Alpenite Report

Martin Braunsperger Adrian Gawor

March 12, 2025

Contents

1 Introduction **2**
1.1 Project Context . . . 2
1.2 Goal . . . 2
1.2.1 Project Scope . . . 2

2 Design & Implementation **3**
2.1 tile-downloader . . . 4
2.2 terrainbuilder . . . 4
2.2.1 Geometry Generation . . . 4
2.2.2 Texture assembling . . . 8
2.2.3 Mesh format . . . 10
2.3 terrainmerger . . . 12
2.3.1 Geometry Merging . . . 12
2.3.2 UV Unwrapping . . . 13
2.3.3 Texture Construction . . . 13
2.3.4 Simplification . . . 16
2.4 terrainconvert . . . 17
2.5 renderer . . . 17
2.5.1 Hierarchy size in practice . . . 19

3 Outlook **19**
3.1 Directed Acyclic LOD Graph . . . 19
3.2 Compression . . . 21
3.3 Advanced UV Unwrapping . . . 21
3.4 Blender Integration for .TILE . . . 21

1 Introduction

1.1 Project Context

AlpineMaps is a multi-platform application for rendering 3D maps with a focus on alpine terrain. It currently uses a combination of heightmaps and orthophotos to render the terrain. This works well on flat areas, but it can struggle in mountainous terrain or cities where the height changes rapidly. There also exist some structures that cannot be represented, such as overhangs, caves and bridges. As these issues are very relevant to the mountains that are the primary focus of the app, a different approach is desired.

1.2 Goal

The central goal of Alpenite is to use meshes instead of heightmaps. This would provide several benefits to AlpineMaps. First, it allows the system to represent any kind of surface geometry. On one hand this eliminates the issues with macroscopic geometric features like overhangs. But it is also important to achieve higher geometric resolution, as the world becomes increasingly non-planar the more we zoom in, exacerbating the issues with heightmaps when increasing their resolution. The switch to a mesh-based system should also be able to more efficiently distribute data resources between regions of high and low frequency. More triangles and larger textures could be used for mountainous or urban areas and less for lakes or plains.

1.2.1 Project Scope

The following are goals encompassed by the scope of this project.

Reference Mesh Generating reference/base meshes from orthophoto tiles and a heightmap dataset. The meshes have to use an interoperable file format and contain all their required textures in one file.

Mesh Simplification To facilitate constant visual quality while keeping memory usage in check, mesh simplification at smaller zoom levels is necessary.

Meshlet Tile Generator Conversion of the reference/base meshes into WebMercator coordinate system tiles.

Integration into the AlpineRenderer The generated tiles from the previous step need to be able to get fetched and rendered in the AlpineRenderer.

(Optional) Spherical World Using a coordinate system that is designed to work with the spherical nature of our planet. Currently WebMercator is being used, which is a flat coordinate system.

(Optional) Border Locking In preparation for the next project, border locking between tiles needs to be implemented. This ensures that tile borders are preserved during simplification and therefore tiles of different LOD levels fit together seamlessly.

(Optional) Clustering instead of 2D Tiling Instead of using 2D Tiles, use a clustering method which does not use fixed grid-like tile borders.

(Optional) Data Consolidation Instead of only using one data source, use multiple sources to generate the data (i.e.: photogrammetry, other countries' data, etc.)

2 Design & Implementation

This section will showcase and explain the structure of the project, the relationships between the various programs and how they work under the hood. At the start of the project, we tried to build a monolithic program that handles everything. However, we quickly realized that some tasks didn't require the functionality from all libraries that we used and split them into smaller programs that perform a single task. For example, all the tile-downloader does is download JPEG files via HTTPS, which doesn't need the functionality of e.g.: CGAL and omitting CGAL reduces compile times drastically.

In Figure 1 we illustrate the dataflow and relationships between the programs, i.e.: what is the input and output data of a program, and how the output data is used by other programs.

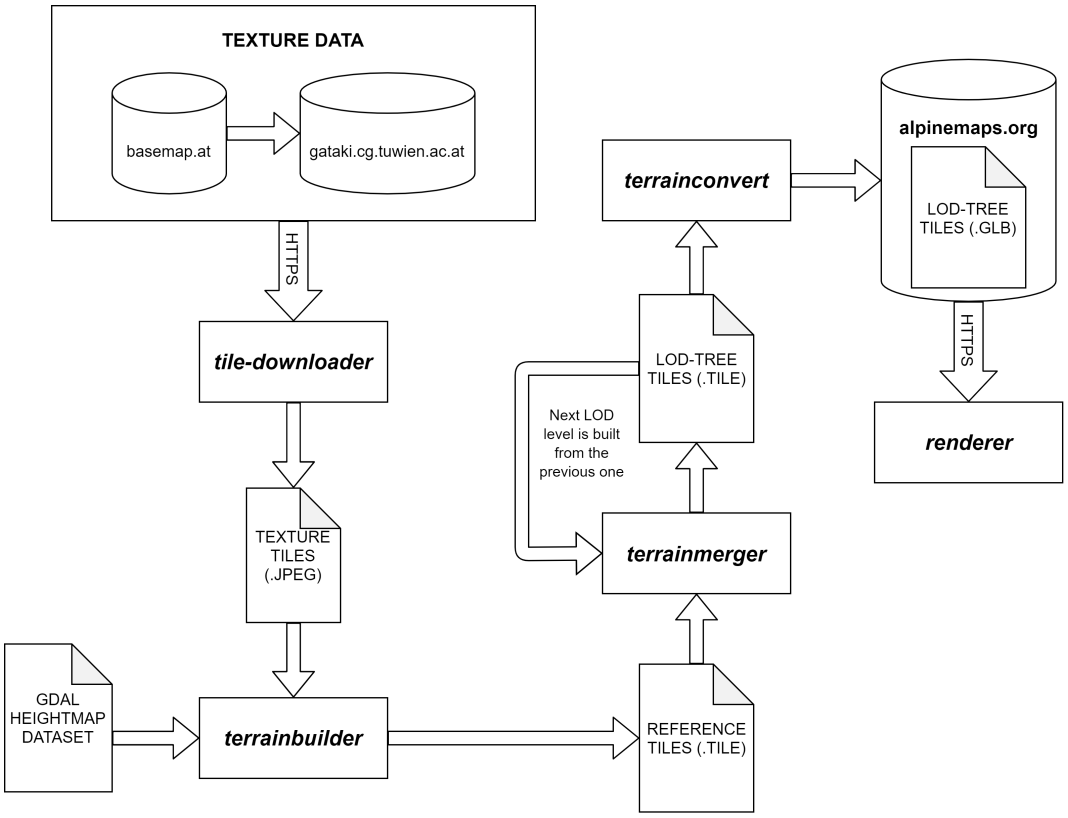


Figure 1: A flowchart representing the data-flow and relationships between the various programs or components described in this report.

2.1 tile-downloader

The `tile-downloader` is a command-line tool designed for downloading tile textures structured as hierarchical grids. At present, it exclusively caters to `basemap.at` (the official geodata of Austrian administrations) and `gataki.cg.tuwien.ac.at` (a private mirror of basemap data). However, its structure allows for straightforward adaptation to other similar services.

2.2 terrainbuilder

The `terrainbuilder` is responsible for generating the reference meshes that form the basis of the LOD tree. These meshes are constructed from a GDAL dataset containing terrain height information and a collection of tile textures. This section first outlines the process of generating geometry from the height map and then assembling the necessary texture data.

2.2.1 Geometry Generation

The generation of mesh geometry involves several key steps:

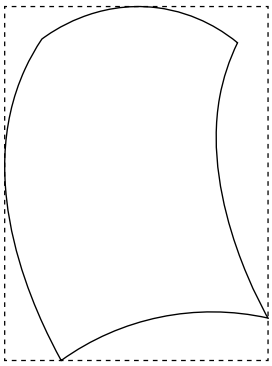


Figure 2: An exemplary depiction of the bounding box transfer. The solid shape represents the target bounds transformed exactly, while the dashed rectangle represents the transformed bounding box. Note that constructing a bounding box from only the corners of the solid shape would be invalid.

1. Retrieving relevant height data from the dataset
2. Converting height data into 3D positions
3. Determining quads within the target bounds
4. Compiling lists of geometry

To begin, the target bounds (given as an AABB) for a given reference mesh are translated into the appropriate spatial reference system (SRS). This is required, since the reference mesh tiles are (currently) generated based on the Webmercator (EPSG 3857) grid, while the height dataset uses the MGI / Austria Lambert (EPSG:31287) format (which is optimized for the Austrian territory). Transforming only the corners of the bounds is insufficient due to the non-linear nature of the transformation between these (and in fact most) SRSs (see Figure 2). The solution is to instead sample points on the border of the AABB, transforming each point, and constructing a new AABB from them. As this transformation can still have some small inaccuracies, we will later round the pixel bounds away from the center and add a small border to it.

Once we have established the bounds within the target SRS, we proceed to identify the precise pixel region containing the data within the dataset. Although this process captures some surplus data, reading rectangular regions is significantly faster and it simplifies subsequent handling. Next, we eliminate any padding data, which are included in the dataset to ensure it covers a rectangular region, even if the actual data is only available in an irregular sub-region. This padding is represented by invalid height values outside the valid range.

The remaining data is then converted from height values to positions within the dataset's reference system. These positions are then used to check each pixel against the target AABB, potentially necessitating further transformations. This check is performed using the center of each prospective mesh quad, to guarantee that each quad is uniquely present in exactly

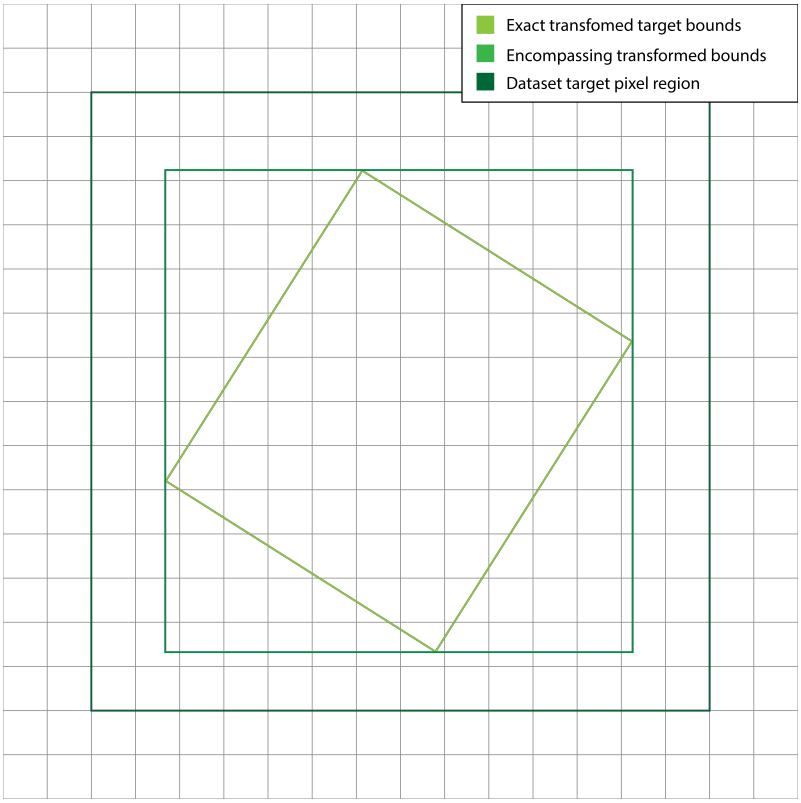


Figure 3: A depiction of the relation between the accurate transformed target bounds and the target pixel region in the height dataset.

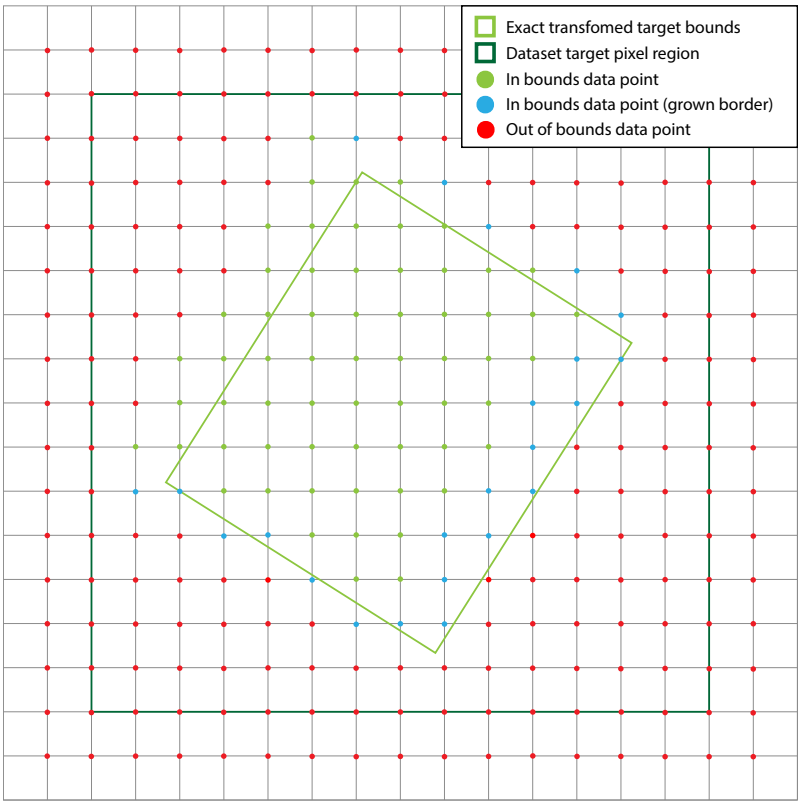


Figure 4: A depiction of the mask storing which quads (represented by their top right corner) are considered in bounds.

one reference mesh tile. Thus, avoiding any overlap with or holes between neighbouring tiles. It's assumed that the actual height measurement corresponds to the center of the pixels, although this detail isn't explicitly stated in the dataset. Consequently, the center of a quad is actually positioned at its bottom right corner. In the practical implementation, the outcome of each quad bounds check is stored as a mask at the top-left corner pixel and then expanded by 1px to the right and bottom (see Figure 4).

This mask can now be used to generate the vertex arrays and triangle list. For every 2x2 arrangement of true values in the mask we generate two triangles forming a quad (see Figure 5 and Figure 6).

The vertex position array is generated by again transforming the 3D positions from the SRS datasets into the mesh SRS (presently ECEF/EPSG:4978). UVs are computed by transforming the source positions into the texture SRS (currently Webmercator/EPSG:3857), determining the resulting bounding box, and then normalizing the texture positions to the unit range. The bounding box of the texture coordinates serves as an output, which is subsequently utilized for generating the texture for the mesh.

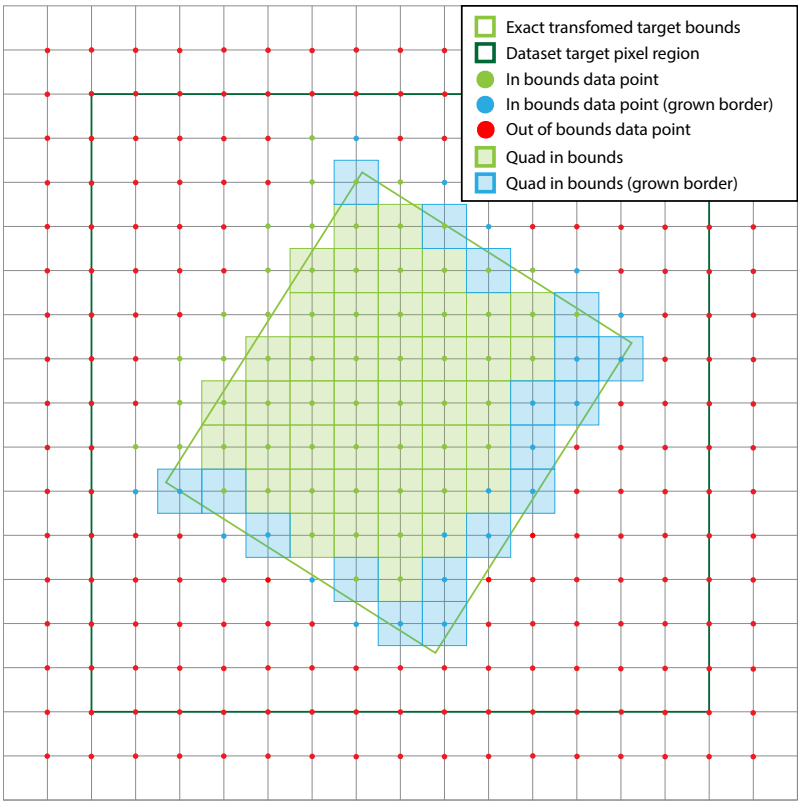


Figure 5: A depiction of the quads generated for some given target bounds.

2.2.2 Texture assembling

Once the geometry for a reference mesh tile is constructed, we proceed to assemble the corresponding texture. This texture is composed of orthophotos sourced from `basemap.at`. These are available for every Webmercator (EPSG 3857) tile, typically down to zoom level 19, although certain areas like Vienna may have higher levels available.

The creation of the texture involves a two-step process. Initially, we identify the highest-level tiles that will be visible in the final texture. Subsequently, in a second step, we merge these tiles into a single texture with the desired resolution.

The selection process is done by first calculating the smallest tile that fully encompasses the target bounds, like is in the geometry building process. This is then used as the root for scanning available textures. We proceed to recursively check every tile. If a tile overlaps the target bounds and not all subtiles (that are in bounds) are available, we have to include it in the final texture. This scan continues at least up to a configurable zoom level that defaults to the approximate zoom level of the target bounds. This is necessary, as otherwise the textures at the root could be missing, if the smallest encompassing tile is very large due to the bounds being on the border between tiles on a high zoom level.

The selection process begins by determining the smallest tile that fully encompasses the target bounds, similar to what is done while building the geometry. This tile serves as the

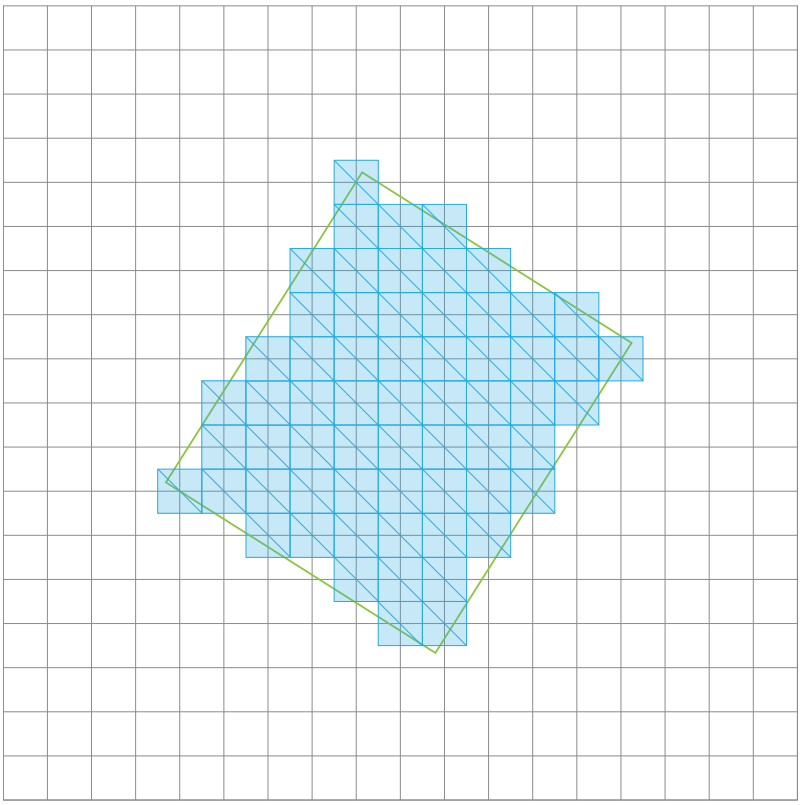


Figure 6: A depiction of the final triangles generated for some given target bounds.

root for scanning and selecting available textures. We then recursively examine each subtile. If a tile overlaps the target bounds and not all of its children (that are within the bounds) are available, it must be included in the final texture. This scanning process continues up to a configurable zoom level, which defaults to approximately the zoom level of the target bounds. This precaution is essential because without it, no textures will be found if the smallest encompassing tile is exceptionally large, due to the bounds lying on the border between tiles at a high zoom level. If it was impossible to completely cover the bounds with the textures available, the parent textures of the smallest encompassing tile are examined until one is available or the root of the grid is reached.

Once we've compiled the list of images relevant for the mesh texture, we arrange them in ascending order by zoom level and sequentially write them into an image buffer. To calculate the required buffer size, we first identify the maximum zoom level image available and determine the resolution of the image tiles. Next, we scale this resolution based on the range of zoom levels covered between the smallest tile and the root tile. Subsequently, we consider the texture SRS bounding box established during the geometry building process and ascertain its relative positions and size within the root tile, calculating the resulting resolution accordingly. The images are then scaled relative to their respective zoom levels, ensuring any excess pixels that would extend beyond the image regions are trimmed.

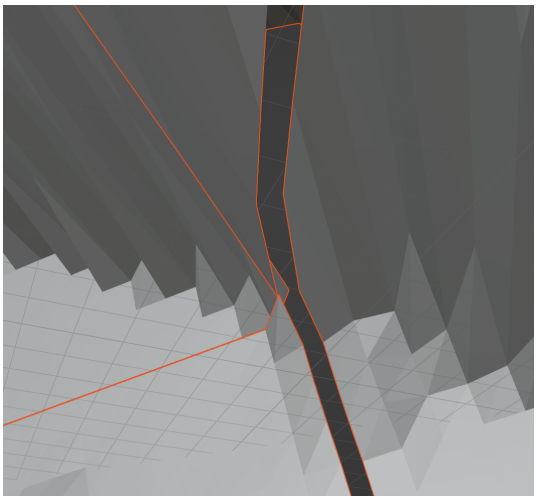


Figure 7: An example mesh consisting of a singular triangle for which the double offset with relative float positions is not exact.

2.2.3 Mesh format

This particular use case presents several unique requirements for storing the reference tile meshes. Firstly, given the need to accommodate approximately 35,000 tiles at level 15 for Austria alone, efficient storage is paramount. Consequently, we aim to avoid text-based formats such as Wavefront (.OBJ) or textual versions of formats like STL, FBX or Collada. Additionally, built-in compression support is desirable to further reduce file size. Moreover, considering that the vertex positions span the entire surface of Earth, maintaining full double accuracy is essential to prevent gaps between neighbouring meshes during rendering, as depicted in Figure 7.

The challenge arises from the fact that many common mesh formats, like GLTF, are limited to float accuracy. This is the case as rendering typically occurs in float accuracy due to the computational expense of double precision. Many formats could theoretically support arbitrary precision since they are text-based. However, even when compressed (like .ZAE Collada archives), this may prove too inefficient. Therefore, formats like GLTF, which store vertex arrays as continuous binary blobs, emerge as the preferred choice. Additionally, the format should enjoy broad support from 3D viewers/editors for both debugging and integrating additional data into the reference meshes (e.g. drone scans or higher resolution textures). Ideally, the format should also accommodate the storage of arbitrary point, text, surface, and volume information for points of interest (e.g., mountain names), metadata, or additional information layers (e.g., avalanche risk).

In summary, the requirements for our mesh storage encompass:

- Double precision
- Efficiency (Binary representation, Compression)

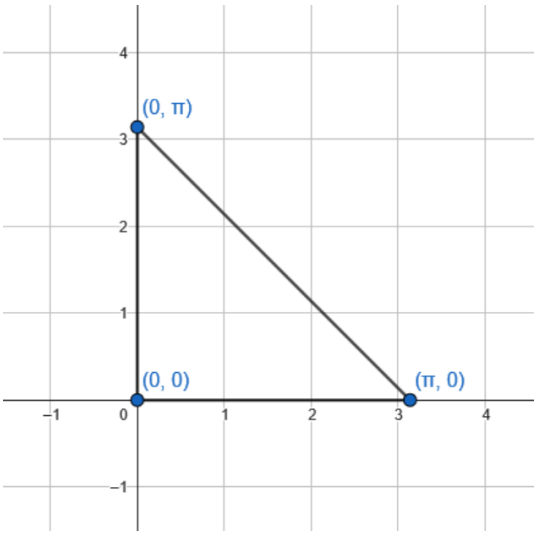


Figure 8: An example mesh consisting of a singular triangle for which the double offset with relative float positions is not exact.

- Viewer/Editor support
- Metadata storage

No format comprehensively supports all the properties required, necessitating a trade-off. Initially, the approach was to use GLTF (in its binary form GLB) due to its efficiency, support for compression (like Meshopt and Draco), and its ability to store various types of additional data. Moreover, it enjoys broad support and can be viewed in the Windows 3D viewer and edited in Blender. However, according to the format specification, the vertex arrays can only be of type `GL_FLOAT` or various integer types. Although some libraries also extend this to support `GL_DOUBLE`, this not widely implemented. To overcome this limitation, a workaround was implemented by storing normalized float positions with an offset. However, as the offset can also only be a float in many implementations, intermediary nodes were added to the GLTF scene, each with their own offset until the exact double offset was achieved. Although one might assume this results in an exact representation by combining a 64-bit double equivalent offset with a 32-bit position to store the original 64-bit double position, this is not the case, as evident when considering a simple example mesh consisting of only a single triangle shown in Figure 8.

Even if the positions are normalized the relative positions can still require more than 64-bit to store. In practise this approximation still performed very well in most cases. But as there are thousands of tiles the error was more pronounced in some of them and caused issues. Thus, it was decided to use a custom format for intermediate meshes during the LOD tree build process and only export to a GLTF for content delivery and debugging.

This custom .TILE format is straightforward, comprising a format version, the vertex and triangle arrays, and the PNG encoded texture, all written sequentially to a binary file.

This format thus ensures lossless storage and can be utilized for any intermediate meshes. Moreover, for debugging or delivery purposes, the option to output to GLTF, as described earlier, is also supported.

2.3 terrainmerger

The `terrainmerger` module is designed to handle the creation of the LOD hierarchy from a collection of reference tile meshes. This process currently involves repeatedly merging the meshes according to the Webmercator grid until a single root mesh is reached.

The merging process involves several steps:

- Merging of the geometry based on the border vertices
- UV unwrapping the merged mesh
- Reconstruction of the mesh texture
- Simplification with a zoom-level dependent error bound

2.3.1 Geometry Merging

Before attempting to merge the meshes, we conduct a sanity check to verify whether they can actually be merged, by checking if the bounding boxes of the meshes overlap. If so, then merging is possible, and the process continues. The merging process is made up of two parts: The actual merge operation that tries to find correspondences between border vertices of different meshes and a control loop that tries merge distances and validates the merge.

This control loop starts by estimating an initial merge distance, derived from a fraction of the minimum edge length across all source meshes. Vertices within this distance of each other from different meshes are candidates for merging. Subsequently, the loop iterates, attempting to merge the set of meshes with incrementally increasing merge distances until successful merging is achieved. Success is validated through a two-step process. Firstly, a union-find data structure confirms that all meshes are interconnected. Following this, we verify if the minimum vertex separation is greater than or equal to that of the original meshes. If these conditions are met, the merging is deemed successful. However, if unmerged vertices remain, indicating failure, the distance epsilon is adjusted accordingly for subsequent attempts.

To facilitate the core merge operation, we employ a 3D grid acceleration structure for efficient spatial lookups of vertex positions. The size of this grid is determined by the number of vertices and the size of the mesh according to this formula:

$$2 \cdot \sqrt[3]{\#\text{ vertices}} \cdot \frac{\text{Mesh size}}{\max\{\text{Mesh size}\}} \tag{1}$$

This results in approximately 30 vertices per grid cell, which showed optimal performance in preliminary tests. The merging process begins by inserting vertices from one of the source meshes into the grid. Subsequently, vertices from other meshes are inserted one by one. But

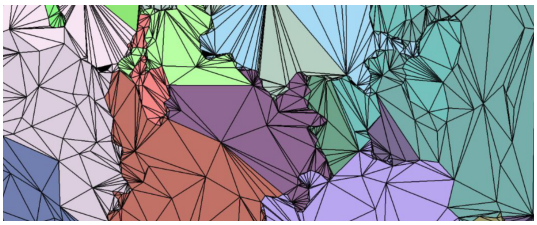


Figure 9: Triangle Cruft along locked borders.

before they are inserted, we check if there was already another vertex (from a different mesh) in the grid near that location and if so, we establish a correspondence between the vertices.

The merge operation establishes a mapping between vertices of original meshes and those in the resulting mesh. This mapping facilitates generating new geometry and aids in determining triangle correspondences, which are essential for texture generation of the merged mesh.

2.3.2 UV Unwrapping

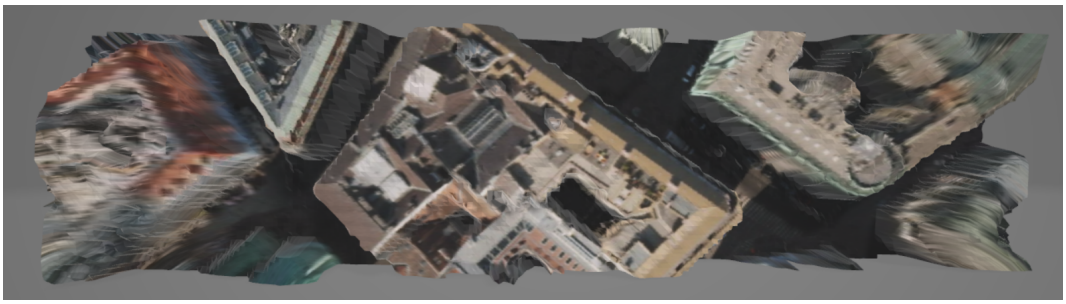
The subsequent phase involves generating a new UV unwrapping of the merged geometry, a process facilitated by the CGAL library. This step is required, as it is infeasible to reliably merge arbitrary UV spaces in such a way, as to not have the texture seams remaining fixed. Vertices on seams have to be locked during simplification, and lots of triangle cruft will result, if the seams stay the same throughout the LOD tree building process (see Figure 9).

Given that the UV coordinates are merely interpolated during simplification, it's imperative for the texture space to be continuous and convex. Failure to meet this criterion could lead to invalid UV coordinates or pronounced texture warping.

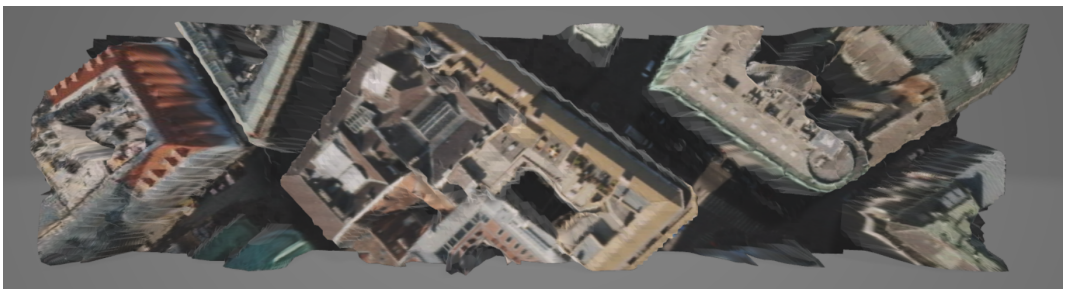
To ensure a continuous and convex UV space, we confine the border to a fixed convex shape. Even though this approach increases texture distortion (in comparison with free-border methods or a texture atlas), it should still produce acceptable results as the tiles are flat and homomorphic to a circle. However, there remains room for improvement as will be discussed in section 3. Our testing indicates that imposing a circular border yields less distortion at the edges and is more uniform compared to a square border (see Figure 10). This constraint limits the applicable algorithms to Tutte Barycentric Mapping, Discrete Authalic Parameterization, Discrete Conformal Maps, and Floater Mean Value Coordinates (compared in Figure 11).

2.3.3 Texture Construction

Based on the UV unwrapping it is now possible to generate a new texture for the merged geometry. This is done by filling in one triangle after the other. First the source triangle in one of the original meshes is found using the vertex correspondences established during merging. Then we copy the texture of this source triangle from the original texture and warp it to the newly unwrapped triangle. To limit the information lost due to resampling, we super



(a) Parameterized mesh with square border constraint.



(b) Parameterized mesh with circular border constraint.

(c) Texture from parameterization with square border constraint.

(d) Texture from parameterization with circular border constraint.

Figure 10: Comparison of square vs circular border parameterization methods. Note that the actual merged meshes are square in shape and the sequential arrangement is only used to exaggerate the distortions for comparison purposes.

(a) Discrete Authalic Parameterization

(b) Discrete Conformal Maps

(c) Floater Mean Value Coordinates

(d) Tutte Barycentric Mapping

Figure 11: Comparison of different surface parameterization algorithms supporting fixed border constraints.

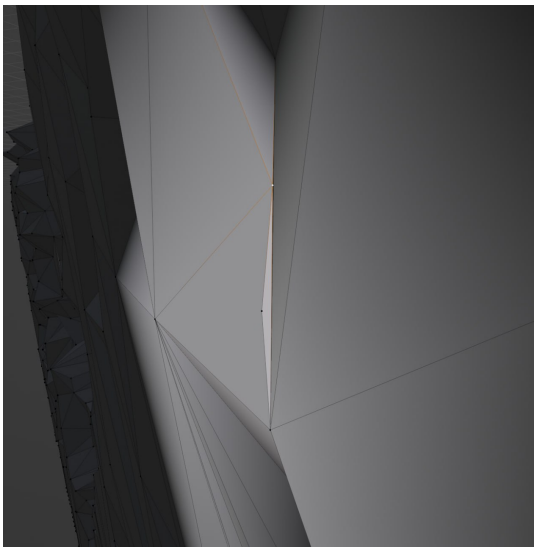


Figure 12: One of the non-manifold triangles resulting from only locking border edges.

sample the new texture. While the original tile texture has a resolution of 256px ×256px, all intermediate meshes use a resolution of 4096px ×4096px.

2.3.4 Simplification

The next step is simplifying the merged geometry. This is also done using the CGAL library. The simplification repeatedly performs edge collapses, until a given absolute error bound (other criteria are also supported by the codebase) is reached. This absolute error bound is given by the number of meters per pixel at that zoom level at the equator for 256 pixel wide tiles (given by `https://wiki.openstreetmap.org/wiki/Zoom_levels`).

Border Locking We cannot modify the border of the merged tile, as this would impede the merging at the next layer and thus result in holes. As such every edge, that connects to a border vertex is locked and is not considered for collapsing. Although locking solely border edges may seem preferable, it led to the formation of zero-sized triangles along the border, subsequently transforming into non-manifold triangles post-merging (see Figure 12). While these are not desirable, the real issue lies in the fact that non-manifold meshes are not supported in many CGAL algorithms, like parameterization.

Stop Condition As CGAL only supports very fundamental stop conditions, such as a target face, edge or vertex count, a custom solution had to be implemented for the required error bound. CGAL provides a mechanism to specify such a condition as a closure to be called after every edge collapse. But as it is invoked so frequently, it has to be cheap as to not slow down the process too much. Given that we employ the Hausdorff distance to gauge the absolute error relative to the original mesh—an inherently costly measure—we can't afford to compute it after every step. Instead, we opt to assess it every time 10% of edges have been collapsed since the last check or initialization. Consequently, we're only able to halt the sim-

plification process after the first time an error surpassing the specified threshold was detected. As such, a snapshot of the last valid mesh version has to be kept around to be restored once the error bound is overstepped. An alternative approach, inspired by binary search, could involve simplifying based on primitive counts (edge, face, or vertex). Initially, we'd simplify until half of the target primitives remain, then evaluate the current error and adjust the next target primitive count accordingly. This method could iteratively converge arbitrarily close towards the optimal target primitive count—minimizing the primitive count without violating the error bound—while requiring fewer steps within a specified edge range. But this approach cannot be implemented using a stop closure like the solution described previously and thus the simplification process would have to be restarted often. Each restart entails rebuilding all acceleration structures used by CGAL, adding to computational overhead. A hybrid approach might offer a faster solution, although it's more suitable for higher simplification ratios and less effective for lower ones. In general, it's essential to acknowledge the computational cost associated with Hausdorff distance checks, as this influences the overall efficiency of the simplification process.

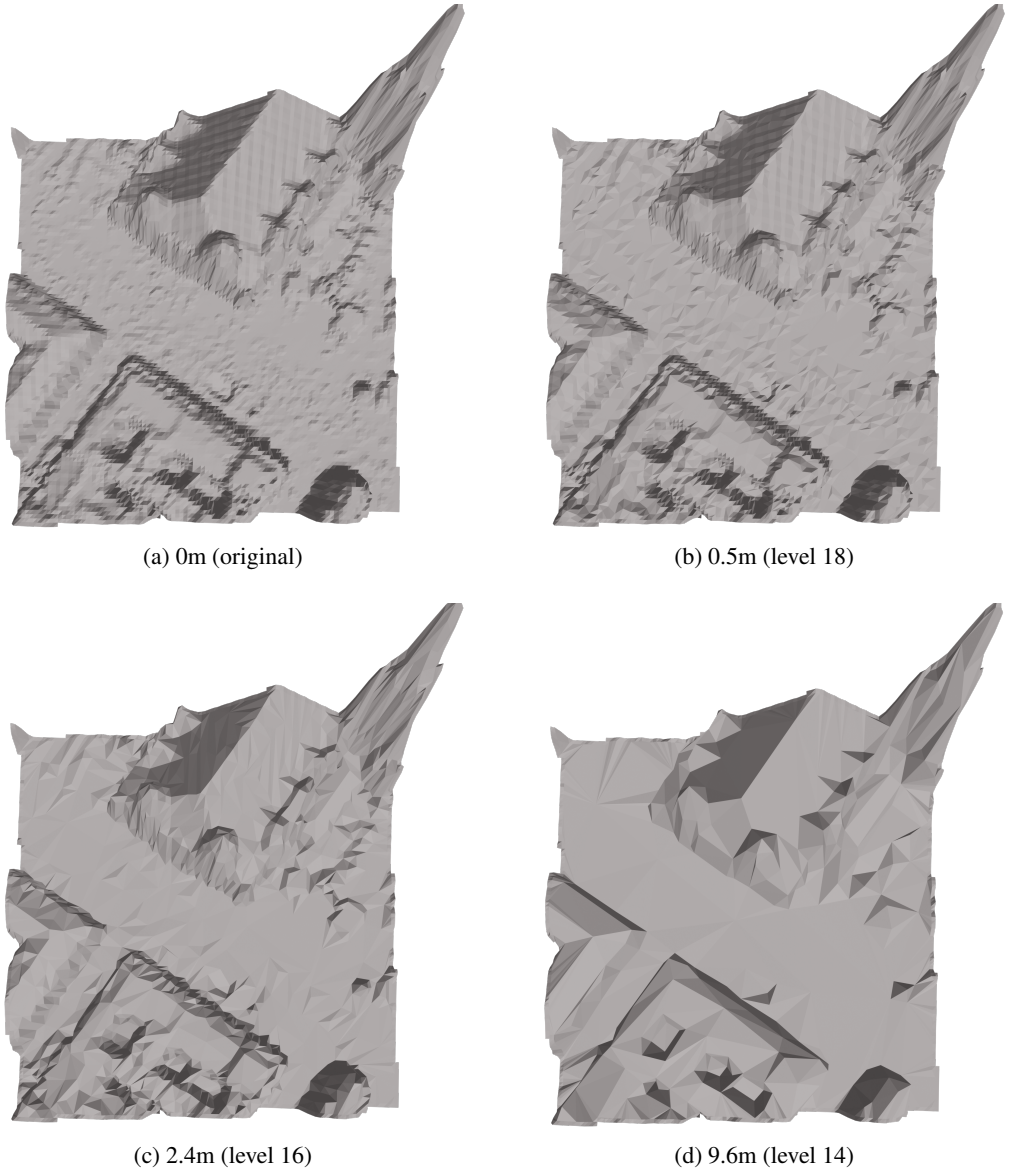
2.4 terrainconvert

The terrainconvert utility offers a straightforward command-line interface for converting between various tile formats. Currently, it supports the custom .TILE format and .GLTF/.GLB formats. This tool is useful for converting the base tiles (highest detail tiles) to .GLB files, since the terrainbuilder only produces .TILE files, while the renderer expects .GLB files.

2.5 renderer

The renderer is based on the already existing and in-use heightmap renderer, which generates geometry from heightmaps on the fly. It is a Qt 6 application, which uses Qt's signals and slots (Documentation: `https://doc.qt.io/qt-6/signalsandslots.html`) to manage the data flow through the application. The code was adjusted so that

- Only one service is used, that returns the tiles in .GLB format
- The LayerAssembler gets omitted, since both the geometry and texture data is already contained in the .GLB file, and nothing needs to get assembled.
- A GLTFReader is introduced to decode the GLTF data into texture and geometry data.
- The types are adjusted to
  - be able to hold the indices, positions, uvs and the texture data, and
  - upload them to the shader
- The vertex shader, that was formerly used to displace the geometry by the heightmap, is adjusted so that it only reads the data provided by the .GLB files, subtracts the



(a) 0m (original)

(b) 0.5m (level 18)

(c) 2.4m (level 16)

(d) 9.6m (level 14)

Figure 13: Comparison of simplified meshes with different absolute error bounds (and the corresponding zoom level).

camera's world position from it to convert it to a camera local space, and passes it on to further shaders.

For the rendering to work, the tiles have to be generated by the tilebuilder using the EPSG:3857 SRS, as the camera is using this SRS for its coordinate system.

In Figure 14 screenshots of the renderer displaying Vienna's city center can be seen. In the tile overlay we can see that high detail tiles are used nearer to the camera, whereas areas further away use lower detail merged tiles. At the same time, in the VertexID view, we can see that triangle sizes are relatively uniform with the exception of large flat areas being simplified more aggressively. In the normals view, we can see that we have good perceived mesh quality even for the lower detail tiles in the distance.

2.5.1 Hierarchy size in practice

For demonstration purposes, only a small region is used as larger regions would take too long to build. The region used is Vienna's city center with the root tile 13/4468/2840 (Google Tile Format: Zoomlevel/X/Y). The generated tile hierarchy for this region has a total possible size of 7,72 GB of which

- 3,78 GB (48,96%) are used by 5461 .TILE files
- 1,53 GB (19,82%) are used by 5461 simplified .GLB files
- 0,99 GB (12,82%) are used by 1365 merged .GLB files

For deployment, the only necessary files are the simplified .GLB files. The merged .GLB files are only saved when terrainmerger is called with `--save-debug-meshes` and the .TILE files are used while merging to preserve the most precision possible. This means that in total the given region of Vienna's city center (Root Tile: 13/4468/2840), uses 1,53 GB of simplified .GLB tile data. In Table 1 we can see the 7 zoom levels with their respective contribution to the total size.

3 Outlook

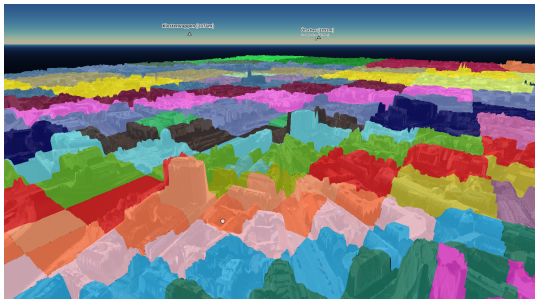
3.1 Directed Acyclic LOD Graph

The current implementation is based on an LOD tree instead of the directed acyclic graph (DAG) used in Nanite. Thus, the same border edges remain locked from reference mesh tile to the top collecting triangle cruft in the process. Constructing such a DAG would entail two additional steps while building the LOD hierarchy: Clustering a set of meshes into clusters to be merged and after merging partitioning each merged mesh again. Clustering is especially involved as to apply this process to the whole of Austria would entail a parallelizable solution and preferably one that can be run on different regions independently on multiple machine

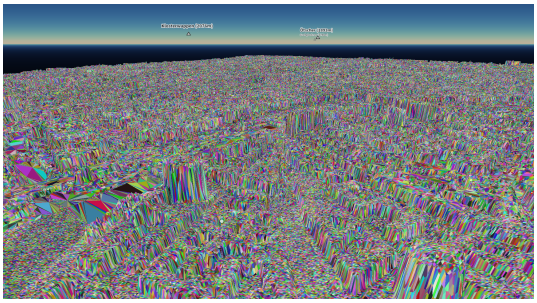
Zoom level	Size in MB	% of total (1,53 GB)	# files	Average size/file
13	10,5	0,7%	1	10,5 MB
14	33,6	2,2%	4	8,4 MB
15	58,2	3,8%	16	3,6 MB
16	100	6,5%	64	1,6 MB
17	180	11,8%	256	703,1 KB
18	323	21,1%	1024	315,4 KB
19	868	56,7%	4096	211,9 KB

Table 1: Comparison of hierarchy sizes between each zoom level. Note: The program used for calculating the sizes of each layer is not consistent in how it sums the file sizes. Therefore, "Size in MB" and "% of total" do not add up to 1.53 GB and 100% exactly and should be interpreted as a ballpark measure for the layer sizes.

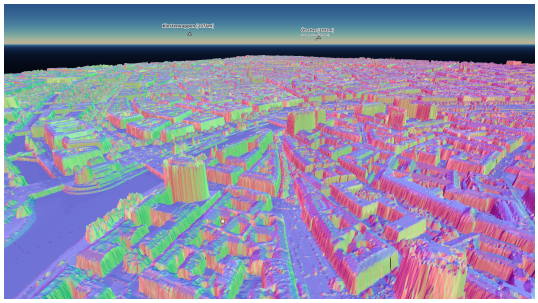
(a) No Overlays



(b) Tile Overlay



(c) VertexID Overlay (i.e. Triangle View)



(d) Normals Overlay

Figure 14: Rendering overlays showing the rendering of various level of detail tiles.

without synchronisation. Such a clustering algorithm would have to provide a predicable result independent from the target/starting area. Partitioning the merged meshes can be achieved using the METIS library and was already demonstrated successfully.

3.2 Compression

At the moment the reference mesh tiles and LOD meshes are uncompressed. This is fine for now but if tens of thousands of meshes are to be stored this should be subject to change. To facilitate this initially Draco and MeshOpt compression were determined to be suitable. But from preliminary testing Draco compression appears superior, as it supports lossless compression and the lossy variant results in higher fidelity with lower file size.

.tile 83.62 MB

.gltf 53.48 MB

.glb 39.35 MB

.glb (lossless draco) 24.11 MB

.glb (lossy draco) 12.78 MB

.glb (lossy meshopt) 15.85 MB

3.3 Advanced UV Unwrapping

The texture resampling solution described earlier generally produces acceptable results but is has some issues. For once currently meshes with multiple connected components are not supported, although it should be possible with limited changes. Another problem is that repeated resampling results in quality loss that accumulates over many merge operations (see Figure 15). This was already partially addressed by increasing the texture resolution for intermediate meshes. On the other hand, a completely new approach for texture handling could be required to ensure the fidelity and alignment of the textures is kept after numerous merge and resampling operations. One such solution could be to remember the vertices that were combined to form each triangle in the simplified mesh and then to project them onto the triangle to generate the new texture for this triangle. Another could be to simply use the orthophotos for the higher zoom level tiles directly if no modification was made to the terrain data (through integration of drone scan data or similar)

3.4 Blender Integration for .TILE

The decision was made to use a custom format for mesh storage, which violated one of the requirement of an easy editing and viewing experience. This could be fixed by implementing a custom import and export for this custom format in blender. The need for such an integration could become more prominent once additional geometry information is to be manually integrated.

Figure 15: A demonstration of the accumulated resample error observed with reduced texture size for intermediate meshes.