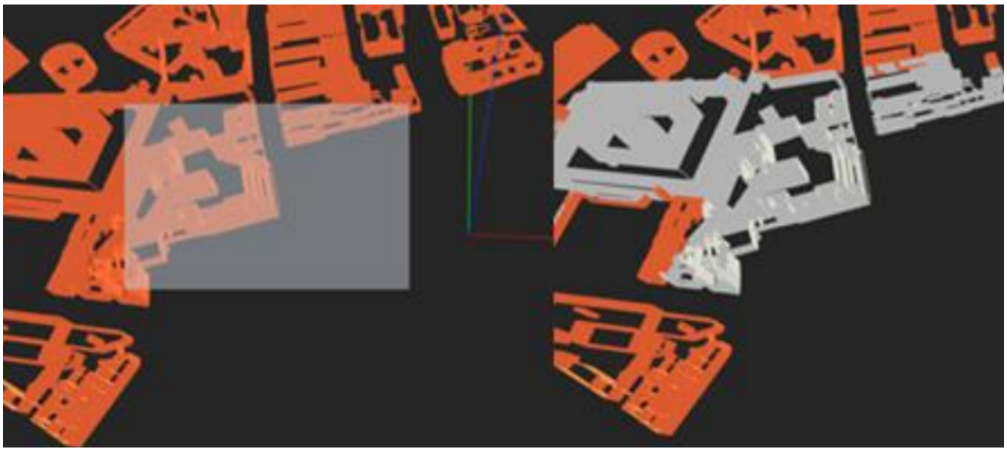


Figure 2.29: Rectangle selection over buildings (left) and the resulting selection (right) from [MWE14].

Section 2.4 and Section 2.5. The second half of this chapter focused on 3D modelling. By listing the theories and concepts of different projects I wanted to describe how I came up with the methodology for the framework that I am going to present in the next chapter.

CHAPTER 3

Methodology

This thesis mainly addresses problems encountered during another project [EM15]. The goal of this former project was to build a skydiving simulator, which enabled the user to feel the thrill of flying through virtual reality by the means of a Head Mounted Display (HMD). In order to create an immersive skydiving experience we needed 3D models of vast areas. Furthermore the jump had to happen over the city of Vienna. After different interviews with urban planners, architects and the municipal authorities for urban surveys it became clear that the creation of such a 3D environment was no trivial task. Thus the research for this thesis started. In the previous chapter I presented the background knowledge I acquired during my research. In this chapter I am going to describe the data and discuss the methods I applied in order to develop the presented framework. This chapter is going to include an in detail description of the framework's requirements, of the data, tools, libraries, and algorithms I used.

3.1 Problem Description

During the interview with the municipal authorities for urban surveys I was able to acquire sources for DTM, DSM and even 3D models of Vienna. However after creating a test scene in Unity3D, the acquired models resulted in a low frame rate at runtime, since they were not optimized for game engines. Furthermore the scene was affected by glitches due to z-fighting, as seen in Figure 2.20. Considering these problems I composed requirements and constraints for a framework capable of creating simpler models. The results are based on the data made available by the government and are used in a game engine for a virtual reality context. Additionally this framework can be used in combination with the tool presented in [EM15] in such a way that the creation of 3D environments for the skydiving simulator is considerably simplified.

3.2 System Requirements

In this section I am going to describe the four main requirements taken into consideration while implementing the presented framework. First and foremost the resulting models have to be optimized for use in conjunction with a HMD. As was explained in Chapter 2 z-fighting is caused by 2 major factors, the application of a HMD being the first one and the second one being the context of the simulation which demanded a clear view of a city's skyline. This setup requires the near clipping plane of the scene to be close to the viewers eyes and the far clipping plane to be at the horizon. Creating models able to perform well under the these conditions constitutes the first requirement of the framework. The second requirement derives from the fact that the simulator may be applied at request from different customers asking for differing environments. Creating scenes for different simulations may be a repetitive but still arduous task. Therefore the tool has to be flexible in order create city models of any city just by inputting data in an easy and fast way. Ideally the tool should be able to be run online for two reasons. The first being the portability. By running the tool as a web application it can be accessed from anywhere without having to take any compatibility issue into account. Furthermore modern web applications can also run locally by using any browser just as a container. The second reason is that by implementing these concepts as a web application the load of the calculations can be distributed at least between the client and the server if not even over a whole distributed system. Of course this last requirement is not going to be fully implemented in our framework, but it is taken into account for its further development. The next requirement addresses the size of the models created using the framework. Keeping the scene as lightweight as possible is crucial, since rendering 3D scenes in a virtual reality context always results in two frames, one for each eye, thus doubling the rendering efforts for the application. Therefore the models created by this framework need to be as small as possible. In this case I consider the size of the models to be directly expressed by the amount of vertices used to render the scene.

Summarizing these four requirements the resulting framework should mostly address the need for an easy to use and portable tool capable of automating most processes that are required for the creation of 3D models of a city. Since the data plays a central role for these processes, in the next section I am going to describe the data used, its sources, and the statistical methods applied to it in more detail.

3.3 Available Data

The Vienna Open Government Initiative offers the possibility to retrieve various data free of charge using their geodata viewer. This tool consents the online view of Vienna's surveys in form of district maps, multi-purpose maps, terrain maps, and aerial images with the additional possibility to export the visualized data for specific areas of the city. In order to reduce the requests on the government's site I implemented scripts to automatically download the data I needed during the development. By looking at the data viewer it becomes obvious that the data is ordered on a grid. Due to the differences in complexity and size of the various downloadable data types, the grid's granularity changes depending on the selected data type. Therefore the higher the level of detail the smaller the grid's sectors become. By varying in size the sectors of the grid increase in number thus are also addressed different ways. In order to script the data-acquisition process I had to make following observations on the government's geodata viewer.

3.3.1 Geodata Viewer Vienna

The lowest level of detail covers the biggest sectors which have a size of 5x5 kilometres each. These sectors are ordered on a 10x5 grid covering the whole city of Vienna. The sectors are addressed by a sequential index which starts at 11 and ends at 60. The first sector lies on the upper left corner of the grid. This results in a grid enumerated as seen in Figure 3.3.

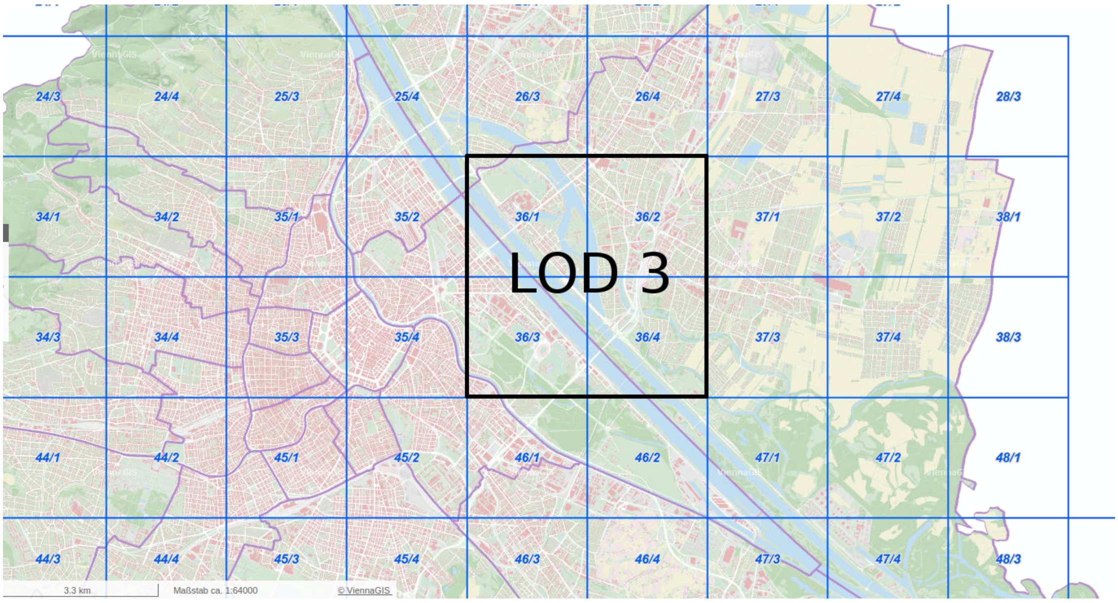


Figure 3.1: Marked section is covered by LOD3 as seen in the geodata viewer for Vienna.

The next smaller level of detail is represented by splitting each sector of the previously described grid into four equal quadrants. Thus each of these quadrants covers an area of 2.5x2.5 kilometers. These quadrants are again addressed by a sequential index which starts at 1 and ends at 4, with the first quadrant being in the upper left corner of the splitted sector. Therefore the sectors for this level of detail are addressed by using the index of the upper level of detail's sectors in conjunction with a number that goes from 1 to 4. In Figure 3.2 this concept can be seen as it is applied on a concrete example.

The sectors used to represent the smallest level of detail are calculated by applying the same principle of splitting the sector of the upper level of detail into smaller sectors. In this case the sectors are split into 5x5 sectors each one covering an area of 0.5x0.5

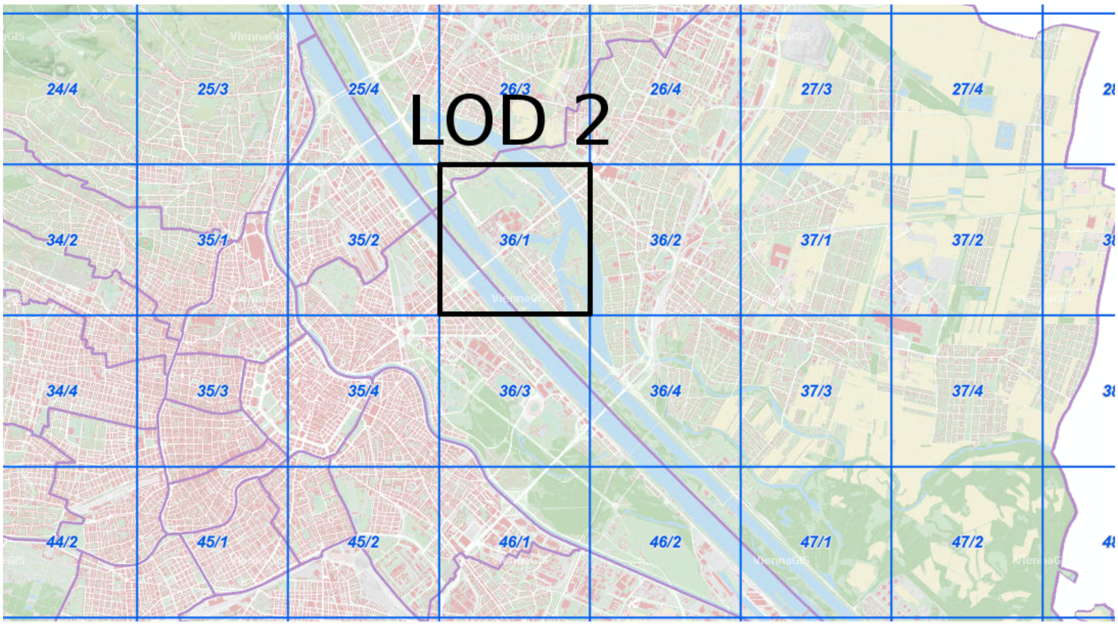


Figure 3.2: Marked section is covered by LOD2 as seen in the geodata viewer for Vienna.

kilometers. Additionally the index used to address each sector is calculated differently. Each of these sectors is addressed by two three-digit numbers. The first of which describes the horizontal while the second describes the vertical position of the sector on the grid as can be seen in Figure 3.3.

By adopting these observations to define the different iterators I implemented a script which iterates over each sector and downloads the data provided by the viewer at each level of detail and stores it in a folder structure that reflects the sequences used by the geodata viewer.

3.3.2 Terrain Elevation Model

This section briefly reviews terrain elevation models. The terrain elevation models are represented as raster information. Therefore the elevation models are stored in matrices of pixels distributed on a regular rectangular grid. In most of our cases terrain elevation models are handled as images as seen in Figure 3.4.

However the image alone describes only the elevation of the displayed landmark. In order for the terrain elevation model to be complete each pixel in the image has to be georeferenced. This is done by adding geospatial information to the raster. A concrete example of such a terrain elevation can be seen by taking a look at the first few lines in the ASCII files that can be downloaded from the geodata viewer.

As can be seen in Table 3.1 the first lines of the ASCII file describe the raster itself. This information about the cell size, the raster's width (ncols), the raster's height (nrows) and

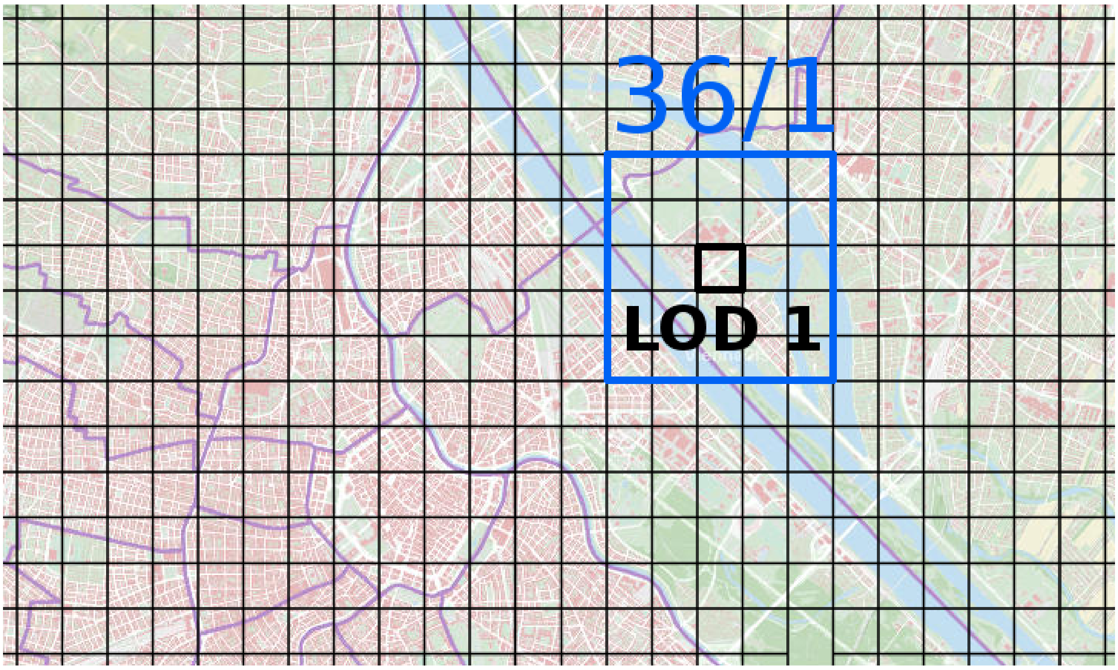


Figure 3.3: Marked section is covered by LOD1 contained in the LOD2 patch with index 36/1.

First 6 lines of a georeferenced ASCII file:
`ncols 5001`
`nrows 5001`
`xllcorner 4999.75`
`yllcorner 342499.75`
`cellsize 0.5`
`nodata_value -9999`

Table 3.1: Example of header for a georeferenced ASCII file.

the rasters origin point (xllcorner and yllcorner) is used for the allocation of each pixel on a map. Note that the projection is not mentioned in the ASCII file. Importing the terrain elevation model with the wrong projection will result in a faulty allocation of each pixel which will become a problem when data from different projection reference systems combined, because the sets of information will not match. Therefore this information has to be looked up. In case of Vienna's geodata viewer the used coordinate reference system can be looked up in the EPSG dataset online registry using the EPSG-code 31256.

In addition to the used coordinate reference system the documentation of the geodata viewer mentions that all elevation measurements are expressed relative to the Viennas zero-height (156,68 meters over the Adriatic mean sea level).

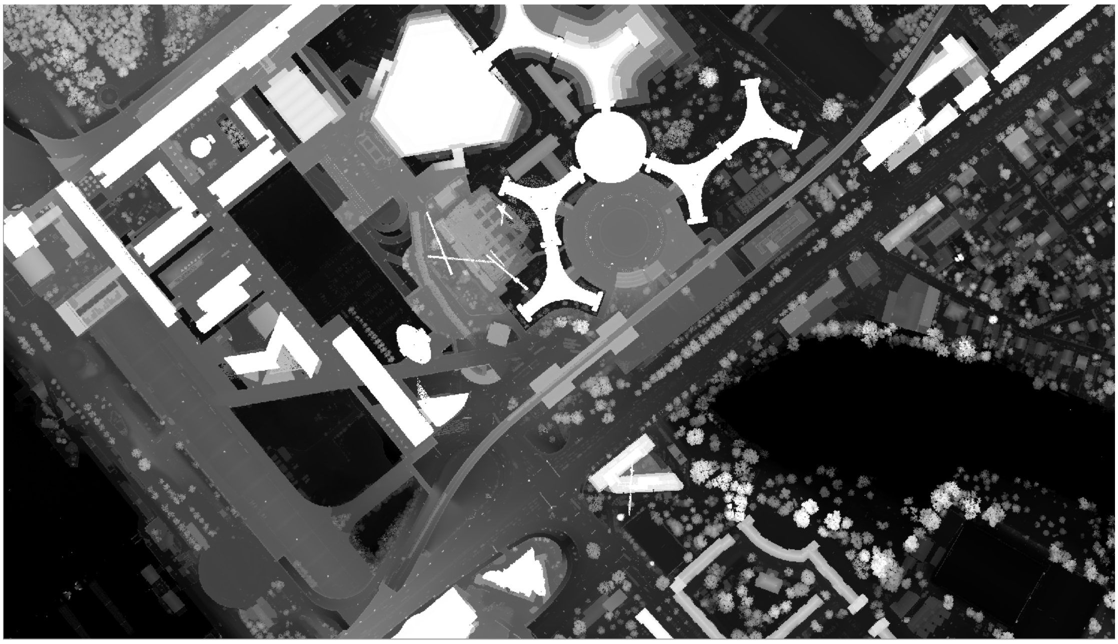


Figure 3.4: Part of the patch's 36/1 terrain elevation model as seen in QGIS after import.

3.3.3 Surface Elevation Model

The representation of building surfaces distinguishes the surface elevation model from the terrain elevation model. However the basic concept remains the same. The geodata viewer offers the download of the same ASCII files described in the previous section. By subtracting the terrain elevation raster from the surface elevation raster the height of each building can be calculated. Once the elevation models were downloaded I used Matlab to preprocess them with a median filter. I chose the median filter because of two well known properties. First of all the median filter is a widely used solution for noise removal. Secondly the median filter functions as an edge preserving filter, which means that applying it the elevation models will not influence the building's structure. Hence the median filter helps me to remove salt and pepper artifacts from the downloaded data as seen in Figure 2.12.

3.3.4 Orthophoto

While the elevation models described in the previous two sections are used for modelling the building structures the orthophotos are used for texturing the resulting 3D models. In this section I will briefly describe the orthophoto's main properties and why these properties qualify orthophotos as textures for georeferenced 3D models.

In [Law13] orthophotos are defined as non distorted aerial photographs of the earth's surface. Major causes for distortion and occlusion in aerial photographs are the sudden changes in the surface's elevation, the camera's lens and the camera's orientation. The

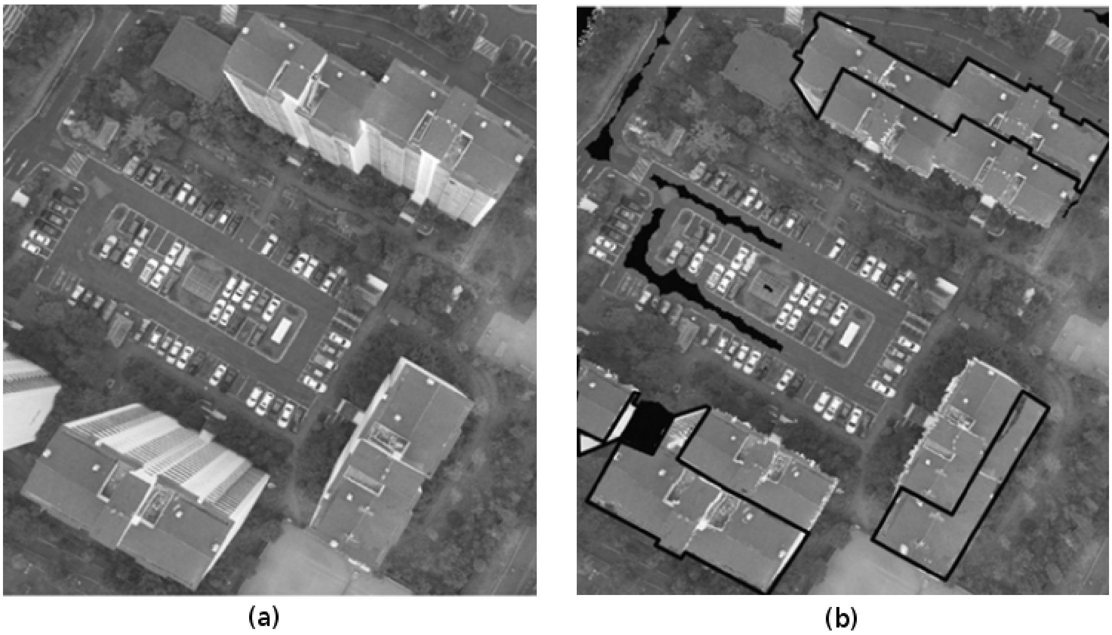


Figure 3.5: Here an aerial photography (a) can be compared with (b) an orthophoto of the same scene [HKK07].

process of removing these mentioned distortions is called ortho-rectification and its product is the orthophoto. In Figure 3.5 the reader can see how an orthophoto distinguishes itself from a simple aerial photograph. Creating an orthographic top-down projection of the earth's surface is not a trivial task. The algorithms applied to remove distortion from aerial photographies are explained in detail in [HKK07]. Because of the ortho-rectification's complexity I will not further describe it here, however the reader should have noted the difference between a simple aerial photograph and an orthophoto. Because the orthophoto represents an orthographic top-down projection of the earth's surface I simply applied it to the 3D models by transforming their two dimensional representations (UV-coordinates) with the same top-down projection, as can be seen in Figure 3.5. The orthophotos are also represented as raster information. As such the same concepts as for the elevation models apply. This means that every orthophoto comes with additional meta-information which has to be used to allocate each pixel of the image to a specific point on the earth's surface. In Vienna's geodata viewer this information was included in a .wld file which comes with the downloaded orthophoto. The .wld file contains the parameters for the image to world projection of the raster. As we will see in the next section most GIS will automatically apply the image to world transformation when importing orthophotos by reading the .wld file.

3.3.5 Combining Raster and Vector Data

Quantum GIS is a Python based framework which includes a multitude of tools for the evaluation and processing of georeferenced data. I chose QGIS over other proprietary frameworks because its tool set fully satisfied all my requirements, which were the support for the file types of the data downloaded from the geodata viewer, statistical evaluation of given areas over a raster and querying of the downloaded vector-data. As such it enables me to create an overlay containing all the vector-layer and elevation models of Vienna, as can be seen in Figure 3.6. In this section I will therefore describe how Quantum GIS can be used to combine the information from the elevation models with the vector data in a step-by-step guide.

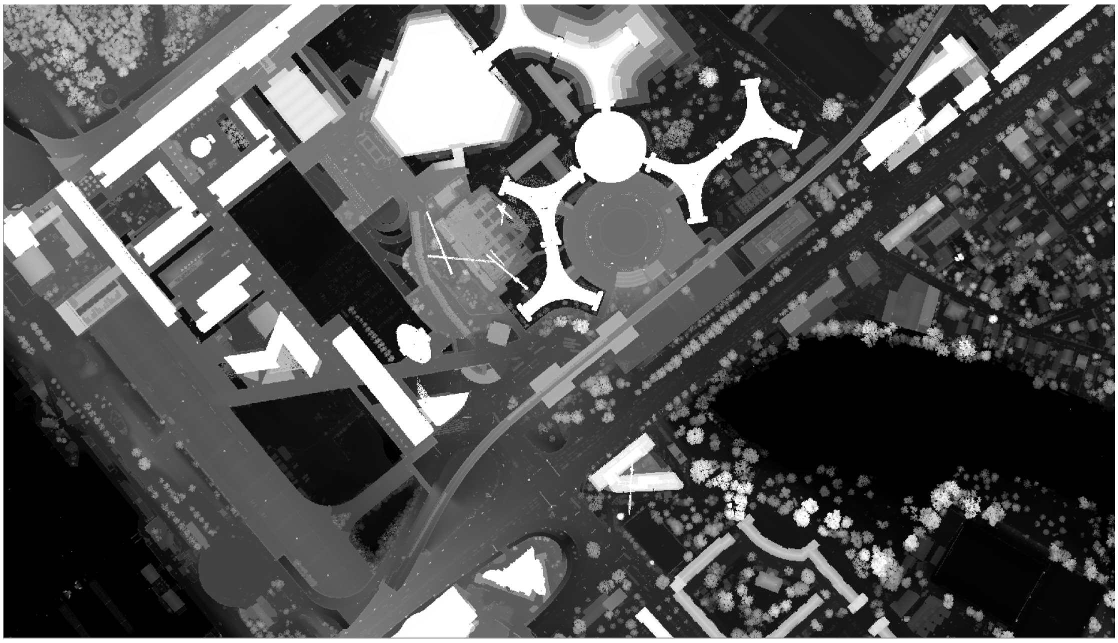


Figure 3.6: DSM as seen after import into QGIS.

The first step after importing the city's vector data, the terrain elevation model and the surface elevation model into a new QGIS project, was to remove the superfluous polygons from the vector-layer. Because the objective of my work is to create a simplistic 3D model of the city's skyline, I focus on the buildings alone. Therefore I remove everything that is not a building from the vector-layer, using the query tool in QGIS. In Figure 3.7 the resulting filtered vector-layer can be seen. Now every polygon in the vector-layer represents a building.

In the second step I let QGIS calculate the mean elevation over each polygon in the previously filtered vector-layer using its "Zonal Statistic" tool. This was done once for each elevation model. By executing this step the relative height of each polygons base (terrain elevation model) and roof (surface elevation model) is calculated and the result is added to the respective polygon in the vector-layer. Figure 3.8 can be observed to better

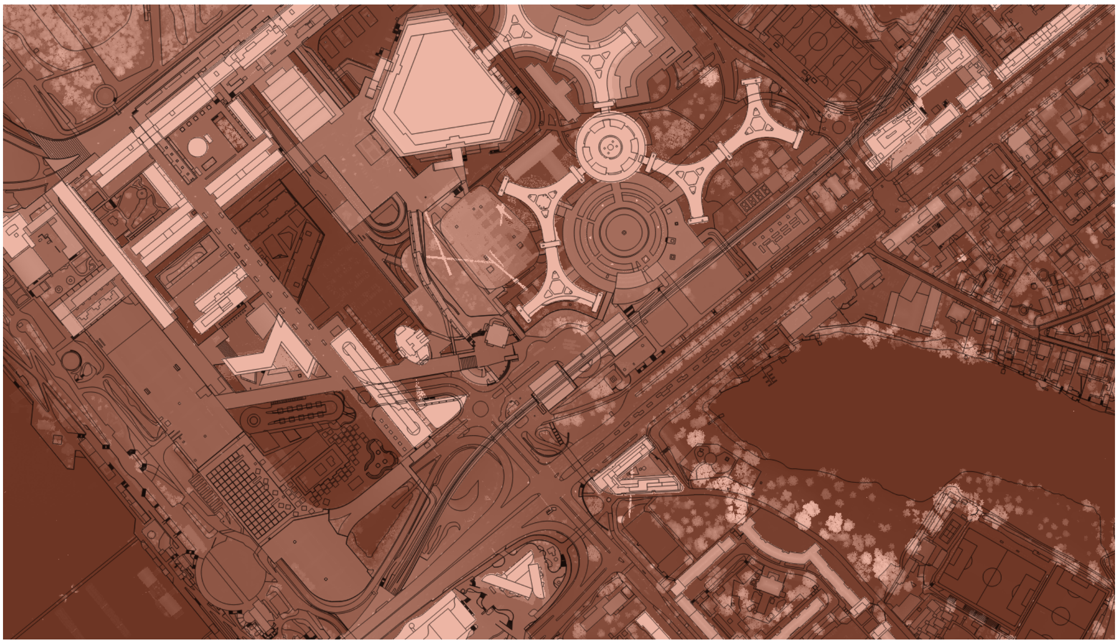


Figure 3.7: DSM and vector layer overlay as seen in QGIS.

understand this step. In the third step, again by using the QGIS query tool, I calculate the height of each building by subtracting the bases' elevation from the roof's elevation, which were calculated in the previous step. Again the result is added to each respective polygon's meta-information. In the fourth and final step I export the vector-layer with the newly calculated meta-information as a JSON file, which can easily be parsed by modern browsers using JavaScript.

3.3.6 Import GeoJSON into City Web Extruder

In this section I am going to describe in detail how I handled the georeferenced JavaScript notated object file, also called GeoJSON, by presenting 3 crucial JavaScript functions I implemented during the development of the City Web Extruder. These three functions are all used during the GeoJSON import and bring together ClipperJS, in order to simplify the two-dimensional data as much as possible, and ThreeJS, in order to extrude the polygons into three-dimensional models and visualize them. By following the steps described in the previous section I was able to combine the raster data describing the building's footprints with approximate values of the building's heights. GeoJSON is defined as a geospatial data interchange format which uses JSON to represent georeferenced data, in the structures that were described in Section 2.1.5, and their features (e.g. the buildings' heights) [BDD+16]. The JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format [Bra14].

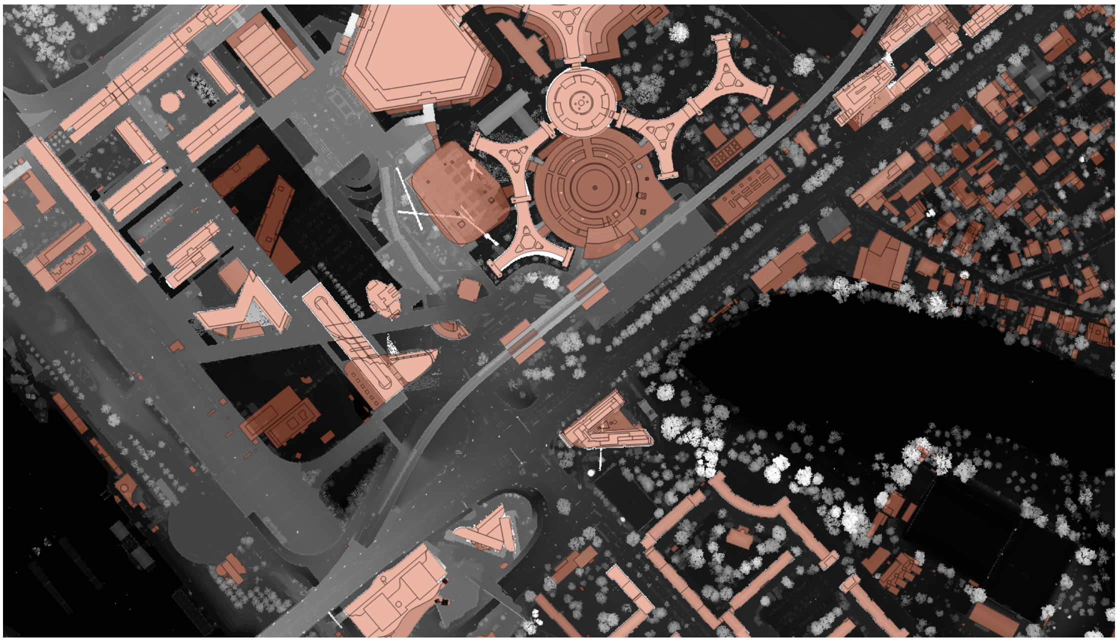


Figure 3.8: DSM with filtered vector layer overlay as seen in QGIS.

```
{
    "type": "FeatureCollection",
    "features": [{
            "type": "Feature",
            "geometry": {
                "type": "Polygon",
                "coordinates": [
                    [1.0, 0.0],
                    [2.0, 2.0],
                    [3.0, 0.0],
                    [1.0, 0.0]
                ]
            },
            "properties": {
                "Height": "5"
            }
        }]
}
```

Listing 3.1: Simplified example of a GeoJSON file describing a triangle polygon.

GeoJSON was chosen as the import format for the City Web Extruder, because it stores the polygons in an easily parsable format, since JSON is perfectly integrated into the JavaScript language. As can be seen in the example shown in Listing 3.1, the GeoJSON file stores GIS related information using JSON objects. The geometry type "Polygon" shown in the example GeoJSON on line 6 relates to the GIS data structures that have been explained in Section 2.1.5. The field "Height", found in the "properties" object, defined on line 9 in Listing 3.1, stores by how many units the polygon should be extruded. The listed GeoJSON has been reduced to a few properties to better explain the concept however in a real case scenario the GeoJSON would also store meta information, like the used projection system of the coordinates used to represent the polygon, as has been explained in Section 2.1.2, and different types of polygon as I will explain in the following paragraphs.The GeoJSON importer I wrote for the ThreeJS library first loads the GeoJSON file using the JavaScript native JSON parser, which can be seen on line 6 of Listing 3.2. After that the deserialized JSON information is passed as an object to the parser function, which maps the acquired information onto the ThreeJS data structures for further processing.

```
1  function ( file , onLoad , onProgress , onError ) {
2      var scope = this;
3      scope.reader.onprogress = onProgress;
4      scope.reader.onError = onError;
5      scope.reader.onload = function (e) {
6        var obj = JSON.parse(scope.reader.result);
7
8        onLoad( scope.parse( obj ) );
9      };
10     scope.reader.readAsText(file);
11   }
```

Listing 3.2: JavaScript function which parses the data from a GeoJSON file.

The parser function shown in Listing 3.3 iterates over all the polygons listed in the GeoJSON file. Depending on the defined polygon type the program simply maps the data or applies additional operations before creating a ThreeJS "Shape" object out of the two-dimensional information stored in the GeoJSON file. In the ThreeJS documentation "Shape" is described as a structure which defines an arbitrary 2d shape using paths with optional holes, thus perfectly fitting for the intended use. However note that depending on the defined polygon type the base geometry is handled differently [Cab]. In the first case the handled polygon type is defined as "Polygon" at line 14. In this case the polygon data is directly mapped onto the ThreeJS "Shape" structure. However in the second case, which is the mapping of the "MultiPolygon" polygon type, another step is adopted before mapping the information onto the ThreeJS "Shape" structure. In this case the building's footprint is described using multiple polygons. Here is were the ClipperJS framework comes into play in order to perform clipping operations on these multi-polygons.

```
function ( obj ) {
  var meta = obj.type;
  var crs = obj.crs.properties.name;
  var buildings = obj.features;
  var totalGeometry = new THREE.Geometry();
  console.log(buildings);
  for (var building of buildings){

    var surface = building.properties.SURFACE;
    var terrain = building.properties.TERRAIN;
    var extrudeSettings = {
      amount: building.properties.HEIGHT,
      bevelEnabled: false
    };
    var coordinates = building.geometry.coordinates;

    if ( building.geometry.type === 'Polygon' ){
      var shapes = this.parsePolygon(coordinates);
      for ( var shape of shapes ) {
        var new_geometry = new THREE.ExtrudeGeometry(
          shape,
          extrudeSettings
        );
        totalGeometry.merge(
          new_geometry,
          new_geometry.matrix
        );
      }
    }else if ( building.geometry.type === 'MultiPolygon' ){
      for ( var polygons of coordinates){
        var shapes = this.parsePolygon(polygons);
        for ( var shape of shapes ) {
          var new_geometry = new THREE.ExtrudeGeometry(
                      shape,
                      extrudeSettings
                    );
                     totalGeometry.merge(
                        new_geometry,
                        new_geometry.matrix
                      );
                 }
      }
    }
  }

  var mesh = new THREE.Mesh( totalGeometry, new THREE.MeshNormalMaterial() );
  mesh.geometry.computeBoundingSphere();
  var center = mesh.geometry.boundingSphere.center;
  mesh.geometry.translate( -center.x, -center.y, -center.z );
  mesh.scale.set( 0.1, 0.1, 0.1 );

  var group = new THREE.Object3D();//create an empty container
  group.add( mesh );//add a mesh with geometry to it

  return group;
}
```

Listing 3.3: JavaScript function mapping the data from a GeoJSON onto the ThreeJS data structures.

In Listing 3.4 the application of ClipperJS can be seen from line 4 to line 20. Here ClipperJS is used to apply the Union operation between the accumulated polygons in order to avoid overlapping of multiple polygons, which would cause glitches as has been explained in Section 2.3. After the clipping the function maps the data on the ThreeJS "Shape" structure as I mentioned in the previous paragraph. The height found in the properties of each polygon in the GeoJSON file is used as an extrusion setting as can be seen on line 14 of the Listing 3.3. On lines 20 and 30 the extrusion settings are then passed to the ThreeJS "ExtrudeGeometry" function in addition to the base shape of the building. The result of the extrusion is a geometry structure, which stores vertices, faces and colors of a three-dimensional model. Each extruded polygon is iteratively accumulated in the total geometry variable on the lines 21 and 31. Finally the total geometry is used to created a ThreeJS "Mesh" object which represents triangular polygon mesh based objects [Cab]. The "Mesh" object's origin point is also moved to the center of the scene. Note that this translation transforms the coordinates. This means that the three-dimensional model loses it's georeference. In the show case presented in this work the georeference is not that important after the extrusion, because the different city patches can be reassembled by using the file's name as has been described in Section 3.3.1. In this section I described how the GeoJSON is handled in the City Web Extruder. By presenting three JavaScript functions, crucial to the import process, I wanted to show how the frameworks ThreeJS and ClipperJS were used together in the presented framework. For an in depth explanation of the theories implemented by these two frameworks I suggest reading Section 2.4 and Section 2.5. By reading this section the reader should have gained a first impression of the City Web Extruders's functionality. This section also concluded the Methodology chapter. This chapter described how I adopted the theories and frameworks presented in the previous chapter. By reading this chapter the reader should be able to get an introspective view of the City Web Extruder. In addition the reader should be able to replicate the methodology in order to further develop the presented concepts and ideas for his/her own projects. In the next chapter I am going to present and discuss the results of the City Web Extruder, their integration into a game engine and how they perform in a virtual reality scene.

```
function( coordinates ){
  var subj = [];
  var clip = [];
  var c = new ClipperLib.Clipper();
  for(var i = 0; i < coordinates.length; i++) {
   for ( var point of coordinates[i] ) {
    if( i < 1 ){
     subj.push(new ClipperLib.IntPoint(point[0],point[1]));
    }else{
     clip.push(new ClipperLib.IntPoint(point[0],point[1]));
    }
   }
   if ( clip.length > 0 ) {
    c.AddPath(subj, ClipperLib.PolyType.ptSubject, true);
      c.AddPath(clip, ClipperLib.PolyType.ptClip, true);
      c.Execute(ClipperLib.ClipType.ctUnion, subj);
        clip = [];
   }
  }

  shapes = [];
  shp_points = [];

  for( var r of subj ){
   if ( r instanceof ClipperLib.IntPoint){
    shp_points.push(new THREE.Vector2(r.x, r.y));
   }else{
    for ( var point of r ){
     shp_points.push(new THREE.Vector2(point.x, point.y));
    }
    shapes.push(new THREE.Shape(shp_points));
    shp_points = [];
   }
  }

  if ( shp_points.length > 0 ){
   shapes.push( new THREE.Shape(shp_points) );
  }

  return shapes;
```

Listing 3.4: JavaScript function mapping a list of points onto the ThreeJS "Shape" data structure.

CHAPTER 4

Results

By implementing the approaches described in the previous chapter a web framework was built that is able to create three-dimensional models of buildings by parsing GeoJSON files. These files contain the two-dimensional information of the buildings' blueprints and their respective heights. In Section 4.1 I am going to describe this resulting framework, how the produced models were textured and how they were imported into a game engine in order to use them in a virtual reality scene. The texturing of the resulting three-dimensional models goes beyond the scope of my thesis, however in Section 4.2 I will describe a fast and efficient method, in order to create a more realistic virtual reality scene.

4.1 City Web Extruder

The City Web Extruder is the framework which was implemented during the course of my master thesis and presented here. It is based on Javascript and WebGL for the automated extrusion of polygons representing buildings' footprints into three-dimensional models. The City Web Extruder offers an import for GeoJSON files, which store georeferenced polygons, representing building blueprints, as described in Section 3.3.6. Once uploaded the file is parsed and the data is processed in order to assign the height to each building's polygon using the ClipperJS library [Joh] and the importer which was specifically written for the import of GeoJSON files in order to use the ThreeJS framework for further processing. During this step the data is read from the file and mapped to the ThreeJS's data structures in order to apply the extrusion functions to the polygons.

Once finished the center of mass for the resulting model is calculated. After that the scene center and the model's origin are both positioned to the calculated center of mass. This step eases the visualization of the resulting model in the browser. By implementing this step I enabled rotation and zoom controls around the created model. The user can use these basic controls to check the resulting model before exporting it. Figure 4.1 shows

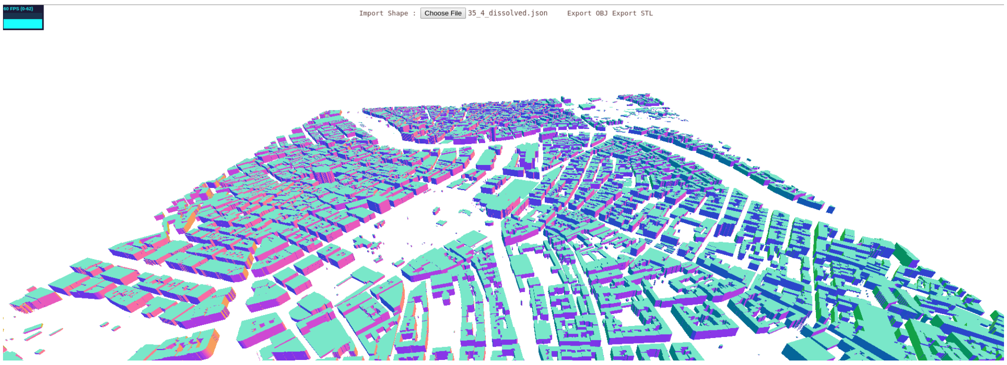


Figure 4.1: View the user is confronted with after successfully importing a GeoJSON in the City Web Extruder.

the described use case. The colors of the models that can be seen in the figure describe the orientation of the faces relative to the camera. Notice how the roofs of the buildings all have the same color. Due to the extrusion process all faces representing roofs look in the same direction. Therefore all the roofs are represented by the same color which is the result of mapping the orientation vector of the face onto the faces RGB values. The ThreeJS library offers a material which can be applied on the model in order to created this visualization automatically. The reader should keep in mind that all roofs of the extruded geometry look in the same direction. In the next section a categorization by this feature is used in order to easily apply textures to the model.

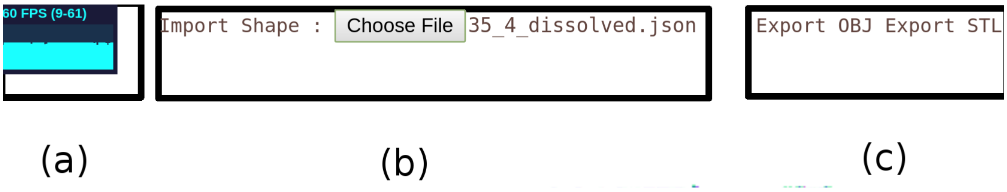


Figure 4.2: Status overview of the City Web Extruder, showing the frames per second (a), the loaded GeoJSON file (b) and the export options (c).

The ThreeJS library also natively offers the .stl and .obj file export implementations. Both file types can be used for the import into any arbitrary game engine. In Figure 4.2 the Web City Extruder handles the rendering of a 6,25 qm patch of Vienna at constant 60 FPS due to the simplified geometry. Even though at this point the textures are still missing they will not cost much performance once integrated in the game engine. However I am going to discuss how the textures were applied to the three-dimensional models in the next section.

As I briefly mentioned before there are similar frameworks which can create three-dimensional models out of blueprints. However at the time none were solving this problem free of cost and as an online service. ArcGis [arc] and QGis [qgi] are probably the most notable alternatives. Both frameworks bundle multiple features needed for the evaluation of georeferenced data. ArcGis is the commercial alternative, while QGis is an open source project. While QGis lacks features when compared to ArcGIS (e.g. export of three-dimensional models and online service variant), the City Web Extruder synergizes very well with QGis, as it can offer an online interface to QGis's functionality. Additionaly the City Web Extruder expands QGis by the three-dimensional model export while using its ability to combine the raster and vector data as has been described in the previous chapter.

4.2 Wall Texturing

In this section I am going to describe how I used the orthophotos to apply textures to the city models produced using the Web City Viewer. This step was necessary in order to create a credible and realistic scene in the virtual reality environment, and may be automated in the future. However in the current framework the texturing is is not part of the automated process. The approach I adopted to texture the models uses the simplicity of their geometry. Due to the models being extrusions of two dimensional polygons the orientation of their faces can only fall into one of the three following categories:

- Down-looking faces
- Up-looking faces
- Faces' orientation is parallel to the floor

Figure 4.1 not only shows the City Web Extruder in action, but also visualizes the three variations of the polygons' faces orientation. Nowadays most computer graphics software suites implement a function to select faces by their orientation. To implement this step I used the Blender graphics suite. Blender offers the possiblity to select these three different face types separately. By doing so I was able to categorize them as floor (down-looking faces), roof (up-looking faces) and walls (faces' orientation is parallel to the floor). This categorization is crucial, because floors, roofs and walls are all texturized in different ways. In this case the floor and roofs are handled the same way, because they are applied in an outdoor visualization. The building's interior is neglectable for the same reason. However for the texturing of the floor and roofs Blender was used again to project the orthophoto onto the models using an orthographic top-down projection, as Figure 4.3 demonstrates. Notice how this process replicates the mechanics used to produce the orthophoto, as has been described in Section 3.3.4. Therefore the orthographic top-down projection of the model perfectly matches the orthophoto, which means that the result of the projection can be used as UV-Map of the orthophoto resulting in a perfect texturing of the roofs and terrain.

The remaining walls were textured using a similar approach. In this case UV-coordinates were projected using a front view projection. The resulting UV-coordinates distort the

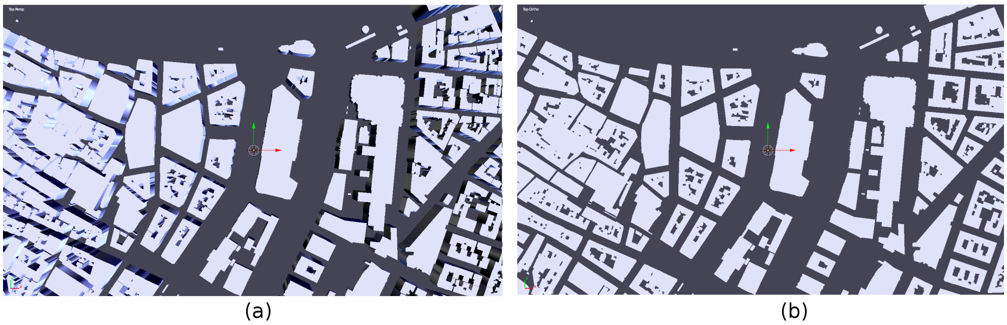

(a) (b)

Figure 4.3: A perspective top-down view on the three-dimensional model (a) and an orthographic top-down projection (b) of the same model using Blender.

texture along curved walls but the defects were not recognized by the users. In Figure 4.4 the final result can be seen with some textures applied onto it. The scene renders Vienna's center using a textured model created with the City Web Extruder however some higher level-of-detail models where used in order to increase the recognizability of Vienna's more picturesque sites. Therefore the Stephan's Cathedral in the middle of the scene is a handmade model.

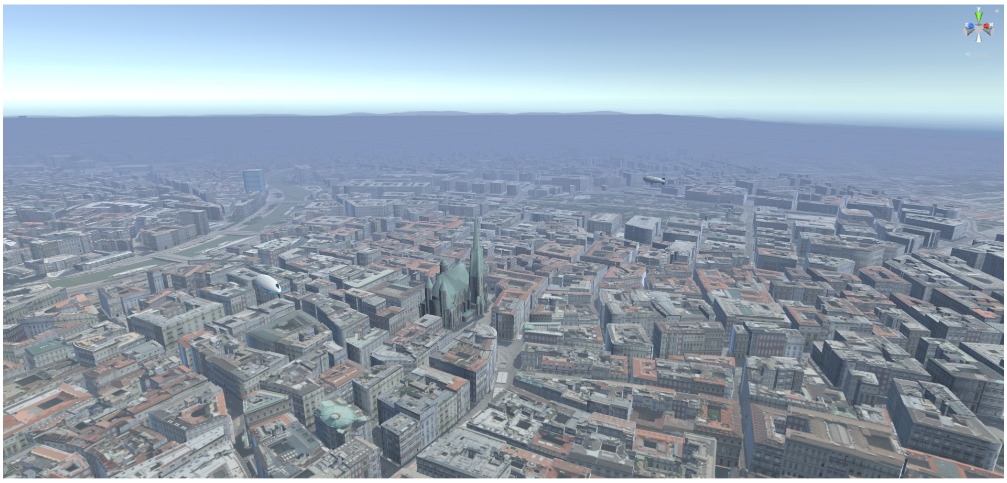


Figure 4.4: The textured models used in a scene which was created using Unity.

By following the steps described in this section the results of the City Web Extruder can be textured in a fast and easiliy automatable process, however depending on the context the models are going to be applied in their distortions on curved side walls, which can be unsatisfactory. Implementing a procedural approach for the textures of the buildings' facades would be ideal, but goes beyond the frame of this work. Due to the main focus being the efficient rendering of a city in a virtual reality environment the texturizing of the side walls was not a crucial feature. In the next section I am going to show how the models performed once integrated in a game engine and show the final results as the models were used for [EM15].

4.3 Game Engine Integration

In this section I am going to discuss how the three-dimensional models performed once integrated in a game engine and compare them to the models which are processed for the integration into CAD programs instead of game engines. These files can be downloaded in the *.dwg file format from Vienna's geodata viewer [Mag16]. The models stored in the *.dwg files are based on the same data as the models created using the City Web Extruder. Comparing these files will show that the processes which produced them implemented different constraints thus offering areas where one performs better than the other. The game engine chosen for the comparison was Unity, the same engine used for the creation of the virtual reality scene in [EM15]. The downloadable models come without any texture therefore the comparison is done without taking them into account for both models. Therefore except for the models both scenes are exactly the same, containing one-directional light for the lighting and the default procedural skybox offered by Unity.

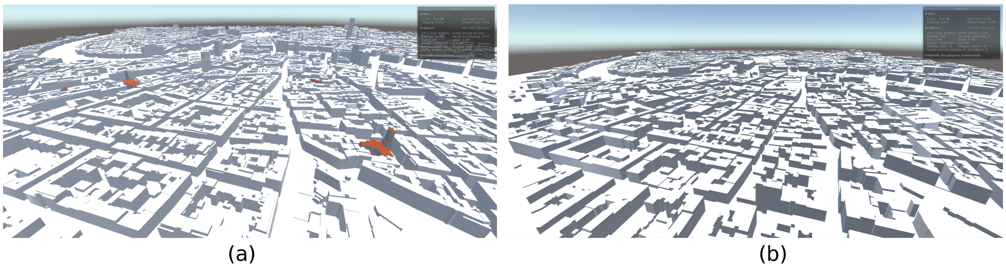


Figure 4.5: Vienna's city center rendered using the downloadable models from the geodata viewer (a) and the same scenery visualized using the models created using the presented framework (b).

Figure 4.5 shows the rendered images using the implemented direct comparison of the models. Both models contain the exact same buildings in a 6.25x6.25 kilometers quadrant around Vienna's city center. The City Web Extruder results miss details on roofs due to the union operations on the polygons of the same building, however the rendered scene (b) runs at 82.8 FPS compared to the 64.2 FPS achieved by geodata viewer's scene (a). Joining all the buildings into one object also drastically reduced the draw calls from as scene (a) uses up to 31 draw calls while scene (b) only uses 15 draw calls per rendered frame, which also increases the performance. Finally I also compared the size of the models. The geodata viewer's model is stored in a *.fbx file of 340 MB while the model created using the City Web Extruder has a size of 14.6 MB once also converted into a *.fbx file using Blender for the comparison.

Finally the figures 4.6 and 4.7 show the city models in action as seen by the user of [EM15]. The reader may notice some additional models and effects (e.g. zeppelin, images, day-night cycle, fireworks, ... etc.) in the shown images that were added to the scene for story telling purposes.

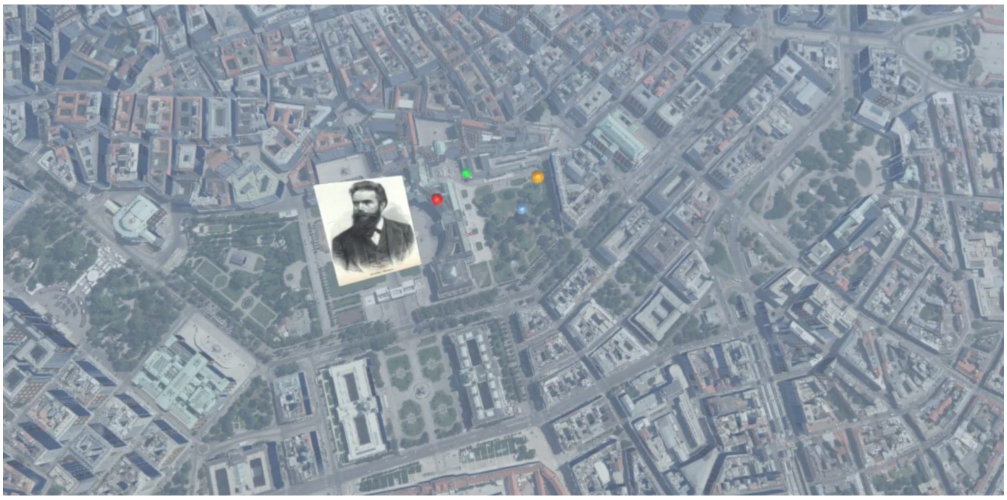


Figure 4.6: The textured models as applied in the skydiving simulation described in [EM15].

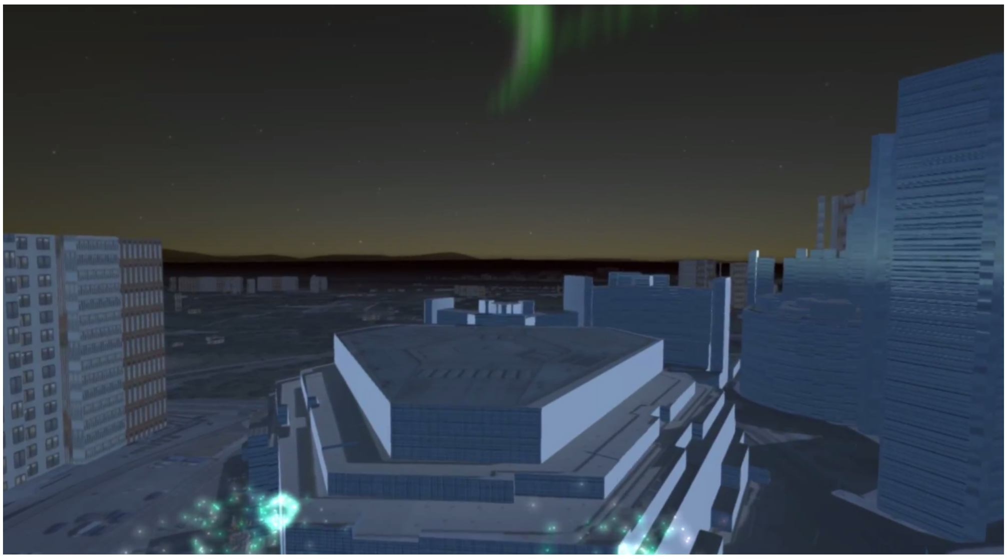


Figure 4.7: The textured models as seen during the landing phase of the skydiving simulation described in [EM15].

CHAPTER 5

Conclusion

The research presented in this master thesis examined the possibilities to create three-dimensional models of Vienna's buildings using freely accessible georeferenced data. Additionally the question whether the resulting models could be used in a virtual reality oriented scene was answered. The resulting framework, a web application capable of creating three-dimensional models out of a GeoJSON file storing building footprints and their respective heights, has been documented in this work. By doing so it has been shown that it is fully possible to recreate a full city model of Vienna by using free data and without the use of any commercial service. Additionally the results showed that using the three-dimensional buildings created using the Web City Extruder instead of the prefabricated models or simple surface interpolations of the DSM, drastically reduced the models size and improved the performance. The created models may have a lower level-of-detail, when compared to the models created using commercial alternatives or CAD suites, however this loss of detail was accepted during the implementation of the presented framework in order to further improve the performance for a virtual reality context. The loss of detail is caused by performing a union operation over all polygons of the same buildings, which creates a single polygon for each building, but removes the possibility to visualize different layers of the same building. The presented framework serves as a proof of concept and can be further improved by adopting smarter algorithms for the buildings' extrusion and polygon clipping. Procedural texturing of the building's facades and automated roof creation are topics that expand the presented framework reasonably, but would have gone beyond the scope of this work.

The main focus of this work lied on the optimization of the model's geometry in order to save computational power and reduce error-proneness during rendering for a virtual reality experience. The concept has been further expanded by the idea of implementing the service as a web application using the relatively young WebGL technology. Thus the presented framework offers the first non commercial alternative for three-dimensional model creation using georeferenced data. The City Web Extruder has opened a number of development and research possibilities. For instance, in the future it may be possible to fully automate the process. Additionally the framework could be expanded to a full blown service-oriented web platform capable of processing the data on the client side by using the client's graphics card through WebGL. This way the workload on the servers could be considerably reduced, making the whole web platform easily scalable. From there on it would be possible to iteratively add services and features to the platform. For example, procedural facade and roof creation could be added to the framework's pipeline. Until now the City Web Extruder has been used to easily create scenes for experiments on virtual reality and motion sickness in a fast and easy way [EM15]. The resulting scenes have also been applied in simulations which were presented during public events. Still the City Web Extruder was mostly used by informatics students, however by further automating and simplifying the whole process the framework could be used by users of different disciplines like urban planners or architects, which often have the need for three-dimensional models like for instance scene visualizations, lighting simulations and flooding simulations [ZSP][DH07][RV01].

List of Figures

1.1 A patch of Vienna rendered in the browser using the presented framework. . 2
1.2 3D model used for a light simulation as seen in [ZSP]. 2
1.3 A simple buildings footprint created using GIS data. 4

2.1 John Snow's map overlay. 8
2.2 GPS data overlaid on a simple map. 10
2.3 Unwrapping a cylindrical projection [Law13]. 11
2.4 The UTM raster from [Dom]. 11
2.5 Unwrapping a conic projection [Sny87]. 12
2.6 From unprojected to projected coordinates [Law13]. 13
2.7 Geoid describing the Earth's surface from [NASa]. 14
2.8 Simple visualization of Airborne Laser Scan (ALS) data acqusition [Mag16]. . 15
2.9 DEM of Vienna visualized as a grayscale image. 16
2.10 The basic structures used for all geometries in GIS. 17
2.11 A screenshot taken from the online tool to download GIS data for Vienna [Mag16]. 19
2.12 Imperfections of DSM's showing a helicopters rotor in the final result. 20
2.13 Difference between DTM and DSM [Mar]. 21
2.14 Example of DSM from the St. Othmar church in Vienna. 21
2.15 Example of DTM of same area seen in Figure 2.14. 22
2.16 Textured surface created by interpolating raster data of DEM. 23
2.17 Three-dimensional surface which can be created using a DSM. [BRBM10] . . 24
2.18 Gray scale image of the world's DEM. [NASb] 25
2.19 QGIS overlay of the DSM and the building footprints. 26
2.20 Showing z-fighting in *Blender 2.5* (left) and *Google SketchUp 8* (right) from [VF13]. 27
2.21 Visualization of the view frustum between clipping planes [Rea16]. 28
2.22 Depth of the closest object to each pixel is stored in the z-buffer [LKR]. . . . 29
2.23 Example showing difficulties of overlapping polygons [RF00]. 30
2.24 Simplified overlapping polygons using union from [RF00]. 31
2.25 Illustration of the winding number (ω) as shown in [GH98]. 32
2.26 The polygons in (a) represent a building's footprint which is extruded in (b) and (c) shows the overlapping of two common sidewalls [LM11]. 32

2.27 Boundary representation of different polygons comparing Node-Edge-Face structure (left) and triangulated structure (right). 34
2.28 Structure of a WebGL application from [FWLL11]. 35
2.29 Rectangle selection over buildings (left) and the resulting selection (right) from [MWE14]. 36

3.1 Marked section is covered by LOD3 as seen in the geodata viewer for Vienna. 39
3.2 Marked section is covered by LOD2 as seen in the geodata viewer for Vienna. 40
3.3 Marked section is covered by LOD1 contained in the LOD2 patch with index 36/1. 41
3.4 Part of the patch's 36/1 terrain elevation model as seen in QGIS after import. 42
3.5 Here an aerial photography (a) can be compared with (b) an orthophoto of the same scene [HKK07]. 43
3.6 DSM as seen after import into QGIS. 44
3.7 DSM and vector layer overlay as seen in QGIS. 45
3.8 DSM with filtered vector layer overlay as seen in QGIS. 46

4.1 View the user is confronted with after successfully importing a GeoJSON in the City Web Extruder. 52
4.2 Status overview of the City Web Extruder, showing the frames per second (a), the loaded GeoJSON file (b) and the export options (c). 52
4.3 A perspective top-down view on the three-dimensional model (a) and an orthographic top-down projection (b) of the same model using Blender. . . . 54
4.4 The textured models used in a scene which was created using Unity. 54
4.5 Vienna's city center rendered using the downloadable models from the geodata viewer (a) and the same scenery visualized using the models created using the presented framework (b). 55
4.6 The textured models as applied in the skydiving simulation described in [EM15]. 56
4.7 The textured models as seen during the landing phase of the skydiving simulation described in [EM15]. 56

List of Tables

2.1 List of files of the SHP standard [Ins98]. 18

3.1 Example of header for a georeferenced ASCII file. 41

Acronyms

ALS Airborne Laser Scan. 13, 14, 20, 24

DEM Digital Elevation Model. 14, 16, 20–25, 59

DSM Digital Surface Model. 21, 22, 24, 26, 59

DTM Digital Terrain Model. 21, 22, 59

GIS Geographic Information Systems. 7, 9, 13, 17, 19, 22, 25, 28, 32, 33, 35, 59

GPS Global Position Systems. 9, 10, 12, 59

HMD Head Mounted Display. 37, 38

OGC Open Geospatial Consortium. 9, 19

OSM Open Street Map. 9

QGIS Quantum Geographic Information System. 22, 24, 26, 44, 59

WebGL Web Graphics Library. 34, 35

WGS World Geodetic System. 12

Bibliography

[Ago05] Max K Agoston. *Computer graphics and geometric modeling*, volume 1. Springer, 2005.

[arc] Arcgis. https://www.arcgis.com/features/index.html [Accessed: June 2017].

[BDD$^+$16] Howard Butler, Martin Daly, Allan Doyle, Sean Gillies, S Hagen, and T Schaub. The geojson format. Technical report, 2016.

[Bra14] Tim Bray. The javascript object notation (json) data interchange format. 2014.

[BRBM10] CARLO Baroni, ADRIANO Ribolini, GIUSEPPE Bruschi, and Paolo Mannucci. Geomorphological map and raised-relief model of the carrara marble basins, tuscany, italy. *Geografia Fisica e Dinamica Quaternaria*, 33(2):233–243, 2010.

[C$^+$15] PostGIS Project Steering Committee et al. Postgis documentation, 2015.

[Cab] Ricardo Cabello. Three.js homepage. https://threejs.org/ [Accessed: May, 2017].

[Cha14] Kuang-Hua Chang. *Product Design Modeling using CAD/CAE: The Computer Aided Engineering Design Series.* Academic Press, 2014.

[DH07] J Döllner and Benjamin Hagedorn. Integrating urban gis, cad, and bim data by servicebased virtual 3d city models. *R. e. al.(Ed.), Urban and Regional Data Management-Annual*, pages 157–160, 2007.

[Dom] Public Domain. The longitude and latitude zones in the universal transverse mercator system. URL: http://goo.gl/5CMBIk [Accessed: August, 2016].

[EM15] Horst Eidenberger and Annette Mossel. Indoor skydiving in immersive virtual reality with embedded storytelling. In *Proceedings of the 21st ACM Symposium on Virtual Reality Software and Technology*, pages 9–12. ACM, 2015.

[FWLL11] Lei Feng, Chaoliang Wang, Chuanrong Li, and Ziyang Li. A research for 3d webgis based on webgl. In *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, volume 1, pages 348–351. IEEE, 2011.

[GH98] Günther Greiner and Kai Hormann. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics (TOG)*, 17(2):71–83, 1998.

[HKK07] Ayman F Habib, Eui-Myoung Kim, and Chang-Jae Kim. New methodologies for true orthophoto generation. *Photogrammetric Engineering & Remote Sensing*, 73(1):25–36, 2007.

[HW08] Mordechai Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 2008.

[Ins98] Environmental Systems Research Institute. Shapefile technical description. *http://www.esri.com*, 1998.

[Joh] Angus Johnson. The clipper library. URL: http://goo.gl/RpfDB4 [Accessed: September, 2016].

[Law13] Joel Lawhead. *Learning Geospatial Analysis with Python*. Packt Publishing Ltd, 2013.

[LB83] You-Dong Liang and Brian A Barsky. An analysis and algorithm for polygon clipping. *Communications of the ACM*, 26(11):868–877, 1983.

[LKR] Ph.D. Loren K. Rhodes. z-buffer algorithm. URL: http://goo.gl/Ju3qmy [Accessed: August, 2016].

[LM11] Hugo Ledoux and Martijn Meijers. Topologically consistent 3d city models obtained by extrusion. *International Journal of Geographical Information Science*, 25(4):557–574, 2011.

[Mag16] Stadtvermessung Wien Magistratsabteilung 41. Geodatenviewer. *URL: https://goo.gl/vtTavi*, January 2016.

[Mar] MartingOver. Surfaces represented by a digital surface model and digital terrain model.

[Mar11] Chris Marrin. Webgl specification. *Khronos WebGL Working Group*, 2011.

[Mica] Microsoft. Direct3d primitives documentation. URL: http://goo.gl/GMh1Tp [Accessed: September, 2016].

[Micb] Microsoft. Direct3d primitives documentation. URL: http://goo.gl/GMh1Tp [Accessed: September, 2016].

[MKK$^+$14] Finian Mwalongo, Michael Krone, Grzegorz Karch, Michael Becher, Guido Reina, and Thomas Ertl. Visualization of molecular structures using state-of-the-art techniques in webgl. In *Proceedings of the 19th International ACM Conference on 3D Web Technologies*, pages 133–141. ACM, 2014.

[MWE14] James Milner, Kelvin Wong, and Claire Ellul. Beyond visualisation in 3d gis. 2014.

[NASa] NASA. Gravimetry map from the gravity recovery and climate experiment—grace, a joint mission of nasa and the german aerospace center. URL: http://goo.gl/54rrFZ [Accessed: August, 2016].

[NASb] NASA. Gravimetry map from the gravity recovery and climate experiment—grace, a joint mission of nasa and the german aerospace center. URL: http://https://goo.gl/FfZGZE [Accessed: May, 2017].

[Ope] OpenStreetMap. Openstreetmap data to postgresql converter. URL: https://github.com/openstreetmap/osm2pgsql [Accessed: August, 2016].

[PS12] Emil Persson and Avalanche Studios. Creating vast game worlds: Experiences from avalanche studios. In *ACM SIGGRAPH 2012 Talks*, page 32. ACM, 2012.

[qgi] Quantum gis. https://www.qgis.org/en/site/ [Accessed: June 2017].

[Rea16] Reallusion. Clipping planes of the camera. *URL: https://goo.gl/94UeWz*, August 2016.

[RF00] Marilina Rivero and Francisco R Feito. Boolean operations on general planar polygons. *Computers & Graphics*, 24(6):881–896, 2000.

[RV01] Ismo Rakkolainen and Teija Vainio. A 3d city info for mobile users. *Computers & Graphics*, 25(4):619–625, 2001.

[SH74] Ivan E Sutherland and Gary W Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[Sny87] John Parr Snyder. *Map projections–A working manual*, volume 1395. US Government Printing Office, 1987.

[TBB01] James A Thompson, Jay C Bell, and Charles A Butler. Digital elevation model resolution: effects on terrain attribute calculation and quantitative soil-landscape modeling. *Geoderma*, 100(1):67–89, 2001.

[TM98] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839–846. IEEE, 1998.

[Tom74] Roger F Tomlinson. *The Application of Electronic Computing Methods and Techniques to the Storage, Compilation and Assessment of Mapped Data.* PhD thesis, University College London (University of London), 1974.

[Vat92] Bala R Vatti. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63, 1992.

[VF13] Andreas-Alexandros Vasilakis and Ioannis Fudos. Depth-fighting aware methods for multifragment rendering. *IEEE transactions on visualization and computer graphics*, 19(6):967–977, 2013.

[Wes10] Erik Westra. *Python geospatial development.* Packt Publishing Ltd, 2010.

[WL99] Aloysius Wehr and Uwe Lohr. Airborne laser scanning—an introduction and overview. *ISPRS Journal of photogrammetry and remote sensing*, 54(2):68–82, 1999.

[ZSP] Peter Zeile, Ralph Schildwächter, and Tony Poesch. 3d-stadtmodell-generierung aus kommunalen geodaten und benutzerspezifische echtzeitvisualierung mithilfe von game-engine-techniken.