

# Topological Modeling for Vector Graphics

by

Boris Dalstein

B.Sc., École Normale Supérieure de Lyon, 2010

M.Sc., Université Joseph Fourier, Grenoble, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF  
DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2017

© Boris Dalstein 2017



This work is licensed under a Creative Commons

Attribution 4.0 International License.

<https://creativecommons.org/licenses/by/4.0/>

# Abstract

In recent years, with the development of mobile phones, tablets, and web technologies, we have seen an ever-increasing need to generate vector graphics content, that is, resolution-independent images that support sharp rendering across all devices, as well as interactivity and animation. However, the tools and standards currently available to artists for authoring and distributing such vector graphics content have many limitations. Importantly, basic topological modeling, such as the ability to have several faces share a common edge, is largely absent from current vector graphics technologies. In this thesis, we address this issue with three major contributions.

First, we develop theoretical foundations of vector graphics topology, grounded in algebraic topology. More specifically, we introduce the concept of Point-Curve-Surface complex (PCS complex) as a formal tool that allows us to interpret vector graphics illustrations as non-manifold, non-planar, non-orientable topological spaces immersed in  $\mathbb{R}^2$ , unlike planar maps which can only represent embeddings.

Second, based on this theoretical understanding, we introduce the vector graphics complex (VGC) as a simple data structure that supports fundamental topological modeling operations for vector graphics illustrations. It allows for the direct representation of incidence relationships between objects, while at the same time keeping the geometric flexibility of stacking-based systems, such as the ability to have edges and faces overlap each others.

Third and last, based on the VGC, we introduce the vector animation complex (VAC), a data structure for vector graphics animation, designed to support the modeling of time-continuous topological events, which are common in 2D hand-drawn animation. This allows features of a connected drawing to merge, split, appear, or disappear at desired times via keyframes that introduce the desired topological change. Because the resulting space-time complex directly captures the time-varying topological structure, features are readily edited in both space and time in a way that reflects the intent of the drawing.

# Lay Summary

In recent years, it has become increasingly important for digital artists to be able to draw images, animations, and interactive graphics that can be displayed equally well on a wide range of screens, from mobile phones to movie theaters. Unfortunately, the tools currently available for this task suffer from many limitations, such as the inability to easily animate two shapes sharing a common edge, for example two countries sharing a border.

To overcome these limitations, we have developed new mathematical models to represent 2D drawings and animations, and have built new digital tools based on these models. They allow artists to draw and animate shapes which are connected to one another, using novel interactive techniques. Positive initial reception suggests that these new techniques are readily embraced by artists, thus we expect our work to have a large influence in the future of digital drawing tools.

# Preface

A version of Chapter 4 has been published in the following:

- ACM Transactions on Graphics, 33(4), 2014 [[Dalstein et al. 2014b](#)]  
(Proceedings of SIGGRAPH 2014)

The ideas originated in discussions between myself and Michiel van de Panne. It comes from the realization that fundamental research on *non-animated* vector graphics was needed first, before being able to properly tackle the problems of *animated* vector graphics, addressed in Chapter 5 (whose ideas originated first). I conducted the implementation, testing, and contributed to writing the manuscript. Michiel van de Panne and Rémi Ronfard provided guidance, insights, and contributed to the writing of the manuscript.

A version of Chapter 5 has been published in the following:

- ACM Transactions on Graphics, 34(4), 2015 [[Dalstein et al. 2015](#)]  
(Proceedings of SIGGRAPH 2015)

The ideas originated in discussions between myself and Rémi Ronfard, as part of my master’s thesis. I conducted the implementation, testing, and contributed to writing the manuscript. Michiel van de Panne and Rémi Ronfard provided guidance, insights, and contributed to the writing of the manuscript.

A version of the content provided as appendices of this dissertation has been informally distributed as the following:

- Point-Curve-Surface Complex: A Cell Decomposition for Non-Manifold Two-Dimensional Topological Spaces. University of British Columbia, technical report, 2014 [[Dalstein et al. 2014a](#)]

The ideas originated in discussions between myself and Michiel van de Panne. I conducted most of the theoretical analysis and derivations, and wrote the manuscript. Michiel van de Panne and Rémi Ronfard provided guidance and insights.



We have not yet submitted Chapter 3 for publication. This chapter builds on the ideas developed in [Dalstein et al. 2014a], providing important improvements and clarifications. I conducted most of the theoretical analysis and derivations, and wrote the manuscript. Michiel van de Panne provided guidance.

# Table of Contents

<b>Abstract</b>	ii
<b>Lay Summary</b>	iii
<b>Preface</b>	iv
<b>Table of Contents</b>	vi
<b>List of Figures</b>	xi
<b>Acknowledgments</b>	xvi
<b>1 Introduction</b>	1
1.1 Contributions	2
1.2 Informal Definition of the Vector Graphics Complex	3
1.3 Outline of Dissertation	5
<b>2 Background and Related Work</b>	6
2.1 Historical Background	6
2.2 Related Work in Vector Graphics	13
2.3 Related Work in Topological Modeling	16
2.4 Related Work in Animation	30
<b>3 The Theoretical Foundations of Vector Graphics Topology</b>	37
3.1 First Concepts of Topology	38
3.1.1 Topology According to Computer Scientists	39
3.1.2 Topology According to Mathematicians	40
3.1.3 Topology According to Computational Geometers	42
3.1.4 Topology According to 3D Modeling Artists	43
3.2 The Non-Planar Nature of Vector Graphics	47
3.2.1 Design Decisions	48
3.2.2 Non-Planarity and Overlapping	49

## Table of Contents

3.2.3	Non-Orientability . . . . .	50
3.2.4	Non-Manifoldness . . . . .	50
3.2.5	N-Sided Faces . . . . .	51
3.2.6	Closed Edges . . . . .	51
3.2.7	Faces with Inner Holes . . . . .	52
3.2.8	Non-Planar Faces . . . . .	53
3.2.9	Faces without Boundary . . . . .	54
3.2.10	Cut and Glue Closed Edges . . . . .	55
3.2.11	Cut Faces at Vertices and along Closed Edges . . . . .	56
3.2.12	Cut Faces with Inner Holes . . . . .	56
3.2.13	Cut Non-Planar Faces . . . . .	58
3.2.14	The Face-Cut Classification . . . . .	60
3.3	PCS Complexes . . . . .	62
3.3.1	Abstract PCS complexes . . . . .	63
3.3.2	PCS complexes . . . . .	64
3.3.3	Examples and Discussions . . . . .	66
3.3.4	Vector Graphics Complexes . . . . .	72
3.4	Conclusion . . . . .	76
<b>4</b>	<b>Vector Graphics Complexes: The Topology of Vector Illustrations . . . . .</b>	<b>77</b>
4.1	Introduction . . . . .	77
4.2	Motivation and Overview . . . . .	79
4.3	Vector Graphics Complex . . . . .	82
4.3.1	Topology . . . . .	82
4.3.2	Geometry . . . . .	85
4.3.3	Vector Graphics Complexes as Colored Incidence Graphs . . . . .	86
4.3.4	Implementation . . . . .	90
4.4	Topological Operators . . . . .	92
4.4.1	Creation and Deletion Operators . . . . .	93
4.4.2	Glue and Unglue Operators . . . . .	94
4.4.3	Cut and Uncut Operators . . . . .	95
4.5	Depth Ordering . . . . .	96
4.6	User Interface . . . . .	98
4.7	User Feedback . . . . .	99
4.8	Limitations and Future Work . . . . .	102
4.9	Conclusion . . . . .	103
<b>5</b>	<b>Vector Animation Complexes: The Topology of Vector Animations . . . . .</b>	<b>105</b>

## Table of Contents

---

5.1	Introduction . . . . .	105
5.2	Space-Time Topology . . . . .	106
5.2.1	Animating Vertices . . . . .	106
5.2.2	Animating Stroke Graphs . . . . .	107
5.2.3	Animating Vector Graphics Complexes . . . . .	109
5.3	Formal Definition . . . . .	112
5.3.1	Vector Animation Complex . . . . .	112
5.3.2	Key Vertex . . . . .	113
5.3.3	Key Closed Edge . . . . .	113
5.3.4	Key Open Edge . . . . .	113
5.3.5	Key Face . . . . .	114
5.3.6	Inbetween Vertex . . . . .	114
5.3.7	Inbetween Closed Edge . . . . .	114
5.3.8	Inbetween Open Edge . . . . .	115
5.3.9	Inbetween Face . . . . .	115
5.3.10	Halfedge . . . . .	116
5.3.11	Path . . . . .	116
5.3.12	Cycle . . . . .	117
5.3.13	Animated Vertex . . . . .	117
5.3.14	Animated Cycle . . . . .	117
5.4	Interpolation Scheme . . . . .	122
5.5	User Interface . . . . .	123
5.6	Results . . . . .	125
5.7	Discussion . . . . .	128
5.8	Conclusion . . . . .	131
<b>6</b>	<b>Conclusion . . . . .</b>	<b>133</b>
	<b>Bibliography . . . . .</b>	<b>136</b>
	<b>Index . . . . .</b>	<b>143</b>
 <b>Appendices</b>		
<b>A</b>	<b>Concepts of Algebraic Topology . . . . .</b>	<b>145</b>
A.1	Topological Spaces and Homeomorphisms . . . . .	145
A.2	Manifolds with Boundary and Compact Manifolds . . . . .	146
A.3	Points, Curves, and Surfaces . . . . .	147

A.4	Classification of Compact $n$ -Manifolds for $n \leq 2$	147
A.5	Non-Manifold Topological Spaces	150
A.5.1	Abstract Simplicial Complexes	151
A.5.2	CW Complexes	151
A.6	Geometric Realizations and Quotient Spaces	152
A.7	Immersions vs. Embeddings	155
<b>B</b>	<b>Non-Combinatorial Definition of PCS Complexes</b>	156
B.1	Cell Complex	156
B.2	Relation Between $\partial c$ and $\mathcal{B}_c$ , Compactness, and Subcomplexes	159
B.3	Comparison with CW Complexes	160
B.4	PCS Complex	162
<b>C</b>	<b>Equivalence between PCS-Decomposable and 2-Triangulable Spaces</b>	168
<b>D</b>	<b>Topological Operators on PCS Complexes</b>	171
D.1	Notations	171
D.2	Algebraic Operations on Halfedges, Paths and Cycles	173
D.2.1	Paths	173
D.2.2	Flipping Halfedges, Paths and Cycles	174
D.2.3	Converting Open Halfedges to Paths and Paths to Cycles	174
D.2.4	Concatenating Paths	175
D.2.5	Rotating Non-Simple Cycles	175
D.2.6	Extracting Subpaths from Paths and Non-Simple Cycles	177
D.3	Cell Creation	177
D.4	Cell Deletion	181
D.5	Glue Cells	183
D.6	UnGlue Cells	185
D.7	Cut Cells	190
D.7.1	Cutting an Open Edge (at a Vertex)	192
D.7.2	Cutting a Closed Edge (at a vertex)	193
D.7.3	Cutting a Face at a Vertex	194
D.7.4	Cutting a Face at an Edge	194
D.7.5	Cutting an Orientable Face at a Closed Edge	198
D.7.6	Cutting a Non-Orientable Face at a Closed Edge	199
D.7.7	Cutting a Face at an Open Edge Starting and Ending at the Same Hole	202
D.7.8	Cutting a Face at an Open Edge Starting and Ending at Different Holes	208
D.7.9	Flipping Cycles of Non-Orientable Faces	208

*Table of Contents*

---

D.8	Uncut Cells . . . . .	210
<b>E</b>	<b>Simplification of PCS Complexes . . . . .</b>	<b>220</b>
E.1	Simplification of Cell Complexes . . . . .	220
E.2	Equivalence of Cell Complexes . . . . .	223
E.3	Uniqueness of Minimal PCS Complex . . . . .	225

# List of Figures

1.1	Ivan Sutherland's Sketchpad Program . . . . .	1
1.2	Vector Graphics Complex: Example C++ Implementation . . . . .	4
2.1	Early computers . . . . .	7
2.2	Vector Displays vs Raster Displays . . . . .	7
2.3	PostScript . . . . .	9
2.4	Adobe Illustrator 1 . . . . .	10
2.5	Animating a 3-Way Join . . . . .	11
2.6	European Union as SVG . . . . .	12
2.7	SVG Representation . . . . .	13
2.8	Dynamic Planar Maps . . . . .	14
2.9	Stroke Graphs . . . . .	15
2.10	Diffusion Curves . . . . .	16
2.11	Comparison with Existing Topological Structures . . . . .	17
2.12	Winged-Edge Data Structure . . . . .	18
2.13	Halfedge Data Structure . . . . .	19
2.14	Quad-Edge Data Structure: Edge Direction and Orientation . . . . .	20
2.15	Quad-Edge Data Structure . . . . .	20
2.16	Radial-Edge Data Structure: Edge-Uses . . . . .	21
2.17	Radial-Edge Data Structure . . . . .	22
2.18	Combinatorial Maps . . . . .	24
2.19	Generalized Maps . . . . .	25
2.20	Self-Sewing Faces . . . . .	26
2.21	Two-Dimensional Simplicial Complexes . . . . .	27
2.22	CW Complexes . . . . .	28
2.23	Selective Geometric Complexes . . . . .	29
2.24	Keyframes with Inconsistent Topology . . . . .	31
2.25	Stroke Correspondence using Manifold Learning . . . . .	31
2.26	Cartoon Retargeting . . . . .	32
2.27	Texture Transfer . . . . .	33

2.28 Shape Morphing . . . . .	33
2.29 Non-Photorealistic Rendering . . . . .	34
2.30 Animation as Space-Time Modeling . . . . .	35
3.1 Non-Planarity of Vector Graphics Topology . . . . .	37
3.2 Effect of Connectedness on the Behavior of Graphical Objects . . . . .	38
3.3 Homeomorphic Graphs . . . . .	39
3.4 Homeomorphism Counter Example . . . . .	40
3.5 Homeomorphic Spaces . . . . .	41
3.6 Catmull-Clark Subdivision Surfaces . . . . .	42
3.7 Head Topologies . . . . .	43
3.8 Quad Mesh Isomorphism . . . . .	44
3.9 Topology vs Geometry . . . . .	45
3.10 Geometric Realization vs Immersion . . . . .	45
3.11 Vector Graphics Topological Space and Immersion . . . . .	47
3.12 Proof of Non-Planarity and Non-Orientability . . . . .	49
3.13 Proof of Non-Manifoldness . . . . .	50
3.14 N-Sided Faces . . . . .	51
3.15 Closed Edges . . . . .	51
3.16 Face with One Inner Hole . . . . .	52
3.17 Faces with N Inner Holes . . . . .	52
3.18 Non-Planar Faces . . . . .	53
3.19 Faces without Boundary . . . . .	54
3.20 Cut and Glue Closed Edges . . . . .	55
3.21 Cut Faces at Vertices and along Closed Edges . . . . .	56
3.22 Cut Planar Faces with Inner Holes . . . . .	57
3.23 Cut Non-Planar Faces . . . . .	58
3.24 Cut Non-Planar Faces with Closed Edges . . . . .	59
3.25 Face-Cut Classification . . . . .	61
3.26 Formal Steps to Define a PCS Complex . . . . .	62
3.27 Examples of PCS Complexes . . . . .	66
3.28 Examples of PCS Complexes . . . . .	68
3.29 Consistent Parameterization . . . . .	69
3.30 Homeomorphism and Isomorphism of PCS Complexes . . . . .	70
3.31 Multiplicative Notation and Cut Operators on Möbius Strips . . . . .	72
3.32 Sequence of VGC Operators on a Möbius strip . . . . .	73
3.33 Relations between PCS Complexes and VGCs . . . . .	74



3.34	Relevance of the Cut-Face Classification for VGCs . . . . .	75
4.1	Vector Graphics Illustrations and Their Topology . . . . .	77
4.2	SVG Representation . . . . .	79
4.3	Limitations of Existing Representations . . . . .	79
4.4	Vertices, Open Edges, and Closed Edges . . . . .	80
4.5	Faces . . . . .	81
4.6	Cell . . . . .	82
4.7	Topology of a Square . . . . .	84
4.8	Face with Many Cycles . . . . .	85
4.9	Incidence Graph . . . . .	87
4.10	Incidence Graph of a Möbius Strip . . . . .	88
4.11	Different Renders for Different Cycles . . . . .	88
4.12	Coloring of Incidence Graph for Open Edges . . . . .	89
4.13	Coloring of Incidence Graph for Faces . . . . .	89
4.14	Glue and Unglue Operators . . . . .	94
4.15	Cut and Uncut Operators . . . . .	95
4.16	Simplify Operator via Global Uncut . . . . .	96
4.17	Illustration of the Raise Algorithm . . . . .	97
4.18	Partial Ordering . . . . .	97
4.19	Examples of Non-Manifold Topologies . . . . .	99
4.20	User Example . . . . .	100
4.21	More User Examples . . . . .	100
4.22	More User Examples . . . . .	101
4.23	Partial Unglue . . . . .	103
4.24	Comparison of Multiway Joins . . . . .	103
4.25	Possible Artefacts of Invisible Edges . . . . .	104
5.1	Overview of Vector Animation Complexes Cell Types . . . . .	105
5.2	Sequential Keyframing vs. Topological Keyframing . . . . .	107
5.3	Stroke Graph Animation With Time-Varying Topology . . . . .	108
5.4	Topology of Inbetween Edges . . . . .	109
5.5	Intuition behind animated cycles . . . . .	110
5.6	Example of Valid Animated cycle . . . . .	118
5.7	Node-Cell Consistency Invariants . . . . .	120
5.8	Invalid Doubly-Linked Lists . . . . .	121
5.9	Animated Cycle Violating the Cycle Uniqueness Invariant . . . . .	122
5.10	Interpolation Scheme . . . . .	123

5.11 Result: Double Torus . . . . .	126
5.12 Result: Animated Ribbon . . . . .	126
5.13 Result: Flapping Bird . . . . .	127
5.14 Result: Head Turning . . . . .	128
6.1 Example of Challenging Figure to Author with Current Tools . . . . .	133
A.1 Example of 2-Manifold without Boundary . . . . .	146
A.2 Example of 2-Manifold with Boundary . . . . .	146
A.3 Classification of $n$ -manifolds for $n \leq 2$ . . . . .	147
A.4 Polygonal Presentation of the Torus . . . . .	148
A.5 Polygonal Presentation of the Klein Bottle . . . . .	148
A.6 Polygonal Presentation of the Sphere . . . . .	148
A.7 Polygonal Presentation of the Projective Plane . . . . .	148
A.8 Example of Non-Manifold Space . . . . .	150
A.9 Example of Simplicial Complex . . . . .	150
A.10 Different Cell Decompositions of the Sphere . . . . .	150
A.11 Example of Valid Cell Decomposition, but Invalid CW complex . . . . .	152
A.12 Example of Quotient Space . . . . .	153
A.13 CW Complex Seen as a Quotient Space . . . . .	154
A.14 Immersion vs. Embedding . . . . .	155
A.15 Different Immersions of the Klein Bottle . . . . .	155
B.1 Gluing Conditions . . . . .	158
B.2 Example of Valid CW Complex but Invalid PCS Complex . . . . .	161
B.3 Example of Valid CW Complex but Invalid PCS Complex . . . . .	161
B.4 Example of Valid 1-Complex . . . . .	162
B.5 Examples of Invalid 1-Complexes . . . . .	163
B.6 Cell Decompositions of Face Boundary . . . . .	164
B.7 Examples of Valid PCS Complexes . . . . .	165
B.8 More Examples of Valid PCS Complexes . . . . .	166
C.1 Illustration of the Proof of Proposition 7 . . . . .	168
D.1 Examples of SmartDelete Operator . . . . .	181
D.2 Cut-Torus and Cut-Möbius . . . . .	187
D.3 Example of Cut Operator . . . . .	190
D.4 Face-Cut Classification . . . . .	195
D.5 Illustration of the 9 Ways to Cut a Face at a Closed Edge . . . . .	196

*List of Figures*

---

D.6	Illustration of the 10 Ways to Cut a Face At an Open Edge . . . . .	197
D.7	Specifying a Cut using Edge-Uses . . . . .	204
E.1	Reduced Star and Atomic Simplification . . . . .	220

# Acknowledgments

First and foremost, I would like to thank my supervisor Michiel van de Panne. Michiel has been an inspiration at every single step of the program. He was an exceptional mentor, a sincere friend, and an unconditional believer in my ideas, no matter how crazy they were. It is often said that finding the right supervisor is one of the most important steps towards a successful PhD program; I am very glad that I got that step right. Whenever I was not sure which path to take, he would ask me: "Which do you think would have the most impact?". This question is now shaping almost every decision I take, thank you for adding this new lens to my set of world views.

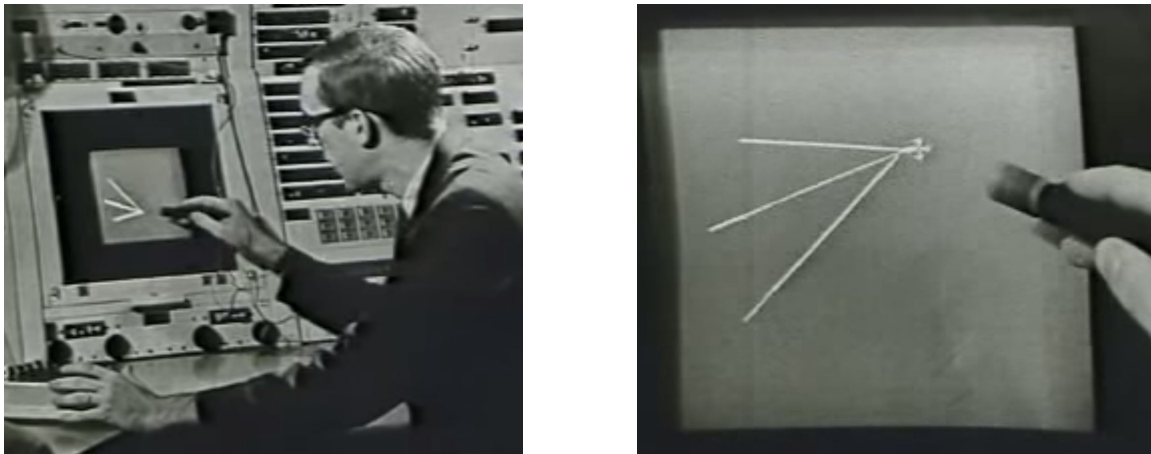
I would like to thank Rémi Ronfard, Joanna McGrenere, Will Evans, Karon MacLean, Dinesh Pai, Alla Sheffer, Hsi-Yung Feng, Daniel Sýkora, and all the anonymous reviewers for their insightful feedback and discussions on the content of my research. There is no doubt that it would be of much lesser quality without you. Feeling part of such a friendly but rigorous academic community was a true privilege. Part of this work was supported by ERC Advanced Grant "Expressive", by GRAND, and by NSERC.

I would like to thank Estelle Charleroy, Etienne Colas, anonymous artists, and all of the members of the VPaint forum for their invaluable user feedback. This entire work would be meaningless without you. Hearing your excitement for the technology gave me confidence that I was on the right track, more than any mathematical proof could. Special thanks to James Lopez for both his feedback and allowing me to use his work from Hullabaloo. Your dedication to hand-drawn animation is an inspiration, and I too wish that this beautiful art form never dies.

I would like to thank all my friends who were a constant reminder that there is more to life than Computer Science. I will never thank you enough for the friendly breakfasts, lunches, and dinners, the Swing and Salsa dancing, the camping, hiking, kayaking, skiing, and mountaineering, the ping-pong playing, the movie nights, and other types of nights I may not remember well. I would like to thank St. John's College for exposing me to so many diverse cultures and opinions, the Varsity Outdoor Club for converting me to an outdoor enthusiast, and Pixar for realizing one of my childhood dreams while taking a useful break from my PhD. Last but not least, I would like to thank my parents, my sister, and all my family for always being supportive despite my tendency to fly very far away from where they live. Most importantly, I would like to thank my partner Elodie, writing this thesis would have been much harder without her love and support.

# Chapter 1

## Introduction



**Figure 1.1:** *Ivan Sutherland's Sketchpad program in action. Source: [MIT Lincoln Laboratory 1964], Copyright 1964 MIT Lincoln Laboratory, permission pending.*

Pioneered in 1963 by Ivan Sutherland in his highly influential PhD thesis introducing the graphical program *Sketchpad* [Sutherland 1963], *vector graphics* refers to the use of basic mathematical primitives, such as straight lines, ellipses, rectangles, and Bézier curves, to represent two-dimensional images, animations, and interactive graphics.

One of the core functionality of Sketchpad was to allow users to simultaneously edit three or more incident lines by simply moving the point where they meet, as demonstrated in Figure 1.1. In the abstract of his PhD dissertation, Sutherland writes:

“Sketchpad stores explicit information about the topology of a drawing. If the user moves one vertex of a polygon, both adjacent sides will be moved. If the user moves a symbol, all lines attached to that symbol will automatically move to stay attached to it. The topological connections of the drawing are automatically indicated by the user as he sketches.”

Therefore, already back in 1963, Sutherland understood the value of explicitly storing topological information about a drawing, such as the incidence relationship between lines: it allows users to interact with the drawing in intuitive ways which would otherwise not be possible. In fact, not

only did Sutherland pioneer vector graphics, but he also pioneered *topological modeling*.

However, something surprising happened. Fast-forward to 2016. The leading commercial vector graphics editor in the industry, Adobe Illustrator CC 2016, poorly supports this feature. The leading open-source vector graphics editor, Inkscape 0.91, does not support this feature at all. The leading and widely adopted open file format for vector graphics, W3C SVG 1.1, does not support this feature at all. Despite how obviously useful this feature is, despite being described in the abstract of one of the most well-known PhD dissertations in computer graphics history, and despite the apparent simplicity of implementing it, this feature is largely absent in today’s vector graphics landscape.

How can this be possible? What went wrong? One can only speculate. However, it may be a hint that in fact, there are some non-obvious, intrinsic properties of vector graphics that make implementing this feature harder than it looks. Some fundamental problems that 50 years of research failed to identify and solve. Somehow, a fascinating subfield of computer graphics, *topological modeling for vector graphics*, failed to materialize in the 1970s to tackle these problems. This thesis is a long overdue analysis of these problems, and a first attempt to solve a few of them.

## 1.1 Contributions

The first contribution of this thesis, presented in Chapter 3, is the development of theoretical foundations of vector graphics topology, grounded in algebraic topology. Importantly, we introduce the concept of Point-Curve-Surface complex (PCS complex) as a formal tool to interpret vector graphics illustrations as *non-planar* topological spaces *immersed* in  $\mathbb{R}^2$ . This contrasts with planar maps which interpret vector graphics illustrations as *planar* topological spaces *embedded* in  $\mathbb{R}^2$ . Using immersions of non-planar spaces instead of embeddings of planar spaces is a key paradigm shift that allows us to rigorously reconcile *overlapping* with *shared boundary*, two useful vector graphics features that were mutually exclusive in previous work.

Our second contribution, and probably the most impactful in practice, is the introduction in Chapter 4 of the vector graphics complex (VGC). The VGC is a data structure that supports topological modeling operations for vector graphics illustrations, based on the theoretical foundations developed in Chapter 3. Unlike stack-based representations, adopted for instance by the W3C SVG standard and the vast majority of today’s vector graphics systems, the VGC allows for the coordinated editing of edges sharing a common vertex, and faces sharing a common edge. Unlike planar maps, much rarer but implemented for instance in Adobe Flash, and Adobe Illustrator since the introduction of the *LivePaint* tool in 2005, the VGC allows edges and faces to overlap, keeping the geometric flexibility of stack-based representations. The VGC is a strict superset of all representa-

tions currently in use, adding useful functionalities without sacrificing any existing ones. Also, it is reasonably simple to understand and easy to implement (see Section 1.2). Therefore, we believe that it has the potential to become a *de facto* standard representation for vector graphics in the future.

The third and last contribution of this thesis is the introduction in Chapter 5 of the vector animation complex (VAC). The VAC is a data structure for vector graphics animation, extending the VGC to the time dimension, and designed to support the modeling of time-continuous topological events. In other words, it allows the representation of drawings animated continuously over time, instead of using discrete frames, including when their topology is non-constant. Examples include two lines merging into one, a line growing from a vertex, or one face splitting into two, such as cell mitosis. Such topological events are extremely common in 2D traditional hand-drawn animation (there are typically several of them per second), but until now there was no practical data structure to model them. Our representation is based on a novel animation paradigm, which we call *topological keyframing*, that is designed to overcome the topological limitations of traditional keyframing. This paradigm can also be applied outside the scope of vector graphics animation, but is particularly well suited in our application case, since topological events are more common in 2D animation than 3D animation.

The VGC and the VAC have been implemented as an open-source vector graphics editor called *VPaint*, which can be freely downloaded at <http://www.vpaint.org>, with binaries available for Windows, MacOS X, and Linux. Experimenting with the topological tools of *VPaint* before or while reading this thesis can be a good way to build some initial intuition before delving into technical details. Note that you are free to use, modify, and/or extend *VPaint* for your own research.

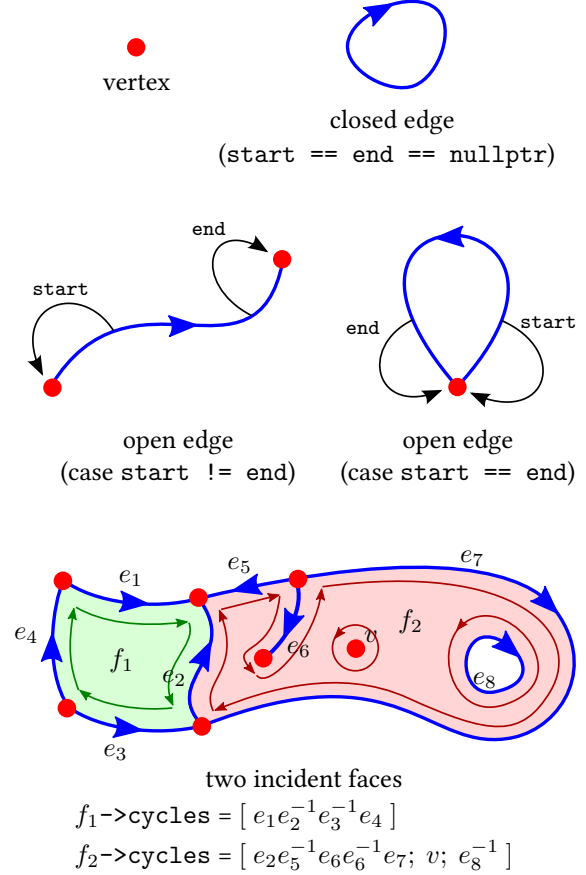
## 1.2 Informal Definition of the Vector Graphics Complex

Let us now provide an informal definition of the VGC, since it is concise enough to fit in this introduction and is a helpful piece of knowledge to have before reading any other chapter of this dissertation. To illustrate the definition, we also provide a C++ implementation and first examples in Figure 1.2.

A VGC is a topological structure made of four types of **cells**: vertices, open edges, closed edges, and faces. A **vertex** is simply defined as a 2D point. An **open edge** is defined as a 2D directed curve that starts and ends at some vertices, possibly equal. A **closed edge** is defined as a 2D directed closed curve, with no start or end vertex at all. Finally, a **face** is defined as a 2D closed region delimited by vertices and edges, ordered in *cycles*. A **cycle** is defined as either: a closed sequence of consecutive open edges; or a single closed edge; or a single vertex.

```

1  class Cell {
2      std::unordered_set<Cell*> star;
3  };
4
5  class Vertex: public Cell {
6      Point p;
7  };
8
9  class Edge: public Cell {
10     Vertex *start, *end;
11     DirectedCurve curve;
12 };
13
14 class Halfedge {
15     Edge *edge;
16     bool direction;
17 };
18
19 class Cycle {
20     Vertex *steiner;
21     std::vector<Halfedge> halfedges;
22 };
23
24 class Face: public Cell {
25     std::vector<Cycle> cycles;
26 };
27
28 class VGC {
29     std::unordered_set<Cell*> cells;
30 };
    
```



**Figure 1.2:** Example C++ implementation of the VGC. In addition to the minimal information required, this implementation also stores the **star** of each cell  $c$ , which is the set of cells  $c'$  whose boundary contains  $c$ .

Typically, one of the cycles represents the *outer boundary* of the face, and the other cycles represent *inner holes*, possibly degenerate (missing curves/points). However, because the cycles of a given face may intersect, the distinction between outer boundary and inner holes is in general ill-defined. For this reason, the VGC does not explicitly store whether a cycle is an outer boundary or an inner hole.

Cycles made of consecutive open edges may *use* the same edge  $e$  any number of times, and each of these *edge-uses* independently specifies whether  $e$  is traversed in its intrinsic direction, or in the opposite direction. We use the term **halfedge** to refer to an edge with such specified direction, and we respectively denote by  $e$  and  $e^{-1}$  the two possible directions. This allows us to denote cycles using a convenient multiplicative notation, such as  $\gamma = e_2 e_5^{-1} e_6 e_6^{-1} e_7$ . Cycles made of a single closed edge  $e$  may use it multiple times, but all uses must be with the same direction, i.e., either  $\gamma = e^n$  or  $\gamma = e^{-n}$ . This represents a cycle that circles around the same closed edge multiple times. Finally, cycles made of a single vertex  $v$  are called *Steiner cycles* and denoted  $\gamma = v$ .



## 1.3 Outline of Dissertation

**Chapter 2. Background and Related Work.** In this chapter, we provide a brief history of vector graphics, computer animation, and topological modeling, as well as an overview of relevant work and the state of the art in these fields.

**Chapter 3. The Theoretical Foundations of Vector Graphics Topology.** In this chapter, we first give precise definitions of many concepts used throughout this dissertation, such as the definition of *topology* and *overlapping*. Then, we introduce the concept of PCS complex as a formal tool to interpret vector graphics illustrations as two-dimensional topological spaces immersed in  $\mathbb{R}^2$ . This provides an in-depth understanding of the non-planar, non-orientable nature of vector graphics shapes, and shows why topological modeling for vector graphics is in fact much harder than it seems. This chapter can be read independently of the other chapters.

**Chapter 4. Vector Graphics Complexes: The Topology of Vector Illustrations.** This chapter provides an analysis of the vector graphic complex, together with many practical considerations, results, and informal discussions. This chapter can be read independently of the other chapters, but reading Chapter 3 beforehand makes it more insightful, as it provides a rationale for most design decisions behind the VGC, and a framework to rigorously infer the set of topological operators acting on VGCs.

**Chapter 5. Vector Animation Complexes: The Topology of Vector Animations.** This chapter introduces the vector animation complex, a topological data structure to represent vector graphics animations whose topology can change over time. Reading Chapter 4 is recommended before reading this chapter. Indeed, each time-slice of a VAC is nothing else but a VGC, thus a good understanding of the VGC is required to understand the VAC.

**Chapter 6. Conclusion.** Finally, we conclude the dissertation with a summary of the contributions of this thesis, and discuss its implications and the road that is still ahead of us.

## Chapter 2

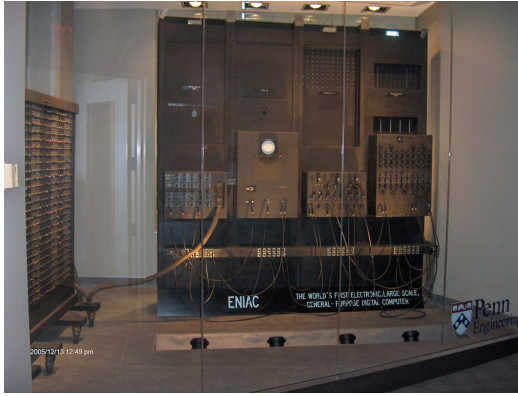
# Background and Related Work

Vector graphics, computer animation, and topological modeling are some of the core subfields that have driven research in the early years of computer graphics. In order to better understand where this thesis fits in the big picture, let us recall some of the most relevant bits of the fascinating history of computer graphics. Most of the historical elements presented here come from various sources which include the two textbooks *Fundamentals of Computer Graphics* [Shirley and Marschner 2009] and *Fundamentals of Interactive Computer Graphics* [Foley and Van Dam 1982], the books *CG 101* [Masson 1999] and *Creativity, Inc.* [Catmull and Wallace 2014], and the online content of a course from Ohio State University [Carlson 2003]. However, please note that our aim here is not to be exhaustive. For conciseness, we omit a lot of key players, and only focus on those that we think better help understand why and how we arrived where we are today, in the fields most relevant to this thesis. For a more exhaustive and accurate history, I highly recommend to read the references provided above, especially [Carlson 2003] which is easily accessible and a fascinating read. Then, following Section 2.1, we survey the state of the art in the fields most related to this thesis.

## 2.1 Historical Background

Computers are useful to humans only to the extent that we are able to communicate with them, and improving this communication has always been of critical importance to the development of computing technology. Computer graphics, and especially the introduction of graphical user interfaces (GUIs), was one of the key communication medium that has allowed computers to reach the general public, without which we wouldn't have computers in our pockets today.

**From Punched Cards to Graphical Displays** In the early age of computing, a standard communication medium was the 80-column IBM punched card, introduced by Hollerith in 1928, and used until the 1970s. During this period, there was barely any interactivity: computers were given data and programs to execute from humans via punched cards (and later tapes or keyboards) and gave an answer to humans also via punched cards, text printed on paper, lights, bells, or switches. For instance, The ENIAC computer (Figure 2.1a), completed in 1946 at the University of Pennsylvania, used punched cards both as input and output. A few years later, the more advanced

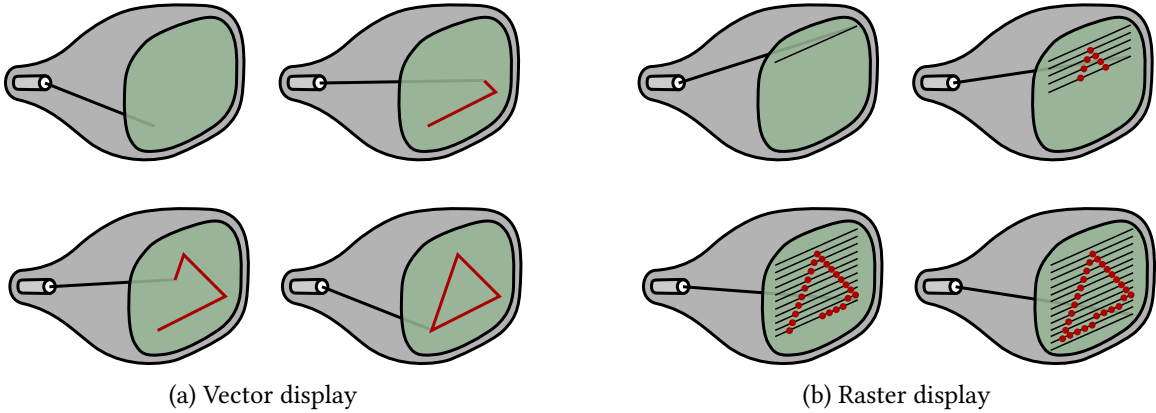


(a) The ENIAC computer



(b) The UNIVAC I computer

**Figure 2.1:** (a) The ENIAC computer. Source: Wikipedia, Copyright 2005 Paul W Shaffer, University of Pennsylvania, licensed under CC BY-SA 3.0. (b) The UNIVAC I computer. Source: Wikipedia, submitted by user 'Daderot', Public Domain.



(a) Vector display

(b) Raster display

**Figure 2.2:** The two types of CRT displays, at the origin of the terminology vector graphics vs raster graphics. Source: inspired from [Carlson 2003].

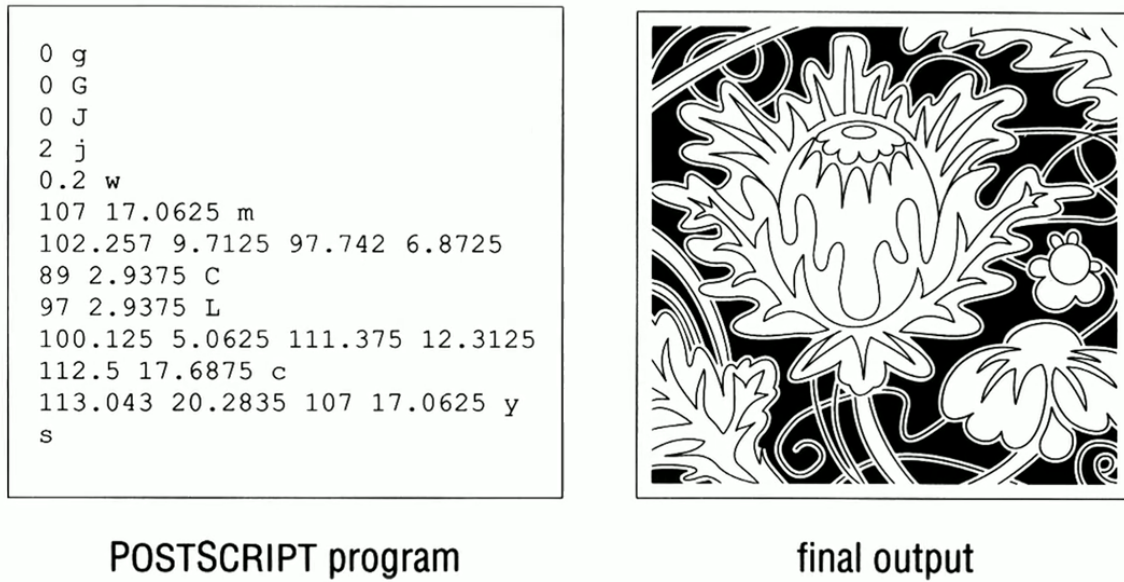
UNIVAC I computer (Figure 2.1b), commercialized in the United States starting 1951, also accepted punched cards as input/output. However, it also accepted magnetic tapes, and could be hooked to other output devices such as an electric typewriter, and more importantly in the context of this dissertation, an early Tektronix oscilloscope.

**Raster Displays vs Vector Displays** Oscilloscopes, and more generally cathode ray tubes (CRTs), are indeed the first type of graphical display systems that were used with computers. There are two types of CRT displays: *vector displays* and *raster displays*, also called random-scan displays and raster-scan displays [Foley et al. 1990, Chapter 4]. In both cases, a CRT is made of an electron gun emitting a stream of electrons that eventually hits a phosphor-coated screen, emitting light. However, in CRT vector displays, the trajectory of the electron gun follows a sequence

of user-specified `MoveTo(X,Y)` and `LineTo(X,Y)` commands (Figure 2.2a), while in CRT raster displays it follows a fixed trajectory covering the whole screen with a predefined number of scan lines (Figure 2.2b). Note that vector displays were incapable of drawing many features that are now common in vector graphics, such as lines of varying width, filled interiors of given contours, and specific join styles between vector lines (e.g., miter joins, as seen in Figure 2.5). All of these are possible with raster displays via rasterization algorithms. This limitation of vector displays, and thus of Sketchpad, may in fact be an important reason why Sketchpad was able to feature topological modeling, while current vector graphics applications do not. Vector displays were used up to the 1970s due to their more affordable price, but are now obsolete in favor of today's raster displays. The distinction between these two types of displays is at the origin of the terminology *vector graphics* versus *raster graphics*.

**Sketchpad: Pioneering Interactive Computer Graphics** In 1959, the TX-2 computer was built at the MIT [Reilly 2003, p261], and included a nine-inch CRT vector display as output, as well as a *light pen* as input, a technology also developed at the MIT as part of the Whirlwind project a few years earlier. The combination of this graphical input/output enabled Ivan Sutherland to build Sketchpad [Sutherland 1963], the first ever GUI, which is one of the most significant milestones ever achieved in computer graphics history, and especially interactive graphics. Notably, the ancestor of vector graphics was born. As mentioned in the Introduction, it is interesting to note that this ancestor of vector graphics featured some aspects of topological modeling, such as the correct representation of 3-way joins (see Figure 1.1), which Adobe Illustrator does not properly support today. However, it is important to recall that due to the limitations of the vector display, Sketchpad's topological model did not have to support faces (2D regions filled with a given color), nor to support the rendering of various join styles, both of which are important features that Adobe Illustrator had to support early on. These features make topological modeling much harder, as we will see in Chapter 3.

**DAC-1: Pioneering Computer-Aided Design** In the same period of time, roughly from 1959 to 1967, computer-aided design (CAD) also started to emerge, with IBM and General Motors working together on a project called DAC-1 (Design Augmented by Computers). This project aimed at proving the feasibility of using computers to solve problems related to vehicle-body design [Krull 1994]. They were using a similar system as Sketchpad, with a CRT display as output and a light pen as input, but instead were working on three-dimensional mathematical car models. Very early, they also noticed the importance of storing the topology of the model, i.e., not just the shape of the surfaces, but also the incidence relationship between the different surfaces. From this point on, the CAD industry became one of the strongest driving forces of computer graphics, with topological modeling at its core. In contrast, within the field of vector graphics, topological modeling failed to take root, which motivates this thesis.

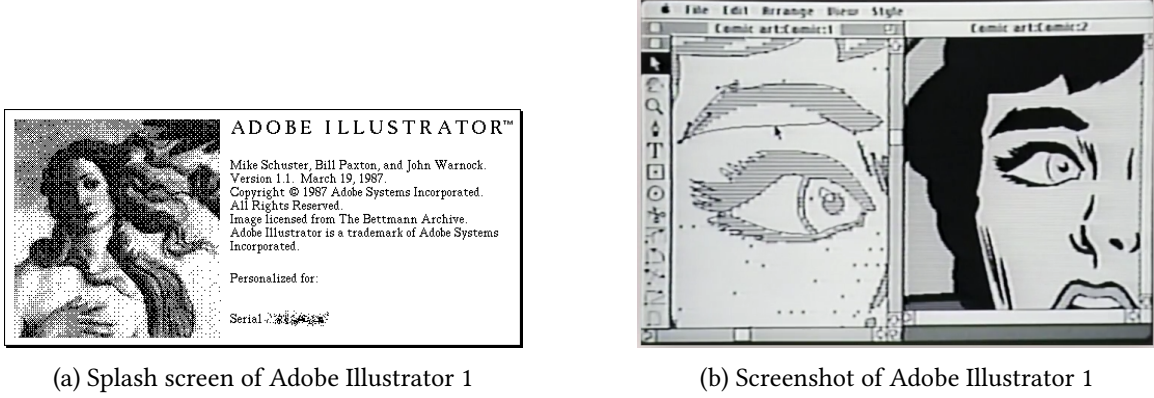


**Figure 2.3:** *The PostScript graphics language for printing.* Source: [Capen et al. 2014], Copyright 2014 Adobe Systems Inc, permission pending.

**From Sketchpad to Pixar and Adobe** In 1968, Ivan Sutherland, the author of Sketchpad, moved to the University of Utah to become a professor, and co-founded *Evans and Sutherland* with his friend and colleague David Evans. With their colleagues, students, and employees, in a few years, they went on to make many early breakthroughs in computer graphics, such as texture mapping, Z-buffering, subdivision surfaces, and hidden surface removal. One of their doctoral students was Edwin Catmull, who started his PhD in 1970. After graduating in 1974, Edwin Catmull led the new and highly influential Computer Graphics Lab at New York Institute of Technology (NYIT), then moved to California in 1979 to work for George Lucas at LucasFilm, where he led the Graphics Group that eventually became Pixar when spun-off as a separate corporation in 1986, with the financial support of Steve Jobs. Another employee of *Evans and Sutherland* was John Warnock, whose importance is even greater in the context of this thesis. John Warnock started working for *Evans and Sutherland* in 1976, then moved in 1978 to Palo Alto to join Xerox PARC, also a very influential research center in computer graphics. John Warnock was interested in developing a graphics language to control printing, and he left Xerox in 1982 to co-found Adobe, where he developed PostScript (released in 1984), then Adobe Illustrator (released in 1987), which is the first software product of Adobe [Capen et al. 2014].

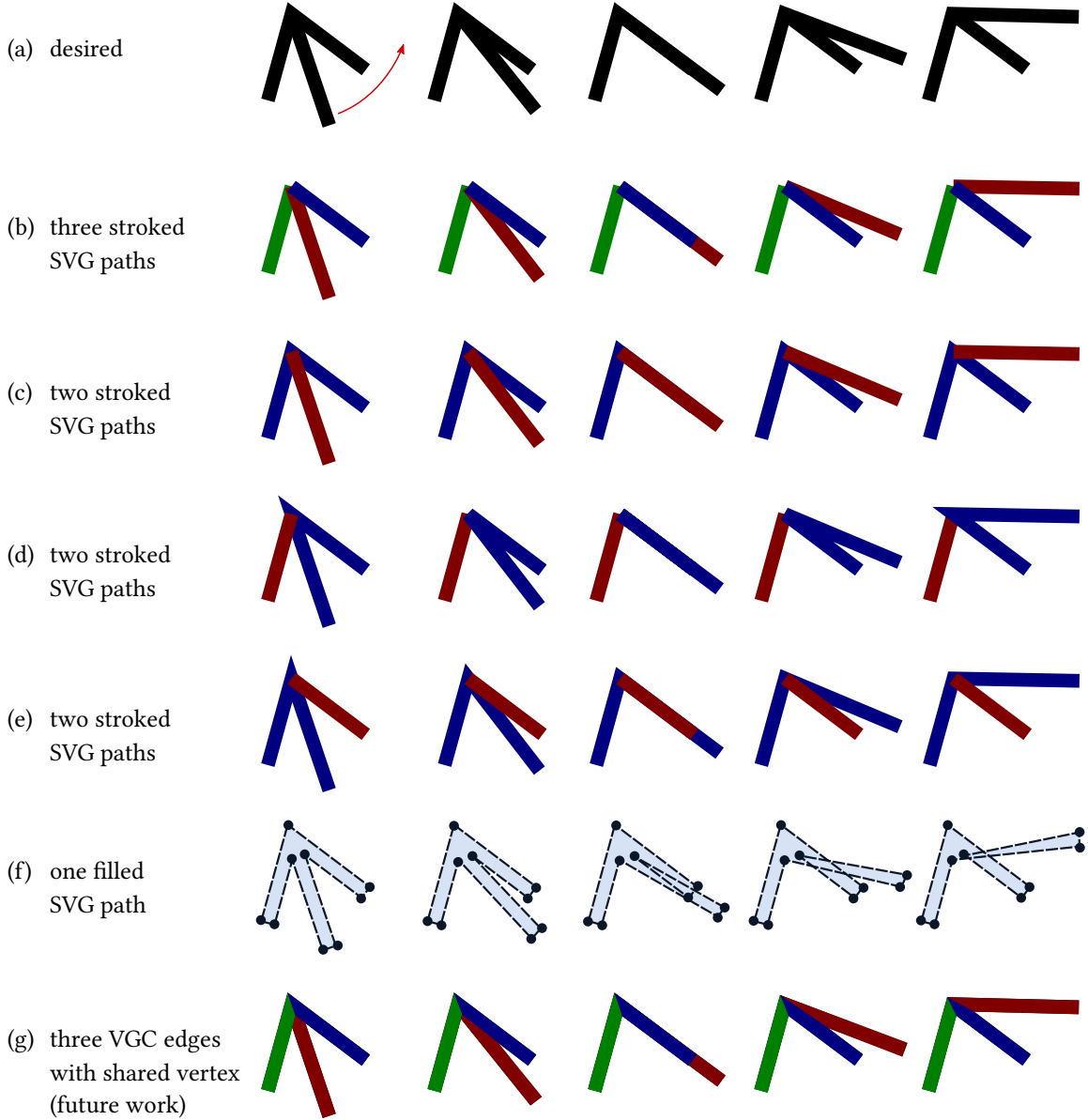
**PostScript and Adobe Illustrator** Released in 1984, PostScript (see Figure 2.3) can be seen as the first widely adopted vector graphics file format, hence the first widely adopted vector graphics representation. However, it was designed as a unifying language that every printer could under-



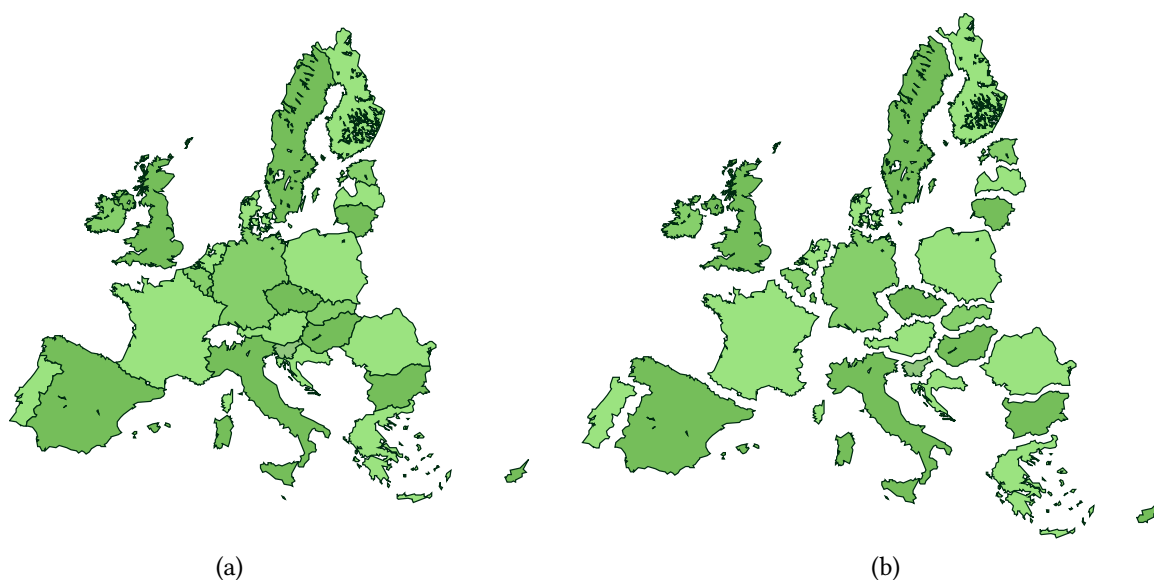


**Figure 2.4:** *Adobe Illustrator 1, released in 1987. Source: [Capen et al. 2014], Copyright 2014 Adobe Systems Inc, permission pending.*

stand, and therefore focused on simplicity, not art-directability. In addition, in its early days (before Illustrator was released), it had to be written by hand by programmers, which further emphasizes the focus on simplicity. These practical considerations may be one of the reasons why topological information, such as incidence relationships between paths, was left out of this standard. In 1987, Adobe released the first version of Illustrator (see Figure 2.4) as a graphical front-end to create PostScript art, as well as typographic fonts. Illustrator did not support the topological modeling features that were pioneered within Sketchpad, perhaps as a natural consequence of PostScript not supporting the encoding of topological information. Therefore, the first version of Adobe Illustrator did not support the coordinated editing of edges sharing a common vertex (supported in Sketchpad), or the coordinated editing of faces sharing a common edge (not supported in Sketchpad either, since Sketchpad did not support filled regions at all). This choice was understandable at the time: as we will see in Chapter 3 and 4, implementing these features is in fact far from trivial when color filling and edge styling is also a requirement, and therefore given the many technical constraints of writing software back in 1987, the features that did make it to Illustrator 1.0 were already a big achievement. However, it is surprising that very little progress has been made since then to identify and address these issues. Notably, nearly 30 years later, Adobe Illustrator CC 2016 still does not properly support these features. Even though the *LivePaint* tool (Adobe’s implementation of dynamic planar maps [Asente et al. 2007]) provides some aspects of coordinated editing, the objects created with this tool cannot interact with the more traditional paths which are still widely used and required as input to most Illustrator’s tools. Converting LivePaint groups to paths causes the topological information to disappear, and therefore causes edge joins not to be properly rendered. Finally, the full range of topological operators common in every CAD application, such as merging or splitting edges, is not supported.



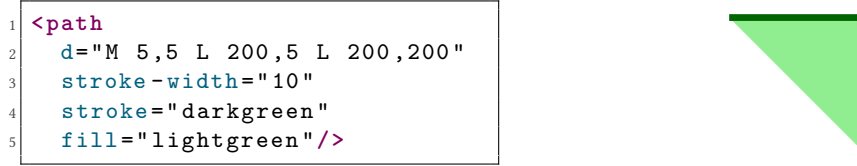
**Figure 2.5:** Styling issues when animating—or interacting with—a 3-way ‘miter’ join. (a) Desired style. (b–f) Incorrect styling using SVG in various ways. (g) Example of artefact-free styling that might be achieved using VGC edges. However, note that in this thesis we only solve how to store the topological information, and the five images in the last row were manually produced individually. How to computationally produce a good-looking, artefact-free miter join style from the topological information is a hard problem on its own, left for future work.



**Figure 2.6:** *In order to represent geographic maps in SVG, such as here the 2016’s European Union, one must duplicate geometric information at the boundary between countries, because SVG does not allow the representation of shared edges between faces. (a) SVG content as seen by end users. (b) Underlying topological representation. Source: modified from Wikipedia, submitted by user ‘Ssolbergj’, licensed under CC BY-SA 3.0.*

**Adobe’s Heritage** Most vector graphics file formats and tools that followed PostScript and Adobe Illustrator used a similar data representation, and the topological modeling features of Sketchpad seem to have been forgotten by the vector graphics community. One can only speculate why, but it is probably an unfortunate combination of many factors. Perhaps subsequent tools had to stay compatible with the industry standard set by Adobe. Perhaps they were simply heavily influenced by PostScript and Illustrator. Perhaps the few industrial players who tried to develop topological modeling for vector graphics did not have the mathematical background or time required to tackle the issues that we identify and/or address in this thesis. Perhaps the academic researchers who could have tackled these issues were not aware of them, because vector graphics topology *appears* like a solved topic. Perhaps there is too little research on vector graphics in general, because most research leans towards 3D graphics which is more appealing to the majority of scientists. In any case, the consequence of the lack of research in vector graphics is that to this day, in 2016, the currently leading open standard for vector graphics, W3C SVG 1.1, still does not include any topological information [SVG Working Group 2011]. Therefore, something as seemingly simple as three edges sharing a vertex cannot be intuitively modeled, stylized, and animated in your browser (see Figure 2.5). Also, any geographic map must store redundant geometric information at the boundary between countries, because the current standard does not allow faces to share common edges (see Figure 2.6). The latter issue has already been identified by the community, and extensions of SVG have been proposed [Moissinac 2010] to solve it, but they are not





**Figure 2.7:** The SVG representation, with an example SVG code (left), and the final output (right). The *d* attribute specifies the geometry of the path, which in this case should be read as *MoveTo*(5,5), then *LineTo*(200,5), then *LineTo*(200,200).

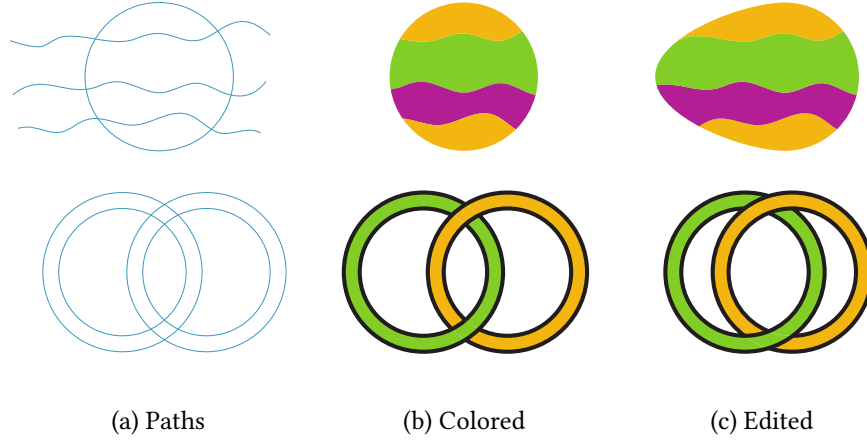
part of the current SVG 2.0 Candidate Recommendation [SVG Working Group 2016], thus are very unlikely to be part of the future SVG 2.0 standard. Also, the solution is an ad-hoc primitive to support this specific feature, and not the fundamental change that is required to make topological information a first-class citizen of the representation, as we propose in this thesis.

In the three remaining sections of this chapter, we respectively review the most relevant subset of the state of the art in vector graphics, topological modeling, and animation.

## 2.2 Related Work in Vector Graphics

**SVG Representation** The most common way to represent vector graphics illustrations, both in academic and commercial systems, consists of using a set of independent paths which are *layered*, or *stacked*, on top of one another, such as the SVG representation [SVG Working Group 2011]. Even though there exist some nuances between different systems (e.g., Adobe Illustrator vs. Inkscape), they are essentially identical from a topological standpoint, therefore, for conciseness, we collectively refer to all such stacking-based representations as *the SVG representation*. Each path is a one-dimensional curve, for example represented via Bézier control points, which can be either open or closed. To specify how each path is to be rendered, style attributes are attached to them. For instance, in SVG, the `stroke` attribute specifies the color of the curve, the `stroke-width` attribute specifies the width of the curve, and the `fill` attribute specifies the color of the 2D region “inside” the curve (where “inside” is defined by a winding rule, and open paths are implicitly closed using a straight line), as illustrated in Figure 2.7. The SVG representation is fundamentally limited in its 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, as we have seen in Section 2.1.

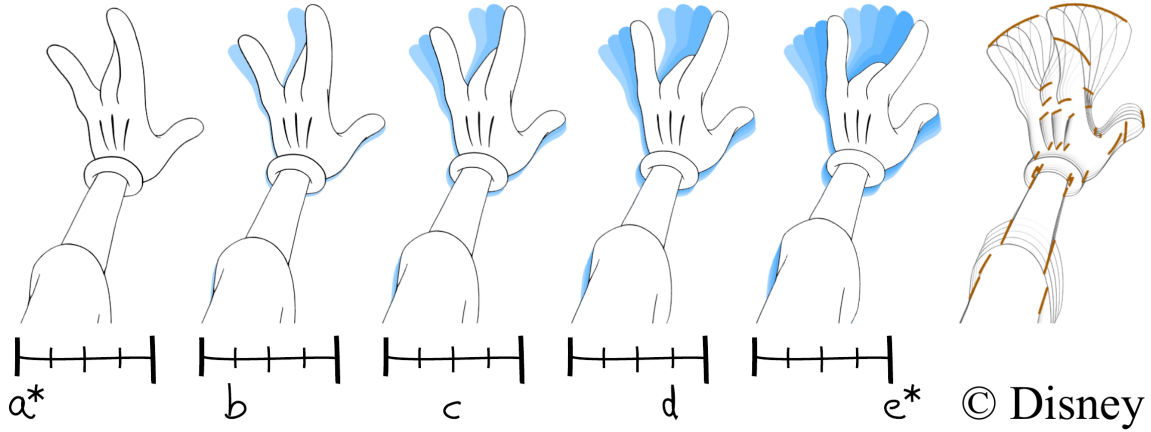
**Depth Ordering** Most stacking-based systems use a back-to-front rendering of paths, group of paths, and layers to designate the occlusion relationships among objects, which are then composed together using alpha-blending [Porter and Duff 1984]. However, illustrations with cycles



**Figure 2.8:** Two examples of dynamic planar maps illustrations. (a) Open and/or closed paths are given as initial input. (b) These paths define a partition of the canvas into regions that can be independently colored. (c) Heuristics are used to preserve colored regions while the input paths are edited. Source: [Asente et al. 2007], used with permission.

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 stacking order 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 stacking order 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].

**Planar Maps** Besides stacking-based systems, it is also common—although less common—to represent vector graphics illustrations using *planar maps* [Baudelaire and Gangnet 1989]. A planar map is a topological structure that is able to represent a partition of  $\mathbb{R}^2$  into regions, delimited by a set of intersecting 2D paths given as input (see Figure 2.8a), which can then be independently assigned style attributes such as color filling (see Figure 2.8b). One advantage of planar maps over stacking-based systems is that they can represent faces sharing common edges, which makes them more suitable for a wide class of illustrations that cannot be easily decomposed into independent paths. However, a difficulty of this approach is that the planar map needs to be recomputed whenever 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] (see Figure 2.8c). Also, planar maps, as their name suggests, are unfortu-

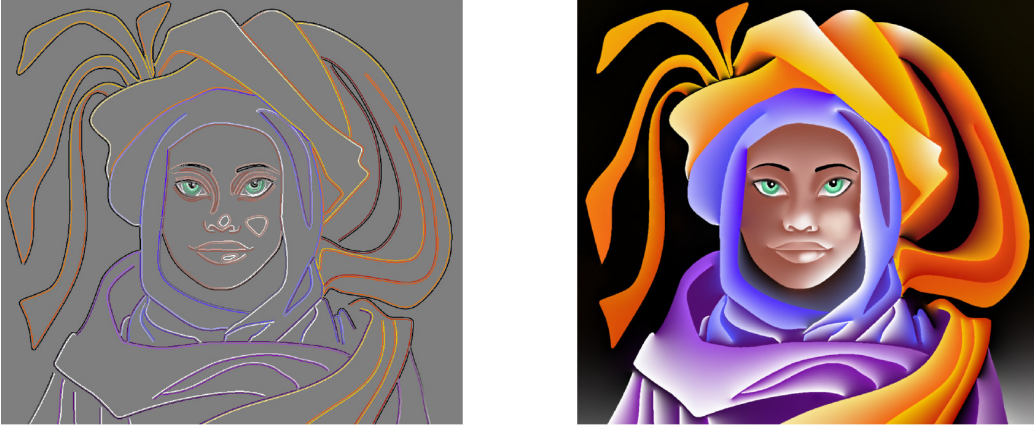


**Figure 2.9:** Example of inbetween frames that can be automatically generated by leveraging the topological information encoded in stroke graphs. Source: [Whited et al. 2010], permission pending.

nately not able to represent regions that overlap with one another, which is a significant hindrance for artistic freedom. Several planar maps can be stacked as layers, but then shapes in different layers cannot be topologically connected, unlike with our representation. 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.

**Stroke Graphs** Stroke graphs [Whited et al. 2010] represent the topology of a drawing as a graph, where nodes represent where strokes end or intersect, and edges represent the strokes themselves. This topological information can be used to establish automatic stroke correspondences between two keyframes of a traditional 2D animation, and provide topology-aware interpolation of the strokes (see Figure 2.9). Also, it can be used to vectorize pencil line drawings with accurate topology [Noris et al. 2013, Favreau et al. 2016]. However, stroke graphs do not address the issue of the representation of faces, and, to the best of our knowledge, no previous work with stroke graphs discussed overlapping edges (even though the data structure supports them), or the usefulness of this structure as an interactive drawing paradigm. Our work can be seen as an extension of stroke graphs to support closed edges and faces, together with a theoretical analysis of the consequences of allowing overlapping, and its application as an interactive drawing paradigm.

**Smooth-Shaded Vector Images** In order to represent vector graphics images with smooth shading, the simplest option is to fill each path with a linear or circular color gradient instead of a solid color. This method is standard and has been widely available across all vector graphics systems for decades. A more recent technique is to use a *gradient mesh*, for instance defined as a Coons surface [Coons 1967]. This technique is implemented for instance in recent versions



**Figure 2.10:** *Example of smooth-shaded vector illustration obtained using diffusion curves. Source: [Orzan et al. 2008], used with permission.*

of Adobe Illustrator and Corel CorelDraw, and has been successfully used to accurately vectorize photograph images [Sun et al. 2007]. More recently, [Orzan et al. 2008] introduced diffusion curves, where smooth-shaded vector images are represented by two-sided colored paths, possibly intersecting, whose color affect nearby regions via a diffusion equation (see Figure 2.10). However, none of these representations are meant to or are able to represent the topology of line drawings, and therefore are not suitable for topological modeling. Though, they can potentially be used together with our topological representation to provide smooth shading, which we leave for future work.

## 2.3 Related Work in Topological Modeling

In Section 2.2, we have discussed the limitations of existing vector graphics representations: the SVG representation [SVG Working Group 2011] and diffusion curves [Orzan et al. 2008] do not store any incidence relationships, stroke graphs [Whited et al. 2010] do store incidence relationships between edges but do not represent faces, and planar maps [Baudelaire and Gangnet 1989, Asente et al. 2007] do store incidence relationships between both edges and faces but can represent neither overlapping edges nor overlapping faces. Therefore, we found that none of these representations were good candidates to support topological modeling for vector graphics. In this section, we now review representations that have actually been successful to support topological modeling, but outside of the realm of vector graphics. In this thesis, we took inspiration from these existing topological structures, and adapted them to the problems at hand.

In Figure 2.11, we provide a comparison between these existing structures and the VGC, highlighting how none of the existing structures satisfied our requirements. We wanted a topological

	Scales to arbitrary dimension		Admits combinatorial representation		Overlapping		Shared boundaries		Non-planar topologies		Non-orientable topologies		N-sided faces		Faces with inner holes		Closed edges		Non-planar / non-orientable faces		Unique minimal decomposition	
Winged-Edge / Halfedge	✗	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Quad-Edge	✗	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Radial-Edge / Partial Entity	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	
Combinatorial Maps	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Cell-Tuples / Generalized Maps	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Chains of Maps	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Simplicial Complexes	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
CW Complexes	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Geometric Complexes / SGC / STC	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
PCS Complexes	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓?	✓?	✓?	
SVG-Like Representations	✗	✓	✓	✗	✗	✗	✗	✓	✓	✓	✓	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	(✓)	N/A	N/A	N/A	
Planar Maps	✗	✓	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	N/A	N/A	N/A	
Vector Graphics Complexes	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(✓)	(✓)	(✓)	✓?	✓?	✓?	

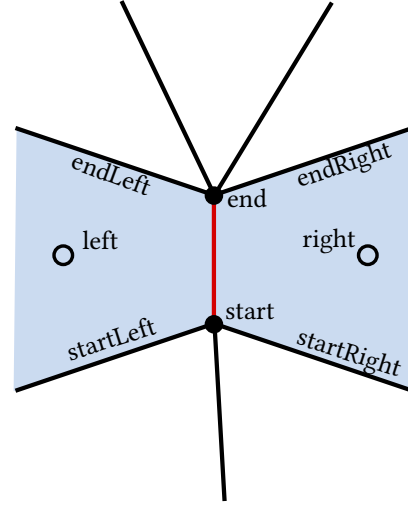
**Figure 2.11:** Comparison between existing topological structures and ours (in bold). The two question marks mean that we conjecture this property to be true, although we have no formal proof yet. The parentheses mean that even though vector graphics complexes do not explicitly encode non-planarity or non-orientability information for individual faces, they do allow for the representation of faces which humans would perceptually interpret as being non-planar or non-orientable. Note that by non-planar faces, we mean non-genus-0 faces, i.e., faces which cannot be embedded in a plane. Most of these data structures do support “curved” faces (such as a half-sphere), but very few support non-planar faces (such as a torus or a Möbius strip).

structure with a representational power similar to the SGC [Rossignac and O’Connor 1989], but with the geometric flexibility of stacking-based systems (the SGC can only represent embeddings, and does not admit a purely combinatorial representation, unlike the SVG representation). We were able to achieve this by narrowing our scope to dimension 2, allowing us to make less compromises on other features. Ultimately, the VGC can be seen as a variant of the radial-edge data structure [Weiler 1986], extending it to support closed edges and non-planar faces, but with no radial ordering of edge-uses, no face-uses, no shells, and no regions.

**Array-Based Data Structures** One of the simplest and most compact ways to represent a triangle mesh is as an array  $P = [p_1, \dots, p_n]$  of  $n$  positions  $p_i \in \mathbb{R}^3$ , together with an array

```

1 class Vertex {
2     Point p;
3     Edge *edge;
4 };
5
6 class Edge {
7     Vertex *start, *end;
8     Face *left, *right;
9     Edge *startLeft, *startRight;
10    Edge *endLeft, *endRight;
11 };
12
13 class Face {
14     Edge *edge;
15 };
16
17 class WingedEdgeDS {
18     std::set<Vertex*> vertices;
19     std::set<Edge*> edges;
20     std::set<Face*> faces;
21 };
    
```



**Figure 2.12:** Example implementation of the winged-edge data structure.

$I = [i_1, i_2, i_3, \dots, i_{3m-2}, i_{3m-1}, i_{3m}]$  of  $3m$  indices  $i_j \in [1..n]$ , where each consecutive triplet of indices represents one triangle. The array  $I$  stores the *topology* of the mesh, while the array  $P$ , optional for applications that only care about topology, stores the *geometry* of the mesh. The same idea can be easily adapted to store quad meshes, or arbitrary polygonal meshes. In some cases, the size of the array  $I$  can be reduced by grouping triangles into “triangle strips”, avoiding to repeat redundant indices. In such cases, the amortized per-triangle storage cost can be as low as one index  $i \in \mathbb{N}$ . Due to their simplicity and compactness, these types of representations are very popular as data formats (e.g., OBJ) or APIs (e.g., OpenGL), but they are generally not suitable for interactive topological modeling sessions or geometry processing, since most topological edits or adjacency queries have a  $O(n + m)$  complexity.

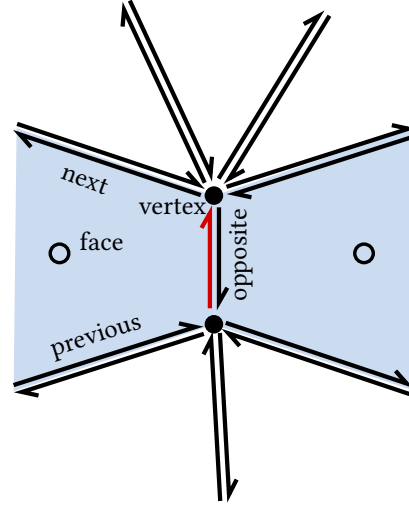
**Winged-Edge Data Structure** The winged-edge data structure [Baumgart 1972] is a simple data structure that supports topological modeling for manifold, orientable polygonal meshes. Most local topological edits and adjacency queries can be achieved in amortized  $O(1)$  or  $O(\ln(n))$  time. The idea is to store, for each edge  $e$ , its two end vertices  $v_{\text{start}}$  and  $v_{\text{end}}$  (in an arbitrary order), its two incident faces  $f_{\text{left}}$  and  $f_{\text{right}}$  (in an order consistent with  $v_{\text{start}}$  and  $v_{\text{end}}$ ), and its four “wing edges”, that is, the four edges which are incident to  $e$  and in the boundary of either  $f_{\text{left}}$  or  $f_{\text{right}}$  (see Figure 2.12). Each vertex (resp. face) arbitrarily points to one of its incident (resp. boundary) edges. The other incident (resp. boundary) edges can be discovered by traversing the structure.



```

1 class Vertex {
2     Point p;
3     Halfedge *halfedge;
4 };
5
6 class Halfedge {
7     Vertex *vertex;
8     Face *face;
9     Halfedge *previous, *next;
10    Halfedge *opposite;
11 };
12
13 class Face {
14     Halfedge *halfedge;
15 };
16
17 class HalfedgeDS {
18     std::set<Vertex*> vertices;
19     std::set<Halfedge*> halfedges;
20     std::set<Face*> faces;
21 };

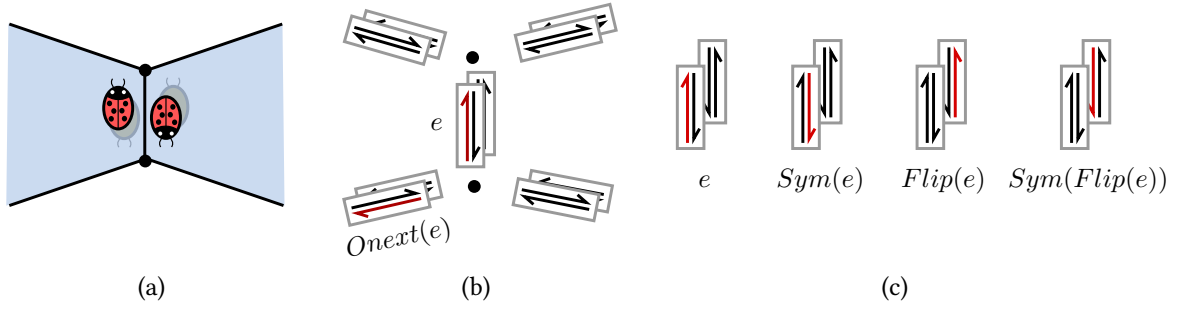
```



**Figure 2.13:** Example implementation of the halfedge data structure.

**Halfedge Data Structure** The halfedge data structure [Weiler 1985, Kettner 1999] is equivalent to the winged-edge data structure, but allows for slightly more efficient traversal by splitting each edge into two *halfedges* (see Figure 2.13). For instance, traversing all boundary edges of a given face can be directly achieved via `face->halfedge->next->next->next->...`, while with the winged-edge data structure, it requires some bookkeeping and a conditional statement to decide whether "next" should be `endLeft` or `startRight`. However, this performance gain is in fact negligible, and not the primary reason to prefer this structure. More fundamentally, it is useful to be able to refer to "a given edge, as seen from the perspective of a given face", which is one way to interpret the concept of halfedge. This interpretation turned out to be a key idea that most subsequent topological data structures relied upon, equivalent to the concept of *edge-use* extensively analyzed by Weiler in his PhD dissertation [Weiler 1986]. This concept was later formalized and generalized to arbitrary dimension by [Brisson 1989] with the introduction of cell-tuples, which we detail later in this section. But for now, let us just observe that in the case of orientable surfaces without boundary, each edge is *used* exactly two times (i.e., each edge has exactly two incident faces), which is why we only need two "halfedges" to represent this class of surfaces.

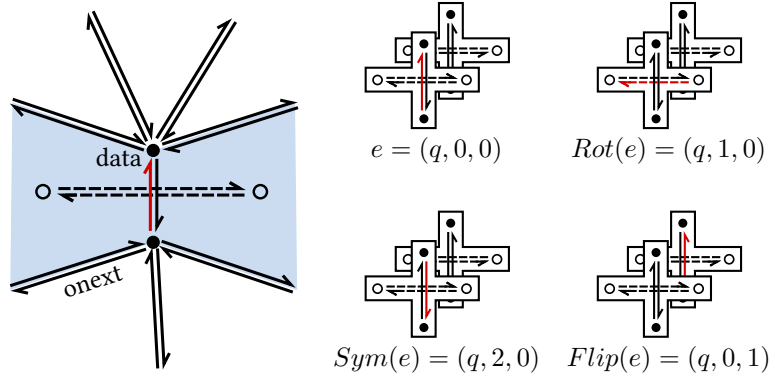
**Quad-Edge Data Structure** The quad-edge data structure [Guibas and Stolfi 1985] extends the ideas of the halfedge data structure in order to represent non-orientable surfaces. Intuitively, the ability to refer to "an edge, as seen from a face" is extended to also specify from which side of the surface the edge is considered. As suggested by the original authors, imagine a small bug



**Figure 2.14:** Illustration of the concept of edge direction and edge orientation in the quad-edge data structure. (a) The four different ways to walk along an edge, keeping the edge on your right. (b)  $Onext(e)$  is the edge obtained by rotating counterclockwise around the origin vertex of  $e$ . (c)  $Sym(e)$  is the edge obtained by switching the direction of  $e$ , and  $Flip(e)$  is the edge obtained by switching the orientation of  $e$ .

```

1 class EdgeRef {
2     QuadEdge *q;
3     int r;
4     int f;
5 };
6
7 class QuadEdge {
8     Data data[4];
9     EdgeRef onext[4];
10 };
11
12 class QuadEdgeDS {
13     std::set<QuadEdge*>
14         quadEdges;
15 };
    
```



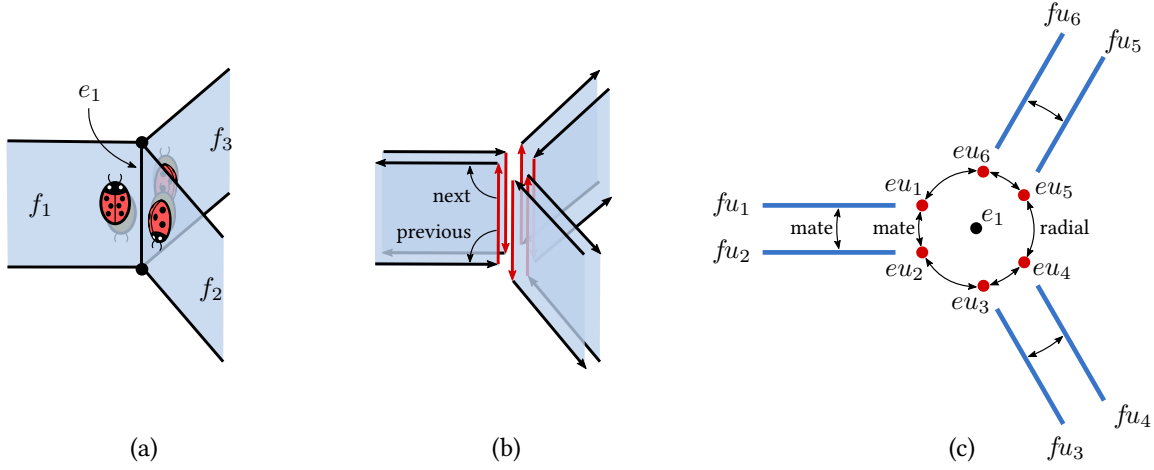
**Figure 2.15:** Example implementation of the quad-edge data structure.

walking on the surface, along an edge (see Figure 2.14a). The bug can be on either side of the edge, and on either side of the surface, resulting in four different configurations. In the quad-edge data structure, these four configurations are represented by splitting each edge into four edges, each with a given *direction* (from which vertex to which vertex), and a given *orientation* (which side of the surface)<sup>1</sup>. In order to traverse the data structure, each directed oriented edge  $e$  points to three other directed oriented edges:  $Sym(e)$ ,  $Flip(e)$ , and  $Onext(e)$  (see Figure 2.14b,c).

In addition, the authors make the choice to explicitly represent in their data structure the dual of the mesh. This means that each of the four directed oriented edge  $e$  is paired with a dual

<sup>1</sup>Interestingly, while many authors use *direction* and *orientation* interchangeably, those who do make a difference are generally consistent with the usage of Guibas and Stolfi. Fundamentally, *direction* is a purely one-dimensional concept, while *orientation* requires the existence of two-dimensional objects (e.g., “edge direction” vs. “surface orientation”). One can say that a closed edge is “oriented clockwise”, but note how this only makes sense if the edge is embedded in an orientable surface, and an orientation of this surface is chosen (this choice may be implicit). If the edge is embedded in a non-orientable surface, one can still choose a *local orientation* of the surface, which is what the quad-edge data structure effectively does.





**Figure 2.16:** Illustration of the concept of edge-use and face-use in the radial-edge data structure. (a) The six different ways to walk along an edge shared by three faces. (b) The corresponding six edge-uses (red), six face-uses (blue), and the previous and next pointers of one of the six edge-uses. (c) Cross-section of these edge-uses and face-uses, and their corresponding mate and radial pointers. Source: inspired from [Weiler 1986], Figure 17 – 6, page 198.

edge, effectively splitting in eight the original non-directed non-oriented edges of the mesh (see Figure 2.15, right). To navigate between primal and dual elements, each of the eight directed oriented edge  $e$  points to  $Rot(e)$ , the dual edge obtained after a “90° counterclockwise rotation around its midpoint”. While this representation of the dual mesh is quite elegant and allows for a very compact implementation, it is in fact less fundamental than the aforementioned concept of edge direction vs. orientation. Indeed, as noted by [Brisson 1989], the dual elements are not necessary for representational power, and a variant of the quad-edge data structure without these dual elements would still be able to represent non-orientable surfaces.

As detailed in [Guibas and Stolfi 1985], implementations of the quad-edge data structure do not need to allocate separate objects for each of the eight directed oriented edges. Instead, a single *quad-edge*  $q$  can be used as reference, and all eight edges  $e$  can be referred to as triplets  $e = (q, r, f)$ , with  $r \in \{0, 1, 2, 3\}$  and  $f \in \{0, 1\}$ . This way, the functions  $Rot(e)$ ,  $Flip(e)$ , and  $Sym(e)$  are implicitly given by  $Rot(q, r, f) = (q, r + 1, f)$ ,  $Flip(q, r, f) = (q, r, f + 1)$ , and  $Sym(q, r, f) = Rot^2(q, r, f) = (q, r + 2, f)$ , thus do not require any additional storage. Only  $Onext(e)$  needs to be explicitly stored, and taking advantage of symmetries, storing four of the eight  $Onext(e)$  per quad-edge is actually enough (see Figure 2.15, left).

**Radial-Edge Data Structure** The radial-edge data structure [Weiler 1986] further extends the ideas of the halfedge and quad-edge data structures, in order to represent non-manifold surfaces, such as when an edge is shared by three faces (Fig. 2.16a). Intuitively, in this specific case, the edge is *used* six times: once for each of the two sides of each of the three incident faces (Fig. 2.16b).

```

1  class RadialEdgeDS {
2      Model *models;
3  };
4
5  class Model {
6      Model *previous, *next;
7      Region *regions;
8  };
9
10 class Region {
11     Model *owningModel;
12     Region *previous, *next;
13     Shell *shells;
14 };
15
16 class Shell {
17     Region *owningRegion;
18     Shell *previous, *next;
19     // Mut. excl. alternatives:
20     // 1. Non-degenerate
21     FaceUse *faceUses;
22     // 2. Wireframe
23     EdgeUse *edgeUses;
24     // 3. Steiner vertex
25     VertexUse *vertexUse;
26 };
27
28 class Face {
29     FaceUse *faceUses;
30     FaceGeometry geometry;
31 };
32
33 class Loop {
34     LoopUse *loopUses;
35 };
36
37 class Edge {
38     EdgeUse *edgeUses;
39     EdgeGeometry geometry;
40 };
41
42 class Vertex {
43     VertexUse *vertexUses;
44     VertexGeometry geometry;
45 };
46
47 class FaceUse {
48     Shell *owningShell;
49     FaceUse *previous, *next;
50     FaceUse *mate;
51     LoopUse *loopUses;
52     Face *face;
53     bool orientation;
54 };
55
56 class LoopUse {
57     FaceUse *owningFaceUse;
58     LoopUse *previous, *next;
59     LoopUse *mate;
60     Loop *loop;
61     // Mut. excl. alternatives:
62     // 1. Non-degenerate
63     EdgeUse *edgeUses;
64     // 2. Steiner vertex
65     VertexUse *vertexUse;
66 };
67
68 class EdgeUse {
69     VertexUse *startVertexUse;
70     EdgeUse *mate;
71     Edge *edge;
72     // Mut. excl. alternatives:
73     // 1: Used as wireframe of
74     //    shell
75     Shell *owningShell;
76     // 2: Used as edge of
77     //    loop use
78     LoopUse *owningLoopUse;
79     EdgeUse *previous, *next;
80     EdgeUse *radial;
81     bool direction;
82 };
83
84 class VertexUse {
85     VertexUse *previous, *next;
86     Vertex *vertex;
87     // Mut. excl. alternatives:
88     // 1. Used as Steiner vertex of
89     //    shell
90     Shell *owningShell;
91     // 2. Used as Steiner vertex of
92     //    loop use
93     LoopUse *owningLoopUse;
94     // 3. Used as start vertex of
95     //    edge use
96     EdgeUse *owningEdgeUse;
97 };

```

**Figure 2.17:** Example implementation of the radial-edge data structure. “Mut. excl.” is a shorthand for “Mutually exclusive”, which means that only one of the alternatives has non-null pointers.

The two sides of each face are explicitly represented as two *face-uses*, all uses of each edge are explicitly represented as *edge-uses*, and all uses of each vertex are explicitly represented as *vertex-uses*. By “explicitly represented”, we mean that objects of type `VertexUse`, `EdgeUse` and `FaceUse` are dynamically allocated (i.e., they have an *identity*), and they store pointers to neighboring uses in a linked-list fashion (see Figure 2.16b,c and Figure 2.17). This concept of explicit uses also exists in the halfedge data structure (each halfedge can be interpreted as one edge-use), but contrasts with the quad-edge data structure and the VGC, where these uses are implicit<sup>2</sup>.

In addition, unlike the other data structures that we have seen so far in this section (but like the W3C SVG representation, planar maps, and the representations we introduce in this thesis), the radial-edge data structure allows faces to have inner holes. In other words, faces are not necessarily topological disks. However, they must still be genus-0 (i.e., orientable and without handles), unlike in our PCS complex where we also waive this restriction. Both the outer boundary and inner holes (which, for curved surfaces, are topologically indistinguishable) are represented in the radial-edge data structure using the concept of *loop*, which is analogous to our concept of *cycle*<sup>3</sup>. Each loop is used exactly twice—once for each side of the face it belongs to—and these two loop-uses point to one another via their *mate* pointer. However, one can notice that the edge-uses that compose a given loop-use essentially contain the same information as the edge-uses that compose its mate loop-use. There exist many variants of the radial-edge data structures (e.g., [Gursoz et al. 1990, Marcheix and Gueorguieva 1995]), and the partial entity structure [Lee and Lee 2001] is one that factorizes this duplicated information to propose a more compact representation.

Finally, the radial-edge data structure also defines the concepts of *regions* and *shells*, which are the three-dimensional counterparts of *faces* and *loops*, in order to represent solid objects. Since the VGC only aims at representing two-dimensional objects, it has no equivalent of these concepts.

**Nef Polyhedra** A Nef polyhedron [Nef 1978] is a point-set of  $\mathbb{R}^n$  defined via boolean operations of halfspaces. In other words, in the case of  $\mathbb{R}^3$ , it represents an arrangement of 3D solids, possibly non-manifold and with degeneracies (e.g., isolated points), whose boundaries are defined by planes, i.e., edges are straight lines and faces are non-curved surfaces. [Granados et al. 2003] in-

---

<sup>2</sup>In the case of the VGC, this is reflected by the fact that the classes `Vertex`, `Edge`, and `Face` are used with pointer semantics, but the classes `Halfedge` and `Cycle` are used with value semantics, making it impossible to “point” to a halfedge. However, we acknowledge that this implicit vs. explicit choice is more an implementation detail than a fundamental property of the model. For instance, instead of being implemented as “`std::vector<Halfedge> halfedges`”, VGC cycles may be implemented as “`EdgeUse *edgeUses`”, where `edgeUses` is the first node of a doubly linked-list, each node pointing to the “previous” and “next” edge-use, and possibly also pointing to “radial” edge-uses. These choices correspond to trade-offs between readability, maintainability, storage costs, and performance costs, which we do not analyze in this thesis.

<sup>3</sup>In the VGC, we favored the terminology *cycle* instead of *loop* for consistency with graph theory and algebraic topology: a (directed) path is a sequence of (directed) edges, and a cycle is simply a closed path.

```

1  template <unsigned int n>
2  class Dart {
3      Dart* beta[n];
4  };
5
6  template <unsigned int n>
7  class CombinatorialMap {
8      std::vector<Dart<n>*> darts;
9  };
    
```

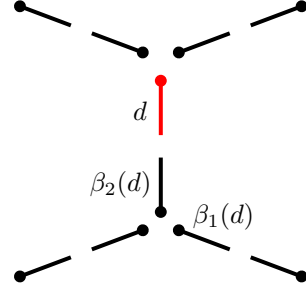


Figure 2.18: Example implementation of combinatorial maps.

introduces a data-structure similar to the radial-edge data structure to represent such non-manifold spaces. The concept of sphere map, first introduced in [Dobrindt et al. 1993], is used to provide an explicit representation of the possibly non-manifold geometry around each vertex, unlike in the radial-edge data structure where all vertex-uses of a given vertex are stored in no particular order, and unlike in the VGC where even incident faces of a given edge are not ordered (since this information is meaningless for vector graphics). However, since Nef polyhedra only represent non-curved surfaces, non-orientable surfaces or edges used three times by the same face are not supported, unlike in the radial-edge data structure and the VGC.

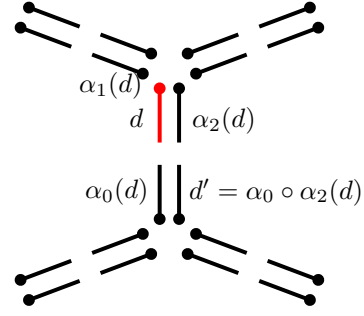
**Combinatorial Maps** Combinatorial maps [Edmonds 1960, Lienhardt 1989, Lienhardt 1991, Lienhardt 1994, Damiand and Lienhardt 2014], or  $n$ -maps, can be seen as an extension of the halfedge data structure for arbitrary dimension. An  $n$ -dimensional combinatorial map is defined as a tuple  $(D, \beta_1, \dots, \beta_n)$  where:

1.  $D$  is a finite set of elements called *darts*;
2.  $\beta_1$  is a permutation on  $D$ ;
3.  $\beta_2, \dots, \beta_n$  are involutions on  $D$ ;
4.  $\forall i \leq [1..n-2], \forall j \leq [i+2..n], \beta_i \circ \beta_j$  is an involution on  $D$ .

Let us clarify this definition in the case of the dimension 2. The first condition states that the structure is made of objects called *darts*. Imagine cutting each edge in two pieces at the midpoint, each of the two pieces would be a dart (see Fig. 2.18). You can also interpret each dart as a pair (vertex, edge), i.e., a vertex "as seen from an edge", or a halfedge. The second condition states that each dart  $d$  points to a "next" dart  $\beta_1(d)$ . This is the equivalent of the "next" pointer in the halfedge data structure.  $\beta_1$  is a permutation, which simply means that it is invertible. In other words, each dart also has a "previous" dart  $\beta_1^{-1}(d)$ . The third condition states that each dart  $d$  points to an "opposite" dart  $\beta_2(d)$ . This dart corresponds to the second piece of the original edge that was cut at the midpoint.  $\beta_2$  is an involution, which simply means that the opposite of the opposite of a

```

1  template <unsigned int n>
2  class Dart {
3      Dart* alpha[n+1];
4  };
5
6  template <unsigned int n>
7  class GeneralizedMap {
8      std::vector<Dart<n>*> darts;
9  };
    
```



**Figure 2.19:** Example implementation of generalized maps.

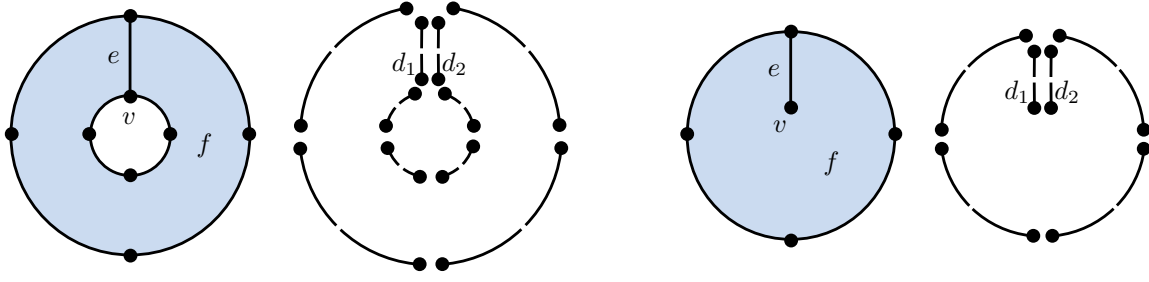
dart is the same dart. In other words,  $\beta_2$  groups the darts by pairs, each pair being one edge. The last condition ensures that topological constraints are satisfied, which is always true for  $n = 2$ . Please refer to [Lienhardt 1989, Levy and Mallet 1999] for more details on why the condition is required for higher dimensions. Note that unlike the VGC, combinatorial maps do not support non-orientable or non-manifold surfaces.

**Generalized Maps** Generalized maps [Lienhardt 1989, Lienhardt 1991, Lienhardt 1994, Damiand and Lienhardt 2014], or  $n$ -G-maps, are a generalization of combinatorial maps to support non-orientable surfaces. An  $n$ -dimensional generalized map is defined as a tuple  $(D, \alpha_0, \dots, \alpha_n)$  where:

1.  $D$  is a finite set of elements called *darts*;
2.  $\alpha_0, \dots, \alpha_n$  are involutions on  $D$ ;
3.  $\forall i \leq [0..n-2], \forall j \leq [i+2..n], \alpha_i \circ \alpha_j$  is an involution on  $D$ .

Let us clarify this definition in the case of the dimension 2. Unlike combinatorial maps, each original edge of the represented surface is not split in two, but in four darts (see Fig. 2.19). Similarly to the quad-edge data structure, this additional granularity permits the representation of non-orientable surfaces. Each dart can be interpreted as a triplet (vertex, edge, face), i.e., a vertex "as seen from an edge, as seen from a face". For each dart  $d = (v, e, f)$ , the involution  $\alpha_0$  pairs  $d$  with the only dart  $\alpha_0(d) = (v', e, f)$  seen from the same edge and the same face, but a different vertex. The involution  $\alpha_1$  pairs  $d$  with the only dart  $\alpha_1(d) = (v, e', f)$  seen from the same vertex and the same face, but a different edge. Finally, the involution  $\alpha_2$  pairs  $d$  with the only dart  $\alpha_2(d) = (v, e, f')$  seen from the same vertex and the same edge, but a different face (see Fig. 2.19).

The involution  $\alpha_2$  can be interpreted as "sewing" together the edges of different faces. Using this sewing interpretation, one can observe that  $\alpha_0$  and  $\alpha_2$  are not independent, since sewing one dart  $d$  mandates how  $\alpha_0(d)$  should be sewed. More precisely, for any given  $d$ , once  $\alpha_0(d)$  and  $\alpha_2(d)$  are defined (see Fig. 2.19), then it should be clear that  $d' = \alpha_0 \circ \alpha_2(d)$  is the fourth dart composing



**Figure 2.20:** Two examples of "self-sewing" faces, i.e., faces whose boundary is partly sewed to itself. In other words, the same edge is "used" two times by the same face. Unlike  $n$ -G-maps, cell-tuples cannot represent such faces, because the tuple  $(v, e, f)$  corresponds in fact to two darts  $d_1$  and  $d_2$ , instead of a single dart. Formally, this limitation is expressed by the fact that cell-tuples can only represent "regular" cell decompositions of  $n$ -manifolds (we give more detail in our following paragraph on regular CW complexes).

the edge, and that we also have  $d = \alpha_0 \circ \alpha_2(d')$ . Therefore,  $\alpha_0 \circ \alpha_2$  must be an involution, which is expressed by the third condition of the definition.

**Cell-Tuples** In the same year that Lienhardt introduced generalized maps<sup>4</sup>, Brisson introduced the cell-tuple structure [Brisson 1989]. Apart from not supporting "self-sewing" faces (see Figure 2.20), the cell-tuple structure is equivalent to generalized maps, but with a different formalism, centered around this idea of vertex "as seen from an edge, as seen from a face". The connection between cell-tuples and generalized maps is well explained and illustrated in [Levy and Mallet 1999]. Given a subdivided<sup>5</sup>  $n$ -manifold, Brisson first defines the familiar concept of incidence: a cell  $c$  is incident to another cell  $c'$  if and only if  $c$  is contained in the boundary of  $c'$ . In this case, we write  $c < c'$ . If more specifically, we have  $\dim(c) = \dim(c') - 1$ , then we write  $c \prec c'$ . This defines a partial ordering of the cells, which in turns defines the **incidence graph** of the subdivided manifold. Finally, a **cell-tuple** is defined as an  $n + 1$ -tuple  $(c_0, c_1, \dots, c_n)$  such that  $c_0 \prec c_1 \prec \dots \prec c_n$  (i.e., a path in the incident graph from a leaf to a root [Levy and Mallet 1999]). In the case  $n = 2$ , this simply means that a cell-tuple is a triplet  $(v, e, f)$  where  $v$  is an endpoint of  $e$ , and  $e$  is a boundary edge of  $f$ . Brisson then proves that for every  $c_{k-1} \prec c_k \prec c_{k+1}$ , there is a unique  $c'_k \neq c_k$  such that  $c_{k-1} \prec c'_k \prec c_{k+1}$ . For each  $k$  in  $[0..n]$ , this defines a function  $\text{switch}_k(c_0, \dots, c_{k-1}, c_k, c_{k+1}, \dots, c_n) = (c_0, \dots, c_{k-1}, c'_k, c_{k+1}, \dots, c_n)$  on cell-tuples. These functions are equivalent to the involutions  $\alpha_k$  that define generalized maps.

**Chains of Maps** Chains of maps [Elter and Lienhardt 1992, Elter and Lienhardt 1993, Elter and Lienhardt 1994, Damiand and Lienhardt 2014], or  $n$ -chains, are an extension of generalized maps to support non-manifold spaces. For conciseness, we only provide in this paragraph a simpli-

<sup>4</sup>In fact, at the same conference, Symposium on Computational Geometry '89.

<sup>5</sup>Formally, a manifold with a finite regular CW decomposition; informally, a manifold subdivided into vertices, edges, faces, etc.

```

1 class Vertex {
2     std::vector<Edge*> incidentEdges;
3 };
4
5 class Edge {
6     Vertex *startVertex, *endVertex;
7     std::vector<Triangle*> incidentFaces;
8 };
9
10 class Triangle {
11     Edge *e1, *e2, *e3;
12 };
13
14 class TriangleMesh {
15     std::vector<Vertex*> vertices;
16     std::vector<Edge*> edges;
17     std::vector<Triangle*> triangles;
18 };

```

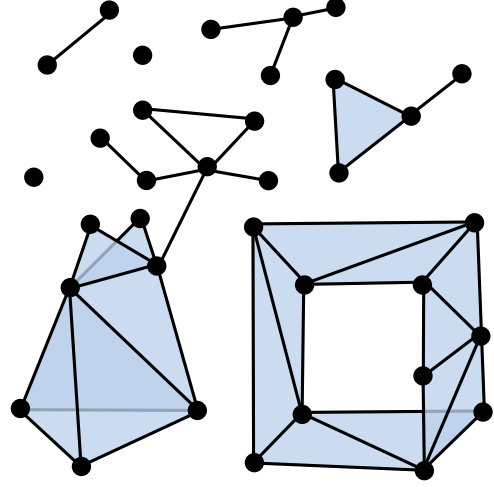
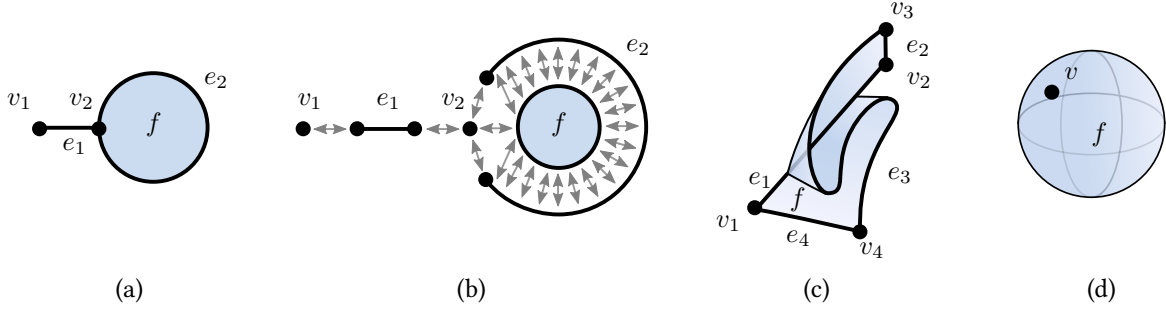


Figure 2.21: Example implementation of two-dimensional simplicial complexes.

fied, slightly inaccurate definition of  $n$ -chains; please refer to the original papers (e.g., [Elter and Lienhardt 1993]) for details. The idea is to iteratively build a non-manifold space by gluing the boundary of cells of higher dimensions to cells of lower dimension, a common technique used in algebraic topology (see CW complexes). More specifically, the  $i$ -th step of the process defines all cells of dimensions  $i$  via an  $i$ -G-map  $G^i = (D^i, \alpha_0^i, \dots, \alpha_i^i)$ , together with a tuple  $(\sigma_0^i, \dots, \sigma_{i-1}^i)$ , where  $\sigma_j^i$  is an application from  $D^i$  to  $D^j$ , gluing the boundary of  $i$ -cells to lower-dimensional  $j$ -cells. An  $n$ -chain is then defined as the structure  $((G^i)_{i=0, \dots, n}, (\sigma_j^i)_{0 \leq j < i \leq n})$  composed of all  $n+1$  generalized maps and all  $\sigma_j^i$ . While chains of maps are indeed able to represent non-manifold spaces, and provide a useful theoretical insight, they do not seem to have been widely used in practice. One reason may be that most application cases only focus on the dimension 2 or 3, for which more compact (and arguably more intuitive) representations exist.

**Simplicial Complexes** Simplicial complexes are one of simplest and most popular structures to represent non-manifold topological spaces. They are taught in most introductory class in algebraic topology [Munkres 2000, Hatcher 2001, Lee 2011], and are widely used in practice in the computer graphics industry [De Floriani et al. 2010]. In the two-dimensional case, a simplicial complex is simply a *triangle mesh* (possibly non-manifold and with isolated vertices and edges). We give an example and one of many possible implementations in Figure 2.21. Simplicial complexes easily scale to arbitrary dimension: we give a formal definition in Appendix A, together with some context and discussion. The simplicity and representational power of triangle meshes makes them highly suitable for many applications. However, their obvious disadvantage is that the represented objects must be decomposed into triangles. For some applications, this is neither desired



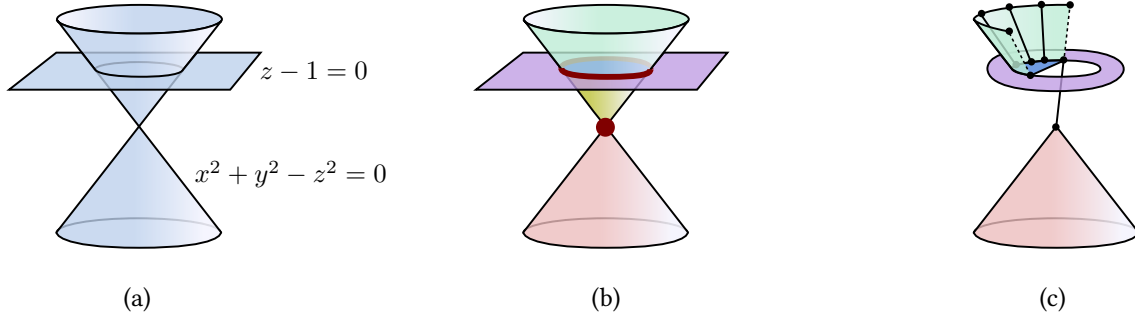


**Figure 2.22:** (a) Example of CW complex. (b) Illustration of the glue operations that define the CW complex. (c,d) Two more examples of valid CW complexes.

or practical, since it would result in many triangles, when a more compact subdivision with less cells could be used. For instance, in the case of vector graphics, shapes are naturally authored by drawing their boundary (a directed closed curve), and rendered using a winding rule. It would be prohibitive and non-intuitive to ask artists to manually author triangles, and while a triangulation can be automatically computed, a triangle mesh still would not capture the higher-level *semantics* of the drawing (for instance, defining per-triangle style attributes wouldn't make sense). One approach could be to use a triangle mesh together with a partition of the triangles into “meta-cells” (for instance, nearly manifold cells [De Floriani et al. 2003]), and only attach style attributes to these meta-cells. Our approach can be seen as directly representing these meta-cells without the underlying triangulation, which is possible for the dimension 2 thanks to the known classification of compact 2-manifolds (see Appendix A), but would be impossible for higher dimensions.

**CW Complexes** CW complexes [Munkres 2000, Hatcher 2001, Lee 2011] are one of the most (perhaps *the* most) used structures in modern algebraic topology to represent non-manifold spaces. Their definition, which we recall in Appendix A, provides a sweet balance between simplicity and genericity that makes them a powerful tool as a proof mechanism. Unfortunately, not all CW complexes can be represented with a computer due to their non-combinatorial nature, thus they are “too generic” for use within computer graphics. Nonetheless, their theoretical importance makes them highly relevant to the field of topological modeling, including this thesis, and we informally describe them in this paragraph. Like chains of maps, CW complexes are built iteratively by gluing the boundary of cells of higher dimension to cells of lower dimension (see Figure 2.22a,b). In the case of two-dimensional CW complexes, a “cell” means a point, an edge, or a topological disk. Note that such cells are continuous pointsets, that is, geometric objects (unlike in chains of maps, where cells are represented as darts, that is, combinatorial objects). For each cell  $c$ , a “glue” operation is defined as a *continuous* function  $\Phi_c$  from the boundary of the cell  $c$  to cells of lower dimension (unlike in chains of maps, where the glue operation is a *combinatorial* function from darts to darts). The function  $\Phi_c$  does not have any other restrictions than to be continuous, in particular, it does





**Figure 2.23:** (a) Example of algebraic variety. (b) Decomposition of the variety into extents, with singularities highlighted in red. (c) Example selective geometric complex (SGC) whose cells are open subsets of extents.

not have to be differentiable or invertible, which is what makes the class of all CW complexes very generic and not representable combinatorially (the set of all continuous functions is not countable). For example, in Figure 2.22c, the function  $\Phi_f$  makes a “switch-back” in the interior of  $e_1$ , and thus is not invertible. There isn’t even a requirement that a vertex exists at the switch-back. In Figure 2.22d, the whole boundary of the disk  $f$  is mapped by  $\Phi_f$  to a single vertex  $v$ , thus creating a sphere. This last situation is allowed neither by chains of maps nor by the radial-edge structure, but is allowed by our PCS complexes and vector graphics complexes. In fact, notice how in this example, there isn’t even a single edge! Therefore, it is obvious that the concept of cell-tuple cannot be used to represent such a subdivision.

**Regular CW Complexes** If for each cell  $c$ , the function  $\Phi_c$  is in fact invertible and its inverse is continuous (i.e., if  $\Phi_c$  is an *homeomorphism*, more on this later), then the CW complex is called *regular*. In other words, regular CW complexes do not allow the boundary of cells to be glued to themselves, as in Figure 2.22c,d, or in Figure 2.20. With this additional restriction, all (finite) regular CW complexes can in fact be represented by any of the aforementioned non-manifold structures, except simplicial complexes. However, in the context of vector graphics, it is sometimes useful to represent non-regular CW complexes. In fact, as we detail in Chapter 3, it is even useful to represent cell decompositions which are not CW complexes at all, for instance decompositions that involve disks with inner holes represented as one single face. Such faces are already allowed by SVG, planar maps, the radial-edge data structure, and selective geometric complexes.

**Selective Geometric Complexes** Simplicial complexes and CW complexes share the common idea to build non-manifold spaces by gluing together simple manifold pieces (respectively,  $n$ -simplices and  $n$ -disks). Selective geometric complexes (SGCs) [Rossignac and O’Connor 1989] and their extension structured topological complexes (STCs) [Rossignac 1997] take the opposite approach by starting with a given non-manifold space, and decomposing it into manifold pieces, defining what they call a *geometric complex*. More specifically, the definition of a SGC starts with

the concept of *algebraic variety*, that is, a subset of  $\mathbb{R}^n$  defined as the zero level set of a given finite set of degree- $n$  polynomials (see Figure 2.23a). Such variety generally has *singularities* (cusps, self-crossing, isolated lower-dimensional pieces, etc.), and these singularities define a partition of the variety into *extents*, where each extent is a connected sub-manifold of the variety (see Figure 2.23b). A *cell* is then defined as any connected open subset of an extent, and a *geometric complex* is defined as a finite collection of cells  $c_j$ , with  $j \in J$ , such that:

1.  $\forall i \neq j, c_i \cap c_j = \emptyset$
2.  $\forall j \in J, \exists I \subset J, \partial c_j = \bigcup_I c_i$

A *selective* geometric complex is then defined as a geometric complex where each cell stores an extra attribute called “active”, specifying whether the cell should be included in the pointset defined by the SGC (see Figure 2.23c, where dashed lines represent inactive edges). This allows SGCs to represent non-closed pointsets.

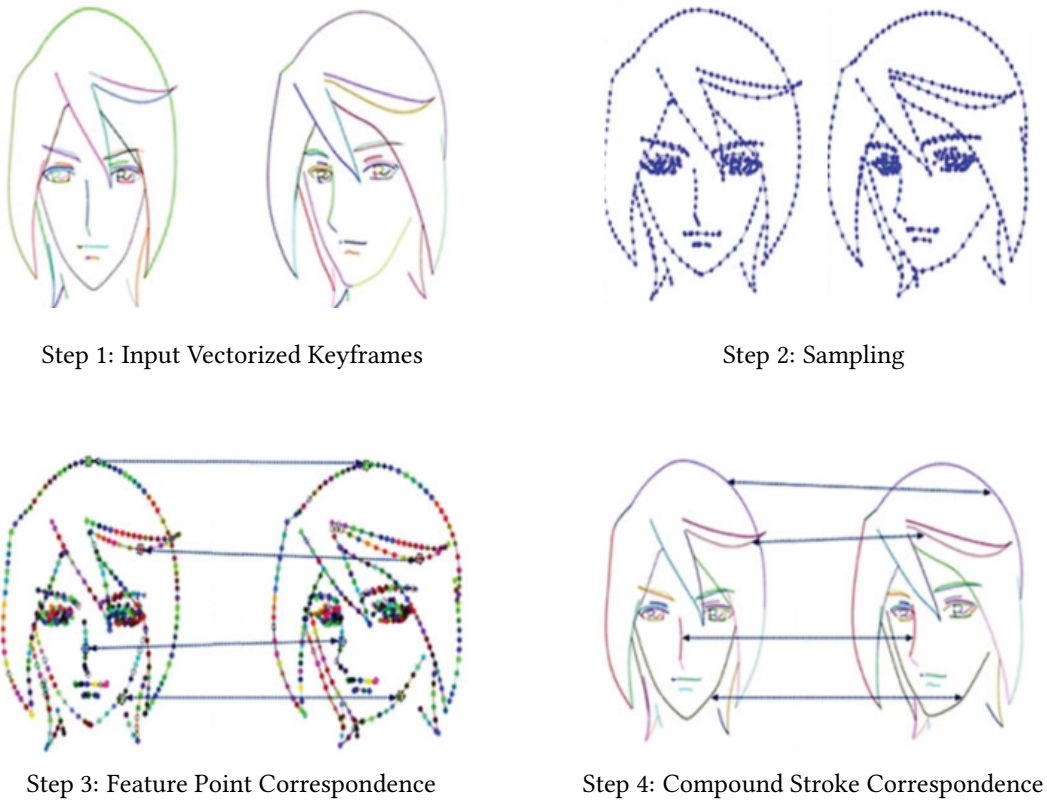
Note how the definition of geometric complexes allows closed edges and disks with inner holes, as illustrated in Figure 2.23c (more generally, it allows cells which are not necessarily homeomorphic to  $n$ -disks). This was a deliberate and important design decision taken by the authors, which they motivate in [Rossignac 1997, Section 3.6]. Our PCS complexes and vector graphics complexes also allow such cells, for very similar reasons, which we detail in Chapter 3. This makes SGCs and PCS complexes very similar in terms of intent and topological expressiveness. In fact, the notion of SGC was an important source of inspiration for the design of our structures. Unfortunately, due to their definitions relying on explicit embeddings in  $\mathbb{R}^n$ , SGCs lack the geometric flexibility that we desire for vector graphics. Notably, like planar maps, SGCs do not support the concept of *overlapping*. This stems from the lack of a purely combinatorial definition from which to define an immersion. More precisely, while a combinatorial structure is associated with every SGC (an augmented incidence graph), this combinatorial structure alone does not fully specify the non-manifold object up to homeomorphism. In other words, some topological information is encoded in the polynomial coefficients of the underlying algebraic variety, such as whether a face is orientable, or how many times a bounding edge is “used” by the face. Finally, even though STCs cyclically order the faces around any given edge, neither SGCs nor STCs cyclically order the edges bounding a given face. For vector graphics applications, we require such cycles for the unambiguous computation of winding numbers, used for rendering.

## 2.4 Related Work in Animation

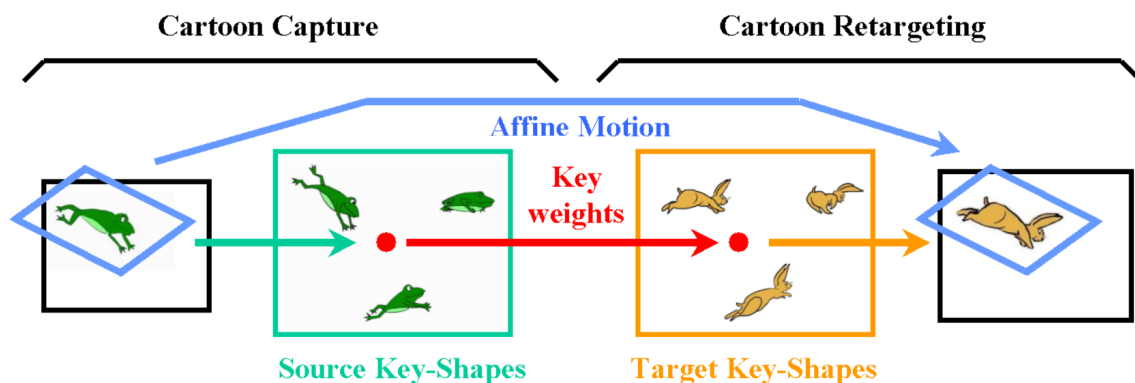
**Topology-unaware inbetweening** Cartoon animation [Thomas and Johnston 1987, Blair 1994, Williams 2009] consists in drawing a finite sequence of pictures that gives the illusion of motion.



**Figure 2.24:** Three keyframes with inconsistent topology, making automatic inbetweening challenging. Original design by James Lopez, used with permission.



**Figure 2.25:** Illustration of the different steps to compute stroke correspondence using shape descriptors and manifold learning. Source: modified from [Liu et al. 2011], permission pending.

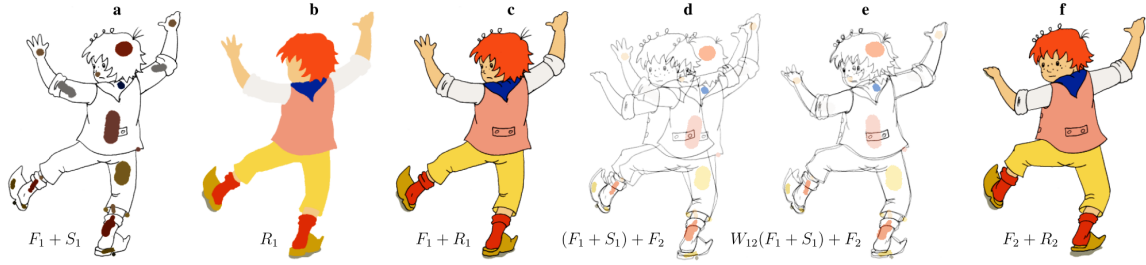


**Figure 2.26:** Illustration of an approach that transfer existing cartoon motion to new shapes. Source: [Bregler et al. 2002], used with permission.

It was expected that automatic inbetweening of vectorized strokes would make cartoon animation easier, but this task appeared to be much more complex than expected [Catmull 1978], one reason being inconsistent *topology* between keyframes (see Figure 2.24). Early stroke-based approaches [Burtnyk and Wein 1971, Reeves 1981, Fekete et al. 1995] are *manual* (the animator selects pairs of strokes to interpolate) and *topology-unaware* (strokes are interpolated independently, unaware of their neighbors). More recently, [Liu et al. 2011, Yu et al. 2012] have proposed to use shape descriptors and machine learning techniques to compute stroke correspondences automatically, however, the shape descriptors are based on stroke sampling and therefore are also topology-unaware (see Figure 2.25). Nonetheless, once the per-sample correspondence is computed, this information is aggregated per-stroke, possibly allowing one stroke to split into several strokes, as we also allow in our vector animation complex, among many other topological events.

**Topology-aware inbetweening** [Kort 2002] introduces semantic relations between strokes (e.g., *intersecting*, or *dangling*), together with inference rules to find stroke correspondences automatically. To the best of our knowledge, this is the first work attempting to leverage topology to solve the correspondence problem. However, this approach could only be applied for cases that can be resolved via an invariant layering, and it is unclear whether it can scale to complex examples. Later, [Whited et al. 2010] introduced stroke graphs, a concept which we already presented in Section 2.2. Given two stroke graphs and initial stroke correspondences, the two graphs can be traversed in parallel to propagate stroke correspondences, stopping at topological inconsistencies. Unfortunately, unlike the method we introduce in Chapter 5, none of these methods can produce space-time continuous animations with time-varying topology, since it is not allowed by their representation. Also, none of these methods address coloring.

**Data-driven inbetweening** An alternative approach to generate cartoon animations is to reuse existing content. [Bregler et al. 2002] extracts animated affine transformations and weight coeffi-



**Figure 2.27:** Automatic painting using LazyBrush: (a) original drawing  $F_1$  with color scribbles  $S_1$ , (b) segmentation  $R_1$  of  $F_1$  using scribbles  $S_1$ , (c) painted drawing  $(F_1 + R_1)$ , (d) initial overlap of the source  $F_1$  with scribbles  $S_1$  and the target frame  $F_2$ , (e) registration of the source frame  $F_1$  with the target frame  $F_2$ , and transfer of scribbles  $S_1$  using as-rigid-as-possible deformation field  $W_{12}$ , (f) painted drawing  $(F_2 + R_2)$  using transferred scribbles  $W_{12}(S_1)$ . Source (image + caption): [Sýkora et al. 2011], used with permission.



**Figure 2.28:** Example inbetweens obtained using shape morphing. Source: [Baxter et al. 2009], used with permission.

cients from existing cartoons, which can be transferred to new shapes (see Figure 2.26). [de Juan and Bodenheimer 2006] performs a semi-automatic segmentation of the input video to combine parts of existing content together. New inbetweens can be generated by defining an implicit space-time surface interpolating extracted contours, however, no change in topology is allowed, since interpolated contours are always closed curves. [Zhang et al. 2009] proposes a method to vectorize input cartoon animations, allowing to edit them. However, their outputs are stacked layers of paths, thus cannot represent topological events.

**Shape morphing** Another way to generate inbetweens is shape morphing, where a *shape* is a closed curve (its boundary), together with a raster image (its interior). The simplest method to interpolate two given shapes consists in computing a linear interpolation of the positions of all samples along the boundary. Unfortunately, this method typically causes "shrinkage", that is, it does not preserve the area of the shapes to interpolate. Instead, [Sederberg et al. 1993] proposes to interpolate the lengths and relative angles of curve segments. Later, [Alexa et al. 2000] proposes to interpolate compatible triangulations of the shapes by minimizing an as-rigid-as-possible energy, and uses texture blending for the interior pixels. This technique was widely successful, and several improvements have been developed [Fu et al. 2005, Baxter et al. 2009]. Also, the approach has been adapted for interactive shape manipulation [Igarashi et al. 2005]. One of the first steps in these methods consist in finding an initial arc-length correspondence between the two closed curves, which can be achieved using curvature-based methods [Sebastian et al. 2003]. An alternative approach to shape morphing is introduced by [Sýkora et al. 2009], where they align the two

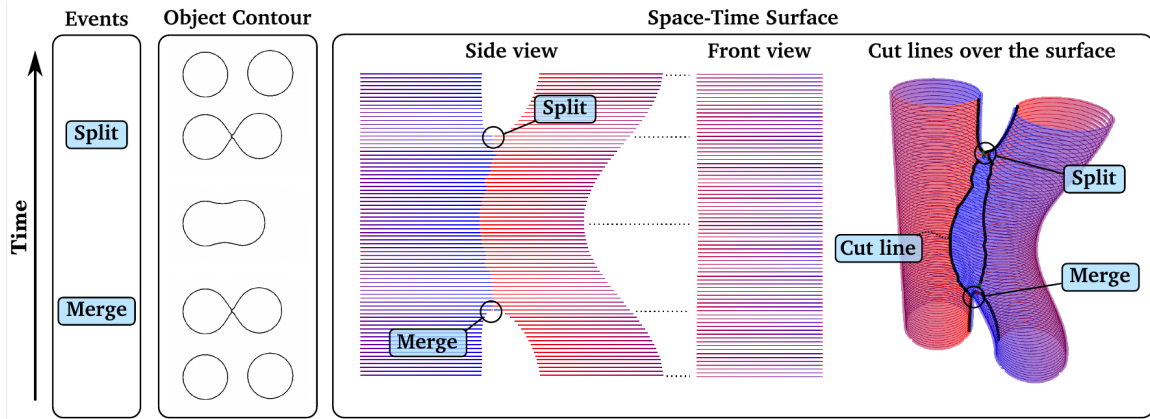


**Figure 2.29:** Example of non-photorealistic rendering obtained using *snaxels*. Source: [Karsch and Hart 2011], used with permission.

shapes using an iterative method, alignment which can be used for temporal noise control [Noris et al. 2011], or texture transfer [Sýkora et al. 2011], as illustrated in Figure 2.27. Unfortunately, none of these methods can produce space-time continuous animations of vectorized curves with changing topology, since by definition every shape has the topology of a disk, and their interior is not vectorized, typically leading to blurring artifacts (see Figure 2.28).

**Stylizing 3D animation** A natural approach to handle image-space topological events is to animate in a different space where no topological events occur, e.g., 3D animation [Lasseter 1987], in which case a fixed number of degrees of freedom can be keyframed independently. From a 3D model, one can compute vectorized 2D feature lines (e.g., [Bénard et al. 2014]), from which it is possible to extract cycles for coloring using depth-ordered paths [Eisemann et al. 2009] or planar maps [Karsch and Hart 2011], which can be further processed in 2D for stylization (see Figure 2.29). However, the 3D-to-2D conversion is typically a per-frame process and therefore does not output a time-continuous 2D animation. To address this issue, [Karsch and Hart 2011] track *snaxels*' 3D positions in the original mesh to generate correspondences across 2D frames, [Buchholz et al. 2011] computes a parameterization of the space-time surface swept by the silhouette lines, and [Bénard et al. 2012] uses an image-space relaxation method to deform, split and merge active strokes at frame  $i$  to match the feature lines of frame  $i + 1$ . Unfortunately, unlike the method we introduce in Chapter 5, all of these methods require to create a 3D animation beforehand. In addition, their output representation either does not support vectorized coloring [Buchholz et al. 2011, Bénard et al. 2012], or breaks the animation into contour sequences that do not change in topology [Karsch and Hart 2011]. In Chapter 5, we present a novel representation that we believe could be used as output of these existing methods to address their limitations.





**Figure 2.30:** Representing an animation as a space-time non-manifold object. Source: [Buchholz et al. 2011], used with permission.

**Using hybrid “2.5D” models** To have better image-space control of style, but still animate in a space free from topological events, [Fiore et al. 2001, Rivers et al. 2010] introduce hybrid models where shapes are defined in 2D, but their interpolation and depth-ordering is guided by 3D information. Unfortunately, these approaches tend to reduce the space of possible animations (compared to freeform hand-drawn animation), and only allow the representation of topological events which are solved by depth ordering.

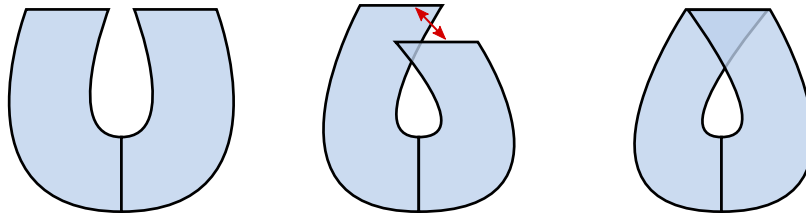
**Space-time modeling** Finally, another approach to generate 2D animations, the one we adopt in Chapter 5 of this thesis, is to consider animated lines as surfaces in space-time [Fausett et al. 2000, Kwarta and Rossignac 2002, de Juan and Bodenheimer 2006, Southern 2008, Buchholz et al. 2011], and animated faces as volumes in space-time [Fausett et al. 2000, Southern 2008]. Therefore, *animating* becomes *modeling in space-time*, which makes possible to represent topological events, unlike when using the model-then-animate paradigm (see Figure 2.30). The time dimension can also be replaced by more abstract parameters [Ngo et al. 2000, Fausett et al. 2000], leading to 4D or even higher dimensional objects. Recently, space-time meshes have also been used for fluid simulation inbetweening [Raveendran et al. 2014]. In theory, any non-manifold topological representation could be used to apply these concepts, as long as they can represent objects of sufficiently high dimension. For instance, simplicial complexes [De Floriani et al. 2010] could be a natural choice for their simplicity and scalability in dimension. Other popular non-manifold representations which we have presented in Section 2.3, such as selective geometric complexes [Rossignac and O’Connor 1989], or the radial-edge data structure [Weiler 1985], could be used as well. However, none of these representations have been *designed* to represent space-time objects. Therefore, while they are very appropriate for their intended use (e.g., solid modeling), they are less intuitive to manipulate for space-time modeling. What makes our concept of vector anima-

tion complex stands out is that by design, it treats the time dimension separately from the space dimensions, enabling a keyframing paradigm similar to what animators are already familiar with. For instance, instead of a 1D entity called “edge”, we make the distinction between two types of 1D entities: a *key edge* (1D in space; 0D in time), which represents an edge at a given time, and an *inbetween vertex* (0D in space; 1D in time), which represents an interpolation between two key vertices. Even though they are both 1D in space-time, they are created, edited, and visualized differently, reflecting a cleaner semantics.



## Chapter 3

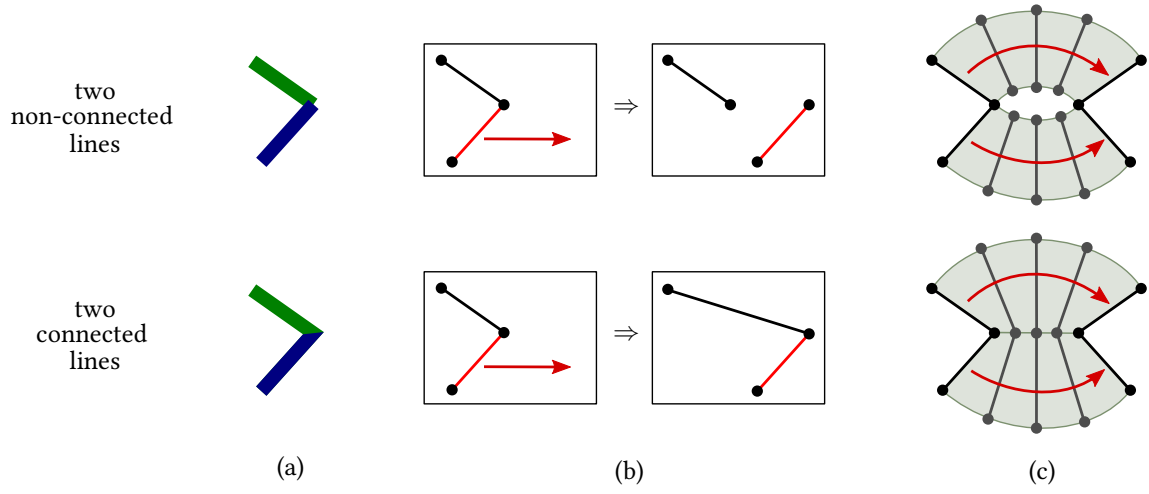
# The Theoretical Foundations of Vector Graphics Topology



**Figure 3.1:** *Allowing 2D vector graphics shapes to overlap and to share edges enables users to depict non-planar topologies, possibly non-orientable, such as here a Möbius strip.*

Vector graphics technologies have been around for a long time. In fact, the vector graphics system Sketchpad [Sutherland 1963] is generally considered to have pioneered interactive computer graphics, which makes vector graphics one of the oldest subfields of computer graphics. However, despite its seniority, the theoretical foundations required to rigorously study topological modeling for vector graphics are still lacking. In this chapter, we introduce the notion of PCS complex as a formal tool to better understand the mathematical nature of vector graphics topology, and from which modern vector graphics systems can be derived. In particular, vector graphics complexes, which we introduce in Chapter 4, are largely based on PCS complexes. Those readers who prefer to get a better understanding of the practical applications before diving into the theory, you may safely skip this chapter for now and read it last. However, for those who do not mind a little theoretical detour before getting to the more practical contributions of this thesis, reading this chapter first is likely to make reading the other chapters much more insightful and enjoyable.

In Section 3.1, we shall start by clarifying the meaning of *topology*, which is either combinatorial or geometric depending on the context. In Section 3.2, we show that in the case of vector graphics, reasonable design decisions result in a topology that includes some of the most generic classes of curves and surfaces. Importantly, it includes non-planar surfaces (such as non-orientable surfaces, see Figure 3.1), an unfortunate reality which brings many complications that were largely overlooked by previous work in vector graphics. In particular, the *cut* topological operators are far more complicated for non-planar surfaces than for planar ones. As an example, it is not obvious



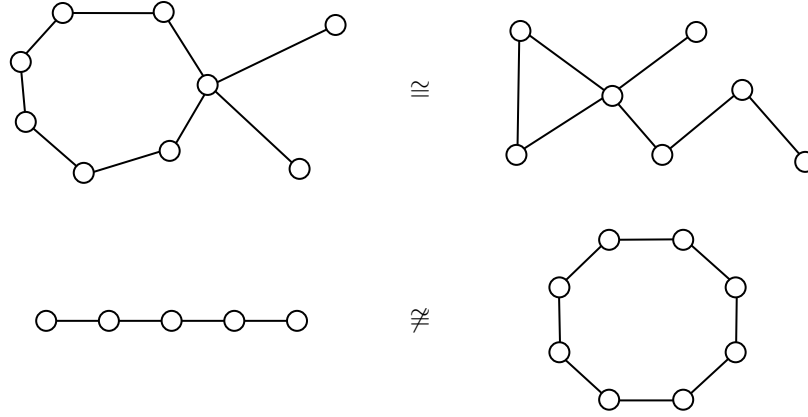
**Figure 3.2:** Whether or not two lines are connected affects: (a) rendering, such as here the rendering of a Miter join; (b) user interaction, such as here a drag-and-drop action of one of the two lines; and (c) keyframe interpolation.

what happens if you cut a Möbius strip. Once the topological nature of vector graphics objects has been determined, we introduce the notion of PCS complex in Section 3.3. It is a topological structure that satisfies our design decisions, and that admits a combinatorial representation from which vector graphics complexes are derived. Finally, we summarize and conclude in Section 3.4.

### 3.1 First Concepts of Topology

*Topology* is a word whose precise meaning depends on the context it is used. This section is about clarifying this terminology: What do we mean by “topology”? What do we mean by: “These two line drawings have the same topology”? Or: “This surface has the topology of a disk”?

At its core, topology is about representing *how objects are connected to one another*. And this is why topology obviously matters for interactive graphics: the behavior of graphical objects depends on whether or not they are connected to one another, and the nature of this connection. For instance, rendering two lines depends on whether or not the two lines are connected (see Figure 3.2a). The result of a *drag-and-drop* user interaction depends on whether or not lines are connected (see Figure 3.2b). And in the context of computer animation, interpolation between two keyframes depends on whether or not lines are connected (see Figure 3.2c).



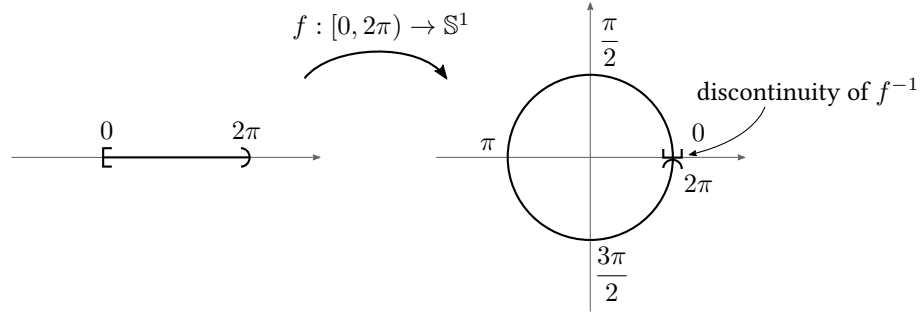
**Figure 3.3:** Top: two homeomorphic graphs. Bottom: two non-homeomorphic graphs.

### 3.1.1 Topology According to Computer Scientists

As computer scientists, we typically work within a discrete world, most often finite. Therefore, we only manipulate a finite number of objects, and connections between these objects can be easily expressed as a **graph**  $\mathcal{G} = (V, E)$ , where every object is represented as a node  $n_i \in V$ , and where a “direct connection” between two objects is represented as an edge  $(n_i, n_j) \in E$ . It should be clear that this graph represents *how objects are connected to one another*, which was our informal definition of topology. This is the reason why, within computer science, the word *topology* often refers to a graph.

For instance, in computer network architecture, the *topology* of a network refers to a graph where each node  $n_i$  represents a computer (either a server, a router, or an end-user device), and each edge  $(n_i, n_j)$  is a pair that indicates that there is a direct communication link between the two computers. This graph contains all the information about how the computers are connected, such as: Are two computers direct neighbors? Are they at all connected to one another, possibly indirectly? If yes, what is the shortest communication path between these two computers? Is the network robust to link failure, i.e., are there at least two paths between any pair of computers? Note that this representation not only stores *whether or not* any pair of computers are connected (=connect-*edness*), but also stores *how* any pair of computers are connected (=topology). For instance, linear and circular networks (see Figure 3.3, bottom) are both fully connected, but they do not have the same topology: the first one becomes disconnected after any given *cut*, while the second one stays connected. In fact, we will see throughout this chapter that the concept of cut, and whether or not it disconnects objects, plays a fundamental role in topology. For instance, cutting a Möbius strip along its centerline does not disconnect it, while cutting a cylinder does.

In most academic contexts where the topology of a model (e.g., a computer network) is expressed



**Figure 3.4:** The function  $f(\theta) = (\cos(\theta), \sin(\theta))$  defined from  $[0, 2\pi)$  to the unit circle  $\mathbb{S}^1$  is continuous and bijective, but its inverse is not continuous. Therefore, it is not a homeomorphism. In fact, it is possible to prove that there exist no homeomorphism from  $[0, 2\pi)$  to  $\mathbb{S}^1$ , and therefore that these two spaces are not homeomorphic, i.e., they “have a different topology”. Source: inspired from [Lee 2011], Figure 2.5.

as a graph, it is said that two models “have the same topology” if and only if the two graphs are **homeomorphic** (see Figure 3.3). By definition, two graphs are said to be homeomorphic if and only if they can be obtained from one another with a finite number of edge subdivisions (splitting one edge into two edges), and edge simplifications (the inverse operation).

However, there are some contexts where “have the same topology” means instead that the two graphs are **isomorphic**, which is much a stronger concept that simply means that the two graphs are identical up to node renaming. We will discuss this in more details in Section 3.1.4.

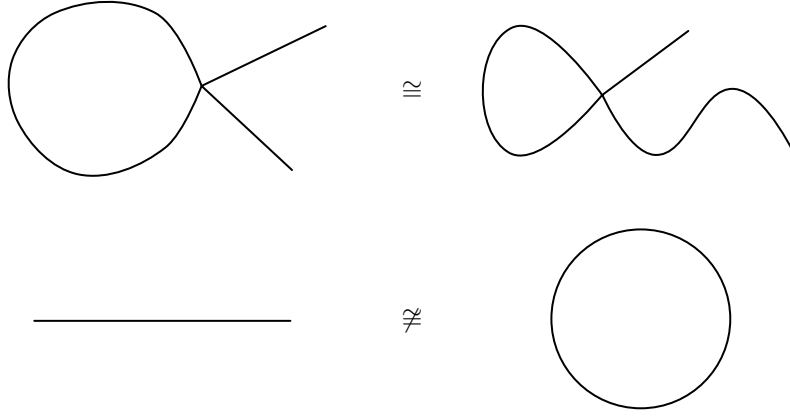
### 3.1.2 Topology According to Mathematicians

On the other hand, most mathematicians who study topology work with continuous geometric entities, such as curves, surfaces, and manifolds of higher dimensions. These geometric entities are made of an uncountable infinite number of *points*, which are “continuously connected” to one another. Because representing this “continuous connectedness” cannot be done using graphs, mathematicians use the concept of *open sets* instead.

More precisely, if  $X$  is a set (for instance,  $X = \mathbb{R}^n$ ), a **topology** on  $X$  is defined as a collection  $\mathcal{T}$  of subsets of  $X$ , satisfying the three following properties:

1.  $X$  and  $\emptyset$  are elements of  $\mathcal{T}$ .
2. Any finite intersection of elements of  $\mathcal{T}$  is also an element of  $\mathcal{T}$ .
3. Any (possibly infinite) union of elements of  $\mathcal{T}$  is also an element of  $\mathcal{T}$ .

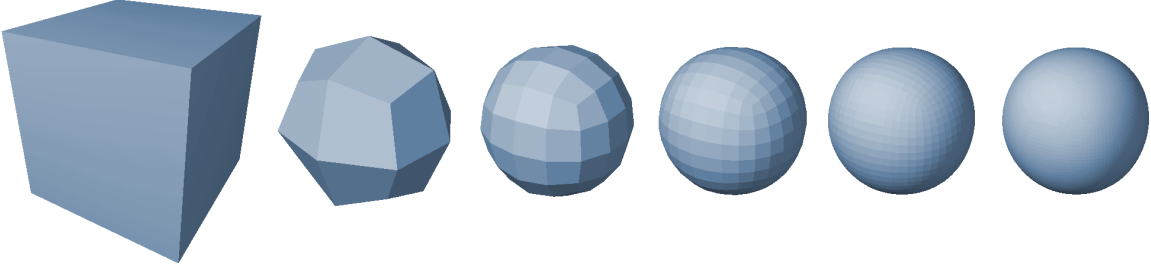
An ordered pair  $(X, \mathcal{T})$  consisting of a set  $X$  together with a topology  $\mathcal{T}$  on  $X$  is called a **topological space**. Elements of  $X$  are usually called **points**, and elements of  $\mathcal{T}$  are called the **open sets** of  $X$ . If the choice of  $\mathcal{T}$  is clear from the context, we simply say that  $X$  is a topological space.



**Figure 3.5:** Top: two homeomorphic point-sets of  $\mathbb{R}^2$ . Bottom: two non-homeomorphic point-sets of  $\mathbb{R}^2$ .

While this definition may seem very abstract to the reader unfamiliar with point-set topology, it is in fact the building block behind all topological concepts such as neighborhood, boundary, and continuous functions. For instance, a function is said to be **continuous** if and only if the pre-image of any open set by this function is also an open set. Such definition is equivalent to the one you may have learned in an analysis class for functions from  $\mathbb{R}^n$  to  $\mathbb{R}^m$  (using limits), but has the advantage that it generalizes well to many other spaces, including non-metric spaces (where the concept of “distance” does not exist). Readers interested to learn more on this fascinating topic are encouraged to get their hands on any introductory book in general or algebraic topology (e.g., [Munkres 2000, Hatcher 2001, Lee 2011]). However, our aim in this section is not that the reader fully understands these concepts, but only to give a sense of what *topology* and *topological space* means in a continuous world. In one sentence, a topological space  $(X, \mathcal{T})$  is a set  $X$  of points together with a collection  $\mathcal{T}$  of subsets of  $X$  called open sets. It is quite different from the concept of a graph  $(V, E)$ , which is a set  $V$  of nodes together with a set  $E$  of pairs of nodes called edges. However, notice the similarity: in both cases, it is a set  $X$  or  $V$  of “objects”, together with the information  $\mathcal{T}$  or  $E$  of *how they are connected to one another*. In Section 3.3, we will see the same similarity between PCS complexes and abstract PCS complexes.

We can now define one of the most important concepts of this chapter: a **homeomorphism** between two topological spaces is a bijective function which is continuous, and whose inverse is also continuous (continuity of the inverse is important, see Figure 3.4 for a classical counter-example). Two spaces are said to be **homeomorphic** if and only if there exists a homeomorphism between them (see Figure 3.5). Homeomorphisms are extremely important because, as a direct consequence of their definition, they happen to preserve open sets (i.e., the image or the pre-image of an open set by a homeomorphism is also an open set). Since topology is defined via open sets, this means that all topological properties are preserved by homeomorphisms. For this reason, it is common to say that two spaces “have the same topology” if and only if the two spaces are homeomorphic.



**Figure 3.6:** Many smooth surfaces made of an uncountable infinite number of points can be finitely represented using computers. Here, we illustrate how a smooth surface (i.e., a geometric object) can be defined as the limit of Catmull-Clark subdivision steps, starting from a polygonal mesh (i.e., a combinatorial object).

### 3.1.3 Topology According to Computational Geometers

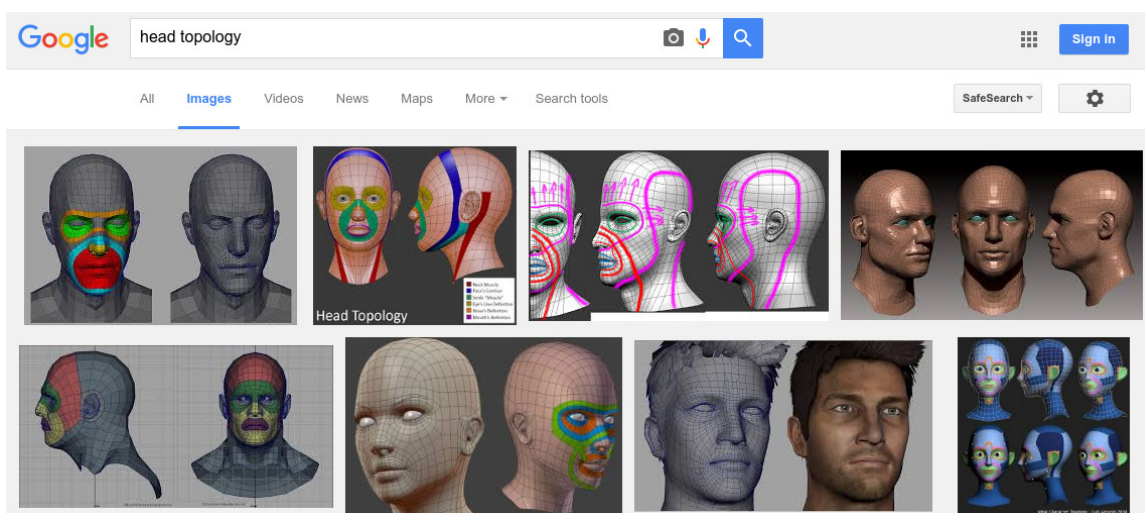
In the previous two sections, we have described topology as having either a combinatorial nature (a graph, typically used in computer science), or a geometric nature (a topological space, typically used in mathematics). However, there are many fields of research at the boundary between these two interpretations, such as *computational geometry*.

In fact, very fundamentally, the core idea of algebraic topology is to study the combinatorial properties of geometric objects, and an important part of graph theory focuses on embedding graphs in geometric spaces. For instance, planar graphs are defined as the graphs that can be embedded in  $\mathbb{R}^2$ , that is, whose vertices can be embedded as distinct points, and whose edges can be embedded as curves connecting these endpoints that are interior disjoint. Within algebraic topology, such interpretations of combinatorial objects as topological spaces are usually called **geometric realizations**. For instance, every graph can be realized as a specific kind of topological space called a (one-dimensional) simplicial complex. As you can guess by comparing Figure 3.3 with Figure 3.5, the concept of homeomorphism between graphs is equivalent to the concept of homeomorphism between topological spaces, once graphs are realized as topological spaces.

But let us go back to computational geometry, and more specifically to its subfield *geometric modeling*, to which this thesis belongs. Within geometric modeling, a goal is to design computer representations of various geometric objects, typically curves and surfaces. Given that computers have a finite memory, this means designing finite representations of these geometric objects. For instance, a 2D line segment between two points  $P$  and  $Q$  can be mathematically represented as:

$$\{ (1 - \lambda)P + \lambda Q \mid \lambda \in [0, 1] \} \quad (3.1)$$

where  $P \in \mathbb{R}^2$  and  $Q \in \mathbb{R}^2$ . Even though this line segment is made of an infinite number of points, all we need to *represent* all of these points is to store  $P$  and  $Q$ , which only uses a finite amount of memory (as long as  $P$  and  $Q$  can themselves be represented with a finite amount of memory, for



**Figure 3.7:** Google image search results for “head topology”. © 2016 Google Inc. Google and the Google logo are registered trademarks of Google Inc., used with permission.

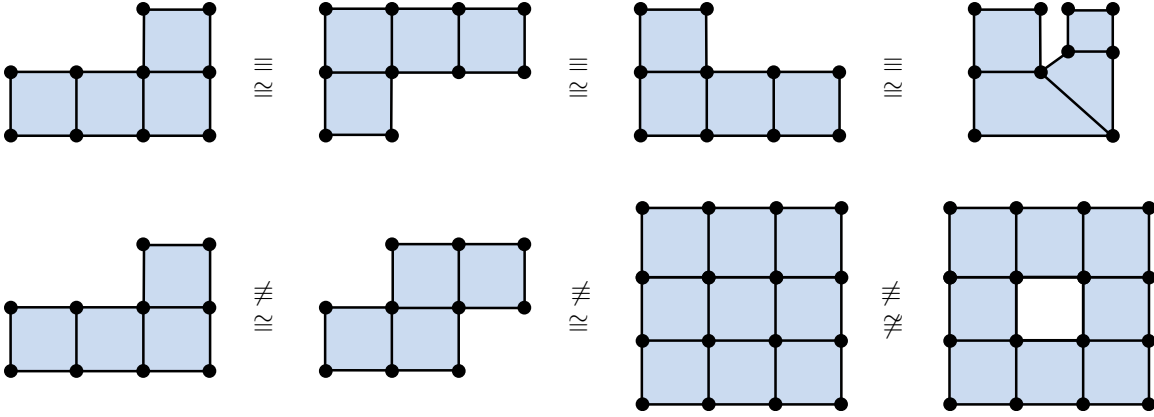
instance using floating point arithmetic). In order to represent a piecewise linear curve, one can use a list of points. In order to represent a smooth curve, one can use a list of points and tangent vectors (e.g., a Hermite or Bézier curve), or simply a list of points (e.g., a Catmull-Rom or Chaikin subdivision curve). In order to represent a piecewise linear surface, one can use a triangle mesh. In order to represent a smooth surface, one can use a NURBS surface, or a Catmull-Clark subdivision surface (see Figure 3.6).

Since all these combinatorial objects are in fact *representing* continuous point-sets, it is possible to apply to them the mathematician’s definition of topology. For instance, to compute a mesh parameterization, it is often said that the mesh must be cut until “it has the topology of a disk”. This means that after the cuts, the represented surface must be homeomorphic to the point-set  $\{ |x| \leq 1 \mid x \in \mathbb{R}^2 \}$ . Being able to apply the mathematician’s definition of topology to combinatorial objects is essential to study many of their properties, and is the reason why we will introduce the concept of PCS complexes in Section 3.3. PCS complexes provide a well-defined geometric interpretation of vector graphics combinatorial objects, and in particular, they allow us to precisely define what homeomorphism means for these objects.

#### 3.1.4 Topology According to 3D Modeling Artists

However, there are many contexts within geometric modeling, and especially among 3D modeling artists, where the word “topology” does not refer to the mathematician’s geometric definition, and where “having the same topology” does not mean “being homeomorphic”. For example, if you ask a 3D modeling artist whether the two leftmost objects in Figure 3.6 have the same topology,



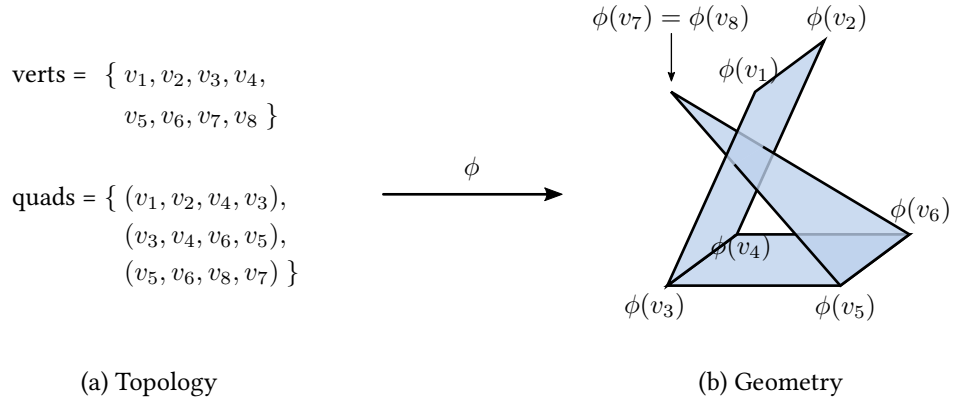


**Figure 3.8:** Top: Examples of isomorphic quad meshes. Bottom: Examples of non-isomorphic quad meshes. Note that isomorphic ( $\equiv$ ) implies homeomorphic ( $\cong$ ), but the opposite is not true. All these quad meshes are pairwise homeomorphic, apart from the bottom-right example which contains a hole.

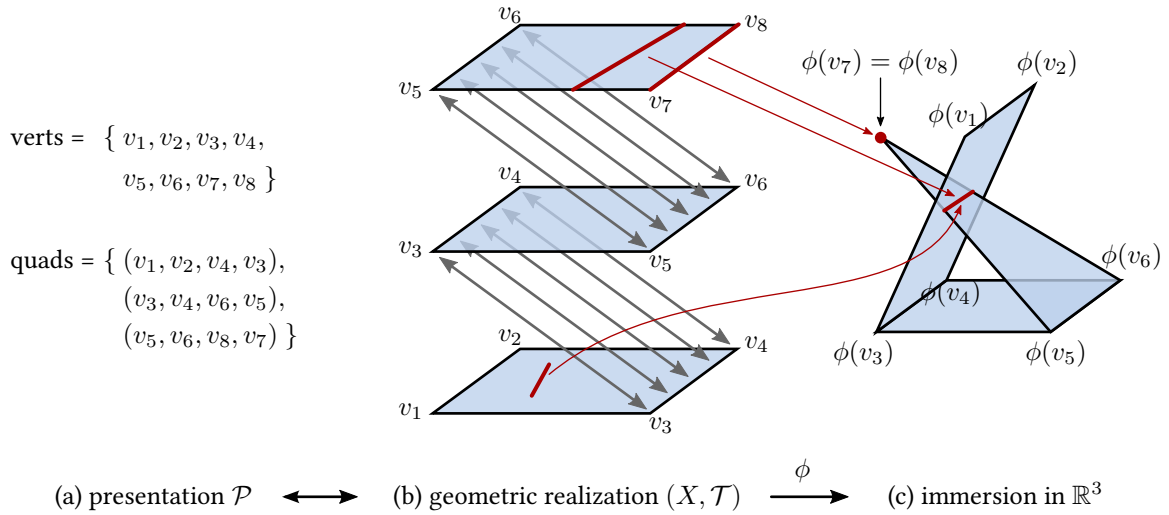
their answer is likely to be: “no, for instance the first one has only 6 quads, while the second one has 24 of them”. But if you ask a mathematician: “yes, they both have the topology of a sphere”. When artists use the word “topology”, they usually refer to how many polygons are used, and how they are arranged into coherent and esthetically pleasing *loops*, typically following muscle lines, which produce the best smoothing and deformation results (see Figure 3.7). They would say that the first two head models in the first row of Figure 3.7 have different topologies because the first model has a loop of quads enclosing both eyes (in orange), while the second model has no such loop. Formally, what they mean is that two meshes have the same topology if and only if they are **isomorphic**. A precise definition of **isomorphism** for polygonal meshes (or other combinatorial structures, such as graphs, multigraphs, triangle meshes, simplicial complexes, etc.) depends on how exactly the combinatorial structure is defined, but essentially it means that they have the same number of vertices, edges, and faces, and that they are arranged in the same way. Equivalently, one can say that isomorphism means that the meshes are equal up to vertex/edge/face renaming and geometric deformation (see Figure 3.8). In section 3.3.2, we precisely define what isomorphism means for PCS complexes, and since most mesh data structures are a subset of PCS complexes, the definition apply to these other data structures as well.

This meaning of topology, which emphasises more on isomorphism than homeomorphism, comes from the fact that in most mesh data structures, it is possible to decorrelate the combinatorial information from the geometric information of the represented surface. Therefore, the data structure can be implemented as a combinatorial structure called the *topology* of the mesh, on top of which can be added geometric attributes, called the *geometry* of the mesh (see Figure 3.9). As a consequence, any edit to the combinatorial structure (e.g., an edge-collapse) is often called a **topological operator** and said to “change the topology” of the mesh, even if the new mesh is still homeomor-





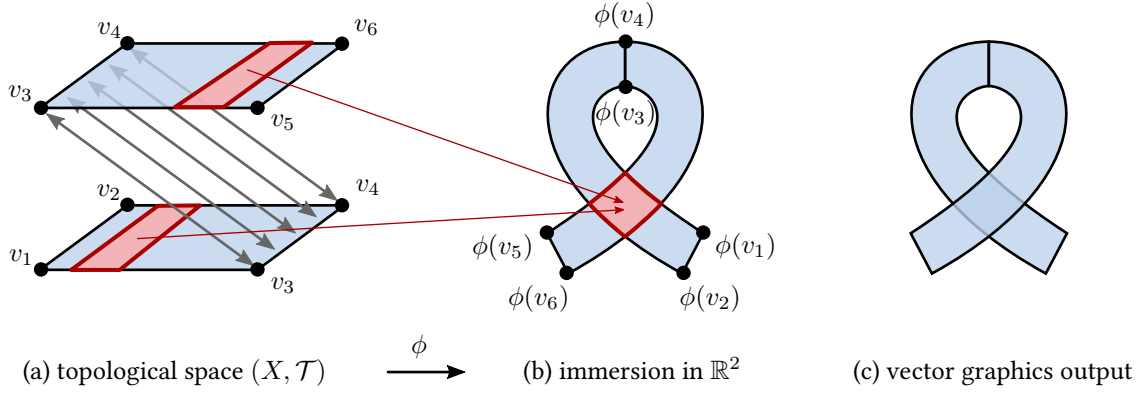
**Figure 3.9:** Most mesh data structures, such as here a simple quad mesh structure, are usually implemented as: (a) a combinatorial structure called “topology”; and (b) geometric attributes called “geometry”. Typically, a 3D position  $\phi(v_i)$  is assigned to each vertex  $v_i$ , which defines an immersion of the topology into  $\mathbb{R}^3$ .



**Figure 3.10:** (a) In algebraic topology, any combinatorial structure  $\mathcal{P}$  describing topology is usually called a presentation. (b) Each presentation  $\mathcal{P}$  can be interpreted as a continuous topological space  $(X, \mathcal{T})$  called its geometric realization. (c) Assigning a 3D position  $\phi(v_i) \in \mathbb{R}^3$  to each vertex  $v_i \in \mathcal{P}$  can be interpreted as defining a continuous function  $\phi : X \mapsto \mathbb{R}^3$ , generally non-injective. We have highlighted in red every point  $x \in \mathbb{R}^3$  whose pre-image by  $\phi$  contains two or more points in the geometric realization. This formalism rigorously captures the intuitive idea that there are several “overlapping points” at such 3D location  $x$ .

phic to the old one. This meaning of topology is the one we use when we say, for instance, “vector graphics animation with time-varying topology”. We mean that there exist at least two frames in the animation whose respective vector graphics complexes are not isomorphic. Whether they are homeomorphic is irrelevant.

While this definition of topology is combinatorial in nature, it is essential to understand that it actually *represents* a continuous geometric object, even before geometric attributes are specified. More precisely, this combinatorial structure called “topology” is usually called a **presentation**  $\mathcal{P}$  by algebraic topologists (see Figure 3.10a), which can be interpreted as a continuous topological space  $(X, \mathcal{T})$  called its **geometric realization** (see Figure 3.10b). Generally, this geometric realization is defined via the formalism of **quotient spaces**, which consists in abstractly gluing together duplicated pieces of  $\mathbb{R}^n$  (see Section A.6 for details). After this process, the topology  $\mathcal{T}$  of the geometric realization accurately represents the intended semantics of the combinatorial structure, i.e. how the points of the underlying geometric object  $X$  are continuously connected to one another. Once a presentation  $\mathcal{P}$  is interpreted as a continuous topological space  $(X, \mathcal{T})$ , then assigning a 3D position  $\phi(v_i) \in \mathbb{R}^3$  to each vertex  $v_i \in \mathcal{P}$  can be interpreted as defining an **immersion** of this topological space into  $\mathbb{R}^3$ , i.e. a continuous function  $\phi : X \mapsto \mathbb{R}^3$ , possibly not injective (see Figure 3.10c). This means that two different points of the geometric realization may be mapped to the same point in  $\mathbb{R}^3$ , creating what we call **overlapping points**: two or more points that share the same spatial location, but have their own “topological identity”. In the next section, we will see that understanding this concept is critical for understanding vector graphics topology, since overlapping plays a very important role in vector graphics.



**Figure 3.11:** A vector graphics illustration represented as: (a) a continuous topological space  $(X, \mathcal{T})$ , such as here a quotient space, decomposed into vertices, edges, and faces; and (b) a continuous function  $\phi$  that immerses the topological space into the canvas  $\mathbb{R}^2$ . We have highlighted in red every point  $x \in \mathbb{R}^2$  whose pre-image by  $\phi$  contains two or more points in  $X$ . (c) The final vector graphics illustration.

## 3.2 The Non-Planar Nature of Vector Graphics

In this section, we show that assuming a few reasonable design decisions, any topological structure that *represents* vector graphics combinatorial objects must support non-manifold topologies and surfaces of arbitrary orientability and genus (the definitions of manifoldness, orientability, and genus are recalled in Appendix A). In particular, this includes non-planar objects, such as non-orientable surfaces (e.g., a Möbius strip), or orientable surfaces of non-zero genus (e.g., a torus). We will also cover a few other topological properties, but we want to emphasize non-planarity because it is the least studied by previous research, the one that brings most complications, and also perhaps the least intuitive.

By **topological structure**, we mean a topological space  $(X, \mathcal{T})$  that is partitioned into subsets called **cells**, such as vertices, edges, and faces. The continuous point-set  $X$  might be defined as either a quotient space (see Figure 3.11a), or a subset of  $\mathbb{R}^n$  in which the cells do not intersect (this may require  $n \geq 4$ , e.g. for a Klein bottle). It is then immersed into the canvas  $\mathbb{R}^2$  via a continuous function  $\phi : X \mapsto \mathbb{R}^2$ , possibly non-injective, which defines the final vector graphics illustration (see Figure 3.11b and 3.11c). We call **overlapping** any point of  $\mathbb{R}^2$  whose pre-image by the immersion  $\phi$  contains two or more points of the topological space.

We can observe that this concept of topological structure *immersed* in  $\mathbb{R}^2$  is more generic than the concept of planar maps [Baudelaire and Gangnet 1989], which is directly a partition of  $\mathbb{R}^2$  into vertices, edges, and faces, i.e. a topological structure *embedded* in  $\mathbb{R}^2$ .

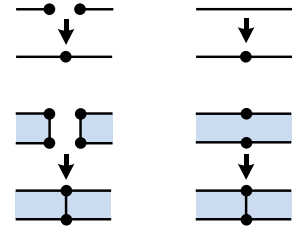
### 3.2.1 Design Decisions

Let us start with a short list of design decisions, or desiderata, that we believe should ideally be supported by any vector graphics system, thus by our topological structure.

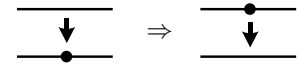
**A. Basic Primitives** At the very least, the following cells must be supported: vertices (single points in space); open edges (open curves starting and ending at a vertex); and triangles (surfaces homeomorphic to disks, bounded by three edges).



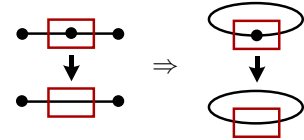
**B. Basic Topological Operators** Any two vertices can be glued. Any two edges can be glued using any of the two possible directions. Any edge can be cut by inserting an additional vertex. Any face can be cut by inserting an additional open edge starting and ending at existing boundary vertices.



**C. Operator Invertibility** The inverse of any valid operator is also a valid operator.

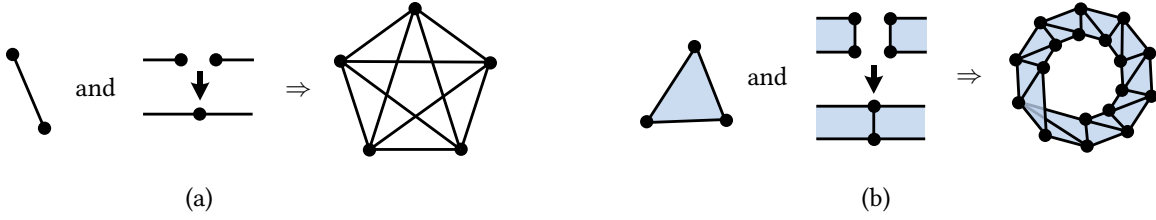


**D. Operator Locality** The validity of any topological operator (i.e., whether or not it is allowed to apply it) only depends on local topological properties.



These design decisions adhere to two important user interface design principles: learnability (which includes familiarity, predictability, and consistency) and flexibility. Basic Primitives (A) are obvious for familiarity: vertices, open edges and triangles are already standard in all existing topological modeling tools. Among Basic Topological Operators (B), gluing two vertices and cutting edges are also standard in most vector graphics tools, except that not “any” two vertices can be glued. Our addition of the word “any” is key for predictability and consistency, and is what allows to represent three or more edges sharing a common vertex. If instead, users are sometimes allowed to glue two vertices or two edges, but sometimes not (for technical reasons not obvious to them), then not only does it impede predictability and consistency, but it also decreases flexibility. In other words, it limits their artistic freedom, which is likely to frustrate them. Unfortunately, this is the case in most existing vector graphics tools.

Gluing two edges or cutting a face is not standard in existing vector graphics tools, since most of them do not even support shared edges in the first place. However, not only these are two operations extremely useful, but they are the direct counterpart of gluing two vertices or cutting an edge. Therefore, they increase both flexibility and consistency, and it makes sense to include them. Operator Invertibility (C) is very important for familiarity and predictability: if an operation



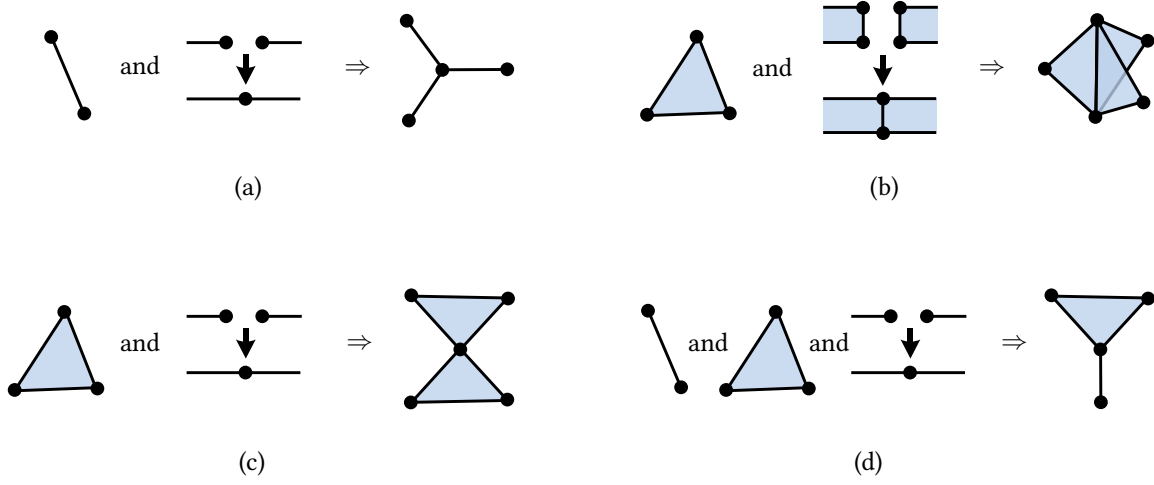
**Figure 3.12:** (a) Proof of non-planarity: vector graphics topologies are a superset of graphs, some of which are not planar. (b) Proof of non-orientability: vector graphics topologies are a superset of triangle meshes, some of which are non-orientable, such as here a Möbius strip. We note that (b)  $\Rightarrow$  (a), but proving (a) independently shows that even without faces, the topology can be non-planar.

can be done, surely it can be undone. This is a basic expectation that users have learned from all other applications, and not just graphics applications. Operator Locality (D) is more subtle but also very important for predictability, and is correlated with the usage of “any” in our design decisions. The idea is that *global* topological properties, such as planarity or orientability, are hard to understand and assess at a glance. Therefore, not allowing a topological operation due to global topological constraints is likely to be an unexpected behavior for most users. However, *local* topological properties are more intuitive and directly visible. Therefore, it is acceptable to forbid users from doing some topological operations due to local topological constraints, since they can intuitively understand why the operation is not possible. For instance, it should be obvious to most users that “uncut” cannot be applied at a vertex with three incident edges, since such topology could not have possibly been achieved via “cut”.

### 3.2.2 Non-Planarity and Overlapping

Let us now start to analyze the consequences of the design decisions above. The first and foremost consequence is that non-planar topologies must be supported, as illustrated in Figure 3.12. Indeed, allowing for the representation of open edges and allowing for any two vertices to be glued means that the topological structure at hand is a superset of graphs. Since some graphs are non-planar, such as  $K_5$  (see Figure 3.12a), then some vector graphics topologies are non-planar.

Since there exists no injective immersion in  $\mathbb{R}^2$  of such non-planar topologies, this means that edges and faces *must* be allowed to overlap. Of course, allowing edges and faces to overlap is an extremely desirable feature in itself. Intuitively, it should be seen as a design decision, but since it turns out to be a consequence of other design decisions, then we present it as a consequence, in order to keep the list of design decisions minimal and non-redundant. It should be understood that if it was not a consequence, then we would have *added* it as a design decision.



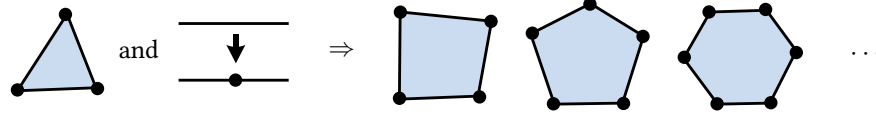
**Figure 3.13:** Examples of non-manifold topologies, together with the corresponding design decisions which allow these topologies. In fact, gluing objects together, which is formalized via quotient spaces, is one of the fundamental tools used in algebraic topology to represent non-manifold topological spaces.

### 3.2.3 Non-Orientability

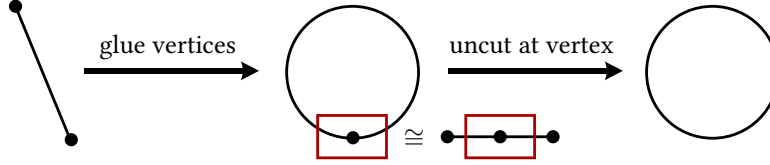
Similarly, it is straightforward to prove that given the design decisions of Section 3.2.1, then some vector graphics topologies are non-orientable (that is, they contain a non-orientable surface, see Appendix A.4). Indeed, allowing for triangles to be represented and allowing to glue any two edges in any of the two possible directions means that the topological structure at hand is a superset of triangle meshes. Since some triangle meshes are non-orientable, such as a triangle mesh representing a Möbius strip (see Figure 3.12b), then some vector graphics topologies are non-orientable.

### 3.2.4 Non-Manifoldness

Finally, as a superset of graphs and triangle meshes, an obvious but important consequence is that some vector graphics topologies are non-manifold (see Figure 3.13). More precisely, vector graphics topologies are a superset of two-dimensional simplicial complexes, which have been for decades the long-established standard representation for non-manifold spaces. Nowadays, many algebraic topologists tend to prefer CW complexes for their flexibility, but in the field of computational topology, simplicial complexes are still the *de facto* representation for non-manifold spaces. The reason is that unfortunately, CW complexes do not admit a purely combinatorial representation, and therefore are not suitable for computation. In a sense, vector graphics topologies can be seen as a balance between simplicial complexes and CW complexes: they are more generic than simplicial complexes in order to be artist-friendly, but they are less generic than CW complexes in order to be computer-friendly.



**Figure 3.14:** As a consequence of allowing triangles and allowing any edge to be cut, vector graphics topologies must support  $n$ -sided faces.



**Figure 3.15:** As a consequence of allowing open edges, of allowing any two vertices to be glued, of allowing edges to be cut, of Operator Invertibility (C), and of Operator Locality (D), vector graphics topologies must support closed edges.

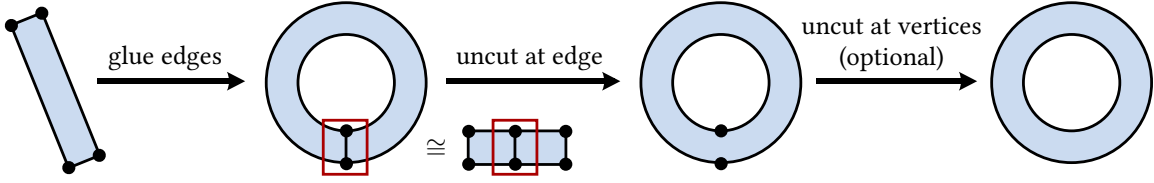
### 3.2.5 N-Sided Faces

So far, we have only analyzed the consequences of using open edges, triangles, and the glue operators. More specifically, we have seen that they allow for non-manifold and non-planar topologies, possibly non-orientable. From now on, let us focus on the consequences of the cut operators. The most obvious consequence of allowing to cut edges is that the topological structure must support  $n$ -sided faces (i.e., faces bounded by  $n$  edges, for arbitrary  $n$ ). Indeed, when iteratively applying a cut to the boundary of a triangle, then we obtain a quad, then a five-sided face, and so on, as illustrated in Figure 3.14.

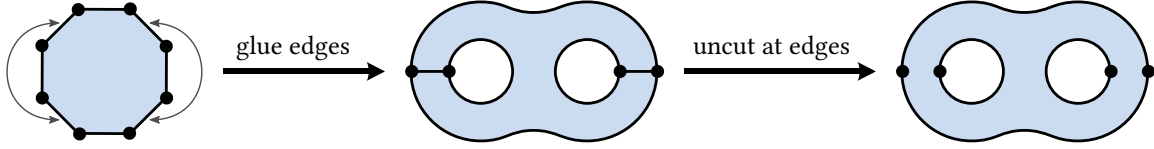
### 3.2.6 Closed Edges

However, the most significant consequences of the cut operators happen when they are combined with Operator Invertibility (C) and Operator Locality (D). Indeed, this turns the inverse of the cut operators into powerful *simplification operators*, extending significantly what topologies can be expressed *with a single cell*. For instance, a **closed edge** can be created in two simple steps, illustrated in Figure 3.15. First, one can create an edge that start and end with the same vertex, by gluing the two end vertices of an open edge. At the remaining vertex, one can observe that the topological structure is *locally homeomorphic* to an open edge that has been cut (see Figure 3.15, middle). Therefore, as a consequence of Operator Invertibility (C) and Operator Locality (D), “uncutting” at this vertex must be allowed, which results in a closed edge.

Such a closed edge can in fact be seen as a new **type** (= homeomorphism class) of cell! Indeed, it is not homeomorphic to either a vertex, an open edge, or a face. This makes vector graphics



**Figure 3.16:** As a consequence of allowing  $n$ -sided faces, of allowing any two edges to be glued, of allowing faces to be cut, of Operator Invertibility (C), and of Operator Locality (D), vector graphics topologies must support faces with inner holes. Note the similarity between this figure and Figure 3.15.



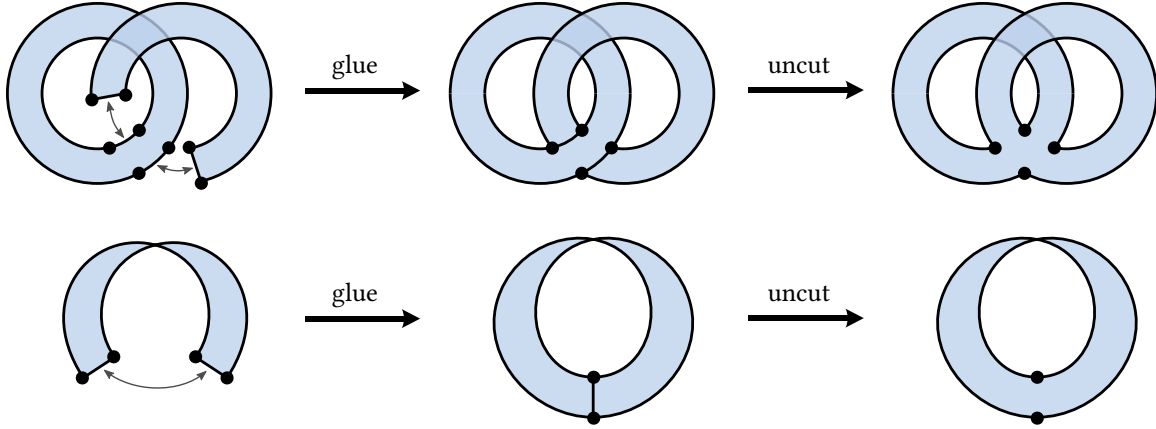
**Figure 3.17:** For any given  $n \in \mathbb{N}$ , it is possible to create a disk with  $n$  inner holes from a  $4n$ -sided disk, by gluing every  $4i$ -th edge with the  $(4i + 2)$ -th edge, then uncutting all resulting edges.

topologies unique compared to most existing topological structures. For instance, neither simplicial complexes or CW complexes allow closed edges! The reason is that in algebraic topology, cells are designed to be “simple enough” to be a useful tool for mathematical proofs. Indeed, the simpler the cells, the stronger their topological properties. For instance, in the case of CW complexes, all cells are homeomorphic to the interior of  $\mathbb{D}^n = \{ x \in \mathbb{R}^n \mid ||x|| \leq 1 \}$ , which is a very useful property that can be used in proofs. But in our case, we are not very concerned about its utility as a proof mechanism, but rather we are concerned about user experience. We want a definition of cells that provides the best possible user experience. In a sense, defining what qualifies as a *valid cell* and what does not can be seen as a delicate balance between expressiveness and simplicity, and because artists have very different needs than mathematicians, our definition of cell leans more towards expressiveness than simplicity. Unfortunately, this more complex definition results in topological operators which are harder to implement, but this is a reasonable price to pay to improve user experience. In other words, there is no such thing as a free lunch.

### 3.2.7 Faces with Inner Holes

Following the exact same reasoning, vector graphics topologies must also support faces with inner holes. Indeed, a face with an inner hole can be easily created in two steps: first gluing the two opposite edges of a quad, then uncutting the resulting edge (see Figure 3.16). More generally, given any  $n \in \mathbb{N}$ , it is possible to create a face with  $n$  inner holes from a  $4n$ -sided face (see Figure 3.17). Like with closed edges, this in fact creates new *types* of faces, since a face with  $i$  holes is never homeomorphic with a face with  $j$  holes (for  $i \neq j$ ). Like closed edges, none of these new types of cells are valid in simplicial complexes and CW complexes, but it is clear that they must be





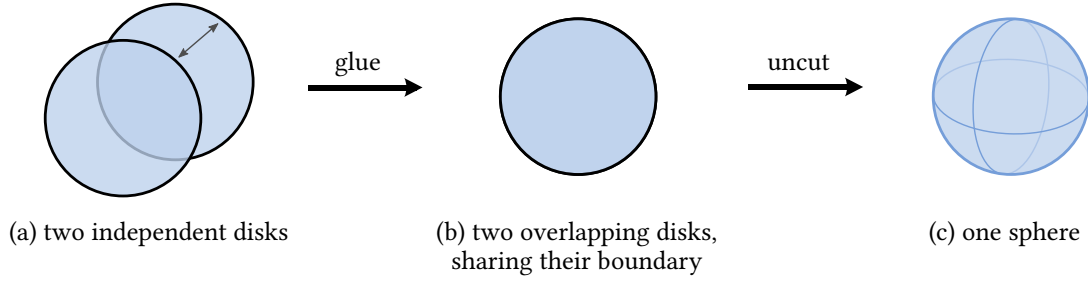
**Figure 3.18:** Examples of non-planar faces. Top: A face homeomorphic to a torus with one hole. Bottom: A face homeomorphic to a Möbius strip. Like a disk, the boundary of both these faces is made of a single curve.

supported by vector graphics topologies. Indeed, they are obviously useful for artists, and are in fact already supported by both SVG and planar maps.

### 3.2.8 Non-Planar Faces

Using the same two steps (glue + uncut), it is also possible to create **non-planar faces**, i.e. cells which are homeomorphic to non-planar surfaces (see Figure 3.18). Indeed, we had already seen in Section 3.2.2 that due to the glue operators, non-planar topologies must be supported. But in fact, by applying the inverse of the cut operators to these topologies, even *a single cell* could be a non-planar point-set. For instance, a cell can be a point-set homeomorphic to a Möbius strip, or homeomorphic to a torus with a hole, or more generally homeomorphic to any orientable or non-orientable surface of any genus, with any number of holes. Like closed edges, such faces are usually not allowed in topological structures in order to ensure that cells are as “simple” as possible, which is very desirable for many use cases, such as mathematical proofs. But in our case, we allow these non-planar faces as a consequence of our design decisions. They provide additional flexibility, consistency, and expressiveness which have practical benefits for artists, and which we believe is worth the extra complexity of supporting them.

With all these different types of faces, the word “face” is becoming more and more ambiguous. Is it orientable? What is its genus? Does it have holes? How many? The same way that we have a different name for *closed edges* and *open edges*, it would be nice to have a different name for each of these types of faces. Unfortunately, while there exist only two types of edges (open and closed), there exist infinitely many types of faces, which makes naming them inconvenient. However, it turns out that the type of a face is fully determined by three attributes: its orientability  $\epsilon \in \{\circ, \emptyset\}$  (= orientable or not), its genus  $g \in \mathbb{N}$ , and its number of holes  $k \in \mathbb{N}$ . This is a consequence



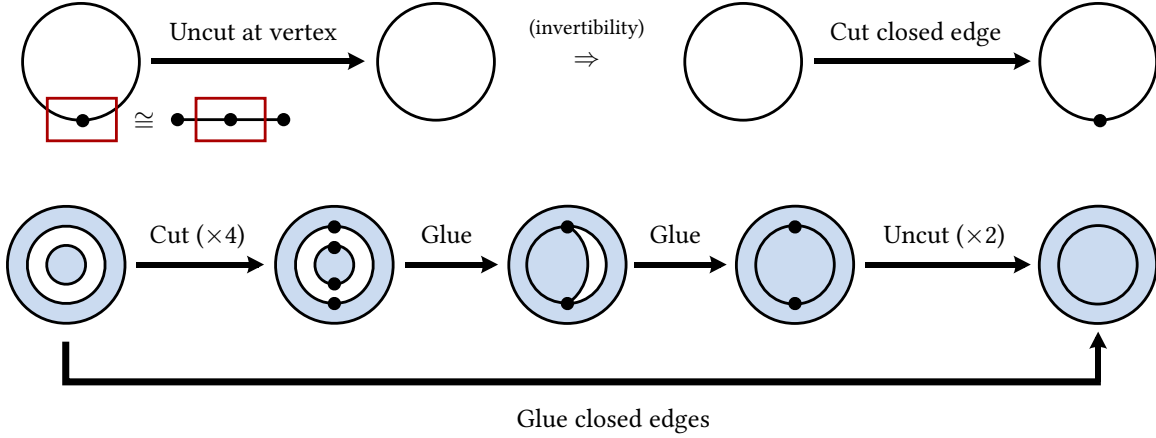
**Figure 3.19:** A sequence of topological operators may result in a face without boundary, such as here a sphere. Since these topologies cannot be rendered via “winding rule filling”, most implementations may want to detect them and handle them as a special case.

of an important result of algebraic topology called *the classification of compact 2-manifolds* (see Section A.4). Therefore, we can refer to each type of face as  $\epsilon$ - $g$ - $k$ -face. For instance, a face is planar (i.e., can be embedded in  $\mathbb{R}^2$ ) if and only if it is an  $\odot$ -0- $k$ -face, for some  $k > 0$ . In other words, planar faces are those homeomorphic to a sphere with  $k > 0$  holes. All other types of faces are non-planar, including the case  $k = 0$  which we discuss in the next section.

Note that in Section 3.2.7, we used the phrase “face with  $n$  inner holes” to mean “cell homeomorphic to a disk with  $n$  holes”. Indeed, it is the intuitive meaning of “hole” in vector graphics, where a 2D shape “without hole” intuitively means a topological disk. However, this intuitive meaning of “hole” is *different* from the number  $k$  above, which includes not only inner holes but also the outer boundary. The reason is that from a topological perspective, they are no ways to distinguish the outer boundary from an inner hole: both are one connected component of the face’s boundary. Topologically, a “disk with one hole” is nothing else than a (finite) cylinder, which is nothing else than a sphere with 2 holes. To summarize, what we called a “face with  $n$  inner holes” was in fact an  $\odot$ -0- $(n + 1)$ -face: an orientable, genus-0 surface with  $k = n + 1$  holes. In this thesis, whenever we use the term “hole”, we refer to the number  $k$ . If we need to refer to the more intuitive number  $n = k - 1$ , we use the term “inner hole” instead.

### 3.2.9 Faces without Boundary

In the previous section, we have seen that each face is of type  $\epsilon$ - $g$ - $k$ -face, for some  $\epsilon$ ,  $g$ , and  $k$ . In particular, we have seen that the number  $k$  represents the number of holes of the face, which means the number of connected components of the face’s boundary. The case  $k = 0$  is very special and represents surfaces without boundary (e.g., a sphere, a torus, or a Klein bottle), which are never planar. Because vector graphics faces are typically rendered using a winding rule to “fill” in  $\mathbb{R}^2$  the immersion of their boundary, faces without boundary cannot be rendered and thus have very limited use. However, we do allow them for various reasons, including the fact that they are the

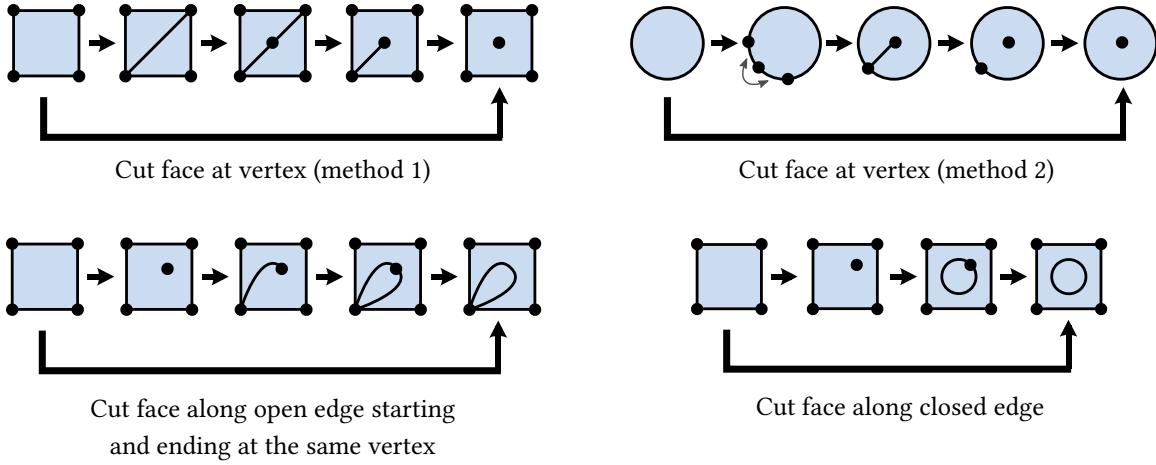


**Figure 3.20:** Top: Cutting a closed edge into an open edge can be defined as the inverse of uncutting an open edge starting and ending at the same vertex. Bottom: Gluing together two closed edges can be defined as a sequence of existing operators.

mathematically correct result of some topological operators which should be valid according to our design decisions (see Figure 3.19). However, in practice, some implementations may want to forbid them, warn the user when they occur, or make it a user preference. However, it is important to keep in mind that even if they cannot be rendered, they can be extremely useful as temporary objects, either in the middle of an algorithm or during the user’s editing process. In addition, some advanced users, for instance math students studying topology, may find them useful as a pedagogical tool or as an abstract representation not meant to be rendered. Also, it is possible to imagine alternative rendering methods not based on winding numbers, which would make them renderable (for instance, by immersing their geometry using a triangulation). Finally, we note that they are necessary for uniqueness of a minimal cell decomposition (see Section E), which may be useful to determine whether two vector graphics illustrations are homeomorphic.

### 3.2.10 Cut and Glue Closed Edges

Now that we have shown the existence of all these new types of cells (closed edges, faces with inner holes, and non-planar faces), we can define new topological operators involving these cells. They can all be defined using the existing operators, Operator Invertibility (C), and Operator Locality (D). For instance, we illustrate in Figure 3.20 (top) that cutting a closed edge into an open edge can be defined as the inverse of uncutting an open edge into a closed edge. In Figure 3.20 (bottom), we show that gluing two closed edges together can be defined as a sequence of cutting the closed edges into open edges, then gluing the open edges together, and finally uncutting back the open edges into closed edges.



**Figure 3.21:** Using a sequence of existing operators, *Operator Invertibility (C)*, and *Operator Locality (D)*, it is possible to define cutting a face at a (Steiner) vertex, cutting a face along an open edge that starts and ending at the same vertex, and cutting a face along a closed edge.

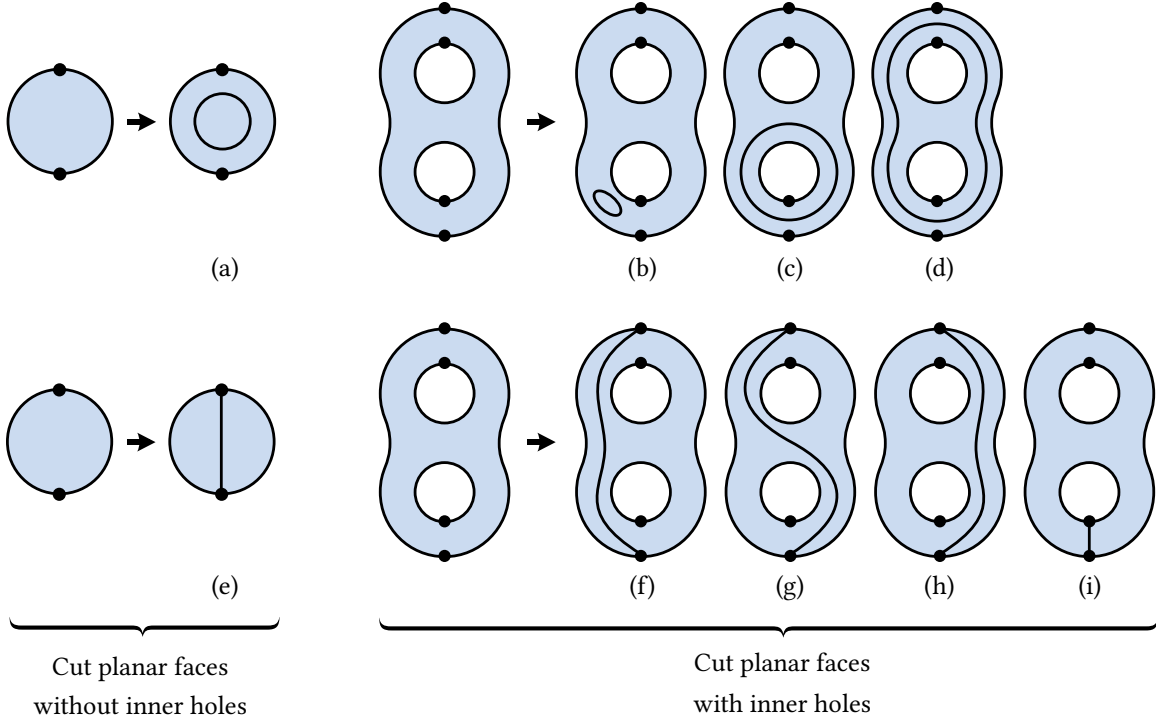
### 3.2.11 Cut Faces at Vertices and along Closed Edges

In the design decisions, we mentioned cutting faces along open edges. In Figure 3.21, we show that it is also possible to cut them at vertices, and cut them along closed edges. In addition, we show that they can be cut along open edges starting and ending at the same vertex, a special case of cutting along an open edge, which was not explicitly mentioned in the design decisions. All of these new cut operators can be defined as a sequence of operators already defined.

Cutting a face at a vertex results in what we call a **Steiner vertex**, also called “point-in-face” in some contexts. It is a point in the interior of a face that has been taken out to become its own separate cell. However, we note that a Steiner vertex is not a new *type* of vertex, and its surrounding face is not a new type of face either. Indeed, a Steiner vertex is still homeomorphic to a “normal” vertex, and the surrounding face is still homeomorphic to one of the types of face we have already seen: each Steiner vertex counts as a hole (for instance a disk with a Steiner vertex is a  $\odot$ -0-2-face). Therefore, the term “Steiner vertex” does not refer to a new type of vertex, but instead refers to the specific way in which the topological space  $X$  is locally decomposed into existing types of cells. More specifically, any vertex that has one or more incident face(s) but no incident edges is called a Steiner vertex.

### 3.2.12 Cut Faces with Inner Holes

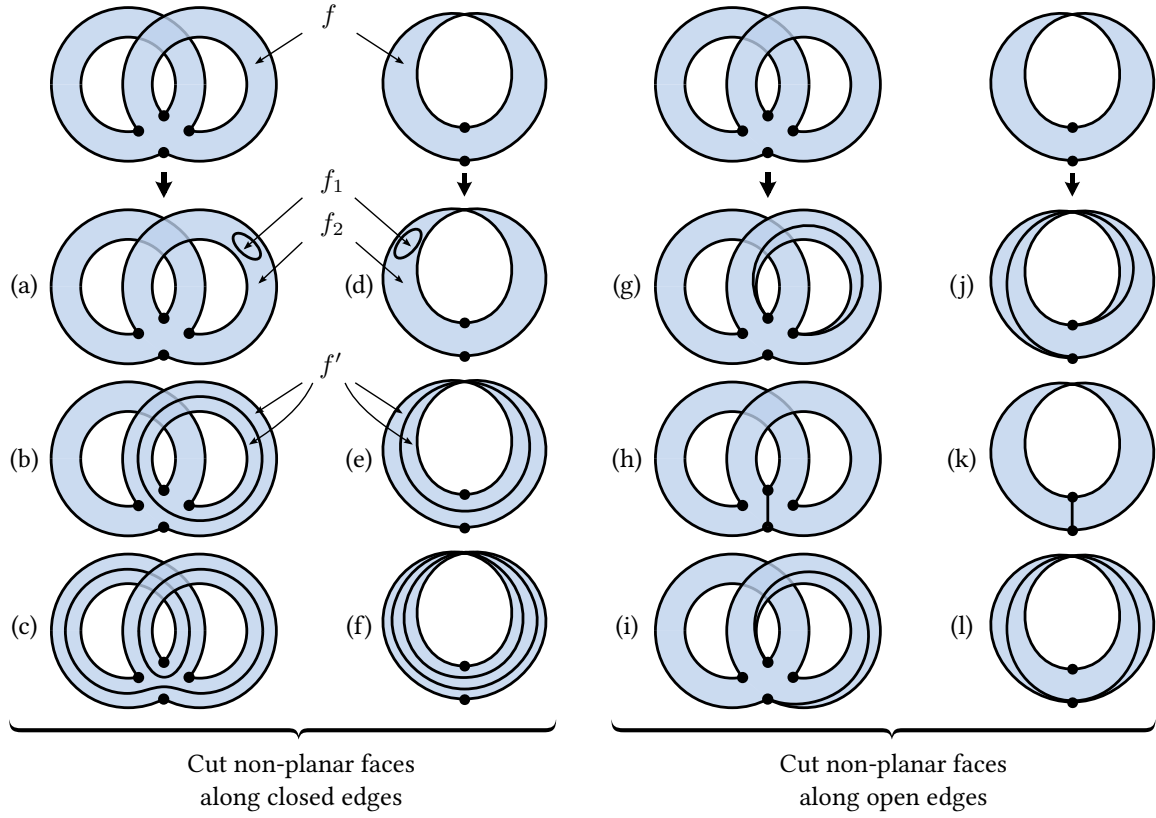
Cutting a planar face without inner holes along open or closed edges is easy and non-ambiguous: it always disconnects the face into two faces (see Figure 3.22a,e). However, if the face has one or



**Figure 3.22:** Cutting a planar face along a closed edge always disconnects the face (not true for non-planar faces). Cutting a planar face along an open edge starting and ending at the same hole always disconnects the face (not true for non-planar faces). Cutting a planar face along an open edge starting and ending at different holes never disconnects the face (also true for non-planar faces).

more inner holes, things get more complicated. For instance, let us look at the case of a planar face with  $k = 3$  holes, which we cut along an open edge (Fig. 3.22f–i). If the cut edge starts and ends at the same hole (Fig. 3.22f–h), we can observe that it also disconnects the face into two faces, like when cutting a disk into two half-disks. However, depending on which side of the cut each hole belongs to, different topologies will be obtained. A similar situation occurs if we cut the face along a closed edge (Fig. 3.22b–d): it disconnects the face, and the holes get distributed among the two resulting faces, depending on which side of the cut they belong.

But if we cut the face along an open edge that starts and ends at different holes (Fig. 3.22i), then an entirely different situation occurs: it does not disconnect the face. Instead, the two holes are simply merged into one, and the cut is actually non-ambiguous, regardless of the path taken by the cut edge. In fact, this property turns out to be also true for non-planar faces. In a sense, cutting a face along an open edge that starts and ends at different holes is the simplest way to cut a face along an edge. It is never ambiguous and can always be computed purely combinatorially, whether the face is planar or not.

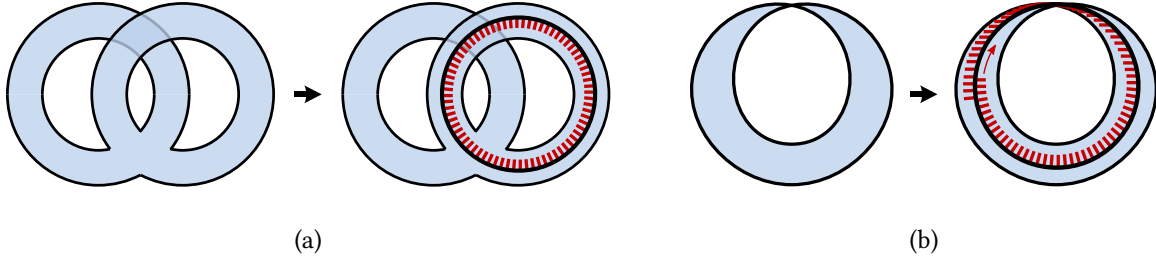


**Figure 3.23:** Some of the many ways non-planar faces can be cut along open and closed edges. Most of these cuts are not topologically equivalent: some disconnect the face into two faces, some do not. Some decrease the genus of the face, some do not. Some transform a non-orientable face into an orientable face, some do not. It is a good exercise to try to determine the values of  $\epsilon$ ,  $g$ , and  $k$  before and after the cut, in each of these examples. For an exhaustive list of the 19 non-equivalent ways to cut faces along edges, see Figure 3.25.

### 3.2.13 Cut Non-Planar Faces

In the previous section, we have discussed cutting planar faces along any type of edge, and cutting faces—planar or not—along open edges starting and ending at different holes. In this section, we finally discuss the two most complicated categories of cuts: cutting non-planar faces along closed edges, or along open edges starting and ending at the same hole.

**Orientable Faces** Let us take the example of a torus with one hole (i.e., an  $\odot$ -1-1 face), which we cut along a closed edge (see Figure 3.23a–c). One possibility is to cut along a tiny closed edge (Fig. 3.23a), which disconnects the face  $f$  into two faces  $f_1$  and  $f_2$ . This is always possible since by definition, every surface is locally homeomorphic to  $\mathbb{R}^2$ , thus one can cut out a tiny disk  $f_1$  anywhere on the surface. The remaining face  $f_2$  has the same orientability and genus as the original face  $f$ , but with an additional hole. In other words, the  $\odot$ -1-1 face is disconnected into an



**Figure 3.24:** (a) Cutting an orientable face along a closed edge always increases the total number of holes by exactly 2. The two additional holes correspond to the two sides of the closed edge. We highlighted one of these two sides using red stripes. (b) Cutting a non-orientable face along a closed edge increases the total number of holes by either 1 or 2. As we illustrate in this figure, the former case happens when the closed edge is globally one-sided, despite being locally two-sided. If you were walking on the face along the hole, you would need to walk twice the length of the closed edge before reaching your start position again.

$\odot$ -0-1 face and an  $\odot$ -1-2 face. We can observe that the total number of faces increases by 1, the total genus has not changed, and the total number of holes increases by 2.

However, since  $f$  has a non-zero genus, it is also possible to cut it in a way that does not disconnect it (Fig. 3.23b). Indeed, the number of times a surface can be cut without disconnecting it is in fact one possible definition of the genus of a surface. The resulting face  $f'$  is an  $\odot$ -0-3 face: a topological sphere with three holes. One of the holes is the original boundary, and the two new holes are the two sides of the cut edge, sides that just happen to belong to the same face. We can observe that the total number of faces has not changed, the total genus decreased by 1, and the total number of holes increased by 2.

**Non-Orientable Faces** Let us now take the example of a Möbius strip (i.e., a  $\oslash$ -1-1 face), which we also cut along a closed edge (Fig. 3.23d–f). Like for the torus, it is possible to cut out a tiny disk out of the Möbius strip (Fig. 3.23d), which disconnect the  $\oslash$ -1-1 face  $f$  into an  $\odot$ -0-1 face  $f_1$  and a  $\oslash$ -1-2 face  $f_2$ . However, it is also possible to cut  $f$  in a way that does not disconnect it, such as cutting along its centerline<sup>6</sup> (Fig. 3.23e). The resulting face  $f'$  is an  $\odot$ -0-2 face: a topological sphere with two holes, i.e., a topological cylinder. In addition to decreasing the genus (which was expected), two remarkable things happened: not only the face became orientable, but its number of holes increased by only 1, instead of increasing by 2 as expected. The reason is that despite being locally two-sided, the cut edge is in fact globally one-sided, and then only generates one additional hole, which circles along the closed edge two times (Fig. 3.24b).

Finally, we have yet to discuss cutting non-planar faces along open edges starting and ending at the same hole (see Figure 3.23, right). However, it turns out that this case shares many similarities with cutting non-planar faces along closed edges. Therefore, for conciseness, we do not discuss it

<sup>6</sup>If you have never done it before, I highly recommend to take an actual strip of paper (about 2cm  $\times$  20cm), tape its two ends after a half-twist to create a Möbius strip, then cut along its centerline with scissors to see what happens.

here and refer to Appendix D for more details. As a quick way to get more insight, we encourage the reader to try to answer the following four questions (answers given in this footnote<sup>7</sup>):

1. We cut an  $\odot$ -1-1 face along an open edge *that does not disconnect it* (Fig. 3.23h) . How many holes does it have after the cut?
2. Do the two sides of the cut edge belong to the same hole?
3. Consider the same two questions as applied to a  $\oslash$ -1-1 face (Fig. 3.23k).
4. Compare these results with the analogous case of cutting these same faces along a closed edge.

### 3.2.14 The Face-Cut Classification

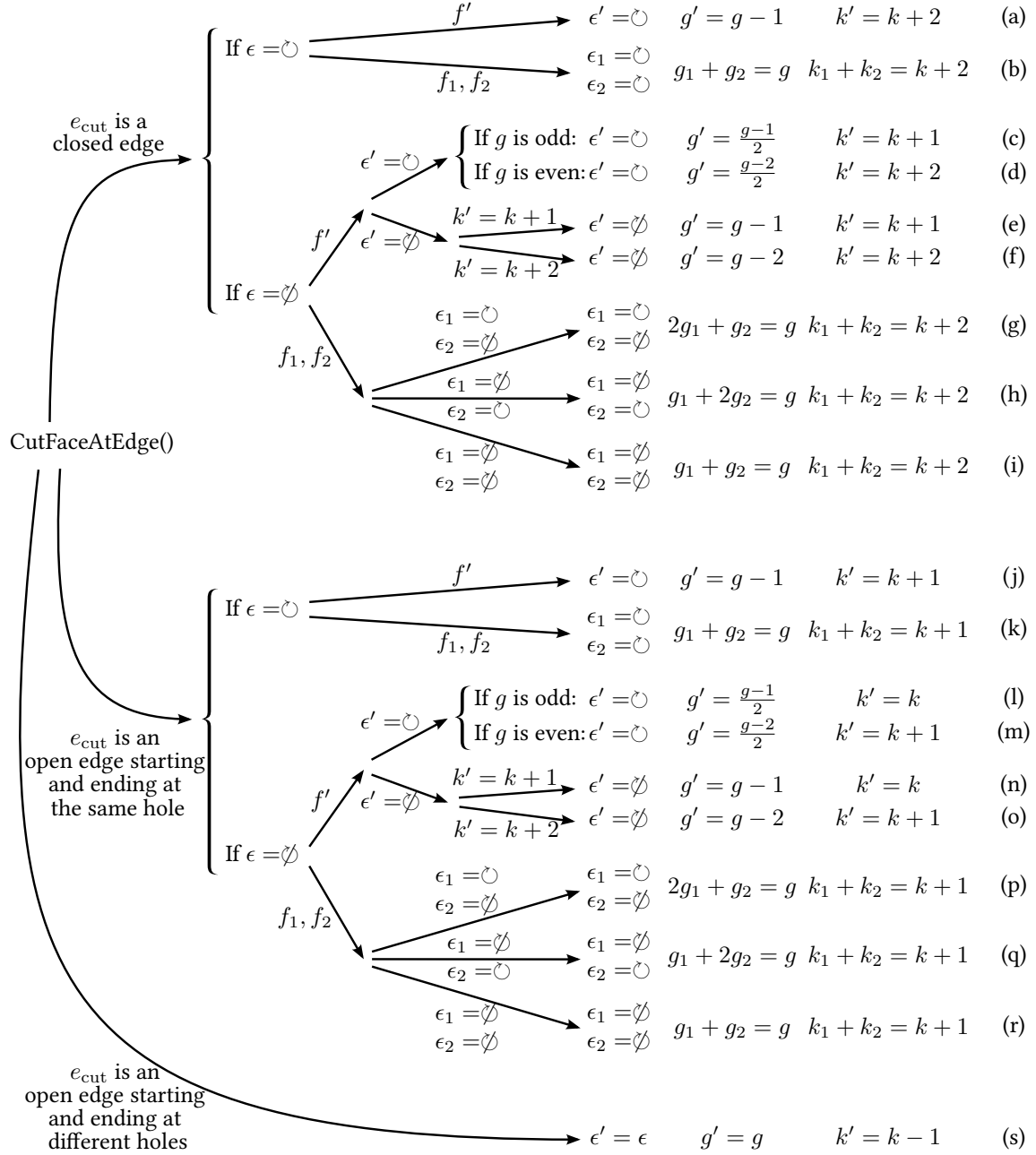
At this point, we hope to have successfully conveyed the idea that vector graphics topology is not as trivial as it may seem. In particular, cutting non-planar faces, and especially non-orientable faces, is far from obvious. And in fact, we even have omitted the most complicated cases for conciseness. For instance, there are 6 non-equivalent ways to cut a  $\oslash$ -3-1 face along a closed edge. Half of these do not disconnect the face, and respectively transform it into an  $\odot$ -1-2 face, a  $\oslash$ -2-2 face, and a  $\oslash$ -1-3 face. This is very different from the cases we had already seen, where there was at most one way to cut each face along a closed edge without disconnecting it.

One may start to worry that by considering faces of even higher genus, for instance  $\oslash$ -7-1 faces, then there would be even more ways to cut them. Fortunately, this is not the case: there are only 3 ways to cut a  $\oslash$ -7-1 face along a closed edge without disconnecting it, which are the same 3 ways that apply to a  $\oslash$ -3-1 face. In other words, cutting faces gets more complicated up to genus 3, at which point all possible cuts have already been discovered. The exhaustive list of the 19 possible cuts is illustrated in Figure 3.25, and we call this list the **face-cut classification**. We prove this classification in Appendix D, using the formalism of **PCS complexes** which we define in the next section. Together with the definition of PCS complexes itself, the face-cut classification is perhaps the most significant theoretical contribution of this thesis.

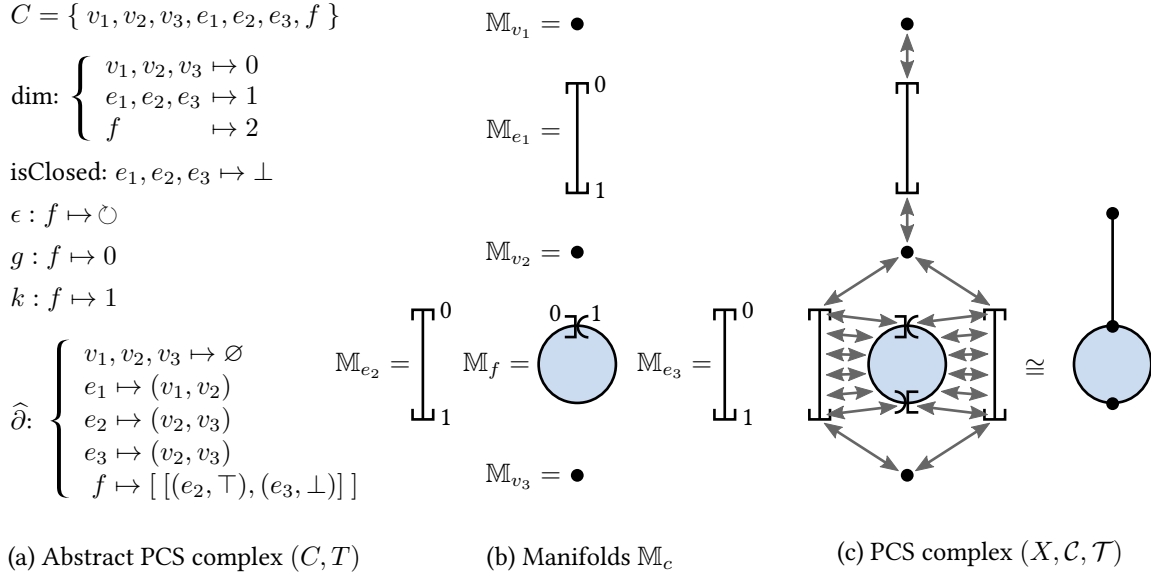
---

<sup>7</sup>(1) 2. (2) No. (3) 1, Yes (4) When cutting with a closed edge, the answers were 3-No-2-Yes, i.e. the same answers with a one-hole offset.





**Figure 3.25:** Exhaustive classification of the 19 different ways a face can be cut along an edge. The branching “if”s represent known information about the face to be cut (e.g., is it orientable or not?). The branching arrows represent different cuts that can be done, each leading to different topological properties (e.g., either disconnects the face, or not). We illustrate all these 19 types of cuts in Figure D.5 and Figure D.6. Note that planar maps only support (b), (k), and (s), and which of the three to apply can be fully determined combinatorially, unlike in the general case.



**Figure 3.26:** (a) Example of abstract PCS complex  $\mathcal{P} = (C, T)$ , where  $T = (\text{dim}, \text{isClosed}, \epsilon, g, k, \hat{\partial})$ , consisting of 7 abstract cells  $c \in C$ . (b) The characteristic manifold  $\mathbb{M}_c$  associated with each abstract cell, as specified by the functions  $\text{dim}$ ,  $\text{isClosed}$ ,  $\epsilon$ ,  $g$ , and  $k$ . (c) The geometric realization  $|\mathcal{P}| = (X, \mathcal{C}, \mathcal{T})$  of  $\mathcal{P}$ , defined by gluing the boundary of each manifold  $\mathbb{M}_c$  to manifolds of lower dimension, as specified by  $\hat{\partial}$ .

### 3.3 PCS Complexes

In Section 3.2, starting from a short list of design decisions, we informally deduced various topological objects and operators that vector graphics tools should support, such as non-planar topologies, and even non-planar individual cells. In this section, we introduce the concept of **PCS complex**, which is a topological structure satisfying these requirements. This particular section is more technical in nature and provides insights into the key theoretical contributions of this thesis. It is optional reading for the reader with more applied interests, who will already have developed a useful intuition of the problems at hand from the first two sections of this chapter.

Very briefly, a PCS complex (= Point-Curve-Surface complex) is a topological space—in other words, a continuous point-set—which is partitioned into cells that have to satisfy some constraints (see Figure 3.26c). It is similar to the concept of CW complex, but has different definitions of cells and constraints, in order to make it suitable for vector graphics. Importantly, unlike CW complexes, PCS complexes can be defined in terms of a combinatorial structure which we call **abstract PCS complex** (see Figure 3.26a).

This section is organized as follows. In Section 3.3.1, we define the concept of *abstract PCS complex*. It is a combinatorial structure consisting of symbols  $c \in C$  and some topological information  $T$  about them. Then, in Section 3.3.2, we define the concept of *PCS complex*, as geometric realizations

of abstract PCS complexes. In order to clarify these formal definitions, we discuss them and provide examples in Section 3.3.3. Finally, in Section 3.3.4, we show how the definition of *vector graphics complexes* is derived from the definition of abstract PCS complexes.

### 3.3.1 Abstract PCS complexes

An **abstract PCS complex** is an ordered pair  $\mathcal{P} = (C, T)$ , such that:

- $C$  is a finite set of symbols called **abstract cells**
- $T = (\text{dim}, \text{isClosed}, \epsilon, g, k, \hat{\partial})$  is a tuple of functions together called **abstract topology**. They assign topological information to abstract cells, as we detail below.
- $\text{dim} : C \rightarrow \{0, 1, 2\}$  is a function that assigns a **dimension** to each abstract cell. This defines a partition of  $C$  into three sets  $V$ ,  $E$ , and  $F$  of elements respectively called **abstract vertices**, **abstract edges**, and **abstract faces**. For conciseness, we now omit the adjective *abstract* when it is clear that we are referring to combinatorial objects.
- $\text{isClosed} : E \rightarrow \{\top, \perp\}$  is a function that assigns a **closedness** to each edge. This defines a partition of  $E$  into two sets  $E_{\circ}$  and  $E_{\perp}$  of elements respectively called **closed edges** and **open edges**.
- $\epsilon : F \rightarrow \{\circ, \emptyset\}$  is a function that assigns an **orientability** to each face.
- $g : F \rightarrow \mathbb{N}$  is a function that assigns a **genus** to each face. This genus must be non-zero for non-orientable faces.
- $k : F \rightarrow \mathbb{N}$  is a function that assigns a **number of holes** to each face.
- $\hat{\partial}$  is a function that assigns an **ordered boundary** to each cell. This ordered boundary, which we detail below, is what defines the incidence relationship between cells.
- For each  $v \in V$ , we have  $\hat{\partial}v = \emptyset$ .
- For each  $e \in E_{\circ}$ , we have  $\hat{\partial}e = \emptyset$ .
- For each  $e \in E_{\perp}$ , we have  $\hat{\partial}e \in V \times V$ . We denote by  $v_{\text{start}}(e)$  and  $v_{\text{end}}(e)$  the first and second element of the ordered pair.
- For each  $f \in F$ , we have  $\hat{\partial}f \in \Gamma^{k(f)}$ . In other words, it is an ordered sequence of  $k(f)$  cycles  $\gamma_i \in \Gamma$ , where  $\Gamma$  is the set of all possible cycles on  $\mathcal{P}$ , which we define below.
- A **halfedge**  $h = (e, \beta)$  is a pair of an edge  $e \in E$  and a direction  $\beta \in \{\top, \perp\}$ . If  $e$  is a closed edge, then  $h$  is called a **closed halfedge**, otherwise it is called an **open halfedge**. For each

open halfedge  $h$ , we denote by  $v_{\text{start}}(h)$  and  $v_{\text{end}}(h)$  the following vertices:

$$v_{\text{start}}(h) = \begin{cases} v_{\text{start}}(e), & \text{if } \beta = \top \\ v_{\text{end}}(e), & \text{if } \beta = \perp \end{cases} \quad \text{and} \quad v_{\text{end}}(h) = \begin{cases} v_{\text{end}}(e), & \text{if } \beta = \top \\ v_{\text{start}}(e), & \text{if } \beta = \perp \end{cases} \quad (3.2)$$

• A **cycle**  $\gamma \in \Gamma$  is either:

1. a vertex  $v \in V$ , or
2. a pair  $(h, N)$  consisting of a closed halfedge  $h$  and an integer  $N > 0$ , or
3. a non-empty, ordered sequence  $(h_j)_{j \in [1..N]}$  of open halfedges such that:

$$\forall j \in [1..N], \quad v_{\text{end}}(h_j) = v_{\text{start}}(h_{(j+1) \bmod N}) \quad (3.3)$$

In the first case, the cycle is called a **Steiner cycle**; in the second case, it is called a **simple cycle**; and in the third and last case, it is called a **non-simple cycle**.

### 3.3.2 PCS complexes

In this section, for each abstract PCS complex  $\mathcal{P} = (C, T)$ , we define a structure called the **geometric realization** of  $\mathcal{P}$ . It is a triplet  $|\mathcal{P}| = (X, \mathcal{C}, \mathcal{T})$  consisting of a point-set  $X$ , a partition  $\mathcal{C}$  of  $X$ , and a topology  $\mathcal{T}$  on  $X$ . The elements of  $\mathcal{C}$  are called the (non-abstract) **cells** of  $|\mathcal{P}|$ .

A **PCS complex** is then defined as any triplet  $\mathcal{K} = (X, \mathcal{C}, \mathcal{T})$  such that there exists an abstract PCS complex  $\mathcal{P}$  whose geometric realization  $|\mathcal{P}| = (X', \mathcal{C}', \mathcal{T}')$  is **isomorphic** to  $\mathcal{K}$ . By isomorphic, we mean that there exists a homeomorphism  $\phi : X \rightarrow X'$ , continuous with respect to  $\mathcal{T}$  and  $\mathcal{T}'$ , such that each cell  $c \in \mathcal{C}$  is mapped by  $\phi$  to a cell  $c' \in \mathcal{C}'$ .

Let  $\mathcal{P} = (C, T)$  be an abstract PCS complex. We define  $|\mathcal{P}| = (X, \mathcal{C}, \mathcal{T})$  as follows, where the topology  $\mathcal{T}$  of each topological space is assumed to be the usual topology on compact manifolds and quotient spaces.

**Characteristic Manifolds** For each abstract cell  $c \in C$ , we define a compact manifold  $\mathbb{M}_c$ , called the **characteristic manifold** of  $c$ , as follows:

- The characteristic manifold of each vertex  $v \in V$  is the single point  $\{0\}$ .
- The characteristic manifold of each closed edge  $e \in E_o$  is the unit circle  $\mathbb{S}^1$ .
- The characteristic manifold of each open edge  $e \in E_l$  is the segment  $[0, 1]$ .

- The characteristic manifold of each face  $f \in F$  is the unit<sup>8</sup> compact surface of orientability  $\epsilon(f)$ , genus  $g(f)$ , with  $k(f)$  holes.

**Disjoint Union** We define  $Y$  as the **disjoint union**  $Y = \coprod_{c \in C} \mathbb{M}_c$ . This simply means that  $Y$  is a topological space composed of all  $\mathbb{M}_c$  together, preventing intersections by attaching a unique ID to each  $\mathbb{M}_c$ . A formal definition can be found in [Lee 2011, p64].

**Quotient Space** We define  $X$  as the **quotient space** obtained from  $Y$  by gluing<sup>9</sup> the boundary  $\partial \mathbb{M}_c$  of each characteristic manifold to lower-dimensional manifolds, as follows:

- For vertices and closed edges, there is nothing to glue since  $\partial \mathbb{M}_c = \emptyset$ .
- For each open edge  $e \in E|$ , we respectively glue the start and end point of  $\mathbb{M}_e = [0, 1]$  to the point  $\mathbb{M}_{v_{\text{start}}(e)}$  and the point  $\mathbb{M}_{v_{\text{end}}(e)}$ .
- For each face  $f \in F$ , we glue the points of  $\partial \mathbb{M}_f$  as follows.

First, we recall that  $\mathbb{M}_f$  is a compact surface with  $k(f)$  holes. This means that  $\partial \mathbb{M}_f$  consists of  $k(f)$  connected components  $\mathbb{B}_i$ , each homeomorphic to a circle. For clarity, we assume that all manifolds  $\mathbb{M}$  homeomorphic to a circle are consistently parameterized<sup>10</sup> by  $\theta \in [0, 1)$ , and we use the notation  $\mathbb{M}[\theta]$  to refer to the point of  $\mathbb{M}$  corresponding to the parameter  $\theta$ . If  $\theta$  lies outside the range  $[0, 1)$ , we implicitly use its fractional part  $\theta - \lfloor \theta \rfloor$  instead.

Back to the definition. For each cycle  $\gamma_i$  of  $\hat{\partial} f$ , we glue the following points:

1. If  $\gamma_i$  is a vertex  $v$ , we glue the whole circle  $\mathbb{B}_i$  to the point  $\mathbb{M}_v$ .
2. If  $\gamma_i$  is a pair  $(h, N)$  consisting of a closed halfedge  $h = (e, \beta)$  and an integer  $N > 0$ , then we glue each point  $\mathbb{B}_i[\theta]$  to the point  $\mathbb{M}_e[\theta']$ , where:

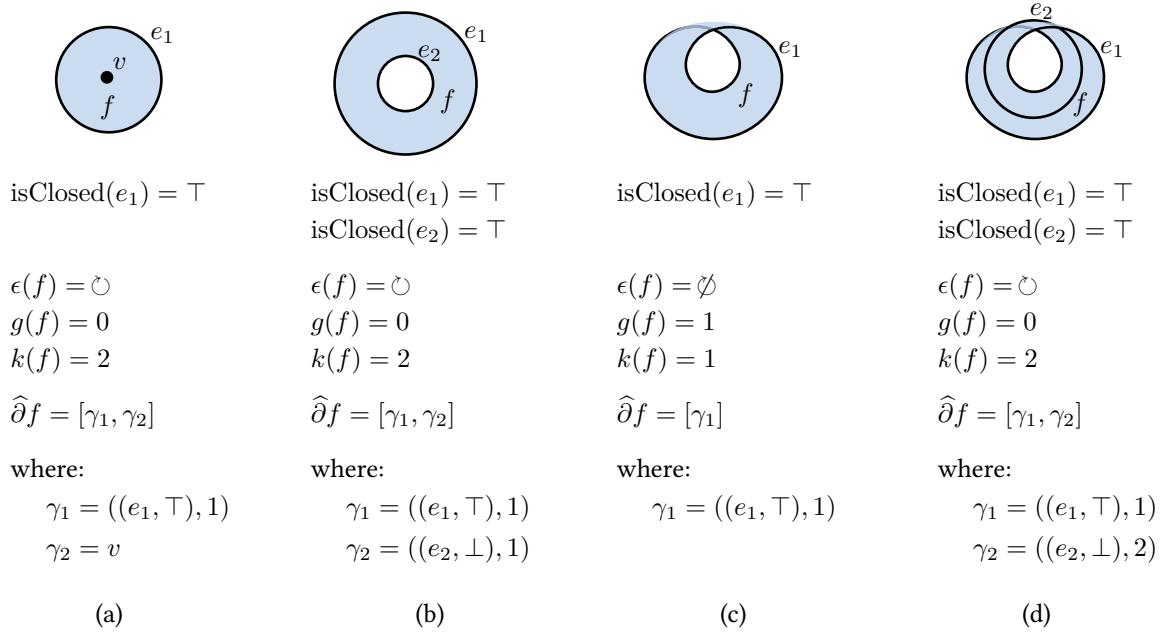
$$\theta' = \begin{cases} N\theta, & \text{if } \beta = \top \\ -N\theta, & \text{if } \beta = \perp \end{cases} \quad (3.4)$$

3. If  $\gamma_i$  is a sequence  $(h_j)_{j \in [1..N]}$  of open halfedges  $h_j = (e_j, \beta_j)$ , then we first partition the hole  $\mathbb{B}_i$  into  $N$  sub-arcs  $\mathbb{B}_{i,j}$ , each defined by  $\theta \in [\frac{j-1}{N}, \frac{j}{N})$ . Then, for convenience, we reparameterize these sub-arcs by  $u \in [0, 1)$ . Finally, for each sub-arc  $\mathbb{B}_{i,j}$ , we glue

<sup>8</sup>Here, the term *unit* refers to any surface of reference which we assume has been arbitrarily fixed beforehand, in order to uniquely define  $|\mathcal{P}|$ . The choice of this unit surface influences the definition of  $|\mathcal{P}|$ , but only within the same isomorphism class. Thus, it does not influence the definition of PCS complexes, which are defined via isomorphisms.

<sup>9</sup>See Appendix A.6 for the precise meaning of *quotient space* and *gluing*.

<sup>10</sup>We detail what this means in Section 3.3.3, paragraph *Consistent Parameterization*. This disambiguates which direction is “clockwise”, and which direction is “counter-clockwise”.



**Figure 3.27:** Four examples of PCS complexes  $\mathcal{K}$ , together with their combinatorial representation  $\mathcal{P}$ . For conciseness, we did not mention the set  $C$  or the function  $\text{dim}$ , which can be easily inferred from the figure. Also, we did not mention the value of  $\hat{\partial}$  for vertices and closed edges, which is always equal to the empty set.

each point  $\mathbb{B}_{i,j}[u]$  to the point  $\mathbb{M}_{e_j}[u']$ , where:

$$u' = \begin{cases} u, & \text{if } \beta_j = \top \\ 1 - u, & \text{if } \beta_j = \perp \end{cases} \quad (3.5)$$

**Cell Decomposition** Finally, for each abstract cell  $c \in C$ , we define its **geometric realization**  $|c|$  as the subset of  $X$  that corresponds to the interior of  $\mathbb{M}_c$ . Since the boundary of each characteristic manifold  $\mathbb{M}_c$  is glued to characteristic manifolds of lower dimension, and since their interior is left untouched, we can deduce that the subsets  $|c|$  define a partition of  $X$ . We define  $\mathcal{C}$  to be this partition, which completes the definition of  $|\mathcal{P}| = (X, \mathcal{C}, \mathcal{T})$ .

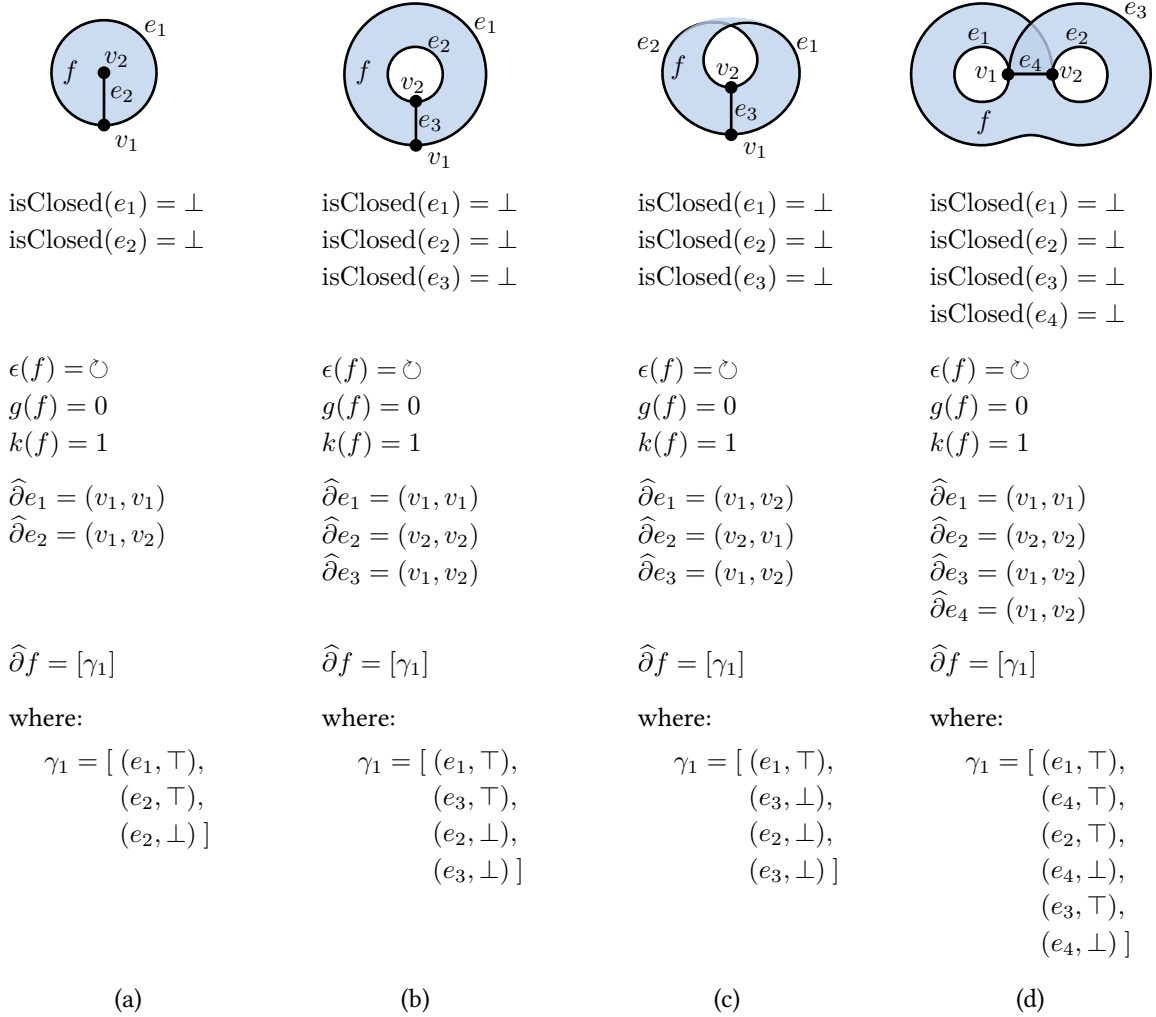
### 3.3.3 Examples and Discussions

In Figure 3.26, we give a simple but detailed example of abstract PCS complex  $\mathcal{P} = (C, T)$ , together with its geometric realization  $|\mathcal{P}| = (X, \mathcal{C}, \mathcal{T})$ , and an illustration of how  $|\mathcal{P}|$  is defined by stitching together the characteristic manifolds  $\mathbb{M}_c$ . We recommend to spend some time to analyze this example while reading the definitions, which by itself should already provide a good understanding of the structure. In this section, we clarify the most important or least obvious points.

**Cell Types** An abstract PCS complex is a combinatorial structure made of different types of cells: vertices, closed edges, open edges, and faces of various orientability, genus, and number of holes. The list of cells is given by the set  $C$ , and their topological type is given by the functions  $\dim$ ,  $\text{isClosed}$ ,  $\epsilon$ ,  $g$ , and  $k$ . This choice of types is not arbitrary but a direct consequence of the classification of compact  $n$ -manifolds for  $n \leq 2$  (for more details, see Section A.4).

**Hole Types** The incidence relationship between cells is given by the function  $\hat{\partial}$ , which specifies the start and end vertex of open edges, and the bounding vertices and edges of faces. Since the former is quite straightforward, let us only clarify the latter. As we have seen in Section 3.2, the boundary of every face is composed of a given number of holes (this includes both the outer boundary and inner holes, if any). Each of these holes is composed of vertices and/or edges, organized in a specific order, and with a specific direction. We represent this organization via the concept of *cycle*. We can identify three categories of holes, which leads to three categories of cycles:

1. There are holes that are composed of a single vertex and no edges, i.e. a *point-in-face*, or *Steiner vertex*. These holes are very easy to represent combinatorially: one must just refer to this vertex (see Figure 3.27a,  $\gamma_2$ ).
2. There are holes that are composed of a single closed edge, no vertices, no open edges (see Figure 3.27b–d). One may think that these can also be easily represented combinatorially, by simply referring to the closed edge. This is almost true, but with two subtleties:
  - First, one must choose a direction for the closed edge. Indeed, if you glue together the two holes of a cylinder, you can either get a torus or a Klein bottle, depending on which direction is chosen. Therefore, instead of simply specifying the closed edge  $e$ , we need to specify a pair  $h = (e, \beta)$  of the closed edge with a direction.
  - Second, the hole may “circle around” the closed edge several times. This is for instance what happens when you cut a Möbius strip in its centerline (see Figure 3.24b). The non-orientable face becomes orientable, and it gains one new hole which is composed of a closed edge “used two times”. This latter information is critical: without it, there would be no way to distinguish this “cut-Möbius” face (Fig. 3.27d) from a disk with one inner hole (Fig. 3.27b). There would be no way to know that it is possible to uncut at this closed edge, and that such uncut makes the face non-orientable. Therefore, instead of specifying a closed halfedge  $h$ , we need in fact to specify a pair  $(h, N)$ , where  $N > 0$  specifies the number of times the hole circles around the closed edge. Typically,  $N = 1$ , but in the case of the cut-Möbius,  $N = 2$ , and in fact any value of  $N > 0$  is possible.
3. Finally, there are holes that are composed of vertices and open edges (see Figure 3.28). This is the most typical case, the most intuitive, but the most complex from a combinatorial

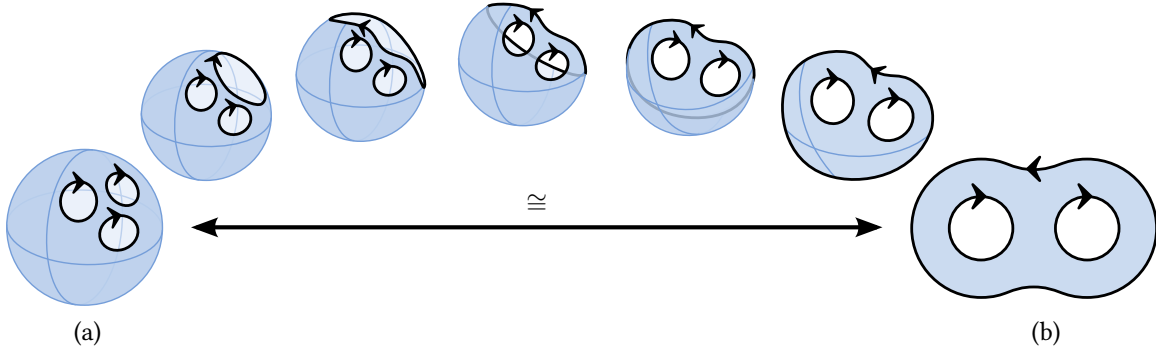


**Figure 3.28:** More examples of PCS complexes  $\mathcal{K}$ , together with their combinatorial representation  $\mathcal{P}$ .

standpoint. They are combinatorially represented by a *cycle*: a path of consecutive directed open edges, which starts and ends at the same vertex. Intuitively, this cycle is obtained by starting at any arbitrary vertex of the hole, then walking on the surface, alongside the hole, until we are back at the same position *on the surface*. Note that being back at the start vertex is not enough, since it can appear multiple times in the cycle. Also, edges can appear any number of times in the cycle, with any direction. In Figure 3.28, we give many examples where an edge is used two times by the same cycle. In Figure 3.28d, there is even an edge used three times by the same cycle. Note that if an edge is used three or more times (by the same cycle or not), then the overall topology is both non-manifold and non-planar.

**Analogy with Graphs** The set of abstract cells  $C$  plays the same role as the set of nodes  $V$  in a graph  $\mathcal{G} = (V, E)$  (which is to provide an “identity” to every object in the structure), and the



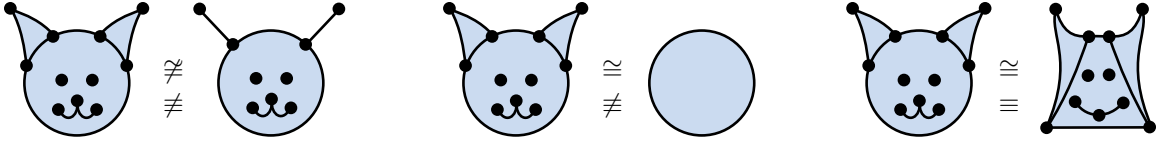


**Figure 3.29:** (a) In order to unambiguously define the geometric realization of an abstract PCS complex, all holes must be oriented “consistently”. For instance, here, walking along any hole leaves the surface “to the left”. (b) This means that if a face is embedded in  $\mathbb{R}^2$ , then the outer boundary must have the opposite clockwiseness than the inner holes. Indeed, when stretching this outer boundary oriented like so, we see that it turns into a hole of a sphere, with an orientation consistent with the other holes.

tuple of functions  $T$  plays the same role as  $E$  (which is to represent *how these objects are connected to one another*). However, in the case of graphs all nodes have the same “type”, while in the case of abstract PCS complexes there exist different types of cells. Assigning this type is another role of  $T$ , which can be seen as coloring the nodes of a graph. Besides the usual graphs  $\mathcal{G} = (V, E)$  where edges are unordered pairs of vertices without identity, it is perhaps more interesting to make an analogy with directed multigraphs, defined as  $\mathcal{G} = (V, E, r)$ , where  $V$  and  $E$  are two sets of symbols (i.e., edges also have an identity), and  $r : E \rightarrow V \times V$  assigns a *source* vertex and a *target* vertex to every edge. Such structure could be equivalently defined as  $\mathcal{G} = (C, T)$ , with  $T = (\text{dim}, r)$ , clearly classifying information as being either “identity” or “topology”, like in our formalism but without closed edges and faces, and where  $r$  plays the role of  $\hat{\partial}$ . This shows that PCS complexes are in fact a superset of directed multigraphs, and among them stroke graphs.

**Geometric Realization** In Section 3.3.2, we defined the geometric realization  $|\mathcal{P}|$  of each abstract PCS complex  $\mathcal{P}$ , by defining a characteristic manifold  $\mathbb{M}_c$  to each abstract cell, then gluing these manifolds together in a way specified by  $\hat{\partial}$ . Then, we defined a PCS complex as any topological space, together with a cell decomposition, which is isomorphic to the geometric realization of an abstract PCS complex. However, it is also possible to define PCS complexes independently from abstract PCS complexes, similarly to how CW complexes are usually defined. We provide such a definition in Appendix B.

**Consistent Parameterization** In the definition of the geometric realization  $|\mathcal{P}|$  of an abstract PCS complex  $\mathcal{P}$ , we mentioned that the holes of the characteristic manifold  $\mathbb{M}_f$  of a given face  $f$  are assumed to be *consistently* parameterized by  $\theta \in [0, 1)$ . For conciseness, we did not define the exact meaning of “consistently”, but it is actually a critical part of the definition, and we clarify it



**Figure 3.30:** Examples of PCS complexes which are either homeomorphic, or isomorphic, or both, or neither. Note that if two PCS complexes are isomorphic, then they are also homeomorphic.

here. It means that for any given  $\mathbb{M}_f$ , the chosen orientations for all its holes are consistent with one another. More precisely, this means that if we walk along any hole with the chosen orientation, the surface is always on the same side of the hole, e.g., “to the left” (see Figure 3.29a). As a consequence, this means that for planar faces—which are the most typical in vector graphics—the “outer boundary” is oriented with the opposite clockwiseness than the inner holes (see Figure 3.29b). Besides, we note that any orientable surface has two sides A and B. When we say “walk along any hole”, we mean more precisely “walk on side A of the surface, alongside any hole”. If you were walking on side B, then the surface would be “to the right” instead of “to the left”. For any given face, which side or which orientation is chosen does not matter, as long as the choice is *consistent* across all holes. Finally, we note that this whole concept of consistency only applies to orientable faces. Non-orientable faces only have one side, like in a Möbius strip, and the orientations of their holes are irrelevant. They can be chosen arbitrarily and independently; in each case it will define the same geometric realization  $|\mathcal{P}|$  up to isomorphism.

**Homeomorphism and Isomorphism** We recall that a PCS complex  $\mathcal{K}$  is a topological space together with a (valid) cell decomposition. Therefore, we define a **homeomorphism** between two PCS complexes  $\mathcal{K}$  and  $\mathcal{K}'$  to be a homeomorphism between their topological spaces (ignoring their cell decompositions). However, in Section 3.3.2, we also defined the stronger concept of **isomorphism**, which is a homeomorphism between two PCS complexes such that every cell of  $\mathcal{K}$  is mapped into a cell of  $\mathcal{K}'$ . In other words, if  $\mathcal{K}$  and  $\mathcal{K}'$  are isomorphic, not only they are homeomorphic, but they also have the “same” cell decomposition. In the algebraic topology literature, this concept is sometimes called *cellular homeomorphism*, or *cellular equivalence*. We use the notation  $\mathcal{K} \cong \mathcal{K}'$  to denote homeomorphic PCS complexes, and the notation  $\mathcal{K} \equiv \mathcal{K}'$  to denote isomorphic PCS complexes (see Figure 3.30 for examples). These definitions can be extended for abstract PCS complexes: two abstract PCS complexes  $\mathcal{P}$  and  $\mathcal{P}'$  are said homeomorphic (resp. isomorphic) if and only if their geometric realizations  $|\mathcal{P}|$  and  $|\mathcal{P}'|$  are homeomorphic (resp. isomorphic). Abstract PCS complex homeomorphism is very similar to graph homeomorphism. However, abstract PCS complex isomorphism is a bit more subtle than graph isomorphism. Indeed, within an isomorphism class, not only can cells be renamed (as in graph isomorphism), but in addition, cycles can be re-ordered, rotated (= different start vertex), sometimes reversed (e.g., in the case of non-orientable faces), and edge directions can sometimes be flipped. This captures the intuitive concept

that for any (non-trivial) PCS complex, there are many abstract PCS complexes that can represent it, which are *technically* different, but should really be considered the same. Deciding whether two abstract PCS complexes are isomorphic is a NP-hard problem<sup>11</sup>, since it is at least as hard as graph isomorphism, which is NP-complete.

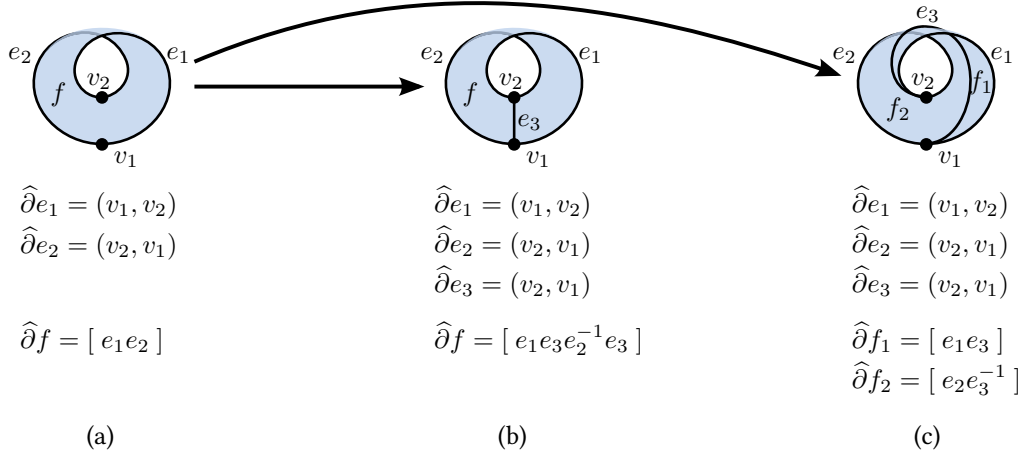
**Boundary and Incidence Graph** From the definition of ordered boundary  $\hat{\partial}$ , we can define the concept of **boundary**  $\partial$  for abstract PCS complexes. For each abstract cell  $c$ , its boundary  $\partial c$  is defined as the set of cells “involved” in  $\hat{\partial}c$ . More precisely, it is defined as follows:  $\partial c = \emptyset$  for vertices and closed edges;  $\hat{\partial}e = \{v_{\text{start}}(e), v_{\text{end}}(e)\}$  for open edges; and for faces,  $\partial f$  is composed of its Steiner vertices, bounding closed edges, bounding open edges, and their start and end vertices. Equivalently, it can also be defined as the set of abstract cells  $c'$  whose geometric realization  $|c'|$  is contained in the point-set defined by  $\partial \mathbb{M}_c$ . In any case, this allows us to define the **incidence graph** of any abstract PCS complex, as the directed graph  $\mathcal{G} = (C, E)$  where  $C$  is the set of abstract cells, and  $(c, c') \in E$  if and only if  $c' \in \partial c$ . One may think that this incidence graph, or perhaps simply replacing  $\hat{\partial}$  by  $\partial$  in the definition of  $\mathcal{P}$ , is enough to uniquely define an abstract PCS complex. Unfortunately, this is not true: there exist many abstract PCS complexes which are not isomorphic, but that have the same incidence graph. In other words, the concept of *ordered* boundary is critical. This order specifies direction of edges, which can make the difference between a planar or a non-planar topology. This order specifies which edges are used several times by the same face, which can make the difference between a manifold or non-manifold topology. In practice, all of this has an influence on the behavior of topological operators.

**Multiplicative Notation for Cycles** For convenience, it is useful to denote cycles via a multiplicative notation, which is a very common practice in algebraic topology (see Figure 3.31). In fact, one may argue that it is what makes algebraic topology, well, *algebraic*. Using this notation, the sequence of open halfedges  $[(e_1, \top), (e_2, \perp), (e_3, \top)]$  can simply be written as  $e_1 e_2^{-1} e_3$ . A cycle consisting of a Steiner vertex  $v$  is obviously written “ $v$ ”, and a cycle consisting of a pair  $((e, \beta), N)$  of a closed halfedge and integer is written as  $e^N$ , or  $e^{-N}$  (or simply  $e$  if  $N = 1$  and  $\beta = \top$ ). If  $\beta$  is an unknown variable, intuitive notations such as  $e^\beta$ ,  $e^{-\beta}$ , or  $e^{\beta N}$  can of course be used.

**Topological Operators** Given an abstract PCS complex  $\mathcal{P}$ , it is possible to transform it to a different abstract PCS complex  $\mathcal{P}'$ , by using a **topological operator**. Each topological operator is a combinatorial algorithm that implements a behavior *described in terms of point-sets*, such as glue and cut (see Section 3.2). In fact, being able to do so is one of the most important reason why we developed the formalism of PCS complexes in the first place. It allows us to have a one-to-one

---

<sup>11</sup>More precisely, it is very likely a NP-complete problem, but proving that it is actually in NP is not obvious. We believe that it can be done by realizing the spaces as triangulations, and using some trick with the genus to keep the size of the triangulation polynomial in the size of  $\mathcal{P}$ , but we leave a formal proof for future work.

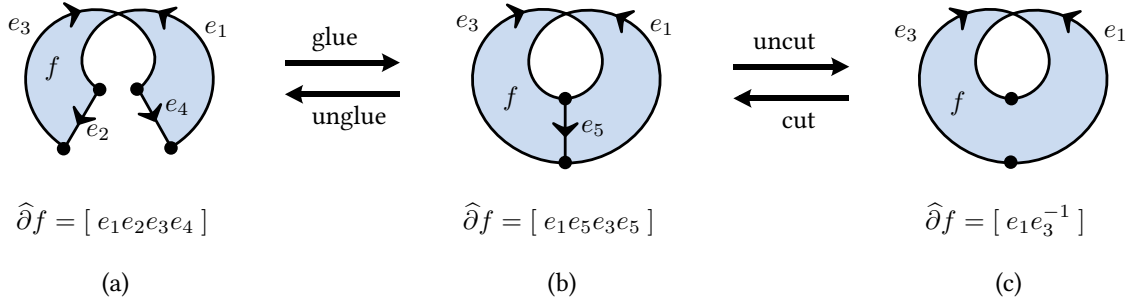


**Figure 3.31:** Two examples of cut operators on a Möbius strip. We use here a multiplicative notation for cycles, where  $e$  means the halfedge  $(e, \top)$ , and  $e^{-1}$  means the halfedge  $(e, \perp)$ . Note that for both cut operators, the cut edge  $e_3$  is an open edge starting at  $v_2$  and ending at  $v_1$ , but still the cuts are non-equivalent.

mapping (up to isomorphism) between combinatorial structures  $\mathcal{P}$  and point-set decompositions  $\mathcal{K}$ , supporting closed edges and non-planar faces. To the best of our knowledge, this is the first time it has ever been achieved, at the very least applied for vector graphics. This makes possible to rigorously *infer* any topological operator algorithm from a behavior specified in terms of point-sets, instead of arbitrarily picking an algorithm that *seems* to implement the desired behavior. For instance, cutting a face  $f$  along an open edge can be defined as extracting from its geometric realization  $|f|$  any open curve starting and ending at a vertex, then decomposing the remaining point-set into connected components. With this definition, we have been able to methodically discover and classify all **non-equivalent cuts**, where non-equivalent means that the resulting PCS complexes are not isomorphic. In Figure 3.31, we illustrate two of such non-equivalent cuts, and in Appendix D, we detail all topological operators and their corresponding algorithms.

### 3.3.4 Vector Graphics Complexes

A **vector graphics complex** (VGC) is defined as an abstract PCS complex with “unknown” orientability and genus. In other words, a VGC is a pair  $\mathcal{P} = (C, T)$  where  $T = (\dim, \text{isClosed}, k, \widehat{\partial})$ , and where  $C$ ,  $\dim$ ,  $\text{isClosed}$ ,  $k$ , and  $\widehat{\partial}$  have the same definition as for abstract PCS complexes (see Section 3.3.1). Of course, there are many other ways to define this structure, such as  $\mathcal{P} = (V, E, F, \widehat{\partial})$ , especially since  $\text{isClosed}$  and  $k$  can in fact be deduced from  $\widehat{\partial}$ . All of these are equivalent and simply a matter of taste. In Chapter 4, we discuss vector graphics complexes at length, focusing mostly on practical considerations such as how to immerse them in  $\mathbb{R}^2$ , how to specify a depth ordering for overlapping cells, and how to build an intuitive user interface around this concept. In this section, we only discuss how vector graphics complexes relate to PCS complexes,

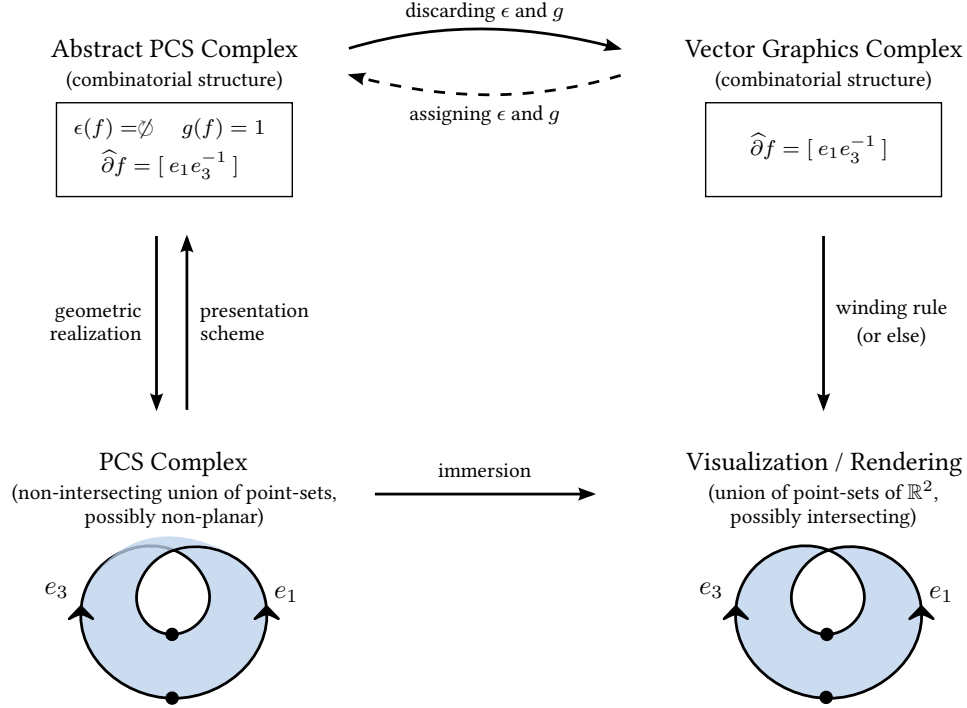


**Figure 3.32:** Examples of topological operators applied to a VGC. Top: Expected behavior. Bottom: Algebraic operations to perform on the face’s cycle in order to implement this behavior.

and in particular, we discuss why the concept of PCS complex is useful to understand and define vector graphics complexes.

Consider the sequence of VGC operations illustrated in Figure 3.32. The initial shape on the left (Fig. 3.32a) can be represented as a single face  $f$ , since edges of vector graphics complexes are allowed to overlap. The face is rendered using the even-odd winding rule, which is the most standard rendering method used in vector graphics systems. Then, a user may decide to glue  $e_2$  with  $e_4$ , and finally to uncut at  $e_5$ , resulting in the shapes depicted in Figure 3.32b and 3.32c. While this behavior is *arguably* a design decision, it is fair to assume that it captures what most users would intuitively expect. For this specific example, based on this expected behavior, it is possible to infer an implementation of glue, uncut, and their inverse operations, such as cut. Importantly, notice that this cut does not disconnect the face! Also, it even reverses the direction of  $e_3$ . This makes it significantly different from a “usual” cut, which should have disconnected the face into two faces (e.g., cutting a disk gives two half-disks). If we assume that  $f$  represents some kind of surface, with any reasonable definition of surface, then such cut only makes sense if we interpret this surface to be a Möbius strip. In fact, the cycle  $\gamma = e_1e_5e_3e_5$  from Figure 3.32b strongly supports this interpretation: it is *the* textbook example of an algebraic Möbius strip. It may not exactly look like a Möbius strip (due to how vector graphics shapes are rendered), and users may not think of it as a Möbius strip, but nonetheless it *behaves* like a Möbius strip, and it has the same *algebraic representation* as a Möbius strip. Therefore, it is in our best interest to interpret  $f$  as a Möbius strip immersed in  $\mathbb{R}^2$ , rather than a non-manifold surface embedded in  $\mathbb{R}^2$ .

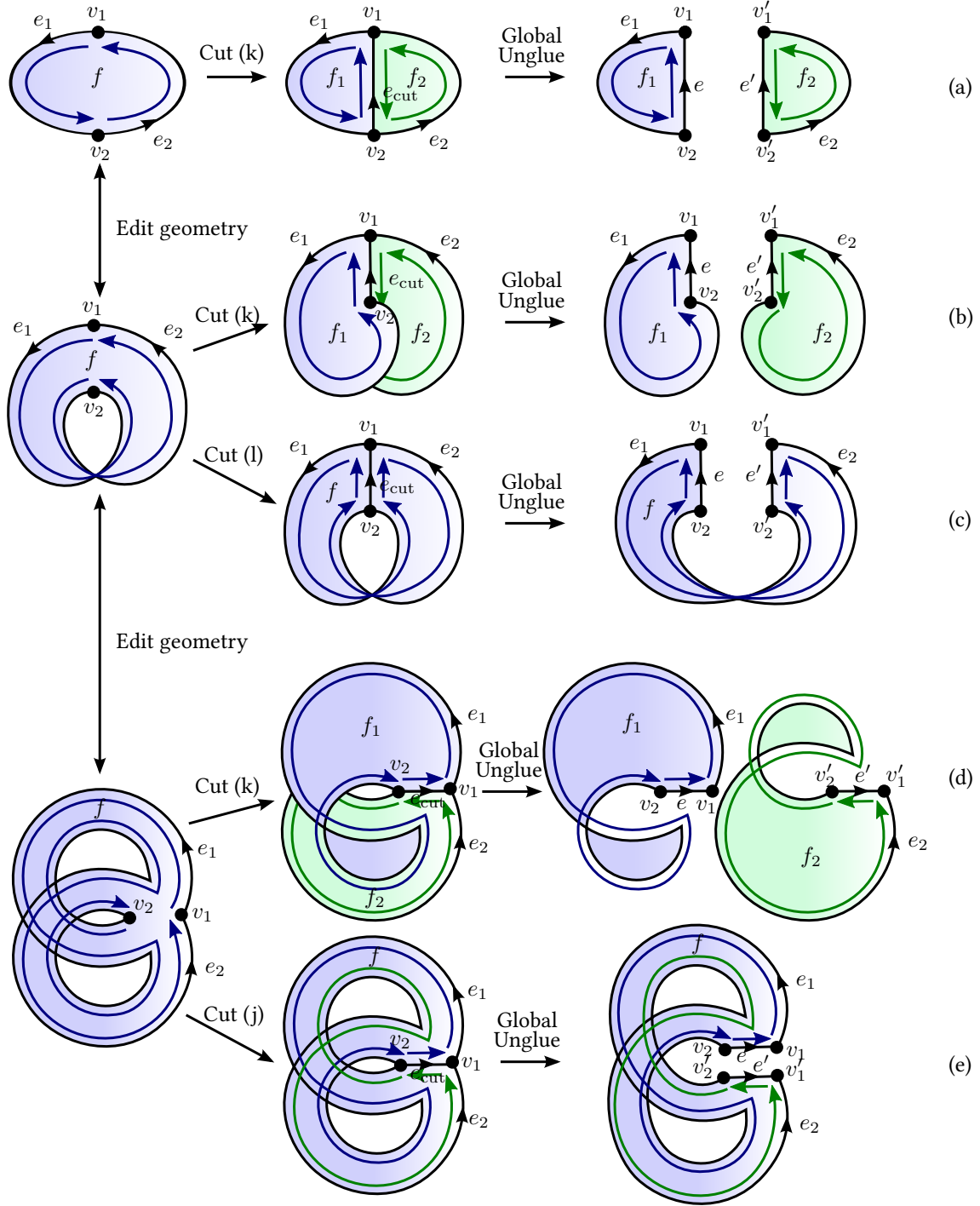
The concept of PCS complex allows us to formalize this interpretation (see Figure 3.33). Indeed, by assigning an orientability and genus to every face, any VGC can be interpreted as an abstract PCS complex  $\mathcal{P} = (C, T)$ , from which can be defined a PCS complex  $\mathcal{K} = |\mathcal{P}| = (X, \mathcal{C}, \mathcal{T})$ , which we recall is a (possibly non-planar) point-set  $X$ , such as a Möbius strip, decomposed into non-intersecting subsets  $c \in \mathcal{C}$  called *cells*. Rendering the VGC, for instance using the even-odd winding rule, can then be interpreted as defining an immersion  $\phi : X \rightarrow \mathbb{R}^2$  from this point-



**Figure 3.33:** Diagram comparing the related concepts of vector graphics complexes and PCS complexes. In particular, notice the two-way mapping between PCS complexes and abstract PCS complexes, which allows to define topological operators in terms of point-sets. By contrast, rendering a VGC in  $\mathbb{R}^2$  using a winding rule (or any other method), is only a one-way mapping.

set  $X$  into the canvas  $\mathbb{R}^2$ , possibly creating intersections between cells (= “overlapping points”). The strength of this formalism is that while topological operators on VGCs may be seen as *design decisions*, topological operators on PCS complexes are perfectly well-defined. Indeed, we have seen that they can be non-ambiguously defined in terms on point-sets, from which we can infer their algebraic counterparts, which is possible due to the one-to-one mapping between PCS complexes and abstract PCS complexes (Fig. 3.33, left). By contrast, due to the existence of overlapping points, rendering a VGC into subsets of  $\mathbb{R}^2$  can only be a one-way mapping no matter which rendering method is used (Fig. 3.33, right), therefore the same approach cannot be used. However, we can now simply define VGC operators in terms of abstract PCS complex operators, since they share the same underlying algebraic structure.

Of course, there is a catch: assigning orientability and genus to VGC faces is ambiguous in the general case, and can only be achieved via imperfect perceptual heuristics, or by asking the user. Therefore, it is still ambiguous *which* PCS topological operator should be applied on any given VGC. But at the very least, studying PCS complexes allowed us to rigorously identify an exhaustive list of topologically meaningful operators to choose from. Also, in practice, the large majority of faces should simply be interpreted as orientable and of genus 0. Among the rare cases which



**Figure 3.34:** Left column: Three different immersions of the same VGC. Middle column: the result of applying a given cut algorithm to the VGC (the letters (k), (l) and (j) refer to Figure 3.25). Right column: the result of applying unglue to all cells and modifying slightly the geometry, for better visualization of the cut. This illustrates that unlike planar maps, cutting a VGC is an ambiguous operation. While there is only one way a planar map face can be cut (i.e., applying Cut (k), cf. row (a)), there are many non-equivalent ways a VGC face can be cut. Choosing the "planar map way" may lead to unexpected results (cf. rows (b) and (d)), in which case choosing an alternative cut algorithm may better capture the user's intent (cf. rows (c) and (e)).



should be interpreted as something else, such as a Möbius strip, there is also almost always an obvious choice. Besides, PCS complexes provide us with an important theoretical understanding of the underlying topological spaces that vector graphics complexes represent, or more precisely, of the family of topological spaces that they *may* represent. In other words, it formally clarifies what we mean by “a VGC can represent any arbitrary non-manifold topology as an immersion in the plane, unlike planar maps which can only represent embeddings”.

An important difference between planar immersions (= vector graphics complexes) and planar embeddings (= planar maps) is that the latter, by definition, can only represent faces that are planar, that is, orientable and of genus 0. This largely simplifies the cut operator. For instance, there is only one way to cut a genus-0 orientable face along an open edge starting and ending at the same hole. This cut is labeled (k) in the face-cut classification provided in Figure 3.25, and is illustrated in Figure 3.34a. However, mistakenly applying this “planar cut” (k) to a face that “looks like” a Möbius strip (Fig. 3.34b), or “looks like” a genus-1 orientable face (Fig. 3.34d) leads to unexpected behaviors. Instead, one should in these cases apply the “non-planar cut” labeled (l) (Fig. 3.34c), or the “non-planar cut” labeled (j) (Fig. 3.34e). Deciding which cut to apply on any given situation is still an open problem, but the formalism of PCS complexes allowed us to discover the exhaustive list of possible cuts to choose from. Note how the three input examples in Figure 3.34 (left column) all have the same topology as vector graphics complexes, but should be interpreted as different PCS complexes (that is, assigning different genus and/or orientability).

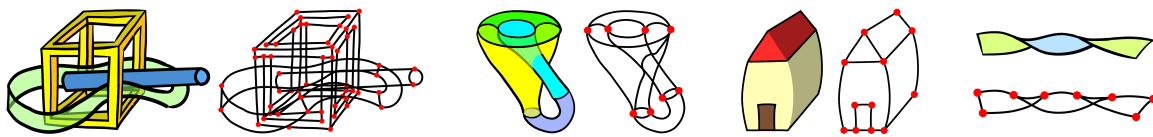
## 3.4 Conclusion

In this chapter, we have introduced important topological concepts to help us understand the mathematical nature of vector graphics objects, which is not as trivial as it may seem. Importantly, as a consequence of allowing vector graphics shapes to overlap and to share edges (two very desirable features), vector graphics topologies are possibly non-planar, or even non-orientable. In fact, even shapes represented as a single vector graphics face may be non planar and/or non-orientable, by allowing users to uncut shapes in predictable ways, where *predictable* means that the behavior only depends on local topological properties. This theoretical knowledge was critical to design and implement the concept of vector graphics complexes, which we detail in the next chapter. It is a combinatorial structure that can be interpreted as a (possibly non-planar) topological space called a PCS complex  $\mathcal{K}$ , in which cells are disjoint point-sets. Rendering a VGC, using for instance winding rules, can then be interpreted as defining an immersion of this underlying space  $\mathcal{K}$  into  $\mathbb{R}^2$ . This interpretation of VGCs as a non-planar point-set immersed in  $\mathbb{R}^2$  allowed us to rigorously define and classify all topological operators that can be applied to VGCs, instead of arbitrarily designing such operators based on intuition.



## Chapter 4

# Vector Graphics Complexes: The Topology of Vector Illustrations



**Figure 4.1:** *Vector graphics illustrations and their underlying topology.*

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.

### 4.1 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

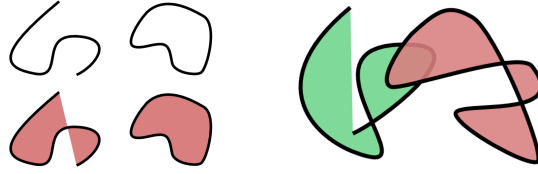
faces. A rich set of features and editing operations can then be integrated into this framework to yield powerful systems for 2D illustration.

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.

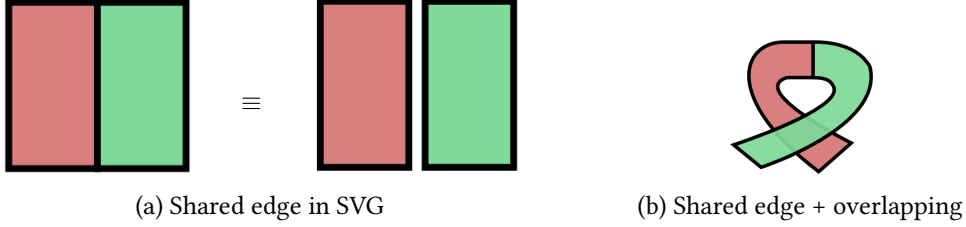
Another important observation is that vector graphics illustrations often consist of 2D depictions of 3D objects [Durand 2002, Eisemann et al. 2009], with the important consequence that a representation of vector graphics objects as strictly two-dimensional entities, such as planar maps, may be counter-productive. 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.



**Figure 4.2:** 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.



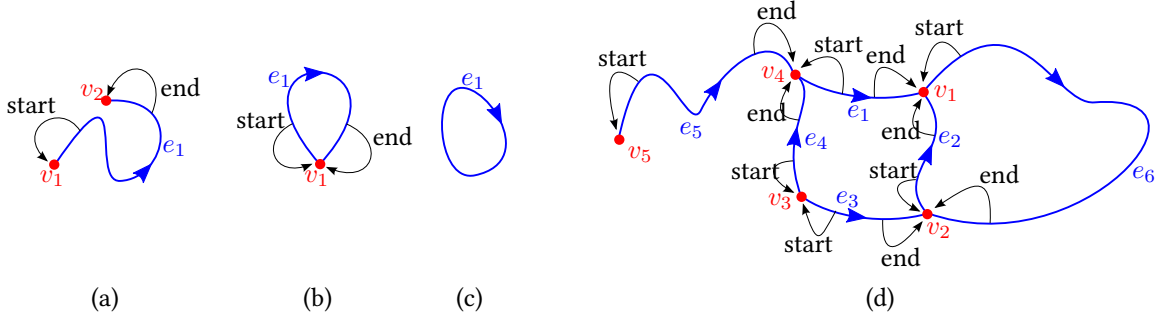
**Figure 4.3:** The limitations of existing representations. SVG cannot represent two faces sharing a common edge as in (a), therefore must duplicate the shared edge. Planar maps can represent shared edges, but cannot represent overlapping faces, thus neither SVG nor planar maps can represent the illustration in (b).

## 4.2 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 more formal definition 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 4.2. 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 4.2.

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 4.3a.

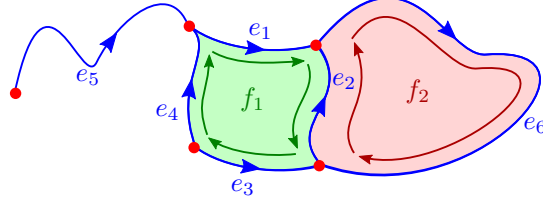


**Figure 4.4:** (a) Example of open edge  $e_1$ , with its two end vertices  $v_1$  and  $v_2$ . (b) Special case of open edge where its two end vertices are equal. (c) Example of closed edge, i.e., an edge with no end vertices at all. (d) A VGC composed of vertices and edges only, similarly to a stroke graph.  $v_1, v_2$  and  $v_4$  are each shared by three edges.

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 alignment features, or the shape builder tool in Illustrator), the topological information between the two shapes is still not explicitly 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 video accompanying [Dalstein et al. 2014b]). Planar maps and their extension, dynamic planar maps (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 4.3a, they cannot faithfully represent the semantics of the illustration shown in Figure 4.3b. This second figure shares the same topology and can therefore be obtained from Figure 4.3a 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 4.3b 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 four types: *vertices*, *open edges*, *closed 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 **open edge** is similar to an SVG open path: it defines a 2D directed open curve on the canvas, for instance using Bézier control points. The significant difference between an open edge and a SVG open path is that an open edge starts at a *start vertex*,

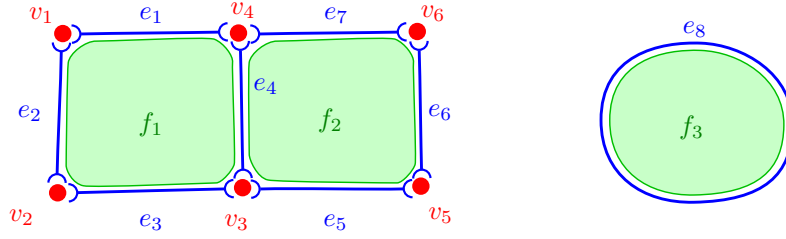


**Figure 4.5:** Two faces  $f_1$  and  $f_2$  each defined by one cycle. The cycle defining  $f_1$  is  $\gamma_1 = [(e_1, \top); (e_2, \perp); (e_3, \perp); (e_4, \top)]$ , while the cycle defining  $f_2$  is  $\gamma_2 = [(e_2, \top); (e_6, \top)]$ . For convenience, we also use the multiplicative notation  $\gamma_1 = e_1 e_2^{-1} e_3^{-1} e_4$  and  $\gamma_2 = e_2 e_6$ .

and ends at an *end vertex*, and these vertices can be shared with other edges. On the contrary, all the control points of a given SVG path, including the first and last control points, belong to the path and cannot be shared with other paths. In practice, the end vertices of open edges are stored as pointers (see Figure 4.4a). As illustrated in Figure 4.4d, 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, our definition 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 to the concept of open edge, we define the concept of **closed edge**, which is a 2D directed closed curve no end vertices at all, as illustrated in Figure 4.4c. We note that it is allowed for an open edge to have its start vertex be equal to its end vertex (see Figure 4.4b). On the contrary to some existing representations, we consider this to be a special case of open edge, i.e., we *do not* call this 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 **cycle**. Typically, one of the cycle of the face represents its outer boundary, while the other cycles represent inner holes. Like in SVG, we do not explicitly store in our data structure whether a cycle represents an outer boundary or an inner hole, since users can freely edit their geometry, which could change these perceptual roles, or make them ill-defined. Typically, a cycle is defined as a sequence of  $(e, \beta)$  pairs that we call **halfedges**, where  $e$  is an open edge and  $\beta$  is a boolean indicating whether the edge should be considered with its intrinsic direction (from start to end,  $\beta = \top = \text{True}$ ), or with the opposite direction (from end to start,  $\beta = \perp = \text{False}$ ), as illustrated in Figure 4.5. However, a cycle can also be defined as a unique directed closed edge. Finally, a cycle can also be defined as a unique vertex, and we call these cycles **Steiner cycles**. 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



**Figure 4.6:** Two incident squares and a disk represented as a single VGC with 17 cells: six vertices  $v_1$  to  $v_6$ , seven open edges  $e_1$  to  $e_7$ , one closed edge  $e_8$ , and three faces  $f_1$  to  $f_3$ .

ignore the intersection and create overlapping edges, not topologically connected. To create faces, the artist uses a paint bucket tool, which automatically compute cycles defining closed regions of the canvas bounded by edges. 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 video accompanying [Dalstein et al. 2014b] 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 4.3b.

## 4.3 Vector Graphics Complex

In this section, we provide a formal definition of the concept of vector graphics complex. This definition is inspired from concepts of algebraic topology (see Chapter 3), but the formalism itself is more typical of graph theory. In fact, we show in Section 4.3.3 how the definition can be interpreted as a colored graph. Readers who prefer a more practical definition can also jump directly to Section 4.3.4 where we give a C++ implementation with invariants.

### 4.3.1 Topology

First, let us provide a purely combinatorial definition. In Section 4.3.2, we subsequently assign geometric attributes to each cell, in order to immerse this combinatorial structure in  $\mathbb{R}^2$ .

A **vector graphics complex** is an ordered pair  $\mathcal{P} = (C, \text{dim}, \text{isClosed}, k, \widehat{\partial})$ , such that:

- $C$  is a finite set of symbols called **cells**. We illustrate in Figure 4.6 what cells *represent*.
- $\text{dim} : C \rightarrow \{0, 1, 2\}$  is a function that assigns a **dimension** to each cell. This defines a partition of  $C$  into three sets  $V$ ,  $E$ , and  $F$  of elements respectively called **vertices**, **edges**, and **faces**.

- $\text{isClosed} : E \rightarrow \{\top, \perp\}$  is a function that assigns a **closedness** to each edge ( $\top$  and  $\perp$  are the two symbols we use to denote the booleans “True” and “False”). This defines a partition of  $E$  into two sets  $E_{\circ}$  and  $E_{|}$  of elements respectively called **closed edges** and **open edges**.
- $k : F \rightarrow \mathbb{N}$  is a function that assigns a **number of cycles** to each face.
- $\hat{\partial}$  is a function that assigns an **ordered boundary** to each cell. This ordered boundary, which we detail below, is what defines the incidence relationship between cells.
- For each  $v \in V$ , we have  $\hat{\partial}v = \emptyset$ .
- For each  $e \in E_{\circ}$ , we have  $\hat{\partial}e = \emptyset$ .
- For each  $e \in E_{|}$ , we have  $\hat{\partial}e \in V \times V$ . We define  $v_{\text{start}}(e)$  and  $v_{\text{end}}(e)$  to be the first and second element of the ordered pair.
- For each  $f \in F$ , we have  $\hat{\partial}f \in \Gamma^{k(f)}$ . In other words,  $\hat{\partial}f$  is an ordered sequence of  $k(f)$  cycles  $\gamma_i \in \Gamma$ , where  $\Gamma$  is the set of all possible cycles on  $\mathcal{P}$ , which we define below, after first defining the concepts of open and closed halfedges.
- A **closed halfedge**  $h = (e, \beta)$  is defined as a pair of a closed edge  $e \in E_{\circ}$  and a direction  $\beta \in \{\top, \perp\}$ .
- An **open halfedge**  $h = (e, \beta)$  is defined as a pair of an open edge  $e \in E_{|}$  and a direction  $\beta \in \{\top, \perp\}$ . For each open halfedge  $h$ , we define  $v_{\text{start}}(h)$  and  $v_{\text{end}}(h)$  as follows:

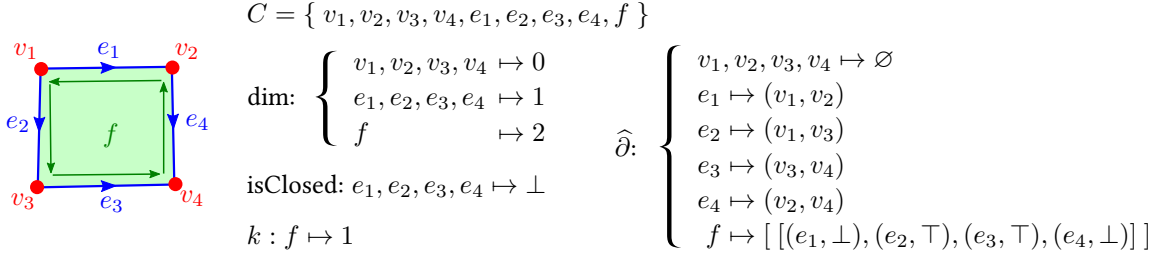
$$v_{\text{start}}(h) = \begin{cases} v_{\text{start}}(e), & \text{if } \beta = \top \\ v_{\text{end}}(e), & \text{if } \beta = \perp \end{cases} \quad \text{and} \quad v_{\text{end}}(h) = \begin{cases} v_{\text{end}}(e), & \text{if } \beta = \top \\ v_{\text{start}}(e), & \text{if } \beta = \perp \end{cases} \quad (4.1)$$

- Finally, a **cycle**  $\gamma \in \Gamma$  is defined as either:
  1. a vertex  $v \in V$ , or
  2. a pair  $(h, N)$  consisting of a closed halfedge  $h$  and an integer  $N > 0$ , or
  3. a non-empty, ordered sequence  $(h_j)_{j \in [1..N]}$  of open halfedges such that:

$$\forall j \in [1..N], \quad v_{\text{end}}(h_j) = v_{\text{start}}(h_{(j+1) \bmod N}) \quad (4.2)$$

These three types of cycles are respectively called **Steiner cycles**, **simple cycles**, and **non-simple cycles**.

We illustrate this definition with a simple example in Figure 4.7. Since the concept of vertices and edges should be fairly self-explanatory, let us simply give more clarifications on the concept of



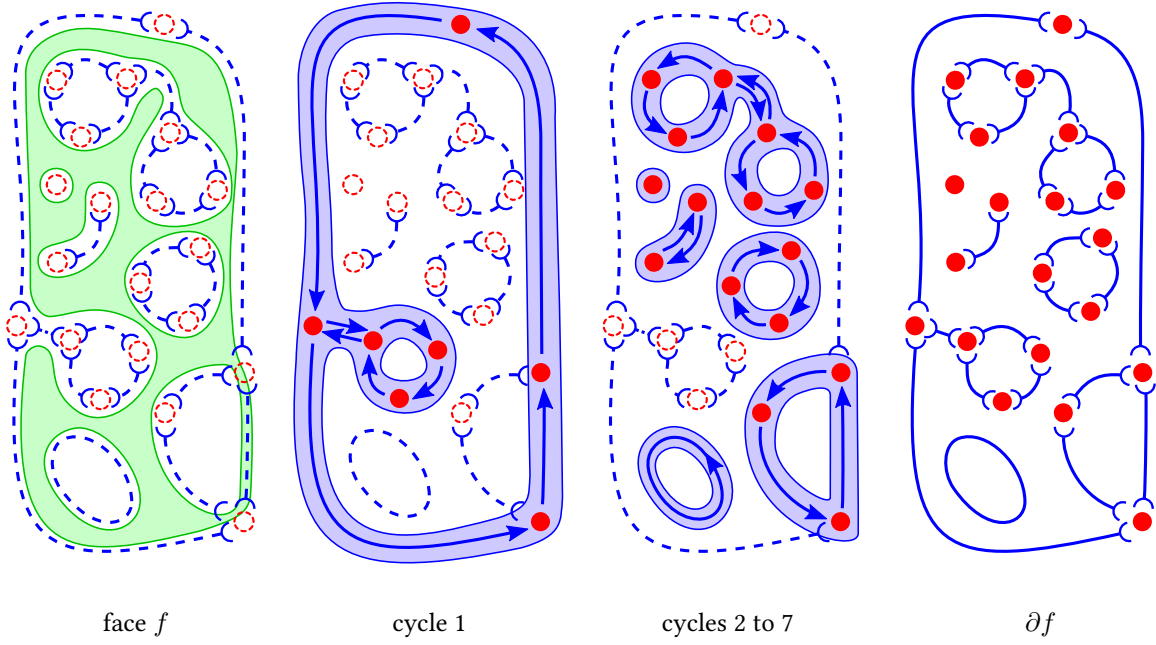
**Figure 4.7:** Combinatorial representation of a square as a VGC with 4 vertices, 4 open edges, and 1 face.

faces and cycles. As one can notice, the *number of cycles*  $k(f)$  of any given face  $f$  can be any integer in  $\mathbb{N}$ . In particular, the definition allows  $k(f) = 0$ , which corresponds to a face without cycles. Such faces are not particularly useful in vector graphics (they cannot be rendered in any meaningful way), but they have theoretical relevance (see Chapter 3 for details), and it is sometimes useful to allow their representation as a transient state occurring in the middle of an operation.

Also, note that cycles are allowed to repeat any edge any number of times, including three times or more. Not only such cycles have theoretical relevance, but they are actually useful in practice (see Figure 4.19, middle and right). This corresponds to the typical “fan edge” non-manifold configuration, but with the less typical flavor that all the “fans” actually belong to the same face and the same cycle. Perhaps more surprisingly, we also allow simple cycles (i.e., cycles defined via a closed edge) to repeat their closed edge any number of times (though, the chosen direction must be fixed). This number of repetitions is given by the integer  $N > 0$  in the definition. Similarly to faces without cycles, simple cycles with repeated closed edge correspond to configurations which are theoretically meaningful (and may be the results of valid topological operators), but are not very useful in practice. Though, one notable example is a Möbius strip whose centerline is represented as a closed edge. Both sides of the centerline belong to the same face and the same cycle, which is a simple cycle repeating the closed edge two times. Another interesting example is to take three long strips of paper, and glue them all together along one of their long edge (this gives a long open edge shared by three faces). Then, if you glue the two star-shaped ends of this construction after applying a third-twist, the three faces become one, and the center open edge becomes closed. One of the cycle of the unique face is a simple cycle repeating this closed edge three times.

Finally, cycles are not necessarily disjoint, whether they are from the same face or different faces. In other words, they can freely share common vertices or edges, with no limitations, as we illustrate in Figure 4.8. In other words, the validity of a cycle does not depend on any other cycle. In fact, two different faces may even have equal cycles. This represents two faces with exactly the same shape, stacked on top of one another, with their boundary glued.





**Figure 4.8:** 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).

### 4.3.2 Geometry

Now that we have defined the combinatorial structure of a VGC, let us define how we can immerse each cell  $c \in C$  as a pointset  $|c| \subseteq \mathbb{R}^2$ , by assigning them geometric attributes.

**Vertex** Each vertex  $v$  is assigned a point  $p(v) \in \mathbb{R}^2$ . We define  $|v| = \{p(v)\}$ , that is, the subset of  $\mathbb{R}^2$  reduced to the point  $p(v)$ .

**Open Edge** Each open edge  $e$  is assigned a continuous curve  $\Gamma(e) : [0, 1] \rightarrow \mathbb{R}^2$ , satisfying  $\Gamma(e)(0) = p(v_{\text{start}}(e))$  and  $\Gamma(e)(1) = p(v_{\text{end}}(e))$ . We define  $|e| = \Gamma(e)((0, 1))$ , that is, the image of the open interval  $(0, 1)$  by  $\Gamma(e)$ .

**Closed Edge** Each closed edge  $e$  is assigned a continuous closed curve  $\Gamma(e) : \mathbb{S}^1 \rightarrow \mathbb{R}^2$ , where  $\mathbb{S}^1$  is the unit circle. We define  $|e| = \Gamma(e)(\mathbb{S}^1)$ .

**Face** Each face  $f$  is assigned a **winding rule**  $R(f) \subseteq \mathbb{Z}$ . This winding rule allows us to define the pointset  $|f| \subseteq \mathbb{R}^2$  as follows. First, for each simple or non-simple cycle  $\gamma$ , we define the closed curve  $\Gamma(\gamma) : \mathbb{S}^1 \rightarrow \mathbb{R}^2$  by concatenating together the curves  $\Gamma(e)$  for all the edges  $e$  involved in the cycle, and we define  $|\gamma| = \Gamma(\gamma)(\mathbb{S}^1)$ . If  $\gamma$  is a Steiner cycle  $\gamma = v$ , we simply define  $|\gamma| = |v|$ . This allows us to define  $|\partial f| = \bigcup_{\gamma \in \partial f} |\gamma|$ , which is the immersion in  $\mathbb{R}^2$  of the boundary of the face.

Next, for each cycle  $\gamma$ , and for each point  $p \in \mathbb{R}^2 \setminus |\gamma|$ , we denote by  $N(\gamma)(p) \in \mathbb{N}$  the winding number of  $\Gamma(\gamma)$  at  $p$  (cf. [Edelsbrunner and Harer 2010, p12]). Now that we have defined per-cycle winding numbers  $N(\gamma)(p)$ , we can define per-face winding numbers  $N(f)(p)$  by summing the  $N(\gamma)(p)$  for all cycles  $\gamma$  of the given face  $f$ , that is:

$$\forall f \in F, \forall p \in \mathbb{R}^2 \setminus |\widehat{\partial}f|, N(f)(p) = \sum_{\gamma \in \widehat{\partial}f} N(\gamma)(p). \quad (4.3)$$

With all these prerequisites,  $|f|$  can finally be defined as:

$$\forall f \in F, |f| = \{ p \in \mathbb{R}^2 \setminus |\widehat{\partial}f| \text{ s.t. } N(f)(p) \in R(f) \}. \quad (4.4)$$

Currently, our implementation uses the OpenGL GLU polygon tessellator [Shreiner et al. 2004] to compute and render  $|f|$ , using the even-odd winding rule, that is,  $R(f) = 2\mathbb{Z} + 1$  for all faces. Other rules could be considered as well, and users could choose different rules for different faces, as is already common in many vector graphics applications (e.g., Inkscape).

### 4.3.3 Vector Graphics Complexes as Colored Incidence Graphs

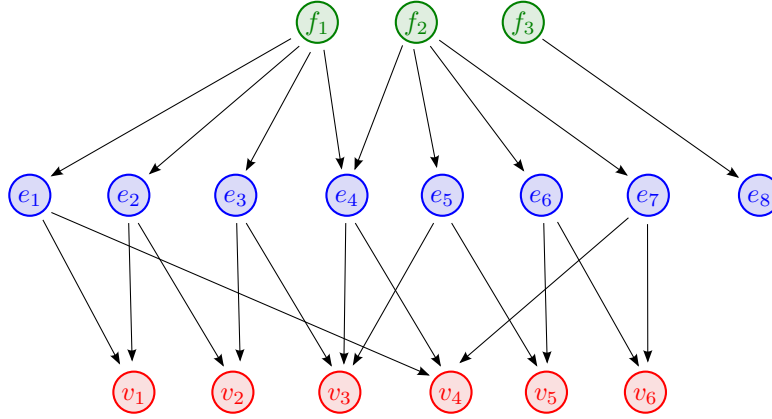
In this section, we show how the concept of vector graphics complex, which we have just defined, can be interpreted as a colored graph. More specifically, it can be interpreted as a coloration of the *incidence graph* of the cell complex.

#### Incidence Graph

Many readers may already be familiar with the concept of incidence graph. However, let us take the time to formally define it in the case of vector graphics complexes. The **incidence graph** of a given vector graphics complex  $\mathcal{P} = (C, \text{dim}, \text{isClosed}, k, \widehat{\partial})$  is the directed graph  $\mathcal{G} = (C, A)$  where the nodes are the cells  $c \in C$ , and where there is a directed arc<sup>12</sup>  $a \in A$  from  $c$  to  $c'$  if and only if  $c'$  is in the boundary of  $c$ .

Though, we have yet to define what “ $c'$  is in the boundary of  $c$ ” actually means. Indeed, all we have formally defined so far is the concept of *ordered boundary*  $\widehat{\partial}c$ , which isn’t a subset of  $C$  but some more complex ordered structure (e.g., an ordered pair of vertices for open edges, or an ordered sequence of cycles for faces). Now, we define  $\partial c$  as the subset of  $C$  consisting of “all cells involved in  $\widehat{\partial}c$ ”, that is, we forget the order. More precisely, we define  $\partial v = \partial e = \emptyset$  for vertices and closed edges, we define  $\partial e = \{v_{\text{start}}(e), v_{\text{end}}(e)\}$  for open edges (reduced to a unique element when

<sup>12</sup>We use the terminology “arc” instead of the more usual “edge” to avoid confusion with VGC edges, which are actually nodes of the graph.



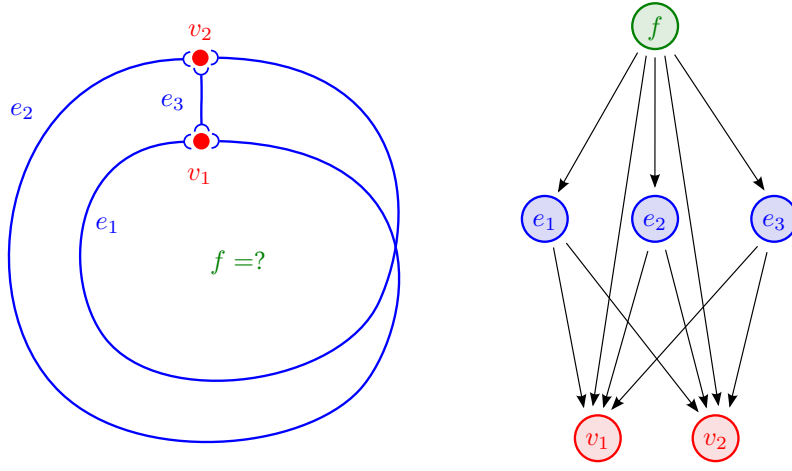
**Figure 4.9:** The incidence graph of the VGC illustrated in Figure 4.6. For clarity, we do not show in this figure the arcs from faces to vertices, since they can be inferred by transitivity.

$v_{\text{start}}(e) = v_{\text{end}}(e)$ , and in the case of a face  $f$ , we define  $\partial f$  to be the union of all its Steiner vertices, all the closed edges of its simple cycles, and all the open edges and their end vertices of its non-simple cycles. With this definition, we can now safely define “ $c'$  is in the boundary of  $c$ ” as meaning  $c' \in \partial c$ .

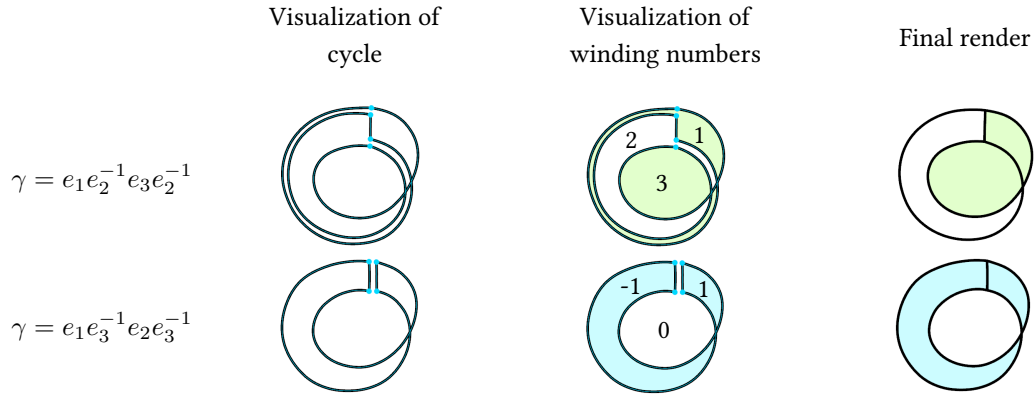
It can be easily shown that the relation “ $c' \in \partial c$ ” is transitive and irreflexive, thus defines a strict partial order, which in turns means that  $\mathcal{G}$  is a transitive directed acyclic graph. By assigning the dimension of each cell  $c$  as a color of each node of the graph, we obtain a graph such as illustrated in Figure 4.9.

### Non-Sufficiency of the Incidence Graph

One may wonder whether this incidence graph alone encodes all the topological information of the vector graphics complex. The answer is no. Fundamentally, the incidence graph only encodes  $\partial c$ , and the conversion from  $\widehat{\partial c}$  to  $\partial c$  causes information loss (i.e., it is not invertible). In more practical terms, the edges in the boundary of a face need to be organized into cycles if we want to render the face using winding numbers, but the automatic computation of cycles from a set of edges is in general ambiguous. We illustrate this ambiguity in Figure 4.10 and 4.11 with the example of a Möbius strip. In this example, the (non-ordered) boundary of the face  $f$  is  $\partial f = \{ e_1, e_2, e_3 \}$ , and it is impossible to organize these three edges in one cycle without repeating at least one of its edges. For instance, starting at  $v_1$ , one can go along  $e_1$ , then  $e_2$ , then  $e_3$ , then  $e_2$  again back to  $v_1$ . Alternatively, still starting at  $v_1$ , one can go along  $e_1$ , then  $e_3$ , then  $e_2$ , then  $e_3$  again back to  $v_1$ . These two different cycles result in different winding numbers, thus in different final renders. The explicit cycle ordering provided in  $\widehat{\partial c}$  allows disambiguation (potentially authored by artists), and ensures consistent rendering across implementations.



**Figure 4.10:** The incidence graph of a Möbius strip. From this incidence graph alone, it is unclear how the face  $f$  should be rendered.

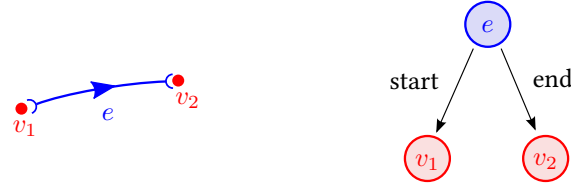


**Figure 4.11:** Illustration of two possible choices of cycles for the incidence graph of Figure 4.10. We assume that  $v_1$  is the start vertex of all three edges, and that  $v_2$  is their end vertex. These two cycles repeat a different edge, which results in different winding numbers, and therefore different final renders. We use here the even-odd winding rule, but using the non-zero winding rule would also result in different renders.

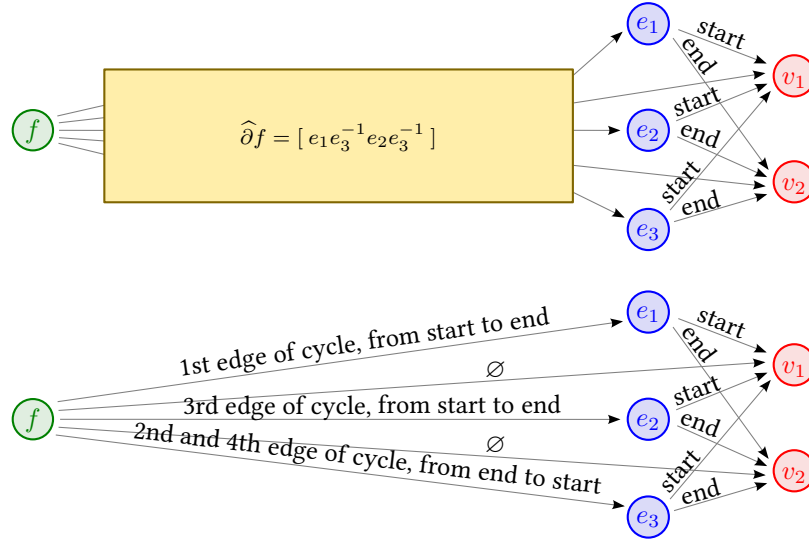
### Coloring the Incidence Graph with Topological Order

In the previous paragraph, we have seen that the incidence graph of a given vector graphics complex does not encode all its topological information. We have also seen that the additional information not encoded in the incidence graph corresponds to an “ordering” of the boundary of faces into cycles of edges.

One can imagine encoding this ordering into the incidence graph by “coloring” (=annotating) its



**Figure 4.12:** The ordered boundary  $\hat{\partial}e = (v_{\text{start}}, v_{\text{end}})$  of an open edge  $e$  can be encoded in the incidence graph by coloring the arcs of the graph, thus defining a topological direction for the edge.



**Figure 4.13:** Illustration of how the ordered boundary  $\hat{\partial}f$  of each face  $f$  can be encoded in the arcs of the incidence graph. This example corresponds to coloring the incidence graph from Figure 4.10.

arcs with additional information. For instance, the arcs between an open edge and its end vertices might be annotated “start” or “end” (or “both”), in order to indicate which vertex is the start vertex of the edge, and which is the end vertex (see Figure 4.12). The same idea can be applied to annotate the boundary of faces. It is a bit less practical, but with some care, it is possible to define a bijective function to map the information contained in  $\hat{\partial}f$  into annotation of the outgoing arcs of  $f$  in the incidence graph (see Figure 4.13). Therefore, one can see that the concept of vector graphics complex can be interpreted as an incidence graph *augmented* with order information, and that this order is required to disambiguate winding numbers.

More fundamentally, we see in Chapter 3 that the order is critical to determine the homeomorphism class of the underlying topological space, which is required to know which topological operators can be applied to the complex.

### 4.3.4 Implementation

Below, we give one possible C++ implementation of the VGC data structure:

```
1 class Cell
2 {
3     std::set<Cell*> star;
4 };
5
6 class Vertex: public Cell
7 {
8     Point p;
9 };
10
11 class Edge: public Cell
12 {
13     Vertex * start;
14     Vertex * end;
15     Curve curve;
16 };
17
18 class Halfedge
19 {
20     Edge * edge;
21     bool b;
22 };
23
24 class Cycle
25 {
26     Vertex * steiner;
27     std::vector<Halfedge> halfedges;
28 };
29
30 class Face: public Cell
31 {
32     std::vector<Cycle> cycles;
33 };
```

Of course, the above code is only an implementation of how to *store* the data, and most of the hard work is implementing how to *edit* this data via topological operators (see Section 4.4). However, the code illustrates that the *structure* itself is quite straightforward and intuitive. Though, let us provide some details and clarifications in the next few paragraphs.

As detailed in Section 4.3.1 and 4.3.2, a vertex is simply a 2D position, an edge is a 2D directed curve pointing to its start and end vertex (if any), and a face is a 2D region of the plane delimited by cycles (where a cycle is either a sequence of consecutive halfedges, or a single vertex).

Checking whether an edge  $e$  is closed or open is simply done by checking whether  $e \rightarrow \text{start}$  is NULL or not. Indeed, closed edges do not have end vertices, and therefore  $e \rightarrow \text{start}$  and  $e \rightarrow \text{end}$  are both NULL for closed edges. Conversely, open edges always have a valid start vertex and end vertex, and therefore  $e \rightarrow \text{start}$  and  $e \rightarrow \text{end}$  are both non-NULL for open edges.

Similarly, checking whether a cycle  $\text{cycle}$  is a Steiner cycle or not is done by checking whether  $\text{cycle} \rightarrow \text{steiner}$  is NULL or not. Equivalently, one may check whether  $\text{cycle} \rightarrow \text{halfedges}$  is empty or not, since  $\text{cycle} \rightarrow \text{halfedges}$  is non-empty if and only if  $\text{cycle} \rightarrow \text{steiner}$  is NULL.

Note how in this implementation, the classes `Vertex`, `Edge`, and `Face` all inherit the class `Cell`, while `Halfedge` and `Cycle` do not. This is because halfedges and cycles are only *helper classes* to define faces, but are not cells themselves. More specifically, halfedges and cycles are just convenient container classes (you can see them as ad-hoc `std::pair` and `std::vector`) to store the incidence relationships between cells. In particular, note how the classes `Cell`, `Vertex`, `Edge`, and `Face` are used with *pointer semantics*, while the classes `Halfedge` and `Cycle` are used with *value semantics*. This is because cells have an *identity*: it is important for a cell to be able to *refer* to other cells (e.g.: “Who is my start vertex?”). On the other hand, halfedges and cycles do not need to have an identity (though, for various reasons, a particular implementation may *choose* to give them an identity, similarly to the concept of *uses* in the radial-edge data structure).

The **star** of a cell  $c$  is defined as  $\text{star}(c) = \{ c' \mid c \in \partial c' \}$ , that is, as the set of cells  $c'$  whose boundary contains  $c$ . Strictly speaking, this is redundant topological information (reason why it is not in the theoretical definition of the VGC), but in practice, we need to store the star of each cell for obvious performance reasons. Indeed, not doing so would make most adjacency queries or topological operators have a linear-time complexity instead of constant-time. Storing the star of the cells is the equivalent of storing back-pointers to parent nodes in a tree data structure. Note that we do not store this star in any particular order, which we emphasized by using a `std::set` (in practice, it may be more efficient to use a `std::vector`). If we ever need to have more information about the incident relationship between a cell  $c$  and one of its star cell  $c'$  (e.g., is the vertex a Steiner vertex?), one can simply traverse the boundary of  $c'$  and inspect how  $c$  is *used* by  $c'$  (e.g., as a Steiner cycle, or as the end vertex of an open edge of a non-simple cycle?).

We did not describe 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, in our implementation we added vertex radius, variable edge width, cell color (possibly transparent), and edge junctions style (miter join or bevel join).

To ensure that the data structure is topologically valid, we ensure that all public methods (e.g., topological operators, see Section 4.4) preserve the following invariants:

- 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  $i \in \{1, \dots, n-1\}$ ,  
`halfedges[i] == halfedges[0]`
    - \* `halfedges` only contains valid open halfedges, and for all  $i \in \{0, \dots, n-1\}$ ,  
`halfedges[i].end() == halfedges[(i+1) % n].start()`
 (where `Halfedge::start()` returns `(b ? edge->start : edge->end)`, and  
`Halfedge::end()` returns `(b ? edge->end : edge->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.

## 4.4 Topological Operators

Since topology lies at the heart of the vector graphics complex, and that we aim at porting topological modeling into 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



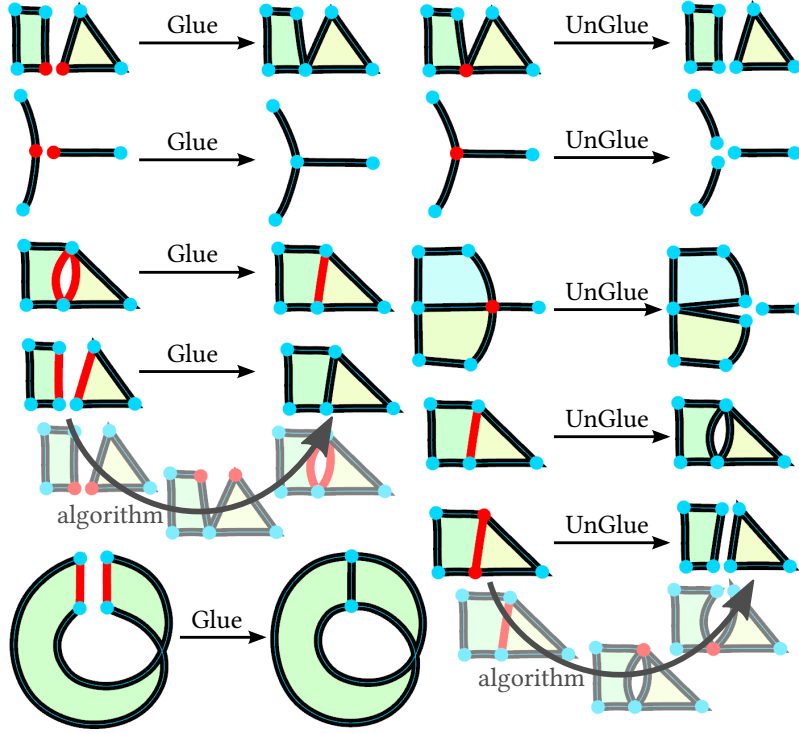
number of connected components and the number of holes. Designing an Euler formula in the case of our non-manifold, non-orientable, mixed-dimensional objects is likely possible and definitely interesting, but it is far from trivial and rather irrelevant as a safety check. Instead, one just has to ensure that the *invariants* which we detailed in the previous section are preserved. In this section, we informally present the reversible operators create/delete, glue/unglue and cut/uncut, which all preserve these invariants and are analyzed in more details in [Dalstein et al. 2014a].

These topological operators 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 4.1. In fact, we show in Chapter 3 that the VGC can represent any regular non-manifold two-dimensional topological space.

#### 4.4.1 Creation and Deletion Operators

Creating a vertex or a closed edge does not require special care. Creating an open edge requires referring to existing start and end vertices. These topological operators are automatically invoked when the user draws a stroke, as detailed in Section 4.6. Creating a face requires as input its list of valid cycles, and robustly obtaining these valid cycles from intuitive user input is still an open problem. Currently, the user can either manually select a set of edges which is automatically converted to cycles, or use a “paint bucket” tool which tries to find appropriate cycles from surrounding edges. Unfortunately, both of these techniques are in general ambiguous due to potential overlapping between edges. To achieve faces such as a Möbius strip, one option for the user is to explicitly draw the repeated edge twice (as in Figure 4.11, left), use the paint bucket, then glue together the two repeated edges.

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 is 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

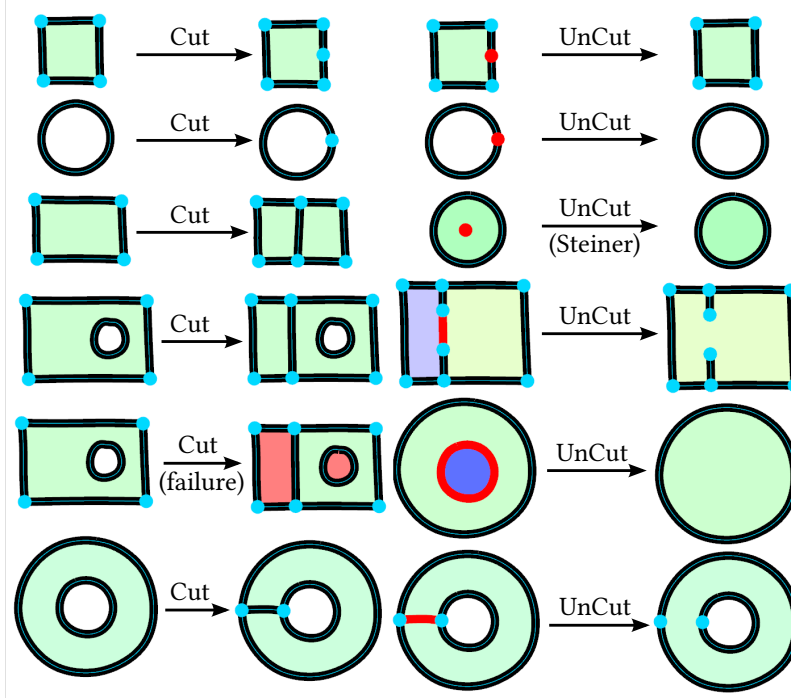


**Figure 4.14:** Examples of glue and unglue operations on vertices, edges, and a set of cells (bottom-right).

are three or more incident edges. The semantics of our “smart delete” are thus defined by “uncut if possible; otherwise, hard delete”.

#### 4.4.2 Glue and Unglue Operators

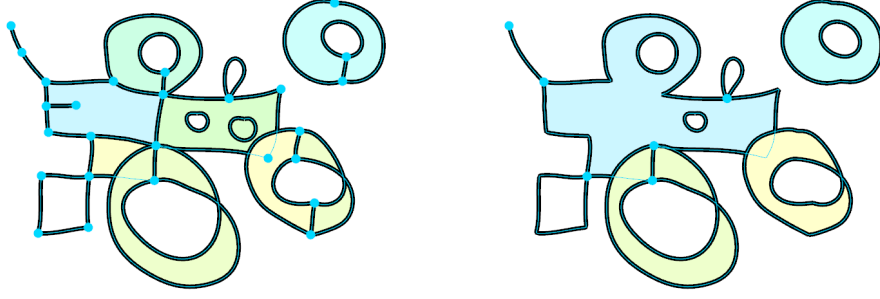
The glue operator on vertices and edges, as well as the unglue operator on vertex, edge, or set of vertices and edges are illustrated in Figure 4.14. Gluing two vertices is the equivalent of the operation *join* in existing vector graphics software, where two paths are appended by gluing together two selected path end-nodes. However, with such a classical *join* operation, the selected end-nodes are transformed into a middle Bézier control point that cannot be *joined* 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. More specifically, there are always two ways to glue two given edges: one has to specify their relative *relative direction*. Our implementation uses a simple heuristic to predict the most relevant direction 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.



**Figure 4.15:** 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.

#### 4.4.3 Cut and Uncut Operators

The results of the cut and uncut operators are illustrated in Figure 4.15. 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  $\Gamma$  starting and ending on  $\partial f$ , it cuts the face into two faces  $f_1$  and  $f_2$ , separated by a new edge  $e$  whose geometry is  $\Gamma$ . This may involve cutting first  $\partial f$  to create the end vertices of  $e$  if they do not already exist. Alternatively, if  $\Gamma$  starts and ends at two different cycles of  $f$  (cf. Figure 4.15, 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  $(e, \text{true})$  and  $(e, \text{false})$ . 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, as we detail in Chapter 3. A simple example is given in Figure 4.15 (failure case): if  $f$  contains 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



**Figure 4.16:** The effect of a global “uncut”. Left: Original VGC. Right: VGC resulting from applying “select all” and then “uncut”.

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. In Appendix E, we refer to this operation as *atomic simplification* and study it in more details. 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* (concept that we define in Appendix E) is a manifold space. 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 theoretically simpler than cutting: it is a bit tedious to implement to handle all possible cases, but 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 4.16: performing this operation on the whole VGC is equivalent to the simplification operation described in [Rossignac and O’Connor 1989]. We conjecture in Appendix E that it results in a unique minimal decomposition.

## 4.5 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

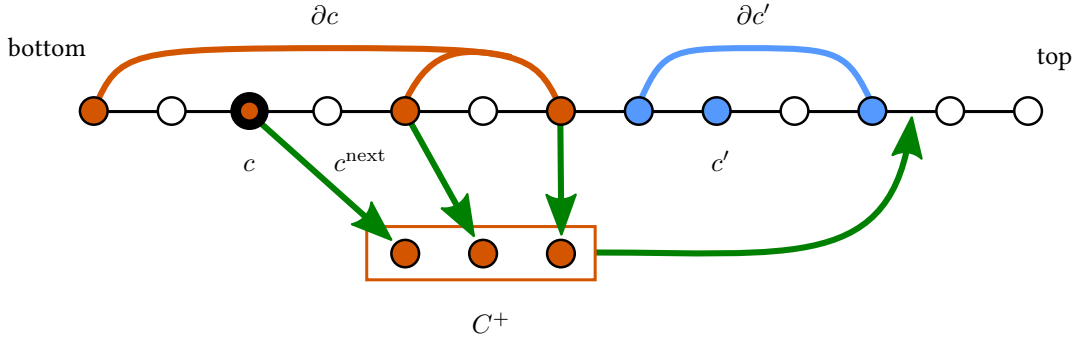


Figure 4.17: Raising a cell.

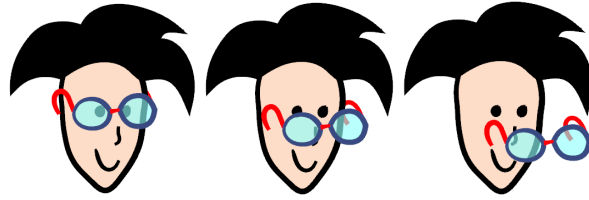


Figure 4.18: An example of the flexible occlusion interactions enabled by the VGC.

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  $\partial c$ .

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,  $c^{\text{next}}$ , as depicted in Figure 4.17. 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 the algorithm *Raise* below, and illustrated in Figure 4.17. The *lower* operation is the counterpart to *raise* and is implemented in a largely symmetric fashion, where the directions are reversed and  $\partial c$  is replaced by  $\text{star}(c)$ .

---

**Raise** (selected cell  $c$ )

---

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

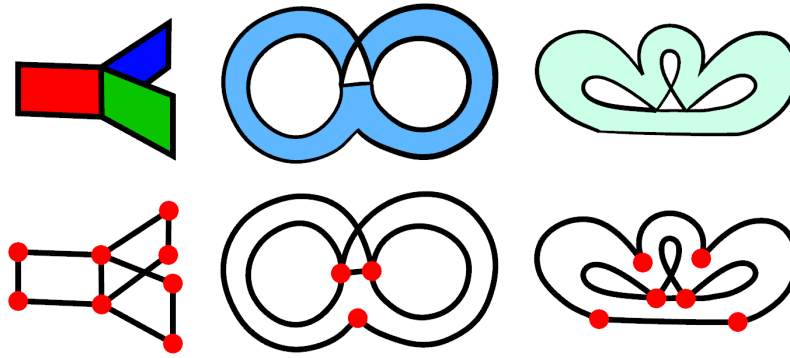
The ability to manipulate depth orderings for components within objects and between objects allows for partial orderings such as that shown in Figure 4.18. One arm of the glasses is stacked so as to be behind the face while the other remains in front, along with the rest of the frame. More examples involving depth manipulations are shown in Figure 4.1 and in the video accompanying [Dalstein et al. 2014b].

## 4.6 User Interface

Many aspects of the user interface can be readily understood from the video accompanying [Dalstein et al. 2014b]. 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  $\epsilon$ . Self-intersection junctions can also snap to each other, thereby allowing multiple approximately collocated self-intersections to automatically coalesce into a single junction.



**Figure 4.19:** 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)

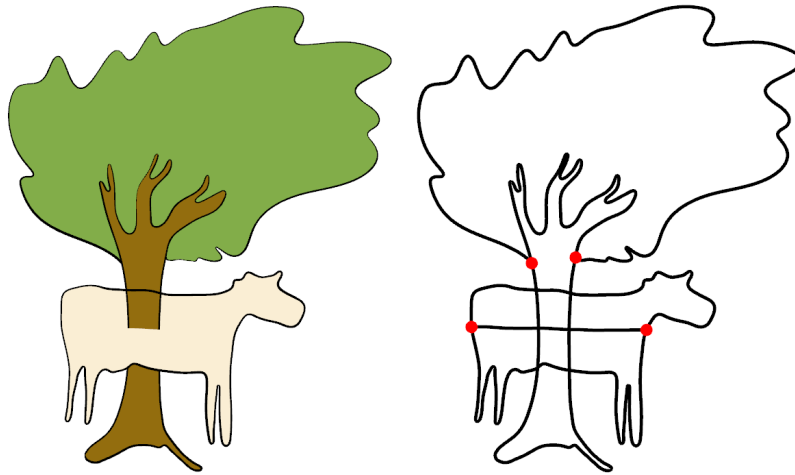
**Creating faces and holes** Faces and holes can be created in multiple ways, as demonstrated in the video. The simplest way is to use a “paint bucket” tool that attempts to infer cycles of edges enclosing the current mouse position. Alternatively, users can create faces by manually selecting the edges that will serve as a boundary, which is sometimes useful to resolve ambiguities that may arise due to overlapping edges. Multiple holes can be added to a face, with the resulting fill determined by their winding number, as shown in Figure 4.19.

**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.

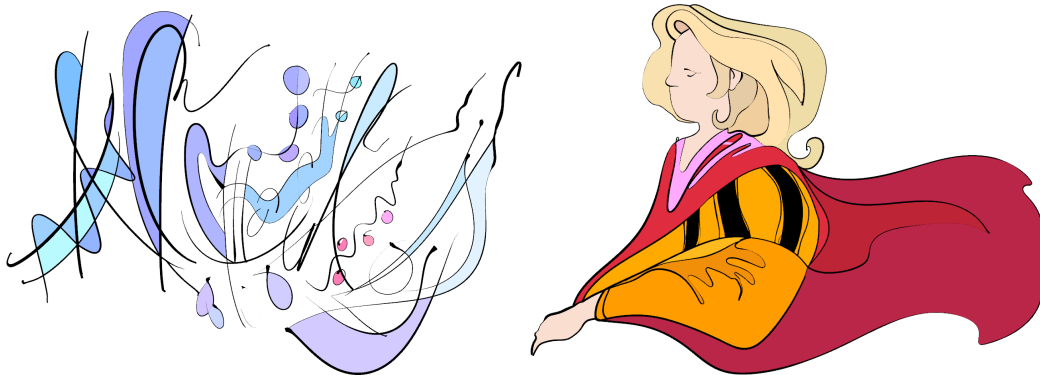
**Performance** The performance of our implementation is currently limited by the naive dynamic tessellation that we perform for each render. Edges are rendered according to their width attribute through generation of quadrilaterals that are centered around the polyline that represents the edge path,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

## 4.7 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 VGC by non-technical users.



**Figure 4.20:** A user experimenting with the possibilities offered by invisible cuts and depth ordering.



**Figure 4.21:** The VGC can be used for abstract art or stylized figurative art. Examples drawn by Etienne Colas.

Users consistently report that using the VGC is significantly different from current SVG tools, and that it opens exciting new creative 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 4.22).

Concerns have also been raised. Not surprisingly, one of the most disliked aspects of the prototype was the necessity to select all the boundary edges to create a face (at the time, our prototype did not yet have the “paint bucket” feature). Sometimes, due to an unclear topology and tiny edges, this is hard to achieve. Also, rendering artefacts at junctions (see Section 4.8) and the difficulty to sculpt incident edges across a vertex, specifically to get a smooth transition, have been mentioned.





**Figure 4.22:** 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.

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 4.20), but reported difficulties to master the feature. However, we believe that further interface revisions and video tutorials would ease the learning experience.

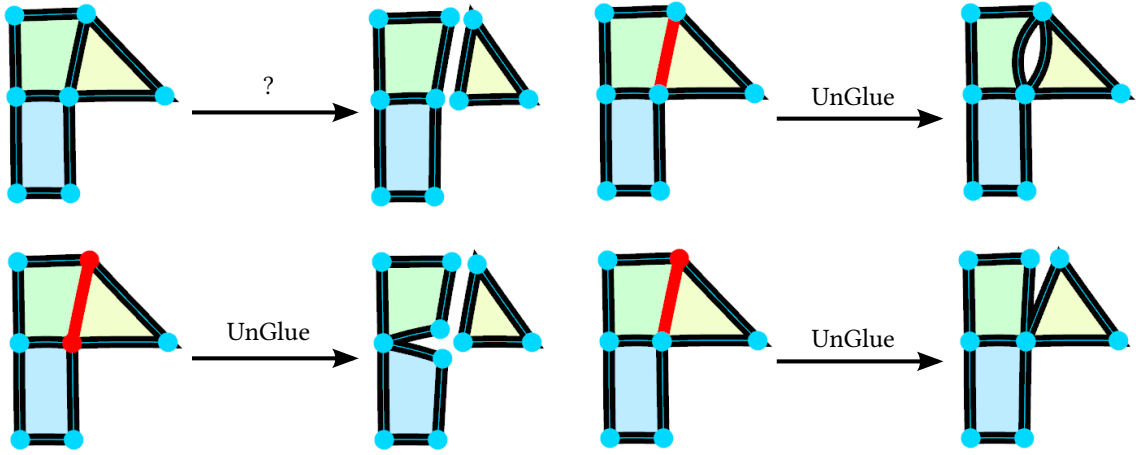
## 4.8 Limitations and Future Work

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 tessellator 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.

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 outweigh the added complexity.

Some desired topological operations must currently be performed using multiple steps, such as the “partial unglue” shown in Figure 4.23. 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 4.24, 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-



**Figure 4.23:** 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.

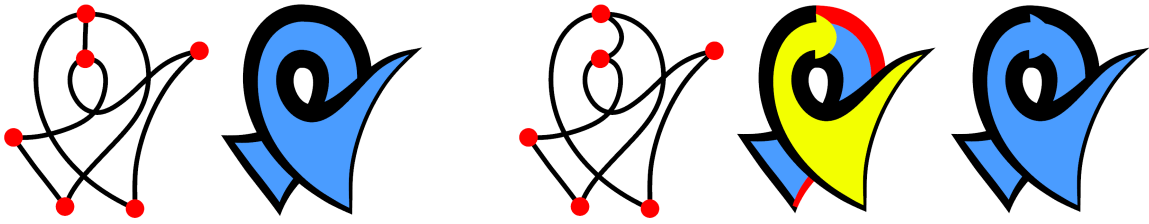


**Figure 4.24:** Illustrator (except when using LivePaint) cannot represent multiway joins, thus fails to render them correctly: incident faces are rendered independently. With the VGC, we are aware of multiway joins and hence can improve the rendering, as illustrated here with the two common styles “bevel” and “miter”.

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 4.25, we illustrate a typical artefact that occurs due to the overlapping ability of VGC cells and the presence of zero-width or transparent edges.

## 4.9 Conclusion

We have introduced the concept of *vector graphics complex*, a novel and powerful data structure for topology-aware design of 2D illustrations. It 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. Vector

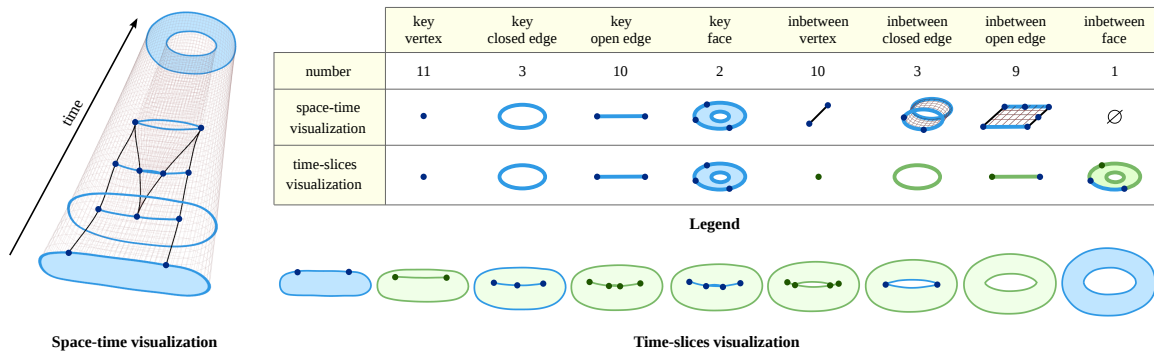


**Figure 4.25:** *Left: To obtain a self-overlapping object, one “invisible edge” is necessary, to define two faces with different depth orders. Right: This ordering implies that the red edge is below the yellow face. Depending on the geometry of the invisible edge, this situation often leads to artefacts in our implementation.*

graphics complexes neatly separate 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.

# Chapter 5

## Vector Animation Complexes: The Topology of Vector Animations



**Figure 5.1:** A space-time continuous 2D animation depicting a rotating torus, created without 3D tools. First, the animator draws key cells (in blue) using 2D vector graphics tools. Then, he specifies how to interpolate them using inbetween cells (in green). Our contribution is a novel data structure, called Vector Animation Complex (VAC), which enables such interaction paradigm.

In this chapter, we introduce the Vector Animation Complex (VAC), a novel data structure for vector graphics animation, designed to support the modeling of time-continuous topological events. This allows features of a connected drawing to merge, split, appear, or disappear at desired times via keyframes that introduce the desired topological change. Because the resulting space-time complex directly captures the time-varying topological structure, features are readily edited in both space and time in a way that reflects the intent of the drawing. A formal description of the data structure is provided, along with topological and geometric invariants. We illustrate our modeling paradigm with experimental results on various examples.

### 5.1 Introduction

A fundamental difference between raster graphics and vector graphics is that the former is a *discrete* representation, while the latter is a *continuous* representation. Instead of storing individual pixels that our eyes readily interpret as curves, vector graphics stores curves that can be ren-

dered at any resolution. As display devices spanning a wide range of resolutions proliferate, such resolution-independent representations are increasing in importance.

Similarly, a fundamental difference between traditional hand-drawn 2D animation and 3D animation is that the former is *discrete* in time, while the latter is *continuous* in time. Instead of storing individual frames that our eyes interpret as motion, the use of animation curves allows a scene to be rendered at any frame rate.

Space-time continuous representations, i.e., representations that are resolution-independent both in the spatial domain *and* the temporal domain, are ubiquitous within computer graphics for their many advantages. They are typically based on the “model-then-animate” paradigm: a parameterized model is first developed and then animated over time using animation curves that interpolate *key values* of the parameters at *key times*. A limitation of this paradigm is the underlying assumption that the model can be parameterized by a *fixed* set of parameters that captures the desired intent. This is indeed possible for 3D animation and simple 2D animation, but it quickly becomes impractical in any 2D animation scenario where the number of strokes or how they intersect change over time. In other words, the “model-then-animate” paradigm fails when the *topology* of the model is time-dependent, which makes it challenging to represent space-time continuous animated vector graphics illustrations with time-varying topology.

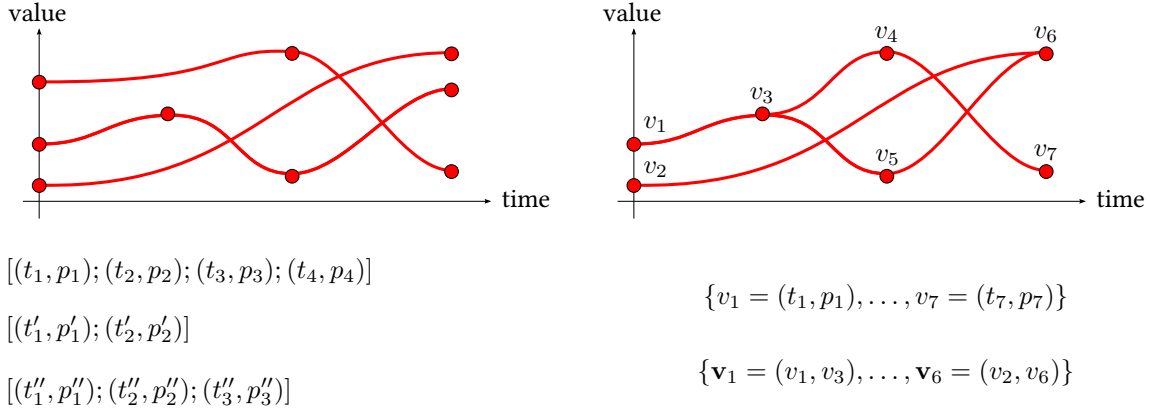
In this chapter, we address this problem by introducing the Vector Animation Complex (VAC). It is a cell complex immersed in space-time, specifically tailored to meet the requirements of vector graphics animation with non-fixed topology. Any time-slice of the complex is a valid Vector Graphics Complex (VGC) which make its rendering consistent with non-animated VGCs.

## 5.2 Space-Time Topology

In this section, we provide an initial intuition behind the vector animation complex, which we formally define in Section 5.3.1.

### 5.2.1 Animating Vertices

Suppose an animator wants to create a time-continuous animation of a single vertex  $v$ . This means that he needs to define its position  $p(t)$  for every time  $t$  in the life-span of the vertex. The existing approach (Fig. 5.2, Left) is to define a sequence of *keys*  $[(t_1, p_1); (t_2, p_2); \dots]$  which are interpolated in time. To animate three vertices, the animator would define three sequences of keys. Let us call this paradigm **sequential keyframing**, since the representation is a set of sequences of keys: one sequence per animated vertex, or more generally, one per animated degree of freedom.



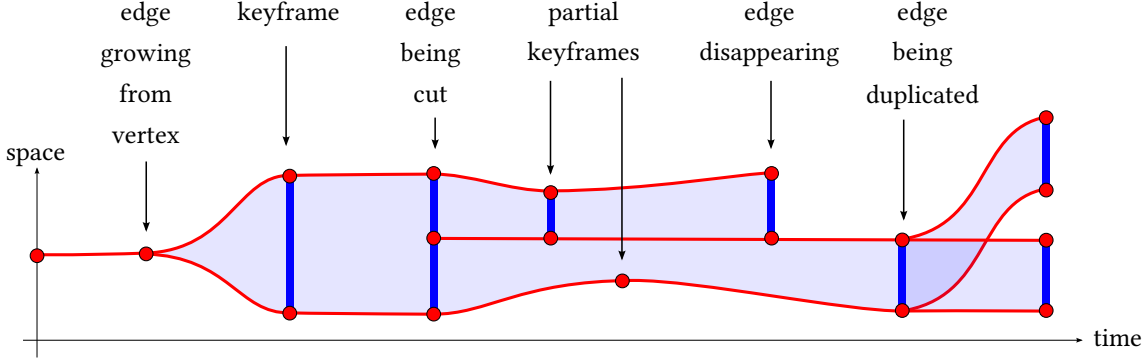
**Figure 5.2:** Left: The existing keyframing paradigm, defining an animation as ordered sequences of key values. Right: Our more general approach, where key values are unordered but labeled, and inbetween values specify which one to interpolate.

But what if the animator wants the number of vertices—or degrees of freedom—to change over time, by splitting or merging? We can observe (Fig. 5.2, Right) that the *space-time topology* of such animation is not anymore disconnected sequences, but a more general graph. Therefore, sequential keyframing fails to represent such animation with time-varying topology, and we need a more general approach to keyframing that we call **topological keyframing**. The animator first defines a set of **key vertices**  $v_i = (t_i, p_i)$ , as in sequential keyframing except that they are not ordered in sequences. Then, he defines a set of **inbetween vertex**  $\mathbf{v}_j = (v_{\text{before}}, v_{\text{after}})$  that reference to two key vertices to interpolate.

In theory, such paradigm can easily be applied to animate any kind of values, say, quaternions. However, in this chapter, we use it to animate the topology of vector graphics illustrations. This poses additional challenges due to the fact that such topology is already a graph-like structure in the space dimension. Therefore, we have to represent incidence relationships both in the temporal domain *and* the spatial domain, resulting in a space-time complex.

### 5.2.2 Animating Stroke Graphs

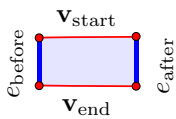
Suppose now that we want to animate a stroke graph [Whited et al. 2010], i.e. not only vertices but also (open) edges, which are 2D curves starting at a start vertex and ending at an end vertex. An easy way to achieve this is to define first a stroke graph, then use sequential keyframing to animate independently its degrees of freedom (e.g., position of the vertices and Bézier control points of the edges). Unfortunately, with this approach, it is impossible to represent animated stroke graphs



**Figure 5.3:** Stroke graph animation with time-varying topology. Red dots are key vertices; (non-vertical) red curves are inbetween vertices; (vertical) blue curves are key edges; and light blue areas are inbetween edges. Each annotation describes either a topological event introduced by key cells, or specifies that key cells are used as conventional “keyframes” (trajectory control, no change in topology). Note: key edges are represented as straight lines (because space is represented as 1D), but are in fact general 2D curves.

with time-varying topology.

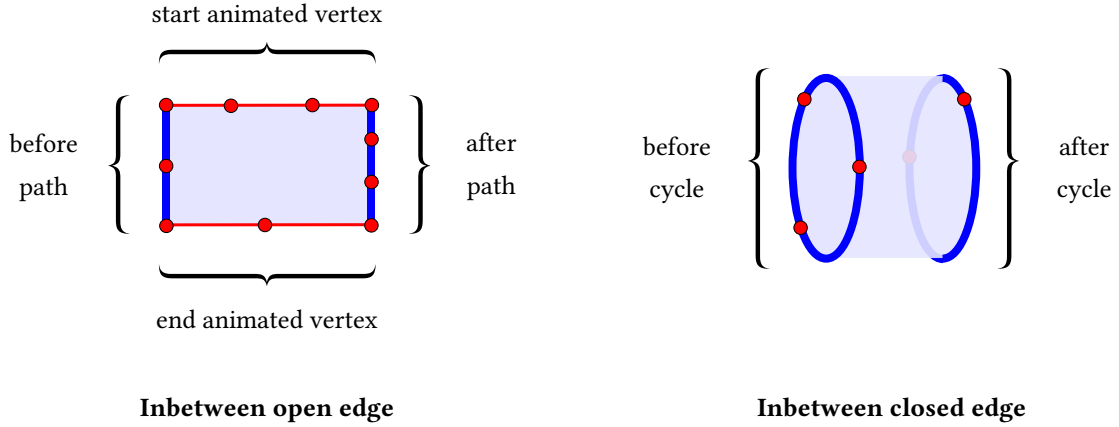
Our solution (Fig. 5.3) is to represent such animation as a space-time complex made of key vertices and inbetween vertices (as defined previously), but also key edges and inbetween edges. A **key (open) edge**  $e_i$  is defined by a time  $t_i$  and a 2D curve  $\phi_i(s)$ , starting at a key vertex  $v_{\text{start}} = (t_i, p_1)$  and ending at a key vertex  $v_{\text{end}} = (t_i, p_2)$ . An **inbetween (open) edge**  $e_j$  is defined by its **temporal boundary** and its **spatial boundary** (detailed in the next paragraph), from which can be computed a time-parameterized 2D curve  $\Phi(s, t)$  (i.e., a surface in space-time) interpolating this boundary.



Naively (Fig. opposite), one might define the temporal boundary as a pair  $(e_{\text{before}}, e_{\text{after}})$  that references to two key edges to interpolate, and the spatial boundary as a pair  $(v_{\text{start}}, v_{\text{end}})$  that references to two inbetween vertices where the time-parameterized curve  $\Phi(s, t)$  should start and end (for  $t$  fixed). Unfortunately,

this naive definition would only enable to represent a very small subset of all possible topological events that can happen to a stroke graph, and therefore we need a more general definition (Fig. 5.4, Left). Indeed, to represent an edge being cut in half by an appearing vertex (Fig. 5.3), or cut in more pieces by several vertices appearing simultaneously, we need the temporal boundary not to be two key edges, but two “sequences of connected key edges”, structure that we call **path**. To represent an edge growing from a vertex, we need to allow paths to be reduced to a key vertex. Finally, to allow **partial keyframing** (e.g., adding a key to an inbetween edge without adding a key to every incident edge, and recursively to every connected edge), we need the spatial boundary not to be two inbetween vertices, but two “sequences of connected inbetween vertices”, structure that





**Figure 5.4:** Topology of inbetween edges. Note: a path or cycle can also consist of a single key vertex (but an animated vertex cannot). A cycle can also consist of a single closed edge, possibly repeated.

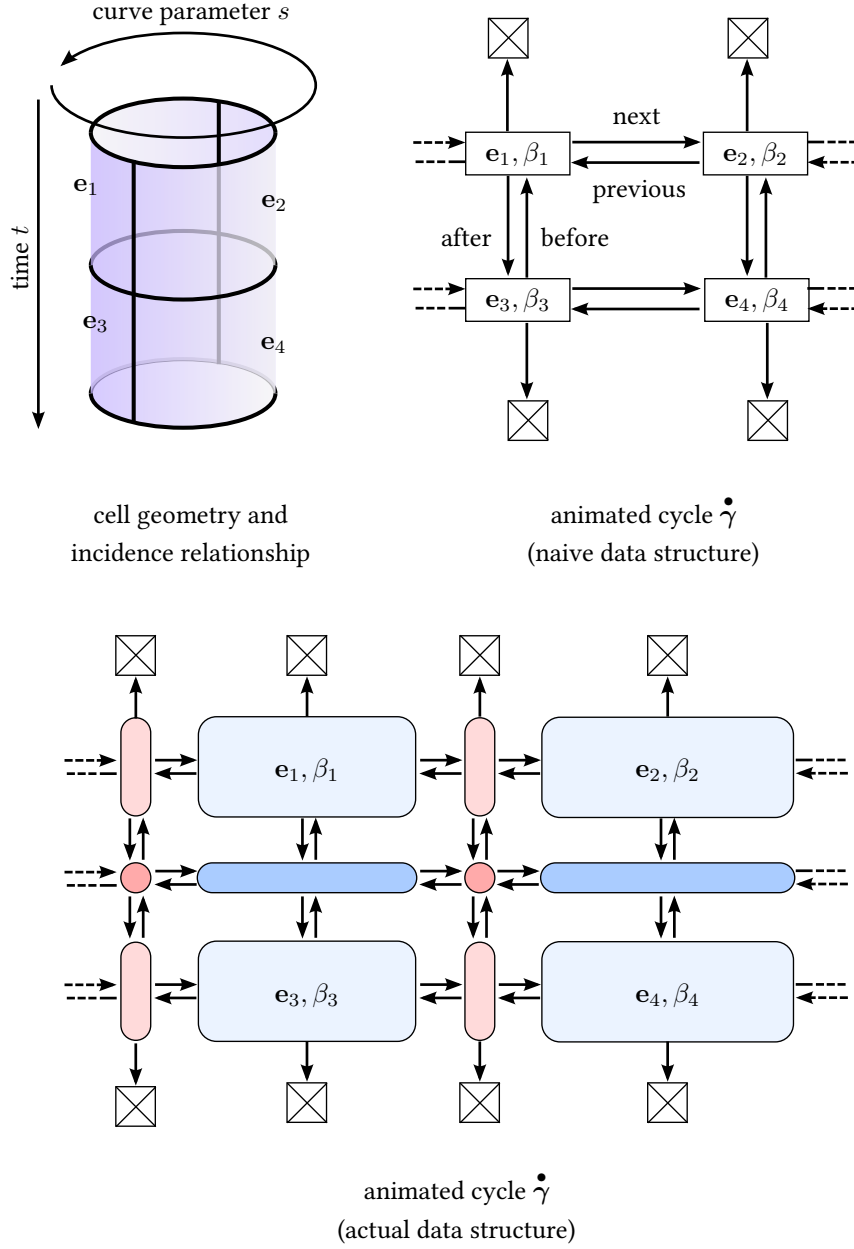
we call **animated vertex** (it is a chain key—inbetween—key—...—key—inbetween—key, which can be interpreted as a vertex animated using conventional keyframing).

### 5.2.3 Animating Vector Graphics Complexes

We extend these ideas further to represent an animated vector graphics complex (see Chapter 4) with time-varying topology. The same way that the VGC extends stroke graphs with closed edges and faces, the VAC extends the representation introduced in the previous section with key closed edges, inbetween closed edges, key faces, and inbetween faces.

A **key closed edge**  $e_i$  is defined by a time  $t_i$  and a 2D closed curve  $\phi_i(s)$  (note that it does not have bounding vertices). An **inbetween closed edge**  $e_j$  is defined by its temporal boundary, made of two **cycles** (Fig. 5.4, Right), from which can be computed a time-parameterized 2D closed curve  $\Phi(s, t)$  interpolating this boundary. Note that unlike inbetween open edges, inbetween closed edges have an empty spatial boundary, since closed edges do not have bounding vertices. To allow all sorts of topological events, cycles can either be reduced to a single key vertex, or made of a single (possibly repeated) key closed edge, or made of connected key open edges (equivalently to the concept of cycle introduced in Chapter 4).

A **key face**  $f_i$  is defined by a time  $t_i$  and a sequence of cycles, all sharing the same time  $t_i$ . Given a winding rule (e.g., *even-odd* or *non-zero*), these cycles define a 2D region of the time-plane  $t = t_i$ . An **inbetween face**  $f_j$  is defined by its temporal boundary and its spatial boundary. Its temporal boundary is defined by two sequences of faces, the *before faces* and the *after faces*. Its spatial boundary is defined by a sequence of **animated cycles**, structure that we introduce informally in the next three paragraphs.



**Figure 5.5:** Top: Intuitively, an animated cycle  $\dot{\gamma}$  is a two-dimensional doubly linked list where every node holds a reference to an inbetween edge  $e$  and a direction  $\beta$ . The structure is circular in the space dimension, and non-circular in the time dimension. Unfortunately, this naive structure is not expressive enough to capture all possible scenarios. Bottom: The actual data structure includes additional nodes to explicitly hold a reference to shared vertices, key edges, and inbetween vertices.

**Animated cycle** We have seen that an *animated vertex* is a combinatorial structure that stores references to existing inbetween vertices, which define a time-parameterized position  $p(t)$ . Similarly, our goal is now to define a time-parameterized closed curve  $\Phi(s, t)$ , via a combinatorial structure storing references to existing cells (a “cylinder in space-time”, cf. Fig. 5.5, Top-left).

A simple option would be to define this boundary as a set  $\{c_1, \dots, c_n\}$  of references to cells. However, for the same reasons that this approach is not sufficient to define the boundary of key faces (see paragraph *Non-Sufficiency of the Incidence Graph* from Section 4.3.3), it is not sufficient either to define the boundary of inbetween faces. More specifically, because overlapping of cells is allowed (i.e., the complex is only *immersed* in space-time, as opposed to *embedded*), then the *set* of boundary cells does not contain enough information to unambiguously define the geometry of the face. Instead, it is necessary to *organize* this set using an ordered structure, possibly referring to the same cell multiple times. This additional information explicitly defines a *parameterization* of the boundary. For key faces, this is achieved via the structure called *cycle*. For inbetween faces, this is achieved via the structure called *animated cycle*.

Intuitively (Fig. 5.5, Top-right), a naive structure to define such parameterization would be a two-dimensional doubly linked list of directed inbetween edges, where the first dimension corresponds to the curve parameter  $s$ , and the second dimension correspond to the time  $t$ . This structure is circular in the  $s$ -dimension, but non-circular in the  $t$ -dimension. Like a doubly linked list, it is composed of nodes which store: 1) per-node data; and 2) references to adjacent nodes. But unlike a doubly linked list, each node stores four references instead of two: **previous** and **next** to navigate in the  $s$ -dimension; and **before** and **after** to navigate in the  $t$ -dimension. The node data itself is a reference to an inbetween edge  $e$ , and a boolean  $\beta$  that orients  $e$  with respect to curve parameterization.

Unfortunately, this naive structure cannot handle inbetween edges bounded by more than two key edges, more than two inbetween vertices, or that shrink to a key vertex, and thus cannot represent general time-parameterized cycles (e.g., Fig. 5.6, Left). Our solution is to include all the lower dimensional cells shared between inbetween edges as explicit nodes of the structure (Fig. 5.5, Bottom; Fig. 5.6, Right). It introduces a little redundancy to the structure, but makes it significantly more expressive.

## 5.3 Formal Definition

### 5.3.1 Vector Animation Complex

A **vector animation complex**  $\mathcal{K}$  is defined as a tuple

$$\mathcal{K} = (C, \dim_T, \dim_S, \dots) \quad (5.1)$$

where  $C$  is a finite set of abstract symbols called **cells** (think of them as *identifiers*, or *addresses*), and  $\dim_T, \dim_S, \dots$  are functions defined on  $C$  or a subset of  $C$ , assigning to relevant cells some **attributes**, that have to satisfy some **invariants**. These numerous attributes and invariants are detailed in the remainder of this section. In our C++ implementation, an element  $c \in C$  is a *pointer* to an object inheriting the class `Cell`, and an attribute  $\alpha(c)$  is typically a data member `c->m_alpha`.

Cell attributes can be classified in two types: **topological attributes**, which are combinatorial objects defining incidence relationship between cells; and **geometrical attributes**, which are continuous objects immersing the cells in space-time. The two most important attributes of any cell  $c \in C$  are topological:

- its **temporal dimension**  $\dim_T(c) \in \{0, 1\}$
- its **spatial dimension**  $\dim_S(c) \in \{0, 1, 2\}$

Cells of temporal dimension 0 are called **key cells**, and cells of temporal dimension 1 are called **inbetween cells**. Orthogonally, cells of spatial dimension 0 are called **vertices**, cells of spatial dimension 1 are called **edges**, and cells of spatial dimension 2 are called **faces**. In addition, each edge  $e$  is assigned the topological attribute  $\text{isClosed}(e) \in \{\top, \perp\}$ , where  $\top$  means true and  $\perp$  means false. Therefore, all cells can be partitionned into eight finite sets which define their **type**:

$\dim_T$	$\dim_S$	isClosed	Type	Notation
0	0	n/a	<b>key vertex</b>	$v \in V$
0	1	true	<b>key closed edge</b>	$e \in E_\circ$
0	1	false	<b>key open edge</b>	$e \in E_\perp$
0	2	n/a	<b>key face</b>	$f \in F$
1	0	n/a	<b>inbetween vertex</b>	$\mathbf{v} \in \mathbf{V}$
1	1	true	<b>inbetween closed edge</b>	$\mathbf{e} \in \mathbf{E}_\circ$
1	1	false	<b>inbetween open edge</b>	$\mathbf{e} \in \mathbf{E}_\perp$
1	2	n/a	<b>inbetween face</b>	$\mathbf{f} \in \mathbf{F}$

For convenience, we define  $E = E_{\mid} \cup E_{\circ}$  and  $\mathbf{E} = \mathbf{E}_{\mid} \cup \mathbf{E}_{\circ}$ . From Section 5.3.2 to Section 5.3.9, we define all the remaining attributes and invariants for each type of cells. For clarity, some of these attributes are expressed using auxiliary/helper structures (halfedges, paths, cycles, animated vertices, and animated cycles), which are defined from Section 5.3.10 to Section 5.3.14.

### 5.3.2 Key Vertex

A key vertex  $v \in V$  represents a single point in space-time:

topological attributes:  $\emptyset$   
 geometrical attributes: **position**  $p(v) \in \mathbb{R}^2$   
                                   **time**  $t(v) \in \mathbb{R}$   
 invariants:  $\emptyset$

### 5.3.3 Key Closed Edge

A key closed edge  $e \in E_{\circ}$  represents a closed curve contained in a time-plane:

topological attributes:  $\emptyset$   
 geometrical attributes: **curve**  $\phi(e) : s \in [0, 1] \rightarrow \mathbb{R}^2$   
                                   **time**  $t(e) \in \mathbb{R}$   
 invariants:  $\phi(e)$  continuous  
                    $\phi(e)(0) = \phi(e)(1)$

### 5.3.4 Key Open Edge

A key open edge  $e \in E_{\mid}$  represents an open curve contained in a time-plane, starting and ending at two key vertices (possibly equal):

topological attributes: **start vertex**  $v_{\text{start}}(e) \in V$   
                                   **end vertex**  $v_{\text{end}}(e) \in V$   
 geometrical attributes: **curve**  $\phi(e) : s \in [0, 1] \rightarrow \mathbb{R}^2$   
                                   **time**  $t(e) \in \mathbb{R}$   
 invariants:  $\phi(e)$  continuous  
                    $\phi(e)(0) = p(v_{\text{start}}(e))$   
                    $\phi(e)(1) = p(v_{\text{end}}(e))$   
                    $t(v_{\text{start}}(e)) = t(e) = t(v_{\text{end}}(e))$

### 5.3.5 Key Face

A key face  $f \in E$  represents a region of a time-plane delimited by closed curves (possibly self-intersecting, including going back and forth the same path or being reduced to a single point):

$$\begin{aligned}
 \text{topological attributes: } & \mathbf{cycles} & \forall i \in [1..k(f)], \gamma_i(f) \in \Gamma \\
 & & \text{where } k(f) \geq 0 \\
 \text{geometrical attributes: } & \mathbf{winding rule} & R(f) \subseteq \mathbb{N} \\
 & \mathbf{time} & t(f) \in \mathbb{R} \\
 \text{invariants: } & \forall i \in [1..k(f)], t(f) = t(\gamma_i(f))
 \end{aligned}$$

### 5.3.6 Inbetween Vertex

An inbetween vertex  $\mathbf{v} \in \mathbf{V}$  represents an interpolation in time between two key vertices:

$$\begin{aligned}
 \text{topological attributes: } & \mathbf{before vertex} & v_{\text{before}}(\mathbf{v}) \in V \\
 & \mathbf{after vertex} & v_{\text{after}}(\mathbf{v}) \in V \\
 \text{geometrical attributes: } & \mathbf{animated position} & \mathbf{p}(\mathbf{v}) : t \in [t_1, t_2] \rightarrow \mathbb{R}^2 \\
 & & \text{where } t_1 = t(v_{\text{before}}(\mathbf{v})) \\
 & & t_2 = t(v_{\text{after}}(\mathbf{v})) \\
 \text{invariants: } & t_1 < t_2 \\
 & \mathbf{p}(\mathbf{v}) \text{ continuous} \\
 & \mathbf{p}(\mathbf{v})(t_1) = p(v_{\text{before}}(\mathbf{v})) \\
 & \mathbf{p}(\mathbf{v})(t_2) = p(v_{\text{after}}(\mathbf{v}))
 \end{aligned}$$

### 5.3.7 Inbetween Closed Edge

An inbetween closed edge  $\mathbf{e} \in \mathbf{E}_o$  represents an interpolation in time between two cycles:

$$\begin{aligned}
 \text{topological attributes: } & \mathbf{before cycle} & \gamma_{\text{before}}(\mathbf{e}) \in \Gamma \\
 & \mathbf{after cycle} & \gamma_{\text{after}}(\mathbf{e}) \in \Gamma \\
 \text{geometrical attributes: } & \mathbf{animated curve} & \Phi(\mathbf{e}) : (s, t) \in [0, 1] \times [t_1, t_2] \rightarrow \mathbb{R}^2 \\
 & & \text{where } t_1 = t(\gamma_{\text{before}}(\mathbf{e})) \\
 & & t_2 = t(\gamma_{\text{after}}(\mathbf{e}))
 \end{aligned}$$

$$\begin{aligned}
 \text{invariants: } & t_1 < t_2 \\
 & \Phi(\mathbf{e}) \text{ continuous} \\
 & \forall t \in [t_1, t_2], \Phi(\mathbf{e})(0, t) = \Phi(\mathbf{e})(1, t) \\
 & \forall s \in [0, 1], \Phi(\mathbf{e})(s, t_1) = \phi(\gamma_{\text{before}}(\mathbf{e}))(s) \\
 & \forall s \in [0, 1], \Phi(\mathbf{e})(s, t_2) = \phi(\gamma_{\text{after}}(\mathbf{e}))(s)
 \end{aligned}$$

### 5.3.8 Inbetween Open Edge

An inbetween open edge  $\mathbf{e} \in \mathbf{E}_I$  represents an interpolation in time between two paths, spatially bounded by two animated vertices:

$$\begin{aligned}
 \text{topological attributes: } & \textit{before path} & \pi_{\text{before}}(\mathbf{e}) \in \Pi \\
 & \textit{after path} & \pi_{\text{after}}(\mathbf{e}) \in \Pi \\
 & \textit{start animated vertex} & \dot{\mathbf{v}}_{\text{start}}(\mathbf{e}) \in \dot{\mathbf{V}} \\
 & \textit{end animated vertex} & \dot{\mathbf{v}}_{\text{end}}(\mathbf{e}) \in \dot{\mathbf{V}} \\
 \\
 \text{geometrical attributes: } & \textit{animated curve} & \Phi(\mathbf{e}) : (s, t) \in [0, 1] \times [t_1, t_2] \rightarrow \mathbb{R}^2 \\
 & & \text{where } t_1 = t(\pi_{\text{before}}(\mathbf{e})) \\
 & & t_2 = t(\pi_{\text{after}}(\mathbf{e})) \\
 \\
 \text{invariants: } & v_{\text{start}}(\pi_{\text{before}}(\mathbf{e})) = v_{\text{before}}(\dot{\mathbf{v}}_{\text{start}}(\mathbf{e})) \\
 & v_{\text{end}}(\pi_{\text{before}}(\mathbf{e})) = v_{\text{before}}(\dot{\mathbf{v}}_{\text{end}}(\mathbf{e})) \\
 & v_{\text{start}}(\pi_{\text{after}}(\mathbf{e})) = v_{\text{after}}(\dot{\mathbf{v}}_{\text{start}}(\mathbf{e})) \\
 & v_{\text{end}}(\pi_{\text{after}}(\mathbf{e})) = v_{\text{after}}(\dot{\mathbf{v}}_{\text{end}}(\mathbf{e})) \\
 & t_1 < t_2 \\
 & \Phi(\mathbf{e}) \text{ continuous} \\
 & \forall t \in [t_1, t_2], \Phi(\mathbf{e})(0, t) = \mathbf{p}(\dot{\mathbf{v}}_{\text{start}}(\mathbf{e}))(t) \\
 & \forall t \in [t_1, t_2], \Phi(\mathbf{e})(1, t) = \mathbf{p}(\dot{\mathbf{v}}_{\text{end}}(\mathbf{e}))(t) \\
 & \forall s \in [0, 1], \Phi(\mathbf{e})(s, t_1) = \phi(\pi_{\text{before}}(\mathbf{e}))(s) \\
 & \forall s \in [0, 1], \Phi(\mathbf{e})(s, t_2) = \phi(\pi_{\text{after}}(\mathbf{e}))(s)
 \end{aligned}$$

### 5.3.9 Inbetween Face

An inbetween face  $\mathbf{f} \in \mathbf{F}$  represents an interpolation in time between key faces, spatially bounded by animated cycles:

topological attributes:	<b>before time</b>	$t_{\text{before}}(\mathbf{f}) \in \mathbb{R}$
	<b>before faces</b>	$\forall i \in [1..k_b(\mathbf{f})], f_{\text{before},i}(\mathbf{f}) \in F$
		where $k_b(\mathbf{f}) \geq 0$
	<b>after time</b>	$t_{\text{after}}(\mathbf{f}) \in \mathbb{R}$
	<b>after faces</b>	$\forall i \in [1..k_a(\mathbf{f})], f_{\text{after},i}(\mathbf{f}) \in F$
		where $k_a(\mathbf{f}) \geq 0$
	<b>animated cycles</b>	$\forall i \in [1..k(\mathbf{f})], \dot{\gamma}_i(\mathbf{f}) \in \dot{\Gamma}$
		where $k(\mathbf{f}) \geq 0$
geometrical attributes:	<b>winding rule</b>	$R(\mathbf{f}) \subseteq \mathbb{N}$
invariants:	$\forall i \in [1..k_b(\mathbf{f})],$	$t_{\text{before}}(\mathbf{f}) = t(f_{\text{before},i}(\mathbf{f}))$
	$\forall i \in [1..k_a(\mathbf{f})],$	$t_{\text{after}}(\mathbf{f}) = t(f_{\text{after},i}(\mathbf{f}))$
	$\forall i \in [1..k(\mathbf{f})],$	$t_{\text{before}}(\mathbf{f}) = t_{\text{before}}(\dot{\gamma}_i(\mathbf{f}))$
	$\forall i \in [1..k(\mathbf{f})],$	$t_{\text{after}}(\mathbf{f}) = t_{\text{after}}(\dot{\gamma}_i(\mathbf{f}))$

### 5.3.10 Halfedge

A **halfedge** is a pair  $h = (e, \beta) \in E \times \{\top, \perp\}$ . If  $e$  is closed then it is a **closed halfedge** denoted  $h \in H_{\circ}$ , otherwise it is an **open halfedge** denoted  $h \in H_{\mid}$ . If  $\beta = \top$ , we define  $\phi(h)(s) = \phi(e)(s)$ , otherwise we define  $\phi(h)(s) = \phi(e)(1 - s)$ . If  $h$  is open then we define  $v_{\text{start}}(h) = v_{\text{start}}(e)$  and  $v_{\text{end}}(h) = v_{\text{end}}(e)$  (when  $\beta = \top$ ), or  $v_{\text{start}}(h) = v_{\text{end}}(e)$  and  $v_{\text{end}}(h) = v_{\text{start}}(e)$  (when  $\beta = \perp$ ). Finally, we define  $t(h) = t(e)$ .

### 5.3.11 Path

A **path**  $\pi$  is either:

1. a key vertex  $v \in V$ , or
2. a list of  $N > 0$  open halfedges  $h_1, \dots, h_N \in H_{\mid}$  satisfying:

$$\forall j \in [1..N-1], \quad v_{\text{end}}(h_j) = v_{\text{start}}(h_{j+1}) \quad (5.2)$$

In the first case, we define  $v_{\text{start}}(\pi) = v_{\text{end}}(\pi) = v$ , otherwise we define  $v_{\text{start}}(\pi) = v_{\text{start}}(h_1)$  and  $v_{\text{end}}(\pi) = v_{\text{end}}(h_N)$ . Also, we define the curve  $\phi(\pi) : s \in [0, 1] \rightarrow \mathbb{R}^2$  by concatenating and uniformly reparameterizing the  $\phi(h_j)$ . In the special case  $\pi = v$ , then  $\phi(\pi)$  is the constant



function  $\Phi(\pi)(s) = p(v)$ . Finally, we define  $t(\pi) = t(v)$  (Case 1.), or  $t(\pi) = t(h_1)$  (Case 2.). We denote by  $\Pi$  the set of all possible paths on  $\mathcal{K}$ .

### 5.3.12 Cycle

A **cycle**  $\gamma$  is either:

1. a key vertex  $v \in V$ , or
2. a closed halfedge  $h \in H_o$  repeated  $N > 0$  times, or
3. a circular list of  $N > 0$  open halfedges  $h_j \in H_l$  satisfying:

$$\forall j \in [1..N], \quad v_{\text{end}}(h_j) = v_{\text{start}}(h_{j+1}) \quad (5.3)$$

In addition, a cycle stores a **starting point**  $s_0 \in \mathbb{R}$ . We define the closed curve  $\phi(\gamma) : s \in [0, 1] \rightarrow \mathbb{R}^2$  by concatenating and uniformly reparameterizing the  $\phi(h_j)$ , then offsetting by  $s_0$ . In the special case  $\gamma = v$ , then  $\phi(\gamma)$  is the constant function  $\Phi(\gamma)(s) = p(v)$ . Finally, we define  $t(\gamma) = t(v)$  (Case 1.), or  $t(\gamma) = t(h)$  (Case 2.), or  $t(\gamma) = t(h_1)$  (Case 3.). We denote by  $\Gamma$  the set of all possible cycles on  $\mathcal{K}$ .

### 5.3.13 Animated Vertex

An **animated vertex**  $\dot{\mathbf{v}}$  is a list of  $N > 0$  inbetween vertices  $\mathbf{v}_1, \dots, \mathbf{v}_N \in \mathbf{V}$  satisfying:

$$\forall j \in [1..N - 1], \quad v_{\text{after}}(\mathbf{v}_j) = v_{\text{before}}(\mathbf{v}_{j+1}) \quad (5.4)$$

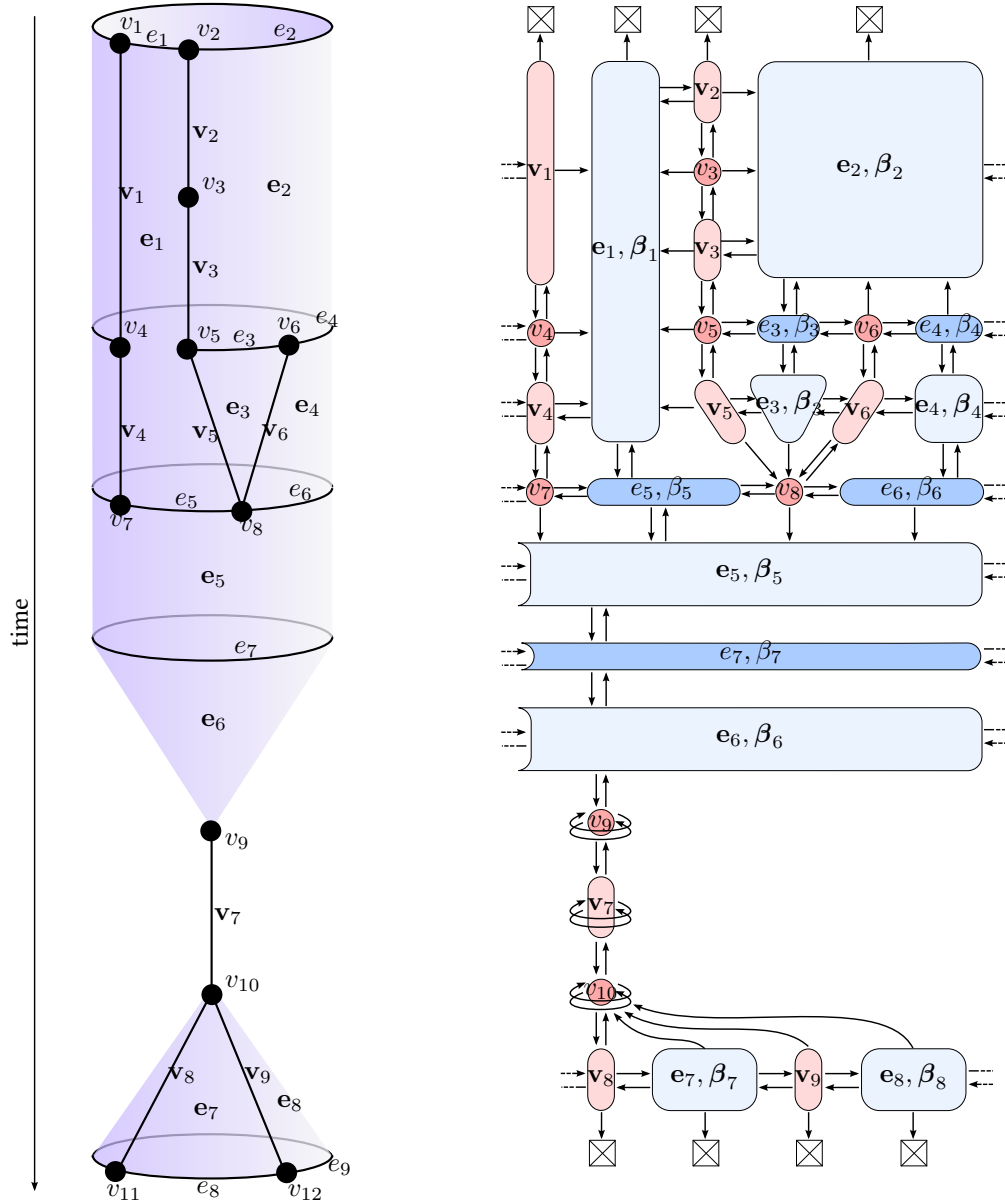
We define  $v_{\text{before}}(\dot{\mathbf{v}}) = v_{\text{before}}(\mathbf{v}_1)$  and  $v_{\text{after}}(\dot{\mathbf{v}}) = v_{\text{after}}(\mathbf{v}_N)$ . Also, we define the time-parameterized position  $\mathbf{p}(\dot{\mathbf{v}}) : t \in [t(v_{\text{before}}(\dot{\mathbf{v}})), t(v_{\text{after}}(\dot{\mathbf{v}}))] \rightarrow \mathbb{R}^2$  by concatenating the  $\mathbf{p}(\mathbf{v}_j)$ . We denote by  $\dot{\mathbf{V}}$  the set of all possible animated vertices on  $\mathcal{K}$ .

### 5.3.14 Animated Cycle

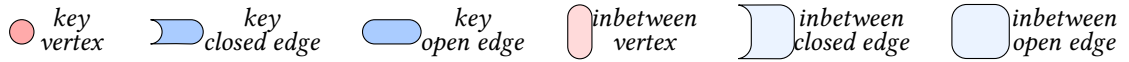
An **animated cycle** (cf. Fig. 5.6) is a tuple

$$\dot{\gamma} = (N, c, \beta, n_{\text{previous}}, n_{\text{next}}, n_{\text{before}}, n_{\text{after}}) \quad (5.5)$$

where  $N$  is a non-empty set of symbols called **nodes**, and:



**Figure 5.6:** A general example of an animated cycle  $\dot{\gamma}$ . Left: Geometry and topology of the cells  $c \in C(K)$  involved in  $\dot{\gamma}$ . It is a sub-complex of the whole VAC. Right: The nodes  $n \in N(\dot{\gamma})$  defining  $\dot{\gamma}$ . Each node  $n$  references to a cell  $c$ , specifies a direction  $\beta$  (ignored if  $c$  is a vertex), and points to a previous, next, before, and after node. The shape/color of the node indicates the type of the referenced cell:



This example illustrates a variety of topological transformations over time for the cycle, including local keyframing using a key vertex ( $v_3$ ), local keyframing using key edges ( $e_3, e_4$ ), keyframing using a closed edge ( $e_7$ ), contraction of an open edge to a vertex ( $e_3 \rightarrow v_8$ ), contraction of a closed edge to a vertex ( $e_7 \rightarrow v_9$ ), cutting an open edge into two open edges ( $e_2 \rightarrow e_3, e_4$ ), among others.

$c : N \rightarrow V \cup \mathbf{V} \cup E \cup \mathbf{E}$	assigns a <b>cell</b> to every node
$\beta : N \rightarrow \{\top, \perp\}$	assigns a <b>direction</b> (ignored if $c(n) \in V \cup \mathbf{V}$ )
$n_{\text{previous}} : N \rightarrow N$	assigns a <b>previous</b> node
$n_{\text{next}} : N \rightarrow N$	assigns a <b>next</b> node
$n_{\text{before}} : N \rightarrow N \cup \{\text{null}\}$	assigns an optional <b>before</b> node
$n_{\text{after}} : N \rightarrow N \cup \{\text{null}\}$	assigns an optional <b>after</b> node

In addition, an animated cycle stores a **starting node**  $n_0 \in N$ . We define the **timespan** of a node  $n$  as being the trivial interval  $T(n) = \{t(c(n))\}$  if  $c(n)$  is a key cell, or the open interval  $T(n) = (t_{\text{before}}(c(n)), t_{\text{after}}(c(n)))$  if  $c(n)$  is an inbetween cell. Despite having a single *next* pointer, one can notice (Fig. 5.6) that when  $c(n)$  is an inbetween open edge, then  $n$  may have several nodes “next to it”, which are stacked in time. The *next* (resp. *previous*) pointer points to the “first” of these, and the others can be accessed by iterating *after* (resp. *before*). To easily traverse the data-structure at  $t$  fixed, we define the two functions  $n_{\text{next}}(n, t)$  and  $n_{\text{previous}}(n, t)$  that return the two nodes “spatially adjacent to  $n$  at time  $t$ ”.

$n_{\text{previous}}(n \in N, t \in \mathbb{R})$	$n_{\text{next}}(n \in N, t \in \mathbb{R})$
<b>Require:</b> $t \in T(n)$	<b>Require:</b> $t \in T(n)$
1 $n' \leftarrow n_{\text{previous}}(n)$	1 $n' \leftarrow n_{\text{next}}(n)$
2 <b>while</b> $t \notin T(n')$ <b>do</b>	2 <b>while</b> $t \notin T(n')$ <b>do</b>
3 $n' \leftarrow n_{\text{before}}(n')$	3 $n' \leftarrow n_{\text{after}}(n')$
4 <b>return</b> $n'$	4 <b>return</b> $n'$

We define the time-parameterized closed curve  $\Phi(\dot{\gamma})(s, t)$  by finding a node  $n$  such that  $t \in T(n)$  (iterating before/after from  $n_0$ ), then concatenating the  $\phi(c(n))$  while iterating  $n_{\text{next}}(n, t)$  (followed by a normalization into  $[0, 1]$ ). We denote by  $\dot{\mathbf{I}}$  the set of all valid animated cycle on  $\mathcal{K}$ , which are the ones whose attributes satisfy the following **invariants**.

**Connectedness** Any node  $n_2$  can be reached from any node  $n_1$  with a finite sequence of *next*, *previous*, *after*, or *before*.

**Node-cell consistency** Informally: “adjacency between nodes must be consistent with incidence relationship between assigned cells”. For instance, if  $c(n)$  is an open key edge and  $\beta(n) = \top$ , then we must have:

$$c(n_{\text{previous}}(n)) = v_{\text{start}}(c(n)) \quad \text{and} \quad c(n_{\text{next}}(n)) = v_{\text{end}}(c(n)) \quad (5.6)$$

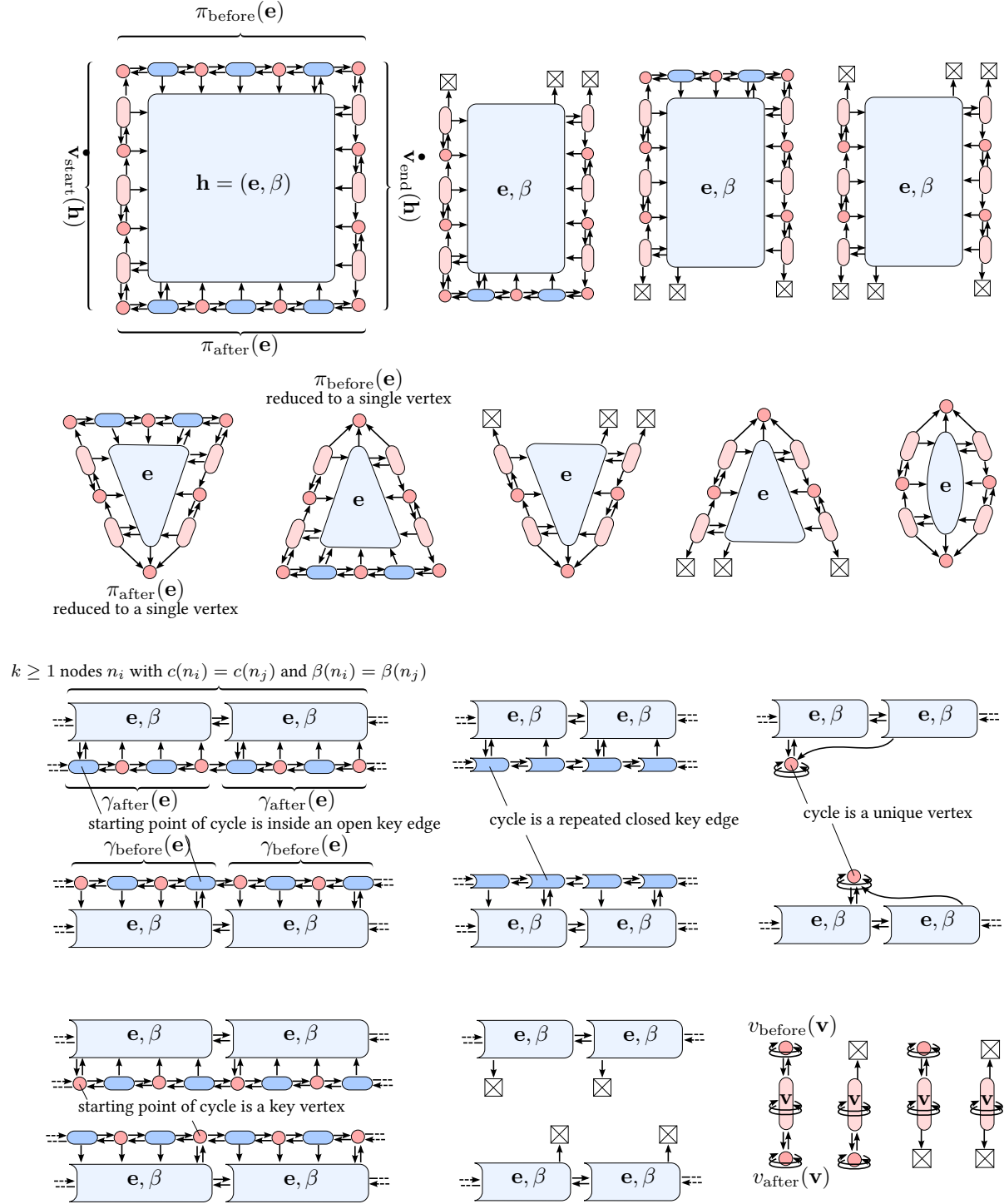
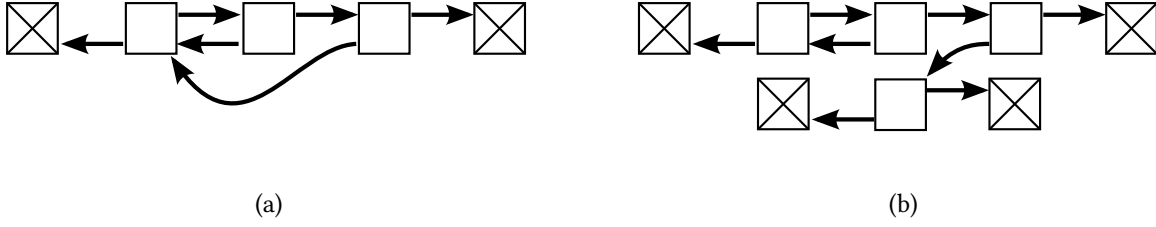
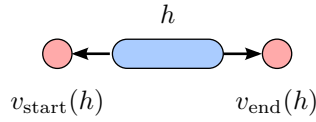


Figure 5.7: Node-cell consistency invariants expressed as a set of allowed configurations.



**Figure 5.8:** Two examples of doubly linked lists which violate the invariant  $n_{\text{next}}(n_{\text{previous}}(n)) = n$ , therefore are invalid.

The exhaustive list of all constraints would be very hard to read if expressed algebraically, as above, despite being intuitive to understand visually. Therefore, for clarity, we express them all in Figure 5.7 as a set of *allowed configurations*, using diagrams. For instance, the above two constraints can be expressed as<sup>13</sup>:



In these diagrams, space is represented horizontally and time vertically. Every colored shape represents a node  $n \in N$ , and its shape and color represents the type of  $c(n)$ , cf. legend of Figure 5.6. Annotations near or inside the shape indicate the value of  $c(n)$  and  $\beta(n)$ , sometimes using a “halfedge notation” for conciseness. Left (resp. right, up, and down) arrows indicate the value of  $n_{\text{previous}}(n)$  (resp.  $n_{\text{next}}(n)$ ,  $n_{\text{before}}(n)$ , and  $n_{\text{after}}(n)$ ). An unspecified annotation/arrow in a diagram indicates that its value is not constrained by this diagram, but the value must still be allowed by at least one of the other diagrams.

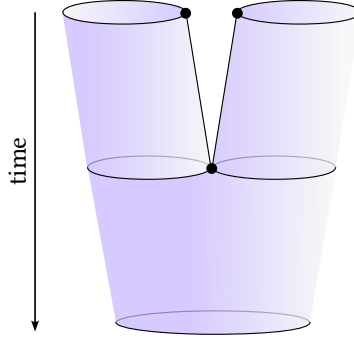
**Back-pointers consistency** For all nodes  $n \in N$ , there exist  $k_1, k_2, k_3, k_4 \in \mathbb{N}$  such that:

$$\left\{ \begin{array}{l} n_{\text{after}}^{k_1}(n_{\text{next}}(n_{\text{previous}}(n))) = n \\ n_{\text{before}}^{k_2}(n_{\text{previous}}(n_{\text{next}}(n))) = n \\ n_{\text{next}}^{k_3}(n_{\text{after}}(n_{\text{before}}(n))) = n \\ n_{\text{previous}}^{k_4}(n_{\text{before}}(n_{\text{after}}(n))) = n \end{array} \right. \quad (5.7)$$

where exponents represent the  $k$ -th iterate of the function. In addition, for all nodes  $n \in N$ , and for all  $t \in T(n)$ :

$$\left\{ \begin{array}{l} n_{\text{previous}}(n_{\text{next}}(n, t), t) = n \\ n_{\text{next}}(n_{\text{previous}}(n, t), t) = n \end{array} \right. \quad (5.8)$$

<sup>13</sup>However, we note that this specific diagram does not appear in Figure 5.7 because it is redundant with other diagrams.



**Figure 5.9:** Example of an invalid animated cycle: it doesn't satisfy the cycle uniqueness invariant.

These invariants play the same role as the simpler invariant  $n_{\text{next}}(n_{\text{previous}}(n)) = n$  that must be satisfied by conventional doubly linked lists. It ensures that back pointers serve their purpose (i.e. point back to the original node), and prevents invalid configurations such as those illustrated in Figure 5.8.

**Cycle uniqueness** For all pairs of nodes  $n_1, n_2 \in N$ , if there exists a time  $t$  such that  $t \in T(n_1)$  and  $t \in T(n_2)$ , then there exists  $k \in \mathbb{N}$  such that:

$$n_{\text{next}}^k(n_1, t) = n_2 \quad (5.9)$$

This invariant makes sure that a scenario such as in Figure 5.9 is not allowed, i.e. that any time-slice of the animated cycle is one and only one cycle.

**Temporal boundary structure** There exist a unique time  $t_1$  and a unique time  $t_2$ , such that for each node  $n \in N$ :

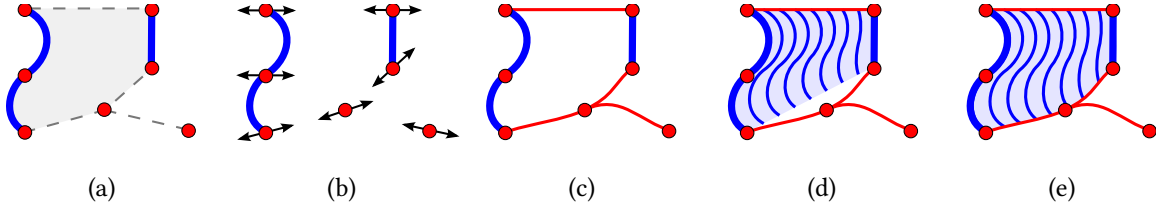
$$\begin{cases} n_{\text{before}}(n) = \text{null} & \Rightarrow & t_{\text{before}}(c(n)) = t_1 \\ n_{\text{after}}(n) = \text{null} & \Rightarrow & t_{\text{after}}(c(n)) = t_2 \end{cases} \quad (5.10)$$

We denote these two times  $t_{\text{before}}(\dot{\gamma})$  and  $t_{\text{after}}(\dot{\gamma})$ .

## 5.4 Interpolation Scheme

The geometry of inbetween cells may be provided explicitly (or in non-photorealistic rendering applications, computed from an animated 3D model), but in our case, it is computed by interpolating the geometry of key cells, as expected from a keyframing system.

First, for each key vertex  $v_i$ , we define a tangent  $q(v_i)$  as the average of the slopes  $\frac{p(v_j) - p(v_i)}{t(v_j) - t(v_i)}$ , for all key vertices  $v_j$  connected to  $v_i$  by an inbetween vertex (Fig. 5.10b). Then, we define the geom-



**Figure 5.10:** Interpolation scheme (time = horizontal axis). (a) Input: geometry of key cells and space-time topology. (b) Compute tangents at key vertices. (c) Compute geometry of inbetween vertices, satisfying tangents. (d) For each inbetween edge, compute linear interpolation of bounding paths/cycles. (e) Output: warp to satisfy spatial boundary conditions.

etry of each inbetween vertex as the unique cubic curve interpolating the positions  $p(v_{\text{before}})$  and  $p(v_{\text{after}})$  with the desired tangents  $q(v_{\text{before}})$  and  $q(v_{\text{after}})$  (Fig. 5.10c). All that is left to do is define the geometry of every inbetween open (resp. closed) edge, by interpolating its two bounding paths (resp. cycles). We recall from Section 5.3.1 that paths/cycles have an explicit parameterization  $[0, 1] \rightarrow \mathbb{R}^2$ , obtained by concatenating and uniformly reparameterizing the key edges' parameterizations (the starting point of cycles is a user-controllable variable). First, we compute a linear interpolation between these two explicit parameterizations (Fig. 5.10d). Finally, in the case of inbetween open edges, for all  $t \in (t_1, t_2)$ , we linearly warp this interpolation  $\Phi(s, t)$  such that  $\Phi(0, t)$  and  $\Phi(1, t)$  coincide with the start and end animated vertices at  $t$  (Fig. 5.10e). There is no need to define an interpolation scheme for inbetween faces, since their geometry is entirely specified by the geometry of their boundary. Indeed, for all  $t \in (t_1, t_2)$ , a closed parameterized curve  $[0, 1] \rightarrow \mathbb{R}^2$  can be extracted from each animated cycle, which, together with the user-specified winding rule (e.g., even-odd), define an area of the 2D plane.

This interpolation scheme is robust and general but limited as it only guarantees  $\mathcal{C}^0$  continuity. More aesthetically pleasing interpolation can be achieved using logarithmic spirals [Whited et al. 2010] or Coons patches. This is left for future work.

## 5.5 User Interface

To create and manipulate VACs, we implemented various visualizations and topological operators, which we present in this section. We refer to the video accompanying [Dalstein et al. 2015] for a demonstration of these tools.

**2D view** We provide a 2D view to render a specific frame of the animation (i.e., a time-slice of the VAC), which can be selected using a timeline similar to any animation system. The 2D view can be split into multiple 2D views to visualize simultaneously different frames of the animation. The user can also toggle “onion skinning” to overlay several frames within a single 2D view, or

render the animation as an animation strip (Fig. 5.1, bottom). The frames can be rendered either in “normal” mode (showing the actual result), or in “topology” mode (using a color code to inform whether a cell is a key cell, or a time-slice of an inbetween cell).

**3D view** We also provide a 3D view to visualize the VAC in space-time. However, we mostly use this view as a debugging tool, as it becomes quickly impractical when the number of cells grow. All interaction happens in the 2D views, and all examples presented in this chapter have been created without using the 3D view at all. At this stage, it is unclear whether it is relevant to expose such visualization to end users.

**Creating key cells** Key cells are created in the 2D view using standard VGC tools. They are assigned the time  $t_i$  selected in the timeline.

**Motion-pasting** The easiest way to create inbetween cells is to select key cells at time  $t_1$ , trigger the *copy* action, then move to time  $t_2$  and trigger the *motion-paste* action. It creates a copy of the key cells, assigns them the time  $t_2$ , and creates inbetween cells that connect in time the old key cells to the new ones. In other words, it corresponds to sweeping key cells in time. Once motion-pasted, the new cells can be edited to create the desired motion. Standard VGC topological operators (extended to support incident inbetween cells) can also be used on the new key cells, which introduce topological events as a result.

**Inbetweening** Another way to create inbetween cells is to select existing key cells at two different times  $t_1$  and  $t_2$  (e.g., using side-by-side 2D views), then trigger the *inbetweening* action. It creates inbetween cells that connect in time the selected key cells. Currently, it works to create an inbetween vertex out of two key vertices; an inbetween edge out of two key edges; an inbetween edge out of more than two key edges that can be organized into two paths or two cycles; or an inbetween edge that grows or shrinks to a vertex. This tool does not yet support the creation of inbetween faces (we can still create them using motion-pasting or manually specifying their boundary), neither the simultaneous creation of multiple inbetween edges, which are both interesting challenges left as future work.

**Inserting keys** A fundamental topological operator on VACs is to cut an inbetween cell in half, in the time dimension, by inserting a new key cell. It is the equivalent of inserting a keyframe in conventional keyframing animation. Similarly to the “auto-key” feature of most animation systems, we automatically call this operator whenever the user performs an action on (the rendered time-slice of) an inbetween cell. For instance, attempting to move an inbetween vertex automatically inserts a key vertex at the time  $t_i$  selected in the timeline, and the new key vertex is the cell actually moved. Note that this *insert key* tool is local in space (in addition to be obviously local in



time): it cuts the selected inbetween cell and its spatial boundary, but does not propagate to any other cell. This allows for local trajectory or topology refinement possible, without keyframing the whole drawing.

**Drag-and-drop** Selected key cells can be drag-and-dropped in space (using the 2D view), but also in time (using the timeline), within a time interval determined by its incident inbetween cells. This allows to easily refine the timing of a motion.

**Depth-ordering** We store a global ordering for all the cells in a complex using a doubly-linked list, and render the cells back-to-front using this ordering. As with VGCs, we provide tools to conveniently alter this ordering.

## 5.6 Results

We create several illustrative examples of vector graphics animations that involve topological changes over time. We briefly summarize them below, although they are best seen in the video accompanying [Dalstein et al. 2015].

**Torus** The torus (Fig. 5.1) is an example use of the VAC to create a clean conceptual animated vector graphics illustration. It is defined using a total of five *keyframes* (frames that contain at least one key cell). As always required, the clip begins and ends with *full keyframes* (frames containing key cells only), which specify the shape of all the drawing elements that exist at those key times. The second keyframe captures the motion of the interior silhouette, marks the initial appearance of the hole with a single vertex, and also keyframes the shape of the outside silhouette. The third keyframe properly introduces the now-visible hole, while the fourth keyframe then ends the growth of the hole by merging the end vertices of the two lines. As seen in this example, keyframes are used either to introduce a change in shape, to introduce a change in topology, or both. Keyframes are typically *local*, i.e., key cells are only inserted where needed, without keyframing the entire drawing.

**Double torus** Once we have the VAC for the single torus, the animation of a double torus (Fig. 5.11) is easy to create. Indeed, all that is needed is: 1) deforming the outside silhouette, 2) copy-pasting to a different space-time location the sub-complex representing the animation of the hole, and 3) gluing the first key edge of this sub-complex to the deformed silhouette. We believe that this type of template-based construction provides a practical way of simplifying the creation of VACs. Figure 5.11 shows a vector graphics animation of a simple torus which is morphed to a double torus, with the two halves rotating asynchronously. Creating such animation would be

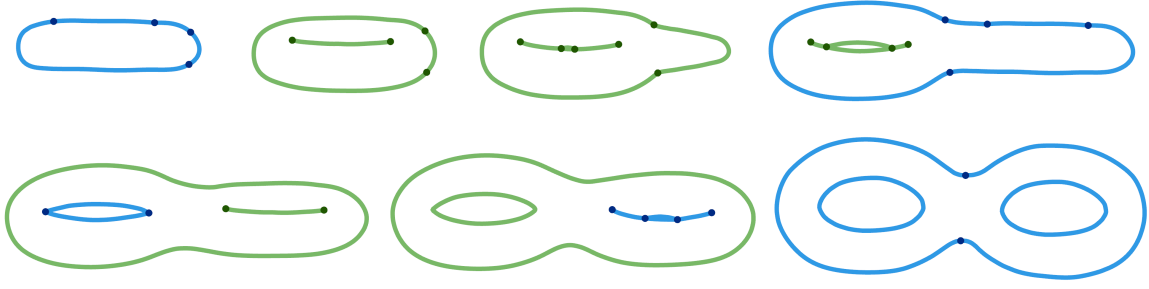


Figure 5.11: Double Torus.

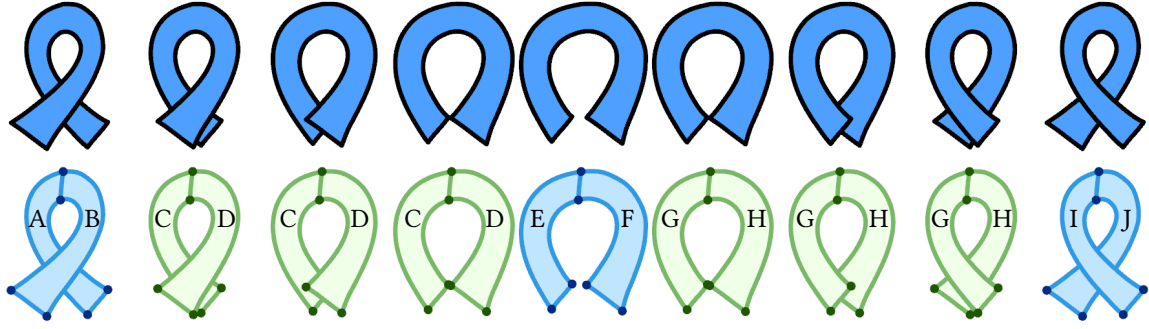
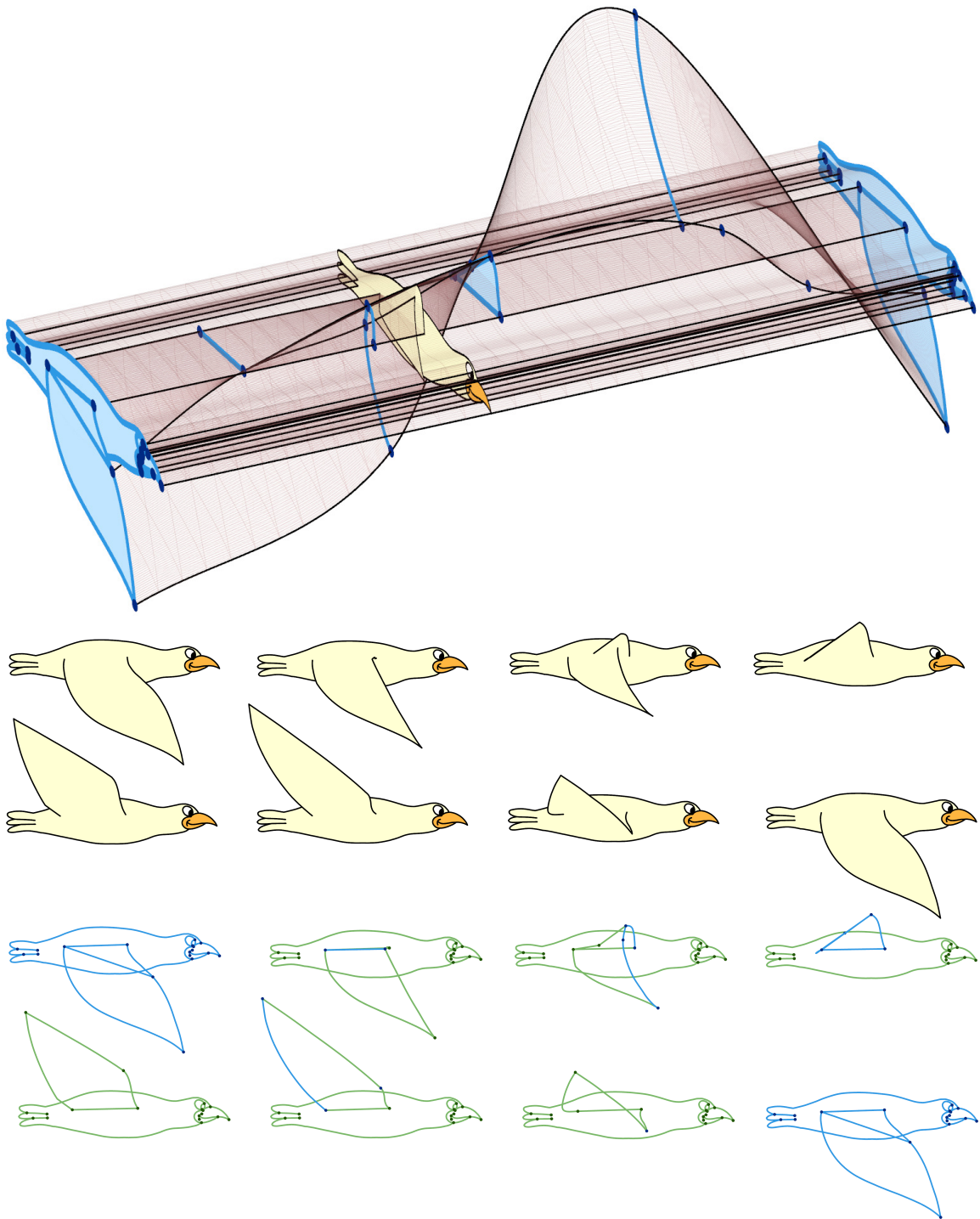


Figure 5.12: Animated ribbon decomposed into 6 key faces (A,B,E,F,I,J) and 4 inbetween faces (C,D,G,H), in order to depict local depth-ordering both in space and time.

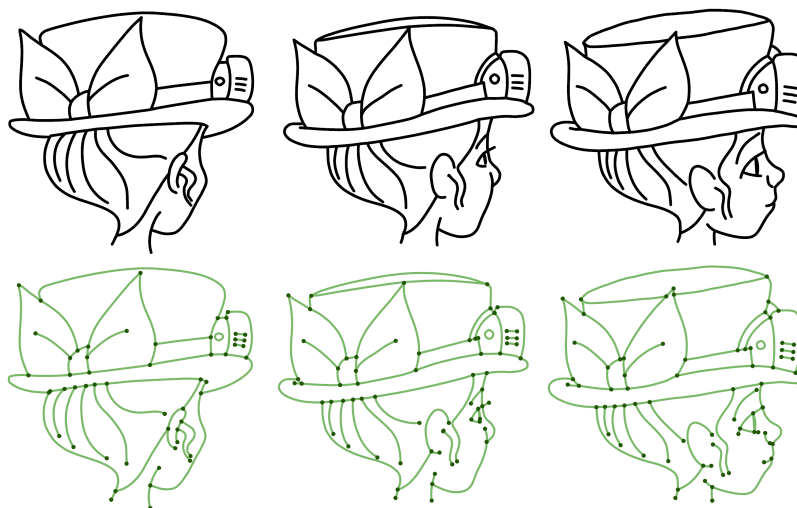
hard using conventional vector graphics tools, but would be equally hard in 3D, since the genus of the depicted surface changes, requiring a topological event in 3D as well.

**Animated ribbon** In a given VAC, any cell is either completely in front, or completely behind, any other cell. However, any cell can be easily split spatially (cutting) or temporally (keyframing) into different cells, and the cells of this new cell-decomposition are assigned their own independent depth orders. This makes possible to depict local depth-ordering, both in space and time, as illustrated in Figure 5.12. Using motion-pasting and basic editing, the space-time topology and geometry of this animation can be created within a few seconds. Then, the user alters the depth-ordering to ensure  $A < B$ ,  $C < D$ ,  $G > H$ , and  $I > J$ , using the “raise” and “lower” actions, as with standard VGCs. Note that this example does not contain topological events: keyframes are only used to introduce a change in geometry, as well as a change in depth-ordering.

**Flapping bird** We demonstrate Figure 5.13 the use of animated faces with depth layering in creating an example of a bird with a flapping wing, as inspired by a hand-drawn animation [Blair 1994, p. 122]. This example is created using 7 keyframes, all of which are local except the ones at the start and end. A looping animation is easily created by copy-and-pasting a second copy of the



**Figure 5.13:** Bird animation. Space-time view (top); output animation (middle); VGC film strip (bottom).



**Figure 5.14:** Turning head animation. Output animation (top); VGC film strip (bottom).

full VAC so that it sits immediately after the first copy. The ending elements of the first animation are then topologically glued to the starting elements of the second animation.

**Head turning** We use a drawn animation sequence by James Lopez (used with permission) as inspiration for a more complex example, shown in Figure 5.14. This involves many drawn elements and a significant number of topological changes, particularly involving the ear, goggles, eye, and mouth. Many topological changes need not be modeled in great detail. The eye is a good example: the features of the eye are simply spawned from an initial vertex that is introduced on the silhouette of the face. For this example, the 3D space-time view is largely unusable because of its complexity, and thus it proved to be a good test case for the capabilities of our user interface.

## 5.7 Discussion

**Creation** Many aspects of working with the VAC are no different than that of creating a conventional keyframe animation. Animation workflows are often classified as being *straight ahead* or *pose-to-pose*, and these working styles can each be reproduced using the VAC. A *straight ahead* workflow is readily reproduced using motion-pasting to create a new keyframe, followed by editing as necessary. A *pose-to-pose* workflow can be modeled by creating independent keyframes, followed by the creation of inbetween cells interpolating the key cells. For other potential applications, such as the vectorization of existing animations or video clips, we expect that the creation of the VAC may be automated.

**Editing** Creating the space-time topology of a complex animation may take more time than via traditional animation but once created, the VAC offers significant benefits as it provides a compact representation that is continuous in space and time. The VAC can be easily edited in ways that are not possible with traditional 2D or 3D animation pipelines. The VAC also provides a compact and convenient representation for algorithms to operate on. For example, we envision algorithms that can produce rich variations of an existing animated drawing by adding stochastic perturbations in space and time to some of the key elements.

**Local keyframing** Conventional keyframing animation allows for independent keyframing of the animation variables, i.e., the keyframe times for an animated knee-joint motion can be different from the keyframe times for the animated ankle motion within the same animation. Similarly, the VAC allows for the asynchronous specification of keyframes for portions of the vector graphics complex. Local keyframes provide better support for the semantics of many vector graphics drawings by allowing different portions of a drawing to be governed by different keyframes. It also allows for many topological changes to be conveniently modeled using instantiated templates.

**Repurposing of existing 3D complexes** It is tempting to believe that modeling an animated 2D complex could be achieved using existing approaches for 3D topological modeling, where the  $z$ -coordinate simply plays the role of time. Unfortunately, this does not reflect the unique semantics of the time axis, and this manifests itself in several ways. An “out of plane” rotation of a vector graphics animation does not usually produce a valid animation because the space is not Euclidean. For similar reasons, others have proposed representing image spaces as a non-Euclidean, Cayley-Klein geometry with one isotropic dimension [Koenderink and Doorn 2002]. Without a special designation for time, specific strategies would be needed to model the changing depth-orderings that can be desired during the course of a vector graphics animation, and which, by contrast, are easily modeled using the VAC. More importantly, cells would not always admit a time-parameterization. By contrast, all cells in our complex have an explicit time-parameterization, by design. This makes extracting time-slices trivial and also guarantees that all topological events are constrained to occur at key cells. This would not be the case if our cells were allowed to do “switch-backs” in time. In addition, despite being both 1D in space-time, the distinction we make between key edges and inbetween vertices is critical since their intersection with a time-plane is an object of different dimension. They must thus be rendered differently and store different attributes. The same is true for key faces and inbetween edges. Also, we allow zero-length edges but not zero-duration inbetween cells, i.e., we enforce  $t_1 < t_2$ . Similarly, paths are allowed to be reduced to a single key vertex, while animated vertices are not.

Using a simplicial complex representation for vector graphics animation [Southern 2008] would

necessitate the use of many cells, which could then be problematic for creation, editing, and visualization, as well as being further removed from the standard keyframing paradigm for animation. Given a simplicial complex that completely reflects the geometry of an VAC, the VAC can be seen as inducing a partition of the simplicial complex, resulting in an output semantics similar to [Buchholz et al. 2011]. In general, geometric complexes allow for models and operations that we wish to forbid in order to reflect the unique nature of the time dimension. Implementing the desired constraints necessitates additional complexity while the VAC implements the desired constraints by design, i.e., as part of its *desiderata*. Also, we note that the intersection of a 3D simplicial complex with a time-plane is not necessarily a 2D simplicial complex (as the intersection between a tetrahedron and a plane can be a four-sided polygon). By contrast, the intersection of a VAC with a time-plane is guaranteed by design to be a VGC, which is trivial to compute due to the explicit time-parameterization.

**Limitations** While there are many benefits to a structure that provides a sound, continuous-time model of the topological events in vector graphics animations, it also comes with additional complexity. In particular, the modeling and editing of *animated cycles*, as required in order to animate faces in the vector graphics complex, embodies much of the complexity of the data structure and its implementation. The space-time topology is also likely to introduce a steep learning curve for artists coming from the world of SVG models where changes in topology are approximated by other means. We currently leave the development of an improved user interface as future work, and as such we have not conducted a formal user study with regard to how end users can best work with the VAC. We believe that the use of topological-event templates may significantly simplify the workflow for modeling and editing. Finally, our system shares the same fundamental limitation of any 2D animation system: the loss of information between the depicted 3D world and the 2D depiction [Catmull 1978]. In other words, the semantics of a rotating 3D object will always be better captured by 3D representations. We believe that the automatic computation of a VAC from an animated 3D model would alleviate this issue.

**Future work** The VAC opens up a number of exciting avenues for future work. Computing aesthetically pleasing interpolation between key cells is a rich and interesting problem. In conventional animation systems, animation curves are defined for any animation variable by keyframes that always have well-defined *before* and *after* keyframes. This allows for well-defined tangent vectors to be specified or inferred at keyframes (e.g., Catmull-Rom). However, the topological events allowed by the VAC means that a key cell can have multiple *before* and *after* key cells, e.g., two or more vertices that join or split at a given time  $t$ , or an entire edge or face that merges or spawns from a given vertex. Developing sound and practical methods for position interpolation or user-based tangency specification is significantly more complex as a result.



Future work is needed to provide high-level manipulation of the VAC. For instance, a space-time paint bucket tool would be useful for creating inbetween faces. The automatic computation of inbetween cells from a general selection of key cells (i.e., automatic inbetweening) is a largely open problem, and extending [Whited et al. 2010] to the VAC is an exciting direction to explore. Also, we have developed a number of visualization tools in support of end user understanding, but much more is possible.

The topological structures could be further extended to allow the creation of motion graphs (equivalently, “move trees”), as is commonly done within game engines for character animation. This would require the ability to follow a given time-indexed “branch” of the VAC, and to rejoin existing branches. The ability to do this with local parts of a VGC would provide even further flexibility, although the resulting complexity might be difficult to develop and debug. One could also imagine creating additional continuous dimensions, such as that created by an aspect graph, i.e., creating a model that is parameterized with respect to the viewing direction as well as time.

An interesting direction is to develop VACs directly from video or rendering of a 3D model. VACs could be used to achieve continuous *space-time tracking*, as a logical extension of keyframe tracking for rotoscoping applications [Agarwala et al. 2004]. Interesting initial steps towards the vectorization of video have recently been explored [Patterson et al. 2012]. The data structure also has potential applications in non-photorealistic rendering, where there is a need for sound time-coherent models of the regions and strokes in an image sequence [Bénard et al. 2014]. Given the separation of topological and geometrical information in the VGC and the VAC, it should also be possible to develop a limited class of 3D animation using the VAC. Both of the above problems point to the need to develop good models for developing or otherwise modeling consistent parameterizations for edges and faces.

Some features supported by traditional vector graphics animation tools are not yet implemented, such as grouping, path-following, clipping, and masking. It is not yet clear how orthogonal this feature set is to the topological modeling aspects that we have focused on. Finally, there are interesting future directions to improve rendering and performance across the wide range of platforms that are a driving force behind increasing popularity of vector graphics.

## 5.8 Conclusion

We have presented a new data structure for representing vector graphics animation: the vector animation complex (VAC). It provides a compact, continuous-time continuous-space representation for vector graphics that is designed to support topological events. We expect that such continuous representations will become increasingly important as content needs to be developed for an ever-

wider range of display resolutions and frame rates. Compared to conventional representations for vector graphics animation, the VAC captures more faithfully the semantics of many animations, therefore provides better support for manual editing or algorithm processing. Local keyframing is supported, i.e., keyframes need only provide information about the topological or shape changes for the subset of parts that require a given change. Topological changes can be modeled where they are desired and can be avoided where they are more simply modeled using other means, such as depth layering.

We envision that the VAC may be used in a wide range of applications, including the traditional domains for vector graphics animations; traditional drawing-based 2D animation, and the image-processing pipelines that are part of video processing and non-photorealistic rendering applications.



## Chapter 6

### Conclusion



**Figure 6.1:** *Left: Example of figure that was challenging to author with current tools. Right: Exploded view of the figure revealing the 9 independent Bézier curves that had to be created in order to achieve the desired outcome. Not only determining how to achieve the desired depth-ordering was a puzzle on its own, but even once solved, any edit of the figure was made painfully complicated due to all pieces having to be edited independently, and carefully aligned to prevent any visible seam.*

In this thesis, first and foremost, we identified important shortcomings of current vector graphics representations. Indeed, despite vector graphics being a mature field with a 50-year-old history, none of the leading vector graphics file standards and applications fully support the representation of faces sharing a common edge, or edges sharing a common vertex, as surprising as it may seem. Artists frequently have to resort to tedious tricks to achieve their desired outcomes, and the resulting illustrations are hard to edit, and even harder to animate. As a matter of fact, the author of this dissertation experienced a lot of this frustration himself when creating many of the figures it contains, such as the one illustrated in Figure 6.1.

After looking back at the history of vector graphics and topological modeling, which we summarized in Chapter 2, it appeared to us that a possible cause of these shortcomings was a lack of theoretical foundations of vector graphics topology, which could have helped the vector graphics community to design representations supporting topological modeling. In order to alleviate this issue, we developed such theoretical foundations in Chapter 3, the take-home message being that it is critical to consider vector graphics illustrations as topological spaces *immersed* in  $\mathbb{R}^2$  (instead of simply *embedded* in  $\mathbb{R}^2$ ), which has direct consequences on what data structures can be used to represent them. In Chapter 4, we introduced such possible data structure which we called the *vector graphics complex*. We expect that the definition of this data structure will have a positive influence on upcoming standards, as we believe that the ability to model incidence relationships between vector graphics shapes is not only desirable, but in fact paramount for user experience, as already stressed in the abstract of Sutherland’s PhD thesis [1963] who pioneered the field, and as suggested by the positive feedback we received from the users of our prototype.

Of course, there still remain important open problems to be solved before the technology is ready for standardization. In particular, there are many open questions related to styling and rendering for which we do not have a clear answer yet. For example, even though vector graphics complexes have the ability to *model*  $n$ -way joins between edges, it is still unclear how they should be rendered, especially if edges are allowed to have variable width. In fact, it is already a non-trivial question for 2-way joins, a case in point being the recent introduction [SVG Working Group 2017] of new join styling attributes in SVG 2.0 to address some of the shortcomings of the prior SVG 1.1 specification.

Besides the many advantages of topological modeling to author static illustrations, this paradigm particularly shines when authoring vector graphics animations. Indeed, there is no better way to ensure that two objects stay connected throughout an animation than to explicitly encode in the representation that they must stay connected, and vector graphics complexes are a perfect representation to encode such connections. In order to develop these ideas further, we introduced in Chapter 5 the concept of *vector animation complexes*, which not only allow the representation of animated vector graphics complexes, but also allow the *topology* of these complexes to change over time. With web animation becoming increasingly popular, and the desire of web artists and developers to have better tools to author such animations, we believe that these types of representations have the potential to be widely adopted by the community.

In this thesis, we focused on solving problems related to vector graphics and hand-drawn 2D animation, but we believe that many of the ideas that have been developed are in fact applicable to a wide range of domains. Among those we can cite non-photorealistic rendering, interactive data visualization, games, geographic information systems, medical imaging, or representations of geological layers, all of which share the need to represent incidence relationships between objects, and possibly represent time-continuous changes of this topology. However, applying the ideas of this thesis to these domains may require more work. For instance, for non-photorealistic rendering, one would need to develop a conversion from 3D mesh silhouettes to vector graphics complexes, which is non-trivial. For interactive data visualization or games, one may need to parameterize a vector graphics complex according to user actions instead of time as we did in Chapter 5. All of these are exciting directions for future work.

Finally, we would like to conclude this dissertation with important thoughts on how the proposed techniques could be evaluated. Indeed, despite an informal positive reception from our users, the general usefulness of the technique remains largely to be proven. A possible approach would be to perform a formal user study where users are asked to perform the same set of tasks using existing tools versus using our tool. However, it is hard if not impossible to objectively design a set of tasks that tells us anything truly conclusive. Using our tool, users would quite obviously perform better on tasks where our structure excels, that is, editing an illustration or an animation featuring

shared boundaries. On the other hand, users would quite obviously perform worse on tasks where our current implementation is too limited compared to existing professional tools, for instance, drawing a complex unstructured illustration with subtle brush styles. Therefore, we believe that the only objective and reliable way to prove the usefulness of the method is to monitor its adoption over decades. Early feedback suggests that such adoption is likely to happen, but only time will tell. In order to make the tool more attractive to use for artists, one would need to implement the large feature set which is commonly found in other professional packages. Some of these features are orthogonal with our proposed approach and therefore not harder to implement than in the existing packages (e.g., layers, text, groups, masks, alignment tools, symmetry mode, cloned instances, sound), while others may require some work to make them compatible with our data structure (e.g., join styles, Bézier curve tangents, keyframe tangents).

# Bibliography

- ADOBE SYSTEMS INC., 2013. Adobe Illustrator: Help and tutorials.
- AGARWALA, A., HERTZMANN, A., SALESIN, D. H., AND SEITZ, S. M. 2004. Keyframe-based tracking for rotoscoping and animation. *ACM Trans. Graph.* 23, 3, 584–591.
- ALEXA, M., COHEN-OR, D., AND LEVIN, D. 2000. As-rigid-as-possible shape interpolation. In *Proceedings of SIGGRAPH 2000*, 157–164.
- ASENTE, P., SCHUSTER, M., AND PETTIT, T. 2007. Dynamic planar map illustration. *ACM Trans. Graph.* 26, 3, 30:1–30:10.
- BAUDELAIRE, P., AND GANGNET, M. 1989. Planar maps: An interaction paradigm for graphic design. In *Proceedings of CHI '89*, 313–318.
- BAUMGART, B. G. 1972. Winged edge polyhedron representation. Tech. rep., DTIC Document.
- BAXTER, W., BARLA, P., AND ANJYO, K.-I. 2009. Compatible embedding for 2d shape animation. *IEEE Trans. on Visualization and Computer Graphics* 15, 5, 867–879.
- BÉNARD, P., LU, J., COLE, F., FINKELSTEIN, A., AND THOLLOT, J. 2012. Active strokes: Coherent line stylization for animated 3d models. In *Proceedings of NPAR '12*, 37–46.
- BÉNARD, P., HERTZMANN, A., AND KASS, M. 2014. Computing smooth surface contours with accurate topology. *ACM Trans. Graph.* 33, 2, 19:1–19:21.
- BLAIR, P. 1994. *Cartoon animation*. How to Draw and Paint Series. W. Foster Pub.
- BREGLER, C., LOEB, L., CHUANG, E., AND DESHPANDE, H. 2002. Turning to the masters: Motion capturing cartoons. *ACM Trans. Graph.* 21, 3, 399–407.
- BRISSON, E. 1989. Representing geometric structures in d dimensions: Topology and order. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, ACM, New York, NY, USA, SCG '89, 218–227.
- BUCHHOLZ, B., FARAJ, N., PARIS, S., EISEMANN, E., AND BOUBEKEUR, T. 2011. Spatio-temporal analysis for parameterizing animated lines. In *Proceedings of NPAR '11*, 85–92.
- BURTNYK, N., AND WEIN, M. 1971. Computer-generated key-frame animation. *Journal of the Society of Motion Picture & Television Engineers* 80, 3, 149–153.
- CAPEN, A., SEVERTSON, J., HEMPHILL, T., AND COWLES, D., 2014. The Adobe Illustrator Story. <https://vimeo.com/95415863>, May. [Online; retrieved 29-July-2016].

- CARLSON, W., 2003. A critical history of computer graphics and animation. <https://design.osu.edu/carlson/history/lessons.html>. [Online; retrieved 21-July-2016].
- CATMULL, E., AND WALLACE, A. 2014. *Creativity, Inc.: Overcoming the Unseen Forces That Stand in the Way of True Inspiration*. Random House of Canada.
- CATMULL, E. 1978. The problems of computer-assisted animation. *SIGGRAPH Comput. Graph.* 12, 3, 348–353.
- COONS, S. A. 1967. Surfaces for computer-aided design of space forms. Tech. rep., Massachusetts Institute of Technology.
- 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.
- DALSTEIN, B., RONFARD, R., AND VAN DE PANNE, M. 2014. Vector graphics complexes. *ACM Trans. Graph.* 33, 4, 133:1–133:12.
- DALSTEIN, B., RONFARD, R., AND VAN DE PANNE, M. 2015. Vector graphics animation with time-varying topology. *ACM Trans. Graph.* 34, 4 (July), 145:1–145:12.
- DAMIAND, G., AND LIENHARDT, P. 2014. *Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing*. CRC Press.
- 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*, 403–420.
- DE JUAN, C. N., AND BODENHEIMER, B. 2006. Re-using traditional animation: methods for semi-automatic segmentation and inbetweening. In *Proceedings of SCA '06*, 223–232.
- DEHN, M., AND HEEGAARD, P. 1907. Analysis situs. In *Enzyklopädie der Math. Wiss.* III.1.1. 153–220.
- DOBRINDT, K., MEHLHORN, K., AND YVINEC, M. 1993. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. In *Workshop on Algorithms and Data Structures*, Springer, 314–324.
- 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.
- EDMONDS, J. 1960. A combinatorial representation for polyhedral surfaces. *Notices of the American Mathematical Society* 7.

- 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, 83:1–83:8.
- ELTER, H., AND LIENHARDT, P. 1992. Extension of the notion of map for the representation of the topology of cellular complexes. In *4th Canadian Conference on Computational Geometry*.
- ELTER, H., AND LIENHARDT, P. 1993. Different combinatorial models based on the map concept for the representation of subsets of cellular complexes. In *Modeling in Computer Graphics*. 193–212.
- ELTER, H., AND LIENHARDT, P. 1994. Cellular complexes as structured semi-simplicial sets. *International Journal of Shape Modeling* 1, 02, 191–217.
- FAUSETT, E., PASKO, A., AND ADZHIEV, V. 2000. Space-time and higher dimensional modeling for animation. In *Proceedings of Computer Animation 2000*, 140–145.
- FAVREAU, J.-D., LAFARGE, F., AND BOUSSEAU, A. 2016. Fidelity vs. simplicity: A global approach to line drawing vectorization. *ACM Trans. Graph.* 35, 4 (July), 120:1–120:10.
- FEKETE, J.-D., BIZOUARN, E., COURNARIE, E., GALAS, T., AND TAILLEFER, F. 1995. TicTacToon: A paperless system for professional 2D animation. In *Proceedings of SIGGRAPH 95*, 79–90.
- FIGLIORE, F. D., SCHAEKEN, P., ELEN, K., AND REETH, F. V. 2001. Automatic in-betweening in computer assisted animation by exploiting 2.5D modelling techniques. In *Proceedings of Computer Animation 2001*, 192–200.
- FOLEY, J. D., AND VAN DAM, A. 1982. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- FU, H., TAI, C.-L., AND AU, O. K.-C. 2005. Morphing with laplacian coordinates and spatial-temporal texture. In *Proceedings of Pacific Graphics 2005*, 100–102.
- GALE, D. 1987. The Classification of 1-Manifolds: A Take-Home Exam. *The American Mathematical Monthly* 94, 2, 170–175.
- 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*, Elsevier, Amsterdam, 107–130.
- HATCHER, A. 2001. *Algebraic Topology*.
- 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.
- IGARASHI, T., MOSCOVICH, T., AND HUGHES, J. F. 2005. As-rigid-as-possible shape manipulation. *ACM Trans. Graph.* 24, 3, 1134–1141.
- INKSCAPE, 2013. <http://www.inkscape.org/en/>. [Online; retrieved 01-July-2017].
- KARSCH, K., AND HART, J. C. 2011. Snaxels on a plane. In *Proceedings of NPAR '11*, 35–42.
- KETTNER, L. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Comput. Geom. Theory Appl* 13, 65–90.
- KOENDERINK, J. J., AND DOORN, A. J. v. 2002. Image processing done right. In *Proceedings of the 7th European Conference on Computer Vision*, 158–172.
- KOENDERINK, J., AND DOORN, A. 2008. The structure of visual spaces. *Journal of Mathematical Imaging and Vision* 31, 2-3, 171–187.
- KORT, A. 2002. Computer aided inbetweening. In *Proceedings of NPAR '02*, 125–132.
- KRULL, F. N. 1994. The origin of computer graphics within general motors. *IEEE Ann. Hist. Comput.* 16, 3 (Sept.), 40–56.
- KWARTA, V., AND ROSSIGNAC, J. 2002. Space-time surface simplification and edgebreaker compression for 2D cel animations. *International Journal of Shape Modeling* 8, 2, 119–137.
- LASSETER, J. 1987. Principles of traditional animation applied to 3d computer animation. *SIGGRAPH Comput. Graph.* 21, 4, 35–44.
- 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.
- LEE, J. M. 2011. *Introduction to Topological Manifolds*. Springer New York.
- LEVY, B., AND MALLET, J.-L. 1999. Cellular modelling in arbitrary dimension using generalized maps. Tech. rep., Technical report, ISA-GOCAD (Inria-Lorraine/CNRS).
- LIENHARDT, P. 1989. Subdivisions of n-dimensional spaces and n-dimensional generalized maps. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, ACM, New York, NY, USA, SCG '89, 228–236.
- LIENHARDT, P. 1991. Topological models for boundary representation: A comparison with n-dimensional generalized maps. *Comput. Aided Des.* 23, 1 (Feb.), 59–82.
- LIENHARDT, P. 1994. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *International Journal of Computational Geometry & Applications* 04, 03, 275–324.
- LIU, D., CHEN, Q., YU, J., GU, H., TAO, D., AND SEAH, H. S. 2011. Stroke correspondence construction using manifold learning. *Computer Graphics Forum* 30, 8, 2194–2207.

- MARCHEIX, D., AND GUEORGUEVA, 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.
- MASSON, T. 1999. *CG 101: A Computer Graphics Industry Reference*. 3D Graphics Other Series. New Riders.
- MCCANN, J., AND POLLARD, N. 2009. Local layering. *ACM Trans. Graph.* 28, 3, 84:1–84:7.
- MIT LINCOLN LABORATORY, 1964. Ivan Sutherland : Sketchpad demo. [https://www.youtube.com/watch?v=USyoT\\_Ha\\_bA](https://www.youtube.com/watch?v=USyoT_Ha_bA). [Online; retrieved 05-March-2017].
- MOISSINAC, J.-C. 2010. SuperPath (vePath): A necessary primitive for vector graphic formats. In *Proceedings of the 8th International Conference on Scalable Vector Graphics*.
- MUNKRES, J. 2000. *Topology*. Featured Titles for Topology Series. Prentice Hall, Incorporated.
- NEF, W. 1978. *Beiträge zur Theorie der Polyeder: mit Anwendungen in der Computergraphik*. Beiträge zur Mathematik, Informatik und Nachrichtentechnik. Lang.
- NGO, T., CUTRELL, D., DANA, J., DONALD, B., LOEB, L., AND ZHU, S. 2000. Accessible animation and customizable graphics via simplicial configuration modeling. In *Proceedings of SIGGRAPH 2000*, 403–410.
- NORIS, G., SÝKORA, D., COROS, S., WHITED, B., SIMMONS, M., HORNUNG, A., GROSS, M., AND SUMNER, R. W. 2011. Temporal noise control for sketchy animation. In *Proceedings of NPAR '11*, 93–98.
- 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, 4:1–4:11.
- ORZAN, A., BOUSSEAU, A., WINNEMÖLLER, H., BARLA, P., THOLLOT, J., AND SALESIN, D. 2008. Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph.* 27, 3, 92:1–92:8.
- PATTERSON, J. W., TAYLOR, C. D., AND WILLIS, P. J. 2012. Constructing and rendering vectorised photographic images. *The Journal of Virtual Reality and Broadcasting* 9, 3.
- 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.
- RAVEENDRAN, K., WOJTAN, C., THUEREY, N., AND TURK, G. 2014. Blending liquids. *ACM Trans. Graph.* 33, 4, 137:1–137:10.
- REEVES, W. T. 1981. Inbetweening for computer animation utilizing moving point constraints. *SIGGRAPH Comput. Graph.* 15, 3, 263–269.
- REILLY, E. 2003. *Milestones in Computer Science and Information Technology*. Greenwood Press.
- RIVERS, A., IGARASHI, T., AND DURAND, F. 2010. 2.5D cartoon models. *ACM Trans. Graph.* 29, 4, 59:1–59:7.



- ROSSIGNAC, J., AND O'CONNOR, M. 1989. *SGC: A Dimension-independent Model for Pointsets with Internal Structures and Incomplete Boundaries*. Research report. IBM T.J. Watson Research Center.
- 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.
- SEBASTIAN, T. B., KLEIN, P. N., AND KIMIA, B. B. 2003. On aligning curves. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 25, 1, 116–125.
- SEDERBERG, T. W., GAO, P., WANG, G., AND MU, H. 1993. 2-D shape blending: an intrinsic solution to the vertex path problem. In *Proceedings of SIGGRAPH 93*, 15–18.
- SHIRLEY, P., AND MARSCHNER, S. 2009. *Fundamentals of Computer Graphics*. Taylor & Francis.
- 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.
- SOUTHERN, R. 2008. Animation manifolds for representing topological alteration. Tech. Rep. UCAM-CL-TR-723, University of Cambridge, Computer Laboratory.
- SUN, J., LIANG, L., WEN, F., AND SHUM, H.-Y. 2007. Image vectorization using optimized gradient meshes. *ACM Trans. Graph.* 26, 3 (July).
- SUTHERLAND, I. E. 1963. *Sketchpad, a man-machine graphical communication system*. PhD thesis, Massachusetts Institute of Technology.
- SVG WORKING GROUP, 2011. Scalable Vector Graphics (SVG) 1.1 (Second Edition). <http://www.w3.org/TR/SVG11/>. [Online; retrieved 01-July-2017].
- SVG WORKING GROUP, 2016. Scalable Vector Graphics (SVG) 2.0 (W3C Candidate Recommendation 15 September 2016). <https://www.w3.org/TR/2016/CR-SVG2-20160915/>. [Online; retrieved 01-August-2016].
- SVG WORKING GROUP, 2017. Controlling line joins: the 'stroke-linejoin' and 'stroke-miterlimit' properties. <https://www.w3.org/TR/SVG2/painting.html#LineJoin>. [Online; retrieved 27-March-2017].
- SÝKORA, D., DINGLIANA, J., AND COLLINS, S. 2009. As-rigid-as-possible image registration for hand-drawn cartoon animations. In *Proceedings of NPAR '09*, 25–33.
- SÝKORA, D., BEN-CHEN, M., ČADÍK, M., WHITED, B., AND SIMMONS, M. 2011. TexToons: practical texture mapping for hand-drawn cartoon animations. In *Proceedings of NPAR '11*, 75–84.
- 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.
- THOMAS, F., AND JOHNSTON, O. 1987. *Disney Animation: The Illusion of Life*. Abbeville Press.
- WEILER, K. 1985. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications* 5, 1, 21–40.

- WEILER, K. 1986. *Topological Structures for Geometric Modeling*. PhD thesis, Rensselaer Polytechnic Institute.
- WHITED, B., NORIS, G., SIMMONS, M., SUMNER, R., 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.
- WILLIAMS, R. 2009. *The Animator's Survival Kit*. Faber and Faber.
- YU, J., BIAN, W., SONG, M., CHENG, J., AND TAO, D. 2012. Graph based transductive learning for cartoon correspondence construction. *Neurocomputing* 79, 0, 105–114.
- ZHANG, S.-H., CHEN, T., ZHANG, Y.-F., HU, S.-M., AND MARTIN, R. R. 2009. Vectorizing cartoon animations. *IEEE Trans. on Visualization and Computer Graphics* 15, 4, 618–629.

# Index

- abstract cell 63
- abstract edge 63
- abstract face 63
- abstract PCS complex 63
- abstract topology 63
- abstract vertex 63
- after 119
- after cycle 114
- after face 116
- after path 115
- after time 116
- after vertex 114
- animated curve 114, 115
- animated cycle 109, 116, 117
- animated position 114
- animated vertex 109, 117
- attribute 112
- before 119
- before cycle 114
- before face 116
- before path 115
- before time 116
- before vertex 114
- boundary 71
- cell 3, 47, 64, 80, 82, 112, 119
- cell-tuple 26
- characteristic manifold 64
- closed edge 3, 51, 63, 81, 83
- closed halfedge 63, 83, 116
- closedness 63, 83
- combinatorial map 24
- consistent parameterization 69
- continuous 41
- curve 113
- CW complex 28
- cycle 3, 64, 81, 83, 109, 114, 117
- dimension 63, 82
- direction 119
- disjoint union 65
- edge 82, 112
- end animated vertex 115
- end vertex 113
- $\epsilon$ - $g$ - $k$ -face 54
- face 3, 81, 82, 112
- face-cut classification 60
- generalized map 25
- genus 63
- geometric realization 42, 46, 64, 66
- geometrical attribute 112
- graph 39
- halfedge 4, 63, 81, 116
- halfedge data structure 19
- homeomorphism (graphs) 40
- homeomorphism (PCS compl.) 70
- homeomorphism (topol. spaces) 41
- immersion 46
- inbetween cell 112
- inbetween closed edge 109, 112, 114

inbetween face 109, 112, 115  
 inbetween open edge 108, 112, 115  
 inbetween vertex 107, 112, 114  
 incidence graph 71, 86  
 invariant 112, 119  
 isomorphism (graphs) 40  
 isomorphism (PCS complexes) 64, 70  
 isomorphism (polygonal meshes) 44  
 key cell 112  
 key closed edge 109, 112, 113  
 key face 109, 112, 114  
 key open edge 108, 112, 113  
 key vertex 107, 112, 113  
 next 119  
 node 117  
 non-equivalent cuts 72  
 non-planar face 53  
 non-simple cycle 64, 83  
 number of cycles 83  
 number of holes 63  
 open edge 3, 63, 80, 83  
 open halfedge 63, 83, 116  
 open set 40  
 ordered boundary 63, 83  
 orientability 63  
 overlapping 46, 47  
 partial keyframing 108  
 path 108, 116  
 PCS complex 64  
 planar map 14  
 point 40  
 position 113  
 presentation 46  
 previous 119  
 quad-edge data structure 19  
 quotient space 46  
 radial-edge data structure 21  
 raster graphics 8  
 selective geometric complex 29  
 sequential keyframing 106  
 simple cycle 64, 83  
 simplicial complex 27  
 spatial boundary 108  
 spatial dimension 112  
 star 4, 91  
 start animated vertex 115  
 start vertex 113  
 starting node 119  
 starting point 117  
 Steiner cycle 64, 81, 83  
 Steiner vertex 56  
 stroke graph 15  
 SVG representation 13  
 temporal boundary 108  
 temporal dimension 112  
 time 113, 114  
 timespan 119  
 topological attribute 112  
 topological keyframing 107  
 topological modeling 2, 8, 16  
 topological operator 44, 71, 92  
 topological space 40  
 topological structure 47  
 topology 38, 40  
 type 51, 52, 112  
 vector graphics 1, 8  
 vector graphics complex 72, 82  
 vertex 3, 82, 112  
 winding number 86  
 winding rule 85, 114, 116  
 winged-edge data structure 18

# Appendix A

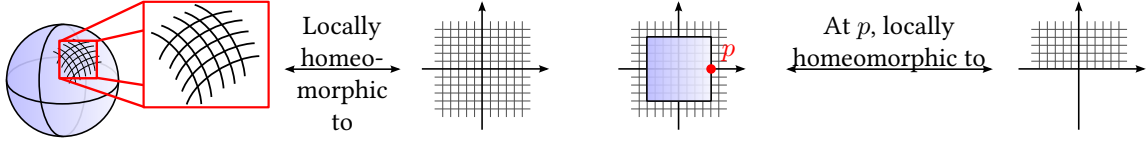
## Concepts of Algebraic Topology

In this appendix, we review a few concepts of algebraic topology which are relevant to this dissertation. These concepts are by no means a prerequisite for understanding most of the dissertation (in fact, the author himself only has a superficial understanding of some of them), but they were an important source of inspiration, and we believe they may provide useful insight to interested readers. For readability, we only provide informal definitions of most of these concepts. Formal definitions can be found in [Lee 2011, Hatcher 2001].

### A.1 Topological Spaces and Homeomorphisms

In this dissertation, whenever we say **topological space**, we mean a Hausdorff topological space. Intuitively, a (general) topological space is a set, for instance  $X = \{x \in \mathbb{R} \mid 0 < x < 5\}$ , together with a definition of **open subsets** that has to satisfy a few properties that capture the notion of “openness”. These properties are given in Section 3.1.2, but informally, it is simply a generalization of the intuitive concepts of open intervals and closed intervals. For instance,  $I = (1, 2) = \{x \in \mathbb{R} \mid 1 < x < 2\}$  is an open subset of  $X$ , while  $J = [1, 2) = \{x \in \mathbb{R} \mid 1 \leq x < 2\}$  is not an open subset of  $X$ . For any point  $x \in X$ , a **neighborhood** of  $x$  is defined as any open subset  $N_x \subset X$  that contains  $x$ . Finally, a *Hausdorff* topological space is a topological space that satisfies the additional property that any two points  $x_1$  and  $x_2$  are *separable* by open sets, that is, there exists neighborhoods of  $x_1$  and  $x_2$  that do not intersect. In practice, most “reasonable” spaces are indeed Hausdorff topological spaces. For instance,  $\mathbb{R}^n$  is a Hausdorff topological space, and any subset of  $\mathbb{R}^n$  is a Hausdorff topological space as well.

Two topological spaces  $X$  and  $Y$  are said to be **homeomorphic**, denoted  $X \cong Y$ , if and only if there exists a **homeomorphism** between  $X$  and  $Y$ , that is, an invertible continuous function  $\phi : X \rightarrow Y$  whose inverse  $\phi^{-1}$  is also continuous.  $X \cong Y$  captures the intuitive concept of “ $X$  and  $Y$  are essentially the same topological space”, meaning that even though they are not necessarily “equal”, they behave similarly and have a similar “shape”. For instance, the two closed intervals  $I_1 = [0, 1]$  and  $I_2 = [1, 2]$  are not equal but they are homeomorphic. The two open intervals  $J_1 = (0, 1)$  and  $J_2 = (1, 2)$  are not equal but they are homeomorphic. Also, the two circles



**Figure A.1:** The sphere  $\mathbb{S}^2 = \{x \in \mathbb{R}^3, \|x\| = 1\}$  is a 2-manifold without boundary, since it is everywhere locally homeomorphic to  $\mathbb{R}^2$ .

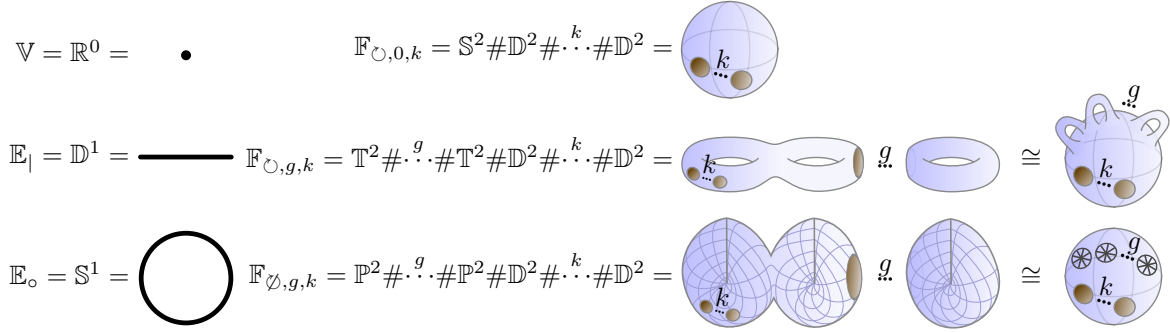
**Figure A.2:** The surface  $[-1, 1] \times [-1, 1]$  is a 2-manifold with boundary, since it is locally homeomorphic either to  $\mathbb{R}^2$  or to  $\mathbb{R} \times [0, +\infty)$ .

$S_1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$  and  $S_1 = \{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 2\}$  are not equal but they are homeomorphic. However, none of  $I_1$ ,  $J_1$  or  $S_1$  are homeomorphic to each other, which is the formal way of saying: “they do not look alike”. This informal statement is so intuitive that we have different names for these objects (closed intervals, open intervals, circles), while there is no terminology to differentiate, say,  $I_1$  and  $I_2$ . Identifying that a topological space  $X$  is homeomorphic to a known topological space  $Y$  is of primary importance because it makes possible to infer properties of  $X$  from the known properties of  $Y$ .

## A.2 Manifolds with Boundary and Compact Manifolds

An ***n*-manifold without boundary**  $\mathbb{M}$  is a topological space that is everywhere locally homeomorphic to  $\mathbb{R}^n$ . More formally,  $\mathbb{M}$  is an *n*-manifold without boundary if and only if for each  $p \in \mathbb{M}$ , there exists a neighborhood  $N_p$  homeomorphic to  $\mathbb{R}^n$ . For instance, the sphere  $\mathbb{S}^2$  is a 2-manifold without boundary because for each point  $p$  on the sphere, it “looks locally like” the plane, as illustrated in Figure A.1. However, the square  $[-1, 1] \times [-1, 1] \subset \mathbb{R}^2$  is *not* a 2-manifold without boundary because at  $p = (1, 0)$ , it is locally homeomorphic to  $\mathbb{R} \times [0, +\infty)$ , as illustrated in Figure A.2. Since we want this last example to be included in our definition of manifold, we use a more general definition: an ***n*-manifold with boundary** is defined as a topological space that is everywhere locally homeomorphic either to  $\mathbb{R}^n$  or to  $\mathbb{H}^n = \mathbb{R}^{n-1} \times [0, +\infty)$ . In this dissertation, whenever we say **manifold**, we mean manifold with boundary, unless *without boundary* is explicitly stated. If  $\mathbb{M}$  is an *n*-manifold, then the **interior** of  $\mathbb{M}$ , denoted  $\text{int}(\mathbb{M})$ , are the points of  $\mathbb{M}$  that have a neighborhood homeomorphic to  $\mathbb{R}^n$ . Conversely, the **boundary** of  $\mathbb{M}$ , denoted  $\partial\mathbb{M}$ , are the points of  $\mathbb{M}$  that have a neighborhood homeomorphic to  $\mathbb{H}^n$ , i.e.  $\partial\mathbb{M} = \mathbb{M} \setminus \text{int}(\mathbb{M})$ .

A **compact manifold**  $\mathbb{M}$  is a manifold that is compact as a topological space. A general definition can be found in [Lee 2011, Chapter 4], but in the specific case where  $\mathbb{M}$  is a subset of  $\mathbb{R}^n$ , then being compact is equivalent to being bounded and topologically closed in  $\mathbb{R}^n$  (i.e.,  $\mathbb{R}^n \setminus \mathbb{M}$  is open). For instance, the closed interval  $[0, 1]$  is a compact manifold, but the real line  $\mathbb{R}$  is not a compact manifold because it is not bounded, and the open interval  $(0, 1)$  is not a compact manifold because it is not closed in  $\mathbb{R}$ . If  $\mathbb{M}$  is a compact *n*-manifold, then  $\partial\mathbb{M}$  is a compact  $(n - 1)$ -manifold but



**Figure A.3:** The classification of points, curves and surfaces. Any connected compact  $n$ -manifold with  $n \leq 2$  is homeomorphic to one and only one of these known compact manifolds. The notations  $\mathbb{V}$ ,  $\mathbb{E}_{|}$ ,  $\mathbb{E}_{\circ}$ ,  $\mathbb{F}_{\cup,g,k}$ , and  $\mathbb{F}_{\emptyset,g,k}$  are non-standard and introduced for conciseness and clarity. They are the characteristic manifolds for, respectively: vertices, open edges, closed edges, orientable faces, and non-orientable faces.

$\text{int}(\mathbb{M})$  is an  $n$ -manifold generally *not* compact. More specifically,  $\text{int}(\mathbb{M})$  is compact iff  $\partial\mathbb{M} = \emptyset$  (i.e., iff  $\mathbb{M}$  is a compact  $n$ -manifold without boundary).

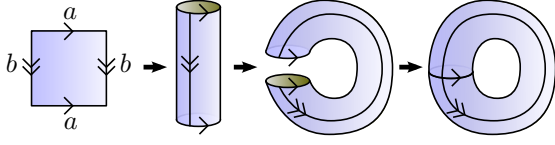
### A.3 Points, Curves, and Surfaces

Compact manifolds are important because their properties capture the intuitive concepts of points, curves, and surfaces. More specifically, a **point** is defined as a connected compact 0-manifold, a **curve** is defined as a connected compact 1-manifold, and a **surface** is defined as a connected compact 2-manifold. Being compact ensures that curves (resp. surfaces) have a finite, well-defined length (resp. area), and avoid having to deal with many pathological cases.

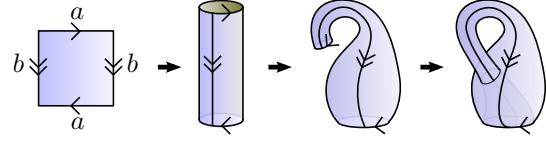
### A.4 Classification of Compact $n$ -Manifolds for $n \leq 2$

Compact manifolds of dimension two and lower, that is, points, curves, and surfaces, have been completely “classified”. This means that we know a very concise list of compact manifolds, illustrated in Figure A.3, such as any point, curve, or surface is necessarily homeomorphic to one and only one of the manifolds in the list. In other words, any surface “looks like” one and only one of the surfaces in the list. In this section, we recall this classification. Following common practice, we only consider here *connected* compact manifolds, but it is trivial to generalize to all compact manifolds since any compact  $n$ -manifold can be decomposed as a finite disjoint union of connected compact  $n$ -manifolds.

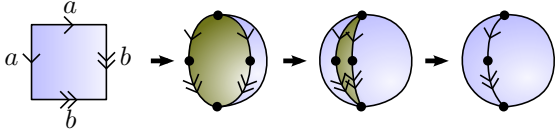
**Dimension 0** A point is always homeomorphic to  $\mathbb{R}^0 = \{0\}$ , that is, a set containing a unique element. In practice, we identify the set to the unique element itself.



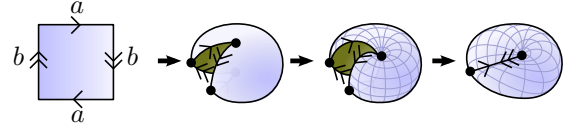
**Figure A.4:** A polygonal presentation of the torus is the single word  $W = aba^{-1}b^{-1}$ .



**Figure A.5:** A polygonal presentation of the Klein bottle is the single word  $W = abab^{-1}$ .



**Figure A.6:** A polygonal presentation of the sphere is the single word  $W = abb^{-1}a^{-1}$ .



**Figure A.7:** A polygonal presentation of the projective plane is the single word  $W = abab$ .

**Dimension 1** A curve is homeomorphic either to the unit circle  $\mathbb{S}^1$  or to the closed interval  $\mathbb{D}^1 = [0, 1]$ . A proof can be found in [Gale 1987] or in [Lee 2011, Ch. 5, p. 143-147].

**Dimension 2** A surface **without boundary** is homeomorphic to either:

- The sphere  $\mathbb{S}^2$ , called the **surface of genus 0**.
- The connected sum of  $g \geq 1$  tori  $(\mathbb{T}^2)^g = \mathbb{T}^2 \# \dots \# \mathbb{T}^2$ , called the **orientable surface of genus  $g$** .
- The connected sum of  $g \geq 1$  projective planes  $(\mathbb{P}^2)^g = \mathbb{P}^2 \# \dots \# \mathbb{P}^2$ , called the **nonorientable surface of genus  $g$** .

We clarify here the terminology. The **torus**  $\mathbb{T}^2$  is the topological space obtained by “gluing” together (or “sewing”) the opposite boundaries of a cylinder as depicted in Figure A.4. Note that direction matters: if you choose to glue using the reverse direction of one of the boundaries, as depicted in Figure A.5, you get the **Klein bottle**  $\mathbb{K}^2$  instead, which is not homeomorphic to the torus but to  $\mathbb{P}^2 \# \mathbb{P}^2$ . The **projective plane**  $\mathbb{P}^2$  is the topological space obtained by “gluing” the unique boundary of a disk to the unique boundary of a Möbius strip. Alternatively, as illustrated in Figure A.7, it can be obtained by gluing together one half of the boundary of a disk to the other half, using the appropriate direction. The **connected sum** of two surfaces consists in removing one disk from each surface, and gluing together the two obtained boundaries.

The classification of surfaces given above has been first proven in [Dehn and Heegaard 1907], and is nicely covered and illustrated in [Lee 2011, Ch. 6]. We recall below the high-level steps of this proof, which involves the concept of *polygonal presentations*, related to our concept of abstract PCS complexes introduced in Section 3.3.1.

- A **word**  $W$  is defined, given a set  $S$ , as a finite sequence of  $k \geq 1$  symbols, each of the form



$a$  or  $a^{-1}$  with  $a \in S$ . It represents a regular polygon with  $k$  edges, where some edges are identified in pairs with a chosen direction. For instance, the word  $W = abab^{-1}$  represents a square, where the first and third edges are identified with the same direction, and the second and fourth edges are identified with opposite directions (cf. Figure A.5, left). A **polygonal presentation**  $\mathcal{P}$  is defined as a set of words.

- The **geometric realization** of  $\mathcal{P}$ , denoted  $|\mathcal{P}|$ , is the topological space obtained by gluing together the paired edges of the polygons described by its words. For instance, the geometric realization of  $\mathcal{P} = \{abab\}$  is a torus (cf. Figure A.4). Other examples are given in Figure A.5, A.6, and A.7. Two polygonal presentations are said to be **topologically equivalent** if and only if their geometric realizations are homeomorphic.
- A **surface presentation** is defined as a polygonal presentation  $\mathcal{P}$  where each element  $a \in S$  occurs exactly twice. In this case, we can prove that  $|\mathcal{P}|$  is a compact 2-manifold without boundary. Conversely, it can be shown that any compact 2-manifold without boundary admits a surface presentation  $\mathcal{P}$ .
- Finally, combinatorial operations on surface presentations prove that any surface presentation is topologically equivalent to either:

- the canonical surface presentation of the sphere:

$$\mathcal{P} = \{aa^{-1}\} \tag{A.1}$$

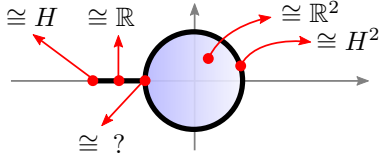
- the canonical surface presentation of the connected sum of  $g \geq 1$  tori:

$$\mathcal{P} = \{a_1b_1a_1^{-1}b_1^{-1} \dots a_gb_ga_g^{-1}b_g^{-1}\} \tag{A.2}$$

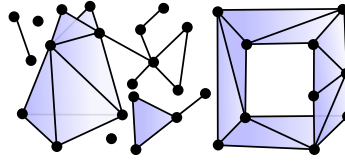
- the canonical surface presentation of the connected sum of  $g \geq 1$  projective planes:

$$\mathcal{P} = \{a_1a_1 \dots a_ga_g\} \tag{A.3}$$

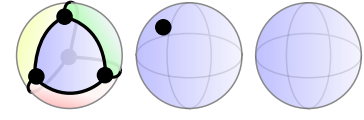
The classification above was only for surfaces *without boundary*. However, it turns out that surfaces with boundary can always be obtained from surfaces without boundary by removing the interior of  $k \geq 0$  disjoint closed disks. We illustrate this complete classification in Figure A.3. Finally, we recall an interesting theorem which is used in the proof of this classification, and is relevant to analyze our cut and uncut topological operators:



**Figure A.8:**  $X = \{(x, y) \mid x^2 + y^2 \leq 1\} \cup \{(x, 0) \mid x \in [-2, -1]\}$  is a non-manifold space: there exists no  $n$  such that  $X$  is everywhere locally homeomorphic to either  $\mathbb{R}^n$  or  $\mathbb{H}^n$ .



**Figure A.9:** A two-dimensional simplicial complex. The union of the simplices is in general a non-manifold space. Though, some connected components may be manifolds.



**Figure A.10:** Left: minimal simplicial decomposition of  $\mathbb{S}^2$  (14 simplices). Middle: minimal CW decomposition (2 CW-cells). Right: minimal PCS decomposition (1 PCS-cell).

**Theorem 1.** The connected sum of a projective plane and a torus is homeomorphic to the connected sum of three projective planes, i.e.:

$$\mathbb{P}^2 \# \mathbb{T}^2 \cong \mathbb{P}^2 \# \mathbb{P}^2 \# \mathbb{P}^2 \quad (\text{A.4})$$

## A.5 Non-Manifold Topological Spaces

Compact manifolds are very convenient to study, but unfortunately not all compact topological spaces are compact manifolds, as for instance the one illustrated in Figure A.8. To include this example and many others (but not *all* compact topological spaces, which would be too general), a wider class of topological spaces has been defined: those that can be obtained by “gluing together” simple manifold pieces. For instance, the topological space in Figure A.8 can be obtained by gluing a segment with a disk. Topological spaces defined within this general framework are commonly referred to as *complexes*, such as simplicial complexes and CW complexes, which we recall in this section. As noted in [Edelsbrunner and Harer 2010, Ch. III, p. 51], one of the most important characteristic that makes each kind of complex different from one another is how “simple” the glued pieces are. The simpler the pieces, the more pieces you need to decompose a given space (cf. Figure A.10). Therefore, choosing the right formalism to tackle a given topological problem is a trade-off between the complexity of each piece, and the number of pieces you need to decompose a given space. For instance, the pieces of a simplicial complex are called *simplices* and are  $n$ -dimensional triangles, while the pieces of a CW complex are called *cells* and are homeomorphic to an  $n$ -dimensional open disk. Thus, a simplex is a special case of a cell, meaning that the pieces of CW complexes are “more complex” than the pieces of simplicial complexes. As a consequence, the sphere can be decomposed with only two cells, while it requires 14 simplices. In this dissertation, we introduce the concept of PCS complex, whose pieces, also called cells for lack of a better name, are only required to be homeomorphic to the interior of a compact manifold. Thus, a “CW-cell” is a special case of a “PCS-cell”, meaning that the pieces of PCS complexes are “even more complex”

than the pieces of CW complexes, and can for instance decompose the sphere as a single PCS-cell.

### A.5.1 Abstract Simplicial Complexes

An abstract simplicial complex [Edelsbrunner and Harer 2010, p. 53] is a finite collection of sets  $A$  such that:

$$(\alpha \in A \text{ and } \beta \subseteq \alpha) \Rightarrow \beta \in A \quad (\text{A.5})$$

The elements  $\alpha$  in  $A$  are called **simplices**, and each simplex is given as the set of its **vertices**. The **dimension** of a simplex is  $\dim \alpha = \text{card } \alpha - 1$ , and the dimension of the complex is the maximum dimension of any of its simplices. Intuitively, a two-dimensional abstract simplicial complex is a triangle mesh made of vertices, edges and triangles (possibly non-manifold, with dangling edges or isolated vertices), as illustrated in Figure A.9. The formal definition ensures that if a triangle defined by the vertices  $\{0, 1, 2\}$  is part of the complex, then all the vertices  $\{0\}, \{1\}$ , and  $\{2\}$ , and all the edges  $\{0, 1\}, \{0, 2\}$ , and  $\{1, 2\}$  are also part of the complex, and they are called the **boundary simplices** of the triangle.

### A.5.2 CW Complexes

Let us first formally define this concept, then right after provide the intuition behind the formalism. First, we define the  **$n$ -disk**, its interior the **open  $n$ -disk**, and its boundary the  **$(n-1)$ -sphere** as:

$$\mathbb{D}^n = \{x \in \mathbb{R}^n \mid \|x\| \leq 1\}, \quad (\text{A.6})$$

$$\mathring{\mathbb{D}}^n = \text{int}(\mathbb{D}^n) = \{x \in \mathbb{R}^n \mid \|x\| < 1\}, \quad (\text{A.7})$$

$$\mathbb{S}^{n-1} = \partial \mathbb{D}^n = \{x \in \mathbb{R}^n \mid \|x\| = 1\}. \quad (\text{A.8})$$

An  **$n$ -cell**  $c$  is defined as a topological space homeomorphic to  $\text{int}(\mathbb{D}^n)$ . A **cell decomposition**  $\mathcal{C}$  of a topological space  $X$  is a collection of disjoint cells  $c_i$  such that  $X = \bigcup_i c_i$ . The  **$n$ -skeleton**  $X^n$  of  $X$  is the union of  $k$ -cells of  $\mathcal{C}$  such that  $k \leq n$ . Finally,  $\mathcal{K} = (X, \mathcal{C})$  is called a **CW complex** if it satisfies the following three axioms:

**Axiom 1 (Characteristic maps).** For each  $n$ -cell  $c \in \mathcal{C}$ , there exists a continuous function  $\Phi_c : \mathbb{D}^n \rightarrow X$  such that the restriction of  $\Phi_c$  to  $\text{int}(\mathbb{D}^n)$  is a homeomorphism from  $\text{int}(\mathbb{D}^n)$  to  $c$ , and such that  $\Phi_c(\partial \mathbb{D}^n) \subseteq X^{n-1}$ .

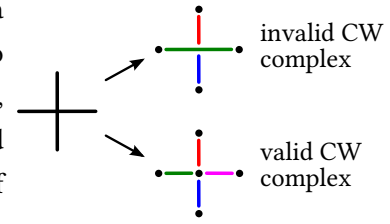
**Axiom 2 (Closure finiteness).** The closure  $\bar{c}$  intersects only a finite number of other cells.

**Axiom 3 (Weak topology).**  $A \subseteq X$  is closed iff  $A \cap \bar{c}$  is closed for each  $c \in \mathcal{C}$ .

Despite the fact that the last two axioms are those responsible for the acronym “CW”, you can safely ignore them in this report, since they are automatically true if the number of cells is finite, which should always be the case for computer graphics applications. Therefore, let us simply clarify this obscure CW complex definition by focusing on the preliminary definitions and the first axiom. Since an  $n$ -cell is a pointset homeomorphic to  $\text{int}(\mathbb{D}^n)$ , this means that a 0-cell (called **vertex**) is a single point in space, a 1-cell (called **edge**) is a pointset homeomorphic to the open interval  $(-1, 1)$ , and a 2-cell (called **face**) is a pointset homeomorphic to the open 2-disk  $\mathring{\mathbb{D}}^2$ . A two-dimensional cell decomposition of a topological space  $X$  is therefore a partition of  $X$  into vertices  $v \cong \{0\}$ , edges  $e \cong (-1, 1)$  and faces  $f \cong \mathring{\mathbb{D}}^2$ .

However, this allows spaces like  $X = \mathbb{R}$  to be decomposed as a single cell  $e = \mathbb{R}$ , since  $\mathbb{R} \cong (-1, 1)$ . Also, this allows a cross to be decomposed as four vertices and three edges (cf. Figure A.11, top). Because we do not want these decompositions to be valid CW complexes, Axiom 1 adds some restrictions. In the case of edges, instead of “simply” requiring  $e \cong (-1, 1)$ , we require the existence of a continuous function  $\Phi_e : [-1, 1] \rightarrow X$  such that  $\Phi_e((-1, 1)) = e$ . This way, even though edges are “open intervals”, they are forced to “look like interior of closed intervals”,

thus the edge  $e = \mathbb{R}$  is not allowed as part of a CW complex. Finally, Axiom 1 also requires that  $\Phi_e(-1)$  and  $\Phi_e(1)$  be included in the 0-skeleton of  $\mathcal{C}$ . In other words, it requires that the “edge boundary”  $\partial e = \bar{e} \setminus e$  is made of vertices that are part of the decomposition. This additional requirement enforces the existence of a vertex at the intersection of the cross (cf. Figure A.11). We note that  $\Phi_e$  restricted to  $(-1, 1)$  must be a homeomorphism (in particular, must be invertible), but it is not required that  $\Phi_e$  be invertible on the whole domain of definition  $[-1, 1]$ . This prevents self-intersections in the interior of the edge, but allows  $\Phi_e(-1)$  to be equal to  $\Phi_e(1)$ . In other words, it is allowed that the start vertex of the edge is equal to the end vertex. All these considerations scale for faces: the closure of a face must be compact, the boundary of a face must be included in a union of vertices and edges, and faces cannot self-intersect in their interior, but their boundary can “use” the same vertex or edge several times. For instance, the boundary of a face can be a single vertex (cf. Figure A.10, middle), or even a single point in the interior of an edge (cf. Figure B.2), or can do “switch-backs” in the interior of an edge (cf. Figure B.3).



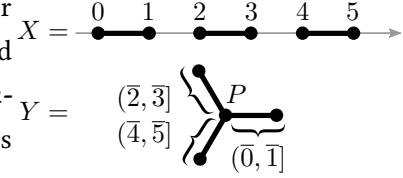
**Figure A.11:** Two valid cell decompositions of a cross, but only one is a valid CW complex.

## A.6 Geometric Realizations and Quotient Spaces

The reader may have noticed that the definition of CW complexes that we have given differs greatly from the definitions of polygonal presentations and abstract simplicial complexes because

it relies on the existence of a topological space  $X$ , that we decompose into cells. On the contrary, a polygonal presentation or an abstract simplicial complex is not a topological space *per se*, but a combinatorial description of one. From such combinatorial description, one can *build* the corresponding topological space, called its *geometric realization*, by “gluing” together known topological spaces, which is formally done using the concept of *quotient space* that we recall in the following paragraph.

Let  $X$  be a set, and let  $\sim$  be an equivalence relation on  $X$ . For instance, let us take as a very simple example  $X = \{1, 2, 3\}$  and  $\sim$  defined such that  $1 \sim 2$ ,  $1 \not\sim 3$ , and  $2 \not\sim 3$ . The **equivalence classes** of  $\sim$  are defined as a partition of  $X$  into subsets regrouping elements that are equivalent to each others. In our example, there are two equivalence classes:  $E = \{1, 2\}$  (since 1 and 2 are equivalent) and  $F = \{3\}$  (since 3 is not equivalent to any other elements). The **quotient set** of  $X$  by  $\sim$ , denoted  $X/\sim$ , is defined as the set of equivalence classes of  $\sim$ . Therefore, in our example, we have  $X/\sim = \{E, F\} = \{\{1, 2\}, \{3\}\}$ . In other words, *quotienting* a set by an equivalence relation can be understood as transforming elements that were equivalent into a single element. The concept of **quotient space** is very similar, except that it acts on topological spaces instead of sets. This means that in addition to define  $Y = X/\sim$  as the set of equivalence classes of  $\sim$ , it also makes  $Y$  a topological space by defining which subsets of  $Y$  are “open”. More specifically, the open subsets of  $Y$  are defined as the sets of equivalent classes whose unions are open sets in  $X$ . This means that two equivalent classes  $E$  and  $F$  are “close-by” in  $Y$  if and only if there exist  $x_E \in E$  and  $x_F \in F$  that were originally “close-by” in  $X$ . For instance, consider  $X = [0, 1] \cup [2, 3] \cup [4, 5]$ , i.e.  $X \subset \mathbb{R}$  is a disjoint union of three closed intervals (cf. Figure A.12, top). Then let us consider the equivalence relation  $\sim$ , defined by  $0 \sim 2$ ,  $0 \sim 4$ ,  $2 \sim 4$ , and  $x_1 \not\sim x_2$  for all other pairs in  $X$ . This means that the set  $P = \{0, 2, 4\}$  is one equivalence class, and every other element  $x \in X$  is its own equivalence class  $\{x\}$ . By using the convenient notation  $(\bar{a}, \bar{b}] = \{\{x\} \mid x \in (a, b]\}$ , then we have  $Y = (\bar{0}, \bar{1}] \cup (\bar{2}, \bar{3}] \cup (\bar{4}, \bar{5}] \cup \{P\}$ . Because 0 was in the closure of  $(0, 1]$  in  $X$ , and  $0 \in P$ , it can be shown that  $P$  is in the closure of  $(\bar{0}, \bar{1}]$  in  $Y$ . Similarly, it can be shown that  $P$  is in the closure of  $(\bar{2}, \bar{3}]$  and  $(\bar{4}, \bar{5}]$ . Therefore, the closures of  $(\bar{0}, \bar{1}]$ ,  $(\bar{2}, \bar{3}]$ , and  $(\bar{4}, \bar{5}]$  intersect in  $Y$  (at  $P$ ), while the closures of  $(0, 1]$ ,  $(2, 3]$ , and  $(4, 5]$  did not intersect originally in  $X$ . This is why it is said that using this operation, the three closed intervals  $[0, 1]$ ,  $[2, 3]$ , and  $[4, 5]$  have been *glued*, by *identifying* the three real numbers 0, 2 and 4 as a single element. Quotienting  $X$  by  $\sim$  has transformed a disjoint union of three closed intervals (a 1-manifold with boundary) into a star-like shape with three branches (a non-manifold space), as illustrated in Figure A.12.



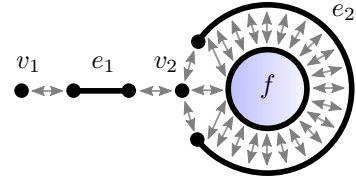
**Figure A.12:** Illustration of  $X = [0, 1] \cup [2, 3] \cup [4, 5]$ , and the quotient space  $Y = X/\sim$  defined by  $0 \sim 2 \sim 4$ .

With this formalism, the **geometric realization** of an abstract simplicial complex, called a **sim-**

**plicial complex** (i.e., not abstract), can be easily defined as a disjoint union of points, segments, triangles, and  $n$ -dimensional triangles that are glued together by identifying their common boundaries with a well-chosen equivalence relation. The reverse viewpoint can also be taken: given a possibly non-manifold topological space  $X$  (in a sense, “already glued”), and a decomposition of  $X$  into subsets homeomorphic to points, interior of segments, interior of triangles, and interior of  $n$ -dimensional satisfying a few properties on their boundaries, then it is called a simplicial complex, and its corresponding abstract simplicial complex can be defined.

Similarly, CW complexes can be defined either as we did (i.e., “Let  $X$  be a topological space. If there exists a cell-decomposition  $\mathcal{C}$  such that there exists functions  $\Phi_c$  satisfying [...], then  $(X, \mathcal{C})$  is called a CW complex”), or by *building* them, via the explicit definition of characteristic maps  $\Phi_c$  gluing disjoint  $n$ -disks together.

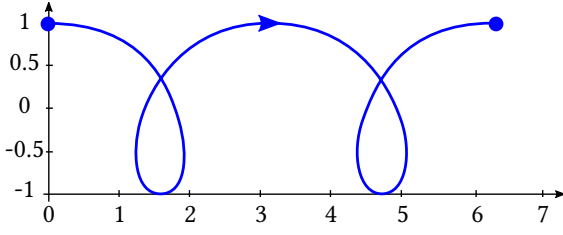
For instance, the space in Figure A.8 is homeomorphic to a CW complex that can be built explicitly as follows. We first define  $X$  as the disjoint union of two points  $v_1$  and  $v_2$  (0-cells), two closed intervals  $e_1$  and  $e_2$  (1-cells, parameterized as  $[-1, 1]$ ), and one disk  $f$  (2-cell, whose boundary is parameterized  $[0, 2\pi]$ ). We then define  $\Phi_{e_1}(-1) = v_1$ ,  $\Phi_{e_1}(1) = v_2$ ,  $\Phi_{e_2}(-1) = v_2$ ,  $\Phi_{e_2}(1) = v_2$ ,  $\Phi_f(\theta = 0) = v_2$ , and  $\Phi_f(\theta \in (0, 2\pi)) = \frac{\theta}{\pi} - 1 \in e_2$ . From these characteristic maps, an equivalence relation  $\sim$  can be defined, identifying each point in the boundary of each  $n$ -cell to a point of a  $k$ -cell,  $k < n$ , as illustrated in Figure A.13. Quotienting  $X$  by this equivalence relation gives the final CW complex  $Y = X/\sim$ .



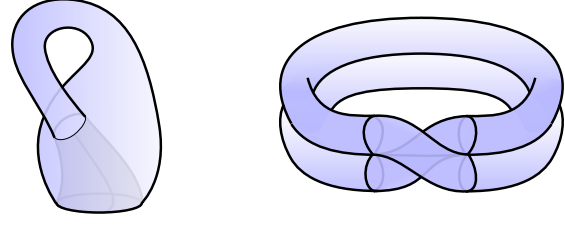
**Figure A.13:** The space Figure A.8 can be seen as a CW complex obtained by gluing together two points, two closed intervals and one disk. The double-arrows represent the equivalence relation for the glue operation.

The geometric realization of a polygonal presentation, informally described in Section A.4, is also formally defined via quotient spaces. Each word of size  $k$  defines a regular polygon with  $k$  edges, and points in the boundary of each polygon are identified to each other via the symbols in the words. This defines an equivalence relation, which subsequently defines the geometric realization as a quotient space.

To summarize, a polygonal presentation is a combinatorial structure from which can be defined a topological space using quotient spaces. An abstract simplicial complex is a combinatorial structure from which can be defined a topological space using quotient spaces. Finally, a CW complex is a topological space that can be defined using quotient spaces. However, CW complexes cannot be described combinatorially because the quotient spaces are defined via continuous functions, not using a combinatorial structure. This makes them very inconvenient to implement on a computer. On the contrary, our concept of PCS complex that does have a combinatorial description (called abstract PCS complex).



**Figure A.14:** The continuous mapping  $\Phi : [0, 4\pi] \rightarrow \mathbb{R}^2$  defined as  $\Phi(t) = (\sin(t) + 0.5t, \cos(t))$  is an immersion. However, since the curve is self-intersecting, the mapping is not injective and hence  $\Phi$  is not an embedding.



**Figure A.15:** Two immersions of the Klein bottle in  $\mathbb{R}^3$ . Both immersions intersect themselves in a closed curve whose preimage consists of two loops. Image and caption inspired from [Edelsbrunner and Harer 2010].

## A.7 Immersions vs. Embeddings

In this dissertation, the term **map** is used to denote a continuous function  $\Phi : X \mapsto Y$  between two topological spaces  $X$  and  $Y$ . Alternatively, we also use the term **immersion**, and we say that  $X$  is *immersed* in  $Y$  by  $\Phi$ . In addition, if  $\Phi$  is injective then it is called an **embedding**, and we say that  $X$  is *embedded* in  $Y$  by  $\Phi$ . Intuitively, an embedding is an immersion that does not produce self-intersections, as illustrated in Figure A.14.

If the space  $Y$  is too low dimensional, there may not exist an embedding of  $X$  into  $Y$ . Classical examples are non-orientable surfaces without boundary, such as the Klein bottle, that can be embedded in  $\mathbb{R}^4$  but not in  $\mathbb{R}^3$ . Hence, if  $\mathcal{S}$  is an abstract simplicial complex representing a Klein bottle, and  $X = |\mathcal{S}|$  is a geometric realization of  $\mathcal{S}$ , then every mapping from  $X$  to  $Y = \mathbb{R}^3$  will produce self-intersections, as illustrated in Figure A.15.

Combinatorial structures have the advantage to enable defining an immersion of their geometric realization (and hence making possible to visualize it and manipulate it in 2D or 3D), without actually constructing the geometric realization itself that would require additional dimensions. This is actually very standard in 3D polygon modeling: a Klein bottle can be easily modeled with any triangle mesh structure supporting non-orientable meshes. This will result in intersections of some triangles, but these intersections are not tracked and the intersecting triangles just ignore each others, which is the behavior that modeling artists expect. We use exactly the same concept for our vector graphics complexes: instead of working with an embedding, like planar maps do, we work with an abstract combinatorial structure that is *immersed* in  $\mathbb{R}^2$ . There is no need to explicitly construct the geometric realization of a vector graphics complex, which may require 3 or 4 dimensions, since the combinatorial structure is enough to characterize this geometric realization.



## Appendix B

# Non-Combinatorial Definition of PCS Complexes

In Chapter 3, we defined a combinatorial structure called *abstract PCS complex*, from which we defined the concept of *PCS complex* as a geometric realization. In this Appendix, we propose an alternative definition of PCS complexes, directly defined as a cell decomposition of an existing topological space, similarly to how we defined CW complexes in Appendix A. This allows to better compare PCS complexes and CW complexes, and also provides some insight on the concept of abstract PCS complex.

This Appendix is organized as follows. First, in Section B.1, we define a concept of *cell complex* for arbitrary dimension, different from the one used for CW complexes, and in Section B.2 we prove a few properties of this complex. In Section B.3, we provide a comparison between our concept of cell complex and the one used for CW complexes. Finally, in Section B.4, we define a PCS complex as a cell complex of dimension at most two, and we exhaustively characterize all possible types of vertices, edges, and faces, and how they are allowed to be glued together. This characterization is much less compact than the original definition, but looks more similar to the definition of abstract PCS complexes and therefore provides a link between the two definitions. Finally, in Section C, we show that the class of topological spaces decomposable as a PCS complex is equal to the class of topological spaces decomposable as a two-dimensional simplicial complex.

### B.1 Cell Complex

Throughout this Appendix, a topological space means a Hausdorff topological space, and an  $n$ -manifold means a topological  $n$ -manifold with boundary.

**Cell** An  *$n$ -cell*  $c$  is defined as a topological space homeomorphic to the interior of a connected compact  $n$ -manifold. The *dimension* of  $c$  is  $\dim c = n$ . A *cell* is an  $n$ -cell for some  $n$ . For  $n = 0$ , 1, and 2, we call them vertices, edges and faces.



**Cell decomposition** Let  $X$  be a topological space. A **cell decomposition**  $\mathcal{C}$  of  $X$  is a finite collection of disjoint cells  $c_i$  such that  $X = \bigcup_i c_i$ . The  **$n$ -skeleton**  $X^n$  of  $X$  (resp. the  **$n$ -skeletonset**  $\mathcal{C}^n$  of  $\mathcal{C}$ ) is the union (resp. the set) of all the  $k$ -cells of  $\mathcal{C}$  such that  $k \leq n$ . The smallest  $n$  such that  $X^n = X$  is called the **dimension** of  $(X, \mathcal{C})$ .

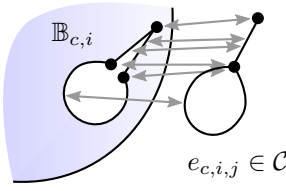
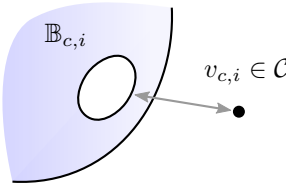
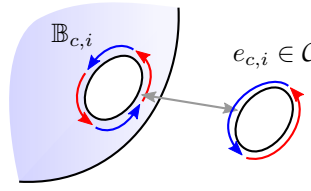
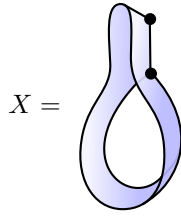
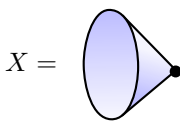
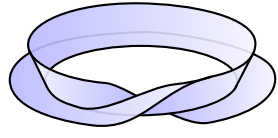
**Pointsets and set of pointsets** We will often refer to objects that are either pointsets or sets of pointsets, and it is important not to confuse them. For instance,  $X$  is a pointset, a cell  $c$  is a subset of  $X$  and hence is also a pointset (a set of points  $p \in X$ ). A **union** of cells is also a pointset. However, a **set** of cells, such as  $\mathcal{C}$ , is not a pointset but a set of pointsets. A subset  $\mathcal{C}' \subset \mathcal{C}$  is also a set of pointset. If  $c_i$  are pointsets, and  $\mathcal{C}_j$  are sets of pointsets, we introduce the notation  $c' = \langle c_1, \dots, c_k, \mathcal{C}_1, \dots, \mathcal{C}_m \rangle$  to conveniently define  $c'$  as the pointset obtained by the union of the pointsets  $c_i$  and of the pointsets in the sets  $\mathcal{C}_j$ . For instance,  $\langle c_1, c_2 \rangle = c_1 \cup c_2$ ,  $\langle \mathcal{C} \rangle = X$ , and  $\langle \mathcal{C}^n \rangle = X^n$ .

**Closure and boundary** The **closure** of a cell  $c \in \mathcal{C}$ , denoted  $\bar{c}$ , is the closure of  $c$  in  $X$ . The **boundary** of a cell  $c \in \mathcal{C}$ , denoted  $\partial c$ , is defined as the set difference  $\bar{c} \setminus c$ . An edge whose boundary is empty is called a **closed edge**, otherwise it is called an **open edge**.

**Cell complex** Let  $X$  be a topological space and  $\mathcal{C}$  be a cell decomposition of  $X$ . The pair  $\mathcal{K} = (X, \mathcal{C})$  is called a **cell complex** if and only if, for each  $n$ -cell  $c \in \mathcal{C}$ , there exists a connected compact  $n$ -manifold  $\mathbb{M}_c$  and a map  $\Phi_c : \mathbb{M}_c \rightarrow X$  satisfying the following **cell complex constraints**:

- The restriction of  $\Phi_c$  to  $\text{int}(\mathbb{M}_c)$  is a homeomorphism from  $\text{int}(\mathbb{M}_c)$  to  $c$ .
- For each connected component  $\mathbb{B}_{c,i}$  of  $\partial \mathbb{M}_c$ , either:
  1. there exists a cell decomposition  $\mathcal{D}_{c,i}$  of  $\mathbb{B}_{c,i}$  such that for all  $d_{c,i,j} \in \mathcal{D}_{c,i}$ , the restriction of  $\Phi_c$  to  $d_{c,i,j}$  is a homeomorphism from  $d_{c,i,j}$  to a cell  $e_{c,i,j} \in \mathcal{C}$ , or
  2. the image of  $\mathbb{B}_{c,i}$  by  $\Phi_c$  is a single vertex  $v_{c,i} \in \mathcal{C}$ , or
  3. the boundary component  $\mathbb{B}_{c,i}$  is homeomorphic to  $\mathbb{S}^1$ , it is mapped by  $\Phi_c$  to a single closed edge  $e_{c,i} \in \mathcal{C}$ , and the restriction of  $\Phi_c$  to  $\mathbb{B}_{c,i}$  “wraps  $N_{f,i}$  times around  $e_{c,i}$ ”, for some  $N_{f,i} \in \mathbb{N}^+$ .

In other words,  $\Phi_c$  must be a “homeomorphism by part” from cells decomposing  $\mathbb{M}_c$  to cells of  $\mathcal{C}$  (case 1.), with the first exception that a connected component of  $\partial \mathbb{M}_c$  is allowed to shrink to a single vertex (case 2.), and the second exception that a connected component of  $\partial \mathbb{M}_c$  homeomorphic to  $\mathbb{S}^1$  is allowed to be mapped to a closed edge by wrapping around it several times (case 3.). We illustrate these cases in Figure B.1, and formalize below what we mean by “wraps  $N_{f,i}$  times around  $e_{c,i}$ ”.

1. Homeomorphic by part	2. Shrink to a vertex	3. Wrap around a closed edge
		
		

**Figure B.1:** The three possible “gluing conditions” that each connected component  $\mathbb{B}_{c,i}$  of the boundary of each cell  $c$  must satisfy. We illustrate them for  $\dim c = 2$ , since it is the dimension for which they have been designed. Top: How each connected component of the boundary of the characteristic manifold is glued to cells of lower dimension. Bottom: The actual, glued topological space  $X$ . In terms of abstract PCS complex, these three cases correspond to the three types of cycle. From left to right: non-simple cycle, Steiner cycle, and simple cycle.

**Wrapping circles around circles** Let  $A$  and  $B$  be two spaces homeomorphic to the circle  $\mathbb{S}^1$ . We say that a map  $\Phi : A \rightarrow B$  **wraps  $N > 0$  times around  $B$**  if and only if there exist two homeomorphisms  $\Phi_A : A \rightarrow \mathbb{S}^1$  and  $\Phi_B : B \rightarrow \mathbb{S}^1$  such that:

$$\Phi = \Phi_B^{-1} \circ W_N \circ \Phi_A \quad (\text{B.1})$$

where, by using the usual parameterization  $\theta \in [0, 2\pi)$  of  $\mathbb{S}^1$ ,  $W_N : \mathbb{S}^1 \rightarrow \mathbb{S}^1$  is the continuous map defined by:

$$W_N(\theta) = N\theta \quad (\text{B.2})$$

**Characteristic objects** The connected compact  $n$ -manifold  $\mathbb{M}_c$  is called the **characteristic manifold** of  $c$ , and the map  $\Phi_c$  is called the **characteristic map** of  $c$ .

**Cell neighborhood** We define the **boundary cells** of  $c$  as the set  $\mathcal{B}_c$  of all  $e_{c,i,j}$ ,  $v_{c,i}$ , and  $e_{c,i}$ . The **star** of a cell  $c \in \mathcal{C}$  is defined as the set of cells

$$\mathcal{S}_c = \{c' \in \mathcal{C} \mid c \in \mathcal{B}_{c'}\}. \quad (\text{B.3})$$

**Dimension** The **dimension** of a cell complex is defined as the dimension of its cell decomposition. A cell complex of dimension  $n$  is also called an  **$n$ -complex** for conciseness.

## B.2 Relation Between $\partial c$ and $\mathcal{B}_c$ , Compactness, and Subcomplexes

For the sake of completeness and comparison with CW complexes, we formally prove in this section a few immediate properties that cell complexes (in a PCS sense) satisfy, for arbitrary dimension. The reader not familiar with CW complexes may safely skip this section. Let  $(X, \mathcal{C})$  be a cell complex. Then we have:

**Lemma 1.**  $\forall c \in \mathcal{C}, \mathcal{B}_c \subseteq \mathcal{C}^{n-1}$ , where  $n = \dim c$ .

*Proof.* If  $n = 0$ , then  $\mathbb{M}_c$  is a singleton and  $\partial \mathbb{M}_c = \emptyset$  so there are no  $\mathbb{B}_{c,i}$  hence no  $v_{c,i}$ ,  $e_{c,i}$ , or  $e_{c,i,j}$  and  $\mathcal{B}_c = \emptyset$ . Let  $n \geq 1$ . Since  $\dim v_{c,i} = 0$ , then  $\dim v_{c,i} \leq n - 1$  and  $v_{c,i} \in \mathcal{C}^{n-1}$ . The case where  $\mathcal{B}_c$  contains a cell of type  $e_{c,i}$  can only occur when  $n \geq 2$ , so we also have  $e_{c,i} \in \mathcal{C}^{n-1}$ . Since  $d_{c,i,j}$  is a cell of  $\mathbb{B}_{c,i}$  and that  $\mathbb{B}_{c,i}$  is a compact  $(n - 1)$ -manifold, we have  $\dim d_{c,i,j} \leq n - 1$ . In addition,  $\dim e_{c,i,j} = \dim d_{c,i,j}$  since  $\Phi_c$  restricts to a homeomorphism from  $d_{c,i,j}$  to  $e_{c,i,j}$ , hence  $\dim e_{c,i,j} \leq n - 1$  and  $e_{c,i,j} \in \mathcal{C}^{n-1}$ .  $\square$

**Lemma 2.**  $\forall c \in \mathcal{C}, \Phi_c(\partial \mathbb{M}_c) = \langle \mathcal{B}_c \rangle$ .

*Proof.* We have  $\partial \mathbb{M}_c = \bigcup_i \mathbb{B}_{c,i}$ , hence  $\Phi_c(\partial \mathbb{M}_c) = \bigcup_i \Phi_c(\mathbb{B}_{c,i})$ . The image of  $\mathbb{B}_{c,i}$  is either a single vertex  $v_{c,i}$  (case 2.), a closed edge  $e_{c,i}$  (case 3.), or  $\mathbb{B}_{c,i} = \bigcup_j d_{c,i,j}$  (case 1.) in which case  $\Phi_c(\mathbb{B}_{c,i}) = \bigcup_j \Phi_c(d_{c,i,j}) = \bigcup_j e_{c,i,j}$ . Hence,  $\Phi_c(\partial \mathbb{M}_c) = \langle \dots, v_{c,i}, \dots, e_{c,i,j}, \dots, e_{c,i}, \dots \rangle = \langle \mathcal{B}_c \rangle$ .  $\square$

**Lemma 3.**  $\forall c \in \mathcal{C}, \bar{c} = \Phi_c(\mathbb{M}_c)$ .

*Proof.* If  $\Phi : X \rightarrow Y$  is a map and  $X' \subseteq X$ , then  $\Phi(\overline{X'}) \subseteq \overline{\Phi(X')}$ . Thus in our case:

$$\Phi_c(\mathbb{M}_c) = \Phi_c(\overline{\text{int}(\mathbb{M}_c)}) \subseteq \overline{\Phi_c(\text{int}(\mathbb{M}_c))} = \bar{c}.$$

In addition,  $\Phi_c(\mathbb{M}_c)$  is compact as a continuous image of a compact, thus  $\Phi_c(\mathbb{M}_c)$  is closed in  $X$  since  $X$  is Hausdorff. Considering that  $\bar{c}$  is defined as the intersection of all closed set in  $X$  containing  $c$ , that  $\Phi_c(\mathbb{M}_c)$  contains  $c$ , and that  $\Phi_c(\mathbb{M}_c)$  is closed in  $X$ , it proves that  $\bar{c} \subseteq \Phi_c(\mathbb{M}_c)$ . Hence, we proved that  $\Phi_c(\mathbb{M}_c) \subseteq \bar{c} \subseteq \Phi_c(\mathbb{M}_c)$  thus  $\bar{c} = \Phi_c(\mathbb{M}_c)$ .  $\square$

**Lemma 4.**  $\forall c \in \mathcal{C}, \bar{c} = \langle c, \mathcal{B}_c \rangle$ .

*Proof.*  $\bar{c} = \Phi_c(\mathbb{M}_c) = \Phi_c(\text{int}(\mathbb{M}_c) \cup \partial \mathbb{M}_c) = \Phi_c(\text{int}(\mathbb{M}_c)) \cup \Phi_c(\partial \mathbb{M}_c) = c \cup \langle \mathcal{B}_c \rangle$ .  $\square$

**Proposition 1.**  $\forall c \in \mathcal{C}, \partial c = \langle \mathcal{B}_c \rangle$ .

*Proof.*  $\partial c = \bar{c} \setminus c = \langle c, \mathcal{B}_c \rangle \setminus c = \langle \mathcal{B}_c \rangle$  since  $\mathcal{B}_c \subseteq \mathcal{C}^{n-1}$  and  $\dim c = n$  thus  $c \notin \mathcal{B}_c$ .  $\square$

**Proposition 2.**  $X$  is compact.

*Proof.*  $\forall c \in \mathcal{C}, c \subseteq \bar{c}$  and  $\bar{c} \subseteq X$  thus  $X = (\bigcup_{c \in \mathcal{C}} c) \subseteq (\bigcup_{c \in \mathcal{C}} \bar{c}) \subseteq X$ . Hence, all inclusions are equalities, and  $X$  is compact as a finite union of compacts.  $\square$

**Proposition 3.**  $\forall c \in \mathcal{C}, \partial c$  is compact and closed in  $X$ .

*Proof.* The boundary of a compact manifold is compact, hence  $\partial \mathbb{M}_c$  is compact, and then  $\Phi_c(\partial \mathbb{M}_c) = \langle \mathcal{B}_c \rangle = \partial c$  is compact. Thus, it is closed in  $X$  since  $X$  is Hausdorff.  $\square$

**Proposition 4.**  $\forall c \in \mathcal{C}, (\partial c, \mathcal{B}_c)$  is a cell complex.

*Proof.*  $\partial c = \langle \mathcal{B}_c \rangle$  hence  $\mathcal{B}_c$  is a cell decomposition of  $\partial c$ . Let  $c' \in \mathcal{B}_c$ , and  $n' = \dim c'$ . The existence of a manifold  $\mathbb{M}_{c'}$ , a map  $\Phi_{c'} : \mathbb{M}_{c'} \rightarrow X$ , decompositions  $\mathcal{D}_{c',i}$  of  $\mathbb{M}_{c'}$  and cells  $v_{c',i} \in \mathcal{C}$ ,  $e_{c',i} \in \mathcal{C}$ , and  $e_{c',i,j} \in \mathcal{C}$  comes directly from the fact that  $(X, \mathcal{C})$  is a cell complex. We only need to verify that  $\Phi_{c'} : \mathbb{M}_{c'} \rightarrow \partial c$  (instead of  $X$ ) and that  $v_{c',i} \in \mathcal{B}_c$ ,  $e_{c',i} \in \mathcal{B}_c$  and  $e_{c',i,j} \in \mathcal{B}_c$  (instead of  $\mathcal{C}$ ).

We know that  $c' \in \mathcal{B}_c$ , thus  $c' \subseteq \partial c$ , thus  $\bar{c'} \subseteq \partial c$  (because  $\partial c$  is closed in  $X$ ), thus  $\Phi_{c'} : \mathbb{M}_{c'} \rightarrow \partial c$  (because  $\Phi_{c'}(\mathbb{M}_{c'}) = \bar{c'}$ ). In addition, the cells  $v_{c',i}$ ,  $e_{c',i}$  and  $e_{c',i,j}$  are images of restrictions of  $\Phi_{c'}$  thus are subsets of  $\Phi_{c'}(\mathbb{M}_{c'})$ , thus are subset of  $\partial c$ , hence are elements of  $\mathcal{B}_c$ .  $\square$

**Corollary 1.**  $\forall c \in \mathcal{C}, (\bar{c}, \{c\} \cup \mathcal{B}_c)$  is a cell complex.

*Proof.* We are just adding  $c$  to the complex above, and we know that  $\Phi_c : \mathbb{M}_c \rightarrow \bar{c}$ , as well as the cells  $v_{c',i}$ ,  $e_{c',i,j}$  and  $e_{c',i}$  are in  $\{c\} \cup \mathcal{B}_c$  since they are by definition in  $\mathcal{B}_c$ .  $\square$

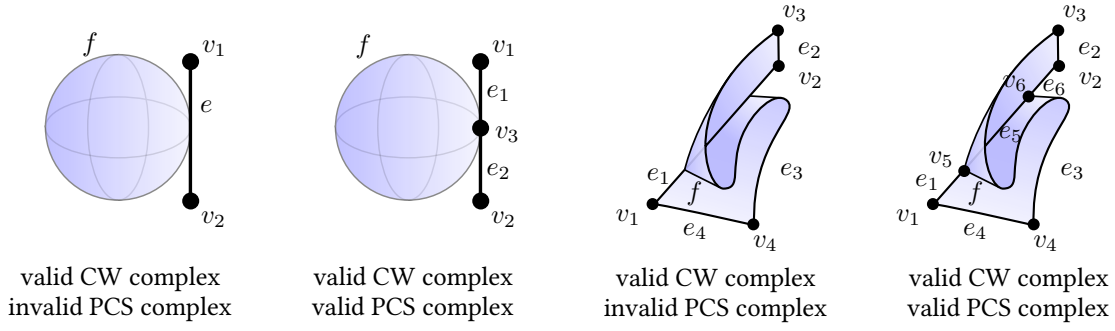
**Proposition 5.**  $\forall c \in \mathcal{C}$ , if  $c' \in \mathcal{B}_c$  then  $\mathcal{B}_{c'} \subseteq \mathcal{B}_c$ . In other words, the relation “ $c'$  is in the boundary of  $c$ ” is transitive:

$$(c'' \in \mathcal{B}_{c'} \wedge c' \in \mathcal{B}_c) \Rightarrow c'' \in \mathcal{B}_c$$

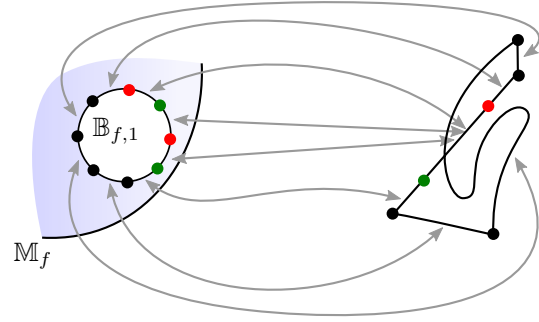
*Proof.* If  $c' \in \mathcal{B}_c$  then  $c' \subseteq \partial c$ . Hence,  $\bar{c'} \subseteq \partial c$  since  $\partial c$  is closed, from which it follows that  $\partial c' \subseteq \partial c$  since  $\partial c' = \bar{c'} \setminus c'$ . Thus  $\langle \mathcal{B}_{c'} \rangle \subseteq \langle \mathcal{B}_c \rangle$ , thus  $\mathcal{B}_{c'} \subseteq \mathcal{B}_c$ .  $\square$

## B.3 Comparison with CW Complexes

Our definitions of cell, cell decomposition and cell complex differ from the ones of CW complexes, thus there are a few differences that are worth noting. In this section, we use the terms PCS-cell, PCS-cell decomposition and PCS-cell complex to refer to our definition (for arbitrary dimension), while we use the terms CW-cell, CW-cell decomposition and CW-cell complex for the classical definition.



$$\begin{aligned}
 X &= \mathbb{S}^2 \cup \{(1, t, 0) \mid t \in [-1, 1]\} \\
 v_1 &= \{(1, 1, 0)\} \\
 v_2 &= \{(1, -1, 0)\} \\
 v_3 &= \{(1, 0, 0)\} \\
 e &= \{(1, t, 0) \mid t \in (-1, 1)\} \\
 e_1 &= \{(1, t, 0) \mid t \in (0, 1)\} \\
 e_2 &= \{(1, t, 0) \mid t \in (-1, 0)\} \\
 f &= \mathbb{S}^2 \setminus \{(1, 0, 0)\}
 \end{aligned}$$

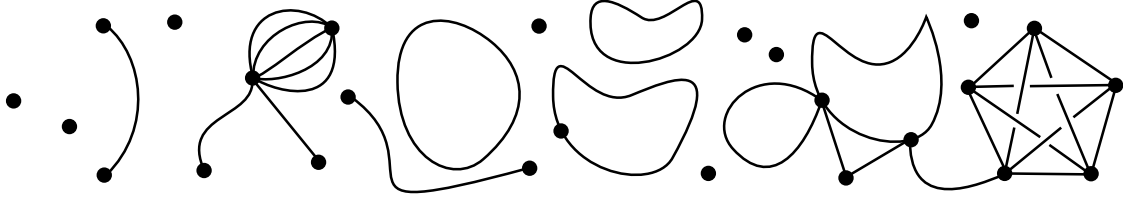


**Figure B.2:** Left: Counter-example showing that Proposition 1 is not true for CW complexes:  $\partial f$  is not equal to any union of other cells, but only “included” in such a union (e.g.,  $\partial f = \{(1, 0, 0)\} \subset e$ ). Right: A valid PCS decomposition of  $X$  requires adding the additional vertex  $v_3$  splitting  $e$  into  $e_1$  and  $e_2$ .

**Figure B.3:** Top-left: The boundary of a CW face is allowed to do “switch-backs” within an edge. Top-right and bottom: A valid PCS decomposition of  $X$  requires the additional vertices  $v_5$  and  $v_6$  so that  $\Phi_f(\mathbb{B}_{f,1})$  is homeomorphic by part from vertices and edges decomposing  $\mathbb{B}_{f,1}$  to vertices and edges of  $\mathcal{C}$ .

An  $n$ -CW-cell is a specific case of  $n$ -PCS-cell, since an  $n$ -CW-cell is homeomorphic to  $\text{int}(\mathbb{D}^n)$ , and that  $\mathbb{D}^n$  is a connected compact  $n$ -manifold. Likewise, a finite CW-cell decomposition is a specific case of a PCS-cell decomposition. Note that PCS-cell decompositions are enforced to be finite, while CW-cell decompositions can be infinite.

However, a finite CW-cell complex is not necessarily a PCS-cell complex. Indeed, even though the definition of PCS-cells is more general than CW-cells, the “gluing conditions” that PCS-cells must satisfy to define a PCS-cell complex are stricter than those for CW-cell complexes. Indeed, we replaced  $\Phi_c(\partial \mathbb{M}_c) \subseteq X^{n-1}$  by a stricter version imposing, for each connected component  $\mathbb{B}_{c,i}$  of  $\partial \mathbb{M}_c$ , that  $\Phi_c(\mathbb{B}_{c,i})$  is either homeomorphic by part from PCS-cells decomposing  $\mathbb{B}_{c,i}$  to PCS-cells of  $\mathcal{C}$  (case 1.), or maps  $\mathbb{B}_{c,i}$  to a single vertex (case 2.), or, if  $\mathbb{B}_{c,i} \cong \mathbb{S}^1$ , wraps it around a closed edge (case 3.). In other words, CW-cells have very little restrictions on how their boundaries are glued to cells of lower dimension, while the boundary of our PCS-cells must be glued to lower dimensional cells in very specific ways. For instance, the boundary of a PCS-face cannot be mapped into a strict subset of an edge (cf. Figure B.2), or do “switch-backs” in the interior of an edge (cf. Figure B.3). This imposes a cleaner incidence structure as illustrated by Proposition 1, which is not true for



**Figure B.4:** A valid 1-complex. It can be seen as an extension of multigraph to support closed edges.

CW-cell complexes (indeed, there may not exist  $\mathcal{B}_c \subseteq \mathcal{C}^{n-1}$  such that  $\partial c = \langle \mathcal{B}_c \rangle$ , a counter-example being Figure B.2). This regularity is what makes possible to describe combinatorially a PCS-cell complex. Note that this regularity is still less strict than the notion of *regular CW complex* [Hatcher 2001, p. 534], which is too strict to provide uniqueness of a minimal complex.

## B.4 PCS Complex

A **PCS complex** is defined as a cell complex of dimension at most two. Hence, its cells are either vertices, edges, or faces. In this section, we analyze in depth what the general definition of cell complex means for each type of cells in a PCS complex, which allows us to provide a detailed characterization of them. This characterization can be seen as an alternate, less compact definition of PCS complex, which provides the link between PCS complexes and abstract PCS complexes.

**Vertices** Vertices are 0-cells, i.e. pointsets homeomorphic to the interior of a connected compact 0-manifold. Up to homeomorphism, there exists only one connected compact 0-manifold:

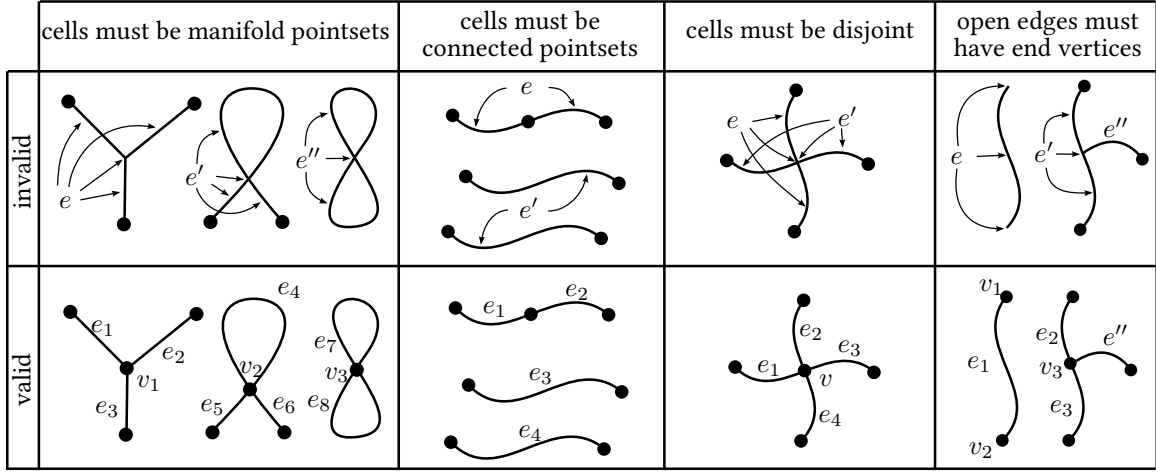
- $\mathbb{V}$ : the singleton  $\mathbb{R}^0 = \{0\}$  whose interior is  $\mathbb{R}^0$  and boundary is  $\emptyset$ .

Hence, a pointset  $v$  is a vertex if and only if it is a singleton, in which case its characteristic manifold is  $\mathbb{M}_v = \mathbb{V}$ .

Since  $\partial \mathbb{V} = \emptyset$ , a vertex automatically satisfies the cell complex constraints. Therefore, cell complexes of dimension 0 are finite sets  $X$ . They admit a unique cell decomposition  $\mathcal{C} = \{\{x\}, x \in X\}$ .

**Edges** Edges are 1-cells, i.e. pointsets homeomorphic to the interior of a connected compact 1-manifold. Up to homeomorphism, there exist only two connected compact 1-manifolds:

- $\mathbb{E}_I$ : the segment  $\mathbb{D}^1 = [0, 1]$  whose interior is  $(0, 1)$  and boundary is  $\{0, 1\}$ .
- $\mathbb{E}_O$ : the circle  $\mathbb{S}^1$  whose interior is  $\mathbb{S}^1$  and boundary is  $\emptyset$ .



**Figure B.5:** Invalid 1-complexes (top), and how to make them valid (bottom).

Hence, a pointset  $e$  is an edge if and only if it is homeomorphic to  $(0, 1)$  or  $\mathbb{S}^1$ . In the first case, it is called an open edge and its characteristic manifold is  $\mathbb{M}_e = \mathbb{E}_\downarrow$ . In the second case, it is called a closed edge and its characteristic manifold is  $\mathbb{M}_e = \mathbb{E}_\circ$ .

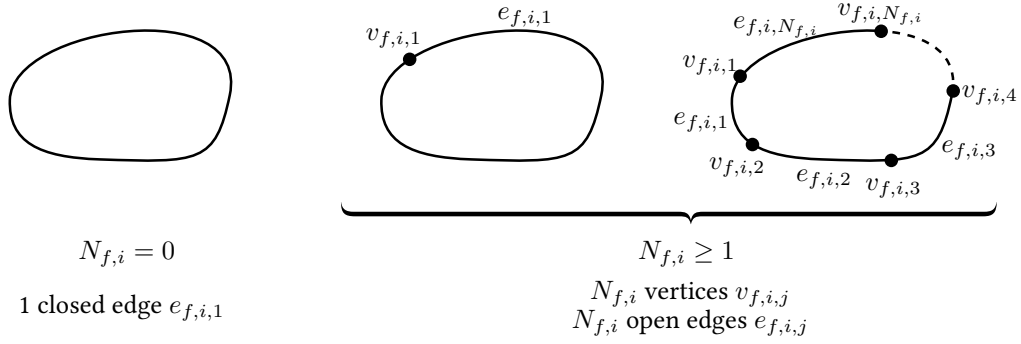
Since  $\partial\mathbb{E}_\circ = \emptyset$ , a closed edge  $e$  automatically satisfies the cell complex constraints. Let  $e$  be an open edge. Its characteristic manifold  $\mathbb{E}_\downarrow = [0, 1]$  has a non-empty boundary  $\partial\mathbb{E}_\downarrow = \{0, 1\}$  made of two connected components  $\mathbb{B}_{e,\text{start}} = \{0\}$  and  $\mathbb{B}_{e,\text{end}} = \{1\}$ . For each  $\mathbb{B}_{e,i}$ , the two cases 1. and 2. of the cell complex constraints are equivalent to:

$$\exists v_{e,i} \in \mathcal{C}, \quad \Phi_e(\mathbb{B}_{e,i}) = v_{e,i}. \quad (\text{B.4})$$

The case 3. does not apply since  $\mathbb{B}_{e,i}$  is not homeomorphic to  $\mathbb{S}^1$ . Therefore, a decomposition of  $X$  into a finite disjoint union of vertices and edges is a cell complex of dimension 1 if and only if:

$$\begin{aligned} & \text{for all open edge } e \in \mathcal{C}, \\ & \text{there exist } \begin{cases} \Phi_e : [0, 1] \rightarrow X \text{ continuous} \\ v_{e,\text{start}} \in \mathcal{C} \\ v_{e,\text{end}} \in \mathcal{C} \end{cases} \\ & \text{such that } \begin{cases} \Phi_e : (0, 1) \rightarrow e \text{ homeomorphism} \\ \Phi_e(0) = v_{e,\text{start}} \\ \Phi_e(1) = v_{e,\text{end}} \end{cases} \end{aligned} \quad (\text{B.5})$$

This is illustrated in Figure B.4. Invalid cell complexes of dimension 1 are illustrated in Figure B.5.



**Figure B.6:** The only possible cell decompositions  $\mathcal{D}_{f,i}$  of a boundary component  $\mathbb{B}_{f,i}$  of  $\mathbb{F}_{\epsilon,g,k}$ .

**Faces** Faces are 2-cells, i.e. pointsets homeomorphic to the interior of a connected compact 2-manifold. Up to homeomorphism, there exist only three “kinds” of connected compact 2-manifolds (cf. Section A.4):

- $\mathbb{F}_{\odot,0,k}$ : the sphere with  $k$  holes.
- $\mathbb{F}_{\odot,g,k}$ : the connected sum of  $g \geq 1$  tori with  $k$  holes.
- $\mathbb{F}_{\oslash,g,k}$ : the connected sum of  $g \geq 1$  projective planes with  $k$  holes.

Hence, a pointset  $f$  is a face if and only if it is homeomorphic to the interior of one of the above manifolds. More formally,  $f$  is a face if and only if there exist  $\epsilon \in \{\odot, \oslash\}$ ,  $g \in \mathbb{N}$  and  $k \in \mathbb{N}$  such that  $f \cong \text{int}(\mathbb{F}_{\epsilon,g,k})$ , in which case it is called an  $(\epsilon, g, k)$ -face and its characteristic manifold is  $\mathbb{M}_f = \mathbb{F}_{\epsilon,g,k}$ . All these characteristic manifolds are illustrated in Figure A.3 (right).

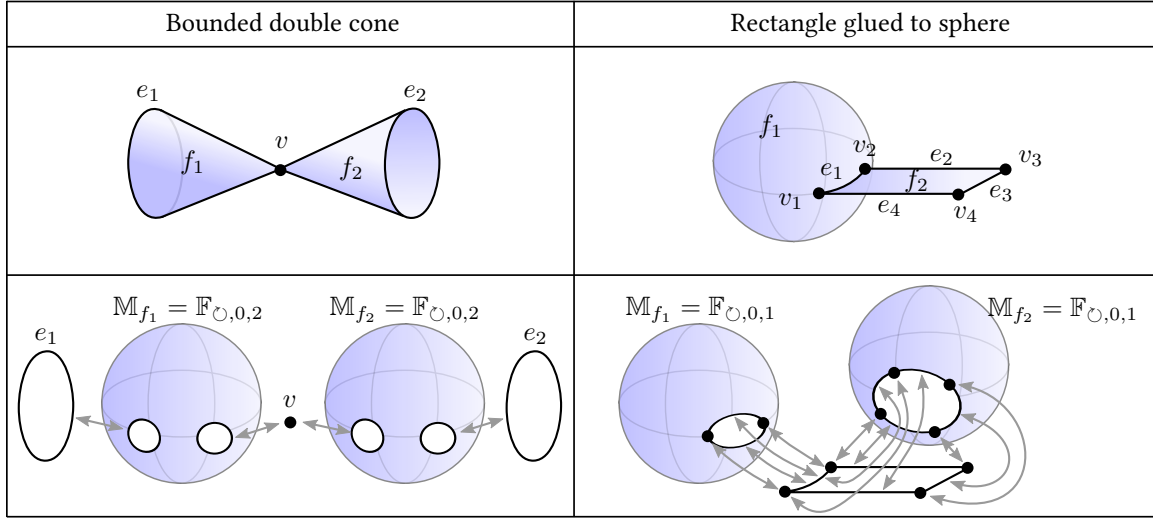
Now, let us characterize how the boundary components  $\mathbb{B}_{f,i}$  of a face  $f$  are allowed to be glued to vertices and edges. First, since  $\mathbb{M}_f$  is a compact 2-manifold, we know that  $\mathbb{B}_{f,i}$  is a compact 1-manifold without boundary, i.e.  $\mathbb{B}_{f,i} \cong \mathbb{S}^1$ . This means that the three cases of the gluing constraints (cf. Figure B.1) have to be considered, and are not equivalent. The cases 2. and 3. do not need further analysis. However, let us explicit what the case 1. means for faces, i.e. let us expand the definition in the specific case where  $\mathbb{B}_{f,i} \cong \mathbb{S}^1$ . Let us start by the following lemma, illustrated in Figure B.6:

**Lemma 5.** Let  $\mathcal{D}_{f,i}$  be a cell decomposition of  $\mathbb{B}_{f,i}$ , and let  $N_{f,i}$  be the number of vertices in  $\mathcal{D}_{f,i}$ .

- If  $N_{f,i} = 0$ , then  $\mathcal{D}_{f,i} = \{\mathbb{B}_{f,i}\}$ , i.e. the decomposition is a single closed edge.
- If  $N_{f,i} \geq 1$ , then  $\mathcal{D}_{f,i}$  is decomposed into  $N_{f,i}$  vertices and  $N_{f,i}$  open edges.

*Proof.* By definition,  $\mathcal{D}_{f,i}$  is a finite collection of disjoint cells  $d_{f,i,j}$  such that  $\mathbb{B}_{f,i} = \bigcup_j d_{f,i,j}$ . Since  $\mathbb{B}_{f,i} \cong \mathbb{S}^1$ , it can only involve vertices and/or edges. Let  $N_{f,i}^{(v)}$  be the number of vertices and  $N_{f,i}^{(e)}$  the number of edges.

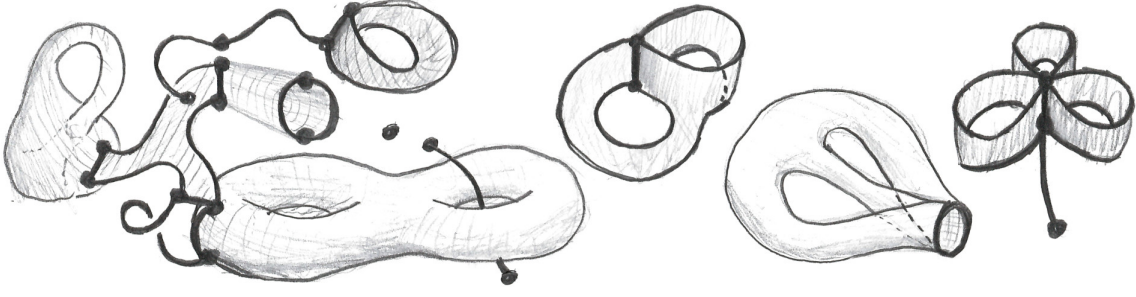




**Figure B.7:** Two examples of valid PCS complexes. Top: The topological space  $X$  and its PCS decomposition. Bottom: Characteristic manifolds of the faces, and how their boundary components are glued to the 1-skeleton.

- Let assume  $N_{f,i}^{(v)} = 0$ . Hence,  $\mathbb{B}_{f,i}$  is a finite disjoint union of edges. Since  $\mathbb{B}_{f,i}$  is compact, but open edges are not compact (thus a finite union of open edges is not compact either), there exists at least an edge  $e$  in  $\mathcal{D}_{f,i}$  that is a closed edge. However, the only closed 1-manifold included in  $\mathbb{S}^1$  is  $\mathbb{S}^1$  itself, thus  $N_{f,i}^{(e)} = 1$  and  $\mathcal{D}_{f,i} = \{e\}$ .
- Let assume  $N_{f,i}^{(v)} \geq 1$ , and let  $v_{f,i,j}$  be the  $N_{f,i}^{(v)}$  vertices of  $\mathcal{D}_{f,i}$ . By fixing  $\theta \in [0, 2\pi)$  a parameterization of  $\mathbb{B}_{f,i}$ , each vertex  $v_{f,i,j}$  corresponds to a unique  $\theta_{f,i,j}$ , and we assume  $\theta_{f,i,1} < \theta_{f,i,2} < \dots < \theta_{f,i,N_{f,i}^{(v)}}$ , without loss of generality. Let  $e_{f,i,j}$  be the pointset  $(\theta_{f,i,j}, \theta_{f,i,j+1}) \subset \mathbb{B}_{f,i}$ . Since  $e_{f,i,j}$  contains no vertices of  $\mathcal{D}_{f,i}$ , then  $e_{f,i,j}$  is included in a disjoint union of edges in  $\mathcal{D}_{f,i}$ . None of them can be a closed edge, because it would contradict  $N_{f,i}^{(v)} \geq 1$  (see bullet above: if  $\mathcal{D}_{f,i}$  contains a closed edge  $e$  then  $\mathcal{D}_{f,i} = \{e\}$ , thus  $N_{f,i}^{(v)} = 0$ ). Hence,  $e_{f,i,j}$  is included in a disjoint union of  $m$  open edges in  $\mathcal{D}_{f,i}$ . Besides, it can be shown that a disjoint union of  $m \geq 2$  open edges in  $\mathbb{S}^1$  is a disconnected set, the connected components being the  $m$  open edges. Therefore, since  $e_{f,i,j}$  is a connected set,  $e_{f,i,j}$  is actually included in a single open edge  $e \in \mathcal{D}_{f,i}$ . In addition, we know that  $e$  contains neither  $\theta_{f,i,j}$  nor  $\theta_{f,i,j+1}$  (since the cells of  $\mathcal{D}_{f,i}$  are disjoint). In conclusion,  $e_{f,i,j} = (\theta_{f,i,j}, \theta_{f,i,j+1})$  is included in  $e$ ,  $e$  is connected, and  $e$  contains neither  $\theta_{f,i,j}$  nor  $\theta_{f,i,j+1}$ , therefore  $e = e_{f,i,j} = (\theta_{f,i,j}, \theta_{f,i,j+1})$ . This proves that the  $e_{f,i,j}$  are actually open edges of  $\mathcal{D}_{f,i}$ . In addition, since the union of the  $v_{f,i,j}$  and  $e_{f,i,j}$  is equal to  $\mathbb{B}_{f,i}$ , it proves that there are no other cells in  $\mathcal{D}_{f,i}$ . In conclusion,  $N_{f,i}^{(e)} = N_{f,i}^{(v)}$ , and  $\mathcal{D}_{f,i}$  is a disjoint union of  $N_{f,i}^{(v)}$  vertices and  $N_{f,i}^{(v)}$  open edges.

□



**Figure B.8:** More examples of valid PCS complexes. *Is has to be imagined embedded in  $\mathbb{R}^4$ , i.e. with no “self-intersection” of the Klein bottle or the sphere with three holes glued together.*

Using the above lemma, the cell complex constraints for a face  $f \in \mathcal{C}$  can be rewritten to:

- Case 1a ( $N_{f,i} = 0$ ):  $\mathbb{B}_{f,i}$  is mapped homeomorphically by  $\Phi_f$  to a single closed edge  $e_{f,i} \in \mathcal{C}$ , or
- Case 1b ( $N_{f,i} \geq 1$ ):  $\mathbb{B}_{f,i}$  is decomposed into  $N_{f,i} \geq 1$  vertices, each mapped by  $\Phi_f$  to a vertex  $v_{f,i,j} \in \mathcal{C}$ , and  $N_{f,i}$  open edges, each mapped homeomorphically by  $\Phi_f$  to an open edge  $e_{f,i,j} \in \mathcal{C}$ , or
- Case 2:  $\mathbb{B}_{f,i}$  is mapped by  $\Phi_f$  to a single vertex  $v_{f,i} \in \mathcal{C}$ , or
- Case 3:  $\mathbb{B}_{f,i}$  is mapped by  $\Phi_f$  by being wrapped  $N_{f,i}$  times around a closed edge  $e_{f,i} \in \mathcal{C}$ .

Finally, we can observe that Case 1a. is already taken into account by Case 3. (wrapping one time around a closed edge), and therefore can be ignored. These three cases are illustrated in Figure B.1.

By combining all the information that we have shown, we are finally able to provide a characterization of PCS complexes: a decomposition of  $X$  into a finite disjoint union of vertices, edges and faces is a PCS complex if and only if (see next page):

For all open edge  $e \in \mathcal{C}$ ,

$$\begin{aligned} \text{there exist } & \begin{cases} \Phi_e : [0, 1] \rightarrow X \text{ continuous} \\ v_{e,\text{start}} \in \mathcal{C} \\ v_{e,\text{end}} \in \mathcal{C} \end{cases} \\ \text{such that } & \begin{cases} \Phi_e : (0, 1) \rightarrow e \text{ homeomorphism} \\ \Phi_e(0) = v_{e,\text{start}} \\ \Phi_e(1) = v_{e,\text{end}} \end{cases} \end{aligned}$$

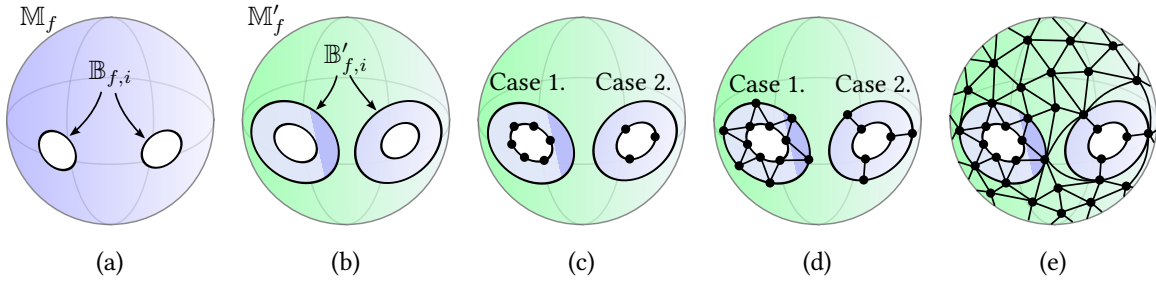
And for all  $(\epsilon, g, k)$ -face  $f \in \mathcal{C}$ ,

$$\begin{aligned} \text{there exist } & \begin{cases} \Phi_f : \mathbb{F}_{\epsilon,g,k} \rightarrow X \text{ continuous} \\ \text{a partition of } [1..k] \text{ into } I_{f,1}, I_{f,2} \text{ and } I_{f,3} \\ \forall i \in I_{f,1}, \begin{cases} N_{f,i} \in \mathbb{N}, N_{f,i} \geq 1 \\ \forall j \in [1..N_{f,i}], v_{f,i,j} \text{ vertex of } \mathcal{C} \\ \forall j \in [1..N_{f,i}], e_{f,i,j} \text{ open edge of } \mathcal{C} \end{cases} \\ \forall i \in I_{f,2}, \begin{cases} v_{f,i} \text{ vertex of } \mathcal{C} \end{cases} \\ \forall i \in I_{f,3}, \begin{cases} N_{f,i} \in \mathbb{N}, N_{f,i} \geq 1 \\ e_{f,i} \text{ closed edge of } \mathcal{C} \end{cases} \end{cases} \quad (\text{B.6}) \\ \text{such that } & \begin{cases} \Phi_f : \text{int}(\mathbb{F}_{\epsilon,g,k}) \rightarrow f \text{ homeomorphism} \\ \forall i \in I_{f,1}, \forall j \in [1..N_{f,i}], \\ \quad \begin{cases} \Phi_f(\mathbf{v}_{f,i,j}) = v_{f,i,j} \\ \Phi_f : \mathbf{e}_{f,i,j} \rightarrow e_{f,i,j} \text{ homeomorphism} \end{cases} \\ \forall i \in I_{f,2}, \Phi_f(\mathbb{B}_{f,i}) = v_{i,f} \\ \forall i \in I_{f,3}, \Phi_f : \mathbb{B}_{f,i} \rightarrow e_{f,i} \text{ wraps } N_{f,i} \text{ times around } e_{f,i} \end{cases} \\ \text{where } & \begin{cases} \mathbb{B}_{f,i} \text{ is the } i\text{-th boundary component of } \mathbb{F}_{\epsilon,g,k} \\ \text{and } \{\dots, \mathbf{v}_{f,i,j}, \dots, \mathbf{e}_{f,i,j}, \dots\} \text{ is a} \\ \text{decomposition of } \mathbb{B}_{f,i} \text{ into } N_{f,i} \text{ vertices and} \\ N_{f,i} \text{ open edges, ordered clockwise or counter-clockwise.} \end{cases} \end{aligned}$$

This is, for the case of the dimension 2, an equivalent formulation of the cell complex constraints described in Section B.1. It is much less compact and does not scale in dimension, but exhaustively describes the different types of cells involved and how they can be glued together. Examples of valid PCS complex are given in Figure B.7 and B.8. By comparing this characterization and the definition of abstract PCS complexes (see Section 3.3.1), one can notice that they are in correspondence, and deduce the equivalence between the two definitions.

## Appendix C

# Equivalence between PCS-Decomposable and 2-Triangulable Spaces



**Figure C.1:** Steps in the construction of the mixed triangulation-quadrangulation of  $\mathbb{M}_f$ , from the proof of Proposition 7.

In this section, we show that the class of topological spaces that can be decomposed as a PCS complex is the same as the class of topological spaces that admits a 2-triangulation. This shows that PCS complexes are able to represent any “reasonable” two-dimensional object.

**Proposition 6.** *A topological space that admits a 2-triangulation can be decomposed as a PCS complex.*

*Proof.* This proposition comes directly from the observation that a 2-triangulation of a space  $X$  is in fact also a valid PCS decomposition. Indeed, a 0-simplex is a PCS vertex, a 1-simplex is a PCS open edge, and a 2-simplex is a PCS face whose characteristic manifold is  $\mathbb{F}_{\circ,0,1}$  (the sphere with one hole), with its unique boundary component  $\mathbb{B}_{f,1}$  decomposed into three vertices and three open edges, each mapped homeomorphically to vertices and open edges.  $\square$

**Proposition 7.** *A topological space that can be decomposed as a PCS complex admits a 2-triangulation.*

*Proof.* To prove this statement, we provide an explicit construction of the triangulation. Let  $\mathcal{K} = (X, \mathcal{C})$  be a PCS complex. Without loss of generality, we assume that  $\mathcal{C}$  does not contain any closed edge. Indeed, any valid PCS decomposition  $(X, \mathcal{C})$  can be preliminary turned into a valid PCS decomposition  $(X, \mathcal{C}')$  that does not contain closed edges, by partitioning every closed edge of  $\mathcal{C}$  into an open edge and a vertex. Now, let us start the construction of the triangulation:

- The vertices in  $\mathcal{C}$  are 0-simplices of the triangulation.
- The open edges in  $\mathcal{C}$  are all split into three 1-simplices and two 0-simplices.

These two first steps have constructed a *valid* 1-triangulation of the 1-skeleton of  $X$ . Splitting each edge in three is necessary to ensure that each 1-simplex has a different start and end 0-simplex, and that any given pair of 0-simplices is connected by *at most* one 1-simplex. Now, let us triangulate the faces:

- Let  $f$  be a face in  $\mathcal{C}$ .
- Let  $\mathbb{M}_f$  be the characteristic manifold of  $f$ ,  $\Phi_f$  be the characteristic map, and  $\mathbb{B}_{f,i} \cong \mathbb{S}^1$  be the boundary components of  $\mathbb{M}_f$ .
- Let  $\mathbb{M}'_f$  be a compact submanifold of  $\mathbb{M}_f$  obtained by “offsetting by a small amount” the holes of  $\mathbb{M}_f$  (cf. Figure C.1(b)). We call  $\mathbb{B}'_{f,i}$  the boundary components of  $\mathbb{M}'_f$ .
- For each boundary component  $\mathbb{B}_{f,i}$  mapped by  $\Phi_f$  to vertices and open edges (Case 1., i.e. when  $i \in I_{f,1}$  from the characterization Eq. B.6), we 1-triangulate  $\mathbb{B}_{f,i}$  by using the pre-image by  $\Phi_f$  of the previously constructed 1-triangulation of  $\partial f$  (cf. Figure C.1(c), left hole). This 1-triangulation has  $3N_{f,i}$  0-simplices, and the same number of 1-simplices.
- For each boundary component  $\mathbb{B}_{f,i}$  mapped by  $\Phi_f$  to a single vertex (Case 2.), we arbitrarily 1-triangulate  $\mathbb{B}_{f,i}$  using three 0-simplices, and three 1-simplices (cf. Figure C.1(c), right hole). We note that Case 3. can be ignored since  $\mathcal{C}$  does not contain any closed edge.
- For each boundary component  $\mathbb{B}_{f,i}$  in Case 1., we arbitrarily 1-triangulate  $\mathbb{B}'_{f,i}$  using  $3N_{f,i}$  0-simplices and  $3N_{f,i}$  1-simplices. Then, we 2-triangulate the topological cylinder between  $\mathbb{B}_{f,i}$  and  $\mathbb{B}'_{f,i}$  using the pattern illustrated in Figure C.1(d), left hole.
- For each boundary component  $\mathbb{B}_{f,i}$  in Case 2., we arbitrarily 1-triangulate  $\mathbb{B}'_{f,i}$  using three 0-simplices and three 1-simplices. Then, we *quadrangulate* the topological cylinder between  $\mathbb{B}_{f,i}$  and  $\mathbb{B}'_{f,i}$  using the pattern illustrated in Figure C.1(d), right hole.
- Finally, we 2-triangulate  $\mathbb{M}'_f$  by preserving the existing 1-triangulation of its boundary, that we know is possible since  $\mathbb{M}'_f$  is a compact 2-manifold (cf. Figure C.1(e)).
- At this stage of the construction, we have obtained a mixed triangulation-quadrangulation  $\mathcal{T}$  of  $\mathbb{M}_f$ . To conclude the construction, we define the triangulation of  $f$  to be the image of  $\mathcal{T}$  by  $\Phi_f$ .

The reader can verify that the quads of  $\mathbb{M}_f$  become triangles of  $f$  since in Case 2.,  $\mathbb{B}_{f,i}$  shrinks to a single vertex. Also, due the homeomorphism properties of  $\Phi_f$ , the triangles stay triangles, no 1-simplex of  $f$  has its start 0-simplex equal to its end 0-simplex, and no pair of 0-simplices

are connected by 2 or more 1-simplices (this is guaranteed by the specific triangle pattern chosen around  $\mathbb{B}_{f,i}$  for Case 1.). Therefore, by performing this process for all faces, we obtain a valid 2-triangulation of  $X$ .  $\square$

# Appendix D

## Topological Operators on PCS Complexes

In this appendix, we detail topological operators acting on PCS complexes. More precisely, we provide algorithms on abstract PCS complexes that correspond to well-defined geometric operations on (non-abstract) PCS complexes. For example, the *cut* operator is geometrically defined as partitioning an existing cell into several cells, and we use this definition to classify all the different types of cuts that are possible (for example, cutting a Möbius strip along an edge may either disconnect it or not, depending on the geometry of the edge). The concept of PCS complex was necessary to infer the combinatorial algorithms from the geometric definitions of operators, but the algorithms themselves can be used on both abstract PCS complexes and vector graphics complexes.

### D.1 Notations

**Vertices** We use the notation  $v$  to refer to a vertex, and the notation  $V$  to refer to the set of all vertices.

**Open edges** We use the notation  $e^|$  to refer to an open edge, or simply  $e$  when it is clear from the context that the edge is open. We denote by  $E_|$  the set of all open edges. Sometimes we use the abuse of notation  $e = (v_{\text{start}}, v_{\text{end}})$  to define or refer to an edge  $e$  whose ordered boundary is  $\widehat{\partial}e = (v_{\text{start}}, v_{\text{end}})$ .

**Closed edges** We use the notation  $e^\circ$  to refer to a closed edge, or simply  $e$  when it is clear from the context that the edge is closed. We denote by  $E_\circ$  the set of all closed edges.

**Edges** We use the notation  $e$  to refer to an edge that can be either open or closed, and we denote by  $E = E_| \cup E_\circ$  the set of all edges.

**Halfedges** We use the notation  $h = (e, \beta)$ , with  $\beta \in \{\top, \perp\}$ , to refer to or define a halfedge that can be either open or closed. We use the notation  $h^\circ$  to refer specifically to a closed halfedge, and the notation  $h^|$  to refer specifically to an open halfedge. Similarly, we denote by  $H_|$  the set of all open halfedges, by  $H_\circ$  the set of all closed halfedges, and by  $H = H_| \cup H_\circ$  the set of all

halfedges. Finally, we use the notation  $e(h)$  and  $\beta(h)$  to refer to the first and second components of the pair defining the halfedge.

**Steiner cycles** We use the notation  $\gamma^\bullet$  to refer to a Steiner cycle, or simply  $\gamma$  when it is clear from the context that we refer to a Steiner cycle. We use the notation  $\Gamma_\bullet$  to refer to the set of all possible Steiner cycles. We use the notation  $v(\gamma^\bullet)$  to refer to the vertex that defines a Steiner cycle, and the following notation to refer to or define a Steiner cycle together with its vertex  $v$ :

$$\gamma^\bullet = [v] \quad (\text{D.1})$$

**Simple cycles** We use the notation  $\gamma^\circ$  to refer to a simple cycle, or simply  $\gamma$  when it is clear from the context that we refer to a simple cycle. We use the notation  $\Gamma_\circ$  to refer to the set of all possible simple cycles. We use the notations  $h^\circ(\gamma^\circ)$  and  $N(\gamma^\circ)$  to refer to the closed halfedge and integer that define a simple cycle. We also use the convenient notations  $e^\circ(\gamma^\circ) = e(h^\circ(\gamma^\circ))$  and  $\beta(\gamma^\circ) = \beta(h^\circ(\gamma^\circ))$ . Finally, we use the following notation to refer to or define a simple cycle together with its defining components:

$$\gamma^\circ = [h^{\circ N}] = [(e^\circ, \beta)^N] \quad (\text{D.2})$$

**Non-simple cycles** We use the notation  $\gamma^+$  to refer to a non-simple cycle, or simply  $\gamma$  when it is clear from the context that we refer to a non-simple cycle. We use the notation  $\Gamma_+$  to refer to the set of all possible non-simple cycles. We use the following notation to refer to or define a non-simple cycle together with its defining open halfedges:

$$\gamma^+ = [h_1, \dots, h_N] = [(e_1, \beta_1), \dots, (e_N, \beta_N)] \quad (\text{D.3})$$

We conveniently refer to these objects by  $N(\gamma)$ ,  $h_j(\gamma)$ ,  $e_j(\gamma)$  and  $\beta_j(\gamma)$ , where  $j \in \mathbb{N}$  is considered modulo  $N$  (e.g.,  $h_0(\gamma)$  is well-defined and refers to  $h_N$ ). We use the notation  $v_i(\gamma) = v_{\text{end}}(h_i(\gamma))$ . In particular, we have  $v_0(\gamma) = v_N(\gamma)$ , and to conveniently visualize all the cells involved in a non-simple cycle, we use the notation:

$$\gamma^+ = [ \begin{array}{c} \bullet \\ v_0 \end{array} (e_1, \beta_1) \begin{array}{c} \bullet \\ v_1 \end{array} \cdots \begin{array}{c} \bullet \\ v_{N-1} \end{array} (e_N, \beta_N) \begin{array}{c} \bullet \\ v_N \end{array} ] \quad (\text{D.4})$$

**Cycles** We use the notation  $\gamma$  to refer to a cycle that can be Steiner, simple or non-simple.



**Faces** We use the notation  $f$  to refer to a face. To conveniently refer to or define a face  $f$  together with its ordered boundary  $\hat{\partial}f$ , we use the following abuse of notation:

$$f = (\epsilon, g, [\gamma_1, \dots, \gamma_k]) \quad (\text{D.5})$$

or simply

$$f = [\gamma_1, \dots, \gamma_k] \quad (\text{D.6})$$

when  $\epsilon$  and  $g$  are irrelevant or clear from context. We conveniently refer to these objects by  $\epsilon(f)$ ,  $g(f)$ ,  $k(f)$ , and  $\gamma_i(f)$ . It is possible that  $f = []$ , in which case  $f$  is a face without boundary (we do not use a special notation for faces without boundary).

## D.2 Algebraic Operations on Halfedges, Paths and Cycles

In order to describe more conveniently the topological operators on abstract PCS complexes, we first introduce the notion of *paths*, and a few basic algebraic operations on halfedges, paths and cycles, which are: flipping a halfedge, a path, or a cycle; converting an open halfedge to a path and a path to a cycle; concatenating paths to create a longer path; rotating a non-simple cycle; and extracting a subpath from a path or a non-simple cycle.

### D.2.1 Paths

Given an abstract PCS complex  $\mathcal{P}$ , a **path** is defined as a triplet  $\pi = (v_{\text{start}}, (h_j)_{j \in [0..N]}, v_{\text{end}}) \in \Pi = V \times H^* \times V$  satisfying the following constraints:

- if  $N = 0$  (i.e., the sequence  $(h_j)$  is empty), then  $v_{\text{start}} = v_{\text{end}}$
- if  $N > 0$  (i.e., the sequence  $(h_j)$  is not empty), then
  - $v_{\text{start}} = v_{\text{start}}(h_1)$
  - $\forall j \in [1..N - 1], v_{\text{end}}(h_j) = v_{\text{start}}(h_{j+1})$
  - $v_{\text{end}}(h_N) = v_{\text{end}}$

Intuitively, a path starts at a given vertex  $v_{\text{start}}$ , then travels along  $N \geq 0$  edges  $e_i$  with a given direction  $\beta_i$ , and finally ends its course at a vertex  $v_{\text{end}}$ . If  $N = 0$ , we conveniently use the notation  $\pi = [v]$  instead of  $\pi = (v, [], v)$ . If  $N > 0$ , we conveniently use the notation  $\pi = [h_1, \dots, h_N]$  instead of  $\pi = (v_{\text{start}}, [h_1, \dots, h_N], v_{\text{end}})$ , since the start and end vertices can be inferred from the halfedges. The integer  $N \in \mathbb{N}$  is called the **length** of the path. While the notion of path shares similarities with the notion of cycle (e.g., can be reduced to a single vertex), we note that

the concept of “simple path” does not exist: a path necessarily starts and ends at given vertices (possibly equal), therefore it cannot contain closed edges. To better emphasize the differences between paths and cycles, we use the following terminology: if  $N = 0$ , we refer to the path as a **trivial path** (rather than a “Steiner path”); and if  $N > 0$ , we refer to the path as a **non-trivial path** (rather than a “non-simple path”).

### D.2.2 Flipping Halfedges, Paths and Cycles

Given a halfedge  $h = (e, \beta)$ , we define its **flipped halfedge** as:

$$\bar{h} = (e, \bar{\beta}), \quad \text{where } \bar{\beta} = \begin{cases} \perp & \text{if } \beta = \top \\ \top & \text{if } \beta = \perp \end{cases} \quad (\text{D.7})$$

Given a path  $\pi = (v_{\text{start}}, [h_1, \dots, h_N], v_{\text{end}})$ , we define its **flipped path** as:

$$\bar{\pi} = (v_{\text{end}}, [\bar{h}_N, \dots, \bar{h}_1], v_{\text{start}}) \quad (\text{D.8})$$

Given a cycle  $\gamma$ , we define its **flipped cycle** as:

$$\bar{\gamma} = \begin{cases} [v] & \text{if } \gamma = [v] \text{ is a Steiner cycle} \\ [\bar{h}^{\circ N}] & \text{if } \gamma = [h^{\circ N}] \text{ is a simple cycle} \\ [\bar{h}_N, \dots, \bar{h}_1] & \text{if } \gamma = [h_1, \dots, h_N] \text{ is a non-simple cycle} \end{cases} \quad (\text{D.9})$$

### D.2.3 Converting Open Halfedges to Paths and Paths to Cycles

An open halfedge  $h$  can always be interpreted as a path  $\pi$  of length  $N(\pi) = 1$ , using the following conversion:

$$\begin{aligned} H| &\rightarrow \Pi \\ h &\mapsto [h] = (v_{\text{start}}(h), [h], v_{\text{end}}(h)) \end{aligned} \quad (\text{D.10})$$

For conciseness, we will often omit the brackets and simply write  $h$  instead of  $[h]$  when it is clear from the context that we interpret  $h$  as a path.

Similarly, a path satisfying  $v_{\text{start}} = v_{\text{end}}$  can always be interpreted as a cycle (more specifically, a

Steiner cycle if  $N = 0$  and a non-simple cycle if  $N > 0$ ), using the following conversion:

$$\begin{aligned} \{\pi \in \Pi \mid v_{\text{start}} = v_{\text{end}}\} &\rightarrow \Gamma \\ \pi &\mapsto [\pi] = \begin{cases} [v_{\text{start}}] & \text{if } N = 0 \\ [h_1, \dots, h_N] & \text{if } N > 0 \end{cases} \end{aligned} \quad (\text{D.11})$$

For conciseness, we will often omit the brackets and parentheses and simply write  $\pi$  instead of  $[\pi]$  when it is clear from the context that we interpret  $\pi$  as a cycle.

#### D.2.4 Concatenating Paths

Given two paths  $\pi = (v_{\text{start}}, [h_1, \dots, h_N], v_{\text{end}})$  and  $\pi' = (v'_{\text{start}}, [h'_1, \dots, h'_{N'}], v'_{\text{end}})$  satisfying  $v_{\text{end}} = v'_{\text{start}}$ , we define the **concatenation** of  $\pi$  with  $\pi'$  by:

$$[\pi, \pi'] = (v_{\text{start}}, [h_1, \dots, h_N, h'_1, \dots, h'_{N'}], v'_{\text{end}}) \quad (\text{D.12})$$

Since this operation is associative (i.e.,  $[[\pi, \pi'], \pi''] = [\pi, [\pi', \pi'']]$ ), we conveniently omit the extra brackets and simply write  $[\pi_1, \dots, \pi_m]$  when concatenating more than two paths together. Also, since open halfedges can be interpreted as paths of length one, we extend the notation to concatenate paths and halfedges, leading to expressions such as  $[\pi_1, (e, \top), \overline{\pi_2}, (e, \perp)]$ . If  $v_{\text{start}}(\pi_1) = v_{\text{start}}(e)$ , this expression can subsequently be implicitly interpreted as a cycle, so we would simply write  $\gamma = [\pi_1, (e, \top), \overline{\pi_2}, (e, \perp)]$  instead of  $\gamma = [[[ \pi_1, [(e, \top)] ], \overline{\pi_2}, [(e, \perp) ] ]]$ .

In pseudocode, we will often use the wording “Append  $h$  to  $\pi$ ”, which means “ $\pi \leftarrow [\pi, h]$ ”. Finally, we note that as per the definitions, concatenating with a trivial path is a null operation. For instance, if  $\pi_2$  is trivial, then  $[\pi_1, \pi_2, \pi_3] = [\pi_1, \pi_3]$ . In other words, all trivial paths are neutral elements for the concatenation operation.

#### D.2.5 Rotating Non-Simple Cycles

Intuitively, we want a cycle to represent a “loop” made of consecutive halfedges, but we would like the starting point of this loop to be irrelevant. However, in our definition, we defined a non-simple cycle as a sequence of halfedges  $[h_1, \dots, h_N]$  which means that a first halfedge  $h_1$  must be arbitrarily chosen among  $h_1, \dots, h_N$ . A negative consequence is that if  $h_1 \neq h_2$ , then the two cycles  $\gamma_1 = [h_1, h_2]$  and  $\gamma_2 = [h_2, h_1]$  are mathematically different even though they intuitively represent the same cycle. To capture this intuitive notion, we define the following equivalence

relation between non-simple cycles:

$$\gamma_1^+ \sim \gamma_2^+ \Leftrightarrow \exists d \in \mathbb{N}, \forall j \in [1..N], h_j(\gamma_1^+) = h_{j-d}(\gamma_2^+) \quad (\text{D.13})$$

In other words, two non-simple cycles  $\gamma_1^+$  and  $\gamma_2^+$  are **equivalent** if and only if  $\gamma_2^+$  can be obtained from  $\gamma_1^+$  by choosing a different “starting point”, formally done via an operation called a **rotation**, defined by:

$$\begin{aligned} \text{Rot}_d : \quad \Gamma_+ &\rightarrow \Gamma_+ \\ \gamma^+ = [h_1, \dots, h_N] &\mapsto \text{Rot}_d(\gamma^+) = [h_{(1+d) \bmod N}, \dots, h_{(N+d) \bmod N}] \end{aligned} \quad (\text{D.14})$$

Using this operation, the equivalence relation can be rewritten as:

$$\gamma_1^+ \sim \gamma_2^+ \Leftrightarrow \exists d \in \mathbb{N}, \gamma_2^+ = \text{Rot}_d(\gamma_1^+) \quad (\text{D.15})$$

We extend the equivalence relation to all types of cycles by defining:

- Two Steiner cycles  $\gamma_1^\bullet$  and  $\gamma_2^\bullet$  are equivalent iff  $v(\gamma_1^\bullet) = v(\gamma_2^\bullet)$ .
- Two simple cycles  $\gamma_1^\circ$  and  $\gamma_2^\circ$  are equivalent iff  $e^\circ(\gamma_1^\circ) = e^\circ(\gamma_2^\circ)$ ,  $\beta(\gamma_1^\circ) = \beta(\gamma_2^\circ)$ , and  $N(\gamma_1^\circ) = N(\gamma_2^\circ)$
- Two cycles  $\gamma_1$  and  $\gamma_2$  of different nature (i.e., non-simple, Steiner or simple) are not equivalent.

We note that while a more carefully crafted definition of cycles would avoid the need for such an equivalence relation, we would lose a lot of clarity and the convenience of referring to a halfedge via its index. In addition, this simpler definition is closer to our actual implementation and hence has a practical value. Finally, we also note that using a circular linked list instead of an indexed sequence does *not* avoid the theoretical and practical need for an equivalence relation, since the circular linked list must still arbitrarily point to one element of the list, and hence testing for “logical equality” between two circular linked lists also requires to take into account rotations, in this case simply achieved by pointing to a different element in the list.

### D.2.6 Extracting Subpaths from Paths and Non-Simple Cycles

Given a path  $\pi = (v_{\text{start}}, [h_1, \dots, h_N], v_{\text{end}})$  and two indices  $j_{\text{start}}$  and  $j_{\text{end}}$  satisfying  $0 \leq j_{\text{start}} \leq j_{\text{end}} \leq N$ , we define the **subpath**  $\pi' = \pi[j_{\text{start}}; j_{\text{end}}]$  by:

$$\pi[j_{\text{start}}; j_{\text{end}}] = \begin{cases} [v_{\text{start}}] & \text{if } j_{\text{start}} = j_{\text{end}} = 0 \\ [v_{\text{end}}(h_{j_{\text{end}}})] & \text{if } j_{\text{start}} = j_{\text{end}} \neq 0 \\ [h_{j_{\text{start}}+1}, \dots, h_{j_{\text{end}}}] & \text{otherwise (i.e., if } j_{\text{start}} < j_{\text{end}}) \end{cases} \quad (\text{D.16})$$

Given a non-simple cycle  $\gamma = [h_1, \dots, h_N]$  and two indices  $j_{\text{start}}$  and  $j_{\text{end}}$ , we define the **subpath**  $\pi' = \gamma[j_{\text{start}}; j_{\text{end}}]$  by:

$$\gamma[j_{\text{start}}; j_{\text{end}}] = \begin{cases} [v_{\text{end}}(h_{\overline{j_{\text{end}}}})] & \text{if } \overline{j_{\text{start}}} = \overline{j_{\text{end}}} \\ [h_{\overline{j_{\text{start}}+1}}, \dots, h_{\overline{j_{\text{end}}}}] & \text{if } \overline{j_{\text{start}} + 1} \leq \overline{j_{\text{end}}} \\ [h_{\overline{j_{\text{start}}+1}}, \dots, h_N, h_1, \dots, h_{\overline{j_{\text{end}}}}] & \text{if } \overline{j_{\text{start}} + 1} > \overline{j_{\text{end}}} \end{cases} \quad (\text{D.17})$$

where  $\overline{j} = j \bmod N$ . The above formal definition can be implemented with the following pseudocode:

---

**SubPath** ( $\gamma \in \Gamma_+$ ,  $j_{\text{start}} \in \mathbb{N}$ ,  $j_{\text{end}} \in \mathbb{N}$ )

---

```

1  $\pi \leftarrow [v_{j_{\text{start}}}(\gamma)]$ 
2  $j \leftarrow j_{\text{start}}$ 
3 while  $j \not\equiv j_{\text{end}} \pmod{N(\gamma)}$  do
4    $j \leftarrow j + 1$ 
5   Append  $h_j(\gamma)$  to  $\pi$ 
6 return  $\pi$ 
```

---

## D.3 Cell Creation

In this section and all following sections, we finally define PCS topological operators, that is, operations that transform a valid abstract PCS complex into another valid abstract PCS complex, given some relevant input. These PCS topological operators apply to vector graphics complexes as well: just ignore all genres and orientabilities. This is possible since, except in two exceptional cases, genres and orientabilities are never used to determine what actions to take: they are only used to compute other genres and orientabilities. The two exceptional cases are `CutNonOrientableFaceAtNonDisconnectingOrientingClosedEdge()` and `CutNonOrientableFaceAtNonDisconnectingOrientingOpenEdge()`. In these cases, the “if” branching can be seen as two alternatives that are

both valid. Since the input always includes a valid abstract PCS complex, and the output is always a valid abstract PCS complex, we do not mention them, and instead we assume that we are working on a globally accessible abstract PCS complex  $\mathcal{P} = (C, \dim, \text{isClosed}, \epsilon, g, k, \hat{\partial})$  that is modified in-place by the topological operator.

Our first and simplest topological operator is *cell creation*. Creating a cell means adding to  $C$  a new symbol  $c$  that is not already contained in  $C$ , and defining the value of  $\dim(c)$  and  $\hat{\partial}c$  for this new symbol. If  $\dim(c) = 1$ , we also need to define  $\text{isClosed}(c)$ , and if  $\dim(c) = 2$ , we also need to define  $\epsilon(c)$ ,  $g(c)$ , and  $k(c)$ . Below are the topological operators for vertices and edges:

---

**CreateVertex ()**


---

```

1 Let  $v \notin C$  ▷ Memory allocation in real-life code, cf. next paragraph
2  $\dim(v) \leftarrow 0$ 
3  $\hat{\partial}v \leftarrow \emptyset$ 
4 Insert  $v$  in  $C$ 
5 return  $v$ 
```

---



---

**CreateClosedEdge ()**


---

```

1 Let  $e^\circ \notin C$ 
2  $\dim(e^\circ) \leftarrow 1$ 
3  $\text{isClosed}(e^\circ) \leftarrow \top$ 
4  $\hat{\partial}e^\circ \leftarrow \emptyset$ 
5 Insert  $e^\circ$  in  $C$ 
6 return  $e^\circ$ 
```

---



---

**CreateOpenEdge ( $v_{\text{start}} \in V, v_{\text{end}} \in V$ )**


---

```

1 Let  $e \notin C$ 
2  $\dim(e) \leftarrow 1$ 
3  $\text{isClosed}(e) \leftarrow \perp$ 
4  $\hat{\partial}e \leftarrow (v_{\text{start}}, v_{\text{end}})$ 
5 Insert  $e$  in  $C$ 
6 return  $e$ 
```

---

In real-life code, “finding a new symbol not already in  $C$ ” is typically a memory allocation. For instance, in C++, “Let  $c \notin C$ ” might translate to “Cell \* c = new Cell;”. Also, defining the value  $\dim(c)$  or  $\text{isClosed}(c)$  may translate to “do nothing” when implemented with an object-oriented language where the type of  $c$  already tells you if it’s a vertex, an edge, or a face, and if the closedness of an edge can be inferred from whether some pointers are null pointers or not. We clarify this with a C++ snippet:

```
1 class Cell
2 {
3 public:
4     virtual int dimension() const=0;
5     virtual std::set<Cell*> boundary() const=0;
6 };
7
8 class Vertex: public Cell
9 {
10 public:
11     // Returns the dimension of this Vertex (= 0)
12     int dimension() const { return 0; }
13
14     // Returns the boundary of this Vertex (= empty set)
15     std::set<Cell*> boundary() const { return std::set<Cell*>(); }
16 };
17
18 class Edge: public Cell
19 {
20 private:
21     Vertex * start_;
22     Vertex * end_;
23
24 public:
25     // Creates a closed edge
26     Edge() : start_(nullptr), end_(nullptr) {}
27
28     // Creates a open edge
29     Edge(Vertex * vs, Vertex * ve) : start_(vs), end_(ve) {}
30
31     // Returns whether this Edge is a closed edge or an open edge
32     bool isClosed() const { return start_ == nullptr; }
33
34     // Returns the dimension of this Edge (= 1)
35     int dimension() const { return 1; }
36
37     // Returns the boundary of this Edge (= its end vertices, if any)
38     std::set<Cell*> boundary() const
39     {
40         std::set<Cell*> res;
41         if(isClosed()) // Closed edge
42         {
43             return res;
44         }
45         else // Open edge
46         {
```

```

47         res.insert(start_);
48         res.insert(end_);
49         return res;
50     }
51 }
52 };

```

And then the method CreateClosedEdge() would simply be

```

1 Edge * createClosedEdge()
2 {
3     Edge * e = new Edge();
4     C.insert(e);
5     return e;
6 }

```

Finally, the most atomic way to create a face is in several steps: one step to create a face without boundary, and then one step per cycle to add. Cycles can also be removed afterwards. Since the number of cycles  $k(f)$  can always be inferred from  $\widehat{\partial}f$ , we omit to specify it.

---

**CreateFace** ( $\epsilon_f \in \{\circ, \emptyset\}$ ,  $g_f \in \mathbb{N}$ )

---

```

1 Let  $f \notin C$ 
2  $\dim(f) \leftarrow 2$ 
3  $\widehat{\partial}f \leftarrow []$  ▷ Empty sequence of cycles
4  $\epsilon(f) \leftarrow \epsilon_f$ 
5  $g(f) \leftarrow g_f$ 
6 Insert  $f$  in  $C$ 
7 return  $f$ 

```

---



---

**AddSteinerCycleToFace** ( $f \in F$ ,  $v \in V$ )

---

```

1 Append  $\gamma^\bullet = [v]$  to  $\widehat{\partial}f$ 

```

---



---

**AddSimpleCycleToFace** ( $f \in F$ ,  $e^\circ \in E_\circ$ ,  $\beta \in \{\top, \perp\}$ ,  $N \in \mathbb{N}$ )

---

```

1 Append  $\gamma^\circ = [(e^\circ, \beta)^N]$  to  $\widehat{\partial}f$ 

```

---



---

**AddNonSimpleCycleToFace** ( $f \in F$ ,  $\gamma \in \Gamma_+$ )

---

```

1 Append  $\gamma$  to  $\widehat{\partial}f$ 

```

---



---

**AddCycleToFace** ( $f \in F$ ,  $\gamma \in \Gamma$ )

---

```

1 Append  $\gamma$  to  $\widehat{\partial}f$ 

```

---



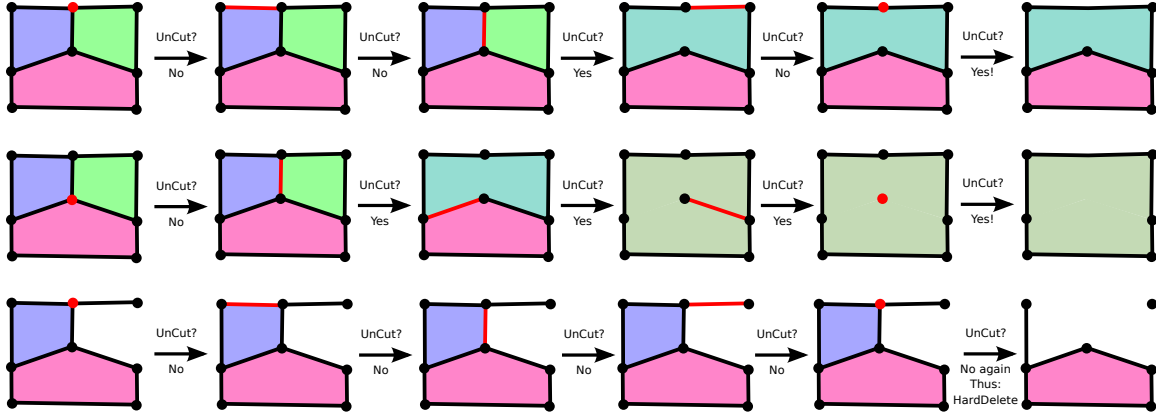


Figure D.1: Three scenarios using *SmartDelete()*.

---

**RemoveCyclesFromFace** ( $f \in F, I \subset \mathbb{N}$ )

---

- 1  $\Delta \leftarrow []$
  - 2 **for all**  $i \in [1..k(f)], i \notin I$  **do**
  - 3     Append  $\gamma_i(f)$  to  $\Delta$
  - 4  $\hat{\partial}f \leftarrow \Delta$
- 

---

**RemoveCycleFromFace** ( $f \in F, i \in \mathbb{N}$ )

---

- 1 RemoveCyclesFromFace( $f, \{i\}$ )
- 

## D.4 Cell Deletion

In the general case, deleting a cell by simply removing it from  $C$  would result in an invalid abstract PCS complex. For instance, if  $C = \{v_1, v_2, e\}$  with  $\hat{\partial}e = (v_1, v_2)$ , then removing  $v_1$  would result in  $C = \{v_2, e\}$  with  $\hat{\partial}e = (v_1, v_2)$ , which is clearly an invalid abstract PCS complex since we must have  $v_{\text{start}}(e) \in V$  which is not anymore the case. More generally, removing a cell  $c$  from  $C$  is valid if and only if we have  $\text{star}(c) = \emptyset$ . Hence, one possible way to define “deletion” is to remove  $c$  and  $\text{star}(c)$  together, an operation that we call “hard delete”, that is safely achieved by the following recursive method:

---

**HardDelete** ( $c \in C$ )

---

- 1 **while**  $\exists c' \in \text{star}(c)$  **do**
  - 2     HardDelete( $c'$ )
  - 3 Remove  $c$  from  $C$  ▷ In real-life code, remove from set, then release memory
-

But there is a less destructive way to remove  $c$  from  $C$ : perform an *atomic simplification* at  $c$ , as defined in Appendix E and illustrated in Figure E.1. This operation is equivalent to the `UnCut()` topological operator defined in Section D.8. However, not all cells are candidate for atomic simplification. So we may think of a method “if can be atomically simplified, atomically simplify; otherwise, hard delete”. But this approach is still too destructive: in the first two scenarios in Figure D.1, it would be equivalent to hard delete, while we can see that a less destructive approach exists. This approach is “if can be simplified, simplify; otherwise, hard delete”, where “simplify cell  $c$ ” corresponds to “recursively simplify all its star cells first, then atomically simplify  $c$ , if possible”. This “smart delete” operation is implemented by the following topological operators:

---

**SmartDelete** ( $c \in C$ )

---

```

1 if  $c \in V$  then
2   SmartDeleteVertex( $c$ )
3 else if  $c \in E$  then
4   SmartDeleteEdge( $c$ )
5 else if  $c \in F$  then
6   SmartDeleteFace( $c$ )
```

---



---

**SmartDeleteFace** ( $f \in F$ )

---

```

1 HardDelete( $f$ )
```

---



---

**SmartDeleteEdge** ( $e \in E$ )

---

```

1 if CanUncutAtEdge( $e$ ) then
2   UnCutAtEdge( $e$ )
3 else
4   HardDelete( $e$ )
```

---



---

**SmartDeleteVertex** ( $v \in V$ )

---

```

1 if CanUncutAtVertex( $v$ ) then
2   UnCutAtVertex( $v$ )
3 else
4   for all Edge  $e \in \text{star}(c)$  do
5     if CanUncutAtEdge( $e$ ) then UnCutAtEdge( $e$ )
6   if CanUncutAtVertex( $v$ ) then
7     UnCutAtVertex( $v$ )
8   else
9     HardDelete( $v$ )
```

---

## D.5 Glue Cells

Gluing is a rather simple topological operator, both conceptually and to implement. To glue two cells  $c_1$  and  $c_2$  of same “type”, the idea is to create a new cell  $c$ , then replace every occurrence of  $c_1$  or  $c_2$  (in the ordered boundary of other cells) by  $c$ , and finally delete  $c_1$  and  $c_2$ . Note that because  $c_1$  or  $c_2$  do not belong anymore to the boundary of any cell, we have  $\text{star}(c_1) = \text{star}(c_2) = \emptyset$ , thus deleting them simply means removing them from  $C$ . In case some geometry is associated to the topology, the geometry of  $c$  would be the “average” of the geometry of  $c_1$  and  $c_2$ , where the exact meaning of “average” depends how the geometry is represented and is a choice of the implementer.

First, let us show what this means for vertices. In order to glue two vertices  $v_1$  or  $v_2$ , you should replace every occurrence of  $v_1$  or  $v_2$  (as a start vertex, end vertex, or Steiner cycle) by the new “glued” vertex  $v$ .

---

**GlueVertices** ( $v_1 \in V, v_2 \in V$ )

---

**Require:**  $v_1 \neq v_2$

```

1  $v \leftarrow \text{CreateVertex}()$ 
2 for all open edge  $e \in \text{star}(v_1) \cup \text{star}(v_2)$  do
3   if  $v_{\text{start}}(e) = v_1$  OR  $v_{\text{start}}(e) = v_2$  then
4      $v_{\text{start}}(e) \leftarrow v$ 
5   if  $v_{\text{end}}(e) = v_1$  OR  $v_{\text{end}}(e) = v_2$  then
6      $v_{\text{end}}(e) \leftarrow v$ 
7 for all face  $f \in \text{star}(v_1) \cup \text{star}(v_2)$  do
8   for all Steiner cycle  $\gamma_i^\bullet \in \hat{\partial}f$  do
9     if  $\gamma_i^\bullet = [v_1]$  OR  $\gamma_i^\bullet = [v_2]$  then
10       $\gamma_i^\bullet \leftarrow [v]$ 
11  $\text{HardDelete}(v_1)$ 
12  $\text{HardDelete}(v_2)$ 
13 return  $v$ 
```

---

Gluing two edges is ambiguous: one needs to decide on a chosen relative direction first. If there is geometry available, simple heuristics should be enough (for instance using the sign of a dot product). Once direction is decided, we are left to glue two halfedges  $(e_1, \beta_1)$  and  $(e_2, \beta_2)$ . To achieve this, we first glue their start vertices and end vertices together (if any), then we create a new edge  $e$ , and replace every occurrence of  $(e_1, \top)$ ,  $(e_1, \perp)$ ,  $(e_2, \top)$ , or  $(e_2, \perp)$  by either  $(e, \top)$  or  $(e, \perp)$ .

---

**GlueClosedHalfedges**  $((e_1^\circ, \beta_1) \in H_\circ, (e_2^\circ, \beta_2) \in H_\circ)$ 


---

**Require:**  $e_1^\circ \neq e_2^\circ$ 

```

1  $e^\circ \leftarrow \text{CreateClosedEdge}()$ 
2 for all face  $f \in \text{star}(e_1^\circ) \cup \text{star}(e_2^\circ)$  do
3   for all simple cycle  $\gamma_i^\circ = [(e_i^\circ, \beta_i)^{N_i}] \in \widehat{\partial}f$  do
4     if  $e_i^\circ = e_1^\circ$  then
5        $\gamma_i^\circ \leftarrow [(e^\circ, (\beta_i \Leftrightarrow \beta_1))^{N_i}]$   $\triangleright$  “ $\beta \Leftrightarrow \beta'$ ” returns  $\top$  if  $\beta = \beta'$ ,  $\perp$  otherwise
6     else if  $e_i^\circ = e_2^\circ$  then
7        $\gamma_i^\circ \leftarrow [(e^\circ, (\beta_i \Leftrightarrow \beta_2))^{N_i}]$ 
8    $\text{HardDelete}(e_1^\circ)$ 
9    $\text{HardDelete}(e_2^\circ)$ 
10 return  $e^\circ$ 

```

---



---

**GlueOpenHalfedges**  $(h_1 = (e_1, \beta_1) \in H|, h_2 = (e_2, \beta_2) \in H|)$ 


---

**Require:**  $e_1 \neq e_2$ 

```

1 if  $v_{\text{start}}(h_1) = v_{\text{start}}(h_2)$  then
2    $v_{\text{start}} \leftarrow v_{\text{start}}(h_1)$ 
3 else
4    $v_{\text{start}} \leftarrow \text{GlueVertices}(v_{\text{start}}(h_1), v_{\text{start}}(h_2))$ 
5 if  $v_{\text{end}}(h_1) = v_{\text{end}}(h_2)$  then
6    $v_{\text{end}} \leftarrow v_{\text{end}}(h_1)$ 
7 else
8    $v_{\text{end}} \leftarrow \text{GlueVertices}(v_{\text{end}}(h_1), v_{\text{end}}(h_2))$ 
9  $e \leftarrow \text{CreateOpenEdge}(v_{\text{start}}, v_{\text{end}})$ 
10 for all face  $f \in \text{star}(e_1) \cup \text{star}(e_2)$  do
11   for all non-simple cycle  $\gamma_i = [h_1, \dots, h_{N_i}] \in \widehat{\partial}f$  do
12     for all halfedge  $h_j = (e_j, \beta_j) \in \gamma_i$  do
13       if  $e_j = e_1$  then
14          $h_j \leftarrow (e, (\beta_j \Leftrightarrow \beta_1))$ 
15       else if  $e_j = e_2$  then
16          $h_j \leftarrow (e, (\beta_j \Leftrightarrow \beta_2))$ 
17    $\text{HardDelete}(e_1)$ 
18    $\text{HardDelete}(e_2)$ 
19 return  $e$ 

```

---

## D.6 UnGlue Cells

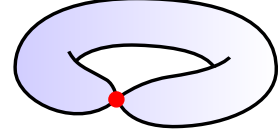
Informally, *unglue* is the “reverse” topological operation of *glue*. However, this is slightly inaccurate. For instance, creating a vertex shared by three edges requires two glue operations, but can be reversed in a single unglue operation. Conversely, gluing two isolated vertices results in a single isolated vertex, but ungluing at this vertex is a null operation instead of reversing back into two isolated vertices. We illustrate this with a few examples below:

$$\begin{aligned}
 & \left\{ \begin{array}{c} v_1 \\ v_2 \end{array} \right\} \xrightarrow{\text{Glue}(v_1, v_2)} \{ v \} \xrightarrow{\text{UnGlueAt}(v)} \{ v \} \\
 \\
 & \left\{ \begin{array}{c} v_1 \\ v_2 \\ v'_1 \\ v'_2 \\ e_1 = (v_1, v'_1) \\ e_2 = (v_2, v'_2) \end{array} \right\} \xrightarrow{\text{Glue}(v_1, v_2)} \left\{ \begin{array}{c} v \\ v'_1 \\ v'_2 \\ e_1 = (v, v'_1) \\ e_2 = (v, v'_2) \end{array} \right\} \xrightarrow{\text{UnGlueAt}(v)} \left\{ \begin{array}{c} v_1 \\ v_2 \\ v'_1 \\ v'_2 \\ e_1 = (v_1, v'_1) \\ e_2 = (v_2, v'_2) \end{array} \right\} \\
 \\
 & \left\{ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v'_1 \\ v'_2 \\ v'_3 \\ e_1 = (v_1, v'_1) \\ e_2 = (v_2, v'_2) \\ e_3 = (v_3, v'_3) \end{array} \right\} \xrightarrow{\text{Glue}(v_1, v_2)} \left\{ \begin{array}{c} v \\ v_3 \\ v'_1 \\ v'_2 \\ v'_3 \\ e_1 = (v, v'_1) \\ e_2 = (v, v'_2) \\ e_3 = (v_3, v'_3) \end{array} \right\} \xrightarrow{\text{Glue}(v, v_3)} \left\{ \begin{array}{c} v' \\ v'_1 \\ v'_2 \\ v'_3 \\ e_1 = (v', v'_1) \\ e_2 = (v', v'_2) \\ e_3 = (v', v'_3) \end{array} \right\} \xrightarrow{\text{UnGlueAt}(v')} \left\{ \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v'_1 \\ v'_2 \\ v'_3 \\ e_1 = (v_1, v'_1) \\ e_2 = (v_2, v'_2) \\ e_3 = (v_3, v'_3) \end{array} \right\}
 \end{aligned}$$

Fundamentally,  $\text{UnGlue}(c)$  duplicates  $c$  as many times as it is “used” by cells of higher dimension, or do nothing if  $\text{star}(c) = \emptyset$ . We formalize now the notion of “use”, which is similar to the vertex-use, edge-use and face-use in the *radial-edge* data structure [Weiler 1985], but not exactly identical. The fundamental difference is that while in the radial-edge data structure, these uses are *explicit objects* (for instance, vertex-uses are ordered in a cyclic doubly-linked list around the vertex they represent), they are only *implicit* in abstract PCS complexes. Another less significant difference is that the radial-edge data structure does not support Steiner cycles and closed edges, but we believe it could be easily extended to support them. Finally, since the radial-edge data structure is designed to represent solid 3D objects, it also defines volumes via *shells* (shells are for volumes what cycles are for surfaces) and hence defines *face-uses*, while we stop at the dimension 2 and hence do not need them.

**Vertex-use** A vertex  $v$  can be used in three different ways:

- As a start or end vertex of an open edge  $e$  that has no incident face (i.e.,  $\hat{\partial}e = \emptyset$ ). Such a use is called **end-vertex-use** and denoted  $\odot_{e,\beta}$  with  $\beta \in \{\text{start}, \text{end}\}$ . Note that an open edge  $e$  uses twice the same vertex if it has no incident faces and  $v_{\text{start}}(e) = v_{\text{end}}(e)$ . Note also that if  $e$  has incident faces, then it is not considered as using any of its end vertices (otherwise redundant with *corner-vertex-uses*).
- As a Steiner cycle  $\gamma_i^\bullet$  of a face  $f$ . Such a use is called **Steiner-vertex-use** and denoted  $\odot_{f,i}$ . Note that the same vertex can be used as Steiner more than once by the same face. For instance, consider the “pinched torus” made of one vertex  $v$  and one face  $f$  such that  $\hat{\partial}f = [[v], [v]]$ .
- As the vertex  $v_j(\gamma_i)$ , junction between the consecutive halfedges  $h_j$  and  $h_{j+1}$  in a non-simple cycle  $\gamma_i$  of a face  $f$ . Such a use is called **corner-vertex-use** and denoted  $\odot_{f,i,j}$



**Open-edge-use** An open edge  $e$  can only be used in one way: as an open halfedge  $h_j$  in a non-simple cycle  $\gamma_i$  of a face  $f$ . Such a use is called **open-edge-use** and denoted  $\odot_{f,i,j}$ .

**Closed-edge-use** A closed edge  $e^\circ$  can only be used in one way: as a simple cycle  $\gamma_i^\circ$  of a face  $f$ . If  $N_i$  (i.e.,  $N(\gamma_i^\circ)$ ) is greater than one, then the closed edge  $e^\circ$  is considered to be used as many times by the face. Such a use is called **closed-edge-use** and denoted  $\odot_{f,i,j}$ , where  $j$  allows to distinguish repeated uses in the same simple cycle. An example where  $N_i = 2$  is the “cut-Möbius” illustrated in Figure D.2, bottom-middle.

Once this notion of *use* is defined, the unglue topological operators are conceptually simple: to unglue at a cell  $c$ , you create a new cell  $c_k$  for each use  $\odot_k$  of  $c$ , and replace  $c$  by  $c_k$  for this specific use. After this operation,  $c$  is not used anymore and hence we delete it, as shown below:

---

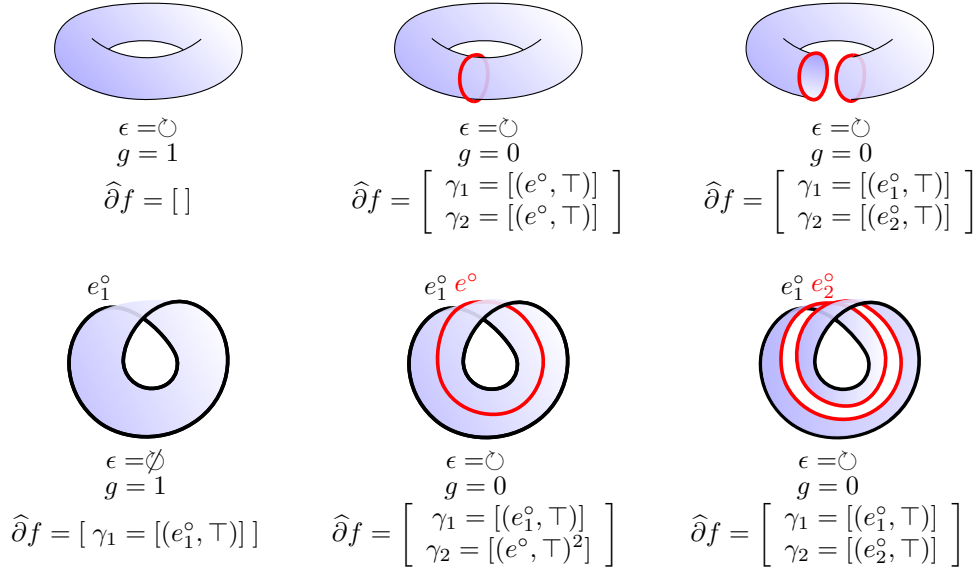
**UnGlueAtOpenEdge** ( $e \in E_l$ )

---

```

1 if star( $e$ ) =  $\emptyset$  then
2   Do nothing.
3 else
4   for all face  $f \in \text{star}(e)$  do
5     for all non-simple cycle  $\gamma_i \in \hat{\partial}f$  do
6       for all halfedge  $h_j \in \gamma_i$  do
7         if  $e(h_j) = e$  then ▷ Found open-edge-use  $\odot_{f,i,j}$ 
8            $e(h_j) \leftarrow \text{CreateOpenEdge}(v_{\text{start}}(e), v_{\text{end}}(e))$ 
9   HardDelete( $e$ )
```

---



**Figure D.2:** The “cut-torus” (top row, middle column) and “cut-Möbius” (bottom row, middle column) are two examples of abstract PCS complexes where a closed edge  $e^\circ$  is used twice by the same face. In the case of the cut-torus, the two closed-edge-uses of  $e^\circ$  come from two simple cycles, while in the case of the cut-Möbius, the two closed-edge-uses of  $e^\circ$  come from a single cycle repeating  $e^\circ$  twice. We show these examples before the cut (left), then after the cut (middle), then after ungluing at the cut edge  $e^\circ$  (right). Ungluing the cut-torus or the cut-Möbius at  $e^\circ$  gives the same abstract PCS complex: the cylinder. It has no vertices, two closed edges  $e_1^\circ$  and  $e_2^\circ$ , and a face  $f$  such that  $\hat{\partial}f = (\circlearrowleft, 0, [[h_1^\circ]; [h_2^\circ]])$ . Indeed, the two surfaces depicted in the right column are both homeomorphic to  $\mathbb{F}_{\circlearrowleft, 0, 2}$ .

The case of closed edges is as easy to implement, but conceptually challenging. If  $N_i = 1$  for all simple cycles  $\gamma_i^\circ$  using  $e^\circ$ , then there are no difficulties. However, the case  $N_i > 1$  is not as straightforward. To understand what the algorithm should do in this case, let us clarify what a “repeated closed edge” represents. Consider a Möbius strip represented by its minimal PCS decomposition (Figure D.2, bottom-left):

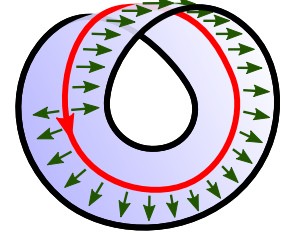
- one closed edge  $e_1^\circ$ : its unique boundary edge
- one face  $f = (\emptyset, 1, [[(e_1^\circ, \top)]])$ : non-orientable, genus-1, one simple cycle

The closed edge  $e_1^\circ$  is only used once by  $f$ . Indeed, if we arbitrarily choose a direction for this closed curve, we can see that locally,  $f$  is only “at the left side” of  $e_1^\circ$ , or “at the right side”, but not on both sides. It is well-known that if you take scissors and cut this Möbius strip in half along its length, you obtain a single orientable surface (Figure D.2, bottom-right). In terms of abstract PCS complexes, this operation can be decomposed into two atomic topological operators:

1. The first topological operator is `CutNonOrientableFaceAtNonDisconnectingOrientingClosedEdge( $f$ )` (see Section D.7.6), and corresponds to “tracing” the red closed edge  $e^\circ$  along

the centerline of the Möbius strip (Figure D.2, bottom-middle). In terms of PCS complexes, this corresponds to partition  $f$  into two cells:  $e^\circ$  and  $f \setminus e^\circ$ . With this new (not minimal) decomposition of the Möbius strip, the cell  $f$  is now homeomorphic to  $\text{int}(\mathbb{F}_{\odot,0,2})$ , while it was homeomorphic to  $\text{int}(\mathbb{F}_{\emptyset,1,1})$  before the cut. However, this does not change the whole topological space  $X$  that the PCS complex represents (i.e., the union of cells), which is still homeomorphic to the Möbius strip  $\mathbb{F}_{\emptyset,1,1}$  (“Cutting” is simply decomposing the same space into more cells as will be discussed later).

After this cut, we can observe that if we arbitrarily choose a direction for this closed curve  $e^\circ$ , then  $f$  is actually “both at the left side and the right side” of  $e^\circ$ . This explains why  $f$  actually uses  $e^\circ$  twice. But unlike the “cut-torus” (Figure D.2, top-middle), these two uses are from the *same* cycle. To understand why, pick a point on  $e^\circ$ , then pick one side of the face (for instance, the “left side”). If you move along  $e^\circ$  while keeping in mind which side of  $f$  you picked, then after one turn you will realize that *you end up at the other side of  $e^\circ$* . Hence, you have to perform two complete turns around  $e^\circ$  to actually complete the cycle, that continuously goes through the two closed-edges-uses.



2. The second topological operator is  $\text{UnglueAtClosedEdge}(e^\circ)$ , that actually changes the topological space  $X$  by “disconnecting” the two closed-edges-uses of  $e^\circ$ . Similarly to the cut-torus example (Figure D.2, top), this is achieved by “duplicating the geometry” of  $e^\circ$ . However, unlike the cut-torus example where this duplicated geometry is distributed among two closed edges  $e_1^\circ$  and  $e_2^\circ$ , the duplicated geometry belongs to the same closed edge  $e_2^\circ$ , making the closed edge twice as long as it was initially (cf. Figure D.2, bottom-right). Combinatorially, this duplication of geometry is conceptual, and it simply means that the simple cycle  $\gamma_i^\circ$  that uses multiple times  $e^\circ$  is transformed into a simple cycle  $\gamma_i^\circ$  that uses only once a new closed edge.

Note that it is also possible to have  $N_i \geq 3$ . For instance, take three rectangles glued together along a long edge, then glue their short edges in the same way you would construct a Möbius strip, but with a third-twist instead of a half-twist. With this understanding, we can finally define the topological operator  $\text{UnglueAtClosedEdge}(e^\circ)$ : for every simple cycle  $\gamma_i^\circ = [(e_i^\circ, \beta_i)^{N_i}]$  that uses  $e^\circ$  (possibly  $N_i > 1$  times), we create a new closed edge  $e_i^\circ$  and change  $\gamma_i^\circ$  into  $[(e_i^\circ, \beta_i)]$ .

Finally, to unglue at a vertex, the important difference is that for this operation to be valid, all cells in the star of  $v$  must be unglued first. Then, we handle independently the different use cases.



---

**UnGlueAtClosedEdge** ( $e^\circ \in E_\circ$ )

---

```

1 if  $\text{star}(e^\circ) = \emptyset$  then
2   Do nothing.
3 else
4   for all face  $f \in \text{star}(e^\circ)$  do
5     for all simple cycle  $\gamma_i^\circ \in \hat{\partial}f$  do
6       if  $e^\circ(\gamma_i^\circ) = e^\circ$  then ▷ Found closed-edge-uses  $\mathbb{C}_{f,i,1}^\circ$  to  $\mathbb{C}_{f,i,N(\gamma_i^\circ)}^\circ$ 
7          $e^\circ(\gamma_i^\circ) \leftarrow \text{CreateClosedEdge}()$ 
8          $N(\gamma_i^\circ) \leftarrow 1$ 
9   HardDelete( $e^\circ$ )

```

---



---

**UnGlueAtVertex** ( $v \in V$ )

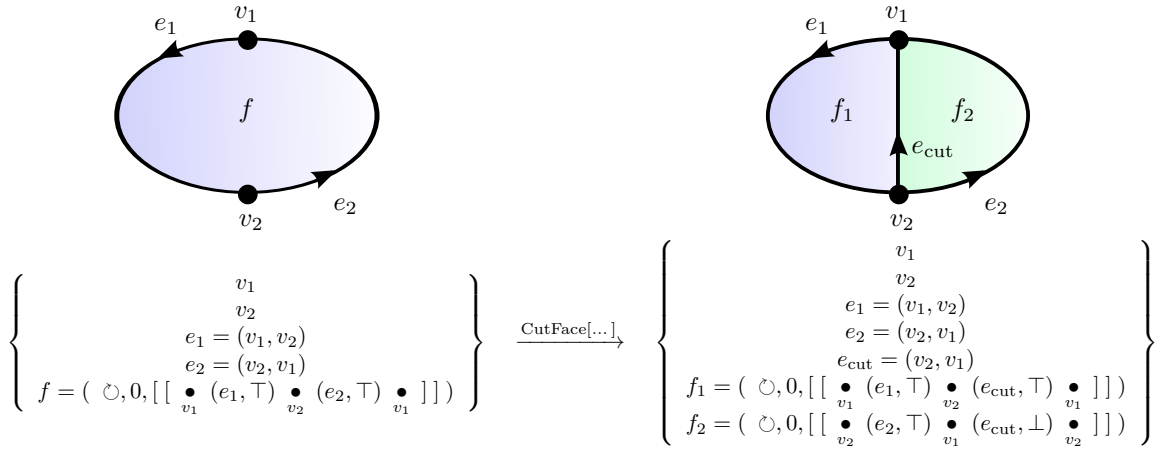
---

```

1 if  $\text{star}(v) = \emptyset$  then
2   Do nothing.
3 else
4   for all edge  $e \in \text{star}(v)$  do ▷ Unglue at all (necessarily open) star edges of  $v$ 
5     UnGlueAtOpenEdge( $e$ )
6   for all edge  $e \in \text{star}(v)$  do
7     if  $\text{star}(e) = \emptyset$  then
8       if  $v_{\text{start}}(e) = v$  then ▷ End-vertex-use  $\mathbb{V}_{e,\text{start}}$ 
9          $v_{\text{start}}(e) \leftarrow \text{CreateVertex}()$ 
10      if  $v_{\text{end}}(e) = v$  then ▷ End-vertex-use  $\mathbb{V}_{e,\text{end}}$ 
11         $v_{\text{end}}(e) \leftarrow \text{CreateVertex}()$ 
12    for all face  $f \in \text{star}(v)$  do
13      for all Steiner cycle  $\gamma_i^\bullet = [v_i] \in \hat{\partial}f$  do
14        if  $v_i = v$  then ▷ Steiner-vertex-use  $\mathbb{V}_{f,i}$ 
15           $v(\gamma_i^\bullet(f)) \leftarrow \text{CreateVertex}()$ 
16      for all non-simple cycle  $\gamma_i \in \hat{\partial}f$  do
17        for all halfedges  $h_j \in \gamma_i$  do
18          if  $v_{\text{end}}(h_j) = v$  then ▷ Corner-vertex-use  $\mathbb{V}_{f,i,j}$ 
19             $v_{f,i,j} \leftarrow \text{CreateVertex}()$ 
20             $v_{\text{end}}(h_j) \leftarrow v_{f,i,j}$ 
21             $v_{\text{start}}(h_{j+1}) \leftarrow v_{f,i,j}$ 
22    HardDelete( $v$ )

```

---



**Figure D.3:** An abstract PCS complex is transformed into another abstract PCS complex, as a result of the cut topological operator  $\text{CutOrientableFaceAtDisconnectingOpenEdge}(f, 1, 1, 2, 0, 0, [ ])$ .

## D.7 Cut Cells

Given a (non-abstract) *PCS complex*, a **cut** is defined as partitioning a cell  $c$  into new cells  $\{c_{\text{cut}}, c_1, c_2, \dots\}$ , where  $c_{\text{cut}}$  is a proper subset of  $c$  and where  $\{c_1, c_2, \dots\}$  are the connected components of  $c \setminus c_{\text{cut}}$ . We say that “ $c$  is cut at  $c_{\text{cut}}$ ”. A **valid cut** is a cut such that the resulting cell decomposition is a valid PCS complex. For instance, if a face  $f$  is cut at an open edge  $e_{\text{cut}} \subset f$ , then  $e_{\text{cut}}$  must start and end at vertices in  $\partial f$ . From now on, whenever we say “a cut”, we mean a valid cut. It can be shown that a cut necessarily satisfies  $\dim(c_{\text{cut}}) < \dim(c)$ . Subsequently, it can be shown that  $c \setminus c_{\text{cut}}$  is either connected or made of two connected components, thus  $c$  is partitioned into either two components  $\{c_{\text{cut}}, c'\}$  or three components  $\{c_{\text{cut}}, c_1, c_2\}$ . This intuitive result should become clear with the different examples illustrated in this section.

The cut topological operator on *abstract PCS complex* that we describe in this section is the combinatorial counterpart of the pointset definition given above. This means that a given abstract PCS complex  $\mathcal{P}$  is transformed into another abstract PCS complex  $\mathcal{P}'$  such that the PCS complex  $|\mathcal{P}'|$  could have been obtained by cutting a cell  $c$  of  $|\mathcal{P}|$  at some subcell  $c_{\text{cut}} \subset c$ . Interestingly, this means that the abstract cell  $c_{\text{cut}}$  of  $\mathcal{P}'$  is actually an *output* of the cut topological operator, as illustrated in Figure D.3, whereas it is more easily interpreted as an *input* with the pointset definition (i.e.: “cut *here*”). Because an abstract PCS complex is a purely combinatorial object, its faces are purely abstract and not assumed to be realized as pointsets (or even as abstract triangulations), and therefore it is not possible to say “cut *here*”, and we should instead say “cut *this way*”. For instance, in Figure D.3, it would be along the lines of “cut  $f$  at an open edge starting at  $v_2$  and ending at  $v_1$ ”. However, while this sentence entirely specifies the cut in the simple example in Figure D.3, things are actually much more complicated in the general case because:

- If  $f$  uses  $v_1$  or  $v_2$  more than once, then the sentence is ambiguous: the actual vertex-uses must be specified instead of “just” the vertices.
- If  $f$  contains several cycles, then the sentence is ambiguous: it is necessary to specify which cycles must be transferred to  $f_1$  and which cycles must be transferred to  $f_2$ .
- If the genus of  $f$  is non-zero (e.g., a torus with a hole), then the sentence is ambiguous: a cut from  $v_2$  to  $v_1$  may or may not disconnect  $f$ , depending on the actual path of  $e_{\text{cut}}$  in a pointset sense, and this information has to be specified combinatorially somehow.
- Other ambiguities that are detailed later.

To exhaustively cover all the possible cases, and find out what combinatorial input is *necessary and sufficient* to fully determine the cut, we have to *classify* all the different *ways* a cell can be cut. Only then we can rigorously define “cut *this way*”, and make sure that we are not missing any *way* to cut a cell. For instance, the cut topological operator performed in Figure D.3 can be more accurately expressed as “cut the orientable face  $f$  at an open edge starting at the vertex-use  $\textcircled{v}_{f,1,1}$ , ending at the vertex-use  $\textcircled{v}_{f,1,2}$ , disconnecting  $f$  into two (necessarily orientable) faces, both of genus zero, and both receiving no cycles from  $f$ ”. To do this, you would call the method `CutOrientableFaceAtDisconnectingOpenEdge( $f,1,1,2,0,0,[ ]$ )`.

We provide below an informal overview of the different cases to consider. The exhaustive list is provided by the following sections, and in particular the different ways to cut a face at an edge are illustrated in Figure D.5 and D.6.

- **Cutting an open edge (at a vertex)**

An open edge  $e$  becomes a vertex  $v$  and two open edges  $e_1$  and  $e_2$ .

- **Cutting a closed edge (at a vertex)**

A closed edge  $e^\circ$  becomes a vertex  $v$  and the open edge  $e' = e^\circ \setminus v$ .

- **Cutting a face at a vertex**

A face  $f$  becomes a vertex  $v$  and the face  $f' = f \setminus v$ .

- **Cutting a face at a closed edge, disconnecting it**

A face  $f$  becomes a closed edge  $e^\circ$  and two faces  $f_1$  and  $f_2$ . The holes, handles or crosscaps of  $f$  are distributed among  $f_1$  and  $f_2$ . The cycle  $[(e^\circ, \top)]$  is added to  $f_1$  and the cycle  $[(e^\circ, \perp)]$  is added to  $f_2$ .

- **Cutting a face at a closed edge, not disconnecting it**

A face  $f$  becomes a closed edge  $e^\circ$  and the face  $f' = f \setminus e^\circ$ . If  $f$  is orientable, its genus is decreased by one, and the two cycles  $[(e^\circ, \top)]$  and  $[(e^\circ, \perp)]$  are added. If  $f$  is non-orientable, it may become orientable or not, its genus may be decreased by one or more, and either the

cycle  $[(e^\circ, \top)]$  is added twice, or the single cycle  $[(e^\circ, \top)^2]$  is added.

- **Cutting a face at an open edge starting/ending at the same hole, disconnecting it**

A face  $f$  becomes an open edge  $e$  and two faces  $f_1$  and  $f_2$ . The handles or crosscaps of  $f$  are distributed among  $f_1$  and  $f_2$ . All cycles except one are distributed among  $f_1$  and  $f_2$ . The last cycle  $\gamma_i = [\pi_1, \pi_2]$  is split by adding  $[\pi_1, (e, \top)]$  to  $f_1$  and adding  $[\pi_2, (e, \perp)]$  to  $f_2$ .

- **Cutting a face at an open edge starting/ending at the same hole, not disconnecting it**

A face  $f$  becomes an open edge  $e$  and the face  $f' = f \setminus e$ . If  $f$  is orientable, its genus is decreased by one, and the cycle  $\gamma_i = [\pi_1, \pi_2]$  is split into  $[\pi_1, (e, \top)]$  and  $[\pi_2, (e, \perp)]$ . If  $f$  is non-orientable, it may become orientable or not, its genus may be decreased by one or more, and either the two cycles  $[\pi_1, (e, \top)]$  and  $[\overline{\pi_2}, (e, \top)]$  are added, or the single cycle  $[\pi_1, (e, \top), \overline{\pi_2}, (e, \top)]$  is added.

- **Cutting a face at an open edge starting/ending at different holes**

A face  $f$  becomes an open edge  $e$  and the face  $f' = f \setminus e$ . Two cycles  $\gamma_{i_1}$  and  $\gamma_{i_2}$  are merged into the single cycle  $[\gamma_{i_1}, (e, \top), \gamma_{i_2}, (e, \perp)]$ .

### D.7.1 Cutting an Open Edge (at a Vertex)

Cutting an open edge  $e$  is a very simple operation that consists in splitting  $e$  in half by inserting a vertex in its interior, resulting in two open edges  $e_1$  and  $e_2$  and one vertex  $v$ . One only has to take care of replacing the halfedges using  $e$  by two halfedges using respectively  $e_1$  and  $e_2$  with the appropriate direction. The reason we put “at a vertex” in parenthesis in the title of this section (and the reason why “AtAVertex” is not part of the topological operator name) is that cutting an open edge is necessarily done at a vertex due to the requirement  $\dim(c_{\text{cut}}) < \dim(c)$ .

**CutOpenEdge** ( $e \in E_l$ )

---

```

1  $v \leftarrow \text{CreateVertex}()$ 
2  $e_1 \leftarrow \text{CreateOpenEdge}(v_{\text{start}}(e), v)$ 
3  $e_2 \leftarrow \text{CreateOpenEdge}(v, v_{\text{end}}(e))$ 
4 for all face  $f \in \text{star}(e)$  do
5   for all non-simple cycle  $\gamma_i \in \widehat{\partial}f$  do
6      $\gamma'_i \leftarrow []$ 
7     for all halfedges  $h_j \in \gamma_i$  do
8       if  $e(h_j) = e$  then
9         if  $\beta(h_j) = \top$  then
10           Append  $(e_1, \top)$  to  $\gamma'_i$ 
11           Append  $(e_2, \top)$  to  $\gamma'_i$ 
12         else
13           Append  $(e_2, \perp)$  to  $\gamma'_i$ 
14           Append  $(e_1, \perp)$  to  $\gamma'_i$ 
15       else
16         Append  $h_j$  to  $\gamma'_i$ 
17      $\gamma_i(f) \leftarrow \gamma'_i$ 
18 HardDelete( $e$ )

```

---

**D.7.2 Cutting a Closed Edge (at a vertex)**

Cutting a closed edge  $e^\circ$  follows the same idea, resulting in one open edge  $e$  and one vertex  $v$ .

**CutClosedEdge** ( $e^\circ \in E_\circ$ )

---

```

1  $v \leftarrow \text{CreateVertex}()$ 
2  $e \leftarrow \text{CreateOpenEdge}(v, v)$ 
3 for all face  $f \in \text{star}(e^\circ)$  do
4   for all simple cycle  $\gamma_i^\circ = [(e_i^\circ, \beta_i)^{N_i}] \in \widehat{\partial}f$  do
5     if  $e_i^\circ = e^\circ$  then
6        $\gamma_i(f) \leftarrow [ \underset{v}{\bullet} (e, \beta_i) \underset{v}{\bullet} \cdots \underset{v}{\bullet} (e, \beta_i) \underset{v}{\bullet} ]$ 

```

Replace the simple cycle by a non-simple cycle that repeats  $N_i$  times the open edge  $e$

```

7 HardDelete( $e^\circ$ )

```

---

▷

### D.7.3 Cutting a Face at a Vertex

A trivial way to cut a face is via a vertex. This means that we decompose the face  $f$  into a new vertex  $v$  and the new face  $f \setminus v$ .

---

**CutFaceAtVertex** ( $f \in F$ )
 

---

```

1  $v \leftarrow \text{CreateVertex}()$ 
2  $\text{AddSteinerCycleToFace}(f, v)$ 

```

---

### D.7.4 Cutting a Face at an Edge

Cutting a face at an open or closed edge is a much harder operation because there are many non-trivial and non-equivalent ways a face can be cut (cf. Figure D.4): the face may become disconnected or not, its orientability and genus may change or not, and different cycles may be added, split, or merged together. If the abstract PCS complex is realized as a triangulation, and the cut edge is given as a subset of the edges in the triangulation, then it is possible to *compute* in which case we are, and perform the appropriate operation. However, at the combinatorial level of the abstract PCS complex, no such realization as triangulation is assumed, therefore “how” the face is cut has to be specified somehow. For instance, if a face  $f$  is a sphere with  $k$  holes, and we cut it at a closed edge  $e^\circ$ , then we know for sure that this disconnects  $f$  into two faces  $f_1$  and  $f_2$ . However, without more information, there is no way to know combinatorially which holes of  $f$  must be transferred to  $f_1$ , and which holes must be transferred to  $f_2$ . Hence, this information has to be given as input of the topological operator. In this case, the operator could be:

---

**CutSphereAtClosedEdge** ( $f \in F, I \subseteq \mathbb{N}$ )
 

---

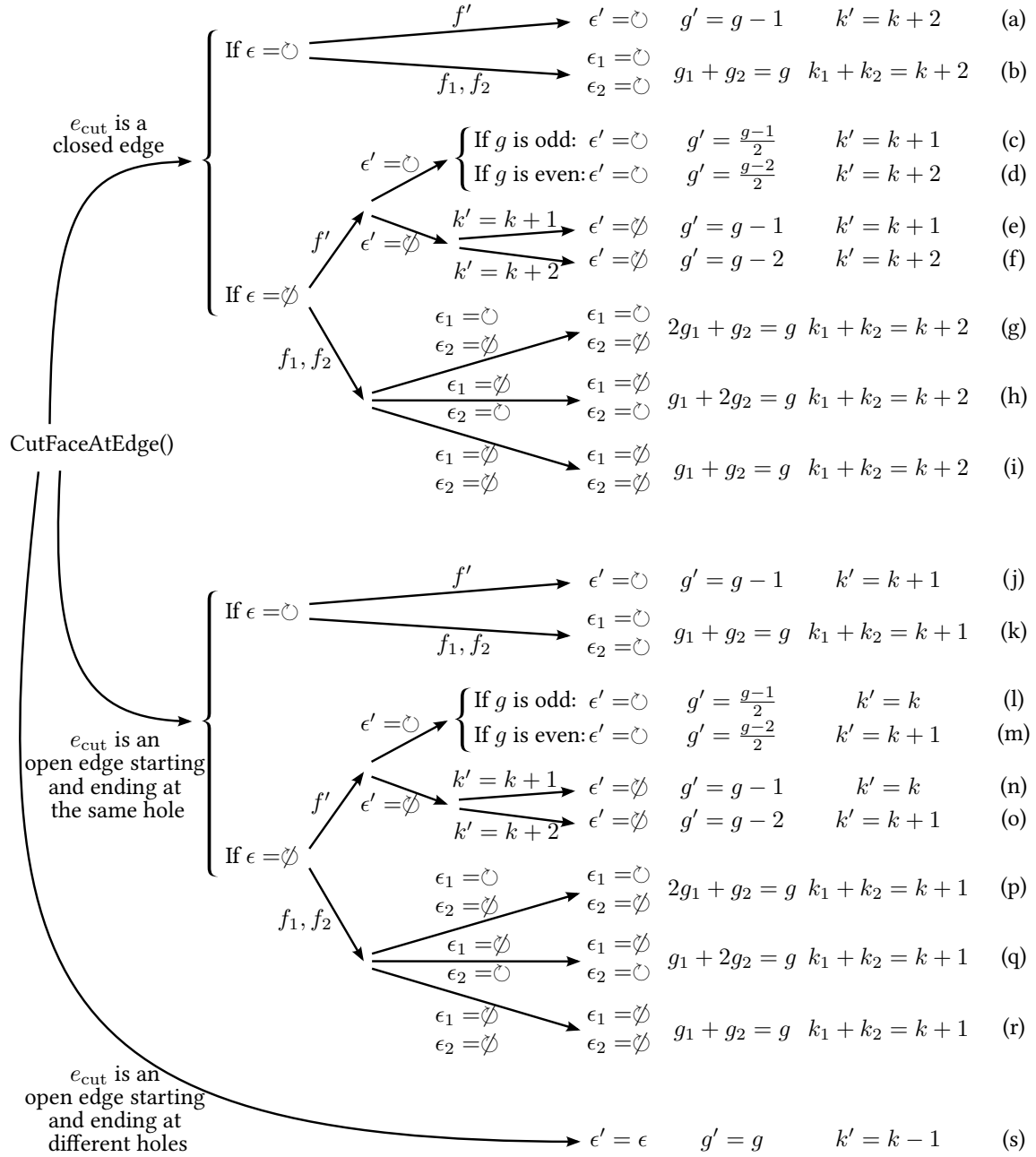
**Require:**  $\epsilon(f) = \circ$  and  $g(f) = 0$

```

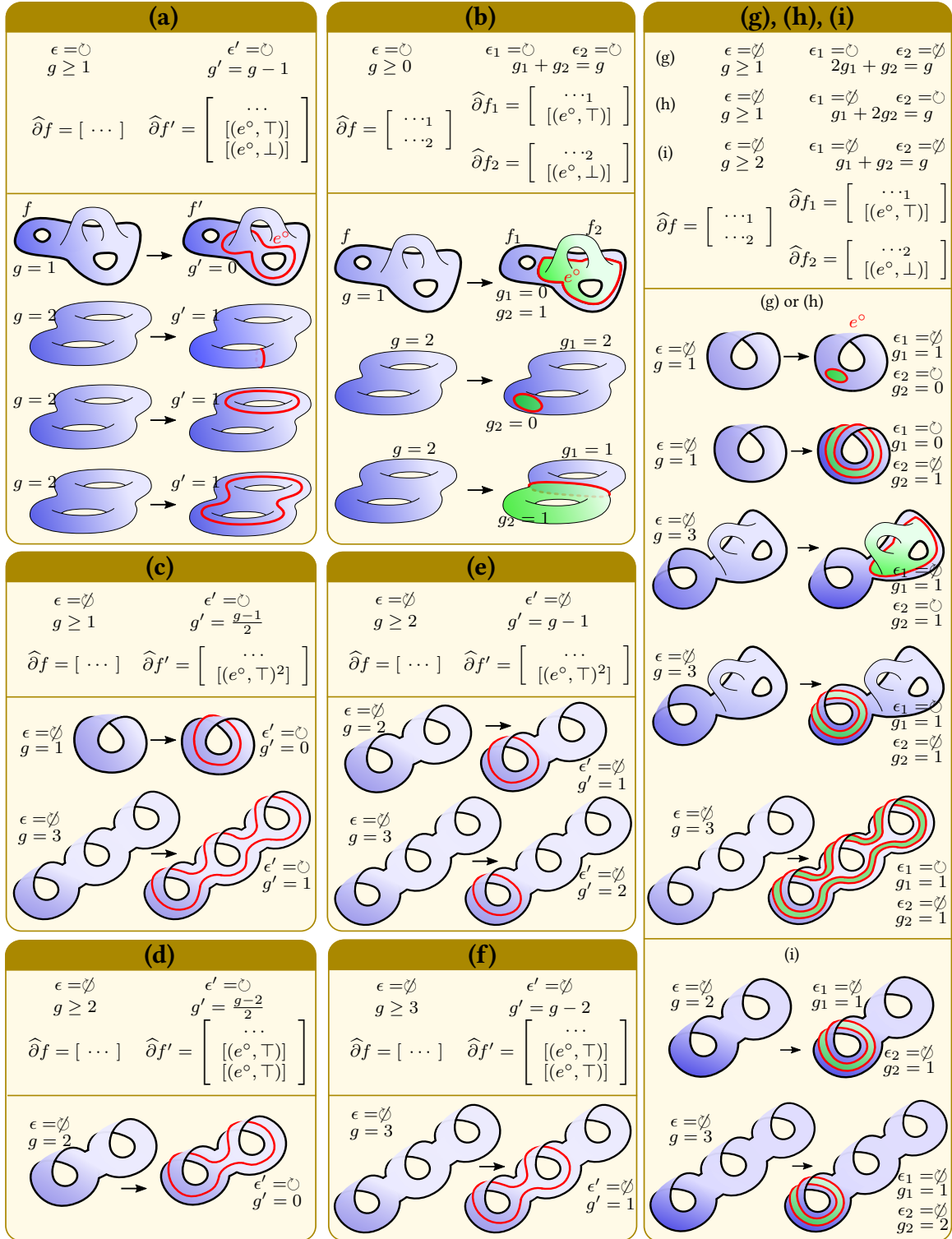
1  $e^\circ \leftarrow \text{CreateClosedEdge}()$                                  $\triangleright$  Create the cut edge  $e^\circ$  and the two faces  $f_1$  and  $f_2$ 
2  $f_1 \leftarrow \text{CreateFace}(\circ, 0)$ 
3  $f_2 \leftarrow \text{CreateFace}(\circ, 0)$ 
4 for all cycle  $\gamma_i$  of  $f$  do                                        $\triangleright$  Distribute the cycles of  $f$  among  $f_1$  and  $f_2$ 
5   if  $i \in I$  then
6      $\text{AddCycleToFace}(f_1, \gamma_i(f))$ 
7   else
8      $\text{AddCycleToFace}(f_2, \gamma_i(f))$ 
9  $\text{AddSimpleCycleToFace}(f_1, e^\circ, \top, 1)$                          $\triangleright$  Add cycle  $[(e^\circ, \top)]$  to  $f_1$  and cycle  $[(e^\circ, \perp)]$  to  $f_2$ 
10  $\text{AddSimpleCycleToFace}(f_2, e^\circ, \perp, 1)$ 
11  $\text{HardDelete}(f)$                                                  $\triangleright$  Delete  $f$ 

```

---

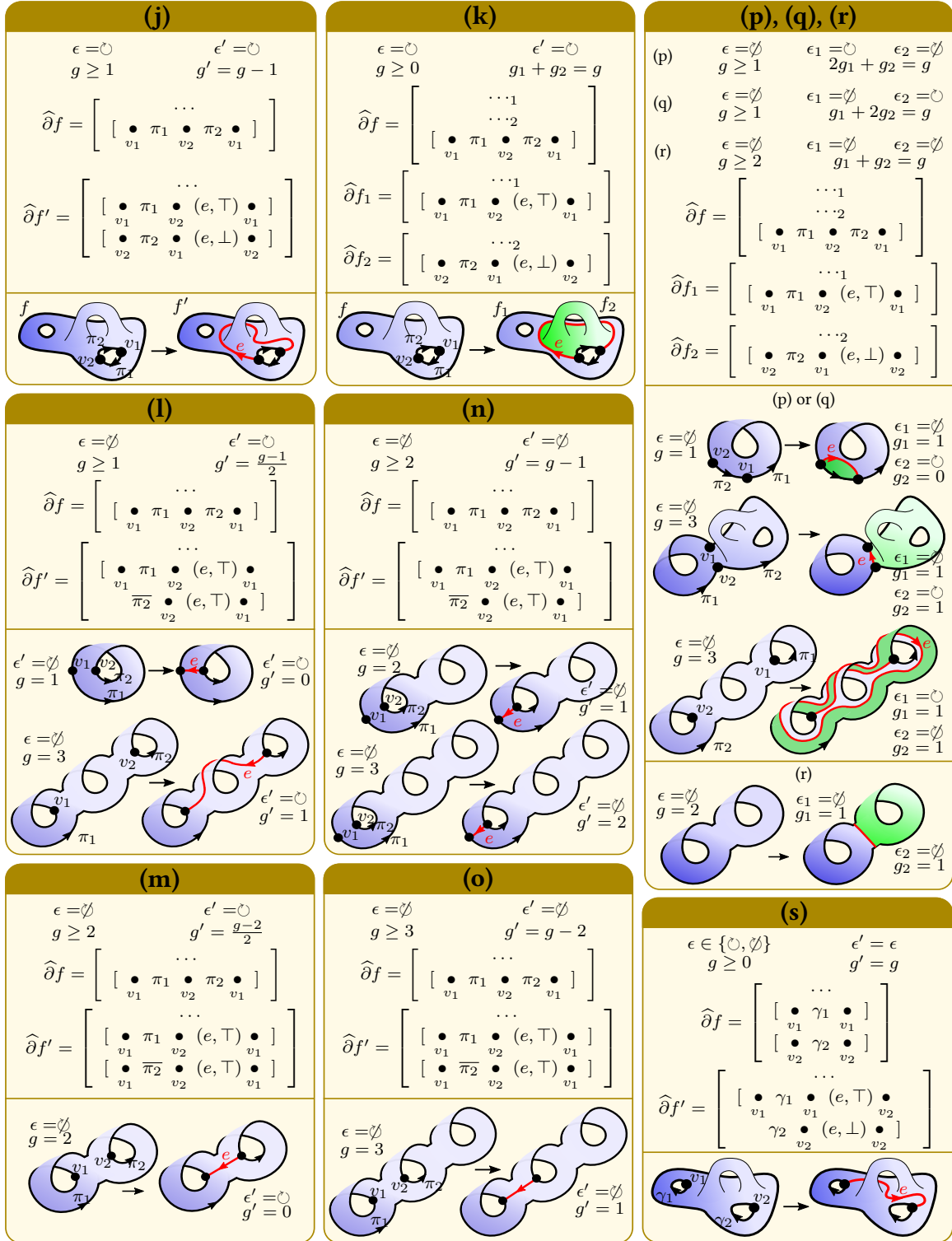


**Figure D.4:** Exhaustive classification of the 19 different ways a face can be cut at an edge. The branching “if”s represent known information about the abstract PCS complex that is about to be cut. The branching arrows represent information about  $e_{\text{cut}}$  that cannot be algorithmically determined, and hence that has to be given as input to the PCS topological operator (either as parameters or by calling different methods). We only show here the unknown information that leads to different orientabilities, genus formulas, number of faces, or number of cycles (e.g.: is  $e_{\text{cut}}$  closed or not? Does  $e_{\text{cut}}$  disconnect the face?). Additional parameters to give to the topological operators include: if  $e_{\text{cut}}$  disconnects  $f$ , which cycles to transfer to  $f_1$  or  $f_2$ ? What are the new genres  $g_1$  and  $g_2$ ? If  $e_{\text{cut}}$  is open, at which vertex-uses does it start and end? Should some cycles be flipped?



**Figure D.5:** The different ways to cut a face at a closed edge. The labelling letters refer to the classification provided in Figure D.4.





**Figure D.6:** The different ways to cut a face at an open edge. The labelling letters refer to the classification provided in Figure D.4.

The above topological operator is quite simple, but things get much more complicated when  $f$  is not a sphere, and especially when  $f$  is non-orientable. In order to cover all the different cases with a finite but exhaustive set of topological operators, it is necessary to *classify* all these different cases. We call this the **face-cut classification**, summarized in Figure D.4, illustrated in Figure D.5 and Figure D.6, and detailed in the following subsections.

### D.7.5 Cutting an Orientable Face at a Closed Edge

The first way to cut a face is via a closed edge included in the face. We recall that the geometric realization of a face is  $\text{int}(\mathbb{F}_{\epsilon,g,k})$ , and all the possibilities are illustrated in Figure A.3. As can be seen in Figure D.4, there exist many ways to choose a closed edge  $e^\circ$  inside the interior of a face  $f$ . In this section and the following, we classify all of them.

First, let us consider the case where  $f$  is orientable. Let  $e^\circ$  be a closed edge included in  $f$ . Thus, the pointset  $f \setminus e^\circ$  is either connected or it is not. If  $f \setminus e^\circ$  is connected, this completely determines the cut, i.e. any choice of  $e^\circ$  included in an orientable face  $f$  such that  $f \setminus e^\circ$  is connected leads to the same PCS complex up to homeomorphism, i.e. they have the same abstract PCS complex. This abstract PCS complex is obtained from the abstract PCS complex before the cut by decreasing the genus of  $f$  by one, and adding the two cycles  $[(e^\circ, \top)]$  and  $[(e^\circ, \perp)]$  to  $f$ . This is performed by the topological operator below:

---

#### CutOrientableFaceAtNonDisconnectingClosedEdge ( $f \in F$ )

---

**Require:**  $\epsilon(f) = \circ$  and  $g(f) \geq 1$

- 1  $g(f) \leftarrow g(f) - 1$
  - 2  $e^\circ \leftarrow \text{CreateClosedEdge}()$
  - 3  $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 1)$
  - 4  $\text{AddSimpleCycleToFace}(f, e^\circ, \perp, 1)$
- 

If, on the contrary,  $f \setminus e^\circ$  is not connected, then this means that it has two connected components  $f_1$  and  $f_2$ , both orientable and satisfying  $g(f) = g(f_1) + g(f_2)$ , where the cycles of  $f$  are distributed among  $f_1$  and  $f_2$ , the cycle  $[(e^\circ, \top)]$  is added to  $f_1$ , and the cycle  $[(e^\circ, \perp)]$  is added to  $f_2$ . However, the actual values of  $g(f_1)$  and  $g(f_2)$ , as well as which cycles are transferred to  $f_1$  and which cycles are transferred to  $f_2$  cannot be determined combinatorially without an underlying triangulation, and must therefore be an input of the following PCS topological operator. We note that the “CutSphereAtClosedEdge” operator that we have presented as a motivating example is redundant with this operator and therefore is not part of the classification.

---

**CutOrientableFaceAtDisconnectingClosedEdge** ( $f \in F, g_1 \in \mathbb{N}, g_2 \in \mathbb{N}, I \subseteq \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \circlearrowleft$  and  $g(f) = g_1 + g_2$ 

```

1  $e^\circ \leftarrow \text{CreateClosedEdge}()$   $\triangleright$  Create the cut edge  $e^\circ$  and the two faces  $f_1$  and  $f_2$ 
2  $f_1 \leftarrow \text{CreateFace}(\circlearrowleft, g_1)$ 
3  $f_2 \leftarrow \text{CreateFace}(\circlearrowleft, g_2)$ 

4 for all cycle  $\gamma_i$  of  $f$  do  $\triangleright$  Distribute the cycles of  $f$  among  $f_1$  and  $f_2$ 
5   if  $i \in I$  then
6      $\text{AddCycleToFace}(f_1, \gamma_i(f))$ 
7   else
8      $\text{AddCycleToFace}(f_2, \gamma_i(f))$ 

9  $\text{AddSimpleCycleToFace}(f_1, e^\circ, \top, 1)$   $\triangleright$  Add cycle  $[(e^\circ, \top)]$  to  $f_1$  and cycle  $[(e^\circ, \perp)]$  to  $f_2$ 
10  $\text{AddSimpleCycleToFace}(f_2, e^\circ, \perp, 1)$ 

11  $\text{HardDelete}(f)$   $\triangleright$  Delete  $f$ 

```

---

### D.7.6 Cutting a Non-Orientable Face at a Closed Edge

In this section, let us consider the case where a non-orientable  $f$  is cut at a closed edge  $e^\circ$ . The pointset  $f' = f \setminus e^\circ$  is either connected or it is not, and let us first consider the case where it is connected. On the contrary to the orientable case presented in the previous section, the information “ $f \setminus e^\circ$  is connected” does not fully determine the cut, unless  $g(f) = 1$ . More specifically, it is in fact always possible to choose  $e^\circ$  such that  $f'$  becomes orientable, but if  $g(f) \geq 2$  then it is also possible to choose  $e^\circ$  such that  $f'$  stays non-orientable. Reasoning with the Euler characteristic, it can be shown that if  $f'$  is orientable, then this information fully determines the cut, which is given by the following topological operator:

---

**CutNonOrientableFaceAtNonDisconnectingOrientingClosedEdge** ( $f \in F$ )
 

---

**Require:**  $\epsilon(f) = \emptyset$ 

```

1 if  $g(f)$  is odd then
2    $\epsilon(f) \leftarrow \circlearrowleft$ 
3    $g(f) \leftarrow \frac{g-1}{2}$ 
4    $e^\circ \leftarrow \text{CreateClosedEdge}()$ 
5    $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 2)$ 
6 else
7    $\epsilon(f) \leftarrow \circlearrowright$ 
8    $g(f) \leftarrow \frac{g-2}{2}$ 
9    $e^\circ \leftarrow \text{CreateClosedEdge}()$ 
10   $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 1)$ 
11   $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 1)$ 

```

---

However, if  $f'$  is non-orientable, then there still remains some ambiguity. Specifically, whenever  $g \geq 2$  it is possible to cut by preserving non-orientability and adding only one boundary (i.e., adding the cycle  $[(e^\circ, \top)^2]$ ), and whenever  $g \geq 3$  it is also possible to cut by preserving non-orientability and adding two boundaries (i.e., add the cycle  $[(e^\circ, \top)]$  twice). In the first scenario, it can be shown that the genus is decreased by one, and in the second scenario it can be shown that the genus is decreased by two. Therefore, this leads to the following two topological operators:

---

**CutNonOrientableFaceAtNonDisconnectingNonOrientingOddClosedEdge** ( $f \in F$ )
 

---

**Require:**  $\epsilon(f) = \emptyset$  and  $g \geq 2$ 

```

1  $g(f) \leftarrow g - 1$ 
2  $e^\circ \leftarrow \text{CreateClosedEdge}()$ 
3  $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 2)$ 

```

---



---

**CutNonOrientableFaceAtNonDisconnectingNonOrientingEvenClosedEdge** ( $f \in F$ )
 

---

**Require:**  $\epsilon(f) = \emptyset$  and  $g \geq 3$ 

```

1  $g(f) \leftarrow g - 2$ 
2  $e^\circ \leftarrow \text{CreateClosedEdge}()$ 
3  $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 1)$ 
4  $\text{AddSimpleCycleToFace}(f, e^\circ, \top, 1)$ 

```

---

Now that we have finished to consider all the cases where  $f \setminus e^\circ$  was connected, we are about to consider the cases where  $f \setminus e^\circ$  is not connected, and therefore has two connected components  $f_1$

and  $f_2$ . In this case, as with the orientable case, the cycles of  $f$  must be distributed among  $f_1$  and  $f_2$ , the cycle  $[(e^\circ, \top)]$  is added to  $f_1$ , and the cycle  $[(e^\circ, \perp)]$  is added to  $f_2$ . Since  $f$  is non-orientable, it can be shown that  $f_1$  and  $f_2$  cannot be both orientable, but all three other combinations are possible:  $f_1$  orientable and  $f_2$  non-orientable;  $f_1$  non-orientable and  $f_2$  orientable; or both  $f_1$  and  $f_2$  non-orientable (however, the latter is only possible if  $g(f) \geq 2$ ). Reasoning with the Euler characteristic, it can be shown that for each of these three cases, we have the genus relation, respectively:  $g(f) = 2g(f_1) + g(f_2)$ ;  $g(f) = g(f_1) + 2g(f_2)$ ; and  $g(f) = g(f_1) + g(f_2)$ . However, whether  $f_1$  and  $f_2$  are orientable and the actual values of  $g(f_1)$  and  $g(f_2)$  cannot be determined algorithmically without an underlying triangulation. Therefore, they are all input to the following topological operator that spans all the three cases:

---

**CutNonOrientableFaceAtDisconnectingClosedEdge** ( $f \in F$ ,  $\epsilon_1, \epsilon_2 \in \{\circlearrowleft, \emptyset\}$ ,  $g_1, g_2 \in \mathbb{N}$ ,  $I \subseteq \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \emptyset$

**Require:**  $\epsilon_1 = \emptyset$  or  $\epsilon_2 = \emptyset$

**Require:**  $(\epsilon_1 = \circlearrowleft \text{ and } \epsilon_2 = \emptyset) \Rightarrow (g_2 \geq 1 \text{ and } g(f) = 2g_1 + g_2)$

**Require:**  $(\epsilon_1 = \emptyset \text{ and } \epsilon_2 = \circlearrowleft) \Rightarrow (g_1 \geq 1 \text{ and } g(f) = g_1 + 2g_2)$

**Require:**  $(\epsilon_1 = \emptyset \text{ and } \epsilon_2 = \emptyset) \Rightarrow (g_1 \geq 1, g_2 \geq 1, \text{ and } g(f) = g_1 + g_2)$

```

1  $e^\circ \leftarrow \text{CreateClosedEdge}()$                                  $\triangleright$  Create the cut edge  $e^\circ$  and the two faces  $f_1$  and  $f_2$ 
2  $f_1 \leftarrow \text{CreateFace}(\epsilon_1, g_1)$ 
3  $f_2 \leftarrow \text{CreateFace}(\epsilon_2, g_2)$ 

4 for all cycle  $\gamma_i$  of  $f$  do                                        $\triangleright$  Distribute the cycles of  $f$  among  $f_1$  and  $f_2$ 
5   if  $i \in I$  then
6      $\text{AddCycleToFace}(f_1, \gamma_i(f))$ 
7   else
8      $\text{AddCycleToFace}(f_2, \gamma_i(f))$ 

9  $\text{AddSimpleCycleToFace}(f_1, e^\circ, \top, 1)$                          $\triangleright$  Add cycle  $[(e^\circ, \top)]$  to  $f_1$  and  $[(e^\circ, \perp)]$  to  $f_2$ 
10  $\text{AddSimpleCycleToFace}(f_2, e^\circ, \perp, 1)$ 

11  $\text{HardDelete}(f)$                                                  $\triangleright$  Delete  $f$ 
```

---

### D.7.7 Cutting a Face at an Open Edge Starting and Ending at the Same Hole

As illustrated in Figure D.4, this case is very similar to cutting at a closed edge, and follows the same classification. The difference is that instead of adding the two cycles  $[(e^\circ, \top)]$  and  $[(e^\circ, \perp)]$  (resp., twice the cycle  $[(e^\circ, \top)]$ , or the single cycle  $[(e^\circ, \top)^2]$ ), we remove one cycle  $\gamma_i$  (the cycle corresponding to the starting/ending hole), split it into two paths  $\pi_1$  and  $\pi_2$ , then add the two cycles  $[\pi_1, (e, \top)]$  and  $[\pi_2, (e, \perp)]$  (resp., the two cycles  $[\pi_1, (e, \top)]$  and  $[\pi_2, (e, \top)]$ , or the single cycle  $[\pi_1, (e, \top), \pi_2, (e, \top)]$ ).

As illustrated in Figure D.7, one issue is that the same vertex may be used several times by the same cycle, and hence knowing  $v_{\text{start}}(e)$  and  $v_{\text{end}}(e)$  is in general not enough information to combinatorially determine at which two indices the cycle  $\gamma_i$  must be split. Therefore, these indices must be explicitly provided as input of the topological operator, in the form of two integers  $j_{\text{start}}$  and  $j_{\text{end}}$ , in addition to the integer  $i$  specifying  $\gamma_i$ . Finally, we note that the same vertex-use  $\textcircled{v}_{f,i,j}$  can be specified as both start and end vertex-use of the cut (cf Figure D.7, bottom-left), in which case either  $\pi_1$  or  $\pi_2$  is empty, while the other is equal to the whole cycle  $\gamma_i$ . To disambiguate which is which, the caller of the operator must indicate either  $j_{\text{end}} = j_{\text{start}}$  (to get  $\pi_1 = \gamma_i$  and  $\pi_2 = []$ ), or  $j_{\text{end}} = j_{\text{start}} + N(\gamma_i)$  (to get  $\pi_1 = []$  and  $\pi_2 = \gamma_i$ ). In the special case where  $\gamma_i$  is a Steiner cycle, then  $j_{\text{start}}$  and  $j_{\text{end}}$  are not necessary and are simply ignored. We note that  $\gamma_i$  cannot be a simple cycle, since a simple cycle do not have any vertex-use.

Combining the above observations with the classification already given for cutting at a closed edge, we obtain six topological operators that are reported in this section. But first, we define the helper method `SplitCycle()` splitting a cycle  $\gamma$  into two paths  $\pi_1$  and  $\pi_2$ , given two indices  $j_{\text{start}}$  and  $j_{\text{end}}$  indicating where to split  $\gamma$ :

---

**SplitCycle** ( $\gamma \in \Gamma, j_{\text{start}} \in \mathbb{N}, j_{\text{end}} \in \mathbb{N}$ )
 

---

**Require:**  $\left\{ \begin{array}{l} \gamma \text{ is a Steiner cycle, or} \\ \left\{ \begin{array}{l} \gamma \text{ is a non-simple cycle, and} \\ j_{\text{start}} \in [1..N(\gamma)], \text{ and} \\ \left\{ \begin{array}{l} j_{\text{end}} = j_{\text{start}}, \text{ or} \\ j_{\text{end}} = j_{\text{start}} + N(\gamma), \text{ or} \\ j_{\text{end}} \in [1..N(\gamma)] \text{ and } j_{\text{start}} \neq j_{\text{end}} \end{array} \right. \end{array} \right. \end{array} \right.$

```

1  if  $\gamma$  is a Steiner cycle then  $\triangleright \gamma = [v]$ 
2       $v_{\text{start}} \leftarrow v(\gamma)$ 
3       $v_{\text{end}} \leftarrow v(\gamma)$ 
4       $\pi_1 \leftarrow [v(\gamma)]$ 
5       $\pi_2 \leftarrow [v(\gamma)]$ 

6  else
7      if  $j_{\text{end}} = j_{\text{start}}$  then  $\triangleright \gamma = [h_1 \cdots h_{j_{\text{start}}} \bullet_{v_{j_{\text{start}}}} h_{j_{\text{start}}+1} \cdots h_N]$ 
8           $v_{\text{start}} \leftarrow v_{j_{\text{start}}}(\gamma)$ 
9           $v_{\text{end}} \leftarrow v_{j_{\text{start}}}(\gamma)$ 
10          $\pi_1 \leftarrow [h_{j_{\text{start}}+1} \cdots h_N h_1 \cdots h_{j_{\text{start}}}]$ 
11          $\pi_2 \leftarrow [v_{j_{\text{start}}}(\gamma)]$ 

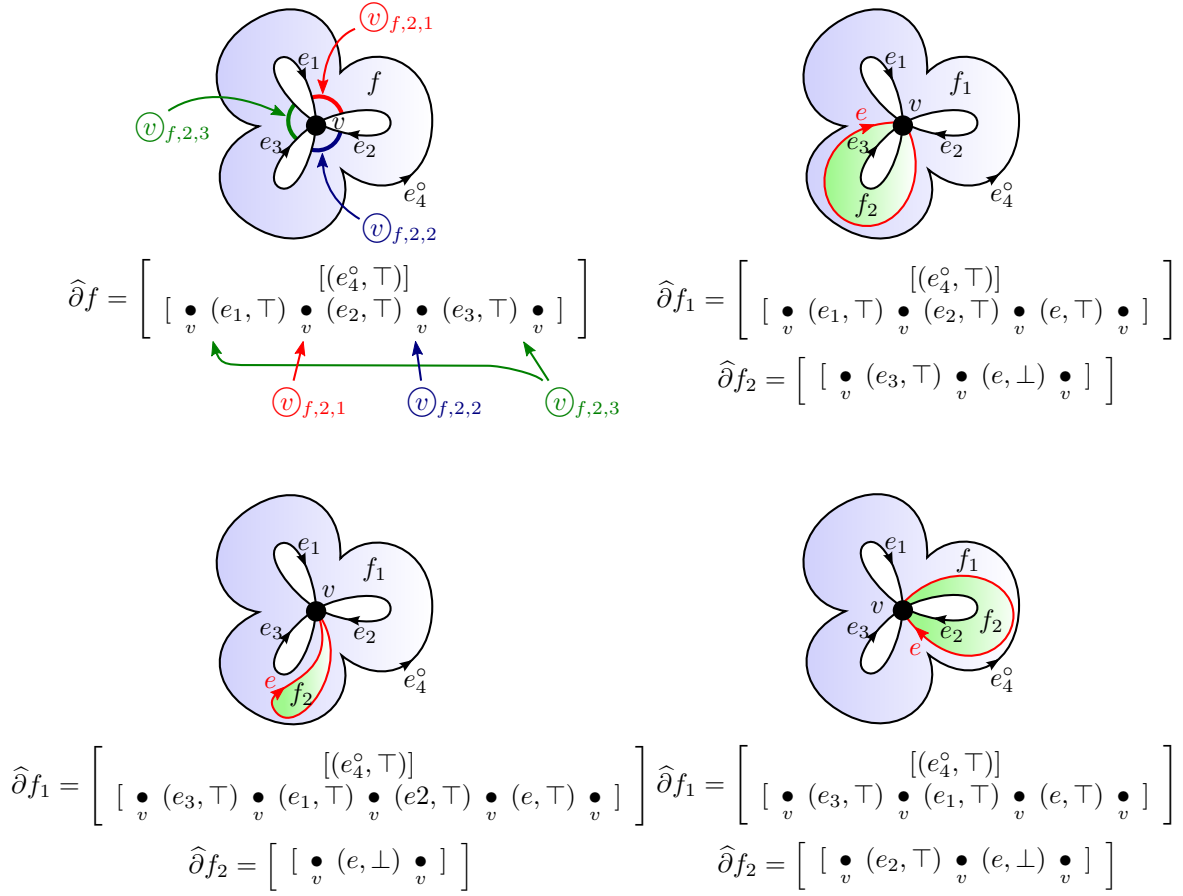
12        else if  $j_{\text{end}} = j_{\text{start}} + N(\gamma)$  then
13             $v_{\text{start}} \leftarrow v_{j_{\text{start}}}(\gamma)$ 
14             $v_{\text{end}} \leftarrow v_{j_{\text{start}}}(\gamma)$ 
15             $\pi_1 \leftarrow [v_{j_{\text{start}}}(\gamma)]$ 
16             $\pi_2 \leftarrow [h_{j_{\text{start}}+1} \cdots h_N h_1 \cdots h_{j_{\text{start}}}]$ 

17        else if  $j_{\text{start}} < j_{\text{end}}$  then  $\triangleright \gamma = [h_1 \cdots h_{j_{\text{start}}} \bullet_{v_{j_{\text{start}}}} h_{j_{\text{start}}+1} \cdots h_{j_{\text{end}}} \bullet_{v_{j_{\text{end}}}} h_{j_{\text{end}}+1} \cdots h_N]$ 
18             $v_{\text{start}} \leftarrow v_{j_{\text{start}}}(\gamma)$ 
19             $v_{\text{end}} \leftarrow v_{j_{\text{end}}}(\gamma)$ 
20             $\pi_1 \leftarrow [h_{j_{\text{end}}+1} \cdots h_N h_1 \cdots h_{j_{\text{start}}}]$ 
21             $\pi_2 \leftarrow [h_{j_{\text{start}}+1} \cdots h_{j_{\text{end}}}]$ 

22        else  $\triangleright \gamma = [h_1 \cdots h_{j_{\text{end}}} \bullet_{v_{j_{\text{end}}}} h_{j_{\text{end}}+1} \cdots h_{j_{\text{start}}} \bullet_{v_{j_{\text{start}}}} h_{j_{\text{start}}+1} \cdots h_N]$ 
23             $v_{\text{start}} \leftarrow v_{j_{\text{start}}}(\gamma)$ 
24             $v_{\text{end}} \leftarrow v_{j_{\text{end}}}(\gamma)$ 
25             $\pi_1 \leftarrow [h_{j_{\text{end}}+1} \cdots h_{j_{\text{start}}}]$ 
26             $\pi_2 \leftarrow [h_{j_{\text{start}}+1} \cdots h_N h_1 \cdots h_{j_{\text{end}}}]$ 

27 return  $(v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2)$ 
    
```

---



**Figure D.7:** Three different cuts that start and end at the same vertex, but with different vertex-uses.

We now report all the six different methods that can be used to cut a face at an open edge starting and ending at the same hole. In addition to specific parameters, all these methods have in common the parameters  $f \in F$ ,  $i \in \mathbb{N}$ ,  $j_{\text{start}} \in \mathbb{N}$ ,  $j_{\text{end}} \in \mathbb{N}$  with the following requirement that we report here for conciseness:

$$\left\{ \begin{array}{l} i \in [1..k(f)], \text{ and} \\ \left\{ \begin{array}{l} \gamma_i(f) \text{ is a Steiner cycle, or} \\ \left\{ \begin{array}{l} \gamma_i(f) \text{ is a non-simple cycle, and} \\ j_{\text{start}} \in [1..N(\gamma_i(f))], \text{ and} \\ \left\{ \begin{array}{l} j_{\text{end}} = j_{\text{start}}, \text{ or} \\ j_{\text{end}} = j_{\text{start}} + N(\gamma_i(f)), \text{ or} \\ j_{\text{end}} \in [1..N(\gamma_i(f))] \text{ and } j_{\text{start}} \neq j_{\text{end}} \end{array} \right\} \end{array} \right\} \end{array} \right. \end{array} \right. \quad (\text{D.18})$$



---

**CutOrientableFaceAtNonDisconnectingOpenEdge** ( $f \in F, i \in \mathbb{N}, j_{\text{start}} \in \mathbb{N}, j_{\text{end}} \in \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \circlearrowleft$  and  $g(f) \geq 1$

**Require:** Equation D.18

- 1  $(v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2) \leftarrow \text{SplitCycle}(\gamma_i(f), j_{\text{start}}, j_{\text{end}})$
  - 2  $g(f) \leftarrow g(f) - 1$
  - 3  $e \leftarrow \text{CreateOpenEdge}(v_{\text{start}}, v_{\text{end}})$
  - 4  $\text{AddNonSimpleCycleToFace}(f, [\pi_1, (e, \top)])$
  - 5  $\text{AddNonSimpleCycleToFace}(f, [\pi_2, (e, \perp)])$
  - 6  $\text{RemoveCycleFromFace}(f, i)$
- 

---

**CutOrientableFaceAtDisconnectingOpenEdge** ( $f \in F, i \in \mathbb{N}, j_{\text{start}}, j_{\text{end}} \in \mathbb{N}, g_1, g_2 \in \mathbb{N}, I \subseteq \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \circlearrowleft$  and  $g(f) = g_1 + g_2$

**Require:** Equation D.18

- 1  $(v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2) \leftarrow \text{SplitCycle}(\gamma_i(f), j_{\text{start}}, j_{\text{end}})$
  - 2  $e \leftarrow \text{CreateOpenEdge}(v_{\text{start}}, v_{\text{end}})$  ▷ Create the cut edge  $e$  and the two faces  $f_1$  and  $f_2$
  - 3  $f_1 \leftarrow \text{CreateFace}(\circlearrowleft, g_1)$
  - 4  $f_2 \leftarrow \text{CreateFace}(\circlearrowleft, g_2)$
  - 5 **for all** cycle  $\gamma_{i'}$  of  $f, i' \neq i$  **do** ▷ Distribute the cycles of  $f$ , except  $\gamma_i$ , among  $f_1$  and  $f_2$
  - 6     **if**  $i' \in I$  **then**
  - 7          $\text{AddCycleToFace}(f_1, \gamma_{i'}(f))$
  - 8     **else**
  - 9          $\text{AddCycleToFace}(f_2, \gamma_{i'}(f))$
  - 10  $\text{AddNonSimpleCycleToFace}(f_1, [\pi_1, (e, \top)])$  ▷ Add cycle  $[\pi_1, (e, \top)]$  to  $f_1$  and  $[\pi_2, (e, \perp)]$  to  $f_2$
  - 11  $\text{AddNonSimpleCycleToFace}(f_2, [\pi_2, (e, \perp)])$
  - 12  $\text{HardDelete}(f)$  ▷ Delete  $f$
-

---

**CutNonOrientableFaceAtNonDisconnectingOrientingOpenEdge** ( $f \in F, i \in \mathbb{N}, j_{\text{start}} \in \mathbb{N}, j_{\text{end}} \in \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \emptyset$

**Require:** Equation D.18

```

1  ( $v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2$ )  $\leftarrow$  SplitCycle( $\gamma_i(f), j_{\text{start}}, j_{\text{end}}$ )
2  if  $g(f)$  is odd then
3       $\epsilon(f) \leftarrow \circlearrowleft$ 
4       $g(f) \leftarrow \frac{g-1}{2}$ 
5       $e \leftarrow$  CreateOpenEdge( $v_{\text{start}}, v_{\text{end}}$ )
6      AddNonSimpleCycleToFace( $f, [\pi_1, (e, \top), \overline{\pi_2}, (e, \top)]$ )
7      RemoveCycleFromFace( $f, i$ )
8  else
9       $\epsilon(f) \leftarrow \circlearrowright$ 
10      $g(f) \leftarrow \frac{g-2}{2}$ 
11      $e \leftarrow$  CreateOpenEdge( $v_{\text{start}}, v_{\text{end}}$ )
12     AddNonSimpleCycleToFace( $f, [\pi_1, (e, \top)]$ )
13     AddNonSimpleCycleToFace( $f, [\overline{\pi_2}, (e, \top)]$ )
14     RemoveCycleFromFace( $f, i$ )

```

---



---

**CutNonOrientableFaceAtNonDisconnectingNonOrientingOddOpenEdge** ( $f \in F, i, j_{\text{start}}, j_{\text{end}} \in \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \emptyset$  and  $g \geq 2$

**Require:** Equation D.18

```

1  ( $v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2$ )  $\leftarrow$  SplitCycle( $\gamma_i(f), j_{\text{start}}, j_{\text{end}}$ )
2   $g(f) \leftarrow g - 1$ 
3   $e \leftarrow$  CreateOpenEdge( $v_{\text{start}}, v_{\text{end}}$ )
4  AddNonSimpleCycleToFace( $f, [\pi_1, (e, \top), \overline{\pi_2}, (e, \top)]$ )
5  RemoveCycleFromFace( $f, i$ )

```

---

---

**CutNonOrientableFaceAtNonDisconnectingNonOrientingEvenOpenEdge** ( $f \in F, i, j_{\text{start}}, j_{\text{end}} \in \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \emptyset$  and  $g \geq 3$

**Require:** Equation D.18

- 1  $(v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2) \leftarrow \text{SplitCycle}(\gamma_i(f), j_{\text{start}}, j_{\text{end}})$
  - 2  $g(f) \leftarrow g - 2$
  - 3  $e \leftarrow \text{CreateOpenEdge}(v_{\text{start}}, v_{\text{end}})$
  - 4  $\text{AddNonSimpleCycleToFace}(f, [\pi_1, (e, \top)])$
  - 5  $\text{AddNonSimpleCycleToFace}(f, [\pi_2, (e, \top)])$
  - 6  $\text{RemoveCycleFromFace}(f, i)$
- 

---

**CutNonOrientableFaceAtDisconnectingOpenEdge** ( $f \in F, i, j_{\text{start}}, j_{\text{end}}, \epsilon_1, \epsilon_2, g_1, g_2, I \subseteq \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \emptyset$

**Require:**  $\epsilon_1 = \emptyset$  or  $\epsilon_2 = \emptyset$

**Require:**  $(\epsilon_1 = \circlearrowleft \text{ and } \epsilon_2 = \emptyset) \Rightarrow (g_2 \geq 1 \text{ and } g(f) = 2g_1 + g_2)$

**Require:**  $(\epsilon_1 = \emptyset \text{ and } \epsilon_2 = \circlearrowleft) \Rightarrow (g_1 \geq 1 \text{ and } g(f) = g_1 + 2g_2)$

**Require:**  $(\epsilon_1 = \emptyset \text{ and } \epsilon_2 = \emptyset) \Rightarrow (g_1 \geq 1, g_2 \geq 1, \text{ and } g(f) = g_1 + g_2)$

**Require:** Equation D.18

- 1  $(v_{\text{start}}, v_{\text{end}}, \pi_1, \pi_2) \leftarrow \text{SplitCycle}(\gamma_i(f), j_{\text{start}}, j_{\text{end}})$
  - 2  $e \leftarrow \text{CreateOpenEdge}(v_{\text{start}}, v_{\text{end}})$   $\triangleright$  Create the cut edge  $e$  and the two faces  $f_1$  and  $f_2$
  - 3  $f_1 \leftarrow \text{CreateFace}(\epsilon_1, g_1)$
  - 4  $f_2 \leftarrow \text{CreateFace}(\epsilon_2, g_2)$
  - 5 **for all** cycle  $\gamma_{i'}$  of  $f, i' \neq i$  **do**  $\triangleright$  Distribute the cycles of  $f$ , except  $\gamma_i$ , among  $f_1$  and  $f_2$
  - 6     **if**  $i \in I$  **then**
  - 7          $\text{AddCycleToFace}(f_1, \gamma_{i'}(f))$
  - 8     **else**
  - 9          $\text{AddCycleToFace}(f_2, \gamma_{i'}(f))$
  - 10  $\text{AddNonSimpleCycleToFace}(f_1, [\pi_1, (e, \top)])$   $\triangleright$  Add cycle  $[\pi_1, (e, \top)]$  to  $f_1$  and  $[\pi_2, (e, \perp)]$  to  $f_2$
  - 11  $\text{AddNonSimpleCycleToFace}(f_2, [\pi_2, (e, \perp)])$
  - 12  $\text{HardDelete}(f)$   $\triangleright$  Delete  $f$
-

### D.7.8 Cutting a Face at an Open Edge Starting and Ending at Different Holes

Finally, the last case to consider is when the cut edge  $e$  is an open edge that starts and ends at different holes, represented by different cycles  $\gamma_{i_1}$  and  $\gamma_{i_2}$  of  $f$ . Fortunately, this case is actually very easy to handle, as it can be shown that it never disconnects  $f$ , and preserves its orientability and genus. Therefore, its only action is to merge the two cycles  $\gamma_{i_1}$  and  $\gamma_{i_2}$  into a single cycle, by joining them with  $e$ , as per the algorithm below:

---

**RotatedCycle** ( $\gamma \in \Gamma, j \in \mathbb{N}$ )

---

**Require:**  $\gamma$  is a Steiner cycle, or a non-simple cycle with  $j \in [1..N(\gamma)]$

```

1 if  $\gamma$  is a Steiner cycle then  $\triangleright \gamma = [v]$ 
2    $v' \leftarrow v(\gamma)$ 
3    $\gamma' \leftarrow \gamma$ 
4 else  $\triangleright \gamma = [h_1 \cdots h_j \bullet_{v_j} h_{j+1} \cdots h_N]$ 
5    $v' \leftarrow v_j(\gamma)$ 
6    $\gamma' \leftarrow [h_{j+1} \cdots h_N h_1 \cdots h_j]$ 
7 return  $(v', \gamma')$ 
```

---



---

**CutFaceAtOpenEdge** ( $f \in F, i_1 \in \mathbb{N}, i_2 \in \mathbb{N}, j_1 \in \mathbb{N}, j_2 \in \mathbb{N}$ )

---

**Require:**  $(i_1, i_2) \in [1..k(f)]^2$

**Require:**  $\gamma_{i_1}(f)$  is a Steiner cycle, or a non-simple cycle with  $j_1 \in [1..N(\gamma_{i_1}(f))]$

**Require:**  $\gamma_{i_2}(f)$  is a Steiner cycle, or a non-simple cycle with  $j_2 \in [1..N(\gamma_{i_2}(f))]$

```

1  $(v_1, \gamma_1) \leftarrow \text{RotatedCycle}(i_1, j_1)$ 
2  $(v_2, \gamma_2) \leftarrow \text{RotatedCycle}(i_2, j_2)$ 
3  $e \leftarrow \text{CreateOpenEdge}(v_1, v_2)$ 
4  $\text{AddNonSimpleCycleToFace}(f, [\gamma_1, (e, \top), \gamma_2, (e, \perp)])$ 
5  $\text{RemoveCyclesFromFace}(f, \{i_1, i_2\})$ 
```

---

### D.7.9 Flipping Cycles of Non-Orientable Faces

When cutting a non-orientable face, there is one additional subtlety that has been omitted for clarity. It starts with the observation that directions of cycles matter for orientable faces, but do not matter for non-orientable faces. This means that it is always possible to flip the direction of any cycle of any non-orientable face, and this will result in a homeomorphic abstract PCS complex (i.e., their geometric realization is homeomorphic). Therefore, the topological operator below is

essentially a null operation, and can be performed at any time without changing what PCS complex it represents:

---

**FlipCycle** ( $f \in F, i \in \mathbb{N}$ )

---

**Require:**  $\epsilon(f) = \emptyset$  and  $i \in [1..k(f)]$

$\gamma_i(f) \leftarrow \overline{\gamma_i(f)}$

---

However, it *cannot* be performed for orientable faces since it could lead to non-homeomorphic PCS complexes. For instance, consider two abstract PCS complexes, each of them being made of one closed cycle  $e^\circ$  and one face  $f$ . In the first abstract PCS complex, the face is

$$f = (\circlearrowleft, 0, [[(e^\circ, \top)], [(e^\circ, \top)]]), \quad (\text{D.19})$$

while in the second abstract PCS complex, the face is

$$f = (\circlearrowleft, 0, [[(e^\circ, \top)], [(e^\circ, \perp)]]). \quad (\text{D.20})$$

In both cases, it is possible to uncut at  $e^\circ$ , but the resulting PCS complexes are not homeomorphic: one leads to a Klein bottle while the other leads to a torus, as formalized below:

$$f = (\circlearrowleft, 0, \left[ \begin{array}{c} [(e^\circ, \top)] \\ [(e^\circ, \top)] \end{array} \right]) \xrightarrow{\text{UnCutAt}(e^\circ)} f' = (\emptyset, 2, []) \quad (\text{D.21})$$

$$f = (\circlearrowleft, 0, \left[ \begin{array}{c} [(e^\circ, \top)] \\ [(e^\circ, \perp)] \end{array} \right]) \xrightarrow{\text{UnCutAt}(e^\circ)} f' = (\circlearrowleft, 1, []) \quad (\text{D.22})$$

This has to be compared with the non-orientable case, where indeed direction does not matter, as illustrated by the examples below:

$$f = (\emptyset, 1, \left[ \begin{array}{c} [(e^\circ, \top)] \\ [(e^\circ, \top)] \end{array} \right]) \xrightarrow{\text{UnCutAt}(e^\circ)} f' = (\emptyset, 3, []) \quad (\text{D.23})$$

$$f = (\emptyset, 1, \left[ \begin{array}{c} [(e^\circ, \top)] \\ [(e^\circ, \perp)] \end{array} \right]) \xrightarrow{\text{UnCutAt}(e^\circ)} f' = (\emptyset, 3, []) \quad (\text{D.24})$$

Therefore, when a non-orientable face  $f$  generates an orientable face  $f'$ ,  $f_1$  or  $f_2$  under the action of a cut, in addition to give as input which cycles to transfer to the orientable face, it is also necessary to give as input what directions to give to these cycles, directions that could be computed

if  $e_{\text{cut}}$  was given as edges of an underlying triangulation. Also, in the case where a non-orientable face is cut at an open edge starting and ending at the same hole, there are in fact two possible non-homeomorphic outcomes of the cut: either merging  $\gamma_{i_1}$  and  $\gamma_{i_2}$  into  $[\gamma_{i_1}, (e, \top), \gamma_{i_2}, (e, \perp)]$ , or merging them into  $[\gamma_{i_1}, (e, \top), \overline{\gamma_{i_2}}, (e, \perp)]$ .

Instead of making the input of the topological operators more complicated than it already is, this can simply be achieved by calling `FlipCycle()` as many times as necessary before calling one of the `CutNonOrientableFace[...]` methods (or `CutFaceAtOpenEdge()` if  $f$  is non-orientable), and this sequence can be seen as the whole cut operator.

## D.8 Uncut Cells

We now present the uncut topological operator, which is the reverse of the cut operator. Since all the important ideas have already been covered in the previous section, we provide here the algorithm but do not comment it extensively. Nevertheless, here are two important observations:

- Given a cell  $c$ , it is not always possible to “uncut at  $c$ ”. More specifically, it is possible to uncut at  $c$  if and only if  $c$  may have been created as the cut cell of a cut topological operator.
- On the contrary to the cut operator, the uncut operator is *not* ambiguous. This means that indicating which cell to uncut at is the only necessary input. One way to interpret this fundamental difference between cut and uncut is that before the cut, we *do not know yet*  $e_{\text{cut}}$ , and hence we have to fully specify combinatorially *how* it cuts a given face. However, for the reverse operation,  $e_{\text{cut}}$  does exist, and hence we know exactly how it is used, e.g. as a frontier between two known faces. Merging back these two faces into one face is a non-ambiguous process, but during which information about  $e_{\text{cut}}$  is lost, reason why the reverse process is ambiguous.

---

### CanUnCutAt ( $c \in C$ )

---

```

1 if  $c \in V$  then
2   return CanUnCutAtVertex( $c$ )
3 else if  $c \in E_{\circ}$  then
4   return CanUnCutAtClosedEdge( $c$ )
5 else if  $c \in E_{\perp}$  then
6   return CanUnCutAtOpenEdge( $c$ )
7 else if  $c \in F$  then
8   return false

```

---

**CanUnCutAtVertex** ( $v \in V$ )

---

```

1 if  $\text{star}(v) = \emptyset$  then
2   return false
3 else
4    $N_{\text{incident-edges}} \leftarrow 0$ 
5    $N_{\text{end-vertex-use}} \leftarrow 0$ 
6   for all edge  $e \in \text{star}(v)$  do
7      $N_{\text{incident-edges}} \leftarrow N_{\text{incident-edges}} + 1$ 
8     if  $v_{\text{start}}(e) = v$  then ▷ End-vertex-use  $\textcircled{v}_{e,\text{start}}$ 
9        $N_{\text{end-vertex-use}} \leftarrow N_{\text{end-vertex-use}} + 1$ 
10    if  $v_{\text{end}}(e) = v$  then ▷ End-vertex-use  $\textcircled{v}_{e,\text{end}}$ 
11       $N_{\text{end-vertex-use}} \leftarrow N_{\text{end-vertex-use}} + 1$ 

  ▷ Count the number of Steiner-vertex-uses.

12   $N_{\text{Steiner-vertex-use}} \leftarrow 0$ 
13  for all face  $f \in \text{star}(v)$  do
14    for all Steiner cycle  $\gamma_i^\bullet = [v_i] \in \hat{\partial}f$  do
15      if  $v_i = v$  then ▷ Steiner-vertex-use  $\textcircled{v}_{f,i}$ 
16         $N_{\text{Steiner-vertex-use}} \leftarrow N_{\text{Steiner-vertex-use}} + 1$ 

  ▷ Check if  $v$  could have been created via CutFaceAtVertex().

17  if  $N_{\text{Steiner-vertex-use}} = 1$  and  $N_{\text{end-vertex-use}} = 0$  then
18    return true

  ▷ Check if  $v$  could have been created via CutClosedEdge(). This requires  $v$  to have a single
  incident edge  $e = (v, v)$ , and cycles using  $e$  must be of the form  $[(e, \beta)^N]$ .

19  if  $N_{\text{Steiner-vertex-use}} = 0$  and  $N_{\text{end-vertex-use}} = 2$  and  $N_{\text{incident-edges}} = 1$  then
20     $e \leftarrow$  only edge in  $\text{star}(v)$ 
21    for all face  $f \in \text{star}(v)$  do
22      for all non-simple cycle  $\gamma_i \in \hat{\partial}f$  do
23        if  $\gamma_i$  uses  $e$  and  $\nexists(\beta, N)$  s.t.  $\gamma_i = [(e, \beta)^N]$  then
24          return false
25    return true

```

---

---

▷ Check if  $v$  could have been created via CutOpenEdge(). This requires  $v$  to have exactly two incident edges  $e_1$  and  $e_2$  each using  $v$  once, and cycles using  $v$  must not do any “switch-back” at  $v$ .

```

26   if  $N_{\text{Steiner-vertex-use}} = 0$  and  $N_{\text{end-vertex-use}} = 2$  and  $N_{\text{incident-edges}} = 2$  then
27        $(e_1, e_2) \leftarrow$  the two edges in  $\text{star}(v)$ 
28       for all face  $f \in \text{star}(v)$  do
29           for all non-simple cycle  $\gamma_i \in \hat{\partial}f$  do
30               for all  $j \in [1..N(\gamma_i)]$  do
31                   if  $v_j = v$  and  $e_j(\gamma_i) = e_{j+1}(\gamma_i)$  then
32                       return false
33       return true

▷ All other cases mean that  $v$  could not have been created via a cut
34   return false

```

---



---

**CanUnCutAtClosedEdge** ( $e^\circ \in E_o$ )

---

```

1  if  $\text{star}(e^\circ) = \emptyset$  then
2      return false
3  else
4       $N_{\text{incident-faces}} \leftarrow 0$ 
5       $N_{\text{cycles-using-e}} \leftarrow 0$ 
6       $N_{\text{closed-edge-use}} \leftarrow 0$ 
7      for all face  $f \in \text{star}(e^\circ)$  do
8           $N_{\text{incident-faces}} \leftarrow N_{\text{incident-faces}} + 1$ 
9          for all simple cycle  $\gamma_i^\circ \in \hat{\partial}f$  do
10             if  $e^\circ(\gamma_i^\circ) = e^\circ$  then                                     ▷ Closed-edge-uses  $\mathcal{C}_{f,i,1}^\circ$  to  $\mathcal{C}_{f,i,N(\gamma_i^\circ)}^\circ$ 
11                  $N_{\text{cycles-using-e}} \leftarrow N_{\text{cycles-using-e}} + 1$ 
12                  $N_{\text{closed-edge-use}} \leftarrow N_{\text{closed-edge-use}} + N(\gamma_i^\circ)$ 
13             if  $N_{\text{closed-edge-use}} = 2$  then
14                 return true
15             else
16                 return false

```

---



---

**CanUnCutAtOpenEdge** ( $e \in E_{\mid}$ )

---

```

1 if  $\text{star}(e) = \emptyset$  then
2   return false
3 else
4    $N_{\text{incident-faces}} \leftarrow 0$ 
5    $N_{\text{cycles-using-e}} \leftarrow 0$ 
6    $N_{\text{open-edge-use}} \leftarrow 0$ 
7   for all face  $f \in \text{star}(e)$  do
8      $N_{\text{incident-faces}} \leftarrow N_{\text{incident-faces}} + 1$ 
9     for all non-simple cycle  $\gamma_i \in \widehat{\partial}f$  do
10      CycleAlreadyCounted  $\leftarrow$  false
11      for all  $j \in [1..N(\gamma_i)]$  do
12        if  $e_j(\gamma) = e$  then  $\triangleright$  Open-edge-use  $\odot_{f,i,j}$ 
13           $N_{\text{open-edge-use}} \leftarrow N_{\text{open-edge-use}} + 1$ 
14          if not CycleAlreadyCounted then
15             $N_{\text{cycles-using-e}} \leftarrow N_{\text{cycles-using-e}} + 1$ 
16            CycleAlreadyCounted  $\leftarrow$  true
17   if  $N_{\text{open-edge-use}} = 2$  then
18     return true
19   else
20     return false

```

---



---

**UnCutAt** ( $c \in C$ )

---

```

1 if  $c \in V$  then
2   UnCutAtVertex( $c$ )
3 else if  $c \in E_{\circ}$  then
4   UnCutAtClosedEdge( $c$ )
5 else if  $c \in E_{\mid}$  then
6   UnCutAtOpenEdge( $c$ )
7 else if  $c \in F$  then
8   Do nothing

```

---

**UnCutAtVertex** ( $v \in V$ )

---

```

1  if NOT CanUnCutAtVertex( $v$ ) then
2      Do nothing
3  else
    ▷ Handle case where  $v$  could have been created via CutFaceAtVertex().
4      if  $N_{\text{Steiner-vertex-use}} = 1$  and  $N_{\text{end-vertex-use}} = 0$  then
5           $f \leftarrow$  only face in  $\text{star}(v)$ 
6           $i \leftarrow$  index of Steiner cycle of  $f$  using  $v$ 
7          RemoveCycleFromFace( $f, i$ )
    ▷ Handle case where  $v$  could have been created via CutClosedEdge().
8      if  $N_{\text{Steiner-vertex-use}} = 0$  and  $N_{\text{end-vertex-use}} = 2$  and  $N_{\text{incident-edges}} = 1$  then
9           $e^\circ \leftarrow$  CreateClosedEdge()
10          $e \leftarrow$  only edge in  $\text{star}(v)$ 
11         for all face  $f \in \text{star}(v)$  do
12             for all non-simple cycle  $\gamma_i \in \hat{\partial}f$  do
13                 if  $\gamma_i$  uses  $e$  then
14                      $(\beta, N) \leftarrow$  values such that  $\gamma_i = [(e, \beta)^N]$ 
15                      $\gamma_i(f) \leftarrow [(e^\circ, \beta)^N]$ 
16                 HardDelete( $e$ )
    ▷ Handle case where  $v$  could have been created via CutOpenEdge().
17     if  $N_{\text{Steiner-vertex-use}} = 0$  and  $N_{\text{end-vertex-use}} = 2$  and  $N_{\text{incident-edges}} = 2$  then
        ▷ Compute  $h_1$  and  $h_2$ , the two halfedges such that  $\xrightarrow{h_1} \bullet \xrightarrow{h_2}$ .
18          $(e_1, e_2) \leftarrow$  the two edges in  $\text{star}(v)$ 
19         if  $v_{\text{end}}(e_1) = v$  then  $\beta_1 \leftarrow \top$  else  $\beta_1 \leftarrow \perp$ 
20         if  $v_{\text{start}}(e_2) = v$  then  $\beta_2 \leftarrow \top$  else  $\beta_2 \leftarrow \perp$ 
21          $h_1 \leftarrow (e_1, \beta_1); h_2 \leftarrow (e_2, \beta_2)$ 
        ▷ Create the new open edge  $e = (v_{\text{start}}(h_1), v_{\text{end}}(h_2))$ .
22          $e \leftarrow$  CreateOpenEdge( $v_{\text{start}}(h_1), v_{\text{end}}(h_2)$ )
        ▷ Replace every occurrence of  $\xrightarrow{h_1} \bullet \xrightarrow{h_2}$  by  $(e, \top)$  and every occurrence of  $\xleftarrow{h_2} \bullet \xleftarrow{h_1}$  by  $(e, \perp)$ .
23         for all face  $f \in \text{star}(v)$  do
24             for all non-simple cycle  $\gamma_i \in \hat{\partial}f$  do
25                  $\gamma'_i \leftarrow []$ 

```

---

```

26      for all  $j \in [1..N(\gamma_i)]$  do
27          if  $e_j(\gamma_i) = e_1$  then
28              Do nothing.
29          else if  $e_j(\gamma_i) = e_2$  then
30              if  $\beta_j(\gamma_i) = \beta_2$  then
31                  Append  $(e, \top)$  to  $\gamma'_i$ 
32              else
33                  Append  $(e, \perp)$  to  $\gamma'_i$ 
34          else
35              Append  $h_j(\gamma_i)$  to  $\gamma'_i$ 
36       $\gamma_i(f) \leftarrow \gamma'_i$ 

       $\triangleright$  Delete  $e_1$  and  $e_2$ .
37      HardDelete( $e_1$ )
38      HardDelete( $e_2$ )

       $\triangleright$  In any of the previous cases, delete  $v$ .
39      HardDelete( $v$ )

```

---



---

**UnCutAtClosedEdge** ( $e^\circ \in E_o$ )

---

```

1  if NOT CanUnCutAtClosedEdge( $e^\circ$ ) then
2      Do nothing
3  else
4      if  $N_{\text{incident-faces}} = 1$  then  $\triangleright$  1 face  $f'$ : Case (a), (c), (d), (e), or (f) (cf. Figure D.5)
5           $f' \leftarrow$  face using  $e^\circ$ 
6          if  $N_{\text{cycles-using-e}} = 1$  then  $\triangleright$  1 cycle  $\gamma_i = [(e^\circ, \beta)^2]$ : Case (c) or (e)
7               $i \leftarrow$  index of cycle of  $f'$  using  $e^\circ$ 
8              RemoveCycleFromFace( $f', i$ )
9              if  $\epsilon(f') = \emptyset$  then  $\triangleright f'$  non-orientable: Case (e)
10                  $g(f') \leftarrow g(f') + 1$ 
11             else  $\triangleright f'$  orientable: Case (c)
12                  $\epsilon(f') \leftarrow \emptyset$ 
13                  $g(f') \leftarrow 2g(f') + 1$ 
14         else  $\triangleright$  2 cycles  $\gamma_{i_1} = [(e^\circ, \beta_1)]$  and  $\gamma_{i_2} = [(e^\circ, \beta_2)]$ : Case (a), (d), or (f)
15              $(i_1, i_2) \leftarrow$  indices of cycles of  $f'$  using  $e^\circ$ 
16              $\beta_1 \leftarrow \beta(\gamma_{i_1}(f'))$ 
17              $\beta_2 \leftarrow \beta(\gamma_{i_2}(f'))$ 

```

---

```

18      RemoveCyclesFromFace( $f', \{i_1, i_2\}$ )
19      if  $\epsilon(f') = \emptyset$  then                                      $\triangleright f'$  non-orientable: Case (f)
20           $g(f') \leftarrow g(f') + 2$ 
21      else if  $\beta_1 = \beta_2$  then                                      $\triangleright f'$  orientable,  $\beta_1 = \beta_2$ : Case (d)
22           $\epsilon(f') \leftarrow \emptyset$ 
23           $g(f') \leftarrow 2g(f') + 2$ 
24      else                                                          $\triangleright f'$  orientable,  $\beta_1 \neq \beta_2$ : Case (a)
25           $g(f') \leftarrow g(f') + 1$ 
26  else  $\triangleright 2$  faces  $f_1$  and  $f_2$ , 2 cycles  $\gamma_{i_1} = [(e^\circ, \beta_1)]$  and  $\gamma_{i_2} = [(e^\circ, \beta_2)]$ : Case (b), (g), (h), or (i)
27       $(f_1, f_2) \leftarrow$  faces using  $e^\circ$ 
28       $i_1 \leftarrow$  index of cycle of  $f_1$  using  $e^\circ$ 
29       $i_2 \leftarrow$  index of cycle of  $f_2$  using  $e^\circ$ 
30       $\beta_1 \leftarrow \beta(\gamma_{i_1}(f_1))$ 
31       $\beta_2 \leftarrow \beta(\gamma_{i_2}(f_2))$ 
32      if  $\epsilon(f_1) = \emptyset$  and  $\epsilon(f_2) = \emptyset$  then                  $\triangleright \epsilon_1 = \emptyset, \epsilon_2 = \emptyset$ : Case (i)
33           $f \leftarrow \text{CreateFace}(\emptyset, g(f_1) + g(f_2))$ 
34      else if  $\epsilon(f_1) = \emptyset$  and  $\epsilon(f_2) = \circ$  then                  $\triangleright \epsilon_1 = \emptyset, \epsilon_2 = \circ$ : Case (h)
35           $f \leftarrow \text{CreateFace}(\emptyset, g(f_1) + 2g(f_2))$ 
36      else if  $\epsilon(f_1) = \circ$  and  $\epsilon(f_2) = \emptyset$  then                  $\triangleright \epsilon_1 = \circ, \epsilon_2 = \emptyset$ : Case (g)
37           $f \leftarrow \text{CreateFace}(\emptyset, 2g(f_1) + g(f_2))$ 
38      else                                                          $\triangleright \epsilon_1 = \circ, \epsilon_2 = \circ$ : Case (b)
39           $f \leftarrow \text{CreateFace}(\circ, g(f_1) + g(f_2))$ 
40      if  $\beta_1 = \beta_2$  then
41          for all  $i \in [1..k(f_2)]$  do
42              FlipCycle( $f_2, i$ )
43      for all  $i \in [1..k(f_1)], i \neq i_1$  do
44          AddCycleToFace( $f, \gamma_i(f_1)$ )
45      for all  $i \in [1..k(f_2)], i \neq i_2$  do
46          AddCycleToFace( $f, \gamma_i(f_2)$ )
47      HardDelete( $f_1$ )
48      HardDelete( $f_2$ )
49      HardDelete( $e^\circ$ )

```

---

**UnCutAtOpenEdge** ( $e \in E_l$ )

---

```

1  if NOT CanUnCutAtOpenEdge( $e$ ) then
2      Do nothing
3  else
4      if  $N_{\text{incident-faces}} = 1$  then           ▷ 1 face  $f'$ : Case (j), (l), (m), (n), (o), or (s) (cf. Figure D.6)
5           $f' \leftarrow$  face using  $e$ 
6          if  $N_{\text{cycles-using-}e} = 1$  then       ▷ 1 cycle  $\gamma_i = [\pi_1, (e, \beta_1), \pi_2, (e, \beta_2)]$ : Case (l), (n), or (s)
7               $i \leftarrow$  index of the cycle of  $f'$  using  $e$ 
8               $\gamma_i \leftarrow \gamma_i(f')$ 
9               $(j_1, j_2) \leftarrow$  indices of the two halfedges of  $\gamma_i$  using  $e$ 
10              $\beta_1 \leftarrow \beta_{j_1}(\gamma_i)$ 
11              $\beta_2 \leftarrow \beta_{j_2}(\gamma_i)$ 
12              $\pi_1 \leftarrow \text{SubPath}(\gamma_i, j_1, j_2 - 1)$ 
13              $\pi_2 \leftarrow \text{SubPath}(\gamma_i, j_2, j_1 - 1)$ 
14             RemoveCycleFromFace( $f', i$ )
15             if  $\beta_1 = \beta_2$  then
16                 AddCycleToFace( $f', [\pi_1, \overline{\pi_2}]$ )
17                 if  $\epsilon(f') = \emptyset$  then           ▷  $\beta_1 = \beta_2$ ,  $f'$  non-orientable: Case (n)
18                      $g(f') \leftarrow g(f') + 1$ 
19                 else                             ▷  $\beta_1 = \beta_2$ ,  $f'$  orientable: Case (l)
20                      $\epsilon(f') \leftarrow \emptyset$ 
21                      $g(f') \leftarrow 2g(f') + 1$ 
22             else                                 ▷  $\beta_1 \neq \beta_2$ : Case (s)
23                 AddCycleToFace( $f', [\pi_1]$ )
24                 AddCycleToFace( $f', [\pi_2]$ )
25             else                                 ▷ 2 cycles  $\gamma_{i_1} = [\pi_1, (e, \beta_1)]$  and  $\gamma_{i_2} = [\pi_2, (e, \beta_2)]$ : Case (j), (m), or (o)
26                  $(i_1, i_2) \leftarrow$  indices of the cycles of  $f'$  using  $e$ 
27                  $\gamma_{i_1} \leftarrow \gamma_{i_1}(f')$ 
28                  $\gamma_{i_2} \leftarrow \gamma_{i_2}(f')$ 
29                  $j_1 \leftarrow$  index of the halfedge of  $\gamma_{i_1}$  using  $e$ 
30                  $j_2 \leftarrow$  index of the halfedge of  $\gamma_{i_2}$  using  $e$ 
31                  $\beta_1 \leftarrow \beta_{j_1}(\gamma_{i_1})$ 
32                  $\beta_2 \leftarrow \beta_{j_2}(\gamma_{i_2})$ 
33                  $\pi_1 \leftarrow \text{SubPath}(\gamma_{i_1}, j_1, j_1 - 1)$ 
34                  $\pi_2 \leftarrow \text{SubPath}(\gamma_{i_2}, j_2, j_2 - 1)$ 
35                 RemoveCyclesFromFace( $f', \{i_1, i_2\}$ )

```

---

---

```

36   if  $\epsilon(f') = \emptyset$  then  $\triangleright f'$  non-orientable: Case (o)
37       if  $\beta_1 = \beta_2$  then
38           AddCycleToFace( $f'$ ,  $[\pi_1, \overline{\pi_2}]$ )
39       else
40           AddCycleToFace( $f'$ ,  $[\pi_1, \pi_2]$ )
41            $g(f') \leftarrow g(f') + 2$ 
42       else if  $\beta_1 = \beta_2$  then  $\triangleright f'$  orientable,  $\beta_1 = \beta_2$ : Case (m)
43           AddCycleToFace( $f'$ ,  $[\pi_1, \overline{\pi_2}]$ )
44            $\epsilon(f') \leftarrow \emptyset$ 
45            $g(f') \leftarrow 2g(f') + 2$ 
46       else  $\triangleright f'$  orientable,  $\beta_1 \neq \beta_2$ : Case (j)
47           AddCycleToFace( $f'$ ,  $[\pi_1, \pi_2]$ )
48            $g(f') \leftarrow g(f') + 1$ 
49   else  $\triangleright 2$  faces  $f_1$  and  $f_2$ , 2 cycles  $\gamma_{i_1} = [\pi_1, (e, \beta_1)]$  and  $\gamma_{i_2} = [\pi_2, (e, \beta_2)]$ : Case (k), (p), (k), or (r)
50        $(f_1, f_2) \leftarrow$  faces using  $e$ 
51        $i_1 \leftarrow$  index of cycle of  $f_1$  using  $e$ 
52        $i_2 \leftarrow$  index of cycle of  $f_2$  using  $e$ 
53        $\gamma_{i_1} \leftarrow \gamma_{i_1}(f_1)$ 
54        $\gamma_{i_2} \leftarrow \gamma_{i_2}(f_2)$ 
55        $j_1 \leftarrow$  index of the halfedge of  $\gamma_{i_1}$  using  $e$ 
56        $j_2 \leftarrow$  index of the halfedge of  $\gamma_{i_2}$  using  $e$ 
57        $\beta_1 \leftarrow \beta_{j_1}(\gamma_{i_1})$ 
58        $\beta_2 \leftarrow \beta_{j_2}(\gamma_{i_2})$ 
59        $\pi_1 \leftarrow \text{SubPath}(\gamma_{i_1}, j_1, j_1 - 1)$ 
60        $\pi_2 \leftarrow \text{SubPath}(\gamma_{i_2}, j_2, j_2 - 1)$ 
61       if  $\epsilon(f_1) = \emptyset$  and  $\epsilon(f_2) = \emptyset$  then  $\triangleright \epsilon_1 = \emptyset, \epsilon_2 = \emptyset$ : Case (r)
62            $f \leftarrow \text{CreateFace}(\emptyset, g(f_1) + g(f_2))$ 
63       else if  $\epsilon(f_1) = \emptyset$  and  $\epsilon(f_2) = \circ$  then  $\triangleright \epsilon_1 = \emptyset, \epsilon_2 = \circ$ : Case (q)
64            $f \leftarrow \text{CreateFace}(\emptyset, g(f_1) + 2g(f_2))$ 
65       else if  $\epsilon(f_1) = \circ$  and  $\epsilon(f_2) = \emptyset$  then  $\triangleright \epsilon_1 = \circ, \epsilon_2 = \emptyset$ : Case (p)
66            $f \leftarrow \text{CreateFace}(\emptyset, 2g(f_1) + g(f_2))$ 
67       else  $\triangleright \epsilon_1 = \circ, \epsilon_2 = \circ$ : Case (k)
68            $f \leftarrow \text{CreateFace}(\circ, g(f_1) + g(f_2))$ 
69       if  $\beta_1 = \beta_2$  then
70           for all  $i \in [1..k(f_2)]$  do
71               FlipCycle( $f_2, i$ )
72   for all  $i \in [1..k(f_1)], i \neq i_1$  do

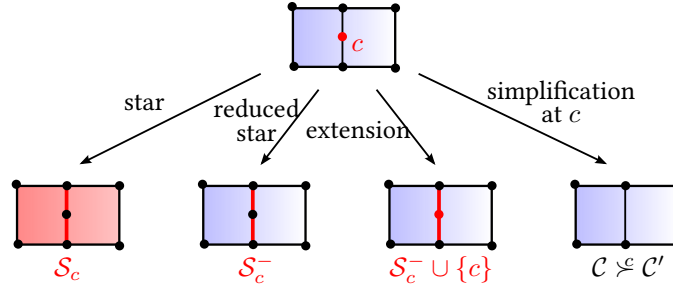
```

```
73      AddCycleToFace( $f, \gamma_i(f_1)$ )
74      for all  $i \in [1..k(f_2)], i \neq i_2$  do
75          AddCycleToFace( $f, \gamma_i(f_2)$ )
76      if  $\beta_1 = \beta_2$  then
77          AddCycleToFace( $f, [\pi_1, \overline{\pi_2}]$ )
78      else
79          AddCycleToFace( $f, [\pi_1, \pi_2]$ )
80      HardDelete( $f_1$ )
81      HardDelete( $f_2$ )
82  HardDelete( $e^\circ$ )
```

---

# Appendix E

## Simplification of PCS Complexes



**Figure E.1:** *Reduced star and atomic simplification.*

In this appendix, we define the concept of simplification of PCS complexes, which is used to define the *uncut* topological operator, and to define the concept of *minimal decomposition*. We conjecture that given a PCS complex  $\mathcal{K} = (X, \mathcal{C})$ , then  $X$  has a unique minimal PCS decomposition  $\mathcal{K}_m = (X, \mathcal{C}_m)$ . However, we leave the proof for future work. One idea to prove it might be to triangulate the PCS complex, and show that any sequence of simplifications leads to the unique decomposition discussed in [De Floriani et al. 2003].

### E.1 Simplification of Cell Complexes

In this section, we define the concept of *simplification*, intuitively an operation transforming a cell complex  $\mathcal{K} = (X, \mathcal{C})$  into another complex  $\mathcal{K}' = (X, \mathcal{C}')$ , decomposing the same space with strictly fewer cells. We recall that a PCS complex is defined as a cell complex whose dimension is at most two (see Appendix B). We also recall that  $\mathcal{S}_c$  denotes the *star* of  $c$ , that is, the set of cells whose boundary contain  $c$ . Finally, we recall that the notation  $\langle \mathcal{C} \rangle$  is used to represent the union of cells in  $\mathcal{C}$ .

**Reduced star** Let  $\mathcal{K} = (X, \mathcal{C})$  be a cell complex and  $c \in \mathcal{C}$ . The **reduced star** of  $c$  is defined as:

$$\mathcal{S}_c^- = \begin{cases} \emptyset & \text{if } \mathcal{S}_c = \emptyset \\ \{c' \in \mathcal{S}_c \mid \dim c' = n_c^-\} & \text{otherwise, where } n_c^- = \min_{c' \in \mathcal{S}_c} (\dim c') \end{cases} \quad (\text{E.1})$$



In other words, the reduced star of  $c$  is defined as the cells of lowest dimension among the cells in the star of  $c$ , as illustrated in Figure E.1. Note that  $n_c^-$  is not necessarily equal to  $\dim(c) + 1$ . For example, the reduced star of a Steiner vertex of a PCS complex is equal to the face(s) surrounding the Steiner vertex (in this case, the reduced star is in fact equal to the star).

**Extension** The *extension* of a cell  $c$  is defined as its reduced star extended by  $c$  itself:

$$\widehat{\mathcal{S}}_c^- = \{c\} \cup \mathcal{S}_c^- \quad (\text{E.2})$$

$$\widehat{c} = \langle \widehat{\mathcal{S}}_c^- \rangle = \langle c, \mathcal{S}_c^- \rangle \quad (\text{E.3})$$

**Atomic simplification** Let  $\mathcal{K} = (X, \mathcal{C})$  be a cell complex and  $c \in \mathcal{C}$ . We say that  $\mathcal{K}$  can be simplified at  $c$  if and only if the following constraints are satisfied:

- $\mathcal{S}_c \neq \emptyset$
- $\mathcal{K}' = (X, \mathcal{C}')$  is a cell complex, where  $\mathcal{C}' = (\mathcal{C} \setminus \widehat{\mathcal{S}}_c^-) \cup \{\widehat{c}\}$

In this case, we write  $\mathcal{K} \searrow \mathcal{K}'$ . For the dimension two or less, checking whether “ $\mathcal{K}$  can be simplified at  $c$ ” can be done combinatorially with the algorithm  $\text{CanUnCut}(c)$ . If yes, then  $\mathcal{K}'$  is obtained by the algorithm  $\text{UnCut}(c)$ .

Let  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X', \mathcal{C}')$  be two cell complexes. We say that  $\mathcal{K}$  can be atomically simplified into  $\mathcal{K}'$ , which we denote  $\mathcal{K} \succ^\bullet \mathcal{K}'$ , if they satisfy the relation defined below:

$$\mathcal{K} \succ^\bullet \mathcal{K}' \Leftrightarrow \exists c \in \mathcal{C}, \mathcal{K} \searrow \mathcal{K}' \quad (\text{E.4})$$

It follows directly that if  $(X, \mathcal{C}) \succ^\bullet (X', \mathcal{C}')$  then  $X' = X$ . Therefore, as an abuse of notation, we will often write  $\mathcal{C} \succ^\bullet \mathcal{C}'$  instead of  $\mathcal{K} \succ^\bullet \mathcal{K}'$ .

**Proposition 8.** *If  $\mathcal{C} \succ^\bullet \mathcal{C}'$ , then  $|\mathcal{C}| > |\mathcal{C}'|$ , where  $|\mathcal{C}|$  denotes the number of cells in  $\mathcal{C}$ .*

*Proof.* If  $\mathcal{C} \succ^\bullet \mathcal{C}'$ , then  $\exists c \in \mathcal{C}, \mathcal{C} \searrow \mathcal{C}'$  and we have  $\mathcal{C}' = (\mathcal{C} \setminus \widehat{\mathcal{S}}_c^-) \cup \{\widehat{c}\}$ , thus  $|\mathcal{C}'| = |\mathcal{C}| - |\widehat{\mathcal{S}}_c^-| + 1$ . Since  $\mathcal{C}$  can be simplified at  $c$ , this means  $\mathcal{S}_c \neq \emptyset$ , thus  $\mathcal{S}_c^- \neq \emptyset$ , thus  $\widehat{\mathcal{S}}_c^-$  contains at least two cells:  $c$  and one belonging to  $\mathcal{S}_c^-$ . Thus  $|\widehat{\mathcal{S}}_c^-| \geq 2$ , thus  $|\mathcal{C}'| \leq |\mathcal{C}| - 2 + 1$ , thus  $|\mathcal{C}'| \leq |\mathcal{C}| - 1$ .  $\square$

**Simplification** We define the binary relation  $\succ$  to be the transitive closure of  $\succ^\bullet$  (i.e., the minimal transitive relation containing  $\succ^\bullet$ ). In other words,  $\mathcal{C} \succ \mathcal{C}'$  if and only if a finite sequence of atomic simplification transforms  $\mathcal{C}$  into  $\mathcal{C}'$ . In this case, we say that  $\mathcal{C}$  can be simplified into  $\mathcal{C}'$ . This

relation can be equivalently defined as:

$$\mathcal{C}_0 \succ \mathcal{C}' \Leftrightarrow \begin{cases} \exists k \in \mathbb{N}^+, \\ \exists \mathcal{C}_1, \dots, \mathcal{C}_{k-1} \text{ decomposing } X, \\ \forall i \in [0..k-1], \exists \mathcal{C}_i \in \mathcal{C}_i, \\ \mathcal{C}_0 \succ^{\mathcal{C}_0} \mathcal{C}_1 \succ^{\mathcal{C}_1} \dots \succ^{\mathcal{C}_{k-2}} \mathcal{C}_{k-1} \succ^{\mathcal{C}_{k-1}} \mathcal{C}' \end{cases} \quad (\text{E.5})$$

**Proposition 9.** *If  $\mathcal{C} \succ \mathcal{C}'$ , then  $|\mathcal{C}| > |\mathcal{C}'|$ .*

*Proof.* If  $\mathcal{C} \succ \mathcal{C}'$ , then  $\mathcal{C} \succ^{\mathcal{C}_0} \mathcal{C}_1 \succ^{\mathcal{C}_1} \dots \succ^{\mathcal{C}_{k-2}} \mathcal{C}_{k-1} \succ^{\mathcal{C}_{k-1}} \mathcal{C}'$ , then  $|\mathcal{C}| > |\mathcal{C}_1| > \dots > |\mathcal{C}_{k-1}| > |\mathcal{C}'|$ , then  $|\mathcal{C}| > |\mathcal{C}'|$ .  $\square$

**Proposition 10.**  *$\succ$  is a strict partial order.*

*Proof.* We verify below that it is irreflexive, transitive and asymmetric:

- Irreflexivity: we have  $\neg(|\mathcal{C}| > |\mathcal{C}|)$ , thus  $\neg(\mathcal{C} \succ \mathcal{C})$ .
- Transitivity: by definition.
- Asymmetry: If  $\mathcal{C} \succ \mathcal{C}'$  then  $(|\mathcal{C}| > |\mathcal{C}'|)$  then  $\neg(|\mathcal{C}'| > |\mathcal{C}|)$  then  $\neg(\mathcal{C}' \succ \mathcal{C})$ .  $\square$

**Minimal complex** Let  $\Omega$  be a set of cell complexes. A cell complex  $\mathcal{K} \in \Omega$  is said to be a minimal element of  $\Omega$  if it is minimal for  $\succ$ , i.e. if there are no  $\mathcal{K}' \in \Omega$  such that  $\mathcal{K} \succ \mathcal{K}'$ . In other words, a cell complex is said to be minimal if it cannot be simplified to another cell complex in  $\Omega$ . Formally:

$$\mathcal{K} \text{ minimal in } \Omega \Leftrightarrow \forall \mathcal{K}' \in \Omega, \neg(\mathcal{K} \succ \mathcal{K}') \quad (\text{E.6})$$

By extension, if no set  $\Omega$  is specified,  $\mathcal{K}$  is said to be *minimal*, or *simple*, if it cannot be simplified:

$$\mathcal{K} = (X, \mathcal{C}) \text{ minimal} \Leftrightarrow \forall c \in \mathcal{C}, \forall \mathcal{C}' = (X, \mathcal{C}'), \neg(\mathcal{K} \succ \mathcal{C}') \quad (\text{E.7})$$

**Proposition 11.**  *$\succ$  is a well-founded strict partial order, i.e. every non-empty set of cell complexes  $\Omega$  has a minimal element.*

*Proof.* Let  $n_m = \min\{|\mathcal{C}| \mid \mathcal{C} \in \Omega\}$  (exists because  $<$  on  $\mathbb{N}$  is well-founded), and  $\mathcal{C}_m$  such as  $|\mathcal{C}_m| = n_m$ . By definition of  $n_m$ , we have  $\neg(|\mathcal{C}'| < |\mathcal{C}_m|)$  for each  $\mathcal{C}'$  in  $\Omega$ , thus  $\neg(\mathcal{C}_m \succ \mathcal{C}')$ , thus  $\mathcal{C}_m$  is a minimal element of  $\Omega$ .  $\square$

**Corollary 2.** *There are no infinite descending chains:*

$$\mathcal{C}_0 \succ \mathcal{C}_1 \succ \dots \succ \mathcal{C}_k \succ \dots \quad (\text{E.8})$$

*Proof.* Well-founded strict partial orders do not have infinite descending chains.  $\square$

**Corollary 3.** *Let  $X$  be a topological space admitting a cell complex  $\mathcal{K}$ . Then there exists  $\mathcal{K}_m$  decomposing  $X$  such that  $\mathcal{K}_m$  is minimal.*

*Proof.* Let  $\Omega$  be the set of all cell complexes decomposing  $X$ . It is non-empty since  $\mathcal{K} \in \Omega$ , thus there exists  $\mathcal{K}_m$  minimal using Proposition 11.  $\square$

Finally, we conclude this section by defining the weak versions of the simplification binary operators.

**Weak atomic simplification** Let  $\mathcal{K} = (X, \mathcal{C})$  be a cell complex and  $c \in \mathcal{C}$ . We conveniently write  $\mathcal{K} \not\asymp \mathcal{K}'$  to define  $\mathcal{K}'$  as being equal to:

- $\mathcal{K}$  if  $\mathcal{K}$  cannot be simplified at  $c$ .
- the atomic simplification of  $\mathcal{K}$  at  $c$  otherwise.

We define the relation  $\asymp^\bullet$  to be the reflexive closure of  $\asymp$ :

$$\mathcal{K} \asymp^\bullet \mathcal{K}' \Leftrightarrow \begin{cases} \mathcal{K} = \mathcal{K}', \text{ or} \\ \mathcal{K} \asymp \mathcal{K}' \end{cases} \quad (\text{E.9})$$

**Weak simplification** We define the relation  $\succsim$  to be the reflexive closure of  $\succ$ :

$$\mathcal{K} \succsim \mathcal{K}' \Leftrightarrow \begin{cases} \mathcal{K} = \mathcal{K}', \text{ or} \\ \mathcal{K} \succ \mathcal{K}' \end{cases} \quad (\text{E.10})$$

## E.2 Equivalence of Cell Complexes

**Bi-directional atomic simplification** We define the relation  $\overset{\bullet}{\leftrightarrow}$  to be the symmetric closure of  $\asymp^\bullet$ , that is:

$$\mathcal{K} \overset{\bullet}{\leftrightarrow} \mathcal{K}' \Leftrightarrow \begin{cases} \mathcal{K} \asymp^\bullet \mathcal{K}', \text{ or} \\ \mathcal{K}' \asymp^\bullet \mathcal{K} \end{cases} \quad (\text{E.11})$$

**Bi-directional simplification** We define the relation  $\leftrightarrow$  to be the transitive closure of  $\overset{\bullet}{\leftrightarrow}$ , that is,  $\mathcal{K} \leftrightarrow \mathcal{K}'$  if and only if a finite sequence of atomic simplification or de-simplification transforms  $\mathcal{K}$  into  $\mathcal{K}'$ :

$$\mathcal{K} \leftrightarrow \mathcal{K}' \Leftrightarrow \mathcal{K} \overset{\bullet}{\leftrightarrow} \mathcal{K}_1 \overset{\bullet}{\leftrightarrow} \dots \overset{\bullet}{\leftrightarrow} \mathcal{K}_{k-1} \overset{\bullet}{\leftrightarrow} \mathcal{K}' \quad (\text{E.12})$$

**Equivalence relation** We define the relation  $\equiv$  to be the reflexive closure of  $\leftrightarrow$ :

$$\mathcal{K} \equiv \mathcal{K}' \Leftrightarrow \begin{cases} \mathcal{K} = \mathcal{K}', \text{ or} \\ \mathcal{K} \leftrightarrow \mathcal{K}' \end{cases} \quad (\text{E.13})$$

It is an equivalence relation, since it is symmetric, transitive and reflexive. Note that it is important to take the transitive closure after the symmetric closure: two decompositions  $\mathcal{C}$  and  $\mathcal{C}'$  can have the same number of cells (and thus we have neither  $\mathcal{C} \succ \mathcal{C}'$  nor  $\mathcal{C}' \succ \mathcal{C}$ ), but still could be obtained via a de-simplification followed by a simplification:  $\mathcal{C} \prec \mathcal{C}'' \succ \mathcal{C}'$ . In fact, we will see later that it is always possible.

**Proposition 12.** *Two cell complexes are equivalent if and only if they are equal or obtained from one another via a finite sequence of atomic simplification or de-simplification:*

$$\mathcal{K} \equiv \mathcal{K}' \Leftrightarrow \begin{cases} \mathcal{K} = \mathcal{K}', \text{ or} \\ \mathcal{K} \xleftrightarrow{\bullet} \mathcal{K}_1 \xleftrightarrow{\bullet} \dots \xleftrightarrow{\bullet} \mathcal{K}_{k-1} \xleftrightarrow{\bullet} \mathcal{K}' \end{cases} \quad (\text{E.14})$$

*Proof.* Combine Definition E.12 with Definition E.13. □

**Proposition 13.** *Let  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X', \mathcal{C}')$  be two cell complexes. Then we have:*

$$\mathcal{K} \equiv \mathcal{K}' \Rightarrow X = X' \quad (\text{E.15})$$

*Proof.* We have  $\mathcal{K} \preceq \mathcal{K}' \Rightarrow X = X'$  directly from the definition of  $\preceq$ , which implies that  $X = X'$  whenever  $\mathcal{K}$  and  $\mathcal{K}'$  are related by any of a closures of  $\preceq$  defined above. □

**Proposition 14.** *Let  $\mathcal{K}$  and  $\mathcal{K}'$  be two cell complexes. Then we have:*

$$\mathcal{K} \succcurlyeq \mathcal{K}' \Rightarrow \mathcal{K} \equiv \mathcal{K}' \quad (\text{E.16})$$

*Proof.*  $\mathcal{K} \succcurlyeq \mathcal{K}' \Rightarrow (\mathcal{K} = \mathcal{K}' \text{ or } \mathcal{K} \succ \mathcal{K}')$ . In the first case,  $\mathcal{K} \equiv \mathcal{K}'$  since  $\equiv$  is reflexive. In the second case, we have  $\mathcal{K} \succ \dots \succ \mathcal{K}'$ , hence  $\mathcal{K} \xleftrightarrow{\bullet} \dots \xleftrightarrow{\bullet} \mathcal{K}'$ , hence  $\mathcal{K} \equiv \mathcal{K}'$ . □

**Conjecture 1.** *Let  $X$  be a topological space, and  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X, \mathcal{C}')$  be two cell complexes decomposing  $X$ . Then they admit a common “ancestor”, i.e.:*

$$\exists \mathcal{K}'' = (X, \mathcal{C}'') \text{ such that } \begin{cases} \mathcal{K}'' \succcurlyeq \mathcal{K}, \text{ and} \\ \mathcal{K}'' \succcurlyeq \mathcal{K}' \end{cases} \quad (\text{E.17})$$

We expect that a proof can be achieved by explicitly constructing  $\mathcal{C}''$  as intersections of cells in  $\mathcal{C}$  with cells in  $\mathcal{C}'$ . Then, we would have to prove that  $\mathcal{K}'' = (X, \mathcal{C}'')$  is a cell complex. The following of this Section E.2 (but no other sections) assumes that this conjecture is true.

**Theorem 2 (Equivalence Theorem).** *Let  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X', \mathcal{C}')$  be two cell complexes. Then we have:*

$$\mathcal{K} \equiv \mathcal{K}' \Leftrightarrow X = X' \quad (\text{E.18})$$

*In other words: the equivalent cell complexes are exactly those decomposing the same space.*

*Proof.* We have already  $\mathcal{K} \equiv \mathcal{K}' \Rightarrow X = X'$ . Let  $X$  be a topological space and  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X, \mathcal{C}')$  be two cell complexes decomposing  $X$ . Let  $\mathcal{K}'' = \text{CommonAncestor}(\mathcal{K}, \mathcal{K}')$ . We have  $\mathcal{K}'' \succcurlyeq \mathcal{K}$  and  $\mathcal{K}'' \succcurlyeq \mathcal{K}'$ , thus  $\mathcal{K}'' \equiv \mathcal{K}$  and  $\mathcal{K}'' \equiv \mathcal{K}'$ , thus  $\mathcal{K} \equiv \mathcal{K}'$  by transitivity.  $\square$

**Corollary 4.** *Let  $X$  be a topological space, and  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X, \mathcal{C}')$  be two cell complexes decomposing  $X$ . Then it is possible to transform  $\mathcal{K}$  into  $\mathcal{K}'$  via a finite sequence of atomic simplification or de-simplification.*

*Proof.* We simply combine the result of the equivalence theorem with Corollary 12.  $\square$

### E.3 Uniqueness of Minimal PCS Complex

We have seen, for arbitrary dimension, that if  $X$  admits a cell complex  $\mathcal{K} = (X, \mathcal{C})$ , then it also admits one  $\mathcal{K}_m$  that is minimal. In fact, regardless whether Conjecture 1 is true or false, it also admits one which is both minimal and equivalent to  $\mathcal{K}$  (i.e., that can be obtained from a finite sequence of simplification or de-simplification):

**Proposition 15 (Minimal decomposition).** *Let  $\mathcal{K} = (X, \mathcal{C})$  be a cell complex. Then there exists a minimal cell complex  $\mathcal{K}_m$  such that  $\mathcal{K} \equiv \mathcal{K}_m$ .*

*Proof.* We have seen that there exists no infinite decreasing sequences of cell complexes. Hence, by defining  $\mathcal{K}_0 = \mathcal{K}$ , we can recursively define  $\mathcal{K}_{i+1}$  by  $\mathcal{K}_i \succcurlyeq^{\mathcal{C}_i} \mathcal{K}_{i+1}$  while there exists a cell  $c_i \in \mathcal{C}_i$  such that  $\mathcal{K}_i$  can be atomically simplified at  $c_i$ . This procedure necessarily stops, and then there exists  $N \in \mathbb{N}$  such that  $\mathcal{K}_N$  cannot be atomically simplified at any cell and  $\mathcal{K}_0 \succcurlyeq^{\mathcal{C}_0} \dots \succcurlyeq^{\mathcal{C}_{N-1}} \mathcal{K}_N$ . Thus,  $\mathcal{K}_N$  minimal and  $\mathcal{K} \succcurlyeq \mathcal{K}_N$ , thus  $\mathcal{K}_N$  minimal and  $\mathcal{K} \equiv \mathcal{K}_N$ .  $\square$

In the case of the dimension two or less, we conjecture that this minimal decomposition is unique, which would imply that performing simplifications in any order until it is not possible anymore leads necessarily to this unique minimal decomposition.

**Conjecture 2 (Unique minimal decomposition).** *Let  $X$  be a topological space, and  $\mathcal{K} = (X, \mathcal{C})$  and  $\mathcal{K}' = (X, \mathcal{C}')$  be two minimal PCS complexes decomposing  $X$ . Then  $\mathcal{K} = \mathcal{K}'$ .*