



Interactive Cross-Section Visualization of RIMFAX Subsurface Data for PRo3D

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Lucia Kloos

Registration Number 12122563

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller

Assistance: Dipl.-Ing. Harald Steinlechner
Dipl.-Ing. Dr. Christoph Traxler

Vienna, June 25, 2026

Lucia Kloos

Eduard Gröller



Interactive Cross-Section Visualization of RIMFAX Subsurface Data for P_{Ro}3D

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Lucia Kloos

Matrikelnummer 12122563

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller

Mitwirkung: Dipl.-Ing. Harald Steinlechner

Dipl.-Ing. Dr. Christoph Traxler

Wien, 25. Juni 2026

Lucia Kloos

Eduard Gröller

Declaration of Authorship

Lucia Kloos

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

I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix “Overview of Generative AI Tools Used” I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Vienna, June 25, 2026

A handwritten signature in black ink, appearing to read 'L. Kloos', written over a horizontal line.

Lucia Kloos

Acknowledgements

I want to thank my supervisor, Harald Steinlechner, for his guidance and feedback during this work. Also, I want to thank my cats Princess Mononoke and Valie Export for their emotional support and my little brother Philipp for brainstorming with me.

Abstract

This thesis presents an interactive visualization approach for integrating subsurface radar data acquired by the Radar Imager for Mars' Subsurface Experiment (RIMFAX) into a three-dimensional terrain context. RIMFAX radargrams represent subsurface reflections recorded along the Perseverance rover traverse and are therefore two-dimensional data embedded in three-dimensional space. To improve their spatial interpretability, this work investigates cross-sections that open the terrain along a selected path and display the corresponding radargram on a curtain-shaped surface.

The developed prototype supports both interactively drawn paths and imported rover traverses. A central challenge is the consistent classification of the terrain for paths containing sharp turns, concave regions, loops, or self-intersections. Several local approaches were investigated, including tiled clipping planes and a nearest-point heuristic based on a frozen viewing direction. These methods produced inconsistent or visually unclear classifications for geometrically complex paths.

The final method reformulates the clipping problem as a two-dimensional point-in-polygon classification. The path is combined with an additional reference point to form a closed polygon, and terrain vertices are projected into a common two-dimensional coordinate system and classified as inside or outside this polygon. The classification is computed on the CPU and transferred to the GPU as a per-vertex attribute, where interpolation and fragment discarding produce the visible terrain cut. For fixed inputs, this approach provides stable and reproducible results and handles concave paths more consistently than the explored local methods.

A continuous curtain is generated along the complete cross-section path by expanding its line segments into vertical quads. Normalized arc-length texture coordinates provide a continuous horizontal parameterization and allow the radargram to be mapped according to distance along the path.

The implementation is written in F# using the Aardvark Platform and processes the terrain through its Out-of-Core Point Cloud (OPC) patch hierarchy. It therefore uses technologies, data structures, and rendering mechanisms compatible with later integration into P_{Ro}3D. Performance was evaluated on a system equipped with an 11th Gen Intel Core i7-1165G7 processor, Intel Iris Xe Graphics, and 16 GB of RAM. For the evaluated terrain view, the classification of approximately 1.91 million vertices required approximately 1.21

seconds on average. After classification, navigation with clipping and curtain rendering enabled remained interactive at approximately 54 frames per second during continuous camera movement.

The results demonstrate that the method provides a practical foundation for embedding RIMFAX radargrams into their corresponding terrain context. Remaining limitations include spikes along the clipped terrain boundary, gaps or overlaps between the terrain and curtain, preprocessing requirements for the radargram imagery, and semantically ambiguous results for some self-intersecting paths.

Kurzfassung

Diese Arbeit stellt einen interaktiven Visualisierungsansatz vor, mit dem unterirdische Radardaten, die vom „Radar Imager for Mars’ Subsurface Experiment“ (RIMFAX) erfasst wurden, in einen dreidimensionalen Geländekontext integriert werden können. RIMFAX-Radargramme stellen unterirdische Reflexionen dar, die entlang der Fahrstrecke des Rovers „Perseverance“ aufgezeichnet wurden, und sind daher zweidimensionale Daten, die in einen dreidimensionalen Raum eingebettet sind. Um ihre räumliche Interpretierbarkeit zu verbessern, untersucht diese Arbeit Querschnitte, die das Gelände entlang einer ausgewählten Route freilegen und das entsprechende Radargramm auf einer vorhangförmigen Fläche anzeigen.

Der entwickelte Prototyp unterstützt sowohl interaktiv gezeichnete Routen als auch importierte Rover-Fahrstrecken. Eine zentrale Herausforderung ist die konsistente Klassifizierung des Geländes bei Pfaden, die scharfe Kurven, konkave Bereiche, Schleifen oder Selbstschnittpunkte enthalten. Es wurden mehrere lokale Ansätze untersucht, darunter gekachelte Clipping-Ebenen und eine auf einer festen Blickrichtung basierende Heuristik des nächstgelegenen Punktes. Diese Methoden führten bei geometrisch komplexen Pfaden zu inkonsistenten oder visuell unklaren Klassifizierungen.

Die endgültige Methode formuliert das Clipping-Problem als zweidimensionale „Punkt-im-Polygon“-Klassifizierung um. Der Pfad wird mit einem zusätzlichen Referenzpunkt zu einem geschlossenen Polygon kombiniert, und die Geländeknoten werden in ein gemeinsames zweidimensionales Koordinatensystem projiziert und als innerhalb oder außerhalb dieses Polygons liegend klassifiziert. Die Klassifizierung wird auf der CPU berechnet und als Attribut pro Knoten an die GPU übertragen, wo Interpolation und Fragmentverwerfung den sichtbaren Geländeschnitt erzeugen. Bei festen Eingaben liefert dieser Ansatz stabile und reproduzierbare Ergebnisse und behandelt konkave Pfade konsistenter als die untersuchten lokalen Methoden.

Entlang des gesamten Querschnittspfades wird ein durchgehender Vorhang erzeugt, indem dessen Liniensegmente in vertikale Quadrate erweitert werden. Normalisierte Bogenlängen-Texturkoordinaten bieten eine durchgehende horizontale Parametrisierung und ermöglichen es, das Radargramm entsprechend der Entfernung entlang des Pfades abzubilden.

Die Implementierung wurde in F# unter Verwendung der Aardvark-Plattform geschrieben und verarbeitet das Gelände über deren Out-of-Core Point Cloud (OPC) Patch Hierarchy.

Sie nutzt daher Technologien, Datenstrukturen und Rendering-Mechanismen, die mit einer späteren Integration in P_{Ro}3D kompatibel sind. Die Leistung wurde auf einem System mit einem Intel Core i7-1165G7-Prozessor der 11. Generation, Intel Iris Xe Graphics und 16 GB RAM bewertet. Für die untersuchte Geländansicht dauerte die Klassifizierung von etwa 1,91 Millionen Scheitelpunkten im Durchschnitt etwa 1,21 Sekunden. Nach der Klassifizierung blieb die Navigation bei aktiviertem Clipping und Curtain-Rendering während kontinuierlicher Kamerabewegungen mit etwa 54 Bildern pro Sekunde interaktiv.

Die Ergebnisse zeigen, dass die Methode eine praktische Grundlage für die Einbettung von RIMFAX-Radargrammen in den entsprechenden Geländekontext bietet. Zu den verbleibenden Einschränkungen zählen Spitzen entlang der beschnittenen Geländegrenze, Lücken oder Überlappungen zwischen Gelände und Vorhang, Vorverarbeitungsanforderungen für die Radargramm-Bilder sowie semantisch mehrdeutige Ergebnisse bei einigen sich selbst schneidenden Pfaden.

Contents

Abstract	ix
Kurzfassung	xi
Contents	xiii
1 Introduction	1
1.1 Problem	2
1.2 Goal and Research Questions	2
1.3 Requirements and Scope	4
1.4 Operational Definitions	5
2 Background and Definitions	7
2.1 Background	7
2.2 Definitions	8
3 State of the Art	11
3.1 VTK and VTK Cutter	11
3.2 ParaView	12
3.3 CGAL: The Computational Geometry Algorithms Library	13
3.4 libigl	14
3.5 BGS Groundhog	15
3.6 GemPy	16
3.7 Comparing Existing Tools	16
3.8 Implicit Slicing for Functionally Tailored Additive Manufacturing	17
3.9 An Optimal Algorithm for 3D Triangle Mesh Slicing	18
3.10 Geometric Foundations of the Classification Problem	19
3.11 The 2D Point-in-Polygon Problem	19
4 Our Method	23
4.1 Method Overview	23
4.2 Explored Clipping Strategies	23
4.3 Point-in-Polygon-Based Classification	30
4.4 Curtain Geometry Concept	32

5	Implementation	35
5.1	Aardvark and PPro3D Integration	35
5.2	Implementation Pipeline Overview	35
5.3	Coordinate Spaces and Floating-Point Precision	36
5.4	Terrain Loading and OPC Patch Processing	37
5.5	CPU-Side Classification	38
5.6	GPU-Side Clipping	40
5.7	User Interaction	41
5.8	Curtain Geometry Generation	42
6	Results and Discussion	45
6.1	RQ1: Classification of the Terrain	45
6.2	RQ2: Curtain Construction and Radargram Mapping	50
6.3	RQ3: Preparation for Integration into PPro3D	53
6.4	Answer to the Main Research Question	57
6.5	Limitations and Future Work	58
	Overview of Generative AI Tools Used	61
	List of Figures	63
	List of Tables	65
	Bibliography	67

Introduction

In this work, we develop an interactive visualization pipeline for Mars 2020 RIMFAX subsurface data. The primary data product of RIMFAX consists of radargrams, which represent subsurface reflections along the rover’s path [Ham20, Ham22]. While these radargrams already provide valuable insight, they are two-dimensional signals embedded in three-dimensional space [Ham22]. This makes it difficult to directly relate them to the surrounding terrain and geological context.

To address this, we explore cross-sections as an intuitive visualization method. Cross-sections allow the radargrams to be spatially embedded into the 3D terrain by cutting the terrain open and revealing the subsurface along the defined path. This provides a clearer spatial understanding and turns cross-sections into an important visualization tool for planetary science. Similar concepts are widely used in scientific visualization systems such as ParaView and VTK, where slicing operations are a standard method to explore volumetric and spatial data [Kit26b, SML06, Kit26d]. However, our implementation is not restricted to globally defined planar cuts. It supports more complex cross-section shapes and is therefore better suited to the visualization of rover-based subsurface measurements in a terrain context.

To realize this form of visualization, we generate a curtain geometry along the cross-section that serves as the spatial carrier of the radar data. This allows subsurface measurements to be integrated directly into the 3D environment and provides an intuitive representation of the Martian subsurface. To support different analysis scenarios, the system provides two ways of defining a cross-section: it can either be sketched interactively on the terrain or derived from a recorded rover traverse path. This makes it possible to support both exploratory visualization and data-faithful inspection along measured paths.

The method was designed for later use in the visualization tool PRo3D, short for Planetary Robotics 3D Viewer, which was developed for planetary scientists working on NASA and ESA Mars missions [VRV25]. After completion of the prototype, the method was

integrated into PPro3D by other developers. This integration is not part of the contribution or evaluation of this thesis, but it demonstrates the relevance of designing the method around PPro3D-compatible technologies and data structures.

Computing and rendering stable, interactive cross-sections on large triangulated terrain meshes is an ongoing research topic in computer graphics, particularly for arbitrarily formed cross-sections and planetary-scale data [MVS⁺17, SIM16].

1.1 Problem

Creating interactive cross-sections on large triangulated Mars terrain meshes is a challenge because the result must be stable, deterministic, and perceptually clear under all conditions. This difficulty is amplified by the characteristics of planetary-scale data. The large coordinate values introduce floating-point precision issues, which can lead to visual artifacts and unstable behavior [Kne17]. At the same time, the flexibility in defining cross-sections increases the complexity of the problem, as arbitrary paths may contain sharp turns, self-intersections, or highly irregular shapes. In addition to these technical challenges, perceptual issues arise if the resulting cross-section is ambiguous or difficult to interpret, for example due to inconsistent orientation, visual clutter, or a lack of clear separation between foreground and background geometry. Such perceptual problems are illustrated in Figure 1.1, where the curtain becomes visible, but its presentation without proper depth relationships leads to a visually confusing result.

The central problem is therefore to construct a stable and consistent cross-section through a complex 3D terrain while preserving numerical robustness and visual clarity. In practice, naive or locally defined clipping strategies can lead to inconsistent inside—outside classifications, resulting in disconnected geometry, missing regions, and visually confusing cross-sections, as illustrated in Figure 1.2. These geometric, computational, and visualization challenges motivate the research questions formulated in the following section.

1.2 Goal and Research Questions

The goal of this thesis is to investigate how RIMFAX subsurface radar data can be embedded into a three-dimensional terrain visualization through cross-sections. The work focuses on three connected challenges: consistently classifying the terrain along geometrically complex paths, constructing a surface on which the radargram can be displayed, and designing the resulting pipeline so that it remains interactive on large terrain datasets and can later be integrated into PPro3D.

This goal is expressed by the following main research question:

How can RIMFAX subsurface radar data be visualized in a 3D terrain context using stable, deterministic, and interactive cross-sections along geometrically

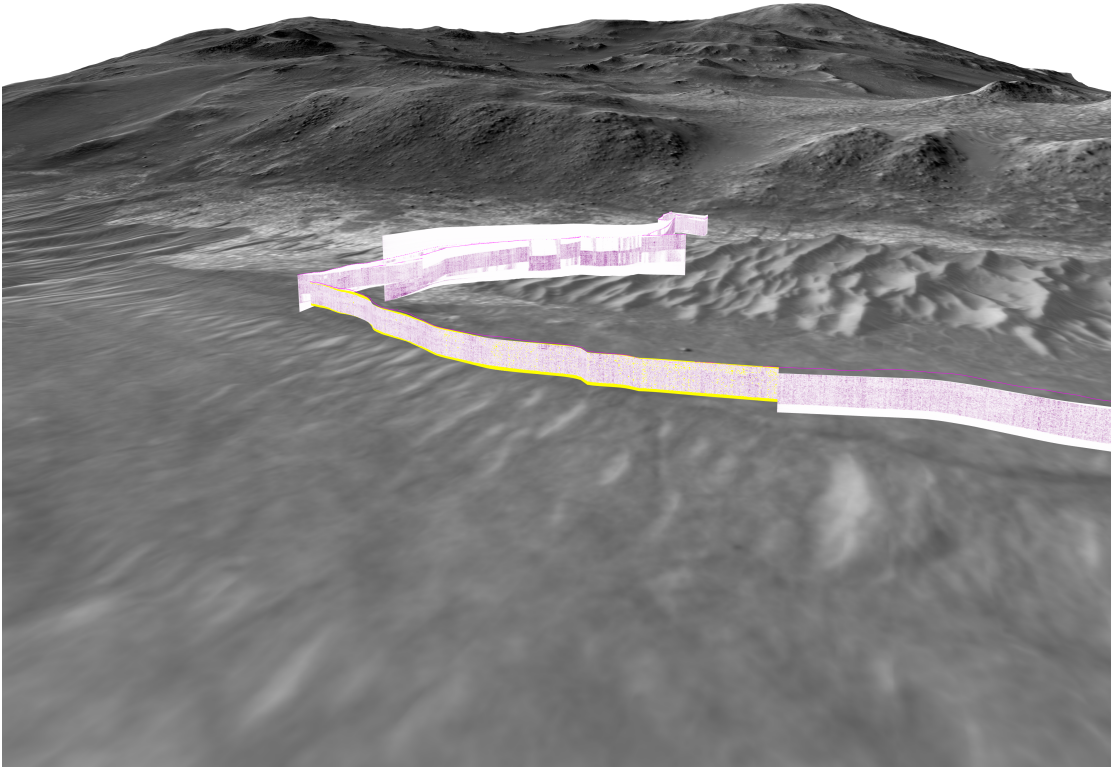


Figure 1.1: Visually confusing results caused by improper depth relationships.

complex paths, while preparing the visualization pipeline for later integration into PPro3D?

To investigate the individual components of this problem, the main research question is divided into three sub-questions:

RQ1: How can the terrain region to be retained or removed be classified consistently for cross-section paths containing concave turns, loops, or self-intersections?

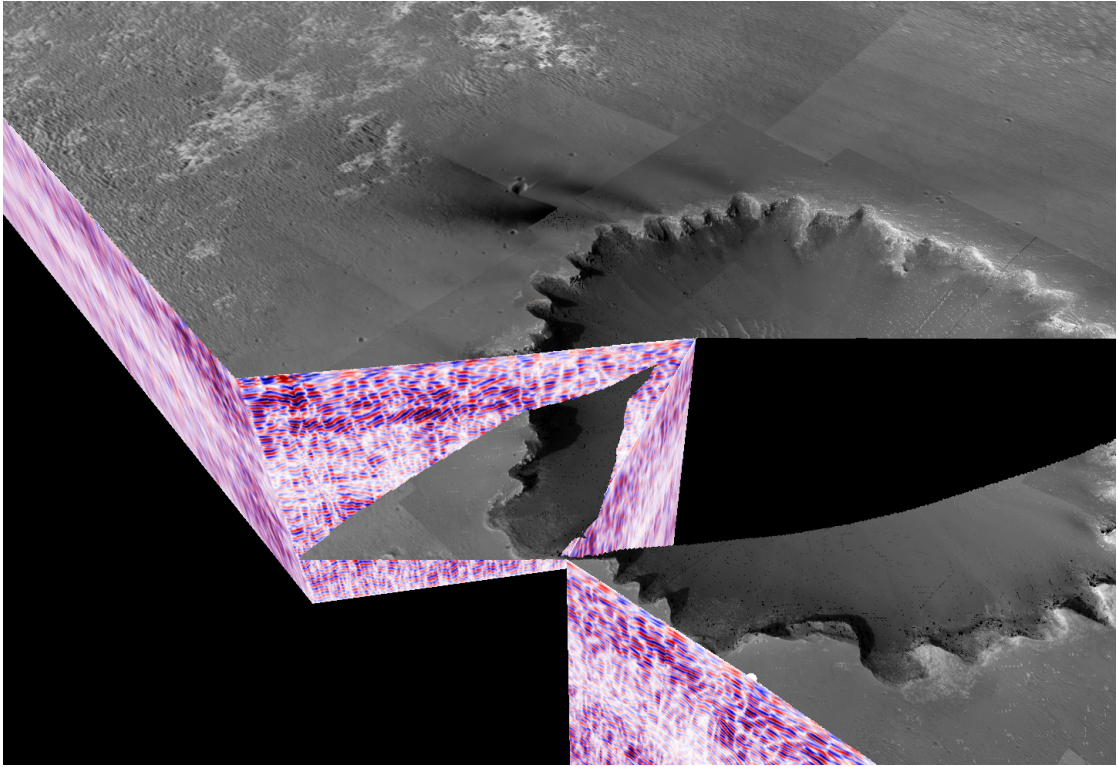


Figure 1.2: Incorrect surface classification resulting from local clipping strategy.

RQ2: How can a curtain geometry be constructed and parameterized along the cross-section path so that RIMFAX radargrams can be displayed continuously in their corresponding 3D terrain context?

RQ3: How should the classification, clipping, and curtain-generation method be designed to support integration into PPro3D while maintaining interactive performance on large terrain datasets?

These questions define the structure of the thesis. RQ1 concerns the geometric classification and clipping of the terrain. RQ2 concerns the construction and parameterization of the curtain that carries the radargram. RQ3 concerns the software architecture, processing pipeline, and performance characteristics required for later integration into PPro3D. Together, the answers to these questions provide the answer to the main research question.

1.3 Requirements and Scope

The requirements of the prototype follow directly from the research questions.

To address RQ1, the system must support cross-section paths that are either drawn interactively or imported from recorded rover traverses. The classification must provide a consistent retained terrain region for paths containing concave turns, sharp directional changes, loops, or self-intersections. Once a path has been finalized, the result must not change unintentionally during camera navigation. Identical path data, reference-point data, terrain geometry, and projection parameters must produce the same classification in repeated executions.

To address RQ2, the system must construct a continuous curtain along the complete cross-section path. The curtain must provide texture coordinates that preserve the progression of distance along the path so that the horizontal axis of a radargram can be mapped continuously to the corresponding spatial trajectory. The generated surface must be suitable for rendering together with the clipped terrain and should make the relationship between the terrain and the subsurface data perceptually understandable.

To address RQ3, the method must be compatible with the central technologies and data structures used by PPro3D. It must operate on the OPC patch hierarchy rather than requiring the complete terrain mesh to be loaded and processed as a single object. The rendering stages must remain composable with the existing scene-graph and shader pipeline from Aardvark [Aar24]. Approaches that require global control over several rendering passes or shared framebuffer state, such as stencil-buffer-based methods, are therefore unsuitable for the intended integration.

The computationally more expensive classification may be performed when a path is finalized or when a terrain patch is prepared. During subsequent navigation, clipping and curtain rendering should introduce sufficiently little overhead to preserve interactive exploration of the terrain.

The direct integration of the prototype into the complete PPro3D application is outside the scope of this thesis. Instead, the work evaluates whether the implementation uses compatible technologies, data structures, and processing stages, and whether its measured performance is suitable for an interactive visualization workflow.

The scientific interpretation of a radargram is additionally constrained by the acquisition process. RIMFAX measurements are available only along the actual Perseverance rover traverse. An interactively drawn path can therefore be used to test and explore the cross-section method, but a radargram can only be interpreted as measured subsurface data if the curtain follows the corresponding recorded rover trajectory.

1.4 Operational Definitions

The main research question uses the terms *stable*, *deterministic*, *interactive*, and *geometrically complex*. This section defines these terms in the context of the proposed cross-section visualization pipeline.

Stable. A cross-section is considered stable if, after the path has been finalized, its classification and visual appearance do not change unintentionally during camera navigation or repeated rendering. In particular, the retained side of the terrain must not switch as the camera moves, and the cut should not exhibit view-dependent flickering or sudden changes caused by the classification method.

Deterministic. The implementation is considered deterministic if identical input data and identical configuration parameters produce the same per-vertex inside–outside classification in repeated executions. The relevant input consists of the finalized cross-section path, the reference point, the terrain patch geometry, and the projection parameters. Determinism therefore concerns the reproducibility of the computed classification, rather than only the visual similarity of the rendered result.

Interactive. The system is considered interactive if the user can draw and finalize a cross-section without disruptive waiting times and can inspect and navigate the terrain interactively after the clipping classification has completed. Navigation is temporarily unavailable while the classification is being computed. Interactivity is evaluated quantitatively using the processing time required after finalizing a path, the steady-state rendering time, and the resulting frame rate, as described in Chapter 6.

Geometrically complex path. In this thesis, a geometrically complex path is a cross-section polyline whose two-dimensional projection contains one or more configurations that cannot be handled reliably by a purely local side-classification rule. These configurations include concave turns, sharp directional changes, loops, and self-intersections.

These definitions provide the criteria used to interpret and evaluate the research questions. Stability and determinism concern the consistency and reproducibility of the clipping result, interactivity concerns processing and rendering performance, and geometric complexity defines the relevant input-path categories considered in the evaluation.

Background and Definitions

2.1 Background

2.1.1 Mars 2020 Mission Motivation

The cross-section visualization developed in this work is motivated by the scientific goals of the Mars 2020 mission and, in particular, by the subsurface measurements acquired with the Radar Imager for Mars' Subsurface Experiment (RIMFAX) [Ham20]. RIMFAX is a ground-penetrating radar (GPR) instrument mounted on NASA's Perseverance rover and operates as part of the broader exploration campaign in the Jezero Crater [Ham20].

The landing site in the Jezero Crater was selected because of its deltaic landform [Ham22]. Figure 2.1 shows the Perseverance rover traverse from Sol 0 to Sol 204 on a High Resolution Imaging Science Experiment (HiRISE) orbital color image. The white numbers mark the rover's sol traverse endpoints. The highlighted traverse segments A–D indicate sections where the rover path is generally perpendicular to the boundary of Séítah and where dipping subsurface reflectors were observed with RIMFAX [Ham22].

Orbital observations indicate that this structure was formed by sediment deposition in an ancient lake environment [Ham22]. Such environments are of high astrobiological interest, as they may have preserved biosignatures of past microbial life. One of the goals of the mission is to assess the ancient habitability of the region and to identify promising geological samples for caching and potential return to Earth [Ham22]. Subsurface information is crucial in this context, as it reveals stratigraphic layering, depositional structures, and buried interfaces that are not visible at the surface [Ham22].

The RIMFAX data products are archived in NASA's Planetary Data System (PDS) and organized into a bundle structure [HP21].

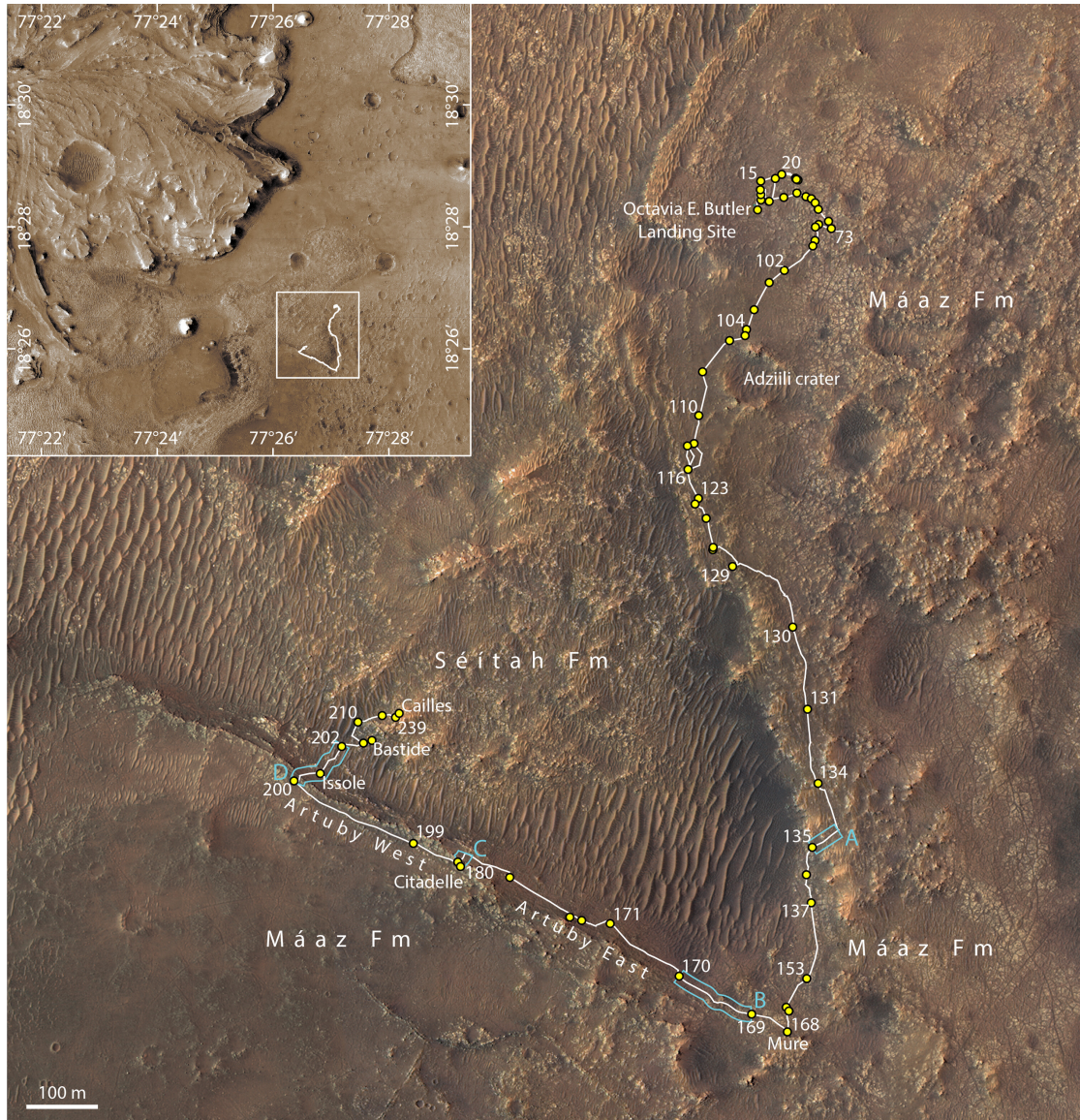


Figure 2.1: Perseverance rover traverse in the Jezero Crater from Sol 0 to Sol 204. Reproduced from [Ham22].

2.2 Definitions

2.2.1 Ground Penetrating Radar

RIMFAX is a ground-penetrating radar (GPR) instrument that is designed to image the shallow subsurface beneath the rover as it traverses the terrain [Ham20]. It is a non-invasive geophysical instrument consisting of an electronics unit and an antenna system [Ham20]. The instrument is designed to detect shallow reflections as well as

deeper subsurface layering down to approximately 10–15 meters, depending on material properties and attenuation [Ham22]. The radar transmits electromagnetic waves into the ground over a range of frequencies and records the reflected signals from subsurface structures [Ham20]. As shown in Figure 2.2, the radargram reveals subsurface layering along the rover traverse.

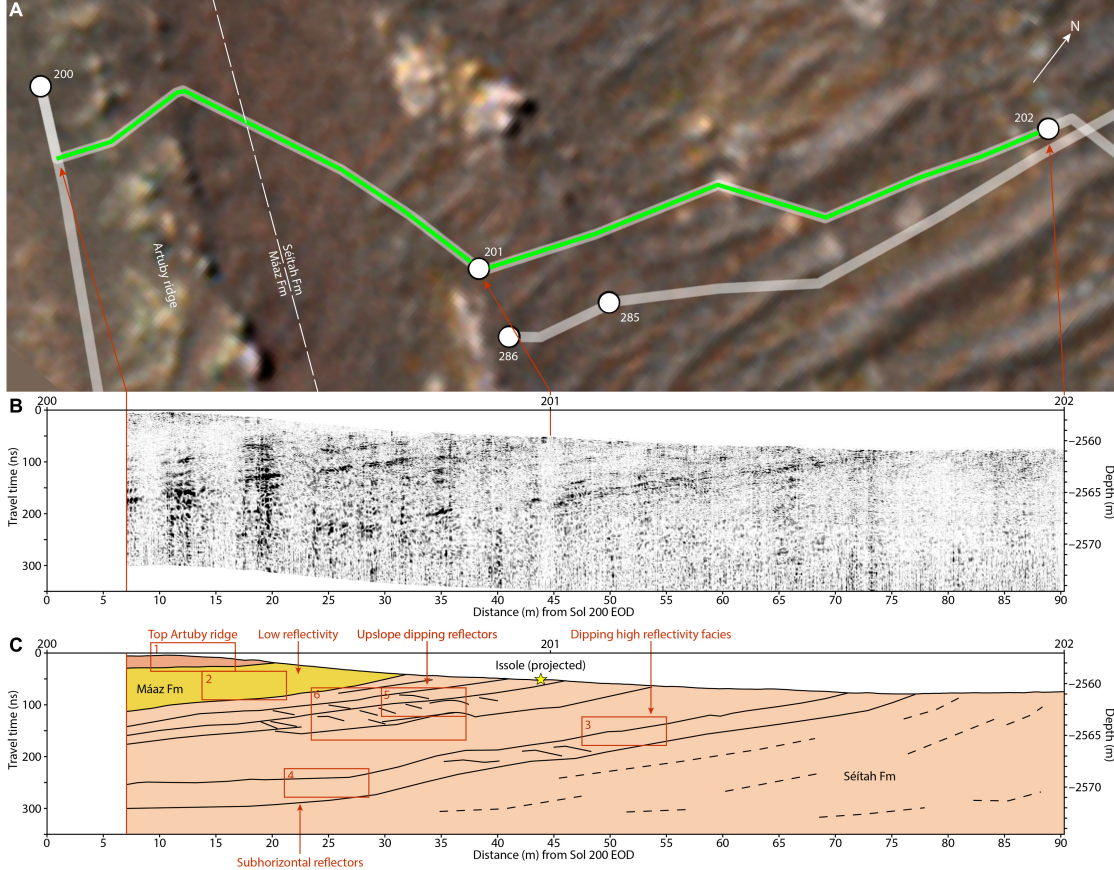


Figure 2.2: RIMFAX radargram and interpretive cross-section along the Brac–Issole traverse. From Hamran et al. (2022), Fig. 3, licensed under CC BY-NC 4.0.

Reflections occur at interfaces where there is a contrast in dielectric properties, i.e., a change in density, porosity, composition, or the presence of buried rocks and sedimentary structures [Ham22]. The fundamental measurement principle of ground-penetrating radar is based on two-way travel time [Ham22]. When a radar pulse is emitted, it propagates through the subsurface until it encounters an interface that reflects part of the energy back to the antenna. The time interval between transmission and reception is recorded. If the wave propagation velocity in the subsurface is known or estimated, the depth of the reflector can easily be calculated [Ham22]. RIMFAX measurements not only provide geometric information about subsurface layering, but also allow inferences about material

properties such as bulk density and composition [Ham22].

2.2.2 Traverse Paths, Soundings and Radargrams

During rover operations, RIMFAX acquires soundings at regular spatial intervals while the rover is driving [Ham22]. Each sounding consists of a time series of reflected power samples over the selected bandwidth [Ham20]. If successive soundings are arranged according to their acquisition position along the rover’s path, they form a two-dimensional image known as a radargram [Ham22]. As also illustrated in Figure 2.2, the horizontal axis of a radargram corresponds to distance along the traverse, while the vertical axis represents two-way travel time or a converted depth estimate [Ham22]. Importantly, RIMFAX does not acquire a volumetric 3D dataset; instead, it records a sequence of 1D soundings along a single spatial trajectory [Ham22]. The resulting data therefore form a 2D profile embedded in 3D space. In addition to continuous traverse measurements, stationary measurements can be performed [Ham22]. For scientific interpretation and visualization, accurate knowledge of the rover position and timing is essential. Each sounding is associated with metadata, including the rover’s position within the local site coordinate frame, which can be transformed into Mars latitude, longitude, and elevation [HP21].

2.2.3 PRo3D

PRo3D, short for Planetary Robotics 3D Viewer, is an interactive visualization platform developed to support planetary science missions, particularly those of NASA and ESA on Mars [VRV25]. It enables scientists to explore large, high-resolution 3D terrain reconstructions derived from orbital and rover-based imagery [VRV25]. PRo3D integrates various mission data directly into a spatially accurate 3D environment. The system is designed to handle very large datasets efficiently and provides tools for measurements, annotation, and collaborative analysis. It therefore supports scientific interpretation, mission planning, and data-driven decision-making in planetary exploration [VRV25].

State of the Art

This chapter provides an overview of existing tools and methods relevant to the visualization of subsurface data through cross-sections. It discusses systems that either directly support cross-sectional visualization or contribute techniques that are useful for constructing and rendering cross-sections in our setting. For each tool, we examine which aspects are useful for our visualization problem, where the limitations lie, and why these approaches do not fully satisfy the requirements of PPro3D.

3.1 VTK and VTK Cutter

VTK, short for Visualization Toolkit, is an open-source software system that provides advanced features for 3D computer graphics, visualization, modeling, and data analysis [SML06]. It provides a rough baseline and an approach for a partial solution for our work. One component of VTK that is relevant for our problem is the filter *vtkCutter*. This filter takes a given VTK data object as input, applies a cutting operation, and outputs the resulting geometry as a new dataset [Uni24, Kit24]. The cutting operation is defined by an implicit function, such as *vtkPlane*, *vtkSphere*, or *vtkBox*, together with one or more cut values c . The filter evaluates this implicit function on the input geometry and extracts the intersection primitives between the geometry and the cutting surface [Kit24]. As a consequence, the cell dimension of the dataset is reduced: for example, cutting a 2D surface mesh produces 1D intersection lines [Kit24].

vtkCutter provides a useful functionality because it is generally the mechanism for extracting intersections between geometric datasets and analytical surfaces. VTK also integrates these operations into a visualization pipeline and provides rendering functionality that allows the resulting geometry to be visualized directly [SML06]. Furthermore, VTK offers a method for slicing a mesh along a polyline through the *SliceAlongPolyline* functionality [Kit26], which demonstrates that the framework supports more flexible cutting definitions than simple implicit functions.

However, these capabilities do not fully address the requirements of our application. The primary goal of *vtkCutter* is to extract intersection primitives rather than to modify or classify the input mesh. As a result, the filter only returns the intersection curves and relies on additional processing steps in a larger pipeline to produce usable geometry [Uni24, Kit24]. In our case, the central task is not only to compute the intersection but also to deterministically classify the terrain mesh fragments so that one side of the polyline is removed while the other remains visible. This problem is not solved by *vtkCutter*.

Another limitation is that slicing along a polyline in VTK requires the polyline to be non-intersecting [Kit26c]. In our setting this restriction is problematic, because the input polyline may originate from the Perseverance rover traverse path or may be drawn interactively by a scientist. In both cases, the polyline can contain complex shapes, including loops or self-intersections, and therefore cannot be constrained to a non-intersecting form.

Furthermore, VTK does not directly provide functionality to construct the curtain geometry that is required for visualizing the cross-section surface. Although it is possible to approximate such a curtain through additional modeling steps, these workarounds significantly increase the complexity of the pipeline [SML06]. Our goal, in contrast, is to construct a cross-section representation that not only reveals the subsurface along a selected path but also provides a curtain geometry to directly support embedding radar data into the 3D terrain context, while requiring as little additional user input as possible.

Finally, the cross-sections used in our application are defined either by user-drawn polylines or by predefined rover traverse paths. These paths can be geometrically complex and cannot easily be expressed as simple implicit functions. As a result, the standard cutting mechanisms provided by VTK are not well suited for the flexible and exploratory workflow required for the P_{Ro}3D application.

3.2 ParaView

ParaView is an open-source visualization application built on top of VTK and provides an interactive Render View for showing derived geometry directly as a 3D scene [Kit26b, Kit26a]. Its *SliceAlongPolyline* filter cuts a dataset along a curve and outputs the cut result [Kit26c, WA20]. However, slicing with ParaView, just like with VTK, reduces the cell dimensions of the input dataset [Kit26d]. So again, we would get one-dimensional intersection segments and not a directly renderable cut surface as output. The output can still be rendered, but typically as lines or tubes, and is therefore mainly useful to visualize the cut trace [Kit26a].

ParaView does not solve the remaining core requirements: it neither classifies the input surface to discard one side of the polyline nor does it automatically build a curtain geometry attached to the cut edge. A curtain-like geometry can only be approximated

via additional modeling steps such as linear extrusion and extra pipeline work [Law08], which is not suitable for our use case.

VTK and ParaView demonstrate that slicing is a standard operation in scientific visualization, but their cutting filters mainly extract intersection geometry and do not directly address deterministic mesh classification, curtain construction, or radargram mapping along arbitrary user-defined polylines.

3.3 CGAL: The Computational Geometry Algorithms Library

CGAL, short for Computational Geometry Algorithms Library, is a C++ library that provides a broad collection of robust geometry algorithms. In contrast to VTK, CGAL is not a visualization tool and does not provide a rendering pipeline. Instead, it focuses on geometry computation, which later can be rendered with the application's own rendering system.

For the task of cutting a triangulated surface mesh, CGAL provides the class `CGAL::Polygon_mesh_slicer` [CGA26c]. It intersects a triangle surface mesh with a plane and records the intersection as a set of polylines, because the plane-mesh intersection can consist of multiple connected components [CGA26c]. It also handles degenerated cases, such as the intersection being a single point [CGA26c]. The class `CGAL::Polygon_mesh_slicer` outputs an explicitly ordered representation of the intersection curves between the plane and mesh, and not a cut version of the input surface [CGA26c].

CGAL uses an AABB tree to speed up geometric queries. An AABB is a box aligned with the coordinate axes that encloses a geometric object. These boxes are organized hierarchically: larger boxes contain groups of primitives, while smaller boxes contain individual primitives or small sets of them. During a query, boxes that cannot intersect the query object are discarded early. Therefore, only a smaller set of primitives must be tested exactly, which is more efficient than checking every mesh edge individually. [Eri05]

The key advantage of CGAL's approach is that it is designed around efficient geometric queries. The implementation for `CGAL::Polygon_mesh_slicer` relies on CGAL's AABB tree package for fast intersection computation [CGA26a]. In more detail, the slicer builds an AABB tree over the mesh edges by wrapping each mesh edge as a 3D segment primitive `AABB_halfedge_graph_segment_primitive` [CGA26b]. This avoids brute-force testing of every edge for intersection and makes repeated slicing queries much simpler [CGA26a]. From a geometric standpoint, the cutting step follows a sign-evaluation idea: for each candidate segment, the plane equation is evaluated at endpoints to determine whether the segment crosses the plane. If a segment does cross the plane, the exact intersection point is evaluated by interpolation. The resulting per-triangle intersection segments are then stitched into polylines representing each connected intersection component [CGA26c].

However, CGAL's slicer is still too narrow in its functionality for the use case of PRo3D. Most importantly, it only supports slicing by a plane [CGA26c] and does not address

cross-sections defined by an arbitrary user-drawn polyline. In our setting, the cross-section must support the visualization of radargram data within the 3D terrain scene and therefore requires a surface representation aligned with the selected path. This is fundamentally not the same as a global cut with a plane. Additionally, CGAL outputs polylines and does not generate a filled curtain surface mesh, nor does it provide a parameterization that could be used to map a radargram texture onto that curtain [CGA26c]. Finally, because CGAL only performs geometric computations, the rendering step is missing completely and would require an additional rendering pipeline.

In summary, CGAL contributes strong and reusable implementations for slicing a triangulated surface mesh with a plane, especially through its efficient use of AABB trees for improved performance [CGA26a] and the production of ordered polylines as intersection output [CGA26c]. However, CGAL does not solve central steps required for our specific problem: it does not construct and render the cross-section curtain surface, does not clip the terrain mesh deterministically with respect to the polyline-defined cross-section, nor does it generate a texture mapping for the curtain.

While CGAL provides efficient and robust algorithms for mesh slicing, its typical usage model assumes that the complete mesh is available in memory. In contrast, our application operates on extremely large planetary terrain datasets that are streamed and processed in smaller patches at runtime. This makes a global preprocessing step with CGAL impractical.

Furthermore, CGAL’s algorithms are designed for CPU-based computation and are not intended for interactive, real-time updates as required in PRo3D. In our setting, cross-sections must be updated dynamically based on user-defined polylines during navigation, which favors a GPU-based approach integrated directly into the rendering pipeline.

CGAL would be suitable for mesh processing on static datasets. However, it does not align with the requirements of interactive visualization and out-of-core rendering for large-scale terrain data.

3.4 libigl

libigl is an open-source C++ library for geometry processing, designed to provide small, reusable building blocks for mesh processing algorithms [lib26c, lib26d]. In contrast to VTK, libigl does not provide a full visualization or rendering pipeline, but focuses, like CGAL, on geometric computations that can be integrated into other applications [lib26c]. Regarding mesh cutting and cross-sections, libigl does not provide one canonical plane-slicer comparable to `vtkCutter` or `CGAL::Polygon_mesh_slicer`. Instead, a common approach in libigl is to formulate a plane cut as an isoline extraction problem on a triangulated surface mesh. For a plane with a normal n and an offset d , a scalar field $F(v) = n * v + d$ is defined on the mesh vertices. The intersection of the plane with the mesh corresponds to the 0-isoline of this scalar field. libigl provides `igl::isolines` for extracting isolines of a scalar field defined on a triangle mesh. In practice, this

equals a marching-triangles-style procedure that generates line segments on faces whose vertex scalar values lie on opposite sides of the chosen iso-value [lib26d]. This approach is attractive because it is conceptually simple and fits well into geometry processing workflows [lib26c]. Because it directly returns the intersection curve, it generalizes beyond planes to any scalar field defined on vertices.

libigl does not provide a dedicated high-level slicer for arbitrary polyline-defined cross-sections. Instead, it offers lower-level geometry-processing building blocks such as scalar-field isoline extraction. In the simple plane case, the cut curve can be obtained directly as the 0-isoline of the plane’s signed-distance field. More generally, this formulation can be extended to other user-defined scalar fields, but deriving a scalar field whose isoline matches an arbitrary user-drawn polyline is itself an additional problem and is not provided as a standard end-to-end solution in libigl. Moreover, the extracted isoline alone does not automatically split the original mesh into a clipped terrain representation, construct the curtain surface, or provide a parameterization for mapping radargram data.

This limitation is explicitly discussed by users in the libigl repository, who note that the isoline vertices are new points and not part of the mesh and therefore cannot be directly used with mesh cutting or splitting operations without additional remeshing or edge-splitting steps [lib26a, lib26b].

Furthermore, even for plane slicing, libigl’s isoline output is not automatically a single ordered polyline with clear semantic meaning regarding which side of the line each surface fragment lies on. However, both topologically ordered segments and classification of the surface fragments are crucial for cutting the surface correctly and deterministically. Finally, as with CGAL, libigl does not address rendering; any visualization must be implemented using an external rendering system [lib26c].

In summary, libigl provides a useful technique to compute plane–mesh intersection curves by reducing the problem to extracting isolines of a scalar field [lib26c]. For the PPro3D cross-section use case, however, libigl does not provide the central missing steps: it does not deterministically classify the fragments and clip the surface mesh with respect to an arbitrary input polyline, nor does it construct the surface representation required to display radargrams along the cross-section in a spatially meaningful way. Additionally, it does not provide a pipeline for rendering.

3.5 BGS Groundhog

BGS Groundhog is a professional geoscience interpretation and modeling tool developed and maintained by the British Geological Survey (BGS) [Bri25]. It is explicitly positioned as a GeoScientific Information System (GSIS) for subsurface interpretation work. The external user manual describes it as a tool to gather, display, filter, and edit subsurface-relevant data such as boreholes, geological map linework, interpreted cross-sections and faults, and reference layers such as elevation models and images [WRRH15]. The BGS software page likewise presents Groundhog as a tool for the visualization, modeling,

and interpretation of geological and environmental data, highlighting its relevance for scientific as well as applied geological workflows. [Bri25]. At the same time, Groundhog is currently described in its documentation as a legacy product that is no longer actively supported [Bri], which limits its relevance as a directly adoptable technical solution today.

From the perspective of cross-sections, Groundhog is interesting because it reflects a real geological workflow: the user defines a cross-section by drawing a line-section (alignment) in map space, then opens a dedicated cross-section window to interpret and draw within the section plane [WRRH15].

However, Groundhog's core step in its cross-section generation is not cutting a triangulated surface mesh. Instead, the manual explains that the terrain or topographic profile in a cross-section is, by default, auto-generated by sampling and projection [WRRH15], rather than constructing a clipped 3D terrain mesh and a textured curtain surface suitable for GPU rendering. Consequently, while Groundhog provides strong domain-relevant state of the art for how geologists define and work with cross-sections [WRRH15], it does not directly address the PPro3D-specific computational requirements.

3.6 GemPy

GemPy is an open-source Python framework for structural geological modelling that follows an implicit potential-field approach for building complex 3D geological models [dlVSW19, Gem25b]. It uses interface points and orientation or gradient data to interpolate a scalar field, where geological interfaces are represented as iso-surfaces of that field [dlVSW19].

In this context, a cross-section is typically generated by sampling the model solution on a plane or along a custom trace, and then visualizing lithology blocks, scalar fields, or interface lines in 2D [dlVSW19]. GemPy explicitly supports plotting sections based on custom section traces via its 2D plotting utilities [Gem25a], and it also provides an API to define section grids from start and end points plus a resolution [Gem25c].

For our work, GemPy is therefore relevant mainly as another example of an established scientific geology workflow for creating and inspecting subsurface models through sections [dlVSW19]. However, GemPy does not address the central technical problem considered in this thesis: it does not explicitly slice or clip a triangulated surface mesh using an arbitrarily drawn polyline, nor does it construct a renderable curtain geometry with a stable UV parameterization required to map a radargram. Instead, it focuses on evaluating an implicit or volumetric model on section domains and visualizing the resulting scalar or categorical fields [dlVSW19, Gem25a].

3.7 Comparing Existing Tools

Table 3.1 compares the software systems discussed in Sections 3.1–3.6 according to the requirements derived from the research questions. The comparison distinguishes

System	Polyline	Complex Paths	Polyline-Based Terrain Classification	Curtain and Mapping	Interactive Large Terrain
VTK	Limited	No	Limited	No	Limited
ParaView	Limited	No	Limited	No	Limited
CGAL	No	No	Limited	No	No
libigl	Limited	No	Limited	No	No
BGS Groundhog	Yes	Limited	No	No	No
GemPy	Limited	Limited	No	No	No
Our work	Yes	Limited	Yes	Yes	Yes

Table 3.1: Comparison of the surveyed systems with respect to the requirements of the proposed cross-section pipeline.

between support for polyline-defined sections, handling of geometrically complex paths, classification and removal of one terrain region, construction and parameterization of a curtain surface, and suitability for an interactive large-terrain workflow. The entry “Limited” indicates that a capability is available only under specific restrictions or does not provide a complete solution for the considered use case. The research algorithms discussed in the following sections are not included because they address individual geometric subproblems rather than providing directly comparable visualization systems.

The following sections discuss general geometric approaches relevant to constructing cross-sections on triangulated terrain meshes

3.8 Implicit Slicing for Functionally Tailored Additive Manufacturing

The method proposed by Steuben et al. [SIM16] is an algorithm for implicit slicing in additive manufacturing. The goal of their slicing algorithm is to compute a parametric toolpath and associated commands that direct an additive manufacturing system to produce a physical realization of a 3D input. Their method enables computationally efficient slicing of arbitrarily complex geometries [SIM16].

Steuben et al. use an implicit, field-based slicing method based on level sets of scalar field functions [SIM16]. Toolpaths are generated as contours of scalar fields defined over each layer.

In addition to slicing, they compute component infill properties such as stress, strain, and temperature, integrating process-aware design heuristics into the slicing workflow [SIM16]. While this represents an important contribution for additive manufacturing, it is not directly relevant for our work. We do not require interior infill computation or level-set-based slicing; instead, we focus on the extraction and processing of slicing contours.

Their slicing algorithm computes triangle–plane intersections using standard segment–plane intersection tests [Eri05]. The output of these intersection tests is a set of line segments γ_i without any inherent ordering [SIM16].

To organize these segments, Steuben et al. reinterpret the set as a graph in which segment endpoints form the vertices and the line segments form the edges [SIM16]. They then

apply the classical connected-components algorithm by Hopcroft and Tarjan [HT73] to partition the planar slice regions into contiguous ordered subsets. Each subset $\gamma_{i,j}$ denotes one connected boundary component of the i -th slice, for example one outer contour or one hole. Hopcroft and Tarjan’s Algorithm 447 is based on depth-first search and runs in linear time with respect to the number of vertices and edges [HT73].

By utilizing this graph-based structure, Steuben et al. reduce the problem of organizing unordered segments to the well-studied problem of finding connected components in a graph [SIM16, HT73]. The resulting γ_{ij} are cohesively ordered loops of line segments suitable for toolpath generation.

From the perspective of P_{Ro}3D, this approach addresses one of the key subproblems of mesh slicing: transforming unordered intersection segments into ordered contour loops. However, for our use case, this is not sufficient. We cannot stop once the line segments are ordered; we must additionally reconstruct the boundary of the clipped terrain mesh and construct and attach a triangulated curtain geometry along that boundary.

Furthermore, Steuben et al.’s [SIM16] method is designed for additive manufacturing, where slicing geometries are typically simple, planar, and layer-wise. In contrast, our use case requires handling arbitrary user-defined polylines, which may also self-intersect. Their workflow is fundamentally layer-wise planar slicing for toolpath generation.

In summary, Steuben et al. provide an efficient and well-structured solution for organizing plane–mesh intersection segments into connected contour loops, but their method does not address the higher-level geometric reconstruction and rendering requirements central to the P_{Ro}3D cross-section pipeline.

3.9 An Optimal Algorithm for 3D Triangle Mesh Slicing

Minetto et al. [MVS⁺17] present an asymptotically optimal algorithm for slicing a triangular mesh with planes. Their method separates the process into two stages: first, computing plane–mesh intersection segments, and second, assembling these unordered segments into ordered contour loops using a hash-based adjacency structure.

This approach efficiently solves one of the central subproblems of mesh slicing, namely the construction of ordered contours from an unordered set of segments. However, the pipeline stops at contour generation and does not address further steps such as reconstructing the clipped mesh boundary or generating additional geometry.

Furthermore, the method is restricted to planar slicing, typically with parallel planes. In contrast, our use case requires cuts along arbitrary user-defined polylines, which introduces geometric and topological cases not covered by this approach. Therefore, the algorithm is not directly suitable for the P_{Ro}3D cross-section pipeline.

3.10 Geometric Foundations of the Classification Problem

Foley, van Dam, Feiner and Hughes' *Computer Graphics: Principles and Practice* [FvDFH90] is one of the foundational textbooks of computer graphics. While it does not present a ready-made solution to our Pro3D cross-section pipeline, it provides crucial theoretical and algorithmic background for two central components of our work. It describes how surface geometry can be extracted from implicit representations and discusses point-in-polygon and point-location problems. In this sense, the book contributes theoretical foundations rather than a complete application-level workflow [FvDFH90]. For surface extraction from an implicit representation, the book assumes the input to be an implicit field. The mesh is then extracted by, e.g., marching cubes or triangle fan to produce a renderable triangle mesh. In our case, the input mesh is already triangulated, but as shown later, we use a triangle fan to triangulate the polygon after extracting it from the input terrain mesh via cutting. They state that the point-in-polygon problem is well understood in planar versions but less so in higher dimensions. We utilize that knowledge by projecting the relevant geometry in our application to 2D before solving the point-in-polygon problem. They show, it is solvable in linear time for simple polygons and faster with preprocessing for convex and star-shaped polygons [FvDFH90]. The book shows that the cost of classification grows linearly with the complexity of the polygon, which makes it predictable and scalable. This is important for our use case, if the input polyline gets denser.

The point-in-polygon problem is therefore the natural abstraction for classifying terrain fragments with respect to a user-defined polyline. Based on this observation, we formulate our approach as a 2D point-in-polygon problem after projecting the geometry into a suitable plane.

3.11 The 2D Point-in-Polygon Problem

The core computational problem for our use case is to classify the terrain fragments left or right of the polyline so it can then be cut deterministically. None of the prior listed tools solves this problem to a degree that satisfies our requirements. However, we found a way to reduce it to a 2D point-in-polygon problem by taking one user input, the reference point, additionally to the polyline. This way we can form a closed polygon. This approach is not novel and is discussed in various works in the literature [Shi62, HA01, GG17]. Given the closed polygon P and the query point Q , the task is to determine whether Q lies inside or outside P . While this sounds trivial for simple convex polygons, it becomes non-trivial for arbitrary and especially self-intersecting polygons. We ran into exactly that problem when testing earlier implementation attempts with more complex polygons. The following discusses the two most influential contributions to the theoretical and algorithmic basis of our implementation. *Shimrat's 1962 Algorithm 112* [Shi62] forms a historic baseline for ray-crossing, and Galetzka & Glauner's 2017 paper [GG17] presents a robust even-odd algorithm for complex polygons.

3.11.1 Historic Baseline: Shimrat’s Algorithm 112

One of the earliest formal descriptions of the point-in-polygon problem in computer science literature is *Algorithm 112: Position of Point Relative to Polygon* by Shimrat (1962) [Shi62]. Shimrat presents a Boolean procedure that determines whether a point (x_0, y_0) lies inside a simple closed polygon defined by vertices $(x[i], y[i])$. The algorithm first closes the polygon and then shoots a horizontal ray to the right from the point it is testing. It counts how many polygon edges the ray intersects, and while doing that, it toggles the Boolean variable for each intersection [Shi62]. If the number of intersections is odd, the point in question is inside the polygon; if even, it is outside. This is the classical even-odd rule, also known as ray-crossing algorithm. Shimrat’s algorithm is quite elegant due to its simplicity and linear runtime $O(n)$ [Shi62]. However, the algorithm assumes that the point in question does not lie exactly on an edge and that the polygon is not self-intersecting. Special cases such as ray-vertex coincidences or horizontal edges require careful handling, which is not fully addressed in the original presentation. Since we cannot assume any preconditions for our polyline in P_{Ro}3D, Shimrat’s algorithm is not completely suitable for us. However, despite its limitations, Shimrat’s algorithm remains the conceptual foundation of most modern even-odd approaches.

3.11.2 A Robust Even-Odd Algorithm for Complex Polygons

A more robust and practically oriented contribution is Galetzka & Glauner’s *A Simple and Correct Even-Odd Algorithm for the point-in-polygon Problem for Complex Polygons* [GG17]. The authors focus explicitly on complex polygons, especially on self-intersecting polygons. The self-intersecting edges of these polygons may create holes or ambiguous regions which often cause the naive even-odd implementation to fail due to incorrect handling of special cases. The algorithm proceeds as follows: First, the polygon is translated, so the query point becomes the origin $(0, 0)$. If the query point lies exactly on a vertex or edge, the result is immediately classified as inside. Next, the positive x-axis is used as the ray for intersection testing [GG17]. While iterating over the polygon edges for the ray-intersection test, vertices lying exactly on the x-axis are skipped. If necessary, auxiliary edges are created between non-axis vertices to replace ambiguous edge sequences. The key innovation lies in the treatment of vertices that lie exactly on the ray. Instead of allowing ambiguous intersection cases, such as counting a vertex twice or not at all, the algorithm skips such vertices and connects the nearest valid ones with an auxiliary edge. Depending on the configuration, it then intersects either the positive half-axis or the entire x-axis. This guarantees that every processed edge either intersects the ray or is clearly not intersecting it. Ambiguous cases are eliminated by construction. Additionally, the paper provides proof of correctness by systematically analyzing all possible quadrant combinations of vertex configurations. It also proves correctness for degenerate cases such as vertices lying exactly on the ray, overlapping or collinear edges, and self-intersecting polygons. In addition, the proof covers all possible configurations of consecutive polygon vertices around the query point, which determine whether an edge should be counted as intersecting the ray [GG17]. The algorithm, like

Shimrat's, requires in linear time $O(n)$ for n vertices and visits each vertex exactly once.

In our P_{Ro}3D case, the cross-section polyline is either user-defined and unrestricted or comes from an original and unrestricted traverse path of the Perseverance rover. The polyline may be self-intersecting and may contain nearly horizontal segments. If inside–outside classification is inconsistent, the resulting clipped terrain may contain visual artifacts such as holes or fragment islands. The naive ray-casting approach of Shimrat [Shi62] provides the conceptual basis but does not sufficiently address degeneracies. The robust strategy of Galetzka & Glauner [GG17] provides a deterministic and well-analyzed solution suitable for complex polygons.

Our Method

4.1 Method Overview

This chapter develops the geometric methods required to answer RQ1 and RQ2. Sections 4.2 to 4.3 investigate how the terrain can be classified and clipped relative to the cross-section path, thereby addressing RQ1. Section 4.4 describes the construction and parameterization of the curtain surface required for RQ2. The architectural realization of these methods and their suitability for later integration into PPro3D, as required by RQ3, are discussed in Chapter 5 and evaluated in Chapter 6.

4.2 Explored Clipping Strategies

This section describes the implementation techniques explored for constructing cross-sections, their drawbacks, and the final solution.

While developing, we explored three different approaches:

- a segment-wise clipping strategy based on tiled clipping planes (Section 4.2.1),
- a view-dependent heuristic based on nearest points on the polyline (Section 4.2.2),
- and a point-in-polygon-based classification that provides the final solution adopted in this work (Section 4.3).

4.2.1 Tiled Clipping Planes

Our first approach of cutting the terrain mesh was to compute one clipping plane for each segment of the polyline and to discard the part of the terrain that lies on the negative side of the selected plane. We tested this approach with a hardcoded polyline.

The process starts on the CPU, which builds a plane for every consecutive pair of points. Each plane contains the corresponding line segment and the approximate planetary up direction. The approximate planetary up direction is a locally constant vector that represents the vertical direction of the terrain, pointing away from the terrain. Geometrically, this creates a vertical slicing wall along each segment of the polyline.

When the OPC patches are loaded, the implementation determines for each terrain vertex which polyline segment is closest to it. The clipping plane of this nearest segment is then selected, transformed into the patch-local coordinate system, and stored as a custom per-vertex attribute. Listing 4.1 shows the relevant CPU-side steps in simplified form.

Listing 4.1: CPU-side assignment of the nearest segment clipping plane

```
1 let assignPlaneToVertex
2   (up : V3d)
3   (localToWorld : Trafo3d)
4   (worldToLocal : Trafo3d)
5   (polylineWS : V3d[])
6   (vertexLocal : V3f) =
7
8   let vertexWS =
9     localToWorld.TransformPos(V3d vertexLocal)
10
11   let segmentIndex =
12     polylineWS
13     |> Array.pairwise
14     |> Array.mapi (fun i (a, b) ->
15       let ab = b - a
16       let t =
17         Vec.dot (vertexWS - a) ab / Vec.dot ab ab
18       |> clamp01
19
20       let closest = a + t * ab
21       i, (vertexWS - closest).LengthSquared)
22     |> Array.minBy snd
23     |> fst
24
25   let a = polylineWS.[segmentIndex]
26   let b = polylineWS.[segmentIndex + 1]
27
28   let normal =
29     Vec.cross (b - a).Normalized up.Normalized
30     |> Vec.normalize
31
32   let planeLocal =
33     Plane3d(normal, a).Transformed(worldToLocal)
34
35   V4f planeLocal.Coefficients
```

On the shader side, the plane is available as a per-vertex attribute. The vertex position is transformed into view space, and the fragment shader transforms the assigned plane into the same coordinate system. The signed distance between the current fragment position and the plane is then evaluated with a dot product. If the distance is negative, the fragment is discarded; otherwise it is kept. Listing 4.2 shows the corresponding shader step.

Listing 4.2: Signed-distance clipping against the assigned segment plane

```

1 let discardBehindAssignedPlane (v : Vertex) =
2   fragment {
3     let planeView =
4       uniform.CrossSectionToViewSpace * V4d(v.clipPlaneLocal)
5
6     let signedDistance =
7       Vec.dot planeView v.viewPos
8
9     if signedDistance < 0.0 then
10      discard()
11
12     return v.color
13   }
```

Because the sign check is fixed, the implementation always removes the same side of the selected segment plane. In that sense, it clips the input mesh on one side of the hardcoded polyline. However, the nearest-segment selection is only local. It does not define a globally consistent inside–outside region for the full polyline.

Figure 4.1 illustrates the main stages of this workflow. First, clipping planes are placed along the polyline segments (Figure 4.1a). Then, the terrain mesh is clipped so that only one side of the polyline remains visible (Figure 4.1b). Finally, a curtain geometry is added along the cut boundary (Figure 4.1c).

This clipping method works well with the hardcoded polyline because the angles between the line segments are rather flat, and the nearest-plane assignment remains locally consistent for most terrain vertices. However, the method fails for polylines with concave spikes and self-intersections. In such cases, the nearest segment may define a clipping plane whose local signed-distance test does not match the intended global keep/discard region.

Figure 4.2 shows such a failure case. The gray area denotes the region that should remain visible. The dashed lines represent the local clipping planes constructed from the individual polyline segments, and the small gray arrows indicate the normal directions used for the signed-distance test. Since each terrain point is classified only with respect to the plane of its nearest segment, points outside the intended kept region may be wrongly kept, while points inside the intended kept region may be wrongly discarded.

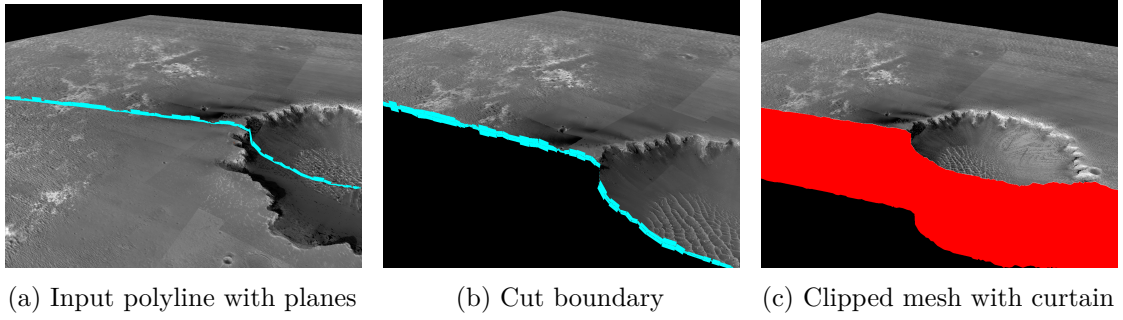


Figure 4.1: Stepwise visualization of the tiled clipping planes workflow. (a) Polyline segments as planes along a polyline on the terrain. (b) The terrain mesh is cut along the polyline, keeping only one side of the mesh. (c) A cross-section geometry is built along the mesh’s cut.

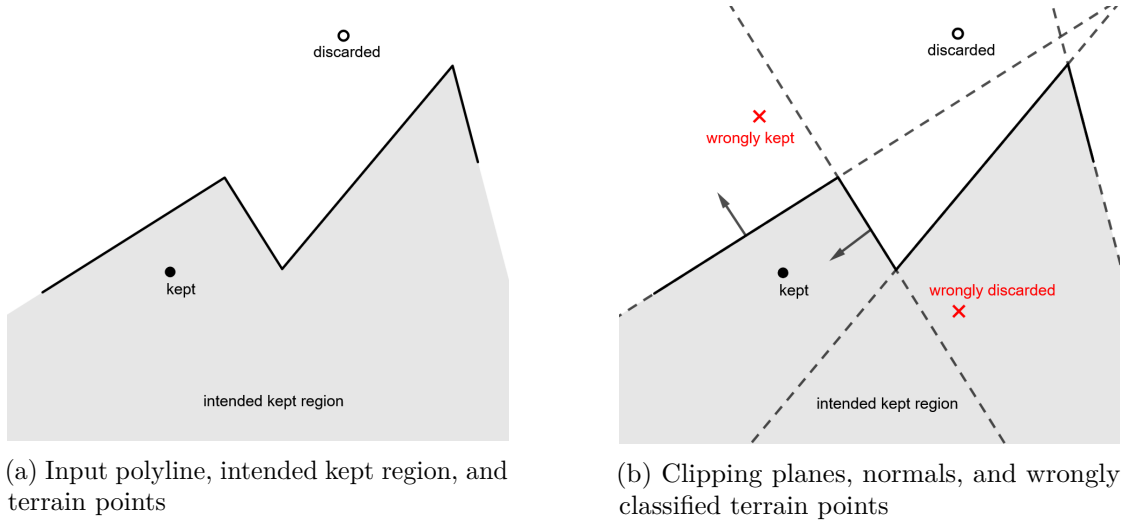


Figure 4.2: Result of the tiled clipping-plane approach for a concave polyline.

4.2.2 Nearest-Point Classification Heuristic

Our next approach was to use the nearest projected point on the polyline for each point on the terrain and the viewing direction to determine the front and back side of the polyline. This method no longer clips by evaluating one explicit plane equation per fragment. Instead, for every terrain vertex, the CPU finds the closest point on the polyline. Then, in the fragment shader, the terrain point is compared to that nearest polyline point in a frozen camera view. If the point lies on the front side of the polyline relative to that frozen viewing direction, the fragment is discarded.

The nearest-point search is performed in a two-dimensional projection of the terrain. For this, both the terrain vertex and the polyline segments are projected into a local 2D coordinate system whose axes span a plane perpendicular to the approximate planetary

up direction. This projected 2D distance ignores elevation differences and only compares the horizontal layout of the terrain point and the polyline.

Listing 4.3 shows the relevant CPU-side part of the implementation. For each terrain vertex, the local patch position is first transformed into world space. The world-space vertex and each world-space polyline segment are then projected into the same 2D basis. The segment parameter t is computed in this 2D projection and clamped to the finite segment. The squared 2D distance is used to select the closest segment. Since only the relative ordering of distances is required, the squared distance can be compared directly and no square root is necessary. After the closest segment has been found, the nearest point is reconstructed on the original 3D polyline segment in world space and transformed back into the local coordinate system of the current terrain patch. This local position is stored as a per-vertex attribute and later used in the shader.

Listing 4.3: Projected 2D nearest-point search for a terrain vertex

```

1 let nearestPointForVertex patch project polylineWS localPosition =
2   let pWS =
3     patch.info.Local2Global.TransformPos (V3d localPosition)
4
5   let p2 = project pWS
6
7   let _, qWS =
8     polylineWS
9     |> Array.pairwise
10    |> Array.map (fun (aWS, bWS) ->
11      let a2 = project aWS
12      let b2 = project bWS
13      let ab2 = b2 - a2
14
15      let t =
16        Vec.dot (p2 - a2) ab2 / Vec.dot ab2 ab2
17      |> clamp01
18
19      let q2 = a2 + t * ab2
20      let qWS = aWS + t * (bWS - aWS)
21
22      (p2 - q2).LengthSquared, qWS)
23   |> Array.minBy fst
24
25   patch.info.Local2Global.Inverse.TransformPos qWS

```

Full 3D Euclidean distance is the straight-line distance between two points in 3D space, including elevation differences. In this approach, however, the nearest point on the polyline should be determined based on the horizontal arrangement of the terrain point and the polyline, not based on their height difference. Therefore, both are projected into a plane orthogonal to the up direction, and the nearest point is found in this 2D projection.

The code stores a frozen view transform to prevent the clipping from changing when the user rotates the camera after defining the path. This transform represents the camera view at the moment the cross-section path is finalized. During rendering, the same frozen transform is used for both the terrain point and its nearest point on the polyline, so both positions are compared in one coordinate system.

In the fragment shader, the terrain point and the corresponding nearest polyline point are transformed into this frozen view space. The shader then computes the vector from the nearest polyline point to the current terrain point and projects this vector, together with the frozen viewing direction, into 2D. Their dot product is used to decide on which side of the polyline the terrain point lies. A positive dot product means that the terrain point lies in the same general direction as the frozen viewing direction. This side is defined as the side to be removed, so the fragment is discarded. If the dot product is negative or zero, the fragment is kept.

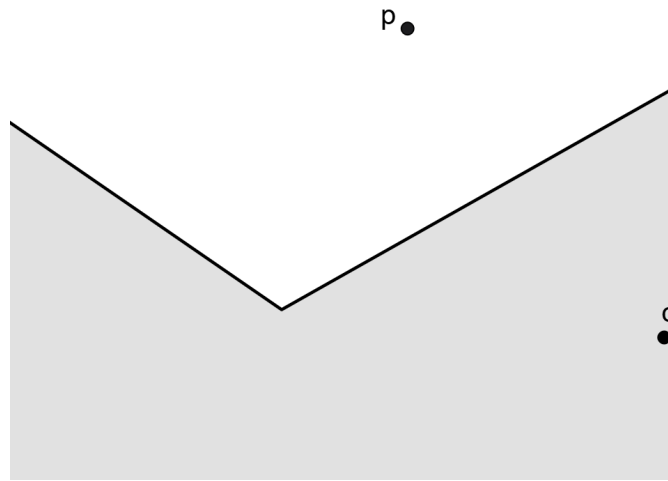
Figure 4.3 illustrates this procedure step by step. Figure 4.3a shows the input polyline and a sample terrain point. Figure 4.3b shows the vector from the terrain point to its nearest point on the polyline together with the frozen viewing direction. Figure 4.3c shows the final dot-product-based classification. In the illustrated failure case, the dot product suggests that the terrain point should be kept, although it lies on the wrong side of the intended cross-section. This demonstrates that a local view-dependent test is not sufficient for robust classification.

4.2.3 Limitations of Local Classification Strategies

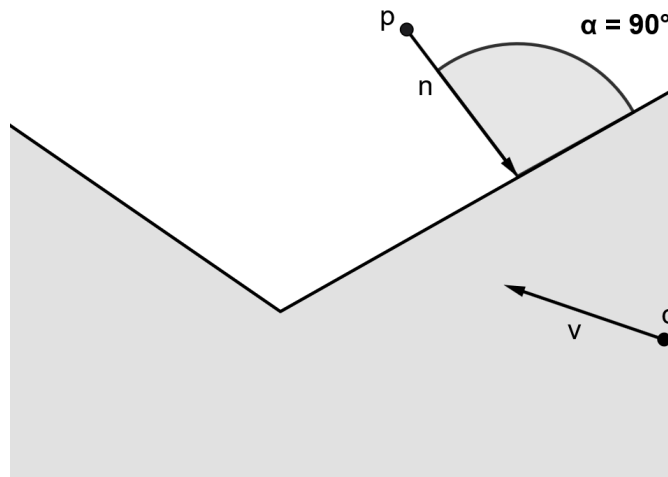
The tiled clipping plane method works well with the hardcoded polyline because neighboring line segments are almost collinear and do not contain sharp turns. Consecutive segments meet at large interior angles, so the nearest-plane assignment remains unambiguous. However, this method fails for polylines with concave spikes and self-intersections. In such cases, a terrain point may be assigned to a nearby clipping plane that does not correctly classify the terrain point. The resulting signed-distance field can therefore be incorrect and inconsistent. As a result, some points that should be removed could be kept, while others could be incorrectly discarded. This leads to holes and falsely rendered terrain elements. A typical failure case is illustrated in Figure 4.2.

The nearest-point classification heuristic too fails for polylines with strong bends, loops, or self-intersections. In such cases, the nearest segment only defines a local side of the path and does not provide a globally consistent inside–outside classification. The problem is especially visible near concave spikes, where different nearby segments may imply different visible sides. If the frozen viewing direction is not approximately perpendicular to the relevant local segment, the dot-product test can classify the wrong side of the cut, leading to incorrectly removed or falsely visible fragments. This behavior is illustrated in Figure 4.3c.

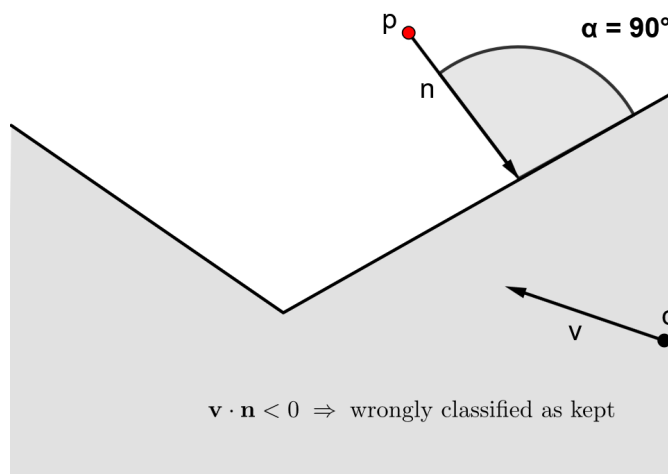
After implementing multiple failing approaches that used only the drawn polyline as input, we concluded that the sequence of points defining the path is not sufficient for



(a) Input polyline and terrain point



(b) Vector to the nearest point on the polyline and the view direction



(c) View-based classification using the dot product

Figure 4.3: Failure case of the view-direction-based classification.

arbitrary cross-section shapes. The polyline specifies where the cut should run, but as an open curve it does not define a closed inside–outside region.

One possible idea would be to extend the polyline at its endpoints so that it separates the terrain into two regions. However, this strategy only works reliably for simple polylines without loops, self-intersections, or sharp turns. For more complex paths, extending the polyline can still produce wrong classifications, because it still does not define globally consistent inside and outside regions.

For this reason, we introduce an additional reference point. Together with the drawn polyline, this point closes the path into a polygon and allows the clipping problem to be formulated as a point-in-polygon classification. As discussed in Section 3.11, research on the point-in-polygon problem provides different solutions for this classification task.

4.3 Point-in-Polygon-Based Classification

Our final approach no longer tries to derive the visible side of the cut from local segment orientation or from the current viewing direction. Instead, the clipping problem is formulated as an inside–outside classification problem. For each terrain point, the method determines whether its position is inside or outside a polygon defined by the cross-section path and reference point. Conceptually, this method differs from the earlier approaches in an important way. The decision is no longer made relative to one nearest segment or one nearest plane, but relative to a globally defined closed polygon. Consequently, all terrain points are evaluated using the same polygon-wide inside–outside criterion. This avoids the potentially contradictory decisions produced by local segment-based heuristics, particularly near concave turns and loops, as well as for self-intersections.

4.3.1 Projection into a Two-Dimensional Coordinate System

The point-in-polygon classification is performed in a two-dimensional coordinate system. For this purpose, a 2D projection basis is constructed from the approximate planetary up vector. This basis spans the local tangent plane of the terrain and is defined by two orthogonal axes perpendicular to the up direction.

Using this basis, any 3D point can be projected into the plane by computing its dot product with the two basis vectors. This produces a 2D coordinate used for the subsequent point-in-polygon classification. Both the polyline points and the reference point are projected from 3D world space into this common 2D coordinate system.

4.3.2 Closing the Polyline with a Reference Point

After drawing or importing the cross-section, it initially is represented as an open polyline. To formulate the classification of terrain fragments as a point-in-polygon problem, this polyline must first be closed into a polygon. For this purpose, the camera position at the moment the user saves the polyline is used as an additional reference point. The open

polyline and reference point are then being projected into the constructed 2D coordinate system. The reference point is then appended to the open polyline to form a closed polygon.

The resulting polygon defines the region used for the clipping decision. Figure 4.4 illustrates how the user-defined polyline is closed into a polygon and how terrain points are classified as inside or outside using a point-in-polygon test.

4.3.3 Inside–Outside Classification and Application to the Terrain

After closing the polygon, we must project all points into the same 2D coordinate system, and then test each terrain point against the closed polygon. This inside–outside test determines whether the terrain point lies inside or outside the polygon.

For every terrain patch, each vertex position is transformed from patch-local coordinates into world space and then projected into the same 2D basis as the polygon. The projected vertex positions are then tested against the polygon. If the point is inside, the vertex receives the value -1 ; if it is outside, it receives the value $+1$.

The result of this classification defines which parts of the terrain remain visible and which parts are clipped. Thus, the clipping decision is based on the global polygon constructed from the user-defined cross-section and the reference point, rather than on local segment directions or view-dependent heuristics.

If no usable clipping polygon can be constructed, the implementation does not perform any clipping and marks all vertices as inside. This can happen, for example, if the saved cross-section contains not enough points to construct a polygon. This fallback is not part of the classification method itself, but a safety mechanism: in an undefined input state, the terrain should remain fully visible instead of being clipped accidentally.

We use the function `Polygon2d.Contains`, which is part of the Aardvark framework [Aar24], to test each vertex position. The method performs the point-in-polygon test by forming a signed triangle fan using the first polygon vertex as common v_0 . The fan triangles are not required to form a valid non-overlapping triangulation. For concave polygons, some triangles may be oppositely oriented; the triangles contribute negatively to an accumulated counter. Therefore, inverted fan triangles are handled as part of the signed containment test. The method assumes a consistent vertex order: counter-clockwise by default, or clockwise if the *CCW* parameter is set to *false*. The choice of v_0 is fixed by the implementation to `poly[0]`; for well-formed polygons it should not change the theoretical result.

The function `Polygon2d.Contains` iterates over the triangles (v_0, v_i, v_{i+1}) of the triangle fan. For each triangle, the helper function `InsideTriangleFlags` evaluates oriented edge tests and returns the signs of the three half-space tests. If all three signs are equal, the query point lies inside the corresponding signed fan triangle. The sign is then added to an accumulated counter. Thus, positively oriented fan triangles contribute $+1$, while oppositely oriented fan triangles contribute -1 . For a counter-clockwise polygon, a

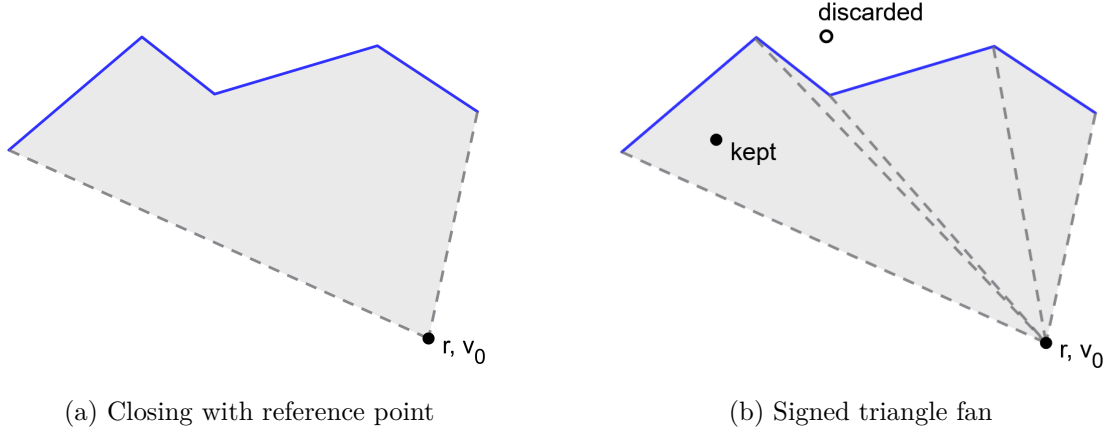


Figure 4.4: Point-in-polygon-based formulation of the clipping problem. Here, the reference point and v_0 share one vertex.

positive final counter classifies the point as inside; for a clockwise polygon, the inside condition is a negative counter. The classification is computed per vertex on the CPU and the result is passed to the GPU as a vertex attribute.

Since the fragment shader operates per fragment, it receives an interpolated value of this attribute from the rasterizer. A sign change of the interpolated value inside a triangle represents the approximate cut boundary on that triangle. Fragments with a positive interpolated value are discarded, while fragments with a negative value remain visible.

This means that the point-in-polygon test itself is exact only at the terrain vertices where it is evaluated. Between vertices, the visible boundary is approximated by linear interpolation across the triangle. Therefore, the method provides a practical clipping solution for the rendered mesh, but the boundary is not recomputed by an exact point-in-polygon test for every fragment. The shader itself only applies the classification that was precomputed on the CPU.

Figure 4.4 summarizes the point-in-polygon formulation used for the final clipping approach. Figure 4.4a shows how the open cross-section path is closed by adding the reference point r , resulting in a polygonal region. Terrain points inside this region are classified as kept, while points outside the region are classified as discarded. Figure 4.4b illustrates the signed triangle fan used by `Polygon2d.Contains`: the first polygon vertex v_0 acts as the common anchor, and each triangle (v_0, v_i, v_{i+1}) contributes to the accumulated inside–outside decision.

4.4 Curtain Geometry Concept

After the cross-section path is finalized, it is converted into a visible geometric surface, a curtain, representing the vertical cross-section. This curtain follows the selected path on the terrain and extends vertically along the planetary up direction.

The curtain geometry is generated along the polyline defining the cross-section. Each segment p_0, p_1 of the polyline is extruded vertically along the planetary up vector to form a quad representing a vertical slice of the subsurface. The resulting triangles form a continuous curtain surface.

To support proper texturing along the curtain, arc-length-based texture coordinates are computed along the polyline. This ensures that textures appear evenly distributed along the curtain even when the polyline segments have different lengths. Radargrams or other visual representations of the Martian subsurface can then be placed onto this curtain so that the recorded data becomes visible in its spatial context.

Implementation

5.1 Aardvark and PPro3D Integration

The application is implemented using the *Aardvark Platform*, an open-source framework for visual computing that combines a large collection of libraries for algorithms, data structures, rendering, and user interaction [VRV24, Aar24]. Aardvark is a cross-platform library and its components are modular and reusable [Aar24]. It originated in 2006 at VRVis as a growing collection of solutions for interactive, high-performance visualization and continues to evolve with ongoing research projects [VRV23].

Aardvark provides a unified ecosystem for building visualization systems, including components for rendering, scene graphs, mesh handling, and user interfaces. Its rendering engine, *aardvark.rendering*, offers abstractions for GPU resource management and shader programming. It enables flexible composition of rendering pipelines while maintaining efficient execution. Additionally, the library *aardvark.base* provides mathematical foundations such as vectors, matrices, and geometric operations, which are used consistently throughout the platform.

Over time, the framework shifted from C# towards functional programming concepts to improve abstraction, reusability, and robustness [VRV23]. As a result, applications built on Aardvark, including PPro3D, are typically implemented in F#. The platform is specifically designed to handle very large datasets by efficiently utilizing hardware resources, enabling significantly faster processing compared to conventional approaches [VRV24].

5.2 Implementation Pipeline Overview

Figure 5.1 provides an overview of the implementation pipeline used for the final cross-section approach. The input is either a manually drawn path or an imported rover

traverse. Together with an additional reference point, the finalized path provides the geometric input for the terrain classification and curtain construction.

The first part of the pipeline addresses RQ1, which concerns the consistent classification of the terrain region to be retained or removed. The path and reference point are transformed into a common coordinate system and projected into a local two-dimensional plane. The projected path is closed with the reference point to form the classification polygon. For every OPC terrain patch, the terrain vertices are transformed from patch-local coordinates into world space, projected into the same two-dimensional coordinate system, and classified as inside or outside the polygon. The resulting classification is stored as a custom per-vertex attribute. During rendering, the GPU interpolates this attribute across the terrain triangles, and the fragment shader discards fragments classified as outside the retained region.

The second part of the pipeline addresses RQ2, which concerns the construction and parameterization of the curtain geometry. The finalized path is converted into a sequence of line segments. Normalized arc-length values are calculated along the path and stored as horizontal texture coordinates. A geometry shader then expands each line segment into a vertical quad. Together, the generated quads form a continuous curtain along the complete path. The arc-length-based horizontal coordinate and the vertical coordinate assigned during shader expansion provide the parameterization required to map a radargram continuously onto the curtain.

The overall implementation design contributes to RQ3, which concerns interactive performance and preparation for later integration into PRo3D. The prototype is implemented in F# using the Aardvark Platform and operates directly on the OPC patch hierarchy used for large terrain datasets. The more expensive point-in-polygon classification is performed on the CPU when terrain patches are prepared or when the classification is invalidated, while the GPU performs only the comparatively lightweight interpolation, clipping, and curtain rendering during navigation. Dynamic level-of-detail selection remains the responsibility of the existing OPC scene graph. This separation allows the method to use the same mathematical types, terrain structures, scene-graph abstractions, and shader mechanisms as PRo3D.

Figure 5.1 therefore summarizes how the three research questions are represented in the implementation: terrain projection, polygon construction, CPU-side classification, and GPU-side clipping address RQ1; curtain generation and texture parameterization address RQ2; and the use of F#, Aardvark, OPC terrain patches, CPU–GPU task separation, and dynamic level-of-detail processing address RQ3.

5.3 Coordinate Spaces and Floating-Point Precision

Since the terrain is represented in Mars-scale world coordinates, position values can become very large. This is problematic for GPU-based rendering, as double precision is often only limitedly supported or considerably less efficient than single precision [Kne17].

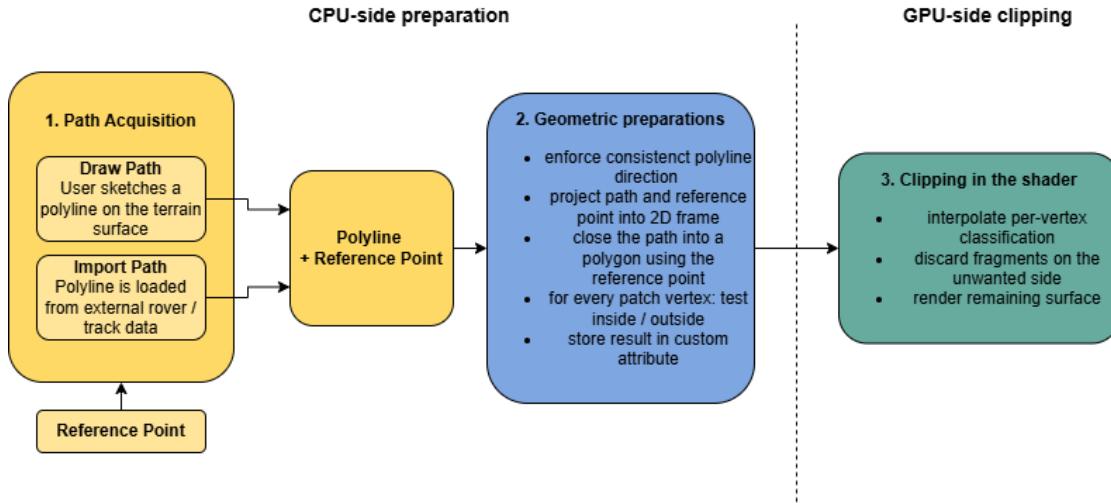


Figure 5.1: Implementation pipeline of the final point-in-polygon-based clipping approach.

To improve numerical stability, all relevant computations for clipping and rendering are therefore performed in view space. In this coordinate system, values are centered around the camera and remain much smaller, which reduces precision-related artifacts and leads to more stable rendering results.

5.4 Terrain Loading and OPC Patch Processing

When the application starts, it initializes the Aardvark rendering environment and prepares the scene graph that will be used to display the terrain data and the interactive cross-section elements. The startup procedure sets up the window, the rendering runtime, and the camera controller that allows the user to navigate the 3D scene.

After initialization, the terrain mesh is loaded from an OPC dataset. OPC is a hierarchical data structure designed for efficiently rendering very large terrain models. Instead of loading the entire dataset into memory, the terrain is organized into smaller patches arranged in a spatial hierarchy. This allows the renderer to load and display only the parts of the terrain that are currently visible.

During startup, the program reads the OPC structure from disk and constructs a scene graph representation of the terrain. Each patch of the dataset contains a triangle mesh with vertex positions and additional attributes. These patches are streamed from disk as needed and integrated into the rendering pipeline. The scene graph then manages the rendering of the terrain by dynamically selecting the appropriate level-of-detail depending on the current camera position.

When a terrain patch is loaded, additional per-vertex attributes required for the cross-section computation are generated on the CPU. These attributes store information that is later used in the shader pipeline to determine whether fragments of the terrain mesh

should be kept or discarded. After these attributes are attached to the patch geometry, the mesh is added to the scene graph and rendered using the configured shader pipeline.

This design allows the application to handle very large planetary terrain datasets efficiently while still supporting interactive operations such as drawing polylines and computing cross-sections directly on the rendered surface.

5.5 CPU-Side Classification

The main part of the point-in-polygon-based clipping approach is computed on the CPU when the OPC patches are loaded. First, a two-dimensional projection basis is constructed from the approximate planetary up vector. This basis is used to project both the user-defined polygon and the terrain vertices into the same 2D coordinate system.

Listing 5.1 shows the construction of this local projection basis and the projection of a 3D point into the resulting 2D coordinate system.

Listing 5.1: Construction of the local 2D projection basis and projection of a 3D point

```
1 let buildProjectionBasis (up : V3d) =  
2   let axisX =  
3     up.Cross V3d.OOI |> Vec.normalize  
4  
5   let axisY =  
6     up.Cross axisX |> Vec.normalize  
7  
8   axisX, axisY  
9  
10 let projectTo2D (p : V3d) (axisX : V3d) (axisY : V3d) =  
11   V2d( Vec.dot p axisX,  
12     Vec.dot p axisY )
```

The function `buildProjectionBasis` derives two orthogonal axes from the approximate up vector. These axes span the local tangent plane used for the two-dimensional classification. The function `projectTo2D` then projects a 3D point into this plane by computing the dot product with both basis vectors.

After the projection basis has been constructed, the polyline points and the reference point are projected into this coordinate system. The projected reference point is appended to the projected polyline, producing the closed polygon used for the classification.

For every terrain patch, the implementation builds a custom per-vertex attribute buffer named *InsideOutsideV4*. To fill this buffer, every vertex position of the patch is transformed from patch-local coordinates into world space and then projected into the same 2D basis as the polygon. The projected vertex position is then tested with `Polygon2d.Contains`, which is part of the Aardvark framework [Aar24].

If the projected vertex lies inside the polygon, the vertex receives the value -1 . If it lies outside, it receives the value $+1$. This value is stored in the first component of the custom

four-component vertex attribute. If no usable clipping polygon can be constructed, the implementation marks all vertices as inside so that no geometry is clipped accidentally. In the implementation, clipping is only enabled if the saved point sequence contains more than three points. If this condition is not fulfilled, or if polygon construction fails, the terrain remains fully visible.

5.5.1 Point-in-Polygon Test

The implementation uses the function `Polygon2d.Contains`, which is part of the Aardvark framework [Aar24], to test each projected vertex position. The method performs the point-in-polygon test by forming a signed triangle fan using the first polygon vertex as common v_0 . The fan triangles are not required to form a valid non-overlapping triangulation. For concave polygons, some triangles may be oppositely oriented; the triangles contribute negatively to an accumulated counter. Therefore, inverted fan triangles are handled as part of the signed containment test.

The method assumes a consistent vertex order: counter-clockwise by default, or clockwise when the *CCW* parameter is set to *false*. The choice of v_0 is fixed by the implementation to `poly[0]`; for well-formed polygons it should not change the theoretical result.

Listing 5.2 shows the relevant part of the `Polygon2d.Contains` implementation used for the inside–outside test.

Listing 5.2: Implementation of `Polygon2d.Contains` used for the inside–outside classification

```

1 let contains (p : V2d) (poly : V2d[]) (ccw : bool) =
2   if poly.Length < 3 then
3     false
4   else
5     let mutable counter = 0
6     let p0 = poly.[0]
7
8     for i in 1 .. poly.Length - 2 do
9       let p1 = poly.[i]
10      let p2 = poly.[i + 1]
11
12      let contribution =
13        signedPointInTriangle p p0 p1 p2
14
15      counter <- counter + contribution
16
17      if ccw then
18        counter > 0
19      else
20        counter < 0

```

The function `Polygon2d.Contains` iterates over the triangles (v_0, v_i, v_{i+1}) of the triangle fan. For each triangle, the helper function `InsideTriangleFlags` evaluates

oriented edge tests and returns the signs of the three half-space tests. If all three signs are equal, the query point lies inside the corresponding signed fan triangle. The sign is then added to an accumulated counter. Thus, positively oriented fan triangles contribute $+1$, while oppositely oriented fan triangles contribute -1 . For a counter-clockwise polygon, a positive final counter classifies the point as inside; for a clockwise polygon, the inside condition is a negative counter.

The classification is therefore precomputed per vertex on the CPU. During rendering, this value is passed to the GPU as an additional vertex attribute and is interpolated across the triangles by the rasterizer.

5.6 GPU-Side Clipping

On the GPU side, the clipping implementation is intentionally simple. The shader does not solve the point-in-polygon problem itself. Instead, it only applies the classification that was precomputed on the CPU and removes fragments that are classified as outside the polygon.

The vertex shader forwards the custom attribute *InsideOutsideV4* to the fragment stage together with the transformed position. Since the attribute is defined per vertex, the rasterizer interpolates it across each triangle. The cut boundary is therefore represented implicitly by the sign change of the interpolated value inside a triangle. Listing 5.3 shows the fragment-shader step that discards fragments classified as outside the polygon.

Listing 5.3: Fragment shader discarding fragments outside the polygon

```
1 fragment {  
2     let inside = v.InsideOutsideV4.X  
3  
4     if inside > 0.0 then  
5         discard()  
6  
7     return v.color  
8 }
```

The first component of *InsideOutsideV4* contains the sign assigned during the CPU-side classification. Negative values represent vertices inside the polygon, while positive values represent vertices outside the polygon. The fragment shader evaluates the interpolated value and discards the fragment whenever *InsideOutsideV4.X* is greater than zero. As a result, fragments classified as outside the user-defined polygon are removed, while fragments classified as inside remain visible.

Since the fragment shader operates per fragment, it receives an interpolated value of the classification attribute from the rasterizer. A sign change of the interpolated value inside a triangle represents the approximate cut boundary on that triangle. Fragments with a positive interpolated value are discarded, while fragments with a negative value remain visible.

This means that the point-in-polygon test itself is exact only at the terrain vertices where it is evaluated. Between vertices, the visible boundary is approximated by linear interpolation across the triangle. Therefore, the method provides a practical clipping solution for the rendered mesh, but the boundary is not recomputed by an exact point-in-polygon test for every fragment. The shader itself only applies the classification that was precomputed on the CPU.

5.7 User Interaction

One goal of the application was to allow users to define cross-sections directly on the terrain in an intuitive way. This allows scientists to explore the Martian subsurface interactively.

The polyline drawing is implemented as a sequence of interactively picked 3D world-space points that lie directly on the rendered terrain. If the user double-clicks the left mouse button, the implementation recovers the 3D point under the mouse cursor on the rendered terrain. It reads the current depth buffer at the mouse pixel, converts the mouse position to normalized device coordinates, combines this position with the sampled depth value, and unprojects the result using the inverse of the view-projection matrix. The resulting world-space point lies on the visible terrain and is appended to the array of the currently drawn points.

The active polyline is rendered from the stored points using its own scene graph. If at least two points are available, the implementation builds an `IndexedGeometry` in `LineStrip` mode. Both `IndexedGeometry` and `LineStrip` are provided by Aardvark. To reduce floating-point issues, the geometry is shifted into a local frame and then transformed back using the model transformation matrix. Pressing *Escape* during the drawing process clears the currently drawn points from the array.

Pressing *C* finalizes the drawn line. The implementation stabilizes the point order by optionally flipping it and stores the finalized line in the array of saved drawn points. In addition, the current camera position is used as a reference point and added to the saved points. The current camera-related data is also stored so that later computation can be performed in a reference frame.

Once the polyline is finalized, it is used as geometric input for two main purposes. First, it is used to construct the curtain geometry that visualizes the cross-section. Second, together with the reference point, it forms the polygon used to classify terrain mesh fragments as inside or outside the cut region.

Besides interactively drawing a polyline on the terrain, the system also allows the cross-section path to be defined by importing a rover traverse. This functionality is particularly relevant for the visualization of RIMFAX radar data, since the subsurface measurements are acquired only along the actual path of the Perseverance rover. As described in Section [2.2.2](#), the radar soundings form a radargram when arranged along

the rover's traverse path. Using the recorded traverse therefore ensures that the generated cross-section corresponds to the spatial locations where measurements were collected.

The traverse path is loaded during program startup from a dataset containing rover positions. These positions are stored as geographic coordinates consisting of latitude, longitude, and elevation values. In order to use them within the 3D visualization environment, the coordinates are transformed into the Cartesian coordinate system of the terrain model using the SPICE Toolkit [ABSW18]. During initialization, the required planetary kernels are loaded, after which each geographic coordinate is converted into a three-dimensional position in the Mars-fixed reference frame.

The resulting sequence of positions forms a polyline in world space. This polyline is inserted into the same data structure that normally stores interactively drawn points. From this point onward, the processing pipeline treats imported and manually drawn polylines identically. The traverse therefore serves as the geometric input for constructing the curtain geometry and for classifying terrain fragments during the clipping step. Because the path corresponds to the real rover trajectory, the resulting curtain geometry aligns with the locations where the radar measurements were acquired.

5.8 Curtain Geometry Generation

After the cross-section path is finalized, the implementation constructs a line geometry from the saved polyline points. The CPU does not generate the full curtain mesh directly. Instead, it submits the polyline as an indexed line list, where each pair of consecutive points forms one line segment. Listing 5.4 shows the essential part of the construction. The vertex positions store the polyline points, while the texture coordinates store the normalized arc length along the path as the horizontal coordinate u .

Listing 5.4: CPU-side construction of the indexed line geometry

```
1 let positions : V3f[] =  
2   ptsLocal |> Array.map V3f  
3  
4 let texcoords : V2d[] =  
5   arcLengths |> Array.map (fun u -> V2d(u, 0.0))  
6  
7 let indices : int[] =  
8   Array.init ((n - 1) * 2) (fun k ->  
9     let i = k / 2  
10    if k % 2 = 0 then i else i + 1  
11  )  
12  
13 IndexedGeometry(  
14   Mode = IndexedGeometryMode.LineList,  
15   IndexArray = indices,  
16   IndexedAttributes =  
17     SymDict.ofList [
```

```

18         DefaultSemantic.Positions, positions :> Array
19         DefaultSemantic.DiffuseColorCoordinates, texcoords :>
    Array
20     ]
21 )

```

The line geometry is then passed into the shader pipeline, where the curtain surface is generated. As shown in Listing 5.5, the geometry shader expands each incoming line segment into one vertical quad. The original segment endpoints p_0 and p_1 form the upper edge of the quad. The lower edge is generated by subtracting an offset vector from both endpoints. This offset is computed from the normalized planetary up vector in view space and the predefined curtain height.

The shader emits four vertices for each segment: two vertices for the upper edge and two vertices for the lower edge. The vertical texture coordinate v is assigned in the shader, with $v = 1$ for the upper vertices and $v = 0$ for the lower vertices. Together with the arc-length-based u -coordinate computed on the CPU, this allows radar images to be mapped continuously along the curtain, even if the individual polyline segments have different lengths.

The geometry shader emits one quad per polyline segment. Together, these quads form a continuous curtain along the entire path. As a result, the selected path directly defines the shape and the position of the curtain that reveals the terrain interior along the chosen trajectory.

Listing 5.5: Geometry-shader expansion of one line segment into a curtain quad

```

1 let curtainGeometry (line : Line<Vertex>) =
2     triangle {
3         let upVS : V3d = uniform?UpVS
4         let height : float = uniform?CurtainHeight
5         let offset = V4d(upVS.Normalized * height, 0.0)
6
7         let p0 = line.P0.vp
8         let p1 = line.P1.vp
9
10        let u0 = line.P0.tc.X
11        let u1 = line.P1.tc.X
12
13        yield { line.P0 with pos = uniform.ProjTrafo * p0
14                    tc = V2d(u0, 1.0) }
15        yield { line.P0 with pos = uniform.ProjTrafo * (p0 - offset)
16                    tc = V2d(u0, 0.0) }
17        yield { line.P1 with pos = uniform.ProjTrafo * p1
18                    tc = V2d(u1, 1.0) }
19        yield { line.P1 with pos = uniform.ProjTrafo * (p1 - offset)
20                    tc = V2d(u1, 0.0) }
21    }

```


Results and Discussion

This chapter evaluates the proposed cross-section pipeline with respect to the research questions formulated in Section 1.2. The evaluation considers the geometric consistency of the terrain classification, the construction and parameterization of the curtain geometry, and the suitability of the implementation for later integration into P_{Ro}3D. Since the integration into P_{Ro}3D itself is outside the scope of this thesis the third research question is addressed by evaluating the architectural compatibility and performance characteristics of the implemented prototype.

The results show that the final point-in-polygon approach provides a more stable and deterministic classification than the previously explored local clipping strategies. However, the quality and semantic clarity of the result still depend on the geometry of the input path. The generated curtain provides a continuous surface and a suitable texture parameterization for displaying a radargram, but small gaps and overlaps remain at the connection between the curtain and the clipped terrain. The implementation was built using the same central technologies and data structures as P_{Ro}3D, which provides a foundation for later integration.

6.1 RQ1: Classification of the Terrain

The first question asked:

How can the terrain region to be retained or removed be classified consistently for cross-section paths containing concave turns, loops, or self-intersections?

The explored approaches show that the difficulty of this classification problem depends strongly on the geometric form of the input polyline. Polyline whose consecutive segments are almost collinear and whose turns are relatively flat represent comparatively

simple inputs. For such paths, local methods such as the tiled clipping-plane approach can already produce plausible results. Since the directions of neighboring segments differ only slightly, their local slipping planes generally agree on which side of the path should remain visible.

The problem becomes considerably more difficult if the polyline contains sharp turns, concave regions, loops, or self-intersections. In these cases, a local classification relative to the nearest segment does not necessarily correspond to the intended global classification. A terrain point may be closest to a segment whose orientation implies a different retained side than another nearby segment. The main source of incorrect classifications encountered during the development were therefore sharp directional changes and self-intersections.

The nearest-point classification heuristic did not resolve this limitation. Although it avoided directly applying one clipping plane per segment, it still classified each point using local information associated with the nearest part of the path. Its result additionally depended on a frozen viewing direction. This produced incorrect classifications if the viewing direction did not correspond well to the orientation of the relevant path segment. Consequently, both explored local approaches worked for simple paths but failed to define a globally consistent retained region for sufficiently complex paths.

This is not only a theoretical edge case. Recorded rover traverses can contain sharp turns, repeated changes of direction, loops, and sections that cross or closely revisit earlier parts of the route. Figure 6.1 shows a section of the traverse displayed by NASA’s interactive Perseverance Rover Location Map at Sol 1883. The route contains several closely spaced and partly overlapping sections, abrupt changes in direction, and loop-like configurations. These geometric features demonstrate that the classification challenges considered in this work are relevant to real rover traverses and are not limited to artificially constructed test paths [NAS].

The final approach improves the consistency of the classification by reformulating the problem as a two-dimensional point-in-polygon test. The input polyline is combined with a reference point to form a closed polygon. All terrain vertices are projected into the same two-dimensional coordinate system and evaluated relative to this globally defined region. In contrast to the local approaches, the classification no longer depends on the nearest segment or the current camera orientation.

For an identical finalized path, reference point, terrain patch, and projection basis, the approach produces the same per-vertex classification in repeated executions. It can therefore be considered deterministic according to the definition in Section 1.5. The classification is also stable during subsequent camera movement because the polygon and the resulting vertex attributes do not change when the user navigates through the scene.

Figure 6.2 compares the classification results for paths with different geometric characteristics. For paths without self-intersections, the retained terrain region remains visually coherent and the cut follows the general shape of the path. This works for paths with flat turns as well as for paths with sharp turns (Figure 6.2a) (Figure 6.2b). For a self-intersecting path, the classification still produces a reproducible result, but



Figure 6.1: Geometrically complex section of the Perseverance rover traverse in the Jezero Crater. Screenshot from the NASA Perseverance Rover Location Map [NAS].

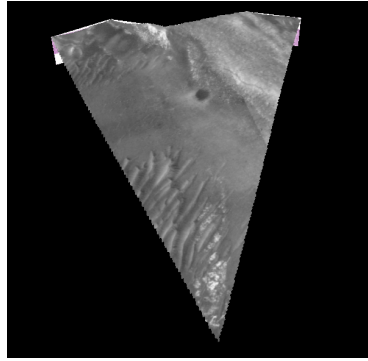
the retained region and the resulting cross-section are difficult to interpret semantically (Figure 6.2c).

These examples show that the point-in-polygon method provides one globally defined classification for all three inputs, but that the visual quality and semantic clarity of the result still depend on the geometry of the path. The images constitute a qualitative comparison of representative cases rather than a numerical accuracy evaluation.

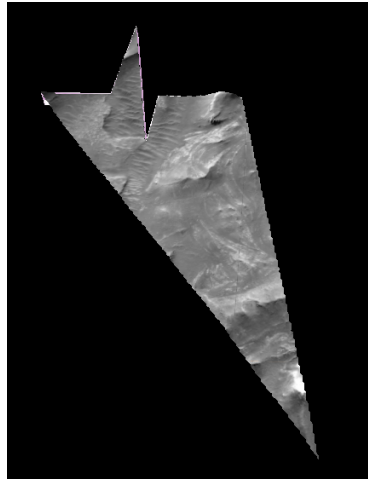
However, the final solution is not perfect. A deterministic result is not necessarily a semantically meaningful result for every possible input. As shown in Figure 6.2c, self-intersecting paths do not always define one intuitively clear inside and outside region. The implemented method may therefore remain computationally consistent while retaining a terrain region that does not correspond to the region a user would intuitively expect.

In contrast, the results for the simpler paths, without self-intersections, in Figure 6.2a Figure 6.2b demonstrate that the same method can produce a visually plausible and coherent cut if the path geometry is less complex.

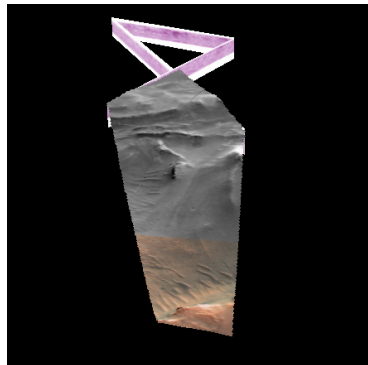
The semantic limitation of highly complex paths also has a perceptual interpretation. A path containing several loops or strongly differently oriented sections would expose cross-section surfaces facing several directions. Such a cross-section cannot necessarily be inspected clearly from one camera position, since it is not possible to view all sides of a complex three-dimensional object simultaneously. Even a geometrically correct classification could therefore lead to a visually confusing result.



(a) Path with relatively flat turns. The resulting retained terrain region is visually coherent.



(b) Path containing a sharp turn.



(c) Self-intersecting path.

Figure 6.2: Qualitative classification results for paths with increasing geometric complexity.

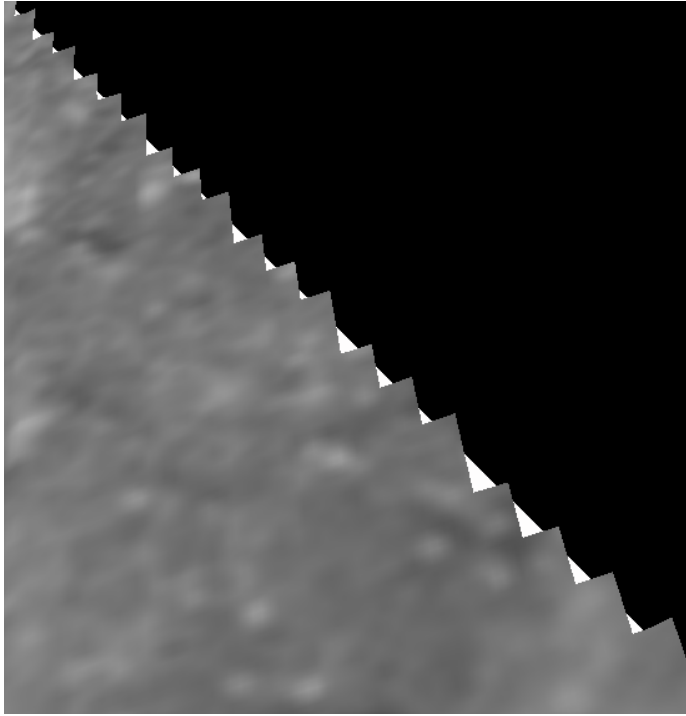


Figure 6.3: Close-up of the clipped terrain boundary. The repeated saw-tooth-like spikes result from applying the per-vertex classification through interpolation across the existing terrain triangles.

For this reason, highly complex rover traverses should not necessarily be represented as one connected cross-section. A more suitable visualization could divide such a path into a sequence of simpler cross-sections. Each sequence could then be inspected from an appropriate viewpoint and associated with the corresponding interval of the radargram. The complexity of the path is therefore not only an algorithmic problem, but also an indication that a different visual representation may be required.

A second limitation is visible at the cut border. As shown in Figure [6.3](#), the cut border can contain pronounced spikes. Since the classification is computed at the terrain vertices and interpolated across the triangles, the rendered boundary only approximates the polygon boundary. These spikes reduce the visual quality of the cross-section and complicate the connection between the terrain and the curtain.

The spikes do not substantially change the overall terrain region that is retained, but they reduce the visual quality of the cross-section and complicate the connection between the clipped terrain and the curtain.

RQ1 can consequently be answered as follows: consistent classification is achieved more reliably by replacing local segment-based decisions with a globally defined point-in-polygon classification. The approach is stable and deterministic for fixed inputs and

handles concave paths more consistently than the explored local methods. Nevertheless, sharp turns and self-intersections remain challenging. Self-intersections can produce semantically ambiguous regions, and the per-vertex representation of the classification produces spikes along the cut boundary. The method should therefore be interpreted as a practical solution for paths of moderate geometric complexity rather than a perfect solution for every arbitrary path.

Figure 6.4 shows the application of the final classification method to the imported traverse. Before clipping, the complete terrain remains visible and the imported path is drawn on top of it (Figure 6.4a). After the path has been finalized, the point-in-polygon classification retains the terrain region associated with the resulting closed polygon (Figure 6.4b). The top view makes the global shape of the retained region visible and illustrates that the classification is applied to the complete path rather than independently to its individual segments. Finally, the retained terrain serves as the basis for generating the Cross-Section curtain onto which the radargram is projected, demonstrating that the classification directly enables the final visualization (Figure 6.4c).

6.2 RQ2: Curtain Construction and Radargram Mapping

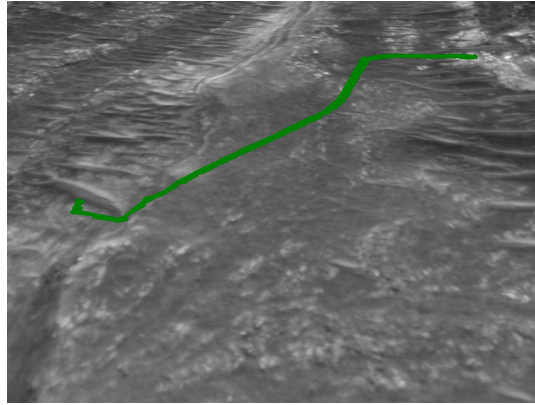
The second research question asks:

How can a curtain geometry be constructed and parameterized along the cross-section path so that RIMFAX radargrams can be displayed continuously in their corresponding 3D terrain context?

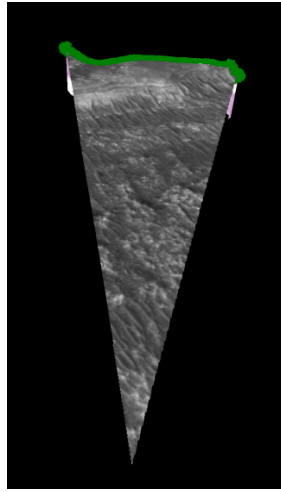
The implemented curtain construction successfully generates a connected geometric surface along the complete cross-section path. The finalized polyline is represented as a sequence of line segments, and the geometry shader expands every segment into a quad. Neighboring quads share the positions associated with their common polyline vertex. The resulting curtain is therefore continuous along the path and covers the complete horizontal extent of the selected cross-section.

The arc-length-based texture parameterization provides a continuous horizontal coordinate along the curtain. The horizontal texture coordinate is calculated from the accumulated physical length of the path. This prevents the texture from being stretched differently on each segment and establishes a direct mapping between the horizontal radargram axis and the distance traveled along the cross-section path.

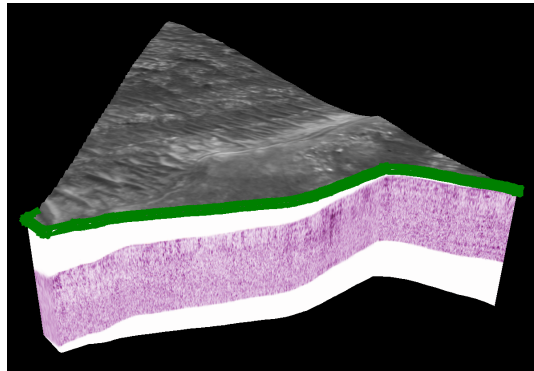
The vertical texture coordinate is assigned during the geometry-shader expansion. The upper vertices receive the upper texture coordinate, and the vertically offset vertices receive the lower texture coordinate. Together, the horizontal arc-length coordinate and the vertical coordinate define a complete parameterization of the curtain. The implementation therefore provides the geometric and texturing foundations required to place a two-dimensional radargram into the three-dimensional terrain scene.



(a) Imported path on the uncut terrain



(b) Retained terrain after classification



(c) Radargram mapped onto the Cross-Section curtain

Figure 6.4: Application of the point-in-polygon classification to the imported traverse. The green line denotes the imported path.

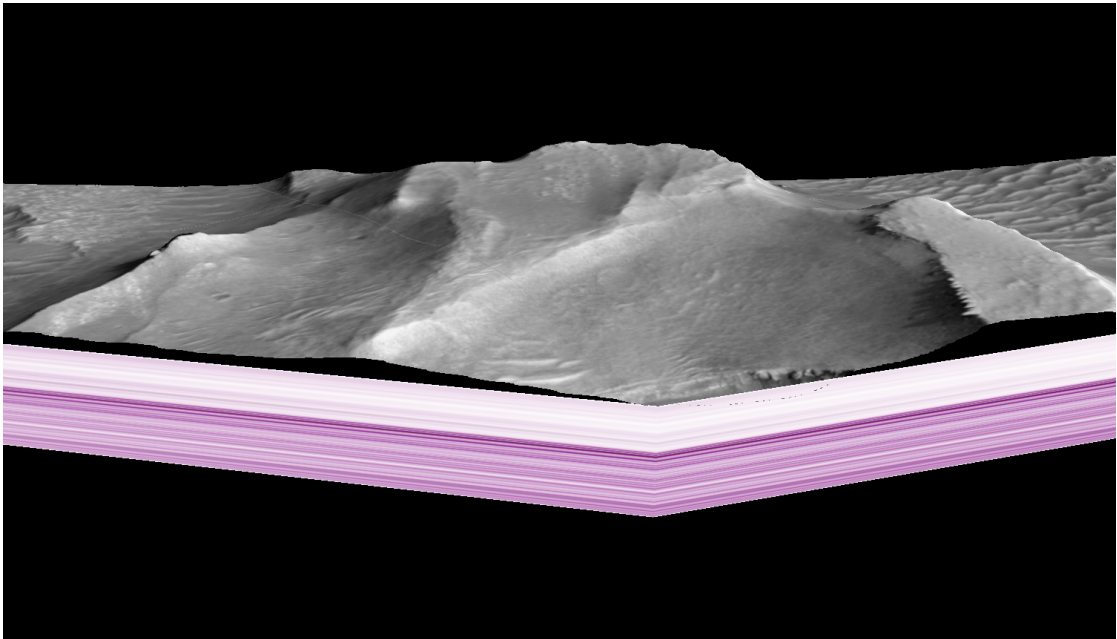


Figure 6.5: Visible gap between the clipped terrain and the generated curtain. The black region between the terrain boundary and the upper curtain edge shows where the two independently generated boundaries do not coincide.

However, the curtain does not reconstruct the exact boundary produced by clipping the terrain. It is generated by extruding the input polyline by a fixed distance along the approximate planetary up direction. The upper curtain border follows the input polyline, while the remaining geometry is produced through a straight offset in one predefined direction. It therefore does not adapt to every local height variation and irregularity of the clipped terrain boundary.

This difference can produce small overlaps or gaps between the upper curtain border and the cut terrain border. Figure 6.5 shows a gap through which the background remains visible between the terrain and the curtain. In Figure 6.5 the changing elevation of the terrain even increases the size of the gap. The problem is also amplified by the spikes created during the terrain clipping. Even when the curtain follows the original polyline correctly, it cannot connect without gaps or overlaps to an irregular border.

The remaining gaps and overlaps are primarily visual artifacts. They do not prevent the radargram from being displayed on the curtain, but they weaken the impression that the curtain is directly and continuously attached to the opened terrain. Improving this connection would require extracting the actual cut contour from the terrain triangles or generating additional transition geometry between the clipped terrain boundary and the curtain.

A further limitation concerns the radargram image itself. The tested radargram contains

white borders that are part of the input image rather than the measured subsurface signal. If the complete image is mapped directly onto the curtain, these borders remain visible in the three-dimensional scene. They should be removed in a preprocessing step, for example by cropping the image to its relevant data region or by storing the valid radargram range as metadata. This preprocessing is separate from the geometric curtain construction but is necessary for a visually clean final presentation.

RQ2 can therefore be answered as follows: a continuous curtain can be constructed efficiently by expanding the path segments into connected quads and parameterizing the horizontal texture coordinate using normalized arc length. This produces a continuous geometric carrier for the complete radargram and preserves the relative spatial extent of its measurements along the path. The curtain does not, however, conform exactly to the clipped terrain boundary. The spikes along the terrain border and the uniform extrusion direction cause small gaps and overlaps, while borders contained in the radargram image require additional preprocessing.

6.3 RQ3: Preparation for Integration into PPro3D

The third research question asks:

How should the classification, clipping, and curtain-generation method be designed to support integration into PPro3D while maintaining interactive performance on large terrain datasets?

The prototype itself was integrated into the complete PPro3D application only after this thesis had been completed. This integration was carried out independently by other developers and is therefore outside the scope of this work. Consequently, the integration process itself is not evaluated as a contribution of this thesis. However, the successful incorporation of the method into PPro3D provides practical evidence that the chosen technologies, data structures, and rendering abstractions are compatible with the target application.

Figure [6.6](#) shows a screenshot of the cross-section visualization being used inside PPro3D after the separate integration step. The figure is included to illustrate the intended application context of the method, not as an evaluation of the integration process itself. The evaluation in this thesis remains limited to the prototype implementation, its geometric behavior, and its measured performance in the standalone test environment.

The direct integration of the prototype into the complete PPro3D application was carried out separately and is not part of this thesis. Consequently, RQ3 is evaluated by considering whether the implementation uses compatible technologies, data structures, and processing stages, and whether its measured performance is suitable for an interactive visualization workflow. The later integration into PPro3D is used only as contextual evidence that the design is compatible with the target application.

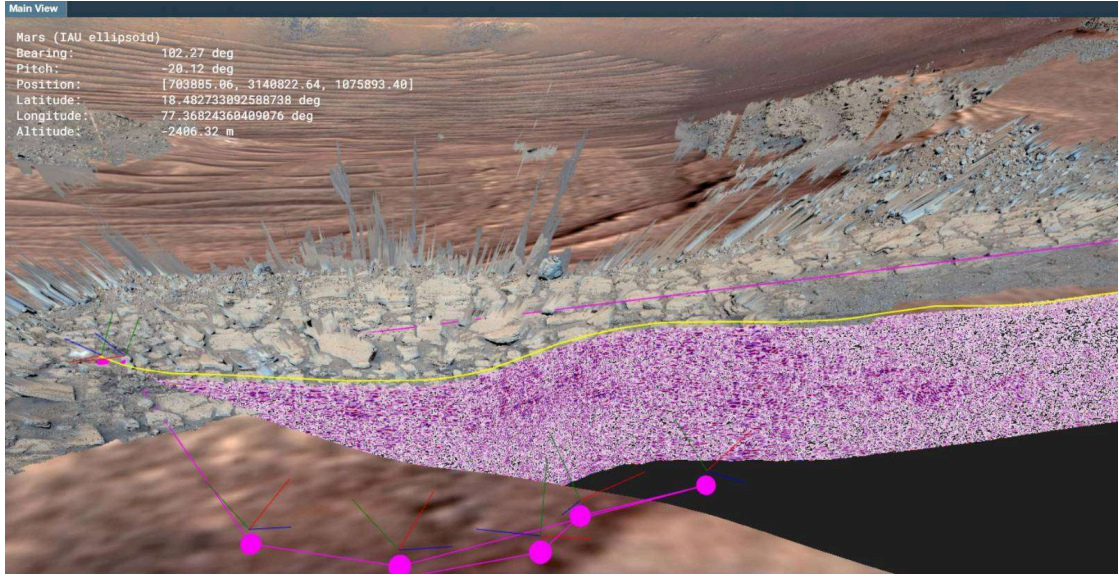


Figure 6.6: Screenshot from the cross-section visualization used inside PRo3D after separate integration into the application.

Several design decisions were made specifically to support later integration. The implementation is written in F# and uses the Aardvark Platform, which is also used by PRo3D. The implementation can therefore reuse the same mathematical types, scene-graph abstractions, rendering mechanisms, shader system, and GPU-resource management. This avoids the need to transfer the method from a separate graphics framework or programming language before it can be incorporated into PRo3D.

The terrain is loaded from an OPC dataset and processed through its patch hierarchy. Rather than requiring the complete terrain mesh to be available in memory, the method operates on the individual patches selected by the existing out-of-core rendering system. The scene graph dynamically selects appropriate levels of detail according to the current camera view. This is important for planetary terrain datasets, whose complete high-resolution geometry is too large to process or render as one mesh.

Aardvark is designed to support interactive visualization of every large datasets, including out-of-core data structures, patch hierarchies, and dynamic level-of-detail selection. Building the prototype on these mechanisms does not guarantee successful integration by itself, but it removes several major architectural incompatibilities.

6.3.1 Evaluation Setup

The performance evaluation was conducted on a system equipped with an 11th Gen Intel Core i7-1165G7 processor, Intel Iris Xe Graphics, and 16 GB of RAM.

The evaluation used the Jezero_04_05 OPC terrain dataset. The dataset consists of 1,507 files with a total on-disk size of 1,375,748,054 bytes, corresponding to approximately

1.28 GiB or 1.38 GB. The imported traverse contained 150 path points. Together with the additional reference point, the resulting classification polygon contained 151 points. The camera was kept stationary, and we waited for the terrain to be loaded completely before the measurements were started. The evaluated view required the classification buffers of 34 terrain patches containing a total of 1,911,628 vertices. The application was executed as a Release build without an attached debugger. It targeted .NET 9.0 and used the OpenGL rendering backend.

Frame measurements were performed at a framebuffer resolution of 1536×1152 pixels. The complete input data, evaluated workload, and execution configuration are summarized in Table 6.1

Measurement	Value
Processor	11th Gen Intel Core i7-1165G7
Graphics	Intel Iris Xe Graphics
Memory	16 GB RAM
Terrain dataset	Jezero_04_05
Dataset size	1,507 files; 1.28 GiB / 1.38 GB
Imported path points	150
Classification polygon points	151
Evaluated patches	34
Processed terrain vertices	1,911,628
Build configuration	Release
Debugger attached	No
Target framework	.NET 9.0
Rendering backend	OpenGL
Framebuffer resolution	1536×1152
Rendering mode	Event-driven rendering; RenderAsFastAsPossible not forced
Frame measurement scenario	Imported traverse after clipping; continuous camera movement
Vertical synchronization	Enabled

Table 6.1: Input data, workload, and execution configuration used for the performance evaluation.

6.3.2 Classification Performance

The classification was measured in ten repeated executions. Every execution processed the same 34 terrain patches and 1,911,628 terrain vertices. For each execution, the wall-clock completion time measured the interval from invalidating the classification buffers until the classifications of the evaluated patches had completed. The measurements varied between the individual executions. This variation concerned execution time only; the number of evaluated patches, processed vertices, path points, and classification polygon points remained unchanged. The runtime measurements therefore characterize the processing delay for a fixed classification workload.

6.3.3 Rendering Performance

Rendering performance was evaluated after the imported traverse had been clipped and the classification had completed. Each run discarded the first 120 frames as warm-up and measured the following 600 frames. Since the application was executed in its normal event-driven rendering mode, frames were only produced while the scene changed. The measurements were therefore collected during continuous camera movement. Terrain clipping and curtain rendering were active during all measurements.

Ten repeated measurements were performed. The mean frame times ranged from 16.361 ms to 21.771 ms. The corresponding frame rates ranged from 45.93 fps to 61.12 fps. Across all runs, the mean of the reported frame rates was 54.60 fps. If the accumulated frame times of all measured frames are considered together, the resulting overall frame rate is 54.12 fps. The average 95th-percentile frame time over the ten runs was 26.62 ms.

Ten runs of 600 frames each were performed, resulting in 6,000 measured frames. The run-level mean frame times ranged from 16.36 ms to 21.77 ms, corresponding to frame rates between 45.93 fps and 61.12 fps. The arithmetic mean of the ten run-level frame rates was 54.60 fps. Based on the accumulated frame times of all measured frames, the aggregated mean frame time was 18.48 ms. The mean of the ten run-level 95th-percentile frame times was 26.62 ms.

These values were measured during continuous camera movement, because the application uses event-driven rendering and does not continuously redraw unchanged frames when `RenderAsFastAsPossible` is not forced. The complete rendering performance evaluation is summarized in Table 6.2

Run	Mean frame time [ms]	Median [ms]	95th percentile [ms]	Frame rate [fps]
1	21.77	19.97	32.12	45.93
2	17.15	14.09	30.59	58.31
3	16.36	14.07	23.77	61.12
4	18.00	15.24	24.68	55.57
5	16.43	14.19	18.51	60.86
6	20.43	17.80	27.25	48.94
7	19.71	17.40	38.00	50.73
8	19.72	15.93	30.91	50.71
9	16.81	14.73	17.90	59.50
10	18.40	14.08	22.45	54.35
Mean	18.48	15.75	26.62	54.60

Table 6.2: Rendering performance after clipping the imported traverse. Each run measured 600 frames during continuous camera movement.

These results indicate that navigation remained interactive while clipping and curtain rendering were enabled.

RQ3 can consequently be answered as follows: the pipeline should be implemented using

the same central technologies and data structures as PPro3D, operate directly on the OPC patch hierarchy, and separate the more expensive CPU-side classification from the comparatively lightweight GPU-side clipping and curtain rendering. The use of F#, Aardvark scene graphs, FShade shaders, OPC terrain patches, and dynamic level-of-detail selection provides an architecturally compatible basis for later integration. For the evaluated workload, the classification required approximately 1.21 s on average, while subsequent navigation with clipping and curtain rendering enabled remained interactive at approximately 54 fps during continuous camera movement.

These results demonstrate the suitability of the design for the evaluated prototype and terrain view, but they do not establish performance for every possible dataset, camera configuration, or path complexity. Although the method has since been integrated into PPro3D by other developers, this thesis does not evaluate the integration process itself or its behavior under complete production workloads.

6.4 Answer to the Main Research Question

The main research question asks:

How can RIMFAX subsurface radar data be visualized in a 3D terrain context using stable, deterministic, and interactive cross-sections along geometrically complex paths, while preparing the visualization pipeline for later integration into PPro3D?

The developed pipeline addresses this problem through three connected stages. First, the input path and an additional reference point are projected into a two-dimensional coordinate system and used to define a closed classification polygon. Terrain vertices are classified relative to this polygon, and the resulting values are passed to the GPU to remove the selected terrain region. Second, a continuous curtain geometry is generated along the path and parameterized using normalized arc length. This curtain provides the geometric carrier on which a RIMFAX radargram can be displayed. Third, the implementation uses F#, Aardvark, OPC terrain patches, and dynamic level-of-detail selection, which provides an architectural basis for later integration into PPro3D.

For fixed input data, the classification does not depend on subsequent camera movement, because the classification polygon and the resulting per-vertex attributes remain unchanged after the path has been finalized. The method is also deterministic by design: identical path points, reference point, terrain geometry, and projection parameters are processed by the same point-in-polygon classification. Compared with the explored local approaches, the global point-in-polygon formulation handles concave paths and strong directional changes more consistently. The generated curtain covers the complete cross-section path and provides continuous texture coordinates for mapping the radargram.

The performance evaluation shows that the CPU-side classification introduces a noticeable one-time delay. For the evaluated view, which contained 34 terrain patches with 1,911,628 vertices, the classification required approximately 1.21 s on average over ten executions. After this processing step had completed, navigation remained interactive with clipping and curtain rendering enabled. Across ten measurements of 600 frames each, the mean frame time was 18.48 ms. The mean of the reported frame rates was 54.60 fps, while the accumulated frame times over all measured frames correspond to an overall frame rate of 54.12 fps. These values were measured during continuous camera movement in the normal event-driven rendering mode.

At the same time, the results identify clear limitations. Self-intersecting paths can remain semantically ambiguous even if their classification is computationally consistent. The cut terrain has spikes along the border. These spikes, together with the uniform curtain extrusion, result in gaps or overlaps between the terrain and the curtain. Furthermore, self-intersecting paths produce cross-sections that are not semantically clear and complex paths whose sections face several directions may not form a perceptually clear cross-section that can be inspected from one camera position.

The main research question can therefore be answered positively with qualifications. The implemented method provides the central components required to embed a RIMFAX radargram into a three-dimensional terrain context through a stable and reproducible cross-section. In the evaluated view, the one-time classification delay remained on the order of one second, while subsequent rendering remained interactive. The implementation also provides a practical architectural basis for later integration into PRo3D. However, the solution has limitations. Further work is required to improve the terrain–curtain connection, preprocess the radargram imagery, evaluate the method inside the complete PRo3D application, and represent highly complex rover paths in a perceptually clearer form.

6.5 Limitations and Future Work

The evaluation identified limitations related to each of the three research questions.

Regarding RQ1, the main limitations are the visible spikes along the clipped terrain boundary and the semantic ambiguity of self-intersecting paths. The classification is computed at the terrain vertices and represented between them through interpolation across the existing triangles. Consequently, the rendered cut only approximates the polygon boundary and can contain pronounced saw-tooth-like spikes. These spikes reduce the visual quality of the cross-section and make it difficult to connect the terrain border cleanly to the curtain.

Future work could replace the interpolated per-vertex boundary with an explicitly reconstructed cut contour. Terrain triangles intersected by the classification boundary could be subdivided, and new vertices could be inserted at the estimated intersection positions. This would produce a more accurate geometric border and could reduce the

spikes visible in the current implementation. A less expensive alternative would be to smooth or simplify the reconstructed boundary before generating the final cross-section geometry.

Self-intersecting paths present a different limitation. Although the classification remains deterministic for fixed inputs, the resulting retained region does not necessarily correspond to the region a user would intuitively expect. A path containing loops or intersections can define several overlapping or nested regions, while the resulting curtain may contain sections facing substantially different directions. The cross-section can therefore be computationally consistent but semantically and perceptually unclear.

Future work should analyze the input path before constructing the classification polygon. Sharp turns, self-intersections, closely repeated segments, and major changes in orientation could be detected automatically. Instead of always generating one connected cross-section, a sufficiently complex rover traverse could be divided into a sequence of simpler paths. Each path interval could then be visualized as a separate cross-section and inspected from a suitable viewpoint.

Regarding RQ2, the main limitation is that the generated curtain does not follow the exact clipped terrain boundary. The curtain is constructed directly from the input polyline, whereas the visible terrain boundary results from the interpolated per-vertex classification. Because these boundaries are generated independently, small gaps or overlaps can occur between the upper curtain edge and the terrain. The spikes along the terrain boundary further amplify this problem.

An improved approach could construct the curtain from the extracted terrain intersection contour rather than directly from the original path. Alternatively, transition geometry could be generated between the curtain and the clipped terrain border. Such geometry could preserve the current path-based curtain parameterization while visually closing remaining gaps.

The radargram input also requires additional preparation. The evaluated image contains borders and regions that are not part of the measured subsurface signal. When the complete image is mapped to the curtain, these regions remain visible in the three-dimensional scene. Future preprocessing should crop the image to the valid measurement region or store its valid extent as metadata. A more precise mapping could additionally associate individual soundings or sounding intervals with their corresponding positions along the traverse instead of mapping the complete image only according to normalized path length.

Although the method has since been integrated into PPro3D by other developers, this thesis does not evaluate the integration process itself or its behavior under complete production workloads. Future work should therefore examine the interaction with other PPro3D rendering components, application state, data-loading mechanisms, and user-interface workflows in detail.

The performance measurements also cover only one terrain dataset, one fixed camera

view, and one imported path. Future evaluation should include additional OPC datasets, complete rover traverses, different camera configurations, different levels of detail, and paths with varying numbers of vertices and geometric complexities. It should measure classification time, rendering time, memory consumption, and scaling behavior as both terrain and path complexity increase. The effects of dynamically loading and unloading OPC patches should also be examined.

Finally, a user study with planetary scientists could evaluate whether the generated cross-sections improve the interpretation of RIMFAX data. Such a study could investigate whether scientists prefer one connected curtain or a sequence of simpler cross-sections for complex rover traverses, whether the terrain clipping communicates the spatial relationship clearly, and which interaction mechanisms are required for practical use inside PPro3D.

Overview of Generative AI Tools Used

Generative AI tools were employed in a supporting role during the preparation of this thesis. ChatGPT (OpenAI, GPT-5.4 Thinking) was used primarily for language refinement, restructuring of text passages, and checking for consistency. Furthermore, the Clause agent integrated into JetBrains Rider was used to support software development tasks, including code comprehension, debugging, and minor revisions within the development workflow. All AI-generated output was carefully reviewed, verified, and selectively incorporated by the author. Responsibility for the correctness, originality, and final form of the thesis remains solely with the author.

List of Figures

1.1	Visually confusing results caused by improper depth relationships.	3
1.2	Incorrect surface classification resulting from local clipping strategy. . . .	4
2.1	Perseverance rover traverse in the Jezero Crater from Sol 0 to Sol 204. Reproduced from [Ham22].	8
2.2	RIMFAX radargram and interpretive cross-section along the Brac-Issole traverse. From Hamran et al. (2022), Fig. 3, licensed under CC BY-NC 4.0.	9
4.1	Stepwise visualization of the tiled clipping planes workflow. (a) Polyline segments as planes along a polyline on the terrain. (b) The terrain mesh is cut along the polyline, keeping only one side of the mesh. (c) A cross-section geometry is built along the mesh's cut.	26
4.2	Result of the tiled clipping-plane approach for a concave polyline.	26
4.3	Failure case of the view-direction-based classification.	29
4.4	Point-in-polygon-based formulation of the clipping problem. Here, the reference point and v_0 share one vertex.	32
5.1	Implementation pipeline of the final point-in-polygon-based clipping approach.	37
6.1	Geometrically complex section of the Perseverance rover traverse in the Jezero Crater. Screenshot from the NASA Perseverance Rover Location Map [NAS].	47
6.2	Qualitative classification results for paths with increasing geometric complexity.	48
6.3	Close-up of the clipped terrain boundary. The repeated saw-tooth-like spikes result from applying the per-vertex classification through interpolation across the existing terrain triangles.	49
6.4	Application of the point-in-polygon classification to the imported traverse. The green line denotes the imported path.	51
6.5	Visible gap between the clipped terrain and the generated curtain. The black region between the terrain boundary and the upper curtain edge shows where the two independently generated boundaries do not coincide.	52
6.6	Screenshot from the cross-section visualization used inside PPro3D after separate integration into the application.	54

List of Tables

3.1	Comparison of the surveyed systems with respect to the requirements of the proposed cross-section pipeline.	17
6.1	Input data, workload, and execution configuration used for the performance evaluation.	55
6.2	Rendering performance after clipping the imported traverse. Each run measured 600 frames during continuous camera movement.	56

Bibliography

- [Aar24] Aardvark Platform Contributors. Aardvark platform. <https://aardvarkians.com/>, 2024. Accessed: 2026-03-04.
- [ABSW18] Charles H. Acton, Nathaniel Bachman, Boris Semenov, and Edward Wright. A look towards the future in the handling of space science mission geometry. *Planetary and Space Science*, 150:9–12, 2018.
- [Bri] British Geological Survey. Bgs groundhog documentation. <https://britishgeologicalsurvey.github.io/Groundhog/>. Accessed: 2026-02-24.
- [Bri25] British Geological Survey. Bgs groundhog v2.8.9. <https://www.bgs.ac.uk/technologies/software/groundhog/>, 2025. Accessed: 2026-02-24.
- [CGA26a] CGAL Editorial Board. Cgal 5.x – aabb tree package. https://doc.cgal.org/latest/AABB_tree/index.html, 2026. Accessed: 2026-02-24.
- [CGA26b] CGAL Editorial Board. Cgal::aabb_halfedge_graph_segment_primitive class reference. https://doc.cgal.org/latest/AABB_tree/classCGAL_1_1AABB_halfedge_graph_segment_primitive.html, 2026. Accessed: 2026-02-24.
- [CGA26c] CGAL Editorial Board. Cgal::polygon_mesh_slicer class reference. https://doc.cgal.org/latest/Polygon_mesh_processing/classCGAL_1_1Polygon_mesh_slicer.html, 2026. Accessed: 2026-02-24.
- [dlVSW19] Miguel de la Varga, Alexander Schaaf, and Florian Wellmann. Gempy 1.0: Open-source stochastic geological modeling and inversion. *Geoscientific Model Development*, 12(1):1–32, 2019.
- [Eri05] Christer Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, San Francisco, 2005.
- [FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. The Systems Programming Series. Addison-Wesley, Reading, MA, 2 edition, 1990.

- [Gem25a] GemPy Project. Gempy documentation: Plotting 2d sections. <https://docs.gempy.org/en/latest/plotting/index.html>, 2025. Accessed: 2026-03-10.
- [Gem25b] GemPy Project. Gempy: Open-source 3d geological modeling in python. <https://www.gempy.org>, 2025. Accessed: 2026-03-10.
- [Gem25c] GemPy Project. Gempy tutorial: Cross sections and section grids. https://docs.gempy.org/en/latest/tutorials/ch1_fundamentals/3_sections.html, 2025. Accessed: 2026-03-10.
- [GG17] Michael Galetzka and Patrick Glauner. A simple and correct even-odd algorithm for the point-in-polygon problem for complex polygons. In *Proceedings of the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2017) - Volume 1: GRAPP*, pages 175–178. SCITEPRESS, 2017.
- [HA01] Kai Hormann and Alexander Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131–144, 2001.
- [Ham20] Hamran, Svein-Erik and Paige, David A. and Amundsen, Hans E. F. and Berger, Tor and Brovoll, Sverre and Carter, Lynn and Damsgård, Leif and Dypvik, Henning and Eide, Sigurd and Ghent, Rebecca and Kohler, Jack and Mellon, Michael T. and Nunes, Daniel C. and Plettemeier, Dirk and Russell, Patrick and Øyan, Mats Jørgen. Radar Imager for Mars’ Subsurface Experiment—RIMFAX. *Space Science Reviews*, 216(128), 2020.
- [Ham22] Hamran, Svein-Erik and Paige, David A. and Allwood, Abigail and Amundsen, Hans E. F. and Berger, Tor and Brovoll, Sverre and Carter, Lynn and Casademont, Titus M. and Damsgård, Leif and Dypvik, Henning and Eide, Sigurd and Fairén, Alberto G. and Ghent, Rebecca and Kohler, Jack and Mellon, Michael T. and Nunes, Daniel C. and Plettemeier, Dirk and Russell, Patrick and Øyan, Mats Jørgen. Ground penetrating radar observations of subsurface structures in the floor of Jezero crater, Mars. *Science Advances*, 8(34):eabp8564, 2022.
- [HP21] Svein-Erik Hamran and David A. Paige. Mars 2020 Perseverance Rover RIMFAX Raw and Calibrated Data Products. NASA Planetary Data System (PDS) Geosciences Node https://pds.nasa.gov/ds-view/pds/viewBundle.jsp?identifier=urn%3Anasa%3Apds%3Amars2020_rimfax&version=1.0, 2021.
- [HT73] John Hopcroft and Robert Tarjan. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- [Kit24] Kitware, Inc. vtkcutter class reference. <https://vtk.org/doc/nightly/html/classvtkCutter.html>, 2024. Accessed: 2026-03-10.

- [Kit26a] Kitware Inc. Paraview user’s guide – displaying data. <https://docs.paraview.org/en/latest/UsersGuide/displayingData.html>, 2026. Accessed: 2026-02-24.
- [Kit26b] Kitware Inc. Paraview user’s guide – introduction. <https://docs.paraview.org/en/latest/UsersGuide/introduction.html>, 2026. Accessed: 2026-02-24.
- [Kit26c] Kitware Inc. paraview.simple.SliceAlongPolyLine. <https://www.paraview.org/paraview-docs/v5.13.2/python/paraview.simple.SliceAlongPolyLine.html>, 2026. Accessed: 2026-02-24.
- [Kit26d] Kitware Inc. vtkCutter Class Reference. <https://vtk.org/doc/nightly/html/classvtkCutter.html>, 2026. Accessed: 2026-02-24.
- [Kne17] Ronald T. Kneusel. *Numbers and Computers*. Springer International Publishing, 2 edition, 2017.
- [Law08] Lawrence Berkeley National Laboratory. Paraview documentation – linear extrusion filter. https://dav.lbl.gov/archive/NERSC/Software/paraview/docs/HTML/filters_linear_extrusion.htm, 2008. Accessed: 2026-02-24.
- [lib26a] libigl contributors. Issue #1537: Discussion on mesh slicing. <https://github.com/libigl/libigl/issues/1537>, 2026. Accessed: 2026-02-24.
- [lib26b] libigl contributors. Issue #851: Mesh cutting / slicing discussion. <https://github.com/libigl/libigl/issues/851>, 2026. Accessed: 2026-02-24.
- [lib26c] libigl contributors. libigl – a simple c++ geometry processing library. <https://libigl.github.io/>, 2026. Accessed: 2026-02-24.
- [lib26d] libigl contributors. libigl github repository. <https://github.com/libigl/libigl>, 2026. Accessed: 2026-02-24.
- [MVS⁺17] Rodrigo Minetto, Neri Volpato, Jorge Stolfi, Rodrigo M. M. H. Gregori, and Murilo V. G. da Silva. An optimal algorithm for 3d triangle mesh slicing. *Computer-Aided Design*, 92:1–10, 2017.
- [NAS] NASA. Where is perseverance? <https://science.nasa.gov/mission/mars-2020-perseverance/location-map/>. Accessed 14 June 2026.
- [Shi62] Shimrat. Algorithm 112: Position of point relative to polygon. *Communications of the ACM*, 5(8):434, 1962.

- [SIM16] John C. Steuben, Andreas Iliopoulos, and John G. Michopoulos. Implicit slicing for additive manufacturing. *Computer-Aided Design*, 77:1–11, 2016.
- [SML06] William J. Schroeder, Ken Martin, and Bill Lorensen. The visualization toolkit: An object-oriented approach to 3d graphics. <https://gitlab.kitware.com/vtk/textbook/-/raw/master/VTKBook/VTKTextBook.pdf>, 2006. Accessed 14 June 2026.
- [Uni24] University of Freiburg. vtkcutter – vtk manual page. <https://www.geologie.uni-freiburg.de/root/manuals/vtkman/vtkCutter.html>, 2024. Accessed: 2026-03-10.
- [VRV23] VRVis Research Center. Aardvark: From a library to an important tool in visual computing. <https://www.vrvis.at/en/news-events/news/aardvark-from-a-library-to-an-important-tool-in-visual-computing>, 2023. Accessed: 2026-03-04.
- [VRV24] VRVis Research Center. Aardvark – visual computing framework. <https://www.vrvis.at/en/products-solutions/products-licenses/aardvark>, 2024. Accessed: 2026-03-04.
- [VRV25] VRVis Research Center. Pro3d – planetary robotics 3d viewer. <https://pro3d.space/>, 2025. Accessed: 2026-03-01.
- [WA20] Shawn Waldon and Utkarsh Ayachit. Slice along a polyline. <https://www.kitware.com/slice-along-a-polyline/>, 2020. Accessed: 2026-02-24.
- [WRRH15] Ben Wood, Tanya Richmond, Jenny Richardson, and John Howcroft. BGS Groundhog[®] Desktop Geoscientific Information System v1.8.0: External User Manual. Technical Report OR/15/046, British Geological Survey, 2015.