Vector Graphics Complexes

Boris Dalstein, Rémi Ronfard, Michiel van de Panne

Vector Graphics Complexes

Boris Dalstein*
University of British Columbia

Rémi Ronfard
Laboratoire Jean Kuntzmann,
University of Grenoble & Inria, France

Michiel van de Panne
University of British Columbia

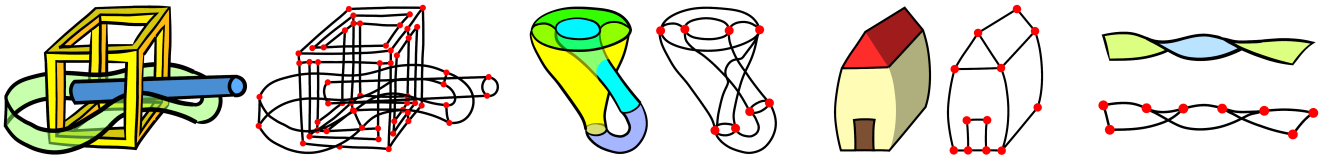


Figure 1: *Vector graphics illustrations made using the vector graphics complex, and their underlying topology.*

Abstract

Basic topological modeling, such as the ability to have several faces share a common edge, has been largely absent from vector graphics. We introduce the vector graphics complex (VGC) as a simple data structure to support fundamental topological modeling operations for vector graphics illustrations. The VGC can represent any arbitrary non-manifold topology as an immersion in the plane, unlike planar maps which can only represent embeddings. This allows for the direct representation of incidence relationships between objects and can therefore more faithfully capture the intended semantics of many illustrations, while at the same time keeping the geometric flexibility of stacking-based systems. We describe and implement a set of topological editing operations for the VGC, including glue, unglue, cut, and uncut. Our system maintains a global stacking order for all faces, edges, and vertices without requiring that components of an object reside together on a single layer. This allows for the coordinated editing of shared vertices and edges even for objects that have components distributed across multiple layers. We introduce VGC-specific methods that are tailored towards quickly achieving desired stacking orders for faces, edges, and vertices.

CR Categories: I.3.4 [Computer Graphics]: Graphics Utilities—Paint systems I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Curve, surface, solid, and object representations;

Keywords: Vector illustration, Topology, Graphics editor

Links: DL PDF WEB VIDEO

1 Introduction

Vector illustrations are widely used to produce high quality 2D drawings and figures. They are commonly based on objects that are assigned to layers, thereby allowing objects on higher layers to obscure others drawn on lower layers. Objects are typically constructed from collections of open and closed paths which are assigned to a single common layer when they are grouped together. Closed paths can be optionally filled by an opaque or semitransparent color in order to represent faces. A rich set of features and editing operations can then be integrated into this framework to yield powerful systems for 2D illustration.

Our work begins with the observation that basic topological modeling is largely absent in vector graphics systems. While 3D modeling systems readily support the creation of geometry having a desired topology, in many vector graphics systems it remains difficult to design objects having edges shared by adjacent faces or vertices shared by sets of incident edges. Our solution is to develop a novel representation which allows users to directly model the desired topology of the elements in a vector graphics illustration.

Another important observation is that vector graphics illustrations often consist of 2D depictions of 3D objects [Durand 2002; Eisemann et al. 2009], with the important consequence that a representation of vector graphics objects as strictly two-dimensional entities, such as planar maps, may be counter-productive. In this context, users may also need to represent aspects of the topological structure of the 3D objects being depicted when creating and editing the visual representation. The topology of the visual objects may therefore not be in correspondence with their 2D geometry, but rather be in correspondence with the 3D geometry of the depicted objects, which are mental entities, constructed by perception [Hoffman 2000]. Such mental visual objects can be represented in an abstract *pictorial space* [Koenderink and Doorn 2008] which is different from both the 2D image space and the 3D world space.

Finally, a third observation is that artists use a variety of techniques that frequently result in non-manifold representations. For example, a flower or tree can be drawn with a combination of strokes and surfaces. As a result, non-manifold, mixed-dimensional objects are the rule in vector graphics, not the exception.

Based on the above observations, we have developed the vector graphics complex (VGC), a novel cell complex that satisfies the following requirements: (a) be a superset of multi-layer vector graphics and planar maps; (b) allow edges to share common vertices and faces to share common edges; (c) allow faces and edges to overlap, including when they share common edges or vertices (d) make it possible to draw projections of 3D objects and their topology without knowing their 3D geometry; (e) represent non-orientable and non-manifold surfaces; (f) allow arbitrary deformation of the geometry without invalidating the topology; (g) offer reversible operators for editing the topology of vector graphics objects; and (h) have the simplicity that would encourage wide-spread adoption.

*dalboris@cs.ubc.ca

2 Related work

Vector graphics systems use a combination of stacked layers of paths [Porter and Duff 1984; SVG Working Group 2011] and planar maps [Baudelaire and Gangnet 1989; Asente et al. 2007], which are the two most common representations in both academic and commercial systems. Stacked layers of paths are fundamentally limited in their ability to model even basic topological constructs, such as joining an edge (or path) to the middle of another edge, or sharing an edge between two faces. We further elaborate on such issues in the following section.

Planar maps [Baudelaire and Gangnet 1989] allow selected sets of intersecting 2D paths to be used as the boundaries for defining closed cells or faces, which can then be independently assigned attributes such as fill color. A difficulty of this approach is that the planar map needs to be recomputed when the original 2D paths are edited. By default, the need to compute a new planar map results in a loss of the attribute information stored with the original faces of the planar map. In practice, it is in fact possible to devise heuristics for establishing correspondences between the new faces and the original faces, thereby allowing the attribute information to be carried over after edits [Asente et al. 2007]. Closely related to planar maps, [Takayama et al. 2013] proposes a curve network representation well-suited for free-form sketching of patches decomposing a 3D mesh, useful for user-guided quad remeshing.

Stacking-based systems use a back-to-front rendering of layers to designate the occlusion relationships among objects. However, illustrations with cycles in the occlusion relations require developing work-arounds or alternate solutions. One approach is to develop local orderings [Wiley and Williams 2006; McCann and Pollard 2009] that allow for the layer orderings to be specific to a particular area of the illustration. For cases where the illustration arises from known 3D geometry, algorithms exist to identify the cycles and automatically split a face so as to break the occlusion cycle [Eisemann et al. 2009]. Another solution that is particularly well suited to the depiction of knots and folds is to implement deformations to the 3D geometry so as to produce a desired local depth ordering as specified by a user [Igarashi and Mitani 2010]. Other work is aimed at the automatic extraction of 2D contiguous faces from underlying 3D geometry [Karsch and Hart 2011].

Our vector graphics complex can also be seen as an extension of stroke graphs [Whited et al. 2010; Noris et al. 2013], that represent the topology of a drawing by nodes (our vertices) and edges. This topological information is used to establish automatic stroke correspondences between two keyframes of a traditional 2D animation, and provide topology-aware interpolation of the strokes. However, they never address the issue of the representation of faces, and since their input is a scanned drawing, they never consider or discuss overlapping edges (even though their data structure would support it) or the usefulness of this structure as an interactive drawing paradigm.

A significant body of work on topological representations is also relevant, including common choices such as, among others: winged-edge and half-edge structures; Nef polygons [Nef 1978]; simplicial complexes, e.g., [De Floriani et al. 2003; De Floriani et al. 2010]; the selective geometric complex and structured topological complex [Rossignac and O'Connor 1989; Rossignac 1997]; radial-edge structures [Weiler 1985]; and the adjacency and incidence framework [Silva and Gomes 2003]. Providing a meaningful comparison with our proposed representation first requires defining the VGC in detail and so comparisons of these representations with the VGC are deferred until after we formally define the VGC, in section 4.4.

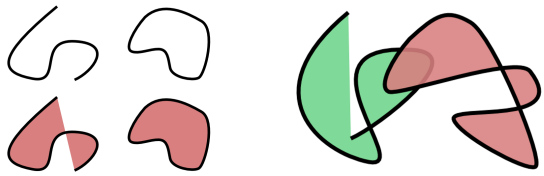
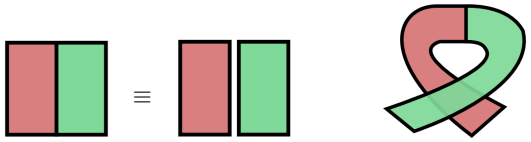


Figure 2: *The "SVG" representation, as used in Illustrator (except for the LivePaint tool) and Inkscape. Left: Open and closed paths, filled or not. Right: Overlapping and self-overlapping paths.*



(a) *Shared edge in SVG* **(b)** *Shared edge + overlapping*

Figure 3: *The limitations of existing representations. SVG cannot represent two faces sharing a common edge as in (a), therefore must duplicate the shared edge. Planar maps can represent shared edges, but cannot represent overlapping faces, thus neither SVG nor planar maps can represent the illustration in (b).*

3 Motivation and overview

In this section, we motivate the vector graphics complex (VGC) and provide an intuition about its structure. This overview provides many of the insights needed to understand and implement the VGC. It also lays the foundation for understanding a formal description rooted in graph theory and combinatorial topology that we provide in the following section.

Let us first recall the traditional vector graphics representation, that we will refer to as "SVG" because of the XML *Scalable Vector Graphics* file specification of the same name. With SVG, a drawing is represented using building blocks called *paths*. A path is typically a list of Bézier control points, that can be either closed or open. It has drawing attributes that indicate how it must be rendered, such as stroke width, stroke color, and fill color, as illustrated Figure 2. Paths are defined independently of each other, which means that if one path is dragged and dropped by the artist on top of another one, they freely overlap and do not interact as shown in Figure 2.

This basic overlapping capability is a very desirable feature, since it allows the artist to freely edit the geometry of the paths and move the objects without any constraints. Nonetheless, there are cases where it would be desirable to model interaction between paths. A canonical example, also described in [Baudelaire and Gangnet 1989], occurs when the illustration represents two shapes that share a common partial contour or edge. In SVG, this must be represented as two independent closed paths, where the common section has exactly the same geometry, as illustrated in Figure 3a.

While common tools such as Adobe Illustrator [Adobe Systems Inc. 2013] or Inkscape [Inkscape 2013] provide convenient tools to *build* such shapes (such as basic duplication or alignement features, or the shape builder tool in Illustrator), the topological information between the two shapes is still not explicitely encoded: the *semantics* of the intended illustration is not correctly represented. In practice, this means that the information about the common portion of the paths is duplicated, and *editing* its geometry is often tedious. Typically, adding Bézier control points or editing tangents must be performed twice (this limitation is demonstrated in the accompanying video). Planar maps and their extension, dynamic planar maps

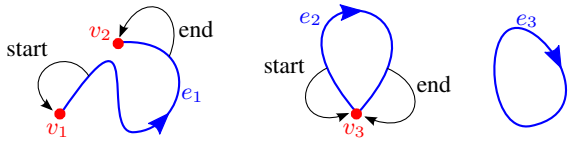


Figure 4: *Two open edges e_1 and e_2, and one closed edge e_3.*

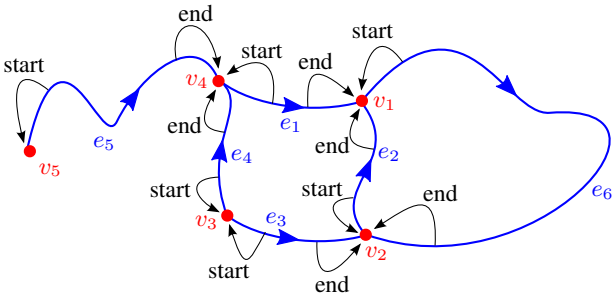


Figure 5: *A VGC composed of vertices and edges only, similarly to a stroke graph. v_1, v_2 and v_4 are each shared by three edges.*

(LivePaint in Illustrator), have been introduced as a solution to this problem. This, however, introduces other compromises, such as the inability to have overlapping faces. While planar maps can faithfully represent the semantics shown in Figure 3a, they cannot faithfully represent the semantics of the illustration shown in Figure 3b. This second figure shares the same topology and can therefore be obtained from Figure 3a via simple editing of the geometry. This limitation seriously impairs artistic freedom and expressiveness.

The vector graphics complex that we present is an alternative solution, much closer to the spirit of SVG. Notably, it retains the ability to represent overlapping objects, and hence it is able to faithfully capture the semantics of the illustration shown in Figure 3b without duplicating any geometric information for the common section.

Whereas in SVG the building block is the *path*, the VGC has building blocks called *cells*, of which there are three types: *vertices*, *edges* and *faces*. A **vertex** is simply a 2D point on the canvas, typically located where strokes meet or end. It can have drawing attributes such as a color and a radius size, but most often you would prefer not to display it at all (just use it as a building block for edges and faces). An **edge** is similar to an SVG path: it defines an oriented 2D curve on the canvas, for instance using Bézier control points. The significant difference with SVG paths is that *edges do not necessarily exist independently from other edges*. For instance, *open edges* must start at a *start vertex*, and end at an *end vertex*. These vertices are stored as pointers in the edge implementation, as illustrated in Figure 4. Therefore, as illustrated in Figure 5, two or more open edges can be connected to each others by sharing a common vertex, and manipulating this vertex would affect all incident edges. So far, this representation is identical to *stroke graphs* [Whited et al. 2010], except that we do not order the incident edges counter-clockwise around a vertex. In addition, we define the notion of *closed edge*, which is an edge with no boundary vertex, as illustrated in Figure 4 (right). We note that it is allowed for an open edge to have its start vertex be equal to its end vertex. At the contrary to some existing representations, we only consider this to be a special case of open edge and *not* a closed edge.

Another difference with SVG paths is that VGC edges do not have a color filling attribute: filling is done via the creation of *faces*, an entity not supported by stroke graphs. A **face** is defined by its boundary via what we call *cycles*. A cycle is a a list of (e, β) pairs

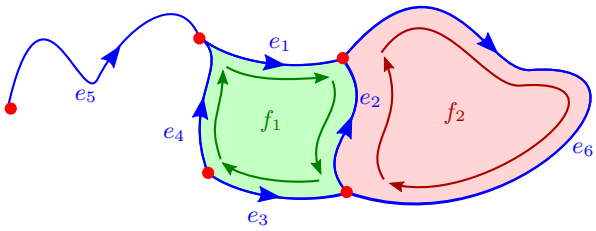


Figure 6: *Two faces f_1 and f_2 each defined by one cycle. The cycle defining f_1 is $[(e_1, \text{true}); (e_2, \text{false}); (e_3, \text{false}); (e_4, \text{true})]$, while the cycle defining f_2 is $[(e_2, \text{true}); (e_6, \text{true})]$.*

that we call *halfedges*. Here, e represents an edge and β is a boolean indicating whether the edge should be considered with its intrinsinc orientation (from start to end, $\beta = \text{true}$), or with the opposite orientation (from end to start, $\beta = \text{false}$), as illustrated in Figure 6. Using several cycles for the same face makes possible to specify holes in the face. Finally, we define the notion of *Steiner cycle*, that is a cycle not defined by a list of halfedges, but defined by a single vertex. A Steiner cycle makes possible to connect the end vertex of an edge to the interior of a face.

An artist creates edges and vertices by drawing strokes. He can choose whether intersections of strokes with existing edges must generate a new vertex and split the incident edges; or must simply ignore the intersection and create overlapping edges, not topologically connected. Then, the artist can freely sculpt the geometry of edges, or drag and drop cells. Also, he can use topological operators, for instance to *glue* two vertices or edges together, or *cut* a face into two faces by inserting a new edge. We refer the reader to the accompanying video for a demonstration of this drawing paradigm. The cells are globally depth-ordered in a doubly-linked list, and we provide intuitive tools to alter this ordering, for instance to decide which face is behind the other in Figure 3b.

4 Vector graphics complex

The vector graphics complex (VGC) is a non-manifold boundary representation for 2D vector graphics, formally defined as a colored incidence graph. In this section, we first describe the topology and geometry of this complex, and then propose an implementation. Finally, we assess its relevance by comparing it with existing non-manifold representations.

4.1 Topology

This section formally presents the theoretical foundations of the topology of a VGC. It does assume a background in graph theory, and a basic understanding of combinatorial topology. It is provided for completeness, but the unfamiliar reader may safely skip it: the actual implementation of a VGC and the remaining of the paper does not assume understanding of this section.

Cell The topology of a VGC is composed of entities called *cells*. There exists three types of cells: *vertices*, *edges*, and *faces*, as illustrated in Figure 7.

Incidence graph Each cell c is a colored node of a directed acyclic graph called the *incidence graph*, illustrated Figure 8. The color indicates the type of the cell: *vertex*, *edge*, or *face*. To avoid

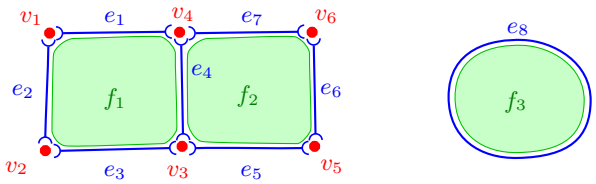


Figure 7: *Two connected squares and an isolated disc represented as a single VGC with 17 cells: six vertices v_1 to v_6, eight edges (e_1 to e_7 are open, e_8 is closed), and three faces f_1 to f_3.*

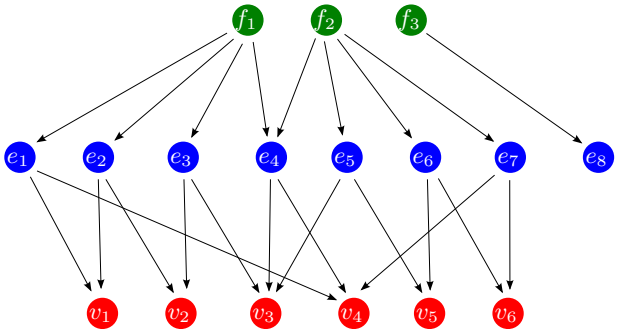


Figure 8: *The incidence graph (without semantics) corresponding to the VGC illustrated in Figure 7. For clarity, we do not show here the pointers from faces to vertices (can be deduced by transitivity).*

confusion with the cell type "edge", the oriented edges of the incidence graph are called *pointers*. Thus, each cell c "points" to other cells, and the set of these pointed cells is called the *boundary* of c, denoted ∂c. The relation $c \in \partial c'$ is a strict partial order. We have:

- irreflexivity: $c \notin \partial c$
- transitivity: if $c \in \partial c'$ and $c' \in \partial c''$ then $c \in \partial c''$
- asymmetry (implied by first two): if $c \in \partial c'$ then $c' \notin \partial c$

Ambiguity of the incidence graph Unfortunately, this incidence graph does not encode enough information for our needs. Indeed, rendering a 2D area of the plane defined by its possibly self-intersecting boundary is traditionally done via the computation of *winding numbers*, which requires as input one or several *closed curves* ("polygon contours" in OpenGL [Shreiner et al. 2004]). The incidence graph only gives a *set* of boundary edges, and converting this set into closed curves is an ambiguous operation: it might involve repeating edges, and the choice of the repeated edge affects the rendering. This issue is illustrated in Figure 9 with the example of a Möbius strip. To make the rendering of the VGC easier and consistent accross implementations, and to give the artist the freedom to choose which closed curves must be used, we do not want this disambiguation to be automatically computed via geometric analysis of the edges. Rather, we directly encode this information in the incidence graph.

Semantic boundary To do this formally, we add a color to each pointer, that we call the *semantics* of the pointer. For instance, an open edge has exactly two pointers colored *start* and *end* (each pointing to a cell of type *vertex*) that defines a topological orientation for the edge, as illustrated in Figure 10. Implementation-wise, we are just saying here that a pointer has a variable name. We define the *semantic boundary* of a cell c as the set of its colored pointers, denoted $\hat{\partial} c$. For instance in Figure 10, we have $\hat{\partial} e = \{(v_1, \text{start}), (v_2, \text{end})\}$, while $\partial e = \{v_1, v_2\}$.

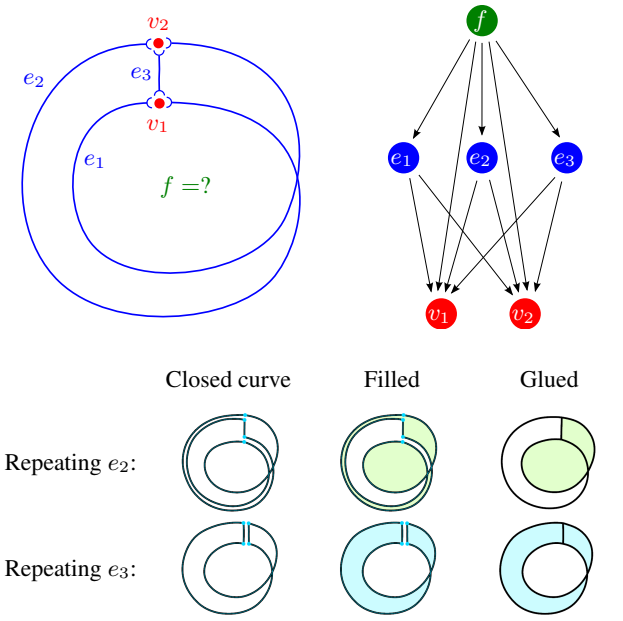


Figure 9: *Top: The incidence graph (without semantics) of a Möbius strip. Bottom: To render it, one needs to decompose its boundary into a closed curve. This requires to repeat (at least) one edge, and this choice is ambiguous: repeating e_2 or repeating e_3 give different rendering results when filling the face using the even-odd winding rule.*

Figure 10: *A VGC composed of three cells (two vertices and one edge), and its incidence graph. The boundary of e is $\partial e = \{v_1, v_2\}$ while its semantic boundary is $\hat{\partial} e = \{(v_1, \text{start}), (v_2, \text{end})\}$. This additional semantics defines a topological orientation for the edge.*

Hence, the topology of a VGC is formally defined as a colored directed multigraph $\mathcal{G} = (C, \hat{\partial})$, where $C \subset \mathbb{N} \times \mathbb{N}$ is a set of colored cells (a cell has an id and a type), and $\hat{\partial} : C \longrightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N})$ is a map giving for each cell c the set of its colored pointers $\hat{\partial} c$ (a pointer has a pointed id and a semantics). Obtaining ∂c from $\hat{\partial} c$ is simply a projection onto the first component of the pair, an operation that can be interpreted as "forgetting the semantics". As usual with projections, this is an irreversible operation: it is not possible to resynthesize the semantics from ∂c, because it is an ambiguous operation, as illustrated in Figure 9.

Helper Figure 11 (top) describes the semantics we would like to encode in the incidence graph of a Möbius strip. Obviously, this is not practical, even more than formally, this semantics must be encoded by a single integer per pointer. Instead, we introduce entities that we call *helpers*, that are a higher level description of this semantics, and can be seen as "meta-pointers": in the same way that a pointer points to a cell with semantics, a helper points to a set of cells with semantics. Figure 11 (bottom) illustrates this concept. Appendix A proves that this description using helpers can be converted into a proper colored incidence graph.

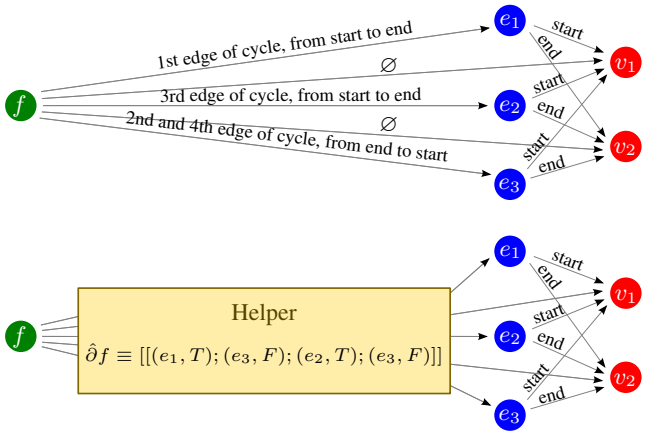


Figure 11: *Top: The incidence graph (with semantics) of a Möbius strip. The semantics disambiguates which edge must be repeated twice, and specifies the traversal orientation of the cycle. Bottom: The same semantics expressed with a helper. Here, T denotes the boolean* true*, while F denotes* false*. By writing "$\hat{\partial} f \equiv$", we mean that it is a convenient description of the semantic boundary of f: a single cycle expressed by a sequence of edges with a specific orientation.*

Topological constraints In order to be *valid*, the semantic boundary of the cells of a VGC must satisfy some constraints. We decribe these constraints in the rest of this section, alongside a description of the semantics we attach to each type of cell.

Vertex A vertex v has a void boundary $\partial v = \hat{\partial} v = \varnothing$.

Edge An edge is either *open* or *closed*.

- An open edge e has two pointers to vertices, one colored *start* and the other colored *end*. We refer to them as $v_{\text{start}}(e)$ and $v_{\text{end}}(e)$, possibly equals.
- A closed edge e has a void boundary $\partial e = \hat{\partial} e = \varnothing$.

Halfedge A halfedge $h = (e, \beta)$ is a pair composed of an edge e and a boolean β representing a chosen orientation of the edge. If e is a closed edge, then h is said to be a closed halfedge. If e is an open edge, then h is said to be an open halfedge, and we define:

if $\beta =$ `true`			if $\beta =$ `false`		
$v_{\text{start}}(h)$	$=$	$v_{\text{start}}(e)$	$v_{\text{start}}(h)$	$=$	$v_{\text{end}}(e)$
$v_{\text{end}}(h)$	$=$	$v_{\text{end}}(e)$	$v_{\text{end}}(h)$	$=$	$v_{\text{start}}(e)$

Cycle A cycle γ is either:

- A single vertex v, or
- A sequence $(h_i)_{i \in \{1,..,n\}}$ of $n > 0$ halfedges satisfying:
  - h_1 is a closed halfedge and $\forall i \in \{1, .., n\}, h_i = h_1$, or
  - $\begin{cases} \text{all halfedges are open, and} \\ v_{\text{start}}(h_1) = v_{\text{end}}(h_n), \text{ and} \\ \forall i \in \{1, .., n-1\}, \;\; v_{\text{end}}(h_i) = v_{\text{start}}(h_{i+1}) \end{cases}$

There are no other restrictions. For instance, a cycle may repeat any number of vertices or edges with arbitrary orientation. A cycle is called a *Steiner cycle* when it is defined as a vertex, called a *simple cycle* when it is defined as (possibly repeated) closed halfedge, and called a *non-simple cycle* otherwise (i.e., when defined as a sequence of open halfedges).

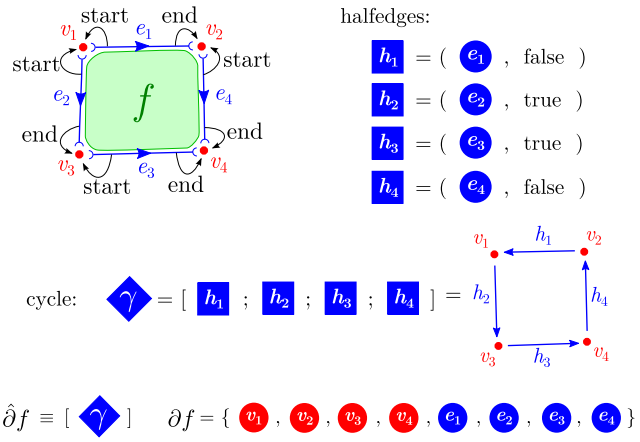


Figure 12: *The complete description of the topology of a VGC representing a square with 4 vertices, 4 (open) edges, and 1 face.*

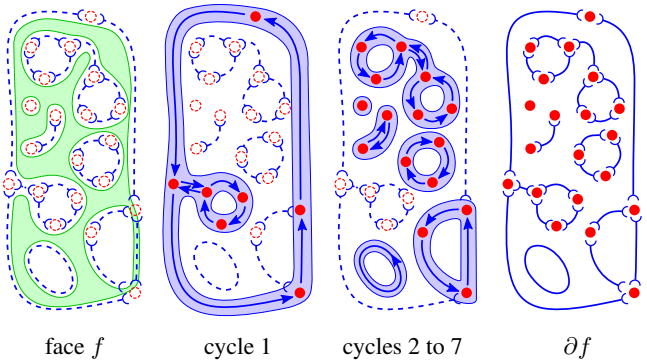


Figure 13: *A valid face, with seven cycles. Cycle 1 represents its external boundary (including a "crack"), and the six other cycles represent holes (one of them being a single missing point in the face, defined by the Steiner cycle).*

Face The semantic boundary of a face f is defined by a sequence $(\gamma_i)_{i \in \{1,..,m\}}$ of $m \geq 0$ cycles. There are no other restrictions. For instance, even though a face without cycle is rather useless for vector graphics, it is valid and theoretically justified in [Dalstein et al. 2014]. Most other oddities, such as repeating an edge three times or more are not only valid and theoretically justified, but actually useful (cf. Figure 19, middle and right). The cycles must not necessarily be disjoint: they can share common vertices or edges (cf. Figure 13). The boundary ∂f of a face is the set of all the edges and vertices involved in the cycles. Intuitively, if only one cycle is given, then it defines the contour of the face. If more than one is given, then it defines holes inside this face. Figure 12 illustrates the topology of a face with a single cycle. A face can represent a non-orientable surface, like the Möbius strip in Figure 11.

4.2 Geometry

The previous section defined the topology of a VGC, a combinatorial structure representing the neigbourhood relationships between abstract cells. This section defines the geometry of a VGC, i.e., how each cell c is realized as a pointset $|c| \subseteq \mathbb{R}^2$.

Vertex The geometry of a vertex v is a point $p_v \in \mathbb{R}^2$, and we define $|v| = \{p_v\}$.

Open edge The geometry of an open edge e is a continuous curve $\Gamma_e : [0, 1] \to \mathbb{R}^2$, satisfying $\Gamma[0] = v_{\text{start}}(e)$ and $\Gamma[1] = v_{\text{end}}(e)$. We define $|e| = \Gamma_e((0, 1))$.

Closed edge The geometry of a closed edge e is a continuous closed curve $\Gamma_e : \mathcal{S}^1 \to \mathbb{R}^2$, where $\mathcal{S}^1$ is the unit circle. We define $|e| = \Gamma_e(\mathcal{S}^1)$.

Face The geometry of a face f is defined by a *winding rule* $R \subseteq \mathbb{N}$, from which can be defined a pointset $|f| \subseteq \mathbb{R}^2$ that we detail in the remainder of this paragraph. For each simple or non-simple cycle γ of a face f, we define the closed curve $\Gamma_\gamma : \mathcal{S}^1 \to \mathbb{R}^2$ by concatenating the functions Γ_e of all edges e involved in the cycle. We denote by $|\gamma| = \Gamma_\gamma(\mathcal{S}^1)$ the set of points $p \in \mathbb{R}^2$ belonging to the cycle. If γ is a Steiner cycle defined by v, we simply define $|\gamma| = |v|$. We define $|\partial f| = \bigcup_{\gamma \in \hat{\partial} f} |\gamma|$, and we note that it is equal to $\bigcup_{c \in \partial f} |c|$. For each $p \in \mathbb{R}^2 \backslash |\gamma|$, Γ_γ defines a winding number $N_\gamma(p) \in \mathbb{N}$ (cf. [Edelsbrunner and Harer 2010, p12]). If γ is a Steiner cycle, we define $N_\gamma(p) = 0$. We define, for each $p \in \mathbb{R}^2 \backslash |\partial f|$, the total winding number $N_f(p) = \sum_{\gamma \in \hat{\partial} f} N_\gamma(p)$ obtained by summing the winding numbers for each cycle of f. Finally, $|f|$ is defined as the pointset $\{p \in \mathbb{R}^2 \backslash |\partial f| \text{ s.t. } N_f(p) \in R\}$. Our implementation uses the OpenGL GLU polygon tesselator [Shreiner et al. 2004] with the even-odd winding rule, but other rules could be considered as well.

4.3 Implementation

Despite the possibly discouraging formalism, the implementation is surprisingly easy and we present here our C++ class structure.

```
class Cell { set<Cell*> star; };
class Vertex: public Cell { Point p; };
class Edge:   public Cell { Vertex * start;
                            Vertex * end;
                            Curve curve;    };
struct Halfedge { Edge * edge; bool b; };
struct Cycle { Vertex * steiner;
               vector<Halfedge> halfedges; };
class Face: public Cell { vector<Cycle> cycles; };
```

As detailed in Section 4.1 and 4.2, a vertex is simply a 2D position, an edge is a 2D curve pointing to its ordered end vertices (if any), and a face is a list of cycles, where a cycle is either a list of halfedges or a single vertex. Checking if an edge `e` is open is simply done via `if(e->start)` (`e->start` and `e->end` are both `NULL` for closed edges). Similarly, checking if a cycle `g` is a Steiner cycle is simply done via `if(g->steiner)`, in which case the vector of halfedges is empty.

The *star* of a cell c is defined by the set of cells whose boundary contains c, i.e. $\text{star}(c) = \{\ c' \mid c \in \partial c'\ \}$. While not stricly required (redundant information), finding the star of a cell is a very frequent query, and storing it in the datastructure makes this query constant time instead of linear in the number of cells (the same way as implementations of directed graphs often contains back-pointers to parent nodes). However, we do not store any semantics in the star and store them indistinctivily in a set. If semantics is needed, then the semantic boundary of the star cells of can be inspected to retrieve it. We tried to encode semantics for the star, but found that it makes maintaining the consistency of the data structure much harder, and that it was not worth the marginal efficiency gain.

We do not described the `Curve` class since it is up to each application to decide on a curve representation adapted to its specific context. For instance, an existing vector graphics system with extensive support of Bézier curves may use Bézier curves. For our prototype, we opted for a simpler dense polyline representation. On top of this core structure, more drawing attributes can be added for fine control on rendering. For instance, we added vertex radius, variable edge width, cell color (possibly transparent), and edge junctions style (mitre join or bevel join).

4.4 Comparison with other representations

We have already discussed (cf. Section 2) the limitations of the SVG representation [SVG Working Group 2011], the limitations of planar maps [Baudelaire and Gangnet 1989; Asente et al. 2007] (also shared by Nef polygons [Nef 1978]), and the connection between the VGC and stroke graphs. In this section, we compare the VGC to representations that belong to the the realm of topological modeling. We took inspiration from these existing topological structures, and adapted them to the problem at hand, resulting in a *pictorial space* that combines the genericity of non-manifold non-orientable 3D topology with the flexibility of 2D geometry. A general observation is that in 2D, there is no need to explicitly describe the geometry of surfaces, since it is completely described by the geometry of its boundary and a winding rule. We take advantage of this in order to propose a simpler ad-hoc data structure.

Well-known topological representations include the winged-edge, half-edge and quad-edge data structures, combinatorial maps, and cell-tuples. These are not suitable, since they cannot represent non-orientable surfaces. The necessity to represent such topologies comes from our requirement to have a data structure closed under the *glue* operation (more detail in Section 5). However, we note that Guibas and Stolfi [1985] define an edge algebra similar to ours.

The selective geometric complex and the structured topological complex [Rossignac and O'Connor 1989; Rossignac 1997] are both a source of inspiration for the VGC and hence have a lot in common with it. However, even though the STC cyclically orders the faces around an edge, neither the SGC nor the STC cyclically orders the edges bounding a face. Such cycles are required for the unambiguous computation of winding numbers, thus the SGC and STC as currently defined are not directly suitable for our purpose.

Suitable representations would include the radial-edge structure [Weiler 1985], extensions and/or specialization [Gursoz et al. 1990; Lee and Lee 2001; Marcheix and Gueorguieva 1995], as well as Nef polyhedra [Granados et al. 2003; Nef 1978], generalized maps [Lienhardt 1994], and the handle-cell data structure [Pesco et al. 2004]. However, these representations are more complex to understand and implement than the VGC, mainly because they target 3D CAD applications and hence have a richer structure to support 3D geometric queries and boolean operations that are less a concern for us. [Gursoz et al. 1991] describes a representation very similar to the VGC, but the authors do not provide any details on an actual data structure, and only consider planar surfaces, and as such do not consider edges shared more than two times by the same face.

Simplicial complexes [De Floriani et al. 2010] are limited to faces bounded by exactly three edges, which is not expressive enough for vector graphics. They can be decomposed into nearly manifold parts equivalent to our cells [De Floriani et al. 2003], but this still requires the underlying triangulation, unlike our approach. The adjacency and incidence framework described in [Silva and Gomes 2003] is a pure incidence graph, and then as we discussed earlier does not contain enough information to disambiguate winding numbers, as required for the 2D rendering of our faces. The VGC extends it with *semantics* to solve this issue.

5 Topological operators

Since topology lies at the heart of the vector graphics complex, and that we aim at porting topological modeling into the the realm of 2D vector graphics, it is highly desirable to be able to manipulate in an intuitive fashion this topology, using *topological operators*. Traditionally, such operators are described as *Euler operators*, since as a safety check one ensures that they are compatible with an *Euler formula*, linking the number of cells of each type, and topological quantities such as the number of connected components and the number of holes. Designing an Euler formula in the case of our non-manifold, non-orientable, mixed-dimensional objects is likely possible and definitely interesting, but it is far from trivial and rather irrelevant as a safety check. Instead, one just has to ensure that the *topological constraints*, or *invariants* using a programming terminology, are preserved. These invariants for our implementation are:

- Vertex: no topological invariants.
- Edge: the `start` and `end` vertices are either both non-`NULL` valid pointers (open edge), or both `NULL` (closed edge).
- Halfedge: `edge` is a non-`NULL` valid pointer.
- Cycle: one of the following is true:
  - `steiner` is a valid non-`NULL` pointer and `halfedges` is empty.
  - `steiner` is `NULL`, `halfedges` has a size $n > 0$, and one of the following is true:
    * `halfedges[0]` is a valid closed halfedge, and for all $i \in \{1, \ldots, n-1\}$,
      ```
      halfedges[i] == halfedges[0]
      ```
    * `halfedges` only contains valid open halfedges, and for all $i \in \{0, \ldots, n-1\}$,
      ```
      halfedges[i].end() ==
            halfedges[(i+1) % n].start()
      ```
- Face: Every cycle in `cycles` is valid.

In addition, since our implementation includes the backpointers `star`, every cell c must satisfy:

- $\forall c' \in \partial c,\ c \in c'.\text{star}$
- $\forall c' \in c.\text{star},\ c \in \partial c'$

A key feature of these invariants is that they can be verified efficiently and robustly without geometric computations. In this section, we informally present the reversible operators create/delete, glue/unglue and cut/uncut. They provably maintain all topological invariants and are detailed in [Dalstein et al. 2014]. They can be intuitively combined together by the artist to achieve the intended topology. In fact, gluing and cutting are not only *useful* operations, but have theoretical roots in algebraic topology. For instance, the proof of the classification of closed two-manifolds involves "cutting" the given manifold until the manifold is represented as a *fundamental polygon*. By gluing together the edges of this polygon, we re-obtain the original manifold. The VGC is a superset of fundamental polygons and is furthermore closed under gluing, which proves that the VGC can represent any closed two-manifold, including non-orientable surfaces such as the Klein bottle in Figure 1. More theoretical considerations show that the VGC can represent any regular non-manifold two-dimensional topological space [Dalstein et al. 2014].

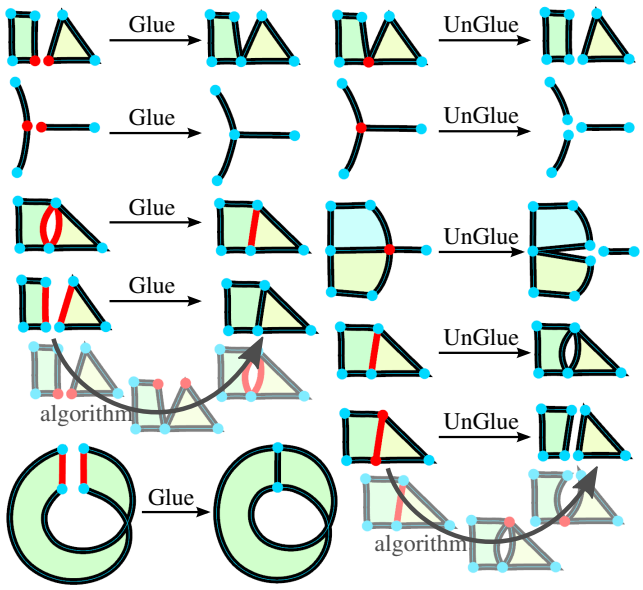


Figure 14: *Examples of glue and unglue operations on vertices, edges, and a set of cells (bottom-right).*

5.1 Creation and deletion operators

Creating a vertex or a closed edge does not requires special care. Creating an open edge requires refering to existing start and end vertices. These topological operators are automatically invoked when the user draws a stroke, as detailed in Section 7. Creating a face requires as input its list of valid cycles, and obtaining these valid cycles from intuitive user input is still an open problem. Currently, we ask the user to select a set of edges, which is automatically converted to cycles. This is not only tedious for the user, but more importantly we recall that this conversion is ambiguous. When ambiguity is detected, we abort the operation. To achieve faces such as the Möbius strip in Figure 9, one option for the user is to explicitly draw the repeated edge twice (left), fill this face (middle), then glue together the two repeated edges (right). We leave as future work the automatic detection of potential faces, which would enable "click to fill" tools that are common for faces in planar maps.

To delete a cell, we first recursively delete its star, otherwise the VGC would become invalid. We refer to this topological operation as a "hard delete". However, by default our delete command enacts a "smart delete", whose semantics are designed to better reflect a user's intentions. If we consider the case of a vertex, v, with two incident edges, e_1 and e_2, and a user that chooses to "delete" v, the intended outcome is more likely to be a single longer edge e_3 that is the geometric union of v, e_1 and e_2, as opposed to an alternative scenario that deletes each of v, e_1, and e_2. The intended outcome is given by the topological operation "uncut at v", as will be detailed later. However, it may not always be possible to uncut at a given vertex v, such as is the case when there are three or more incident edges. The semantics of our "smart delete" are thus defined by "uncut if possible; otherwise, hard delete".

5.2 Glue and unglue operators

The glue operator on vertices and edges, as well as the unglue operator on vertex, edge, or set of vertices and edges are illustrated in Figure 14. Gluing two vertices is the equivalent of the operation *join* in existing vector graphics software, where two paths are appended by gluing together two selected path end-nodes. How-

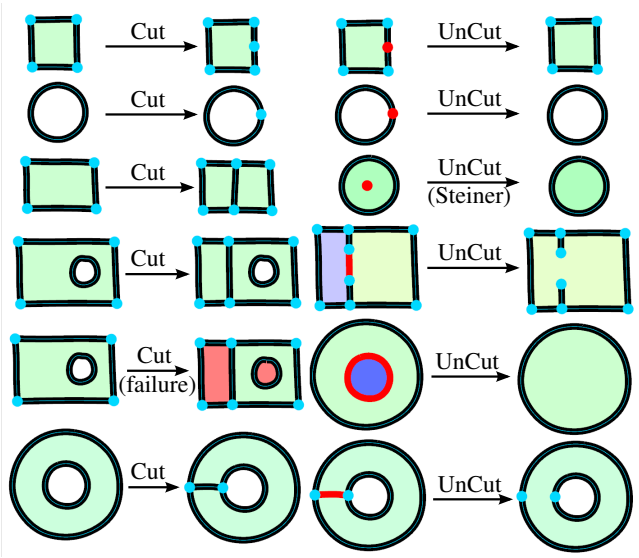


Figure 15: *Examples of cut and uncut operations on vertices and edges. The third operation on the right column illustrates uncutting a Steiner vertex from a face. It is topology equivalent to the fifth operation on the same column. The fifth example on left column is a failure case, when the cut algorithm transfers the "hole cycle" to the wrong face (shown in red). This happens because the cut operator is actually ambiguous and disambiguation require geometric heuristics to capture the user's intent.*

ever, with such a classical *join* operation, the selected end-nodes are transformed into a middle Bézier control point that cannot be *join*ed again to create three-way junctions. In contrast, the VGC is closed under the glue operation: any two vertices, or two open edges, or two closed edges can *always* be glued. In fact, gluing two edges is ambiguous and is only a convenient operation provided for the user. Our implementation uses a simple heuristic to predict the most relevant orientation for the selected edges and then calls the unambiguous *glue halfedges* topological operation. This first glues their respective start and end vertices together, and only then glues the edges together. The unglue operation is the reverse operation, where a vertex or an edge is duplicated as many times as necessary. Ungluing a vertex involves first ungluing its incident edges.

5.3 Cut and uncut operators

The results of the cut and uncut operators are illustrated in Figure 15. Cutting an edge e is the equivalent of inserting a new control point in a SVG path: given a 2D position p on the geometry of an edge e, it creates a new vertex v at position p, and cuts e into two edges e_1 and e_2 separated by v. If e was a closed edge, then it becomes an open edge with its start and end vertices equal to v. Cutting a face f is similar: given a curve Γ starting and ending on ∂f, it cuts the face into two faces f_1 and f_2, separated by a new edge e whose geometry is Γ. This may involve cutting first ∂f to create the end vertices of e if they do not already exist. Alternatively, if Γ starts and ends at two different cycles of f (cf. Figure 15, bottom-left), then instead of cutting f into f_1 and f_2, it simply merges the two cycles into one by concatenating them with the halfedges (e, true) and (e, false). Finally, a face can also be cut by a closed curve contained in its interior, or by a point p (via a Steiner cycle). In the general case, cutting a face is ambiguous and is therefore less trivial than one might think [Dalstein et al. 2014]. A simple example is given in Figure 15 (failure case): if f contains

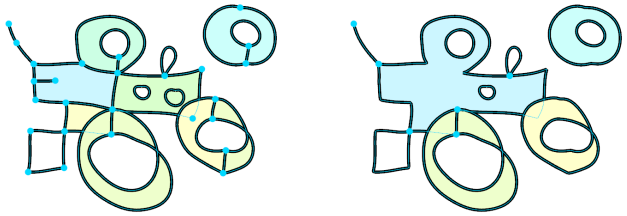


Figure 16: *The effect of a global "uncut". Left: Original VGC. Right: VGC resulting from applying "select all" and then "uncut".*

holes, then one may decide to transfer these holes either to f_1 or f_2, which requires geometric heuristics. Designing a set of infallible heuristics is unfortunately not possible because the boundary of a hole is allowed to overlap the boundary of f, or can even be completely outside f.

Uncutting is the reverse operation: the user chooses a cell c (a vertex or an edge) where to uncut, and the operator merges this cell with its star to obtain a larger cell. Therefore, it can be seen as a "smart delete" or a "local simplification" operation. This operation has a significant theoretical relevance and we refer to this as *atomic simplification* [Dalstein et al. 2014]. While it is always possible to *cut* any given cell, i.e., a face or an edge, it is not always possible to *uncut* at a given cell c. Specifically, this is only possible if c "could have been obtained via a cut". Equivalently, it is only possible if the union of c with its *direct star* is a manifold space [Dalstein et al. 2014]. For instance, if a vertex v has three incident edges, then the union of v with its direct star (in this case the incident edges) is non-manifold, and hence uncutting at v is not possible. Surprisingly, uncutting is "conceptually simpler" than cutting: it is never ambiguous and does not rely on any geometric computation.

Once the uncut operator is implemented both for a single vertex and a single edge, it can be trivially extended to a set of cells: simply uncut all selected edges, then uncut all selected vertices. This is a powerful topological simplification operator, as illustrated in Figure 16: performing this operation on the whole VGC is equivalent to the simplification operation described in [Rossignac and O'Connor 1989]. We conjecture that it results in a unique minimal decomposition [Dalstein et al. 2014].

6 Depth ordering

An important consideration in vector graphics is the ability to order cells from back to front and paint them appropriately. In this section, we describe how this operation is supported with the VGC. Each cell in a VGC (e.g., vertices, edges, and faces) is assigned a unique depth order. The order is maintained via a doubly-linked list containing all the cells, where the 'top' cell of the list will be drawn last and will therefore occlude other parts of the drawing. When a new cell c is created, it is by default inserted just below the lowest cell in its boundary ∂c.

Further alterations of the depth ordering are supported by allowing the user to *raise* a selected cell, c. A trivial implementation of *raise* would be to simply swap the depth order of c and the cell immediately above it in the depth order, c^{next}, as depicted in Figure 17. However, this typically fails to capture the user's intention: there may be no visible change as a results of the raise, and, if a face or edge is selected, the commonly-desired semantics is to have vertices remain on top of the edges and faces that they help define, and edges to remain on top of the faces that they help define. These semantics are implemented by Algorithm 6.1 and illustrated in Figure 17. The *lower* operation is the counterpart to *raise* and is

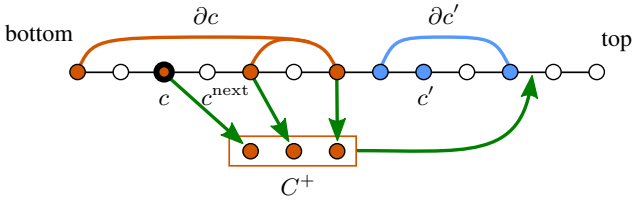


Figure 17: *Raising a cell.*

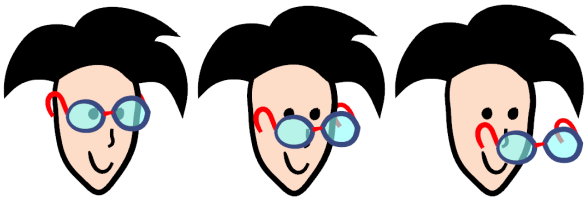


Figure 18: *An example of the flexible occlusion interactions enabled by the VGC.*

implemented in a largely symmetric fashion, where the directions are reversed and ∂c is replaced by $\text{star}(c)$.

Algorithm 6.1: RAISE(selected cell c)

search from bottom to find c
compute $C^+ =$ subset of $(c \cup \partial c)$ that is above c
search up from c for the first cell c' satisfying:
    $c' \notin \partial c$ AND geometry of c' intersects with c
move C^+ above the highest element of $(c' \cup \partial c')$

The ability to manipulate depth orderings for components within objects and between objects allows for partial orderings such as that shown in Figure 18. One arm of the glasses is stacked so as to be behind the face while the other remains in front, along with the rest of the frame. More examples involving depth manipulations are shown in Figure 1 and in the accompanying video.

7 User interface

Many aspects of the user interface can be readily understood from the videos associated with this paper. In what follows we provide a summary of the fundamental concepts and operations.

Edge design: Hand-drawn strokes are the primary method for creating edges. An open stroke drawn on the canvas creates an edge with start and end vertices. Edges can be drawn in a standard fixed-width mode, or their width can vary as a function of stylus pressure. Edges are represented as a densely sampled polyline and can be reshaped using a sculpting tool, either by locally dragging points, smoothing, or editing the width of the curve. If new intersections occur when sculpting an edge, they are ignored and never lead to the creation of a new vertex. The user can always manually insert a vertex at any given intersection.

Intersections and snapping behavior: If desired, edges can be drawn in a mode analogous to working with planar maps. This automatically cuts intersected edges and faces, and cuts the drawn edge at self-intersections. This mode can be enabled or disabled in the GUI by toggling an always-visible icon. A snapping behavior can also be toggled: if enabled, end points of strokes snap to existing vertices if within a distance ϵ. Self-intersection junctions can also snap to each other, thereby allowing multiple approximately colocated self-intersections to automatically coalesce into a single junction.

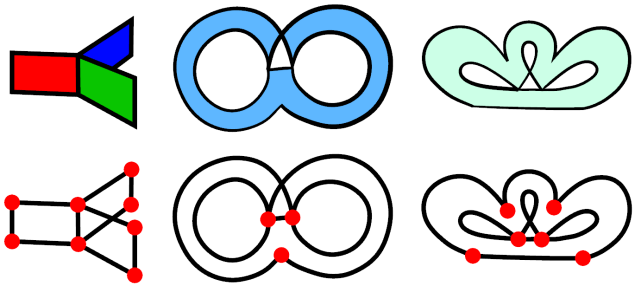


Figure 19: *Examples of non-manifold topologies where an edge has three "face uses". These uses may be by the same face (middle and right), and part of a hole (right)*

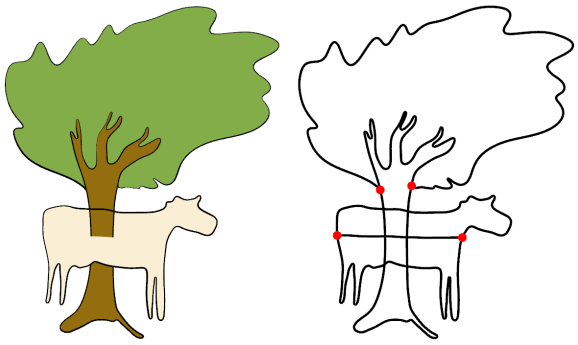


Figure 20: *A user experimenting with the possibilities offered by invisibe cuts and depth ordering.*

Creating faces and holes: Faces and holes can be created in multiple ways, as demonstrated in the videos associated with this paper. Faces are created by selecting the edges that will serve as a boundary. For faces where this does not yield a unique cycle, we require that the face be constructed through intermediate steps. Multiple holes can be added to a face, with the resulting fill determined by their winding number, as shown in Figure 19.

Steiner cycles: Steiner cycles are created by selecting the face and the vertex to add as Steiner cycle. Its primary use is to connect the end vertex of an edge to the interior of a face. Dragging the face would also drag this end vertex, since it translates the whole face boundary. If desired, Steiner cycles can also be used to topologically connect two faces that do not share a common edge. By sharing a common Steiner cycle, they can still be dragged independently, but form a single connected component.

Performance: The performance of our implementation is currently limited by the naive dynamic tesselation that we perform for each render. Edges are rendered according to their width attribute through generation of quadrilaterals that are centered around the polyline that represents the edge path, and these are not cached between renders. Similarly, all faces are retesselated for each render.

8 User feedback

Our prototype has been informally tested by five users ranging from novice to professional artists. Using this feedback, we provide an initial assessment of the usability of the SVG by non-technical users.

Users consistently report that using the VGC is significantly different from current SVG tools, and that it opens exciting new creative

Figure 21: *The VGC can be used for abstract art or stylized figurative art. Examples drawn by Etienne Colas.*

Figure 22: *Three more vector illustrations designed using the VGC. The ellipse surrounding the top illustration is a single edge whose width was sculpted. Examples drawn by Estelle Charleroy.*

workflows. Due to this novelty, the first impressions are generally quite enthusiastic. The topological operations such as glue, unglue, uncut (directly via a tool "simplify", or indirectly when "deleting" a cell) are appreciated and readily adopted. Interestingly, one of the most appreciated features is the ability to sculpt the interior of edges via local dragging or smoothing. Our free-form width editing is reported to be especially useful (see Figure 22).

Concerns have also been raised. Not surprisingly, one of the most disliked aspects of the prototype is the necessity to select all the boundary edges to create a face. Sometimes, due to an unclean topology and tiny edges, this is hard to achieve. Also, rendering artefacts at junctions (see Section 9) and the difficulty to sculpt incident edges across a vertex, specifically to get a smooth transition, have been mentioned. Finally, users currently tend to stay in the (default) "planar map" mode, and hence fail to see the advantage that overlapping faces offer. One user experimented with invisible cuts and depth ordering (Figure 20), but reported difficulties to master the feature. However, we believe that further interface revisions and video tutorials would ease the learning experience.

9 Limitations and future work

The decorrelation of geometry and topology is a powerful means of representing the topology of visual objects as they appear *in the mind's eye*. As a result, it is left to the user to maintain any desired consistency between geometry and topology of VGCs. This can be a limitation in some cases. While the input we provide to the tesselator is the same as used for SVG and is therefore known to be well supported, the VGC offers little support for geometric operations such as boolean operations. The parameterization of VGC faces is left as a separate problem; our current system does not support texture-mapped faces. While a parameterization could be established via triangulation based on a particular geometric configuration, future geometric edits may then become problematic.

It would be useful to extend or augment the VGC to support vector graphics animation, where time is the additional dimension. In the accompanying video, we in fact demonstrate that VGCs can trivially be animated by deforming their geometry in a consistent fashion *as long as their topology does not change*. However, topology changes in the time dimension are left for future work.

Another concern is that even though using VGCs appears reasonably intuitive, they are still a more complex structure than SVG, which may cause confusion and frustration for artists used to the classical representation. For example, a user cannot "uncut" a vertex shared by more than three edges or an edge shared by three or more faces. A vertex that is a Steiner cycle of a face can be moved outside of the face, which can be unintuitive. Constraints could be added to resolve this, but this then removes the independence of the topology and the geometry. Representing an opaque disc involves only one path with SVG (a path with a fill color), while it involves two cells with the VGC (a closed edge and a face whose boundary is this edge). However, the application could easily be adapted to provide further abstractions and tools making the VGC more artist-friendly. For instance, the fill-color property can be simulated by automatically creating a face (and a closing edge for open edges) for every edge in the complex. We believe that the benefits of the VGC largely outweight the added complexity.

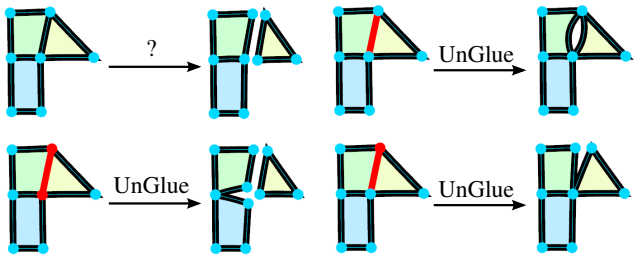


Figure 23: *A user cannot achieve the "partial unglue" operation (shown in the top left) in one step. It can be accomplished indirectly via a sequence of unglues and reglues.*

Figure 24: *Illustrator (except when using LivePaint) cannot represent multiway joins, thus fails to render them correctly: incident faces are rendered independently. With the VGC, we are aware of multiway joins and hence can improve the rendering, as illustrated here with the two common styles "bevel" and "miter".*

Some desired topological operations must currently be performed using multiple steps, such as the "partial unglue" shown in Figure 23. We expect that it is possible to create macros for many such operations.

One advantage of the VGC over traditional vector graphics is that, as with planar maps, it enables the representation of multiway joins (three or more edges sharing a common vertex). As shown in Figure 24, being aware of multiway joins (as opposed to emulating them via duplicated edges) makes it possible to inform the rendering for better results. However, we note that this is a double-edged sword: in the most general case, the correct rendering of multiway joins is a rather difficult and open problem, especially when the incident edges have different and possibly non-uniform widths and colors. This leads to various visible artefacts in our prototype. In Figure 25, we illustrate a typical artefact that occurs due to the overlapping ability of VGC cells and the presence of zero-width or transparent edges.

10 Conclusion

We have introduced the *Vector Graphics Complex*, a novel and powerful data structure for topology-aware design of 2D illustrations. VGC is a superset of multi-layer vector graphics, planar maps and stroke graphs, which significantly extends the range of objects that can be drawn with vector graphics, including 2D projections of 3D objects with imprecise or incomplete geometry, non-manifold surfaces of arbitrary genus, non-orientable surfaces, and overlapping faces. VGC neatly separates the geometry of vector graphics objects from their topology, making it easy to deform objects geometrically in interesting and intuitive ways; and to edit their topology with reversible and provably-correct operators. Components of objects can exist on different layers, which allows for occlusion behaviors to be defined for individual object components rather than objects as a whole. Finally, the explicit representation of multiway joins can be leveraged to improve rendering.

Figure 25: *Left: To obtain a self-overlapping object, one "invisible edge" is necessary, to define two faces with different depth orders. Right: This ordering implies that the red edge is below the yellow face. Depending on the geometry of the invisible edge, this situation often leads to artefacts in our implementation.*

11 Acknowledgements

We thank Estelle Charleroy, Etienne Colas, and other anonymous artists for their invaluable user feedback. We also thank all the reviewers for their precious comments that have helped improving the article in many ways. Part of this work was supported by ERC Advanced Grant "Expressive", by GRAND, and by NSERC.

References

ADOBE SYSTEMS INC., 2013. Adobe Illustrator: Help and tutorials.

ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM Trans. Graph. 26*, 3 (July), 30:1–30:10.

BAUDELAIRE, P., AND GANGNET, M. 1989. Planar maps: An interaction paradigm for graphic design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '89, 313–318.

DALSTEIN, B., RONFARD, R., AND VAN DE PANNE, M. 2014. Point-curve-surface complex: A cell decomposition for non-manifold two-dimensional topological spaces. Tech. rep., University of British Columbia.

DE FLORIANI, L., MORANDO, F., AND PUPPO, E. 2003. Representation of non-manifold objects through decomposition into nearly manifold parts. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, ACM, New York, NY, USA, SMA '03, 304–309.

DE FLORIANI, L., HUI, A., PANOZZO, D., AND CANINO, D. 2010. A dimension-independent data structure for simplicial complexes. In *Proceedings of the 19th International Meshing Roundtable*, Springer Berlin Heidelberg, 403–420.

DURAND, F. 2002. An invitation to discuss computer depiction. In *Proceedings of the 2nd International Symposium on Non-photorealistic Animation and Rendering*, ACM, New York, NY, USA, NPAR '02, 111–124.

EDELSBRUNNER, H., AND HARER, J. 2010. *Computational Topology: An Introduction*. Applied mathematics. American Mathematical Society.

EISEMANN, E., PARIS, S., AND DURAND, F. 2009. A visibility algorithm for converting 3D meshes into editable 2D vector graphics. *ACM Trans. Graph. 28*, 3 (July), 83:1–83:8.

GRANADOS, M., HACHENBERGER, P., HERT, S., KETTNER, L., MEHLHORN, K., AND SEEL, M. 2003. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, and implementation. In *Algorithms - ESA 2003*, vol. 2832 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 654–666.

GUIBAS, L., AND STOLFI, J. 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Trans. Graph. 4*, 2 (Apr.), 74–123.

GURSOZ, E. L., CHOI, Y., AND PINZ, F. B. 1990. Vertex-based representation of non-manifold boundaries. In *Geometric Modeling for Product Engineering*, Eselvier, Amsterdam, 107–130.

GURSOZ, E. L., CHOI, Y., AND PINZ, F. B. 1991. Boolean set operations on non-manifold boundary representation objects. *Comput. Aided Des. 23*, 1 (Feb.), 33–39.

HOFFMAN, D. D. 2000. *Visual Intelligence: How We Create what We See*. Norton.

IGARASHI, T., AND MITANI, J. 2010. Apparent layer operations for the manipulation of deformable objects. *ACM Trans. Graph. 29*, 4 (July), 110:1–110:7.

INKSCAPE, 2013. `http://www.inkscape.org/en/`.

KARSCH, K., AND HART, J. C. 2011. Snaxels on a plane. In *Proceedings of the 9th ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, ACM, New York, NY, USA, NPAR '11, 35–42.

KOENDERINK, J., AND DOORN, A. 2008. The structure of visual spaces. *Journal of Mathematical Imaging and Vision 31*, 2-3, 171–187.

LEE, S. H., AND LEE, K. 2001. Partial entity structure: A compact non-manifold boundary representation based on partial topological entities. In *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, ACM, New York, NY, USA, SMA '01, 159–170.

LIENHARDT, P. 1994. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry & Applications 04*, 03, 275–324.

MARCHEIX, D., AND GUEORGUIEVA, S. 1995. Topological operators for non-manifold modeling. *Proceedings of the Third International Conference in Central Europe on Computer Graphics and Visualisation '95 1* (Feb.), 173–186.

MCCANN, J., AND POLLARD, N. 2009. Local layering. *ACM Trans. Graph. 28*, 3 (July), 84:1–84:7.

NEF, W. 1978. *Beiträge zur Theorie der Polyeder: mit Anwendungen in der Computergraphik*. Beiträge zur Mathematik, Informatik und Nachrichtentechnik. Lang.

NORIS, G., HORNUNG, A., SUMNER, R. W., SIMMONS, M., AND GROSS, M. 2013. Topology-driven vectorization of clean line drawings. *ACM Trans. Graph. 32*, 1 (Feb.), 4:1–4:11.

PESCO, S., TAVARES, G., AND LOPES, H. 2004. A stratification approach for modeling two-dimensional cell complexes. *Computers & Graphics 28*, 2, 235–247.

PORTER, T., AND DUFF, T. 1984. Compositing digital images. *SIGGRAPH Comput. Graph. 18*, 3 (Jan.), 253–259.

ROSSIGNAC, J., AND O'CONNOR, M. 1989. SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries. In *Geometric Modeling for Product Engineering, Proceedings of the IFIP Workshop on CAD/CAM*, IBM T.J. Watson Research Center, 145–180.

ROSSIGNAC, J. 1997. Structured topological complexes: A feature-based API for non-manifold topologies. In *Proceedings of the Fourth ACM Symposium on Solid Modeling and Applications*, ACM, New York, NY, USA, SMA '97, 1–9.

SHREINER, D., WOO, M., NEIDER, J., AND DAVIS, T. 2004. Tesselators and quadrics. In *The OpenGL Programming Guide, Fourth Edition*. Addison-Wesley, ch. 11, 487–514.

SILVA, F. G. M., AND GOMES, A. J. P. 2003. Adjacency and incidence framework: A data structure for efficient and fast management of multiresolution meshes. In *Proceedings of the 1st International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, ACM, New York, NY, USA, GRAPHITE '03, 159–166.

SVG WORKING GROUP, 2011. Scalable Vector Graphics (SVG) 1.1 (Second Edition). `http://www.w3.org/TR/SVG11/`.

TAKAYAMA, K., PANOZZO, D., SORKINE-HORNUNG, A., AND SORKINE-HORNUNG, O. 2013. Sketch-based generation and editing of quad meshes. *ACM Trans. Graph. 32*, 4 (July), 97:1–97:8.

WEILER, K. 1985. Edge-based data structures for solid modeling in curved-surface environments. *Computer Graphics and Applications, IEEE 5*, 1 (Jan), 21–40.

WHITED, B., NORIS, G., SIMMONS, M., SUMNER, R. W., GROSS, M., AND ROSSIGNAC, J. 2010. BetweenIT: An interactive tool for tight inbetweening. *Computer Graphics Forum 29*, 2, 605–614.

WILEY, K., AND WILLIAMS, L. R. 2006. Representation of interwoven surfaces in 2 1/2 D drawing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM, New York, NY, USA, CHI '06, 65–74.

A Semantics – from helpers to pointers

We formally defined the topology of a VGC as a directed graph $\mathcal{G} = (C, \hat{\partial})$, where $C \subset \mathbb{N} \times \mathbb{N}$ is a set of colored cells (a cell has an id and a type), and $\hat{\partial} : C \longrightarrow \mathcal{P}(\mathbb{N} \times \mathbb{N})$ is a map giving for each cell c the set of its colored pointers $\hat{\partial}c$ (a pointer has a pointed id and a semantics). Then we introduced helpers made of halfedges and cycles, to describe in a more convenient a way $\hat{\partial}c$, claiming that it is compatible with the more formal definition. We prove here this claim, by proving that it is possible to convert the information encoded by the helpers into actual pointers of the incidence graph. Even though this conversion could be made explicitely, we propose a more elegant proof of its existence.

Using helpers, the semantic boundary $\hat{\partial}c$ of a cell c can be described as a finite sequence S_c of characters in a countable alphabet A. Since the set $A^\star = \bigcup_{n\in\mathbb{N}} A^n$ of all finite sequences over a countable set A is also countable, there exists an injection:

$$\begin{array}{rccc} \phi: & A^\star & \rightarrow & \mathbb{N} \\ & S_c & \mapsto & \sigma \end{array}$$

It converts in an invertible way S_c (an informal description of $\hat{\partial}c$ via helpers) to a unique integer σ. To properly define the semantic boundary of c as a set of pointers, we can simply use this integer as the color of every pointer in the intended boundary ∂c:

$$\hat{\partial}c = \{\ (c', \sigma) \mid c' \in \partial c\ \}$$

Conversely, given the above formal semantic boundary, we can retrieve the informal description $S_c = \phi^{-1}(\sigma)$ (unless $\hat{\partial}c$ is void, but in such case there are no semantics anyway). This proves that it is possible to define our complex without helpers.