DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Design and Implementation of a System for Visualizing Common 2D and 3D Geospatial Data in a Real-Time 3D Engine

Paul Pernsteiner

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Design and Implementation of a System for Visualizing Common 2D and 3D Geospatial Data in a Real-Time 3D Engine

Design und Implementierung eines Systems zur Visualisierung von verbreiteten 2D und 3D Geodaten in einer Echtzeit-3D-Engine

Author:	Paul Pernsteiner
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Sven Liedtke, M.Sc.
Submission Date:	October 15, 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, October 15, 2021 Paul Pernsteiner

Acknowledgments

Thank you Sven Liedtke for giving me the opportunity to work and grow on this challenging project.

Thank you Professor Klinker for your commitment and support of all students throughout the years.

Thank you Daniel Dyrda for your honest answers and help regarding the topics and initialising the contact to Sven.

Additional thanks to my family including my neighbor Sven as well as Patrik and Johannes for your support and advice during this time.

Abstract

With more geospatial data becoming available each year, the demand for interactive 2D and 3D visualization solutions grows. This includes the still in-development TallShipEngine, whose developers plan to implement a specialized interactive 3D GIS for all device types supported by the engine.

With this thesis, we design the basics of such a system. It provides capabilities for visualizing common geodata sources of different spatial reference systems in the same environment and focuses on creating an easily extendable layer structure with a low impact on the engine's real-time performance. The resulting implementation allows adjustable triangulated and line-based mesh creation for vector data, channel conversion for raster data as well as object creation for POIs.

The work presented in this thesis could be used as a starting point for the implementation of a specialized interactive visualization for a selected use-case with a fixed set of geodata requirements. The idea of a system for fire department coordination was pursued throughout this thesis, hence the focus on German geodata sources and scene preparation in advance.

It was found, that an advanced understanding of any used geodata is required to implement optimized visualizations of this data.

Contents

1. Introduction

In the recent worldwide pandemic, the "*GIS [Geographic Information System] use for COVID-19 has been the most comprehensive and effective one to date*" [23]. Many different regional and global interactive online maps were used by the population to monitor the latest infection trend. In such an overview, regions can be compared by difference in color or symbol size. When selecting a region, its current statistics and past developments are shown. Additionally, a growth is expected for the 3D geospatial market [14] which also results in an increasing demand for customized visualization solutions.

The significant increase of employees working from home showed an additional need for collaborative infrastructure and frameworks in general [8]. According to Sun and Li [29], this is an ongoing research area of the geographic sector in the form of real-time collaborative GIS, which show high potential in application scenarios for urban project planning specialists, emergency operations or public meetings.

In this thesis, a system for visualizing common geodata formats is designed and implemented using the TallShipEngine. This 3D real-time engine was created to support a variety of device types in a collaborative environment, examples include: interactive touch tables, Augmented Reality (AR) and Virtual Reality (VR) hardware as well as tablets. The possibility to visualize geodata on these platforms creates a foundation for further research of specialized interactive geodata visualizations.

Beforehand, geodata basics as well as the used data sources for this thesis are introduced. Followed by a presentation of GIS software and a discussion about the use of real-time 3D engines for the creation of such an application. Then, different details about the implementation results are presented. In the end, current limitations as well as connection points for future work are discussed.

2. Geospatial Data

In this chapter, components of geospatial data (alias geodata) are introduced, followed by a differentiation of the two data types raster and vector data. Afterwards, different file formats for storage and the used library for interfacing this data are presented. Finally, the geodata sources used for the creation of this thesis are introduced.

2.1. Spatial Reference System

A Spatial Reference System (SRS) enables locating geographic objects in a common space, which is based on the earths surface [7]. They have differences in accuracy, computational complexity and applicable regions, but for geodata, coordinate transformations into different SRS are available, for example with tools included in the used Geospatial Data Abstraction Library (GDAL). Coordinates are either angular (longitude and latitude) for geodetic SRS or projected on orthogonal axes (x and y) for cartesian SRS, both types allow usage of height as a third component [7]. Similar to most online maps, mainly projected SRS were used for this implementation.

As spatial coordinates require high precision, they are stored in 64bit floating-point numbers [5]. Popular SRS are listed with an assigned code in the global European Petroleum Survey Group (EPSG) database, managed by the Geodesy Subcommittee of the Oil and Gas Producers [27]. There are several online services for acquiring information about different EPSG codes from this database, for example [13] was used throughout this thesis. The globally most known and used SRS are:

World Geodetic System (WGS) 84 - EPSG: 4326 Used by the Global Positioning System (GPS), unit: degrees, ellipsoid

Pseudo/Web Mercator - EPSG: 3857 Used by most online map services, unit: meters, WGS 84 projected on a flat, rectangular and tileable surface

The details are explained in [7] and [27].

2.2. Raster Data

According to Pingel [22], raster formats are "well suited" for representing continuous spatial data, as storing them is more efficient than equivalent point-based in vector systems. Popular example raster data include: Digital Orthophoto (DOP), meteorological variables such as temperature or rainfall, Digital Elevation Map (DEM) [22], a DOP can be seen in Figure 2.1.

They have regular fixed sample points (pixels) as well as a limited size, and may contain either a single band of continuous or discrete data, or multiple bands, which are then represented by individual color channels [22]. Discrete regional classification is also representable using raster formats and a stored color palette [5]. It is possible to extend the implementation of this thesis when a visualization of this data type is required.

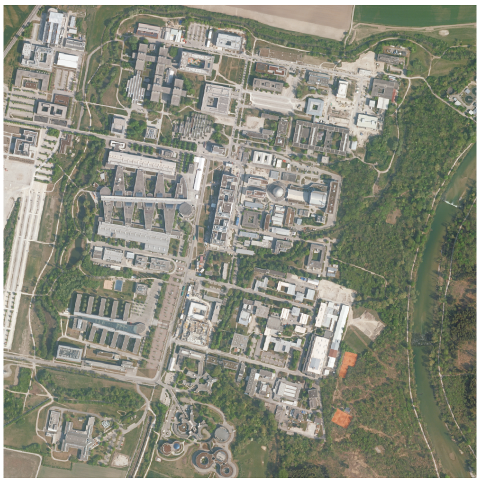

Figure 2.1.: Multi channel DOP raster data of the TUM university campus (via WMS) [10].

2.3. Vector Data

Data represented by or linked to discrete points, edges, areas or volumes is classified as vector data. The OpenGIS Consortium (OGC) that releases standards for geodata, created the Simple Features Specification (SFS) [19] for describing 3D base spatial types that make up more complex multidimensional features. Figure 2.2 shows such a

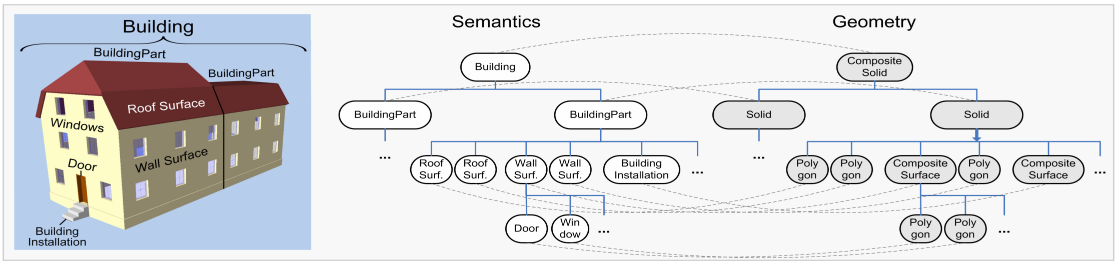

Figure 2.2.: Complex CityGML vector object showing coherence of semantics and geometry (taken from [26]).

complex object, which is used in the CityGML [17] semantic data model [16, 26]. Other example vector data include: Points of Interest (POIs), street networks or regional areas in addition to 3D buildings.

CityGML vector data is represented with different Levels of Detail (LODs) which range from a 2D polygon representation of the building (LOD0), over simplified 3D shapes with flat roofs (LOD1) and more detailed shapes (LOD2) up until even more details and the inclusion of interior structures or rooms with furniture for LOD3-4 respectively [17]. Public available datasets usually contain data between LOD0 and 2.

Every feature may have metadata attributes assigned. Such an attribute field has a name (key) and a value which is given in a data type defined in advance by attribute field definitions. These definitions can also include optional alternative names, default values or other constraints for the value domain of its attribute values [5, 16].

2.4. File Formats

Both vector and raster data occur in a wide variety of different storage and file formats, some are human readable or compressed. These can be read and converted into each other, for example by using GDAL command line applications [5]. Because of large file sizes (especially for raster data), they may be available on an web request base where the currently displayed area boundaries are sent to a server [27]. Often used are Web Map Service (WMS) [20] or Web Coverage Service (WCS) for raster and Web Feature Service (WFS) [18] for vector data that is individually requested by a correctly formatted Uniform Resource Locator (URL).

Most online mapping services use Web Map Tile Service (WMTS) instead of WMS as this improves scalability and performance when tiles are pre-computed [21]. Stefanakis [27] describes the access and storage structure of these tiles, which are usually 256x256 pixel in size, and can correctly be addressed by row, column and zoom-level. They are stored in a quadtree based on halving the initial single tile spanning the entire earth area [27].

2.5. Geospatial Data Abstraction Library

The GDAL open source library [5] supports over 150 different drivers for different geodata formats by default, and over 200 in normal configurations. Therefore it is is suitable for usage in this thesis, as most common geodata file formats are supported. Special examples include direct and random network access, on-the-fly reading of compressed files and hosted files from known cloud services such as Google Cloud Storage, Microsoft Azure or Amazon Web Services. Additionally, it is possible to read and write data from and to database servers. This type of data source was not researched in this thesis, but it is of importance for software directly accessing optimized data storage. An example use-case with potential for geodata writing capabilities is covered in section 5.3. For developers, a documentation is provided on how to create raster and vector drivers for custom formats, as well as usage of the library functions. More advanced file sources or structures require some knowledge of formatting appropriately chained open paths. There are over 100 applications linked on the GDAL website using this library including QGIS (previously Quantum GIS) [24], the GIS used for this thesis, as well as for the upcoming example Graphical User Interface (GUI) in section 3.2.

2.6. Used Geodata Sources

For the development process, it is helpful to have access to different types of geodata of the same physical location, ideally in different SRS, which enables testing temporal data transformation as well as well as visual coherence between them when superimposed. In the beginning, a spare set of datasets were downloaded in order to familiarize with the data, GDAL and QGIS. This was followed by using a variety of locally tested geodata sources for the implementation. While progressing further, the usage of web services was increased due to an observed improvement in speed and predictability of the loading performance, especially for larger raster sources.

In the following subsections, different used geodata sources are presented.

2.6.1. Bavarian Open Data

The regional open accessible data of Bavaria does include 2D vector and raster data, but was mainly used for the implementation of raster support and initial mesh generation. Most vector data is both available as WFS and rasterized as WMS for simplified overview and GIS integration. The datasets are listed and searchable by type, but higher resolutions as well as 3D buildings are not openly available [10].

2.6.2. North Rhine-Westphalia Open Data

In this dataset, all obtainable data for a selected area is available in an online map to allow downloading different sources of only the required region [6]. Especially for a large file sizes, the underlying 2D and 3D datasets are directly downloadable [11]. Of high interest are the open availability of two different LODs for buildings, as well as point cloud laser scanned data. The latter is not covered in this thesis. Additionally, a list of open accessible web services can be used [2].

2.6.3. Berlin Open Data

The city of Berlin offers textured 3D city models both in common geodata format as well as different common rendering 3D mesh formats. These are more optimized for loading into 3D engines, as the data is already triangulated. They can be selected on a map and downloaded as individual tiles or as a single exported region [1], an example mesh is shown in Figure 2.3. Additionally normal 2D geodata web services can be selected from a searchable list including preview and information [25]. Other Berlin data visualizations using the implementation created in this thesis are depicted in Figure 4.15

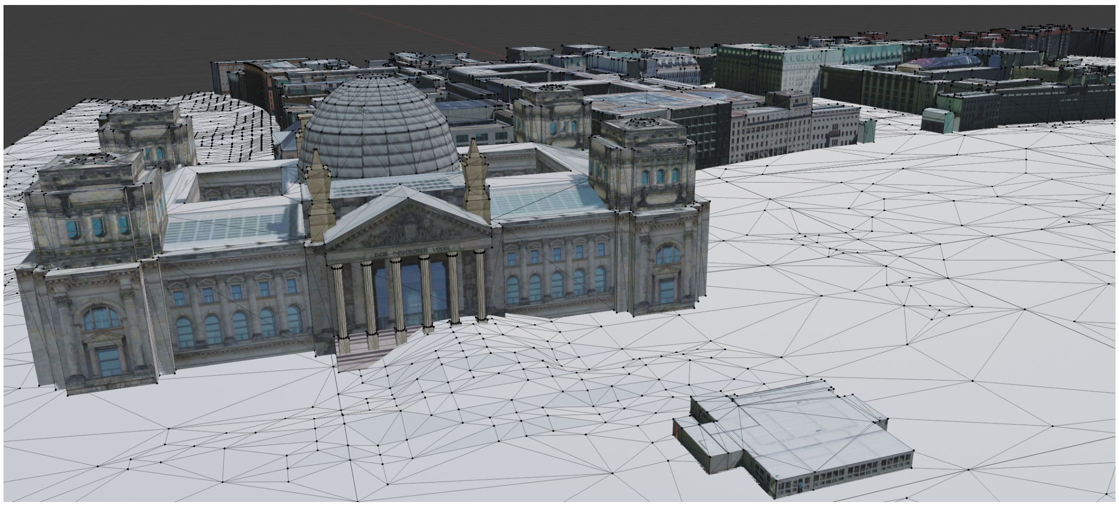

Figure 2.3.: Textured triangulated mesh of Berlin downloaded as *OBJ* file and opened using blender [1]. The Brandenburg Gate is not included in some available datasets.

3. Geographic Information System

In this chapter, basic functionality of a GIS application is shown and some common user interaction elements are explained based on the GUI of QGIS. With regards to the thesis topic, advantages and disadvantages of an implementation using a real-time 3D engine are discussed.

3.1. Definition

According to Steiniger and Weibel [28], GIS software "is used to create, manage, analyze and visualize geographic data". Individual programs offer different specialized tools for their use cases. But most commonly used GIS applications implement some functionality in all three subcategories (Viewer, Editor, Analyst) [28].

Multiple geodata sources can be loaded simultaneously into the current scene, where they are now managed as layers. In case of vector data, similar items may be grouped together into their own sublayers. If the loaded data is not present in the scene's selected SRS, it is temporarily transformed. Subsequently, the layers are superimposed in their current hierarchy order onto the final resulting image. For every layer, different visualization parameters can be changed by the user. Most importantly, this includes changes to color and transparency, but more advanced visual changes are possible as seen on the right side of Figure 3.1. When changing the displayed area, the opened layers are queried again with the current area bounds and the scene gets redrawn.

Most GIS support interaction with the displayed data in form of querying the layers data for the selected point or area. Additionally applications with editing capabilities allow the creation of temporal layers for manual feature creation out of points, these are turned into permanent layers when saved to a system drive by the user. Figure 4.14 contains such manually in QGIS created vector data.

3.2. Example GIS Graphical User Interface

In this thesis, the open source program QGIS [24] (version 3.16.6-Hannover) was used for gaining familiarity with geodata in general as well as comparing expected geodata visualization results and multiple figures.

Basic panels include: the current layer hierarchy, a layer source browser with saved remote services, the main map view, a list of processing tools and, if applicable, information about the current selection. In addition, the GUI shows information about the current viewport on the bottom and a selection of available tools on the top. This can be seen in Figure 3.1.

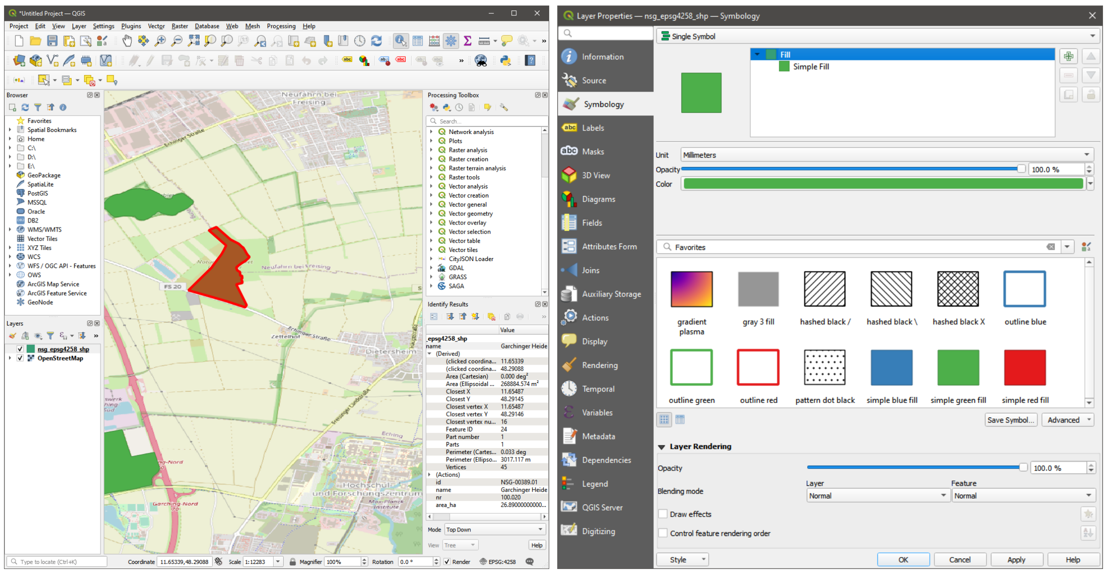

Figure 3.1.: QGIS GUI, displaying a vector layer of local conservation areas on top of an OpenStreetMap raster layer. There is one feature selected and its attributes are shown. On the right, the vector layers property panel is opened with symbology customization options.

3.3. Advantages of Creating a GIS in a Real-Time 3D Engine

For scientific research with the most used CityGML data format, 3DCityDB [30] is a good performing database for accessing large 3D datasets. As with many other 3D GIS applications, the viewing is done in a web client for universal device access. A real-time 3D engine allows advanced use case extension, e.g. other non-geospatial 3D assets, specialized controller support, simultaneous multi-user interaction and most importantly, VR, AR as well as other specialized hardware visualization capabilities in the same environment.

3.4. Disadvantages of Creating a GIS in a Real-Time 3D Engine

The GeoBIM benchmark [15] studied different CityGML implementations on example real-world data and criticized the lack of uniformity in geospatial applications (not limited to GIS) when handling this type of open standard data. The studied users showed a low success rate for opening a large file with a size of 5GB (city of Amsterdam), with half of the successful participants requiring more than one hour to load. For a real-time 3D engine, there is additional effort required to triangulate the data to create optimized for real-time viewing performance. If the interpretation of a standard for interfacing this data limits the performance this much, the resulting time required to open is even more noticeable. For example, a conversion into the more optimized CityJSON [12] format could result in a better loading performance. But any iteration over universal geodata formats solely for mesh creation is still less performing than loading 3D mesh files directly.

Additionally, when dealing with a mixed scene of 2D and 3D data, it is necessary to move the flat results up on the same height level as the already correctly elevated 3D objects. This results in more required user effort (for manual elevation adjustment) or knowledge of their supplied (or now required) geodata in form of a DEM. Without manual adjustment and preparation, visual problems may occur at this stage, an example of this can be seen in Figure 4.14.

For 3D geodata, special live visual customization options (e.g. infill pattern) from section 3.2 would be increasing the generation time, complexity and resource overhead, which results in a limited selection of options available for users. For developers wanting to implement these advanced customization options, this requires an in-depth understanding of the project, but could be added more easily for specific custom layer types if required.

4. Implementation

In this chapter, different implementation decisions are presented. These are divided into individual system components as well as the different layer types.

4.1. Base System Requirements

A solid system should allow a mixture of various different geospatial datasources in the same view. Different hardware platforms should be able to render the same data formats with equal visual results. Each source layer should have a set of exposed controls to enable adjustment by the user or other systems running in the engine (such as User Interface (UI), remote synchronization or other input devices). The displayed area of the visualization is required to be limited within previously set dimensions, this allows one or multiple independently operating maps to be placed in a the 3D environment. Changing the maps coordinate bounds must not lead to a complete rebuild of the scene by first unloading the previously seen data, which is the default behavior for 2D GIS applications without real-time requirements and can be observed when using QGIS.

4.2. Map Instance

Since the TallShipEngine allows creation of multiple independent objects with attached scripts, there can be multiple unique map instances existing in the same scene. Each instance has a SRS set at the beginning, in which the map operates and coordinates forwarded to the individual layers are given. As different data types or future custom layers require different transformation techniques, the layer itself handles this step at some point before displaying. An instance can be created by adding the main map script to an empty GameObject while its creation, which now becomes the parent of all 3D objects later instantiated by this map.

4.3. Data Layers

One map instance handles a list of layers that receive updated coordinate bounds when the displayed area changes. Subsequently, they load new displayable data from their managed geospatial datasource, which is converted into objects displayable by the 3D engine. Through the use of an individual GameObject as a child of the map instance, independent object management between the different layers of a map is realized. This additionally simplifies potential implementations for hiding a complete layer by deactivating its object.

The base class from which all different layer types inherit offers common functionality and attributes. It holds its own SRS, as well as basic coordinate transformation functionality to and from a local copy of the map's SRS. It also manages the flags required for the update process as well as other layer parameters.

4.4. Preloading of Larger Layer Boundaries

To reduce the frequency of data fetching on map movements, it is possible to increase the loaded area individually for each layer. The here self-titled *overprovision factor* is the multiplier of incrementation relative to the visible map dimensions. For example a value of zero is equal to no change and a value of one results in 100% size increase, which will double the loaded bounds dimensions. This behavior was selected over an alternative chunk based solution to reduce the number of active rendering objects. This is currently a major limitation for real-time rendering in the TallShipEngine, which is reasoned in subsection 4.9.5.

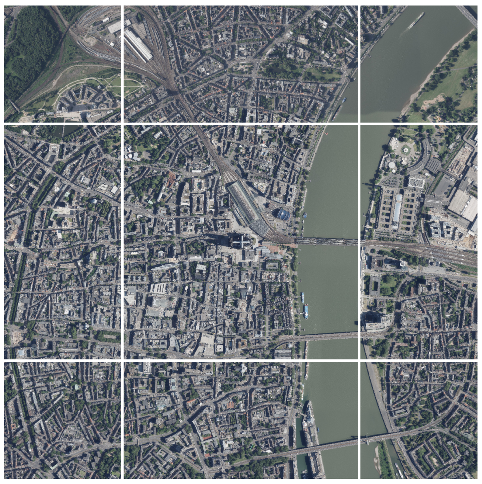
Figure 4.1.: Debug view of a Cologne DOP raster layer with overprovision factor 1 (via WMTS) [2].

For each layer, a debug view can be toggled. It enables displaying the whole currently loaded data, ignoring the usually active restriction to the maps displayed area. This can be seen in Figure 4.1, where the white lines drawn by the shader form a frame around the latter.

4.5. Triggering Layer Updates

Loading geodata requires a varying amount of time and memory, depending on size and complexity. When the visible map area reaches the end of currently displayed data, this layer will display nothing until new data is loaded through an update call. Therefore, a method is implemented to individually customize the behavior with which an update is initiated. Figure 4.2 visualizes two implemented automatic trigger conditions, that are controlled by an *outer trigger multiplier* and an *inner trigger multiplier*.

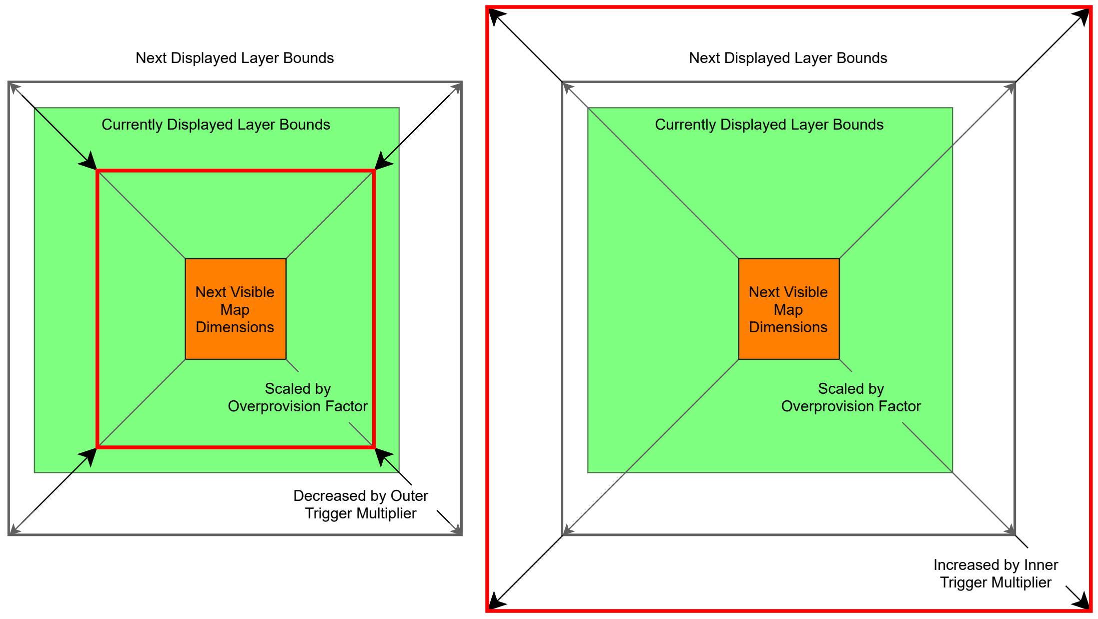
Next Displayed Layer Bounds
Currently Displayed Layer Bounds
Next Visible Map Dimensions
Scaled by Overprovision Factor
Decreased by Outer Trigger Multiplier
Next Displayed Layer Bounds
Currently Displayed Layer Bounds
Next Visible Map Dimensions
Scaled by Overprovision Factor
Increased by Inner Trigger Multiplier

Figure 4.2.: Visualization of two different layer update triggers, not in scale. Trigger conditions are based on the relation between computation result (red) and currently displayed bounds (green).

Both trigger conditions are adjustments relative to the current visible map bounds, that are already scaled up by the overprovision factor and are clamped to be in the range [0;1]. If one of these two trigger conditions apply, this layer is added into the respective update queue. The outer condition (left) triggers if the downsized resulting red bounds are not fully inside the currently displayed (green) layer bounds. The inner condition (right) operates inverted, it triggers when the current layer bounds are not fully inside the expanded result. The debug view mentioned in section 4.4 helps with visual understanding of current tuning parameters to optimize a given scene. The two triggers are required to handle map pan and map zoom with the same functionality. Additionally, each layer has a *dirty flag* to manually force an update at the next opportunity.

4.6. Concept of Map Objects

The TallShipEngine supports 3D object transformation with 32bit floating-point precision, whereas SRS coordinates require the use of 64bit precision. The density of available floating-point numbers increases towards the value zero [4] as seen in Figure 4.3. As the current map view will only display data with close proximity relative to the zoom level, it is possible to transform the global 64bit coordinates into lower precision local 32bit coordinates.

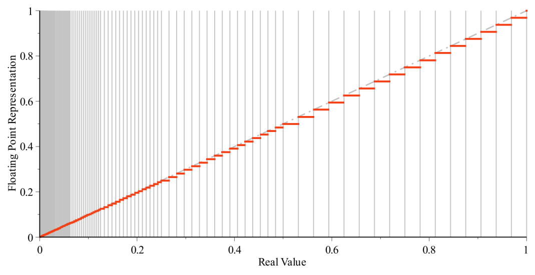

Figure 4.3.: Non-uniform Distribution of Floating-Point Numbers (taken from [4]).

Any displayable object with geospatial high precision coordinate requirements is managed as a *MapObject*. This references the GameObject of the engine, including transformation and rendering material, and stores the 64bit floating-point coordinates. When the map's coordinates are changed, they are forwarded to every layer, each of which can manage their child MapObjects differently.

4.7. Layer Ordering

In a 2D GIS application, overlaying the individual layers is done by the order in which they are rendered. In this project implementation, the layer order is controlled by a vertical offset to the layer's transform. Zooming in the map scales up the coordinate parent in all three dimensions, which also increases the layer objects scale including their vertical offset. This leads to inconsistent layer spacing and even introduces z-fighting when zooming out, which shows floating-point accuracy errors. This problem is resolved by updating every layer's local vertical position to be located at a fixed global height over each other whenever the map changes its zoom value.

For overlapping 3D geometry, this approach only resolves z-fighting on the horizontal plane. As this implementation is designed to be viewed mainly from above, and z-fighting for vertical geometry only occurs when displaying the exact same 3D geometry in different layers, we see this as a valid solution.

4.8. Raster Layer Base

A raster layer is designed for displaying one or more here named *RasterParts*, which consist out of geographic boundaries, a target texture with adjustable size and a MapObject. When displayed, a tessellated quad geometry is instantiated with correct scaling, additionally the texture is created and applied to the objects material. The raster layer differentiates between current and future RasterParts, which by default get exchanged after displaying. But this behavior is overridable by child class implementations.

4.8.1. WMS Prototyping Layer

Initially, a WMS layer was implemented for prototyping as a subclass of the raster layer, with the use of its specification document [20]. This dynamically created correctly formatted request strings for the current map area, followed by loading and displaying the layer data. Later in the the development process we found, that the generalized GDAL raster layer from the upcoming subsection 4.8.2 is able to support these request-based layer types with minor changes. The following data sources were tested: WMS, WMTS and WCS.

4.8.2. Default GDAL Raster Layer

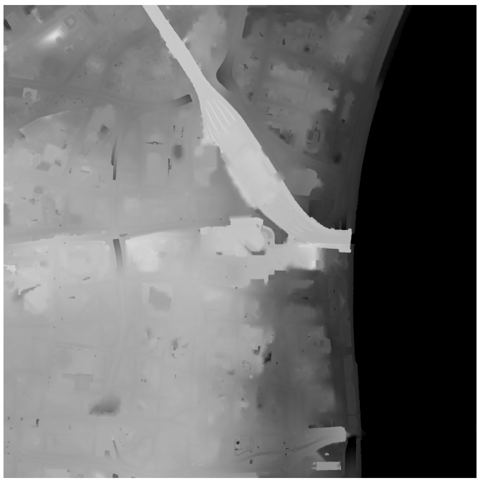
Figure 4.4.: Visualization of Cologne DEM values from 40 to 60 meters (via WCS) [2].

This layer implementation is used for displaying individual GDAL raster layers. The current map boundaries are converted into the correct raster pixel coordinates. With these, the available bands of data are sampled into the four color-channels (red, green, blue, alpha) of the texture. If the source raster does not contain a fourth band, a fixed value is inserted. Additionally, if only a single band is present, its values are copied into all three base color channels, which allows later coloring through the material. As sources may use different value interpretations (e.g. height in meters for DEM, or [0;255] color values for DOP), all values can be clamped to an adjustable range, which then gets mapped to the full displayable color channel range. By default, this is set to display colored raster data, but allows displaying data that is not encoded with red, green and blue color values, which can be seen in Figure 4.4.

4.9. Vector Layer

Visualization of a vector layer requires iterating over the individual features described in section 2.3. These may be grouped into different sublayers with similar data, which are usually exposed to the user as an individual datasource inside GIS applications. Of interest for this implementation are only different geometry groups such as parts of buildings from different LODs, bridges and tunnels. But other data attributes may be present in different sublayers. It it therefore required to allow displaying of a subset of these sublayers. This can be done by sublayer index or name, whereas for larger datasets, the list and name length can become longer. For example, the largest dataset from the GeoBIM benchmark [15] (Amsterdam) lists 1,407 sublayers, of which around 80 contain geometry according to GDAL utilities.

4.9.1. Preloading all Features

For raster layers, displaying the whole dataset in the beginning is limited to the set or largest supported texture resolution of the Graphics Processing Unit (GPU) and therefore of no use. For vector data, preloading the whole geometry independent from the currently visible map area may be desired by the user for known datasets in order to reduce processing delays on movements afterwards. This advanced usage can be be enabled for each layer individually and is disabled otherwise.

4.9.2. Default Feature Iteration Strategy

The default strategy iterates over the available sublayers in the dataset. If the current one is enabled for visualization, every contained feature is iterated over. The pseudo-code for this implementation is shown in algorithm 1.

When updating the layer, it begins by setting a spatial filter on the GDAL sublayers. This results in all fetched features to geometrically intersect with this area. After the mesh creation process, the mesh objects are stored relating to area at the time of creation. When upcoming update boundaries do not intersect with stored ones, and removal of old data is not disabled for this layer, every related MapObject as well as their stored meshes are marked for removal.

4.9.3. Optimized Driver Dependent Feature Iteration Strategy

Especially but not exclusively, the most commonly used CityGML [17] format is internally structured with randomly ordered features. The default iteration approach is performing suboptimal for these datasets. After successful verification of this capability,

```
while geodata has sublayers left do
    if new mesh was started then
        store old meshes and current ID list;
        set dirty to force next update call;
        end current update to display result;
    end
    if end of sublayer reached then
        get next sublayer;
    end
    while sublayer has features left do
        get next feature;
        if feature has geometry and feature ID is not stored then
            recursively generate mesh from geometry;
            add feature ID to current ID list;
        end
    end
end
store old meshes and ID list;
```

Algorithm 1: Update iteration over features per geodata sublayer.

the different update iteration approach from algorithm 2 is used. This primarily iterates over each feature and then checks the layer it is assigned to.

4.9.4. Comparison of Strategy Loading Times

Figure 4.5 shows the scene with the vector data, that was used for comparing the strategies. The downloaded CityGML data should use the optimized strategy. This shows when loading the whole file, which is finished after 10 seconds. For the default strategy with spatial filtering, this speed can only be achieved when loading a small portion, otherwise loading is finished after one minute. But further optimization could be researched with different datasets.

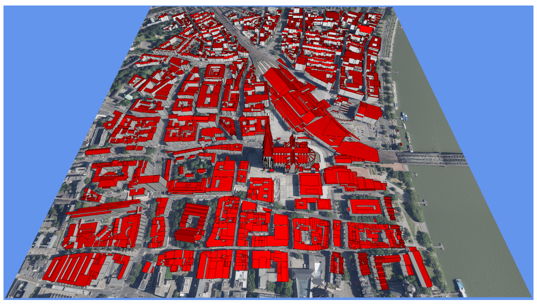

Figure 4.5.: Whole Cologne scene data used for strategy comparison [2, 11].

```
while geodata has features left do
    if new mesh was started then
        store old meshes and current ID list;
        set dirty to force next update call;
        end current update to display result;
    end
    get next feature;
    if feature has geometry and layer of feature is enabled and feature ID is not stored then
        recursively generate mesh from geometry;
        add feature ID to current ID list;
    end
end
store old meshes and ID list;
```

Algorithm 2: Update iteration over randomly ordered geodata features.

4.9.5. Mesh Generation from Geodata Geometry

While iterating over the individual features of a vector layer, the feature geometry has to be converted into 3D meshes for the TallShipEngine. Both algorithms perform the same geometry generation, where the results are added into a single buffer. This increases the required calculation time before the result is displayable, but more importantly it drastically reduces the amount of meshes, GameObjects and therefore engine drawcalls.

Increasing the amount of rendered objects results in a higher required time to render the frame. After initial testing, the limit at which the native refresh rate of current VR Head Mounted Displays (HMDs) (e.g. 90Hz) could not be achieved anymore, was found at at less than 1,500 individual objects for last generation desktop computer hardware. Though, for actual VR hardware this is assumed to be lower. As the GPU usage was not a limiting factor at that time, it showed either a Central Processing Unit (CPU) or Input/Output (I/O) limitation for the current state of rendering in the TallShipEngine.

For this reason, a mesh is only displayed when the maximum mesh buffer size is reached, or there is no geometry left while iterating. This combined vertex and index buffer is currently limited to the maximum size allowed by a 16bit unsigned integer (65,536), where each vertex counts as one, each line segment as additional two or each triangle as additional three. When this limit is increased in the future, implementation of an alternative trigger (e.g. time-based) for displaying the results is advised.

Geometry of the SFS consists out of different types shown in the hierarchy from Fig-

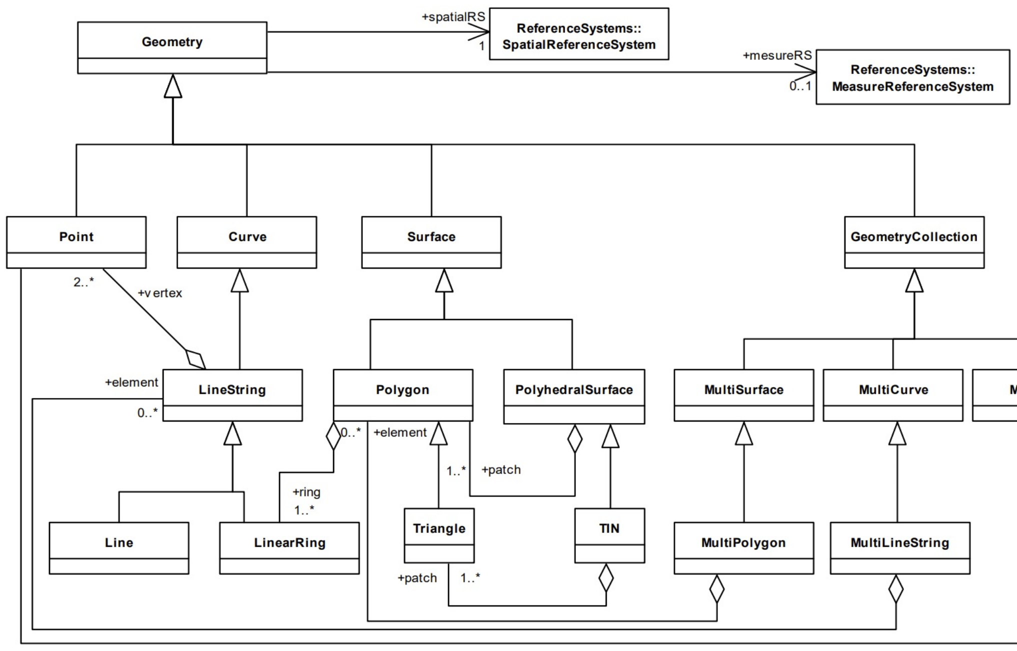

Figure 4.6.: Geometry class hierarchy taken from the SFS [16]. Ultimately consisting out of individual Point class instances.

ure 4.6. All complex types are based on lists of points, which are given another context based on the owning geometry class. All instantiable subclasses of the specification are 2-dimensional geometric objects in an up to 4-dimensional coordinate space (x, y, z and m as a measure) [19]. In this thesis, only up to three dimensional data was used and therefore implemented.

Recursive Traversing through Geometry

As the exact geometry type in a dataset can be different from the available types in the specification, its nearest base class has to be determined. To accomplish this, the inheritance tree is traversed, and for every subclass in a collection, the generating function is called recursively. The mesh generation is split up into line meshes and polygon meshes based on what the vector layer is set to display.

Line Mesh Generation

This geometry type does not need additional processing, as every complex type except MultiPoint consists of elements of the LineString subclass, including surfaces. Each points coordinates get transformed into the correct SRS and the current boundary coordinates get subtracted before getting appended onto the vertex list. Beginning at the second point, for every new point, a line segment between the last two vertices is created by adding their two indices to the index list. If the buffer limit is reached, only the latest vertex has to be copied into the buffer of the next mesh. A 3D geometry visualized as a line mesh is shown in Figure 4.7.

The SFS [19] mentions the possibility for custom variants of the Curve class, where two points could be connected by a different path than the default shortest path. In this implementation such types would be handled as a normal LineString.

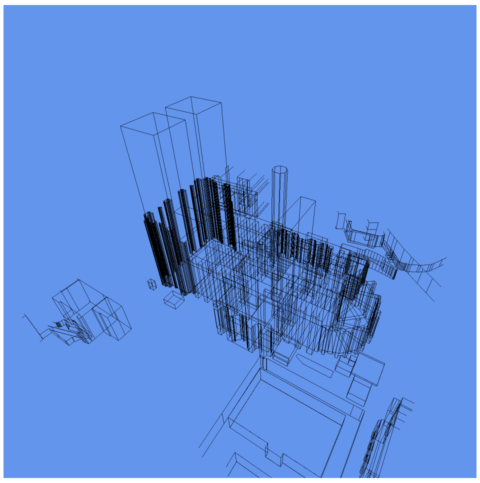
Figure 4.7.: Line mesh of LOD1 Cologne Cathedral [11].

Polygon Mesh Generation

Any geometry surface is built out of polygons. These consist of at least one exterior ring and optionally any amount of interior rings. With the latter being holes in the surface spanned by the exterior ring, some examples are shown in Figure 4.8. It is well defined what visual form classifies as multiple polygons and what is still a single

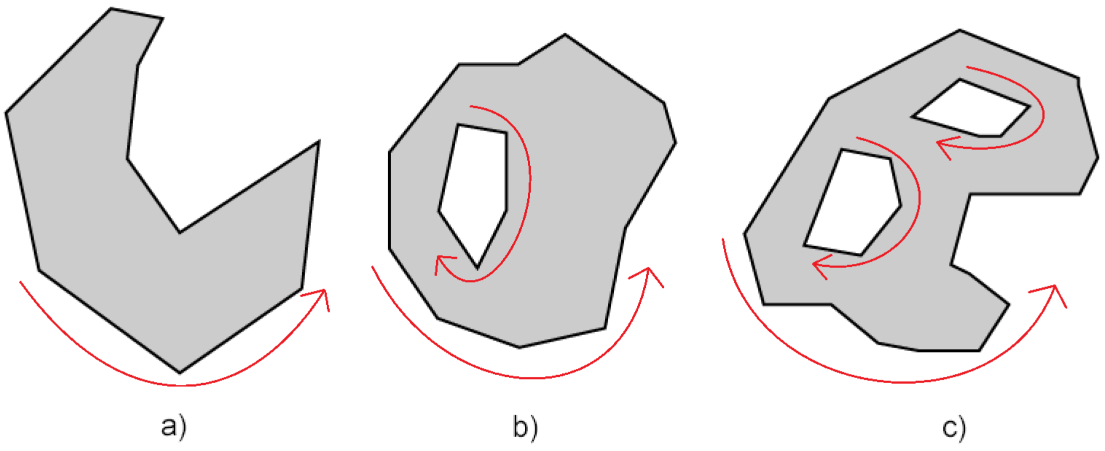

Figure 4.8.: Example polygons taken from the SFS [19] with arrows showing the point order. a, b and c having 0, 1 and 2 holes respectively.

polygon. The order of points from the rings defines the normal direction of the surface, as the top side is where the points appear ordered counter-clockwise. Whereas the order of points is clockwise for holes.

To create a geometry triangulation, an open source implementation based on the ear clipping algorithm is used [3]. This implementation requires structured 2D input geometry and returns an index list for the resulting triangles. In order to span a local 2D coordinate system for each surface, its normal direction is calculated, as this information is not stored in geodata geometry. The ring with the least amount of points is used, which results in flipping the normal if this is an inner ring. This normal in combination with the first two different points define the base of a 2D plane, through which all points are represented by 2D coordinates.

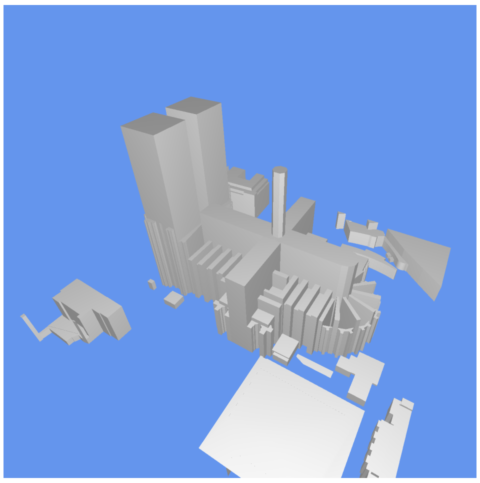
Figure 4.9.: Triangulated LOD1 Cologne Cathedral [11].

After the triangulation step is finished, the points and triangles are appended to the current mesh buffer. Similar to section 4.9.5, the points' coordinates get transformed into the correct SRS, followed by subtracting the center point of the displayed bounds. If the results do not fit into the current mesh, but are small enough to fit into a single mesh, a new mesh is started in advance. Otherwise, it is necessary to split the buffer and create duplicate vertices where required. Figure 4.9 displays a single layer of triangulated vector geometry, whereas a mixture of two polygons and one line visualization is shown in Figure 4.10.

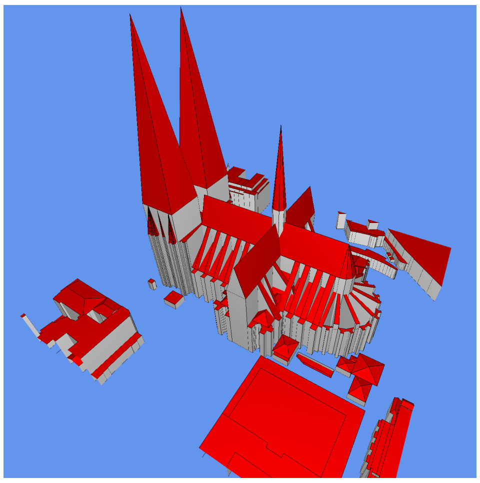
Figure 4.10.: Scene with three instances of LOD2 Cologne Cathedral [11].

Displaying a finished Mesh

Any generated mesh is stored until the layer update is finished. As in the following step update results get displayed. For each mesh an individual MapObject is created with the respective GameObject as a child of the layer. Then the engine mesh and its correct material type get initialized and added, followed by setting the MapObjects' coordinates to the reference point used while creating the mesh.

4.10. Point of Interest Layer

A specialized third layer type was implemented for point data visualization. It allows displaying a customizable textured mesh object for every point of the source geodata. Depending on the used mesh and texture paths in combination with optional automatic scaling based on the current map zoom, different visualization results can be achieved.

For this layer, an attribute filter string can be set, which is a functionality offered by GDAL and explained in the documentation [5]. The string is similar to the *WHERE* clause of a Structured Query Language (SQL) query, to include only a subset of features with matching attribute values. Currently, different GIS applications can be used to view the attribute table of individual features in order to create such queries.

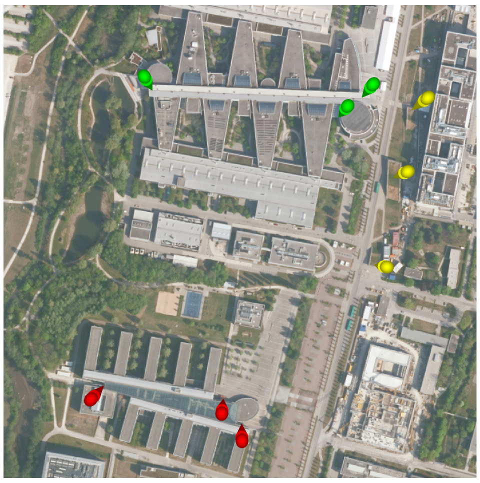
Figure 4.11.: TUM campus visualization with three groups of POIs [10]

Figure 4.11 shows a small dataset created with QGIS containing points with a *title* string attribute, these represent larger entrances of a selection of buildings as well as the train station at the university campus. The dataset is loaded in three differently colored layers, each visualizing a subset of the points using filters.

4.11. Multithreaded Layer Updates

An important aspect of this implementation was the focus on offloading the time-consuming geodata operations onto different threads of the CPU. On the other hand, engine-related object creation is currently not implemented thread-safely, which is why the display operations have to be separated from the geodata updates. The latter are performed by a user-adjustable number of worker threads, which are scheduled by an additional thread. The simplified update procedure is shown in Figure 4.12. As GDAL datasources in general are limited to single-threaded access, the geodata instances are not shareable between the layers, which is a parameter specified while opening them.

4.11.1. Work Scheduler

The singleton work scheduler iterates over the layers of every registered map instance. It uses the latest instance coordinate bounds to check for the layer update triggers from

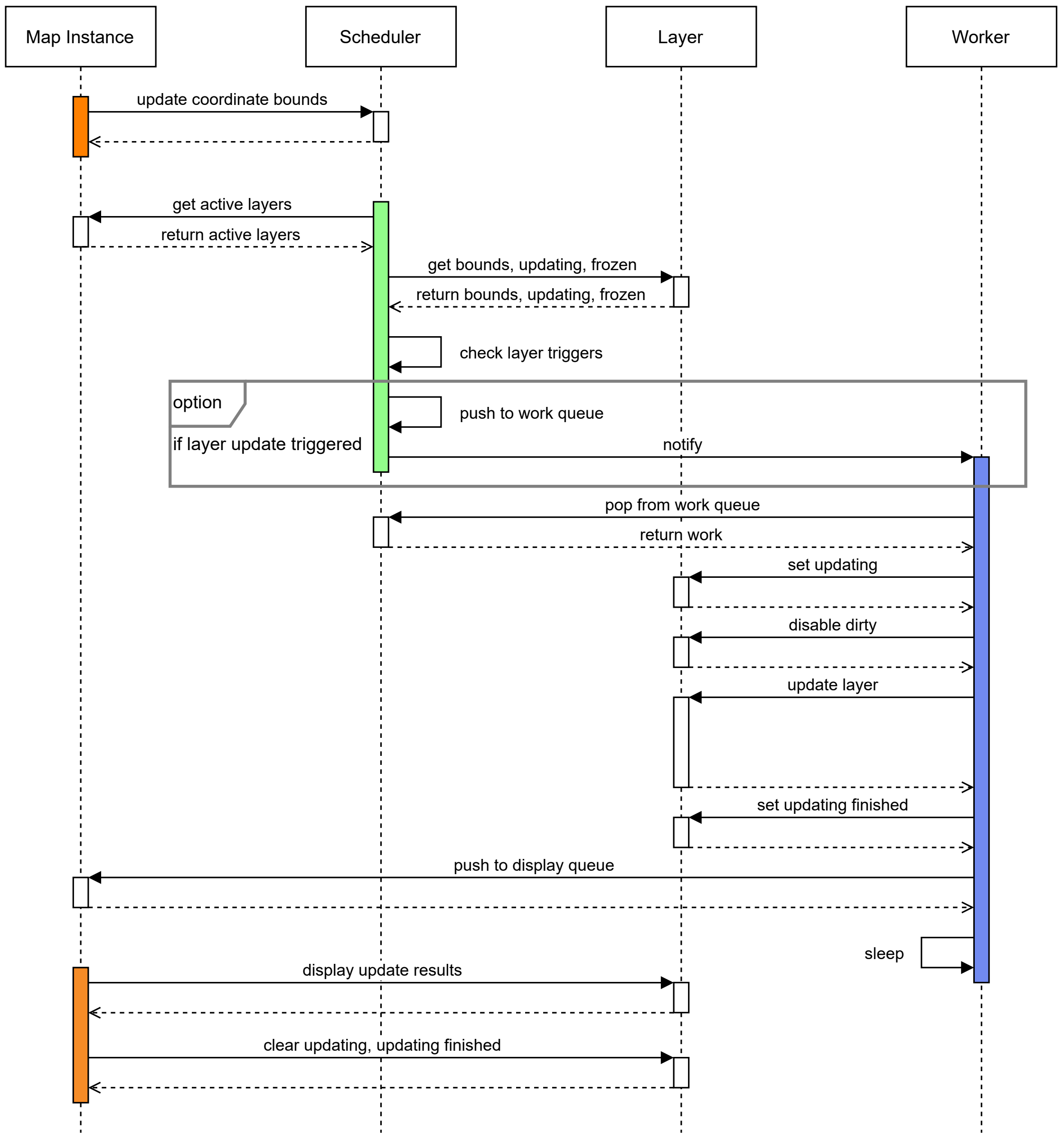

Figure 4.12.: Sequence diagram showing the simplified multithreaded layer update process. Orange colored activity originates from engine scripting updates out of the main application thread, other colored activity from their own looping threads.

section 4.5. Two additional cases are taken into consideration, as they override the triggers from section 4.4, either when the layer is already updating through a worker, or the *frozen* flag is set by the user. Otherwise, if an update is triggered, or the layer is flagged dirty, it is added to the work queue with a copy of the current bounds, followed by notification of a worker thread.

4.11.2. Worker

Each worker thread handles the update procedure of the individual layers. Initially the layer *update flag* is set, which acts as a guard for its attributes. This is followed by disabling the dirty flag and execution of the layer update itself. Subsequently, the implementation marks that the update is finished, and adds the layer into a display queue of the map instance. This handles displaying the results from the main application thread in the next global scripting system update step, as well as clearing of the update and finished flags.

4.12. Object Material and Shader

The map display is limited to specified physical dimensions in the 3D scene. As the visible geodata often exceeds these boundaries, it has to be ensured that out-of-bounds data is not displayed for the default layer types. For this reason, the materials used on these objects were customized. To achieve this effect, the shared vertex shader as well as for each geometry type the geometry and pixel shaders were exchanged.

4.12.1. Restriction to Map Boundary

To know the global offset of each vertex relative to the map's center point, each material instance is given the transposed world matrix of the owning map including its dimensions. Since the world position of every vertex is calculated in the vertex shader, this can now be multiplied by the map matrix to receive its map position. The pixel shader uses this value to allow interpolated per-pixel clipping for every pixel whose map position is outside the displayed boundaries. Changes to this clipping functionality could allow displaying a circular map. When the debug mode from section 4.4 is enabled for an object, its clipping step is skipped and pixels on these lines are colored instead.

4.12.2. Elevating 2D Geodata

When displaying 3D buildings from geodata, these are usually positioned at their correct elevation level. Otherwise, or if additional 2D geodata is displayed, these require manual or automatic height adjustment through the use of a DEM loaded as a layer. This stores the elevation at each pixel in meters or feet (whereas only meters are expected for this thesis) as a floating-point number. It is loaded as a regular raster layer, but instead of displaying on an object, the texture and boundaries are stored in the map instance. They are then assigned to the individual material instances, where for every vertex the 2D map position is used to calculate the height map texture coordinates. Subsequently, the texture is sampled at this coordinates and the vertex is moved in the maps sky direction by the correctly scaled meter amount. In Figure 4.13 a change in elevation can be seen for a DOP, depicting stairs and a neighboring ramp between the Cologne Cathedral and the Rhine.

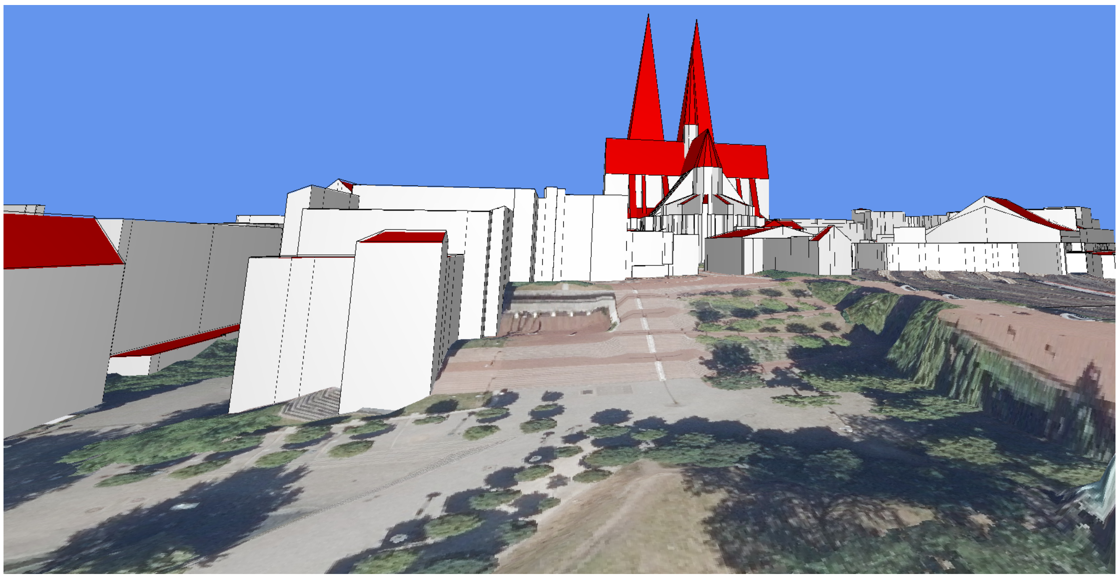

Figure 4.13.: Scene with LOD2 Cologne Cathedral and DOP elevated by a DEM (local, WMTS and WCS respectively) [2, 11].

This solution performs well for raster layers, as they can use an already tessellated quad mesh. For vector layers, an extension of the shader pipeline with tessellation on the GPU is advised but not implemented. Otherwise, the individual source vector area greatly influences the visual result, as the DEM is only sampled at the individual polygon vertex positions. For line meshes, the individual distance between the available data points could be reduced through GDAL (by insertion of additional points at fixed intervals), but for the current default layer implementations this functionality was not used.

An example dataset has been created in QGIS, to explain such problematic data through a visualization. Figure 4.14 contains multiple 2D features: A ramp (yellow) next to equivalent stairs in two different variants (blue and red). The blue stairs are modeled using only the four corner points, whereas the ramp contains more points, defining its curved path. This leads to less problems and a more accurate elevation representation for the ramp similar (but not equal) to results of a tessellation. The points of the blue stairs are connected through two triangles, which are in this case most of the time located below the surface represented by the DEM. The red line contains two points per meter, which allows more accurate elevation sampling. This scene therefore requires further manual adjustment of the height offsets to reduce visual problems, on the cost of the corner points floating above the ground.

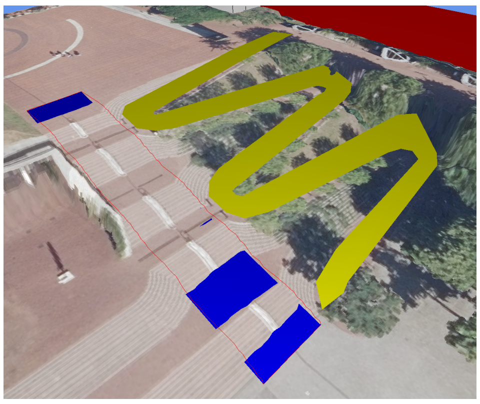
Figure 4.14.: Scene from Figure 4.13 with example vector layers of stairs and ramp created in QGIS.

4.13. Asynchronous Command System

To control individual layer parameters and options mentioned throughout chapter 4 of this thesis, a string-based command system was implemented. This was intended for the initial layer settings loaded from the configuration file, but it is designed to allow thread-safe execution while the application is running. The implementation with a unified command system based on strings enables simple implementation of additional and overriding of existing commands in current and custom layer types.

In order to enable unguarded access to layer settings and objects, it is ensured that commands are executed from the main thread with no simultaneous layer update. Therefore, if an update is in progress, they are queued up until in between displaying update results and clearing of the guarding flags, which are the final two steps seen in Figure 4.12.

A full command string is split into command and arguments at the first existing space character. The former is case-insensitive, whereas the latter is dependant on the specific layer implementation. A full list of currently implemented commands can be seen in Table A.1.

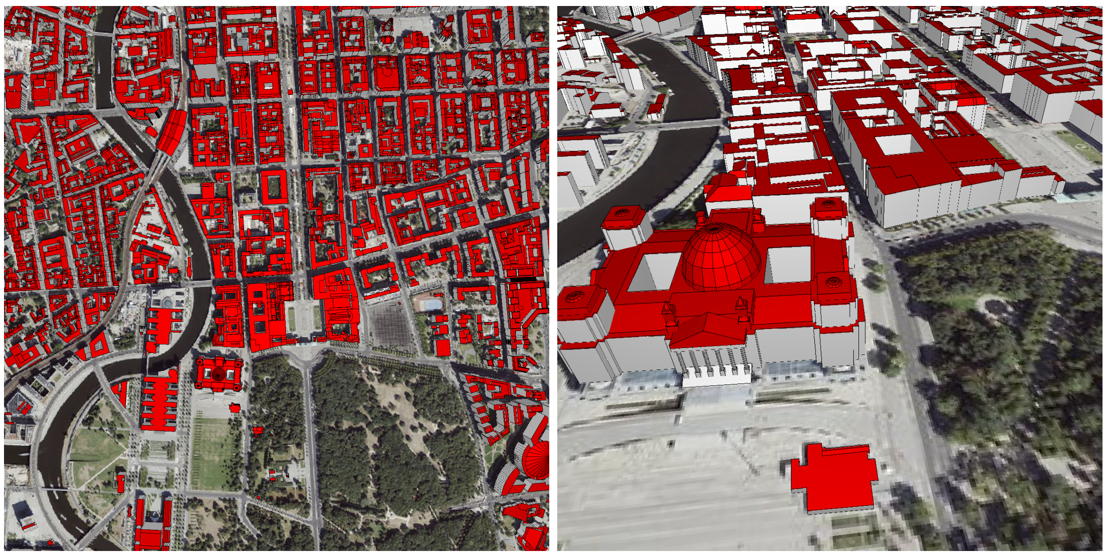
Figure 4.15.: Scene with LOD2 and DOP data of Berlin [1, 25]

5. Outlook and Current Limitations

5.1. Revisit after Specification Updates

As mentioned in section 3.4, the specification implementation can greatly influence the usability of any GIS application. But the same applies to manual preparation of an optimized scene. Especially larger vector datasets contain upwards of 100.000 individual features, any excluded data (mostly whole layers) may lead to improved performance.

Currently, the used CityGML specification is nearing its next revision CityGML 3.0 [9]. This includes for example a space concept to distinguish between occupied and unoccupied space, changes to LOD handling for less redundancy, time versioning and a concept for transportation space. It will take some time until scientific research will be concluded, as the release itself and the following implementations are not completed yet. It would then be useful to revisit this project with matching geodata.

5.2. Categorization as GIS Viewer

This thesis implementation currently enables capabilities for loading and viewing of different geodata. It lacks editing and analysis functionality and is therefore in the "GIS Viewer" category of possible desktop GIS software, according to Steiniger and Weibel [28]. In the following two subsections it is explained why this limitation currently is necessary.

5.2.1. Graphical User Interface Requirement

In the current state, the TallShipEngine does not have a subsystem for simple GUI management. For example, it is only possible to render a string of text at runtime by loading a texture stored on a system drive beforehand. The early stage of this thesis project enabled the support for live instantiation of GameObjects. This was later extended by CPU texture and mesh creation at runtime. Such functionality is a requirement for the future creation of a working GUI system, which in turn is a necessity for usable interaction on HMD platforms. The layers command interface from section 4.13 is a now existing exposed entry point that allows controlled and customized interaction with the different layer types.

Some example functionality of a GUI for interaction with a map instance and its layers is listed in Figure 5.1

Layer list

Minimize option	Hide adjustments for this layer to clean up GUI
Sliders and checkboxes	Adjust individual layer parameters, e.g. disable, freeze, set dirty, change color, change transparency
Information icons	Depicting if errors occured or layer is currently updating
In-depth information table	Displaying information about SRS, the current dataset string and name, number of objects and features or color channels currently displayed

Map Instance

Information fields	Showing current coordinates, if every layer is finished updating
Lock coordinates or zoom	Disable accidental navigation, if e.g. swipe to move is possible

Figure 5.1.: Example GUI functionalities

5.2.2. Layer Interaction using Raycasts

Similar to interaction with a GUI, there is a need to interact with visible 3D geodata. GIS applications offer a tool, that allows clicking on a point in the 2D view to display information including attributes of data at that location from every visible layer, such a tool was used to select the red feature on the left of Figure 3.1 in QGIS. Similar functionality can be achieved by casting a ray onto the displayed geodata meshes, but was not researched for this implementation. Querying of the data has to be implemented asynchronously. Therefore, the command system from section 4.13 is again suitable for initiating this task, that would be executed instead of the next visualization update. To receive feedback on such queries, an asynchronous command callback functionality could be researched. Otherwise, custom layer types could be implemented with that specific feature in mind, for example by deriving from this project's layer implementations. For optimizing load times, it may be helpful to allow disabling of interaction for some layers in a scene, if it is known that no important attributes are available from this dataset.

5.3. Implementation of a Specific Use-Case

There are a multitude of different geodata source types that require some prototyping and a more in-depth expertise in order to implement optimized interactive visualizations. For customized interaction and display strategies of specific available data sources, it is required to focus on a single use-case with a limited subset thereof.

One such use-case could be the the design of a system for *real-time collaborative fire department coordination*, which could be split up into the visualization part of the available geodata and the collaborative interaction between the users and live-updating geospatial information. Implementation of both parts are possible by extension and modification of the implementation resulting from this thesis. Sun and Li [29] discuss key technologies as well as the current development status of this research field. Such a scenario has been as a major thought-focus throughout this thesis. Therefore, it is possible to design for example a networked synchronization module for controlling a remote map instance or a module for saving changes made while the application is running. For a fire department, it is of more importance to handle and perform dynamic changes in a shared database, that may contain information about the current operation. Additional GPS synchronization and therefore visualization of a vehicle's coordinates and status into accessible geodata can also be researched.

This scenario is nearly completely based on a previously known set of static (e.g. DOP, DEM) and dynamic (e.g. vehicle or operation state) geodata. This thesis designed the base system of geodata interfacing in the TallShipEngine, as well as extendable initial visualization capabilities of static vector and raster geodata. The possibility to render custom textured meshes at different points of a dataset allows easier exploration of possible visualization opportunities using the engine. Therefore, researching and implementing such a use-case is a logical step following after this thesis.

6. Conclusion

In this thesis, a geodata visualization system using the TallShipEngine was planned and implemented, which allows displaying common raster and vector data sources in the 3D scene. It was focused on the real-time aspect while loading and converting the data into objects renderable by the engine. This required gaining a deeper understanding of geodata in general as well as GDAL specific functionality for data access.

Initially, a prototype was built in the engine to gain a basic familiarity with its limitations and the datasets. This began with some difficulties of multiple unsuccessful attempts of installing the GDAL library on the system and integrating it into the project. Afterwards, the implementation requirements for the system concerning multi-threading, map instance and layer management were noted. These were implemented while regularly testing with the data, as e.g. GDAL geometry functions operate on a horizontal projection instead of the 3D space. The scarce availability of example code or documentation in combination with the initial lack of support for object creation and removal at runtime changed the initial plan of *only* using the TallShipEngine to also *partially editing* its functionality.

In its current state, the system allows the correct presentation of a previously prepared scene of geodata with different SRS. This supports navigation to update the displayed data at the current visible area. It is possible to implement custom layer types with extended functionality for specialized visualization. For these, the existing codebase shows how to generate and display geometry in the TallShipEngine, as well as how to use the GDAL library to handle geodata in this environment. All of this helps with realizing custom geodata visualizations with focus on a future use-case in the engine.

It was found that preparation and knowledge of planned and available geodata sources, with focus on the final visualization result, are important prerequisites that can greatly influence the real-time performance and program complexity. The growing diversity of existing geodata does not need to be fully accounted for, as a specialized implementation for previously unaccounted data types can be added with manageable resources and effort when enough research about such data types has been made.

List of Acronyms

AR Augmented Reality. 1, 8

CPU Central Processing Unit. 17, 21, 27

DEM Digital Elevation Map. 3, 9, 14, 24, 25, 29, 33, 34

DOP Digital Orthophoto. 3, 11, 14, 24, 26, 29, 33, 34

EPSG European Petroleum Survey Group. 2

GDAL Geospatial Data Abstraction Library. 2, 4, 5, 14, 15, 21, 24, 30, 40–42

GIS Geographic Information System. iv, 1, 5, 7–10, 13, 15, 21, 27, 28, 31

GPS Global Positioning System. 2, 29

GPU Graphics Processing Unit. 15, 17, 24

GUI Graphical User Interface. 5, 7, 8, 27, 28, 33, 34

HMD Head Mounted Display. 17, 27

I/O Input/Output. 17

LOD Level of Detail. 4, 5, 15, 19, 20, 24, 26, 27, 33, 34

OGC OpenGIS Consortium. 3

POI Point of Interest. iv, 4, 21, 33, 42

QGIS previously Quantum GIS. 5, 7, 8, 10, 21, 25, 28, 33, 34

SFS Simple Features Specification. 3, 17–19, 33

SQL	Structured Query Language. 21, 42
SRS	Spatial Reference System. 2, 5, 7, 10, 11, 13, 19, 20, 28, 30, 40
UI	User Interface. 10
URL	Uniform Resource Locator. 4
VR	Virtual Reality. 1, 8, 17
WCS	Web Coverage Service. 4, 14, 24, 33, 34
WFS	Web Feature Service. 4, 5
WGS	World Geodetic System. 2
WMS	Web Map Service. 3–5, 14, 33
WMTS	Web Map Tile Service. 4, 11, 14, 24, 33, 34

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Berlin Business Location Center. *Berlin 3D - Downloadportal*. https://www.businesslocationcenter.de/downloadportal/. 2021.

[2] Bezirksregierung Köln, Geobasis NRW. *Geodatendienste NRW*. https://www.bezreg-koeln.nrw.de/brk_internet/geobasis/webdienste/geodatendienste/index.html. 2021.

[3] D. Eberly. *Triangulation by Ear Clipping*. https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf. 2015.

[4] I. Elshaarawy and W. Gomaa. "Ideal Quantification of Chaos at Finite Resolution." In: July 2014, pp. 162–175. ISBN: 978-3-319-09144-0. DOI: 10.1007/978-3-319-09144-0_12.

[5] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. https://gdal.org. Open Source Geospatial Foundation. 2021.

[6] Geschäftsstelle des IMA GDI Nordrhein-Westfahlen. *GEOportal NRW*. https://www.geoportal.nrw. 2021.

[7] V. Janssen. "Understanding Coordinate Reference Systems, Datums and Transformations." In: *International Journal of Geoinformatics* 5.4 (2009), pp. 41–53. ISSN: 1686-6576.

[8] M. Kaushik. "The Impact of Pandemic COVID -19 in Workplace." In: *European Journal of Business Management and Research* 12 (May 2020), p. 10. DOI: 10.7176/EJBM/12-15-02.

[9] T. Kutzner, K. Chaturvedi, and T. H. Kolbe. "CityGML 3.0: New Functions Open Up New Applications." In: *PFG – Journal of Photogrammetry, Remote Sensing and Geoinformation Science* 88.1 (Feb. 2020), pp. 43–61. DOI: 10.1007/s41064-020-00095-z.

[10] Landesamt für Digitalisierung, Breitband und Vermessung. *Geoportal Bayern*. https://geoportal.bayern.de. 2021.

[11] Landesverwaltung Nordrhein-Westfalen. *Open Geodata NRW*. https://www.opengeodata.nrw.de. 2021.

[12] H. Ledoux, K. A. Ohori, K. Kumar, B. Dukai, A. Labetski, and S. Vitalis. "CityJSON: a compact and easy-to-use encoding of the CityGML data model." In: *Open Geospatial Data, Software and Standards* 4.1 (June 2019). DOI: 10.1186/s40965-019-0064-0.

[13] MapTiler Team, Klokan Technologies GmbH. *EPSG.io*. https://epsg.io/. 2021.

[14] MarketsandMarkets Research Private Ltd. *3D Mapping and Modeling Market by Component (Software Tools and Services), 3D Mapping Application, 3D Modeling Application, Vertical (Government and Defense, Engineering and Construction, Transportation and Logistics), and Region - Global Forecast to 2025*. Research rep. Retrieved from https://www.marketsandmarkets.com/Market-Reports/3d-mapping-market-819.html. Dec. 2020.

[15] F. Noardo, K. Arroyo Ohori, F. Biljecki, C. Ellul, L. Harrie, T. Krijnen, H. Eriksson, J. van Liempt, M. Pla, A. Ruiz, D. Hintz, N. Krueger, C. Leoni, L. Leoz, D. Moraru, S. Vitalis, P. Willkomm, and J. Stoter. "Reference study of CityGML software support: The GeoBIM benchmark 2019—Part II." In: *Transactions in GIS* 25.2 (2021), pp. 842–868. DOI: https://doi.org/10.1111/tgis.12710. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/tgis.12710.

[16] P. van Oosterom, W. Quak, T. Tijssen, and E. Verbree. "The Architecture of the Geo-Information Infrastructure." In: *Proceedings of UDMS 2000, 22nd Urban Data Management Symposium, Delft, September 11-15, 2000*. Retrieved from http://resolver.tudelft.nl/uuid:76338b80-89ae-40f6-8552-f4b6a00219c1. Urban Data Management Society, 2000.

[17] Open Geospatial Consortium. *OGC City Geography Markup Language (CityGML) Encoding Standard*. Retrieved from https://www.ogc.org/standards/citygml. 2021.

[18] Open Geospatial Consortium. *OGC Web Feature Service (WFS) Implementation Specification*. Retrieved from https://www.ogc.org/standards/wfs. 2021.

[19] Open Geospatial Consortium. *OpenGIS Implementation Specification for Gepgraphic information - simple feature access - Part 1: Common architecture*. Retrieved from https://www.ogc.org/standards/sfa. 2021.

[20] Open Geospatial Consortium. *OpenGIS Web Map Service (WMS) Implementation Specification*. Retrieved from https://www.ogc.org/standards/wms. 2021.

[21] Open Geospatial Consortium. *OpenGIS Web Map Tile Service Implementation Standard*. Retrieved from https://www.ogc.org/standards/wmts. 2021.

[22] T. Pingel. "The Raster Data Model." In: *Geographic Information Science & Technology Body of Knowledge* Q3 (July 2018). DOI: 10.22224/gistbok/2018.3.11.

[23] M. Pratt. “GIS Systems Lead Response to COVID-19.” In: *ArcUser* 23.2 (2020). Retrieved from https://www.esri.com/content/dam/esrisites/en-us/newsroom/arcuser/arcuser-spring-2020.pdf, pp. 34–39.

[24] QGIS Development Team. *QGIS Geographic Information System*. http://qgis.osgeo.org. Open Source Geospatial Foundation. 2021.

[25] Staatsverwaltung für Stadtentwicklung und Wohnen. *Geoportal Berlin*. https://www.stadtentwicklung.berlin.de/geoinformation/. 2021.

[26] A. Stadler and T. H. Kolbe. “Spatio-semantic coherence in the integration of 3D city models.” In: *Proceedings of the 5th International ISPRS Symposium on Spatial Data Quality ISSDQ 2007 in Enschede, The Netherlands, 13-15 June 2007*. Ed. by A. Stein. ISPRS Archives. ISPRS, 2007.

[27] E. Stefanakis. “Web Mercator and raster tile maps: two cornerstones of online map service providers.” In: *Geomatica* 71.2 (2017). accessed via https://cdnsciencepub.com/doi/pdf/10.5623/cig2017-203, pp. 100–109. DOI: 10.5623/cig2017-203.

[28] S. Steiniger and R. Weibel. “Encyclopedia of geography.” In: ed. by B. Warf. Retrieved from https://www.zora.uzh.ch/id/eprint/41354/. SAGE Publications, 2010. Chap. Gis software: a description in 1000 words. ISBN: 9781412956970.

[29] Y. Sun and S. Li. “Real-time collaborative GIS: A technological review.” In: *ISPRS Journal of Photogrammetry and Remote Sensing* 115 (May 2016), pp. 143–152. DOI: 10.1016/j.isprsjprs.2015.09.011.

[30] Z. Yao, C. Nagel, F. Kunde, G. Hudra, P. Willkomm, A. Donaubauer, T. Adolphi, and T. H. Kolbe. “3DCityDB - a 3D geodatabase solution for the management, analysis, and visualization of semantic 3D city models based on CityGML.” In: *Open Geospatial Data, Software and Standards* 3.1 (May 2018). DOI: 10.1186/s40965-018-0046-7.

A. Command List

The table below contains all current commands grouped by applicable layer type, the command system is described in section 4.13.

Command	Arguments	Description

Any Layer - section 4.3

ClearVisuals	-	Removes all displayed geometry, usually followed by SetDirty
SetBoundsOverProvision	<multiplier>	Sets the non-negative overprovision factor, see section 4.4
SetBoundsTriggerInner	<0-1>	Sets the overprovision inner trigger factor, see section 4.5
SetBoundsTriggerOuter	<0-1>	Sets the overprovision outer trigger factor, see section 4.5
SetColor	<R,G,B,A>	Sets the layer color as floats [0;1], default: 1,1,1,1
SetDirty	-	Updates this layer on the next occasion
SetFrozen	[true | false]	Disables updating until SetFrozen false
SetHeight	<height>	Sets height above 0 as float, adds to HeightMap
SetIsVisualDebug	[true | false]	Sets Debug mode, see section 4.4
SetSRS	<SRS>	Forces layer SRS, required if not found in dataset, optional speedup of loading, e.g. 'EPSG: 3857', using GDAL OGRSpatialReference.SetFromUserInput(<SRS>)
SetUseHeightMap	[true | false]	Controls if layer is influenced by height map

SetUseShading	[true|false]	Controls if shading is used based on normal direction, disable for raster data

Raster Layer - section 4.8

PrintAllData	-	Prints all information about the GDAL dataset to the console
SetHeightMap	[true|false]	Sets this layer to be used as HeightMap source
SetResolution	<resolution>	Sets raster resolution integer, common values are 256, 512, 1024 or 2048, some web sources are limited to a maximum resolution
SetValueClampEnabled	[true|false]	Sets this layer to clamp the layer values (colors) to the set range
SetValueClampMax	<maxvalue>	Sets the upper clamping value, default: 255
SetValueClampMin	<minvalue>	Sets the lower clamping value, default: 0

Vector Layer - section 4.9

PrintAllData	-	Prints all information about the GDAL dataset to the console
SetLayers	<layername,...>	Enables only the comma-separated list of sublayers by name, listed in output of PrintAllData or GDAL utilities
SetLayersi	<layerindex,...>	See SetLayers, sublayer indices starting at 0 instead of names
SetLineMesh	[true|false]	Changes between surface and edge line generation
SetLoadWithoutSpatialFilter	[true|false]	Sets whether to preload the whole geodata, overrides SetNeverUnload, see subsection 4.9.1
SetNeverUnload	[true|false]	Disables automatic unloading of geometry further away

POI Layer - section 4.10

SetAttributeFilter	<filterQuery>	GDAL attribute query similar to SQL WHERE clause [5]
SetAutoscale	[true | false]	Enables scaling based on current zoom level
SetMesh	<meshPath>	File path for the used mesh
SetObjectLimit	<limit>	Sets the maximum number of visualized objects
SetScale	<scale>	Sets the permanent scale of created objects
SetTexture	<texturePath>	File path for the used texture

Table A.1.: Current layer command list split by layer type. If optional boolean arguments are missing, true will be used.