

Robust Planar Maps for 3D Vectorization

ROBERT FUCHS, Carnegie Mellon University, USA
KEENAN CRANE, Carnegie Mellon University, USA

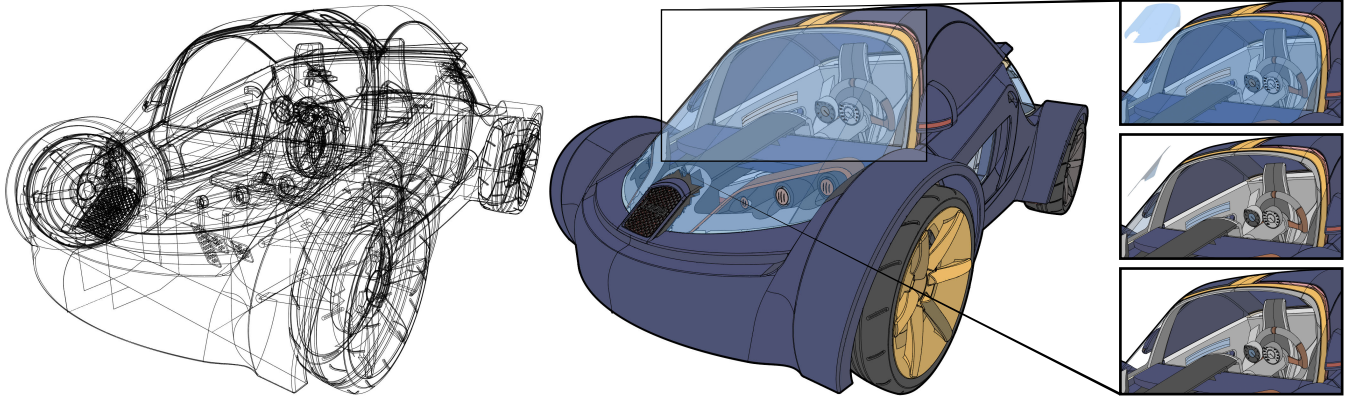


Fig. 1. Vectorization of 3D objects represents an extreme case for 2D planar arrangements, due to numerous near-degeneracies arising from visibility events. Here we convert a rat's nest of projected NURBS curves (*left*) into well-defined planar regions that can be filled with solid colors (*center*)—a task nearly impossible for previous 2D arrangement algorithms. Unlike rasterization-based methods, we also produce artwork with layered transparency—here retaining the windshield (*top right*) and side window (*middle right*) as whole, editable shapes that have not been diced into smaller pieces.

Note: here and throughout, any artifacts in the thin black curves indicate errors in the input—not errors produced by our planar map extraction.

This paper describes a strategy for vectorizing 3D scenes with proper occlusion. Given a collection of curves derived from 3D geometry (silhouettes, isolines, material boundaries, etc.), we produce a 2D vector image that partitions the image plane into a *planar map* of solid shaded regions. The method is agnostic to surface representation, handling for instance curves obtained from polygonal, NURBS or subdivision surfaces. The output likewise supports curve segments of arbitrary parametric type. Our key observation is that the spatial hierarchy used to accelerate curve-curve intersections provides the fundamental representation of the planar map itself. This approach provides geometric flexibility, since general curve-curve intersection problems are replaced with simpler curve-line intersection. Simultaneously, it provides robustness since cells of the spatial hierarchy define a well-defined planar map, even in the presence of numerical errors. For instance, it automatically handles “curve soup” where segment endpoints are not explicitly connected in the input file. The method scales to a large number of primitives, and is output sensitive: it resolves intersections only up to a user-defined precision—while still providing topologically valid output. We evaluate the method on a collection of challenging tests, showing that it is both more robust and orders of magnitude more efficient than existing curve arrangement techniques, such as those found in CGAL.

Authors' Contact Information: Robert Fuchs, robert@fuchsian.net, New York, NY, USA (current address); Keenan Crane, kmcrane@cs.cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA and Roblox, San Mateo, CA, USA..

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7368/2026/7-ART106
<https://doi.org/10.1145/3811357>

CCS Concepts: • **Theory of computation** → **Computational geometry**; • **Computing methodologies** → **Non-photorealistic rendering**; *Parametric curve and surface models.*

Additional Key Words and Phrases: vector graphics, planar arrangements, graphic design

ACM Reference Format:

Robert Fuchs and Keenan Crane. 2026. Robust Planar Maps for 3D Vectorization. *ACM Trans. Graph.* 45, 4, Article 106 (July 2026), 19 pages. <https://doi.org/10.1145/3811357>

1 Introduction

Why vectorize 3D graphics? Vectorized images of three-dimensional geometry enable designers to highlight, abstract, or annotate features that are difficult to understand via more realistic, shaded images. For both scientific and artistic purposes, they offer numerous styles that are difficult to achieve via ray tracing or rasterization (e.g., dashed hidden lines, or silhouette curves of uniform thickness). All the usual arguments for vector images also apply: resolution independence, translation to physical output (e.g., for computer-controlled plotters, cutters, and routers), or dynamic and responsive style adjustments (e.g., maintaining minimum stroke width, or adapting patterns to display size).

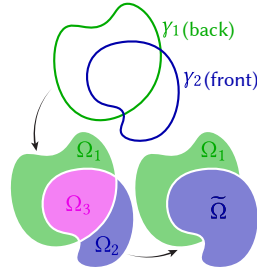
The leaky vectorization pipeline. At present, however, 3D vectorization is far behind existing ray tracing and rasterization pipelines: reliably vectorizing complex geometry is already a difficult task; vectorizing complex material, lighting, and shading effects (such as hard shadows or specular reflections) is generally out of the question [Bénard and Hertzmann 2019]. This pipeline is hard to build for many reasons; one is that vectorization must address difficult

computational-geometric questions that are side-stepped by sample-based methods (ray tracing and rasterization). Moreover, geometric errors made upstream can lead to failure downstream—e.g., poorly-constructed silhouette curves make it difficult to reliably extract 2D regions. For silhouettes in particular, there has been a recent push toward making algorithms more reliable [Capouellez et al. 2023; Liu et al. 2023]. But the equally important task of robustly computing the *planar map* [Bénard and Hertzmann 2019, 4.8] has received less attention.

Problem statement. In particular, given a collection of closed or open curves $\gamma_1, \dots, \gamma_n$ in the plane, we want to compute the collection of disjoint connected regions $\Omega_1, \dots, \Omega_k \subset \mathbb{R}^2$ bounded by these curves. In our case, we aim to describe the region boundaries $\partial\Omega_i$ as segments of the input curves γ_j , rather than (say) approximating polygons. Here we must not answer only a geometric question (“*what does each region look like?*”) but also a topological question (“*how are these regions connected to each other?*”). In the case of 3D vectorization, one must go one step further: since the curves arise from a collection of surface patches $S_1, \dots, S_m \subset \mathbb{R}^3$, which may occlude many curve segments, we wish to find the *coarsest* set of regions that partition the plane into regions with constant patch identity. E.g., if two neighboring regions Ω_i, Ω_j belong to the same surface patch S , and are both visible, then we must produce a single combined patch $\tilde{\Omega} = \Omega_i \cup \Omega_j$. In cases where we have partial transparency, we must also provide a correct back-to-front ordering of regions—again using the coarsest set of patches that preserves this ordering. More precise descriptions of the input and output are given in Section 1.2.

Our system design is based on a set of goals and constraints that are motivated by our own firsthand experience with extensive manual and semi-automatic vectorization of 3D geometry [Crane 2013], which we evaluate via several salient ablations and comparisons (Section 5):

- **Generality.** The system must handle more than just polygonal geometry, including interaction between heterogeneous curve types (e.g., ellipses and Bézier curves). Reducing all curve types to a monolithic representation (e.g., polylines) is unsatisfactory from the perspective of approximation quality (e.g., poorly-approximated curves, or cracks between regions), output file size, and editability of the output (see below).
- **Robustness.** The system should reliably generate well-behaved regions even in the presence of (i) defects in the input curves (e.g., curves that do not form perfectly closed regions), (ii) near-degenerate curve configurations, and/or (iii) imprecision in geometric calculations (e.g., inaccurate evaluation of curve-line intersections). Especially for curves arising from 3D models, cases that are ordinarily pathological (e.g., coincident near-tangential) become quite common, due to frequent visibility events (Figure 1). Moreover, although some work provides strong guarantees about 3D contours [Capouellez



et al. 2023], curves generated by most real-world software—such as the NURBS curves used in most of our examples—exhibit frequent gaps, erroneous intersections, etc..

- **Efficiency and Scalability.** Compute time should scale near-linearly in the size of the output, and should have absolute run times fast enough to provide interactive feedback. (Note that we do not bound complexity in terms of *input* size, since the correct output size can be super-linear in the input—e.g., a regular grid of $O(n)$ overlapping line segments yields $O(n^2)$ square regions.)
- **Editability.** One design goal for a 3D vectorization system is that the vectorized output should not merely reproduce the desired image, but it should also be easy to *edit*. E.g., smooth Bézier curves should maintain a small number of control handles, and visually contiguous regions should correspond to connected pieces of the planar map. In contrast, many planar map algorithms tessellate smooth, concise curves into numerous short segments, and many 3D plotting tools emit correct-looking images, but “dice” up 3D surfaces into tiny regions that are impossible to edit by hand (Figure 2). Likewise, image-based tracing [Selinger 2003] of a “first hit” raster image cannot resolve multiple layers of transparency, again yielding an over-tessellated result that is difficult to edit. Our approach merges together fragments from the same layer of visibility into large pieces that are easier to edit (Figure 3).
- **Non-goal: Geometric Exactness.** A large body of work in computational geometry and digital processing focuses on *exactly* resolving *arrangements* of 2D and 3D objects [Hanniel and Wein 2007; Zhou et al. 2016]. For some applications (e.g., finite element analysis), it is critical to resolve these intersections be in a geometrically exact way. For visual applications, however, we often do not need this level of precision—and instead balance our system design around considerations of flexibility (“does it work on my input?”), visual fidelity (“does it look right?”) and computational cost (“does it finish in a reasonable amount of time?”).

In fact, the central insight of our approach is that, by allowing our planar map to deviate slightly from the exact solution, we obtain substantial flexibility in our system design, enabling us to focus on the more designer-relevant goals above. As a result, we enable users to create informative, high-quality vector illustrations for categories of geometry that are a struggle for existing tools (Section 5). More broadly, it provides foundations for a more comprehensive vectorization pipeline, incorporating challenging effects beyond simple wireframes (e.g., occlusion Section 4, transparency Section 4.4, or reflection Figure 6).

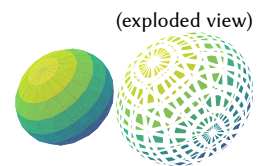


Fig. 2. Many tools for plotting 3D surfaces yield output that is visually reasonable (left) but is diced up into tiny pieces that are impossible to edit (right).



Fig. 3. A robust planar map that preserves large, connected regions is useful not only for display and stylization, but also makes it easier for a designer to edit and interact with vector artwork. Here for instance the exploded view shows a clean segmentation into manageable pieces.

1.1 Related Work

Below, we review work from computational geometry (Section 1.1.1) and vector graphics (Section 1.1.2) on algorithms for planar map construction. More broadly, planar maps [Baudelaire and Gangnet 1989; Asente et al. 2007] and more general cell complexes [Dalstein et al. 2014] have been used as a paradigm for editing vector graphics, though this work does not address the problem of constructing a map from given curves. Likewise, hierarchical decomposition plays a key role in methods for fast rasterization of vector graphics, but this work does not construct an explicit planar map suitable for output and editing. [Nehab and Hoppe 2008; Ganacim et al. 2014; Dokter et al. 2019].

Ultimately, there are surprisingly few software implementations of general planar map algorithms. For instance, *Clipper* [Johnson 2025] supports Boolean operations only on closed regions, and *GEOS* [GEOS contributors 2025] performs polygonization of input curves, followed by rasterization that would require a resolution of approximately $1 \text{ billion} \times 1 \text{ billion}$ pixels to match the precision of our spatially adaptive approach. We hence focus our numerical comparisons in Section 5 primarily on the line and Bézier arrangement algorithms implemented in the *Computational Geometry Algorithms Library* (CGAL) [Fabri and Pion 2009].

1.1.1 2D Arrangements. A planar map encodes an *arrangement* of solid 2D or 3D objects—beyond vector graphics, such arrangements arise broadly in geometric modeling and computer-aided design (CAD) as a way of specifying shapes as composites of more basic primitives [Voelcker and Requicha 1977]. Here, a wide variety of methods have been developed for careful resolution of arrangements in both 2D [Bentley and Ottmann 1979] and 3D [Laidlaw et al. 1986; Bernstein and Fussell 2009; Zhou et al. 2016; Cherchi et al. 2020], with significant emphasis on polygonal or polyhedral geometry. The engineering context of this work also means that exactness of intersections is often prioritized over other criteria (such as speed or

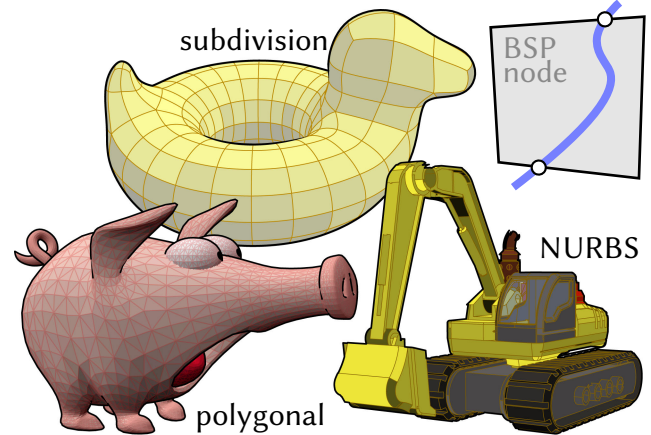


Fig. 4. To generate a planar map, we need only perform line-curve intersections with edges (*top right*), enabling us to handle heterogeneous mixtures of geometry defined by any type of curve.

flexibility), since in a physical context even small geometric defects can have catastrophic consequences [Feynman 1988].

Though there has been some work on, e.g., Bézier arrangements [Gangnet et al. 1989; Hanniel and Wein 2007], we are not aware of any algorithms that (as we do) handle heterogeneous mixtures of arbitrary curve types as input, nor produce such mixtures as output (Figure 4). Moreover, whereas past methods largely treat curve-curve intersection and region identification as two independent tasks, we take a unified approach where a spatial hierarchy (BSP tree) simultaneously serves to define planar map regions, and compute intersections consistent with those region definitions. In essence, we make topological consistency—rather than geometric exactness—the highest priority, though can still achieve excellent geometric precision if desired (by increasing the maximum tree depth). Likewise, while much of the work in this area relies on adaptive precision or exact arithmetic, we use only ordinary floating point computation, achieve robustness by instead leaning on the combinatorics of the BSP tree itself.

In practice our method hence reliably produces meaningful output even on the pathological curve arrangements arising from 3D projection (Figure 1). While Bénard and Hertzmann [2019] suggest that a standard *sweep line algorithm* can be used for this purpose [Bentley and Ottmann 1979], experiments in Section 5 show this not to be the case: even careful implementations of past arrangement algorithms [Flato et al. 1999] fail to be robust and/or scalable enough for complex 3D objects (Section 5).

Output Simplification. Often, we want our vector output from the arrangement algorithm to be simplified or compressed. Existing simplification methods [Nguyen et al. 2024] work downstream of the arrangement algorithm, consuming the planar map in exact resolution in order to produce the simplified output. Our method can produce simplified arrangements just by varying the specified precision. This allows us to skip resolving regions that get simplified away, albeit with less visual control than downstream methods such as Nguyen et al. [2024].

1.1.2 3D Vectorization. 2D arrangement algorithms play a key role in vectorization pipelines, used to resolve intersections between silhouettes that arise from 2D projection. Existing 3D vectorization pipelines, such as Eisemann et al. [2008], rely on the same small set of arrangement kernels, such as Flato et al. [1999]. We hope to provide a robust and efficient alternative to these existing kernels.

BSP Trees. A key component of our system is the use of a *binary space partition (BSP)* tree to identify curve-curve intersections. The only reason we use a BSP, rather than a simpler *quadtree*, is to improve floating point robustness, as detailed in Section 2.1. Importantly, this usage is completely unrelated to the way they are applied in older vectorization methods, where faces of a BSP tree are used to encode polygonal [Winkenbach and Salesin 1994] or polyhedral geometry [Thibault and Naylor 1987]. Winkenbach and Salesin [1994, Appendix A.2] gives an explicit description of how this construction helps build a planar map, essentially by computing (approximate) set unions of visible polygons—a strategy that does not apply to more general curved boundaries, nor contours beyond silhouettes. Winkenbach and Salesin [1996] consider a similar strategy, again only for polygonal geometry, and requiring a dense surface triangulation. Naylor and Rogers [1995] consider Bézier curves, but only for evaluating visibility, and not for building general planar maps.

Occlusion. A classic approach to handling occlusion in 3D vectorization is *quantitative invisibility (QI)* [Appel 1967], which uses local geometric information to propagate visibility across a mesh data structure. This classic approach is insufficient for our needs since (i) it uses rules that assume piecewise planar geometry and (ii) it does not easily extend to partial transparency. We develop an alternative approach to (partial) occlusion that handles both of these phenomena (Section 4). This strategy is closer in spirit to that of Eisemann et al. [2009], except that we work directly with the smooth contours of the planar map, rather than needing to decompose a polygonal mesh.

For partial occlusion, we use a local ordering approach that we then resolve to obtain a global order required for SVG output. This idea of using a local order to define depth has been used in McCann and Pollard [2009] and McCann and Pollard [2012] to improve editability of images with complex depth relations that a single global order cannot resolve. We use a local ordering approach for the same reason, as it allows us to easily resolve global ordering cycles.

1.2 Method Outline

Input. The input to our algorithm is a viewport (i.e., a rectangular region in the image plane), a collection of surface patches $S_1, \dots, S_m \subset \mathbb{R}^3$, and a collection of parameterized curves $\gamma_1, \dots, \gamma_n : [0, 1] \rightarrow \mathbb{R}^2$ encoding the patch boundaries under perspective projection. The patches could represent, e.g., faces of a polyhedron, or more general regions like quadrangles on the sphere bounded by parallels and meridians. They need not all be the same kind, and can be described via a “black box” interface, where the queries we must be able to answer are:

- the first patch S_i hit by a given ray $\mathbf{r}(t) = \mathbf{p} + t\mathbf{u}$, and
- the set of points where a projected curve $\gamma_i(s)$ intersects a given line segment $\mathbf{l}(t) := (1-t)\mathbf{a} + t\mathbf{b}$.

We hence reduce the problem of pairwise intersection between arbitrary curve types (requiring $O(n^2)$ subroutines for n curve types) to the much simpler case of curve-line intersection (requiring only $O(n)$). The algorithm also takes a parameter $\varepsilon > 0$, giving the (linear) error in intersection location that we are willing to tolerate.

For optional simplification via *separating lines* (Section 2.3), we must also be able to evaluate the convex hull of any curve $\gamma(s)$ over an interval $a \leq s \leq b$. However, the core algorithm still works even if these queries are not supported.

Output. The output is guaranteed to be a collection of regions $\Omega_1, \dots, \Omega_k$ that partition the plane, up to small gaps or overlaps of area no greater than ε^2 (Section 3.4). For each such region we also provide the identity of the corresponding surface patch, which we denote by $S(\Omega_i)$. As $\varepsilon \rightarrow 0$ regions approach the partition induced by closest point visibility—at least in exact arithmetic. This first guarantee is an important feature of our algorithm: we always get a valid vectorization, even if, e.g., small regions have been simplified and/or the location of some intersection points has been approximated. Optionally, we can also output a collection of occluded segments $\eta_1, \dots, \eta_k$ (that might be drawn as, e.g., dashed lines).

Nomenclature. Throughout we use the following terms:

- node — node of the BSP tree
- intersection node — leaf node containing more than one curve
- vertex — vertex of the output
- segment — edge of a BSP node
- curve — curve γ_i encoding boundary of a surface patch
- edge — a logical edge of the planar map (not necessarily a straight line segment)

Stages. Given the input, our algorithm proceeds in three stages:

- (1) **Intersection Detection.** (Section 2). We first locate all curve-curve intersections simultaneously, by constructing a *binary space partition (BSP)* of the image plane. Nodes of this BSP that intersect more than one curve γ_i are split recursively, until they fall below size ε . Such nodes in turn represent vertices of the final planar regions.
- (2) **Region Identification.** Using the adjacency of BSP nodes, and the identity of curves intersected by nodes, we construct a *planar map* where each logical edge corresponds to a segment of a curve split by intersection points. Since the curves γ_i are all region boundaries, hence closed loops, faces of our planar map describe a well-defined partition. (Section 3).
- (3) **Merging & Occlusion Culling.** At this point, our partition is finer than necessary, since the planar map includes many curve segments that are occluded. Hence, we use ray tracing to determine the identity of each logical face in the planar map, and remove logical edges between faces that have the same identity (adding these segments to the list of hidden curves η_i). Alternatively, we can generate a “multilayer planar map” that accommodates partial transparency (Section 4.4). This step also provides a correspondence between 2D regions and 3D surface patches S_i , needed for shading/coloring (Section 4).

From here, a user can style the output in any desired way.

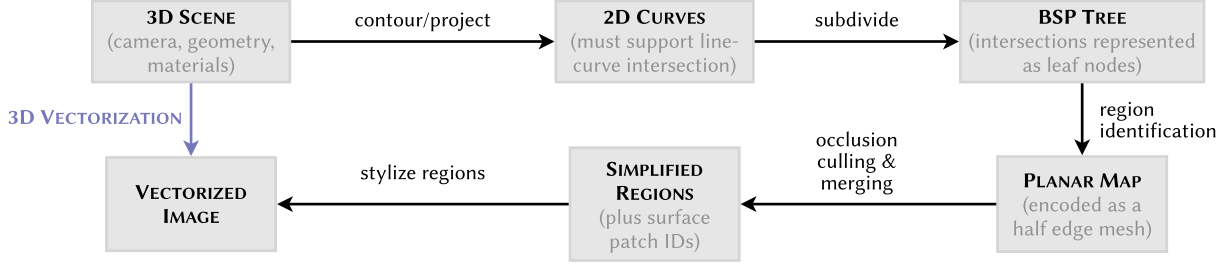


Fig. 5. Our algorithm pipeline. Our starting point is the projection of each 3D curve to a curve in the plane. To simultaneously detect all intersections, we recursively subdivide the image plane until no node contains more than one curve segment. By using a general BSP tree, we can reduce all nodes to a single common case, greatly simplifying subsequent region detection. Once regions have been detected, we can merge regions from the same surface to implement occlusion culling (and to recover occluded curves). This information is then styled to generate a final vectorized image.

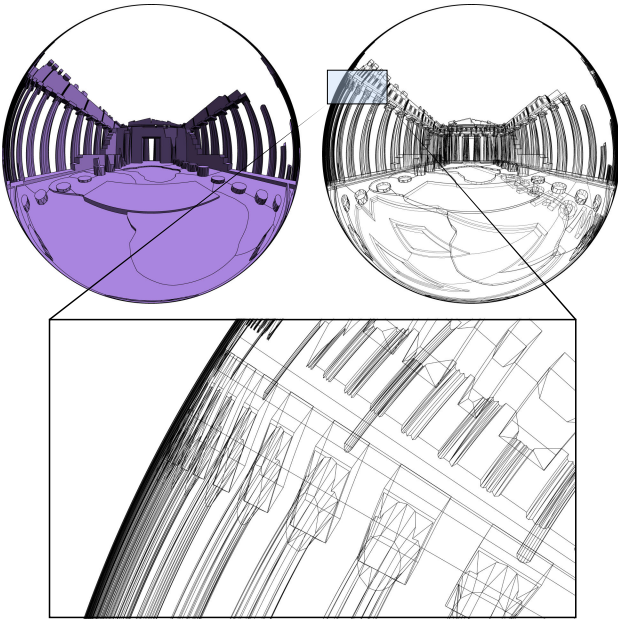


Fig. 6. The robustness and flexibility of our method makes it possible to vectorize effects that are essentially impossible for existing methods, such as global illumination effects. Here, for instance, we correctly extract and shade regions bound by specular reflection lines on a sphere—which bunch up into extremely degenerate bands of curves near the sphere boundary.

An attractive feature of our approach is that it is agnostic to the particular class of projected curve (e.g., Bézier, NURBS, conic section, etc.). More precisely, as long as the user can provide a subroutine that detects intersections between a straight line and a given curve type, we effectively “elevate” these queries to resolve intersections between arbitrary types of curves. This is a critical feature of a real system, because it removes the $O(n^2)$ burden on developing reliable curve-curve intersection routines among n different curve types.

2 Intersection Detection

If individual curves contain self-intersections, we first pre-process them into disjoint curves (e.g., splitting a cubic Bézier segment into two). To detect the points at which distinct curves intersect, we then build a BSP tree in a top-down fashion, using the viewport to define the geometry of the root node. The reason for using a general BSP tree, rather than a more specialized tree like a quadtree, is that it allows us to avoid degenerate cases (detailed in Section 2.1); in turn, by ensuring curves intersect all nodes in a generic way, we greatly simplify region identification (Section 3). Algorithmically, nodes are recursively split until all nodes either (i) intersect no more than a single curve, or (ii) have size less than the target precision ϵ . In practice, if A_0 is the area of the viewport, and d is the depth of the node in the BSP tree, we stop when

$$\frac{1}{2^d} A_0 = \epsilon^2$$

or equivalently, when we reach a maximum depth of

$$d_{\max} := \log_2(A_0/\epsilon^2)$$

(which also ensures that the algorithm always terminates in finite time). In principle one could directly compute, e.g., the node diameter, but in practice we find this level of precision is not necessary. Throughout, we refer to a leaf node intersecting more than one curve as an *intersection node*.

Preprocessing. We require all curves to be non-self-intersecting. By making such an assumption, we guarantee that we terminate splitting early only when a node contains no intersection. Thus we preprocess our curves, splitting self-intersecting curves until no piece contains a self-intersection.

Virtual Intersection Points. Importantly, the intersection points determined by our BSP tree are not meant to correspond to exact, geometric intersection points of two curves. Instead, they *define* what we call *virtual intersection points*, which ultimately become vertices of a topological mesh defining our output regions (Section 3). In fact, depending on the tolerance ϵ and the complexity of the curve arrangement, a single virtual intersection point may serve as a proxy for a large number of geometric intersections within a single node. We make this choice by design, so that we do not needlessly resolve fine-scale intersections in tiny regions, which have no impact on

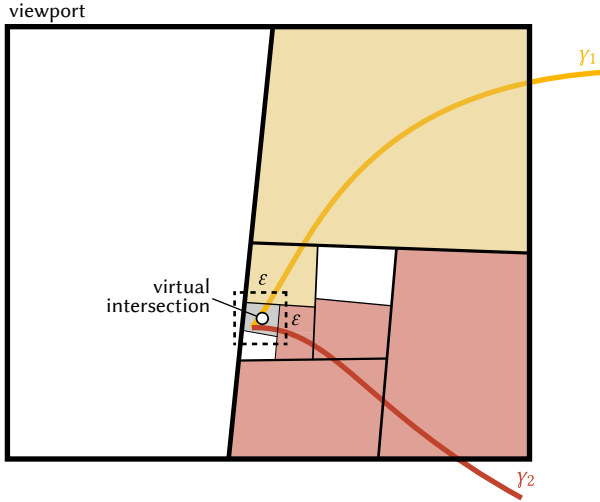


Fig. 7. We use a BSP tree to simultaneously identify pairs of curves that intersect, and also to approximately compute the coordinates of intersection. To do so, we keep splitting nodes until they contain at most one curve, or until their size goes below an accuracy parameter ϵ . Any remaining node that contains more than one curve is identified with a “virtual vertex,” which becomes a vertex in our output vectorization.

the final illustration. Indeed, this approach enables us to generate vectorizations in a multi-scale fashion, and also enables us to make guarantees about the connectivity of the output, by reducing numerous potentially complex topological arrangements of curves to a single common case. Note that traditional approaches, such as sweepline algorithms, fundamentally cannot perform this kind of simplification, since they process only one pair of curves at a time.

BSP Data. Ultimately, for each node of the BSP tree we store (i) the geometry of each BSP node, given as a list of vertices $p_1, \dots, p_k$ for a convex polygon, (ii) the node edges and locations where the split occurs (for interior nodes), and (iii) a collection of tuples (γ_i, t_0, t_1) encoding all segments $\gamma([t_0, t_1])$ that intersect the node (for both interior and leaf nodes).

2.1 Node Splitting

If a node N contains more than one curve and is above the maximal depth $d_{\max}$, we split it into two children N_1, N_2 by splitting along a line L . These splitting lines are alternately oriented along the horizontal and vertical axes. In general, however, we cannot simply draw this line through the midpoints of the node edges, since curves can come arbitrarily close to these midpoints. Instead, let $t_1, \dots, t_k$ be the points of intersection along the node edge E , and let $t_0 = 0, t_{k+1} = 1$. Then we pick a splitting line that passes through the midpoint m of the interval $[t_i, t_{i+1}]$ containing the midpoint of E (choosing the left or right interval arbitrarily if a t_i is exactly $1/2$). Since there can only be finitely many intersections k , the distance between m and the closest intersection t_i will always be a large value in floating point. Moreover, since we pick the midpoint m closest to the midpoint of E , the splitting lines will still tend to be close to horizontal/vertical, producing roughly rectangular cells where our

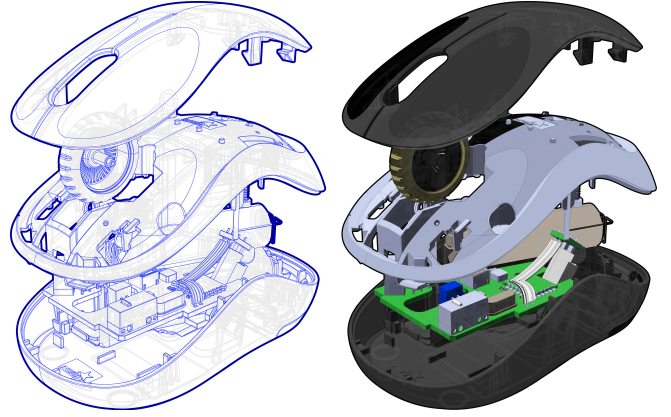


Fig. 8. The spatially adaptive nature of our BSP-based approach enables it to scale to extreme levels of geometric detail, here resolving a detailed schematic for a mouse (left) into well-defined screen-space regions with extraordinarily fine features (right).

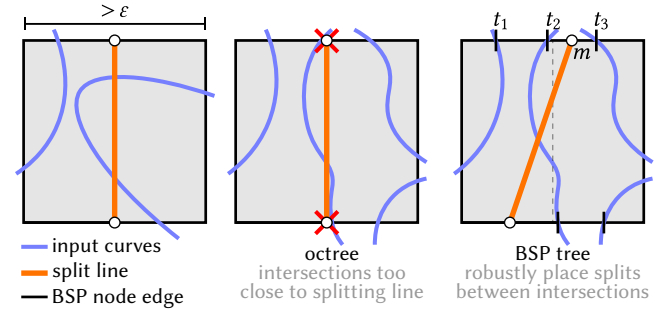


Fig. 9. *Left*: we split any node containing more than one curve along either the horizontal or vertical direction. *Center*: to avoid degenerate cases where curves come arbitrarily close to node corners, we allow non axis-aligned splits, resulting in a BSP tree. *Right*: In particular, we split at the midpoint m of the interval t_i, t_{i+1} containing the edge midpoint.

termination depth (Equation 2) is still meaningfully connected to the target precision ϵ . This scheme also maintains good numerical conditioning of the node geometry (e.g., logically horizontal edges don’t become near-parallel with logically vertical edges).

Each time we split a node N , we also split any curve segments $\gamma([t_0, t_1])$ contained in that node by the split line L . Segments on either side of this split line are then handed to the two children N_1, N_2 . Hence, for each split, we need only find intersections between a single line, and a small, local set of curve segments.

2.2 Intersection Tests

In general, we can associate any split line L with a linear function f such that $f(x) = 0$ whenever $x \in L$. Intersections of a curve γ with L then correspond to parameter values t such that

$$f(\gamma(t)) = 0.$$

Finding intersection points then amounts to a standard root finding problem. In our case, we need to find *all* the roots $t_1, \dots, t_k \in [0, 1]$.

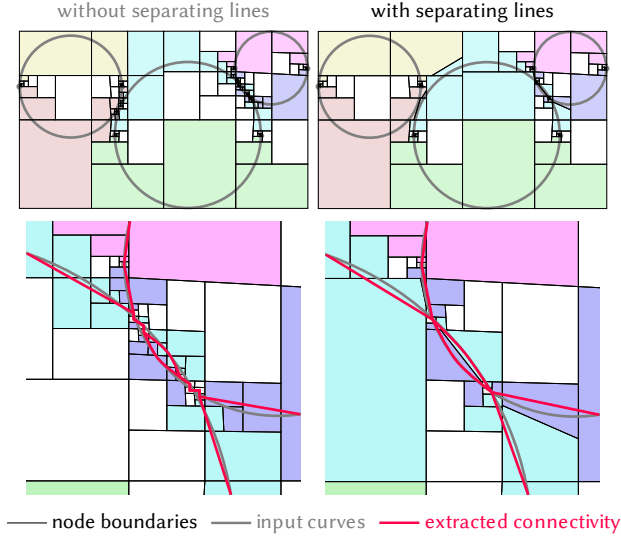


Fig. 10. We can improve both the efficiency and quality of vectorization by more intelligently splitting nodes containing multiple *non-intersecting* curves. *Left*: the BSP tree generated without separating lines is far larger than the one using such lines. *Right*: the use of separating lines also avoids “zipper” lines where two lines that should meet at an ‘X’ instead get vectorized as a pair of ‘Y’s.



Fig. 11. These glasses have tiny details in the input geometry, which we can resolve with a large max depth for our BSP tree (*top left*). If we use a low max depth, these details get simplified into a few regions (*middle left*). However, if we aggressively split along separating lines, we lose our ability to simplify regions, even at the same low max depth (*bottom left*).

Whether this can be done reliably depends on the specific form of the curve γ . In Appendix A and Appendix B we give two common examples: Bézier curves, and ellipses—which in CAD frequently arise as contours and silhouettes of conics (spheres, cones, ellipsoids, etc.). As noted above, each split curve segment is encoded by a tuple (γ_i, t_j, t_{j+1}) .

2.3 Separating Lines

The algorithm we’ve described so far will identify intersections (up to the tolerance ϵ) without any further modification. Optionally, however, we can get better efficiency *and* improve quality by more intelligently splitting nodes that contain multiple *non-intersecting*

curves—at the additional burden of needing to supply a conservative convex bounding polygon, such as the convex hull. Since a convex hull is easily obtained for common curve types (e.g., Bézier or ellipse), we opt for this strategy in all of our examples, apart from the comparison shown in Figure 10. This strategy is more efficient because we build a BSP tree with fewer nodes (Figure 10, *top*). It yields higher-quality results since it avoids “zippering” together lines at small angles. In particular, for any tolerance $\epsilon > 0$, one can always have a pair of lines (say) with angle θ between them so small that the basic procedure from Section 2.1 detects false intersections, i.e., it reaches the maximum depth not only for the node containing the true intersection, but also for nearby nodes where the two curves are extremely close (Figure 10, *bottom*). Likewise, if we have near-parallel lines separated by a distance smaller than ϵ , the separating line strategy can correctly resolve these lines, rather than merging them together.

In more detail, if we have two non-intersecting curves γ_1, γ_2 in a node, the strategy from Section 2.1 will *eventually* separate them by just repeatedly splitting—but often at an unnecessarily large depth. Instead, we can often immediately split a node N into two leaf nodes N_1, N_2 by instead letting the splitting line L be any *separating line* between γ_1 and γ_2 , i.e., such that γ_1 is entirely on one side of L , and γ_2 is entirely on the other side. One way to define such a line is to consider the convex hulls C_1, C_2 of γ_1, γ_2 , *resp*. A supporting line of C_1 is a separating line by if it fails to intersect C_2 , and vice versa. In practice, we compute this line using a standard *separating axis test* (SAT) [Ericson 2004, Section 5.2.1]. If a node contains more than two curves, we apply this procedure only if we can find a separating line between all pairs of hulls; otherwise we apply the procedure from Section 2.1. This strategy is applied even if we are beyond the maximum depth $d_{\max}$.

While it may be natural to consider applying this procedure if we can find a separating line between any two curves in a node, this can increase the effective precision of our BSP tree and ruin any simplification properties obtained from using a lower precision (Figure 11). Splitting a node further refines the convex hulls of the curves within, causing curves to admit new separating lines. Thus if we aggressively split along separating lines, we create new separating lines that we will further split along. This repeats until refining convex hulls no longer creates new separating lines, making our final BSP tree have a maximum depth (and thus effective precision) much higher than specified.

3 Region Identification

To define the regions Ω_i of our partition, which will ultimately correspond to filled shapes, we translate connectivity information from our BSP tree into a planar map. Traditionally this map is built by inferring connectivity from local pairwise intersections—which can fail to be robust (Section 5). Our key observation is that, since a BSP is, by definition, a partition of space, we can rely on the dual structure of the BSP to define regions Ω_i that are always globally consistent.

The connectivity of this mesh is encoded by a standard *halfedge data structure* (Section 3.1); its geometry is comprised of segments of the original curves γ_i , plus some occasional connecting lines. The

process for building up the planar map consists of three sub-steps, detailed below.

3.1 Planar Map

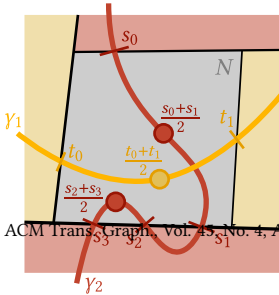
3.1.1 Halfedge Data Structure. Throughout we encode the connectivity of the planar map via a *halfedge mesh* M with vertex set V , edge set E , face set F , and halfedge set H . A halfedge mesh encodes connectivity through the use of halfedges, which are edges with a specified direction. For each edge (u, v) , we assign two halfedges; one travels from u to v , while the other travels from v to u . For each halfedge, we define two operations: twin and next. Using the twin operation on a halfedge gives the halfedge on the same edge that travels in the opposite direction. Using the next operation on a halfedge gives the next connected halfedge within the same face (i.e. for a halfedge $u \rightarrow v$, the next operator will return a halfedge $v \rightarrow w$).

Given the halfedge set H and the connectivity implied by the twin and next operators, we can recover the vertex set V , edge set E , and face set F . For each halfedge, the connected edge can be defined as the orbit of the twin operator. Similarly, the connected face to a halfedge can be defined as the orbit of the next operator. Finally, the connected vertex to a halfedge can be defined as the orbit of the next operator composed with the twin operator.

Notably, a halfedge mesh is equivalent to a *rotation system*, which defines its connectivity by ordering the edges around each vertex. Due to this equivalence, simply ordering the edges around each vertex completely determines the complex formed by a halfedge mesh. We use this equivalence in Section 3.3 to define our halfedge mesh, and then recover the regions of the planar map through the orbit of the next operator.

3.2 Mesh Initialization

Recall that, rather than resolving exact curve-curve intersection points, our BSP tree defines virtual intersection points. The set of vertices in our planar map consists of both (i) the virtual intersection points, and (ii) the endpoints of curves γ_i . We then have one logical edge for each contiguous segment of a curve γ_i between any two vertices of the planar map, and two halfedges for each such edge. Importantly, these are *logical edges*, i.e., they are not literal straight segments in the plane. Moreover, since multiple curves can intersect a single BSP edge, the planar map can have multiple edges between the same two vertices v_1, v_2 (see inset). These edges will later be simplified into single edges (Section 3.3).



More explicitly, suppose that a curve γ_i passes through an intersection node N , intersecting its boundary at parameters $t_0, t_1 \in [0, 1]$. Then we split it into two segments, $\gamma_i([t_0, \frac{t_0+t_1}{2}])$ and $\gamma_i([\frac{t_0+t_1}{2}, t_1])$. More generally, a

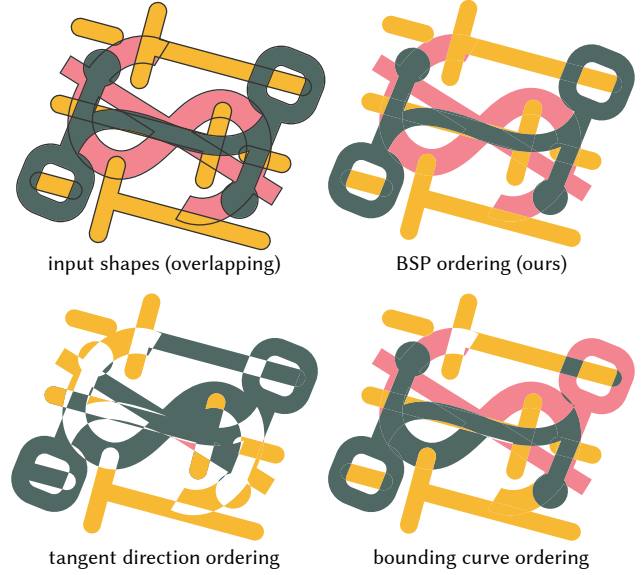


Fig. 12. Small local errors in ordering of curves around vertices of the planar map can yield large global misclassification of regions. Here, ordering by tangent (*bottom left*) or a bounding curve (*bottom right*) lead to holes and miscolored regions. In contrast, our BSP-based regions are consistent with curves by construction (*top right*)

curve γ_i could pass in and out of the same intersection node N multiple times; in this case, we simply split γ_i at the midpoint of each interval made by two intersection points.

(If a curve segment terminates inside an intersection node, we do not split the terminal interval.) Note that the segment endpoints are not (and need not be) located at the same geometric points within a node containing a virtual intersection. For instance, another curve γ_j passing through the same node N will likewise be split in two at a mean parameter value $\frac{s_0+s_1}{2}$ —but in general, $\gamma_i(\frac{t_0+t_1}{2}) \neq \gamma_j(\frac{s_0+s_1}{2})$. This inconsistency will be addressed when we define the region geometry, in Section 3.4.

3.3 Face Connectivity

At this point, we have defined the “1-skeleton” of the planar map (essentially the *view graph* [Bénard and Hertzmann 2019, Section 4.5]), i.e., a graph with just vertices and edges. To fill these edges in with well-defined faces, we need to (i) determine the order of edges around each vertex, and (ii) eliminate any duplicate edges. Consistent ordering of edges around vertices is essential for obtaining a partition into consistent regions (Figure 13); eliminating duplicate edges omits degenerate elements from the output.

Edge Ordering. The particular strategy we use to order edges around a vertex has a significant impact on robustness. In fact, many natural ways one might try to order edges lead to poor region definitions, that yield dramatic visual artifacts (Figure 12). One

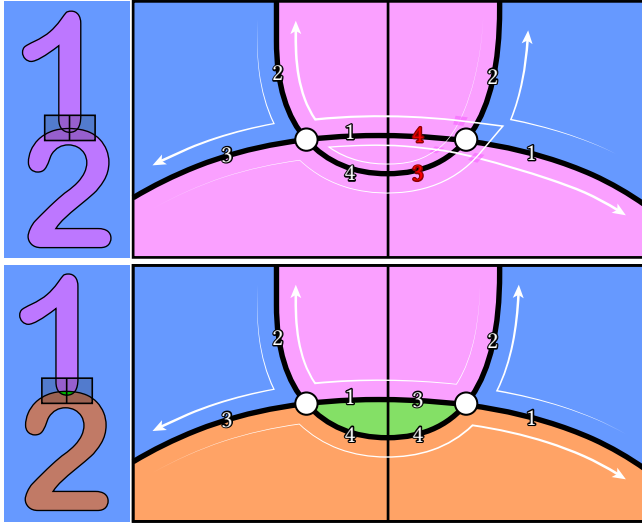


Fig. 13. Even local errors in the ordering can lead to large global inconsistencies in the planar map. For instance, even if just a single pair of edges are incorrectly ordered (in red), we obtain a planar map which erroneously identifies two overlapping regions (*top*), rather than four disjoint pieces that correctly partition the domain (*bottom*). We use a robust ordering strategy based on the BSP tree, rather than any geometric test, to guarantee ordering consistency between neighboring nodes.

possibility, for instance, is to order edges according to the angles of outgoing tangents; another is to draw a larger bounding curve η around the intersection, and order the outgoing edges according to intersections of segments γ_i with η . In practice, neither of these strategies work well. For instance, due to floating point error, we can get inconsistent angular orderings between two neighboring nodes, leading to large global errors in our partition of the image into regions (Figure 13). Likewise, if we use independent bounding curves η for neighboring intersection nodes (e.g., a bounding circle around each node), there is again no reason why floating point calculations should yield consistent ordering from node to node.

Instead, we use the connectivity of the BSP tree to ensure consistent ordering of edges in the mesh. In particular, for each segment between two BSP nodes, we can compute a canonical ordering of the intersections with any segments γ_i . Even if this ordering does not agree with the geometric ordering (e.g., due to floating point error), the nodes on either side still have a shared, canonical ordering that can be used to achieve region identification. To connect edges of the 1-skeleton up into faces, we then walk around the boundary of each BSP node in counter-clockwise order. The order of edges encountered during this traversal determines the ordering of outgoing halfedges from the virtual vertex in our halfedge data structure. As noted in Section 3.1.1, this ordering completely determines the connectivity of the halfedge mesh, i.e., it completely determines the planar map.

Deleting Degenerate Faces. As described thus far, the halfedge mesh has perfectly valid connectivity and consistently identifies regions from node to node. However, since each node contains at

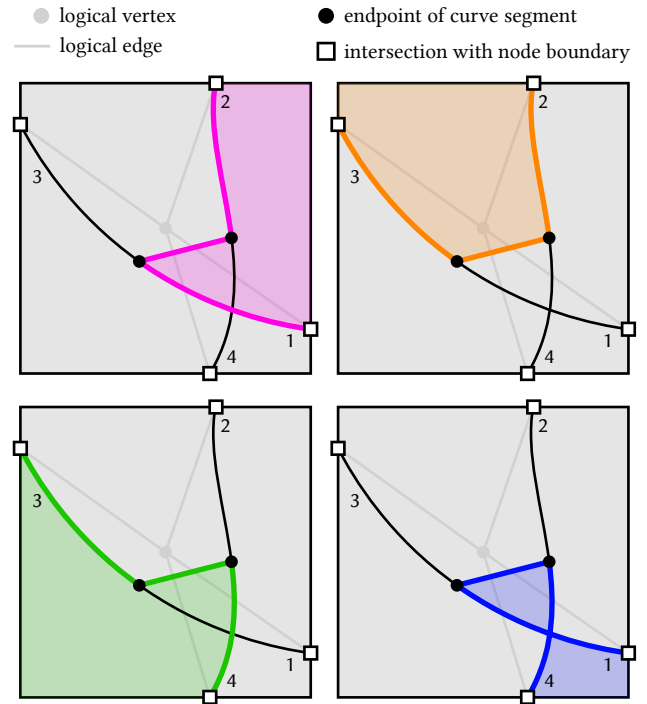


Fig. 14. To define the geometry in the vicinity of any logical vertex of the planar map, we walk around the vertex in counter-clockwise order, connecting segments associated with each logical halfedge. Since segment endpoints will not in general meet, they are connected by a straight line segment.

most one virtual vertex, we might have tiny two-sided regions (such as the green “lens” region in Figure 13, *bottom*). We filter these out by simply removing all but one edge through each BSP node edge, occasionally producing a “zippering” of near-coincident curves into single curves.

3.4 Region Geometry

So far, we have defined the abstract connectivity of the planar map, i.e., we have said how edges connect up to make logical faces. To fully define the regions Ω_i , we also have to define their geometry. Rather than specify, e.g., vertex coordinates (as in a piecewise linear mesh), we assign a closed loop $\Gamma_i(t)$ to each region Ω_i , which in general can be an arbitrary piecewise parameterized curve. Describing the final region geometry as a collection of closed loops is well-aligned with standard vector graphics formats, which express drawings as collections of individual closed shapes (rather than a more general vector graphics complex [Dalstein et al. 2014]). Using general parameterized curves enables region boundaries to exactly follow the input curves (rather than geometric approximations); assigning these curves per face rather than per edge provides some additional flexibility, needed to deal with corner cases near curve-curve intersections.

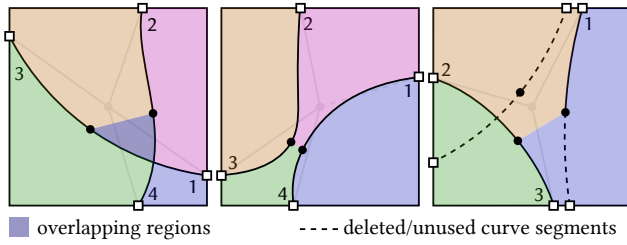


Fig. 15. Our strategy for defining region geometry is extremely robust, handling all possible situations, such as two intersecting curves (*left*), two curves that miss an intersection only up to a tolerance ϵ (*center*), and arbitrary collections of curves that may intersect or slightly miss each other (*right*). The resulting region geometry is defined using exact pieces of the input geometry, which can be a heterogeneous mix of lines, curves, ellipses, etc. The only price we pay for this robustness and flexibility is, in some cases, a tiny amount of overlap, below the visual threshold.

Recall from Section 3.2 that we associate a single parameterized curve segment γ_e with each edge e of the planar map. Hence, to define the curve Γ_i associated with a region Ω_i , we start at any halfedge of the corresponding face f and concatenate¹ consecutive curves $\gamma_{e_1}, \gamma_{e_2}, \dots$. The only exception is that, whenever we encounter a vertex v of face f corresponding to an intersection node, we add a straight line segment between the curve for the incoming halfedge h_k and the curve for the outgoing halfedge h_{k+1} (Figure 14).

Note that region boundaries within an intersection node will not in general fit together perfectly. However, since segments have length of order ϵ , they will generally have no visual impact—rather, they serve only to specify a well-defined closed loop Γ . This choice is in some sense central to our approach: since our goal is to make high-quality images for visualization—rather than, say, high-quality meshes for numerical simulation—we can sacrifice exact conformality of planar regions (i.e., fitting together perfectly at the microscopic scale) for the sake of generality and robustness.

A tempting alternative is to try to “merge” endpoints of curve segments within an intersection node, e.g., nudge them to their mean. However, this idea leads to several undesirable outcomes. First and foremost, by nudging degrees of freedom in one node, we may in fact *change* the set of BSP nodes that the curve intersects, leading to topological inconsistencies. In general, it is difficult to improve robustness of geometric algorithms by making arbitrary perturbations, since these perturbations can have unintended consequences [Edelsbrunner and Mücke 1990]. Second, it makes it difficult for the framework to remain generic—e.g., although it might be easy to perturb the endpoints of line segments, it may be impossible to perturb the parameters of (say) an ellipse so that pieces of the ellipse meet up perfectly in all intersection nodes. Finally, even in cases where such perturbations are possible, they can inflate the size of the output since a single curve (say, a cubic Bézier) may need to be broken up into many smaller segments. Consider for instance a long sequence of T-junctions, as often arises due to occlusion.

¹In a practical implementation, curve concatenation is achieved by essentially just concatenating the curve description, e.g., path data in the *scalable vector graphics (SVG)* format.

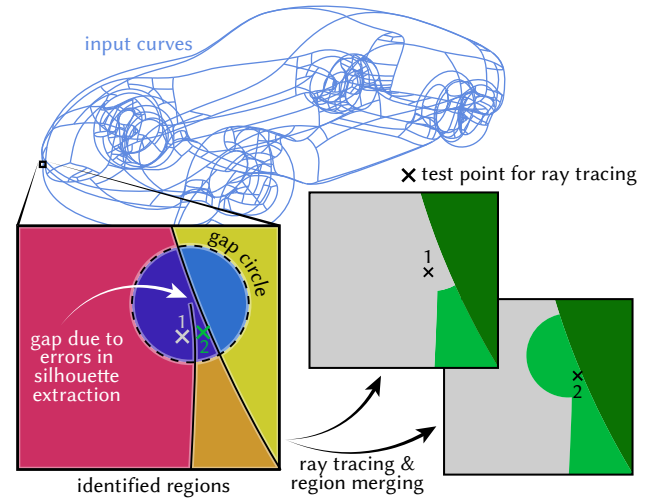


Fig. 16. Our approach is robust to real-world curve data, which can have tiny gaps between curves (here far smaller than a pixel). Here, to mitigate such a gap, we add a small “gap circle” which splits the initial planar graph into additional regions. Identifying regions via ray tracing, and merging contiguous regions, yields a reasonable way of closing the gap—independent of which sample point (1 or 2) we use to determine region identity.

3.5 Robust Gap Handling

So far, we have assumed that the input curves perfectly delineate the desired planar regions. However, many real-world sources of curves exhibit small errors—e.g., tiny gaps due to imprecise evaluation of silhouette curves, on the order of (say) one pixel or less. By default, our algorithm will treat such curves as an exact description of the desired geometry, possibly leading to misclassification over large regions. Optionally, then, we can add tiny “fictitious” proxy geometry to fill in the gaps. Since we do not know *a priori* the direction of the gap, we have found it robust in practice to use small *gap circles*, of a fixed radius $\epsilon > 0$, to fill in the gaps. These circles are added around each open endpoint, prior to building the BSP tree. As seen in Figure 16, there will be some ambiguity about the identity of the regions delineated by the gap circles, depending on which points are used for ray tracing (Section 4.1). However, since these regions are smaller than a pixel, the particular identity of each region has virtually no visible impact on the result. Moreover, after region merging (Section 4.2), the addition of gap circles has minimal impact on the output complexity, typically just adding a single curve segment—which would always be required to close the gap. Finally, in the case of a curve that is genuinely open (such as a visibility contour, see Figure 17) the only consequence is that the open curve gets shorter by about ϵ . Currently we do not try to pick

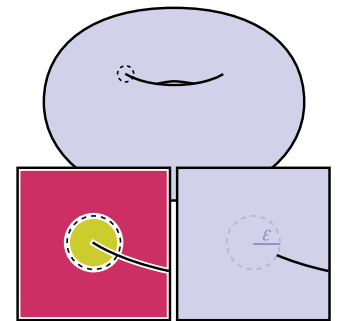


Fig. 17. Our gap filling procedure shortens true open curves by an imperceptible length ϵ .

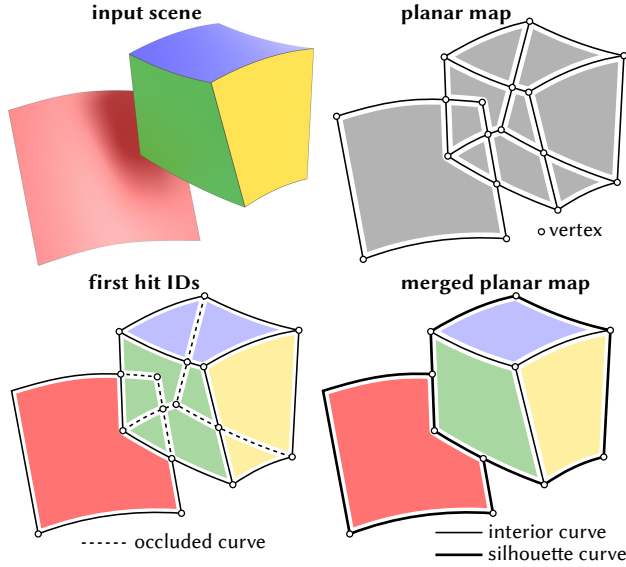


Fig. 18. To handle occlusion, we start with a planar map that includes the maximal set of intersecting regions (*top right*). For each region, we find the first visible patch, using ray tracing in 3D (*bottom left*). A planar map segment is occluded if it has the same closest region on both sides (dashed curves). Optionally, we can distinguish between interior and silhouette curves by seeing if adjacent regions in the planar map are also adjacent regions of the input 3D geometry (*bottom right*).

the gap size automatically, and simply set it to 1/1000th of the image width in all our examples.

4 Occlusion

At this point, the region map delineates all regions formed by intersections of projected curves (Figure 18, *top right*). In order to obtain proper occlusion, we further process this planar map by identifying edges that are occluded (via ray tracing), and merging regions connected by occluded edges. (This step also greatly simplifies our final vector output.) As discussed in Section 1.1, the strategy we outline is quite general, enabling straightforward generalization to partial transparency (Section 4.4), and of course non-polygonal geometry.

Note that our occlusion scheme classifies only logical edges of the planar map, i.e., edges of the halfedge data structure. Hence, the straight segments used to create closed regions in Section 3.4 will never turn into, e.g., spurious silhouette or interior edges in the output.

4.1 Ray Tracing

To determine the identity of each region Ω_i in the planar map, we use a ray tracing strategy that leverages both the 2D BSP tree and the 3D scene geometry. The process proceeds as follows:

For each face f of the halfedge mesh (corresponding to a region Ω_i of the planar map):

- (1) Identify the longest contiguous segment γ of any input curve that lies along the boundary of the region Ω_i . (We use the

longest curve here, since empirically it yields the most stable results for the remaining steps.)

- (2) Compute the midpoint m of the segment γ . From this midpoint, trace a 2D ray into the interior of the region Ω_i , in the normal direction. This step is accelerated using the existing BSP tree (by marching through adjacent BSP nodes, starting at the node containing m).
- (3) Compute the average point $(m + p)/2$, which lies in the interior of region Ω_i . Since all points in the region will see the same closest surface patch, we can trace a ray $r(t)$ from the camera through this point to determine the region identity.
- (4) Trace the 3D ray $r(t)$ through the 3D scene geometry, determining the first hit $t > 0$.
- (5) Assign the identity of the first visible surface patch to region Ω_i , for further processing.

In principle, this process can misidentify the visible surface patch due to external approximation error. For example, we approximate silhouettes using Bézier curves and elliptical arcs, and we use polygonal proxy geometry for ray tracing. In practice, our choice of tracing point in step 3 minimizes any error due to approximation. Additionally, misidentification of a patch only causes local errors in the output vector image, instead of propagating throughout the rest of the image.

4.2 Region Merging

To eliminate occluded segments, and simplify occluding geometry, we merge neighboring regions Ω_i, Ω_j that share the same identity, i.e., which exhibit the same “first hit”. More precisely, we remove any edge $e \in E$ such that the two faces containing e have the same identity, by merging the two adjacent faces.

4.3 Edge Classification

Optionally, if we know the connectivity of regions in the input 3D geometry, we can classify the final output edges e into one of two types. In particular, if Ω_i, Ω_j are the two regions containing e (possibly including the *background region*), then

- Edge e is an *interior edge* if the surface patches $S(\Omega_i)$ and $S(\Omega_j)$ are adjacent in the 3D geometry.
- Edge e is a *silhouette edge* otherwise.

These edge classifications can then be used to style the final vector output (Figure 18, *bottom right*). Note that in general we do *not* require a 3D data structure that explicitly encodes the adjacency of surface patches; this additional information simply helps with certain kinds of stylization.

4.4 Partial Transparency

We can handle scenes with partial transparency by generalizing the region merging strategy discussed in Section 4.2, to a *layer merging strategy*. Layer merging proceeds in several steps:

- During ray tracing (Section 4.1), rather than stop at the first hit, we compute a sequence of surface patches intersected by the ray through each region Ω_i of the planar map, yielding a list of times $0 < t_1 < \dots < t_k$. This list includes all patches up to the first fully opaque patch (or the background).

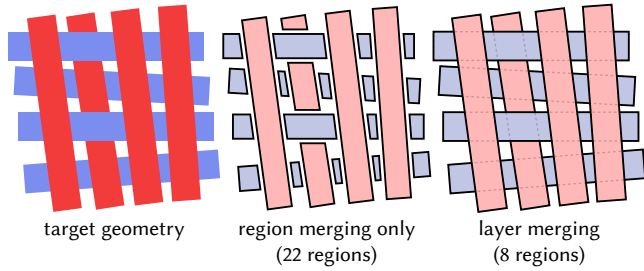


Fig. 19. Even for opaque geometry (left), merging partially-occluded layers, rather than just visible regions, can significantly reduce the size of the output drawing.

- For each region Ω_i , we now consider a “stack” of overlapping layers $\Omega_i^1, \dots, \Omega_i^k$, one for each hit (optionally omitting back-facing layers). Each layer has the same 2D projected geometry as Ω_i , and is viewed as a node in a directed acyclic graph (DAG) that encodes the back-to-front ordering of the regions in this stack.
- To merge neighboring layers in a way that is consistent with visibility, we now greedily merge nodes in this DAG. In particular, for any two neighboring regions Ω_i, Ω_j (i.e., regions sharing an edge), two layers Ω_i^p and Ω_j^q can be merged if (i) they come from the same 3D surface patch and (ii) merging them does not create a cycle in the DAG.
- Once there are no more candidate layer pairs to merge, we emit a single closed 2D path corresponding to each node in the DAG, i.e., each collection of merged layers. (In practice we do this by making a small halfedge mesh that copies each region in the merged collection, and applying the same edge merging operations as in Section 4.2.)

A diagrammatic view of this procedure is shown in Figure 20. Importantly, this strategy properly handles cases where there is no globally consistent ordering, by finding a small set of regions that resolve proper occlusion/visibility (Figure 22, *bottom right*). Moreover, by merging layers together, we get a final output that is much easier to edit than if we simply dice up all patches along all region boundaries (Figure 22, *top left*).

Importantly, note that applying this same layer merging strategy even in the case of fully opaque geometry typically leads to smaller output drawings, since we can merge many visible pieces represented as independent regions into a minimal number of regions. In some cases, this can even *asymptotically* reduce the size of our output, e.g., from quadratic to linear (Figure 19).

4.4.1 Enclosed Regions. The halfedge mesh structure described in Section 3.1.1 nearly encodes the full structure of a planar map, there is an important edge case not covered through the default halfedge structure. Namely, for regions fully contained within another region, a halfedge mesh can not encode which regions contain others, as two regions need to be connected for a halfedge mesh to encode connectivity between the two. While opaque geometry permits workarounds that allow us to ignore these relations, layered transparency requires us to know which regions contain others.

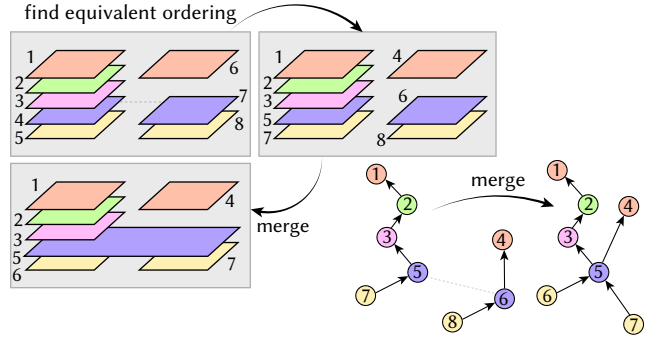


Fig. 20. To obtain a concise, editable vectorization of scenes with partial transparency, we progressively merge stacks of patches associated with each region of the planar map, while maintaining a consistent global ordering. Large, merged pieces can then be correctly drawn in back-to-front order.

To identify these containing relations, we trace a ray outwards from the boundary of each connected component. Once we hit a halfedge not on the boundary of a connected component, we can identify that halfedge’s region as containing the component that originated the ray. This strategy is similar to using the even-odd rule for point-in-polygon tests [Haines 1994], just augmented to identify the innermost containing region for a region.

We perform this identification step before merging any regions for occlusion. As we then simplify the halfedge mesh by deleting edges, we check if new components are created. If a new component is created, we update the containment relations using local connectivity tests. By performing the ray tracing step before region merging, we reduce the number of rays required to identify containing relations.

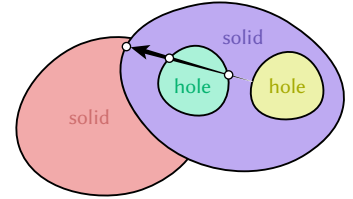


Fig. 21. We trace rays to detect holes and determine layer ordering for nested regions. Our BSP tree already provides exactly the data structure needed to accelerate 2D ray queries.

5 Evaluation, Comparisons, and Examples

We evaluated our system on a variety of different surfaces. Figure 8, Figure 23, and Figure 28 show some of the results of our method. Our method produces clean, topologically sound, and editable curves and regions suitable for stylized output.

5.1 Input Geometry

To evaluate the generality and robustness of our system, we consider scenes comprised of the following types of surface patches:

- **Polygonal** — a patch Ω_i is comprised of one or more planar polygons. We require polygons to be planar so that ray-patch intersection is well-defined (though in principle one could perform ray intersection with nonplanar polygons, *à la* Gillespie et al. [2024]).

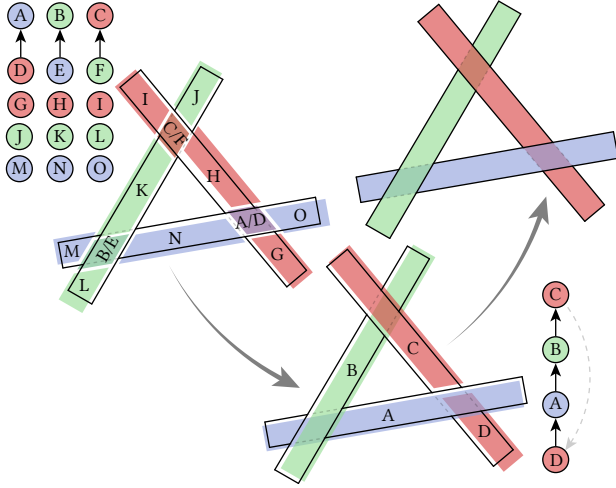


Fig. 22. Even when patches admit no single global ordering, we can find a fairly minimal decomposition into large, editable pieces. Here for instance, the classic counter-example for the *Painter's algorithm* becomes just four regions (*bottom right*) in the final vectorized output. In contrast, naïvely splitting along the planar map boundaries yields a mess of 15 regions that is difficult to edit (*top left*).

- **Catmull-Clark subdivision surface** — a patch Ω_i is the limit surface of one or more polygons under Catmull-Clark subdivision. We approximate the patch boundary by taking the polyline boundary after a finite number of Catmull-Clark subdivisions.
- **NURBS** — a patch Ω_i is a non-uniform rational B-spline surface. We consider this representation as NURBS surfaces allow for non-linear patch boundaries, with existing work for extracting those boundaries available in *Rhino 3D*. Specifically, we use the Make2D command within *Rhino* to project patch silhouettes to curves, which we then export to SVG. Additionally, we export the NURBS patches as high-quality triangle meshes (used for ray tracing), and use *Rhino's* scripting interface to save the camera information used to generate the silhouette curves. We provide the NURBS-derived curves, plus camera configurations and meshed surface patches as a supplemental dataset that can be used to compare against our method in future work.

To demonstrate that our arrangement algorithm accommodates a generic set of curve types, the patch boundaries are composed of polylines, cubic Bézier curves, and elliptical arcs.

5.2 Timing

To test the performance of our system, we ran the vectorization pipeline on a set of polygonal, Catmull-Clark, and NURBS surfaces of varying complexity. Timings were measured on a 2023 Dell XPS laptop with an Intel Core i7 processor and 16GB of RAM. To ensure consistency between tests, we used a maximum depth of 15 for our BSP tree, and we did not use the separating line optimization. For all of the models we tested, our system was able to finish within a reasonable time. For our largest input, consisting of 14486 NURBS

patches, our method took around 50s to complete (excluding the cost of patch boundary calculation in *Rhino*). The method scales approximately linearly with the input size.

If we enable the use of separating lines as described in Section 2.3, we gain a sizable performance increase. On average, enabling separating lines results in a 1.5x to 2x speedup in performance.

Our implementation of the vectorization pipeline is not heavily optimized, and we expect that performance can be improved. Notably, the construction of the BSP tree is done in a serial manner. Since constructing a branch of the BSP tree does not depend on the state of its neighboring branches, it should be feasible to parallelize the construction of the BSP tree.

5.3 2D Arrangements

In order to run a large-scale benchmark of methods, we also considered 2D illustrations in the *scalable vector graphics (SVG)* format. In this context, tools like *Inkscape* and *Adobe Illustrator* often need to construct a planar map for operations like flattening transparency (i.e., turning overlapping partially-transparent regions into corresponding opaque-shaded regions) and more generally for pathfinding/Boolean operations. We hence compare different methods' abilities to correctly resolve the planar graph for vector art found “in the wild.”

As ground truth, we rasterize each example at high resolution. We then run each method that computes a vector planar map, pulling the color for each region from the rasterized image. Finally, we rasterize these re-colored planar maps, and compare them in image space with the ground truth rasterized image. Ideally, the two images should agree pixel-wise, apart from antialiasing at region boundaries (where a rasterizer may adopt different rules for, e.g., antialiasing overlapping vs. abutting shapes). Hence, in the limit as resolution increases (and region boundaries become a vanishingly small fraction of pixels) the two should agree identically. For each method and example we report the percent of pixels that agree with ground truth as a fraction $p \in [0, 1]$.

For the test data set, we generated synthetic examples by transforming icons sourced from Bootstrap, and layering the icons on top of each other. We generated 500 simple examples each consisting of 3-5 layered icons, as well as 100 complex examples consisting of 10-25 layered icons. We compared our BSP arrangement algorithm against the Bentley-Ottmann sweepline algorithm [Bentley and Ottmann 1979], implemented in the industry-standard CGAL package. As we require arrangements of non-linear geometry, we used two variants of the sweepline algorithm. For the first variant, we sample the input curves to a set of polylines and compute the planar map on the polylines using the standard sweepline algorithm. For the second variant, we performed the sweepline algorithm directly on the input Bézier curves using CGAL's bespoke Bézier sweepline implementation.

By running our method and the CGAL algorithms on our synthetic dataset, we are able to show our BSP based planar map algorithm is an improvement on the existing alternatives (Figure 25). On the quality metric, all 3 methods perform identically, consistently producing correct planar maps. However, our method significantly outperforms both CGAL variants in performance and reliability.

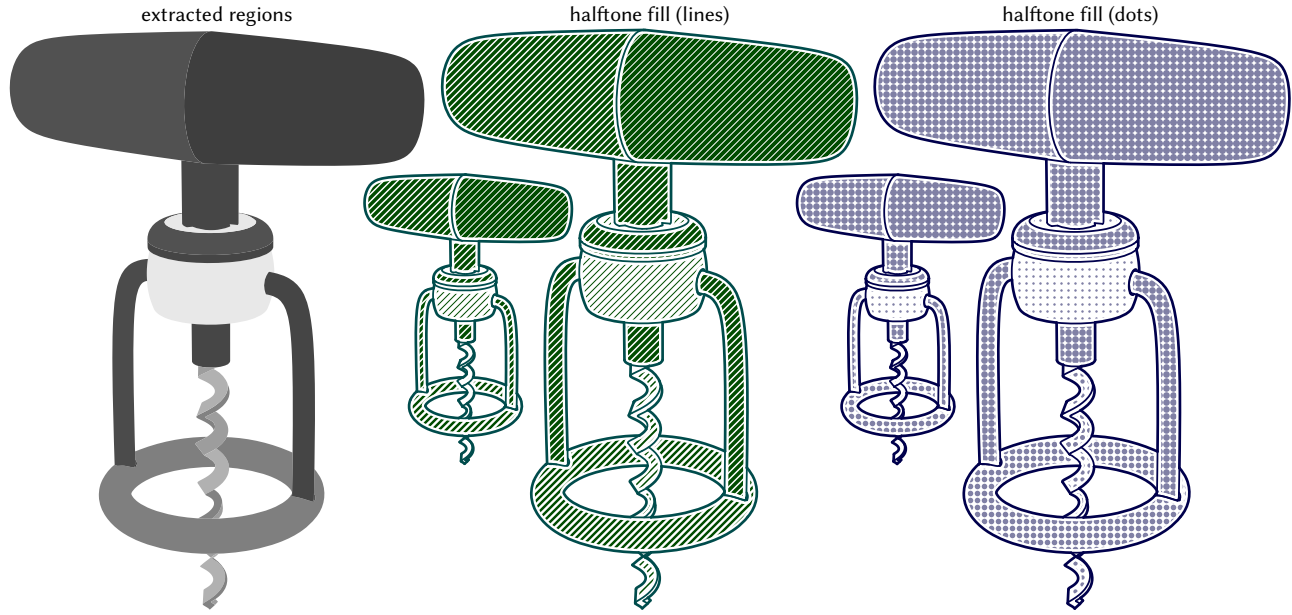


Fig. 23. Vectorized output with clean, well-defined regions enables 3D objects to be stylized using effects that would be difficult or impossible using conventional ray tracing or rasterization. Here, the brightness of extracted regions is used to select halftoning patterns that dynamically scale with the artwork’s display size.

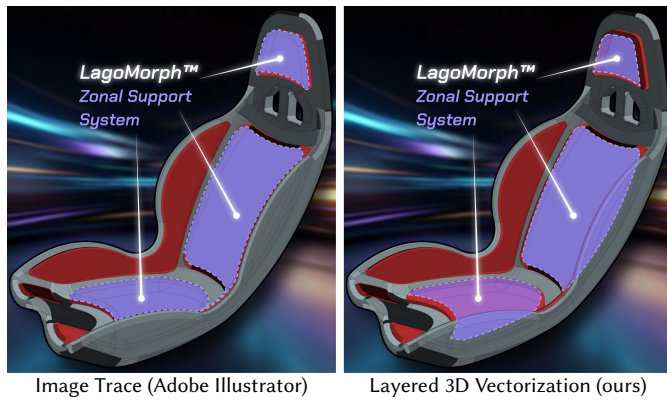


Fig. 24. Our scheme for handling partial transparency can also be used to vectorize additional opaque layers, enabling a graphic designer to highlight occluded features, such as the complete seating pads shown at *right*, that would normally be hidden by standard rasterization-based vectorization techniques (*left*).

Our method consistently ran 62x and 24x faster than the polyline and Bézier approaches respectively. Additionally, our method failed to complete at a rate of only 3%, compared to 48% and 8% for the polyline and Bézier approaches respectively. Of the 3% of cases our method failed to complete, 15 out of 18 crashed due to running out of memory. The other 3 cases were marked as failure due to running into a time limit (chosen to be much longer than the average completion time for the given complexity). Both of these issues were likely caused by input curves overlapping on an interval instead of only intersecting at a finite set of points. These overlapping curves

cause our BSP tree to grow exponentially as depth increases. We do not claim to handle such cases gracefully. However, they will likely succeed given an sufficiently large time and memory budget.

5.3.1 Vector Graphics Editors. Software like *Adobe Illustrator* and *Inkscape* offer powerful tools for Boolean operations on 2D regions—yet do not provide tools to directly construct a planar map from a collection of open and closed curves. Still, we can emulate this functionality by expanding curves into thin regions, taking the union of these thin regions, and then extracting the complementary pieces. Not surprisingly, since this approach is an abuse of the intended use case, results are lackluster—omitting significant regions, even for curves that are not particularly difficult or degenerate. Figure 26 shows one example using Illustrator’s *Pathfinder* tool; experiments with Inkscape gave similar results. Even with the intended use case of boolean operations on closed 2D regions, Illustrator can still produce poor approximations of the output regions (shown in Figure 27).

5.4 3D Comparisons

While we evaluated our planar map algorithm for 2D input data, it is important to consider that the context for our work is in 3D vectorization. Thus we also compared our BSP planar map to the CGAL alternatives in processing input sourced from 3D data for vectorization. In general, the extra complexity from 3D input causes the CGAL algorithms to fail on all but the simplest example scenes. For the scenes where the CGAL alternatives do complete, results are clearly in favor of our algorithm. When run on NURBS surfaces, the CGAL methods produce incorrect and incomplete output, as shown in Figure 29. For subdivision surfaces, the CGAL methods produce correct output (Figure 30), but are significantly slower to

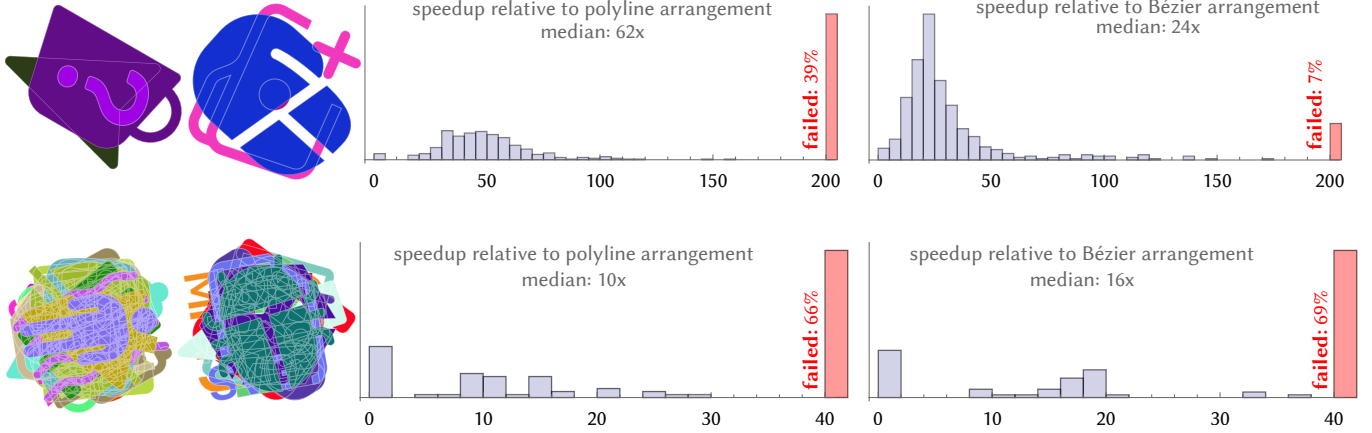


Fig. 25. Even for 2D arrangements, our method is typically one to two orders of magnitude faster than the polyline and Bézier arrangement algorithms implemented in CGAL—and far more robust.

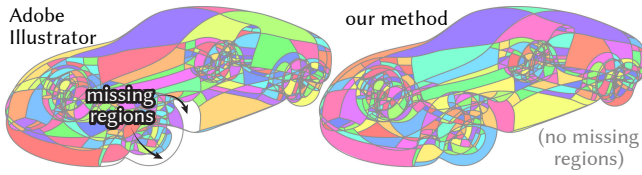


Fig. 26. Commercial vector graphics editors fail to provide a good solution for planar maps, since they do not natively handle “soups” of open curves. If we expand curves into thin regions and take their Boolean union, we obtain a poor planar map approximation that omits large regions (here assigned random colors).

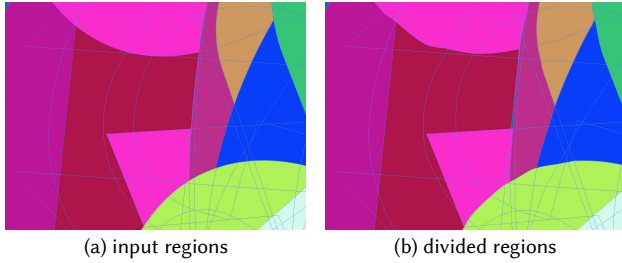


Fig. 27. Illustrator's Pathfinder tool fails to give clean results even in its intended use. We use a zoomed in view of one test case from our 2D arrangement benchmark (left). Using Pathfinder Divide to produce disjoint regions (right) results in major changes to the region boundaries.

complete, taking over 5 minutes in both examples. By contrast, our method always generates correct output and runs quickly, with the equivalent examples only taking 5 seconds.

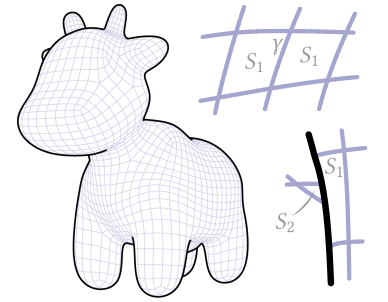
5.5 Robustness of Planar Map Construction

As discussed in Section 3.3, our choice in ordering edges around a vertex of the planar map has a significant impact on robustness. To

quantitatively measure the robustness of our BSP-based ordering, we ran ablation against a dataset of 100 2D arrangements. The method for measuring the quantifying the quality of a given planar map is the same strategy as described in Section 5.3. We compared our method against ordering based on curve tangents and ordering based on intersections with a surrounding loop. Our BSP-based ordering method was the only method of the three to consistently produce perfect planar maps (Figure 31).

5.6 Silhouette Identification

Silhouettes are an essential visual cue in illustration. We identify whether a curve segment γ in our planar map corresponds to a silhouette by simply checking whether the identity of the two neighboring regions Ω_i, Ω_j correspond to neighboring surface patches S_a, S_b in the source geometry. The inset figure shows an example of the silhouettes generated from this approach. This strategy resolves both exterior and interior/open silhouettes.



5.7 Animation

To further evaluate our method, we also considered vectorized *animation*. Animation poses major robustness challenges for vectorization, since it is no longer sufficient for each image to look reasonable: the planar map must stay coherent from frame to frame, to avoid jarring flickering artifacts from frame to frame. Moreover, one inevitably passes through degenerate configurations at moments where visibility changes.

The supplementary video shows a rotation of the NURBS digger from Figure 4, where each frame is vectorized independently, *i.e.*, we do nothing special to enforce frame-to-frame consistency. The

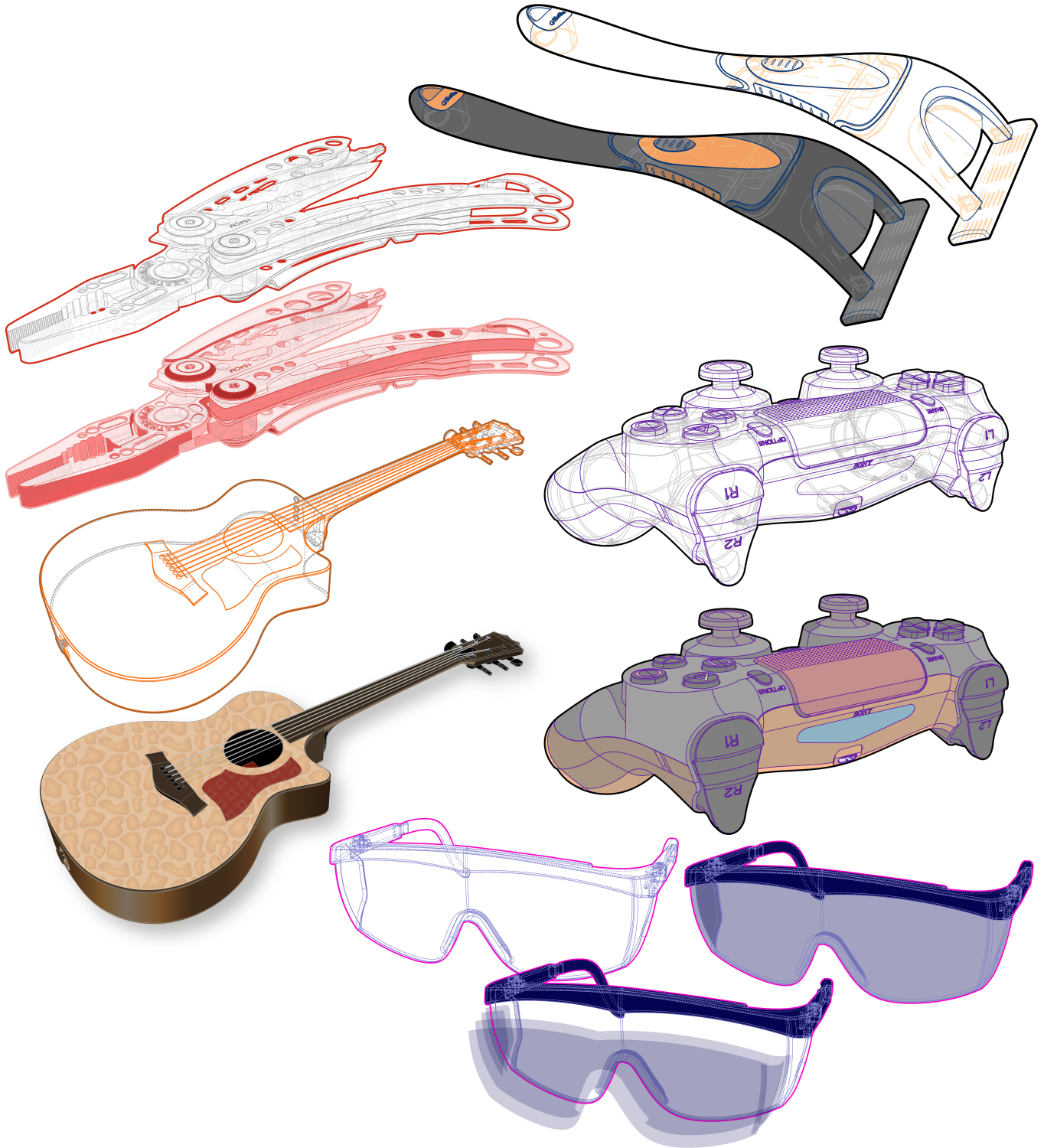


Fig. 28. Gallery of additional results—all previous methods failed to generate output for these models. Each of these objects was originally a NURBS surface, with projected curve contours generated via *Rhino 3D*. Here we take advantage of the vector nature of our output to stylize the final artwork in various ways. The glasses (*bottom*) are exploded to show four distinct layers of transparency.

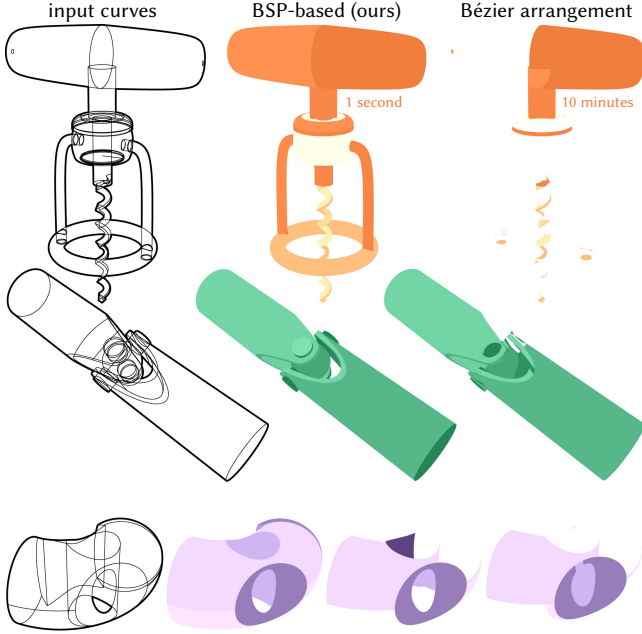


Fig. 29. Classical arrangement methods, including those implemented in the industry-standard CGAL package, failed for most of the 3D examples shown in this paper, resulting in either a crash or unbounded running time. On simpler examples that did terminate, the resulting planar maps have large missing regions—likely due to poor handling of near-coincident and near-tangential curves that commonly arise due to visibility events.

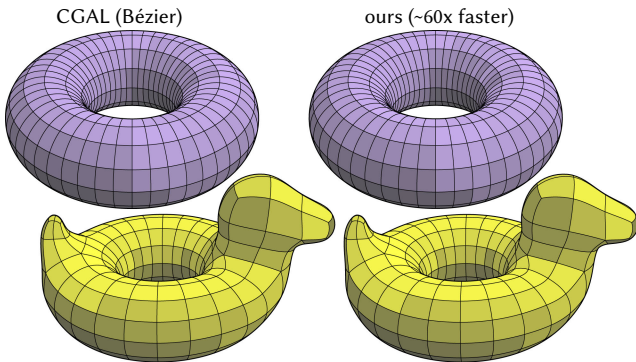


Fig. 30. For the few 3D examples where CGAL's implementation of Bézier arrangements produced correct output, such as these simple subdivision surfaces, our method was 1–2 orders of magnitude faster (and gives visually identical output).

animation shows very few temporal inconsistencies—apart from jumps in shading, since we use an arbitrary point on each visible patch to determine a per-patch color. Treating vectorized animation in a more systematic way (say, explicitly tracing frame-to-frame correspondence between planar regions) is an interesting topic for future work.

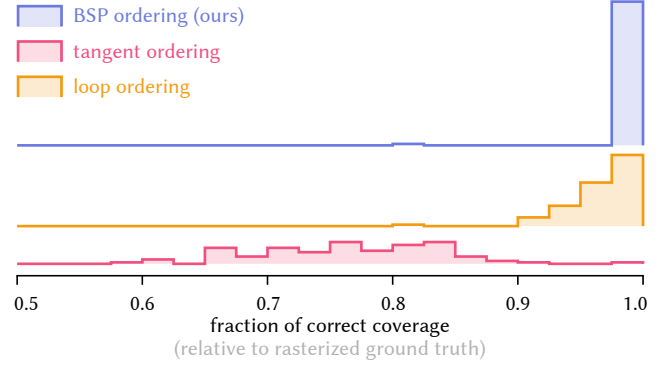


Fig. 31. Over a dataset of 100 challenging 2D arrangements, our BSP-based scheme consistently yields a planar map in better agreement with rasterized ground truth than the more conventional strategy of locally ordering incoming/outgoing curves at each vertex of the planar graph. Here we report the fraction of pixels with the same object ID as the rasterized reference.

5.8 Partial Transparency

As already discussed in Section 4.4, our method can easily handle partial transparency simply by augmenting our raytracing scheme. The ability to handle partial transparency represents an advantage of our method over existing work. The traditional way to resolve visibility when constructing a planar map is to use *quantitative invisibility (QI)*, which uses some elementary observations about relative visibility of neighboring regions to propagate the number of occluders over the planar map. While this is enough information to resolve visibility in the opaque case, partial transparency requires more than the number of occluders. Namely, partial transparency requires the exact identity and ordering of the occluders, as each occluder colors the patches behind it instead of blocking them.

Additionally, we can utilize our partial transparency scheme to generate highly editable output, as shown in Figure 24. All that is needed to produce such output is to treat opaque geometry as if it was transparent. By doing so, we can generate regions for occluded patches, and then merge them to get partially-occluded regions layered on top of each other. This approach is also more robust than the simple painter's algorithm, since our partial transparency method handles circular occlusion properly (Figure 22).

5.9 Reflections

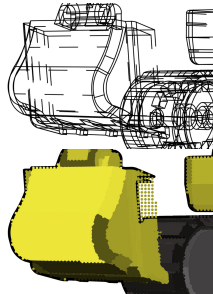
A very challenging example, ignored by essentially all existing vectorization algorithms and software, is vectorizing specular reflections. This example can be especially challenging from the point of view of robustness, since, for instance, reflection lines near the silhouette of a sphere can “bunch up” into highly degenerate planar curves—even for completely reasonable 3D scene geometry. Figure 6 showcases the ability of our method to vectorize specular reflections, robustly handling a large number of degenerate curves.

In order to vectorize specular reflections, we need to make only a few natural changes to our pipeline. First, we take in curves γ_i representing the patch boundaries viewed under specular reflection, instead of just perspective projection. Under spherical reflection,

these boundaries can be computed pointwise by solving a quartic equation [Agrawal et al. 2010]. We then sample the 3D patch boundaries and fit Bézier curves on the point-wise solutions to obtain the patch boundaries under spherical reflection. Second, when ray tracing region identities, we perform a specular bounce and assign the region identity to the first hit of the reflection ray. The ability to extend our visibility model to global illumination effects (reflections, shadows, etc.) represents another advantage of our method over classic strategies such as quantitative invisibility [Appel 1967].

6 Limitations and Future Work

Garbage In, Garbage Out. Planar map construction is only one piece of the broader 3D vectorization pipeline. Though the BSP-based approach significantly increases robustness of region extraction relative to more local approaches, there are of course limits to what we can expect to obtain from very poor input curves. For instance, in our experiments *Rhino* occasionally produced badly broken curves that were nowhere close to forming closed loops (*inset, top*), causing major disagreement with the rasterized reference; here we show rasterized points as small dots (*inset, bottom*). Another common artifact was with the geometry itself: for instance, we found models where an engineer had made multiple coincident NURBS patches, or where models simply had ugly 3D contours (e.g., due to poor approximation of solid models by trimmed NURBS patches). These issues evoke the “leaky pipeline” issue discussed in Section 1, emphasizing the importance of continued work on robustness both upstream and downstream from planar map generation.



3D Intersections and Shadows. Our method focuses on intersection of patches projected into the 2D plane. An orthogonal question is how to handle solid objects that intersect in 3D space. The basic question here is how to reliably evaluate the 3D curve of self-intersection (which our method does not have access to at present). Though we leave this question to future work, some of the strategies we introduce here may point toward general-purpose, robust solutions, e.g., adaptively refining a 3D spatial partition down to a threshold tolerance—rather than attempting the classic approach of attempting to directly parameterize the intersection curve (as done by some CAD packages). Adaptive refinement has already been shown to be useful for discretizing implicit complexes [Ju et al. 2024], which faces similar roadblocks as resolving curves of intersection. Robust handling of 3D intersections would in turn enable visually-critical cues such as shadows, which could be handled by, e.g., computing intersections with *shadow volumes* [Crow 1977]. Nothing then needs to change about the rest of our method—one could simply stylize output regions according to whether the corresponding patches are in/out of shadow.

References

- Amit Agrawal, Yuichi Taguchi, and Srikumar Ramalingam. 2010. Analytical Forward Projection for Axial Non-central Dioptric and Catadioptric Cameras. In *Computer Vision – ECCV 2010*. 129–143.

- Arthur Appel. 1967. The notion of quantitative invisibility and the machine rendering of solids. In *Proceedings of the 1967 22nd national conference*. 387–393.
- Paul Asente, Mike Schuster, and Teri Pettit. 2007. Dynamic planar map illustration. *ACM Transactions on Graphics (TOG)* 26, 3 (2007), 30–es.
- Patrick Baudelaire and Michel Gangnet. 1989. Planar maps: an interaction paradigm for graphic design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 313–318.
- Pierre Bénard and Aaron Hertzmann. 2019. Line Drawings from 3D Models: A Tutorial. *Found. Trends. Comput. Graph. Vis.* 11, 1–2 (Sept. 2019), 1–159. <https://doi.org/10.1561/06000000075>
- Bentley and Ottmann. 1979. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on computers* 100, 9 (1979), 643–647.
- Gilbert Bernstein and Don Fussell. 2009. Fast, exact, linear booleans. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 1269–1278.
- Ryan Capouellez, Jiacheng Dai, Aaron Hertzmann, and Denis Zorin. 2023. Algebraic smooth occluding contours. In *ACM SIGGRAPH 2023 Conference Proceedings*. 1–10.
- Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. 2020. Fast and robust mesh arrangements using floating-point arithmetic. *ACM Transactions on Graphics (TOG)* 39, 6 (2020), 1–16.
- Keenan Crane. 2013. Illustrating Geometry. <https://www.cs.cmu.edu/~kmc Crane/IllustratingGeometry.pdf>. Lecture slides.
- Franklin C Crow. 1977. Shadow algorithms for computer graphics. *Acm siggraph computer graphics* 11, 2 (1977), 242–248.
- Boris Dalstein, Rémi Ronfard, and Michiel Van De Panne. 2014. Vector graphics complexes. *ACM Transactions on Graphics (TOG)* 33, 4 (2014), 1–12.
- Mark Dokter, Jozef Hladky, Mathias Parger, Dieter Schmalstieg, Hans-Peter Seidel, and Markus Steinberger. 2019. Hierarchical rasterization of curved primitives for vector graphics rendering on the GPU. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 93–103.
- Herbert Edelsbrunner and Ernst Peter Mücke. 1990. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.* 9, 1 (Jan. 1990), 66–104. <https://doi.org/10.1145/77635.77639>
- Elmar Eisemann, Sylvain Paris, and Frédo Durand. 2009. A visibility algorithm for converting 3D meshes into editable 2D vector graphics. In *ACM SIGGRAPH 2009 papers*. 1–8.
- Elmar Eisemann, Holger Winnenmöller, John C. Hart, and David Salesin. 2008. Stylized Vector Art from 3D Models with Region Support. *Computer Graphics Forum* 27, 4 (June 2008), 1199–1207. <https://doi.org/10.1111/j.1467-8659.2008.01258.x> <http://artis.imag.fr/Publications/2008/EWHS08/>.
- Christer Ericson. 2004. *Real-time collision detection*. Crc Press.
- Andreas Fabri and Sylvain Pion. 2009. CGAL: The computational geometry algorithms library. In *Proceedings of the 17th ACM SIGSPATIAL international conference on advances in geographic information systems*. 538–539.
- Richard P Feynman. 1988. An outsider’s inside view of the Challenger inquiry. *Physics today* 41, 2 (1988), 26–37.
- Eyal Flato, Dan Halperin, Iddo Hanniel, and Oren Nechushtan. 1999. The design and implementation of planar maps in CGAL. In *International Workshop on Algorithm Engineering*. Springer, 154–168.
- Francisco Ganacim, Rodolfo S Lima, Luiz Henrique De Figueiredo, and Diego Nehab. 2014. Massively-parallel vector graphics. *ACM Transactions on Graphics (TOG)* 33, 6 (2014), 1–14.
- Michel Gangnet, J-C Hervé, Thierry Pudet, and J-M Van Thong. 1989. Incremental computation of planar maps. *ACM SIGGRAPH Computer Graphics* 23, 3 (1989), 345–354.
- GEOS contributors. 2025. *GEOS computational geometry library*. Open Source Geospatial Foundation. <https://doi.org/10.5281/zenodo.11396894>
- Mark Gillespie, Denise Yang, Mario Botsch, and Keenan Crane. 2024. Ray Tracing Harmonic Functions. *ACM Trans. Graph.* 43, 4 (2024).
- Eric Haines. 1994. Point in Polygon Strategies. In *Graphics Gems IV*, Paul Heckbert (Ed.). Academic Press, 24–46.
- Iddo Hanniel and Ron Wein. 2007. An exact, complete and efficient computation of arrangements of Bézier curves. In *Proceedings of the 2007 ACM symposium on Solid and physical modeling*. 253–263.
- Angus Johnson. 2025. *Clipper2*. <https://github.com/AngusJohnson/Clipper2>
- Yiwen Ju, Xingyi Du, Qingnan Zhou, Nathan Carr, and Tao Ju. 2024. Adaptive grid generation for discretizing implicit complexes. *ACM Transactions on Graphics* 43, 4 (July 2024), Article 82. <https://doi.org/10.1145/3658215>
- David H Laidlaw, W Benjamin Trumbore, and John F Hughes. 1986. Constructive solid geometry for polyhedral objects. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. 161–170.
- Chenxi Liu, Pierre Bénard, Aaron Hertzmann, and Shayan Hoshayari. 2023. Contesse: Accurate occluding contours for subdivision surfaces. *ACM Transactions on Graphics* 42, 1 (2023), 1–16.
- James McCann and Nancy Pollard. 2009. Local layering. In *ACM SIGGRAPH 2009 Papers (New Orleans, Louisiana) (SIGGRAPH ’09)*. Association for Computing Machinery, New York, NY, USA, Article 84, 7 pages. <https://doi.org/10.1145/1576246.1531390>

- James McCann and Nancy Pollard. 2012. Soft Stacking. *Proceedings of Computer Graphics Forum* 31, 2 (May 2012), 469–478.
- Bruce Naylor and Lois Rogers. 1995. Constructing Partitioning Trees from Bezier Curves for Efficient Intersections & Visibility. In *Graphics Interface*. Canadian Information Processing Society, 44–44.
- Diego Nehab and Hugues Hoppe. 2008. Random-access rendering of general vector graphics. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 1–10.
- Vivien Nguyen, Matthew Fisher, Aaron Hertzmann, and Szymon Rusinkiewicz. 2024. Region-Aware Simplification and Stylization of 3D Line Drawings. *Computer Graphics Forum* 43, 2 (2024), e15042. <https://doi.org/10.1111/cgf.15042> arXiv:<https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.15042>
- Peter Selinger. 2003. Potrace: a polygon-based tracing algorithm.
- William C Thibault and Bruce F Naylor. 1987. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 153–162.
- Herbert B. Voelcker and Aristides A. G. Requicha. 1977. *Constructive Solid Geometry*. Technical Report Technical Memorandum 25. University of Rochester, Production Automation Project, Rochester, N.Y. <http://hdl.handle.net/1802/26358>
- Georges Winkenbach and David H Salesin. 1994. Computer-generated pen-and-ink illustration. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 91–100.
- Georges Winkenbach and David H Salesin. 1996. Rendering parametric surfaces in pen and ink. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 469–476.
- Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. 2016. Mesh arrangements for solid geometry. *ACM Transactions on Graphics (TOG)* 35, 4 (2016), 1–15.

A Bézier curves

Definition. Bézier curves are a category of curves commonly used within computer graphics. A Bézier curve has an associated degree n , as well as $n + 1$ control points $P_0, \dots, P_n$. Given these control points, we can define the parameterization of our Bézier curve as

$$x(t) = \sum_{i=0}^n \binom{n}{i} t^i (1-t)^{n-i} P_{i,x}, \quad y(t) = \sum_{i=0}^n \binom{n}{i} t^i (1-t)^{n-i} P_{i,y}$$

with our parameter t lying in the domain $[0, 1]$.

Line Intersections. As discussed in Section 2.2, we require only the ability to find all intersections between a curve and a line. For an n th degree Bézier curve, this can be computed by finding all roots of an n th degree polynomial. Suppose our line is expressed parametrically as $L(t) = O + tD$. If we let N be a unit vector orthogonal to D , we can then convert the line to an implicit curve where the function

$$f(x, y) = N_x(x - O_x) + N_y(y - O_y)$$

is 0. If we plug in our Bézier curve parameterization $f(x(t), y(t))$, we obtain the function

$$\hat{f}(t) = N_x(x(t) - O_x) + N_y(y(t) - O_y)$$

The roots where $\hat{f}(t) = 0$ give us the parameter values where our Bézier curve intersects the line. Since $x(t)$ and $y(t)$ are n th degree polynomials, our function $\hat{f}(t)$ is an n th degree polynomial. Thus calculating all intersections between a Bézier curve and a line reduces to computing the polynomial $\hat{f}(t)$ and then solving all roots of an n th degree polynomial. Solving for all roots of an n th degree polynomial is a well studied problem with many well-tested solutions available. As our Bézier curves only take parameters in the domain $[0, 1]$, we filter out all roots outside.

B Ellipses

Definition. An ellipse is the set of points where the sum of the distances to two focal points is constant. Elliptical arcs are an important case for vectorization, describing silhouettes and contours

of conics. Consider a major radius a , minor radius b , center C , and rotation θ . We can parameterize the corresponding ellipse as

$$x(t) = a \cos \theta \cos t - b \sin \theta \sin t + C_x$$

$$y(t) = a \sin \theta \cos t + b \cos \theta \sin t + C_y$$

An elliptical arc is then just the curve $(x(t), y(t))$ where t lies in the domain $[t_1, t_2]$ for some choice of t_1 and t_2

Line Intersections. To compute the intersection of an elliptical arc and a line, we can use the fact that any ellipse can be expressed as the affine transformation of the unit circle centered at the origin. If we reverse the affine transformation to our space, we reduce the problem of intersecting an elliptical arc to that of intersecting the unit sphere. Specifically, this affine consists of a translation by $-C$, then a rotation by $-\theta$, and finally scaling the x and y coordinates by $1/a$ and $1/b$ respectively. If our line is expressed parametrically as $L(t) = O + tD$, then the transformed line can be expressed as $\hat{L}(t) = \hat{O} + t\hat{D}$, where

$$\hat{O} = \begin{bmatrix} \cos(\theta)/a & \sin(\theta)/a \\ -\sin(\theta)/b & \cos(\theta)/b \end{bmatrix} (O - C)$$

$$\hat{D} = \begin{bmatrix} \cos(\theta)/a & \sin(\theta)/a \\ -\sin(\theta)/b & \cos(\theta)/b \end{bmatrix} D$$

The points of intersection are then expressed as the solutions to $\hat{L}(t) \cdot \hat{L}(t) = 1$. Expanding out, we obtain the equation

$$\hat{D} \cdot \hat{D} t^2 + 2\hat{O} \cdot \hat{D} t + \hat{O} \cdot \hat{O} - 1 = 0$$

which can be solved by the quadratic equation. Since an elliptical arc only takes parameter values between $[t_1, t_2]$, we filter extraneous intersections by evaluating the intersections $\hat{L}(t_i)$, converting to the ellipse parameter domain with atan2 , and removing intersections that fall outside the domain.