TU WIEN Informatics

Digital Surveying of Large-Scale Multi-Layered Terrain

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Kevin Streicher, B.Sc.
Matrikelnummer 01025890

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Dipl.-Ing. Dr. Christoph Traxler

Wien, 25. Juli 2022

Kevin Streicher	Michael Wimmer

TU WIEN Informatics

Digital Surveying of Large-Scale Multi-Layered Terrain

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Kevin Streicher, B.Sc.
Registration Number 01025890

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Dr. Christoph Traxler

Vienna, 25th July, 2022

Kevin Streicher	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Kevin Streicher, B.Sc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. Juli 2022

Kevin Streicher

Danksagung

Für Iris.

Ich möchte mich bei meinem Professor Dipl.-Ing., Dipl.-Ing, Dr.techn Michael Wimmer für die Betreuung und Unterstützung bedanken. Im Besonderen möchte ich mich bei ihm für die schier endlose Geduld bedanken und dafür mich stets mit einer guten Idee wieder in die richtige Richtung gestupst zu haben. Ebenso möchte ich mich bei Dipl.-Ing. Dr. Christoph Traxler bedanken, der mein Interesse in die Planetenwissenschaften, Archäeologie und Geowissenschaften überhaupt erst geweckt hat.

Danke an Dr.techn Thomas Ortner, der mir bereitwillig dieselben Fragen wieder und wieder beantwortet hat,

Danke an VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH für die Unterstützung des Projektes.

Danke an JOANNEUM RESEARCH, welche die 3D-Daten für Gale Krater, Victoria Krater, Homeplate, Stimson MSL Sol 1087 und StereoMosaic (MSL Sol 1275) aufbereitet haben.

Ich bedanke mich bei der Österreichischen Post AG, welche meinen Weg unterstützt hat.

Im Besonderen bedanke ich mich bei meiner Familie, die mir diesen Weg eröffnete und mir stets mit Rat beistand.

Acknowledgements

For Iris.

I would like to thank my advisor, Dipl.-Ing., Dipl.-Ing, Dr.techn Michael Wimmer, for his supervision and support. Most of all, I am thankful for his sheer endless patience, and nudging me right back on track with a good idea. I also want to thank Dipl.-Ing. Dr. Christoph Traxler, who was the first to draw my interest to planetary sciences, archaeology and geology.

Thanks to Dr.techn Thomas Ortner, who was willing to answer the same questions over and over again.

Thanks to the VRVis Zentrum für Virtual Reality und Visualisierung Forschungs-GmbH for supporting the project.

Thanks to JOANNEUM RESEARCH, which produced the 3D data for Gale Crater, Victoria Crater, Homeplate, Stimson MSL Sol 1087 and StereoMosaic (MSL Sol 1275) scenes.

Thanks to the Österreichischen Post AG, which supported me all along the way.

Special thanks to my family, which opened up this way for me, and guided me whenever needed.

Kurzfassung

Die Geomorphometrie ist die Wissenschaft der quantitativen Analyse von Terrainoberflächen. Die durch Vermessung des Terrains quantifizierten Oberflächen erlauben die Bestimmung der geomorphometrischen Eigenschaften, wie Höhen, Krümmungen, Neigungen und Distanzen. Diese sind für die Analyse von Terrains in Archäeologie, Geologie und Planetenwissenschaften, sowie anderen, von Bedeutung. Durch digitale Rekonstruktion können Terrains abseits der Forschungsstätte digital vermessen werden. Die durch hohe Auflösung oder große Abmessungen entstehenden großen Datenmengen stellen eine Herausforderung für die Echtzeitvisualisierung mit interaktiven Bildraten dieser Terrains zur explorativen Erkundung dar. Dies erfordert es, geringere Auflösungen oder kleinere Terrainauschnitte zu wählen. Durch Terrainstreaming können größere oder höher aufgelöste Terrains dargestellt werden. Da durch Fehler in der Rekonstruktion Ungenaugikeiten entstehen, ist es wichtig auf diese zu quantifizieren und auf sie aufmerksam zu machen.

In dieser Diplomarbeit wird ein Out-of-Core Rendering Algorithmus für große Terrains mit mehreren Ebenen präsentiert. Der präsentierte Algorithmus erreicht Terrainstreaming in interaktiven Bildraten für Szenen mit bis zu 775 M Dreiecken und 156 GB in der feinsten Detailstufe und einer Gesamtgröße von 222 GB.

Weiters wird ein verbesserter Messalgorithmus für die digitale Vermessung großer Terrains mit mehreren Ebene präsentiert. Der Algorithmus verwendet Unterabtastung mit variabler Rate (VRSS) und Erkennung gemeinsamer Kanten (SED) und wird als VRSS+SED bezeichnet. VRSS+SED erzielt bessere Resultate als Unterabtastung mit fixer Rate (FRSS), der Abtaststrategie die in State-of-The-Art Werkzeugen für planetare Geologie, wie dem *Planetry Robotics 3D Viewer (PRo3D)* verwendet wird. Die frühere Terminierung bei höherer Genaugikeit und gleicher Anzahl an Abstastungen wird erzielt, indem die Ray Casting Ebene mit der geteilten Kante geschnitten wird um analytisch einen Mittelpunkt zwischen zwei benachbarten Primitiven zu berechnen. Weiters wird eine neue Unsicherheits-Metrik namens *On-Data Ratio (ODR)* präsentiert, die es es erlaubt auf Unsicherheiten in State-of-the-Art Messalgorithmen aufmerksam zu machen.

Die präsentierten Algorithmen werden evaluiert indem ein in Unity unter der Verwendung von des *Data-Oriented Techstack (DOTS)* implementierter Prototyp verwendet wird. Die Algorithmen werden gegen PRo3D evaluiert und die Resultate präsentiert und diskutiert. Der vorgestellte Algorithmus erreich, bei ähnlicher Speichergröße, 15x schnellere Ladezeit als PR3oD für die 222 GB große Szene

Abstract

Geomorphometry is the science of quantitative analysis of terrain surfaces. By surveying terrains to quantify their surfaces, it is possible to calculate the geomorphometric properties, such as heights, curvature, slopes, and distances. These are important for the analysis of terrains in archaeology, geology, planetary sciences, and others. By using digital terrain reconstructions, off-site terrain surveying becomes possible. The high resolution or large scale of terrains are a challenge for real-time rendering at interactive frame rates for exploration. This requires limiting resolution or loading smaller terrain parts. The use of terrain streaming allows rendering higher resolution or terrains of greater extents. As errors remain, it is important to quantify and visualize them.

In this thesis, an out-of-core rendering algorithm for large-scale multi-layered terrain is presented. The presented streaming algorithm manages to stream scenes with 775 M triangles and 156 GB on their finest LOD, and a total size of 222 GB, at interactive frame-rates and on commodity hardware.

Additionally, an improved measurement algorithm for digital terrain surveying of large-scale multi-layered terrain is presented in this thesis. The measurement algorithm using variable-rate subsampling (VRSS) and Shared Edge Detection (SED), is called VRSS+SED and achieves better results than the fixed-rate subsampling (FRSS) strategy used in state-of-the-art planetary geology tools such as *Planetry Robotics 3D Viewer (PRo3D)*. It achieves earlier termination at higher precision for the same number of samples by intersecting found shared edges with the ray casting plane to analytically calculate the midpoint between two neighboring primitives. Furthermore, a novel uncertainty metric called *On-Data Ratio (ODR)* is presented which allows raising awareness about the uncertainty in the results of the used state-of-the-art measurements algorithm.

The presented algorithms are evaluated using an implementation in a prototype using the Unity engine and its *Data-Oriented Techstack (DOTS)*. The algorithms are evaluted against PRo3D and the results are presented and discussed. The presented implementation achieves 15x as fast loading times as Pro3D for the 222 GB large scene at a similar storage size.

Contents

CHAPTER 1

Introduction

1.1 Motivation

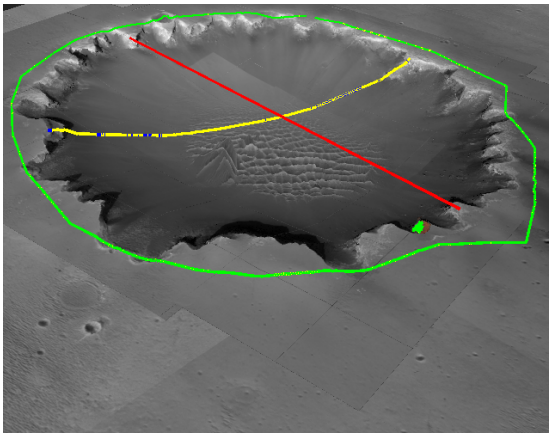
Terrain surveying is the act of measuring a terrain's properties, such as height, curvature, slopes and distance. These are later analysed and interpreted in the sciences of archaeology, geology, planetary sciences, and others. By using digital terrain reconstructions, surveying can be done digitally and off-site. In digital terrain surveying, users explore terrain reconstruction interactively, searching for features of interest. Real-time rendering is in consequence necessary. The data of interest can easily become much larger than the available memory. This is a challenge for realt-ime rendering and measuring algorithms, and thus, out-of-core algorithms are needed. Even with out-of-core solutions, the scale and resolution at which interactive framerates can be achieved are limited. Using a lower resolution leads to inaccuracies and errors, and consequently to uncertainty. To avoid false trust in the result of measurements performed across areas of uncertainty, it is neccessary to quantify such uncertainties and raise the users' awareness of their existence. Figure 1.1 shows a comparison of on-site terrain surveying and digital terrain surveying.

The high-performance requirements for digital terrain surveying software often require specialized tools. Higher-level usually efficient general purpose rendering and physics subsystems. Using the rendering and physics subsystems of a higher-level engine, and only implementing use case specific tools from scratch, could for this reason be one way to make it easier and cheaper to develop specialized digital terrian surveying software. The abstraction and functionality these engines provide come at the cost of less flexibility. A prerequisite to benefit from the provided functionality is often close adherence to the provided data models and algorithms.

The goal of this thesis is to evaluate the benefits and limitations of using such a higher-level engine, Unity, and to present an implementation that supports streaming of large terrains, rendering them at interactive framerates and measuring geomorphometric properties,

while also quantifying the uncertainty of those measurements and visualizing them. The developed software is named *Visionary*.

(a) Geologists discussing an outcrop during a field trip. The widths of thin bedding, as estimated by the professor's hands, are important indicators. Image reprinted from Mike Beauregard [Can m].



(b) Digital terrain surveying of the Victoria Crater with Visionary. The green circle and yellow-blue polylines measure walking distances around and through the crater, the red line evaluates the diameter. The small green dot are multiple annotations on an outcrop.

Figure 1.1: Comparison of terrestrial on-site terrain surveying and digital terrain surveying.

1.2 Problem Definition

In digital terrain surveying of large-scale terrains, a central problem is achieving interactivity. Interactivity is important, as terrain surveying is highly explorative.

The large size of terrain data can be challenging for achieving rendering at interactive framerates and quick measurements. The resolution and scale of the terrain are the main contributing factors to the data size. An increasing resolution allows for more precise terrain reconstructions and hence exact measurements. A larger scale not only allows exploring a larger terrain region, but also can help to understand the surrounding context of a terrain surface.

When the data size exceeds the available memory, out-of-core solutions are required. *Terrain streaming* is such a solution. In terrain streaming the terrain is partitioned into sufficiently small *chunks*. These chunks are then swapped (streamed) between slower bulk memory and fast main and graphics memory. Compact storage and efficient data layout are requirements for fast streaming from slow bulk storage. A streaming and level-of-detail selection that supports efficient rendering is in consequence required. Solutions differ between client-side and server-side storage and the metrics used to decide which chunks to prefetch or stream in.

To increase the rendering performance, *level-of-detail (LOD)* algorithms can be used. LOD algorithms replace objects in visually less important areas with lower-quality representations to increase the rendering performance. LOD algorithms differ in how LODs are generated, if they are adaptive or reactive, discrete or continuous, and how LODs are selected. In digital terrain surveying, measurements have to be performed on the actual data. This means that while we can utilize LODs for rendering, we cannot do so for measurements and might require loading both the low- and high-resolution version of a chunk during a measurement. However, other acceleration algorithms like view-frustum culling and spatial acceleration structures like bounding volume hierarchies can be used to speed up measurements.

At large scales, numeric precison becomes an issue. Single-precision floating-point formats (floats) are the dominant numeric type for efficiency reasons. Floats only provide 6-7 significant digits, which can easily become a problem with real-world quantities. Most of all, the vertex position values can become too large. Coordinate system transfer, e.g., origin shift, is one method to reduce this problem.

One method for digital terrain surveying is via user-drawn polylines and projecting the polyline segments onto the terrain. This method is implemented in PRo3D [Prob], one of the state-of-the-art digital terrain surveying solutions for large planetary terrains. Geomorphometric properties are evaluated at the user-selected feature points. The terrain profile is then reconstructed by connecting two consecutive points by a straight line. Subsampling is used to increase the accuracy of the resulting profile. One way to achieve the projection of the polyline is via ray casting. The performance of these raycasts is thus crucial. The choice of subsamples is important, as it can heavily affect the result. One straightforward subsampling strategy is fixed-rate subsampling. In fixed-rate subsampling, a fixed number of subsamples is evaluated in equal-distance intervals. This method is used in PRo3D, but it does not take the geometry into account, and is for this reason likely to over- or undersample. A better approach is variable-rate subsampling, where more subsamples are used in areas of complex geometry and less in areas with little change in profile. However, the quality and performance of variable-length subsampling is dependent on the subsampling algorithm and termination criteria. A simple termination criteria is a fixed number of rays, but with differently distributed sample points.

Due to missing data, reconstruction errors and early termination, it is not clear how certain a measurement is. When two subsamples are connected by straight line, they are only an exact reconstruction of the surface profile, if they are connected by coplanar primitives, e.g., triangles or quads. A line segment is for this reason considered certain, if it embedded in a planar primitive. All other segments of the reconstructed profile are in consequence a source of uncertainty. To give users the chance to judge if a measurement can be trusted, it is necessary to quantify this uncertainty and visualize its source.

State-of-the-art solutions such as the Planetary Robotics 3D Viewer (PRo3D) are highly specialized tools. Using a higher-level engine as framework instead promises faster development because of already provided functionality. However, this can easily turn out as a false promise when the need to adhere to provided algorithms and data structures

becomes more restricting than helpful. It is consequently required to make such limitations visible.

1.3 Contributions

The main contribution of this thesis is a streaming algorithm for out-of-core rendering of large scale terrain. The algorithm streams large-scale multi-layered terrain in general 3D-mesh representation on commodity hardware. It is based on the out-of-core streaming algorithm and render front selection by Varadhan and Manocha [VM02], which is adapted for independent, asynchronous and parallel streaming of *multiple hierarchical level-of-details (M-LODs)*. Similar to their algorithm, the potentially visible front of the scene graph is calculated. Nodes are then streamed in until a given resource quota is filled. A memory and a target frame-rate scheduler are implemented to ensure efficient streaming. View-frustum culling is used to cull entire HLODs. In contrast to their solution, a Euclidean distance based metric for discrete general meshes using *Axis Aligned Bounding Box (AABB)* and LOD ranges are used. Utilizing a single metric for both streaming selection and LOD selection allows to reuse the calculated results. The streaming algorithm achieves interative frame-rates for a scene with 775 M triangles and 156 GB on its finest LOD, and a total size of 222 GB.

Furthermore, an out-of-core measurement algorithm for digital terrain surveying of large-scale multi-layered terrain is presented. The polyline based surveying tool is based on the method used in the state-of-the-art digital terrain visualization tool *Planetary Robotics 3D Viewer (PRo3D)* [Prob]. The fixed-rate subsampling strategy (FRSS) used in PRo3D is improved by using variable-rate subsampling (VRSS) instead and the Shared Edge Detection (SED) algorithm presented in this thesis. Compared to the FRSS, VRSS+SED does terminate earlier and with higher accuracy at equal sample count. VRSS+SED achieves this by intersecting a shared edge with the ray casting plane to calculate an exact midpoint between neighbouring primitives analytically.

Lastly, a novel but simple uncertainty metric, called On-Data Ratio (ODR) is presented in this thesis, that allows to quantify the uncertainty of such measurements. ODR allows quantifying uncertainty and to raise awareness about its presence in the polyline based state-of-the-art measurement algorithm. ODR also allows gaining trust in a measurement when no uncertainty is detected, and the surface profile reconstructed from subsamples 100 % reflects the actual surface profile.

1.4 Structure of the Work

In Chapter 2 the current state-of-the-art in geospatial visualization and terrain streaming is presented. A summary of the advances of high-level game engines in recent years, and how they were adopted to reduce development time and costs in game development, and other fields, is given. An overview of the state-of-the-art terrain representations used in GIS is given, and their benefits and drawbacks are highlighted.

The terrain streaming rendering algorithm proposed is presented in Chapter 3. The used data structures, implemented serialization methods, the used origin shift and algorithms and metrics for LOD selection and streaming are explained.

The algorithm for measuring in a streamed environment is given in Chapter 4. The two-phase algorithm for asynchronous ray casting against streamed colliders, the *Shared Edge Detection* termination criteria for polyline subsampling and the *On-Data Ratio* metric for uncertainty, are covered.

In Chapter 5 the implementation utilizing Unity's DOTS is presented. The *Ordered Point Cloud (OPC)* format, which is the format of all input data, is explained. The streaming and GPU-optimized binary format, called *Renderable Blob* format, all data is preprocessed into, also is explained.

The results are presented and discussed in Chapter 6.

The conclusion can be found in Chapter 7, where the thesis and gained insights are summarized and a few selected options for potential future research are highlighted.

CHAPTER 2

Background and Related Work

In this chapter, an overview of the current related and ongoing research, methods and available software focused on digital terrain surveying and analysis, rendering large-scale terrains, general-purpose raycasting and spatial acceleration structures is given. Furthermore, an overview on the advancements of higher-level engines and their applications in other industries and sciences is given. Lastly the historical background of some of these advancements is explained.

2.1 Digital Terrain Analysis

The methods of digital terrain analysis (DTA) have been significantly improved since the development of geographical information science (GIS) systems in the 1970s. Xiaong et al. [Xio+21] systematically analyzed the current state of research of geomorphology-oriented DTA. They emphasized how modern remote sensing technologies allow to rapidly and easily acquire multi-temporal, multi-level, and multiscale landform morphology digital elevation model (DEM) data, but as DEM data only contain location and elevation information, it can only express the surface information. However, the DEM is still the most widely used terrain representation as it is capable of representing surfaces well, and many DTA methods based on it are well established. DEM is only one of multiple state-of-the-art terrain representation models. Natali et al. [Nat+13] gave a classification and overview of multiple terrain and subsurface modeling techniques fit for different use cases. They highlight the strengths and weaknesses of them for different requirements, and found that none of the discussed modeling techniques was suitable for all situations. Paar et al. [Paa+15a] showed how mesh-based terrain representation can be used to embed sensor visualization in martian terrain reconstructions. They have used projection of user-drawn polylines for terrain surveying. This is one of the most basic and widely used techniques for interpretation of Digital Outcrop Models (DOM). Barnes et al. [Bar+18] presented and evaluated their use of this technique for geological analysis

in PRo3D against terrain reconstructions based on the Mars Exploration Rover (MER) missions. Their used file format for the Martian terrain reconstructions was the Ordered Point Cloud (OPC) format, as this was supported by their software PRo3D. This format was presented by Ortner et al. [Ort+11] in 2010 for arbitrary level-of-detail hierarchies.

There are several open-source GIS and DTA applications suitable for DOM interpreation available:

Planetary Robotics 3D Viewer (PRo3D) [Prob] was released as open-source software to contribute to open science [Proa]. It supports large-scale terrain by rendering level-of-detail hierarchies in OPC format. It allows geomorphometric analysis via distances and dip-and-strike. Furthermore, it is mainly geared towards planetary geologists, with interactive tools to digitze geological features on digital outcrop models (DOMs) [Bar+16].

GRASS [GRA22] is focused on raster, vector and geospatial processing. It includes tools for terrain and ecosystem modeling. It supports many common GIS file formats that are supported by the *GDAL-OGR* library. Its features include 3D raster (voxel) analysis, point cloud analysis, terrain analysis. Its focus is much stronger on DEMs than on DOMs.

There also is a wide range of commercially available GIS and DTA application, with a few important listed here:

LIME [Buc+19] supports multiple 3D and elevation models, handles massive texture models through out-of-core level-of-detail rendering and allows DTA (dip/slope, dip direction/aspect, brigthness/contracst, etc.). Amongst many other tools, it provides a rich toolset for DTA for DOM.

VRGS: Virtual Reality Geological Studio [Vrg] is dedicated to geology and allows analysis for structural and engineering eology, sedimentology and virtual field trips and site visits. It supports point cloud data and mesh interpretations, as well as LOD generation.

ArcGIS [3dg] is a suite of not only one, but several GIS and DTA tools. It offers a wide range of features for map, terrain, remote-sensing, and city data. Some of them are available as Software-as-a-Service (SAAS) platform, and several tools can be licensed as standalone clients. It provides real-time GIS support. Most of its features are geared towards city, map and in general DEM-based data.

Favorskaya and Jain [FJ17] provide an in-depth analysis of the state-of-the-art advances in remote sensing and geographic information systems for both terrain and vegetation modeling.

The DTA measurements tools in this thesis are based on the polyline-based tools for DOM interpreation as implemented in PRo3D.

2.2 Large-scale terrain rendering

Rendering large-scale scenes interactively is a fundamental problem of computer graphics. The common goal is to achieve high-fidelity representations within the given time budget. It is challenging for all kinds of representations, such as DEM-, voxel- or mesh-based. Each model provides different benefits, differs in its expressiveness, and optimization techniques. In consequence, a lot of research has been conducted to optimize and improve the rendering performance for large-scale data and widely different methods and solutions are available.

As highlighted in Section 2.1 the choice of terrain representation is critical for digital terrain analysis, but it also is important for the rendering performance. The efficiency of DEM for terrain surfaces is the main reason why it is the most widely used format, especially for large-scale terrain rendering in GIS and DTA. In many higher-level engines DEM-, and so heightmap-based terrain is often the only type of terrain representations. Both Unreal Engine [Unrb] and Unity [Tecf] only refer to terrains as heightmap-based terrains and consider other meshes, regardless if they represent terrain or not, as general meshes. To render a DEM terrain it is required to convert it to primitives supported by the graphical processing unit (GPU), with triangle meshings being the most common primitive. Delaunay triangulation [Del+34] is widely used and so are Real-time Optimally Adapting Meshes [Duc+97] (ROAM). A similar CPU-based mesh refinement method was presented by Lindstrom and Cohen [LC10], using binary trees. Other rendering terrain rendering methods are using tiled blocks, where the terrain is partitioned into square patches, tesselated at different resolutions. In tiled approaches seamless stitching between blocks of different tesselation level is difficult. Another method is the use of regular grids using geometry clipmaps, which define a hierarchy centered around the viewer [TMJ98].

Volumetric representations, allow representing not only surface, but also subsurface information. Peytavie et al. [Pey+09] presented material layers as a two dimensional grid alternative less computationally demanding than voxels. FarVoxels [GM05] and GigaVoxels [Cra+09] have been approaches for ray-casting-based rendering of large voxel-based scenes. Graciano et al. [GRF18] presented a real-time visualization system for subsurface geological structures using a stacked layer (voxel) representation.

Another different approach to polygon- or volume-based rendering is point-cloud-based rendering. Instead of polygonal meshes that represent a closed surface, both terrains and objects are represented by points. All points are then rendered using basic primitives, which allows for fast rendering. QSplat [RL00] was the first point renderer being able to handle hundreds of millions of points. Schütz [Sch16] presented a web-based rendering solution, Potree, which is capable of rendering hundreds of billions of points in real time in a standard web browser. This was achieved by using a modifiable nested octree (MNO) structure and out-of-core rendering. Schütz et al. [SKW19] also presented an in-core solution capable of rendering point clouds fast enough for VR application. Schütz [Sch21] presented an algorithm for fast out-of-core generation of octress for massive point clouds.

Level-of-detail (LOD) algorithms are essential for rendering large-scale geometry. Early

pioneering work was done by Schumacher [Sch69] and a lot of improvements have been made since then. Lindstrom et al. [Lin+96] presented a hierarchical solution to optimize rendering of height fields. Erikson et al. [EMBI01] presented hierarchical levels of detail (HLODs) to drastically simplify entire branches of the scene graph to support rendering large environments. Guthe and Klein [GK04] presented an extension to streaming HLODs for an out-of-core framework to visualize huge polygon models.

Once data sizes become larger than working set memory, out-of-core methods are required. Lindstrom and Pascucci [LP02] presented a general framework for view-dependent out-of-core visualization. They presented a method for efficient view-dependent refinement and a memory-friendly indexing strategy. Varadhan and Manocha [VM02] presented an out-of-core rendering algorithm for massive geometric environments. Their algorithm focuses on efficient prefeteching, HLOD selection and parallel rendering and I/O management. Amara and Marsault [AM09] presented an GPU-based architecture for terrain rendering with the use of general meshes called Tile-Load-Map. Their architecture is focused on performing texture selection, LOD and streaming selection and rendering on the GPU.

Parallelization is another approach for large-scale rendering. Dong [Don20] presented a multi-gpu multi-display system for rendering large and complex 3D environments. Distributing the workload not only across multiple GPUs, but across multiple network-connected nodes has been researched for a long time. A networked solution for QSplat was presented by Rusinkiewicz and Levoy [RL01]. Guthe and Klein [GK04] presented a network-based streaming solution for HLOD. Lerbour et al. [LMG10] presented a generic solution for rendering large terrains. Their solution is network-based and they reduce redundancy of data by instead of duplicating it, they reuse it across LODs. By merging and splitting LOD nodes, organized in a quadtree, based on the nodes' importance, they reduce loading of not relevant anymore data. Bethel (Ed) et al. [Str] give an overview of highly parallelized rendering methods using network-based parallelization across many networked rendering clients and methods for distributing and scaling rendering frameworks across them, as well as utilizing GPU-accellerated rendering techniques for high-performance rendering. In recent years, in particular cloud computing became an active research field for rendering large-scale models. Xue et al. [XZQ19] presented a framework for large-scale rendering of CAD models, using parallel rendering using GPU virtualization and cloud computing.

Gobetti et al. [GKY08] provide an overview of techniques for visualizing massive models. Favorskaya and Jain [FJ17] give an overview of the current state-of-the-art in rendering large scenes for both terrain and vegetation.

The input scenes in this thesis are in OPC file format. The stored triangular meshes are preprocessed into a streaming optimized custom file format. The HLOD hierarchies provided by OPC files are used for LOD selection and streaming selection is performed across multiple HLODs. The out-of-core streaming algorithm is based on the streaming approach by Varadhan and Manocha [VM02].

2.3 Ray casting and Spatial Acceleration Structures

2.3.1 Ray casting

One of the most basic algorithms in computer graphics is ray casting, also referred to as ray shooting, or ray tracing. By following the path of an oriented half-line, called a ray, from its origin along its direction, its points of intersection with geometry can be calculated. Shirley presented the use of ray tracing for rendering in 1990 [Shi90]. Since then many improved ray-tracing-based methods have been developed. A lot of research was focused on how to improve the time complexity of $O(N)$ for ray tracing, where N is the number of objects, by the use of spatial accelaration structures. Havra [Hav00] presented several techniques to reduce time and space complexity for *kd*-tree-based ray tracing. Haines (Ed.) and Akenine-Möller (Ed.) [AMHH19] gave an in-depth overview of many algorithms and optimization for high-quality real-time rendering with ray tracing. With improving algorithms, data structures and hardware support, ray tracing became a method not only for rendering, but also for other ray-casting-based geometric queries, e.g., in physics simulations. The most common queries are solving visibility problems, such as first-hit ray traversal and multi-hit traversal. In first-hit traversal the closest point from the origin in direction of the ray is searched for, in multi-hit traversal all hit points are desired. Wald et al. [Wal+19] presented a proof-of-concept using hardware ray tracing for point-in-tetrahedon tests.

The presented measurement algorithm uses ray casting for first-hit traversal. This is used for the projection of polylines for feature and subsample selection.

2.3.2 Spatial Acceleration Structures

Spatial acceleration structures (SAS) are essential for achieving real-time performance for many rendering and physics related algorithms. They are used for node traversal in sublinear time and are important for LOD hierarchies. This is achieved by structuring objects based on position, extents and/or geometry in hierarchical data structues. Since Bentley [Ben75] presented multidimensional binary search trees to improve searching in database in 1975, there has been a lot of research on appliances of acceleration structures for computer graphics algorithms. The two most common types are *bounding volume hierarchies (BVHs)* and variants of *binary space partitioning (BSP)* trees. For brevity, only a few selected historical highlights are listed, without downplaying the important role many of the ommitted algorithms and datastructures had historically.

Clark [Cla76] presented hierarchical structures for geometric models to improve visible surface detection. Meagher [Mea82] presented the use of octress to encode arbitrary geometric models in 1981. Hanan [Sam84] presented quadtrees and many quadtree-related algorithms as general data structures to support image processing, geographic information systems, and robotics, in their survey in 1984. Klosowski et al. [Klo+98] presented an algorithm for collision detection using bounding volume hierarchies in 1998. Gobetti et al. [BHH15] presented an incremential approach for BVH construction for ray tracing in

2015. Zellmann [Zel19] gave an overview of state-of-the-art hierarchical data structures for volume rendering with empty space skipping in 2019. Yesantharao et al. [Yes+21] recently presented a parallel batch-dynamic kd-tree, called BDL-tree, in 2021. Meister et al. [Mei+21] presented a recent survey of the state-of-the-art in bounding volume hierarchies for ray tracing in 2021.

Overall, different SAS excel for different paramters. Some of the most important parameters are the following:

- Performance of algorithms for querying the SAS [GD21].
- Type of queries performed against the SAS [Laz+21; Wal+19].
- Storage and memory efficiency [EBN13; Hua+20].
- Duration to build or update the SAS [Sch21; WH06].
- Availability of algorithms parallelized across multiple CPU [Wan+20], GPU [Hor+07; SOW20] or cluster-based architectures [JSW22].
- Support for static or dynamic (animated or deformable) geometry [Jia+20].
- Type of geometry or volumetric representation (2D, 2.5D, triangular, point cloud [EBN13; Sch21], volumetric and implicit isosurfaces [QLX21; Str+20; Wan+20]).

The cost of network-based solutions or HPC environments justifies the research of algorithms that also are efficient enough for commodity hardware or even restricted environments, such as mobile platforms.

The interested reader is referred to Ericsons [Eri04] book *Real-time collision detection* for detailed information and implementation examples of data structures and algorithms for real-time collision detection. Spatial acceleration structures of various types are covered, as well as many optimizations for cache-aware algorithms. Complementary information also can be found in the book *Real-Time Rendering* by Akenine-Möller et al. [AMHH19]. There they give an in-depth explanation of different spatial acceleration structures and implementations for rendering are provided. The overlap in content shows how relevant SAS are for algorithms for both rendering and physics.

In this thesis, Havok for Unity and raycasting against its AABB-based bound volume hierarchies were used [Unia; Unib; Unih; Uni19].

2.4 Higher-Level Engines

Higher-level engines provide implementations for commonly required datastructures and algorithms. 3D game engines in particular provide functionality for both efficient rendering of mesh-based data and physics, as both are central requirements for many

games. The abstraction to a general purpose usually comes at the drawback of being less efficient for the specialized tasks in science. However, with recent advancement higher-level engines have found usage in a wide range of industries and are becoming a reasonable basis in more and more applications other than games. The most widely used higher-level game engines, which also are available to the public, are Unreal Engine [Unrc] and Unity [Uni22].

With the games' industry being a driving factor for Augmented Reality (AR), Virtual Reality (VR) and Extended Reality (XR), games engines are becoming the foundation for XR-based visualization. Hilfert and König [HK16] presented a low-cost virtual environment using Unreal Engine. Medeiros et al. [Med+22] presented an application for VR-based tactical resource planning using Unity.

Advancements in visual quality and functionality made using higher-level engines a practical solution in the movie industry. Sony Pictures Imageworks [Unra] used Unreal Engine to shorten the prodcution time of their series Love, Death & Robots, by reducing the delay between movie shot and seeing high-fidelity visuals. The Walt Disney Company [Bri19] used Unity for the production of the movie The Lion King, by utilizing its multiplayer VR functionalities to bring director and staff into the scene to help with framing.

Karis et al. [KSW21] presented Unreal Engine 5's new virtual geometry system Nanite [Nan] which achieves state-of-the-art high-fidelity real-time rendering for rigit geometry. They achieve this through fine granularity streaming on the level of mesh simplification groups, called cluster groups. All LODs are structures as HLOD. Each cluster group is as small as the smallest grouped result during mesh simplification. This way a streamed in group can be as small as a dozen triangles. Nanite uses a disk compressed format for storage and a memory and bandwith optimized layout for rendering. Combined with a wide range of other optimizations ranging from GPU-based culling, sparse updates, deferred materials and material culling. Their streaming approach allows for a fine granularity on the size of cluster groups.

Yao et al. [Del+21] presented a rendering algorithm for large-scale terrain in Unity using concurrent binary trees. Concurrent Binary Trees were presented by Jonathan Dupuy [Dup20] as a datastructure for parallel subdivision. It supports parallel refinement and construction of the bounding volume hierarchy. By using integer bitfields the sum-reduction tree is stored compactly and recursively at each higher level of the hierarchy.

Accessiblity to functionality is important in open science, where the goal is to make it as easy as possible to collaborate across fields. Ruzinoor et al. [Mat+14] reviewed the use of game engines for 3D terrain visualization of GIS data and concluded that they have a vast potential for customized applications to make GIS data available in a wide range of fields. Zarco et al. [Zar+21] compared 24 game engines and evaluated both Unreal Engine and Unity for robotics visualization. They conclude that the most evident benefit is the high versatility of both enginges for creation of visualization interfaces, but that further studies regarding the physical simulation accurateness of the underlying physics engines are required, as they have not examined it yet. Nesbit et al. [Nes+20] evaluated

the use of higher-level engines for visualization and sharing of high-resolution 3D data sets and 3D digital outcrop models. They also presented a case study for three different approaches to develop and share such visualizations with frameworks such as Sketchfab, Potree and Unity to verify their usefulness in promoting open science.

The underlying physics engine is often a central component in higher-level engines. As stated in Section 2.3.2, this work relies on Havok for Unity. Hummel et al. [Hum+12] evaluated Havok in 2012 against other open source physics engines and concluded that it was optimized for speed, at the expense of accuracy in some of their benchmarks. This observation was confirmed by Erez et al. [ETT15].

CHAPTER 3

Rendering

This chapter covers the rendering techniques used to implement large-scale terrain streaming for Visionary. It describes the scene structure and optimizations used.

3.1 Overview

Visionary streams large-scale terrain scene stored in Renderable Blob file format.

The source scenes used for this work are in OPC format and are structured in one to many *hierarchical levels of detail (HLOD)*. Erikson and Manocha [EM98] proposed the use of HLOD to perform simplification culling. Their initial work and later improvements by Erikson et al. [EMBI01] focused on the automatic generation of HLODs from sets of geometry with LODs. As the OPC scenes already contain preprocessed HLODs, the generation of HLODs is a prerequisite. In contrast to Erikson et al.'s work, the scenes are structure into multiple HLODs, instead of a single one. This is due to the partitioning into multiple OPC hierarchies for large scenes as presented by Ortner et al. [Ort+11]. Guthe and Klein [GK04] presented the application of HLODs for an out-of-core and network-based visualization framework to render huge polygonal models. The presented streaming algorithm is based on their asynchronous streaming of HLODs over multiple frames, but a different metric for (H)LOD selection and a different parallelization scheme is used.

3.2 Streaming

Streaming is an out-of-core processing technique. Out-of-core processing is required when the data is larger than the available memory. By swapping data from slower storage with data in-memory, an otherwise too large dataset can be processed in smaller than memory-sized chunks. Applied to rendering, this means nodes that contribute little to the

rendered image will be streamed out, while nodes that contribute more will be streamed in. This way, all elements rendered at a time fit into memory, while the dataset as a whole might be substantially larger. A streaming algorithm is presented that supports large-scale multi-layered terrains, multiple hierarchical levels-of-detail (M-HLOD) and measurements on streamed terrain surfaces. *Axis-Aligned Bounding Boxes (AABB)* are used as bounding volumes and the distance between the AABB of the root element of each HLOD and the viewer is used as metric to decide which HLOD should be streamed in, and which should be streamed out. A configurable limit is used to define how many HLODs are kept in memory at once. Only HLODs that are currently streamed-in are processed in later stages.

In this thesis, the rendering of large-scale terrains presented in Pro3d [Prob] is extended by streaming across multiple HLODs (M-HLOD). A similar out-of-core solution for fast display of massive geometric environments was presented by Varadhan and Manocha [VM02]. Their concepts of calculating a rendering front, which is a cut through the scene graph, with integrated simplification culling and view-frustum culling are applied to M-HLODs for efficient independent, asynchronous and parallel streaming. The implementation details for the used binary and streaming optimized Renderable Blobs format are discussed in Chapter 5. Results and a performance comparison to PRo3D are shown in Chapter 6.

3.2.1 LOD

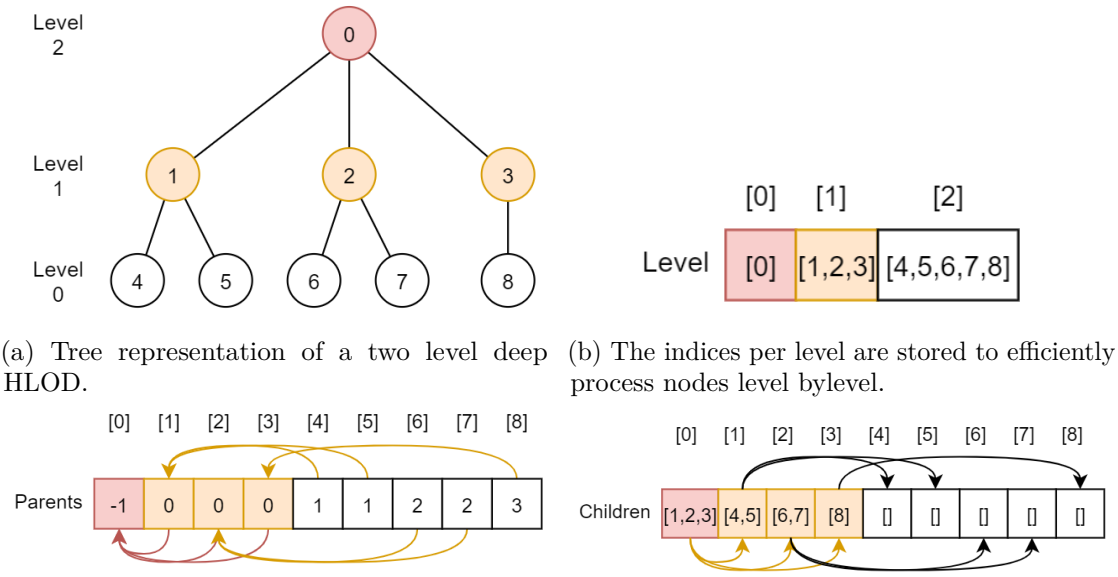
Terrains of interest are often larger than the available memory. Splitting the data into multiple tiles to process only a few at a time is one option. However, in visualization it is often desired to see all tiles at once, and displaying lower-resolution versions instead is a solution to achieve this [Sch16]. In this thesis, both approaches are combined, with tiled streaming across M-HLODs and LOD selection for currently active HLODs.

3.2.2 HLOD

Hierarchical scene structures to optimize visible surface detection were presented by Clark [Cla76] as early as in 1976. Erikson et al. [EMBI01] presented an algorithm for dynamic construction of HLODs to improve rendering performance for large environments. In contrast to their dynamic HLOD construction, this work preprocesses the raw terrain data into a GPU-optimized format and relies on the hierarchy from the input files. A similar method of preprocessing as optimization for GPU-based terrain rendering was presented by Zhai et al. [Zha+16]. They utilize quadtrees and their implicit parent-child relationship via indices to optimize for mesh simplification and GPU-based parallelized tesselation for DEM-based terrain rendering. The generic tree hierarchies presented in this thesis are designed for multi-resolution hierarchies of generic triangular meshes and DOM-, instead of only DEM-based terrains. This allows to model arbitrary mesh hierarchies. In contrast to their logical tree structure, indices are stored instead of pointers. Pointers to arrays containing the child indices are used, as the number of child nodes in a generic tree hierarchy is unbounded, where it is always exactly four in

a quadtree. The presented structure use indices over pointers to densely pack them in CPU-cache-friendly layout. The requirement to store pointers for the array of children is not cache friendly during the total HLOD tree traversal. However, when the children of a node or level are fetched, iteration is cache-optimized again. The presented streaming algorithm uses this by processing the tree level by level.

Node index i allows efficient lookup of all related data via aligned arrays. The data of any node i is stored within an array of the appropriate data type at index i within the hierarchy. Index 0 refers to the root node, with -1 indicating no further parent index. As in the original description of the HLOD structure by Erikson et al. [EMBI01], the finest HLOD is labeled *HLOD* 0 and the coarsest is *HLOD* N. In contrast to their work, no dynamic refinement or HLOD creation from LOD trees is performed, as discrete (H)LODs as decribed by Luebke et al. [Lue+02] are used. Ortner et al. [Ort+10] presented how the LOD levels in OPCs are spatially enclosing their child levels, as higher levels are created bottom-up by simplification of multiple lower level nodes. The streaming and LOD selection algorithms in this thesis utilize this property. Figure 3.1 shows an example of the used HLOD hierarchy in tree and array representation.



(a) Tree representation of a two level deep HLOD.

(b) The indices per level are stored to efficiently process nodes level bylevel.

(c) The parent indices are stored in a cache friendly and densely packed array.

(d) Arrays of child indices are stored to efficiently iterate over the children of each node.

Figure 3.1: Example of a HLOD 2 in tree and array representation. The densely packed indices allow CPU cache optimized traversal of the HLOD tree. The white HLOD 0 nodes contain the finest LOD. The orange HLOD 1 are one level coarser. The coarsest root node is highlighted in red.

3.2.3 Multi-HLOD

Both Erikson et al.'s [EMBI01] original HLOD structure and Guthe and Klein's [GK04] extension to streaming use a single HLOD structure per scene graph. They focus on merging geometry on higher HLOD nodes for efficient LOD algorithms. This work refers to M-HLOD as a set of HLODs. M-HLODs are used based on the scene partioning of OPC as described by Ortner et al. [Ort+10]. This is beneficial for parallel streaming, as each HLOD can be processed independently. With multiple root nodes, a valid rendering front, which must be a complete cut through the scene graph, can be guaranteed, without evaluating other children of the global root. All operations performed on the HLOD are then performed in parallel and independently across all HLODs. The only dependence is through shared resources, such as memory. M-HLOD is in particular useful for multi-resolution data, where lower-resolution HLOD data of the same terrain surface is enriched with higher-resolution HLOD from higher-fidelity data sources. Scenes are partioned both spatially and raster-based, but also across data quality layers. The rendering performance is increased by unloading not required HLODs altogether. This is equivalent to splitting a single HLOD into multiple HLODs, and allowing some subtrees this way to be streamed out.

3.2.4 LOD Streaming

Luebke et al. [Lue+02] give an overview of different LOD selection methods and emphasize the importance of the choice of position for distance-based LOD selection. The algorithm presented in this thesis uses the Euclidean distance between the position of the viewer (camera) and the closest point on the bounding volum. AABBs are used as bounding volumes due to their highly efficient representation and simplicity. Each AABB can be represented by only 24 bytes in single-precision floating-point format. The alignment with cardinal axes also allows for efficient geometry-based tests against them, as most tests can be performed against the slabs forming the AABB.

The LOD metric is evaluated once per frame for all roots, and once after each completed streaming request for all other candidates. The LOD metric for the root node is used for both streaming selection and LOD selection. Reusing it allows evaluating it only once for both phases. The required memory is kept within fast physically available memory by streaming out far away HLODs altogether. The LOD selection algorithm tries to avoid paging altogether. For each streamed-in non-culled HLOD, at least the coarsest node, the root, is required.

To perform streaming selection for streamed-in and streamed-out hierarchies efficiently, their AABBs are kept in memory at all times. This is used as the basis for the measurement algorithm presented in Chapter 4 as well. The memory required for all AABBs is insignificantly small compared to the large-scale terrain scenes. Streaming in bounding volume hierarchies was accordingly not necessary. The spatial enclosing property of the root node's AABB is used to decide if an HLOD is to be streamed in or out. If the viewer is close to the AABB, it is considered close to the encapsulated terrain surfaces.

Every frame the streaming selection is performed. All HLODs are ordered by priority, and low-priority HLODs are streamed out. Not yet streamed-in HLODs are streamed in parallelly. Streaming HLODs out and freeing the occupied memory can be considered an almost instant operation. However, streaming HLODs in, allocating memory, and initializing objects is slow, as it involves in particular reading data from slower bulk storage and copying data across the memory hierarchy. As with Guthe and Klein's [GK04] solution, no geometry is loaded during traversal to collect currently rendered nodes. In consequence, streaming and rendering can be performed completly asynchronously and in parallel. Points of synchronization are required during object destruction, to free the memory, object creation and initialization. As with their approach, loading is allowed to span over multiple rendered frames. This is necessary to achieve interactive real-time rendering.

Guthe and Klein's [GK04] prefetching algorithm for HLOD streaming only considers levels one level higher or lower than the currently rendered node. This reduces visible artifacts caused by rapid LOD changes, as only an iterative change from one level to the next is allowed. Varadhan and Manocha [VM02] prefetch multiple levels of ascendants and descendants. As this work is focused much more on the performance of the streaming algorithm, changes between multiple levels are allowed as in their work. This reduces unecessarily streaming in LOD levels by skipping intermediate streaming operations. However, it is impossible to completly avoid loading unecessary nodes, as during fast movement of the viewer, any arbitrarily defined close neighbourhood around a node could be left. When the viewer leaves the neighbourhood before loading has finished, any node can become unecessary before being available. An extreme example is instantaneous movement (teleportation) of the viewer from the highest priority area to the lowest priority area. This breaks any temporal coherence between frames and makes caching and prefetching challenging. Allowing arbitrary jumps between LOD levels is undesirable in immersive real-time rendering scenarios, e.g., for movie production or video games, where seamless transitions between LODs are important. In non-immersive visualization scenarios, e.g., digital terrain surveying, smooth LODs are a desired, but not a critical requirement. The proposed algorithm could be restricted to a single level of refinement per iteration. Figure 3.2 shows how M-HLODs are processed independently, asynchronously and in parallel, with one rendering front calculated per streamed-in HLOD.

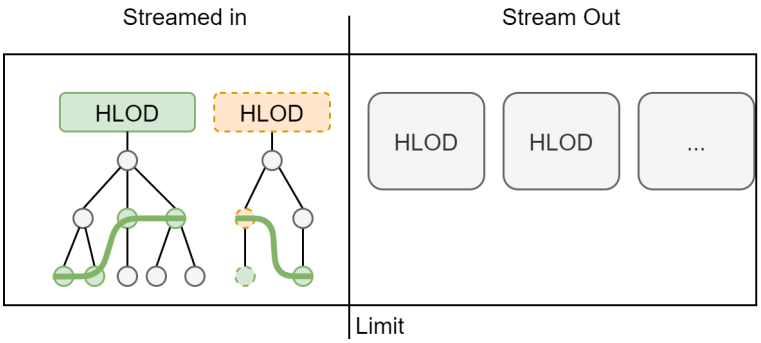


Figure 3.2: Streaming multiple HLODs allows for independent, asynchronous and parallel processing. All HLODs up to a priority limit are streamed in, and all others are streamed out. Green nodes are currently loaded. Orange nodes are currently loading.

Asynchronicity and Parallelization

Interactive applications require at least 20 frames per second to be considered interactive, which results in a time budget of at most 50 ms per frame. To put this duration into a context, an m.2 NVMe SSD with 2000 MB/s sequential read performance could read at most 100 MB per frame. At a 4k random access performance of 35 MB/s, this reduces to only 1.75 MB per frame. This shows how important an efficient memory layout on disk is. As processing and rendering of large-scale terrains also requires significant time, these ideal numbers are reduced further. Asynchronous processing across multiple frames is is for this reason a necessary requirement for streaming of large-scale terrains. This becomes even more drastic environments with a main-thread-model, such as Javascript, where most operations are restricted to a single main thread [Sch16].

Almost all modern systems have multiple cores. Parallelizing the workload across all cores and threads is consequently required to utilize the available hardware to its fullest capacity. Streaming and LOD hierarchies are both well suited for parallelization. Streaming of independent hierarchies can be performed in independent threads. Loading of independent LOD nodes can be performed in parallel the same way. The large number of processed elements in large-scale terrains is ideally suited for parallelization. GPUs also offer a high degree of parallelization. Yangzi [Don20] presented a multi-GPU framework for scalable real-time rendering for extremly complex 3D environments. Wald et al. [Wal+19] showed how hardware ray tracing can be utilized for tet-mesh point location in large data sets. Haidar et al. [Hai+18] showed how the high degree of parallelization in GPUs is beneficial for general high-performance computing (HPC).

The algorithm presented in this thesis is designed to work in environments with a single main thread and support for multithreading. In main-thread-based environments, creational and destructive operations are restricted to this single main thread, but multiple threads are available for other operations. Asynchronous multithreaded parallelization is accordingly possible as long as a context switch to the main thread is performed before continuing with any restricted operations. Long-running operations, potentially

much longer than four frames, such as file access, are offloaded to pooled threads. Managed pooled threads reduce resource exhaustion by spawning only a limited number of threads and distributing the work across them. A job system is used to parallelize computations across large numbers of similar computations. Jobs are short running operations restricted to complete within at most four frames. This restriction allows to optimize their memory allocations for speed over avoidance of memory fragmentaion. The rendering is performed in parallel on the GPU, but is initiated in the main thread, where completion is awaited blockingly. The streaming of LOD nodes for rendering is decoupled entirely from streaming the geometry used for the measurement algorithm presented in Chapter 4. This allows to perform the two most expensive types of work in parallel. Figure 3.3 visualizes the multithreaded design.

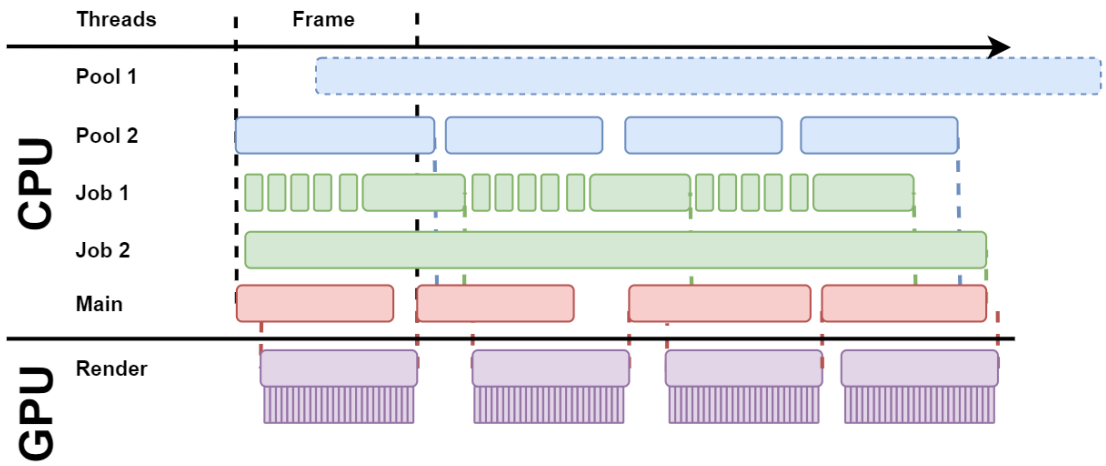


Figure 3.3: Work is distributed across pooled threads for long-running operations and a job system for short-running asynchronous operations. Completion of all parallelized work is observed on the single main thread, where rendering is initiated, and its completion is awaited. The work on pooled thread 1 has not yet completed. The work on job system thread 2 is forced to complete after four frames. Only two threads for each the job system and the thread pool are shown, but many more can be spawned.

3.2.5 LOD Selection

LOD selection is a problem since the introduction of LODs, and different algorithms have already been evaluted in 1998 by Reddy [Red98]. The proposed algorithm uses both a distance metric and LOD ranges for discrete LODs as presented by Luebke et al. [Lue+02]. The Euclidean distance between the closest point on the bounding volume, in this case the AABB, is used. This metric is used both for streaming selection and LOD selection. For streaming M-HLODs, only the distances to the root node of each HLOD is taken into account, and for LOD selection within a streamed-in HLODs the distances to all AABBs are considered. The distance metric is updated every frame, but only sparsely for nodes which are not already covered or culled by higher level.

Guthe and Klein [GK04] also use the closest point on the bounding volume of the current node for their view-dependent prefetching, but do not specify how they treat enclosement of the viewer. In particular, when single nodes of a large-scale terrains span across large distances, it becomes highly likely that the viewer is enclosed by the AABBs of an arbitrarily oriented terrain during a close-to-surface flyover, and thus a solution is required. In this thesis, the highest resolution data is requested in this case. This is a suitable solution if the viewer is enclosed by a small subset of bounding volumes at most. If the viewer is expected to be enclosed by all bounding volumes, a different solution for spatial acceleration structures is required, as culling would lose its efficiency.

The benefit of hierarchical spatial acceleration structures is that decisions can be made on fewer higher up nodes. When any ascendant can be culled, all of its descendants can be culled as well. A top-down tree traversal is used to decide level by level if a node is *sufficient* according to the LOD criteria, or is required to be *refined* by replacing it with its children. The culling information is propagated from top to bottom. No matter if a node is culled by view-frustum culling or culled because it is covered by any ascendant, it is removed from the rendering front and streamed out. A node is considered sufficient if:

- the distance between the viewer and its AABB is larger than the LOD distance d_{LOD} of its LOD level,
- there is insufficent free memory left to replace the node with its children,
- or it is a leaf.

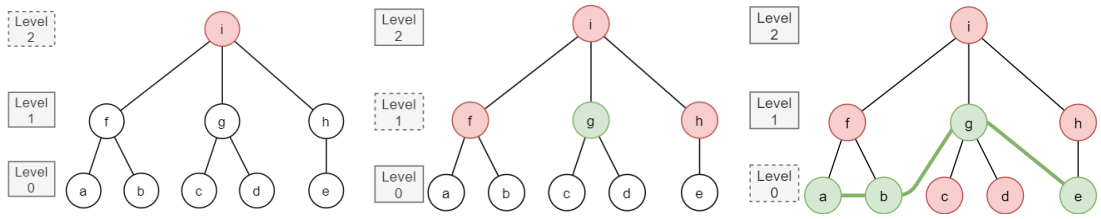
A node is culled if:

- its hierarchy is not streamed in (streaming selection),
- it is not within the view-frustum (view-frustum culling),
- or any ascendant is culled (simplification culling [VM02]).

In immediate rendering of scene graph based scenes, tree traversal can be terminated once a subtree is covered or culled. Varadhan and Manocha [VM02] used the top-down traversal method for calculating the rendering front for HLODs as presented by Erikson [EM98]. This is the recommended traversal order to best utilize the hierarchical properties of HLODs. However, in stateful rendering pipelines and streaming scenarios, the full tree needs to be traversed to deactivate or stream out nodes that were visible during the previous frame. Guthe and Klein [GK04] split traversal from loading and unloading geometry to support asynchronous LOD selection over multiple frames. This separation also is used in this thesis, to achieve asynchronous, independent and parallel LOD selection and streaming.

The result of LOD selection is a rendering front. A rendering front is a complete cut through the LOD tree. A cut is complete when any non-culled subtree is covered by exactly one node.

The proposed algorithm calculates the rendering front by descending level by level. This allows to parallelize across all nodes of all LOD trees of the same level. It is not only parallelizable per HLOD, but across all HLODs of the same level. At each evaluated node, all LOD criteria are evaluated. First it is verified if the subtree is already culled or covered. If a leaf is reached, it is sufficient. Otherwise, the LOD critera are evaluated for this node. Once all nodes of a level are evaluated, the LOD selection continues with the next level. Figure 3.4 visualizes a potential LOD selection traversal and the resulting rendering front.



(a) *Level 0.* The root node i is not selected, e.g., because the camera is too close.

(b) *Level 1.* Node g is selected as sufficient, e.g., because the camera is far away. Nodes f and h are not selected, e.g, because the camera is too close.

(c) *Level 2.* The leaves a, b, e are selected as mandatory. The nodes c and d are already covered by g.

Figure 3.4: Example of the resulting render front (green line) calculated by the presented algorithm. The level processed is shown dashed. Nodes are processed level by level. Nodes selected are shown in green. Nodes not selected are shown in red. White notes have not been processed yet.

LOD Range

LOD ranges are used to select the distance d_{LOD} at which a level is considered sufficient over their finer children. The range distance is increased exponentially, with a small constant base $b > 1$. In this thesis, $b = 1.25$ was chosen. A node at depth d is considered sufficient if it is further away than the LOD range r_d. A hysteresis of $h = \pm d_{LOD} * 10\%$ was chosen. As $d_{LOD} = 0$ is valid, and equivalent to restricting the range to the AABB, a minimum hystereis $h_{min} = 1$ is enforced. The 10% were selected based on experimentation and as suggested by Luebke [Lue+02] according to Astheimer and Pöche[AP94]. The distance per range r_d is calculated as follows:

$$r_b = b^d * d_{LOD} \tag{3.1}$$
$$h = \max(h_{min}, r_b * 10\%) \tag{3.2}$$
$$r_d = r_b \pm h \tag{3.3}$$

Reactive Targe Frame-Rate Scheduler

To maintain a constant and steady frame rate, a fixed frame-rate scheduler can be used [Lue+02]. A fixed frame-rate scheduler increases or decreases the LOD to achieve a fixed-frame rate. In this algorithm, a target frame-rate range was used. It adjusts the LOD distance d_{LOD} to stay within a target range of render time. The scheduler is reactive, as its decisions are based on the average render time t_{avg} over the last N seconds, in contrast to a predictive scheduler, which tries to predict how much work can be done per frame [Lue+02]. When rendering is fast, finer LOD levels are selected, and when rendering becomes slower, coarser LOD levels are selected. The scheduler decreases the LOD distance d_{LOD} when the average render time increases beyond the maxmimum targeted render time t_{max}. This causes coarser LOD nodes to be accepted at closer distances, and consequently reduces the amount of data to be rendered. Inversely, the scheduler increases the LOD distance d_{LOD} when the average render time decreases below the minimum target render time t_{min}. This forces finer LOD nodes to be selected at larger distance, and consequently increases visual fidelity.

A target frame-rate scheduler should react quickly, but a large change in LOD distance can cause resonance. A scheduler is resonating, when every change leads to another change. At the granularity of decisions the scheduler can make, it is possible that there is no decision where the system can stabilize. A finer level could cause too much work load, and the coarser workload could underutilize the system. The implemented scheduler for this reason tries to make as few decisions as possible by selecting a target render time t_{target} as far from t_{min} and t_{max} as possible. This corresponds to the mean value:

$$t_{target} = \frac{t_{max} - t_{min}}{2} \tag{3.4}$$

The scheduler calculates the reactive LOD distance d_{LOD_r} based on the average render time t_{avg}. It tries to adapt the workload to achieve the target render time t_{target}. As changing the LOD distance potentially causes large changes in LOD selection, and thus many new finer nodes to be streamed in, the frame rate can drop quickly due to decisions by the scheduler. A spike in render time can be caused by external factors on the system on which it is running. For this reason, drastic changes are avoided, and multiple finer granular ones preferred. This gives the system sufficient time to recover from large changes, reduces resonance, and stabilizes the frame rate. The following methods are used to dampen reactions of the scheduler:

- The average render time t_{avg} over the last N seconds is used to avoid reacting on single slow frames after a change.
- Decreasing d_{LOD} to the minimum is allowed to happen during a single frame, to quickly achieve interactive frame rates if they become low. It is much better to keep interactivity while increasing fidelity, than having high fidelity, but loosing responsiveness.
- Increasing d_{LOD} is limited to avoid loading more data than the system can handle at once.
- A 10% hysteresis t_h is used to reduce flickering around critical distances.

The ratio between t_{avg} and t_{target} is used to scale the change of d_{LOD}. The LOD distance target range d_{range} is the difference between the maximum LOD distance $d_{LOD_{max}}$ and the minimum LOD distance $d_{LOD_{min}}$.

When t_{avg} is greater than $t_{target} \pm t_h$, the LOD distance d_{LOD} is reduced as follows:

$$t_r = \frac{t_{avg}}{t_{target}} \tag{3.5}$$

$$d_{LOD} = d_{LOD}/t_r \tag{3.6}$$

When t_{avg} is smaller than $t_{target} \pm t_h$ the LOD distance d_{LOD} is increased as follows:

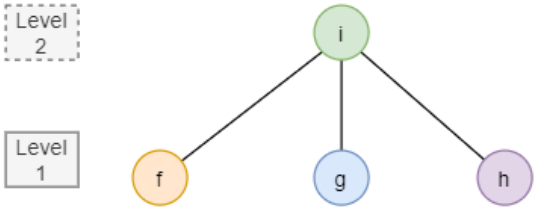
$$d_\delta = d_{range} * 0.1 * t_r \tag{3.7}$$

$$d_{LOD} = d_{LOD} + d_\delta \tag{3.8}$$

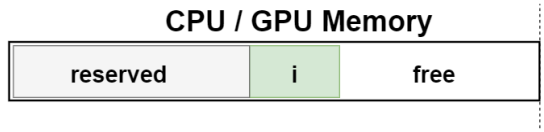
Memory Limitation

When large-scale datasets are handled in real-time rendering, available resources need to be considered. Operating systems are able to provide additional memory via paging up to a certain degree, but access to paged memory residing on disk storage is significantly slower. Historically, paging was only available for CPU memory, but nowadays also is available for GPU memory [lorb]. To avoid paging altogether, memory allocations must be managed carefully. Memory management is a highly complex task, as many factors other than the physical size of the data are relevant. Memory fragmentation and ephemeral copies can significantly increase the actually required memory. Steinberger et al. [Ste+14] proposed a custom allocator for block memory management to efficently generate and render infinite cities on-the-fly. The presented algorithm works on a higher level, leaving allocations to the underlying frameworks and subsystems. However, it tracks memory reservations for LOD nodes, to ensure sufficient memory is available to

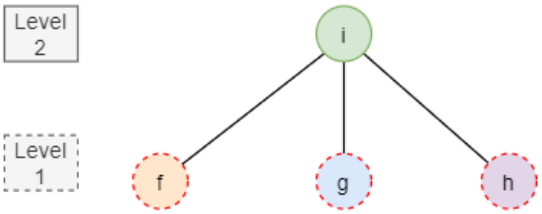
avoid paging. Refinement of a node is restricted such that a refinement is only allowed if sufficient memory for all children can be reserved. It is not a necessary requirement that the children of a node require more memory than their parent, but it can be assumed for the purpose of LOD due to the simplification. Figure 3.5 visualizes an example where a node cannot be further refined because its children would require more memory than available.



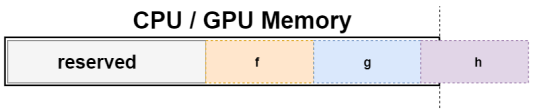
(a) Node i is currently loaded and tested for refinement with descendants f, g and h.



(b) Node i (green) is currently loaded and its memory reserved. Free memory is available.



(c) There is insufficient memory to replace node i with its descendants f, g and g.



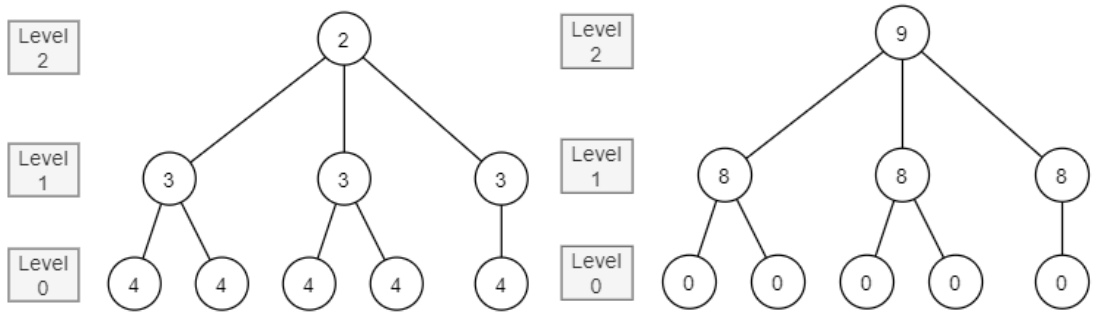
(d) Replacing node i with its children f, g and h would require more memory than available.

Figure 3.5: Example of a node i that cannot be replaced by its children f, g and h, as their combined memory exceeds the available memory.

The memory required per node and its children is static and for this reason precalculated and stored in its hierarchy. The required memory of a node i is m_i. The required memory of its children is $m_{c(i)}$. With current reserved memory m_r and a memory limit of m_{max}, the LOD decision m_{LOD} of the memory scheduler can be calculated as follows:

$$m_{LOD} = \begin{cases} 0, & \text{if } m_r - m_i + m_{c(i)} <= m_{max} \\ 1, & \text{otherwise} \end{cases} \tag{3.9}$$

When $m_{LOD} = 1$ a node is considered sufficient, and no further refinement is allowed. Figure 3.6 shows how both the memory requirement $m_i$for the node, and the memory requirement $m_{c(i)}$ for its children are stored in the hierarchy.



(a) The hierarchy stores the memory required per node.

(b) The summed up memory of the children are stored per node.

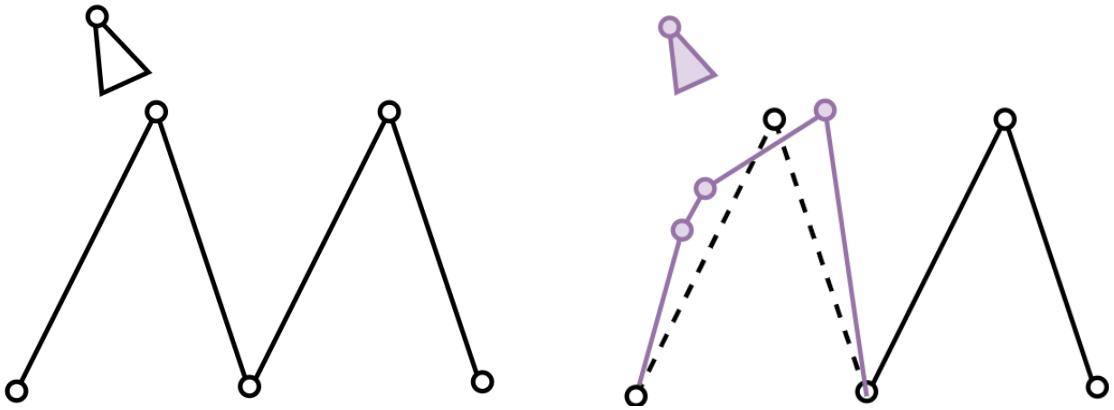
Figure 3.6: Example of how the hierarchy stores the required memory of each node and the sum of required memory of each node's children. The values shown in each node are example memory units, e.g., megabytes.

3.3 Priority Rendering

This work refers to multi-layered terrain, when multiple layers of data are available for the same terrain and meant to be visualized within the same context. Examples for this are data layers from different data sources and reconstructions. While LODs are a form of multiple quality layers, they are not shown at the same time, and are consequently not referred to as multi-layered data.

Visionary supports multi-layered terrain via priority rendering. Each data layer is assigned a priority, and higher priorities are rendered above lower priorities. Paar et al. [Paa+15b] showed this technique for data fusion in their viewer of the PRoViDe (Planetary Robotics Vision Data Exploitation) project. The visualization software PRo3D by Traxler and Ortner [TO15] is the component of the *FP7-PRoViDE* tool set where they have implemented this method for priority rendering. They have used this method to fusion the data between orbiter and rover image products for visualization of Martian terrain.

Data of higher-priority layers is considered of higher quality for all cases. However, visualizing both at the same time can help to understand the context better. An example is the visualization of a high-fidelity outcrop rendered at its positions within a much larger and lower-fidelity terrain. Figure 3.7 visualizes how a higher-quality layer can enrich the quality of the overall data, even if it is only available for a subsection of the terrain.



(a) Example of a simplified terrain with low fidelity and assigned to priority layer 0. In the first render pass, only priority 0 data is rendered from the perspective of the viewer in the top left.

(b) In a second render pass, the data of priority layer 1 (purple) is rendered. It occludes even the mountain peak that would protrude through the priority 1 surface without priority rendering (dashed).

Figure 3.7: Example of priority rendering. The higher-quality data (purple) improves the quality of the terrain surfaces, altough it is only available for the left of the two mountain peaks shown.

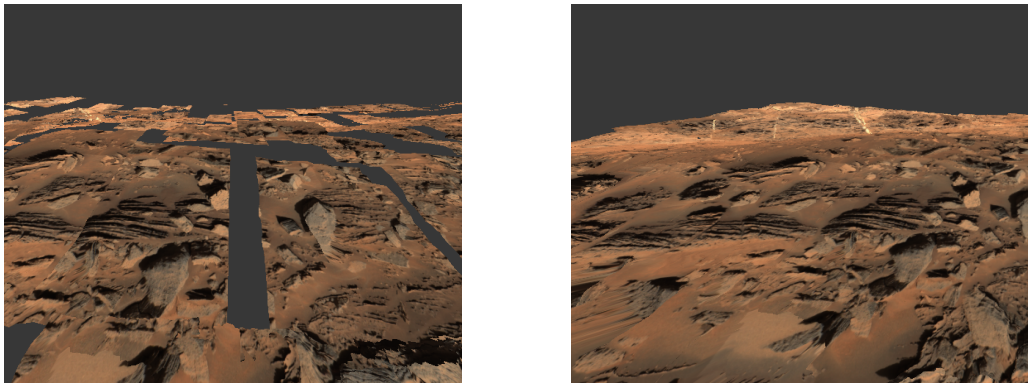
Priority rendering is achieved trivially in Visionary by using multiple render passes. As each priority layer consists of its own M-HLOD, all streaming and LOD selection can be performed just as if only a single priority layer would exist. The only additional requirement is one additional render pass per priority layer. After each render pass, the Z-Buffer is cleared to ensure correct depth testing within each layer, but overdrawing all lower-priority layers. The measurement algorithm presented in Chapter 4 correctly respects the priority of all layers. The resulting images are shown in Chapter 6. The engine-specific implementation for priority rendering via camera stacking is explained in Chapter 5.

3.4 Origin Shift

The coordinates of large-scale terrains can exceed the precision supported by single-precision floating-point values. While graphics cards have been supporting double-precision for a long time [NA11], single-precision floating-point formats have, for performance reasons, remained the de facto standard in real-time rendering. Floating point numbers are usually represented according to the IEEE 754 standard [Iee]. This means that single-precision floating point formats use 32 bits and double-precision floating point formats use 64 bits. Even if a CPU or GPU instruction requires the exact same amount of cycles for both types, the double-precision format will still require twice the memory. This means that only half as many values fit into each cache line and vectorized code has to perform twice as many iterations. In performance-critical applications, such

as streaming large-scale terrain, it is for this reason desired to avoid double-precision operations for any type of large data.

The use of single-precision floating-point formats can cause visual errors, as only 6-9 significant digits are supported. As floating-point formats are most precise around 0, this error becomes larger the further the values are from 0. Spatial jittering is the result. Spatial jittering is a kind of error where the representable positions closest to each other drift visibly apart. Figure 3.8 shows how significant the precision errors of single-precision floating point formats for large coordinate data can become.



(a) Spatial jitter can cause LODs to drift apart in single-precision.

(b) Using double-precision can close the visible gaps.

Figure 3.8: Example of visible spatial-jitter due to insufficient precision by using single-precision floating point formats for large-scale coordinates. The stretched distortion in both images is a result of the stereoscopic reconstruction and not caused by the precision.

A coordinate system transfer is used to render geometry with large-scale coordinates. This allows to render geometry from large-scale coordinate systems while still using the faster single-precision floating point formats. A new origin is selected close to the rendered scene, hence the method is called origin shift. All coordinates are then transformed in relation to this origin, which causes large position values in the spatial transformation matrix to cancel each other out. This is performed in double-precision. An example of this effect on the local-to-world transformation matrix $\mathbf{M}$ will be discussed below. The translation component of the transformation matrix is chosen as origin $\vec{o}$. The shifted transformation matrix with respect to origin $\vec{o}$ is $\mathbf{M_o}$:

$$\mathbf{M} = \begin{bmatrix} \dots & \dots & \dots & x = -2480530.654326382 \\ \dots & \dots & \dots & y = 2301535.936282250 \\ \dots & \dots & \dots & z = -250192.728987073 \\ \dots & \dots & \dots & \dots \end{bmatrix} \tag{3.10}$$

$$\overrightarrow{o} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.11}$$

$$\mathbf{M_o} = \mathbf{M} - \begin{bmatrix} 0 & 0 & 0 & x \\ 0 & 0 & 0 & y \\ 0 & 0 & 0 & z \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \dots & \dots & \dots & x - x \\ \dots & \dots & \dots & y - y \\ \dots & \dots & \dots & z - z \\ \dots & \dots & \dots & \dots \end{bmatrix} = \begin{bmatrix} \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots \end{bmatrix} \tag{3.12}$$

This way the position components x, y and z are reduced to smaller values closer to $(0, 0, 0)$ and can be more precisely represented by single-precision formats. While there is no way to overcome the limitations of lower precision altogether, the goal is to reduce the errors to insignificant levels. Thorne [Tho05] presented a floating origin algorithm to reduce spatial jitter due to precision for large distributed virtual words. By placing the viewer at $(0, 0, 0)$ and moving the scene in relation to this position, the same effect is achieved. In this thesis, only a single origin shift is performed, as no scene was sufficiently large to require updating the origin per frame. By running the origin shift once per frame, with the viewer shifted to $(0, 0, 0)$, the algorithm could be extended to a floating origin model. The origin shift requires the resulting values of the shown measurement algorithm in Chapter 4 to be shifted inversely by $+\vec{o}$. This also needs to be done in double precision.

3.5 Problems and Considerations

In this section, problems, artifacts and considerations about the presented streaming algorithm are discussed.

3.5.1 LOD Selection

The LOD distance metric is cheap to evaluate, but view-dependent or screen-error metrics could improve the quality of the final image. Continuous LOD could further help to smooth LOD transitions. As memory tracking is not trivial, the memory tracking heuristic strongly relies on assumptions of the in-memory size based on the file size. To avoid resonance of the reactive target frame-rate scheduler, magic numbers as dampening factors were used, for the number of seconds to smooth the average frame rate and the weighting of how much to increase the LOD range per refinement.

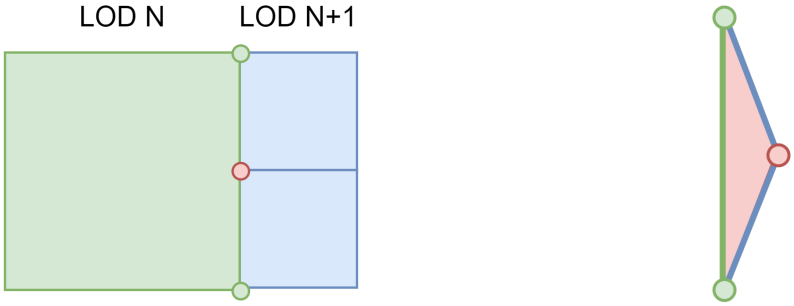
Naive asynchronous streaming can cause popping artifacts when any ascendant is unloaded before all required descendants are loaded, or vice versa. To avoid popping, the unloading

of any node is for this reason delayed until its replacements are fully streamed in. This can still cause popping artifacts due to large geometric or texture changes between two LOD levels, or Z-fighting while elements of two LOD levels are loaded simultaneously. Giegl and Wimmer [GW07] proposed a combination of opaque drawing and blending between two discrete LODs to achieve a smooth transition. Scherzer and Wimmer [SW08] proposed the interpolation of sequential frames as solution for discrete level-of-detail rendering. Schütz et al. [SKW19] presented a method for real-time continuous level of detail for rendering point clouds.

The shown algorithm does not take any measures to guarantee smooth LOD transitions. When arbitrarily large changes in geometry and texture are allowed, blending or interpolation are not sufficent anymore and the additional rendering cost is thus not justified.

3.5.2 Cracks

Different tesselation levels between LODs can cause holes, called *cracks*. Cracks are caused by the resulting *t-vertices* at the border of different LODs. In LOD-based terrain rendering, they occur between two neighbouring tiles of different LOD. Figure 3.9 shows how the different LOD levels can cause cracks.



(a) A t-vertex (red) occurs at the border of two different tesselation levels caused by naively doubling the tesselation level.

(b) The t-vertex (red circle) causes a crack (red triangle) in the terrain.

Figure 3.9: Example of a crack (red triangle) cause by different tesselation level at a t-vertex (red). (a) shows the the different tesselation levels of neighbouring nodes, but the crack is not visible. (b) shows the crack as red triangle.

One solution for this problem is to limit the tesselation level between two neighbouring tiles along the border to the lower tesselation level. This has the drawback of reducing the fidelity of the model along the border. Xingquan et al.[CLS08] solved the problem by filling the crack with additional triangles. Husain et al. [Hus+] avoid cracks during the subdivison of their tesselation algorithm. Lee and Shin [LS19] presented a GPU-based algorithm for real-time landscape visualization that avoids cracks by splitting the edge of the coarser level recursively.

Tesselation and cracks are not handled by Visionary, as the geometry is rendered as provided by the source files.

3.5.3 Engine-Specific Optimizations

To achieve real-time streaming of large-scale terrains with the presented streaming algorithm, several optimizations were required to bypass restrictions imposed by the engine. The input scenes were preprocessed offline into CPU- and GPU-optimized memory layouts. Software caching was used to reduce stalls during garbage collection and to reduce the number of slow creational operations in the engine. File streams were kept open and cached to avoid slowdowns due to slow file system access. Custom serializers were implemented to extend the serialization functionality where the API provided by the engine was insufficent. Thread context management was used to achieve multi-threading in a main-thread model.

In Section 6.2, the engine specific restrictions are discussed, and the implementations used to bypass them are presented.

CHAPTER 4

Measurements

In this chapter, a two-phase out-of-core measurement algorithm for digital surveying of streamed large-scale terrain is presented. It is explained how *Shared Edge Detection (SED)* is used to terminate subsampling earlier by analytically finding exact midpoints between neighbouring primtives and how the *On-Data Ratio (ODR)* is used to quantifiy uncertainty of a measurement, and raise awareness about this uncertainty.

4.1 Overview

Digital terrain surveying allows to measure the geomorphometric quantities of terrain surfaces. One application is the interpretation of digital outcrops for terrestral and planetary geology. User-drawn polylines are used as intuitive digital measuring tape to understand the extents and orientation of surface features. Such polyline-based DOM interpretation tools are implemented, e.g., in VRGS 3.0 [Vrg] and PRo3D [Prob]. The polylines are projected onto the terrain surface to evaluated feature points. An example for an elementary measurement using this method is measuring the extents of a crater or an outcrop. Different projection directions and subsampling strategies are used to analyze finer surface details, e.g., walking distances or length of surface lines.

The presented measurement algorithm uses ray casting against mesh-based terrain for the projection. Ray casting against mesh-based geometry is a well researched and understood method with highly optimized algorithms available. Changing projection directions is done by changing the direction of cast rays and adding more subsamples is achieved by adding additional rays. Further subsamples increase the accuracy of the surface reconstruction of the sample terrain profile as shown in Figure 4.1.

(a) 2 feature points. (b) 1 subsample. (c) 3 subsamples. (d) 4 subsamples.

Figure 4.1: Subsampling for terrain profile reconstruction. (a) shows the user-selected feature points. The orange dashed reconstruction does not reflect the terrain surfaces, as both mountains are missing entirely. (b) shows one added blue subsample in the middle of the polyline segment. The reconstructed surface profile improves only insignificantly. (c) shows two further added blue subsamples. The profile now properly models the two mountains, but no segment is fully accurate yet. (d) shows how one additional added blue subsample leads to an excact reconstruction of the green planar segment.

The difficulties for surveying streamed large-scale terrain are that not all geometry can be kept in memory at the same time and that surveying requires interative exploration of terrains. Performing a measurement on a simplified geometry does not allow for accurate measurements, and for this reason LODs are not useful for the measurement. This means that geometry of the highest-resolution nodes needs to be streamed in for a measurement, even when a lower-resolution node is rendered. This increases the total memory required. The presented algorithm selects potential ray cast hit candidates in the broad phase by raycasting against their AABB. Only required geometry is then streamed in for exact evaluation in the narrow phase.

The difficulty with subsampling is that while more subsamples are desired, in an out-of-core algorithm they take significantly more time, as the geometry must be streamed in. Evaluating a large-scale terrain precisely is consequently challenging, and more so, as the duration required is unpredictible. If all subsamples hit the in-memory geometry, it is significantly faster than if large amounts of geometry need to be streamed in first. This means that only a certain number of rays can be cast per frame, and the choice of subsampling is important for the quality and performance of the algorithm. PRo3D [Prob] relies on fixed-rate subsampling, where a fixed number of equidistant subsamples are evaluated per polyline segment. An improved variable-rate subsampling strategy is presented that uses SED to terminate with exact results when neighbouring primitives are hit. A simple uncertainty metric ODR is presented to quantifiy the quality of a measurement.

4.2 Algorithm

The presented algorithm for ray-casting-based measurements differentiates between *feature points* and *subsamples*. Feature points are the points determined by the user to select a terrain feature of interest and are always projected in view direction. Subsamples are generated algorithmically and are projected in the direction $\vec{d}$ of choice. Directions of choice can be the view direction, to measure what is seen on the screen, or the

down direction, to measure, e.g., the walking distance between two feature points. The up-vector of a terrain is defined by its orientation and down thus means in opposite direction.

The input for each measurement are the start point $\overrightarrow{p_s}$ and the endpoint $\overrightarrow{p_e}$. All segments of a polyline are resolved identically, and so the algorithm is presented for a single segment. The broad phase is as simple as performing all raycasts against all AABBs to identify the potential geometry g to be streamed in. This can be done in parallel. As these raycasts are performed against bounding volumes, all hits, and not only the closest one, must be collected. Algorithm 4.1 lists the pseudocode for the broad phase.

Algorithm 4.1: Pseudocode for the broad phase

input : List of points $p[]$, direction d and all $AABB[]$

output : List of potential geometry $G[]$

```
1 G ← []
2 foreach point p in p[] do
3 |   r_p ←Ray (origin : p, direction : d)
4 |   g[] ←Raycast (AABB[], r_p)
5 |   G[] ← G[] ∪ g[]
```

During the narrow phase, multiple things have to be considered. Geometry that is not currently loaded must now be streamed in for the raycast. The priority layers described in Chapter 3 must be considered, such that hits on higher layers are prioritized as well. Finally, the geometry is usually stored in local space, and transforming all vertices for ray-primitive intersection tests is undesirable performance wise. Instead of transforming all vertices of the geometry by applying the affine local-to-world transformation matrix $\mathbf{M}$, the inverse transformation matrix $\mathbf{M}^{-1}$ is used to transform the ray from world to local space. Ray casting is then performed in local space. The subindex w is used to indicate a coordinate in world space, and the subindex l indicates a coordinate in local space. The origin of the ray $\overrightarrow{r_{ow}}$ is transformed as follows:

$$\overrightarrow{r_{ol}} = \mathbf{M}^{-1} * \overrightarrow{r_{ow}} \tag{4.1}$$

The rotation component of $\mathbf{M}^{-1}$ is extracted as quarternion $q_{M^{-1}}$. This allows to transform the direction of the ray r_{dw} as follows:

$$\overrightarrow{r_{dl}} = q_{M^{-1}} * \overrightarrow{r_{dw}} \tag{4.2}$$

The result for all ray cast hits are the closest hit point $\overrightarrow{h_l}$, the identifier of the hit geometry, and the vertices forming the hit primitive $\overrightarrow{V_l}$. The positions are then transformed from local space back to world space using the transformation matrix $\mathbf{M}$. This is much more efficient, as the number of hit primitives is many orders of magnitude smaller than the

number of vertices of the whole terrain. As this is a simple affine transformation, only the transformation for the hit point $\overrightarrow{h_l}$ is shown. All other positions are transformed to world space identically:

$$\overrightarrow{h_w} = \mathbf{M} * \overrightarrow{h_l} \tag{4.3}$$

The resulting hit points $h_w \in H$ across all geometry are unordered, but only the closest to the ray origin $\overrightarrow{r_{ow}}$ is of interest. A linear search for the closest result in world space is performed. During this step, the priority layers described in Chapter 3 must be considered. As all data on higher-priority layers is considered of better quality, so are hit points against higher-priority layers. A hit point is for this reason considered closer, regardless of the Euclidean distance, if it originates from a higher-priority layer. As the ray cast is performed in the local space of each potential hit target, the distance in world space between the hit $H[i]_w$ and the ray origin r_{ow} must be recalculated. The resulting fraction of the ray cast, which defines the distance along the ray's half line, from ray origin to hit point in local space, does not guarantee proper ordering after local-to-world transformation of the hit point. Algorithm 4.2 lists the pseudocode for the priority-layer-aware search of the closest hit point.

Algorithm 4.2: Pseudocode for finding the closest and highest priority hit

input : Ray origin r_{ow}, list of hit points $H_w[]$ and priorities $p[]$

output : The closest hit point of the highest priority layer h_{cw}

```
1 h_cw ← H_w[0], p_c ← p[0], d_c ←Distance(H_w[0], r_ow)
2 for i ← 1 to Length(H) - 1 do
3 |   h_w ← H_w[i], p ← p[i], d ←Distance(h_w, r_ow)
4 |   if p > p_c or d < d_c then
5 |   |   p_c = p,h_cw = h,d_c = d
```

During the narrow phase, the geometry of the finest LOD is streamed in. Then the raycast is performed in local space. The results are transformed back to world space. Out of all hits, the closest of the highest priority is selected. As rays with similar origin and direction are likely to hit the same geometry, the geometry is not released immediatly. Instead, the geometry is released through a software cache, to keep recently used geometry in-memory. This reduces how often the same geometry is streamed in per measurement. Although the size of the geometry is of the finest detail, it is smaller than the rendered mesh, as no textures are loaded. However, in large-terrain scenarios not all geometry can be kept in-memory at once, consequently it must be processed out-of-core. Algorithm 4.3 lists the per ray pseudocode for the narrow phase.

Algorithm 4.3: Pseudocode per ray for the narrow phase

input : List of potential geometry $G[]$, Ray r_w.

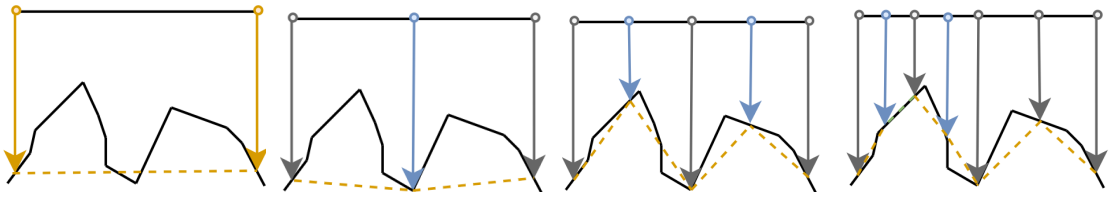
output : The closest hit point of the highest priority layer h_c

```
1  H ← []
2  for i ← 0 to Length(G) do
3      g ← G[i], M ← G[i].localToWorld
4      if not IsStreamed(g) then
5          StreamIn(g)
6      r_l ←WorldToLocal(M^-1, r_w)
7      h_l ←Raycast(g,r_l)
8      h_w ←LocalToWord(M, h_l)
9      H ← H ∪ h_w
10     Cache.Release(g)
11 h_c ← ClosestOfHighestPriority(H)
```

Depending on the selected subsampling strategy, all resulting projected polylines are then refined. During subsampling, the *Shared Edge Detection (SED)* algorithm, presented in this thesis, is used to find exact subsamples and terminate subsampling. The variable-rate subsampling algorithm is described in Section 4.3. Once all subsamples are calculated, they are classified as certain or uncertain and the uncertainty of the measurement is calculated as the *On-Data Ratio (ODR)* presented in this thesis. The ODR is explained in Section 4.5.

4.3 Subsampling

Subsampling is used to evaluate the terrain surface profile per projected polyline segment as described in Section 4.1. However, even with subsampling, many subsamples can be required to accurately reconstruct the terrain surface profile. A subsampling strategy that minimizes the amount of subsamples taken, while leading to accurate results, is consequently required. One strategy is fixed-rate subsampling, where a fixed number of subsamples per polyline segment are taken at equidistant positions. This is the subsampling strategy implemented in PRo3D [Prob]. In variable-rate subsampling additional rays are cast until any of the termination criteria is met. The proposed algorithm uses variable-rate subsampling to make the best use of the same number of rays. When neighbouring primitives are hit, SED will find exact subsamples and terminate earlier. Figure 4.2 visualizes how additional subsamples improve the reconstruction of the terrain surface.



(a) 2 feature points. (b) 1 subsample. (c) 3 subsamples. (d) 5 subsamples.

Figure 4.2: Example of how additional subsamples increase the accuracy of the resulting terrain surface profile. The resulting surface profile is shown as dashed orange polyline.

Not only the number of subsamples matters, but also how they are chosen. Figure 4.3 shows how even if the number of subsamples is always the same, the terrain reconstruction can vary significantly.

(a) 0 subsamples. (b) Subsampling right. (c) Subsampling left. (d) Exact result (SED).

Figure 4.3: Example of how different choices for subsamples result in different terrain reconstructions. The proposed SED algorithm results in exact midpoints (blue circles) whenever neighbouring primitives are hit. No matter where on the blue surface the subsample is evaluated, SED will lead to an exact reconstruction.

The proposed subsampling strategy uses the rays $\overrightarrow{r_i}$ and $\overrightarrow{r_{i+2}}$ to recursively calculate the subsampling ray $\overrightarrow{r_{i+1}}$. For each two rays, one additional ray is cast in-between. The average is sufficient, as all rays lie on the same ray casting plane. When subsamples are cast in down direction, all rays share the same direction and calculating the direction can be skipped. $\overrightarrow{r_o}$ denotes the origin and $\overrightarrow{r_d}$ the direction. The ray $\overrightarrow{r_{i+1}}$ is calculated as follows:

$$\overrightarrow{r_{o_{i+1}}} = \frac{\overrightarrow{r_{o_i}} + \overrightarrow{r_{o_{i+2}}}}{2} \tag{4.4}$$

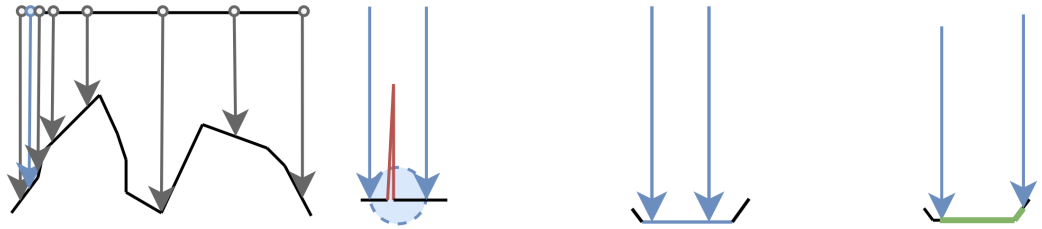
$$\overrightarrow{r_{d_{i+1}}} = \frac{\overrightarrow{r_{d_i}} + \overrightarrow{r_{d_{i+2}}}}{2} \tag{4.5}$$

A subsampling ray is cast between each two rays until one of the following termination criteria is met. To calculate the ODR, the resulting segment also is classified as certain or uncertain based on the termination criteria:

- Two primitives connected by a shared edge are hit (certain, SED).
- The maximum recursion depth is reached (uncertain).

- The origins or results are closer than ϵ (uncertain).

A distance limit of ϵ is used as termination criterion as there is no point to subsample below the precision limit. A maximum recursion depth is enforced as it is not guaranteed that any terrain could be hit. Errors during terrain reconstruction can cause holes within the terrain, and measurements beyond the extents of the terrain have no chance of success. Measuring outside the terrain might be considered an obvious error, but tiny holes in the terrain are not. Tiny holes at subpixel size might not even be visible. Figure 4.4 visualizes the different termination criteria.



(a) (b) (c) (d)

Figure 4.4: Different termination criteria. (a) Maximum recursion depth is reached. (b) ϵ threshold reached, the thin red structure was missed. (c) same primitive hit. (d) neighboring primitives hit (SED).

Algorithm 4.4 lists the pseudocode for the outer loop of the subsampling algorithm.

Algorithm 4.4: Pseudocode for outer loop of the subsampling algorithm using SED.

input : Ordered list of results $R_{in}[]$ and their rays $R_{in}[i].Ray$

output : Refined ordered list of total results $R[]$.

```
1 R ← []
2 R.Add(R_in[0])
3 for i ← 0 to Length(R_in)-1 do
4 |   left ← R_in[i]
5 |   right ← R_in[i + 1]
6 |   R.Add(Subsample(left, right, 0))
```

Algorithm 4.5 shows the recursive subfunction.

Algorithm 4.5: Pseudocode for the recursive subsampling subfunction.

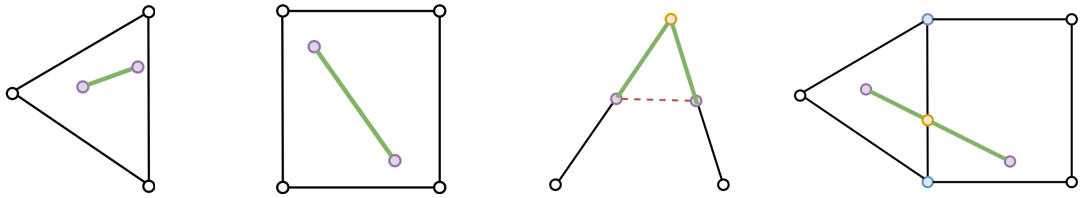
```
1  Function Subsample(left: Result, right: Result, depth: int)
2      if left.IsCertain or depth > max or Distance (left, right) < ε then
3          return
4      SED ← SED(left, right)
5      if SED.HasSharedEdge then
6          left.IsCertain ← true
7          return SED.Result
8      ray ← (left.Ray+right.Ray)/2
9      subsample ← Raycast(ray)
10     results ← Subsample (left, subsample, depth + 1)
11     results.Add(subsample)
12     results.Add(Subsample (subsample, right, depth + 1))
13     return results
```

4.4 Shared Edge Detection

Shared Edge Detection (SED) is a way to improve ray-casting-based measurements of mesh-based terrain. It relies on the vertex positions of hit primitives, but does not require any neighbourhood information. For any two consecutive raycast hits H_i and H_{i+1}, the edges of the hit planar primitives are checked for equality. Edges are considered equal when the positions of the vertices are equal.

When a shared edge is detected, both planar mesh primitives are tested for coplanarity. If both are coplanar, no further subsampling is required, as the direct line $\overrightarrow{H_iH_{i+1}}$ is an exact surface reconstruction for the subsegment. As any primitive is coplanar with itself, no further subsamples are required either. This holds true as long as only convex planar primitives are used. The two most common mesh primitives are triangles and quads, and they also were used for this work. Both primitive types, triangles and quads, are convex.

When the primitives are not planar, then the exact result can be found analytically by calculating the *midpoint* $\overrightarrow{M}$ on the shared edge. In this case, the exact subsegments are formed by $\overrightarrow{H_iM}$ and $\overrightarrow{MH_{i+1}}$. Figure 4.5 shows all SED cases.



(a) (b) (c) (d)

Figure 4.5: Comparison of SED cases. (a) shows two hits within the same triangle. (b) shows two hits within the same quad. (c) shows two hits on neighbouring, non-coplanar primitives. In this side view, the primitive types are not visible, but it is visible how SED improves the direct line result (dashed red line) by finding an exact midpoint (yellow). (d) Case (c) shown from top. SED does not require neighbours to be of the same type.

The shared edge is found by comparing the vertices of both primitives. The first found match is used as shared edge. When the primitives are formed by the same vertices, they are planar. They also are planar, when one primitive is a quad, and the other is a triangle formed by vertices that are a subset of the vertices forming the quad. This edge case can occure during measuring across the borders of multi-layered data. Figure 4.6 visualizes the edge case of a triangle hit of a higher-priority layer that is embedded in the quad of a lower-priority layer.

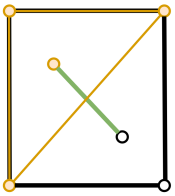


Figure 4.6: Embedded triangle edge case. The orange hit point belongs to the orange triangle on the higher-priority layer. The black hit point belongs to the black quad on the lower-priority layer. As the vertices of the triangle are a strict subset of the vertices of the quad, the triangle must be embededed in the quad and both primitives must be planar.

Algorithm 4.6 lists the pseudocode for finding the vertices of a shared edge.

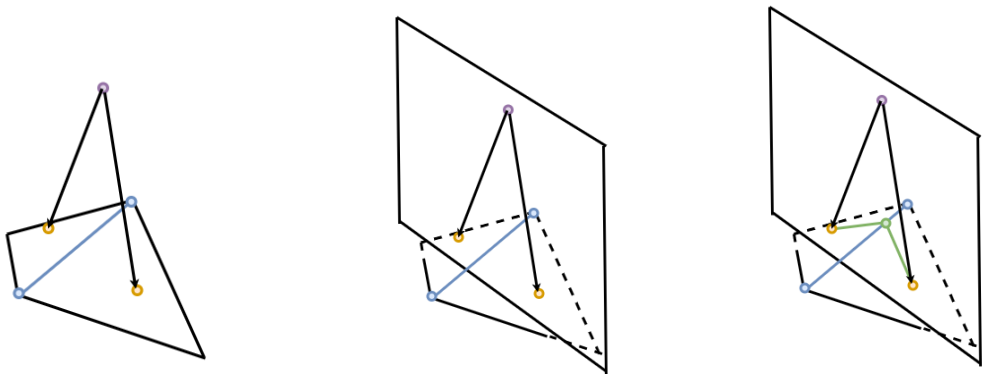
Algorithm 4.6: Pseudocode to find a shared edge if any exists

input : Set of vertices V_0 of the first primitive and V_1 of the second primitive.

output : Shared edge e_0, e_1 if any exists

```
1 hasSharedEdge ← false, isPlanar ← false
2 intersection ← V0 ∩ V1, count ← Count(intersection)
3 if count ≥ 2 then
4 |  e0 ← intersection[0],e1 ← intersection[1], hasSharedEdge ← true
5 if count = 4 or count = 3 and (Count(V0) = Count(V1) or IsQuad(V0) or
    IsQuad(V1)) then
6 |  isPlanar ← true
```

To find the midpoint M when a shared edge was found, the ray casting plane is intersected with the shared edge. The ray casting plane is the plane all rays lie within. All rays from the same polyline lie within the same plane. The ray casting plane can for this reason be constructed from any ray origin $\overrightarrow{r_o}$ and the two hit points $\overrightarrow{H_i}$ and $\overrightarrow{H_{i+1}}$. The point of intersection of the shared edge and the raycasting plane is the midpoint $\overrightarrow{M}$. Figure 4.7 visualizes this.



(a) (b) (c)

Figure 4.7: Step-by-step visualization of SED. (a) Two orange hit points H_i and H_{i+1} lie on primitives with a shared blue edge. The shared edge is formed by the blue vertices e_0 and e_1. (b) The ray casting plane is constructed from the purple ray origin and H_i and H_{i+1}. (c) The green midpoint M is found by intersecting the ray casting plane with the shared edge.

The ray casting plane is constructed by selecting any of the points $\overrightarrow{r_o}$, $\overrightarrow{H_i}$ or $\overrightarrow{H_{i+1}}$ as the plane origin $\overrightarrow{p_o}$. The plane normal $\overrightarrow{p_n}$ is constructed by the cross product of the two sides

of the triangle formed by $\overrightarrow{r_o}$, $\overrightarrow{H_i}$ and $\overrightarrow{H_{i+1}}$:

$$\overrightarrow{p_o} = \overrightarrow{r_o} \tag{4.6}$$

$$\overrightarrow{p_n} = \overrightarrow{r_o H_i} \times \overrightarrow{r_o H_{i+1}} \tag{4.7}$$

The midpoint $\overrightarrow{M}$ can then be calculated by solving the line-plane intersection for fraction f. $\overrightarrow{l}$ is the distance from the point of the line, in direction of the line l, to the point of intersection. p_l is the vector from a point on the line to the plane:

$$\overrightarrow{l} \leftarrow \overrightarrow{e_1} - \overrightarrow{e_0} \tag{4.8}$$

$$\overrightarrow{p_l} \leftarrow \overrightarrow{H_i} - \overrightarrow{e_0} \tag{4.9}$$

$$a \leftarrow \overrightarrow{p_l} \cdot \overrightarrow{p_n} \tag{4.10}$$

$$b \leftarrow \overrightarrow{l} \cdot \overrightarrow{p_n} \tag{4.11}$$

$$f = \frac{a}{b} \tag{4.12}$$

If $b \neq 0$, there is a single intersecton $\overrightarrow{M} = \overrightarrow{e_0} + \overrightarrow{l} * f$. However, when $b = 0$, the line and the plane are parallel. If also $a = 0$, the line is exactly on the plane and any point on the line can be chosen, e.g., $\overrightarrow{M} = e_0$. Otherwise, they are parallel and have no intersection.

4.4.1 Mending vertices

Because of floating-point precision, it is possible that two values that should be equal are not exactly equal. When searching for equal vertices, this becomes a problem. A small distance ϵ can be allowed to meld vertices that drifted apart back together. In this thesis, $\epsilon = 0$ was chosen, as no floating-point math is performed on the vertex positions of the geometry used for measurements. This allows to perform faster equality checks, by testing for bitwise equality. The presented algorithm works equally for $\epsilon > 0$, but the midpoint $\overrightarrow{M}$ then also is within ϵ distance between the two edges. As vertices are unordered, the midpoint will shift slightly based on which vertices are selected for the shared edge. Figure 4.8 visualizes how using $\epsilon > 0$ leads to mutiple possible midpoints $\overrightarrow{M}$.

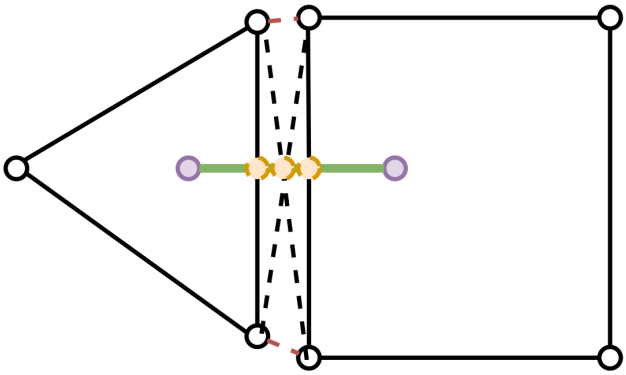


Figure 4.8: When vertices of tiles that drifted apart are melded by accepting a distance $\epsilon > 0$, the resulting midpoint can slightly shift depending on the selected vertices. Additional to a midpoint on any of the two edges of a triangle or quad, a virtual midpoint in-between can be the result. The virtual edges created by allowing $\epsilon > 0$ are shown dashed. Potential virtual midpoints formed by them are shown in orange.

4.5 On-Data Ratio

4.5.1 Assessing Uncertainty

Uncertainty is a key element of intuitive judgments. The trust in a measurement is often based on the trust in the tool used, without necessarily the basis to be as certain about a result as one should be. Amos Tversky formulated it wonderfully in *Assessing Uncertainty* in 1974 [Tve74]:

> UNCERTAINTY is an essential element of the human condition. The decisions we make, the conclusions we reach and the explanations we offer are usually based on beliefs concerning the probability of uncertain events such as the result of an experiment, the outcome of a surgical operation or the future value of an investment. In general, we do not have appropriate models according to which the probability of such events could be computed.

It is important to raise awareness about the existance of uncertainty. Measuring and visualizing it could help to prevent the false trust given in a measurement based on availability of data alone. In recent years, the research of uncertainty gained more relevance in fields other than statistics and cognitive sciences. Rhodes et al. [Rho+03] presented methods to visualize uncertainty in isosurface rendering. Waser et al. [Was+12] used ranged user controls to give users control about the amount of uncertainty in their simulations. Bayburt et al. [Bay+17] analyzed the geometric accuracy of the commercially available "WorldDEM core" height model. They concluded that while it is more accurate than other models they compared, it is significantly influenced by terrain inclination. Klug et al. [Klu+18] analyzed the uncertainty of their measurements for robotic total

simulation and verified it analytically, and via Monte Carlo experiments. Gillman et al. [Gil+21a] list uncertainty visualization as one of ten selected open challenges in medical visualization. Gillman et al. [Gil+21b] provided a state-of-the-art analysis of uncertainty-aware medical imaging in 2021 [Gil+21b].

4.5.2 ODR

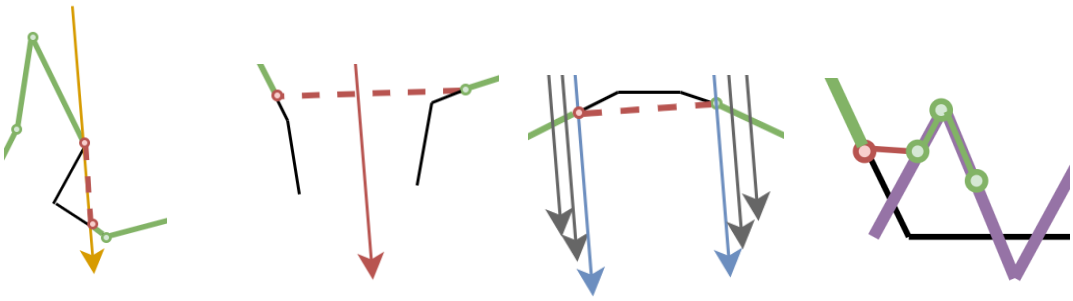
The On-Data Ratio (ODR) is proposed as a simple uncertainty metric to quantify the uncertainty of the measurements presented in this chapter. It is defined as the ratio of the length of polyline segments that are certain over the total length of the polyline. If a segment is marked certain, the start of the segment is classified as certain. For N points $\overrightarrow{p_i}$ forming a polyine, it is calculated as follows:

$$d(a,b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2 + (z_a - z_b)^2} \tag{4.13}$$

$$d_c(a,b) = \begin{cases} d(a,b) & \text{if certain}(a,b) \\ 0 & \text{else} \end{cases} \tag{4.14}$$

$$odr = \frac{\sum_{i=1}^{N-1} d_c(p_i, p_{i+1})}{\sum_{i=1}^{N-1} d(p_i, p_{i+1})} \tag{4.15}$$

Segments are considered on-data, and accordingly certain, when the direct line is a perfect terrain surface reconstruction according to the available data. During subsampling, the subsample is classified as certain or uncertain based on the termination criterion that is met. When subsampling terminates with an exact result through SED, the subsample is classified as certain. In all other cases, the subsamples are classified as uncertain, as it is not known how exact the surface reconstruction is. The direct line between two subsamples on a plane is an exact reconstruction, but this is uncertain if the plane is unknown. Figure 4.9 visualizes how the different termination criteria lead to uncertainty.



(a) (b) (c) (d)

Figure 4.9: Comparison of causes for uncertainty. (a) Occluded areas cannot be hit in the direction of the raycast. (b) Missing data and holes cause uncertainty. (c) Termination due to a maximum number of subsamples. (d) Measuring across multi-layered data is a source of uncertainty. The change from the black low-resolution and low-priority terrain to the purple high-resolution and high-priority terrain is classified as uncertain.

For the user, the ODR value is shown as percentage and by coloring each polyline segment as certain or uncertain. The most relevant difference is between measurements with $odr = 1$, which are completly certain, and those with $odr < 1$, which contain any source of uncertainty. A small uncertain segment across a tiny terrain hole might not be visible to the user, as the render segment might be smaller than a pixel. However, by listing uncertain segments and allowing the user to jump to them, they are made aware of their existance and can investigate why a measurement is uncertain.

Screenhots from Visionary utilizing ODR to classify measurements are shown in Chapter 6. Ideas for future work to account for other sources of uncertainty are shown in Chapter 7.

CHAPTER 5

Implementation

In this chapter, the implementation of Visionary is presented. The program architecture, used frameworks and libraries, and implemented tools are presented. The OPC format and the Renderable Blob format are explained, as well as all preprocessing that is used to convert from OPC to renderable blobs. Optimizations used to achieve real-time streaming performance are explained and code as well as pseudocode for them is shown. The restrictions and limitations of Unity are discussed and solutions used to bypass them are presented.

5.1 Overview

Visionary is a prototype of a multithreaded out-of-core streaming solution with polyline-based surveying tools for geomorphometric analysis of large-scale terrains. It supports scenes in OPC file format and preprocesses them into its custom Renderable Blob file format. Renderable blobs contain textured triangle mesh data. As the input textures contain the lighting information, no further lighting is applied. Textures can be in RGB24 or R8/A8 format. RGB24 textures contain the red, green and blue channel in 8 bit precision each. As the used textures contain lighting information as well, the R8 or A8 textures contain only 8 bit illumination information, but no color information, in either the red or the alpha channel. Other material models could easily be extended, but these covered the types of the used scenes.

Visionary is implemented using the Unity gmae engine and its Data-Oriented Techstack (DOTS).

5.1.1 Frameworks and Languages

Visionary was developed in Unity, a higher-level engine, developed by Unity Technologies. Unity's success as engine for mobile game development, with a reported 71% of top

1,000 mobile games in 2020 [Unip], shows a clear strength as engine for mobile game development. In recent years it also has been used to build tools in a wide range of other fields, such as movie production [Bri19], artificial intelligence [Jul+18], autonomous driving [Ron+20], CAD-based industries [Teca] and others, e.g., VR therapy [Riz+15].

Multiple versions of the Long Term Support (LTS) stream 2020.3 [Tecm] were used to develop Visionary. Unity 2020.3 was the at the time of development the highest version with DOTS support. DOTS is a set of libraries to support high-performance real-time applications. The core of DOTS consists of:

- an Entity-Component-System (ECS) [Tec20] to support the design of data-driven efficient algorithms,
- the highly optimized Burst Compiler [Unic],
- the Burst Compiler compatible math library [Unit],
- the job system, which allows writing efficient parallel algorithms over ECS data [Unir], and
- the new physics system [Unib], with pluggable physics middleware, giving access to, e.g., Havoks Physics for Unity [Unia] and being compatible with ECS data.

Unity supports both a Mono- and a C++-based scripting backend. In both backends, the scripting language is C#. The C++ backend uses Intermediate Language To C++ (IL2CPP) to convert user-written C# code to C++. The IL2CPP transpiler converts Microsoft Intermediate Language (MSIL) code into C++ code. This gives access to performant native binaries for the targeted platform. IL2CPP uses ahead-of-time (AOT) compilation. While IL2CPP does not guarantee code to be faster in every case, many algorithms proved to be faster with IL2CPP than with the Mono runtime. The highest supported language version was C# 8 with .NET STANDARD 2.0 as API compatibility level. Roslyn [Mic] is the compiler used by Unity to create managed assemblies (.NET DLLs). The Unity Burst [Unic] compiler was used to optimize burst compatible jobs. Burst translates IL/.NET bytecode to highly optimized code using LLVM. All performance-critical assemblies were compiled as unsafe [dbo] to allow unchecked access of unmanaged memory.

The DOTS libraries were required to realize this project in Unity. DOTS provides functionality which was not available in Unity before, e.g., collider serialization. Furthermore, it supports writing highly efficient and parallel code for data-oriented algorithms with its powerful job system and burst-compiled job code. The Unity Entities [Tec20] package is the core package for DOTS-based code and contains the framework for the Entity Component System (ECS) architecture. For this thesis, version 0.51 of the Unity Entities package was used. The version of the Entities package is used to identify compatible DOTS libraries. The Unity Job System [Tecd] was used to schedule and parallelize burst-optimized tasks. The Unity Mathematics [Unit] package was used to utilize the general

and SIMD optimization support through the Burst compiler. The Unity Physics [Unib] libraries were used to implement the high-performance ray casting used for the measurements. The Havok Physics for Unity [Unia] package was used as the physics backend. The Unity Collections [Uniq] libraries were used to achieve high-performance allocations of unmanaged memory. Unity's Universal Render Pipeline [Uniu] (URP) was used to access its hybrid renderer (version 2), which is required to render ECS-based objects with Entities version 0.51. The Unity Shadergraph [Tecp] was used to generate URP compatible shader code.

The UnityEngine and UnityEditor packages contain the core functionality of Unity. The core of the Unity framework before DOTS were the types *GameObject (GO)* and *MonoBehaviour (MB)*. GameObjects allow creating objects, whose lifecycle is managed by Unity. MonoBehaviours are the extension point for custom code, integrated into the Unity game loop via overriding predefined callbacks. DOTS is not meant to replace the GameObject- and MonoBehaviour-based workflow, but is designed to synergize with it. Both frameworks are thus used side-by-side. The Unity User Interface (UGUI) [Unis] package was used to create the UI. The Unity TextMeshPro [Tecl] library was used for text rendering of signed-distance-field-based fonts (SDF).

MessagePack-CSharp [Mes] was used to read and write MessagePack files. Custom resolvers and serializers were implemented to support custom types and types from DOTS not yet supported.

UniRx [Unim] was used for reactive programming, which was used to decouple architectural layers. UniTask [Unin] was used for allocation free async/await support. All scheduling between thread pool and main thread was realized via the UniTask scheduler.

Zenject [Zen] was used as dependency injection framework, as Unity does not provide one.

Magick.NET [Lema] was used for general image conversion operations. The MonoGame DDSLoader [Monb] was ported to implemented runtime support for Direct Draw Surface [lora] texture image files.

Microsofts Task Parallel Library [IEv] was used to parallize code via thread pooling. Types and algorithms for managing unmanaged memory from the .NET libraries System.Buffers, System.Memory, System.Net, System.RuntimeCompilerServices.Unsafe, System.Threading.Tasks.Extensions and System.Windows.Forms were used by directly referencing their assemblies. Access to these libraries is in Unity restricted because it uses an older version of .NET.

5.1.2 Main loop

Unity enforces a strict main loop model. A single dedicated main thread is working through this loop. Custom code is executed via predefined methods called at specific points in this main loop. There are over 35 supported callbacks [Teci] with the two

most important ones being *Update*, early at a frame, and *LateUpdate*, late in a frame. Figure 5.1 shows a schematic representaiton of the execution order.

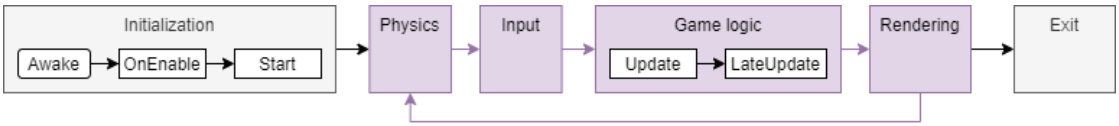


Figure 5.1: A schematic representaiton of the Unity application flow. The main loop highlighted in purple. Only the most important callbacks and none of the internal steps are shown.

Both asynchronous and multithreading tasks are scheduled from and completed in the main thread. If the target platform does not support multiple threads, the tasks are completed in the main thread. Multiple different ways to achieve asynchronous code execution are available, such as coroutines, threads, tasks and jobs. They differ in kind of management, async-await support, overhead, and multithreading support. Coroutines are a low-overhead solution to achieve asynchronous, but not multithreaded, execution. Visionary does not make direct use of coroutines, but indirectly uses them via libraries, e.g., UI libraries that schedule graphical updates by using coroutines. Threads give direct access to common language runtime threads, but also are only used indirectly. Visionary uses tasks, which are executed on one of the threads of C#s *ThreadPool*, to perform long-running operations. As tasks are a way to perform true multithreaded code, they are decoupled from the main loop and are allowed to run for as long as required. However, many operations in Unity are restricted to be performed on the main thread, which requires performing a context switch back to the main thread to make use of a task's results. The Unity Job system is the most recent addition to write multithreaded code in Unity. It not only distributes work, but also schedules job dependencies. It provides many multi-threading safety guards. Job code written within the strict limitations of the Burst compiler, such as restrictions to blittable types, can be highly optimized by it. The Unity Collections [Uniq] library provides Burst compatible collection types to store data efficiently and pass it between jobs and other code. The access to custom allocators gives access to fast allocations of unmanaged memory for short-lived data. However, jobs are restricted to complete within four frames, which does not make them suitable for long-running operations. These restrictions also affect the file system access. Burst-compiled jobs only provide restricted file system access via the burst compatible *AsyncReadManager*. Jobs can be scheduled to be run in the main thread if needed, which leads to the loss of most benefits other than scheduling, but it makes debugging easier. They are executed on a fixed set of worker threads similar to the C# ThreadPool.

5.1.3 Program Architecture

Visionary is structured into the following main components:

- preprocessing (converts OPC scenes into Renderable-Blob-based scenes),
- streaming (performs streaming decisions, streams in and out),
- rendering (converts textured mesh data into renderable objects and performs LOD selection).
- measurement (performs ray casting, subsampling and uncertainty calculations),
- line rendering (renders polylines for measurements),
- and GUI (Graphic User Interface to select scenes to load, configure settings and initiate measurements).

These components will be described in the following sections.

Preprocessing

The prepropcessing component is responsible for loading, cleaning, and converting OPC files to Renderable Blob files. Its GUI allows to select the source and target directories, as well as which of the OPC hierarchies, found during a directory scan, should be loaded and converted. To avoid scanning directory trees with several hundred thousand files, which can take several seconds even on a fast SSD, a once scanned tree is cached and stored as MessagePack file. Large-scale scenes and their directories are rather static and rarely change, and thus the user experience is improved with near-instant access to the available scenes in subsequent executions.

All custom serialization components for DOTS, MessagePack and OPC files are part of the preprocssing component.

Streaming

The streaming component is responsible for managing currently loaded HLODs. It contains the implementation for the M-HLOD streaming algorithm and the LOD selection presented in Chapter 3. The result of each update of the streaming component is a new set of Entities to be rendered based on the currently streamed-in LOD nodes.

Measurement

The measurement component provides the functionality for the two-phase measurement algorithm using SED and ODR presented in Chapter 4. It streams geometry independently of the streaming and LOD decisions for rendering. Measurements are user-initiated tasks, and the measurement component for thsi reason does nothing between two measurements.

Line Rendering

The polylines used for the measurement are rendered by using Unity's *LineRenderer* component. One LineRenderer per currently shown Measurement is spawned. Line points are positioned in world space and the width of the lines is scaled according to the closest point to the camera, to ensure a close to constant screen width.

5.1.4 GUI

The GUI is implemented using Unity's User Interface (UGUI) [Unis]. UGUI provides different rendering modes, with *Screen Space - Overlay* being used in Visionary. This renders the UI on top of the rendered scene.

A perspective camera is used for the viewer, and camera stacking is utilized to achieve priority rendering as described in Chapter 3.

The user-drawn polylines are implemented by reacting to clicks and converting the pixel positions to world space coordinates. The polylines are then drawn in world space using the Linder Renderering system, with lines on the screen being drawn close to the near plane.

The presentation layer is decoupled from the application layer by using UniRx [Unim] as library for reactive programming. This allows to bind data to UI components similar to DataBinding in the C# Model-View-ViewModel pattern [dav].

Tools

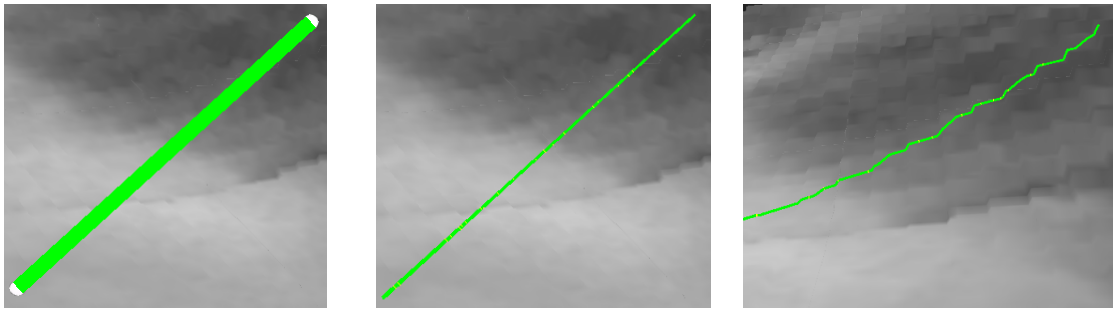
The functionality implemented in Visionary is explained in the following sections.

5.1.5 Navigation

Navigating large-scale terrains requires both fast camera movements across the surface and fine controls close to the surface. A free-fly camera was implemented as Unity does not provide such a camera model. The keys W, A, S, D are used for forward, backward, left and right navigation. Yaw and pitch rotations are enabled while holding MouseRight and react to mouse movement. The keys Q and E allow roll rotations. To achieve quick and precise navigation, the speed accelerates when the users hold LeftShift, and deaccelerates otherwise.

5.1.6 Polyline Drawing

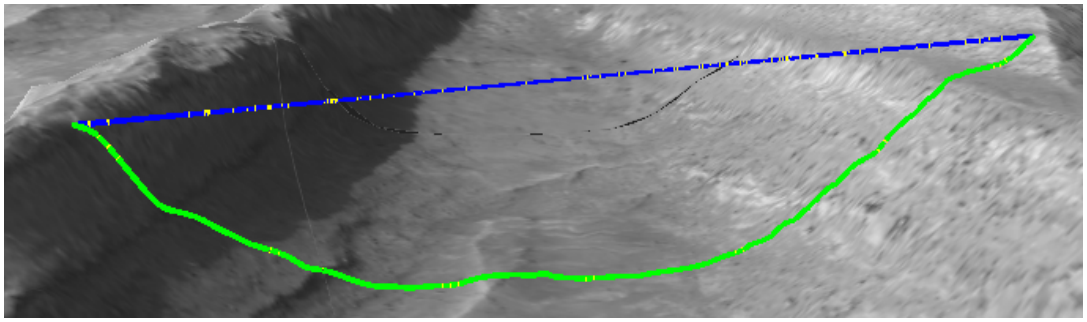
The polylines for the measurement presented in Chapter 4 are user-drawn and then projected based on the selected direction and subsampling strategy. Figure 5.2 shows an example of a polyline before projection, after projection, and after camera movement.



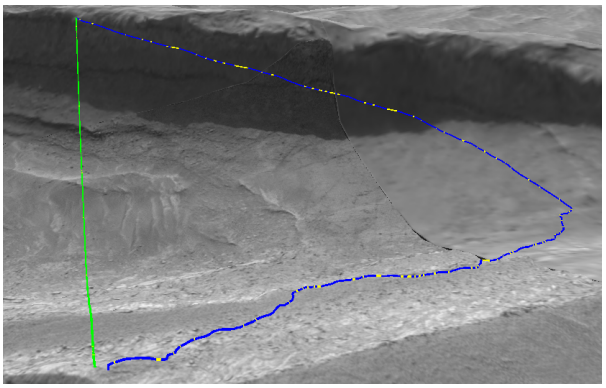
(a) (b) (c)

Figure 5.2: (a) The polyline drawn by the user, before the projection. (b) The polyline after projection in view direction. (c) The polyline after camera movement to a different perspective. The visible steps are due to the resolution of the terrain data and not artifacts caused by the algorithm.

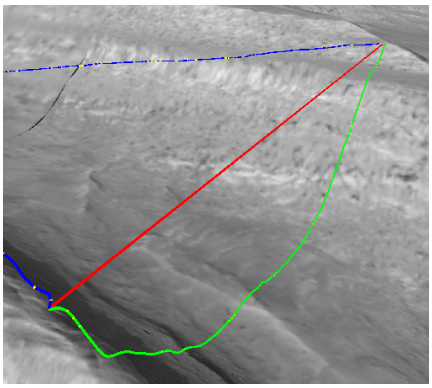
In this thesis, multiple settings for the polyline projection were implemented. Traxler et al. [TO15] used the projection in direction of the view, and downward onto the terrain, to measure different geomorphometric properties of Martian terrains. Both directions were implemented as options in Visionary. Fixed-rate and variable-rate subsampling are available as subsampling strategies. Subsampling can be deactivated altogether by selecting fixed-rate subsampling with 0 subsamples. Polylines can be reprojected under different settings. This method was used to create the data for the evaluations by creating a measurement with one setting, and then reprojecting the exact same feature points with a different setting. Figure 5.3 shows a comparison of the results with view projection, down projection and no subsampling.



(a)



(b)

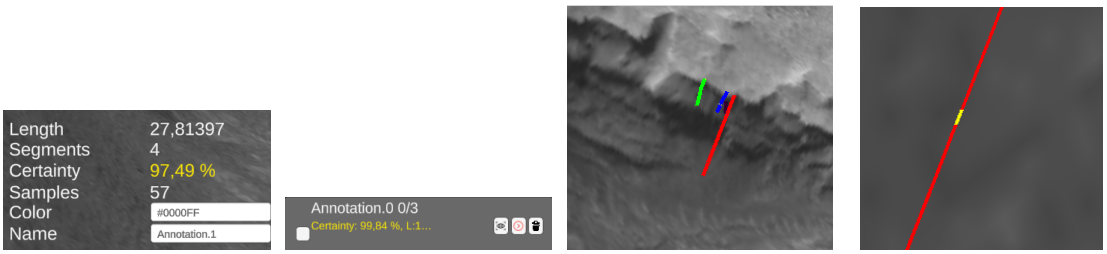


(c)

Figure 5.3: Different types of measurement. (a) shows a blue polyline projected in view direction and a green polyline projected in down direction. Projecting downward is used to measure walking distances and projecting in view direction to survey outcrop features. (b) shows the same polylines after moving the camera. The difference between the directions is emphasized. (c) shows a red direct line measurement without subsamples. This is useful to measure the extents of terrain features.

5.1.7 Annotations

Measurements are used to create annotations. This name is used in PRo3D [Prob] and also in Visionary. Annotations can be renamed and recolored to keep them apart. The annotation list allows cycling through segments and analyzing uncertain segments. Figure 5.4 shows three annotations at a hill wall, the interface to display the length, number of segments, their ODR value, and the number of samples used.



(a) (b) (c) (d)

Figure 5.4: (5.4a) shows the annotation details listing the length, the number of segments, the ODR, and the number of samples. It also allows changing name and color of the annotation. (5.4b) shows the annotation list entry, which allows selecting annotations and cycling through their segments. (5.4c) shows three annotations from far away. The uncertain segment is not visible at this distance. (5.4d) shows a zoomed in view on an uncertain segment after cycling through segments. This allows to identify the uncertain segments.

5.1.8 Selection of Up

To perform projection into the down direction, first the up-vector of a terrain needs to be determined. Paar et al. [Paa] used the NASA SPICE kernels [NAS] to evaluate the planetary coordinate system for their PRoViDE framework, and it is used in PRo3D, which is part of the framework, by Ortner et al. [Prob]. For this work, the Martian epsilloid was approximated as sphere, and a constant up-vector was assumed per measurement. This simplification ignores the effect of the planetary curvatore over long measurements. Figure 5.5 shows the coordinate system gizmo placed, with the green axis facing up.

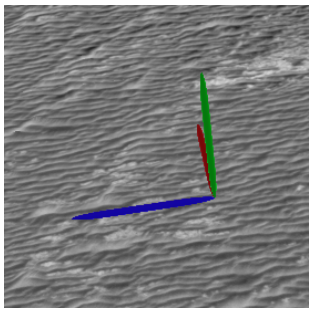


Figure 5.5: The coordinate gizmo shows the point on the surface that is used to evaluate the up-vector. The green axis of the coordinate gizmo visualizes the up-vector.

5.2 Ordered Point Cloud File Format

This section gives an overview of the *Ordered Point Cloud* (OPC) file format. OPCs store arbitrary depth LODs of discrete textured meshes by using a folder hierarchy, with one

folder per LOD node, and one XML-hierarchy file to store the relationship. Mesh data in OPC is stored in binary format *Aardvaark Array* (extension: .aara) files. Large-scale terrains are stored partitioned and tiled across multiple OPCs. All scenes used in this thesis are in OPC file format. This is the file format used in PRo3D. This format was chosen as input format to evaluate the presented algorithms against PRo3D, using the exact same input data.

The OPC file format was first presented by Ortner et al. [Ort+10] to support visualization of infinite surfaces with an arbitrary number of levels of detail. The datastructures for LOD hierarchies in this thesis are based on the LOD hierarchies of the OPC file format.

Paar et al. [Paa+15a] have used the OPC file format to visualize Martian terrain reconstructed from the ExoMars PanCam wide-angle multispectral cameras in PRo3D [Prob]. They converted the original point-cloud information from the *Generic Point Cloud* (GPC) file format to OPC. To support the performance for streaming purposes, the OPC files are further preprocessed into a binary large object file format (blob) in this thesis. Section 5.3 describes the resulting blob file format after preprocessing.

An extract of the OPC file format specifications relevant for this work can be found in Appendix A 7.1. The full specification is available in the Ordered Point Cloud File Format Documentation [TO].

5.2.1 Files and Folders

Scenes in OPC format are stored in a folder-based hierarchy. The *patchhierarchy.xml* file of each hierarchy stores the tree relationship of the LODs. Each LOD node is referred to as *patch*. The data of each patch is stored in its own folder, with the texture images being stored in a shared image folder. The *patch.xml* file contains the relative file paths to images for textures and binary .aara files, which contain vertex positions, normals, texture coordinates, and texture weights.

The folder structure of an OPC hierarchy looks as follows:

```
guid-of-opc/
├── images
│   └── ...
└── patches
    ├── patchhierarchy.xml
    ├── guid-of-patch-1/
    │   ├── patch.xml
    │   ├── Positions.aara
    │   └── ...
    └── ...
```

Figure 5.6: The OPC folder structure.

5.2.2 Patch

The definition of each patch is stored in its *patch.xml* file. It contains the relative paths to the *Aardvark Array* (extension: .aara) files including vertex positions, normals, and texture coordinate data in binary format. The mesh positions are stored in local space and an affine transformation matrix **M** called *LocalToWorld* is used for the coordinate transfer.

5.3 Renderable Blobs

Visionary preprocesses all OPC files into its own streaming optimized binary format, called the *Renderable Blob* file format. This optimization is required to efficiently stream in nodes. Unity does not allow to create the large *Mesh* and *MeshCollider* objects required for rendering and measuring fast enough for interactive streaming, and for this reason preprocessing was required. The textured meshes are preprocessed into the GPU layout used for rendering them. This allows to achieve interactive frame rates for large-scale terrains with the streaming algorithm presented in Chapter 3. The bounding volume hierarchy of the geometry used for ray casting is preprocessed, a *MeshCollider* in Unity terms. This allows to stream in geometry sufficiently fast for interactive measurements on large-scale terrains with the measurement algorithm presented in Chapter 4.

Renderable blobs, in contrast to OPC, are optimized to ensure that:

- All data is ready to use without further processing. No triangulation cleaning or relayouting of memory is required.
- All mesh data is densely packed in the GPU memory layout. Invalid faces are removed.
- Mipmaps are precalculated and texture data reformatted to the memory layout used on the GPU.
- Data that is used together is read together. This increases efficiency of file system accesses.
- Serialization formats used are optimized for deserialization. This is important, as all data is serialized only once, but deserialized many times during streaming.

An extract of the file format specification of renderable blobs can be found in Appendix B 7.1.

5.3.1 Files and Folders

The OPC folder structure is kept, but no image folder is used, as the texture data is part of the renderable blob. Instead of storing images, textures are stored. All data

required to store one OPC patch, which is one textured triangle mesh, is stored as one *renderable.blob* file.

This reorganization of files reduces the number of files to about one third. This is irrelevant in small scenes, but becomes relevant in large-scale terrains with several hundred thousand files. Fewer files means fewer file paths and file handles to process and this helps to keep streaming efficient. Figure 5.7 shows the OPC folder structure compared to the *Renderable Blob* folder structure:

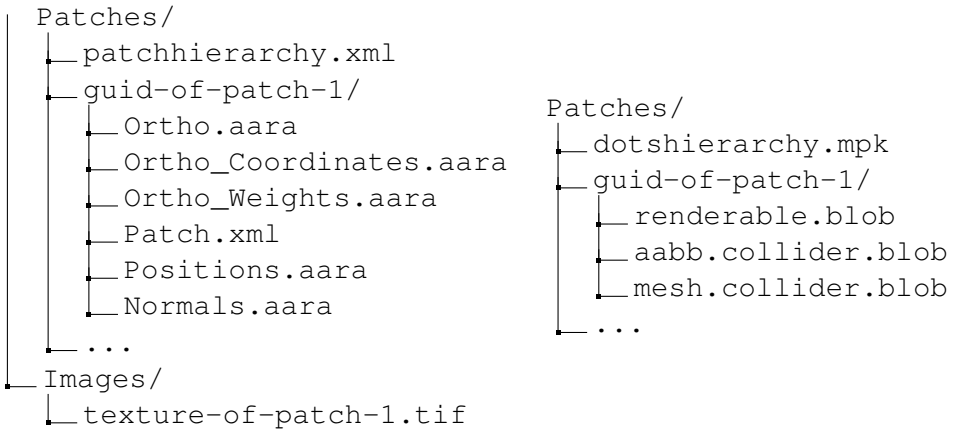


Figure 5.7: Comparison of the folder structure of OPC scenes and scenes in Visionary.

5.3.2 dotsHierarchy.mpk

The XML-hierarchy-based HLOD information from the OPC file *patchhierarchy.xml* is converted to an index-based aligned array structure, and stored as MessagePack file *dotsHierarchy.mpk*.

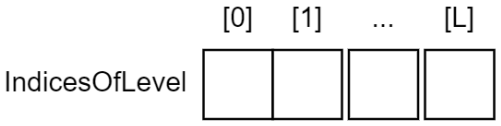
Aligned arrays of value typed data allow iterating linearly in a much more cache efficient way than reference-based structures. During linear array iteration, the cache line is then filled with multiple array elements at once, which cause high cache hits. In contrast, reference-based structures require jumping to the memory section the reference is pointing to, which is less likely to be cached.

In aligned arrays, one array per data type is used. The data of each *node* i is stored at the index i in each array. When the types are value typed instead of reference typed, linear iteration becomes highly cache efficient, as cache lines will be filled with the next accessed array elements.

The LOD tree is stored by storing the indices of the parents and children. The index of the parent of *node* i is stored in *IndexToParentIndex[i]*. The root has no further parent. This is indicated by using -1 as parent index. The indices of all children of *node* i are stored in *IndexToChildIndices[i]*. The leaves have no further children as indicated by an empty array. Frequently accessed data, such as the indices per level, is cached in

the hierarchy in the same way. The LocalToworld affine transformation matrix **M** is elevated from being stored per patch in *patch.xml* to being stored in an aligned array per hierarchy. This allows faster access to this important data. Figure 5.8 visualizes the aligned array-based hierarchy,

(a) Data per node is stored in aligned arrays. The data of *node 1* (*blue*) is stored at *index 1* of each aligned array.



(b) Frequently used lookups such as the indices per LOD level are stored in the hierarchy.

Figure 5.8: The data per node is stored in a cache-friendly way in its hierarchy and accessed by the nodes index. Data of the same type can be iterated over efficiently.

The MessagePack [Mes] format was chosen for the hierarchy file, as it is a general-purpose binary serialization format, it is fast to deserialize, and easy to extend for all used custom non-primitive types.

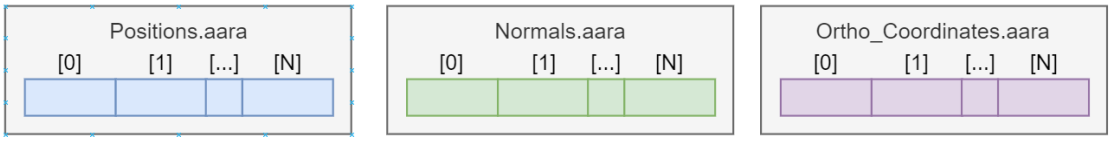
5.3.3 renderable.blob

The *renderable.blob* file contains the textured mesh information for rendering. The mesh information, stored separately in multiple files in OPC, is merged into one array of *VertexData*, which is equivalent to the GPU buffer format. Listing 5.1 shows the definition of VertexData. *float2* and *float3* are 2- and 3-component single-precision float vectors.

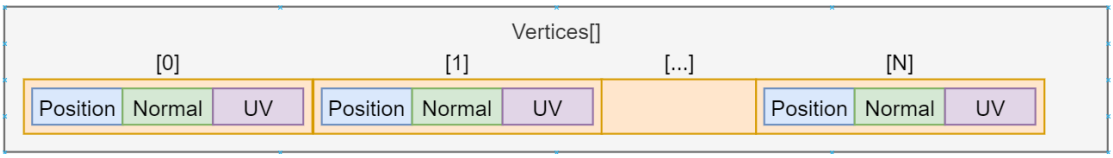
```
VertexData
{
  float3 Position;
  float3 Normal;
  float2 UV;
}
```

Listing 5.1: The definition of VertexData

Figure 5.9 shows the memory layout before and after the merge of multiple OPC files to one array of VertexData.



(a) OPC scenes store positions, normals and texture coordinates in separate files.



(b) Position, normal and texture coordinate are combined per vertex and stored as an array of *VertexData*. This memory layout is used during rendering and allows to copy the read data directly into the target GPU buffer.

Figure 5.9: Comparison of memory layout in Aardvark Array files and Renderable Blobs.

All types used for renderable blobs are blittable [gewa]. This is essential to achieve the serialization and deserialization performance for streaming. The deserialization of a blittable type with known memory layout only consists of reading the data from storage and reinterpreting the read bytes. Copying memory and reinterpretation is the fastest way to deserialize data.

5.3.4 .collider.blob

The serialization of the geometry used for the measurement algorithm utilizes this. To serialize the geometry, its blittable type is reinterpreted as bytes, and then the length is written to the target file, followed by the bytes of the geometry. Deserialization inversely only requires to read the length of the geometry data, copying the bytes and reinterpreting it back to the original format. The geometry information is stored in the *mesh.collider.blob* file.

The rendering and geometry information for measurements was split to avoid overreading data during loading of rendered meshes or loading of geometry for measurements. Each subsystem only loads the data required to render or measure.

The AABB is stored in the exact same layout as the geometry for measurement, but it is separated, as it is only read once. The AABB is stored in the *aabb.collider.blob* file.

5.4 Preprocessing

This section gives an overview of all steps performed during the offline preprocssing stage. During preprocessing, all OPC files are read, cleaned, triangulated and then written back to the file system as Renderable Blob files. Figure 5.10 shows all preprocessing steps.

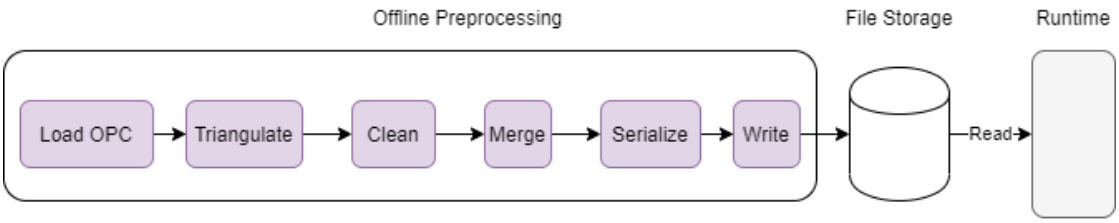


Figure 5.10: The pre-processing stages in order. Pre-processing ends with writing the serialized files back to the filesystem.

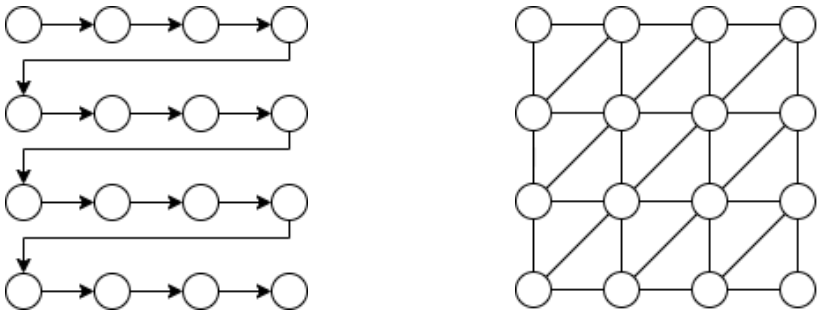
5.4.1 Loading OPC

All scenes used for this work are available in OPC format, which was described in Section 5.2. They are preprocessed to be stored in the Renderable Blob format described in Section 5.3.

Loading an OPC from a target directory requires scanning the directory tree for OPCs. Directories containing OPCs can be identified based on the existance of a patchhierarchy.xml file.

5.4.2 Triangulation

OPC files allow storing meshes in multiple different representations. All scenes relevant for this work were stored as a list of quads, with their data stored in linearized arrays in *row major* order. A list of quads can be triangulated as regular grid. No face indices are stored, and triangulation is based on the implicit grid relationship. Figure 5.11 shows a possible triangulation for a RowMajor array of known width and height.



(a) $4x4$ *RowMajor* grid. (b) Regular grid triangulation

Figure 5.11: RowMajor linearized arrays store the data row by row. Given the width and height of the 2D grid, a regular grid triangulation is possible.

Altough all positions are stored in a linearized 2D array, this does not mean that all stored meshes are rectangular. To indicate empty space, *"Not A Number"* (*NaN*) is used.

Vertices containing *NaN* values are discarded. Figure 5.12 shows an example of such a non-rectangular mesh.

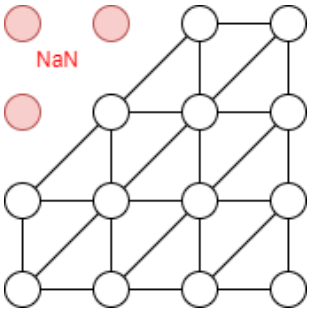
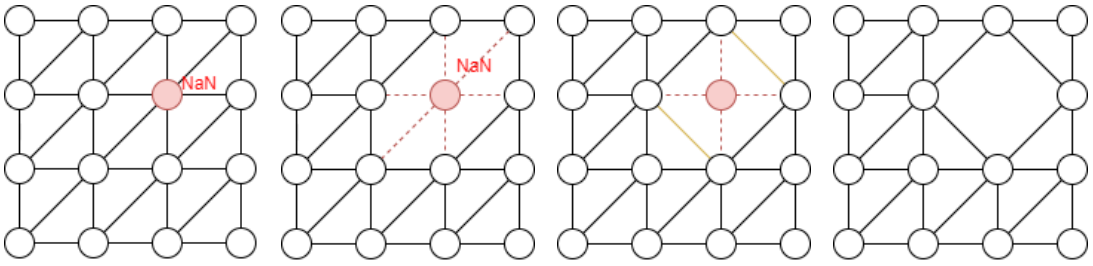


Figure 5.12: *NaN* values are used to mark the border of a non-rectangular patch.

5.4.3 Cleaning

Indication of empty space is not the only source of invalid values, such as *NaN*. Floating-point errors and other reconstruction faults can lead to these values as well. When any component value is invalid, the whole vertex has to be discarded. When a vertex is discarded, the triangle referencing it becomes invalid and has to be discarded too. Discarding a triangle causes a hole. During the cleaning stage, all invalid vertices and triangles are removed. To keep holes small, affected quads are retriangulated where possible. Figure 5.13 shows how a hole is detected and affected quads are retriangulated.



(a) *NaN* value detected. (b) *NaN* value removed (c) Re-Triangulation (d) Resulting hole.

Figure 5.13: Vertices containing invalid values, such as *NaN*, are removed alongside their affected triagnles. This causes a hole. Retriangulation can help to reduce the size of the created hole.

Discarding vertices causes data arrays to contain unused entries. To compact them densely, all elements are shifted left, until no unused entries remain. This requires updating all vertex data arrays. Shifting the data also requires to update the indices accordingly. Figure 5.14 gives an example of such a shift.

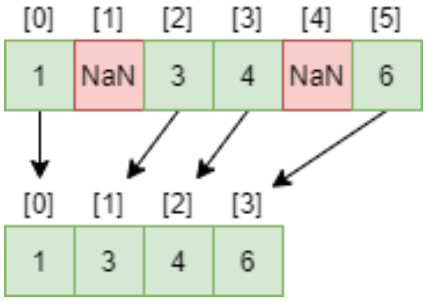


Figure 5.14: Discarding invalid NaN values leaves array indices [1] and [4] unused. By shifting all data left, the arrays become compact again. No unused indices remain. All aligned arrays have to be shifted accordingly.

5.4.4 Merge, Serialize and Write

Through conversion from OPC to , the memory layout is explicitly changed and files are merged, as described in Section 5.3. The output types are all densely packed and identical to the memory layout during runtime. The hierarchy object is serialized to bytes using MessagePack [Kaw] and written to file with *File.WriteAllBytes(path, bytes)*. The renderable blob is serialized to bytes using *Unity.Entities.BlobAssetReference* and written to disk with a custom version of *Unity.Entities.Serialization.StreamBinaryWriter*, the *DeprecatedStreamBinaryWriter*. In Section 6.2 the limitations of the Unity engines are discussed, and why a custom serializer was required. The *Unity.Physics.Collider*, which contains the geometry for the measurement, is written to disk by writing its byte size, followed by its bytes. The *DeprecatedStreamBinaryWriter* also was used for this step.

The multithreaded job system from DOTS and its Burst compiler are used to efficiently create MeshColliders. Listing 5.2 shows a code example for the burst-compiled creation of MeshColliders using jobs.

```
// Allocate NativeArray memory
using NativeArray<float3> vertices =
   v.ToNativeArray(Allocator.TempJob);
using NativeArray<int3> indices =
   i.ToNativeArray(Allocator.TempJob);

// Create the job and pass in data
var job = new CreateMeshColliderJob {
 Vertices = vertices,
 Indices  = indices,
 Colliders = new NativeArray<BlobAssetReference<Collider>>(1,
    Allocator.TempJob, NativeArrayOptions.UninitializedMemory)
};
```

```
// Run the job and complete it
var handle = job.Schedule();
handle.Complete();

// Copy back the results
using NativeArray<BlobAssetReference<Collider>> colliders =
   job.Colliders;
using BlobAssetReference<Collider> meshCollider = colliders[0];

// Serialize the meshCollider
// ...

[BurstCompile]
struct CreateMeshColliderJob : IJob {
 [ReadOnly] public NativeArray<float3> Vertices;
 [ReadOnly] public NativeArray<int3> Indices;
 public NativeArray<BlobAssetReference<Collider>> Colliders;
 public void Execute() => Colliders[0] =
    MeshCollider.Create(Vertices, Indices);
}
```

Listing 5.2: Burst-compiled CreateMeshColliderJob

5.5 Optimizations

5.5.1 Memory Hierarchy

Modern CPU and GPU architectures are designed with complex cache hierarchies. Kumar and Singh [KS16] gave an overview of the performance differences of varying cache types measured in CPU cycles. According to them, the difference in latency is an order of magnitude per layer. The access to the filesystem is multiple orders of magnitude slower than access to the hardware caches available in CPUs. In a real-time out-of-core rendering system, the data has to be copied through the whole memory hierarchy. The slowest layer is the local or remote file system. The next faster layer is the working set memory. The last layer rendering data needs to be copied to is the GPU memory. Efficient memory layout is in consequence essential to achieve the required performance. Figure 5.15 shows a simplication of the complete memory hierarchy, with emphasis put on the CPU cache hierarchy, as the proposed algorithms are CPU-based.

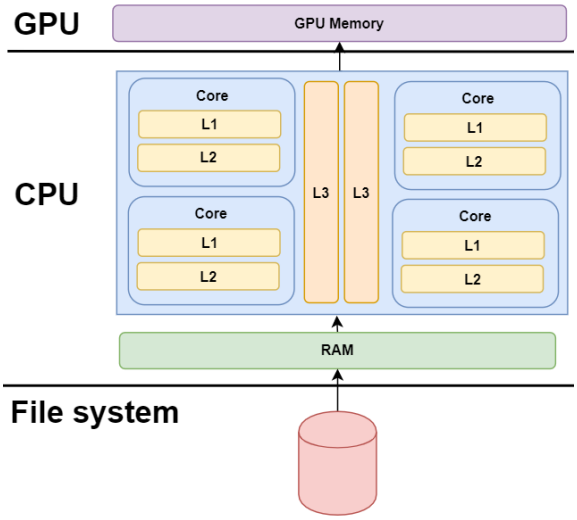


Figure 5.15: Modern CPUs and GPUs have sophisticated multi-layer cache architectures. Both have dedicated caches for specific operations. The CPU cache architecture shown is a schematic representation of the Zen 2 cache architecture and shows a per core L1 and L2 cache model with a shared L3 cache [Naf+21].

One of the simplest, and hence fastest, operations is sequentially copying data from one layer of the memory hierarchy to the next. This is only possible without in-between processing if the data is stored in the memory layout required by the target layer. In this thesis, all data is for this reason stored in either CPU- or GPU-optimized layout, depending on where it is finally processed during runtime. This is common practice when the time for preprocessing is possible and the data is available offline.

Many optimizations in Visionary are designed to utilize this memory hierarchy as efficiently as possible. The following methods were used to achieve a high streaming performance:

- The Renderable Blob format presented in Section 5.3 is designed to avoid processing data between layers. This allows to directly copy the data required for rendering the textured meshes directly into the target buffer.
- The aligned-array-based hierarchy presented in Section 5.3 stores data in iteration order, and it stores it densely packed as continuous memory. This optimizes the read performance of linear iteration during the LOD selection.
- Software caches were used to reduce acccess to the slower file system by keeping the data in faster RAM and the CPU hardware cache.

5.5.2 Memory Management

Unity uses three memory management layers [Tecg]:

- C# memory with a managed heap and garbage copllector. In the following section the term managed memory is generally used for this memory.
- C# unmanaged memory. This memory is accessed by using the *Unity.Collections* library. The term unmanaged memory is used in this thesis for this memory.
- Native memory. Native memory is C++ memory used by Unity to run the engine. This memory is mostly inaccessible to users.

As Visionary runs in a common language runtime (CLR), it uses the CLR garbage collector (GC) [gewb]. During garbage collection, all other threads have to be stopped, to free unused memory. This can have significant impact on the performance. While garbage collection is fast for short-lived objects and memory allocated on the stack, it can stall the system noticeably for large objects and complex reference hierarchies. Avoiding temporary large copies and complex reference hierarchies is therefore important. Thus, the renderable blobs are stored in unmanaged memory and passed by reference within each layer. To copy the data across layers the fast memcpy operation is used.

Steinberger et al. [Ste+14] presented a custom allocation scheme to manually manage memory allocations for large terrain rendering. Visionary instead uses the *Unity.Collections* library, and it's custom allocators, for fast allocation of managed und unmanaged memory.Allocations are kept to a minimum by keeping frequently used memory allocated permanently with the *Allocator.Persistent* option. Temporary allocations are used with the faster allocators *Allocator.Temp* and *Allocator.TempJob*. Memory initialization and array safety checks are skipped.

5.5.3 Pooling Texture Buffers

While the presented memory management is optimized to efficiently allocate and deallocate data, it is still bound by disk bandwith, memory bandwith and garbage collection. The largest streamed data is texture data, which is copied into CPU memory only to buffer it for transfer to the GPU. Ravi et al. [RBK20] presented a method for direct GPU I/O, avoiding the intermediate copy by using Nvidias GPUDirect Storage [Gpu]. However, GPUDirect Storage is not yet available in Unity. Caching is used to reduce the performance impact of texture creation and allocations of bounce buffers. Instead of recreating textures for every streaming request, a limited number of textures is kept in cache. If a cached texture is available, but does not match the required format, it is resized. Resizing to a smaller size does not free the memory, but is instant. Resizing to a larger size is instant as long as the underlying buffer is sufficiently large. In consquence, resizing is still significantly faster than creating textures from scratch, at the cost of lazily releasing memory.

Listing 5.3 shows how textures are rented from the *TexturePool*.

```
Texture2D Rent(ref TextureBlob request)
{
 Texture2D texture = Cache.Request();
 if (texture == null)
 {
  texture = new Texture2D(request.Width,
   request.Height,
   request.Format,
   request.MipmapCount,
   calculatemIpmaps: false);
 }
 else if (texture.width != request.Width
   || texture.height != request.Height
   || texture.format != request.Format)
 {
  texture.Resize(request.Width,
   request.Height,
   request.Format,
   request.MipmapCount > 0);
 }
 TextureFunctions.FillTextureUnsafe(texture, ref request);
 return texture;
}
```

Listing 5.3: Caching texture buffers.

Avoiding long stalls due to garbage collection of large allocated memory caused by unecessary temporary copies is important for real-time streaming. Visionary avoids such copies by using reinterpretation of arrays in unmanaged memory, and copies the memory as-is from layer to layer. Listing 5.4 shows how this is used to copy texture data allocation-free from unmanaged memory into the texture buffer on CPU and GPU.

```
FillTextureUnsafe(Texture2D texture, ref TextureBlob
   textureBlob)
{
 NativeArray<byte> native = texture.GetRawTextureData<byte>();
 textureBlob.TextureData.ToNativeArray().CopyTo(native);
 texture.Apply(false);
}
```

Listing 5.4: Allocation-free filling of texture buffers.

5.5.4 Caching

As explained in Section 4.2, streaming the geometry per raycast is infeasible. Multiple consecutive rays are likely to hit the same geometry. For ray casting, a limited amount of geometry is for this reason kept in memory at all times. Streaming is only performed when the geometry is not yet loaded. Listing 5.5 shows the pseudocode for raycasting against cached colliders.

```
RaycastResult Raycast(Ray ray, Entity entity){
  if(Cache.TryGetValue(entity, out var collider)){
   // Hit
  } else { // Miss
    if(Cache.Full) Cache.ReleaseOne(); // Free Memory
    Cache.Add(entity, collider = LoadCollider(entity));
  }
  return collider.CastRay(ray);
}
```

Listing 5.5: Pseudocode for ray casting against cached colliders.

FileStreams are cached using the same pattern.

CHAPTER 6

Results and Discussions

In this chapter, the performance of the streaming algorithm implemented in Visionary is presented and evaluated against PRo3D. The quality of the measurements using SED is evaluated and compared with fixed-rate subsampling. Lastly, the restrictions and limitations of Unity as engine for this thesis are presented and discussed. Solutions to overcome restrictions caused by Unity are presented, if they were implemented in Visionary.

6.1 Performance

This section lists the performance results of the preprocessor, streaming, rendering and measurement. The overall timings were measured with the C# *System.Diagnostics.Stopwatch* class, and all Unity-related timings, such as delta time between two frames, were measured with Unity's *UnityEngine.Time* class.

The test system is a Windows 10 Education 64-bit (10., Build 19044) PC, with an AMD Ryzen Threadripper 1900x 8-Core Processor, 64 GB RAM and an NVIDIA Gefroce GTX 1080 TI graphics card with 11 GB graphics memory. The system storage, from where the application is run, is a Samsung SSD 960 PRO 512 GB. The larger file storage where both input and output scenes are stored is a Samsung SSD 970 EVO Plus 2 TB.

PRo3D version PRo3D.Viewer.4.6.1-prerelease1 was used to evalute Visionary against.

6.1.1 Evaluation

The streaming and rendering performance was measured by performing a 60s recorded flyover. The minimum, maximum and average frame times were recorded along with the currently used memory and triangles rendered.

PRo3D uses a reactive rendering system, with no renders being made when nothing changes, while Visionary renders once per iteration of the main loop.

The fixed-rate sampling strategy from PRo3D was implemented in Visionary as well, and by implementing the functionality to reproject lines with different sampling strategies, it was possible to compare the two strategies within Visionary.

6.1.2 Scenes and References

Multiple different scenes were tested to evaluate different features:

- *Gale Crater* is the largest used dataset and the main test scene for all streaming and rendering performance related benchmark scenarios. Figure 6.1 shows the Gale Crater scene.
- *Victoria Crater* is the primary test scene for priority rendering and measuring across priority rendered datasets. Figure 6.2 shows the Victoria Crater scene.
- *Homeplate*, *Stereomosaic* and *Stimson1087* are smaller tests scenes with no outstanding features, but were added to expand the test set. Figure 6.3a shows the Homeplate scene, Figure 6.3b the Stereomosaic scene, and Figure 6.3c the Stimson1087 scene.

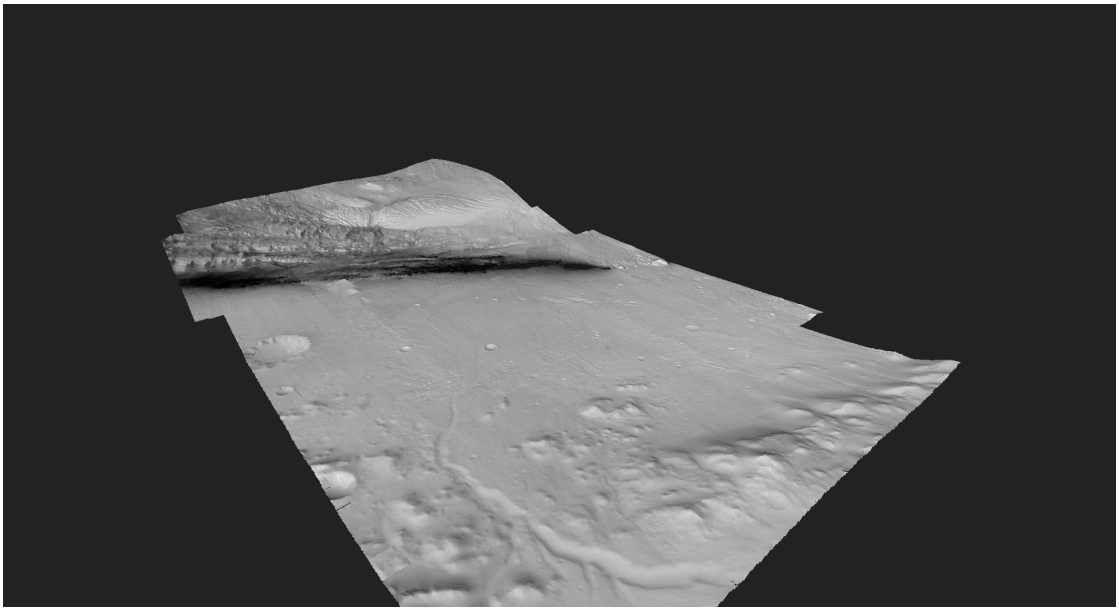


Figure 6.1: The Gale Crater scene has in total 221 GB. LOD 0 has 156 GB with 775.4 million triangles.

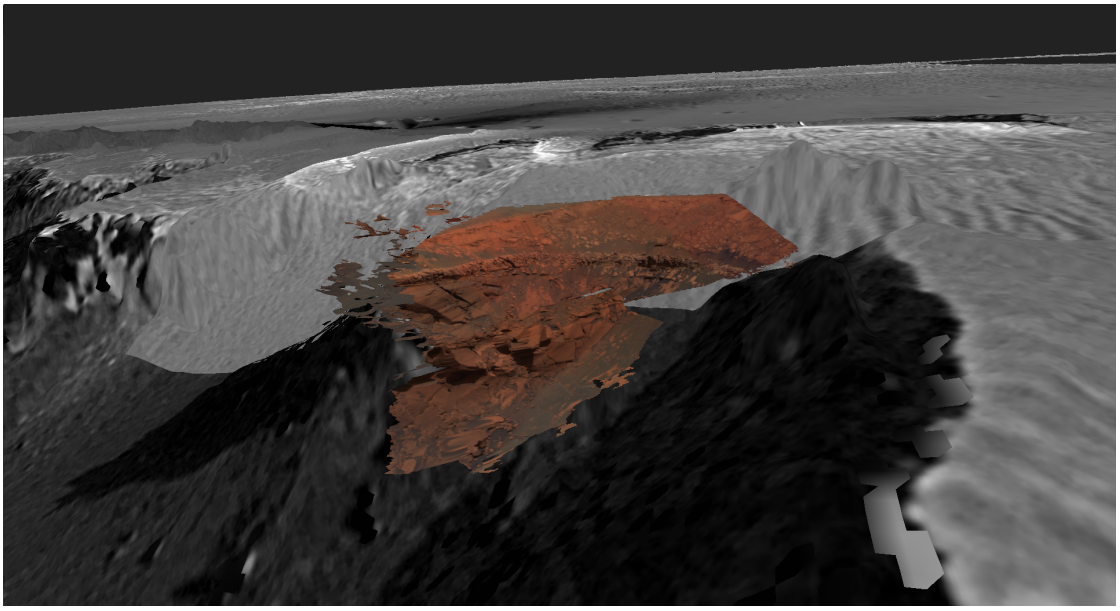
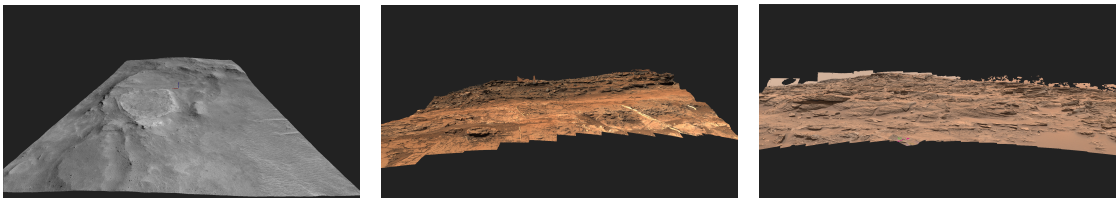


Figure 6.2: The Victoria Crater scene has in total 2.3 GB. LOD 0 has 1.3 GB and 17.7 million triangles. The colored mesh shows data of a higher-quality layer rendered with priority rendering.



(a) Homeplate. (b) Stereomosaic. (c) Stimson1087.

Figure 6.3: Overview of the smaller test scenes.

6.1.3 Preprocessing

In this section, the resulting file counts, file sizes and the time required for preprocessing are discussed.

Table 6.1 shows the size and count comparison of scenes before and after preprocessing. The size and count are shown as reported by WinDirStats [Win]. As can be seen in table 6.1, the Gale Crater scene is the largest and most representative scene. At large scale the file count is reduced to one third, which is expected as shown in Section 5.3.1. The better reduction in other scenes is not relevant for streaming, as those files are used for other functionality in PRo3D. No significant reduction in physical size is expected, and the Gale Crater scene shows this, as most of the data consists of compressed binary textures. The increase by 15% of physical file size for the Victoria Crater scene was

unexpected, and the difference is caused by the physical size of texture data in both formats due to different texture compression algorithms used. The decrease in file size for the scenes Stimson_1087 and Homeplate was not expected. The reduction for these scenes is achieved because the serialized colliders are smaller than the kd-tree of PRo3D. This is particularly interesting, as the AABBs serialized with Unity are 20x larger than required, but the serialized MeshCollider is smaller than the kd-tree format of PRo3D. The decrease in file size for the Stereomosaic scene is expected, as the original image files contain additional layers not supported and discarded by Visionary. Overall, a reduction in file count to one third can be expected, but no reduction in file size. If the additional texture layers were implemented in Visionary, the texture sizes would increase accordingly.

Scene	OPC		Renderable Blobs		%	
	Files	Size	Files	Size	Files	Size
Gale Crater	342 440	222.1 GB	106 185	219.6 GB	31%	99 %
Victoria Crater	2 687	2 GB	671	2.3 GB	25%	115 %
Homeplate	726	706.2 MB	161	381.5 MB	22%	54 %
Stimson_1087	1 594	1.6 GB	555	1.1 GB	35%	69 %
StereoMosaic	3 645	3.3 GB	740	762 MB	20%	22 %

Table 6.1: Comparison of file count and sizes before and after preprocessing.

Table 6.2 shows the preprocessing duration required for all scenes. The preprocessing is bottlenecked by the CPU processing power, as the slowest operations are the texture conversion, mipmap calculation and creation of the Mesh Collider, which all happen on the CPU. Most of these Unity operations are restricted to the main thread, which means that only a single thread is used. As these operations are restricted, they also are not allowed to be performed on background threads as discussed in Section 6.2.4.

Scene	Renderable Blobs			Preprocessing	
	HLODs	Files	Size	Duration	MB/s
Gale Crater	367	106 185	219.6 GB	72 min	52
Victoria Crater	4	671	2.3 GB	73 s	32
Homeplate	1	726	381.5 MB	16 s	23
Stimson_1087	2	555	1.1 GB	21 s	53
StereoMosaic	1	740	762 MB	13 s	58

Table 6.2: Time required for preprocessing.

6.1.4 Opening Scenes

In this section, the duration for the initial opening time of the scenes is compared to PRo3D. The opening time is measured as the time from clicking the load button in each tool until all OPCs are loaded.

Table 6.3 shows the performance comparison between PRo3D and Visionary when opening scenes. Scanning large file system hierarchies is slow in Visionary, but the directory tree is cached, and subsequent executions are faster. The smaller scenes open within 1s in both tools and the differences can be neglected. PRo3D has much more features, and a difference in the order of miliseconds is expected. However, the large Gale Crater scene opens 15x faster in Visionary. This takes 1 minute loading time in PRo3D, which is significant.

Scene	Pro3D	Visionary	
	Duration	Initial	Cached
Gale Crater	1 min	10 s	4 s
Victoria Crater	226 ms	40 ms	17 ms
Homeplate	-	30 ms	7 ms
Stimson_1087	230 ms	33 ms	19 ms
StereoMosaic	199 ms	32 ms	20 ms

Table 6.3: Opening scenes in PRo3D and Visionary. There is no difference when opening scenes in PRo3D for the first or subsequent times. In Visionary the directory tree is cached, which makes subsequent openings faster.

The Homeplate scene failed to load in all tested versions of PRo3D.

6.1.5 Rendering

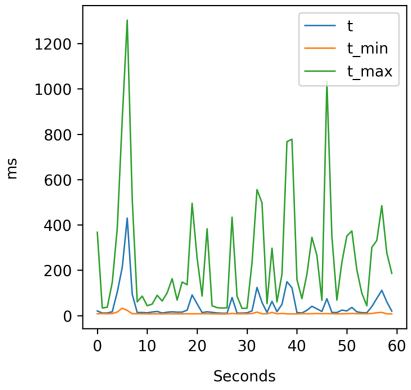
In this section, the streaming and rendering performance is shown and discussed. As streaming solutions are highly sensitive to the current camera position, camera view, the LOD selected, and the current status of the caches, it is difficult to create valid benchmarks. Kang et al. [KSH18] used a flyover with metrics aggregated per second to evaluate their streaming solution for terrain rendering with unlimited detail and resolution. A similar approach was used to evaluate the performance of Visionary. The streaming benchmarks were measured by recording a 60s flyover in-engine, measuring the statistics per frame, and evaluating the per second averages, minimum values and maximum values. A flyover is used to ensure changing from highest to lowest LOD and vice versa. To account for variance, all recordings are replayed five times. The minimum and maximum are global minima and maxima to ensure the whole bandwith is accurately represented.

It is important to know the size of LOD 0 to put the reduction through LOD selection into context. Table 6.4 lists the size and triangle count of LOD 0, which is the finest LOD, per scene.

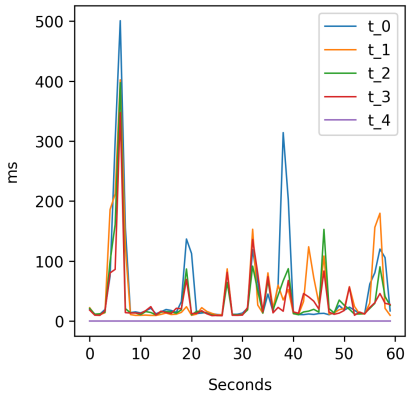
Scene	LOD 0	
	Size	Triangles
Gale Crater	156 GB	775.4 M
Victoria Crater	1.3 GB	17.7 M
Homeplate	171 MB	4.6 M
Stimson_1087	818 MB	3.7 M
StereoMosaic	558 MB	2.4 M

Table 6.4: Size and triangle count of LOD 0.

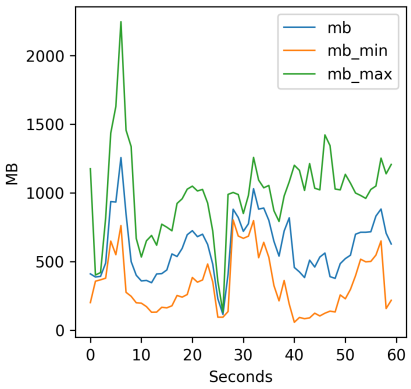
Figure 6.4 shows the performance metrics for a 60s flyover of the Gale Crater scene.



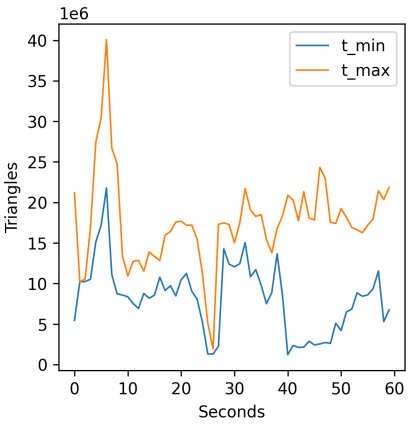
(a) Render time [ms].



(b) Render time / run [ms].



(c) Memory [MB].



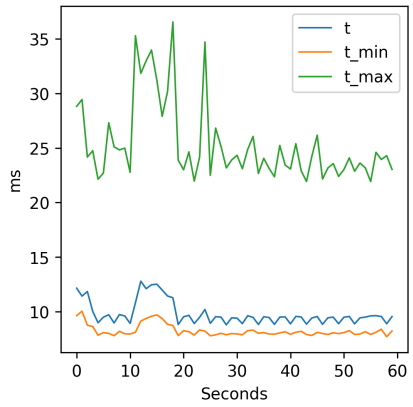
(d) Triangles [M].

Figure 6.4: Gale Crater with camera flyover. (a) shows the averaged timing over 5 runs. (b) shows the timings per run. Most of the time the rendering is interactive, but stuttering after large LOD changes is clearly visible. (c) shows the loaded memory over 5 runs. (d) shows the rendered triangles per frame (in millions).

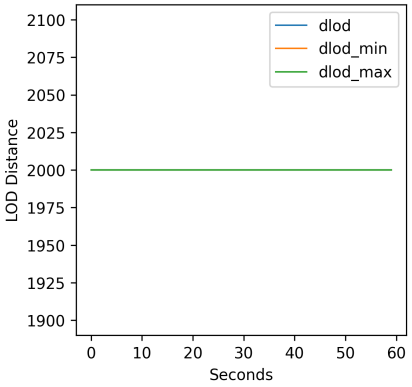
Noticable spikes in render time during large LOD changes are clearly visible. When a run is started, the camera is teleported back to the start position of the recording. This is visible around the 5 s mark, and the number of triangles rendered reflects this, and the used memory shows that the system tries to stream in 2 GB of data for the next run. The LOD selection reduces the number of triangles rendered to below 20% of the triangle count of LOD 0 at all times, with keeping it below 40% most of the time.

While the average render time is overall sufficient for interactive real-time rendering, the maximum render time clearly shows the stuttering during large LOD changes. After each time a replay of the recorded flyover has finished, the camera is teleported back to the start for the next replay. This causes a drastic spike during the initial first 10 seconds, when LOD nodes are changed. A similar drastic change happens during camera panning as seen between the 20 s and 30 s marks, when many LOD nodes become invisible and are culled. Future versions of Visionary need to restrict the number of LOD changes allowed per frame, as it is more desirable to teleport to the same LOD level, and then gradually improve the LOD, beginning with close hierarchies, and refining further away hierarchies later.

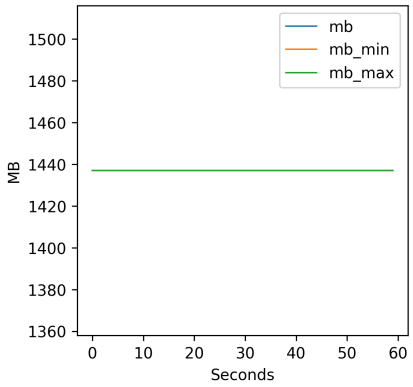
Figure 6.5 shows the performance when the camera is not moving. The camera is positioned close to the surface to ensure loading finer LODs. This test shows that the unrestricted LOD changes are the cause of the spikes seen in Figure 6.4. The constant values for LOD distance, loaded data, and rendered triangles indicate no system activity except Unity's own systems for the Universal Render Pipeline, Hybrid Renderer and Physics. While the render time is on average below 15 ms, the maximum is closer to 25 ms. This indicates a huge variability of the render times caused by the underlying render pipeline, even without other system activity. The constant values for memory and triangle count verify that there is no streaming activity. The rendered scene is shown and the LODs are highlighted using false-color rendering.



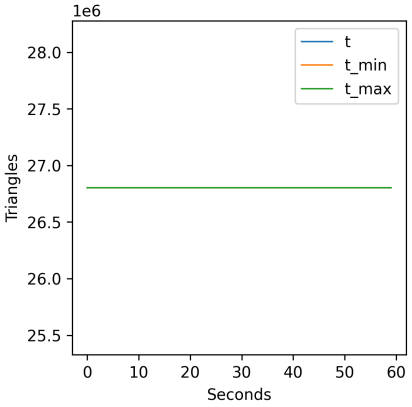
(a) Render time [ms].



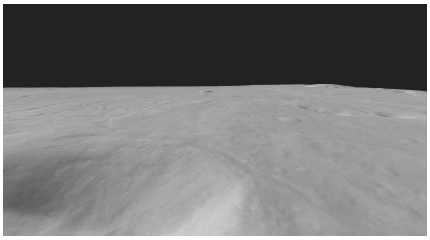
(b) LOD Distance.



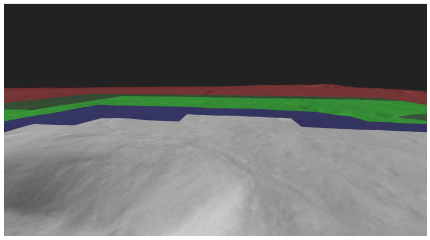
(c) Memory [MB].



(d) Triangles [M].



(e) Gale Crater with static camera.

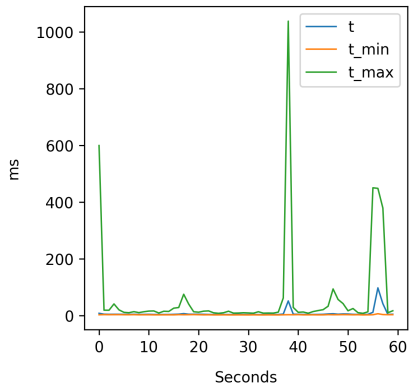


(f) LODs shown.

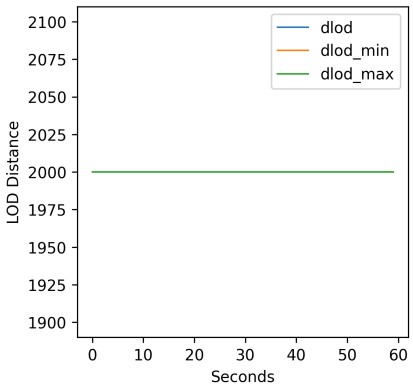
Figure 6.5: Gale Crater with static camera. (a) Altough there is no streaming activity, as indicated by (b), the render times vary strongly, but the average render time stays below 15 ms. (c, d) The constant values for MB loaded and triangles verify that there is no streaming system activity. (e) shows the rendered scene. (f) shows the rendered LODs.

The flyover benchmark was repeated for the Victoria Crater scene, to show the difference between smaller and larger scenes. Figure 6.6 shows the performance in the Victoria Crater scene. The camera zooms close to the surface between the 20 s and 30 s mark, which causes most LOD nodes to be culled, and zooms out to the highest LOD used.

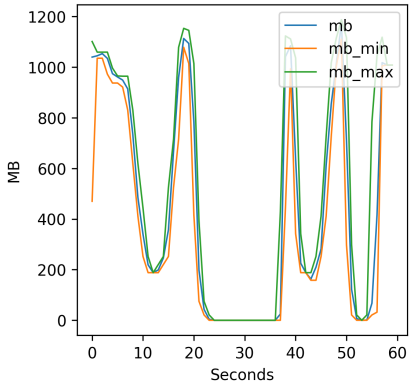
Loading all culled nodes back in almost instantly causes a sharp spike in render times around the 40 s mark. The cause of the sharp spike in render times is explained by zooming out and many nodes not being view-frustum culled anymore. When almost all nodes were culled during a close zoom in, the rapid change in culling cause a large spike when the culled nodes are loaded back in during zooming out. Future versions of Visionary need to keep these LOD nodes in-memory, as this benchmark shows that releasing them is unecessary and causes a lot of pressure on the CPU and disk requesting 1 GB of data to be loaded.



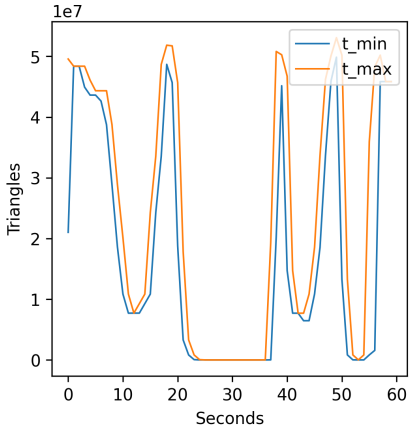
(a) Render time [ms].



(b) LOD Distance.



(c) Memory [MB].



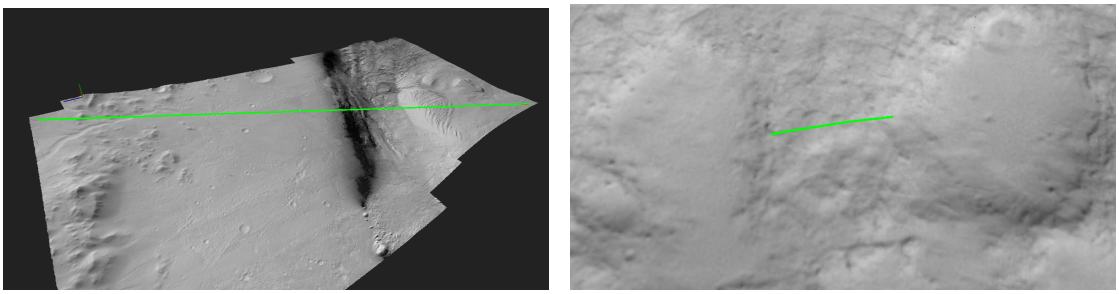
(d) Triangles [M].

Figure 6.6: (a) The average render time is well below 50 ms. (b) The reactive target frame-rate scheduler does not react to the single spike around the 40 s mark, caused by loading almost all LOD nodes back in when zooming out. (c) The sharp drop in memory and triangle counts (d) close to 0 MB used of memory clearly show that Visionary is too aggressively releasing currently not required nodes.

6.1.6 Measurements

In this section, the results of doing measurements using SED and ODR, are presented, and their performance is discussed.

The quality of results for fixed-rate subsampling (FRSS) and variable-rate subsampling (VRSS) are both highly sensitive to the number of samples in relation to the distance measured. The presented measurements and methods are for this reason evaluated both at large distance and at short distance. Figure 6.7 shows the test setups in Gale Crater for both longer and shorter measurements.



(a) A measurement across 58 km. (b) A measurement across 30 m.

Figure 6.7: (a) A long distance measurement across 58 km is performed to test the streaming and measurement performance and the quality of VRSS+SED over FRSS. (b) A shorter distance measurement of 30 m is used to evaluate performance at short distances.

The presented variable-rate subsampling with SED (VRSS+SED) was evaluated against VRSS without SED (VRSS), as well as the fixed-rate subsampling strategy used in PRo3D with 100 subsamples (FRSS 100). It also was evaluated against the fixed-rate subsampling strategy, using the same number of subsamples as VRSS+SED, to compare them as if the numbers of subsamples were known. Table 6.5 shows the results of these measurements.

Method	Samples	ms	ODR	m	Error	δ %
VRSS+SED	1025	3962	100 %	58707.65	0.00	0.00 %
VRSS	1003	274	0 %	58653.27	54.38	0.09 %
FRS	102	48	0 %	58255.98	451.67	0.77 %
FRS	1025	702	0 %	58295.32	412.33	0.70 %

Table 6.5: Quality of subsampling strategies for a 58 km distance.

As VRSS+SED guarantees that the result is a perfect cut through the terrain at 100 % ODR, it is used as a refernece point comparing all other methods. The relative error for all methods is below 1%, but the absolute error is a few hundred meters at a 58 km distance.

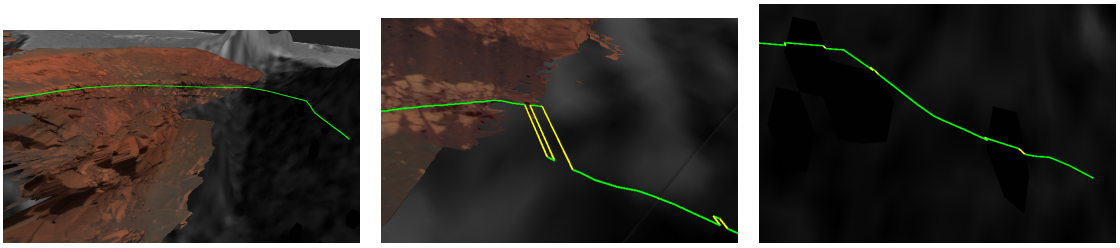
The choice of number of samples in relation to the distance is important for subsampling quality, as it defines the sampling frequency. Table 6.6 shows the result from the comparison of the methods at a shorter distance of 30 m.

Method	Samples	ms	ODR	m	Error	δ %
VRSS+SED	89	22 ms	100 %	30.41047	0.00	0.00 %
VRSS	87	6 ms	100 %	30.37083	0.04	0.13 %
FRS	102	13 ms	0 %	30.49761	0.09	0.29 %
FRS	89	10 ms	0 %	30.46393	0.05	0.18 %

Table 6.6: Quality of subsampling strategies for a 30 m distance.

The relative error remains similar and below 1 %. In consequence, the absolute error also is significantly lower.

Lastly, the measurement algorithm and ODR was evaluated in the Victoria Crater scene with priority rendering. Figure 6.8 shows a measurement with ODR 82 %. Cycling through the uncertain segments reveals multiple sources of uncertainty. Measuring across priority layers causes uncertainty, and uncertainty due to holes in the terrain, are revealed by this presented method.



(a) Measuring across priority layers

(b) Uncertainty because of a change of priority layer.

(c) Uncertainty because of holes.

Figure 6.8: (a) shows a measurement across priority layers at the Victoria Crater scene. The ODR is 82 %. (b) shows a segment that is uncertain because the priority layer was changed. (c) shows a segment that is uncertain because it passes a hole in the terrain.

6.2 Engine-Specific Restrictions

In this section, restrictions caused by using Unity as an engine are discussed, and solutions implemented to bypass them are listed.

6.2.1 Universal Rendering Pipeline

Unity's Built-In Render Pipeline does not support rendering DOTS-based Entities. The only rendering pipelines that support Entities are the High Definition Render Pipeline (HDRP) and the Universal Render Pipeline (URP) [Uniu]. Visionary uses the faster,

but less feature-rich one, which is URP. In addition, the Hybrid Renderer package is required [Unii].

The Hybrid Renderer does not support all the features of both rendering pipelines and DOTS. Unity lists the following limitations for version 0.51 [Unii]:

- Only a single DOTS World is supported, with support for multiple DOTS Worlds intended for a later version.
- Desktop OpenGL or GLES are currently not supported, but planned for a future version.
- Vulkan drivers are the only recommended graphics drivers for Android and Linux. The Vulkan driver is considered in a bad state on many older Android devices and Unity warns that it will never be upgraded.
- Hybrid Renderer is not yet validated on mobile or console platforms.
- Hybrid Renderer is not yet tested on XR devices, but XR support is intended for later versions.
- Ray-tracing (DXR) is currently not supported.

6.2.2 Duplicate Datatypes

The parallel development of DOTS and the UnityEngine core namespace causes type duplications. However, these duplicated types, which are not compatible with each other, cause the need of frequent conversion. The recommended pattern is to use the types *GameObject*, *MonoBehaviour* and *ScriptableObject* from the UnityEngine namespace for authoring, converting the data once during startup into ECS data, and then avoiding copying data back from the ECS [Unie]. This is not a feasible solution in many cases, as not all functionality is available in both worlds.

This parallel development also causes multiple implementations being available for the most basics types. These types often do not provide the same functionality and at worst are semantically contradicting. The types to model AABB in Unity are, e.g., :

- *UnityEngine.Bounds* [Tecn] models the AABB with a *Vector3 center* and *Vector3 extents*. The extents are half-extents on each axis. Size is equal to the extents per axis: $extents * 0.5 = size$.
- *Unity.Mathematics.AABB* [Unij] is defined as *UnityEngine.Bounds*, with the difference of using *float3* instead of *Vector3* and size being equal to half the extents: $size = extents * 2$.
- *Unity.Physics.Aabb* [Unik] uses *float3* as *Unity.Mathematics.AABB*, but its *Extents* property is defined as the *Size* of the AABB with $\overrightarrow{Extents} = \overrightarrow{Max} - \overrightarrow{Min}$ and *Aabb.Extents* * 0.5 = *AABB.Extents*.

Conversion between those types requires extra care. The semantic incompatibility of the types also makes it impossible to use reinterpretation as fast method of conversion, even when their memory layout was otherwise be equal. A temporary copy is required, which contradicts the recommended optimizations for cache-efficient usage of the memory hierarchy as explained in Section 5.5.

6.2.3 Blob Storage

Unity manages a special section of memory for BlobStorage. This limits the types that can be stored in *Unity.Entities.BlobAssetReference* blobs. Arrays stored in blobs must be of type *Unity.Entities.BlobArray*. However, the Unity API mostly only accepts arrays of type *Unity.Collections.NativeArray*. To avoid allocations for temporary copies between the two types, the BlobArray is reinterpreted allocation-free as NativeArray, which is shown in Listing 6.1.

```
NativeArray<T> ToNativeArray<T>(ref this BlobArray<T> ba)
   where T : struct
{
 NativeArray<T> arr = NativeArrayUnsafeUtility
 .ConvertExistingDataToNativeArray<T>(ba.GetUnsafePtr(),
    ba.Length, Allocator.Invalid);
 return arr;
}
```

Listing 6.1: Definition of the renderable blob. It contains both mesh and texture information.

6.2.4 Multithreading

Unity is strongly designed around a single threaded main loop model. Many operations, in particular creation and destruction of the two main types *UnityEngine.Object* and *UnityEngine.MonoBehaviour* and their derivatives, are strictly limited to be executed in the main thread. This limits multithreading in Unity, as all primary types such as *Mesh*, *Texture*, and *GameObject* are derived from them.

Access to managed C# tasks and threads is available, but no restricted type may be modified from threads other than the main thread. Long-running operations are executed multithreaded in Visionary by the use of tasks. Unity does not provide any support for context switching. Visionary uses UniTask [Unin], which provides support for context switching. Listing 6.2 shows the pseudocode for performing long-running operations in worker threads and continue with restricted operations on the main thread using UniTask.

```
var task = Task.WhenAll(requests.Select(r => Task.Run(async ()
   => await Load(r)));
async UniTask Load(...){
```

```
    // Load data from filesystem ...
    await UniTask.SwitchToMainThread();
    // Create, Update, Modify UnityEngine.Objects ...
    // Create, Update, Modify Unity.Entities ...
}
```

Listing 6.2: Pseudocode for using UniTask to perform a context switch to the main thread.

DOTS provides new ways to write highly efficient parallel code with its job system [Unir]. Unity describes their job system as follows [Tecd]:

> The Unity C# Job System allows users to write multithreaded code that interacts well with the rest of Unity and makes it easier to write correct code.

It is most efficiently used together with Unity's Burst compiler. The Burst compiler optimized code written within its restrictions for efficient parallelization. Unity describes their Burst compiler as follows [Unic]:

> Burst is a compiler that translates from IL/.NET bytecode to highly optimized native code using LLVM. It is released as a Unity package and integrated into Unity using the Unity Package Manager. ... Burst is primarily designed to work efficiently with the Job system.

While burst-compiled job code can be highly efficient, it imposes further restrictions compared to C# threads and tasks:

- Only blittable [gewa] data types are supported. NativeContainers to share data with jobs, or return results from them, are provided in the *Unity.Collections* namespace.
- Structural changes of ECS data cause sync points. Structural changes are creating entities, deleting entities, adding components to an entity, removing components from an entity, and changing the value of shared components [Syn].
- Memory allocated with the *Allocator.TempJob* must dispose the memory within four frames, which means jobs in general are limited to run for at most four frames [Tech].
- Accessing static data is not recommened and will be prevented in future versions [Tece].
- Jobs can only be created and completed in the main (control) thread [Tecj].
- Allocating memory within jobs is incredibly slow and prevents the Burst compiler from optimizing the job code [Tece].

All operations that are restricted to the main thread also are not allowed on jobs. The streaming performance could have been improved if the following operations were allowed to be performed on a background thread:

- Creation and initialization of mesh objects, and filling their all related buffers (vertex positions, normals, texture coordiantes, ...).
- Creation and initialization of texture objects, and filling all related buffers, as well as uploading the texture data to the GPU.

Regardless of the restrictions, the combination of the job system and the burst compiler are significant additons to Unity. Jobs are used in Visionary for both collider serialization as shown in Section 5.4.4 and to execute burst-compiled ray casts using jobs.

6.2.5 Mesh duplication

Unity does not allow sharing geometry between the *Mesh* used for rendering and the *MeshCollider* used for physics. This requires to duplicate the geometry for rendering and measurement.

6.2.6 Collider Creation

Unity simplifies created MeshColliders [Unil] in undocumented ways. An example for this is the combination of coplanar triangles to quads. While this simplification is safe and has no effect on the accuracy of the measurement algorithm, as shown in Chapter 4, it is unclear what other simplifications are performed. The API does not offer any control of the behaviour, e.g, turning simplification off, or querying the resulting mesh to validate it against the input mesh. Further research on the degree of simplification and the impact on the accuracy of measurements is required, before measurements using *Unity.Physics* can be considered trustworthy.

DOTS provided the necessary functionality for multithreaded jobs and mesh collider serialization. Without it, the presented solution would not have been possible because mesh creation would have been infeasibly slow.

6.2.7 Serialization

Visionary requires quick serialization of geometry in the form of *Unity.Physics.Collider* for the measurement algorithm presented in Chapter 4. This was implemented using the *Unity.Entities.Serialization.StreamBinaryReader* and *Unity.Entities.Serialization.-StreamBinaryWriter* introduced with early version of DOTS. In a late phase of the project, version 0.50 and 0.51 of *Unity.Entities* were released, and those classes were made *internal* [Unid] by Unity without further notice or explanation. Visionary uses a *public* implementation of the now *internal* classes, named *DeprecatedStreamBinaryReader* and *DeprecatedStreamBinaryWriter*. This breaking change in DOTS introduces a major risk

of the implementation to stop working in future version of DOTS, unless a replacement in the form of serializing DOTS Worlds becomes available. Otherwise, an alternative spatial acceleration structure must be implemented for the ray-casting-based measurements. One solution is the hierarchical KD-Tree structure as used in PRo3D [Prob].

6.2.8 Textures and Image Formats

The Unity editor supports more image formats than the runtime API. Both image formats used for OPC scenes, *Direct Draw Surface* (.dds) and *Tagged Image File Format* (.tif or .tiff) are not supported by the runtime. The editor supports .tiff images but not .dds images.

To solve the missing support for .dds images, the dual licensed Monogame *DdsLoader* was ported under the MIT license.

6.2.9 IL2CPP

The preprocessing and streaming performance was achieved by utilizing the faster ahead-of-time (AOT) IL2CPP scripting backend. While not all operations were faster in IL2CPP, the overall frame times were insufficient for interactive streaming with the same code under the Mono C# backend. The build time using IL2CPP is significantly longer, which slows down development. Even more important, the AOT framework restricts using libraries not compatible with an AOT framework (e.g., emitting code as commonly used during serialization). A full list of restrictions can be found in the official documentation [Teck].

This most of all makes it more difficult to rely on already available functionality. Magick.Net [Lemb] would have made it trivial to add support for over 100 image formats, including .tif and .dds, but the currently available version 7.14.5.0 is not compatible with the IL2CPP scripting backend.

6.2.10 Compatibility

The Unity scripting backend is based on Mono C# [Mona], while the engine itself is written in C++. The use of the multiplatform Mono supports targeting multiple platforms, at the cost of causing a large gap in functionality between the latest available version of .NET.

At the time of writing, the highest API Compatibility supported by Unity was .NET Standard 2.0, with the latest API compatiblity level being .NET Standard 2.1. The latest supported C# language version is C# 9.0 when version 2021.2 or higher of the Editor is used [Tecb]. However, not all language features are supported [Tecc]. The latest available version of C# at the moment is C# 10.0 in .NET 6.0 [Net].

Unity plans to bring the scripting backend from Mono closer to the state-of-the-art .NET ecosystem, and converging the JIT and AOT solutions were publicly announced [Unio]. At the moment, there is still a large gap between .NET and Unity's C# scripting backends.

DOTS is currently only supported in selected Unity versions. Version 0.51 is restricted to the Long-Term Support (LTS) version of the Unity Editor, with the lowest version supported being 2020.3.30 [Unif] and the 2021 LTS version being the latest supported version [Unig]. The highest available version is Unity 2022 [Teco].

6.2.11 DOTS

DOTS consits of many libraries. Some of them are released production ready, while others are still in development and released only experimentally. This thesis uses the Unity convention to refer to the version of DOTS-based on the version of the used core packge *com.unity.entities*. Visionary was implemented in 0.17 and upgraded later to 0.51 when it became available in the project [Unif].

6.2.12 AABB colliders

The size of the serialized *Box Colliders* in Unity is 492 bytes. Compared to the 24 bytes required to represent an AABB in single-precision, it is more than 20x larger. While it is small enough to be kept in memory at all times, it is considerably larger and thus extremly inefficient in terms of disk space, memory and performance.

6.2.13 Bounding Volume

The only bounding volume Unity supports as bounding volume for the render bounds used during view-frustum culling are AABB. Even if another type of bounding volume was available for the data, it would have to be converted and duplicated. Other geometry types are supported for physics simulations.

6.2.14 View-Frustum Culling

Culling is the procedure to filter, or cull, elements before further processing. View-frustum culling against the AABB was implemented to cull hierarchies and LOD nodes to avoid streaming invisible objects. The Unity render pipeline performs another view-frustum culling pass before rendering. This means rendered objects are view-frustum culled twice, as there is no way to share the culling status with Unity's render pipeline.

6.2.15 File System Access

Opening and closing file streams during streaming quickly became a bottleneck. Importantly, the safety checks and garbage created by long file paths caused it to be inefficient. Visionary caches file streams, as explained in Section 5.5.4. This is only viable as long as the data remains static.

The restrictions of multi-threading in Unity are most of all hindering long-running operations. The job system and the *AsyncReadManager* are the only way Unity supports multithreaded file system access, and both are highly restricted compared to .NET file system access.

CHAPTER 7

Conclusion and Future Work

In this thesis, we presented a streaming algorithm for out-of-core rendering of large-scale terrain in general discrete 3D-mesh representation on commodity hardware. Furthermore, we presented an improved polyline-based measurement algorithm, using variable-rate subsampling and Shared Edge Detection (VRSS+SED). Such polyline-based measurement tools are used in state-of-the-art geology software such as PRo3D [Prob] and VRGS 3.0 [Vrg]. Lastly, we presented a simple but to our knowledge novel uncertainty metric, On-Data Ratio (ODR). ODR allows raising awareness about the uncertainty caused by holes in the terrain or early termination during subsampling using such measurment algorithms.

For the streaming algorithm we combined the methods of asynchronous out-of-core rendering as described by Varadhan and Manocha [VM02] with the streaming method for HLODs as described by Guthe and Klein [GK04]. We adapted the methods by allowing for a higher degree of parallelization by using multiple HLODs (M-HLOD). We evaluated our method against PRo3D, a state-of-the-art visualization tool for planetary geology, and achieve a 15x faster loading time for the largest tested scene. Our streaming algorithm manages to stream scenes with 775 M triangles and 156 GB on their finest LOD, and a total size of 222 GB, at interactive frame-rates and on commodity hardware. The algorithm uses a memory and target frame-rate scheduler to ensure efficient streaming. However, the algorithm releases unused LOD nodes unecessarily aggressively, which led to spikes in rendering time after large LOD changes. We would like to address this in future versions.

The presented VRSS+SED measurement algorithm results in exact polyline segments when neighbouring primitives are hit. It achieves this by intersecting the shared edge with the ray casting plane to find an exact midpoint on the shared edge. In contrast to a fixed-rate subsampling (FRSS) strategy, as used in PRo3D [TO15], VRSS+SED terminates earlier, and is more precise at the same number of samples. In particular at large

distances, VRSS+SED outperformed FRSS, as it is not dependent on an appropriately selected number of samples.

The presented ODR is valuable as it raises awareness about uncertainty, where existing methods did not consider any uncertainty. ODR proved to be especially valuable in combination with SED, as the exact results found by SED allow classifying resulting polyline segments as certain, where the fixed number of subsamples used by FRSS does not allow any conclusion about the uncertainty.

The presented algorithms were implemented in a prototype, called Visionary, using the Unity game engine and its Data-Oriented Techstack (DOTS). This showed both the benefits of reusing functionality when using a higher-level engine and the restrictions imposed by it. These restrictions and solutions to overcome them were discussed.

Minor contributions of this thesis are:

- The presented two-phase measurement algorithm, supports streamed large-scale and multi-layered terrain. It achieves efficient measurement on streamed terrain by performing the ray casting in local space.
- The presented *Renderable Blob* file format stores the terrain data in contrast to the *Ordered Point Cloud (OPC)* file format as used by Pro3D [Prob] in CPU- and GPU-cache-efficent memory layout.
- The presented algorithms were implemented in a prototype, called Visionary, using the Unity Engine. This proves the viability of Unity as engine for digital terrain surveying software for out-of-core rendered large-scale terrains and reveals the benefits and drawbacks using it for such a use case. The benefits and drawbacks are discussed and solutions to work around the restrictions are presented.

7.1 Future Work

Potential future works includes:

- The presented streaming algorithm was inferior in situations of large LOD changes. We would like to address the spikes in render times due to large LOD changes in future versions and add smooth blending between LODs.
- Our On-Data Ratio considers only a few selected sources of uncertainty, but others could be considered as well. Most of all, the resolution of the terrain reconstruction is a parameter usually available in GIS terrain and worth considering. Future work could also extend the visualization methods of ODR by using false-color and heat maps to visualize the sources of uncertainty better.
- Schütz [Sch21] suggested the combination of discrete LOD and continuous LOD for efficient rendering of point clouds. We believe this concept would be a good fit

for the presented streaming algorithm as well. It could be a solution to increase the rendering performance while performing less of the much slower streaming operations.

- DOTS is only available as an experimental release and future releases could improve the performance, in particular if support for runtime created objects via serialization of DOTS worlds is added.
- The accuracy of the created MeshColliders in Unity and the impact of the performed mesh simplification requires further research before any measurement relying on it can be trusted.
- Unreal Technologies presented Nanite [KSW21], a highly performant streaming solution for static geometry. Future research could explore wether it is a suitable visualization component for digital terrain surveying software.

Bibliography

[3dg] *3D-GIS | ArcGIS Desktop*. http://desktop.arcgis.com/de/3d/. (Visited on 02/02/2022).

[AM09] Yacine Amara and Xavier Marsault. „A GPU Tile-Load-Map architecture for terrain rendering: theory and applications“. In: *The Visual Computer* 25.8 (2009), pp. 805–824.

[AMHH19] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. eng. 3rd ed.. Boca Raton, FL: Taylor and Francis, an imprint of A K Peters/CRC Press, 2019. ISBN: 1498785638.

[AP94] PETER ASTHEIMER and LUISE PöCHE. „LEVEL-OF-DETAIL GENERATION“. In: *Virtual Reality Software And Technology-Proceedings Of The Vrst'94 Conference*. World Scientific. 1994, p. 299.

[Bar+16] Robert Barnes et al. „PRo3D: A Tool for Geological Analysis of Martian Rover-Derived Digital Outcrop Models“. In: *Proceedings of the 2nd Virtual Geoscience Conference*. Ed. by Simon J Buckley. VGC '16. 2016, p. 49.

[Bar+18] Robert Barnes et al. „Geological Analysis of Martian Rover-Derived Digital Outcrop Models Using the 3-D Visualization Tool, Planetary Robotics 3-D Viewer—PRo3D“. In: *Earth and Space Science* 5.7 (2018), pp. 285–307. DOI: https://doi.org/10.1002/2018EA000374. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1002/2018EA000374. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2018EA000374.

[Bay+17] S Bayburt et al. „Geometric accuracy analysis of WorldDEM in relation to AW3D30, SRTM and ASTER GDEM2“. In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences-ISPRS Archives 42 (2017), Nr. 1W1* 42.1W1 (2017), pp. 211–217.

[Ben75] Jon Louis Bentley. „Multidimensional binary search trees used for associative searching“. In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[BHH15] Jiří Bittner, Michal Hapala, and Vlastimil Havran. „Incremental BVH construction for ray tracing“. In: *Computers & Graphics* 47 (2015), pp. 135–144. ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2014.12.001. URL: https://www.sciencedirect.com/science/article/pii/S0097849314001435.

[Bri19] Brian Tong. *How Disney's The Lion King was made in VR*. May 31, 2019. URL: https://www.youtube.com/watch?v=hIjicwrpgNM (visited on 02/08/2022).

[Buc+19] Simon J. Buckley et al. „LIME: Software for 3-D Visualization, Interpretation, and Communication of Virtual Geoscience Models“. In: *Geosphere* 15.1 (Jan. 2019), pp. 222–235. ISSN: 1553-040X. DOI: 10.1130/GES02002.1. eprint: https://pubs.geoscienceworld.org/gsa/geosphere/article-pdf/15/1/222/4619098/222.pdf. URL: https://doi.org/10.1130/GES02002.1.

[Can m] Mike Beauregard from Nunavut Canada. *Geologists on a Conference Field Trip Are Unraveling a Sequence of Avalanches in Ocean Bottom Sediments at a Mid-Continent Precambrian Outcrop Outside Wawa Ontario, Canada.* 10 May 2017, 11:51. URL: https://commons.wikimedia.org/wiki/File:Geoscience_crowd_sourcing_(37633979015).jpg (visited on 07/05/2022).

[Cla76] James H Clark. „Hierarchical geometric models for visible surface algorithms“. In: *Communications of the ACM* 19.10 (1976), pp. 547–554.

[CLS08] Xingquan Cai, Jinhong Li, and Zhitong Su. „Large-scale terrain rendering using strip masks of terrain blocks“. In: *2008 7th World Congress on Intelligent Control and Automation*. IEEE. 2008, pp. 1768–1772.

[Cra+09] Cyril Crassin et al. „Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering“. In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 15–22.

[dav] davidbritch. *The Model-View-ViewModel Pattern - Xamarin*. URL: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm (visited on 07/18/2022).

[dbo] dotnet bot. *Unsafe Code - C# Language Specification*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code (visited on 06/30/2022).

[Del+21] Thomas Deliot et al. *Experimenting with Concurrent Binary Trees: Large Scale Terrain Rendering*. 2021. URL: https://advances.realtimerendering.com/s2021/ (visited on 02/17/2022).

[Del+34] Boris Delaunay et al. „Sur la sphere vide“. In: *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7.793-800 (1934), pp. 1–2.

[Don20] Yangzi Dong. *Scalable Real-Time Rendering for Extremely Complex 3D Environments Using Multiple GPUs*. Rochester Institute of Technology, 2020.

[Duc+97] Mark Duchaineau et al. „ROAMing terrain: Real-time optimally adapting meshes“. In: *Proceedings. Visualization'97 (Cat. No. 97CB36155)*. IEEE. 1997, pp. 81–88.

[Dup20] Jonathan Dupuy. „Concurrent Binary Trees (with application to longest edge bisection)“. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), pp. 1–20.

[EBN13] Jan Elseberg, Dorit Borrmann, and Andreas Nüchter. „One billion points in the cloud – an octree for efficient processing of 3D laser scans“. In: *ISPRS Journal of Photogrammetry and Remote Sensing* 76 (2013). Terrestrial 3D modelling, pp. 76–88. ISSN: 0924-2716. DOI: https://doi.org/10.1016/j.isprsjprs.2012.10.004. URL: https://www.sciencedirect.com/science/article/pii/S0924271612001888.

[EM98] Carl Erikson and Dinesh Manocha. *Simplification Culling of Static and Dynamic Scene Graphs*. Tech. rep. USA, 1998.

[EMBI01] Carl Erikson, Dinesh Manocha, and William V Baxter III. „HLODs for faster display of large static and dynamic environments“. In: *Proceedings of the 2001 symposium on Interactive 3D graphics*. 2001, pp. 111–120.

[Eri04] Christer Ericson. *Real-time collision detection*. Crc Press, 2004.

[ETT15] Tom Erez, Yuval Tassa, and Emanuel Todorov. „Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX“. In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 4397–4404. DOI: 10.1109/ICRA.2015.7139807.

[FJ17] Margarita N Favorskaya and Lakhmi C Jain. *Handbook on Advances in Remote Sensing and Geographic Information Systems : Paradigms and Applications in Forest Landscape Modeling*. eng. Intelligent Systems Reference Library 122. Cham: Springer International Publishing Imprint: Springer, 2017. ISBN: 3319523082. URL: 10.1007/978-3-319-52308-8.

[GD21] Baiqiang Gan and Qiuping Dong. „An improved optimal algorithm for collision detection of hybrid hierarchical bounding box“. In: *Evolutionary Intelligence* (2021), pp. 1–13.

[gewa] gewarren. *Blittable and Non-Blittable Types - .NET Framework*. URL: https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types (visited on 07/19/2022).

[gewb] gewarren. *Fundamentals of Garbage Collection*. URL: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (visited on 07/16/2022).

[Gil+21a] Christina Gillmann et al. „Ten Open Challenges in Medical Visualization“. In: *IEEE Computer Graphics and Applications* 41.5 (2021), pp. 7–15. DOI: 10.1109/MCG.2021.3094858.

[Gil+21b] Christina Gillmann et al. „Uncertainty-aware Visualization in Medical Imaging-A Survey“. In: *Computer Graphics Forum*. Vol. 40. 3. Wiley Online Library. 2021, pp. 665–689.

[GK04] M Guthe and R Klein. „Streaming HLODs: an out-of-core viewer for network visualization of huge polygon models“. eng. In: *Computers & graphics* 28.1 (2004), pp. 43–50. ISSN: 0097-8493.

[GKY08] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. „Technical Strategies for Massive Model Visualization“. In: *Proceedings of the 2008 ACM Symposium on Solid and Physical Modeling*. SPM '08. Stony Brook, New York: Association for Computing Machinery, 2008, 405–415. ISBN: 9781605581064. DOI: 10.1145/1364901.1364960. URL: https://doi.org/10.1145/1364901.1364960.

[GM05] Enrico Gobbetti and Fabio Marton. „Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms“. In: *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. Los Angeles, California: Association for Computing Machinery, 2005, 878–885. ISBN: 9781450378253. DOI: 10.1145/1186822.1073277. URL: https://doi.org/10.1145/1186822.1073277.

[Gpu] *Boosting Data Ingest Throughput with GPUDirect Storage and RAPIDS cuDF*. NVIDIA Technical Blog. URL: https://developer.nvidia.com/blog/boosting-data-ingest-throughput-with-gpudirect-storage-and-rapids-cudf/ (visited on 07/19/2022).

[GRA22] GRASS Development Team. *Geographic Resources Analysis Support System (GRASS GIS) Software*. Open Source Geospatial Foundation. USA, 2022. URL: https://grass.osgeo.org.

[GRF18] Alejandro Graciano, Antonio J Rueda, and Francisco R Feito. „Real-time visualization of 3D terrains and subsurface geological structures“. In: *Advances in Engineering Software* 115 (2018), pp. 314–326.

[GW07] Markus Giegl and Michael Wimmer. „Unpopping: Solving the image-space blend problem for smooth discrete LOD transitions“. In: *Computer Graphics Forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 46–49.

[Hai+18] Azzam Haidar et al. „Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers“. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2018, pp. 603–613.

[Hav00] Vlastimil Havran. „Heuristic ray shooting algorithms“. PhD thesis. Ph. d. thesis, Department of Computer Science and Engineering, Faculty of . . ., 2000.

[HK16] Thomas Hilfert and Markus König. „Low-cost virtual reality environment for engineering and construction“. In: *Visualization in Engineering* 4.1 (2016), pp. 1–18.

[Hor+07] Daniel Reiter Horn et al. „Interactive K-d Tree GPU Raytracing“. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. Seattle, Washington: Association for Computing Machinery, 2007, 167–174. ISBN: 9781595936288. DOI: 10.1145/1230100.1230129. URL: https://doi.org/10.1145/1230100.1230129.

[Hua+20] Lila Huang et al. „OctSqueeze: Octree-Structured Entropy Model for LiDAR Compression“. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.

[Hum+12] Johannes Hummel et al. „An evaluation of open source physics engines for use in virtual reality assembly simulations“. In: *International Symposium on Visual Computing*. Springer. 2012, pp. 346–357.

[Hus+] Noor Asma Husain et al. „AN EFFICIENT AND SIMPLE ADAPTIVE SUBDIVISION SURFACES METHOD FOR HANDLING CRACKS IN TRIANGULAR MESHES“. In: ().

[Iee] *IEEE SA - IEEE 754-2019*. IEEE Standards Association. URL: https://standards.ieee.org/ieee/754/6210/ (visited on 07/16/2022).

[IEv] IEvangelist. *Task Parallel Library (TPL)*. URL: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl (visited on 06/30/2022).

[Jia+20] Shiyu Jia et al. „Using pseudo voxel octree to accelerate collision between cutting tool and deformable objects modeled as linked voxels“. In: *The Visual Computer* 36.5 (2020), pp. 1017–1028.

[JSW22] Dylan Jude, Jayanarayanan Sitaraman, and Andrew Wissink. „An octree-based, cartesian navier–stokes solver for modern cluster architectures“. In: *The Journal of Supercomputing* (2022), pp. 1–32.

[Jul+18] Arthur Juliani et al. *Unity: A General Platform for Intelligent Agents*. 2018. DOI: 10.48550/ARXIV.1809.02627. URL: https://arxiv.org/abs/1809.02627.

[Kaw] Yoshifumi Kawai. *MessagePack for C# (.NET, .NET Core, Unity, Xamarin)*. URL: https://github.com/neuecc/MessagePack-CSharp (visited on 06/07/2022).

[Klo+98] J.T. Klosowski et al. „Efficient collision detection using bounding volume hierarchies of k-DOPs“. In: *IEEE Transactions on Visualization and Computer Graphics* 4.1 (1998), pp. 21–36. DOI: 10.1109/2945.675649.

[Klu+18] Christoph Klug et al. „Measurement Uncertainty Analysis of a Robotic Total Station Simulation“. In: *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*. 2018, pp. 2576–2582. DOI: 10.1109/IECON.2018.8592768.

[KS16] Swadhesh Kumar and P K Singh. „An overview of modern cache memory and performance analysis of replacement policies“. In: *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. 2016, pp. 210–214. DOI: 10.1109/ICETECH.2016.7569243.

[KSH18] HyeongYeop Kang, Yeram Sim, and JungHyun Han. „Terrain rendering with unlimited detail and resolution“. In: *Graphical Models* 97 (2018), pp. 64–79. ISSN: 1524-0703. DOI: https://doi.org/10.1016/j.gmod.2018.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1524070318300109.

[KSW21] Brian Karis, Rune Stubbe, and Graham Wihlidal. *Nanite: A Deep Dive*. 2021. URL: https://advances.realtimerendering.com/s2021/ (visited on 02/17/2022).

[Laz+21] Lazaros Lazaridis et al. „Hitboxes: A Survey About Collision Detection in Video Games“. In: *International Conference on Human-Computer Interaction*. Springer. 2021, pp. 314–326.

[LC10] Peter Lindstrom and Jonathan Cohen. „On-the-fly decompression and rendering of multiresolution terrain“. In: Feb. 2010, pp. 65–73. DOI: 10.1145/1730804.1730815.

[Lema] Dirk Lemstra. *The .NET Library for ImageMagick: Magick.NET*. URL: https://github.com/dlemstra/Magick.NET (visited on 06/30/2022).

[Lemb] Dirk Lemstra. *The .NET Library for ImageMagick: Magick.NET*. URL: https://github.com/dlemstra/Magick.NET (visited on 06/19/2022).

[Lin+96] Peter Lindstrom et al. „Real-time, continuous level of detail rendering of height fields“. In: *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 1996, pp. 109–118.

[LMG10] Raphael Lerbour, Jean-Eudes Marvie, and pascal Gautron. „Adaptive Real-Time Rendering of Planetary Terrains“. In: *The 18th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2010*. Vaclav Skala, Feb. 2010. ISBN: 978-80-86943-93-0.

[lora] lorihollasch. *DirectDraw Surfaces - Windows Drivers*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/display/directdraw-surfaces (visited on 06/30/2022).

[lorb] lorihollasch. *Paging Video Memory Resources - Windows Drivers*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/display/paging-video-memory-resources (visited on 07/15/2022).

[LP02] P. Lindstrom and V. Pascucci. „Terrain simplification simplified: a general framework for view-dependent out-of-core visualization“. In: *IEEE Transactions on Visualization and Computer Graphics* 8.3 (2002), pp. 239–254. DOI: 10.1109/TVCG.2002.1021577.

[LS19] Eun-Seok Lee and Byeong-Seok Shin. „Hardware-Based Adaptive Terrain Mesh Using Temporal Coherence for Real-Time Landscape Visualization“. In: *Sustainability* 11.7 (2019), p. 2137.

[Lue+02] David Luebke et al. *Level of Detail for 3D Graphics*. eng. 1st edition. Morgan Kaufmann, 2002. ISBN: 0080510116.

[Mat+14] Ruzinoor Che Mat et al. „Using game engine for 3D terrain visualisation of GIS data: A review“. In: *IOP Conference Series: Earth and Environmental Science*. Vol. 20. 1. IOP Publishing. 2014, p. 012037.

[Mea82] Donald Meagher. „Geometric modeling using octree encoding“. In: *Computer graphics and image processing* 19.2 (1982), pp. 129–147.

[Med+22] Marina L. Medeiros et al. „The Potential of VR-based Tactical Resource Planning on Spatial Data“. In: *2022 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 2022, pp. 176–185. DOI: 10.1109/VR51125.2022.00036.

[Mei+21] Daniel Meister et al. „A survey on bounding volume hierarchies for ray tracing“. In: *Computer Graphics Forum*. Vol. 40. 2. Wiley Online Library. 2021, pp. 683–712.

[Mes] *MessagePack: It's like JSON. but Fast and Small.* URL: https://msgpack.org/ (visited on 06/13/2022).

[Mic] *The .NET Compiler Platform*. .NET Platform. URL: https://github.com/dotnet/roslyn (visited on 06/30/2022).

[Mona] *Home | Mono*. URL: https://www.mono-project.com/ (visited on 06/19/2022).

[Monb] *MonoGame/TextureImporter.Cs at Develop · MonoGame/MonoGame*. GitHub. URL: https://github.com/MonoGame/MonoGame (visited on 06/30/2022).

[NA11] Nathan Whitehead and Alex Fit-Florea. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. 2011. URL: Precision&Performance:FloatingPointandIEEE754ComplianceforNVIDIAGPUs (visited on 06/07/2022).

[Naf+21] Samuel Naffziger et al. „Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families: Industrial product“. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2021, pp. 57–70.

[Nan] *Nanite Virtualized Geometry*. URL: https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/Nanite (visited on 02/17/2022).

[NAS] NASA's Navigation and Ancillary Information Facility. *Spice Kernel*. URL: https://naif.jpl.nasa.gov/naif/ (visited on 06/28/2022).

[Nat+13] Mattia Natali et al. „Modeling Terrains and Subsurface Geology". In: *Eurographics 2013 - State of the Art Reports*. Ed. by M. Sbert and L. Szirmay-Kalos. The Eurographics Association, 2013. DOI: 10.2312/conf/EG2013/stars/155-173.

[Nes+20] Paul Ryan Nesbit et al. „Visualization and sharing of 3D digital outcrop models to promote open science". In: *GSA Today* 30.6 (2020), pp. 4–10.

[Net] *Download .NET 6.0 (Linux, macOS, and Windows)*. Microsoft. URL: https://dotnet.microsoft.com/en-us/download/dotnet/6.0 (visited on 06/19/2022).

[Ort+10] Thomas Ortner et al. „Towards True Underground Infrastructure Surface Documentation". In: *Proceedings of CORP 2010* (2010), pp. 783–792. URL: https://www.vrvis.at/publications/PB-VRVis-2010-008.

[Ort+11] T Ortner et al. „Hochauflösende Flächendeckende Dokumentation von Tunneloberflächen". In: *in Proceedings of Symposium und Fachmesse für Angewandte Geoinformatik (AGIT 2011)* (2011), pp. 914–923.

[Paa] Gerhard Paar. *SOLUTIONS AND RESULTS | Www.Provide-Space.Eu*. URL: https://provide-space.eu/solutions-and-results/ (visited on 06/28/2022).

[Paa+15a] Gerhard Paar et al. „EMBEDDING SENSOR VISUALIZATION IN MARTIAN TERRAIN RECONSTRUCTIONS". In: *ASTRA* (Jan. 2015).

[Paa+15b] Gerhard Paar et al. „PRoViDE: Planetary robotics vision data processing and fusion". In: *European Planetary Science Congress*. 2015, EPSC2015–345.

[Pey+09] Adrien Peytavie et al. „Arches: a Framework for Modeling Complex Terrains". In: *Computer Graphics Forum* 28 (Apr. 2009), pp. 457 –467. DOI: 10.1111/j.1467-8659.2009.01385.x.

[Proa] *PRo3D License*. pro3d-space. URL: https://github.com/pro3d-space/PRo3D/blob/661f415a3761ff00c9b62c888b325ef9aa346192/LICENSE (visited on 02/17/2022).

[Prob] *PRo3D Viewer*. URL: https://pro3d.space/ (visited on 02/08/2022).

[QLX21] Zizheng Que, Guo Lu, and Dong Xu. „VoxelContext-Net: An Octree Based Framework for Point Cloud Compression". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 6042–6051.

[RBK20] John Ravi, Suren Byna, and Quincey Koziol. „GPU direct i/o with HDF5". In: *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*. IEEE. 2020, pp. 28–33.

[Red98] M Reddy. „Specification and evaluation of level of detail selection criteria“. eng. In: *Virtual reality : the journal of the Virtual Reality Society* 3.2 (1998), pp. 132–143. ISSN: 1359-4338.

[Rho+03] Philip J Rhodes et al. „Uncertainty visualization methods in isosurface rendering.“ In: *Eurographics (Short Presentations)*. 2003.

[Riz+15] Albert Rizzo et al. „Virtual reality exposure for PTSD due to military combat and terrorist attacks“. In: *Journal of Contemporary Psychotherapy* 45.4 (2015), pp. 255–264.

[RL00] Szymon Rusinkiewicz and Marc Levoy. „QSplat: A Multiresolution Point Rendering System for Large Meshes“. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, 343–352. ISBN: 1581132085. DOI: 10.1145/344779.344940. URL: https://doi.org/10.1145/344779.344940.

[RL01] Szymon Rusinkiewicz and Marc Levoy. „Streaming QSplat: a viewer for networked visualization of large, dense models“. eng. In: *Proceedings of the 2001 symposium on interactive 3d graphics*. I3D '01. ACM, 2001, pp. 63–68. ISBN: 1581132921.

[Ron+20] Guodong Rong et al. „Lgsvl simulator: A high fidelity simulator for autonomous driving“. In: *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*. IEEE. 2020, pp. 1–6.

[Sam84] Hanan Samet. „The quadtree and related hierarchical data structures“. In: *ACM Computing Surveys (CSUR)* 16.2 (1984), pp. 187–260.

[Sch16] Markus Schütz. „Potree: Rendering Large Point Clouds in Web Browsers“. MA thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Institute of Computer Graphics and Algorithms, Vienna University of Technology, Sept. 2016. URL: https://www.cg.tuwien.ac.at/research/publications/2016/SCHUETZ-2016-POT/.

[Sch21] Markus Schütz. *Interactive exploration of point clouds*. eng. Wien, 2021.

[Sch69] Robert Schumacher. *Study for applying computer-generated images to visual simulation*. Vol. 69. 14. Air Force Human Resources Laboratory, Air Force Systems Command, 1969.

[Shi90] Peter Shirley. „A ray tracing method for illumination calculation in diffuse-specular scenes“. In: *Proceedings of Graphics Interface'90*. 1990, pp. 205–212.

[SKW19] Markus Schütz, Katharina Krösl, and Michael Wimmer. „Real-time continuous level of detail rendering of point clouds“. In: *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. IEEE. 2019, pp. 103–110.

[SOW20] Markus Schütz, Stefan Ohrhallinger, and Michael Wimmer. „Fast Out-of-Core Octree Generation for Massive Point Clouds“. In: *Computer Graphics Forum*. Vol. 39. 7. Wiley Online Library. 2020, pp. 155–167.

[Ste+14] Markus Steinberger et al. „On-the-fly generation and rendering of infinite cities on the GPU“. In: *Computer graphics forum*. Vol. 33. 2. Wiley Online Library. 2014, pp. 105–114.

[Str] „Streaming and Out-of-Core Methods“. eng. In: *High Performance Visualization*. Chapman and Hall/CRC, 2013, pp. 237–258. ISBN: 9780429105357.

[Str+20] Daniel Ströter et al. „OLBVH: octree linear bounding volume hierarchy for volumetric meshes“. In: *The Visual Computer* 36.10 (2020), pp. 2327–2340.

[SW08] Daniel Scherzer and Michael Wimmer. „Frame sequential interpolation for discrete level-of-detail rendering“. In: *Computer Graphics Forum*. Vol. 27. 4. Wiley Online Library. 2008, pp. 1175–1181.

[Syn] *Sync Points | Entities | 0.51.0-Preview.32*. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/sync_points.html (visited on 07/19/2022).

[Teca] Unity Technologies. *Real-Time CAD & 3D Data Visualization Software | Unity*. URL: https://unity.com/products/unity-industrial-collection (visited on 06/30/2022).

[Tecb] Unity Technologies. *Unity - Manual: C# Compiler*. URL: https://docs.unity3d.com/2021.2/Documentation/Manual/CSharpCompiler.html (visited on 06/19/2022).

[Tecc] Unity Technologies. *Unity - Manual: C# Compiler*. URL: https://docs.unity3d.com/2022.2/Documentation/Manual/CSharpCompiler.html (visited on 06/19/2022).

[Tecd] Unity Technologies. *Unity - Manual: C# Job System Overview*. URL: https://docs.unity3d.com/Manual/JobSystemOverview.html (visited on 06/18/2022).

[Tece] Unity Technologies. *Unity - Manual: C# Job System Tips and Troubleshooting*. URL: https://docs.unity3d.com/Manual/JobSystemTroubleshooting.html (visited on 07/19/2022).

[Tecf] Unity Technologies. *Unity - Manual: Creating and Editing Terrains*. URL: https://docs.unity3d.com/Manual/terrain-UsingTerrains.html (visited on 07/06/2022).

[Tecg] Unity Technologies. *Unity - Manual: Memory in Unity*. URL: https://docs.unity3d.com/Manual/performance-memory-overview.html (visited on 07/16/2022).

[Tech] Unity Technologies. *Unity - Manual: NativeContainer*. URL: https://docs.unity3d.com/Manual/JobSystemNativeContainer.html (visited on 07/19/2022).

[Teci] Unity Technologies. *Unity - Manual: Order of Execution for Event Functions.* URL: https://docs.unity3d.com/Manual/ExecutionOrder.html (visited on 07/01/2022).

[Tecj] Unity Technologies. *Unity - Manual: Scheduling Jobs.* URL: https://docs.unity3d.com/Manual/JobSystemSchedulingJobs.html (visited on 07/19/2022).

[Teck] Unity Technologies. *Unity - Manual: Scripting Restrictions.* URL: https://docs.unity3d.com/Manual/ScriptingRestrictions.html (visited on 06/19/2022).

[Tecl] Unity Technologies. *Unity - Manual: TextMeshPro.* URL: https://docs.unity3d.com/Manual/com.unity.textmeshpro.html (visited on 06/30/2022).

[Tecm] Unity Technologies. *Unity - Manual: Unity User Manual 2020.3 (LTS).* URL: https://docs.unity3d.com/2020.3/Documentation/Manual/UnityManual.html (visited on 06/30/2022).

[Tecn] Unity Technologies. *Unity - Scripting API: Bounds.* URL: https://docs.unity3d.com/ScriptReference/Bounds.html (visited on 07/19/2022).

[Teco] Unity Technologies. *Unity 2022.1 Tech Stream | Unity.* URL: https://unity.com/de/releases/2022-1 (visited on 07/19/2022).

[Tecp] Unity Technologies. *Wie Shader Graph auf Ihr 2D- oder 3D-Spiel angewendet werden kann | Unity.* URL: https://unity.com/de/features/shader-graph (visited on 06/30/2022).

[Tec20] Unity Technologies. *Scripting API: Unity.Entities.* https://docs.unity3d.com/Packages/com.unity.entities@0.17/api/Unity.Entities.BlobAssetReference-1.html. Accessed: 2022-06-13. 2020.

[Tho05] C Thome. „Using a floating origin to improve fidelity and performance of large, distributed virtual worlds“. In: *2005 International Conference on Cyberworlds (CW'05)*. IEEE. 2005, 8–pp.

[TMJ98] Christopher C Tanner, Christopher J Migdal, and Michael T Jones. „The clipmap: a virtual mipmap“. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, pp. 151–158.

[TO] Robert F. Tobler and Thomas Ortner. *VILMA Data Stucture Design.* https://github.com/aardvark-platform/OpcViewer/blob/master/VilmaDataStructures_20101803.pdf. Accessed: 2022-06-10.

[TO15] C Traxler and T Ortner. „PRo3D—A tool for remote exploration and visual analysis of multi-resolution planetary terrains“. In: *Proceedings of the European Planetary Science Congress, Nantes, France*. Vol. 27. 2015.

[Tve74] Amos Tversky. „Assessing uncertainty“. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 36.2 (1974), pp. 148–159.

[Unia] *About Havok Physics for Unity | Havok Physics for Unity | 0.6.0-Preview.3*. URL: https://docs.unity3d.com/Packages/com.havok.physics@0.6/manual/index.html (visited on 06/30/2022).

[Unib] *About Unity Physics | Unity Physics | 0.51.0-Preview.32*. URL: https://docs.unity3d.com/Packages/com.unity.physics@0.51/manual/index.html (visited on 06/30/2022).

[Unic] *Burst User Guide | Burst | 1.7.2*. URL: https://docs.unity3d.com/Packages/com.unity.burst@1.7/manual/index.html (visited on 06/18/2022).

[Unid] *Changelog | Entities | 0.50.1-Preview.2*. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.50/changelog/CHANGELOG.html (visited on 07/19/2022).

[Unie] *Conversion Workflow | Entities | 0.16.0-Preview.21*. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.16/manual/conversion.html (visited on 07/18/2022).

[Unif] *Official - Experimental Entities 0.50 Is Available*. Unity Forum. URL: https://forum.unity.com/threads/experimental-entities-0-50-is-available.1253394/ (visited on 06/19/2022).

[Unig] *Official - Experimental Entities 0.51 Is Available*. Unity Forum. URL: https://forum.unity.com/threads/experimental-entities-0-51-is-available.1281233/ (visited on 07/19/2022).

[Unih] *Physics updates in Unity 2019.3*. Unity Blog. URL: https://blog.unity.com/technology/physics-updates-in-unity-2019-3 (visited on 02/02/2022).

[Unii] *Requirements and Compatibility | Hybrid Renderer | 0.51.0-Preview.32*. URL: https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.51/manual/requirements-and-compatibility.html (visited on 07/19/2022).

[Unij] *Struct AABB | Package Manager UI Website*. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.0/api/Unity.Mathematics.AABB.html (visited on 07/19/2022).

[Unik] *Struct Aabb | Unity Physics | 0.5.1-Preview.2*. URL: https://docs.unity3d.com/Packages/com.unity.physics@0.5/api/Unity.Physics.Aabb.html (visited on 07/19/2022).

[Unil] *Struct MeshCollider | Unity Physics | 0.51.0-Preview.32*. URL: https://docs.unity3d.com/Packages/com.unity.physics@0.51/api/Unity.Physics.MeshCollider.html?q=meshcollider (visited on 07/19/2022).

[Unim] *UniRx/Assets/Plugins/UniRx/Scripts at Master · Neuecc/UniRx*. GitHub. URL: https://github.com/neuecc/UniRx (visited on 06/30/2022).

[Unin] *UniTask*. Cysharp, Inc. URL: https://github.com/Cysharp/UniTask (visited on 06/30/2022).

[Unio] *Unity and .NET, What's next?* Unity Blog. URL: https://blog.unity.com/technology/unity-and-net-whats-next (visited on 06/19/2022).

[Unip] *Unity Announces Fourth Quarter and Full Year 2020 Financial Results*. URL: https://investors.unity.com/news/news-details/2021/Unity-Announces-Fourth-Quarter-and-Full-Year-2020-Financial-Results/default.aspx (visited on 06/30/2022).

[Uniq] *Unity Collections Package | Collections | 1.3.1*. URL: https://docs.unity3d.com/Packages/com.unity.collections@1.3/manual/index.html (visited on 06/30/2022).

[Unir] *Unity Jobs Package | Jobs | 0.51.0-Preview.32*. URL: https://docs.unity3d.com/Packages/com.unity.jobs@0.51/manual/index.html (visited on 06/18/2022).

[Unis] *Unity UI: Unity User Interface | Unity UI | 1.0.0*. URL: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html (visited on 06/30/2022).

[Unit] *Unity.Mathematics | Package Manager UI Website*. URL: https://docs.unity3d.com/Packages/com.unity.mathematics@1.0/manual/index.html (visited on 06/30/2022).

[Uniu] *Universal Render Pipeline Overview | Universal RP | 13.1.8*. URL: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@13.1/manual/index.html (visited on 06/30/2022).

[Uni19] Unity, director. *Overview of Havok Physics in Unity - Unite Copenhagen*. Nov. 6, 2019. URL: https://www.youtube.com/watch?v=Uv7DWq6KFbk (visited on 07/08/2022).

[Uni22] Unity Technologies. „Unity Game Engine“. English. In: (2022). (Visited on 02/02/2022).

[Unra] *How Sony Pictures Imageworks Created a Real-Time Thriller for Netflix's 'Love, Death + Robots'*. Unreal Engine. URL: https://www.unrealengine.com/en-US/spotlights/how-sony-pictures-imageworks-created-a-real-time-thriller-for-netflix-s-love-death-robots (visited on 07/04/2022).

[Unrb] *Landscape Edit Layers*. URL: https://docs.unrealengine.com/4.27/en-US/BuildingWorlds/Landscape/Layers/ (visited on 02/17/2022).

[Unrc] *Unreal Engine Developers Shine at PlayStation 5 Showcase*. Unreal Engine. URL: https://www.unrealengine.com/en-US/blog/unreal-engine-developers-shine-at-playstation-5-showcase (visited on 02/17/2022).

[VM02] Gokul Varadhan and Dinesh Manocha. „Out-of-core rendering of massive geometric environments“. In: *IEEE Visualization, 2002. VIS 2002.* IEEE. 2002, pp. 69–76.

[Vrg] *VRGeoScience*. URL: https://www.vrgeoscience.com/overview/ (visited on 02/08/2022).

[Wal+19] Ingo Wald et al. „RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location.“ In: *High Performance Graphics (Short Papers)*. 2019, pp. 7–13.

[Wan+20] Feng Wang et al. „CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data“. In: *Computer Graphics Forum*. Vol. 39. 3. Wiley Online Library. 2020, pp. 1–12.

[Was+12] Jürgen Waser et al. „Sketching Uncertainty into Simulations“. In: *IEEE Transactions on Visualization and Computer Graphics, 18(12), pp. 2255-2264 / Proceedings IEEE Scientific Visualization 2012* (2012). URL: https://www.vrvis.at/publications/PB-VRVis-2012-031.

[WH06] I. Wald and V. Havran. „On Building Fast Kd-Trees for Ray Tracing, and on Doing That in O(N Log N)“. In: *2006 IEEE Symposium on Interactive Ray Tracing*. Sept. 2006, pp. 61–69. DOI: 10.1109/RT.2006.280216.

[Win] *WinDirStat*. URL: https://windirstat.net/ (visited on 07/20/2022).

[Xio+21] Liyang Xiong et al. „Geomorphology-oriented digital terrain analysis: Progress and perspectives“. In: *Journal of Geographical Sciences* 31.3 (2021), pp. 456–476.

[XZQ19] Junjie Xue, Xiang Zhai, and Huiyang Qu. „Efficient Rendering of Large-Scale CAD Models on a GPU Virtualization Architecture with Model Geometry Metrics“. In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 251–2515. DOI: 10.1109/SOSE.2019.00043.

[Yes+21] Rahul Yesantharao et al. „Parallel Batch-Dynamic k d-Trees“. In: *arXiv preprint arXiv:2112.06188* (2021).

[Zar+21] Liliana Zarco et al. „Scope and delimitation of game engine simulations for ultra-flexible production environments“. In: *Procedia CIRP* 104 (2021), pp. 792–797.

[Zel19] Stefan Zellmann. „Comparing Hierarchical Data Structures for Sparse Volume Rendering with Empty Space Skipping“. In: *arXiv preprint arXiv:1912.09596* (2019).

[Zen] *What Is Dependency Injection?* Modest Tree Media Inc. URL: https://github.com/modesttree/Zenject (visited on 06/30/2022).

[Zha+16] Rui Zhai et al. „GPU-based real-time terrain rendering: Design and implementation“. In: *Neurocomputing* 171 (2016), pp. 1–8. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2014.08.108. URL: https://www.sciencedirect.com/science/article/pii/S0925231215012096.

Appendix A.

Ordered Point Cloud File Format

In this appendix, the most important parts of the OPC specification are included. Fields not relevant for this work were ommitted. The full specification is available in the OPC documentation [TO].

patchhierarchy.xml

The patchhierarchy.xml file stores the relationship of LOD nodes in it's *SubPatchMap* XML node. Each entry consists of a *key* and a *val* node. The *key* node is the guid, and folder name, of the patch. The *val* contains all children of this LOD node. *RootPatchName* is the root of the LOD tree hierarchy.

```
<?xml version="1.0" encoding="UTF-8"?>
<Aardvark version="0">
   <PatchHierarchy version="0">
      <RootPatch>name-of-root-patch</RootPatch>
      <SubPatchMap>
         <item>
            <key>guid-of-patch-1</key>
            <val>
               <sw>guid-of-south-west-sub-patch-of-patch-1</sw>
               <se>guid-of-south-east-sub-patch-of-patch-1</se>
               <nw>guid-of-north-west-sub-patch-of-patch-1</nw>
               <ne>guid-of-north-east-sub-patch-of-patch-1</ne>
            </val>
         </item>
         . . .
      </SubPatchMap>
   </PatchHierarchy>
</Aardvark>
```

Listing 1: Definition of patchhierarchy.xml containing the order of files. Reprinted from Tobler and Ortner [TO]

patch.xml

The patch.xml file stores the relative path of the image to the images folder and for the positions, normals, and texture corodinate files relative to the patch folder. It furthermore contains the local-to-world transformation matrix.

```
<?xml version="1.0" encoding="UTF-8"?>
<Aardvark version="0">
   <Patch version="0">
      <GeometryType>QuadList</GeometryType>
      <Local2Global>[[ 1.0, 0.0, 0.0, 396.0 ], [ 0.0, 1.0,
         0.0, 400.0 ], [ 0.0, 0.0, 1.0, 112.0 ], [ 0.0, 0.0,
         0.0, 1.0 ]]</Local2Global>
      <Positions>guid-of-positions-array-file.aara</Positions>
      <Normals>guid-of-normals-array-file.aara</Normals>
      <Coordinates>
         <DiffuseColor1Coordinates>guid-of-coords-1-array-file.aara</Diffus
      </Coordinates>
      <Textures>
         <DiffuseColor1Texture>guid-of-image-1/tile.jpg</DiffuseColor1Textu
      </Textures>
   </Patch>
</Aardvark>
```

Listing 2: Definition of patch.xml containing the relative file paths and local-to-world transformation matrix. Reprinted from Tobler and Ortner [TO]

Aardvark Array File

The Aardvark Array File (extension: .aara) contains array data in binary format. It is defined as follows [TO]:

> The Aardvark array file consists of 4 parts, stored in succession: the **Type**, the **Dimension**, the **Size**, and the **Data**. All primitive data types are stored in little-endian (Intel) format.

The mesh and texture coordinate information relevant for this work is of type *V3f* or *V2f* (3- or 2-component vector, single-precision float). The dimension is 2 and stored as a single byte. All example scenes are in *RowMajor* order. These are binary files and thus no example is given.

Appendix B.

Visionary File Format

In this appendix, an overview of the resulting file format all OPC files are converted into is given. Most language specific keywords and optional fields were omitted. Custom types provided by Unity are *Unity.Physics.Aaab* (axis-aligned bounding box defined by min and max position), *Unity.Mathematics.float3* (3-component single-precision float vector) and *Unity.Mathematics.double4x4* (4 by 4 double-precision matrix). Othert custom non-primitve types are included in the relative listing.

Hierarchy

The file *dotshierarchy.mpk* is the equivalent to the *patchhierarchy.xml* file of OPC. Instead of using XML files MessagePack is used as binary serialization format. To serialize the arrays of non-standard types *Unity.Mathematics.double4x4* and *Unity.Physics.Aabb* the serializer is extended via MessagePack extension points for custom formatters and resolvers. They are serialized as the length of the following array data and the array data serialized linearly in binary.

There are a few key differences between both formats. The main difference is that LOD hierarchies in OPC are stored as hierarchical XML nodes, but the presented LOD hierachies flattens them to aligned arrays. All aligned arrays have a length equal to the number of LOD nodes. The index of each LOD node refers to the position in those arrays, e.g., components for node *index 0* are stored in *Names[0]* and *LocalToWorld[0]*. *Index 0* is the root of the LOD hierarchy. Parent-child relationships are stored in separate arrays where *IndexToParentIndex* contains the relationship to the parent node and *IndexToChildIndices* contains an array of indices of all children of a node. The

patchhierarchy.xml	dotshierarchy.mpk
XML	MessagePack
Hierarchical	Aligned Arrays
LocalToWorld stored in patch.xml	contains aligned LocalToWorld[]

Table 1: Differences between patchhierarchy.xml and dotshierarchy.mpk

local-to-world transformation amtrix is elevated from one *patch.xml* file per node the the hierarchy in one aligned array *LocalToWorld[]*.

```
[MessagePackObject, NoReorder]
DotsPatchHierarchyMpk
{
 // Patches/<Name[i]>:
 // [0]: 04-Patch-00001~0099
 // ...
 [Key(0)]
 string[] Names;

 // Mapping from patch to sub patches
 // [0]: [1, 2]
 [Key(1)]
 SubPatchMapping[] IndexToChildIndices;

 // Mapping from patch to parent
 // [0]: -1 (none)
 // [1]: 0
 // [2]: 0
 [Key(2)]
 int[] IndexToParentIndex;

 // IndicesOfLevel[0] =[0]
 // IndicesOfLevel[1] =[1, 2]
 [Key(4)]
 IndicesOfLevel[] IndicesOfLevel;

 // Local Bounds
 [Key(7), MessagePackFormatter(typeof(ArrayFormatter<Aabb>))]
 Aabb[] LocalAabb;

 // Global Bounds (without origin shift)
 [Key(8), MessagePackFormatter(typeof(ArrayFormatter<Aabb>))]
 Aabb[] GlobalAabb;

 // LocalToWorld in double-precision
 [Key(10),
    MessagePackFormatter(typeof(ArrayFormatter<double4x4>))]
 double4x4[] LocalToWorld;
}

[MessagePackObject]
```

```
SubPatchMapping
{
 // Indices of sub patches
 [Key(0)]
 int[] Indices;
}

Aabb
{
 float3 Min;
 float3 Max;
}
```

Listing 3: Definition of dotshierarchy.mpk containing the aligned arrays for relative file paths and local-to-world transformation matrix.

Renderable Blobs

The file *renderable.blob* contains the mesh and texture data, including texture coordinates. As the used serializer only supports blittable types, it is not possible to use it for serialization of *UnityEngine.Mesh* and *UnityEngine.Texture*. Accordingly, the mesh and texture data must be serialized, and the objects need to be recreated and reinitialized after deserialization.

```
RenderableBlob
{
 MeshBlob Mesh;
 TextureBlob Texture;
}

MeshBlob
{
 double4x4 LocalToWorld;
 AABB  RenderBounds;

 BlobArray<VertexData> Vertices;
 BlobArray<int> Indices;
}

VertexData
{
 float3 Position;
 float3 Normal;
 float2 UV;
```

```
}

TextureBlob
{
 TextureFormat Format;
 int MipmapCount;
 int Width;
 int Height;
 BlobArray<byte> TextureData;
}
```

Listing 4: Definition of the renderable blob. It contains both mesh and texture information.

*.collider.blob

The file *aabb.collider.blob* contains the final collider serialized as bytes. Colliders are static and storing precalculated colliders avoids costly recreation everytime a node is streamed in.

The file *mesh.collider.blob* contains the final mesh collider serialized as bytes. Creation of MeshColliders in Unity is a slow operation, and performing it during runtime is infeasible for large-scale terrain.