INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

An End-to-End Compute Shader Pipeline for Real-Time Cartographic Projection and Rendering of Vector Geometry

Thore Schillmann

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

An End-to-End Compute Shader Pipeline for Real-Time Cartographic Projection and Rendering of Vector Geometry

Thore Schillmann

Supervisor:	Prof. Dr. Dieter Kranzlmüller
Advisor:	Dr. Rubén Jesús García-Hernández
Submission Date:	25.03.2026

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln verfasst habe. Alle Passagen, die ich aus einer Literatur oder anderen Quellen übernommen habe, sind eindeutig als Zitate mit Quellenangabe gekennzeichnet. Diese gedruckte Fassung ist ein Ausdruck der eingereichten elektronischen Fassung.

I hereby declare that I have written this work independently and only with the aids specified. All passages that I have taken from literature or other sources have been clearly marked as quotations with reference to the source. This printed copy is a printout of the submitted electronic copy.

München, den 25. März 2026 / Munich, March 25, 2026

. . . . . .

(Unterschrift / Signature)

Abstract

All major web mapping services default to the Web Mercator projection. This choice is rooted in both technical advantages and historical precedents. Unique in preserving angles and compass bearings, it is a natural fit for its intended use in nautical navigation. Yet, its significant drawbacks, notably extreme areal distortion at higher latitudes and an inability to represent polar areas, foster widespread misconceptions about the relative sizes of countries, continents, and oceans.

While recent research proposes to leverage the interactivity of digital maps to adapt projections by scale and location, most efforts focus heavily on geographic theory. Substantial computer-graphics challenges persist:

- Computational Cost: Continuous re-projection at interactive frame rates often exceeds the capabilities of current hardware.
- Vector Tile Complexity: Modern web mapping systems have moved towards vector-geometry, for which the necessary processing steps are more complex and resource-intensive.

This research capitalizes on recent advancements in graphics APIs—specifically the support for compute shaders in web browsers through WebGPU—to develop an end-to-end, GPU-accelerated pipeline for the projection of spherical vector geometry. The pipeline features two novel algorithms: one for clipping geometry by a spherical circle or the antimeridian and another for adaptively sampling lines based on projection-induced curvature to enhance visual fidelity.

Contents

1. Introduction

Because it is impossible to perfectly represent the spherical Earth on a flat, 2D surface, every map projection introduces some form of distortion. Therefore, any commonly used projection either excels at preserving specific properties—like area or shape—or strikes a compromise between different types of inaccuracies.

One of the most famous examples of preserving shape at the expense of area is the Mercator projection. Originally designed by Gerardus Mercator in 1569 for nautical navigation, its ability to preserve angles and constant compass bearings meant a straight line on the map corresponded to a constant heading. Google Maps and other vendors adopted a modern variant of this, known as Web Mercator (EPSG:3857) [OGP08]. Modern web maps use Web Mercator for a related reason: its conformality preserves the local shapes of features, ensuring that intersecting streets meet at correct angles and North remains consistently oriented upwards at any zoom level [BFUY14]. For these reasons, Web Mercator has become ubiquitous in digital mapping, despite the severe areal distortion it introduces the further one moves from the equator.

1.1. Motivation

While the fundamental restrictions of map projections cannot be avoided by static maps, interactive digital maps have the potential to bypass some of their shortcomings. Ideally, we would want to be able to continuously reproject or interpolate between quasi-optimal projections based on the current map scale and central latitude of the viewport. Although some projections can be continuously transformed into other projections, this is the exception not the rule [Jen12, JŠ18]. Even when such a transformation exists, intermediate deformation stages are often not guaranteed to exhibit the desired properties.

Nonetheless, there have been recent advances in the field of cartography—specifically the mathematics of equal-area projections [Str18]—that warrant a closer look into the work that needs to be done on the computer-science and -graphics side to make such a vector-based map rendering system a reality. This work aims to explore if a GPU-accelerated projection pipeline can be implemented that is fast and robust enough to be used in real-time interactive web applications, and if so, give an outlook on further work that would be required to make it production-ready.

1.2. Contribution and Scope

This thesis focuses on real-time GPU-accelerated projection of geometric features as defined by the Simple Features standard (point, line, polygon, multipoint, multi-line, etc.) [Ope11]. Instead of interpreting the coordinates as planar x, y values, they are treated as spherical longitude and latitude.

Geographic Information Systems (GIS) typically support Coordinate Reference Systems (CRS) which may use an ellipsoidal model of the Earth. The corresponding math is significantly more complex than that of the spherical model and—more importantly—computationally expensive. For this reason, I restricted myself to a spherical model. While sufficient for general purpose mapping, this precludes usage of the current implementation in more specialized software.

To this purpose, I made the following contributions: First, I implemented a mapping library that renders the GeoJSON [BDD+16] format using the compute-centric vector graphics renderer Vello[1], which utilizes the WebGPU[2] API for GPU acceleration.

Processing GeoJSON data on the GPU through a custom intermediate representation would have meant one of two things: Either a costly data transfer from the GPU to the CPU for every frame, followed by creating and encoding a Vello scene that would then have to be sent back to the GPU. Or, a custom GPU-based scene builder that would need to be able to translate the intermediate representation into Vello's internal format. The first option would have been prohibitively expensive, while the second is a non-trivial engineering effort that is out of scope for this thesis. Instead, I opted to directly digest Vello scene encodings, which are designed to be processed efficiently on the GPU. At the moment, we still read the projected encoding back to the CPU, as there is currently no way to pass the data directly into the Vello shader stages. This does not represent a fundamental limitation, and the pipeline is designed to be easily integrated into Vello once such a feature is implemented.

Next, I wrote an end-to-end GPU pipeline for arbitrary rotation, clipping and projection of spherical geometry using WebGPU though the `wgpu`[3] library. All projections with well-behaved, continuous boundaries that require the globe to be cut along the antimeridian or a spherical circle are supported. Most mathematical groundwork was adapted from the `d3-geo`[4] module of D3.js[5]. This data visualization library has been the de facto standard library for highly custom and dynamic visualizations on the web. The algorithms have been heavily modified to run efficiently on the GPU.

The clipping algorithm in particular handles intersection calculations, sorting, point-in-polygon tests as well as graph building and traversal to produce valid polygons without any CPU intervention. To the best of my knowledge, this is the first such algorithm to be published.

The adaptive sampling algorithm also represents a novel contribution, porting the recursive subdivision scheme of `d3-geo` to the GPU. As recursion is not natively supported by compute

[1] `https://github.com/linebender/vello` (last accessed 28.02.2026)
[2] `https://www.w3.org/TR/webgpu/` (last accessed 01.03.2026)
[3] `https://github.com/gfx-rs/wgpu` (last accessed 01.03.2026)
[4] `https://github.com/d3/d3-geo` (accessed 28.02.2026)
[5] `https://github.com/d3/d3` (accessed 23.03.2026)

shaders, I emulated it by converting the algorithm into an iterative loop that manages an explicit stack stored in thread-private memory (see Section 2.7.2 for details).

The dynamic memory requirements of these tasks are handled through an efficient retry mechanism that only checks allocation failure once at the end of the pipeline when we stall for read-back of data anyway. Currently, this requires re-executing the entire pipeline except rotation, but future work may introduce checkpointing to only re-execute the failing stages.

The implementation focuses heavily on correctness and robustness as well as execution performance. To this end, I utilized many techniques to mitigate the common pitfalls of GPU programming, such as atomic contention, bank conflicts and warp divergence, which are explained in more detail in Section 2.7.2. VRAM usage, on the other hand, is not optimized at this stage. The pipeline is designed with an efficient but currently unused bump-allocation strategy in mind. Moving to it fully would greatly reduce memory usage and also increase performance due to better memory locality and less buffer-binding overhead.

Finally, I provide a comprehensive benchmark of the implementation, comparing it to the state-of-the-art `d3-geo` library. The benchmarks additionally include a detailed breakdown of execution time by stage, as well as an analysis of how performance scales with different input sizes and algorithm parameters. Drawing on these insights, I discuss the viability of the current implementation for production use in interactive web mapping.

2. Background & Related Work

2.1. Adaptive Composite Map Projections

Adaptive Composite Map Projections (ACMP) by B. Jenny [Jen12] and Bojan Šavrič [JŠ18] represents the first attempt at designing a digital alternative to Mercator by creating a dynamically adapting map that continuously transforms quasi-optimal map projections into one another based on the scale and central latitude. It mostly follows the recommendations given by *Map projections - a working manual* by Snyder [Sny87], which is often considered the most systematic and comprehensive guide on selecting map projections to date. Due to design considerations and to achieve visual continuity, some deviations from the guideline were shown to be necessary. Visual continuity means that when discontinuities could not be avoided, an effort was made to hide them beyond the current extent of the viewport. To achieve this, various tricks were employed. Even so, for many transitions and edge cases, even visual continuity could not be upheld.

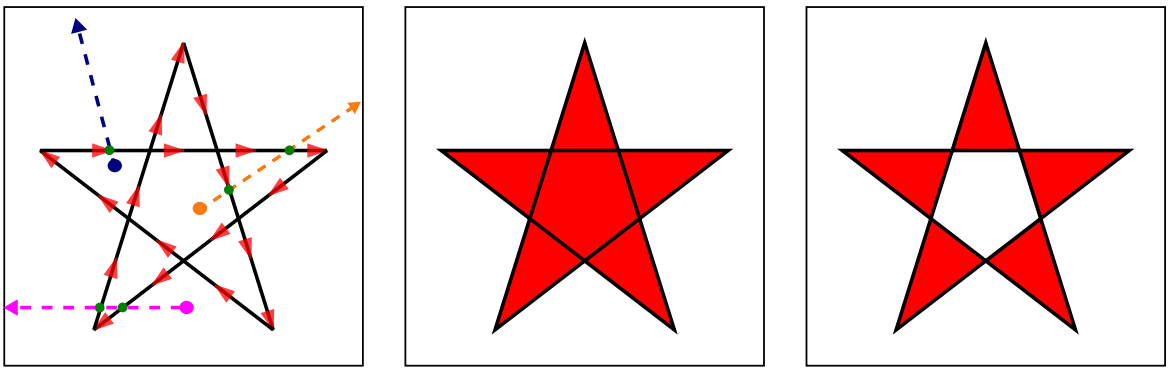
They also provide a WebGL[1]-based demo that projects a raster image of the world[2], but acknowledge the need for vector graphic support. In their short proceedings paper *Rendering Vector Geometry with Adaptive Composite Map Projections* [JŠ13] a quick sketch of such a vector map renderer is laid out.

2.2. Clipping on the Plane

Geometry that is panned across a designated visible area (e.g., the viewport) needs to be clipped at the boundary. Clipping is a ubiquitous operation in computer graphics, with distinct algorithms developed for line and polygon clipping. Classic algorithms for clipping, such as Sutherland–Hodgman and Weiler–Atherton, are well-documented in foundational graphics texts [HDM+14].

In the following sections, we focus on an algorithm that rejoins polygons along the clipping boundary, in contrast to those that simply clip pixels. This more complex class of algorithms is used for viewport clipping, GIS workloads, semiconductor design and many other applications.

[1] `https://get.webgl.org/` (last accessed 28.02.2026)
[2] `https://berniejenny.info/demos/AdaptiveCompositeMapProjections/` (accessed on 28.02.2026)



(a) Analysis of winding number in two locations.

(b) The star painted using the non-zero fill rule

(c) The star painted using the even-odd fill rule

Figure 2.1.: Illustration of the two most common fill rules. Reproduced from [Sta25].

2.2.1. Winding Number and Fill Rules

Polygons are described by a sequence of vertices connected by edges. The winding number of a point with respect to a polygon indicates how many times the polygon winds around that point [HDM+14]. Conceptually, it is calculated by defining a ray from the point to infinity and counting how many times it crosses the polygon's edges, adding one for each clockwise crossing and subtracting one for each counter-clockwise crossing (see Figure 2.1a).

The winding number is crucial for determining whether a point lies inside or outside a polygon. Two common rules based on the winding number are:

- **Even-Odd Rule:** A point is inside the polygon if the winding number is odd and outside if it is even. This rule is simple and works well for polygons without holes.
- **Non-Zero Rule:** A point is inside the polygon if the winding number is non-zero (i.e., not equal to zero) and outside if it is zero. This rule can handle polygons with holes and complex shapes more effectively.

2.2.2. Greiner–Hormann

The Greiner–Hormann algorithm is an efficient and elegant method for clipping arbitrary polygons, including those that are concave or self-intersecting [GH98]. It operates on two input polygons, a subject S and a clip C, and represents them as doubly linked lists of vertices. This data structure allows for the efficient insertion of new vertices during the clipping process. Each vertex node stores its coordinates, pointers to the `next` and `prev` vertices, and several additional fields for managing the clipping logic:

- `intersect`: A boolean flag marking the vertex as an intersection.
- `entry_exit`: A boolean flag to label an intersection as an entry or exit point.
- `neighbor`: A pointer linking an intersection vertex on one polygon to its counterpart on the other.

- `alpha`: A float storing the parametric position of an intersection along an edge, used for sorting.

The algorithm proceeds in three distinct phases.

Phase 1: Intersection Detection and Insertion

The algorithm iterates through each edge $[S_i, S_{i+1})$ of the subject polygon S and tests it for intersection against every edge $[C_j, C_{j+1})$ of the clip polygon C. When an intersection point I is found, its parametric position $\alpha \in (0, 1)$ along the edge $[S_i, S_{i+1})$ is calculated such that $I = (1 - \alpha)S_i + \alpha S_{i+1}$. Two new vertices representing I are created and inserted into the respective lists of S and C, sorted by their α values. These new vertices are then cross-linked using their `neighbor` pointers.

Phase 2: Entry and Exit Labeling

This phase labels each intersection vertex as an `entry` or `exit` with respect to the other polygon's interior. The algorithm traverses each polygon, beginning with a non-intersection vertex S_k. It first determines if S_k is inside or outside the other polygon, typically using the even-odd rule. Based on this initial state, it proceeds labeling the intersections alternatingly along the traversal.

Phase 3: Result Construction

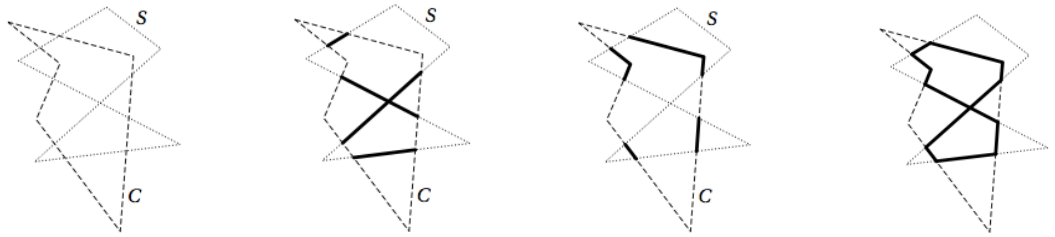
The final phase builds the resulting clipped polygon(s) by tracing the modified vertex lists. The process starts at an unprocessed intersection vertex on S. Depending on the desired Boolean operation (intersection, union, or difference), the algorithm traverses either forward or backward along the polygon's boundary, collecting vertices. At the next intersection, it uses the `neighbor` pointer to switch to the corresponding vertex on C and continues the trace in the direction dictated by the operation. This traversal, alternating between the boundaries of S and C, repeats until the path returns to the starting vertex, completing one result polygon. The process is repeated until all intersection vertices have been visited (see Figure 2.2).

The algorithm handles degenerate cases, such as a vertex of one polygon lying on an edge of the other, by suggesting a slight perturbation of the vertex coordinates to create a non-degenerate configuration.

Parallelization Characteristics

The Greiner–Hormann algorithm's intersection phase has proven to be especially parallelizable. Puri et al. were the first to develop a functional CUDA-based[3] implementation [PP15].

[3] `https://developer.nvidia.com/cuda/toolkit` (last accessed 01.03.2026)



(a) Subject (S) and clip (C) polygons (b) Parts of S inside C ($S \cap C_{int}$) (c) Parts of C inside S ($C \cap S_{int}$) (d) The clipped polygon

Figure 2.2.: Illustration of the Greiner–Hormann clipping process. Reproduced from [Joy15].

The main contribution of Ashan et al. was the introduction of highly effective filtering stages—using common and per-segment bounding boxes—to eliminate 80-99% of the quadratic edge-pair comparisons before the expensive intersection calculations [APP23].

All previously-mentioned works are hybrid algorithms that move the intersection and graph-building stages to the GPU while leaving the traversal of disjoint line segments to produce valid polygons to the CPU. This has been justified with benchmarks that show that labeling and tracing phases together consume less than 1% of the total execution time. Additionally, tracing in particular is inherently sequential.

To date, the only effort to move the reconstruction stage fully to the GPU has been by Ji et al. They use parallel hash tables and memory pools to efficiently traverse and connect edges [JNC25]. They motivate their work with the observation that if subsequent processing stages (e.g., projection with adaptive sampling) require the data to be on the GPU, the cost of copying intermediate results from and to the CPU may outweigh the benefits of a hybrid approach.

2.3. Spherical Geometry

Many of the geometric derivations and algorithmic solutions detailed in the following sections are based on the open-source reference implementation provided by the `d3-geo` library.

2.3.1. Models of the Earth

The true physical shape of the Earth is too complex for geometric computations. Thus, the Earth is modeled by a simpler surface. The standard model for modern geodesy and navigation is a biaxial reference ellipsoid—such as the commonly used WGS 84—which provides a high fidelity approximation of the Earth's actual shape [Env00, Pan14].

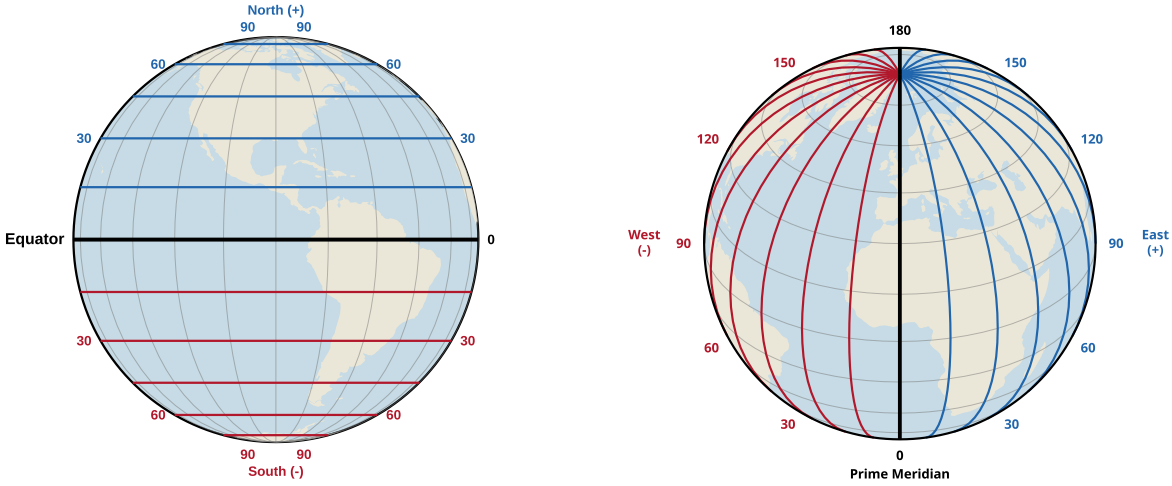
However, for many global-scale applications where high precision is not the primary objective, a spherical model is often sufficient.

2.3.2. Geographic Coordinate Systems

A coordinate system is required to uniquely define the position of a point on the reference surface. A Geographic Coordinate System (GCS) uses angular measurements to specify locations on a three-dimensional sphere or ellipsoid. A GCS is defined by an axis of rotation, a prime meridian, and an angular unit [Env00].

Latitude The Earth's axis of rotation provides a natural reference, defining the North Pole and South Pole. The plane passing through the Earth's center perpendicular to this axis intersects the surface at a great circle known as the Equator. Geodetic latitude (ϕ) is the angle between the equatorial plane and the line normal to the reference surface at a given point [Pan14]. On a sphere, this normal line passes through the center. Latitude values range from -90° (South Pole) to $+90^\circ$ (North Pole), with the Equator at 0°. Circles of constant latitude are known as parallels. The Equator divides the Earth into the Northern and Southern Hemispheres. See Figure 2.3a.

Longitude To define the east-west position, a reference meridian is required. By international convention, this is the Prime Meridian, which passes through the Royal Observatory in Greenwich, London. Longitude (λ) is the angle in the equatorial plane between the plane of the Prime Meridian and the plane of the meridian passing through the point in question. Longitude is measured from -180° (west) to $+180^\circ$ (east) relative to the Prime Meridian (0°). Lines of constant longitude, which are semi-great circles, are known as meridians. The Prime Meridian and its antimeridian (180°) form a great circle that separates the Eastern and Western Hemispheres. See Figure 2.3b.



(a) Parallels of latitude relative to the Equator.

(b) Meridians of longitude relative to the Prime Meridian.

Figure 2.3.: The primary 0° reference planes are denoted in black. Angular measurements are marked in degrees, with positive coordinates (North and East) depicted by blue lines, and negative coordinates (South and West) depicted by red lines.

2.3.3. Mathematical Formulations on the Sphere

Coordinate Transformation

A point P defined by its geographic coordinates (λ, ϕ) on a sphere of radius R can be expressed in a 3D geocentric Cartesian coordinate system (X, Y, Z) using the following transformation [Pan14]:

$$\begin{aligned} X &= R\cos\phi\cos\lambda \\ Y &= R\cos\phi\sin\lambda \\ Z &= R\sin\phi \end{aligned} \tag{2.1}$$

This transformation is fundamental for many geodetic calculations, including vector analysis and coordinate system conversions.

Great Circles and their Intersection

A great circle can be represented by the plane that contains it. This plane is defined by its normal vector $\vec{n}$, which is perpendicular to the surface of the plane. For a great circle passing through two points with Cartesian vectors $\vec{p}_1$ and $\vec{p}_2$, the normal vector is found using the cross product [Whi19]:

$$\vec{n} = \vec{p}_1 \times \vec{p}_2 \tag{2.2}$$

Consequently, the intersection of two great circles with normal vectors $\vec{n}_1$ and $\vec{n}_2$ is a line defined by the vector $\vec{v}_{\text{int}} = \vec{n}_1 \times \vec{n}_2$. The two antipodal intersection points on the sphere are found by normalizing this vector: $\vec{p}_{\text{int}} = \pm\frac{\vec{v}_{\text{int}}}{\|\vec{v}_{\text{int}}\|}$ (see Figure 2.4a).

Small Circles and their Intersection with Great Circles

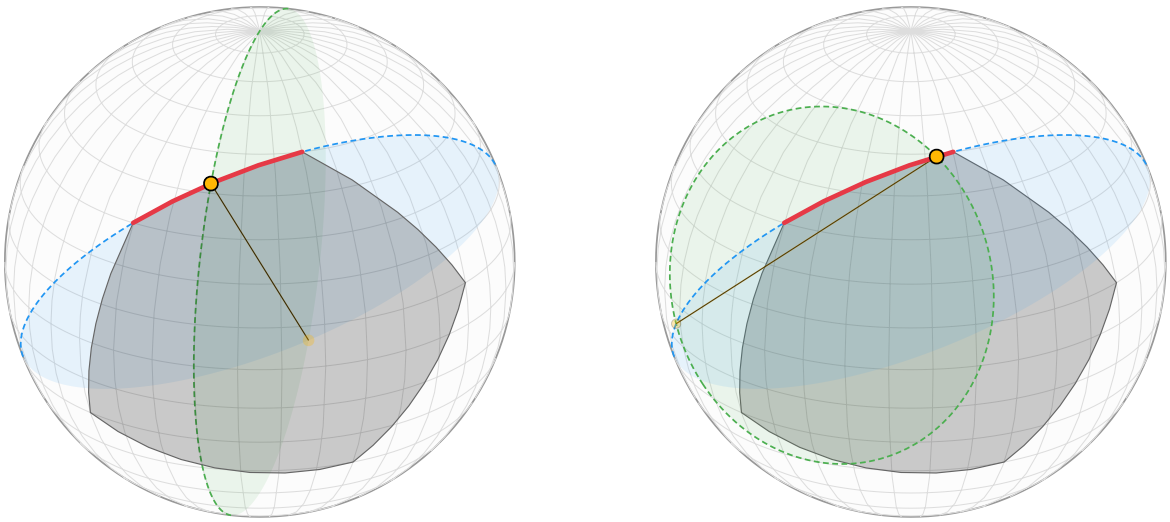
Unlike a great circle, a small circle lies on a plane that does not pass through the origin. It is defined by a normal vector $\vec{n}_1$ and an angular radius r. For a point $\vec{p}$ on the sphere to lie on the small circle, it must satisfy the plane equation $\vec{n}_1 \cdot \vec{p} = \cos(r)$. In the specific context of geographic clipping algorithms, the small circle is often transformed to be centered at $(0, 0)$ in spherical coordinates, corresponding to a normal vector $\vec{n}_1 = (1, 0, 0)$.

To find the intersection of this small circle with a great circle defined by the normal $\vec{n}_2$ (where $\vec{n}_2 \cdot \vec{p} = 0$), we must solve for the line of intersection between the two planes (see Figure 2.4b). The direction of this line is given by $\vec{u} = \vec{n}_1 \times \vec{n}_2$. Any point $\vec{p}(t)$ on this line can be expressed as a linear combination of the normal vectors and the direction vector [Whi19]:

$$\vec{p}(t) = c_1\vec{n}_1 + c_2\vec{n}_2 + t\vec{u} \tag{2.3}$$

The coefficients c_1 and c_2 determine the particular solution and are found by solving the linear system formed by the two plane equations, using the determinant $D = \|\vec{n}_2\|^2 - (\vec{n}_1 \cdot \vec{n}_2)^2$:

$$c_1 = \frac{\cos(r)\,\|\vec{n}_2\|^2}{D}, \qquad c_2 = \frac{-\cos(r)\,(\vec{n}_1 \cdot \vec{n}_2)}{D} \tag{2.4}$$



(a) Intersection of a great circle with a spherical pentagon's geodesic edge.

(b) Intersection of a small circle with a spherical pentagon's geodesic edge.

Figure 2.4.: Geometric intersections on a spherical model. In both figures, a geodesic edge of a shaded spherical pentagon is highlighted in red, with its extended great circle shown as a blue dashed line. The exact intersection between this edge and a secondary intersecting circle (green dashed line) is marked by a yellow point.

Letting $\vec{a} = c_1\vec{n}_1 + c_2\vec{n}_2$ denote this particular solution, the intersection points on the sphere are determined by enforcing the unit magnitude constraint $\|\vec{a} + t\vec{u}\|^2 = 1$. Setting $w = \vec{a} \cdot \vec{u}$, this expands to a quadratic equation for t:

$$t = \frac{-w \pm \sqrt{\Delta}}{\|\vec{u}\|^2} \tag{2.5}$$

where $\Delta = w^2 - \|\vec{u}\|^2 (\|\vec{a}\|^2 - 1)$ is the discriminant. A negative discriminant indicates that the great circle does not intersect the small circle; a zero discriminant corresponds to the tangent case.

Geodesic Distance Calculation

The shortest distance between two points on the surface of a sphere is the arc of the great circle connecting them. This is the spherical geodesic. While the Haversine formula is widely used [Whi19], a vector-based formulation—the spherical case of Vincenty's formula—is often preferred to maintain numerical accuracy for both very small distances and antipodal points.

For two points $P_1(\lambda_1, \phi_1)$ and $P_2(\lambda_2, \phi_2)$, let $\Delta\lambda = |\lambda_1 - \lambda_2|$. We first derive the Cartesian components that relate the two points in geocentric space:

$$\begin{aligned} X &= \cos\phi_2 \sin\Delta\lambda, \\ Y &= \cos\phi_1 \sin\phi_2 - \sin\phi_1 \cos\phi_2 \cos\Delta\lambda, \\ Z &= \sin\phi_1 \sin\phi_2 + \cos\phi_1 \cos\phi_2 \cos\Delta\lambda. \end{aligned} \tag{2.6}$$

The central angle σ is then determined using the two-argument arctangent:

$$\sigma = \operatorname{atan2}\left(\sqrt{X^2 + Y^2}, Z\right). \tag{2.7}$$

The geodesic distance d is finally given by:

$$d = R \cdot \sigma, \tag{2.8}$$

where R denotes the radius of the sphere.

2.4. Clipping on the Sphere

Instead of clipping planar geometry that is panned across a designated visible area, we are mostly interested in clipping geometry that is spherically rotated across the boundary of the projection domain. Clipping geometries on a sphere introduces complex challenges not present on the Euclidean plane ($\mathbb{R}^2$). Foundational work in addressing these challenges was undertaken by Mike Bostock, Jason Davies, and Philippe Rivière for the D3.js library. Their initial implementation, housed within the core `d3-geo` module, focuses on the most common cartographic clipping scenarios: antimeridian cutting and circle clipping[4]. To achieve this, the `d3-geo` algorithm operates across both spherical geometry ($\mathbb{S}^2$) and 3D Cartesian space ($\mathbb{R}^3$). While the input geometries are natively defined by longitudes and latitudes, the pipeline adaptively shifts to 3D vector algebra to efficiently compute rotations and intersections. Despite this non-Euclidean mathematical basis, the library successfully adapts high-level topological concepts from the Greiner–Hormann algorithm to manage polygon re-entry and segment rejoining. Crucially, this core module also implements robust schemes to handle complex topological edge cases, such as polygons that fully encompass the clipping boundary, polygons larger than a hemisphere, and pole crossings.

Building upon this highly capable mathematical foundation, the logic was later generalized into the `d3-geo-polygon`[5] module. This extension's primary contribution is the ability to clip a subject polygon against an arbitrary spherical clipping polygon, rather than being restricted to circles or the antimeridian. Through these advancements, their approach has become the de facto standard for projections in web cartography, having been partially re-implemented across numerous libraries and programming languages.

2.4.1. Spherical Polygons

While planar polygons are constructed from straight line segments, spherical polygons are defined by arcs of great circles. These arcs act as geodesics—the shortest paths between two points on the sphere ($\mathbb{S}^2$)—which inevitably appear curved when projected onto a two-dimensional map.

Because the sphere is a closed, compact surface (having finite area and no edges), a single closed loop does not enclose a distinctly bounded "inside" versus an unbounded "outside"

[4] `https://www.jasondavies.com/maps/clip/` (accessed on 28.02.2026)
[5] `https://github.com/d3/d3-geo-polygon` (accessed 28.02.2026)

as it would on an infinite plane. Instead, it simply partitions the spherical surface into two finite regions. To resolve the ambiguity of which region is the intended polygon, `d3-geo` employs the *right-hand rule*: as one traverses the boundary, the area to the observer's right is designated as the interior. For typical, small polygons, this convention results in a clockwise (CW) winding order when projected into 2D space. However, for polygons larger than a hemisphere, the winding order flips; the "interior" becomes the larger surface area, appearing counter-clockwise (CCW).

This topological distinction invalidates standard planar point-in-polygon methods, such as ray-casting. In the Euclidean plane, a point is considered "outside" if one can trace a path to infinity without crossing a border. On a sphere, however, there is no "infinity"—any ray projected outward eventually wraps around to return to its origin. Consequently, containment cannot be determined simply by counting intersections toward a distant point. Instead, the test must account for the sphere's global topology and the orientation of the boundary arcs to correctly identify which of the two partitioned regions constitutes the polygon's interior.

2.4.2. Calculating Boundary Intersections

Antimeridian Intersections

Intersections are computed between a great-circle arc $[P_1, P_2]$ and the fixed antimeridian, replacing the parametric linear algebra of the planar case with spherical trigonometry. Because the antimeridian is a geodesic of constant longitude ($\lambda = \pm\pi$), the intersection calculation reduces to finding a single unknown latitude ϕ_I. For points $P_1 = (\lambda_1, \phi_1)$ and $P_2 = (\lambda_2, \phi_2)$, this is given by [Ven22]:

$$\phi_I = \operatorname{atan2}\left(\sin\phi_1 \cos\phi_2 \sin\lambda_2 - \sin\phi_2 \cos\phi_1 \sin\lambda_1, \cos\phi_1 \cos\phi_2 \sin(\lambda_1 - \lambda_2)\right). \tag{2.9}$$

By solving directly for ϕ_I using the two-argument arctangent, `d3-geo` bypasses the computational overhead of 3D Cartesian conversions and the subsequent inverse transformations, as subsequent steps require spherical coordinates.

However, the analytic formula can become numerically unstable in certain edge cases. To handle this correctly, endpoints falling exactly on the antimeridian ($\lambda = \pm\pi$) are nudged slightly inward by ϵ. Furthermore, if the longitudinal span $|\sin(\lambda_1 - \lambda_2)| < \epsilon$, the algorithm substitutes the midpoint latitude $(\phi_1 + \phi_2)/2$.

Segments traversing a pole (where $|\lambda_1 - \lambda_2| \approx \pi$) require distinct handling. Even if Equation 2.9 yielded two valid polar points, supplying them directly would cause the sorting function to fail. This function orders intersections by their distance along the vertical clipping boundary ($\lambda = \pm\pi$). Points with intermediate longitudes (e.g., at the pole but not at $\pm\pi$) do not lie on this edge; the function would provide incorrectly sorted values, potentially scrambling the winding order. To guarantee correct topology, the algorithm explicitly generates corner points at $(\pm\pi, \pm\pi/2)$ at the extrema of the sorting domain.

Circle Intersections

For circle clipping, `d3-geo` transitions from spherical coordinates to 3D Cartesian space ($\mathbb{R}^3$) because an arbitrary small circle is a complex curve in (λ, ϕ) space. By lifting the geometry into $\mathbb{R}^3$, the algorithm treats both the arc and the clipping boundary as planes, reducing the intersection calculation to a linear system (see Section 2.3.3).

To avoid unnecessary calculations, the implementation adapts a spherical analogy of the bounding box test of the Cohen–Sutherland [HDM+14] line clipping algorithm. The geometry is first rotated into a local spherical coordinate system where the clipping circle is centered at the origin. The domain is then partitioned into nine regions relative to the circle's angular radius r. For each endpoint, a 4-bit *outcode* c is computed where each bit is set if the point lies outside the central clipping window:

- **Bit 0 (Left):** $\lambda < -r$
- **Bit 1 (Right):** $\lambda > r$
- **Bit 2 (Below):** $\phi < -r$
- **Bit 3 (Above):** $\phi > r$

An arc with endpoint codes c_0 and c_1 is trivially rejected if the bitwise AND $(c_0 \wedge c_1) \neq 0$, identifying arcs that lie entirely within a single exterior half-plane. This pre-filter significantly reduces the number of arcs requiring expensive $\mathbb{R}^3$ intersection math.

The logic handles both small circles ($r < 90°$) and their larger complements by inverting the visibility check accordingly. The clipper detects two primary intersection scenarios: a boundary crossing, where a segment transitions between the interior and exterior regions, generating a single intersection point; and a secant traversal, where a segment begins and ends outside the visible region but dips into the interior, generating both an entry and an exit point.

Unlike clipping against great circles (such as the antimeridian) where numerical degeneracies can be sidestepped by perturbing input coordinates, altering vertex positions near a small circle is geometrically unsafe. Because a great circle arc can be perfectly tangent to a small circle, a spatial perturbation in 2D risks artificially creating secant intersections or destroying valid contact points.

To preclude topological ambiguity without altering the underlying geometry, the algorithm explicitly flags exact boundary collisions—scenarios where a computed intersection coincides precisely with an original polygon vertex. During the rejoining phase, if a clipped segment collapses into a single point (meaning its entry and exit coordinates are identical), the logic inspects this flag. If the flag is absent, the segment is treated as an isolated, fully visible microscopic feature and emitted directly. If the flag is present, the algorithm recognizes a degenerate boundary tangency or vertex collision. It artificially separates the coincident entry and exit points by nudging one coordinate by 2ϵ. Crucially, this microscopic separation occurs strictly within the 1D topological sorting space of the clipping boundary. It preserves the exact 2D geometric integrity of the polygon while guaranteeing a deterministic sorting order, ensuring the alternating entry-exit traversal of the rejoining algorithm does not break down.

2.4.3. Sorting Intersections along the Clipping Boundary

To stitch the clipped polygons back together, the algorithm traverses the clipping boundary to determine when the subject polygon enters or leaves the visible region. `d3-geo` employs a unified sorting strategy for both antimeridian cutting and small-circle clipping by generalizing the intersection coordinates.

For the antimeridian, the sorting coordinates are the spherical coordinates (λ, ϕ). For small circles, the intersections are mapped to a local reference frame (u, v) where the circle behaves geometrically like a line of constant latitude. The sorting function then "unwraps" the circular boundary into a linear sequence S:

$$S(u,v) = \begin{cases} v - \frac{\pi}{2} - \epsilon & \text{if } u < 0 \\ \frac{\pi}{2} - v & \text{if } u \geq 0 \end{cases} \tag{2.10}$$

Here, u acts as the separator between the "left" and "right" sides of the loop, while v orders the points along those sides.

- **Left Side ($u < 0$):** The value increases with v, sorting points upward (e.g., south to north along the west edge).
- **Right Side ($u \geq 0$):** The value decreases as v increases, sorting points downward (e.g., north to south along the east edge).

This logic creates a consistent counterclockwise traversal around the clipping aperture. The ϵ offset at the apex ensures that the end of the ascending path is strictly ordered before the start of the descending path, preventing topological ambiguity where the two sides meet (i.e., at the poles).

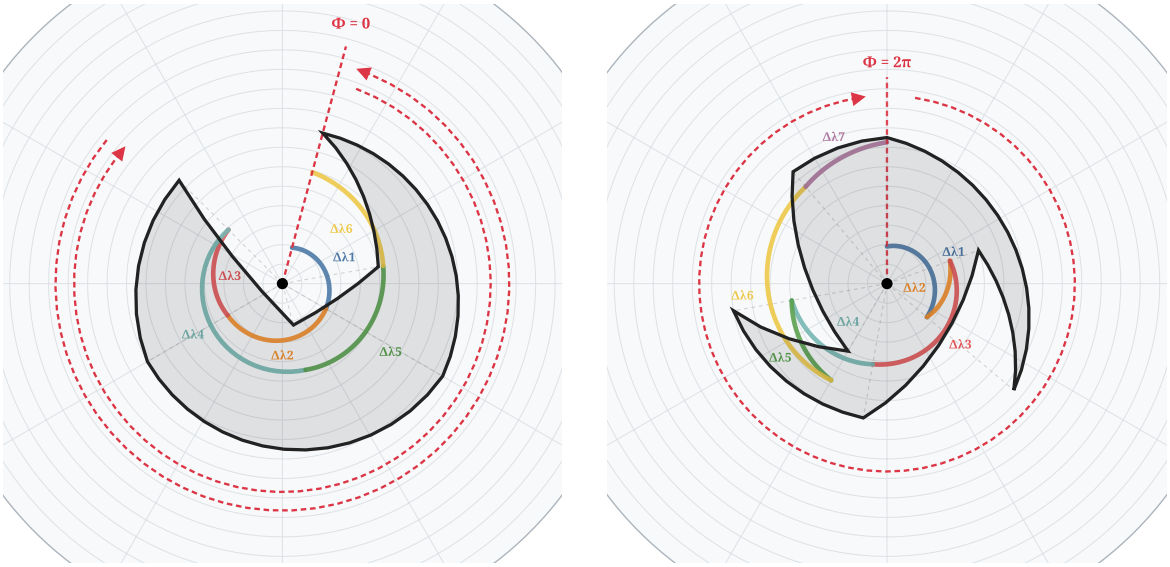
2.4.4. Point-in-Polygon Test

The above scheme requires establishing an initial inside/outside state for a reference point M on the boundary (e.g., the southern apex $(0, -r)$ for small circles, or the South Pole for the antimeridian) from which the traversal begins. As mentioned previously, spherical polygons partition the surface of a sphere into two finite regions, creating a topological ambiguity—without knowing the polygon's orientation relative to the reference, it is impossible to distinguish a small enclosed "island" from the "ocean" surrounding it. Simple ray casting cannot resolve this.

`d3-geo` determines whether M falls inside the subject polygon (the geometry being clipped) by first anchoring the topology to a fixed reference—the South Pole—and then extending this status to M. To do this, the algorithm iterates over the polygon edges to accumulate two distinct metrics relative to the South Pole: the longitudinal winding Φ and the spherical excess Σ.

The longitudinal winding Φ is the accumulated sum of the shortest-path longitude deltas $\Delta\lambda$ across all polygon edges. Because the boundary forms a closed loop, this sum must mathematically evaluate to an integer multiple of a full rotation: $\Phi = 2\pi n$ (see Figure 2.5).

The winding number n establishes how the polygon's boundary encircles the polar axis. A sum of $\Phi \approx 2\pi$ ($n = 1$) indicates the boundary loops around the North Pole, while $\Phi \approx -2\pi$ ($n = -1$) implies a wrapping that physically encircles the South Pole.



(a) Accumulated longitude deltas ($\Delta\lambda$) of a non-enclosing polygon. The eastward and westward traversals cancel out, resulting in a net winding of $\Phi = 0$.

(b) Accumulated longitude deltas ($\Delta\lambda$) of a complex, non-convex polygon. The sequence sums to a full positive rotation ($\Phi = 2\pi$), indicating it encloses the North Pole.

Figure 2.5.: Visualization of the longitudinal winding metric (Φ) used to resolve topological ambiguity. To illustrate the accumulation of angles without overlapping paths, the shortest-path longitude delta ($\Delta\lambda$) of each consecutive polygon edge is mapped to an outward-expanding spiral.

However, a winding number of $\Phi \approx 0$ ($n = 0$) only guarantees that the boundary itself does not loop around the polar axis. This does not definitively prove the South Pole is outside the polygon's interior. Because spherical polygons partition the sphere into two finite regions, a polygon drawn with a reverse winding order (clockwise, per the right-hand rule) designates the larger spherical area as its interior. Such an inverted polygon encompasses the South Pole even if its boundary resides entirely within the Northern Hemisphere.

To resolve cases where $\Phi \approx 0$, the algorithm must evaluate the polygon's signed spherical excess Σ. This metric represents the polygon's spherical area and orientation. If the signed area is negative, the polygon is inverted, meaning its interior covers more than a hemisphere and inherently contains the South Pole.

Computing this area directly on the sphere, however, introduces mathematical instability near the poles. Because all meridians converge at the polar extremes, longitude becomes undefined. This coordinate singularity means that any polygon edge crossing or closely approaching the pole will produce extreme, ambiguous, or undefined longitude deltas ($\Delta\lambda$), causing standard trigonometric area summations to break down.

To evaluate the area and orientation without encountering these artifacts, the algorithm

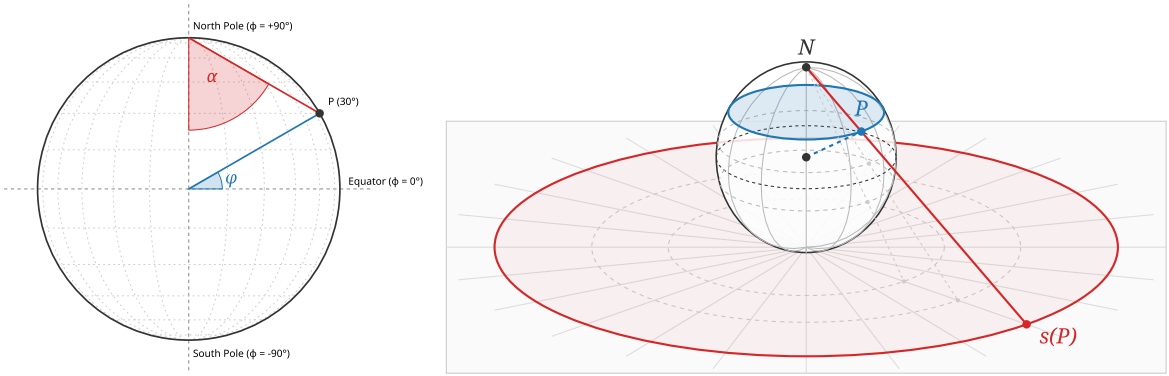


Figure 2.6.: Stereographic projection from the North Pole. The algorithm avoids polar singularities by transforming the spherical latitude ϕ into the half-angle α. This geometric substitution maps the North Pole to infinity and the South Pole to the planar origin ($\alpha = 0$), where the projected radius corresponds directly to the half-angle, ensuring numerical stability when calculating the spherical excess Σ.

employs a stereographic projection originating from the North Pole (see Figure 2.6). By projecting from the North Pole, the northern singularity is mathematically pushed to infinity, effectively "flattening" the sphere without a top boundary. This transforms the spherical latitude $\phi \in [-\pi/2, \pi/2]$ into a polar half-angle $\alpha \in [0, \pi/2]$:

$$\alpha = \frac{\phi}{2} + \frac{\pi}{4} \tag{2.11}$$

Geometrically, this maps the South Pole to a stable planar origin ($\alpha = 0$). Because the stereographic projection is angle-preserving, it provides a rigorous geometric basis for substituting 3D spherical arcs with robust 2D planar-equivalent calculations. By utilizing these projected half-angles, the excess contribution of each segment simplifies to:

$$\Sigma = \sum \operatorname{atan2}\left(k \sin \Delta\lambda, \cos \alpha_i \cos \alpha_{i+1} + k \cos \Delta\lambda\right), \tag{2.12}$$

where the helper constant k is defined for each segment as:

$$k = \sin \alpha_i \sin \alpha_{i+1}. \tag{2.13}$$

The sign and magnitude of the resulting sum Σ allow the algorithm to definitively distinguish between the two regions partitioned by the polygon boundary.

Once the loop completes, these two metrics are synthesized to definitively determine the South Pole's status. The Pole is considered inside the polygon if it is physically enclosed by the winding ($\Phi < -\epsilon$) or if the polygon does not wind around the pole ($\Phi \approx 0$) but defines a negative, clockwise signed area ($\Sigma < -\epsilon^2$). A negative excess implies that the "interior" of the polygon is actually the larger spherical region encompassing the South Pole.

Finally, the status of the target reference point M is derived from the South Pole's status by tracking the parity of edge crossings along the meridian segment connecting M to the

Pole. By anchoring to the projected South Pole, the 3D topological problem is reduced to a standard 2D ray-casting parity check:

$$\text{Contains}(M) = S_{in} \oplus (\text{Meridian Crossings is Odd}) \tag{2.14}$$

This boolean result initializes the rejoining algorithm, allowing it to correctly interpret the intersection sequence along the clipping boundary as alternating entries and exits.

2.4.5. Rejoining

Once the polygon has been cut into isolated visible segments, the final stage reconstructs these linear strips into closed spherical loops. The algorithm maintains two distinct circular lists to manage connectivity:

- Subject List: Preserves the topological order of the original polygon, linking intersection points as they appear along the source geometry. All elements are initially marked as unvisited.
- Clip List: Links the same intersection points but orders them along the clipping boundary (the antimeridian or small circle) as described by Section 2.4.3.

Each node contains a pointer to its counterpart in the other list, acting as a bridge that allows the traversal to switch contexts between following the polygon's edge and following the clipping boundary.

To reconstruct the geometry, the algorithm repeatedly scans the subject list for any remaining unvisited intersection, using it to begin a new traversal that switches between the two lists to close the loop:

- Subject List: Traversed in forward direction. Vertex coordinates of the visible polygon segments are emitted into the output stream until an exit is encountered.
- Clip List: Traversed in backward direction. It generates artificial boundary points by interpolating along the boundary between two intersections.

2.5. Map Projection

A map projection is a mathematical transformation that converts geographic coordinates on a curved surface (sphere or ellipsoid) to planar coordinates. As we use a (unit) spherical model exclusively, coordinates on the Earth's surface for a point P can be fully described by $P = (\lambda, \phi)$, with λ denoting longitude and ϕ latitude.

Most map projections are conceptually derived by projecting the sphere onto a *developable surface*—a geometric shape that can be flattened into a plane without stretching or tearing (see Figure 2.7) [Nat06, Env00]. The three principal families are *cylindrical*, where the surface wraps around the globe (usually touching the equator); *conic*, where a cone sits atop the sphere; and *azimuthal* (or planar), where the map is projected directly onto a flat plane tangent to the Earth.

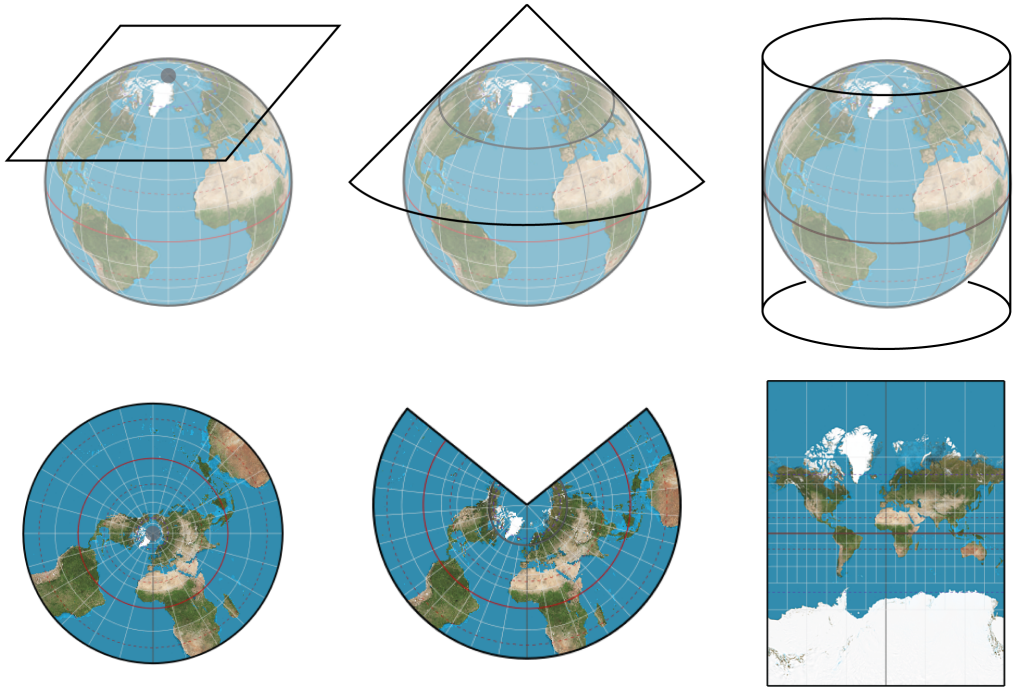


Figure 2.7.: The three principal families of map projections based on developable surfaces. From left to right: an azimuthal (planar) projection tangent to the North Pole, a conic projection over the northern hemisphere, and a cylindrical projection tangent at the equator. The top row illustrates the geometric conceptualization, while the bottom row displays the resulting flattened 2D maps. Reproduced from [Bat17].

The orientation of this surface relative to the globe is defined as the *aspect* of the projection. The *normal* aspect aligns the surface's axis of symmetry with the Earth's polar axis. A *transverse* aspect rotates the surface by 90°, while an *oblique* aspect orients the surface at any other arbitrary angle. Changing the aspect preserves the mathematical properties of the projection (such as conformality or equal-area) but shifts the center of the map and the pattern of distortion, typically ensuring scale is true along the new lines or points of tangency [Sny87].

Many standard projections are defined by smooth, bijective formulas on their intended domain—excluding singularities like the poles—and thus have well-defined inverse formulas. Others intentionally lose information or introduce cuts, making them non-invertible in a global sense. We define a map projection as

$$\begin{cases} x = f(\lambda, \phi) \\ y = g(\lambda, \phi) \end{cases}$$

and:

$$\begin{cases} \lambda = f^{-1}(x, y) \\ \phi = g^{-1}(x, y) \end{cases}$$

as its forward and inverse projections, respectively [Pan14].

When choosing a map projection, one has to consider multiple trade-offs, as all transformations of a sphere to a plane introduce some form of distortion and discontinuity. The main rivaling properties a cartographer may want to preserve are [Sny87]:

- **Distance:** Preservation of scale between specific points (*equidistant*).
- **Shape:** Preservation of local angles and shapes (*conformal* or *orthomorphic*).
- **Area:** Preservation of relative size across the map (*equal-area*, *homolographic*, or *authalic*).
- **Direction:** Preservation of directions from a central point (*azimuthal* or *zenithal*).

Apart from distortion, other quality criteria include graticule and border shape as well as personal aesthetic preferences [Jen12]. To counteract the biggest shortcoming of the Mercator projection, we are mostly interested in the authalic family of projections (see Figure 2.8 as an example).

2.5.1. Fitting of Projection Curvature

When a great-circle arc on a sphere is projected onto a 2D plane, it becomes a curve. Simply projecting the arc's endpoints and connecting them with a straight line is often highly inaccurate. The goal is to approximate this projected curve with a discrete series of straight line segments. However, many traditional curve-fitting algorithms require analytical knowledge of the target curve, such as its derivatives, which is impractical for a general mapping library that must treat arbitrary projections as mathematical "black boxes". A naive approach, such as uniform sampling, is also inefficient; it oversamples regions of low

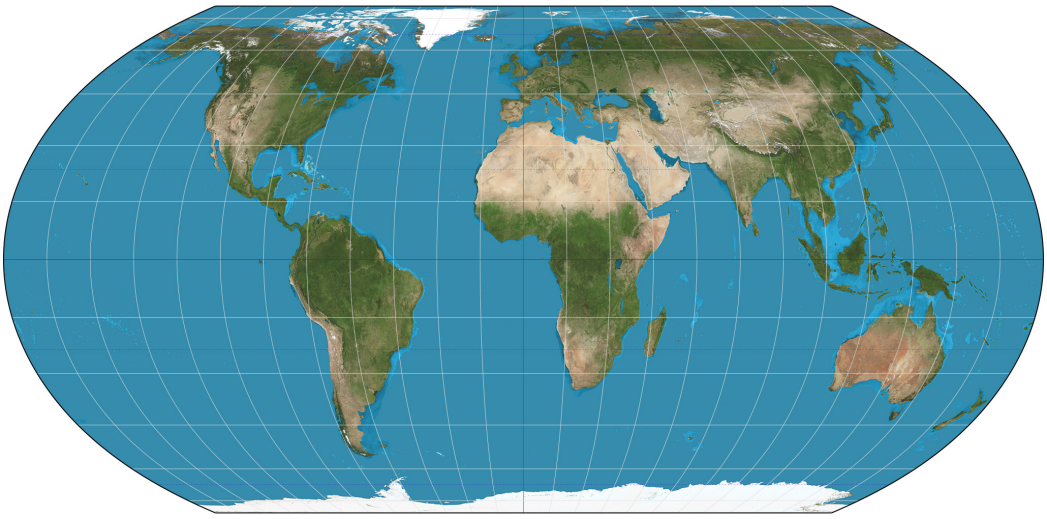


Figure 2.8.: Equal Earth projection. 15° graticule. Imagery is a derivative of NASA's Blue Marble summer month composite with oceans lightened to enhance legibility and contrast. Image created with the Geocart map projection software. Image by Strebe, licensed under CC BY-SA 4.0.

curvature while undersampling regions of high curvature, failing to capture critical details without generating an excessive number of vertices.

A more robust solution can be derived from the principles of line and polygon simplification. Traditionally used to reduce the number of vertices in an existing geometry to improve rendering performance and visual clarity, their underlying logic—evaluating geometric error to retain or discard details—can be effectively adapted to control sampling density during curve fitting. Two of the most well-known algorithms in this domain are the Ramer–Douglas–Peucker [Ram72, DP73] (RDP) and Visvalingam–Whyatt [VW93] algorithms.

Ramer–Douglas–Peucker

The RDP algorithm simplifies a curve using a top-down, recursive splitting approach based on deviation from a baseline.

For a curve segment defined by 2D endpoints p_{start} and p_{end}, the algorithm identifies the intermediate point p_{max} that has the maximum perpendicular distance $d_{\perp}$ from the line segment $[p_{start}, p_{end}]$. If this distance exceeds a specified tolerance δ, the curve is split at p_{max}, and the process is applied recursively to the two new sub-curves. Conversely, if $d_{\perp} \leq \delta$, the segment is considered sufficiently linear, and all intermediate points are discarded. This hierarchical process excels at preserving sharp, characteristic corners and the global structure of the curve.

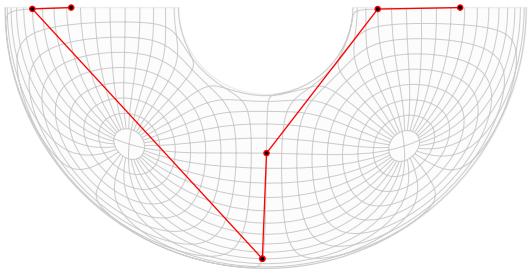
Visvalingam–Whyatt

In contrast to the recursive nature of RDP, the Visvalingam–Whyatt algorithm employs a bottom-up, iterative elimination approach based on local feature significance.

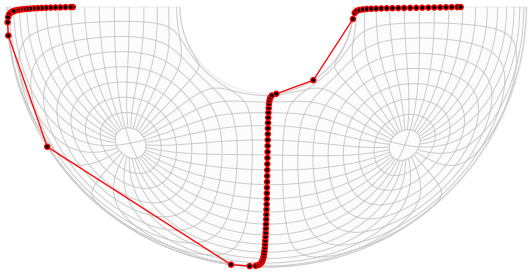
The algorithm assigns an "effective area" to every non-terminal point p_i, calculated as the area of the triangle formed by the point and its immediate neighbors: $A(p_{i-1}, p_i, p_{i+1})$. This area represents the visual importance of the point; smaller areas indicate features that contribute less to the overall shape. The algorithm repeatedly identifies and removes the point with the globally smallest effective area, then updates the areas of the adjacent neighbors. This method progressively removes the "least perceptible" details first, making it particularly effective for producing smoother, more aesthetically pleasing simplifications, especially for natural features.

Adaptive Resampling

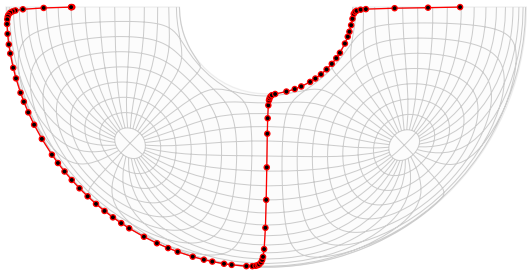
To accurately render map projections, the `d3-geo` module employs an adaptive resampling algorithm[6] that inverts the principles of these simplification techniques to approximate projection curvature.



(a) No sampling (original geometry).



(b) Uniform sampling.



(c) Adaptive sampling.

Figure 2.9.: Adaptive sampling takes ideas from line simplification and prioritizes samples based on curvature, producing high-quality results with low overhead.

For a given great-circle arc between spherical points P_1 and P_2, the algorithm finds the spherical midpoint P_{mid}. All three points are then projected to the 2D plane as p_1, p_2, and p_{mid}. The algorithm calculates the perpendicular distance $d_\perp$ of the projected midpoint p_{mid}

[6] `https://observablehq.com/@d3/adaptive-sampling` (last accessed 01.03.2026)

(a) The outline of a dragon.

(b) The dragon painted using a blue fill.

(c) The dragon painted using a blue stroke of width 2 pixels.

Figure 2.10.: Illustrations of the different drawing modes. Reproduced from [Sta25].

from the straight line segment $[p_1, p_2]$. If this distance exceeds a tolerance δ, the original arc is subdivided at P_{mid}, and the process is applied recursively to the two new, smaller arcs. This method ensures that the sampling density is automatically adjusted based on the local curvature of the projection, providing an accurate representation of the curve with fewer vertices in flatter regions and more vertices in areas of high curvature (see Figure 2.9).

2.6. Vector Graphics

Vector graphics represent images using mathematical primitives such as points, lines, curves, and polygons. Unlike raster graphics, which are composed of a fixed grid of pixels, vector graphics are defined by geometric formulas. This gives them their hallmark characteristic: resolution independence. A vector image can be scaled to any size without loss of quality, as the underlying math is simply re-evaluated for the new dimensions. This makes them essential for GIS and web mapping systems, where maps must remain clear and precise at any zoom level. Each point is stored as a coordinate pair (e.g., (X, Y)), and features are styled using two primary operations: *stroking*, which applies a style to a path's outline, and *filling*, which applies a style to the interior of a closed shape (see Figure 2.10) [Neh20a, Neh20b].

Rendering a vector graphic involves converting its mathematical definition into a grid of pixels for display, a process called *rasterization* [Neh20a]. Traditionally, this was handled by the CPU, which is inefficient for complex scenes like detailed maps and causes performance bottlenecks during interactive operations like panning and zooming.

Most early attempts at hardware acceleration shifted this workload to the GPU through *tessellation* [LB05]. This technique approximates vector paths by breaking them down into a large number of simple triangles that the GPU can process efficiently. While an improvement, it often required significant pre- or post-processing on the CPU. Modern GPU-only tessellation approaches exist, but they are heavily patented or hard to implement [Kil20].

Some cutting edge research on vector graphics rendering has moved beyond tessellation, adopting algorithms that perform rasterization directly on the GPU with massive par-

allelism [GLFN14, NH08, LU24, LHZ16]. Many modern renderers like Google's `Forma`[7], Mozilla's `Pathfinder`[8] as well as `Vello` and `Slug`[9] make use of some of these new techniques to enable real-time rendering of complex vector graphics. They all use compute shaders (more on this in Section 2.7.2) to some extent to calculate the exact pixel coverage for each path segment simultaneously, handling complex fills, strokes, and high-quality anti-aliasing with remarkable speed.

In GIS and digital cartography, common vector data formats are:

- Shapefile: A popular, albeit older, format for storing non-topological geometry and attribute data [Env98].
- GeoJSON & TopoJSON: A lightweight, JSON-based open standard widely used in web-mapping for representing geographical features. TopoJSON is an extension of GeoJSON that encodes topology [10] by storing shared boundaries (arcs) only once. This eliminates redundancy for smaller file sizes and prevents visual artifacts like gaps or overlaps when line simplification algorithms are applied.
- SVG (Scalable Vector Graphics): An XML-based format for displaying high-quality, scalable graphics on the web [SVG18].
- Vector Tiles: Multiple standards for delivering mapping vector data in small, tiled chunks (e.g., `.mvt` format) [Map16]. This allows clients to fetch only the data needed for the current map view, dramatically improving performance and bandwidth efficiency for large datasets.

2.7. GPU APIs

The practice of using a GPU, traditionally designed for rendering graphics, for general computing tasks is known as General-Purpose GPU (GPGPU) computing. This approach leverages the massively parallel architecture of GPUs to accelerate parallel workloads far beyond the capabilities of traditional CPUs.

2.7.1. GPGPU

Initially, GPGPU compute was accomplished by mapping data to textures and using pixel shaders to perform calculations—a cumbersome workaround [HDM+14]. Adding to the difficulty, there exist a number of low-level graphic APIs (e.g., Direct3D[11]/DirectX[12] for Windows, Metal[13] for Apple, and OpenGL[14]/Vulkan[15] as cross-platform options), often

[7] `https://github.com/google/forma` (last accessed 28.02.2026)
[8] `https://github.com/servo/pathfinder` (last accessed 28.02.2026)
[9] `https://sluglibrary.com/` (last accessed 28.02.2026)
[10] `https://github.com/topojson/topojson-specification` (last accessed 08.03.2026)
[11] `https://learn.microsoft.com/en-us/windows/win32/direct3d` (last accessed 01.03.2026)
[12] `https://learn.microsoft.com/en-us/windows/win32/directx` (last accessed 01.03.2026)
[13] `https://developer.apple.com/metal/` (last accessed 01.03.2026)
[14] `https://www.opengl.org/` (last accessed 01.03.2026)
[15] `https://www.vulkan.org/` (last accessed 01.03.2026)

with varying performance characteristics across GPU architectures (AMD/NVIDIA/Intel). Writing code that performs well across these different platforms is a significant challenge. Developers often have to introduce separate code paths or rely on complex abstraction layers that may introduce overhead.

While the landscape matured with the introduction of dedicated compute frameworks, similar fragmentation persists. NVIDIA's CUDA (Compute Unified Device Architecture) emerged as the dominant but proprietary framework, offering a robust ecosystem but locking developers into NVIDIA hardware. In response, open standards like OpenCL[16] (Open Computing Language) were created to provide a cross-vendor solution. However, OpenCL historically lagged behind CUDA in terms of features, performance, and developer tooling. A newer alternative is AMD's ROCm[17] (Radeon Open Compute), which is optimized for AMD GPUs but never achieved widespread adoption, due to the same shortcomings as OpenCL while also providing inadequate documentation and library support.

2.7.2. Compute Shaders

Some graphics tasks require compute capabilities that do not fit into the classic graphics pipeline stages, or an application may require GPU compute for both graphics and non-graphics workloads. As an alternative to compute-centric APIs, modern graphics APIs added support for compute shaders as first-class citizens. Compute shaders are programs that run outside the traditional rendering pipeline, designed specifically for general-purpose computation.

Execution Model

The fundamental computing unit of a compute shader is a *workgroup*, which may contain between 1 and 1024 threads for consumer hardware. These threads can be arranged in one, two, or three dimensions to naturally map onto multidimensional data structures like textures or volumetric grids, thereby simplifying coordinate math and improving memory cache performance. Workgroups share a common fast memory space and can synchronize with each other, making them ideal for tasks that require inter-thread communication or collective operations.

A workgroup is a logical abstraction defined in software. The GPU hardware physically executes threads in strictly synchronized bundles known as *warps* (on NVIDIA) or *wavefronts* (on AMD) [WHKH22]. These follow a *SIMT* (Single Instruction, Multiple Threads) architecture, where every thread in a warp shares a single instruction pointer and advances in lockstep to execute the exact same instruction at the exact same moment. This rigid synchronization means that if threads diverge (e.g., via conditional if-else logic), the hardware cannot run both branches simultaneously. Instead, it must serialize them—pausing one set of threads while the others execute—which significantly reduces efficiency.

[16] `https://www.khronos.org/opencl/` (last accessed 01.03.2026)
[17] `https://www.amd.com/en/products/software/rocm.html` (last accessed 01.03.2026)

The standardized name for a warp or wavefront is *subgroup*. But not all APIs expose subgroup primitives as they are meant to provide access to features that older hardware-subgroups don't actually support. On modern GPUs, they have shared register access, allowing for even faster memory transfers than workgroup-level shared memory. Providing access to subgroup primitives no matter the underlying GPU architecture would sometimes require emulating the functionality, which may lead to performance traps. Still, we will refer to warps/wavefronts as subgroups going forward.

Subgroups inside a workgroup can be scheduled independently, allowing the GPU to hide latency by switching between them when one is stalled (e.g., waiting for memory). This means that while threads within a subgroup must execute in lockstep, different subgroups can run concurrently, maximizing hardware utilization.

GPUs are organized into blocks generally referred to as *Compute Units* (CUs). A typical consumer GPU might have anywhere from 20 to over 100 CUs. Inside each Compute Unit is a massive array of simple *Arithmetic Logic Units* (ALUs), often referred to as "lanes" or "stream processors." When a dispatch occurs, a scheduler assigns workgroups to available CUs. While a single CU can manage multiple workgroups and store their data in its registers, it can only physically execute a limited number of subgroups at any given instant based on its available ALUs.

Here we differentiate between the parallelism and concurrency GPUs provide. Parallelism happens across the GPU, where different Compute Units run instructions simultaneously. Concurrency happens within a specific Compute Unit: To keep its ALUs busy, the Unit holds many more subgroups "resident" in memory than it can execute at once. It aggressively hides latency by context-switching—instantly swapping a stalled bundle (waiting on slow memory) for one that is ready to calculate [HP17].

Memory Hierarchy

GPUs have a distinct memory hierarchy (see Figure 2.11) [KP25]. The largest but slowest tier is *global memory* (VRAM). Because VRAM is accessed in fixed-size chunks, scattered access patterns waste bandwidth by fetching unneeded data. To prevent this, developers must ensure *memory coalescing*, where threads read consecutive addresses so that a single memory transaction can provide data for the entire subgroup.

Closer to the hardware execution units lies *shared memory*, a small, high-speed block of on-chip memory visible only to threads within a single workgroup. Unlike transparent CPU caches, shared memory is explicitly programmable; data can be loaded here manually to enable fast inter-thread communication and reduce access to global memory.

At the finest level of granularity are *registers*, which serve as private, zero-latency storage for individual threads. While registers are the fastest resource, they are scarce; excessive usage can limit the number of active subgroups, reducing the GPU's ability to hide latency. If registers are exhausted, data is "spilled" to slow off-chip memory, severely degrading performance.

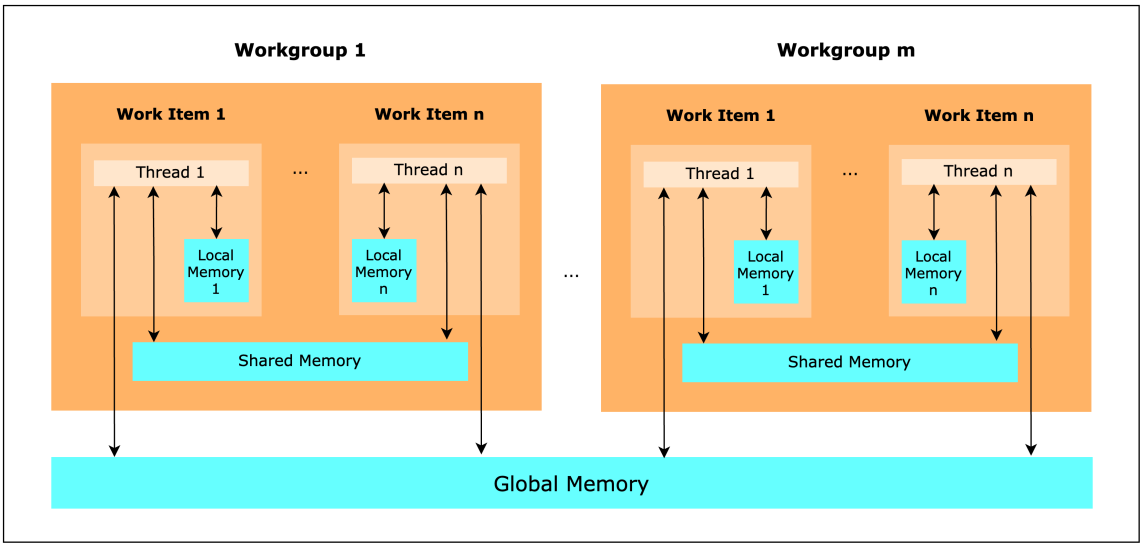


Figure 2.11.: Hierarchical GPU memory architecture illustrating data visibility and access scopes across individual work items, workgroups, and global device memory. Reproduced from [KP25].

Synchronization and Atomics

Because thousands of threads may access the same memory buffers in parallel, managing race conditions—where multiple threads attempt to read and write to the same memory address simultaneously—is critical.

The primary mechanism for synchronization within a workgroup is an *execution barrier*. When a thread encounters a barrier, it stalls execution until all other threads in the same workgroup have reached that specific point. This guarantees that all instructions up to the barrier have been completed by all threads and consequently that data written to shared memory is current.

For finer-grained control, compute shaders provide *atomic operations* (or simply "atomics"). Atomics allow threads to perform Read-Modify-Write (RMW) operations on integer values in global or shared memory without distinct locks. Commonly supported atomics are:

- **Atomic Add/Subtract:** Adds/subtracts a value from a variable.
- **Atomic Exchange:** Replaces the value of a variable with a new value.
- **Atomic Compare-and-Swap (CAS):** Compares the current value of a variable to an expected value, and if they match, swaps it with a new value.
- **Atomic Min/Max:** Updates a variable to the minimum or maximum of its current value and a new value.
- **Atomic Bitwise Operations:** Performs bitwise AND/OR/XOR on a variable.

All atomic operations return the original value of the variable before the operation.

The hardware ensures these operations occur sequentially. While convenient, atomics serialize access to specific memory addresses. If many threads contend for the same atomic variable, performance degrades significantly.

Performance Characteristics

The GPU scheduler's capacity can be defined by two distinct limits [WHKH22]. First, the total number of threads that can be resident on the chip (hiding latency) is:

$$N_{\text{resident}} = N_{\text{CU}} \times W_{\text{limit}} \times S_{\text{subgroup}} \tag{2.15}$$

where:

- N_{CU} is the number of Compute Units (or SMs).
- W_{limit} is the maximum resident subgroups per CU.
- S_{subgroup} is the size of a subgroup (threads per subgroup).

Second, the theoretical limit for instantaneous parallel execution (threads physically firing per clock cycle) is:

$$N_{\text{active}} = N_{\text{CU}} \times W_{\text{issue}} \times S_{\text{subgroup}} \tag{2.16}$$

where W_{issue} is the number of subgroups that can issue instructions simultaneously per CU (typically 1 to 4 depending on architecture). The peak throughput $W_{\text{issue}} \times S_{\text{subgroup}}$ is equal to the number of ALUs in a Compute Unit.

Fundamentally, GPUs prioritize *throughput* over *latency*. While CPUs are designed to minimize the execution time of sequential instruction streams, GPUs offset relatively slow per-core performance by processing large volumes of data simultaneously. Optimizing GPU performance requires balancing *occupancy*—the ratio of active subgroups to the maximum supported subgroups—against resource constraints like register pressure and shared memory usage. While high occupancy helps hide memory latency, it is not always beneficial if achieved at the cost of excessive register spilling or cache contention.

Workloads are generally categorized as either *compute-bound* (limited by ALU throughput) or *memory-bound* (limited by memory bandwidth). Compute-bound kernels benefit from algorithmic optimizations, while memory-bound kernels require improved access patterns (coalescing) or increased arithmetic intensity (performing more operations per byte loaded).

For modern GPUs, maximum throughput equates to around 0.2 TFLOPS (trillions of floating-point operations per second) for older low-end mobile devices and up to 100 TFLOPS for high-end consumer cards of the current generation. By comparison, general-purpose CPUs can achieve approximately 0.05 TFLOPS on mobile to just 2–4 TFLOPS on flagship desktop processors, illustrating the GPU's significant throughput advantage for parallel workloads.

It is important to note that these figures refer specifically to 32-bit Single Precision (FP32). On consumer-grade GPU hardware, 64-bit Double Precision (FP64) is frequently either

unsupported or subject to a significant performance penalty, typically operating at 1/32 or 1/64 the throughput of FP32 due to a lack of sufficient dedicated 64-bit ALUs.[18]

Common Optimization Strategies

The following strategies are often employed to maximize compute shader efficiency [WHKH22, KP25].

Thread Coarsening Instead of a 1:1 mapping between threads and data elements, thread coarsening involves assigning multiple data elements to a single thread. This technique increases the arithmetic intensity per thread and reduces the overhead associated with launching threads. Also, by processing multiple items serially within a register file, the GPU can often achieve better instruction-level parallelism and reduce redundant memory fetches.

Atomic Aggregation To mitigate atomic contention, *aggregation* (or privatization) can be employed. Instead of accessing global memory immediately, threads accumulate results into registers or fast shared memory first. Once the workgroup completes its local batch—or reaches a specific data boundary—a single representative thread performs one global atomic operation to commit the aggregate result.

Minimizing Bank Conflicts Shared memory is physically divided into 32 distinct modules known as *banks*, one for each thread in a subgroup. If multiple threads in a subgroup attempt to access different addresses that map to the same memory bank simultaneously, the requests are serialized, causing a *bank conflict*. Developers can align data access patterns—often by padding arrays or permuting indices—to ensure that threads within a subgroup access distinct banks in parallel, thereby preserving full bandwidth.

Subgroup Divergence Reduction As noted in the execution model, branching logic splits subgroups. Optimization often involves restructuring algorithms to ensure that conditional checks result in the same boolean evaluation for all threads in a subgroup, using *branchless programming* techniques to replace 'if-else' control flow with mathematical selection, or simply using a selector function that computes both branches and chooses the result based on a condition.

2.7.3. WebGPU: Cross-Platform Compute for the Web

While compute shaders as a whole provide a semi-standardized set of GPGPU compute capabilities across hardware, they are still part of the fractured landscape of native graphics APIs. Several functional differences persist. Although abstraction layers have been developed, they are often either hard to use or come with a performance penalty. WebGPU is the first modern API that is designed from the ground up to be supported by all major platforms as

[18] `https://www.techpowerup.com/gpu-specs/` (last accessed 01.03.2026)

well as browsers. Consequently, it only supports a common subset of features that can be efficiently implemented across all GPU architectures.

Browsers implement this standard by translating its API calls into commands for the underlying native APIs, though fully native WebGPU implementations are also emerging. This solves the cross-platform problem at a higher level, while also bringing support for compute shaders to the web. The previous web standard for GPU access, WebGL, focused only on the traditional graphics pipeline and efforts to retrofit compute capabilities never came to fruition.

WGSL and WESL WebGPU introduces its own purpose-built shading language, WGSL[19] (WebGPU Shading Language). Designed to be both safe and efficient, WGSL provides a high-level abstraction that simplifies the inherent complexities of GPU programming. It is a statically typed language that enforces rigorous rules for memory access and resource management. Ultimately, WGSL's greatest strength mirrors the core promise of WebGPU itself: a guarantee of seamless cross-platform compatibility.

While WGSL is already quite powerful, it lacks many of the "niceties" one might have come to expect from a general-purpose programming language. Apart from a rather bare-bones standard library, it also misses features such as a module system with imports, generics, conditional compilation at run- and compile-time as well as packaging. WESL[20] (WGSL Enhanced Shading Language) is a community-driven superset of WGSL that adds these missing features and compiles down to standard WGSL.

[19] `https://www.w3.org/TR/WGSL/` (last accessed 01.03.2026)
[20] `https://wesl-lang.dev/` (last accessed 01.03.2026)

3. Method

The map projection renderer is implemented in Rust[1], as it is one of three languages that provide a mature WebGPU API (i.e. the `wgpu` crate). It is also the language the rendering backend Vello is written in. WESL is used for defining and importing shader packages as well as for dynamic shader compilation. The project started as a fork of the `cartography`[2] crate, which provided a skeleton for rendering GeoJSON using Vello. Apart from the GPU projection pipeline described in this chapter, our implementation makes better use of Vello's capabilities through an improved styling and transformation system.

3.1. Winding Order

As hinted at in the Background section, GeoJSON (RFC 7946) [BDD+16] and `d3-geo` employ opposing topological conventions for defining the interior of a bounded area, despite both confusingly terming their approach the "right-hand rule." RFC 7946 applies the mathematical right-hand rule for normal vectors on a flat plane, placing the interior to the *left* of the directed boundary (dictating counter-clockwise exterior paths). Conversely, `d3-geo` applies a *surveyor's rule* for spherical geometry, defining the interior as the region to the *right* of the traversal path (requiring a clockwise exterior boundary for polygons smaller than a hemisphere).

For this reason, rendering GeoJSON-compliant paths smaller than a hemisphere with `d3-geo` often yields the exact spherical complement of the intended shape. As the vast majority of sources contain "small" polygons exclusively our pipeline instead applies a spherical *left-hand* rule to determine the interior of paths (see Figure 3.1). This way, our implementation does not require pre-processing of input data most of the time.

3.2. Encoding

Vello runs a sophisticated compute shader pipeline to render vector graphics efficiently on the GPU. It takes a host-side scene description and produces an encoding that can be efficiently decoded and rendered on the GPU. The encoding is separated into functionally distinct streams, some of which are compacted. Most important to us are the path data and path tags streams, which describe the geometry of the scene. The tag stream contains information about the type of each path segment (see Table 3.1) and the path data stream contains the

[1] `https://rust-lang.org/` (last accessed 23.03.2026)
[2] `https://gitlab.com/cyloncore/cartography-rs` (last accessed 01.03.2026)

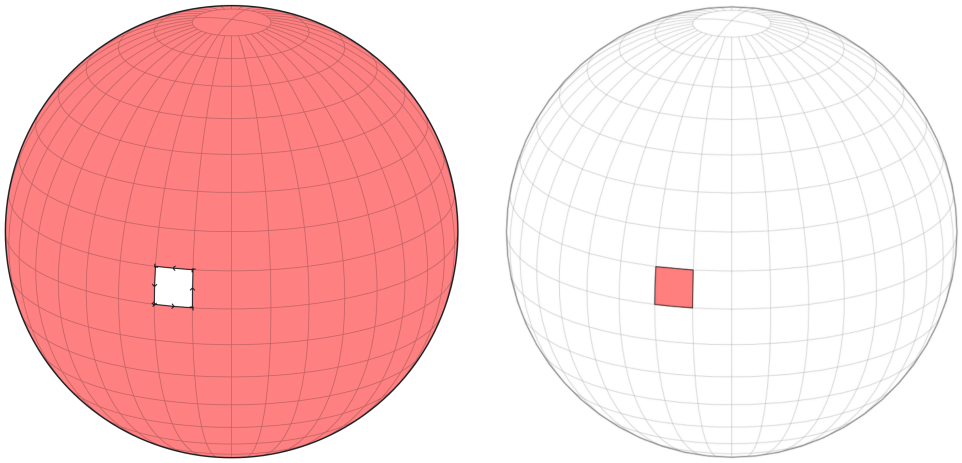


Figure 3.1.: Left: `d3-geo` rendering an orthographic projection of a spherical rectangle with counter-clockwise winding using its right-hand rule. The arrows indicate the direction of traversal. Right: Our tool rendering the same GeoJSON using the left-hand surveyor's rule.

actual vertices and control points. Other streams include style information (e.g. fill and stroke color, stroke width, etc.), affine transformations and more.

Bits 0–1 of the one-byte path tags indicate the type of path segment: 1 for `LINETO`, 2 for `QUADTO` and 3 for `CUBETO`. Multiplying the value of this two-bit `PATH_TAG_SEG_TYPE` with the value of the `PATH_TAG_F32` bit provides the exact amount of space the segment takes up in the path data stream. Other bits may mark delimiting segments of a path/subpath or the beginning of a new style definition.

Tags are one byte each, a granularity not supported by WebGPU. For this reason, four tags are packed into one 32-bit integer. To compute spatial offsets, we employ a *parallel prefix sum* (or *scan*) algorithm. Formally, a scan takes a sequence of elements and an associative binary operator, generating an output sequence where each element at index i is the cumulative result of applying the operator to all elements up to i (inclusive scan) or $i-1$ (exclusive scan). By defining carefully-designed "plus" and reduction operators tailored to our packed integer representation, the scan yields structures termed `TagMonoids`. From these monoids, precise offset information for a specific tag can be derived efficiently using bit manipulation.

While the encoding is multi-threading friendly and very efficient in general, it still becomes a bottleneck, as we need to be able to recreate the full scene every frame. Vello supports affine transformations with almost no overhead, as these can be applied in a single matrix multiplication on the GPU. However, map projections are not affine transformations. We thus need to process Vello scene encodings in multiple compute shader passes directly.

3.3. "Dynamic" Memory Allocation

Compute shaders do not support dynamic memory allocation. This means that we cannot create new arrays or lists of arbitrary size at runtime. This restriction to pre-allocated

Table 3.1.: Path tag encoding.

Bit	Hex	Constant	Purpose
7	`0x80`	(unused)	This bit is currently not assigned a meaning.
6	`0x40`	`PATH_TAG_STYLE`	Indicates that a new style definition begins at this point.
5	`0x20`	`PATH_TAG_TRANSFORM`	Indicates that a new affine transform begins at this point.
4	`0x10`	`PATH_TAG_PATH`	Marks the end of a path.
3	`0x08`	`PATH_TAG_F32`	Indicates if the path coordinate is `f32` or `i16`.
2	`0x04`	`PATH_TAG_SUBPATH_END`	Marks the last segment of a subpath.
1	`0x02`	`PATH_TAG_SEG_TYPE`	Part of a 2-bit field that defines the segment type (Line, Quad, or Cubic).
0	`0x01`	`PATH_TAG_SEG_TYPE`	Part of a 2-bit field that defines the segment type (Line, Quad, or Cubic).

memory is a significant challenge for GPGPU programming and our pipeline specifically. Our algorithm contains multiple stages with dynamic memory requirements and later stages rely on data generated in earlier stages to determine their own memory needs.

There are multiple ways one can deal with this limitation, and the best approach depends on the specific use case. One common method is to over-allocate memory based on worst-case scenarios. This means that we allocate more memory than we expect to need. This is often combined with *memory pool* (arena) allocation. If a good heuristic exists, this approach is often preferred due to its simplicity and speed.

If no such heuristic is available, another approach is to use multiple passes. In the first pass, we analyze the data to determine memory requirements for subsequent passes. This allows exact memory allocation, but comes at the cost of having to read back data from the GPU to the CPU, which can be detrimental to performance. Every data transfer represents a hardware synchronization point, which completely stalls the GPU pipeline.

A more advanced technique—especially well suited for real-time multi-stage pipelines like ours—is to use *indirect dispatching* (explained further below) combined with a single read-back at the end of the pipeline. In this approach, we may use a heuristic that under-allocates memory. Every pass with dynamic memory needs computes and writes its buffer size requirements to a dedicated `bump` allocation buffer during normal execution. If a thread detects that it has run out of memory, it writes a special failure flag to the same buffer but still computes its required memory. All subsequent passes have to be made failure-aware, meaning that they should only perform the minimal amount of work if a prior stage failed. Some stages can completely skip all work, others may still need to calculate their own memory requirement. At the end of the pipeline, we read back the allocation-and-failure buffer to

Listing 1 Bump Allocators Struct

```
// Bitflags for each stage that can fail allocation.
const STAGE_INTERSECTION: u32 = 1u << 0u; // 0x1u
const STAGE_WRITE_CLIPPED: u32 = 1u << 1u; // 0x2u
const STAGE_SUBPATH_INFO_PROJECTION: u32 = 1u << 2u; // 0x4u
const STAGE_ADAPTIVE_SAMPLING: u32 = 1u << 3u; // 0x8u

struct BumpAllocators {
    failed: atomic<u32>,
    intersections_open: atomic<u32>,
    intersections_closed: atomic<u32>,
    path_data_clipped: atomic<u32>,
    path_tags_clipped: atomic<u32>,
    n_subpaths_clipped: atomic<u32>,
    path_data_projected: atomic<u32>,
    path_tags_projected: atomic<u32>,
}
```

the CPU. If no failure flag is present, we can proceed as normal. If a failure flag is set, we re-allocate all buffers based on the exact memory requirements and re-run the pipeline. This approach allows us to have a single read-back stall for the entire pipeline.

This is the scheme chosen for our implementation. Due to current limitations of the Vello API, we need to download the scene geometry after every iteration and splice it into the host-side scene encoding anyway. Vello then re-uploads it to the GPU. Even if this were no longer necessary, we can not avoid at least one stall, but the overhead is manageable as the `bump` structure currently has a size of 32 bytes (see Listing 1).

While already very performant, several optimizations have been designed for but not yet fully implemented. All stages are made failure aware, but except for the rotation stage, we re-run the entire pipeline on under-allocation. This is not a big problem for a real-time application with relatively stable memory requirements, but could nonetheless be improved. Additionally, we almost never re-use buffers across stages. Performance-wise, this increases the bookkeeping overhead of buffer binding management on the host side. More importantly, it significantly increases the amount of VRAM used. A final optimization would be to implement actual bump buffer allocation, where we upload only a few large buffers and corresponding configuration buffers that contain offsets into these buffers for each stage. This is likely to noticeably improve memory locality, overall buffer- and cache sizes as well as reduce the overhead of buffer binding management.

Indirect Dispatch is a capability of GPU APIs where the number of workgroups to be launched for a compute shader is not specified directly in the host-side dispatch call, but rather read from a buffer on the GPU. This allows the GPU to dynamically determine the amount of work to be done based on data generated during previous compute passes, even if the buffer sizes are over-allocated. This can even be used to dispatch a compute shader with zero workgroups if a previous stage failed, effectively skipping the entire pass.

3.4. Stage 0: Setup

This shader stage runs whenever the scene geometry changes due to user interaction (adding new layers, modifying existing paths, etc.). It is the first stage of the pipeline and prepares the data for subsequent stages. It is not projection-specific and does not need to be re-run when projection parameters change.

We initially compute the number of subpaths in the scene on the CPU as this information is crucial for our pipeline but not provided by a Vello scene encoding. This is done by iterating over the tag stream and counting the number of tags with the `PATH_TAG_SUBPATH_END` bit set.

Mirroring this decision, the `TagMonoid` (see Listing 2) structure used by Vello contains an index field for paths but not for subpaths. Here, a slightly modified version of Vello's parallel prefix sum algorithm is used to compute the subpath indices and store them in our own `TagMonoids`. To achieve this the unused `pathseg_ix` field is replaced by a `subpath_ix` field, which is populated by changing the `reduceTag` function to count `PATH_TAG_SUBPATH_END` bits instead of the number of segment tags. We also chose to upload only the path tags and path data instead of the full scene encoding, as we currently can not pass on our updated scene encoding to the Vello pipeline directly anyway. Our custom `TagMonoids` contain the following fields:

Listing 2 TagMonoid Struct

```
struct TagMonoid {
    trans_ix: u32,
    subpath_ix: u32,
    pathseg_offset: u32,
    style_ix: u32,
    path_ix: u32,
}
```

Next, a simple kernel processes the new tag monoids as well as path tags and styles to compute additional metadata describing the path data and path tag streams. This metadata is used by almost all subsequent shader stages.

With respect to the path data streams, the following information is recorded for each (sub)path:

Listing 3 PathInfo Struct

```
struct PathInfo {
    flags: u32,
    start_offset: u32,
    end_offset: u32,
}
```

The `SubpathInfo` struct additionally contains a `parent_path_ix` field. The `start_offset` and `end_offset` fields refer to the index at which the (sub)path's vertices begin and end in the path data stream.

The `flags` field is a bitfield that encodes various properties of the (sub)path, and can have the following flags set:

- `PATH_FLAG_FILL`: Indicates if the path is a fill.
- `PATH_FLAG_CLOSED_STROKE`: Indicates if the path is a closed stroke.
- `PATH_FLAG_OPEN_STROKE`: Indicates if the path is an open stroke.
- `PATH_FLAG_HAS_TRANSFORM`: Indicates if the path is preceded by a transform tag.
- `PATH_FLAG_START_STYLE`: Indicates if the path is preceded by a style tag.

Additionally, subpaths may have the following flags set:

- `SUBPATH_FLAG_IS_FIRST_IN_PATH`: Indicates if the subpath is the first subpath of a path.
- `SUBPATH_FLAG_IS_LAST_IN_PATH`: Indicates if the subpath is the last subpath of a path.

For the path tag stream, only start- and end offsets of both paths and subpaths are recorded in two `TagRangeInfo` arrays.

The buffers produced by this stage are only freed when the geometry changes, not every frame. A reference to them is inserted into a host-side buffer cache easily accessible by all subsequent stages.

3.5. Stage 1: Spherical Rotation

We implement general spherical rotation as a separate step to projection. This allows us to support arbitrary projection centers and aspects for all projections without requiring corresponding parameters in the projection functions.

This is an "embarrassingly parallel" problem, as every vertex can be rotated independently. We launch a compute shader with one thread per vertex.

The shader provides both a forward and an inverse spherical rotation. The rotation is defined by three Euler angles, received as uniform parameters: a yaw (λ_r), a pitch (ϕ_r), and a roll (γ). The implementation includes an identity check to bypass all calculations if the rotation parameters are zero. It also normalizes all output longitude values to the range $[-\pi, \pi]$.

The Two Modes of Rotation A critical aspect of this implementation, shared by `d3-geo`, is how it handles polar coordinates. A naive 3D rotation (converting to a Cartesian position vector $\vec{p}$, applying a rotation matrix, and converting back) would collapse all longitudes λ at a pole ($\phi = \pm\pi/2$) to the same vector $(0, 0, \pm 1)$. While geometrically correct (longitude is undefined at a pole), this irretrievably loses information essential for many projections that must "fan out" points at the pole based on their original longitude (e.g., Equirectangular and Mercator).

Thus, the implementation follows two distinct modes:

1. **Mode 1: Pure Yaw Rotation** (where $\phi_r = 0$ and $\gamma = 0$)

   If only a yaw is applied, the 3D Cartesian conversion is skipped entirely. The rotation is implemented as a simple 2D shift in spherical space:

   $$\lambda' = \lambda + \lambda_r$$
   $$\phi' = \phi$$

2. **Mode 2: Oblique Rotation**

   If any pitch or roll is involved, the rotation composes them with the yaw from step 1. This step does use a 3D Cartesian conversion, but it uses the already-shifted longitude λ' to form the vector $\vec{p}$:

   $$\vec{p} = \begin{pmatrix} \cos\phi \cos\lambda' \\ \cos\phi \sin\lambda' \\ \sin\phi \end{pmatrix}$$

   In this path, if ϕ is a pole, the conversion does lose the longitude information from step 1.

   The 3D rotation is applied to $\vec{p}$ to produce $\vec{p}''$, and the resulting coordinates are converted back to spherical coordinates $(\lambda_{\text{new}}, \phi_{\text{new}})$:

   $$\lambda_{\text{new}} = \operatorname{atan2}(y'', x'')$$
   $$\phi_{\text{new}} = \arcsin z''$$

Forward and Inverse The *forward rotation* process (applied to input degrees, outputting radians) follows the dual-path logic just described.

The *inverse rotation* (applied to input radians, outputting degrees) performs the exact opposite operation, also respecting the two modes:

1. **Mode 1 (Pure Yaw):** A simple inverse shift is applied:

   $$\lambda_{\text{new}} = \lambda' - \lambda_r$$

2. **Mode 2 (Oblique):** The full inverse-composed operation is applied.

   a) Inverse Pitch & Roll: The inverse of the oblique rotation is applied (3D conversion, inverse roll, inverse pitch, convert back to spherical).

   b) Inverse Yaw: The inverse longitude shift is applied to the result.

3.6. Stage 2: Boundary Clipping

While the rotation step itself is straightforward, it introduces challenges when applied to vector data. Even for world maps where no segments are clipped, we at least need to cut the geometry along the boundary of the projection domain. Naively retaining a `LINETO` segment between projected endpoints produces visual artifacts (see Figure 3.2). Circle clipping describes a visible domain and an invisible domain, and segments crossing from one to the other need to be cut similarly, with the portion of the segment outside the visible domain discarded. Closed paths (polygons) also need to be rejoined along the boundary seam.

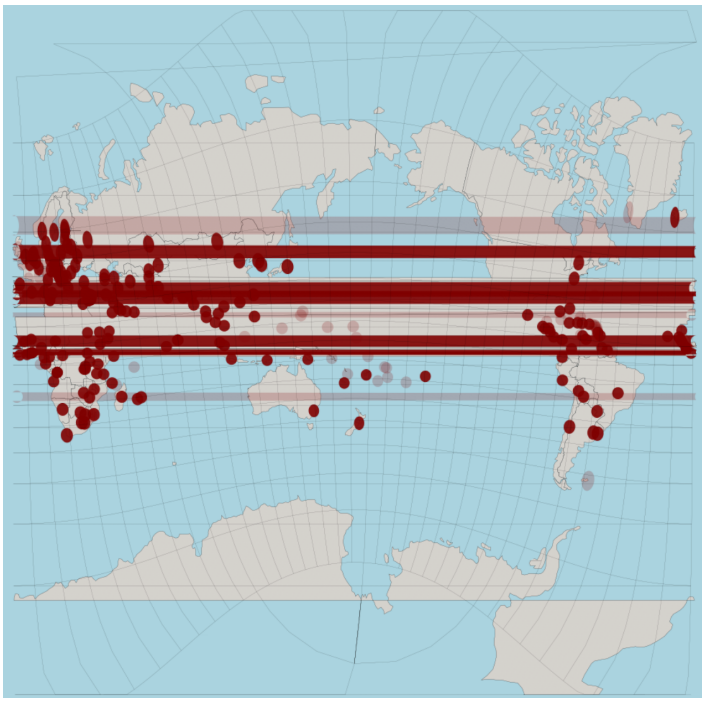


Figure 3.2.: A rotated Mercator map with major cities plotted as points. The points' transparency varies in proportion to the cities population. Clipping is disabled and lines as well as polygons crossing the antimeridian are drawn incorrectly.

We adapt the `d3-geo` clipping pipeline (based on Greiner–Hormann), optimizing it for GPU execution. The key stages are outlined below.

3.6.1. Intersection Calculation

The first stage of the clipping pipeline computes the intersection points between the input geometry and the chosen clipping boundary. This is relatively easy to parallelize, as each segment can be processed independently. Nonetheless, it is not quite as straightforward as it seems at first glance, as spherical geometry may produce a variable number of intersections per segment and thus prevents a static mapping between input threads and output buffer indices. Unlike planar clipping—which yields at most one intersection per segment—clipping a great circle arc against a small circle can yield zero, one, or two intersections. This unpredictable

output distribution necessitates the dynamic memory management and parallel compaction strategies detailed below.

Because the mathematical intersection logic depends on the boundary type (e.g., antimeridian or small circle), the shader is compiled dynamically. The host application links a specific `calculateIntersections` module at runtime, allowing various clipping strategies to share the same core shader logic, memory layout, and parallelization structures.

Pipeline Execution and Memory Management

The shader evaluates the geometry by processing the encoded `path_tags` and `path_data` streams in parallel, assigning one thread per path tag. To correctly interpret the topology of the segments, the shader relies on the `SubpathInfo` array generated during the preceding setup stage. This metadata is essential for filtering out rendering artifacts: Vello appends artificial geometric segments to strokes to ensure proper stroke expansion and coverage calculation. For instance, closed strokes receive an extra `LINETO` segment to render the final *join*, while open strokes end with a `QUADTO` segment to render the initial *cap* (see Section 3.6.5 for a more in-depth discussion). Both carry a `SUBPATH_END` flag. While geometrically present, these segments are artifacts of the rendering backend and are not part of the logical path. By reading the `SubpathInfo` flags, the shader identifies whether a path is stroked and explicitly excludes these auxiliary segments from boundary testing.

For valid segments, the shader invokes the dynamically linked intersection math. This module determines if the segment crosses the boundary, returning up to two intersection points along with topological flags. These flags denote whether the line is entering (`PATH_ENTERS_DOMAIN`) or exiting (`PATH_EXITS_DOMAIN`) the target geometric region. Depending on the active clipping strategy, this "domain" physically represents either the visible extent of the projection (during small-circle clipping) or the transition between the eastern and western hemispheres (during antimeridian cutting). Furthermore, to support downstream graph construction, the shader calculates the segment's incident angle. By evaluating the segment's trajectory $(\Delta\lambda, \Delta\phi)$, normalizing the resulting angle to a $[0, 1]$ range, and quantizing it into a 24-bit integer, this geometric data is bit-packed directly into the upper bits of the topological flag. As detailed in Section 3.6.2, this embedded angle acts as a critical tie-breaker for untangling multi-segment collisions.

Writing these resulting intersections to global memory presents a significant synchronization challenge. Each thread may yield zero, one, or two intersections. A naive approach—requiring each thread to perform an atomic addition to a global bump allocator—would result in severe memory contention and serialize execution across the GPU.

To mitigate this bottleneck, we employ an exclusive workgroup-level Blelloch scan [Ble89] to manage memory allocations (3.3). We utilize this primitive to coalesce many small, individual thread allocation requests into a single bulk allocation, circumventing atomic contention on global memory.

Within each workgroup, threads first store their local intersection counts into shared memory. The Blelloch scan processes these counts in two phases (an up-sweep and a down-sweep) to compute both a workgroup-wide total and the prefix sum for each thread. Consequently, only

a single thread (thread 0) is required to perform one `atomicAdd` to the global bump allocator (`bump.intersections_open` or `bump.intersections_closed`), reserving a contiguous block of memory for the entire workgroup at once. Each thread then uses its calculated prefix sum as a deterministic local offset to safely write its intersections into the globally allocated block.

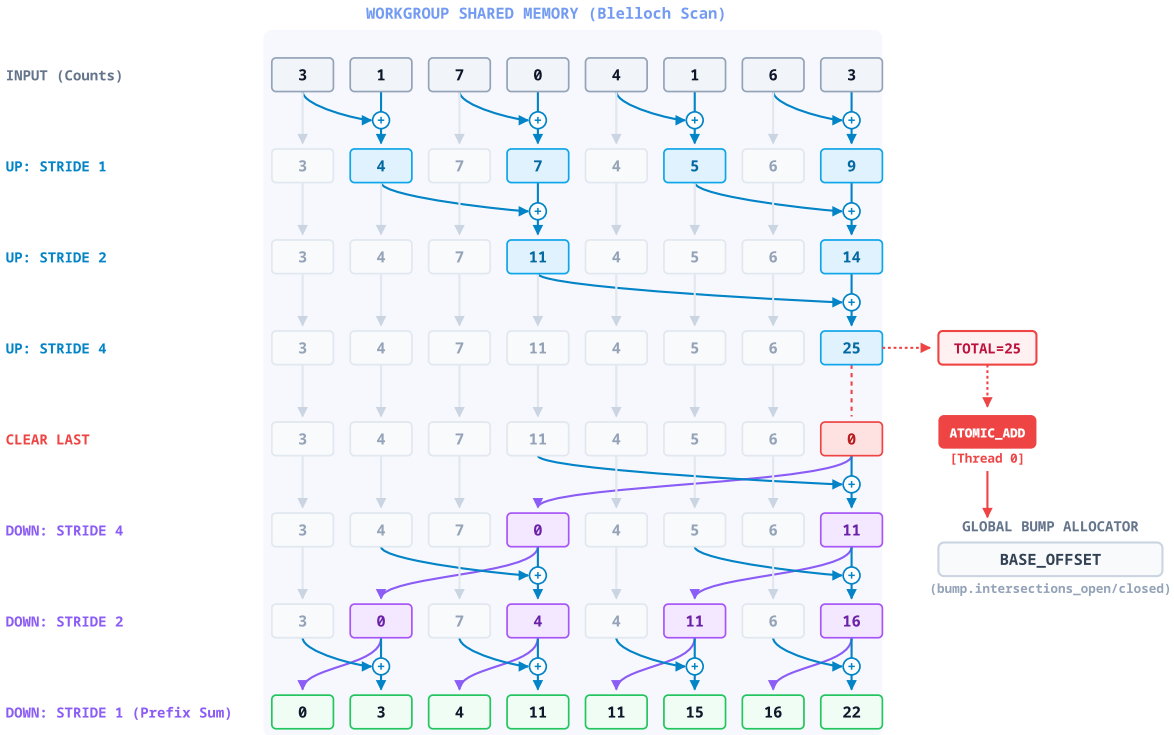


Figure 3.3.: Blelloch scan for memory allocation in intersection stage.

During this output phase, intersections of open and closed paths are routed to separate global buffers. For open paths, the coordinates and the combined entry/exit and angle flags are stored together within `IntersectionOpen` structures. For closed paths, the intersections are written to an `IntersectionClosed` array, while their corresponding flags are written to a separate, parallel `intersections_closed_flags` array (see Listing 4). This explicit separation is not strictly required by the downstream Greiner–Hormann algorithm, but has been implemented for semantic clarity and VRAM optimization, as the flags for closed paths can be discarded much earlier in the pipeline. The buffer also need to be handled separately by the sorting stage, as outlined in Section 3.6.2.

Listing 4 Intersection Structs

```
struct IntersectionOpen {
    point: vec2<f32>,
    pathseg_offset: u32,
    path_ix: u32,
    subpath_ix: u32,
    flags: u32,
}
```

```
struct IntersectionClosed {
    point: vec2<f32>,
    pathseg_offset: u32,
    path_ix: u32,
    subpath_ix: u32,
}
```

Because the initial buffer allocation relies on a heuristic, the shader must actively guard

against buffer overflows. It validates its target global index against a `clipping_config` uniform. If a thread detects an out-of-bounds write, it safely discards the geometric data and instead flags the global `bump.failed` field with a bitmask identifying the current stage. This signals the host to reallocate buffers based on the true calculated requirements and re-run the pipeline following its single end-of-pipeline read-back.

In addition to recording coordinates, the shader generates metadata essential for downstream graph creation and traversal. Each thread tracks local flags indicating whether its assigned segment intersects the boundary (`FLAG_SEGMENT_INTERSECTS`) and whether any part of the segment's endpoints is visible (`FLAG_SEGMENT_VISIBLE`). Similar to the intersection output, updating the global clipping information arrays directly per thread is inefficient. Instead, threads aggregate these flags using a segmented parallel reduction in shared memory (see Figure 3.4).

Over $\log_2(N)$ iterations, the stride between active threads doubles at each step. An active thread reads the data of a neighboring thread located at the current stride distance. If both threads belong to the same grouping—in this case, the same subpath—the active thread merges the neighbor's flag into its own via a bitwise OR, while the neighbor's flag is zeroed out. This hierarchical aggregation continues until all flags for a given subpath are completely accumulated into a single thread. Once the local reduction is complete, only the final aggregated flags are dispatched as a single atomic write to the global `subpath_clipping_info` and `path_clipping_info` buffers. This aggressively culls atomic traffic to global memory.

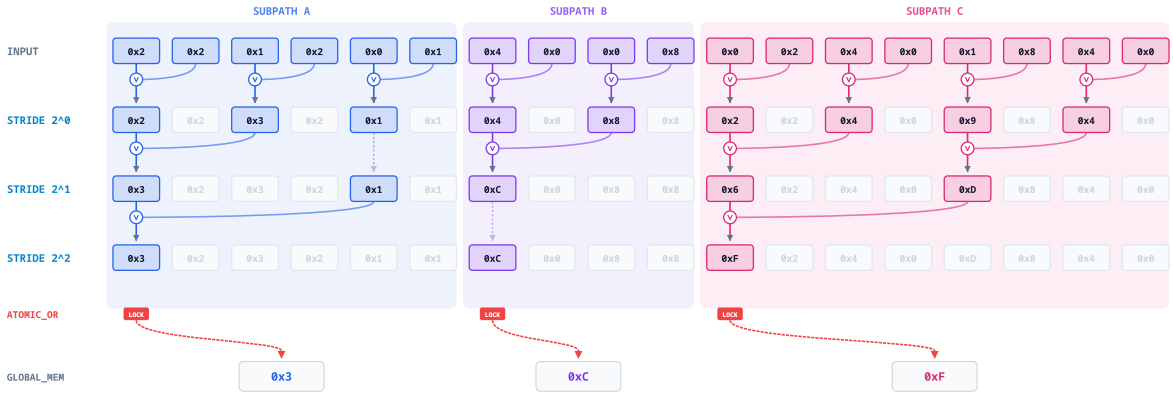


Figure 3.4.: Segmented parallel reduction of bitflags.

Antimeridian Intersection

The core task of antimeridian clipping is identifying whether a segment crosses the boundary at $\lambda = \pm\pi$ and, if so, calculating the exact latitude ϕ_I of the intersection. The shader first performs a fast rejection test: a segment crosses the antimeridian if the longitudes of its endpoints P_1 and P_2 have opposing signs and their absolute difference is greater than or equal to π, or if the segment crosses the pole (where $\Delta\lambda \approx \pi$). If neither condition is met, the thread exits early.

For valid crossings, finding ϕ_I can be approached in two distinct ways. A direct translation of the `d3-geo` algorithm uses the analytical spherical trigonometric formulation (Equation 2.9).

While mathematically elegant and computationally cheap—as it avoids converting coordinates out of the spherical domain—implementing this on a GPU reveals significant numerical instability. Because the analytical formula is highly sensitive to degeneracies, the shader must actively nudge endpoints away from exactly $\pm\pi$ using an ϵ offset, and implement explicit fallbacks for nearly identical longitudes.

GPU-based Greiner–Hormann implementations are much more reliant on stable intersection calculations to generate valid graph structures than sequential ones due to floating-point precision issues; a fact that will be explained in more detail in Section 3.6.2. For this reason, we discarded a verbatim `d3-geo` spherical implementation in favor of a 3D Cartesian approach. While converting between spherical and Cartesian coordinates introduces a performance overhead due to additional trigonometric instructions, it provides superior numerical precision and eliminates the need for ϵ-nudging.

Building upon the formulations established in the Background chapter (see Section 2.3.3), the Cartesian shader determines ϕ_I by finding the intersection of two planes. To compute the normal of the great circle plane containing the segment ($\vec{n} = \vec{p}_1 \times \vec{p}_2$), naively converting endpoints to 3D Cartesian coordinates and evaluating the cross product introduces catastrophic cancellation for short segments. Because the coordinates are nearly identical, the standard cross-product terms (e.g., $y_1 z_2 - z_1 y_2$) result in the subtraction of nearly equal large floating-point numbers, destroying critical mantissa bits.

To resolve this, the shader computes $\vec{n}$ directly from the spherical coordinates by algebraically isolating their differences $(\Delta\lambda, \Delta\phi)$. By leveraging the versine function, $\mathrm{vers}(\theta) = 1 - \cos(\theta) = 2\sin^2(\theta/2)$, the components of the normal vector $\vec{n} = (n_x, n_y, n_z)$ are evaluated using only products and sums of sine terms, which remain numerically stable as the segment length approaches zero:

$$\begin{aligned} n_x &= \cos\phi_1 \cos\phi_2 (\cos\lambda_1 \tan\phi_2 - \cos\lambda_2 \tan\phi_1) \\ n_y &= \cos\phi_1 \cos\phi_2 (\sin\lambda_1 \tan\phi_2 - \sin\lambda_2 \tan\phi_1) \\ n_z &= \cos\phi_1 \cos\phi_2 \sin(\lambda_2 - \lambda_1) \end{aligned} \tag{3.1}$$

While n_z factors into a pure product, the x and y components are reformulated to eliminate catastrophic cancellation. By substituting $\cos\lambda_2$ with $\cos\lambda_1 - (\cos\lambda_1 - \cos\lambda_2)$ and utilizing $\mathrm{vers}(\Delta\lambda)$, the shader evaluates these deltas within the highly precise near-zero domain of the sine function. This cancellation-free cross product (`sphericalCross`) allows the pipeline to leverage the geometric stability of 3D Cartesian intersection math without the traditional loss of precision for short line segments.

Before proceeding, the shader checks whether $\vec{n}$ has near-zero magnitude (i.e. $\vec{n} \cdot \vec{n} < \epsilon$), which indicates numerically coincident endpoints. In this degenerate case no meaningful great circle plane can be determined, and ϕ_I is approximated as the arithmetic mean of the two endpoint latitudes: $\phi_I = \frac{1}{2}(\phi_1 + \phi_2)$.

Because the antimeridian lies strictly on the $y = 0$ plane, the intersection between the great circle plane and the antimeridian plane is a line representing a 3D direction vector. This

vector is efficiently found by crossing $\vec{n}$ with the y-plane normal $(0, 1, 0)$, which simplifies to:

$$\vec{v}_{\text{dir}} = \begin{pmatrix} -n_z \\ 0 \\ n_x \end{pmatrix} \tag{3.2}$$

This intersection line pierces the sphere at two antipodal points: one on the Prime Meridian ($x > 0$) and one on the Antimeridian ($x < 0$). The shader selects the vector pointing towards the negative x-hemisphere, normalizes it, and extracts the latitude via $\phi_I = \arcsin(z)$.

Once ϕ_I is established, the shader constructs the two intersection records. For a standard antimeridian crossing, the longitude of each record is snapped to exactly $\pm\pi$ based on the sign of the corresponding endpoint's longitude, rather than using the raw coordinate value. This guarantees that boundary vertices are placed precisely on the antimeridian irrespective of any floating-point deviation in the input geometry. The point on the exiting side of the boundary (λ_{exit}) is flagged as `PATH_EXITS_DOMAIN`, while the corresponding point on the entering side (λ_{entry}) is flagged as `PATH_ENTERS_DOMAIN`.

If the segment instead represents a pole crossing, the Cartesian intersection calculation is bypassed. Sequential algorithms can handle this case by dynamically emitting intermediate vertices to resolve the polar singularity for map projections that unroll the poles into a line. Specifically, `d3-geo` emits four vertices: A pair routing from the intersections analytically derived starting longitude to the antimeridian boundary ($\lambda = \pm\pi$) on the exiting hemisphere, and a second pair resuming on the entering hemisphere to route from the boundary to the ending longitude, all fixed at the extreme polar latitude ($\phi = \pm\pi/2$). Because a compute shader environment is poorly suited for dynamic data emission, our implementation simply generates the two intersections with analytically derived longitudes retained. The latitude, however, is explicitly set to $\pm\pi/2$, with the sign determined by the arithmetic mean of the endpoint latitudes: $\phi_I = \pi/2$ if $\frac{1}{2}(\phi_1 + \phi_2) > 0$, and $-\pi/2$ otherwise. This exact assignment serves as a sentinel that the subsequent sorting phase intercepts to gracefully handle these intersections which do not lie on the 1D sorting domain. The mechanism by which this is achieved is detailed in Section 3.6.2.

Circle Intersection

While antimeridian clipping focuses on a single fixed longitude, clipping against an arbitrary small circle requires solving a more general plane intersection problem. The first step in the shader evaluates the visibility of the segment's endpoints. A point $P = (\lambda, \phi)$ is deemed strictly visible if $\cos(\lambda)\cos(\phi) > \cos(r)$. By evaluating this for both endpoints, the shader categorizes the segment into one of two potential intersection scenarios: a boundary crossing or a secant traversal.

Before committing to expensive Cartesian intersection math for the secant case (where both endpoints share the same visibility state), the shader aggressively culls non-intersecting segments using the adapted Cohen–Sutherland trivial rejection logic described in the Background chapter (see 2.4.2). The `calculateOutcode` function determines a 4-bit region code for each endpoint. If the bitwise AND of both codes is non-zero, the entire segment lies strictly outside the clipping domain in one specific direction and is immediately discarded.

To elegantly support large circles ($r > 90°$), the shader simply shifts the test longitude by $\pm\pi$ and evaluates the region code against the complementary radius $\pi - r$. The secant path is additionally gated by a hemisphere guard: if $|\cos(r)| \leq \epsilon$, the clipping boundary is an exact hemisphere and the secant path is skipped entirely, as a geodesic arc that does not cross the hemisphere boundary cannot pierce it twice.

For segments that cannot be trivially rejected, the `intersectSegmentCircle` function lifts the spherical coordinates into 3D Cartesian space ($\mathbb{R}^3$) to calculate the precise intersection points, implementing the linear system defined in Section 2.3.3.

The small circle's normal is fixed at $\vec{n}_1 = (1, 0, 0)$, while the great circle's normal $\vec{n}_2$ is derived via the spherical cross product. Because $\vec{n}_1$ is strictly axis-aligned, dependent operations are analytically simplified: the intersection line's direction vector reduces to $\vec{u} = (0, -n_{2z}, n_{2y})$ and the system determinant to $n_{2y}^2 + n_{2z}^2$. Bypassing standard vector built-ins eliminates trivial zero-multiplications and enables the use of hardware Fused Multiply-Add (FMA) instructions to preserve the determinant's precision. A determinant of zero indicates that the great circle is tangent to or coincident with the small circle. In the secant case, no intersections are emitted. In the crossing case, however, the function falls back to emitting the visible endpoint (always passed as the first argument by the caller, as described below) as a degenerate intersection record, preserving topological continuity in the downstream graph.

For the non-degenerate case, the shader calculates the coefficients for the linear combination and solves the quadratic equation for t to find the two potential intersection points along the direction vector $\vec{u}$. Here, the discriminant $\Delta = w^2 - \|\vec{u}\|^2 (\|\vec{a}\|^2 - 1)$ approaches zero for nearly tangent segments, making it highly susceptible to catastrophic cancellation. To prevent floating-point noise from producing a mathematically invalid negative discriminant, the shader again employs FMA. Fusing the multiplication and subtraction guarantees a single rounding step, strictly protecting the discriminant's precision at the tangent limit.

Endpoint argument order is a key implementation detail, as it controls which root of Equation 2.5 resolves to the intersection point on the arc. In the crossing case, the caller always passes the visible endpoint first, swapping P_1 and P_2 when the second endpoint is the visible side (i.e., v_2 is true). This ensures that the $(-w - \sqrt{\Delta})$ root consistently resolves to the point lying on the arc, rather than its antipodal counterpart. In the secant case the arguments are always reversed, which flips the direction of $\vec{u}$ and thereby ensures that the first root corresponds to `PATH_ENTERS_DOMAIN` and the second to `PATH_EXITS_DOMAIN`, consistent with the hard-coded flag assignment in the caller. The shader routes the results based on the initial visibility categorization:

- **Crossing Case ($v_1 \neq v_2$):** The segment clearly crosses the boundary once. The shader computes the first root of the intersection equation and flags the resulting point as `PATH_ENTERS_DOMAIN` or `PATH_EXITS_DOMAIN` based on the direction of the boundary crossing (exit if v_1 is visible).
- **Secant Case ($v_1 = v_2$):** This occurs when both endpoints share the same visibility state but the segment potentially pierces the boundary twice. For small circles, this is triggered when both points are invisible; for large circles, it occurs when both are visible, as the geodesic may cut across the invisible "hole" of the projection domain. The shader computes the first root Q_1 and validates it against the arc using `isPointOnSegment`. If

Q_1 is rejected, no intersections are emitted. If it lies on the arc, the second root Q_2 is computed and both points are emitted with their respective entry and exit flags. The second root is not independently validated: by the geometry of the secant configuration, if one intersection lies within the arc, the other is guaranteed to as well.

Since the underlying plane equations compute intersections for the full great circle, the shader must verify that a calculated point Q lies on the finite geodesic arc $[P_1, P_2]$. This is implemented in the `isPointOnSegment` utility, which adapts interval testing to the sphere's periodic domain. For segments crossing the antimeridian ($\Delta\lambda > \pi$), the containment logic is inverted: a point is valid if its longitude lies outside the numeric range of the endpoints. For meridian-aligned segments that do not cross a pole, the check degrades to a simple latitudinal range test. Polar segments require the most nuanced treatment: the threshold latitude against which ϕ_Q is compared is selected based on which of the two endpoint meridians Q's longitude is closest to, and the sense of the inequality is then flipped depending on whether the arc resides in the northern or southern hemisphere (determined by the sign of $\phi_1 + \phi_2$). This prevents "ghost" intersections produced by the underlying plane equations from being incorrectly accepted as belonging to a short polar arc.

Finally, to prevent topological ambiguities during the downstream graph traversal phase, the shader checks if any calculated intersection point is numerically coincident with the original segment endpoints (within a defined ϵ). If so, it appends an `INTERSECTION_ON_ENDPOINT` flag, signaling the rejoining stage to handle the degenerate vertex gracefully.

3.6.2. Sorting Intersections

Because the intersection calculation stage runs in parallel, the resulting intersection records are written to the global output buffers in a non-deterministic order. To create an analogue to the clip- and subject lists of a sequential Greiner–Hormann implementation, we have to sort the intersections not just along the boundary, but also by the original order of the source geometry in memory.

To achieve this, we adapt a GPU parallel merge sort taken from the Google Forma vector renderer repository for our purposes. While state-of-the-art radix sorts often provide maximum throughput for massive, uniform datasets, a parallel merge sort was chosen for its relative ease of implementation and its superior performance profile for smaller input sizes. The performance trade-off between the two algorithms largely hinges on GPU dispatch overhead and memory hierarchy constraints. A radix sort requires a constant number of compute dispatches determined by the key size rather than the element count (e.g., four dispatches to sort a 32-bit key in 8-bit passes).

In contrast, a parallel merge sort requires $\mathcal{O}(\log N)$ dispatches. Once the input size exceeds the capacity of a single workgroup's shared memory, the algorithm must write intermediate sorted blocks back to global VRAM, synchronizing across the entire GPU before the host can dispatch the next merge pass to double the run lengths. For large arrays, these repeated global memory round-trips and growing dispatch counts become a significant bottleneck. However, typical geographical geometry often yields a total intersection count small enough to be processed within very few rounds or even a single dispatch if the input size is small enough

to processed by one workgroup. In these common scenarios, the merge sort comfortably outperforms the constant-dispatch overhead of a radix sort.

The sorting pipeline executes in four compute passes:

1. **Key Construction:** Threads map geometric intersection data into sortable numerical keys.
2. **In-Block Sort:** Threads perform a local odd-even transposition sort on small per-thread arrays, followed by a hierarchical shared-memory merge to produce sorted workgroup blocks.
3. **Merge Offsets:** A parallel binary search computes precise global indices (offsets) to determine where each sorted block belongs in the final array.
4. **Global Merge:** Threads use the calculated offsets to merge the sorted blocks into the final, strictly ordered output buffer.

We run this pipeline three times in total. The first two runs restore the original segment order for open and closed path intersections, respectively. The final run sorts the closed path intersections along the clipping boundary.

While the current implementation shares intermediate buffers between these runs to conserve VRAM, it forces the API to inject implicit synchronization barriers to prevent data races, strictly serializing the dispatches. Allocating dedicated buffers would remove these dependencies and allow the driver to submit the dispatches for concurrent and possibly parallel processing at the cost of increased memory usage. A potentially more robust optimization would be to fuse the first two runs by writing the keys for both open and closed paths into a single, shared key buffer. By constructing a composite key—using either two u32 values or a topology sentinel encoded in the most significant bit—the sorting pass will partition the open and closed paths in-place. Consequently, this approach requires an additional uniform buffer to provide subsequent pipeline stages with the index offset where the open and closed keys diverge.

Restoring Segment Order

For both open and closed paths we must sort intersections by the order of their generating segments in memory. Because Vello encodes a monotonically increasing `pathseg_offset` for each segment in the corresponding `TagMonoid`, the shader simply uses this 32-bit unsigned integer as the sorting key.

Sorting Along the Clipping Boundary

Sorting intersections of closed paths along the clipping boundary for later graph construction is substantially more complex. Because WGSL lacks native 96-bit or 128-bit integer types, the shader constructs a custom 3-part composite key, represented as a `_u96` struct containing `hi`, `mid`, and `lo` 32-bit integers. Comparisons are performed lexicographically from highest to lowest significance.

Primary Key (`hi`): Path Index The most significant 32 bits store the `path_ix`. This ensures that intersections belonging to different independent polygons are cleanly separated into contiguous blocks. We do not want to separate by `subpath_ix` because interior rings (holes) may need to be joined to the exterior ring along the boundary seam, and thus need to be sorted together.

Secondary Key (`mid`): Boundary Seam Traversal The middle 32 bits establish the traversal order of vertices along the 1D clipping boundary (e.g., the antimeridian). Unrolling a 2D spherical boundary into a continuous 1D domain requires mapping the 2D spherical coordinates (λ, ϕ) to a 1D scalar $s \in [-\pi, \pi]$, as described by Equation 2.10. Crucially, this function traverses the two opposing sides of the antimeridian seam ($\lambda < 0$ vs. $\lambda \geq 0$) in opposite directions, ensuring that a polygon crossing the boundary stitches together in the correct winding order.

Tertiary Key (`lo`): Incident Angle and Topological Flags When intersections collide in the secondary key—which may occur due to collinear input geometry but also due to microscopic arithmetic noise—the sorting algorithm resolves the ambiguity using the tertiary key. To guarantee a deterministic and topologically valid graph order for coincident vertices, this final 32-bit integer utilizes two distinct pieces of information packed during the intersection stage.

First, the upper 24 bits contain the quantized incident angle of the intersecting segment. As introduced in Section 3.6.1, this approach angle is derived from the segment's trajectory, normalized to $[0, 1]$, and scaled to fill the 24-bit space. Leveraging this pre-calculated data allows the sort to geometrically untangle complex, multi-segment collisions at a single point (such as a shared boundary knot).

Second, the lowest bits encode the topological `PATH_EXITS_DOMAIN` and `PATH_ENTERS_DOMAIN` flags. These are structured such that `PATH_EXITS_DOMAIN` evaluates to a numerically smaller value. If intersections are perfectly coincident and arrive from the exact same quantized angle, this acts as the absolute final tie-breaker. Because we use the left-hand surveyor's rule for polygon winding, we always want to join exiting intersections to entering intersections along the boundary seam, ensuring valid graph traversal during the final linking phase.

3.6.3. Spherical Point-in-Polygon Test

As established, determining the initial containment state of the reference point M relies on accumulating the subject polygon's longitudinal winding Φ and spherical excess Σ (Equation 2.12), followed by a parity check against the South Pole (Equation 2.14). Providing this reliable boolean state to the Greiner–Hormann traversal is critical for resolving the inside/outside status of the entire resulting graph.

While the mathematical core of the compute shader closely mirrors this sequential `d3-geo` formulation, executing a global accumulation within a massively parallel GPU environment requires significant structural deviations. Rather than iterating through a polygon's segments

linearly, the shader must evaluate segments concurrently. Translating this sequential point-in-polygon (PIP) test to a parallel architecture introduces distinct challenges regarding memory bandwidth bottlenecks, floating-point catastrophic cancellation, and cross-workgroup synchronization.

Cancellation-Free Edge Evaluation

Each thread computes the longitudinal winding and spherical excess contributions for a localized block of line segments (`PIP_WORDS_PER_THREAD`), storing the high-precision intermediate outputs in an `EdgeResult` structure (Listing 5). Much like the intersection stage, evaluating the geometric relationship between the segment and the reference point invites severe precision loss if handled naively.

Listing 5 Local thread result structure preserving 32-bit floating-point precision during sequential edge evaluation.

```
struct EdgeResult {
    sum: f32,
    angle: f32,
    winding: i32,
};
```

To determine the segment's orientation relative to the reference point, the shader computes the normal of the segment's great circle. Rather than lifting the endpoints to Cartesian coordinates and subtracting them—which triggers catastrophic cancellation for short segments—we again rely on the `sphericalCross` function described in Section 3.6.1.

Furthermore, the intersection between the segment's great circle and the meridian passing through the reference point M is determined via a cross product. Because the reference meridian's normal always lies exactly on the equatorial plane ($z = 0$), we analytically simplify this cross product, eliminating unnecessary zero-multiplications. Where subtractions remain, such as in the z-component of this simplified cross product and the denominator of the spherical excess equation (a critical input to the atan2 function), the shader again uses FMA instructions.

Kahan Summation vs. Exact Adders

Spherical excess calculations are highly susceptible to floating-point drift, particularly when accumulating small, alternating signed values around a complex polygon. The sequential `d3-geo` pipeline mitigates this by utilizing `d3-array`'s `Adder`. This relies on Shewchuk's adaptive-precision summation algorithm [She97], which meticulously tracks floating-point rounding errors by maintaining a variable-length array of non-overlapping partial sums.

While perfectly suited for a single-threaded CPU environment, porting exact summation directly to a massively parallel compute shader fundamentally conflicts with GPU memory architectures. Achieving functionally identical precision in WGSL would require threads to perform complex "expansion merges"—dynamically combining sorted arrays of floating-point

components rather than executing simple scalar additions. Because WGSL lacks hardware mechanisms to atomically merge variable-length arrays in global memory, a direct port would induce severe thread serialization and register exhaustion.

A viable GPU alternative is substituting Shewchuk's expansion with a fixed-memory precision technique, such as Kahan summation or float-float (emulated double precision), aggregated via a multi-pass device-wide parallel prefix scan. However, implementing a high-precision scan exposes strict limitations within WebGPU's memory model. Advanced, high-throughput single-pass scan architectures rely on either device-wide memory fences or packing state flags and data payloads into a single 32-bit integer atomic to guarantee synchronized visibility across workgroups. Because a Kahan state or float-float pair requires a minimum of 64 bits, the payload must be decoupled from the synchronization flag.

In WGSL, atomics strictly enforce relaxed memory ordering and synchronization barriers are limited to workgroup scope. Without acquire-release semantics to enforce a strict execution order, this decoupling introduces unresolvable cross-workgroup race conditions, as the hardware is permitted to publish the atomic flag before the 64-bit payload physically lands in global memory. Consequently, a mathematically safe high-precision scan in WebGPU necessitates abandoning single-pass designs in favor of a multi-pass, map-reduce architecture.

Furthermore, emulating double-precision within a parallel scan requires merging partial sums using exact addition algorithms, such as Knuth's Two-Sum [Knu97]. These algorithms fundamentally rely on strict IEEE 754 execution order to isolate and capture truncated mantissa bits. Because WebGPU acts as a translation layer, it inherits the aggressive compiler optimizations of its underlying graphic APIs. Notably, backend compilers (such as Apple's Metal Shading Language) frequently apply `fast-math` algebraic transformations by default. Without explicit compiler guards—which introduce additional overhead—to disable associative reordering, underlying drivers can dynamically optimize away the error-recovery algebra entirely, rendering float-float emulation dangerously brittle across different hardware.[3][4]

To balance numerical stability with WebGPU's hardware and compiler realities, the shader instead employs local Kahan summation. By carrying a single running compensation factor (c), Kahan summation effectively doubles the working precision of standard 32-bit floats.

Two-Pass Map-Reduce Architecture

Trying to aggregate these high-precision floating-point partial sums across the entire device is challenging. A single-pass reduction would need to rely on global atomic operations to safely accumulate values from independent workgroups. Because WebGPU currently lacks native atomic support for floating-point numbers, this would have to be achieved by scaling and quantizing the floats into 32-bit integers (`i32`) to utilize standard integer atomics. While this circumvents the hardware limitation, fixed-point quantization introduces rigid fractional truncation and overflow risks, effectively destroying the dynamic precision benefits of the Kahan algorithm.

[3] `https://github.com/gpuweb/gpuweb/issues/2076` (last accessed 01.03.2026)
[4] `https://github.com/gpuweb/gpuweb/pull/2080` (last accessed 01.03.2026)

To safely aggregate the partial sums while strictly preserving the floating-point state through the entire reduction tree, the pipeline abandons a single-pass atomic approach in favor of a two-pass map-reduce architecture, separated by an API-level pipeline barrier.

In the first pass, threads complete their local processing loops and enter a workgroup-level reduction tree in shared memory. The workgroup aggregates this data using a segmented parallel scan, intelligently tracking whether a path's data is fully contained within the workgroup or spans across its boundaries. If a path begins and ends entirely within a single workgroup, its global topological parity is fully resolved immediately, bypassing global memory entirely. If a path crosses a workgroup boundary, the active thread writes the unresolved `PartialResult`—containing the exact Kahan sum, compensation factor, and winding—into a pre-allocated global buffer.

The second pass executes a device-wide aggregation. This pass launches with exactly one thread per unique path. Each thread calculates the boundary limits of its assigned path across the workgroups, iterates over the `global_partials` buffer, and performs a final sequential Kahan reduction. This effectively collapses the complex synchronization problem into an embarrassingly parallel gather operation, sacrificing some theoretical single-pass compute throughput to guarantee global numerical stability.

Dynamic Error Bounds and Deadbanding

In spherical geometry, the accumulated spherical excess (Σ) corresponds directly to the polygon's surface area. To determine if a polygon's winding order signifies an inverted topology, the test evaluates whether this accumulated area is negative. In edge cases where the true area is analytically zero, microscopic numerical noise could push the final value slightly below zero, inverting the global parity and causing the test to fail.

To guard against this, the sequential `d3-geo` algorithm uses a mathematically-motivated deadband, evaluating $\Sigma < -\epsilon^2$. However, in a massively parallel floating-point reduction, accumulated errors grow predictably with the number of operations. Without access to the Shewchuk algorithm for full precision summation, a static ϵ is insufficient for complex polygons comprising thousands of vertices.

To construct a mathematically robust deadband, the shader explicitly calculates dynamic error bounds by modeling the floating-point drift as a random walk. The expected error scales by $\sqrt{N}$, where N is the total number of segments in the path. This scaling factor is applied to the fundamental machine epsilon ($\sim 1.19 \times 10^{-7}$).

Crucially, the pipeline differentiates the baseline error magnitude based on the arithmetic intensity of the accumulation:

- **Angle Accumulation:** Driven by simple linear additions, this is tightly bound to a 2-ULP (Unit in the Last Place) hardware precision limit.
- **Area (Sum) Accumulation:** Driven by complex trigonometric evaluations, this utilizes a wider 8-ULP bound to absorb standard GPU hardware approximation errors inherent in atan2, sin, and cos instructions.

These dynamic bounds are added to the foundational geometric ϵ values. This way, the pipeline gracefully absorbs both intrinsic geometric drift and heavy parallel computation noise without falsely inverting the polygon's topology in most cases.

Polar Singularities

If the reference point lies on a pole ($\phi = \pm\pi/2$), the Cartesian cross products in the winding calculation degenerate to zero, making the inside/outside determination undefined. The check mirrors the approach used in `d3-geo`: rather than testing ϕ directly, the shader tests whether $\sin(\phi)$ has saturated to exactly ± 1.0 in floating-point. This is the correct condition because a latitude slightly off from $\pm\pi/2$ can still produce $\sin(\phi) = 1.0$ due to rounding, while testing $\phi = \pm\pi/2$ directly would miss those cases.

When saturation is detected, ϕ is nudged away from the equator by `EPSILON`, placing it just beyond the pole:

$$\phi \leftarrow \pm\Big(\frac{\pi}{2} + \epsilon\Big)$$

This pushes the test point to a latitude that no great-circle arc's ϕ_{arc} can equal, so the strict inequality $\phi > \phi_{\text{arc}}$ in the winding accumulation always resolves unambiguously. The resulting coordinate is geometrically invalid as a spherical position, but it is never used to compute arc geometry—it only participates in that scalar comparison, where the out-of-range value is both harmless and intentional.

3.6.4. Graph Construction

With the intersection records computed and ordered along both the source geometry and the clipping boundary, the next stage of the pipeline constructs the topological graph required for the Greiner–Hormann traversal stage. Sequential implementations typically build a doubly-linked list by dynamically inserting intersection nodes into the subject and clip polygons and linking from one into the other in alternating fashion. In contrast, our massively parallel architecture requires constructing this routed graph in-place across pre-allocated, flat arrays (`PolygonGraphNode` and `LineGraphNode`).

This graph construction is divided into four dedicated compute passes: identifying the array bounds of individual paths and subpaths, linking closed polygons along the clipping boundary (the seam), linking those same polygons along the original subject geometry, and extracting the topological segments of open paths. Furthermore, the pipeline strictly separates the routing logic of closed polygons from open paths (lines), as the latter do not enclose an area and thus never route along the boundary seam.

Path Block Boundary Detection

Before links can be established, the pipeline must identify the boundaries of paths and subpaths within the sorted arrays. The `markBlockBounds` kernel achieves this without cross-thread synchronization by having each thread inspect a single intersection and its immediate neighbor.

Threads evaluating closed paths determine block boundaries by detecting divergences between adjacent elements in the sorted arrays. A change in the primary key (`path_ix`) indicates a polygon boundary. Similarly, to identify subpath boundaries, the shader simply fetches and compares the `subpath_ix` of the current intersection with its neighbor's. Threads that detect these transitions write their global array index to the `start_idx` or `end_idx` fields of the corresponding `PathBlockInfo` and `SubpathBlockInfo` metadata arrays. Open paths undergo an identical process, evaluating only the `subpath_ix`. This metadata effectively creates an index map, granting subsequent passes $O(1)$ access to the array limits of any distinct geometric entity. Especially for polygons, this explicit knowledge of boundaries is critical: because a closed subpath forms a continuous geometric ring, its traversal does not strictly terminate at the end of the array block. Instead, the downstream linking logic uses these bounds to wrap from the final intersection in a block back to the first, bridging the remaining geometric data between the subpath's structural start and end points in memory to maintain a closed topological loop.

Polygon Edge Routing

To extract the clipped polygons, the graph must provide the information to generate a contiguous path by alternating between the subject's internal geometry and the boundary seam. For closed paths, this is accomplished in two separate linking dispatches.

The first kernel, `linkAlongSeam`, establishes the routing along the clipping boundary. Each thread uses the `PathBlockInfo` to determine its local index within a contiguous `path_ix` block. The shader derives the topological entry/exit state of each intersection node via the parity check described by Equation 2.14. By comparing the local index parity against the polygon's initial containment state (`contains_reference_point` evaluated during the spherical point-in-polygon test), the thread assigns the `SEAM_ENTERS_SUBJECT` flag. Due to the left-hand winding convention, `ENTRY` nodes must point forward to the next node along the seam, while `EXIT` nodes record incoming edges from the previous seam node. To ensure the seam routing forms a continuous loop, the shader explicitly wraps pointers at the boundaries of the `path_ix` block: the first intersection sets its incoming pointer (`prev_ix`) to the block's `end_idx`, while the last intersection directs its outgoing pointer (`next_ix`) back to the `start_idx`. For correct traversal, only `next_ix` would be required, `prev_ix` is only stored for a later performance optimization, described in more detail in Section 3.6.5.

To avoid confusion, we explicitly draw attention to the semantic distinction between the `PATH_ENTERS_DOMAIN`/`PATH_EXITS_DOMAIN` flags set in the intersection stages of the pipeline and the `SEAM_ENTERS_SUBJECT` flag set during the graph building phase. The former strictly describe the points at which an intersecting subject segment must either be terminated or a new segment must begin when splitting the original subject path, while the latter encodes the coordinates at which we move into or out of the subject polygon when tracing along the seam from a reference point on the boundary. For instance, a node flagged as `PATH_EXITS_DOMAIN` may still be marked as `SEAM_ENTERS_SUBJECT`. Going forward, we will refer to the former as *domain* entry/exit and the latter as *seam* entry/exit to maintain this distinction.

The second pass, `linkAlongPathdata`, establishes the graph edges that follow the original subject geometry by utilizing the arrays sorted by `pathseg_offset`. The shader assigns

the remaining topological pointers based on the node's domain entry or exit state. In the resulting clipped polygon, the boundary must follow the subject geometry strictly from an `EXIT` node to the next `ENTRY` node. Consequently, an `EXIT` node directs its outgoing pointer to the next intersection along the subject path, while an `ENTRY` node receives its incoming pointer from the preceding intersection on that same path.

Line Segment Routing

Unlike closed polygons, open paths do not inherently partition space into inside and outside regions. Consequently, clipping a line against a boundary yields independent, disjoint line segments rather than a continuous loop. The `buildLineGraph` kernel therefore does not link along the seam at all, but solely along the subject paths segments.

Memory and Caching Strategies

To ensure the graph can be traversed efficiently in subsequent compute passes, the construction phase implements two specific memory optimizations.

Pre-Calculated Memory Slices: Both the line and polygon construction passes pre-calculate the memory extraction bounds (`mem_slice_size`) required to copy the original geometry's path data. The graph nodes contain fields that define the byte length of the geometric data they encompass (Listing 6). The primary field, `mem_slice_size_1`, records the forward byte distance from the current intersection's memory offset to the next logical boundary (either the subsequent intersection's offset or the end of the subpath data). The secondary field, `mem_slice_size_2`, strictly captures the slice from the start of the subpath to its first intersection.

While computing these memory slices on-the-fly during the traversal phase would reduce VRAM usage—and might even be faster for a single pass by avoiding global memory reads—our pipeline must traverse the clipped paths twice: once to calculate and allocate the required output buffer size, and again to perform the actual geometric write. Consequently, pre-computing and storing these slices during graph construction is more performant.

Sorting for Spatial Locality: Because intersections are generated in a non-deterministic order by the parallel intersection kernels, building the graph directly from unorganized data would destroy spatial memory locality during the traversal phase, resulting in severe cache thrashing. To mitigate this, the pipeline explicitly sorts the constructed graph nodes by iterating over the already-sorted key arrays, incurring only a vanishingly small sorting overhead during this phase.

For open paths, the `line_graph` is sorted exclusively by the original subject geometry. Because open paths never route along the clipping boundary, traversal strictly follows the path from start to finish, yielding near 100% cache locality. Conversely, the `polygon_graph` for closed paths is sorted along the boundary seam. Traversing a clipped polygon inherently requires ping-ponging between the subject path and the seam. It is mathematically impossible to achieve perfect memory locality for this operation, as sorting by one boundary inherently scatters the memory accesses of the other.

Listing 6 Graph Structs and Bitflags

```
const INTERSECTION_FLAG_FIRST_IN_SUBPATH: u32 = 1u << 0u; // 0x1
const INTERSECTION_FLAG_LAST_IN_SUBPATH: u32 = 1u << 1u; // 0x2
const SEAM_ENTERS_SUBJECT: u32 = 1u << 2u; // 0x4

struct MemorySliceSize {
    mem_slice_size_1: u32,
    mem_slice_size_2: u32,
}

struct PolygonGraphNode {
    next_ix: u32,
    prev_ix: u32,
    intersection_closed_ix: u32,
    mem_slice_size: MemorySliceSize,
    flags: u32,
}

struct LineGraphNode {
    intersection_open_ix: u32,
    mem_slice_size: MemorySliceSize,
    flags: u32,
}
```

3.6.5. Tracing and Reconstruction

The final stage of the pipeline traverses the constructed graph to extract the clipped geometry. For closed paths, this involves alternating between seam and subject edges to reconstruct the resulting polygons. For open paths, the traversal simply follows the subject edges between domain entry and exit nodes to extract the clipped line segments.

As we cannot dynamically insert or remove slices of memory from the original path data and tag arrays, the shader must write the clipped geometry into new output buffers. The pipeline executes in three logical stages:

1. **Memory Reservation:** Multiple passes that calculate the total byte length of the geometry to be reconstructed, writing these values to reservation buffers.
2. **Device-wide Scan:** A GPU parallel prefix sum converts the per-path byte lengths into write offsets.
3. **Writing Geometry:** A number of passes that use the calculated offsets to write the clipped geometry into the final output buffers.

Decoupled Fallback [SLO25]—the chosen scan algorithm—is a state-of-the-art parallel prefix sum that executes in a single pass. It retains the performance characteristics of Merill and Garland's *Decoupled Lookback* [MG16], while not being restricted to CUDA and NVIDIA hardware. The reference implementation makes use of subgroup-level primitives via a

WGSL extension. For maximum compatibility, it was modified to use only workgroup-level synchronization.

As we need to interpolate vertices along the clipping boundary, the shaders are compiled dynamically to import the clip-dependent interpolation functions.

Boundary Interpolation

Both clip domains share the same interpolation interface, which is imported into the tracing shaders at compile time. It exposes four functions:

- `getInterpolationCount(p_start, p_end)`: Returns the number of intermediate points to insert between two consecutive intersection points on the boundary.
- `getInterpolationPoint(i, clip_params, p_start, p_end)`: Returns the i-th intermediate point between two intersections.
- `getFullInterpolationCount(clip_params)`: Returns the number of points required to trace the entire boundary.
- `getFullInterpolationPoint(i, clip_params)`: Returns the i-th point on the full boundary contour.

For antimeridian clipping, a crossing is detected when endpoint longitudes differ by more than ϵ, and three intermediate points are inserted at the corresponding pole latitude: one on each side of the antimeridian and one at the prime meridian. The pole sign follows the crossing direction—eastward crossings route through the North Pole, westward through the South Pole. When the full seam boundary is required, eight waypoints trace the complete rectangular domain $[-\pi, \pi] \times [-\frac{\pi}{2}, \frac{\pi}{2}]$.

For circle clipping, the boundary is a small circle on the sphere. Interpolating directly in spherical coordinates is impractical, so each intersection point is mapped to a scalar phase angle $\theta = \text{atan2}(z, y)$ measured in the plane perpendicular to the clip circle's polar axis. The arc between two consecutive intersection phases is subdivided into steps of 2°, and each intermediate point is reconstructed by placing it on the unit sphere at the corresponding phase and converting back to spherical coordinates (λ, ϕ). A full boundary interpolation thus yields $\lceil 360/2 \rceil = 180$ segments.

Memory Reservation

We reserve space for intersecting and non-intersecting geometry in two separate kernels, each of which handles both open and closed paths.

Reserving Intersected Closed Paths The `reserveIntersected` kernel is responsible for calculating the memory requirements of the clipped geometry resulting from both closed and open paths. For closed paths specifically, the number of subpaths that results from the clipping operation is not a function of the number of intersections and can only be calculated by tracing. Clipping may generate either fewer, more or the same number of polygons. Interior rings may be merged into their exterior subpaths along the seam. Polygons may also be split into multiple new rings which are either all retained (antimeridian cutting) or partially discarded (circle clipping). For this reason, we dispatch one thread per intersection.

Both to avoid redundant work and to guarantee one unique writer per reconstructed polygon ring, the shader must elect a leader thread to trace each closed loop. This cannot be achieved via a per-subpath atomic-lock where each thread attempts to acquire the lock for its assigned intersection's `subpath_ix` precisely because of the reasons described above. Instead, we perform a neighborhood check: a thread only assumes ownership of the loop if neither of its graph neighbors (`next_ix` and `prev_ix`) is assigned to a thread with a higher global index. As the polygon graph is ordered along the seam, this can be interpreted geometrically as the southern- or northernmost intersection of a new loop along the 1D sorting domain. This early-out check is also the reason why we store both `next_ix` and `prev_ix` in the graph nodes, even though only one of them is strictly required for traversal: with only the forward pointer, each non-owner thread would have to trace the graph until it finds a thread with a higher global index.

The leader thread traverses a full loop, accumulating the required memory for path data and tags (see Figure 3.5). Because clipping can split a single polygon into multiple disjoint rings, or merge interior holes into the exterior shell along the seam, the geometry to be reconstructed must be assigned to a valid target subpath. As the thread traces the loop, it tracks the minimum `subpath_ix` encountered. It then atomically pushes its computed memory reservation into a linked list anchored to this minimum `subpath_ix` via the atomic `subpath_heads` buffer. So instead of just reserving "global" memory per new loop in the output buffers, we also reserve relative memory within some subpaths allotted slice. Both the lowest or highest `subpath_ix` could act as this fixed reference; the lowest was chosen arbitrarily.

Instead of building a linked list, one might be tempted to generate the subpath-relative write offsets for new loops using atomic additions directly onto the output counters. However, we must reserve space for both path tags and path data in their respective buffers perfectly in sync. Because GPUs generally lack the ability to perform atomic additions on two separate buffers simultaneously, using independent `atomicAdd` operations is unsafe. For example, if two threads (each handling a different loop but assigning it to the same target subpath) interleave their atomic additions, Thread A's data offset might be paired with Thread B's tag offset, irreversibly corrupting the final memory layout.

To solve this, we construct a per-subpath linked list using atomic operations. Here is how the lock-free binning works: each leader thread represents a unique loop and possesses a unique global invocation ID. After calculating the total `traced_path_data` and `traced_path_tags` for its loop, the thread stores these values in the `reconstruction_reservations` buffer at the index of its own invocation ID.

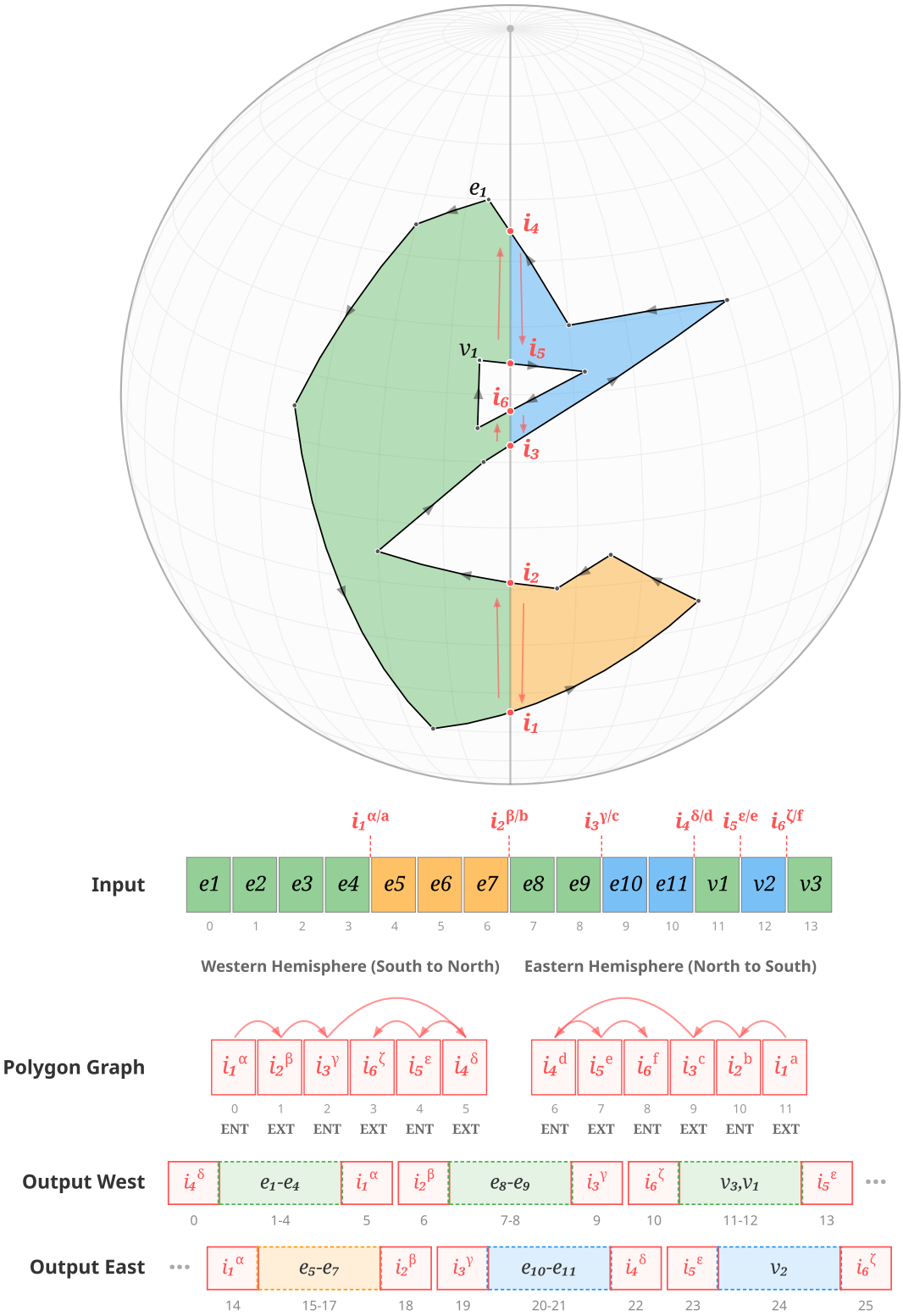


Figure 3.5.: Data flow for reconstructing a polygon cut by the antimeridian. A seam-sorted polygon graph is built from the input array and traversed to isolate closed loops (Output West/East) and compute exact memory reservations.

Next, it performs an `atomicExchange` on the `subpath_heads` array at the `min_subpath_ix`. It swaps in its own ID to become the new head of the list, and retrieves the previous head. It then sets this previous head as its `next_ix` pointer. This creates a last in, first out (LIFO) stack of memory reservations. Later, the separate `resolveIntersectedMemReqs` kernel assigns one thread per subpath to linearly traverse this list, accumulating the required memory and safely writing synchronized, non-overlapping relative offsets for both tags and data.

Each subpath must contain segment data for an implicit `MOVETO` command for which no corresponding tag exists. The thread reserves space for this before entering the traversal loop. During the loop, when routing along the subject geometry (from an `SEAM_EXITS_SUBJECT` node), it sums the pre-computed `mem_slice_size_1` fields and adds one `LINETO` with corresponding data for each node (intersection). When routing along the seam, it adds the intermediate points as needed by calling the interpolation count function with the current seam entry/exit point pair. We break if we reach the initial node, but reserve space for the intersection again to close the shape.

Vello draws line-joins between segments of a stroke (see Figure 3.6). The data required for these can be calculated on the fly, except for the first and last segment of the original subpath. For this reason, the data is pre-calculated during the encoding phase and appended to closed strokes as the `LINETO | SUBPATH_END` tag with corresponding path data. We reserve space for this additional segment for each new stroked loop.

Figure 3.6.: Four types of line joins. Reproduced from [LBS26].

During tracing, if a new loop contains the original subpath's structural start and end points in memory (wrap-around), we must carefully handle a number of corner cases:

- At the last node before the wrap around segments, the thread must reserve the correct amount of memory by also adding its `mem_slice_size_2` field which contains the memory from the start of the subpath to the first intersection.
- We must not to reserve redundant `MOVETO` data for this particular loop. This has already been handled in the graph construction kernel by subtracting the size of the implicit `MOVETO` from the first node's `mem_slice_size_1` field.
- We must not reserve a redundant join segment for stroked loop.

Reserving Intersected Open Paths Threads with indices $i \geq$ num_intersections_closed process open subpaths. Because open paths do not form loops, leader election is considerably simpler than for closed paths: only the thread assigned to the first intersection node of its subpath—identified via the `isFirstInOpenSubpath` predicate—proceeds. All remaining

threads for the same subpath early-out immediately, writing sentinel values (`U32_MAX`) into their `reconstruction_reservations` entries.

The leader traverses the subpath's intersection nodes in linear order, advancing from its base index k to $k+1, k+2, ...$ until a node marked with `isLastInOpenSubpath` is reached. For each node, the `PATH_ENTERS_DOMAIN` flag of the corresponding `IntersectionOpen` record determines whether the crossing is a *domain* entry or exit.

At an entry node, the path crosses the boundary from outside to inside, beginning a new visible segment. Space is reserved for an implicit `MOVETO`, followed by `mem_slice_size_1`, the pre-computed byte size of all subject segments from this entry point to the next exit or natural endpoint. The corresponding tag count is $\lfloor$`mem_slice_size_1/LINETO_DATA_SIZE`$\rfloor$. A correction applies when the entry node is also the last node in the subpath: the visible segment then terminates at the path's natural endpoint, which is encoded as a quadratic Bézier. As `QUADTO_DATA_SIZE` consume four not two `path_data` elements, the integer division overcounts the tags by exactly one, and this excess is subtracted explicitly in that case.

At an exit node, the path leaves the domain. The exit intersection point is emitted as a `LINETO` and its end cap as a `QUADTO | SUBPATH_END`, contributing two additional tags. A special case applies when the first intersection of the subpath is an exit — meaning the path originates inside the clipping domain. Here, `mem_slice_size_2` holds the pre-computed segment data from the structural start of the subpath to this first exit, including the implicit `MOVETO`. Provided `mem_slice_size_2` is non-zero, this block is added to the data reservation and its tag count decremented by one to account for the untagged `MOVETO`. If `mem_slice_size_2` is zero, the path begins exactly on the boundary and no preceding segment data exists.

Open strokes are encoded with an additional delimiting `QUADTO | SUBPATH_END` segment for similar reasons as closed strokes: the data for the closing cap (see Figure 3.7) of the start segment can not be calculated by Vello on the fly.

'butt' cap 'round' cap 'square' cap

Figure 3.7.: Three types of line caps. Reproduced from [LBS26].

Unlike closed paths, each open subpath creates exactly one `reconstruction_reservations` entry — no ring merging or polygon splitting can occur. The destination `subpath_ix` is read from the last traversed intersection node, though any node in the subpath would yield the same value. Even so, the reservation is pushed into the atomic linked list at `subpath_heads`, keeping `resolveIntersectedMemReqs` as the sole writer to the reservation buffers for both open and closed paths.

Resolving Reservations of Intersected Paths The `resolveIntersectedMemReqs` kernel dispatches one thread per subpath. Threads whose subpath carries no intersections—determined by the `FLAG_SEGMENT_INTERSECTS` bit in `subpath_clipping_info`—exit immediately.

For each intersected subpath, the thread traverses the per-subpath linked list which was populated atomically by `reserveIntersected`. During traversal it performs an in-place conversion: it reads the raw byte sizes stored in each `reconstruction_reservations` entry and overwrites them with the current running totals `running_data` and `running_tags` before accumulating. After this step, every entry holds its subpath-relative write offset rather than its raw size.

Two categories of tags are accounted for alongside the geometric data. First, non-geometric structural tags—`PATH_TAG_TRANSFORM` and `PATH_TAG_STYLE`—are seeded into `running_tags` before the list traversal begins, conditioned on the subpath's flags. This offsets all geometric tag offsets written into the linked list entries by the correct amount. Second, `PATH_TAG_PATH` is appended to the running total after traversal if the subpath is the last in its path, terminating the path record. These structural tags are emitted by the `reconstructUnintersected` kernel on a per-subpath basis for every subpath, including those whose geometry has been merged into a different subpath; their tag budget must therefore be reserved here regardless of whether the linked list is empty. Once traversal completes, the accumulated totals are written to `subpath_req_memory` and `subpath_tags_req_memory`, which serve as the definitive memory budgets for the subsequent reconstruction passes.

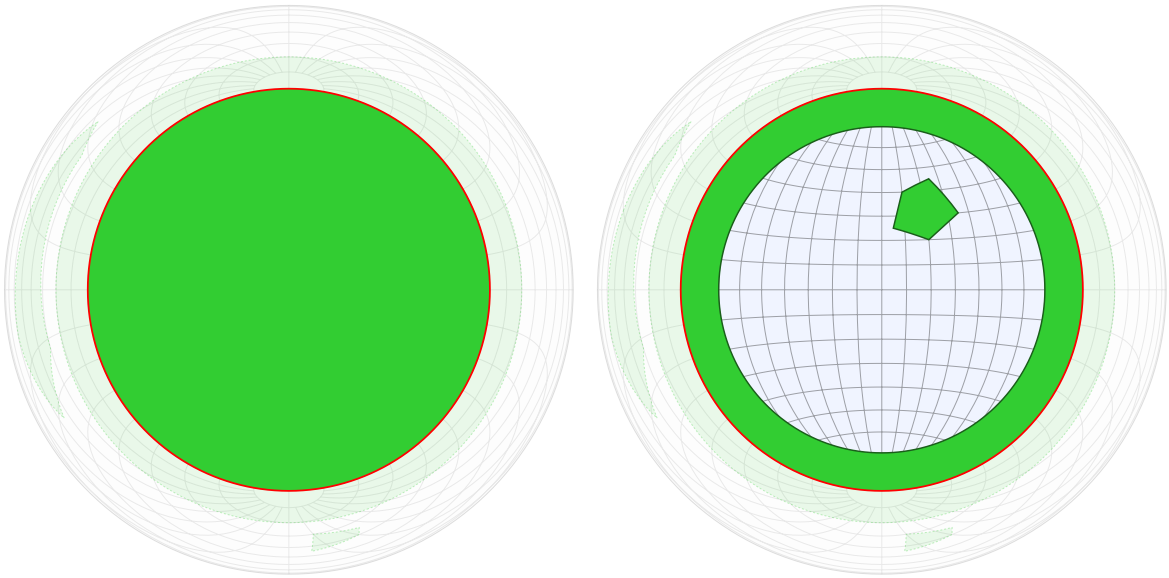
Reserving Non-Intersecting Geometry The `reserveUnintersected` kernel dispatches one thread per subpath. Any subpath carrying at least one intersection (deducible by the `FLAG_SEGMENT_INTERSECTS` bit being set) exits immediately—its memory has already been reserved by `reserveIntersected`.

For non-intersecting subpaths the outcome is binary, determined by `FLAG_SEGMENT_VISIBLE`. If set, the subpath lies entirely inside the clipping domain and is retained verbatim: its original byte extents from `subpath_data_info` and `subpath_tag_info` are written directly into the reservation buffers. If clear, the subpath lies entirely outside the domain and is discarded; no path data is reserved. In both cases, structural tags—`PATH_TAG_TRANSFORM`, `PATH_TAG_STYLE`, and `PATH_TAG_PATH`—are still accounted for as required to keep the downstream path stream well-formed.

A more subtle case arises when a closed polygon fully encloses the clipping boundary without any of its edges crossing it. This has to be detected on a path—not subpath—level. If the parent path contains this reference point and no subpath in the path intersects the boundary, the polygon must wind entirely around the clipping region. In this case the clipping boundary must be synthesized explicitly as a new subpath ring, since it cannot emerge from intersection traversal (see Figure 3.8). To emit it at most once per path, only the thread for the first closed subpath in the path assumes responsibility. The fully visible or invisible subpaths are retained or discarded as usual.

The synthesized ring reserves one implicit `MOVETO`, N `LINETO` segments as returned by `getFullInterpolationCount`, and—for stroked paths—the additional join closing `LINETO`. The tag count follows the same convention as the intersected case, including `PATH_TAG_PATH` if this is the last subpath in the path.

This condition occurs in two geometrically distinct configurations:



(a) Subject edges outside. The polygon's boundary lies entirely outside the clipping region and is discarded. The synthesized clipping boundary (red) is emitted as the sole output.

(b) Subject edges inside. The polygon's inner boundary (the hole) falls within the clipping region and is retained. The synthesized clipping boundary (red) is appended to form the new outer edge.

Figure 3.8.: Configurations requiring clipping boundary synthesis. When a closed polygon fully encompasses the clipping domain without its edges crossing the boundary, the clipping boundary cannot be extracted via standard intersection traversal. Instead, the boundary is explicitly synthesized (shown in red) to properly bound the visible geometry.

- **Subject edges outside, domain enclosed.** All subpath segments lie outside the clipping domain (`FLAG_SEGMENT_VISIBLE` clear), yet the polygon winds around it. None of the original subject edges fall within the visible region, so the original geometry is fully discarded. The only output is the synthesized boundary ring, which represents the entire clipping region as the complete visible area (Figure 3.8b).
- **Subject edges inside, domain enclosed.** All subpath segments lie inside the clipping domain (`FLAG_SEGMENT_VISIBLE` set), and the polygon is large enough that the clipping boundary also falls inside it. The original subpaths are either fully visible (retained) or invisible (discarded), and the synthesized boundary ring is appended as an additional subpath within the same path (Figure 3.8a).

In the second configuration, both the original geometry and the synthesized ring contribute to the same path, and `PATH_TAG_PATH` must be emitted exactly once. The boundary synthesis section claims this tag when the subpath is the last in its path; the original geometry's tag range is decremented by one to compensate.

Writing Geometry

We perform the Decoupled Fallback scan between the reservation and writing phases to obtain offsets into the final output buffers. A small kernel first checks if the provided output buffers are allocated with enough memory for the reserved geometry, and prepares the corresponding indirect dispatch arguments. If the buffers are under-allocated, it makes use of the `bump` allocator struct to set the failed state; the subsequent write passes are dispatched with zero workgroups. The writing phase is then divided into two separate kernels for intersected and non-intersected geometry, each of which handles both open and closed paths.

Writing Byte-Sized Tags In GPU environments, memory architectures natively operate on 32-bit or larger word boundaries, and compute APIs like WGSL generally lack native support for byte-level atomic operations. Because Vello path tags are 8-bit values tightly packed into `u32` arrays, concurrent threads attempting to write independent tags to adjacent byte lanes within the same memory word would inevitably induce data races and corrupt the output. To safely and efficiently serialize these sub-word writes, the shader utilizes a `TagBatcher` abstraction.

The `TagBatcher` acts as a thread-local accumulator. Instead of writing tags directly to global memory one by one, it packs up to four 8-bit tags into a single 32-bit `pending_data` register. It simultaneously maintains a 32-bit `pending_mask` that explicitly tracks which byte lanes (0 through 3) have been populated.

When a word is fully populated, or when a geometric sequence terminates and forces a flush, the batcher commits the accumulated word to the output buffer via `tagBatcherCommitWord`. The underlying commitment strategy dynamically adapts to the accumulated mask to minimize atomic overhead:

- **Exclusive Ownership:** If the thread has populated all four byte lanes (`mask == 0xFFFFFFFF`), it performs a direct, high-performance `atomicStore`, entirely overwriting

the destination word. This operation is faster than an `atomicOr` because it avoids the read-modify-write cycle at the hardware level.

- **Partial Ownership:** If the word is only partially filled—typically at the unaligned start or end of a subpath's tag stream—the batcher must preserve the neighboring bytes. Assuming the destination buffer is zero-initialized prior to the pass, the batcher simply executes a single `atomicOr` to safely inject the `pending_data`. Because concurrent threads operate on strictly disjoint byte lanes within the same word, their write masks never overlap. This allows the atomic operations to safely composite the final word without data races or clobbering adjacent bytes processed by other threads.

To further reduce global memory transactions, the API provides optimized routines for bulk operations. The `tagBatcherPushN` function is designed for emitting long sequences of identical tags, such as consecutive `LINETO` commands. Rather than pushing these tags individually, the function operates in three distinct phases to maximize memory throughput. First, the function evaluates the batcher's current `byte_lane`. If the cursor is unaligned, it pushes single tags sequentially until it reaches a 32-bit word boundary. Once aligned, it constructs a bulk 32-bit word by taking the single 8-bit tag and multiplying it by the hexadecimal constant `0x01010101`. This mathematical trick acts as a pattern broadcast, instantly replicating the tag across all four byte lanes of the register (e.g., an `0x09` tag becomes `0x09090909`). Because the cursor is perfectly aligned and the word is fully populated, the shader can bypass the `pending_data` accumulator entirely. It iterates through the remainder of the sequence in chunks of four, writing the pre-packed 32-bit pattern directly to global memory via a rapid, unmasked `atomicStore`. Any trailing tags (fewer than four) are then pushed back through the standard accumulator to prime the state for the next command.

Similarly, the `tagBatcherAppend` function accelerates the bulk copying of existing tag slices from the original geometry buffer. Because the source and destination byte offsets are rarely word-aligned in the exact same phase, this function dynamically manages alignment drift. After stepping forward to align the source cursor to a word boundary, it reads full 32-bit words directly from the source array. If the destination is out of phase, a single source word is split mathematically: the appropriate bits are shifted and deposited into the remaining lanes of the current destination word to trigger a flush, while the overflow bytes are carried over into the `pending_data` of the subsequent word. This bit-shifting pipeline drastically reduces the number of individual `readTagByte` invocations, transforming what would be a slow, byte-by-byte memory copy into a rapid stream of 32-bit loads and stores.

Writing Intersected Closed Paths The winning leader threads from the reservation phase re-traverse the `polygon_graph`. The topological traversal and routing logic remain identical to the reservation phase, but the shader now actively copies the geometry into the output buffers. The exact destination addresses are calculated by combining the subpath's absolute base offset—yielded by the parallel prefix sum—with the loop's relative offset computed during the linked-list resolution. Path data is written directly to this target slice. Meanwhile, path tags, which are 8-bit bytes rather than 32-bit words, are sequentially packed and written to the tag buffer using the `TagBatcher` API.

Writing Intersected Open Paths Similarly, the designated leader threads for open paths re-traverse the `line_graph` in the exact same linear sequence as they did during memory reservation. Final memory addresses are computed using the identical combination of absolute subpath offsets and relative reservation offsets. The shader copies the visible subject data slices, emits the required entry and exit intersection points, and writes any necessary bounding cap geometries. The corresponding tags for these segments are simultaneously encoded and flushed to memory via the `TagBatcher`.

Writing Non-Intersecting Geometry The `reconstructUnintersected` pass dispatches one thread per subpath to route unmodified geometry, synthesize encompassed boundaries, and emit non-geometric structural tags. For subpaths already processed by the intersected pass, threads bypass geometry entirely and exclusively write their structural preamble and terminator tags (`TRANSFORM`, `STYLE`, `PATH`). For non-intersecting subpaths, the pre-computed visibility state determines the outcome: fully visible subpaths copy their raw data directly via `writeSubjectSlice` and block-copy their tags using `tagBatcherAppend`, while fully clipped subpaths are discarded. Finally, if a path completely encompasses the clipping domain, the thread explicitly generates the full boundary contour via `getFullInterpolationPoint`. This synthesized ring is seamlessly appended to the parent path by deferring the terminating `PATH_TAG_PATH` until the new boundary geometry is fully constructed.

Dispatch Granularity Trade-offs While the current architecture dispatches the reservation and reconstruction kernels primarily with one thread per intersection, an alternative approach could optimize work distribution by dispatching per intersected subpath (for open paths) or per newly formed topological loop (for closed paths). This coarser granularity would require earlier kernels, such as `markBlockBounds`, to pre-compute these entity counts and allocate dispatch parameters via the `bump` allocator, inherently converting them into failable stages. Given that intersected subpaths are typically sparse, but the average number of intersections per subpath is small, the thread utilization gains of this approach must be weighed against the overhead of additional atomic allocations and pipeline failure points. Consequently, the per-intersection dispatch remains the baseline, with coarser granularities representing a potential optimization.

3.7. Stage 3: Projection and Adaptive Sampling

As the clipping operation can remove or add new segments and subpaths, the old `TagMonoid` array created in the setup phase is no longer valid; the same is true of the `SubpathInfo` structs. For this reason, we first have to run the full Vello scan pipeline again to generate an up-to-date `TagMonoid` buffer and use these in the `recomputeSubpathInfo` kernel to generate the `subpath_start_offsets` array. All other information that the `SubpathInfo` structs contained is not required in the projection and adaptive sampling passes.

The shader are compiled dynamically to import the chosen projection function. Analogous to the clipping stage, we run three passes: a reservation pass to calculate the memory requirements of the sampled and projected geometry, a device-wide scan to convert these into write offsets, and a final pass to write the geometry into the output buffers.

3.7.1. Reservation Pass

We dispatch one thread per path tag in the clipped geometry. Threads whose tags are non-geometric exit immediately.

For standard `LINETO` segments, projecting from spherical to Cartesian coordinates introduces curvature. To maintain visual fidelity, these segments are adaptively subdivided into piecewise linear approximations. The thread computes the required number of intermediate leaf points to satisfy a projection tolerance δ^2 via the `countAdaptiveLinetoPoints` routine.

Because compute shaders lacks native support for recursion, subdivision is implemented as an iterative depth-first search (DFS) constrained to a maximum depth of 16. The spatial and recursion state for each deferred branch is packed into a `StackItem` struct (Listing 7).

Listing 7 WGSL struct representing a node in the iterative depth-first search stack for adaptive curve subdivision.

```
struct StackItem {
    m_lon: f32, m_lat: f32, m_x: f32, m_y: f32,
    e_lon: f32, e_lat: f32, e_x: f32, e_y: f32,
    depth: u32
};
```

Noticeably absent from the `StackItem` are the 3D Cartesian unit vectors $\vec{p}_1$ and $\vec{p}_2$ associated with the segment endpoints. In a naive implementation, caching these vectors would increase the struct size by 24 bytes, expanding the maximum 16-deep stack footprint from 576 bytes to 960 bytes per thread. This nearly guarantees register spilling into slower thread-local memory. To aggressively optimize memory bandwidth, we trade ALU instructions for a smaller memory footprint: the struct strictly stores the essential 2D coordinates and recursion depth. When an item is popped from the stack, the 3D vectors are re-evaluated on-the-fly via the `cartesianUnit` routine.

Furthermore, to mitigate the latency of dynamic indexing, the traversal stack employs a register-caching optimization: the two most recently pushed `StackItem` elements are held in statically addressed thread-local variables (`r0` and `r1`). This distinction is critical because GPU hardware registers generally lack support for dynamic indexing. When a shader indexes into a function-scoped array using a variable stack pointer, the compiler is forced to spill that array into thread-local ”scratch” memory. Physically, this scratch space resides in off-chip global VRAM; while it is heavily backed by the L1 cache, accessing it still incurs higher latency and consumes valuable cache lines. By keeping the hottest elements of shallow branches in true hardware registers and delaying the spill to the VRAM-backed array, the shader aggressively optimizes for high-occupancy throughput. This transition between registers and local memory, along with the refinement logic, is visualized in Figure 3.9.

At each step, the algorithm must compute the true spherical midpoint of the segment. Rather than naively interpolating longitude and latitude—which ignores the Earth's curvature and produces incorrect geodetic trajectories (see Figure 2.9b)—the shader evaluates the 3D Cartesian unit vectors $\vec{p}_1$ and $\vec{p}_2$ of the endpoints, averages them, normalizes the result, and projects the vector back into spherical coordinates.

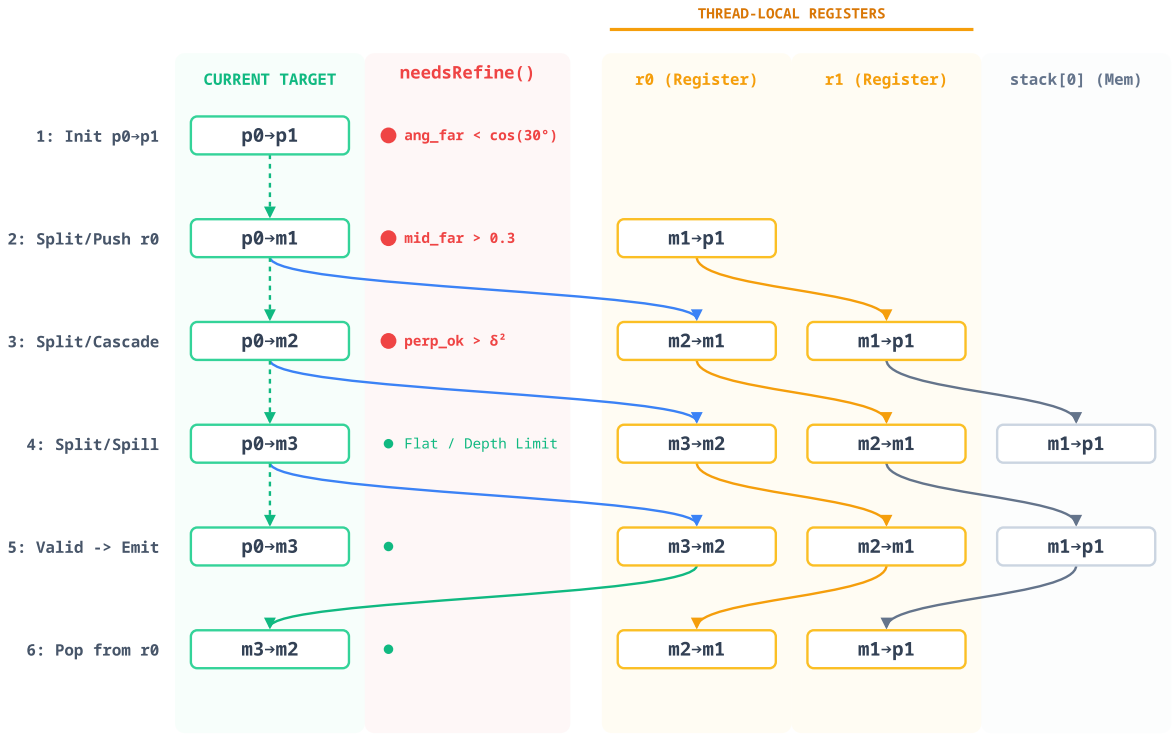


Figure 3.9.: Visualization of the iterative DFS stack execution. The diagram illustrates the `needsRefine()` gatekeeper logic and the register-caching mechanism, where the top of the stack is prioritized in fast registers `r0` and `r1` before spilling to the thread-local `stack` array.

The segment is recursively subdivided if the squared Euclidean distance d^2 between the projected endpoints (x_1, y_1) and (x_2, y_2) is greater than $4\delta^2$ and any of the following geometric refinement criteria are triggered:

- **Perpendicular Distance:** The squared perpendicular distance from the projected midpoint (x_{mid}, y_{mid}) to the projected segment chord exceeds δ^2. This is evaluated as $\Delta z^2 / \max(d^2, \epsilon)$, where $\Delta z = \Delta y \Delta x_{mid} - \Delta x \Delta y_{mid}$.
- **Midpoint Skew:** The parameter t of the projected midpoint—representing its relative orthogonal projection onto the chord—deviates from the linear interpolation center (0.5) by more than 0.3. This detects non-linear parametric distortions where the projected midpoint is pulled disproportionately toward either endpoint.
- **Angular Distance:** The dot product $\vec{p}_1 \cdot \vec{p}_2$ falls below $\cos(30^\circ)$. This prevents highly curved spherical arcs from being severely undersampled, ensuring sufficient vertex density regardless of their current projected footprint.

For subpaths at the very beginning of a path, an additional two 32-bit words are reserved to encode the implicit initial `MOVETO` coordinate.

Vello's parallel stroke expansion dynamically generates offset curves, caps, and joins by extracting local information of adjacent segments. We do not want to subdivide these segments, but instead synthesize them such that they are understood and rendered correctly by the Vello rendering backend. This is done in the write pass, we bypass adaptive subdivision counting entirely.

3.7.2. Write Pass

Following the prefix sum, the write-pass is dispatched with one thread per tag. Threads read their absolute offsets from the `offsets_u32` and `offsets_tags` arrays to route the generated geometry into the pre-allocated output buffers.

Tags are sequentially packed and committed to the tag buffer utilizing the `TagBatcher` abstraction to safely serialize concurrent sub-word byte writes.

Emitting Adaptive Geometry

For standard `LINETO` commands, the thread mirrors the exact iterative depth-first traversal executed in the reservation pass. The validated leaf coordinates are projected into 2D Cartesian space via the imported `project` function.

The resulting x and y float coordinates are written directly to the `path_data_clipped` array. As all adaptively sampled points are `LINETO` segments we can batch many corresponding tags using `tagBatcherPushN` at once before flushing.

Synthesizing Stroke Joins and Caps

The local information required to correctly draw cap and join segments are the tangents of neighboring segments. While these tangents are contiguous for consecutive continuous segments, connectivity is broken at structural boundaries such as closed path wraps or open path starts. Because the projection from spherical coordinates introduces non-linear curvature, the Cartesian tangent at these junctions cannot be inferred via linear interpolation of the original endpoints and must be explicitly synthesized.

For closed strokes, the `LINETO | SUBPATH_END` tag denotes the final segment connecting back to the structural start of the loop. To construct the correct closing join geometry, the thread uses the pre-computed `subpath_start_offsets` array to locate and read the initial coordinates of the subpath. Because spherical coordinates are periodic, segments crossing the antimeridian exhibit a longitudinal discontinuity between π and $-\pi$. To prevent the tangent vector from incorrectly routing the long way around the globe (similar to the artifact with clipping disabled, see Figure 3.2), the `unwrapLongitude` function adjusts the second coordinate's longitude relative to the first, adding or subtracting 2π if their absolute difference exceeds π. The thread then evaluates the true Cartesian tangent on this continuous local segment via `findProjectedStartTangent`.

Vello internally degree-elevates `LINETO` segments to cubic Bézier curves, calculating the relevant control point as $p_2 = p_3 + \frac{1}{3}(p_0 - p_3)$. To match this expected data layout and supply the correct arrival angle for join evaluation, the shader explicitly outputs a synthesized `LINETO` containing the projected start point p_{proj} offset by one-third of the tangent vector: $p_{\text{proj}} + \frac{1}{3}\vec{v}_{\text{tan}}$.

Similarly, start caps for open strokes cannot be deduced from preceding geometry. Vello dictates a strict structural encoding for this boundary condition: an open subpath start cap is represented by a preceding `QUADTO | SUBPATH_END` marker. When the thread encounters a `QUADTO` segment in this phase, it serves exclusively as this cap marker. The thread reads the structural start coordinates via `subpath_start_offsets`, applies `unwrapLongitude` to ensure geodetic continuity, computes the projected start tangent, and writes the origin p_{proj} alongside the pre-scaled tangent point. This reserves a fixed four 32-bit words and one tag byte, providing Vello with the data required to draw oriented start caps.

4. Quantitative Evaluation

This chapter evaluates the performance of the proposed end-to-end compute shader pipeline across various datasets and metrics, benchmarking the results against the `d3-geo` library.

4.1. Introduction

The benchmark is mainly focused on measuring the execution time and memory usage of the compute shader pipeline and comparing it to the `d3-geo` library, heavily focusing on the scaling properties given the size and complexity of the input data. To achieve this, the benchmarking harness defines a strict matrix of configurable parameters that stress different stages of the mathematical and rendering pipelines.

The test matrix evaluates performance across three primary dimensions:

- **Resolution & Complexity:** Benchmarks span low-resolution maps (`110m`), medium-resolution maps (`50m`), and a combined configuration (`110m_poly`) introducing complex custom polygons (e.g., spirals, poles, and donuts) alongside standard graticules. These shapes act as geometric edge-cases specifically designed to trigger heavy branching behavior within the compute shaders. Combined with the relatively small input size, this configuration is intended to represent a best-case performance scenario for `d3-geo` relative to the GPU pipeline.
- **Clipping Strategy:** Scenes are rendered using either Antimeridian clipping (for equirectangular projections) or Circle clipping at varying radii (45°, $90^\circ + \epsilon$, and $180^\circ - \epsilon$). These boundaries test the pipeline's intersection, sorting, and polygon-reconstruction phases.
- **Kinematics:** Viewport rotation is driven by two distinct generation strategies. A *Smooth* provider simulates natural, frame-to-frame coherent user interaction via interpolated waypoints, while a *Random* provider induces worst-case spatial thrashing by jumping between uncorrelated rotations.

The resulting cross-product of the resolution and clipping configurations is visualized in Figure 4.1.

4.2. Setup

All benchmarks were executed on a consumer desktop machine to reflect a realistic usage environment. The system is powered by an AMD Ryzen 5 3600 CPU (Zen 2 architecture), featuring 6 performance cores and 12 threads with a base clock of 3.6 GHz, a maximum

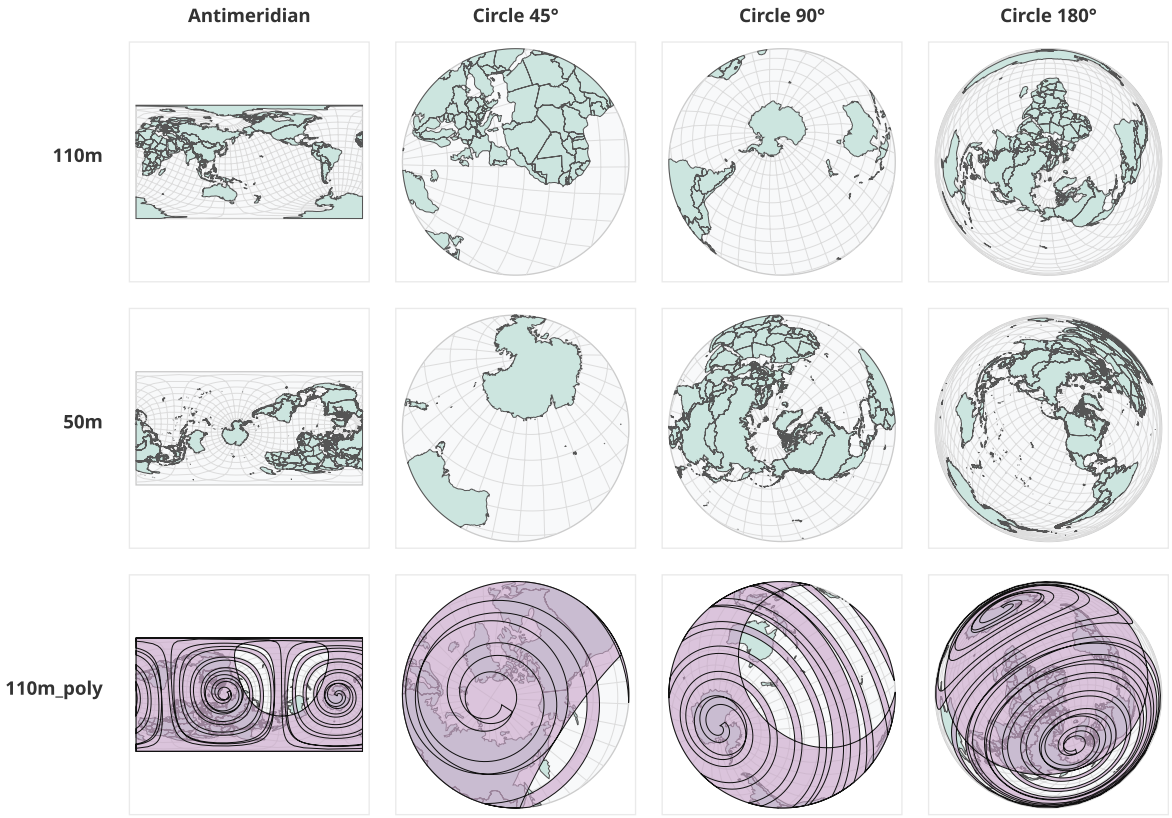


Figure 4.1.: A 3×4 visual matrix of the benchmarking configurations. The rows represent the three dataset complexities (`110m`, `50m`, and `110m_poly`), while the columns illustrate the four clipping strategies (Antimeridian, Circle at 45°, Circle at $90^\circ + \epsilon$, and Circle at $180^\circ - \epsilon$).

boost clock of 4.2 GHz, and a combined 35 MB of L2 and L3 cache. Graphics processing and hardware acceleration are handled by an NVIDIA GeForce RTX 2070 Super GPU (Turing architecture) equipped with 8 GB of GDDR6 VRAM and 2,560 CUDA cores. Operating at a base clock of 1,605 MHz, the GPU delivers a theoretical single-precision compute performance of approximately 9.06 TFLOPs. The system is provisioned with 16 GB of main system RAM. To minimize interference and variance during profiling, non-essential background processes were terminated prior to execution, ensuring the discrete GPU and CPU were as isolated as possible for the benchmark workloads.

The benchmark compares a JavaScript web library with a Rust/WGPU graphics pipeline. Since these technology stacks are fundamentally different, they require two distinct but symmetrically designed testing harnesses. To guarantee deterministic and comparable workloads, both environments use an identical `mulberry32`[1] pseudorandom number generator. Symmetrical seeding ensures both systems generate the exact same sequence of rotation parameters.

4.2.1. Operational Semantics

For every unique configuration in the test matrix, both harnesses perform six sequential passes to cleanly separate raw throughput measurements from profiling overhead.

1. **Passes 1–5 (Throughput):** Each pass renders 1,000 consecutive frames. Frame-to-frame deltas are aggregated to calculate the average inter-frame execution time, the 99$^{\text{th}}$ percentile (P99) frame time, and the overall Frames Per Second (FPS).

2. **Pass 6 (Deep Profiling):** A final pass is executed to gather per-function and per-stage timing distributions. Because deep profiling introduces a measurable overhead, this pass is excluded from the throughput metrics. The profiling methodology differs between the two harnesses: the `d3-geo` benchmark utilizes a sampling-based profiler that periodically interrupts execution (e.g., every 1 ms) to record the current call stack, building a statistical distribution of time spent in functions. To ensure this statistical sampling catches enough occurrences of shorter, relevant JavaScript function calls, this pass runs for 3,000 frames. Conversely, the GPU pipeline utilizes `wgpu-profiler` to inject exact hardware timer queries into the command buffers. For this deterministic measurement, running the pass for 1,000 frames is sufficient.

4.2.2. Test Datasets

To evaluate pipeline performance across varying levels of topological density, the benchmark utilizes three dataset configurations derived from the Natural Earth *Admin 0 – Countries* cultural vector suite.[2] These datasets, converted to GeoJSON format, represent distinct spatial resolutions (110m and 50m) to simulate different computational loads.

[1] Tommy Ettinger, *Mulberry32 PRNG*. Reference implementation available at `https://gist.github.com/tommyettinger/46a874533244883189143505d203312c`

[2] `https://www.naturalearthdata.com/` (last accessed 09.03.2026)

To ensure a consistent geometric baseline, each configuration is augmented with a background graticule. This grid is generated with 90° major and 10° minor intervals, utilizing linear segment interpolation capped at a maximum step of 10° to preserve projection smoothness and prevent edge artifacts. The resulting geometric complexity and memory footprint for each configuration encoded into a Vello scene are summarized in Table 4.1.

Config	Paths	Subpaths	Open Strokes	Fills & Closed Strokes	Segments	Size
`110m`	568	621	53	284	22,133	0.20 MB
`110m_poly`	578	635	53	291	29,814	0.27 MB
`50m`	3,232	3,307	53	1,627	198,702	1.76 MB

Table 4.1.: Geometric complexity of the benchmark datasets. Subpaths are categorized into lines (*Open Strokes*, comprising the graticule) and closed polygons. Note that the value in the *Fills & Closed Strokes* column denotes the count for *each* category individually; for example, the `110m` dataset contains 284 fills *and* 284 closed strokes, generating two independent subpaths per polygon. The raw *Size* represents the combined memory footprint of the vertex coordinates, path tags, and style definitions prior to any pipeline processing.

4.2.3. The `d3-geo` Baseline Harness

The baseline measurements are gathered using a custom web-based harness rendering `d3-geo` paths via the HTML5 Canvas 2D API[3]. Gathering detailed function timings and unthrottled performance data requires a highly automated approach:

- **Execution:** The harness is executed in a Google Chrome instance controlled by Puppeteer[4]. Chrome is explicitly launched with the `--disable-frame-rate-limit` and `--disable-gpu-vsync` flags to bypass the browser's compositor VSync capping, allowing the `requestAnimationFrame` loop to run unconstrained.
- **Profiling:** To avoid the overhead of manual timing in `d3-geo`, we considered the JS Self-Profiling API[5]. However, its 10 ms minimum interval is too coarse for frame-by-frame graphics analysis. While Chrome DevTools provides higher resolution, it lacks the automation required for matrix testing. We therefore use Puppeteer to interface with the Chrome DevTools Protocol[6] (CDP), allowing us to configure a high-resolution 1 ms sampling rate.
- **Memory:** System RAM footprint is captured at the end of a run using Puppeteer's `page.metrics()` to record the `JSHeapUsedSize`.

[3] `https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API` (last accessed 08.03.2026)
[4] `https://pptr.dev/` (last accessed 08.03.2026)
[5] `https://developer.mozilla.org/en-US/docs/Web/API/JS_Self-Profiling_API`
[6] `https://chromedevtools.github.io/devtools-protocol/`

4.2.4. The GPU Pipeline Harness

The compute shader pipeline is benchmarked using a Rust application built upon the `egui`[7] framework and the `vello` rendering engine.

- **Execution:** The `wgpu` surface is configured with `PresentMode::AutoNoVsync` and the rendering strategy is locked to a synchronous mode. This prevents asynchronous thread yielding and message channel overload caused by the overhead of constantly firing update messages between the UI and render threads.
- **Profiling:** Granular GPU performance data is extracted using `wgpu-profiler`[8]. Unlike statistical sampling, this relies on explicitly requesting the `TIMESTAMP_QUERY` feature from the device. This allows the harness to inject precise timer queries directly into the WGPU command buffers, yielding absolute execution times for individual compute stages alongside CPU-side recording, encoding, polling, and downloading phases.
- **Memory:** VRAM footprint is measured by querying the custom `WgpuEngine` resource manager, which aggregates the exact byte sizes of all active GPU buffers (`BindMap`) and inactive cached buffers (`ResourcePool`). Because host-side buffer allocations are quantized to power-of-two-like size classes (aligned to 16 bytes) to reduce fragmentation, this measurement reflects the true memory overhead allocated on the device. Additionally, the harness tracks the frequency of GPU-side bump allocator retries, which occur when dynamic pre-allocated shader memory limits are exceeded.

4.3. Performance Benchmark

4.3.1. Maximum Throughput

The throughput evaluation relies on two primary metrics: average execution time per frame (measured in milliseconds) and the resulting FPS. In addition to average throughput, the 99^{th} percentile (P99) frame time is recorded. In real-time graphics pipelines, average FPS is insufficient for determining perceived smoothness; if a small percentage of frames take significantly longer to render, the user experiences visible micro-stutter. The P99 metric isolates these worst-case latencies, serving as a critical indicator of pipeline stability and frame pacing.

Initial baseline comparisons evaluated execution times under both *Smooth* (frame-to-frame coherent) and *Random* (uncorrelated) rotation kinematics. The empirical data indicates almost no performance discrepancy between the two generation strategies. For instance, evaluating the `110m` dataset under Antimeridian clipping yields an average frame time of 7.6 ms (Smooth) versus 7.9 ms (Random) for `d3-geo`, and 2.0 ms versus 2.1 ms for the GPU pipeline. Because neither pipeline relies on temporal coherence or spatial caching across frames, viewport thrashing does not meaningfully stall execution. Consequently, the remainder of this analysis focuses exclusively on the *Smooth* rotation datasets (Figure 4.2).

[7] `https://github.com/emilk/egui` (last accessed 28.02.2026)
[8] `https://github.com/Wumpf/wgpu-profiler` (last accessed 08.03.2026)

Figure 4.2.: Pipeline throughput comparison under Smooth rotation kinematics. The chart displays average frame times alongside P99 error bars across varying clipping strategies and dataset complexities, referenced against standard 30, 60, and 120 FPS rendering budgets.

On lower-resolution datasets (`110m` and `110m_poly`), the GPU pipeline consistently outperforms the `d3-geo` baseline. While `d3-geo` maintains acceptable real-time performance, averaging between 4.6 ms and 11.3 ms (roughly 88 to 215 FPS), the GPU compute architecture operates between 1.5 ms and 2.4 ms (exceeding 400 FPS across all configurations).

This fundamental difference in scaling behavior is most pronounced when processing the high-resolution `50m` dataset. The `d3-geo` pipeline exhibits severe performance degradation, with average frame times increasing from 32.7 ms (Circle 45°) up to 137.1 ms (Circle 180° and Antimeridian). This linear scaling effectively drops the throughput to between 7 and 30 FPS, failing to meet the strict 16.6 ms budget required for standard 60 FPS rendering. Furthermore, its P99 frame times stretch up to 153.3 ms, indicating severe frame pacing instability and frequent stuttering.

In contrast, the GPU pipeline amortizes the increased geometric complexity across its parallel compute units, resulting in highly stable sub-linear scaling. Frame times for the `50m` dataset increase only to an envelope of 3.1 ms to 7.0 ms (142 to 313 FPS). Additionally, the GPU pipeline maintains exceptionally tight P99 variances. For example, under Antimeridian clipping on the `50m` dataset, the average frame time of 7.0 ms is accompanied by a P99 of only 7.9 ms.

4.3.2. Macro-Architectural Breakdown

Analyzing the macro-architectural timing breakdown (Figure 4.3) demonstrates the distribution of workload between the host (CPU) and device (GPU). The data indicates that the pipeline's overall throughput is primarily governed by GPU execution time rather than host-side orchestration.

The actual GPU execution and polling phase dictates the majority of the frame time, particularly as dataset complexity increases. This compute phase scales proportionally with both the dataset's density and the resulting volume of topological intersections. While GPU execution accounts for only 0.6 ms to 0.8 ms on the `110m` datasets, it dominates the pipeline on the significantly denser `50m` datasets. Under Antimeridian clipping on the `50m` input, for example, GPU execution time reaches approximately 5.1 ms out of the 7.0 ms total frame time.

In contrast, the CPU incurs a comparatively stable baseline cost for its core responsibilities: allocating resources, uploading data, building command buffers, and encoding shader dispatches. Across the `110m` and `110m_poly` configurations, CPU command encoding requires approximately 0.7 ms. When transitioning to the `50m` dataset, this host-side overhead increases only moderately, ranging from 0.8 ms to 1.3 ms depending on the specific clipping strategy.

This variance in CPU time is an artifact of the pipeline's current CPU-GPU synchronization boundary. Although the CPU requires relatively little time to build the command buffers, it cannot proceed asynchronously. After submitting the command queue, the CPU must explicitly block and wait for the GPU to finish execution. This hard synchronization barrier is mandatory for two architectural reasons. First, the host must read back the bump allocation buffer to verify that pre-allocated dynamic memory limits were not exceeded during the

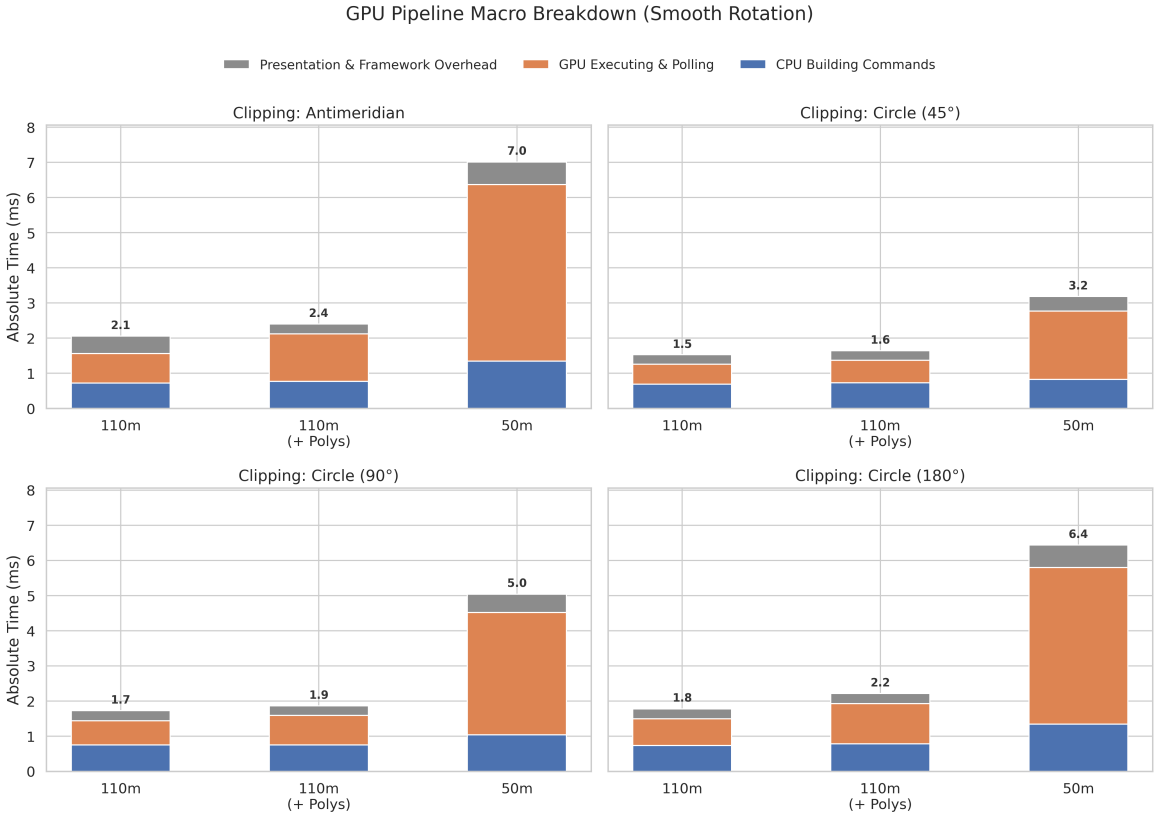


Figure 4.3.: Macro-architectural timing breakdown of the GPU pipeline under Smooth rotation. The stacked bars illustrate the absolute execution time (in milliseconds) distributed across CPU command building, GPU execution and polling, and presentation/framework overhead.

compute passes, triggering a pipeline retry if an under-allocation occurred. Second, because the current Vello rendering API does not support ingesting pre-encoded scene buffers directly from GPU memory, the computed vertices and path tags must be downloaded back to system memory. The CPU must then explicitly splice this geometric data into the mutable Vello scene encoding before re-uploading it to the GPU for rasterization.

Consequently, the CPU overhead scales dynamically with the volume of data that survives the clipping pass, forcing the system into an expensive VRAM-to-RAM-to-VRAM data round-trip. Highly restrictive geometries, such as the 45° circle, cull the majority of the dataset and result in minimal host-side transfer and processing times. Conversely, global projections like the Antimeridian clip force the CPU to download, process, and re-upload the entire surviving dataset, effectively stalling the CPU during the compute phase.

The system's remaining execution time is categorized as Presentation & Framework Overhead, which occupies roughly 0.3 ms to 0.6 ms per frame across all tested configurations. This fraction accounts for OS windowing event polling, WGPU queue submission, swapchain acquisition, and the implicit synchronization of the final frame presentation. Crucially, because the aforementioned synchronization point occurs before the Vello rendering commands are encoded, the measured GPU execution time strictly isolates the geographic compute shaders from the final rasterization workload. Vello operates asynchronously at the end of the frame; if its GPU rasterization time forces the CPU to wait during swapchain acquisition on the subsequent frame, this delay is captured entirely within the Presentation & Framework Overhead rather than the compute polling time. Resolving the current API limitations to allow a pure, GPU-resident pipeline—thereby eliminating the synchronous readback—would remove this host-side bottleneck and unlock significantly higher maximum throughput.

4.3.3. Micro-Profiling and Algorithmic Trade-offs

The GPU pipeline utilizes precise hardware timer queries (`TIMESTAMP_QUERY`) to measure the exact execution time of compute dispatches. In contrast, the HTML5 Canvas 2D API rasterizes geometry asynchronously, entirely outside the main JavaScript thread. Profiling the execution context in JavaScript captures only the time required to queue rendering commands (e.g., `moveTo`, `lineTo`) rather than the true cost of pixel rasterization. Furthermore, because the W3C proposal for a `requestPostAnimationFrame` API stalled, developers lack a reliable hook to determine exactly when the GPU finishes drawing a frame.

Consequently, the actual rasterization stage of the `d3-geo` pipeline remains opaque and cannot be profiled from the main thread. To ensure a fair, "apples-to-apples" evaluation, the rendering compute time of the GPU pipeline (specifically the `vello.*` passes) is excluded from this granular comparison. The resulting analysis focuses strictly on the core computational geometry pipeline.

To effectively compare the highly parallel compute shader pipeline with the single-threaded execution of `d3-geo`, the raw profiling traces from both environments are grouped into shared, high-level logical stages (the precise mapping of individual functions to these stages is detailed in Appendix A).

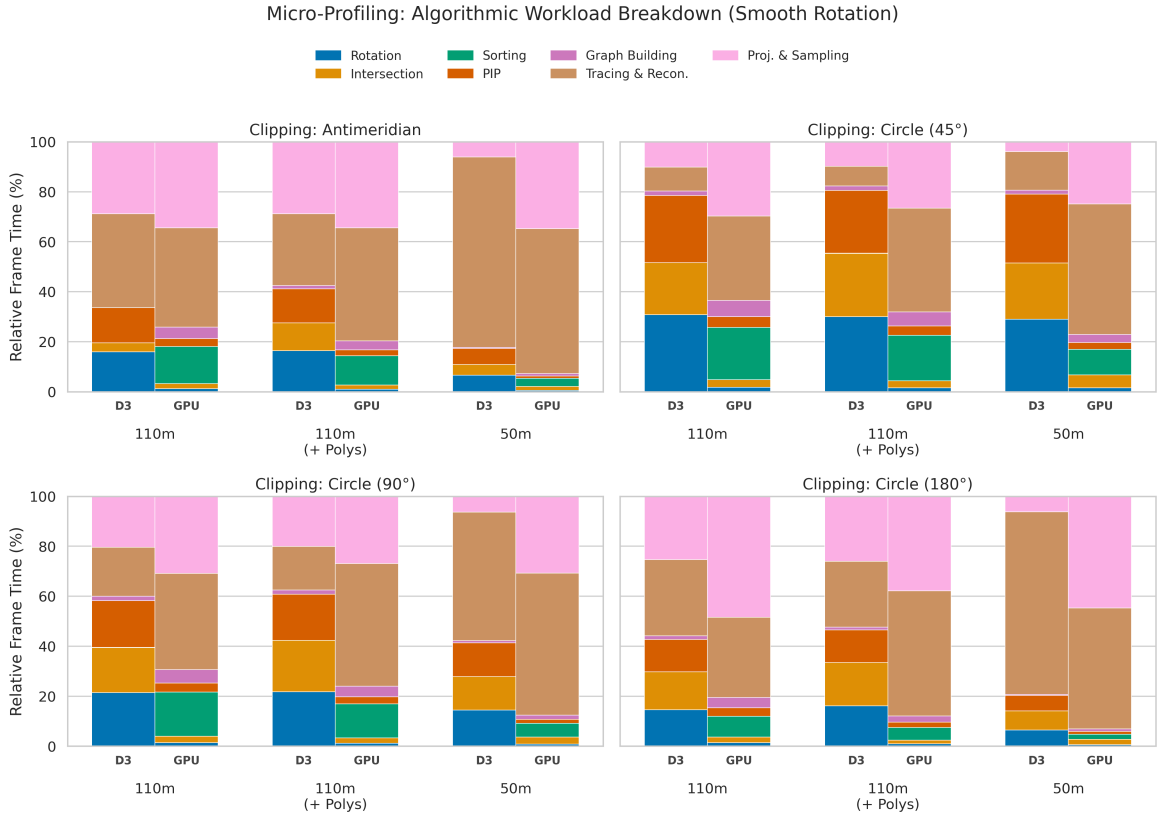


Figure 4.4.: Micro-profiling breakdown of relative frame time (%) for the `d3-geo` (D3) and GPU pipelines. The comparison spans three dataset complexities (110m, 110m + Polygons, and 50m) across four clipping configurations: Antimeridian, and Small-Circle (45°, 90°, 180°). Rendering/rasterization time is excluded to isolate core geometric computation.

As illustrated in the micro-profiling distributions in Figure 4.4, migrating from a sequential CPU environment to a massively parallel GPU architecture necessitates fundamental algorithmic trade-offs:

- **1. Rotation (Blue):** A heavy trigonometric bottleneck on the CPU that vanishes almost entirely on the GPU due to perfect spatial parallelism.
- **2. Intersection (Yellow):** A highly variable cost in `d3-geo` dependent on the clipping geometry. On the GPU, this independent, ALU-bound workload scales optimally across the streaming multiprocessors, reducing its relative impact to a minor fraction of the frame time.
- **3. Sorting (Green):** A necessary architectural tax. While `d3-geo` processes small intersection arrays so rapidly that the cost frequently evades the 1 ms sampling profiler, the GPU pipeline must dispatch three separate sorting passes. This dispatch overhead accounts for a significant fraction of the GPU frame time, despite the intersections often fitting efficiently within local workgroup memory.
- **4. Point-in-Polygon (Dark Orange):** A severe sequential bottleneck for the CPU due to linear edge iteration and adaptive-precision summation. On the GPU, concurrent edge evaluation and hierarchical reduction absorb this heavy trigonometric workload, reducing it to a minimal fraction of the frame time.
- **5. Graph Building (Purple):** The preamble to reconstruction. A minor cost for both architectures, though slightly amplified on the GPU due to the dispatch overhead of constructing and linking flat topological arrays in-place without dynamic memory allocation.
- **6. Tracing & Reconstruction (Brown):** The most severe bottleneck in the GPU pipeline. Sequentially traversing graphs and writing dynamically sized, rejoined polygons back to global VRAM across multiple reservation and write passes starves the GPU's wide memory bus, whereas the CPU handles dynamic array allocation efficiently.
- **7. Projection & Sampling (Pink):** A highly efficient stage on the CPU, which handles recursive curve subdivision within hot L1/L2 caches. On the GPU, the lack of dynamic allocation forces the pipeline to execute the adaptive subdivision logic twice (count, then write), bounding the stage by the high latency of writing massive arrays of transformed vertices out to global VRAM.

Rotation

Spherical rotation applies three Euler angles (yaw, pitch, and roll) to incoming geographical coordinates. Both the CPU and GPU pipelines implement this mathematically through two distinct execution paths. For pure yaw rotations, a computationally inexpensive 2D shift is applied to the longitude. However, oblique rotations (involving pitch or roll) require converting the spherical coordinates into 3D Cartesian space, applying a rotation matrix, and converting the resulting vector back to spherical coordinates.

In `d3-geo`, this stage accounts for a significant portion of the total execution budget. Under the `110m` resolution, rotation consumes 14.5% to 30.8% of the CPU's processing time, depending

on the clipping configuration. The requirement to evaluate trigonometric functions—such as `sin`, `cos`, `asin`, and `atan2`—sequentially across all vertex coordinate pairs creates an arithmetic logic unit (ALU) bound workload on the single-threaded CPU.

Conversely, spherical rotation exhibits high spatial data independence, allowing for straightforward parallelization. By mapping one thread to exactly one vertex, the GPU's floating-point units process the trigonometric workload concurrently. Consequently, the rotation phase constitutes a minimal fraction of the GPU pipeline, never exceeding 1.75% of the compute budget across any tested configuration. On the denser `50m` datasets, where the absolute execution time of other pipeline stages increases significantly, the relative cost of rotation on the GPU decreases to as low as 0.48%. This contrast demonstrates how parallel ALU throughput effectively mitigates the sequential trigonometric bottlenecks observed on the CPU.

Intersection

The intersection stage computes the precise topological collision points between individual line segments and the projection's active clipping boundary.

In `d3-geo`, the computational overhead of this stage exhibits high variance due to a compounding of geometric density and mathematical complexity. For a standard antimeridian clip, the algorithm stays within 2D spherical coordinates, utilizing a relatively lightweight trigonometric formula to evaluate boundary crossings. Because the mathematics are computationally inexpensive, this sequence consumes a modest 3.6% to 4.2% of the CPU's processing time.

However, small-circle clipping requires a fundamentally heavier mathematical algorithm. To find intersections with an arbitrary circle, `d3-geo` must first convert the 2D spherical coordinates into 3D Cartesian space. It then computes plane intersections using vector cross and dot products before converting the resulting 3D vectors back into latitude and longitude. When this expensive 3D sequence is executed sequentially to resolve potentially many boundary crossings—such as in the 45° small-circle configuration—the workload escalates into a primary architectural bottleneck, consuming up to 25.3% of the CPU's total execution time.

In contrast, the GPU pipeline maps line-segment intersections to a massively parallel compute pass. Unlike the rotation stage, intersection testing is not entirely free of inter-thread dependencies. Because each line segment can yield a variable number of intersections (zero, one, or two), the threads must synchronize to tightly pack the dynamic results into dense global buffers. This is achieved by executing local prefix sums (Blelloch scans) within shared workgroup memory, followed by atomic additions to a global bump allocator. Despite this synchronization overhead, the GPU's massive ALU throughput easily absorbs the heavy Cartesian mathematics. The micro-profiling data confirms that this parallel architecture effectively neutralizes the latency penalty of complex clipping shapes: even when processing the dense `50m` dataset against a 45° small circle, the intersection stage accounts for only 5.2% of the GPU compute budget.

Sorting

Once intersections are generated, they must be deterministically ordered to allow for the correct topological reconstruction of the clipped geometries.

In `d3-geo`, the cost of sorting is statistically negligible. Across all tested datasets and clipping configurations, the sorting phase accounts for merely 0.00% to 0.05% of the total execution time. Because the CPU executes sequentially, it isolates and sorts small, per-polygon arrays of intersections on-the-fly using native array sorting routines. The comparison function relies on just a few basic arithmetic operations to order points along the clipping boundary. This simplicity, combined with the small size of the arrays, allows the operation to complete so rapidly that it consistently evades the 1 ms sampling interval of the JavaScript profiler.

Conversely, the GPU sorting pipeline incurs a substantial architectural tax as we must execute the sorting sequence three distinct times per frame: sorting open path intersections by segment order, closed path intersections by segment order, and closed path intersections by their traversal order along the clipping boundary.

Crucially, the total volume of intersections generated—even on the dense `50m` dataset under a 45° small-circle clip—is small enough to fit entirely within the shared memory capacity of a single compute workgroup. As a result, the parallel merge sort completes entirely within the local `blockSort` pass, effectively bypassing the severe latency penalties of global VRAM multi-pass merging.

Despite this, the sorting phase represents a major fraction of the frame on the lower-density `110m` configurations, consuming between 8.2% and 20.9% of the total GPU compute budget depending on the clip. This is driven by global memory latency and poor GPU occupancy. Because the intersection arrays are so small, most of the GPU's streaming multiprocessors remain idle, yet the active workgroups must still endure the fixed latency of reading structs from global VRAM, generating complex 96-bit keys, and writing the results back. When processing the denser `50m` dataset, the absolute time spent sorting increases, but its relative share of the frame time drops to between 2.0% and 10.1%. This proportional decrease occurs because the memory-bandwidth-bound reconstruction stages expand exponentially with the denser geometry, dwarfing the sorting cost.

Point-in-Polygon (PIP) Test

To determine the initial containment state of a polygon relative to the projection's reference point, the pipeline must accumulate the polygon's longitudinal winding and spherical excess.

In `d3-geo`, this stage represents a major sequential bottleneck. The CPU must linearly iterate over every line segment of a subject polygon to evaluate its geometric contribution, accumulating the results using Shewchuk's adaptive-precision floating-point summation algorithm to prevent numerical drift. Because this heavy loop of trigonometric functions and Cartesian cross products scales directly with the number of vertices, it consumes a significant portion of the total execution time, ranging from 6.2% to 27.6% depending on the dataset density and clipping configuration.

In contrast, the native GPU pipeline parallelizes the point-in-polygon test by assigning distinct chunks of path segments to concurrent threads. Translating this sequential summation to a parallel architecture introduces necessary synchronization and algorithmic overhead to maintain numerical stability. Specifically, the pipeline employs localized Kahan summation, segmented parallel scans in shared memory, and a decoupled two-pass map-reduce architecture to safely aggregate high-precision floating-point partial sums across workgroups.

Despite this architectural overhead—including the necessity of an API-level pipeline barrier between the two passes—the GPU's massive ALU throughput easily absorbs the mathematical workload. Consequently, the relative cost of the PIP stage drops dramatically, never exceeding 4.2% of the compute budget across any `110m` configuration. The parallel scaling advantage becomes most apparent on the exceptionally dense `50m` dataset; while the sheer volume of segments drives the sequential CPU cost up to 27.6% of the frame time, the GPU processes the denser geometry concurrently, reducing the relative cost of the PIP phase to a minimal 0.8% to 2.7%.

While this two-pass architecture with dynamic error bounding significantly improves precision compared to naive parallel floating-point accumulation, it is not immune to numerical instability. Even Kahan summation can suffer from catastrophic cancellation under specific geometric configurations, leading to rare but persistent topological inversions. These precision limits and their effect on the final rendering output are discussed in detail in Section 5.2.

Graph Building

To generate correctly clipped and stitched geometries, the intersection points must be linked into a topological graph that defines the final traversal order.

In `d3-geo`, this stage constitutes a statistically negligible portion of the execution time, typically accounting for less than 1.9% of the frame across all configurations. The CPU constructs the graph sequentially by dynamically allocating objects and linking them via standard pointer references. Because the CPU possesses a highly optimized memory hierarchy and branch predictor, it handles this linked-list construction with minimal overhead, particularly since the total number of intersections is relatively small.

In contrast, the native GPU pipeline must construct this routed graph in-place across pre-allocated, flat arrays without relying on dynamic memory allocation. The pipeline achieves this through independent compute kernels that first identify subpath boundaries and then establish topological pointers (`next_ix`, `prev_ix`) by inspecting adjacent intersections in the sorted arrays.

To optimize the subsequent memory-bound reconstruction passes, this stage introduces two minor architectural overheads. First, it pre-calculates and stores the byte sizes of all geometric memory slices to avoid recomputing them during the dual-pass traversal. Second, it populates the node arrays using specific sorting orders to maximize spatial cache locality—arranging open paths by segment order and closed polygons by the boundary seam, though the alternating nature of polygon routing inherently prevents perfect memory contiguity.

Despite the necessity of multiple dispatches to build and link these flat arrays, the absolute workload remains minimal. The micro-profiling data confirms that graph building acts as a

minor architectural tax on the GPU, scaling from 4.1% to 6.5% on the `110m` datasets down to roughly 1.0% on the `50m` configurations.

Tracing and Reconstruction

The final stage of the pipeline extracts the clipped geometry by traversing the topological graphs. Because graphics hardware cannot dynamically insert or remove memory slices from the original arrays, the GPU must write the resulting clipped geometry into entirely new, pre-allocated output buffers.

To accomplish this without cross-thread data races, the native GPU pipeline must execute a highly serialized, multi-pass sequence:

1. **Memory Reservation:** Threads traverse the graphs to calculate the exact byte length of the resulting geometry and tags, writing these lengths into intermediate buffers. For closed paths, threads must use atomic linked-list binning to group multiple new polygon rings back to their original parent subpath.
2. **Device-Wide Scan:** A Decoupled Fallback parallel prefix sum calculates the absolute global memory offsets for every subpath.
3. **Geometry Writing:** Threads re-traverse the graphs a second time, using the calculated offsets to write the final `LINETO` coordinates and tightly packed 8-bit path tags (via atomic bit-masking) to global VRAM.

In `d3-geo`, this stage scales poorly as geometry density increases. While the single-threaded CPU avoids the complexities of parallel synchronization, dynamically pushing coordinates into standard JavaScript arrays and synthesizing new boundary geometry sequentially incurs a heavy computational penalty. Specifically, the mathematical overhead of interpolating boundary segments to close cut polygons forces the CPU into expensive, sequential trigonometric loops. Across all clipping configurations, the reconstruction phase accounts for between 7.8% and 76.1% of the CPU's frame time, with the heaviest relative costs occurring on the massive `50m` dataset where vast amounts of geometry must be generated and copied to memory.

Similarly, tracing and reconstruction represents the most severe architectural bottleneck in the native GPU pipeline. While the GPU excels at the independent, ALU-bound mathematics of rotation and intersection, it is fundamentally starved by the memory-bound, pointer-chasing nature of graph traversal. The necessity of traversing the graph twice—once to reserve memory and once to write it—forces the streaming multiprocessors to endure the latency of global VRAM reads repeatedly. Furthermore, because writing the tightly packed 8-bit path tags requires atomic sub-word masking to prevent data races, the final write pass incurs additional memory contention.

The introduction of closed polygons in the `110m_poly` dataset exacerbates this bottleneck for both architectures, though it manifests differently. While the CPU stalls on sequential interpolation loops, the GPU exposes severe SIMT (Single Instruction, Multiple Threads) control-flow divergence. When a polygon fully encompasses the clipping domain without intersecting it, the pipeline must explicitly synthesize the entire clipping boundary. Within the `reconstructUnintersected` pass, while most threads execute a fast, contiguous block-copy

of their visible subpath data, the single thread assigned to an encompassing polygon diverges into a heavy loop to interpolate the full boundary (e.g., 180 segments for a full circle) and push tags via the `TagBatcher`. This forces the rest of the workgroup to stall. This divergence is distinctly visible in the profiling data: simply introducing edge-case polygons causes the relative reconstruction cost of the `110m` dataset to surge, jumping from 32.0% to 50.0% under the 180° circular clip, and from 38.5% to 49.0% under the 90° clip.

This memory bandwidth bottleneck scales directly with the volume of processed geometry. When shifting to the denser `50m` dataset—which contains nearly an order of magnitude more line segments—the absolute amount of data that must be read, quantified, synthesized, and written expands drastically. This surge in memory transactions adversely affects both the CPU and the GPU, causing the reconstruction stage to occupy a dominant relative share of the frame time across both implementations, reaching as high as 57.9% in the compute pipeline.

This stark reality highlights the primary algorithmic ceiling of geographic clipping: while massively parallel architectures can virtually eliminate the mathematical cost of complex intersections, both CPU and GPU implementations incur a massive latency penalty when forced to dynamically restructure, interpolate, and write large, irregular datasets back to memory.

Projection and Adaptive Sampling

The final stage of the pipeline transforms the clipped spherical coordinates into 2D Cartesian screen space. Because projecting line segments introduces non-linear curvature, both pipelines must adaptively subdivide segments into piecewise linear approximations to maintain visual fidelity.

In `d3-geo`, this stage relies on a recursive depth-first search (DFS) to evaluate subdivision criteria and generate intermediate points. This algorithmic pattern maps optimally to CPU architectures. The hardware call stack natively manages the recursion, allowing the newly generated coordinates to be evaluated and pushed to the rendering context within a tight, highly optimized L1/L2 cache loop. As a result, projection operates efficiently. On the `110m` datasets, it accounts for 10.1% to 28.7% of the frame time. On the denser `50m` datasets, its relative share drops to between 3.9% and 6.3%, largely because sequential bottlenecks elsewhere in the CPU pipeline expand.

Conversely, the native GPU pipeline must map this dynamic geometry generation to a massively parallel, fixed-memory architecture. Because WGSL lacks native support for recursion, the shader implements the adaptive subdivision as an iterative DFS using a manually managed, function-scoped array stack.

This architectural mismatch introduces a severe performance penalty: thread-local memory spilling. Because GPU registers cannot be dynamically indexed, the compiler is forced to allocate the iterative stack array in thread-local memory, which physically resides in global VRAM. With a maximum subdivision depth of 16, the stack requires hundreds of bytes per thread. When multiplied across thousands of concurrent threads, this footprint vastly

exceeds the Streaming Multiprocessor's (SM) L1 data cache capacity, resulting in massive cache thrashing.

While the shader employs a clever thread-local register cache (`r0`, `r1`) to keep the top two stack frames in fast, statically indexed registers, this only shields shallow branches. The moment a curve requires deeper subdivision to meet the error threshold, the shader falls off a performance cliff as state spills into the thrashed VRAM array.

This spilling penalty is compounded by the strict memory safety requirements of the pipeline. Because the shader cannot dynamically allocate memory for the newly subdivided points, the entire iterative DFS must be executed twice. The GPU endures this local memory thrashing once during the reservation pass merely to count the required leaf points, and then endures it a second time during the write pass to compute and output the final coordinates.

Driven by this thread-local register spilling, L1 cache thrashing, and double-execution, projection and adaptive sampling solidifies as a major bottleneck on the GPU. Across all datasets and clipping configurations, it consumes a substantial 24.8% to 48.4% of the total GPU compute budget.

4.4. Memory Footprint

The memory footprint comparison (Figure 4.5) demonstrates distinct allocation behaviors between the two rendering pipelines. While both systems exhibit memory scaling dependent on input complexity and clipping parameters, the magnitude of this scaling differs significantly. The `d3-geo` baseline maintains a relatively modest system RAM footprint, fluctuating within an envelope of approximately 10 MB to 35 MB across all tested configurations. The GPU pipeline's VRAM allocation scales much more aggressively, peaking at 130.8 MB for the `50m` dataset under a 90° circle clip.

The behavior of the 180° circle clip on the `50m` dataset (40.6 MB) illustrates how the GPU pipeline's memory allocation is directly proportional to the volume of topological intersections. A $180° - \epsilon$ clipping boundary encompasses almost the entire sphere, meaning the effective clipping geometry is reduced to a minuscule antipodal circle of radius ϵ. Consequently, this configuration generates negligible line segment intersections. Because the sizes of dynamic intermediate buffers—such as those used for sorting, metadata tracking, and polygon reconstruction—are calculated based on the total number of intersections rather than strictly the initial input size, the memory footprint remains correspondingly low.

This divergence in memory scaling profiles is rooted in the fundamental architectural differences between the two systems. The `d3-geo` library implements a sequential streaming architecture via its `d3.geoStream` API. It processes GeoJSON features individually, passing each primitive through a pipeline of sequential mathematical transformations (e.g., rotation, projection, clipping, and resampling) before dispatching the result to the Canvas 2D context. While this approach requires retaining local state for the currently active feature—such as intersection coordinate queues for polygon rings—it avoids holding the intermediate geometric data of the entire dataset in memory simultaneously. Consequently, its memory footprint is bounded by the complexity of individual features rather than the global input size, explaining the modest variance.

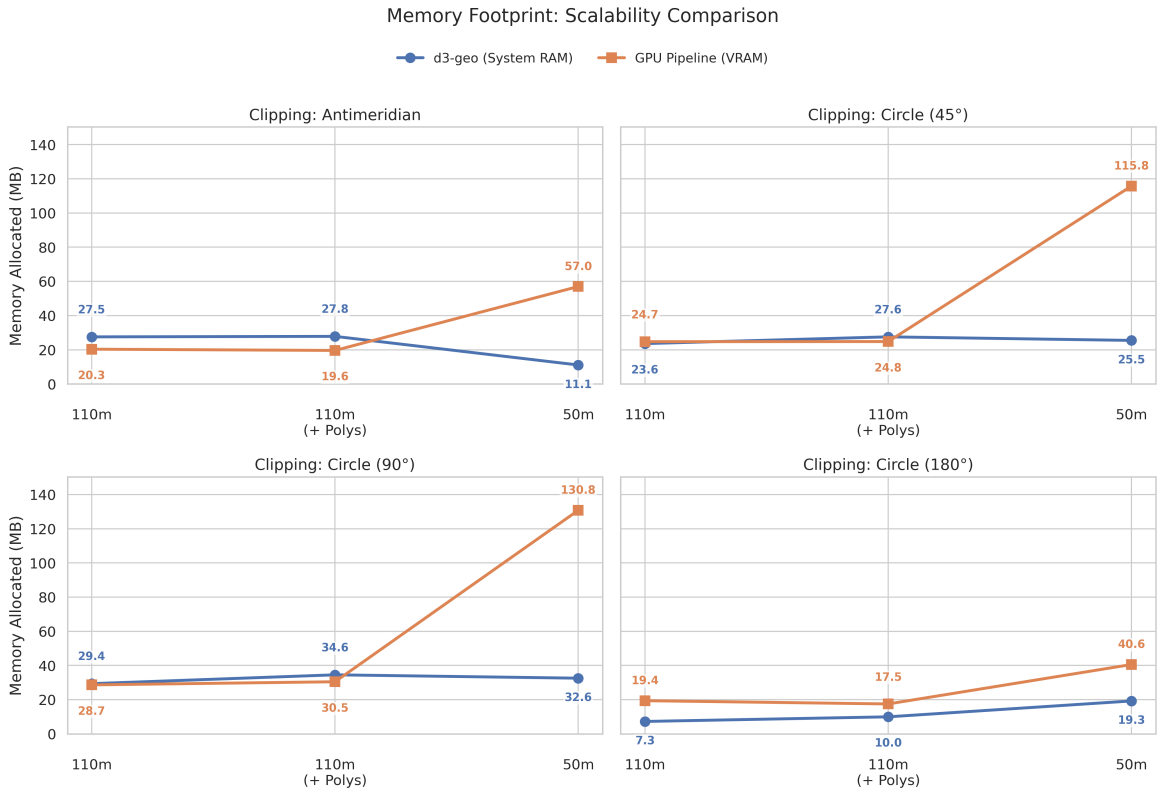


Figure 4.5.: Memory scalability comparison between the `d3-geo` baseline (System RAM) and the GPU Compute Pipeline (VRAM). The charts illustrate memory allocation across varying dataset complexities and clipping configurations.

In contrast, the GPU compute pipeline operates via multi-stage parallel processing. To saturate the hardware's compute units, the pipeline evaluates the geometry globally per frame. It must pre-allocate VRAM buffers capable of simultaneously storing the entire input dataset, alongside all dynamically computed intersections, sorting metrics, and reconstructed vertices. The GPU architecture mandates carrying this global geometric state through the entire pipeline across discrete compute shader dispatch barriers. This fundamentally trades memory efficiency for high-throughput parallel execution, resulting in an allocation profile that scales directly with the total volume of global intersections.

Furthermore, the measured VRAM footprint is amplified by the engine's buffer allocation strategy. To mitigate runtime allocation overhead and prevent severe memory fragmentation, the `WgpuEngine` utilizes a `ResourcePool` that quantizes host-side buffer requests using a log-linear size class allocator. Rather than strictly rounding up to the nearest power of two—a strategy that can yield nearly 50% internal fragmentation when an allocation request marginally exceeds a power-of-two boundary, forcing the allocator to provision a block twice the required size—the allocator uses a 4-bit mantissa. This approach divides the space between consecutive powers of two into 16 uniform sub-intervals, enforcing a minimum baseline alignment of 16 bytes. When dynamically generated geometry exceeds a buffer's current capacity, the allocator provisions a new buffer from the next discrete size class. While this successfully bounds the maximum memory over-provisioning to approximately 6.25% per allocation, this step-function scaling still introduces cumulative VRAM bloat across the pipeline, as intermediate buffers are consistently sized to their upper quantized bounds rather than their exact byte requirements.

Finally, the current GPU implementation does not aggressively pool or reuse intermediate buffers across distinct compute passes within the same frame. It also lacks a global bump allocator, which would allow sub-allocating dynamic data structures from a single, contiguous memory block. Therefore, the measured VRAM values represent an unoptimized upper bound, identifying memory management as a primary architectural target for future pipeline optimization.

5. Discussion

5.1. Limits of Parallelization

Transitioning geographic projection and clipping to WebGPU exposes a strict boundary between theoretical parallelizability and practical hardware constraints. While GPUs provide immense throughput for independent, ALU-bound transformations, they aggressively penalize algorithms reliant on dynamic memory allocation, global synchronization, and pointer-chasing.

Memory Contention and Pointer Chasing

The topological reconstruction of disjoint path segments into closed loops highlights the limitations of GPU memory hierarchies. Graph traversal is inherently sequential; resolving the next vertex requires evaluating the current vertex's pointer. Theoretical parallel graph algorithms, such as flooding or pointer jumping, assume uniform memory access and largely ignore hardware contention. On physical GPUs, traversing spatially scattered intersection nodes generates uncoalesced global VRAM accesses.

SIMT Divergence vs. Dynamic Load Balancing

Adaptive curve subdivision demonstrates a fundamental limit of SIMT (Single Instruction, Multiple Threads) execution. Because threads within a warp execute in lockstep, a single thread evaluating a deeply subdivided curve forces its entire subgroup to stall. While dynamic work-stealing queues could theoretically resolve this ALU underutilization by redistributing sub-segments to idle threads, implementing global task queues requires continuous atomic operations. Accepting SIMT divergence is therefore a necessary, calculated trade-off to prevent memory bandwidth saturation.

Dispatch Overhead and Intentional Serialization

Finally, the high latency of GPU synchronization primitives often renders parallelization counterproductive for sparse workloads. For example, simultaneously reserving relative memory offsets across two distinct global buffers (path data and tags) cannot be done atomically without race conditions. While this could be resolved lock-free using a parallel segmented prefix scan, dispatching device-wide barriers and forcing VRAM round-trips for a handful of intersections introduces massive API overhead. Instead, intentional serialization—assigning a single thread to linearly traverse a localized atomic linked list—bypasses these

heavy parallel primitives entirely. For small, localized datasets, this sequential approach guarantees memory safety while comfortably outperforming theoretical parallel schemes.

5.2. Precision Limits

Modern consumer graphics hardware is fundamentally optimized for 32-bit single-precision (`f32`) and 16-bit half-precision (`f16`) floating-point operations, prioritizing raw throughput for pixel and vertex shading over strict numerical accuracy. While workstation-class GPUs offer robust 64-bit double-precision (`f64`) support, consumer silicon typically executes `f64` instructions at a severely degraded rate (often 1/16th to 1/64th the speed of `f32`), or lacks hardware support entirely. Consequently, WebGPU currently offers no native `f64` types or operations. Translating a geographic pipeline—where CPU ecosystems like JavaScript natively evaluate all numbers as 64-bit floats—into a 32-bit WebGPU environment exposes immediate limits in precision and aggregation capability.

The 32-Bit Floating-Point Ceiling

An IEEE 754 single-precision float allocates only 24 bits to its mantissa, granting approximately 7.2 significant decimal digits of precision. In the context of global mapping, Earth's equatorial circumference is roughly $40,075,000$ meters. Representing a global coordinate in a 32-bit float limits the maximum theoretical spatial resolution to approximately 5 meters.

While this static resolution is sufficient for many visualization tasks, it becomes a severe liability during intermediate geometric calculations. Operations inherent to spherical geometry—such as computing the intersection of two nearly parallel great circles or evaluating Cartesian cross products between short, adjacent line segments—are highly susceptible to catastrophic cancellation. When subtracting two similar `f32` values, the most significant bits cancel out, leaving the result entirely dependent on the lowest mantissa bits, which are often heavily contaminated by prior rounding errors. In a CPU environment, the 53-bit mantissa of an `f64` safely absorbs this noise; on a GPU, `f32` arithmetic natively degrades to zero or arbitrary noise, corrupting subsequent topological logic.

Emulation and Compiler Interference

To circumvent 32-bit or even 64-bit precision ceilings, numerical analysts often employ software-emulated precision techniques. Algorithms such as Shewchuk's adaptive-precision summation, Kahan summation, and float-float (emulated double precision) utilize arrays or pairs of 32-bit floats to capture and retain the truncated bits of standard addition operations.

However, translating these exact-addition algorithms to a shader environment is inherently brittle. Techniques like Knuth's Two-Sum rely heavily on strict execution order and explicit associativity to successfully isolate rounding errors. Because WebGPU acts as an abstraction layer over native graphics APIs (Vulkan, Metal, Direct3D 12), it inherits the aggressive, real-time compiler optimizations of its underlying drivers. By default, shader compilers

frequently apply `fast-math` algebraic transformations, assuming that floating-point addition is associative and commutative—which is mathematically true, but strictly false under IEEE 754 rounding rules. Without explicit and often expensive compiler barriers, the driver will silently reorder instructions, effectively optimizing away the error-recovery algebra and collapsing the emulated precision back into standard `f32` noise.

Furthermore, adaptive-precision algorithms typically demand dynamic memory allocation, exhibit unpredictable memory footprints, and cause significant thread divergence. As previously discussed, these characteristics clash fundamentally with the GPU's highly parallel architecture. Consequently, attempting to enforce CPU-like precision guarantees on consumer GPUs requires trading away the exact compiler optimizations and architectural strengths that make the hardware fast, establishing a rigid ceiling on the numerical robustness of parallel geographic algorithms.

Topological Sorting Errors and Broken Graphs

A core limitation of parallel clipping without double precision is handling nearly coincident intersections. When multiple segments cross the boundary at microscopically close locations, floating-point imprecision can scramble their sorting order. Even with the robustness techniques described in Section 3.6.2, the pipeline's topological sorting logic is vulnerable to catastrophic cancellation, especially on high-density geometry.

The resulting interleaving corrupts the graph. The `linkAlongSeam` pass relies on strictly alternating `EXIT` and `ENTRY` nodes; consecutive nodes of the same type create invalid subcycles or infinite loops. Traversing these during reconstruction would permanently hang the GPU. To prevent device crashes, the kernels enforce a strict safety break:

```
loop_iterations += 1u;
if (loop_iterations > num_intersections_in_path) { break; }
```

This breaks the infinite loop but forces the pipeline to output truncated, malformed path geometry. The downstream Vello renderer receives this garbage data and attempts to mathematically resolve the unclosed or self-intersecting paths. While Vello is robust enough to eventually render them without crashing, processing these highly degenerate paths stalls its optimized rendering loops. In addition to the resulting rendering artifacts (see Figure 5.1), this often causes severe frame-time spikes during affected frames.

The prevalence of these edge cases in high-resolution datasets presents a significant hurdle to the pipeline's practical viability, particularly for the large-scale cartographic tasks that would otherwise benefit most from GPU acceleration.

Catastrophic Cancellation in the PIP Test

As established, globally aggregating spherical excess across large, highly complex polygons using 32-bit floats introduces unavoidable numerical drift. In the Point-in-Polygon (PIP) stage, the initial containment state relies on evaluating whether a polygon's accumulated area is negative to determine its topological winding order.

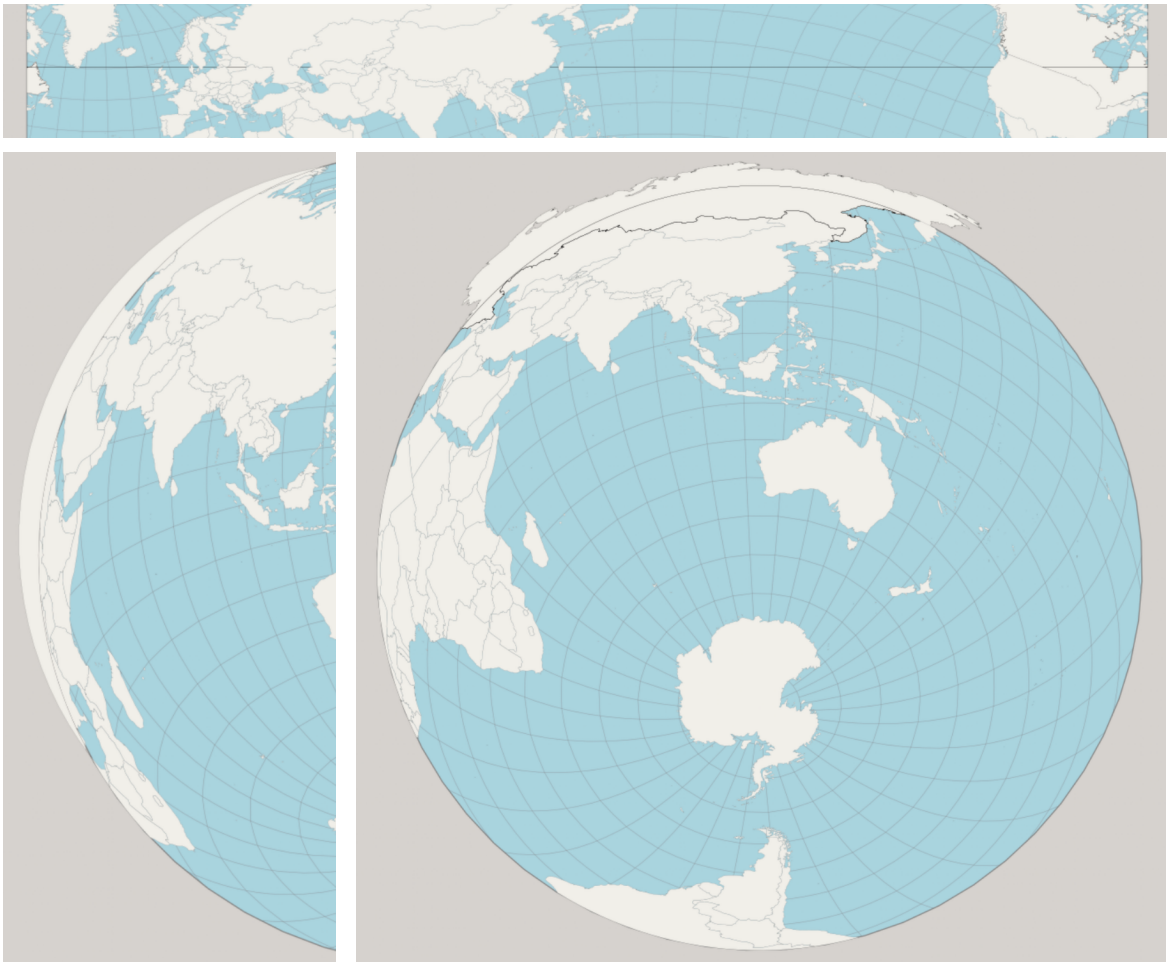
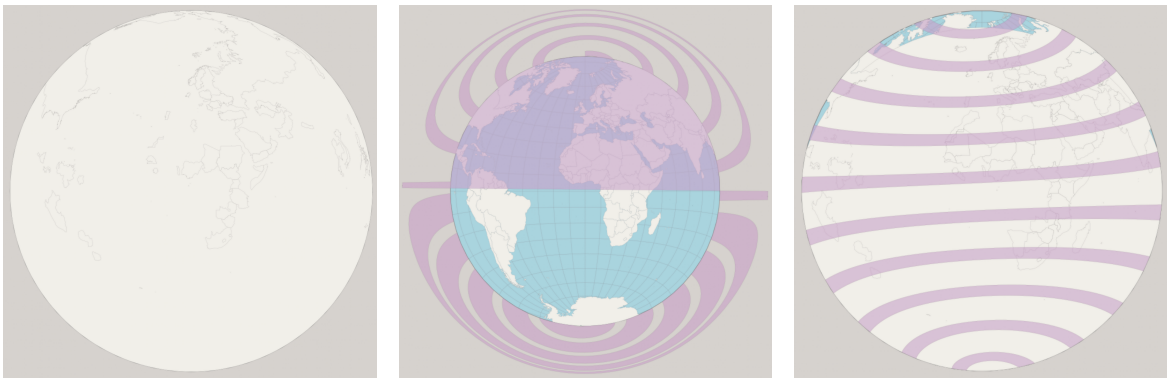


Figure 5.1.: Visual anomalies resulting from topological sorting errors. When micro-collisions corrupt the intersection graph, the traversal becomes trapped in invalid sub-cycles, repeatedly constructing the same malformed geometry until halted by the safety heuristic. Certain configurations lead to local coordinate jumps similar to not clipping at all. Others lead to the downstream renderer attempting to mathematically resolve these unclosed or self-intersecting paths, which leads to graphical artifacs as well as micro-stutter.

While the PIP stage's two-pass map-reduce architecture and Kahan summation drastically reduce this error compared to naive accumulation, they do not eliminate it entirely. In extreme edge cases, catastrophic cancellation during the accumulation of thousands of microscopic trigonometric values can still overcome the pipeline's dynamic $\sqrt{N}$ error bounds. If this residual geometric noise exceeds the dynamically scaled deadband, the parity check flips, permanently corrupting the pipeline's topological interpretation of that specific geometry.

Because the downstream renderer relies on this initial parity to determine the "inside" of a clipped polygon, an inverted winding order compromises the boundary synthesis logic. A primary manifestation of this error occurs when a single, fully visible polygon is falsely evaluated as enclosing the entire clipping domain. The pipeline reacts by synthesizing the full clipping boundary as an outer ring for this specific polygon. The rendering backend then fills the entire domain with the polygon's color, leaving its actual geometry punched out as a hole. Because all other geographic polygons render correctly on top of this erroneous global fill using the same color, they visually blend into the background, leaving only their boundaries visible as negative-space strokes (see Figure 5.2a).



(a) A single inverted polygon draws the full boundary, acting as a solid background that reduces correctly rendered polygons to outlines.

(b) The corrupted winding order of intersected geometry generates arcs that escape the valid projection bounds.

(c) The inverted topology triggers the synthesis of an incorrect geometric bound, erroneously filling a massive portion of the globe.

Figure 5.2.: Visual consequences of PIP parity inversion. Accumulated floating-point drift corrupts the initial topological winding order of subject polygons. Depending on the clip configuration and intersection state, this triggers the synthesis of incorrect geometric bounds and results in severe rendering artifacts.

Rather than presenting a single consistent error mode, this topological collapse is highly unpredictable. In different clip configurations, or when intersected polygons suffer parity inversion, the pipeline generates entirely different structures. These range from erroneous fills that escape the valid projection boundaries entirely (Figure 5.2b) to broad, partial domain fills that obscure large regions of the globe beneath an incorrect geometric arc (Figure 5.2c).

Because floating-point drift scales directly with the number of processed vertices, high-resolution geographic datasets featuring detailed coastlines are inherently more susceptible to these parity inversions. This establishes a practical limitation, as the dense workloads that

stand to benefit the most from GPU acceleration are simultaneously the most vulnerable to catastrophic cancellation.

5.3. Limits of Clipping Composition

A notable algorithmic limitation of the current pipeline involves the application of mutually exclusive clipping boundaries. Projections that map the global longitudinal domain to a finite Cartesian width (such as the Equirectangular projection) inherently require antimeridian cutting. If a line segment crosses the date line without being explicitly severed, the projection math will map these vertices to opposite ends of the Cartesian domain, resulting in a massive, erroneous line tearing horizontally across the map.

Currently, the parallel pipeline supports single-domain clipping: geometry is checked against either the antimeridian *or* a small-circle boundary. If a user applies a wide circle clip (e.g., a clip angle $> 90°$) to a cylindrical projection, the circle clipping logic completely overrides the antimeridian cut. Consequently, any geometry crossing the antimeridian that falls within the valid clip circle remains uncut, triggering the wrap-around tearing artifact upon projection (see Figure 5.3).

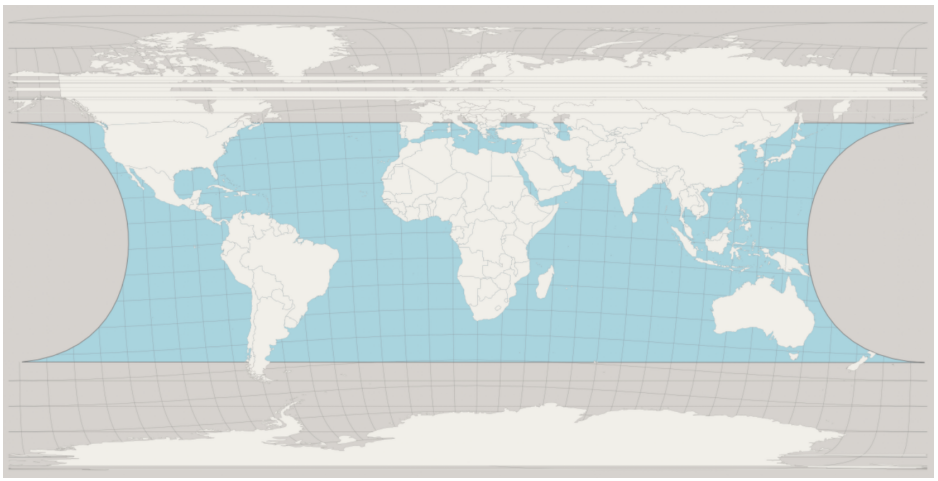


Figure 5.3.: Wrap-around artifacts caused by mutually exclusive clipping. Applying a large circle clip to an Equirectangular projection overrides the necessary antimeridian cut. Geometry crossing the date line within the valid circle domain remains uncut, wrapping across the entire Cartesian space.

It is important to note that this is not exclusively a limitation of our parallel architecture, but a fundamental topological challenge shared by `d3-geo`. In `d3-geo`, clipping operations cannot be natively composited or combined. Configuring a cylindrical projection with a custom `clipAngle` forcefully replaces its default antimeridian stream with the circle clip, resulting in the exact same visual tearing artifacts.

Resolving this mathematically requires true composite spherical clipping—subjecting the geometry to the antimeridian cut, and subsequently clipping the resulting segments against

the small circle. Implementing this in a single parallel pass represents a significant architectural challenge, as it requires handling multiple overlapping boundary synthesis states, tracking complex inside/outside parities for intersecting domains, and sorting multi-boundary intersections without introducing further topological corruption.

6. Conclusion & Future Work

6.1. Conclusion

The primary objective of this thesis was to design, implement, and evaluate a massively parallel geographic projection and polygon clipping pipeline utilizing WebGPU. By transitioning these workloads from traditional sequential CPU environments to a GPU compute architecture, we sought to determine if the strict data dependencies of geographic topology could be efficiently mapped to a SIMT execution model.

Our benchmarking demonstrates that the compute-centric pipeline achieves massive performance improvements in raw processing throughput, frequently outperforming `d3-geo` by an order of magnitude on dense cartographic datasets. However, the evaluation also clearly establishes that the current iteration of the pipeline is not yet robust enough for serious GIS, or even web mapping and navigation tasks.

This is a direct consequence of API and hardware precision limits. While sequential CPU libraries can guarantee topological safety by leveraging 64-bit floating-point precision (`f64`) and Shewchuk's exact adaptive-precision summation, our WebGPU implementation is bound to 32-bit floats (`f32`). The necessary compromises prove mathematically insufficient for massive global geometries. This lack of exact arithmetic directly triggers the two critical failure modes observed in our evaluation: micro-collisions that scramble intersection sorting to break traversal graphs, and accumulation drift that inverts Point-in-Polygon (PIP) parity to cause massive erroneous fills.

Despite these mathematical stability limits, the architectural foundation has proven highly successful. The sheer magnitude of the observed speedups validates the multi-pass compute paradigm, confirming that GPU acceleration for dynamic vector mapping is a highly promising avenue that warrants continued research.

6.2. Future Work

While the pipeline demonstrates high-performance projection, several immediate and long-term pathways for improvement remain. The absolute priority is resolving numerical instability and topological breakdowns caused by floating-point drift. Beyond these critical fixes, we propose a series of architectural refinements to optimize memory usage and expand the pipeline's cartographic capabilities.

Precision Emulation for Intersections and PIP Testing

To definitively eliminate the remaining micro-collisions and parity inversions caused by the hardware limitations of 32-bit floating-point arithmetic, future iterations must explore software-emulated precision within the compute kernels.

For Cartesian intersection calculations, introducing *float-float* arithmetic—which utilizes pairs of 32-bit floats to capture and retain truncated bits—yields approximately 48 bits of mantissa precision. Applying this extended arithmetic to the intersection modules would minimize the rounding errors that cause tightly clustered vertices to collapse into identical spatial buckets. This guarantees topologically robust graph sorting without relying on rigid spatial quantization heuristics.

Similarly, for the Point-in-Polygon (PIP) evaluation, future research should investigate advanced exact summation techniques to improve upon the current Kahan summation. While a truly exact summation using dynamically sized arrays remains incompatible with GPU memory architectures, a restricted form of Shewchuk's algorithm utilizing a fixed number of `f32` accumulators could achieve significantly higher precision. This would proactively absorb the floating-point drift that currently leads to topological parity inversions when processing massive global geometries.

Because these intersection and accumulation computations represent a very small fraction of the overall GPU execution time, the pipeline can likely absorb the significant instruction overhead of emulated precision. The primary engineering challenge for this architectural upgrade will be successfully navigating the aggressive compiler optimizations (as discussed in Section 5.2) that frequently break exact-addition algebra on consumer graphics hardware.

Hybrid CPU-GPU PIP Testing

As an alternative approach to solving catastrophic PIP parity inversions, a hybrid architecture should be explored. Rather than accumulating spherical excess on the GPU using lossy 32-bit atomics, the initial containment state could be evaluated on the CPU utilizing Shewchuk's robust summation. To avoid the overhead of rotating massive coordinate arrays on the host, the CPU could maintain spherical bounding boxes or bounding circles for the unrotated polygons. By applying an inverse rotation to the projection's reference point, the CPU can execute a mathematically exact PIP test against these bounds and asynchronously upload the parity flags to the GPU before reconstruction begins.

VRAM Heuristics and Direct Render Integration

To reduce the severe latency penalty of CPU-GPU synchronization, future iterations should minimize host-side memory management. By monitoring memory usage over an initial set of frames, the pipeline could "freeze" its VRAM allocations at a safe heuristic multiplier (e.g., 2× the maximum observed capacity) and only trigger a reset when the source geometry explicitly changes. This would allow the CPU to stop synchronously reading back the bump allocator buffer, switching to asynchronous readbacks or completely headless GPU execution. Ultimately, discarding intermediate buffers and piping the reconstructed geometry directly

into Vello would drastically reduce memory bandwidth, though this requires extending the Vello architecture to accept scene encodings directly from device memory.

Workgroup-Local Fast Paths

For geographic features that are small enough to be fully contained within a single compute workgroup's execution block, the pipeline could introduce an optimized fast path. As observed during the clipping phase, all intersections were already sorted in their initial blocks and did not actually require global merge passes. By collapsing multi-pass pipeline stages into a single megakernel for small datasets and relying purely on fast workgroup barriers for synchronization, API dispatch overhead and global VRAM round-trips could be significantly curtailed.

Checkpointing and Smart Retry Logic

The current fallback mechanism forces a complete re-execution of the entire compute pipeline if a downstream memory reservation fails and triggers a reallocation. Future implementations should adopt a smarter retry loop utilizing intermediate checkpointing. By caching the state of successfully completed early kernels—such as initial projection or intersection discovery—a retry triggered by an out-of-memory error during the final reconstruction phase would only need to reallocate buffers and re-run the specifically affected downstream stages. While the memory footprint stabilized quickly in the benchmark, highly dynamic high-resolution scenes could profit from this greatly.

Unified Bump Allocation Strategy

To optimize both VRAM consumption and CPU-side dispatch overhead, future iterations should transition from managing numerous discrete WebGPU buffers to a unified bump allocation strategy. By allocating a single monolithic global memory arena, the host can utilize a lightweight configuration buffer to dynamically supply byte-offsets to the compute shaders, effectively sub-allocating the arena into logical slices for each pipeline stage. This approach significantly reduces the API overhead associated with constant buffer creation and complex bind group management. Furthermore, it enables explicit memory aliasing: transient data slices required only during early passes—such as intersection sorting keys or temporary graph structures—can be safely reused and overwritten by the final path geometry during the reconstruction phase, drastically lowering the pipeline's peak memory footprint.

Cross-Platform and Architectural Evaluation

The current performance evaluation establishes a baseline on a discrete desktop GPU and an x86-64 CPU. However, because WebGPU is explicitly designed as a ubiquitous, cross-platform API, validating the pipeline's scalability requires extensive benchmarking across a much wider array of hardware architectures. Future work must evaluate the compute kernels on divergent discrete GPU architectures—such as AMD RDNA and Intel Arc—to identify vendor-specific

bottlenecks, warp/wavefront execution variances, and register allocation limits. Furthermore, the pipeline's memory bandwidth constraints must be tested on integrated graphics (iGPUs) and unified memory architectures, such as Apple Silicon, where traditional host-to-device transfer penalties are fundamentally altered. Evaluating the host-side orchestration across different CPU Instruction Set Architectures (ISAs), particularly ARM64, is equally critical. Ultimately, to prove the pipeline's viability for consumer-facing web mapping, profiling must be extended to mobile devices and low-power hardware, where limited compute unit counts, aggressive thermal throttling, and constrained memory pools will rigorously test the architecture's efficiency.

Expanded Cartographic Features

To reach feature parity with established CPU libraries like `d3-geo`, the pipeline should expand its cartographic toolset. Future iterations could implement:

- **Arbitrary Spherical Clipping:** Extending the intersection logic beyond simple antimeridian and small-circle clipping to support clipping against arbitrary spherical polygons.
- **Post-Projection Clipping:** Adding a secondary Cartesian pass (`PostClip`) to cull geometry falling outside the 2D viewport after spherical projection.
- **Dynamic Vector Tile Projection:** Standard web mapping relies on static, pre-clipped vector tiles optimized for fixed Cartesian projections. Applying dynamic spherical rotation fundamentally breaks this paradigm, as geographic features constantly migrate across tile boundaries depending on the projection's center. Future research must address the open challenge of dynamically projecting, rotating, and re-tiling vector data in real-time. This requires investigating the optimal split between client-side and server-side processing—drawing inspiration from dynamic indexers like GeoJSON-VT or Tippecanoe to generate ephemeral tiles on the fly. Solving this necessitates novel spatial indexing, real-time tile boundary clipping on the GPU, and efficient caching strategies for continuously shifting global geometry.
- **Advanced CRS and Datums:** Once high-precision emulated floats are implemented, the pipeline can safely support complex Coordinate Reference Systems (CRS) and datum transformations, which require strict numerical fidelity to maintain geographic accuracy.

Geometric Refinement and Dynamic LOD

Finally, the generated geometry can be heavily optimized before being handed off to the 2D renderer. The current reliance on dense, piecewise linear segments for projection curvature is memory-intensive. This could be improved by outputting adaptive Bezier curves, or by utilizing the mathematical derivative of the projection function (if available) to inform a much more precise and sparse curvature approximation.

Building on this, future work could investigate offloading line simplification to the GPU. Implementing algorithms such as Ramer–Douglas–Peucker or Visvalingam–Whyatt within

compute shaders could enable high-throughput, dynamic Level of Detail (LOD). This would further reduce the data payload sent to the rendering backend without sacrificing visual fidelity, though the practical efficiency of a GPU-based implementation remains to be verified.

A. Profiling Stage Assignments

As discussed in Chapter 4, comparing the highly parallel compute shader pipeline with the single-threaded, sequential execution of `d3-geo` requires grouping operations into shared, high-level logical stages.

Tables A.1 and A.2 detail the specific assignment of GPU compute passes and CPU function calls to these logical operations, respectively. Because the architectural paradigms differ significantly, some `d3-geo` functions inherently overlap across conceptual boundaries. Note that `d3-geo`'s initial GeoJSON parsing and stream dispatch overhead has been excluded from these mappings, as it represents pipeline-setup overhead on the CPU without a direct mathematical equivalent in the GPU compute sequence.

Logical Stage	**GPU Passes**
Totals (Macro)	`project`, `render_to_texture`
1. Rotation	`rotate`
2. Intersection	`intersect`
3. Sort	`blockSort`, `buildKeys`
4. Point-in-Polygon	`polygonContains`
5. Graph Building	`linkAlongPathdata`, `linkAlongSeam`, `markBlockBounds`
6. Tracing & Reconstruction	`checkOutputBufferSizes`, `exclusiveScan`, `reconstructIntersected`, `reconstructUnintersected`, `reserveIntersected`, `reserveUnintersected`, `resolveIntersectedMemReqs`
7. Projection & Adaptive Sampling	`computeSubpathInfo`, `exclusiveScanWithTotal`, `main`, `resampleCountEntry`, `resampleWriteEntry`, `setupIndirectDispatchProjection`, `setupIndirectDispatchProjectionWrite`, `setupIndirectDispatchSubpathInfoProjection`
Rendering	`vello.*`

Table A.1.: Mapping of discrete GPU compute passes to logical pipeline stages.

Logical Stage	JavaScript Functions (`d3-geo`)
Totals (Macro)	*Aggregated projection vs. rendering*
1. Rotation	`rotation`, `rotateRadians`, `projection2.rotate`, `scaleTranslateRotate`
2. Intersection	`intersect`, `Intersection`, `clipAntimeridianIntersect`, `clipAntimeridianInterpolate`, `clipLine`, `code`[1], `visible`[2], `point`[3], `pointLine`
3. Sort	`compareIntersection`
4. Point-in-Polygon	`polygonContains_default`
5. Graph Building	`link`[4], `merge`[5]
6. Tracing & Reconstruction	`rejoin_default`, `validSegment`, `clean`[6], `buffer_default`, `pointEqual_default`, `polygonEnd`[7], `ringEnd`[8], `pointRing`, `polygonStart`, `ringStart`, `lineStart`, `lineEnd`, `moveTo`, `lineTo`, `closePath`
7. Projection & Adaptive Sampling	`resampleLineTo`, `interpolate`, `linePoint2`, `equirectangularRaw`, `transform`, `cartesian`, `spherical`[9], `recenter`[10]
Rendering	*– (Opaque browser rasterization)*[11]

Table A.2.: Mapping of sampled `d3-geo` CPU function calls to logical pipeline stages.

[1] `code` generates a 4-bit outcode vector for fast bounding-box intersection culling during small-circle clipping.
[2] `visible` evaluates the coordinate's angular distance to determine if it lies within the visible, clipped hemisphere.
[3] `point` is intercepted in `clip/index.js` to run `pointVisible` checks.
[4] `link` constructs a doubly-linked list (`n` and `p` pointers) of intersection vertices used to traverse and trace the rejoined polygon.
[5] `merge` flattens isolated line segments collected during clipping so they can be fed into the graph reconstruction loop (`clipRejoin`).
[6] `clean` evaluates a bitmask to determine if line segments intersected the clip edge and if the first and last segments must be reconnected.
[7] `polygonEnd` triggers `polygonContains` and `clipRejoin` to reconstruct the final shape.
[8] `ringEnd` evaluates ring intersections and filters valid segments for the graph.
[9] `cartesian` and `spherical` are assigned here as their execution is overwhelmingly dominated by `resampleLineTo`, which projects coordinates to 3D space to find geometric midpoints for adaptive subdivision.
[10] `recenter` recalculates the projection's internal transformation matrices, rotation closures, and resampling streams whenever projection properties change.
[11] Canvas 2D rasterization occurs outside the JS call stack (in C++) and cannot be profiled via standard JavaScript sampling intervals.

List of Figures

Bibliography

[APP23] Ashan, M. K. B. ; Puri, Satish ; Prasad, Sushil K.: Efficient PRAM and Practical GPU Algorithms for Large Polygon Clipping with Degenerate Cases. In: *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2023, S. 579–591

[Bat17] Battersby, Sarah: Map Projections. Version: 2nd Quarter 2017, 2017. http://dx.doi.org/10.22224/gistbok/2017.2.7. In: Wilson, John P. (Hrsg.): *The Geographic Information Science & Technology Body of Knowledge*. 2nd Quarter 2017. University Consortium for Geographic Information Science (UCGIS), 2017. – DOI 10.22224/gistbok/2017.2.7

[BDD+16] Butler, H. ; Daly, M. ; Doyle, A. ; Gillies, S. ; Hagen, S. ; Schaub, T.: The GeoJSON Format / Internet Engineering Task Force. Version: August 2016. http://dx.doi.org/10.17487/RFC7946. RFC Editor, August 2016 (7946). – RFC

[BFUY14] Battersby, Sarah E. ; Finn, Michael P. ; Usery, E. L. ; Yamamoto, Kristina H.: Implications of Web Mercator and its Use in Online Mapping. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 49 (2014), Nr. 2, S. 94–109. http://dx.doi.org/10.3138/carto.49.2.2313. – DOI 10.3138/carto.49.2.2313

[Ble89] Blelloch, Guy E.: Scans as primitive parallel operations. In: *IEEE Transactions on Computers* 38 (1989), Nr. 11, S. 1526–1538. http://dx.doi.org/10.1109/12.42122. – DOI 10.1109/12.42122

[DP73] Douglas, David H. ; Peucker, Thomas K.: Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. In: *The Canadian Cartographer* 10 (1973), Nr. 2, S. 112–122. http://dx.doi.org/10.3138/FM57-6770-U75U-7727. – DOI 10.3138/FM57–6770–U75U–7727

[Env98] Environmental Systems Research Institute: ESRI Shapefile Technical Description / ESRI. Redlands, CA, USA, July 1998. – White Paper

[Env00] Environmental Systems Research Institute (Esri) ; Esri (Hrsg.): *Understanding Map Projections*. Redlands, CA: Esri, 2000. – Available at: https://content.esri.com/support/documentation/ao_/710understanding_map_projections.pdf

[GH98] Greiner, Günther ; Hormann, Kai: Efficient Clipping of Arbitrary Polygons. In: *ACM Trans. Graph.* 17 (1998), Nr. 2, S. 71–83. http://dx.doi.org/10.1145/274363.274364. – DOI 10.1145/274363.274364

[GLFN14] Ganacim, Francisco ; Lima, Rodolfo S. ; Figueiredo, Luiz H. ; Nehab, Diego: Massively-parallel vector graphics. In: *ACM Trans. Graph.* 33 (2014), November, Nr. 6. http://dx.doi.org/10.1145/2661229.2661274. – DOI 10.1145/2661229.2661274. – ISSN 0730–0301

[HDM+14] Hughes, John F. ; Dam, Andries van ; McGuire, Morgan ; Sklar, David F. ; Foley, James D. ; Feiner, Steven K. ; Akeley, Kurt: *Computer Graphics: Principles and Practice*. 3rd. Addison-Wesley Professional, 2014. – ISBN 978–0321399526

[HP17] Hennessy, John L. ; Patterson, David A.: *Computer Architecture: A Quantitative Approach*. 6th. Morgan Kaufmann, 2017. – ISBN 978–0128119051

[Jen12] Jenny, B.: Adaptive Composite Map Projections. In: *IEEE Transactions on Visualization and Computer Graphics* 18 (2012), dec, Nr. 12, S. 2575–2582. http://dx.doi.org/10.1109/TVCG.2012.192. – DOI 10.1109/TVCG.2012.192. – ISSN 1077–2626

[JNC25] Ji, Ruian ; Niu, Zhirui ; Chen, Lan: GPU-Accelerated Algorithm for Polygon Reconstruction. In: *Applied Sciences* 15 (2025), Nr. 3. http://dx.doi.org/10.3390/app15031111. – DOI 10.3390/app15031111. – ISSN 2076–3417

[Joy15] Joyrexus: *Greiner-Hormann polygon clipping algorithm implementation*. GitHub Gist. https://gist.github.com/joyrexus/eb7248ad5e3612d4447a. Version: 2015. – Accessed: February 25, 2026

[JŠ13] Jenny, Bernhard ; Šavrič, Bojan: Rendering Vector Geometry with Adaptive Composite Map Projections. In: *Proceedings of the 26th International Cartographic Conference*. Dresden, Germany, August 2013. – Extended Abstract

[JŠ18] Jenny, Bernhard ; Šavrič, Bojan: Enhancing adaptive composite map projections: Wagner transformation between the Lambert azimuthal and the transverse cylindrical equal-area projections. In: *Cartography and Geographic Information Science* 45 (2018), Nr. 5, 456–463. http://dx.doi.org/10.1080/15230406.2017.1379036. – DOI 10.1080/15230406.2017.1379036

[Kil20] Kilgard, Mark J.: Polar stroking: new theory and methods for stroking paths. In: *ACM Trans. Graph.* 39 (2020), August, Nr. 4. http://dx.doi.org/10.1145/3386569.3392458. – DOI 10.1145/3386569.3392458. – ISSN 0730–0301

[Knu97] Knuth, Donald E.: *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd. Reading, MA : Addison-Wesley Professional, 1997

[KP25] Kim, Jaehun ; Park, Jason: *Accelerating ZK Proving with WebGPU: Techniques and Challenges*. zkSecurity Blog. https://blog.zksecurity.xyz/posts/webgpu/. Version: April 2025. – Accessed: February 25, 2026

[LB05] Loop, Charles ; Blinn, Jim: Resolution independent curve rendering using programmable graphics hardware. In: *ACM SIGGRAPH 2005 Papers*. New York, NY, USA : Association for Computing Machinery, 2005 (SIGGRAPH '05). – ISBN 9781450378253, 1000–1009

[LBS26] Laxström, Niklas ; Bah, Tavmjong ; Schulze, Dirk: *SVG Strokes*. W3C Working Draft. https://www.w3.org/TR/svg-strokes/. Version: feb 2026. – Accessed: 2026-02-25

[LHZ16] Li, Rui ; Hou, Qiming ; Zhou, Kun: Efficient GPU path rendering using scanline rasterization. In: *ACM Trans. Graph.* 35 (2016), Dezember, Nr. 6. http://dx.doi.org/10.1145/2980179.2982434. – DOI 10.1145/2980179.2982434. – ISSN 0730–0301

[LU24] Levien, Raph ; Uguray, Arman: GPU-friendly Stroke Expansion. In: *Proc. ACM Comput. Graph. Interact. Tech.* 7 (2024), August, Nr. 3. http://dx.doi.org/10.1145/3675390. – DOI 10.1145/3675390

[Map16] Mapbox: *Mapbox Vector Tile Specification*. https://github.com/mapbox/vector-tile-spec, January 2016. – Version 2.1. Accessed: 2025-09-08

[MG16] Merrill, Duane ; Garland, Michael: Single-pass Parallel Prefix Scan with Decoupled Look-back / NVIDIA Research. Version: 2016. https://research.nvidia.com/publication/2016-03_single-pass-parallel-prefix-scan-decoupled-look-back. 2016 (NVR-2016-001). – Forschungsbericht

[Nat06] National Council of Educational Research and Training: Map Projections. In: *Practical Work in Geography, Part I*. NCERT, 2006, Kapitel 4. – Textbook for Class XI.

[Neh20a] Nehab, Diego: *2D Computer Graphics*. Lecture Slides, Instituto de Matemática Pura e Aplicada (IMPA). https://w3.impa.br/~diego/teaching/vg/. Version: 2020. – Accessed: 2025-09-08

[Neh20b] Nehab, Diego: Converting stroked primitives to filled primitives. In: *ACM Trans. Graph.* 39 (2020), August, Nr. 4. http://dx.doi.org/10.1145/3386569.3392392. – DOI 10.1145/3386569.3392392. – ISSN 0730–0301

[NH08] Nehab, Diego ; Hoppe, Hugues: Random-access rendering of general vector graphics. In: *ACM Trans. Graph.* 27 (2008), Dezember, Nr. 5. http://dx.doi.org/10.1145/1409060.1409088. – DOI 10.1145/1409060.1409088. – ISSN 0730–0301

[OGP08] OGP Geomatics Committee: *EPSG Geodetic Parameter Dataset: Coordinate Reference System 3857*. https://epsg.org/crs_3857/WGS-84-Pseudo-Mercator.html, 2008. – Accessed: 2026-03-10

[Ope11] Open Geospatial Consortium ; Herring, John R. (Hrsg.): OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture / Open Geospatial Consortium Inc. Version: may 2011. http://www.opengis.net/doc/is/sfa/1.2.1. 2011 (OGC 06-103r4). – OpenGIS Implementation Standard. – Version 1.2.1

[Pan14] Panigrahi, Narayan: *Computing in Geographic Information Systems*. Boca Raton, FL, USA : CRC Press, 2014. http://dx.doi.org/10.1201/b17147. http://dx.doi.org/10.1201/b17147

[PP15] Puri, Satish ; Prasad, Sushil K.: A parallel algorithm for clipping polygons with improved bounds and a distributed overlay processing system using MPI. In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, IEEE Press, 2015 (CCGRID '15). – ISBN 9781479980062, 576–585

[Ram72] Ramer, Urs: An iterative procedure for the polygonal approximation of plane curves. In: *Computer Graphics and Image Processing* 1 (1972), Nr. 3, 244-256. http://dx.doi.org/https://doi.org/10.1016/S0146-664X(72)80017-0. – DOI https://doi.org/10.1016/S0146–664X(72)80017–0. – ISSN 0146–664X

[She97] Shewchuk, Jonathan R.: Adaptive precision floating-point arithmetic and fast robust geometric predicates. In: *Discrete & Computational Geometry* 18 (1997), Nr. 3, S. 305–363. http://dx.doi.org/10.1007/PL00009321. – DOI 10.1007/PL00009321

[SLO25] Smith, Thomas ; Levien, Raph ; Owens, John D.: Decoupled Fallback: A Portable Single-Pass GPU Scan. In: *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures*, ACM, 2025 (SPAA '25), S. 255–268

[Sny87] Snyder, John P.: *Map Projections: A Working Manual*. Washington, D.C. : U.S. Government Printing Office, 1987 (U.S. Geological Survey Professional Paper 1395). http://dx.doi.org/10.3133/pp1395. http://dx.doi.org/10.3133/pp1395

[Sta25] Stampfl, Laurenz: *High-performance 2D graphics rendering on the CPU using sparse strips*. Zurich, Switzerland, ETH Zurich, Master's Thesis, October 2025. – Supervised by Dr. Raph Levien and Prof. Dr. Ralf Jung at the Programming Language Foundations Lab

[Str18] Strebe, Daniel "daan": An efficient technique for creating a continuum of equal-area map projections. In: *Cartography and Geographic Information Science* 45 (2018), Nr. 6, 529–538. http://dx.doi.org/10.1080/15230406.2017.1405285. – DOI 10.1080/15230406.2017.1405285

[SVG18] SVG Working Group: Scalable Vector Graphics (SVG) 2 / W3C. Version: October 2018. https://www.w3.org/TR/SVG2/. Cambridge, MA, USA, October 2018. – Candidate Recommendation

[Ven22] Veness, Chris: *Calculate distance and bearing between two Latitude/Longitude points using haversine formula in JavaScript*. https://movable-type.co.uk/scripts/latlong.html. Version: 2022

[VW93] Visvalingam, M. ; Whyatt, J. D.: Line generalisation by repeated elimination of points. In: *The Cartographic Journal* 30 (1993), Nr. 1, S. 46–51. http://dx.doi.org/10.1179/000870493786962263. – DOI 10.1179/000870493786962263

[Whi19] Whittlesey, Marshall A.: *Spherical Geometry and Its Applications*. Boca Raton, FL : CRC Press, 2019 (Textbooks in Mathematics). – ISBN 9780367196905

[WHKH22] W. Hwu, Wen mei ; Kirk, David B. ; Hajj, Izzat E.: *Programming Massively Parallel Processors: A Hands-on Approach*. 4th. Morgan Kaufmann, 2022. – ISBN 978–0323912310