UNIVERSITY OF CALGARY

Cell Complexes: The Structure of Space and the Mathematics of Modularity

by

Brendan Joseph Lane

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 2015

Abstract

The modeling of growing multicellular structures is of fundamental importance in investigating plant development. A distinctive feature of plants is that, with rare exceptions, cells do not move with respect to each other; the only differences in tissue topology are due to cell divisions. Corresponding features are found in other application domains, such as geometric modeling. Models from these diverse domains, taken together, constitute the field of *developmental modeling*. This dissertation is concerned with devising a mathematical and computational formalism for such modeling.

Examining simple examples of one-dimensional models shows that the mathematical structure of the *cell complex* is ideal for developmental modeling. The cell complex consists of mathematical cells of different dimensions, letting physical quantities of different inherent dimension sit in their proper place in the structure. A cell complex can be represented in an index-free manner, and topological operations on it, including the important case of cell division, can be effected in a local manner. Finally, the cell complex is built on neighbourhood relations which let developmental rules easily access values in neighbouring cells.

A novel data structure, the *flip*, records a single adjacency between cells in a cell complex; a *flip table*, the collection of all flips in the complex, is in turn sufficient to represent the complex itself. This representation is used to build the *Cell Complex Framework*, a C++ API which can be used for computational modeling of development in any number of dimensions. The framework provides basic operations such as iterating over a cell complex, adding, removing, dividing, and merging cells, and computing geometric information such as orientation, measure, and centroid. Some developmental models from the existing literature, in both two and three dimensions, are reproduced using the Cell Complex Framework, demonstrating its workability and expressiveness; these include both geometric models as well as models of biological systems. New models are also shown, including a model of turtle geometry on the surface of a 2D mesh and a three-dimensional model of the apex of the moss *Physcomitrella patens*.

Word of Thanks

During the seven years (and longer) that cell complexes have been percolating through my head, many people have supported me both personally and professionally. I would like to single out just a few here.

First, my thanks to the current and past members of the Biological Modeling and Visualization research group, for their comments on my work and conversation about biology, graphics, mathematics, and every other topic under the sun. Thanks to Pierre Barbier de Reuille, whose advice led to the foundational place of relative orientation in the CCF. My special thanks to Adam Runions, who dropped in to my office every day to drop a bug in my ear about a new and fascinating mathematical or computational problem that could surely be solved with only a few hours or days or weeks of work. Thanks also to Richard Smith, whose comments on this thesis were only slightly less valuable than his gentle pressure to finish it.

Thanks to our biological collaborators, especially Drs. Jill Harrison and Elizabeth Barker; I'm not sure how good a student I was, but I learned a lot about plant structure. Thanks to the external members of my defense committee, Drs. Leah Edelstein-Keshet and Elena Braverman, for their rigorous examination and their helpful comments on my work and this thesis. Thanks to my supervisory committee for all of their help and guidance: Dr. Faramarz Samavati inducted me into the world of geometric modeling, while Dr. Eric Mjolsness' mathematical suggestions have been invaluable. It was an early email exchange with Eric that led to my discovering the beauty of the flip structure.

The greatest of thanks have to go to my supervisor, Dr. Przemyslaw Prusinkiewicz. It was his enthusiasm for science that brought me back to research after finishing my Masters degree. His wide range of knowledge and interests kept me busy while I worked as his research associate. During my work on cell complexes and the CCF, he was always able to identify where a concept was uncertain or a line of reasoning was murky, and his advice always kept me on track. Przemek is my mentor and my friend, and he has my deepest gratitude always.

Finally, I have to thank my family, especially my parents. My return to academia meant that they had to put up with their professional student for longer than expected, but they supported me all the way. Thank you.

Contents

List of Figures

List of Algorithms

List of Terms

adjacent In an n-dimensional cell complex, two n-cells c_1 and c_2 are adjacent (or *neighbouring*) if they share an $(n-1)$-dimensional boundary cell. More generally, we say that any two k-cells are adjacent if they share both a $(k-1)$-dimensional boundary cell and a $(k+1)$-dimensional coboundary cell.

atlas An atlas is a set of coordinate frames on a manifold such that at least one of the frames applies at every point on the manifold.

boundary An n-cell c in a cell complex is homeomorphic to a closed n-dimensional ball. The boundary of c is the set of $(n-1)$-cells which, taken together, are homeomorphic to the surface of the ball. The boundary of c is denoted ∂c. Furthermore, we write $c_1 \prec c_2$ when c_1 lies within the boundary of c_2.

boundary chain The boundary chain of a k-cell c is a chain that assigns a value of +1 to a $(k-1)$-cell in the boundary of c if it is positively oriented with respect to c, and a value of −1 if it is negatively oriented with respect to c.

CCF The Cell Complex Framework is a C^{++} API for developmental modeling using cell complexes.

cell complex An n-dimensional cell complex is a collection of mathematical n-cells along with all of the $(n-1)$-cells which make up their boundaries, the $(n-2)$-cells which make up *those* boundaries, and so on down to 0-cells.

cell tuple A cell tuple is a sequence (tuple) of incident cells, one of each dimension from 0 to the maximal dimension of the cell complex.

chain A chain is an assignment of a value to every cell in a cell complex. It is expressed as a linear combination of cells. For example, the chain $c_1 + 2c_2 - c_4$ assigns the value 1 to cell c_1, 2 to cell c_2, −1 to cell c_4, and zero to all other cells in the complex.

coboundary The coboundary of a k-cell c_k is the collection of $(k+1)$-cells in whose boundary c_k lies. Equivalently, a $(k+1)$-cell c_{k+1} is in the coboundary of c_k if $c_k \prec c_{k+1}$.

dart A dart is a simplex of maximal dimension in the combinatorial map. The vertices of a dart are a set of incident cells, one of each dimension from 0 to the maximal dimension of the cell complex.

embedding An embedding of a n-dimensional cell complex $\mathbb{C}$ is an assignment of a geometric interpretation to every cell in $\mathbb{C}$. An embedding relates $\mathbb{C}$ to a subdivision of the n-dimensional manifold in which the geometry is defined.

facet The facets are the two cells of equal dimension in a flip; the facets of the flip $\langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1}\rangle$ are c_k and c'_k.

flip A k-flip is a relation between four cells: one of dimension $(k-1)$, two of dimension k, and one of dimension $(k+1)$. The flip $\langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1}\rangle$ states that the involution σ_k applied to any cell tuple of the form $(c_0, \dots, c_{k-1}, c_k, c_{k+1}, \dots, c_n)$ results in the unique cell tuple $(c_0, \dots, c_{k-1}, c'_k, c_{k+1}, \dots, c_n)$ and vice versa. An alternative notation for the flip $\langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1}\rangle$ is

$$\left\langle \begin{matrix} & c_{k+1} & \\ c_k & & c'_k \\ & c_{k-1} & \end{matrix} \right\rangle.$$

flip operation Applying the flip operation **flip**(k) to the cell tuple $(c_0, \dots, c_{k-1}, c_k, c_{k+1}, \dots, c_n)$ returns the unique cell tuple $(c_0, \dots, c_{k-1}, c'_k, c_{k+1}, \dots, c_n)$, where the flip $\langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1}\rangle$ is one of the flips defining the cell complex.

G-map A G-map is a topological representation of a cell complex based on the combinatorial map and using a collection of darts as the primary data structure (Lienhardt 1994).

incident Two cells c_1 and c_2 are directly incident if one of them lies within the boundary of the other. The relation of incidence is the reflexive and transitive closure of direct incidence. We say that $c_1 \prec c_2$ if c_1 and c_2 are incident and the dimension of c_1 is less than the dimension of c_2.

interior The interior is the highest-dimensional entry in a flip; in the flip $\langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1}\rangle$, the interior is cell c_{k+1}.

joint The joint is the lowest-dimensional entry in a flip; in the flip $\langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1}\rangle$, the joint is cell c_{k-1}.

manifold A manifold is a mathematical object locally homeomorphic to Euclidean space.

mathematical cell A mathematical k-cell is a topological entity homeomorphic to a closed k-dimensional ball. A k-cell is defined by the set of $(k-1)$-cells which make up its boundary.

membrane When splitting a k-cell c into two child k-cells c_L and c_R, the membrane is the new $(k-1)$-dimensional cell which separates the two child cells.

neighbourhood The neighbourhood of a k-cell c is the set of all k-cells adjacent to c.

neighbouring See *adjacent*.

relative orientation The relative orientation between a k-cell c_k and an incident $(k-1)$-cell c_{k-1} is a number equal to either $+1$ or -1.

simplex A simplex is the generalization of the notion of a triangle or tetrahedron to n dimensions. Each k-simplex is uniquely defined by a set of $(k+1)$ vertices.

vv vv is a modeling system based on the vertex-vertex algebra and graph rotation systems (Smith 2006).

Part I

Cell Complexes and the Cell Complex Framework

1 Introduction

This dissertation deals with computer simulation of dynamic processes, of change through space and over time. Change over time has been extensively studied, from discrete-system simulation to difference equations to numerical and analytic solution of differential equations. Change through space, however, has not been studied nearly so thoroughly. Simulating change over space is not unknown, of course: we can analytically solve partial differential equations, or convert them into a set of ordinary differential equations coupled through space. Such techniques involve discretizing space, just as we discretize time.

A discretization imposes a structure on space. Each discrete element represents a piece of space. Moreover, the discrete adjacencies of the elements reflect the topological relationships within space. This means that the discretization defines a *discrete topological space*. (For the purposes of this thesis, a discrete topological space is a set X of objects, where each object x has a *neighbourhood* $N(x) \subseteq X$. The elements of $N(x)$ are said to be *adjacent* to x.[1]) These discrete elements of space can then carry information and computation according to both the subdivision of space they represent and to the neighbourhood relationships between them. In converting a partial differential equation into a set of coupled ordinary differential equations, for example, we assign parameters to each element, assumed to be valid on the subdivision that element represents; the parameters change according to ODEs that are coupled with the corresponding ODEs in the adjacent elements.

We might adaptively change the discretization of time, for example making it finer as a differential equation becomes stiffer. In the same way, we must be able to change the discretization of space as the needs of the simulation dictate. This may be adaptive, subdividing space to obtain numerically more precise results near a region of interest. It may be structured by the problem, subdividing the space representing a plant cell when that cell divides. Even if the discretization itself does not change, the topological space might; the fracturing of a material alters the neighbourhood relations between elements.

Many processes can be simulated by creating a dynamic subdivision of space. The finite element method (Braess 2007) can solve a large family of partial differential equations over space by subdividing that space into elements. Some applications of the finite

[1] A discrete topological space is distinct from the *discrete topology*, in which $N(x) = \{x\}$.

element method adaptively refine this subdivision for improved accuracy (e.g. Federl (2002)). In the field of geometric modeling, a polyline or mesh is a discrete subdivision of a curve or surface, and changes dynamically when the representation is refined by subdivision (Smith, Prusinkiewicz, and Samavati 2003).

In processes involving multicellular development, a dynamic subdivision of space is inherent in the problem. The modeling of growing multicellular structures is of fundamental importance in investigating plant development. In creating the framework described in this dissertation, I am guided by Prusinkiewicz's (2009) concept of *developmental computing*, which draws inspiration from the computational modeling of plant development. Prusinkiewicz extends ideas from the field of plant development into other fields which share the same characteristics:

1. A plant is a *dynamic system with a dynamic structure*: it consists of discrete components (modules), the number and states of which change over time.
2. Development takes place in a *topological space*, which determines the neighboring relations between components.
3. The development is *symplastic*: the neighborhood relations can only be changed as a result of the addition or removal of components (in contrast to animal cells, plant cells do not move with respect to each other).
4. The state of each component advances over time as a function of its own state and the state of the neighboring components, i.e., according to *temporally and spatially local* rules or equations.
5. There is *feedback* between the topology of the system and the state of its components. On one hand, changes in topology (the addition or removal of components and connections between them) are controlled by the state of the components. On the other hand, the state of each component depends on the input from its neighbors.

These characteristics are familiar from the discussion above: the spatial subdivision defines a topological space for simulating a dynamic process and may change over time. One addition is symplasticity (item 3); this seems to limit how the subdivision can change, but in fact any change to the subdivision can be captured by a series of symplastic changes (Rosenfeld and Strong 1969). A more important addition by developmental computing is the requirement that the rules governing behaviour be *local* (item 4). The locality requirement means that the behaviour of spatial elements is *modular*: it depends only on the element's own history and neighbourhood. Thus, calling the elements *modules* in item 1 is justified. Requiring that rules be local might seem to be overly limiting; a large number of dynamic processes, however, feature local behaviour, including mechanical and chemical systems, biological processes, and geometric subdivision.

A key requirement of the framework developed in this dissertation is to represent the topological structure of a dynamic spatial subdivision. The representation I use is the *cell complex*. A cell complex is a mathematical structure that can represent a subdivided space; its key feature is that not only are there subdivisions, or *cells*, of the same dimension

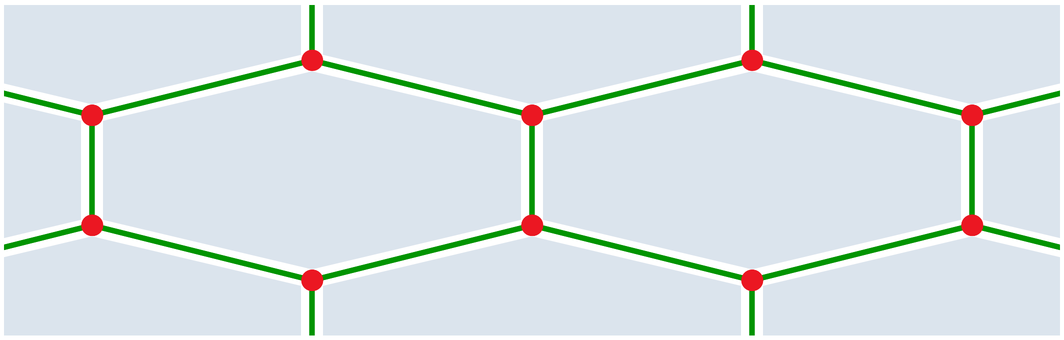


Figure 1.1: A two-dimensional cell complex representing a biological tissue. Two-dimensional mathematical cells, or “2-cells” (blue), represent biological cells; 1-cells (green) form the boundary of 2-cells and represent biological cell walls; 0-cells (red) form the boundary of 1-cells and represent the junctions between biological cell walls.

as the space represented, but there are also cells representing the boundaries between them (of one lower dimension), the boundaries between *these* (one dimension lower still), and so on. Thus, for instance, a two-dimensional representation of a biological tissue (Figure 1.1) contains two-dimensional mathematical cells representing biological cells,[2] one-dimensional mathematical cells representing biological cell walls, and zero-dimensional mathematical cells representing the junctions between biological cell walls.

In order to explore how cell complexes are a good fit to the class of problems considered within developmental computing, in the remainder of this chapter I will discuss the simplest case: *developmental computing with cell complexes in one spatial dimension*. Through the development of some one-dimensional examples, I will show the advantages of the cell complex approach: first, that they are a natural data structure for representing physical quantities that are inherently of different dimensions, such as “position”, “flux”, or “pressure”; and second, that a topological representation built on cell complexes is inherently modular and lets us refer to neighbours and divide cells locally.

1.1 Diffusion

The first one-dimensional example is a model of *diffusion*. Diffusion is a physical process in which random molecular motion results in a flow of a substance within a medium from areas of high concentration toward areas of low concentration; see, for example, Crank (1975). Despite the randomness of the motion of any particular molecule, the change in the distribution of the substance is coordinated; the amount of substance leaving some region of the medium is on average proportional to the concentration of substance within that region in the first place (Figure 1.2). The net flux J across some notional boundary is the difference between the amount of substance entering a region across the boundary and the amount of substance leaving across this boundary, and is thus proportional to

[2]The word “**cell**” can refer to a biological entity or a mathematical structure; as these uses are longstanding within both biology and mathematics, in this dissertation I will use the word with both meanings. Hopefully the particular meaning will usually be clear from context; in other cases, I will explicitly distinguish between “**mathematical cells**” and “**biological cells**”.

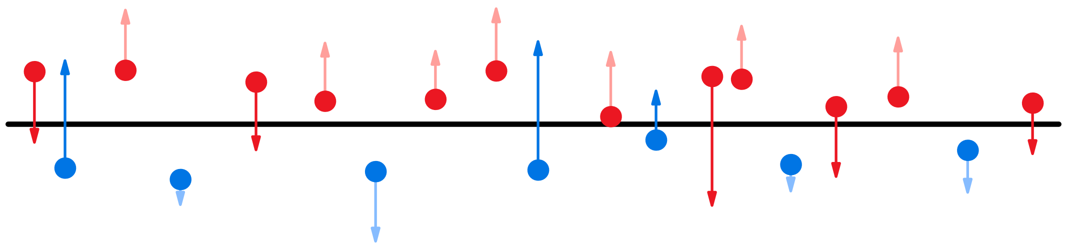


Figure 1.2: Random motion of particles across a notional boundary between two areas, one of high concentration (red particles) and one of low concentration (blue particles). The number of particles leaving the high concentration area is greater than the number of particles entering it.

the difference in the concentration c of the substance on either side of the boundary:

$$J_{i\to j} \propto c_i - c_j. \tag{1.1}$$

In a continuous space, the difference between (infinitesimally) adjacent regions is the gradient, and the flux at a point is given by *Fick's law*:

$$\vec{J}(x) \propto \nabla c(x), \tag{1.2}$$

where $c(x)$ is the concentration of the substance at position x. The accumulation or depletion of the substance at any point is the divergence of the flux, so Fick's law gives us the *diffusion equation*:

$$\frac{\partial c}{\partial t} = D\,\nabla^2 c, \tag{1.3}$$

where the coefficient of proportionality D is called the *diffusion coefficient*.

Here, however, we will consider diffusion within an inherently discrete structure, a *filament* of uniformly-sized biological cells numbered $\{\dots, i-1, i, i+1, \dots\}$. We assume that the concentration c of the substance is approximately constant within each cell (Figure 1.3); this is reasonable if the cell walls pose the greatest barrier to diffusion within the filament. Instead of re-discretizing Equation 1.3, we proceed directly from Equation 1.1; if we use k_i for the coefficient of proportionality for diffusion between cells i and $i+1$, then

$$J_{i\to i+1} = k_i\,(c_i - c_{i+1})\,.$$

Because the change of concentration in cell i is the sum of the flux entering on the left and the flux leaving on the right,

$$\frac{dc_i}{dt} = J_{i-1\to i} - J_{i\to i+1},$$

we see that

$$\frac{dc_i}{dt} = k_{i-1}\,(c_{i-1} - c_i) - k_i\,(c_i - c_{i+1})\,. \tag{1.4}$$

This is a family of equations, one for each cell except for the cells at the left and right sides of the filament, with indexes 1 and N; with only one neighbour each, the equations for these cells are

$$\frac{dc_1}{dt} = -k_1\,(c_1 - c_2) \qquad \text{and} \qquad \frac{dc_N}{dt} = k_N\,(c_{N-1} - c_N)\,.$$

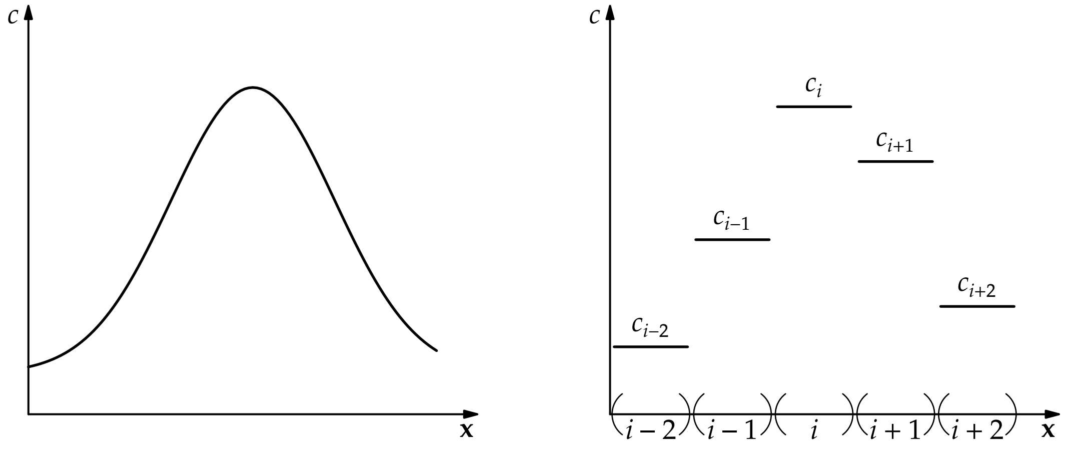


Figure 1.3: Plots of the concentration of a diffusing substance in one dimension. Left: The concentration varies continuously along a continuous filament. Right: The concentration along a filament of cells is approximately constant within each cell and changes discontinuously on the boundaries between cells.

This family of equations is all we need to simulate the diffusion of the substance within the filament. There are a couple of points to make, however, regarding the form of these equations: their *repeated terms* and their use of *indices*. In addressing these points, we will see how one-dimensional cell complexes form a natural structure for this diffusion problem.

1.2 Repeated terms

The first point to take note of in Equation 1.4 is the repeated terms; not in any single equation itself, but in the equations for neighbouring cells:

$$\frac{dc_i}{dt} = k_{i-1}(c_{i-1} - c_i) - k_i(c_i - c_{i+1}) \tag{1.5a}$$

$$\frac{dc_{i+1}}{dt} = k_i(c_i - c_{i+1}) - k_{i+1}(c_{i+1} - c_{i+2}). \tag{1.5b}$$

The quantity $k_i(c_i - c_{i+1})$ represents the substance *leaving* cell i and the substance *entering* cell $i+1$, so it makes sense that it appears in both equations. The question that immediately arises is *when* do we actually compute this quantity? At first sight, computing this value twice, once to calculate dc_i/dt and once to calculate dc_{i+1}/dt, may seem merely redundant, or possibly inefficient. Unfortunately, there is a deeper problem with evaluating this value twice.

Nothing has yet been said about the diffusion coefficients k_i; in fact, these can be constants, or depend on time, position, or concentration. Remembering that diffusion is caused by random molecular motion, it is reasonable that they can also be *random variables*; indeed, constant diffusion coefficients are only a large-number approximation, and random coefficients are to be expected if the number of diffusing molecules is small (Gillespie 1976). But if k_i is a random variable and the term $k_i(c_i - c_{i+1})$ is calculated twice (once in Equation 1.5a, once in Equation 1.5b), the values will likely be different. This

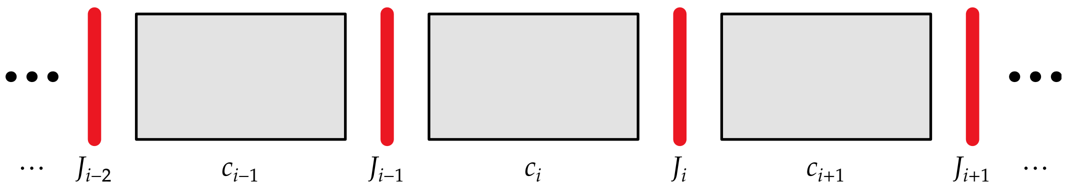


Figure 1.4: Representing a filament by interleaved cells (black) and walls (red). The concentration c is a property of cells, while the flux J is a property of walls.

means that the computed amount of the substance leaving cell i for cell $i+1$ will differ from the computed amount of substance entering cell $i+1$ from cell i. This contradicts the conservation of mass by non-physically creating or destroying the substance.

The basic problem with the repeated term is that we are computing its value both when changing the concentration c_i and the concentration c_{i+1}; that is, when updating both cell i and cell $i+1$. However, it is clear that the amount of substance travelling between cells i and $i+1$ is *not* a property of either cell; instead, it is a property of the *interface* between them. This means that a sequence of cells alone is not an adequate representation for the diffusion problem; we also need to consider the interfaces, leading us to a sequence of cells *separated by walls* (Figure 1.4).

Each equation 1.4 will then be replaced by two equations, one describing the change of flux through walls and one describing the change of concentration in cells:

$$\begin{aligned} J_{i-1\to i} &= k_{i-1}\,(c_{i-1} - c_i) \\ \frac{dc_i}{dt} &= J_{i-1\to i} - J_{i\to i+1} \end{aligned} \tag{1.6}$$

Equations 1.6 computes fluxes and changes in concentrations in a manner similar to Equation 1.4, but the computation of each flux is carried out only once, so mass is conserved even if the coefficients k_i are random variables.

By explicitly incorporating walls in our description of diffusion, we have a place to compute and store all of the important quantities: concentration in cells, and flux in walls. Introducing the (zero-dimensional) boundaries between (one-dimensional) cells turns the representation into a *cell complex*, and we can now see one significant advantage of cell complexes in developmental modeling: that physical quantities representing a problem have *different inherent dimensionalities*, and cell complexes offer places to store these quantities.

1.3 Indices

Three cells are referred to in Equation 1.4: cells $i-1$, i, and $i+1$. These indices describe an adjacency between the cells: a cell i has two neighbouring cells, cell $i-1$ and cell $i+1$. This neighbour relationship, where each cell has one neighbour in the "positive" direction and one neighbour in the "negative" direction, defines the one-dimensional topological space of the diffusion model. Equations 1.4 implicitly define the topological space through *index arithmetic*; the cells of the filament are numbered 1 through N, and a cell with index $j \in \{2, 3, \ldots, N-1\}$ has one neighbour in the negative direction, with index $j-1$, and one

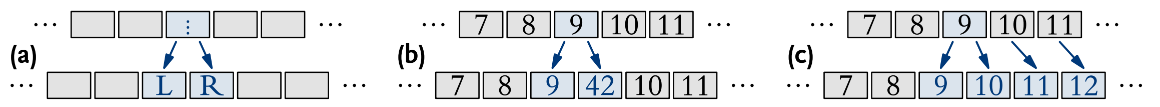


Figure 1.5: (a) The change in neighbourhood relationships when a cell in a filament divides. (b) Giving a new index to a child cell breaks index arithmetic. (c) Index arithmetic can be maintained by a non-local operation: renumbering all cells to the right.

neighbour in the positive direction, with index $j + 1$. The use of indices in this way is nearly ubiquitous in mathematics; for example, Crank (1975), in discussing solutions to the diffusion equation, introduces indices representing neighbourhood relationships without comment.

The use of indices, however, introduces two properties to the topological space which are *not* inherent in the problem of diffusion in a filament. The first is that the use of index arithmetic to find neighbours means that repeated arithmetic finds neighbours of neighbours. This means that cell $(i + 4)$ is always four cells to the right of cell i. The problem can be seen as soon as cells are allowed to *divide*. When a cell divides in two, its child cells each inherit one neighbour (Figure 1.5a); the "negative", or *left* child has the parent cell's *left* neighbour, while the "positive", or *right* child has the parent's *right* neighbour. The child cells are also neighbours of each other; the *left* cell is the *left* neighbour of the *right* cell, and vice versa. Now, if the cells are all numbered, what numbers do the child cells receive? One option is to give them new numbers, or possibly to have one keep the parent's number and the other a new number (Figure 1.5b). This choice, however, breaks index arithmetic; we can no longer add 1 to find the cell to the right, or subtract 1 to find the cell to the left, and cell $(i + 4)$ is no longer four cells to the right of cell i. The other option is to maintain index arithmetic by renumbering other cells. One way to do this is to give the *left* child the parent's old number, then reassign numbers to the remaining cells to the *right* (Figure 1.5c). The index arithmetic still defines the topology; however, the cost is that the division operation is no longer *local*, since cells even far to the right of the division must be renumbered. This contravenes the *locality* requirement of developmental computing.

If we don't use index arithmetic, how else can we describe the topological space? In one dimension, this is particularly easy: we can use a *string*. For the diffusion problem, we can use a *parametric string* (Prusinkiewicz and Lindenmayer 1990; Hanan 1992), a sequence of modules of the form $\boldsymbol{Cell}(c)$, where c is the concentration. The filament itself would then be represented by a string of the form

$$\cdots \boldsymbol{Cell}(c)\ \boldsymbol{Cell}(c)\ \boldsymbol{Cell}(c) \cdots .$$

In Section 1.2, we introduced a cell complex representation by explicitly considering walls between the cells. In the string, we can represent a wall by a module $\boldsymbol{Wall}(J)$. The entire string will then have the form

$$\cdots \boldsymbol{Wall}(J)\ \boldsymbol{Cell}(c)\ \boldsymbol{Wall}(J)\ \boldsymbol{Cell}(c)\ \boldsymbol{Wall}(J) \cdots .$$

The second of the two unnecessary properties introduced by indices is introduced by the topological space of a string as well. This is the fact that one of the neighbours of a module has been identified as the "left" neighbour, while the other is the "right" neighbour. This in turn implies a left-right orientation of the entire filament. An implied orientation of the space is also introduced in two-dimensional representations such as vv (Section 2.1.4), but can be avoided using cell complexes.

With a string representation of the filament, we are ready to write a purely local, index-free version of Equation 1.4.

1.4 L-systems

To operate on the string representing our filament, we will use a *context-sensitive L-system* (Prusinkiewicz and Lindenmayer 1990). An L-system provides rules (*productions*) of the form

$$\boldsymbol{\ell}_m \cdots \boldsymbol{\ell}_1 < \boldsymbol{a} > \boldsymbol{r}_1 \cdots \boldsymbol{r}_n \rightarrow \boldsymbol{b}_1 \cdots \boldsymbol{b}_k.$$

This production states that module $\boldsymbol{a}$ (the *strict predecessor*), when found in the string with substring $\boldsymbol{\ell}_m \cdots \boldsymbol{\ell}_1$ to its left and substring $\boldsymbol{r}_1 \cdots \boldsymbol{r}_n$ to its right, is replaced by the *successor* string $\boldsymbol{b}_1 \cdots \boldsymbol{b}_k$. Each module in the string is rewritten by the first applicable production, and all of the applicable productions are used *in parallel*.

Because of this parallel application of productions, a *derivation step* is unambiguously defined; each step can then be considered as advancement in time (Lindenmayer 1968). This means that, just as the symbols $<$ and $>$ denote neighbourhood in space, $\rightarrow$ denotes neighbourhood in time; any L-system production is thus local in both.

Assuming sufficiently small time steps Δt, Equations 1.6 can be solved numerically by the forward Euler method:

$$\begin{aligned} J^t_{i-1\rightarrow i} &= k^t_{i-1}\left(c^{t-1}_{i-1} - c^{t-1}_i\right) \\ c^t_i &= c^{t-1}_i + \Delta t\left(J^t_{i-1\rightarrow i} - J^t_{i\rightarrow i+1}\right) \end{aligned} \tag{1.7}$$

which has a clear translation into productions:

$$\textbf{\textit{Cell}}(c_L) < \textbf{\textit{Wall}}(J) > \textbf{\textit{Cell}}(c_R) \rightarrow \textbf{\textit{Wall}}\left(k\left(c_L - c_R\right)\right) \tag{1.8a}$$

$$\textbf{\textit{Wall}}(J_L) < \textbf{\textit{Cell}}(c) > \textbf{\textit{Wall}}(J_R) \rightarrow \textbf{\textit{Cell}}\left(c + \Delta t\left(J_L - J_R\right)\right). \tag{1.8b}$$

Because Production 1.8b depends on the current flux values, these productions must be applied in two alternating phases: first, Production 1.8a is applied to the entire string to calculate the fluxes, then Production 1.8b is applied to calculate the new concentrations.

A sample run of a simulation using these two productions is shown in Figure 1.6. Starting from an initially unbalanced distribution of the substance, diffusion rapidly spreads the substance over the entire filament.

1.5 Heterocyst differentiation in *Anabaena*

The diffusion simulation shown in Figure 1.6 has a fixed number of cells. To see how to adapt the simulation to dividing cells, in this section I consider a simulation of the development of the cyanobacterium *Anabaena catenula*.

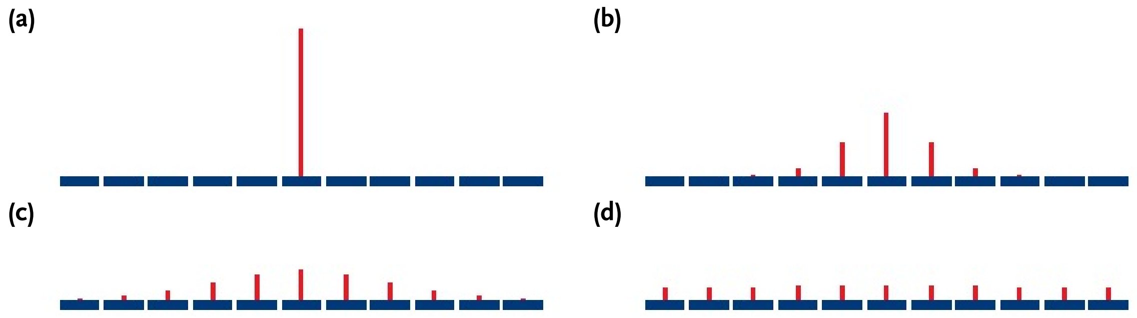


Figure 1.6: Sample run of a diffusion simulation using Productions 1.8a and 1.8b. A filament of blue cells extends from left to right. The concentration in each cell is indicated by a red vertical line. (a) Starting from an initial concentration of 10 units of the substance in the middle cell and none in the rest of the cells, the substance diffuses out to the entire filament. (b) Time $t = 0.5$. (c) Time $t = 2$. (d) Finally, after some time ($t = 10$), the substance is evenly distributed. For this simulation, the diffusion coefficient k is 1 everywhere.

Anabaena is a simple organism that grows in multicellular filaments containing two types of cells. *Vegetative* cells are capable of photosynthesis and produce sugars. *Heterocysts* fix nitrogen from the atmosphere to produce the nitrogen compounds needed for growth. Nitrogen fixation can only take place in the absence of oxygen, which is a product of photosynthesis, so *Anabaena* separates the two processes by restricting them to different cells (Haselkorn 1978). Heterocysts appear in the filament at regular intervals; they cannot divide, but vegetative cells grow and divide, causing the filament to elongate. New heterocysts differentiate when the dividing vegetative cells need nitrogen to grow further but are too far from the existing heterocysts. This differentiation is controlled by the diffusion of a small protein called PatS which is produced by the heterocysts and inhibits the differentiation of vegetative cells into heterocysts (Yoon and Golden 1998). As the vegetative cells divide and existing heterocysts move further apart, the concentration of PatS in the vegetative cells far from the heterocysts gradually decreases and eventually drops below a threshold in a cell near the center of the segment. This drop triggers the genetic mechanism which causes the cell to differentiate into a heterocyst.

A simple L-system which illustrates this process is given in Algorithm 1.1. The values of various parameters are given in the **#define** section at the top; note that the diffusion coefficient k in this example is a random variable. The **Axiom** is the initial string: a filament of three ***Cell***s separated by two ***Wall***s. The first and last cells are heterocysts, while the middle cell is vegetative. The ***Wall*** module is the same as in the diffusion example. The ***Cell*** module in this model has three parameters: the first is either H or V, indicating the cell type (heterocyst or vegetative); the second parameter s is the length of the cell, always 1 for heterocysts; and the third parameter c is the concentration of inhibitor in the cell.

The computation is divided into two alternating phases. In the first phase, production p_1 sets the flux in all ***Wall*** modules; this is the same as Production 1.8a from the diffusion example. In the second phase, extending Production 1.8b, the state of each ***Cell*** is altered. Productions p_2–p_5 use logical guards which limit the conditions under

Algorithm 1.1 L-system model of heterocyst differentiation in *Anabaena*.

```
#define  $H$          0          // Heterocyst cell type
#define  $V$          1          // Vegetative cell type
#define  $k$          ran(2.0)   // Diffusion coefficient
#define  $v$          0.5        // Inhibitor turnover rate
#define  $R$          1.1        // Cell expansion rate
#define  $\Theta$     0.1        // Threshold for heterocyst differentiation
#define  $s_{MAX}$    0.8        // Cell size at division
#define  $\Delta t$   0.01       // Time step

Axiom: $\mathbf{Cell}(H,1,1)\ \mathbf{Wall}(0)\ \mathbf{Cell}(V,1,1)\ \mathbf{Wall}(0)\ \mathbf{Cell}(H,1,1)$
phase 1:
$p_1$: $\mathbf{Cell}(a_L,s_L,c_L) < \mathbf{Wall}(J) > \mathbf{Cell}(a_R,s_R,c_R) \rightarrow \mathbf{Wall}(k \cdot (c_L - c_R))$
phase 2:
$p_2$: $\mathbf{Cell}(a,s,c) : a = H \rightarrow \mathbf{Cell}(H,1,1)$
$p_3$: $\mathbf{Wall}(J_L) < \mathbf{Cell}(a,s,c) > \mathbf{Wall}(J_R)$:
    $\{c \leftarrow c + ((J_L - J_R) - vc)\Delta t;\ s \leftarrow sR^{\Delta t};\}$
    $c < \Theta \rightarrow \mathbf{Cell}(H,1,1)$                       // Differentiate into heterocyst
$p_4$: $\mathbf{Wall}(J_L) < \mathbf{Cell}(a,s,c) > \mathbf{Wall}(J_R)$:
    $\{c \leftarrow c + ((J_L - J_R) - vc)\Delta t;\ s \leftarrow sR^{\Delta t};\}$
    $s > s_{MAX} \rightarrow \mathbf{Cell}(V,s/2,c)\ \mathbf{Wall}(0)\ \mathbf{Cell}(V,s/2,c)$       // Divide
$p_5$: $\mathbf{Wall}(J_L) < \mathbf{Cell}(a,s,c) > \mathbf{Wall}(J_R)$:
    $\{c \leftarrow c + ((J_L - J_R) - vc)\Delta t;\ s \leftarrow sR^{\Delta t};\}$
    $\rightarrow \mathbf{Cell}(V,s,c)$                                  // Expand
```

which they are applied.[3] Production p_2 has a guard $a = H$;[4] this means that it will only be executed if the cell is of type H, that is, a heterocyst. In this case, p_2 merely sets the size and concentration to 1. All heterocysts will be caught by this production, so productions p_3–p_5 will only affect vegetative cells.

The vegetative cell productions p_3–p_5 all have the same code block[5]

$$\left\{c \leftarrow c + ((J_L - J_R) - \nu c)\,\Delta t;\ s \leftarrow sR^{\Delta t};\right\},$$

which computes the new values of c and s. The computation of the new concentration takes place in the first statement in the block; the formula is the same as in Production 1.8a plus an additional term, representing the turnover of a fraction ν of all of the inhibitor in the cell. The second statement in the block computes the new size of the cell. According to this statement, each cell expands exponentially with rate R.

The productions now use the newly computed concentration and size to determine the fate of the cell. Productions p_3 through p_5 are tried in order; if a production's condition is not satisfied, the next production is tried, until p_5, which is applied to the remaining modules unconditionally. Production p_3 is applied if the concentration is less than the differentiation threshold Θ, and turns the vegetative cell into a heterocyst. Production p_4 is applied if the cell is larger than the size threshold s_{MAX}; if so, then it divides. The division replaces the single ***Cell*** with two new vegetative cells, each half the size of the old cell but with the same concentration of inhibitor. These cells are separated by a new ***Wall***. Finally, if neither of the previous cases applies, production p_5 simply updates the ***Cell*** module with the new concentration and size.

Figure 1.7 shows some steps of a simulation using this L-system. As the vegetative cells expand and divide, the heterocysts are pushed apart. The concentration of the inhibitor in the vegetative cells decreases with their distance from the heterocysts, and when the heterocysts become far enough apart, the threshold in the cell halfway between them drops below the differentiation threshold and it becomes a heterocyst. The average distance between the heterocysts is thus maintained despite the growth of the filament.

In the *Anabaena* model, we have easily adapted the diffusion productions 1.8a and 1.8b to a growing filament of cells. Because of the topological nature of the rules, which refer to cells only through neighbourhood relations, the form of the productions is the same in the growing filament as in a filament where the number and neighbourhoods

[3]These productions are written in the extended notation described in Měch (2005):

left context < ***predecessor*** > ***right context*** :
{ ***code block*** } ***guard*** → ***successor***

where the **code block** is a block of C-like code which can be used to compute new values based on the parameters of the modules, and the **guard** is a logical statement which may depend on parameters or new values. Both the code block and guard are optional. If the guard is present, the production will be applied only if it evaluates to **true**.

[4]Note that here, and in all Algorithms, the sign = denotes logical equality, as in **if**$(a = H)$, while assignment is denoted by $\leftarrow$, as in $c \leftarrow c + J\Delta t$.

[5]The repetition of this block can be avoided using, for instance, the L+C language; in that language, the conditional semantics described here can be implemented in a single production.

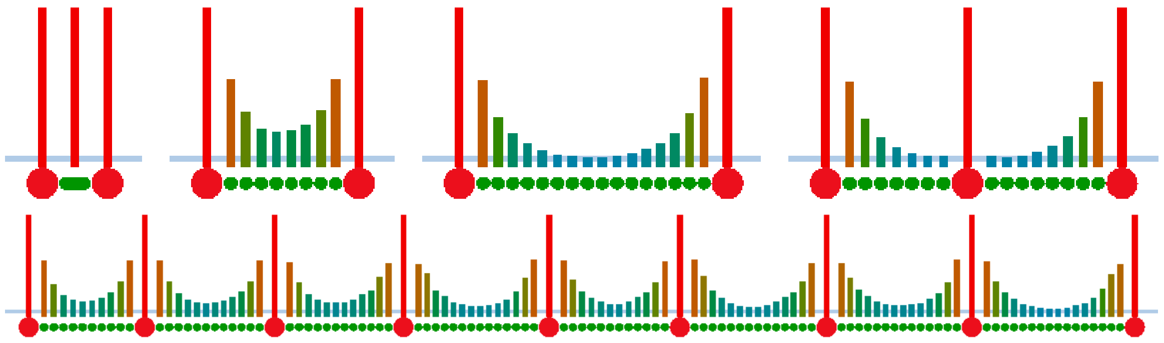


Figure 1.7: Snapshots of a simulation of heterocyst differentiation in a growing *Anabaena* filament. Non-differentiated vegetative cells are shown in green and heterocysts are shown in red. Vertical bars indicate concentration of a diffusing inhibitor produced by the heterocysts. The blue horizontal line indicates the threshold of heterocyst differentiation.

of cells does not change. The rule governing cell division, production p_4, is local in nature. The *Anabaena* example thus demonstrates the use of cell complexes for a complete developmental computing model.

1.6 Moving beyond one dimension

I have shown two examples of computing in one-dimensional cell complexes: simple diffusion in a filament, and heterocyst differentiation in *Anabaena*. Both work on a topological space modeled in an index-free manner with a one-dimensional cell complex. We have seen that this choice has several advantages for developmental modeling. First, cell complexes let physical quantities of different inherent dimension sit in their proper place in the structure, on mathematical cells of the corresponding dimension. In these models, concentration of a substance is a property of one-dimensional cells, while the flux of the substance is a property of zero-dimensional cells. Second, the index-free structure lets all of the operations, including cell division, remain inherently local. Third, neighbourhood relations let developmental rules easily access values in neighbouring cells.

Motivated by these advantages, in this dissertation I develop and explore the use of cell complexes as a modeling framework for problems in developmental computing. The text is divided into two parts: Part I deals with the theory and implementation of this modeling framework, while in Part II I show examples of its application to problems in geometric computing and biological modeling.

Part I continues in Chapter 2, where I discuss previous work in this area, focusing on systems which operate in more than one dimension. I also review data structures used to model cell complexes and similar structures in more than one dimension. In Chapter 3, I introduce the mathematics of cell complexes, both as an abstract topological space and with the geometric interpretation of the *divided manifold*. I then derive the *combinatorial map*, the powerful structure upon which my modeling framework is built. In Chapter 4 I describe the *Cell Complex Framework*; I discuss the algorithms underlying its atomic operations, the theoretical basis behind them, and their computer implementation.

In Part II, I present modeling with the Cell Complex Framework and show some specific developmental models created using it. In Chapter 5, I describe some basic techniques which are common to many models. These include methods for iterating over cells in the cell complex, for subdividing cells, and for computing and incorporating geometric information such as a cell's orientation, size, or centroid. The examples in this chapter also serve as an introduction to modeling with the Cell Complex Framework.

Chapters 6 and 7 cover the two problem domains of geometric modeling and biological modeling, respectively. In these chapters, I describe models which apply the Cell Complex Framework to these two domains. Most of these examples are taken from the literature and have been previously implemented in other developmental computing frameworks. My reimplementations with cell complexes serve to demonstrate the workability and expressiveness of the Cell Complex Framework. In addition, the power of CCF as a modeling system is illustrated using several original models. Specifically, in Chapter 6 I discuss the geometric modeling of surfaces and volumes, including global mesh subdivision in two and three dimensions, polygonization of implicit surfaces, and a method for finding geodesics on surfaces. In Chapter 7, I cover biological modeling with the Cell Complex Framework, including models of cell growth and division. I also present two models of the moss *Physcomitrella*: a two-dimensional model of its leaves, and a three-dimensional model of its apical development. Finally in Chapter 8 I discuss the current state of the Cell Complex Framework, modeling with cell complexes, and future directions in this area.

2 Previous Work

The work described in this chapter comes from two areas. First is previous work in developmental computing; in the broad sense defined in Chapter 1, this is a recent field, but in the sense of *computational models of plant development*, the field has been around for many decades. In the modeling of one-dimensional structures (including branching structures) the field has been dominated by L-systems, a variant of sequential grammars. The influence of L-systems has been so pervasive that even in more than one dimension, developmental models are often based on grammars or seen as higher-dimensional extensions of L-systems.

The second field of work described in this chapter is *data structures for cell complexes* and similar topological objects, such as polygon meshes. This work comes largely from geometric modeling and includes representations intended for different purposes, with various strengths and limitations. Special attention will be paid to data structures based on the *combinatorial map*, which the Cell Complex Framework described in this dissertation is built on.

2.1 Languages for developmental computing

2.1.1 One-dimensional grammars

The L-system formalism was introduced by Lindenmayer (1968), based on *sequential grammars* (Chomsky 1956) and *sequential machines* (Ginsburg 1960). Lindenmayer's original conception was of a number of cells, each of which was at in one of a number of states representing physiological or morphological conditions. In a discrete time step, each cell may change to a different state and produce one of a discrete set of *outputs*, representing diffusing chemicals or the like. Both the output and state change are deterministic, and the transition rules depend on the cell's previous state and outputs of the cell's neighbours.

In most of the models described by Lindenmayer (1968), the outputs are identified with the states; a cell's output is the same as its state, and so the transition rules are functions of the states of the cell and its neighbours. This is the interpretation which has been carried forward in work on L-systems. The first topological space — a set of objects and neighbourhood relations between them — described by Lindenmayer is that of the *string*. The cells are arranged in a sequence, and a cell's neighbours are those cells preceding and succeeding it in the sequence. This means that, as discussed in Section 1.4, the transition rules, or *productions*, can be written in the form

$$\boldsymbol{\ell}_m \cdots \boldsymbol{\ell}_1 < \boldsymbol{a} > \boldsymbol{r}_1 \cdots \boldsymbol{r}_n \rightarrow \boldsymbol{b}_1 \cdots \boldsymbol{b}_k.[1]$$

This rule states that a cell in state a (the *predecessor*), preceded by m cells in states $\boldsymbol{\ell}_m \cdots \boldsymbol{\ell}_1$ (the *left context*) and succeeded by n cells in states $\boldsymbol{r}_1 \cdots \boldsymbol{r}_n$ (the *right context*), can be replaced by k cells (the *successor*) in states $\boldsymbol{b}_1 \cdots \boldsymbol{b}_k$. The death of the predecessor cell is represented by an empty successor, while the division of the predecessor is represented by a successor of more than one cell. Lindenmayer (1968) also describes models with a tree topology, represented by a bracketed string; the form of productions is the same, with a branch being enclosed in brackets [].

The formalism of L-systems as described by Lindenmayer (1968) has been extremely successful in biological modeling, and has been used to model everything from cellular filaments (Prusinkiewicz, Hammel, and Mjolsness 1993) to herbaceous plants and trees (Prusinkiewicz and Lindenmayer 1990). In later formulations, each symbol, or *module*, does not necessarily represent a single biological cell. Modules could directly correspond to plant organs, such as leaves and flowers; or these could be represented by a number of modules, which divide its behaviour into different functions (Cieslak et al. 2011); modules might even represent subcellular elements, such as an interface between cells, as in the L-systems described in Chapter 1.

Most subsequent formalisms for developmental modeling have been influenced by L-systems and their success. Indeed, many have been explicitly formulated as an application of L-systems to multidimensional topological spaces.

2.1.2 Graph grammars

One early approach to rewriting systems on multidimensional structures is *graph grammars*. For instance, Pfaltz and Rosenfeld (1969) define a formalism for rewriting "webs": directed graphs with labelled vertices. In a string, the assignment of neighbourhoods to new modules is clear; the left neighbour of the old module is the left neighbour of the leftmost new module, and the right neighbour of the old module is the right neighbour of the rightmost new module. In a graph, however, reassigning neighbourhoods is ambiguous. Because of this, a production in a web grammar must include not only predecessor, context, and successor, but also an "embedding", a description of how to attach the old neighbourhood to the new modules. Pfaltz and Rosenfeld (1969) consider both acyclic and cyclic directed graphs; on acyclic graphs their embeddings are simple, and usually of the form "if there was an edge from a node x to predecessor node a in the old graph, there will be an edge from x to each of a's successors in the new graph". On directed graphs with cycles, the question of embedding is more complex, and Pfaltz and Rosenfeld do not answer it.

A related publication (Rosenfeld and Strong 1969) discusses in more detail the problem of embedding in "map grammars"; that is, grammars on planar graphs with a planar

[1]Older publications on sequential grammars use varying, inconsistent notation; for simplicity I use here the notation for productions described by Prusinkiewicz and Lindenmayer (1990).

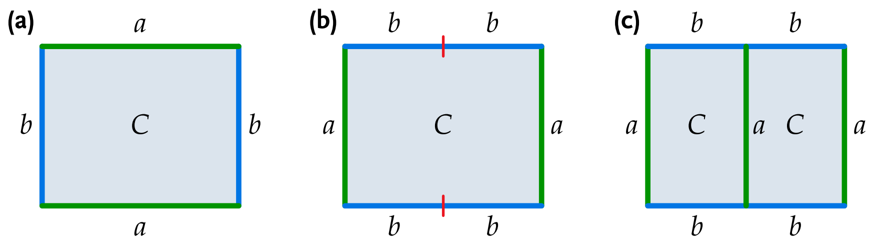


Figure 2.1: Cell division defined by a map L-system. (a) The initial state of the cell. (b) Walls have been altered according to the one-dimensional productions $\{a \rightarrow bb,\ b \rightarrow a\}$. (c) The cell C has been split according to the two-dimensional production $C \rightarrow ((bab, C), (bab, C), a)$.

embedding. To create an embedding, Rosenfeld and Strong (1969) use what they call the "map multigraph", which records at each vertex a cyclic ordering of the neighbouring vertices. This structure is more commonly called the *graph rotation system*; Edmonds (1960) showed that this information uniquely identifies the graph's planar embedding. The graph rotation system is a convenient local specification of the graph, and as such it is used in many grammars and developmental computing formalisms, in particular vv (Section 2.1.4). Rosenfeld and Strong (1969) demonstrate that in a split of one vertex into two, the vertex's cyclic neighbourhood must be split into exactly two parts which may only overlap on the endpoints; two adjacent vertices whose neighbourhoods have this structure can be merged. They also show that any graph alteration can be performed by a succession of binary splits and merges; the problem of the embedding of an arbitrary production is thus reduced to splitting and merging the cyclic neighbourhoods of a number of single vertex divisions.

Graph rotation systems only uniquely identify planar graphs, which are two-dimensional structures; in higher dimensions, another solution must be found. Mayoh (1974) pointed out that the surface of any polyhedral volume is itself a planar graph, and can thus be represented and manipulated by graph rotation systems, but did not pursue this further.

2.1.3 Multidimensional extensions to L-systems

A more direct two-dimensional successor to L-systems is provided by *map L-systems* (Lindenmayer and Rozenberg 1979). Map L-systems act on subdivisions of the plane; each two-dimensional division is a *cell*, and their boundaries are *walls*. The development of the system is broken into two phases: a one-dimensional step, in the style of L-systems, operating on walls, then a two-dimensional step which divides the cells. Walls are broken into segments, each represented by a symbol (Figure 2.1a); the one-dimensional step then modifies the walls using L-system productions (Figure 2.1b):

$$a \rightarrow bb$$
$$b \rightarrow a.$$

The two-dimensional step then describes the actual cell division. In this step, each cell is represented by a symbol. A production specifies explicitly how the cyclic neighbourhood of the cell is split by giving the wall sequence of each child cell along with the

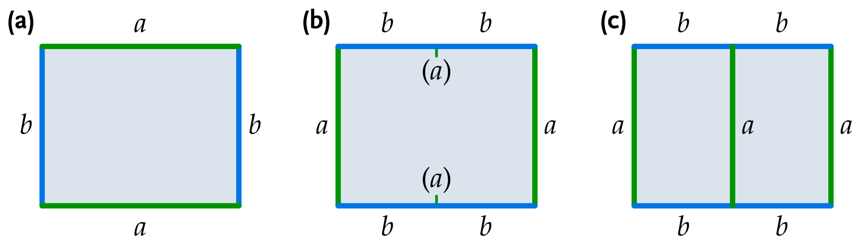


Figure 2.2: Cell division defined by a map L-system with markers. (a) The initial state of the cell. (b) Walls have been altered according to the one-dimensional productions $\{a \to b(a)b,\ b \to a\}$. (c) The markers (a) are joined automatically to divide the cell into two, separated by a new wall with symbol a.

child's symbol, as well as the wall sequence of the new division wall (Figure 2.1c):

$$C \to ((bab, C), (bab, C), a).$$

An extension of map L-systems to three dimensions (Lindenmayer 1984) adds a third phase, for the splitting of three-dimensional cells. Here, faces and cells do not have symbols; rather, they are specified by listing their boundaries. Two-dimensional faces are denoted by their cyclic bounding edges, while three-dimensional faces are defined by a multiset of face descriptions. A three-dimensional production may have the form

$$[(ABAB)^2, (BCBC)^2, (ACAC)^2] \to [(ABAB)^2, (BCBC)^2, ACAC, \underline{ACAC}]^2,$$

where a squared entity indicates two copies, and the underlined face is the new division wall.

A different version of map L-systems, the *edge-controlled formalism*, or *map L-systems with markers* (Nakamura, Lindenmayer, and Aizawa 1986), also privileges edges over cells. Again, cells do not have symbols, but cell division is now controlled entirely by the edge productions. This is done by adding new *marker* symbols, which indicate the endpoint of a division wall. A pair of markers on the wall of a cell are matched up automatically to divide the cell. The productions describing the cell division in Figure 2.2 are

$$a \to b(a)b$$
$$b \to a$$

where (a) is a marker for the endpoint of a division wall with symbol a.

Map L-systems, especially map L-systems with markers, are a powerful formalism for describing two-dimensional spaces of dividing cells. However, while they work well for specifying the division of regular arrangements of cells, they become less controllable and more complex as the shape of the cells becomes more irregular. Partly in response to this, *cell systems*, a cell-controlled formalism for dividing cells, were developed by de Boer, Fracchia, and Prusinkiewicz (1992). In cell systems it is cells, not edges, which are explicitly denoted by symbols and which control division. For example, a cell system production may have the form

$$C \to C\,|(90^\circ, 0.6)\,C.$$

The notation $|(90^\circ, 0.6)$ specifies a *division wall*; the first argument defines the angle the wall makes with an external vector field, the second gives the relative size of the two child cells. Thus, the described division takes place perpendicular to the vector field and creates two child cells which take up 60% (= 0.6) and 40% of the area of the parent cell. The external vector field may be constant or may depend on the cells themselves, for instance as the gradient of some substance produced by the cells. The reliance on the external vector field and the relative sizes of the child cells means that, unlike map L-systems, cell systems must have a defined geometry. In (de Boer, Fracchia, and Prusinkiewicz 1992), this geometry was provided by a physical simulation on the cell structure, which assigned pressures to cells and tensions to walls and brought the arrangement of cells into mechanical equilibrium before each production step. While the particular geometrical realization could be changed, this could not be described within the formalism of cell systems itself.

A different direction is taken by 3Gmap L-systems (Terraz et al. 2009). These also work by productions on cells, rather than walls, but the cells in question are three-dimensional. The three-dimensional topology of the cells is recorded as a cell complex using a *G-map*, a data structure I will discuss in more depth in Section 2.2.3. Despite using a data structure that allows access to an entire cell complex, 3Gmap L-systems only use the adjacencies between 3-dimensional cells.

In order to simplify productions, the shape of cells is restricted to prisms (Figure 2.3). Each prism has polygonal endcaps A_O and A_E, the "origin" and "end" faces, and rectangular side walls, denoted $\{A_{C1}, \dots, A_{Cn}\}$. These wall identifiers are used to describe adjacency; for example, the production

$$A : A_O < B_E \rightarrow C$$

means that cell A is relabelled C if its origin face is adjacent to the end face of a B cell. New cells arise in one of two ways: either an existing cell is split parallel to one of its faces (Figures 2.3a, b), or the face of an existing cell is extruded to form a new prism (Figure 2.3c). Productions can also join together two faces and create an adjacency (Figure 2.3d).

Geometrically, a prism has a specified size (height, width, and length) and rotation (yaw, pitch, and roll relative to its parent prism). Vertices which are shared between two or more prisms are placed at the average position of the corresponding vertices in "ideal" forms of those prisms. There is no mechanism to guard against self-penetration by non-adjacent prisms.

2.1.4 Two recent formalisms

Two recent formalisms for developmental computing in higher dimensions are MGS (Giavitto and Michel 2001) and VV (Smith, Prusinkiewicz, and Samavati 2003). A contrast of these two systems will highlight some of the ideas that went into the development of the cell complex framework.

MGS (Modèle Géneral de Simulation) is a general declarative programming language unifying computing in different topological spaces. Like L-systems, it is based on productions; the predecessor, however, can describe any pattern of adjacency between modules, while the successor is a transformation made on the matched pattern. The patterns are

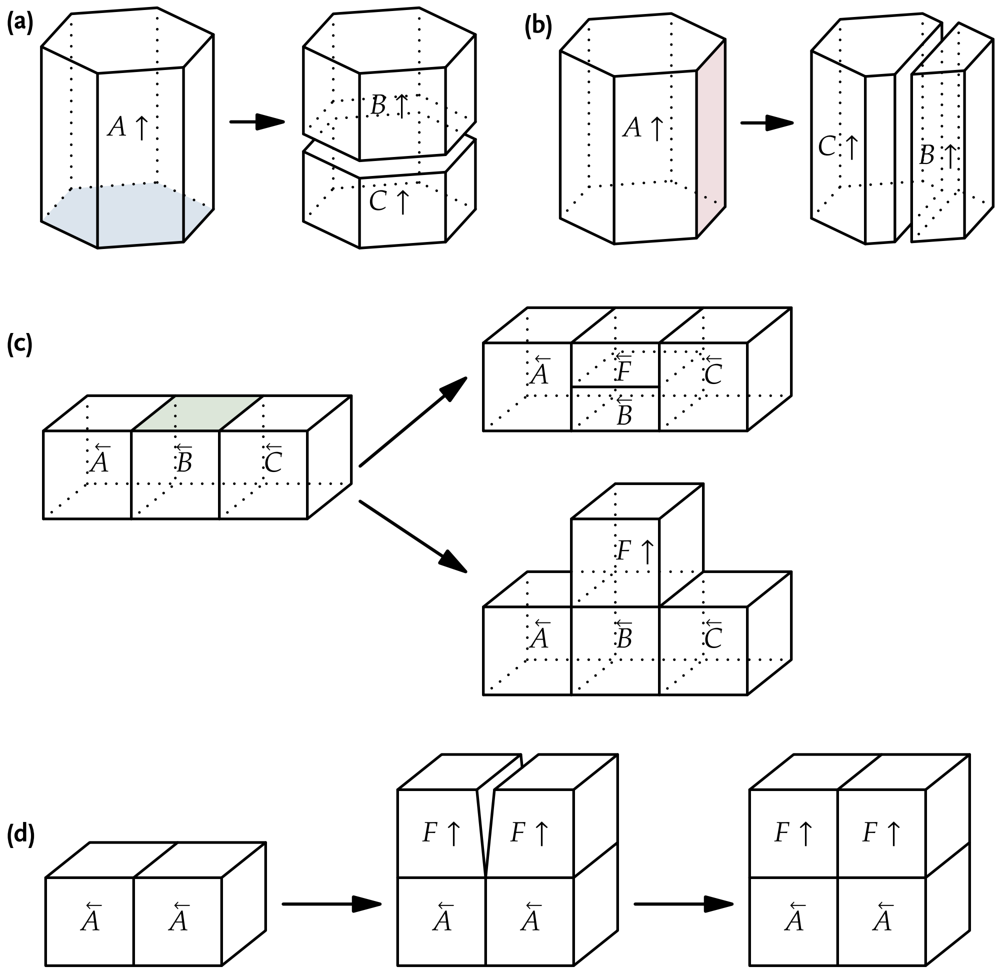


Figure 2.3: Operations on a three-dimensional structure specified in 3Gmap L-systems. Each cell is identified by a letter and has an orientation, from origin to end faces, indicated by an arrow. (a) A prism-shaped cell is split along a plane parallel to its origin face (blue) ($A \rightarrow^{O} BC$) (b) A cell is split along a plane parallel to a side face (red) ($A \rightarrow^{C1} BC$) (c) Having specified a face (green), a new cell can be produced either by splitting parallel to it (top, $B \rightarrow^{C1} BF$) or by extruding across it (bottom, $B \rightarrow B[F]_{C1}$) (d) After adding new cells across side faces ($A \rightarrow A[F]_{C1}$), the new cells can be made adjacent by identifying the walls ($F : F_{C1} < F_{C3} \rightarrow F_{C1}|F_{C3}$).

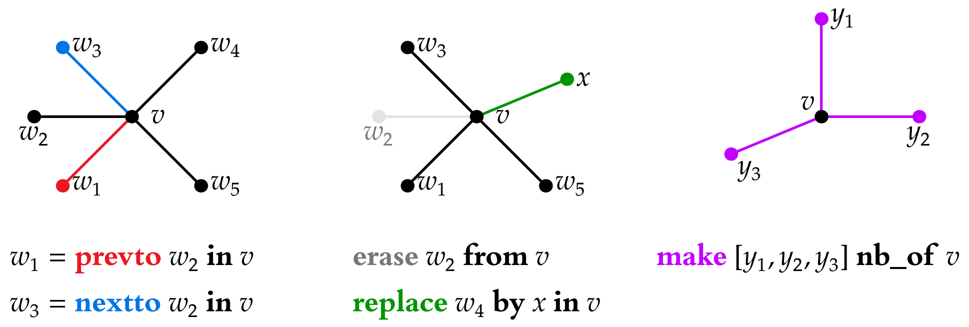

w_3
w_4
w_2
v
w_1
w_5
w_3
x
w_2
v
w_1
w_5
y_1
v
y_2
y_3
w_1 = **prevto** w_2 **in** v
w_3 = **nextto** w_2 **in** v
erase w_2 **from** v
replace w_4 **by** x **in** v
make $[y_1, y_2, y_3]$ **nb_of** v

Figure 2.4: Some of the operations defined in vv.

based on neighbourhood relations, but what "neighbourhood" means depends on the particular topological space used. Thus, the modules might be vertices in a graph, cells in a a grid, nodes in a tree, elements of a multiset, symbols in a string, and so on, with the corresponding definitions of "neighbourhood" between modules. A particular rule might be

$$x.y.z.x \to \mathbf{op}(x, y, z),$$

where the notation $x.y$ means that module x is adjacent to the module y. The pattern identifies all sets of three modules x, y, and z such that x is adjacent to y, y is adjacent to z, and z is adjacent to x, i.e. the three are mutually adjacent. The successor combines x, y, and z using some operator **op**. Used in a graph topology, this operation would be applied to all 3-cliques; on a hexagonal grid, to all triangular arrangements of three cells. In the topology of point sets with adjacency defined as adjacency of Voronoi regions, this pattern would identify all Delaunay triangles.

A supported topology of particular interest is the cell complex (Spicher and Michel 2007); in this case, the topological relations used are not mere adjacency, but incidence; that is, whether one cell lies in another's boundary. If v is in the boundary of e, then the relation is written $v \prec e$. The pattern matching a triangular face, for example, is

$$v_1 \prec e_{12} \succ v_2 \prec e_{23} \succ v_3 \prec e_{13} \succ v_1,$$
$$e_{12} \prec f, e_{23} \prec f, e_{13} \prec f$$

Unfortunately, the complexity of such pattern specifications increases as the dimensionality and the number of cells involved goes up. Consequently, to implement processes such as mesh subdivision in MGS, its creators have had to implement an interactive graphical editor explicitly for specifying these adjacency patterns (Jullian 2005).

Much of this difficulty comes from having to define adjacencies in a purely declarative manner. The vv modeling system avoids this by changing to an imperative formalism. This formalism is based on the *vertex-vertex algebra* (Smith, Prusinkiewicz, and Samavati 2003), which defines operations on a graph rotation system. Recall that a graph rotation system is based on the cyclic ordering of the neighbours of a vertex. The operations provided by vv provide an interface for the modeler to examine and modify these cyclic orderings (Figure 2.4). The examination operations include **nextto** and **prevto**, which make it possible to iterate around a vertex's cyclic neighbourhood; for instance,

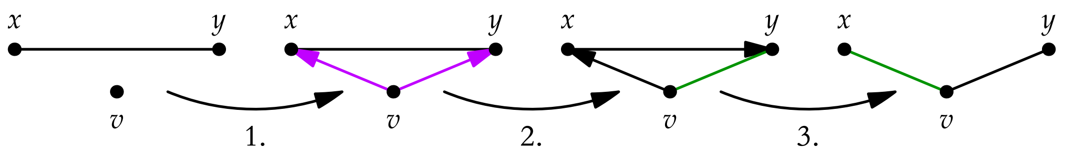


Figure 2.5: Inserting a new vertex v between vertices x and y is a three-step process in vv.

1. **make** $[x, y]$ **nbof** v sets the cyclic neighbourhood of v to $[x, y]$. As v is is not in the cyclic neighbourhood of either x or y, this one-sided relation is depicted by a directed edge.
2. **replace** x **by** v **in** y replaces the entry for x in y's cyclic neighbourhood by v; now the neighbourhood relation between v and y is mutual and the relations with x are one-sided.
3. **replace** y **by** v **in** x replaces the entry for y in x's cyclic neighbourhood by v. The neighbourhood relationship is again symmetric and the graph is consistent.

nextto x **in** v gives the vertex after x in the neighbourhood of v. The modification operations include **erase**, which removes a vertex from a cyclic neighbourhood; **replace**, which replaces one vertex with another in the cyclic neighbourhood; **splice**, which inserts a new vertex in the cyclic neighbourhood adjacent to a specified vertex; and **make nbof**, which replaces the existing cyclic neighbourhood with a new cyclic list.

Operations in vv can be combined into an imperative program manipulating the topological structure. For example, inserting a new vertex v between x and y is done by the sequence of operations shown in Figure 2.5:

$$\begin{aligned}&\textbf{make } \{x, y\} \textbf{ nbof } v\\&\textbf{replace } x \textbf{ by } v \textbf{ in } y\\&\textbf{replace } y \textbf{ by } v \textbf{ in } x.\end{aligned}$$

Using imperative operations, more complex manipulations are easier to specify in vv. Note, however, that the operations do not maintain the validity of the topology; after one or two of the operations depicted in Figure 2.5, the neighbourhood relationship is not symmetric, and it is not until the third operation that the topology becomes consistent again. Further, a graph rotation system gives an explicit representation only of vertices; in vv edges are represented as vertex pairs, and faces have no built-in representation at all. Finally, graph rotation systems and the vv algebra can only work in two dimensions; the theorem of Edmonds (1960) only applies to planar graphs, and a direct extension to three dimensions is impossible.

2.1.5 Other approaches

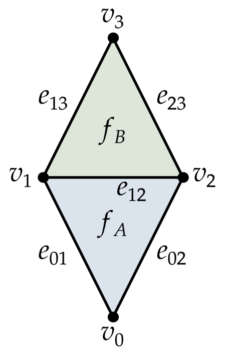
In addition to those describe above, several other environments that have been proposed or applied to model plant development should be mentioned here. The spatial dynamics of growth and development have been captured by a cellular Potts model of the root of *Arabidopsis* (Grieneisen et al. 2007). The topological entities in the cellular Potts framework are points on a regular grid, each of which records which *biological* cell it is a part of. Biological cell interactions are then handled at the grid points between cells, flagged as "cell walls". This approach implies that interactions between points within a cell are implemented at the same level as interactions between different cells. In other words, qualitatively different entities such as cells and cell walls do not have qualitatively different representations. Growth is modeled by changing the cell associated with a grid point as the growing tissue "flows" through space. A drawback of this technique in the context of plant modeling is that preventing relative motion of neighouring cells is quite difficult. For example, the cells in the model of Grieneisen et al. (2007) do not exhibit the symplastic growth characteristic of plant development; they can slide past one another and alter the topology of the cell arrangement during growth.

Two more explicitly cell-based frameworks are CellModeller (Dupuy et al. 2008) and VirtualLeaf (Merks et al. 2011), which both seek to provide multicellular spaces to examine models of gene function in the context of cell growth, division, and intercellular signalling. VirtualLeaf is the simpler of these two systems. The modeled two-dimensional structure consists of polygonal cells separated by straight line segment walls. The built-in geometric evolution of the structure is physically-based; the structure seeks a balance between matching each cell's preferred area with each wall's preferred length, similar to the physically-based model behind the geometry of cell systems (de Boer, Fracchia, and Prusinkiewicz 1992). The biochemical networks within and between cells are modelled with ordinary differential equations.

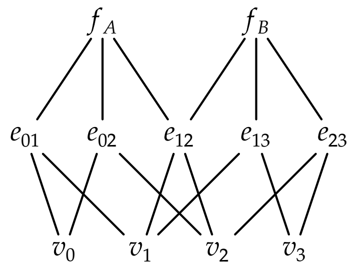
The CellModeller system is somewhat more complex, and includes an element of multiscale modeling. At the lowest of the three scales in this system are cell walls, and processes at this level include both membrane-bound processes and signalling between cells across walls. In the middle scale are individual plant cells; processes like a cell's genetic regulatory network take place at this level. Finally, the topmost scale is an organ or tissue; modeled at this level are the interactions between organs or tissues. The system records both the neighbourhood relations within each scale (for example, which cells are adjacent) and the incidence relationships between the scales (for example, recording the walls of each particular cell).

The incidence relationship between walls and cells is the same as the topological boundary relationship. It can therefore be useful in the same situations as described in Chapter 1; for instance, the differential equation for the accumulation of a diffusing substance within a cell simply adds the flux across the walls from the next scale down to contributions from the cell's internal processes. However, the incidence relationship between cells and tissues seems to be more like "is part of"/"contains", not topological in character. More recent development of CellModeller, such as a model of biofilm growth (Rudge et al. 2012), has focused on these upper levels and interaction between cells and the tissues they are part of.

Both of these systems support only a limited topological expressiveness: polygonal cells separated by cell walls, in the case of Virtual Leaf, or what amounts to the same, with the addition of tissue-level processes, for CellModeller. Neither system can easily be used for general developmental computing.



(a) A simple mesh with two faces, five edges, and four vertices



(b) Incidence graph

v	$x_0\, y_0\, z_0$	*vertex positions*
v	$x_1\, y_1\, z_1$	
v	$x_2\, y_2\, z_2$	
v	$x_3\, y_3\, z_3$	
f	1 2 3	*vertices of face* f_A
f	2 3 4	*vertices of face* f_B

(c) Face-vertex list representation

$$v_0 : [v_1, v_2]$$
$$v_1 : [v_3, v_2, v_0]$$
$$v_2 : [v_0, v_1, v_3]$$
$$v_3 : [v_2, v_1]$$

(d) Vertex-vertex representation

$$f_A : \langle v_0, v_1, v_2 \rangle \quad \mathscr{A} f_A : \langle f_B, -, - \rangle$$
$$f_B : \langle v_1, v_2, v_3 \rangle \quad \mathscr{A} f_B : \langle -, -, f_A \rangle$$

(e) Winged incidence representation

Figure 2.6: (a) A simple mesh and (b)-(e) several incidence-based representations of it

2.2 Data structures for cell complexes and polygon meshes

Structured topological representations are important in the fields of computer graphics and geometric modeling. This is especially true for modeling in two dimensions, where there are a large number of representations for *polygon meshes*. A polygon mesh is made up of points (the vertices of the mesh), one-dimensional curves connecting the points (the edges of the mesh), and polygons bounded by a curved sequence of edges (the faces of the mesh). In many representations, polygon meshes are equivalent to two-dimensional cell complexes, but some definitions may allow meshes which are not cell complexes, for example by allowing degenerate or self-intersecting polygons, or polygons with holes on the interior.

The many different representations in the literature can largely be categorized by whether they are based on recording *incidence* between cells, or based on connecting *handles*, data structures representing some combination of a small number of adjacent cells. An important class of the latter is representations based on the *combinatorial map*, upon which the Cell Complex Framework described in this thesis is based.

2.2.1 Incidence-based representations

The first class of mesh data structures to consider is *incidence-based* representations. These structures store the cell complex by recording incidence and adjacency between cells. A simple incidence-based representation is the *incidence graph* (Figure 2.6b). The boundary relation in a cell complex is a partial ordering (Section 3.2), and the incidence graph representation merely records this relation. This is enough information to represent the structure of the cell complex.

The incidence graph gives a complete list of all of the incidence relationships in a mesh. Some other incidence-based representations give only a subset of the complete incidence graph; the representation of the mesh is still unambiguous even with very few incidences marked, especially if these incidences are given with an implied ordering. For example, the face-vertex list (Botsch et al. 2010) explicitly defines vertices, then defines each polygonal face by the ordered list of vertices in its boundary (Figure 2.6c). This format is used by many mesh file formats, such as OBJ and VRML; polygonal meshes are sent to graphics hardware for rendering in a similar format. Note that this representation defines vertices (as primitives) and faces (as lists of vertices), but defines edges only implicitly, and multiply defines edges which are on more than one face.

As the example of the face-vertex list shows, it is not necessary for a data structure to explicitly represent cells of every dimension. In the case of the vertex-vertex data structure used by vv (Section 2.1.4), only vertices are explicitly represented. Associated with each vertex is the cyclic list of adjacent vertices (Figure 2.6d), and this information is sufficient to uniquely define the topological structure (Edmonds 1960).

Incidence-based representations can also include more structure than a basic incidence graph. Storing this additional information can make tasks such as accessing neighbourhoods or adding or removing cells easier. For example, the *winged incidence graph* (Paoluzzi et al. 1993) adds adjacency information to the incidence graph (Figure 2.6e). Associated with each vertex v in the boundary of c is the adjacent cell c' which lies on the other side of the face opposite v. For instance, in the mesh shown in Figure 2.6a, the edge e_{12} is opposite vertex v_0 in face f_A, and on the other side of that edge is face f_B; thus, in its winged incidence representation the adjacency to f_A associated with v_0 (i.e. the first entry in the adjacency list) is f_B. This association relies on the duality between k-cells and $(d-k)$-cells, and so is only defined for *simplicial* complexes (Section 3.1), for example, two-dimensional triangular meshes.

2.2.2 Handle-based representations

Another class of data structures for meshes is what I call *handle-based* representations. These are distinguished by the main entity, a *handle*, which in some way represents a position in the mesh; this is not (necessarily) a simple geometric object, and may encode an orientation or multiple incident objects. A small number of operations can be applied to a handle, each resulting in another handle; these are implemented as links between handle data structures. A program can iterate around the mesh by following links between handles.

One example of a handle-based representation can be created from the vertex-vertex *path algebra*, which was introduced by Smith (2006) as a shortened form for successive vv operations. The path algebra is based on three operations which can be applied to a pair of adjacent vertices (v, w). The pair (v, w) is distinct from the pair (w, v), so each edge in the adjacency graph corresponds to two pairs; each pair is thus a *half-edge*. A number of handle-based representations are based on half- (or quarter-) edges or facets.

The handle in the vv path algebra is a single half-edge (v, w). The three operations which define the path algebra are (a) **next**, which changes w to the next clockwise neighbour of v; (b) **prev**, which changes w to the next counterclockwise neighbour of v; and

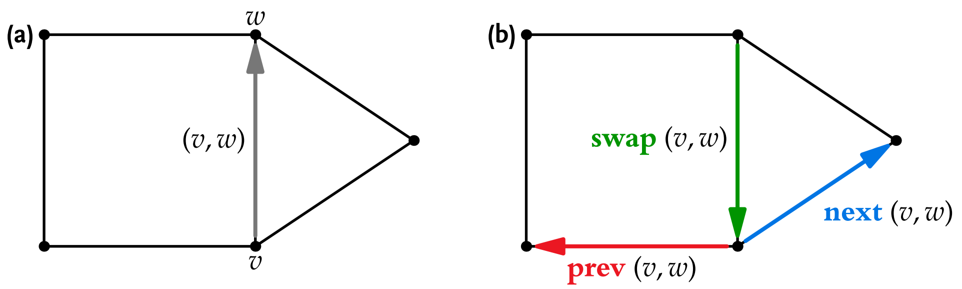


Figure 2.7: Vertex-vertex path algebra operations. (a) A planar graph with half-edge (v, w) defined. (b) The action of the three operations next, prev, and swap.

(c) **swap**, which interchanges v and w. These operations are demonstrated in Figure 2.7.

In a handle-based representation, if applying an operation **op** to a handle h results in a handle h', then there is a pointer from the data structure representing h to the data structure for h'. A data structure representing a half-edge in the vv path algebra is then

```
struct handle:
    vertex    v
    vertex    w
    handle⋆   next
    handle⋆   prev
    handle⋆   swap
```

Applying an operation is simply performed by following the pointer to find the resulting handle. It is clear that the **prevto** and **nextto** operations of the standard vv algebra (Section 2.1.4) can be performed easily using **prev** and **next**. The other operations can also be converted into operations on handles. For example, Algorithm 2.1 removes an edge from the mesh. This is done by pointing the **next** and **prev** handles at each other for each half of the edge. For instance, the **next** operation applied to the **prev** handle should now bypass the deleted edge and give that edge's own **next** handle. The isolated half-edges are then deleted.

Algorithm 2.1 Deleting an edge in the vv path algebra

```
Input: A half-edge $h$
1: procedure DeleteEdge($h$)
2:     $h' \leftarrow h$.swap
3:     $h$.prev.next $\leftarrow h$.next
4:     $h$.next.prev $\leftarrow h$.prev
5:     delete $h$
6:     $h'$.prev.next $\leftarrow h'$.next
7:     $h'$.next.prev $\leftarrow h'$.prev
8:     delete $h'$
9: end procedure
```

An advantage of the handle-based representation over the implementation of vv based on adjacency is that operations which leave an inconsistent state, such as delet-

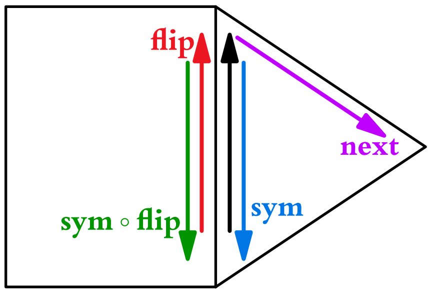


Figure 2.8: In the quad-edge representation of a two-dimensional mesh, each edge corresponds to four handles (black, red, blue, green). The operations sym and flip are used to shift between the different handles on the same edge; the operation next moves to the next edge in the direction of the handle.

ing only one half of the adjacency relation, can be detected immediately: one of the pointers will be invalid. On the other hand, it is simpler to confirm the validity of the entire mesh using the graph rotation system representation. Chen and Akleman (2003) combine the two, storing both a handle-based *winged-edge* representation with a graph rotation system to simplify the tasks of confirming that a mesh is valid and maintaining its validity.

There are a variety of handle-based representations which also choose part-edges as handles, with different choices of operations. Some examples of these are the *winged-edge* representation (Baumgart 1975); the *doubly-connected edge list* (Muller and Preparata 1978); the *half-edge* representation (Weiler 1985); and the *quad-edge* representation (Guibas and Stolfi 1985). Unlike incidence-based representations, most of these data structures have been introduced in the context of computational geometry; they are proposed for use in algorithms in which the topological relations between objects is continually changing.

The quad-edge representation is particularly relevant here, as it relates quite closely to representations based on the combinatorial map. In the quad-edge data structure, each edge is represented by exactly four handles (Figure 2.8). A single handle ("quad-edge") represents the edge in one direction, next to one of the two adjacent faces. The operations on a handle are (a) **sym** changes the direction of the quad-edge; (b) **flip** changes the associated face of the quad-edge; and (c) **next** returns the next quad-edge found cycling around the face.

There are a few handle-based representations for three-dimensional volumetric meshes. One example is the *facet-edge* representation (Dobkin and Laszlo 1987), designed as an extension of the quad-edge representation into three-dimensional meshes. Since an edge in a three-dimensional mesh may be incident to more than two facets, the handle changes from an oriented edge in quad-edge to an oriented facet-edge pair. There are two orientations considered: the direction of the edge between its endpoints, and the orientation of the facet within the greater mesh (what Dobkin and Laszlo call "clocking"). There are thus four handles for each facet-edge pair, corresponding to all combinations of these orientations (Figure 2.9). A single handle then supports four operations: (a) **rev** changes the direction of the edge; (b) **clock** changes orientation of the facet; (c) **enext** changes the edge to the next in cycle about the facet; (d) **fnext** changes the facet to the next in cycle about the edge. The direction of the edge determines which edge is next under **enext**; the clocking determines which facet is next under **fnext**.

Other three-dimensional handle-based representations tend to be based on the combinatorial map. As these representations are the inspiration for the system described in

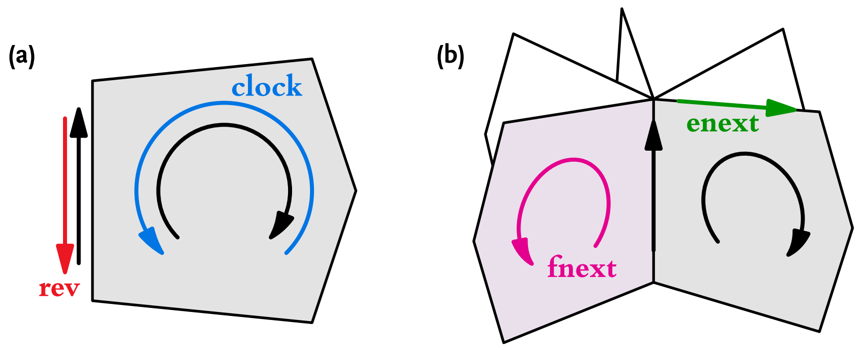


Figure 2.9: The facet-edge representation for three-dimensional polyhedral meshes. (a) For a given facet and edge, there are four possible orientations; the facet can be oriented clockwise or counterclockwise (black or blue), and the edge can be oriented with or against the orientation of the facet (black or red). Navigation between these four handles is done with the operations **rev** and **clock**. (b) The next edge along the facet is chosen with operation **enext**. The next facet around the edge in a "positive" direction (determined by the orientation of the facet) is chosen with operation **fnext**.

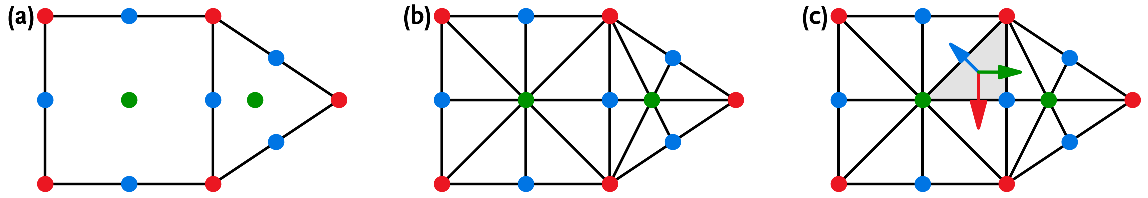


Figure 2.10: Constructing the combinatorial map of a two-dimensional mesh. (a) The vertices of the combinatorial map consist of vertex-vertices (red), edge-vertices (blue), and face-vertices (green), placed in the interior of cells of the corresponding dimension in the original mesh. (b) The vertices corresponding to incident cells are joined by an edge, forming a triangular subdivision of the original mesh. Each dart is a simplex with one vertex of each dimension. (c) For each dimension k, the dart has exactly one neighbour which shares all of the vertices except the vertex of dimension k. For example, the dart pointed to by the green arrow (σ_2) shares the vertex and edge vertices, but has a different face vertex.

this thesis, they will be discussed separately in the next section.

2.2.3 Representations based on the combinatorial map

The two mesh representations described in this section, the *cell tuple* (Brisson 1990) and *G-maps* (Lienhardt 1991), are both based on the *combinatorial map*, a new cell complex built by subdividing the complex to be represented. The combinatorial map will be discussed more completely in Section 3.3; briefly, a combinatorial map is constructed from a given cell complex by placing a vertex in the interior of every cell (Figure 2.10a), then subdividing the highest-dimension cells into simplexes called *darts* (Figure 2.10b). In a n-dimensional cell complex, these darts will have $n + 1$ vertices; one will be in the interior of an n-cell (an n-vertex), another in the interior of an incident $(n - 1)$-cell (an $(n - 1)$-vertex), and so on.

Each dart is adjacent to $n + 1$ others, and each adjacent dart shares all of the vertices with its neighbour except one. This means that, for each dart, exactly one of its neighbours has a different 0-vertex; exactly one has a different 1-vertex, and exactly one has a different 2-vertex (Figure 2.10c). Further, the unique k-neighbour of a dart's own k-neighbour is the original dart itself. We thus call these neighbourhood relationships "involutions", and write the involution which produces the k-neighbour as σ_k.

This structure leads to an obvious definition of the handle:

```
struct handle:
    handle★ $\sigma_0$
    handle★ $\sigma_1$
    ⋮
    handle★ $\sigma_n$
```

As with the other handle-based representations, a complete collection of darts and their corresponding involutions serves to completely define a cell complex; a proof of this is given by Brisson (1990) and is recapitulated in brief in Chapter 3. Darts and involutions are useful not only in representing the cell complex, but also in exploring and manipulating it; some of these operations are described in Chapters 4 and 5.

Both cell tuples (Brisson 1990) and G-maps (Lévy and Mallet 1999) have been implemented with a handle of the above form. The primary difference between the two is the assumed underlying structure. While cell tuples assume an underlying manifold,[2] G-maps assume only the existence of a set of darts and involutions; this structure is not necessarily the combinatorial map of a divided manifold, but is the "generalized map" giving G-maps their name. Lienhardt (1994) defines the class of spaces which can be modeled with G-maps as "quasi-manifolds"; this is a broader class than simple manifolds, and includes manifolds with singularities, as well as cells with discontinuous or nonspherical boundaries.

Brisson (1990), on the other hand, starts from the assumption of a divided manifold and from this develops the combinatorial map as a representation. While the more general G-maps can essentially *only* be modeled by an explicit representation of the involution functions, an underlying manifold structure allows more freedom in the implementation of cell tuples. Brisson (1990) suggests several possible alternatives: a database of all possible cell tuples, with involutions determined by queries containing wildcards, or a direct representation of the combinatorial map using a graph data structure. Other alternatives described by Brisson rely on Theorem 3.3 (Section 3.3), that the involution σ_k depends only on the incident cells of dimensions $k-1$, k, and $k+1$. This fact can be leveraged to create a representation where each involution is described by the connection between (c_{k-1}, c_k, c_{k+1}) and (c_{k-1}, c'_k, c_{k+1}). The data structure I have used, described in Chapter 4, is similar to this proposed implementation.

[2]Terms such as *manifold* and *divided manifold* will be defined more concretely in Chapter 3. For the distinction between cell tuples and G-maps, the key feature is that, like familiar Euclidean space, a manifold is a smooth space with a constant dimension.

3 Mathematical background

The characteristics of developmental computing given in Chapter 1 refer to the *topological space* represented; that is, the *neighbourhood relations* between the entities modeled. I have proposed the *cell complex* as a topological space which is well-suited to developmental computing, and in this chapter I present the mathematical foundations of cell complexes and their representation by combinatorial maps.

While cell complexes are useful as an abstract topological space, within developmental computing we will usually use them as a model for a particular geometry. This chapter therefore begins by discussing the underlying geometry which I assume in this work: the *subdivided manifold*. Cell complexes are introduced here as a representation for this subdivision. After demonstrating some properties of these particular cell complexes I will go on to present the properties of complexes in a more abstract setting, including the idea of geometry-independent orientation. Finally, I construct the combinatorial map and show how it is used to represent and manipulate cell complexes. This lays the groundwork for Chapter 4, where I use this structure to build the Cell Complex Framework.

Throughout this chapter, I will state a number of theorems about cell complexes. These theorems were originally proved rigorously by Brisson (1990); I state them here, along with Arguments which show the intuition behind each proof, to help explain some of the decisions made in the design of the Cell Complex Framework.

3.1 Geometry and topology

The topology modeled by the Cell Complex Framework (CCF) is that of the cell complex. The underlying *geometry* assumed by the CCF, however, is the *manifold* (Hatcher 2002). The defining feature of a manifold is that small portions are indistinguishable from flat space, a feature shared by geometric curves, surfaces, and cell layers, as well as objects in normal flat three-dimensional space. Representing a manifold computationally is made easier if the manifold is subdivided by a cell complex; the local topology and geometry of each cell is simple, and the global topology described by the cell complex describes how these simple forms are combined into the manifold's global geometry and topology. Of course, subdividing a manifold is also important in developmental modeling as the definition relies on interactions between neighbouring components.

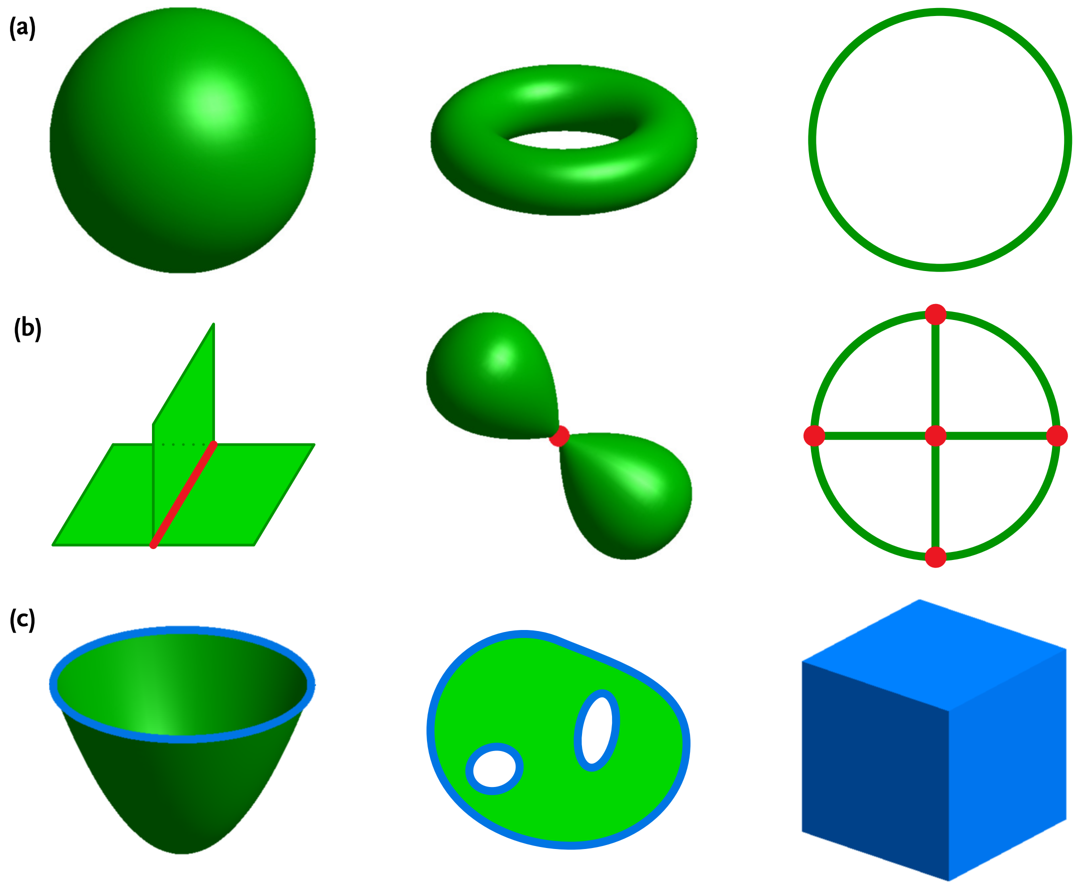


Figure 3.1: a. Manifold surfaces: the surface of a sphere is a 2-dimensional manifold; the surface of a torus is also a 2-dimensional manifold; the loop is a 1-dimensional manifold. b. Nonmanifold surfaces: three sheets attached along a curve; two teardrops joined at a point; a loop with spokes. Points whose neighbourhoods are not locally Euclidean are indicated in red. c. Manifolds with boundary, indicated in blue: the truncated paraboloid is a 2-dimensional manifold with 1-dimensional boundary; the plane figure shown is a 2-dimensional manifold with 1-dimensional boundary in three parts; the solid cube is a 3-dimensional manifold with 2-dimensional boundary.

Intuitively, a *manifold* is a collection of points that, at any point, looks like Euclidean space. More precisely, two spaces "look like" each other when there is a continuous mapping between them; such a mapping is called a *homeomorphism*, and such a mapping preserves topological properties. If M is the manifold, then for any point $p \in M$ there is an associated homeomorphism ψ_p between the neighbourhood of p and a subset of n-dimensional Euclidean space E^n.[1] The dimension n of the associated Euclidean space must be the same at all points p; it defines *dimension* of M. Each mapping ψ_p is called a *chart* and the collection of all of the mappings is called an *atlas* for M. Some surfaces which are manifolds are shown in Figure 3.1a.

If there are points where an otherwise manifold object M is not locally homeomorphic to Euclidean space, these are called *nonmanifold points*. Some nonmanifold objects are shown in Figure 3.1b, with points which are not locally Euclidean indicated in red. For example, the leftmost example, of three sheets joined along a curve, is nonmanifold along the curve; in two-dimensional Euclidean space, such a curve would divide its neighbourhood in two, while here its neighbourhood is divided in *three*. The example in the middle, the teardrops joined at a point, is nonmanifold at that point because the surface is split by a single point, not a curve as would be the case with a two-dimensional manifold like the rest of the surface. The rightmost example, the wheel with spokes, is similar to the leftmost, with nonmanifold points where more than two curves meet.

There is one important type of point which is not locally homeomorphic to Euclidean space: the points in the *boundary* of an otherwise manifold space. We want to treat spaces with boundaries, so we extend our definition to the *manifold with boundary*. A manifold with boundary M is a collection of points p, divided into two sets: the *boundary* ∂M and the *interior* $M \backslash \partial M$. Interior points are locally Euclidean, just as in the case of a manifold; the neighbourhood of a point in the boundary is homeomorphic to a bounded Euclidean space.[2] The boundary of an n-dimensional manifold is an $(n-1)$-dimensional manifold (without boundary). Some manifolds with boundary are shown in Figure 3.1c; their respective boundaries are outlined in blue.

A *subdivision* of an n-dimensional manifold M is a set $\{S_i \subset M\}$ of manifolds with boundary, where the submanifolds are all of the same dimension as M and cover all of M ($M = \bigcup_i S_i$), and any two submanifolds intersect only on their boundaries ($S_i \cap S_{j \neq i} \subset \partial S_i \cap \partial S_j$). Figure 3.2 shows a manifold subdivided into submanifolds.

The submanifolds of a divided manifold are discrete components which still describe the topology and geometry of the manifold. However, as Figure 3.2 suggests, a general subdivision of a manifold may still be too general to easily represent computationally. Indeed, without constraints on the form of the submanifolds, they may be even more complex than the original manifold; for instance, the submanifold marked by a star in Figure 3.2 has a boundary in two parts, unlike the undivided manifold. By constrain-

[1]Formally, p is surrounded by an open set $U \subset M$ with an associated open subset $V \subset E^n$ related by the homeomorphism $\psi_p : U \to V$. Without loss of generality, we can assume that V is the open ball $\left\{(x_1, \dots, x_n) \in E^n \mid \sum_i x_i^2 < 1\right\}$.

[2]Without loss of generality, the *half-ball* $\left\{(x_1, \dots, x_n) \in E^n \mid x_1 >= 0, \sum_i x_i^2 < 1\right\}$. The half-ball is the same as the open ball, but with points in one dimension restricted to one side of the origin, giving a boundary at $x_1 = 0$.

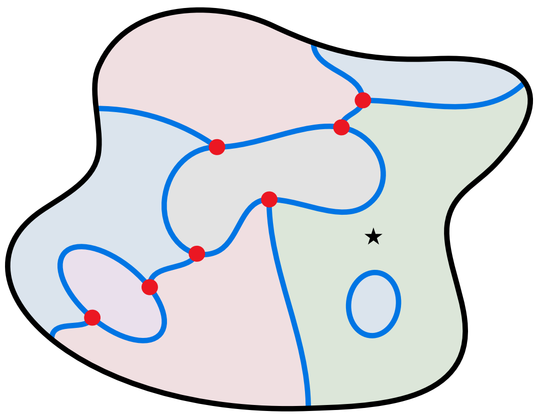


Figure 3.2: A 2-dimensional manifold subdivided into eight submanifolds. Note that segments of the boundaries of each submanifold are either on the boundary of the entire manifold (black), or are on the shared boundary of two submanifolds (blue), except at discrete points where more than two submanifolds meet (red). Subdivided, however, does not mean simpler; the starred submanifold (green) has a divided boundary, unlike the entire manifold.

ing the form of the submanifolds, however, the subdivision can be used to effectively represent the entire manifold.

The first special subdivision to consider is the *simplicial complex*. A *simplex* is a generalization of the triangle or tetrahedron to an arbitrary dimension. Recall that three non-collinear points in a two-dimensional space uniquely identify a triangle, while four non-coplanar points in a three-dimensional space uniquely identify a tetrahedron. Generalizing this notion, we say that an n-simplex is uniquely defined by $n+1$ distinct points, or *vertices*. Thus, a single vertex is a 0-simplex; a 1-simplex, or *edge*, is uniquely defined by two vertices, its *endpoints*. A triangular 2-simplex is uniquely defined by three vertices, a tetrahedral 3-simplex by four vertices, and so on.[3] Figure 3.3 illustrates a single 3-simplex and its boundary simplexes. Note that the boundary of a tetrahedron consists of four triangles, the boundary of a triangle is three edges, and the boundary of an edge is two vertices. In general, the boundary of an n-simplex consists of $n+1$ $(n-1)$-simplexes, defined by the $\binom{n+1}{n} = n+1$ possible n-element subsets of its $n+1$ vertices.

We denote the set of $(n-1)$-simplexes which make up the boundary of n-simplex S by ∂S, and say that the simplexes X in the boundary of S are *incident* to S, $X \prec S$. A boundary simplex X is itself bounded by $(n-2)$-simplexes Y, $Y \prec X$, and so on. We extend the notion of incidence to all of these boundaries of boundaries; equivalently, all simplexes defined by any nonempty subset of the n vertices of S are incident to S, or the incidence operation is transitive, with $Y \prec X \prec S$ implying $Y \prec S$.

A subdivision consisting of simplexes is a *simplicial complex*. We can construct a simplicial complex on a manifold by iteratively joining $(n-1)$-simplexes into n-simplexes. For example, consider the construction of a simplicial subdivision of a two-dimensional manifold:

[3]The definitions of "triangle" and "tetrahedron" often require that the sides be *straight*; here I use them (and "polygon" below) to imply only the *topological* relationship between them and their sides, edges, and vertices (e.g. a triangle has three edges and three vertices; a tetrahedron has four triangular sides, six edges, and four vertices) and not their *geometry*.

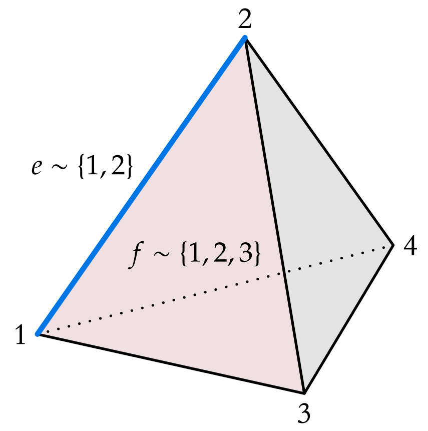
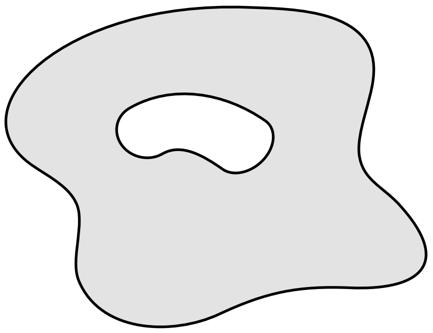
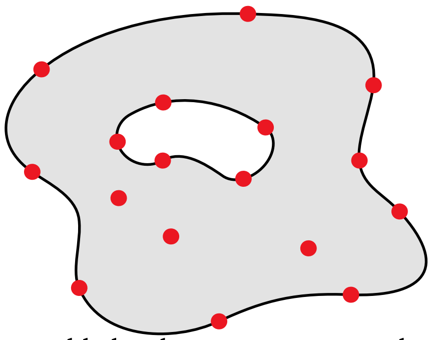


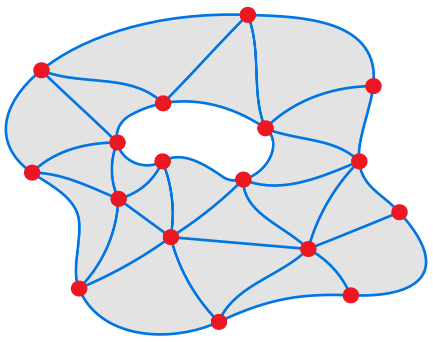
Figure 3.3: A 3-simplex is defined by four vertices, here numbered 1, 2, 3, 4. Its boundary contains four triangular faces (2-simplexes), defined by $\{1,2,3\}$, $\{1,2,4\}$, $\{1,3,4\}$, and $\{2,3,4\}$; note that these are the four three-element subsets of $\{1,2,3,4\}$. The 3-simplex contains six edges (1-simplexes), defined by $\{1,2\}$, $\{1,3\}$, $\{1,4\}$, $\{2,3\}$, $\{2,4\}$, and $\{3,4\}$, all of the two-element subsets of $\{1,2,3,4\}$. The triangle f, defined by $\{1,2,3\}$, has a boundary consisting of three edges, defined by $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$. The edge e, defined by $\{1,2\}$, has a boundary consisting of those two vertices.



We start with a two-dimensional manifold with boundary.



Several distinct points (red) are identified as vertices (0-simplexes).



Edges (blue) are added. These are curves on the manifold between vertices; there is at most one edge between any pair of vertices. Edges do not intersect vertices other than their own endpoints; they do not intersect other edges, except at shared endpoints. Since these edges are the boundaries of the submanifolds (triangles), the boundary of the manifold must be covered entirely by edges.

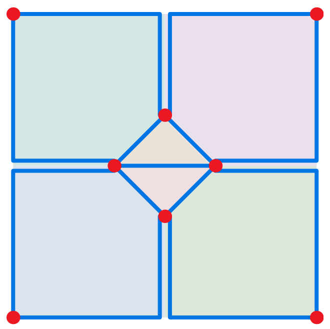
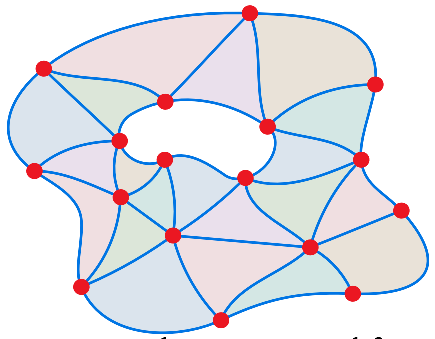


Figure 3.4: An invalid simplicial subdivision of a two-dimensional manifold. Each 2-simplex has three edges and three vertices, but some edges partly overlap. Overlapping edges have been moved apart slightly for illustration.



Faces are created from non-overlapping patches of manifold bounded by exactly three edges. (If there are patches not bounded by three edges, they are further subdivided by adding edges.) These faces are then 2-simplexes defined by the three points delimiting the edges. Faces do not intersect with edges or vertices not in their boundary.

In higher dimensions, we continue in the same vein; 3-simplexes are created from volumes of the manifold bounded by four triangles, and so on. The non-intersection requirement, in general, is this:

> Two simplexes do not intersect except on their shared boundaries, or where one lies entirely within the boundary of the other.

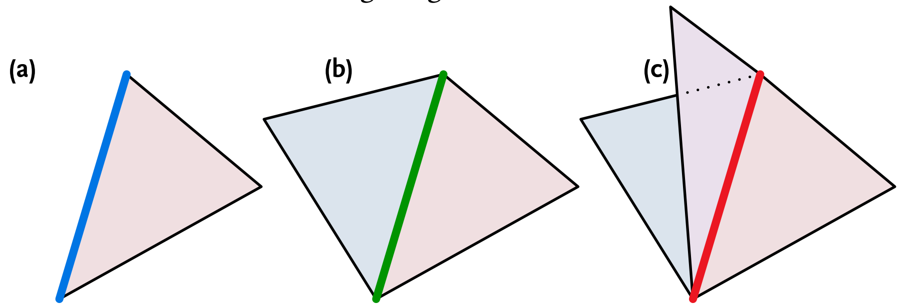
This requirement rules out subdivisions like that shown in Figure 3.4. We can then see that the boundary of an n-dimensional manifold must be covered by entire $(n-1)$-simplexes, whose respective boundaries consist of $(n-2)$-simplexes on the whole manifold's boundary.

One very useful consequence of subdividing a manifold with a simplicial complex is Theorem 3.1:

Theorem 3.1 (Lemma 4.2 from (Brisson 1990)). *Let $\mathbb{S}$ be a simplicial subdivision of n-dimensional manifold M, and S be an $(n-1)$-simplex in $\mathbb{S}$. One of two cases is true:*

1. *S is in the boundary of M and there is exactly one n-simplex with S in its boundary.*
2. *S does not lie in the boundary of M and there are exactly two such n-simplexes.*

Argument. Since M is covered by n-simplexes, the interior of S must intersect at least one of them, and by the non-intersection property, S must therefore be in its boundary. Three cases can be seen in the following image:



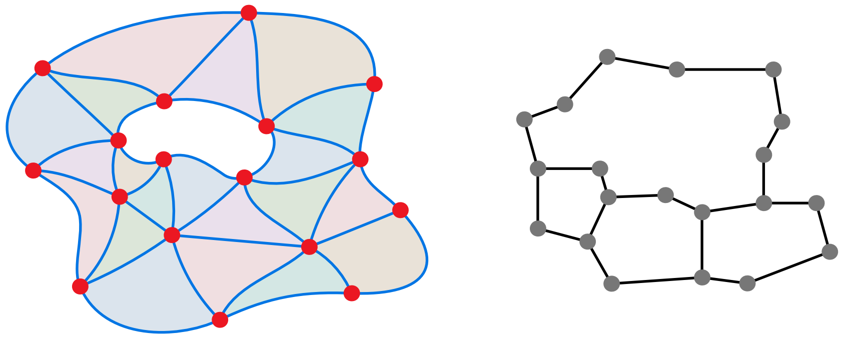


Figure 3.5: A subdivided 2-dimensional manifold and the neighbourhood graph of 2-simplexes in the manifold.

Edges on the boundary of M are like the blue edge in (a); they are in the boundary of only one triangle. Edges in the interior of M are like the green edge in (b); they are on the boundary of two triangles. The red edge in (c) is on the boundary of more than two triangles; however, if M is a 2-dimensional manifold, the complex is nonmanifold on the red edge. In an n-dimensional manifold, then, the only two possibilities are (a) or (b). $\square$

Theorem 3.1 tells us that we can identify pairs of n-simplexes which share a common $(n-1)$-simplex. We say that these simplexes are *neighbours*. We can represent this relationship by a neighbourhood graph (Figure 3.5). A path on this graph corresponds to a path between n-simplexes which passes from neighbour to neighbour through $(n-1)$-simplexes. This leads to Theorem 3.2:

Theorem 3.2 (Lemma 4.11 from (Brisson 1990)). *If M is a connected n-dimensional manifold subdivided by a simplicial complex $\mathbb{S}$, then between any pair of n-simplexes S_0, S_k there is a sequence of n-simplexes $[S_1, \ldots, S_{k-1}]$ such that S_i and S_{i+1} are neighbours.*

Argument. Since M is connected, there must be a curve connecting the two simplexes. If there were no neighbour-sequence, there would be some point where the curve is forced to cross between simplexes through a simplex of dimension less than $n-1$ (Figure 3.6). At this point, though, the complex would be nonmanifold. There must therefore be a neighbour-sequence. $\square$

Theorem 3.2 is not immediately useful, but is vital for proving fundamental statements about combinatorial maps in Section 3.3.

Simplicial complexes are a good choice as a topological structure for developmental computing: they have a well-defined geometry (as manifold subdivisions), a clear notion of neighbourhood, and can be easily represented digitally (as sets of vertices). However, the shapes we deal with in developmental computing are not necessarily as simple as simplexes; in particular, an n-simplex has $(n+1)$ boundary faces, so can have no more than $(n+1)$ neighbouring simplexes. We may model geometric surfaces with quadrilaterals as well as triangles, though, and a three-dimensional plant cell may have more than four neighbours. Ideally, we would like a topological representation which allows more general submanifold shapes and neighbourhood relations, while retaining the advantages of

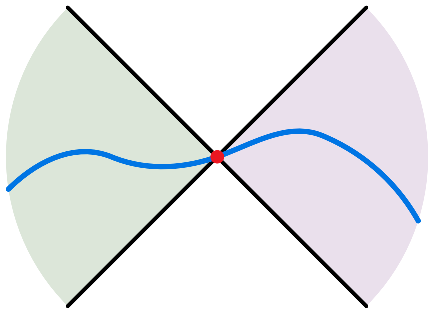
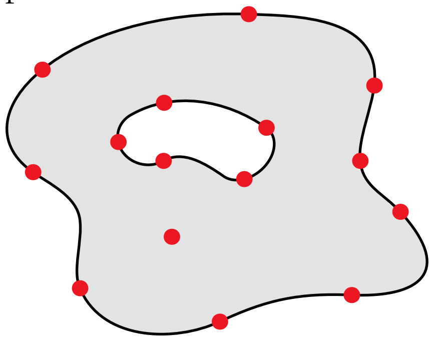


Figure 3.6: The curve (blue) connecting two n-simplexes (green, purple) cannot pass through an $(n-1)$-simplex (black) and must pass through a simplex of dimension less than $n-1$ (red). On this simplex, the complex is nonmanifold.

simplicial complexes. For this purpose, we can move to the more general idea of *cell complexes*.[4]

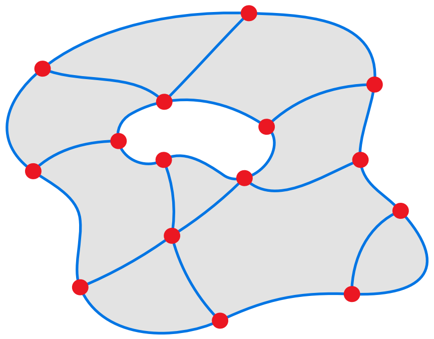
Like a simplicial complex, a cell complex is composed of objects of different dimensions. These objects are called *cells*; just as for simplexes, a 0-cell is a single vertex, and a 1-cell is an edge between vertices. The difference between simplexes and cells comes in higher dimensions: whereas an n-simplex is bounded by exactly $n+1$ $(n-1)$-simplexes, an n-cell is a polytope bounded by at least two $(n-1)$-cells. Thus, while a 2-simplex is a triangle, with three edges, a 2-cell is a polygon, with at least two edges. Naturally, this means that n-simplexes are just special cases of n-cells; however, while an n-simplex can be defined uniquely by its $n+1$ vertices, an n-cell is defined by the set of all of the $(n-1)$-cells in its boundary.

We can construct a cell complex in a similar way as we construct a simplicial complex:

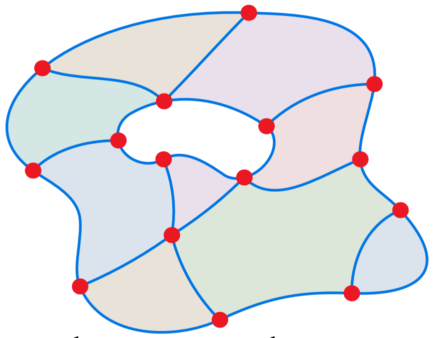


0-cells (red) are identified on the manifold.

Edges (blue) are added between distinct vertices. There can be any number of edges between a given pair of vertices, though the non-intersection property still holds; the edges cannot intersect except at their shared vertices.



[4]There are a number of similar mathematical structures. Most prominent is *CW complexes* (Hatcher 2002), which differ from my definition of cell complexes by being more permissive in the choice of the boundary of a cell. I have not seen any definition of “cell complex” which corresponds exactly to the definition I use here.



Polygons are created from non-overlapping patches of manifold bounded by edges. Each 2-cell can have any number of edges, and each is defined by the set of its bounding edges.

Theorems 3.1 and 3.2 remains true for cell complexes. Because of this, the *neighbour* relationship between n-cells sharing a $(n - 1)$-cell boundary is well-defined on any n-dimensional cell complex. It is somewhat inconvenient that each cell requires the set of all of its boundary cells to be specified, but we can use some concepts from combinatorial maps (Section 3.3) to both make defining cells somewhat simpler, and to make the operations we can use much more powerful.

3.2 Abstract complexes

This chapter has so far described simplicial and cell complexes as subdivisions of an underlying manifold. There is another perspective on complexes, however. They can be defined abstractly, independent of an immediate geometric meaning. An abstract complex can then be *embedded* in a manifold, giving each cell a geometric interpretation. This is in fact how the Cell Complex Framework models a subdivided manifold: as an abstract complex with an embedding. Abstract complexes are discussed in this section, while Section 3.2.1 describes embeddings.

Abstract complexes are defined purely combinatorially from lower to higher dimensional cells, much like complexes defined as manifold subdivisions:

1. *Vertices* are no longer points on a manifold; rather, they are just objects.
2. *Edges* have no geometrical meaning; they are not curves. They are now merely objects whose boundary is a pair of vertices. In the case of a simplicial complex, we still require that edges are uniquely defined by these two vertices, but in a cell complex, we allow any number of edges sharing a single pair of vertices.
3. n-cells (and n-simplexes) are not subdivided pieces of manifold; they are, again, just objects with a set of $(n - 1)$-cells which define their boundary (or by $n + 1$ vertices, in a simplicial complex).

As manifold subdivisions, the incidence relation $\prec$ between cells is defined geometrically; the m-cell c is in the boundary of the n-cell c' (where $m < n$) if the points which make up c are on the exterior of the closed manifold c'. In an abstract complex, on the other hand, the incidence relation defines the complex. As before, if c is in the boundary of c', which is in turn in the boundary of c'', then c is in the boundary of c''; the relation is transitive. In addition, at most one of $c \prec c'$ or $c' \prec c$ is true, so the incidence relation is a *partial order*. We can therefore completely describe the topology of the complex with the Hasse diagram of this relation, otherwise called the *incidence graph* (Figure 3.7).

With a few additions, we can extend the partial order into a *lattice*, which lets us define the operations **meet** ($\wedge$) and **join** ($\vee$) (Figure 3.7c). The **meet** of two cells c_1 and

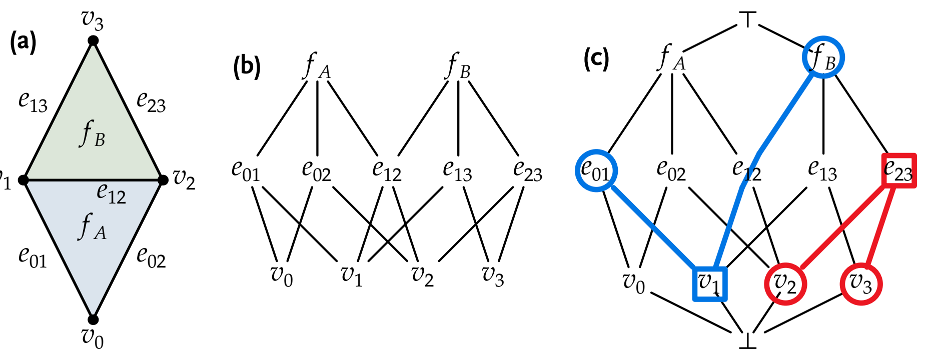


Figure 3.7: (a) An abstract simplicial complex and (b) its incidence graph. (c) The incidence relation is extended to a lattice by adding pseudocells $\top$ and $\perp$. **meet** and **join** relationships are illustrated: $e_{01} \wedge f_B = v_1$ and $v_2 \vee v_3 = e_{23}$.

c_2 is the cell of highest dimension which lies in the boundary of both c_1 and c_2; in a subdivided manifold, it is the cell along which c_1 and c_2 *meet*. The **join** of c_1 and c_2, on the other hand, is the cell of lowest dimension which has both c_1 and c_2 in its boundary; it is the cell which *joins* c_1 with c_2.[5] In order for **meet** and **join** to be defined for all cells, we must add two *pseudocells* $\top$ and $\perp$. $\top$ is the *supremum*; all cells lie in its boundary, while $\perp$ is the *infimum*; it is in the boundary of all cells. While the addition of these two pseudocells may seem arbitrary, they become very useful in implementing the Cell Complex Framework in Chapter 4.

3.2.1 Embedding

We can assign geometric shapes to the cells of an abstract complex to turn it into a manifold subdivision. Only geometric realizations that follow the rules described in Section 3.1, especially the non-intersection requirement, are valid; we call these *embeddings*.

One reasonable question is whether a given abstract complex can be embedded into a given manifold. In the extreme case, *any* simplicial complex with simplexes of dimension no greater than n can be embedded in $\mathbb{R}^{2n+1}$ (Giblin 1977). In general, however, the embedding question is undecidable (Markov 1958).

One embedding for complexes is so simple I call it the *default embedding*; it is used as the embedding for most of the models described in this thesis. A complex is embedded in $\mathbb{R}^n$ by (i) placing vertex v_i at position $\vec{p}_i$; (ii) interpreting an edge from v_i to v_j as the line segment between p_i and p_j; and (iii) interpreting a k-dimensional cell as the flat k-dimensional subspace bounded by its $(n-1)$-dimensional boundary cells. This embedding is defined for simplicial complexes. For general cell complexes, k-dimensional cells (for $1 < k < n$) may not be realizable as flat k-dimensional spaces in a space of n dimensions; for example, not all quadrilaterals in three dimensions are flat. This leads to ambiguity in the geometric interpretation of these cells.

A distinction that comes up in relation to embedding is between *intrinsic* and *extrinsic*

[5]Note that the symbols of these operations are the opposite of the shape they make on the incidence graph; the **meet** $\wedge$ picks out a single *lower*-dimensional cell, while the **join** $\vee$ picks out a single *higher*-dimensional cell.

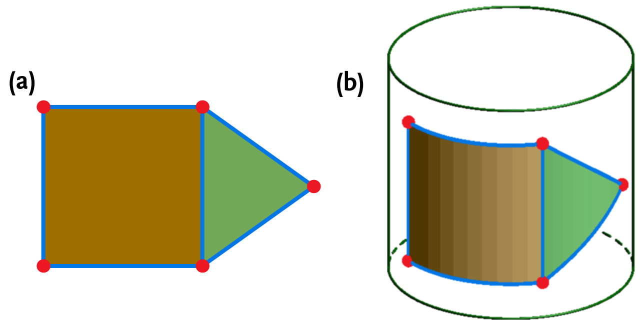


Figure 3.8: A cell complex in two isometric embeddings: (a) in the plane; (b) on the surface of a cylinder.

geometric features. The distinction hinges on geometric qualities that depend only on the *size* of the embedded cells; these are *intrinsic* qualities, while all other geometric features are called *extrinsic*. For instance, Figure 3.8 shows the same cell complex with an embedding in the plane and in the surface of a cylinder. Corresponding cells are of the same size: edges are of the same length, faces of the same area. We call such embeddings *isometric*. Extrinsic qualities, like the curvature of edges, depend on the particular embedding; intrinsic qualities, like the angles between edges, depend only on the assignment of sizes.

Two subdivided manifolds are isometric and have the same intrinsic qualities if there is a homeomorphism between them.

3.2.2 Orientation and cell chains

We have so far only touched on *orientation* with regard to the *global* orientation of a manifold: for example, the inherent left-right orientation of a string, or the distinction in graph rotation systems between "clockwise" and "counterclockwise". Orientation is important as it lets us compare numerical quantities on incident cells of different dimension. For example, recall the diffusion model from Section 1.1. A wall holds the flux between two cells. Because of the inherent left-right orientation, we know that this flux is positive if the flow is from left to right, and negative if from right to left. Without orientation, however, we *could not know* the direction of the flux.

Orientation is a binary choice: in an embedding, it is the choice between right- and left-handed coordinate systems for the cell. The orientation of an edge is then a choice of the two possible directions along the edge (Figure 3.9a); the orientation of a face is a choice of the two-dimensional coordinate frame (Figure 3.9b); the orientation of a three-dimensional volume is a choice of the three-dimensional coordinate frame (Figure 3.9c), and so on.

The handedness of coordinate systems doesn't hold in cell complexes without embeddings, but a concept of orientation exists in even abstract complexes. The orientation of an abstract edge is an assignment of a "from" vertex and a "to" vertex, while the orientation of an abstract face is equivalent to an ordering of its edges (Figure 3.10ab). The orientation of an abstract cell of higher dimension is harder to visualize, but one possibility is to see it as the combination of an orientation of one lower dimension, combined with a notion of "inside" (Figure 3.10c).

Since the distinction between left-handed and right-handed coordinate systems is

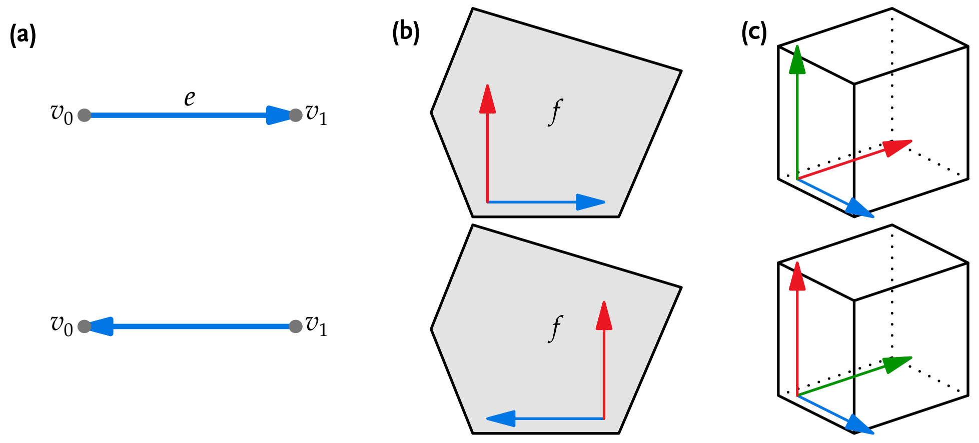


Figure 3.9: The two different orientations for cells of dimension 1, 2, and 3 correspond to a choice between right- and left-handed coordinate systems.

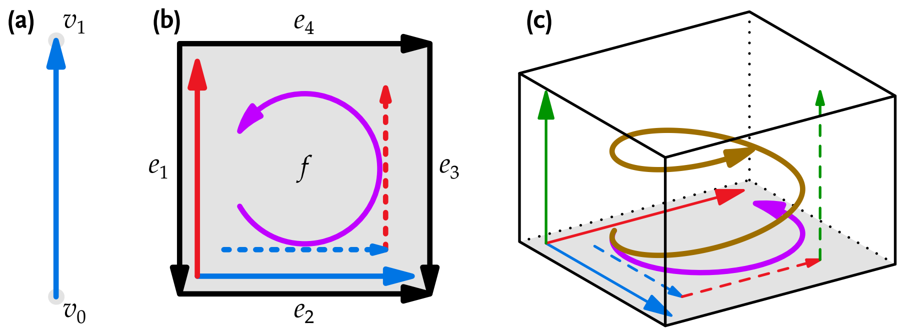


Figure 3.10: (a) The orientation of an abstract edge is an assignment of "from" and "to" vertices (such as $v_0 \to v_1$). (b) The orientation of an abstract face is equivalent to a circular ordering of its edges (such as $[e_1, e_2, e_3, e_4]$). If there is a coordinate frame defined on the face, the circular orientation can be created by moving along the blue axis, then rotating in the direction of the red axis. (c) A helical orientation (brown) of a volume can be created by moving along the blue axis, rotating in the direction of the red axis, and moving in the direction of the green axis. Alternately, the helix can be created by starting with the purple face orientation and moving inwards along the green axis. Note that in all of these cases, the orientation of an n-cell is equivalent to the orientation of one of the $(n-1)$-cells in its boundary moved inwards: trivially so in the case of (a) (the blue line is v_0 moved inwards), but also in (b) (the purple circle is the blue edge-orientation moved inwards) and (c) (the brown helix is the purple face-orientation moved inwards).

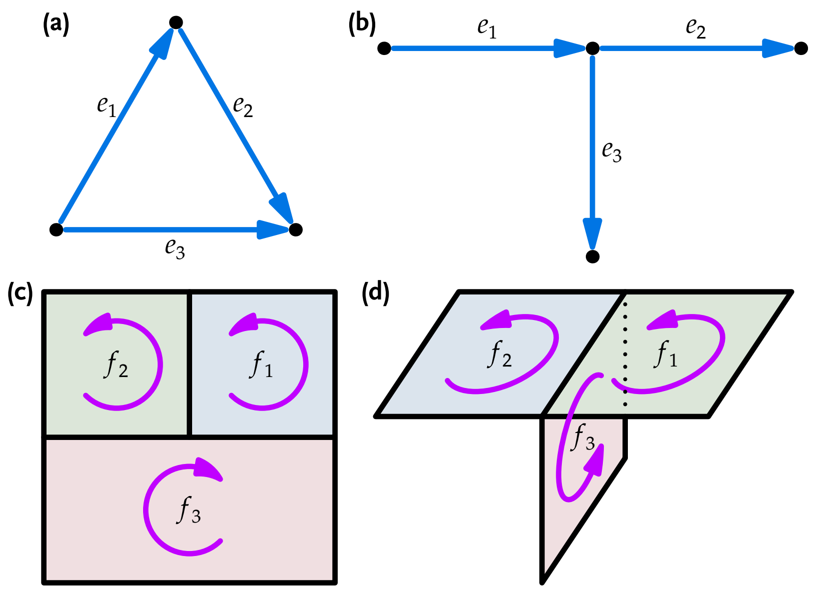
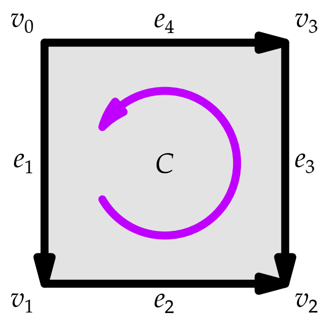


Figure 3.11: The relative orientation of adjacent cells. (a) e_1 and e_2 are of the same orientation, while both are of opposite orientation to e_3. (b) If the connectivity between edges is not one-dimensional, relative orientations cannot be assigned consistently: e_1 is of the same orientation as e_2, but e_1 is of the same orientation as e_3, while e_2 is of opposite orientation to e_3. (c) f_1 and f_2 are of the same orientation, while both are of opposite orientation to f_3. (d) If the connectivity between faces is not two-dimensional, relative orientations cannot be assigned consistently: f_1 is of the same orientation as f_2, but f_1 is of the same orientation as f_3, while f_2 is of opposite orientation to f_3.



(a) $$\partial C = e_1 + e_2 - e_3 - e_4$$

(b) $$\begin{aligned}\partial(\partial C) &= \partial e_1 + \partial e_2 - \partial e_3 - \partial e_4\\ &= (v_0 - v_1) + (v_1 - v_2) - (v_3 - v_2) - (v_0 - v_3)\\ &= 0\end{aligned}$$

Figure 3.12: (a) The boundary chain of the face C is the linear combination of its bounding edges, scaled by their orientation relative to C. e_1 and e_2 are oriented consistently with C; ∂C therefore contains those edges with coefficient +1. e_3 and e_4 are oriented in the other direction, so ∂C contains those edges with coefficient −1. (b) The boundary chain operator can be applied to the boundary chain of C; the boundary of the boundary is zero.

dependent on the particular embedding, the choice of the *absolute* orientation of a cell is arbitrary. We therefore look at the *relative* orientation between cells. First, can we meaningfully compare the orientation between *neighbouring* cells? It may initially look that way: in one dimension (Figure 3.11a), edges are oriented either around in a clockwise direction (e_1 and e_2) or a counterclockwise direction (e_3). In two dimensions (Figure 3.11c), faces are oriented either clockwise (f_1 and f_2) or counterclockwise (f_3). However, if the connectivity between n-cells is not n-dimensional, the orientations *cannot* be assigned consistently (Figure 3.11bd).

What about the relative orientation between *incident* cells? Here we are immediately on better ground; recall that the boundary of an n-cell is an $(n-1)$-dimensional space, so orientations can be assigned consistently to each of the $(n-1)$-cells in that boundary. The only question left is how to relate these absolute orientations to the n-cell itself; but remember that one way we are thinking about the orientation of an n-cell is as the combination of the orientation of an $(n-1)$-cell with an "inside" movement. This lets us consistently define the relative orientation between a n-cell C and the $(n-1)$-cells in its boundary:

Definition 1. *Let ∂C be the $(n-1)$-dimensional boundary of an n-cell C. Then the* relative orientation *$\rho(D, C)$ between C and an $(n-1)$-cell D in its boundary is determined by finding the n-dimensional orientation defined by moving the orientation of D "into" C; if this matches the orientation of C, then we say that D* is oriented consistently with *C $(\rho(D,C) = +1)$; otherwise, C* and *D* are not oriented consistently *$(\rho(D,C) = -1)$.*

The relative orientation between a cell and all of the cells in its boundary can be succinctly represented in its *boundary chain* (Palmer and Shapiro 1993; Egli and Stewart 1999). A *chain* is an assignment of a number to each cell in the complex; chains can be combined linearly, so the assignment of 20 to cell C_0 and 14 to C_1 (and zero to all other cells) is written $20C_0+14C_1$. An n-chain is a chain whose only nonzero values are assigned to n-cells. The boundary chain ∂C of an n-cell C, is an $(n-1)$-chain where elements of the boundary of C are assigned either +1 or −1, corresponding to a positive or negative

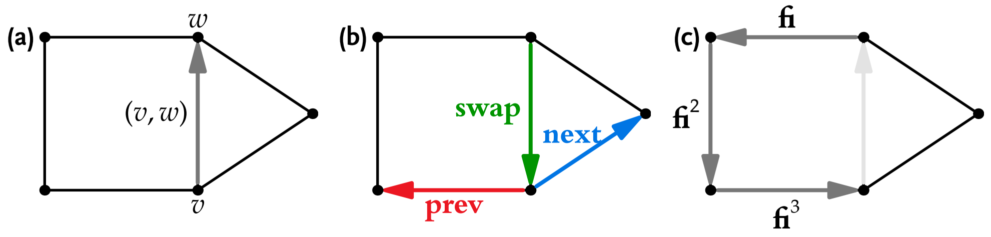


Figure 3.13: (a) A planar graph with half-edge (v,w) defined. (b) The action of the three operations **next**, **prev**, and **swap** on (v,w) in the vv path algebra. (c) The operation **fi** = **next** ∘ **swap** can be applied repeatedly to iterate through all edges incident on a face.

relative orientation to C, respectively:

$$\partial C = \sum_{D \in \partial C} \rho(D,C)D.$$

(See Figure 3.12).

Viewing the boundary in this way has several advantages. For one, the boundary operator ∂ can be applied to any chain; the operator distributes linearly to each cell. The operator can even be applied to a boundary chain (Figure 3.12b); this can be useful as a check that a boundary is valid. Recall that the boundary of an n-dimensional manifold is an $(n-1)$-dimensional manifold *without boundary*; if a supplied chain χ is indeed the boundary of an n-cell, we expect that there are no cells in its own boundary, and the chain $\partial\chi \equiv 0$. Because chains are such a useful representation of a boundary, we also use them to explicitly declare the boundary when constructing a cell (Section 4.5).

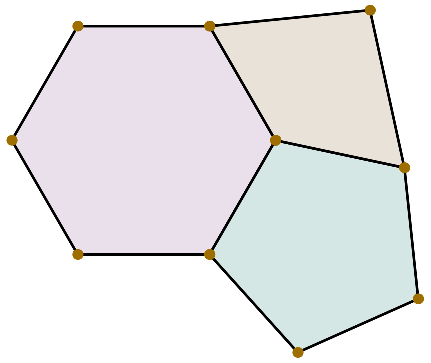
3.3 Combinatorial maps

While a cell is defined by the set of its bounding faces, this is not the most effective representation if we have to examine the boundary in some order. For example, we may have to look at the facets of a convex polytope which lie to one side of a given plane, or look at the sides of a polygon in order. Either of these problems can still be addressed using only unordered bounding faces, but they could be done more easily if the neighbourhood structure of the cells could be explored more explicitly.

As an example of such a structure, recall the *vertex-vertex (vv) path algebra* from Section 2.2.2. In a planar graph we identify a vertex v and one of its neighbours w; this pair is the half-edge (v,w) (Figure 3.13a). There are three operations which apply to each half-edge (Figure 3.13b): (a) **next** changes w to the next clockwise neighbour of v; (b) **prev** changes w to the next counterclockwise neighbour of v; and (c) **swap** interchanges v and w. We can move from any half-edge to any other by composition of these operations. We can also use these operations to visit vertices in a defined order. For example, repeated application of **next** lets us visit every neighbour of a vertex in clockwise order. We can also visit all of the vertices around a polygon by applying the composite operation **next** ∘ **swap** repeatedly (Figure 3.13c).

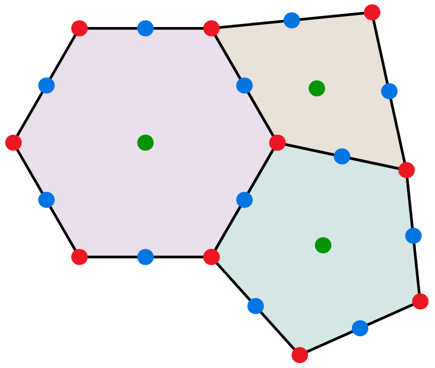
The vv path algebra, and graph rotation systems upon which it is based, rely on the clockwise ordering among a vertex's neighbours. This structure only exists in one and two dimensions; in three and more dimensions, there is no implicit ordering in these neighbourhoods. However, we can create a structure on a cell complex in any number of dimensions and use this to apply orderings to certain relationships, in addition to using operations to travel around the cell complex in a structured way. The particular structure I will discuss is the *combinatorial map* (Vince 1983), which I briefly discussed in Chapter 2.

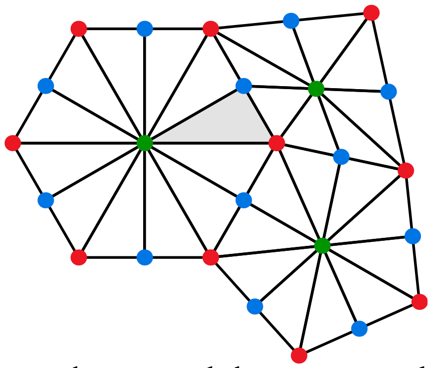
The combinatorial map of a cell complex is based on the *barycentric subdivision* of that complex. The complex is subdivided by the following process:



We start with a cell complex $\mathbb{C}$.

In addition to the already-existing vertices (red), we add new vertices in the middle of each edge ("edge-vertices", blue) and face ("face-vertices", green). In general, a new vertex is placed at the interior of each cell of dimension greater than zero. A vertex representing a k-cell is called a k-*vertex*.



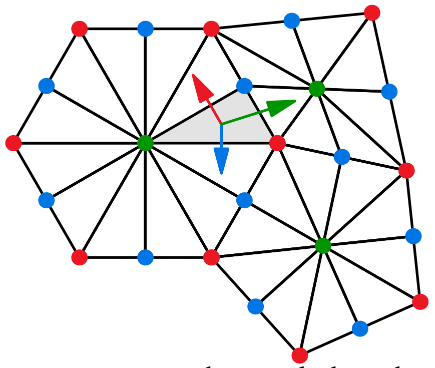


For each pair of incident cells $C_i < C_j$ we add an edge between their respective vertices. In general, we add a k-simplex for each set of $k + 1$ mutually incident cells $C_i < \dots < C_j$. If the cell complex is embedded in a manifold and each cell is convex, then each simplex in this simplicial complex is non-intersecting.

The maximal-dimension simplexes are defined by $n + 1$ vertices, representing a cell of each dimension from 0 to n. These simplexes are variously called *flags*, *darts*, or *tuples*. In this thesis, I will call the abstract n-simplexes *darts*, and will reserve the term *tuple* for their representation in the data structure described in Chapter 4.

From Theorem 3.1 we know that for any dart S with $(n-1)$-simplex $F < S$, there is either exactly one other dart S' with F in its boundary, or there is no such dart. Consider the former case. As S and S' share all of the n vertices in F, they must differ on only one vertex, the one representing a k-cell. Since there are $n + 1$ faces, then, for any dimension

$0 \le k \le n$, there is at most one neighbouring dart S_k which shares all of the vertices of S except the vertex representing a k-cell. We can define a function σ_k that captures this relationship: $\sigma_k(S) = S_k$, and by symmetry $\sigma_k(S_k) = S$. (Note that σ_k is not defined if there is no dart S_k.)



For example, σ_0 (red) takes the shaded dart to its neighbour which has a different 0-vertex; σ_1 (blue) takes the shaded dart to its neighbour which has a different 1-vertex; and σ_2 (green) takes the shaded dart to its neighbour which has a different 2-vertex. Figure 3.14 shows all of the defined σ functions.

There are a number of ways to extend the σ functions to every dart, including those with no n-neighbour (e.g. darts incident to the boundary). One possibility is to let $\sigma_n(S) = S$ in that case; the existence of the boundary is recorded, but the functions are still involutions. Another possibility, used by the Cell Complex Framework, is to have the exterior of the complex be a pseudo-n-cell ∞; then if S is defined by $\{C_0, \dots, C_n\}$, $\sigma_n(S)$ is a dart defined by $\{C_0, \dots, C_{n-1}, \infty\}$.

There is a nice limitation on the possible definitions of the σ functions. Theorem 3.3 tells us that each function σ_k only depends on the $k-1$, k, and $(k+1)$-vertices of each dart:

Theorem 3.3 (Corollary 4.3 from (Brisson 1990)). *If c_{k-1}, c_k, and c_{k+1} are cells of dimension $k-1$, k, and $k+1$, respectively, with $c_{k-1} \prec c_k \prec c_{k+1}$, then there is a unique k-cell $c'_k \ne c_k$ with $c_{k-1} \prec c'_k \prec c_{k+1}$.*

Argument. As the three cells are incident, there must be a dart D with vertices $c_0 \prec \cdots \prec c_{k-1} \prec c_k \prec c_{k+1} \prec \cdots \prec c_n$. Then the dart $D' = \sigma_k(D)$ has vertices $c_0 \prec \cdots \prec c_{k-1} \prec c'_k \prec c_{k+1} \prec \cdots \prec c_n$. To show that c'_k is unique, we suppose there is yet another cell c''_k with $c_{k-1} \prec c''_k \prec c_{k+1}$. But then there is a dart with vertices $c_0 \prec \cdots \prec c_{k-1} \prec c''_k \prec c_{k+1} \prec \cdots \prec c_n$. This means that there are three darts (which are n-simplexes) incident with the $(n-1)$-simplex with vertices $\{c_0, \dots, c_{k-1}, c_{k+1}, \dots, c_n\}$, which contradicts Theorem 3.1. ☐

By examining the actions of sequences of certain subsets of the σ functions, we can extract more information about the cell complex. The first important subset of operations is the *orbit*, defined as all sequences of functions from $\{\sigma_0, \dots, \sigma_{k-1}, \sigma_{k+1}, \dots, \sigma_n\}$ (i.e. all σ functions except σ_k). Any such sequence of functions will fix the k-vertex of any dart it is applied to. Since all darts which contain a k-vertex c_k are contiguous, we can use Theorem 3.2 to see that there is some sequence of σ operations which do not change c_k which take any such dart to any other. Therefore, the set of darts reachable by a function in a k-orbit is exactly those darts which share a k-vertex. We thus see that

Theorem 3.4 (Corollary 4.14 from (Brisson 1990)). *There is an isomorphism between k-orbits and k-cells.*

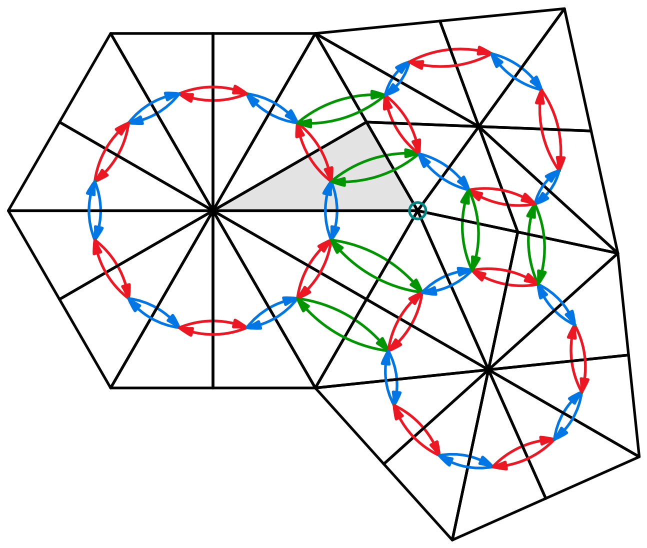


Figure 3.14: A cell complex subdivided into its combinatorial map, with arrows connecting all k-neighbouring darts and coloured according to the dimension k: applications of σ_0 are red, applications of σ_1 are blue, and applications of σ_2 are green. Note the cycles created by alternating σ_k functions of different dimension, such as the 0–1 (red–blue) cycle around faces or the 1–2 (blue–green) cycle around the marked vertex.

The second important subset of operations is those which are sequences of two functions σ_{k-1} and σ_k. It can be shown that

Theorem 3.5 (Theorem 4.5 from (Brisson 1990))**.** *If $c_0 \prec \cdots \prec c_n$, where c_i is an i-vertex, then the functions σ_{k-1} and σ_k induce a circular ordering on all darts with vertices $\{c_0, \ldots, c_{k-2}, c_{k+1}, \ldots, c_n\}$, and on all cells s_{k-1} and s_k such that $c_0 \prec \cdots \prec s_{k-1} \prec s_k \prec \cdots \prec c_n$.*

Argument. Name the set of all darts with vertices $\{c_0, \ldots, c_{k-2}, c_{k+1}, \ldots, c_n\}$ as $\mathscr{D}$. Consider the graph $(\mathscr{D}, \{\sigma_{k-1}, \sigma_k\})$, whose nodes are darts and whose edges are applications of σ functions. This graph is connected and each node is incident to one edge labelled k and one labelled $k-1$. This means that the graph must be a simple cycle, with edges alternately labelled k and $k-1$. Alternate composition of σ_{k-1} and σ_k thus imposes this circular ordering on $\mathscr{D}$ (Figure 3.14). The k-cells fixed by the applications of σ_{k-1} (every second operation) form an ordering of s_k, and vice versa for s_{k-1}. $\square$

Half of the possible combinations of σ_{k-1} and σ_k start with an application of σ_{k-1}, and traverse the order in one direction; the other compositions start with σ_k and traverse in the other direction. Thus, in Figure 3.14, starting from the shaded dart, sequences starting with σ_1 (blue) traverse the polygon in clockwise order, while sequences starting with σ_0 (red) traverse the polygon in counterclockwise order.

For $k = 1$, then, we see that the induced circuit traverses around a polygon. For $k > 1$, the circuit traverses the $k-1$ and k-cells incident to a particular $(k-2)$-cell. For $k = 2$, for instance, the induced circuit traverses the edges and faces incident to a vertex. In Figure 3.14, again starting from the shaded dart, σ_1–σ_2 sequences starting with σ_1 (blue) circle the marked vertex in counterclockwise order, while sequences starting with σ_2 (green) circle the vertex clockwise. In this case, we see the ordering from graph rotation systems. Indeed, the primitive operations from the vv path algebra are equivalent to composite operations on the combinatorial map: one possible mapping is **prev** $\equiv \sigma_1 \circ \sigma_2$, **next** $\equiv \sigma_2 \circ \sigma_1$, and **swap** $\equiv \sigma_2 \circ \sigma_0$.

We can see that the combinatorial map offers us powerful operations for traversing a cell complex. Its power becomes especially evident, however, in the following Theorem:

Theorem 3.6 (Theorem 4.4 from (Brisson 1990)). *Suppose two manifolds M_1 and M_2 are subdivided by cell complexes $\mathbb{C}_1$ and $\mathbb{C}_2$, respectively. The following are equivalent:*

1. *The subdivided manifolds $(M_1, \mathbb{C}_1)$ and $(M_2, \mathbb{C}_2)$ are homeomorphic.*
2. *The abstract cell complexes $\mathbb{C}_1$ and $\mathbb{C}_2$ are isomorphic.*
3. *The combinatorial maps of $(M_1, \mathbb{C}_1)$ and $(M_2, \mathbb{C}_2)$ are isomorphic.*

Argument. If the subdivisions of M_1 and M_2 are homeomorphic, the cell complexes must be isomorphic; thus, 1 implies 2. The parallel construction of the combinatorial maps of isomorphic complexes will clearly lead to isomorphic combinatorial maps; therefore, 2 implies 3. In the other direction, we can use Theorem 3.4, the isomorphism between k-orbits and k-cells, to reconstruct the cell complex from its combinatorial map, showing that 3 implies 2. Finally, there is a local homeomorphism between corresponding submanifolds defined by corresponding cells on $(M_1, \mathbb{C}_1)$ and $(M_2, \mathbb{C}_2)$. As the connectivity between these submanifolds is isomorphic, these local homeomorphisms can be extended into a global homeomorphism. In this way we can show that 2 implies 1. $\square$

This means that the combinatorial map has all of the information needed to reconstruct the subdivided manifold. We can thus construct a representation of subdivided manifolds using combinatorial maps. This representation will be at the heart of the Cell Complex Framework. In the next chapter, I describe how this representation works and is implemented.

4 Implementation of the Cell Complex Framework

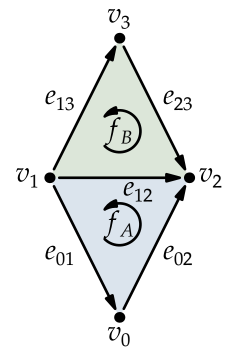
In Chapter 1 I discussed the advantages of cell complexes in developmental computing: first, that having cells of different dimensions lets physical quantities sit in their proper place in the structure; second, that operations such as cell division are inherently local in space; and third, that a cell complex provides local context for changing parameters on individual cells. This suggests the creation of a Cell Complex Framework for developmental computing, and in this chapter I describe the design and implementation of this framework. The Cell Complex Framework described here is based on the combinatorial map representation of a cell complex, in particular the Cell-Tuple data structures proposed by Brisson (1990).

4.1 Design considerations

In choosing design and implementation details for the Cell Complex Framework, we have several considerations, including space and time efficiency, implementation complexity, and ease of use. These have to be considered for all of the operations we might want to carry out. In implementing the developmental model examples described in Part II, I found several useful operations and manipulations which we would like to implement cleanly and efficiently. A user of the Framework should be able to perform any required exploration or alteration of the cell complex using only these operations, independent of the underlying implementation.

First are the basic topological operations: **boundary** and **coboundary** (Figure 4.1). Let C be a cell of dimension k. Then **boundary**(C) is the chain of oriented $(k-1)$-cells which make up the boundary of the k-cell C. Similarly, **coboundary**(C) is the *coboundary* of C, that is, the chain of all of the $(k+1)$-cells in whose boundary C lies. Also useful is **neighbours**; **neighbours**(C) is the set of all k-cells which share both a common boundary cell and a common coboundary cell with C. For a cell C of maximal dimension, **neighbours**(C) is the set of cells adjacent to C; for a vertex v, **neighbours**(v) is the set of all vertices joined to v by an edge. For cells of intermediate dimension, the property is less obvious, but there is a particularly simple intuition involving the combinatorial map (Section 4.5).

There are also a few other derived operations which query the structure. The **incident** operation determines whether two given cells are incident; thus, **incident**(C_0, C_1)



$$\begin{aligned}
\textbf{boundary}(f_A) &= e_{01} + e_{02} - e_{12} \\
\textbf{coboundary}(e_{12}) &= -f_A + f_B \\
\textbf{neighbours}(v_1) &= \{v_0, v_2, v_3\} \\
\\
\textbf{incident}(f_A, v_0) &= \textbf{true} \\
\textbf{border}(e_{01}) &= \textbf{true} \\
\textbf{border}(e_{12}) &= \textbf{false} \\
\textbf{meet}(e_{01}, f_B) &= v_1 \\
\textbf{join}(v_2, v_3) &= e_{23}
\end{aligned}$$

Figure 4.1: Query operations defined by the Cell Complex Framework, applied to a simple cell complex (left).

is **true** exactly when $C_0 \preceq C_1$ or $C_1 \preceq C_0$. The **border** operation reports whether a given cell lies on the boundary of the cell complex, while **meet** and **join** perform the lattice operations $\wedge$ and $\vee$ (Section 3.2). We should also be able to easily find the dimension of a cell; if C is a k-cell, then **dimension of** $C = k$.

Other operations are used to alter the cell complex (Figure 4.2). We need to be able to create new cells, as long as their boundary already exists; the **addCell** operation does this. To remove cells, we can use the **deleteCell** operation. We also recall, from Rosenfeld and Strong (1969), that any topological manipulation can be reduced to a series of binary splits and merges. The **splitCell** operation performs a binary division; it splits a k-cell into two child k-cells, separated by a $(k-1)$-cell, called the *membrane* of the split operation. The input to **splitCell** is the boundary of the membrane cell. Binary merges are handled by the **mergeCells** operation, which removes a membrane and rejoins two child cells.

We want to store numerical information on cells in the cell complex, and transfer information between these cells, so we need to know the relative orientations between them to ensure that signs are computed correctly. We also want to navigate around the cell complex; in particular, the cycles induced by alternating involutions of neighbouring dimension (Section 3.3) prove to be very useful. Beyond these, it is often useful to perform a breadth- or depth-first search of neighbouring cells. It is also useful in some cases to visit a number of adjacent *darts*.

Brisson (1990) discusses several possible implementations for his Cell-Tuple framework. These include a handle-based representation like G-maps, a database of all cell tuples, or a representation of the combinatorial map as a graph. He also suggests possible hybrid representations: for instance, the *augmented incidence graph*: an incidence graph in which each cell has pointers to structures representing the involution functions σ_i. I have developed a different data structure to implement the Cell Complex Framework: the *flip table*. This is a single simple data structure which simultaneously represents both incidence information and the involution functions σ_i. In the next section I will describe the data structure and its underlying elements, *flips*. In Section 4.3 I discuss how relative orientation information is maintained in the flip table. Section 4.4 describes the imple-

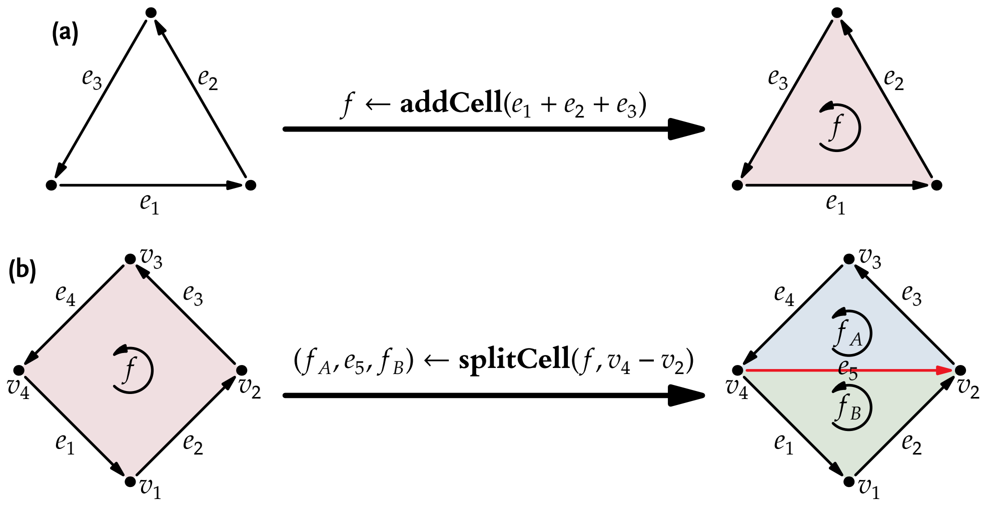

(a)
e_3
e_2
e_1
$f \leftarrow$ **addCell**$(e_1 + e_2 + e_3)$
f
(b)
v_3
v_4
v_2
v_1
e_4
e_3
e_1
e_2
$(f_A, e_5, f_B) \leftarrow$ **splitCell**$(f, v_4 - v_2)$
f_A
e_5
f_B

Figure 4.2: Cell Complex Framework operations which add new cells. (a) **addCell** creates a new face f with the supplied boundary chain $e_1 + e_2 + e_3$. (b) **splitCell** creates a new edge e_5 with the supplied boundary chain $v_4 - v_2$; this new edge is the *membrane* which splits the old face f into two child faces f_A and f_B.

mentation of *cell tuples*, which let us access the combinatorial map structure to traverse the cell complex. Section 4.5 describes the algorithms used to implement operations like **join** or **splitCell**. Finally, in Section 4.6 I talk about the actual implementation of the Framework in C⁺⁺ and about optimizations made to improve its performance.

4.2 Flips and the Flip Table

A common implementation of cell complex representations based on the combinatorial map (e.g. Brisson 1990, Lévy and Mallet 1999) is the handle-based data structure I described in Section 2.2.3. Every dart in the combinatorial map is explicitly represented by a data structure of the form

```
struct Dart {
  Dart *σ₀;
  Dart *σ₁;
  ⋮
  Dart *σₙ;

  CellData *c₀;
  ⋮
  CellData *cₙ; };
```

Each `Dart` object has $n + 1$ pointers to other `Dart` objects, the result of applying the appropriate involution. It also has pointers to data structures representing the $n + 1$ cells that are part of the dart. The cell complex is thus represented by a collection of `Dart` objects, connected by involutions σ_i.

The augmented incidence map suggested by Brisson (1990), on the other hand, has cells as objects:

```
struct Cell {
  Cell *boundary[];
  Cell *coboundary[];
  Cell *σ[][];

  CellData data; };
```

Each Cell has an array of links to its boundary and coboundary cells; this makes the data structure an incidence map. The data member σ is an array indexed by one boundary and one coboundary cell. When indexed by these cells, σ contains a pointer to the cell corresponding to this one in the dart produced after applying the σ involution in this cell's dimension. By Theorem 3.3, specifying one boundary and one coboundary cell uniquely defines these involutions. In the augmented incidence map, the cell complex is represented by a collection of `Cell` objects, again connected by involutions σ_i. Darts are not explicitly part of the data structure.

In the `Dart` data structure, the darts are objects; in the augmented incidence map, the cells are objects. In the flip table data structure, it is *the involutions themselves* which are the objects. Each involution of a given dart is represented by a minimal form, a *flip*, and the collection of all of these flips completely defines the involutions and thus the combinatorial map and the cell complex.

Recall that a single dart in an n-dimensional manifold is an n-simplex and is thus defined by the set of its $n + 1$ vertices. Each of these vertices corresponds to a cell of the cell complex, with one cell of each dimension. We can therefore write a single dart as the *cell tuple* $(c_0, c_1, \cdots, c_n)$. The involution σ_k relates this cell tuple to its k-neighbour, which differs only on its k-dimensional entry:

$$\sigma_k(c_0, \cdots, c_{k-1}, c_k, c_{k+1}, \cdots, c_n) = (c_0, \cdots, c_{k-1}, c'_k, c_{k+1}, \cdots, c_n).$$

According to Theorem 3.3, c'_k depends only on c_{k-1}, c_k, and c_{k+1}. Furthermore (by Theorem 3.1), c_k and c'_k are the only two cells in the boundary of c_{k+1} that are themselves bounded by c_{k-1}. This means that, for *any* cell tuple τ containing c_{k-1}, c_k, and c_{k+1}, $\sigma_k \circ \tau$ is the same tuple, substituting c'_k for c_k, and vice versa. The value of σ_k on any of these cell tuples τ depends only on the relationship between c_{k-1}, c_k, c'_k, and c_{k+1}, which I call the *flip relationship* and denote by its *flip*

$$\left\langle \begin{matrix} & c_{k+1} & \\ c_k & & c'_k \\ & c_{k-1} & \end{matrix} \right\rangle \quad \text{or} \quad \langle c_{k-1}, c_k \leftrightarrow c'_k, c_{k+1} \rangle.$$

The flip determines the value of σ_k applied to any tuple containing c_{k-1}, c_{k+1}, and one of c_k or c'_k. In other words, as noted above, *the σ functions are completely defined by the collection of all flips in the cell complex*. The Cell Complex Framework stores all of these flips in the *flip table*.[1]

[1] Of course, only a *set* of flips is needed, but for the purposes of optimization (Section 4.6) the flips are stored in an indexed table.

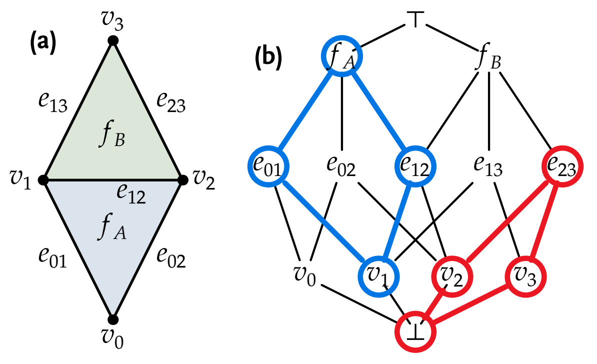
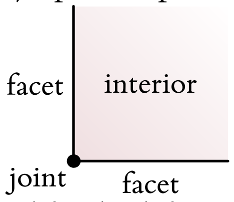


Figure 4.3: (a) A two-dimensional cell complex consisting of four vertices, five edges, and two faces. (b) The incidence graph corresponding to (a). Two flips are highlighted: $\langle v_1, e_{01} \leftrightarrow e_{12}, f_A\rangle$ and $\langle \perp, v_3 \leftrightarrow v_2, e_{23}\rangle$.

We call a flip which contains two possible values for c_k, and thus partially defines σ_k, a *k-flip*. The flip table defining an n-dimensional divided manifold contains 0-flips (or *vertex* flips), 1-flips (*edge* flips), 2-flips (*face* flips), up to n-flips. The cells in a flip are named

$$\left\langle \begin{array}{c} \text{interior} \\ \text{facet}\ \ \text{facet} \\ \text{joint} \end{array} \right\rangle$$



The order of the facets in the flip is immaterial for the definition of the involution; it is, however, used to encode orientation information (Section 4.3).

As an example, we will list the flips which define the 2-dimensional cell complex shown in Figure 4.3a. The simplest to start with are the edge flips. These are flips of the form $\langle v, e \leftrightarrow e', f\rangle$, where v is a vertex, e and e' are edges, and f is a 2-cell (face). There will be one edge flip for each incident pair of v and f; for instance, the two edges incident with both v_1 and f_A are e_{01} and e_{12}, giving a flip of

$$\left\langle \begin{array}{c} f_A \\ e_{01}\ \ e_{12} \\ v_1 \end{array} \right\rangle.$$

Note that this, like every flip, corresponds to a diamond of cells (blue) in the incidence graph (Figure 4.3b). Face f_A has two more vertices, v_0 and v_2, so there are a total of three flips with f_A in interior position:

$$\left\langle \begin{array}{c} f_A \\ e_{02}\ \ e_{01} \\ v_0 \end{array} \right\rangle, \left\langle \begin{array}{c} f_A \\ e_{01}\ \ e_{12} \\ v_1 \end{array} \right\rangle, \left\langle \begin{array}{c} f_A \\ e_{12}\ \ e_{02} \\ v_2 \end{array} \right\rangle.$$

We see that, in some sense, the set of flips with f in interior position defines the structure of the boundary of f; not only which cells make up that boundary, but how to traverse between them (across the joints). Another three flips have f_B in interior position, for a total of six edge flips in the complex:

$$\left\langle \begin{array}{c} f_A \\ e_{02}\ \ e_{01} \\ v_0 \end{array} \right\rangle, \left\langle \begin{array}{c} f_A \\ e_{01}\ \ e_{12} \\ v_1 \end{array} \right\rangle, \left\langle \begin{array}{c} f_A \\ e_{12}\ \ e_{02} \\ v_2 \end{array} \right\rangle, \quad \left\langle \begin{array}{c} f_B \\ e_{13}\ \ e_{23} \\ v_3 \end{array} \right\rangle, \left\langle \begin{array}{c} f_B \\ e_{12}\ \ e_{13} \\ v_1 \end{array} \right\rangle, \left\langle \begin{array}{c} f_B \\ e_{23}\ \ e_{12} \\ v_2 \end{array} \right\rangle.$$

Now we come to the vertex flips. These flips will have an edge in the interior position and vertices in the facet positions; these vertices will be the endpoints of the edge. The joint position should be filled by a (-1)-dimensional cell, which of course we don't have. Recall the discussion in Section 3.2 of the cell complex as a lattice. There we added pseudocells $\perp$ and $\top$ as the infimum and supremum of the incidence graph. We will continue this practice and fill the joint position in vertex flips with the pseudocell $\textcircled{$\perp$}$, pronounced "bottom". This pseudocell does not have an immediate geometric interpretation, but acts as an formal "shared boundary" for all vertices. The vertex flips in the complex are then

$$\left\langle\begin{matrix} & e_{01} & \\ v_0 & & v_1 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & e_{02} & \\ v_0 & & v_2 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & e_{12} & \\ v_1 & & v_2 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & e_{13} & \\ v_1 & & v_3 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & e_{23} & \\ v_3 & & v_2 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle;$$

note that, again, each serves to define the boundary of the edge in the "interior" position.

Finally, we look at the 2-flips. The only faces are f_A and f_B, which are adjacent over the edge e_{12}. This flip relationship has f_A and f_B in facet position, and e_{12} in joint position. The cell in interior position should be a 3-dimensional cell, but there are none in this 2-dimensional complex. Again, we introduce a pseudocell $\textcircled{$\top$}$, or "top", to fill this place:

$$\left\langle\begin{matrix} & \textcircled{$\top$} & \\ f_A & & f_B \\ & e_{12} & \end{matrix}\right\rangle.$$

$\textcircled{$\top$}$ has a clear geometric interpretation: it can be seen as an imaginary 3-cell whose boundary contains the 2-complex under consideration. (In general, $\textcircled{$\top$}$ in an n-dimensional complex is an $(n+1)$-dimensional pseudocell.)

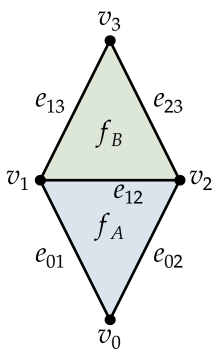
We could stop here; these twelve flips completely define the combinatorial map, and thus the cell complex. However, with only a few more flips we can make it much easier to handle the boundary of the complex. This information is present in the flips we already have: any edge which is in joint position of a 2-flip is in the interior, and any edge which does not show up in such a flip is in the exterior. We can make it easier to perform tasks such as traversing this boundary by introducing a third pseudocell $\textcircled{$\infty$}$, representing the 2-dimensional space outside the cell complex. Then we add 2-flips with the boundary edges in joint position, with the single incident face in one facet position, and $\textcircled{$\infty$}$ in the other:

$$\left\langle\begin{matrix} & \textcircled{$\top$} & \\ \textcircled{∞} & & f_A \\ & e_{01} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{$\top$} & \\ \textcircled{∞} & & f_A \\ & e_{02} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{$\top$} & \\ f_B & & \textcircled{∞} \\ & e_{13} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{$\top$} & \\ f_B & & \textcircled{∞} \\ & e_{23} & \end{matrix}\right\rangle.$$

Finally, we add edge flips with $\textcircled{$\infty$}$ as "interior":

$$\left\langle\begin{matrix} & \textcircled{∞} & \\ e_{02} & & e_{01} \\ & v_0 & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{∞} & \\ e_{01} & & e_{13} \\ & v_1 & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{∞} & \\ e_{23} & & e_{02} \\ & v_2 & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{∞} & \\ e_{13} & & e_{23} \\ & v_3 & \end{matrix}\right\rangle.$$

The entire cell complex is then represented by these twenty flips (Figure 4.4).



$$
\left\langle\begin{matrix} e_{01} \\ v_1 \;\; v_0 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{02} \\ v_0 \;\; v_2 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{12} \\ v_1 \;\; v_2 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{13} \\ v_1 \;\; v_3 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{23} \\ v_3 \;\; v_2 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} f_A \\ e_{02} \;\; e_{01} \\ v_0 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_A \\ e_{01} \;\; e_{12} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_A \\ e_{12} \;\; e_{02} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_B \\ e_{13} \;\; e_{23} \\ v_3 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_B \\ e_{12} \;\; e_{13} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_B \\ e_{23} \;\; e_{12} \\ v_2 \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} \textcircled{∞} \\ e_{02} \;\; e_{01} \\ v_0 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{01} \;\; e_{13} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{23} \;\; e_{02} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{13} \;\; e_{23} \\ v_3 \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_A \;\; f_B \\ e_{12} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ \textcircled{∞} \;\; f_A \\ e_{01} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ \textcircled{∞} \;\; f_A \\ e_{02} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_B \;\; \textcircled{∞} \\ e_{13} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_B \;\; \textcircled{∞} \\ e_{23} \end{matrix}\right\rangle.
$$

Figure 4.4: A simple cell complex and the twenty flips which serve to define it. The order of facets within each flip are irrelevant for the definition of the cell complex, but serve to define the relative orientations between cells (Section 4.3).

The flips defining a cell complex are stored in a searchable collection, the *flip table*. The flip table can be searched by queries against template flips which may contain a wild-card pseudocell $\textcircled{?}$. Thus, performing a flip operation requires matching against the template $\langle c_{i-1}, c_i \leftrightarrow \textcircled{?}, c_{i+1}\rangle$. For example, to find the value of σ_1 from the cell tuple (v_0, e_{01}, f_A) in the above cell complex, we match against the template $\langle v_0, e_{01} \leftrightarrow \textcircled{?}, f_A\rangle$. This matches exactly one flip, $\langle v_0, e_{02} \leftrightarrow e_{01}, f_A\rangle$ (note that the facets can match in either order), so we replace e_{01} in the tuple by e_{02}, and find that $\sigma_1(v_0, e_{01}, f_A) = (v_0, e_{02}, f_A)$.

As another example, we can find cells on the boundary by matching against the template $\langle \textcircled{?}, \textcircled{?} \leftrightarrow \textcircled{∞}, \textcircled{?}\rangle$. This template matches

$$
\left\langle\begin{matrix} \textcircled{$\top$} \\ \textcircled{∞} \;\; f_A \\ e_{01} \end{matrix}\right\rangle, \quad
\left\langle\begin{matrix} \textcircled{$\top$} \\ \textcircled{∞} \;\; f_A \\ e_{02} \end{matrix}\right\rangle, \quad
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_B \;\; \textcircled{∞} \\ e_{13} \end{matrix}\right\rangle, \quad \text{and} \quad
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_B \;\; \textcircled{∞} \\ e_{23} \end{matrix}\right\rangle.
$$

We can then collect the cells in joint or facet position to find that the boundary of the entire cell complex consists of the edges e_{01}, e_{02}, e_{13}, and e_{23}, and that the faces which adjoin the boundary are f_A and f_B.

4.3 Orientation

Recall from Section 3.2.2 that an n-cell c and an $(n-1)$-cell f in its boundary have a *relative orientation* $\rho(c, f)$ which is either $+1$ (meaning the two cells are consistently oriented) or -1 (meaning the two cells are inconsistently oriented). The single piece of ambiguity in the flip table representation is the order of the facets. In this section, I will show that in fact this order can be used to encode the orientation of the flips themselves. This flip orientation can be then used to record the relative orientations between all pairs of cells in the complex.

The boundary of c can be represented by the chain

$$\partial c = \sum_i \rho(t, f_i) f_i$$

where $\{f_0, f_1, \dots, f_m\}$ are the $(n-1)$-cells in c's boundary. Remember that the boundary of any n-cell is an $(n-1)$-dimensional manifold *without boundary*, so applying the boundary operator ∂ again should give zero:

$$\begin{aligned} 0 = \partial\partial c &= \sum_i \rho(c, f_i)\, \partial f_i \\ &= \sum_i \rho(c, f_i) \sum_j \rho(f_i, g_{ij})\, g_{ij} \end{aligned} \tag{4.1}$$

where $\{g_{i0}, g_{i1}, \dots\}$ are the $(n-2)$-cells in f_i's boundary. We reverse the order of the summation: rather than summing first over the f cells, then over the incident g cells of each, we rewrite Equation 4.1 to sum first over g, then over their incident f cells.

$$0 = \sum_j \left(\sum_i \rho(g_j, f_{ji}) \rho(c, f_{ji}) \right) g_j.$$

But there are exactly two f cells whose boundaries contain each g cell; this in fact defines a flip

$$\left\langle \begin{matrix} & c & \\ f_{j0} & & f_{j1} \\ & g_j & \end{matrix} \right\rangle$$

where we have named the two f cells associated with g_j f_{j0} and f_{j1}.

We thus see that

$$0 = \sum_j \big(\rho(g_j, f_{j0}) \rho(f_{j0}, c) + \rho(g_j, f_{j1}) \rho(f_{j1}, c) \big)\, g_j;$$

as the coefficient of each g_j must be zero, then, for every flip $\langle g, f_0 \leftrightarrow f_1, t \rangle$ it must be the case that

$$\begin{aligned} \rho(g, f_0)\rho(f_0, c) + \rho(g, f_1)\rho(f_1, c) &= 0 \\ \rho(g, f_0)\rho(f_0, c) &= -\rho(g, f_1)\rho(f_1, c) \\ \rho(g, f_0)\rho(f_0, c)\rho(g, f_1)\rho(f_1, c) &= -1. \end{aligned} \tag{4.2}$$

In other words, for any flip $\langle g, f_0 \leftrightarrow f_1, t \rangle$ the product of the relative orientations of the four described incidence relationships must be negative. There are eight possible cases; four cases in which three of the orientations are positive and one negative, and four cases with three negative orientations and one positive orientation:

$$\left\langle \begin{matrix} {}_{-}c_{+} \\ f_0\ f_1 \\ {}^{+}j^{+} \end{matrix} \right\rangle, \left\langle \begin{matrix} {}_{+}c_{+} \\ f_0\ f_1 \\ {}^{-}j^{+} \end{matrix} \right\rangle, \left\langle \begin{matrix} {}_{+}c_{-} \\ f_0\ f_1 \\ {}^{-}j^{-} \end{matrix} \right\rangle, \left\langle \begin{matrix} {}_{-}c_{-} \\ f_0\ f_1 \\ {}^{+}j^{-} \end{matrix} \right\rangle, \quad \left\langle \begin{matrix} {}_{+}c_{-} \\ f_0\ f_1 \\ {}^{+}j^{+} \end{matrix} \right\rangle, \left\langle \begin{matrix} {}_{+}c_{+} \\ f_0\ f_1 \\ {}^{+}j^{-} \end{matrix} \right\rangle, \left\langle \begin{matrix} {}_{-}c_{+} \\ f_0\ f_1 \\ {}^{-}j^{-} \end{matrix} \right\rangle, \left\langle \begin{matrix} {}_{-}c_{-} \\ f_0\ f_1 \\ {}^{-}j^{+} \end{matrix} \right\rangle. \tag{4.3}$$

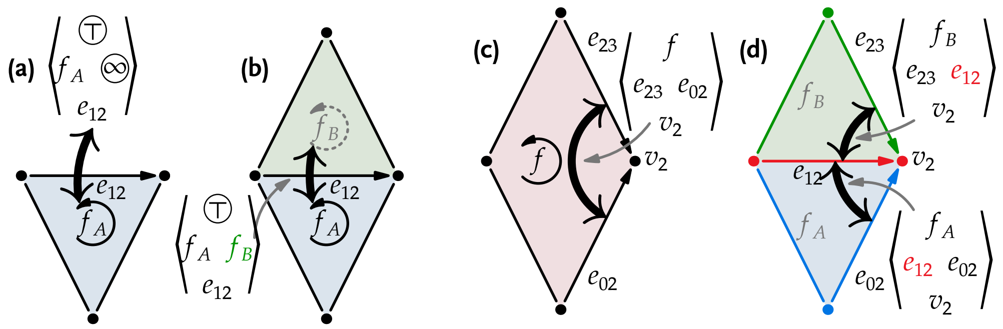

(a)
$\left\langle \begin{array}{cc} \multicolumn{2}{c}{\textcircled{$\top$}} \\ f_A & \textcircled{∞} \\ \multicolumn{2}{c}{e_{12}} \end{array} \right\rangle$
e_{12}
f_A
(b)
f_B
e_{12}
f_A
$\left\langle \begin{array}{cc} \multicolumn{2}{c}{\textcircled{$\top$}} \\ f_A & f_B \\ \multicolumn{2}{c}{e_{12}} \end{array} \right\rangle$
(c)
e_{23}
$\left\langle \begin{array}{cc} \multicolumn{2}{c}{f} \\ e_{23} & e_{02} \\ \multicolumn{2}{c}{v_2} \end{array} \right\rangle$
f
v_2
e_{02}
(d)
e_{23}
$\left\langle \begin{array}{cc} \multicolumn{2}{c}{f_B} \\ e_{23} & e_{12} \\ \multicolumn{2}{c}{v_2} \end{array} \right\rangle$
f_B
v_2
e_{12}
f_A
$\left\langle \begin{array}{cc} \multicolumn{2}{c}{f_A} \\ e_{12} & e_{02} \\ \multicolumn{2}{c}{v_2} \end{array} \right\rangle$
e_{02}

Figure 4.5: Some of the flip modifications which happen during **addCell** and **splitCell** operations. (a) The face f_A is initially adjacent to the outside pseudocell $\textcircled{\infty}$ across edge e_{12}; (b) after adding the face f_B, its adjacency to f_A is recorded by replacing $\textcircled{\infty}$ in the flip with f_B. Relative orientations on the left side of the flip are unaltered. (c) Before splitting the face f, the edges e_{02} and e_{23} flip to one another across the vertex v_2; (d) after splitting, two flips are created by replacing a single edge only in f_A or f_B by the shared edge e_{12}. Since both child faces have the same orientation as their parent, orientations down one side of the flip are again unaltered.

In each of these flips, a + or – has been placed between each pair of cells representing their relative orientation. For example, in the first case shown in Equation 4.3, the relative orientation between c and f_0 is negative, while the relative orientations between f_0 and j, between c and f_1, and between f_1 and j are all positive.

The orientation information could conceivably be stored with the flip in many ways; we could store all four relative orientations, or just three bits indicating which of the eight above cases holds. However, we note that a flip has exactly one bit of ambiguity: the order of the facets. This suggests that we can use the facet order to store a single orientation. We can look at how the flip table changes under cell complex operations to figure out precisely which orientation to store (Figure 4.5).

During both **addCell** and **splitCell** operations (Section 4.5), many of the new flips created are just modifications of existing flips. For instance, when a new face f_B is created (Figures 4.5ab), face f_A that was formerly adjacent to the external pseudocell $\textcircled{\infty}$ is now adjacent to the new face. The face flip between f_A and f_B is created by simply taking the defunct flip between f_A and $\textcircled{\infty}$ and replacing $\textcircled{\infty}$ by f_B:

$$\left\langle \begin{array}{ccc} - & \textcircled{$\top$} & + \\ f_A & & \textcircled{∞} \\ - & e_{12} & - \end{array} \right\rangle \Rightarrow \left\langle \begin{array}{ccc} - & \textcircled{$\top$} & ? \\ f_A & & f_B \\ - & e_{12} & ? \end{array} \right\rangle. \tag{4.4}$$

Note that the left-hand side of the flip is unchanged; both of the unknown relative orientations are on the right-hand side.

Similarly, when a face f is split (Figures 4.5cd), new flips involving the new membrane edge e_{12} are created by duplicating original flips involving f, with the entry for f in one changed to f_A and in the other to f_B. The facet corresponding to the edge of the other face is then replaced by the common edge e_{12}:

$$\left\langle \begin{array}{ccc} - & f & + \\ e_{23} & & e_{02} \\ - & v_2 & - \end{array} \right\rangle \Rightarrow \left\langle \begin{array}{ccc} ? & f_A & + \\ e_{12} & & e_{02} \\ ? & v_2 & - \end{array} \right\rangle, \left\langle \begin{array}{ccc} - & f_B & ? \\ e_{23} & & e_{12} \\ - & v_2 & ? \end{array} \right\rangle. \tag{4.5}$$

During a **splitCell** operation, the orientation of the child cells is the same as that of the parent, so again both of the unknown relative orientations are on one side or another of the flip.

In all of these flip modifications, then, we see that one side of the flip preserves its relative orientations, while on the other side the relative orientations are unknown. However, because the product of all four relative orientations is constant, we know that the *product* of the two relative orientations on each side is *unchanged*. If a flip's orientation is one of these products then we can perform flip table manipulations (4.4) and (4.5) without changing that orientation.

I therefore define the *orientation of a flip* $\vartheta\langle j, f_0 \leftrightarrow f_1, c\rangle$ as the product of the relative orientations down the left side; that is, $\vartheta\langle j, f_0 \leftrightarrow f_1, c\rangle = \rho(j, f_0)\,\rho(f_0, c)$. The flip with facets reversed then has the opposite orientation; $\vartheta\langle j, f_0 \leftrightarrow f_1, c\rangle = -\vartheta\langle j, f_1 \leftrightarrow f_0, c\rangle$. We say that a flip is in *normal form* if its orientation is positive. All flips are stored in the flip table in normal form.

To determine the relative orientation between any pair of cells, we can work from known relative orientations. First, we arbitrarily state that the relative orientation between any vertex v and $\circledperp$ is positive. Then

$$\vartheta\left\langle \begin{matrix} & e & \\ v_0 & & v_1 \\ & \circledperp & \end{matrix} \right\rangle = \rho(\circledperp, v_0)\,\rho(v_0, e) = \rho(v_0, e).$$

Now if we need to find the relative orientation between e and some cell f in the coboundary of e, we need only find the flip $\langle v_0, e \leftrightarrow e', f\rangle$; then we see that

$$\vartheta\left\langle \begin{matrix} & f & \\ e & & e' \\ & v_0 & \end{matrix} \right\rangle = \rho(v_0, e)\,\rho(e, f) \qquad \text{so} \qquad \rho(e, f) = \vartheta\left\langle \begin{matrix} & f & \\ e & & e' \\ & v_0 & \end{matrix} \right\rangle \vartheta\left\langle \begin{matrix} & e & \\ v_0 & & v_1 \\ & \circledperp & \end{matrix} \right\rangle.$$

In general, if there is a sequence of cells $c_0 \prec c_1 \prec \cdots \prec c_k \prec c_{k+1}$ and flips $\mathscr{F}_i = \langle c_{i-1}, c_i \leftrightarrow c_i', c_{i+1}\rangle$, the relative orientation between c_k and c_{k+1} is $\rho(c_k, c_{k+1}) = \prod_i \vartheta\mathscr{F}_i$. This is exactly the structure of the *flip tower* (Section 4.4); the Cell Complex Framework constructs a flip tower containing c_k and c_{k+1} in order to evaluate their relative orientation.

4.4 Cell Tuples

In Section 4.1, I mentioned that we want a way to access the underlying combinatorial map structure in order to traverse the cell complex, particularly to perform iterations around the boundaries of cells. The Cell Complex Framework provides *cell tuples* as the data structure which provides this ability. In this section, I will describe how cell tuples are created and modified, as well as the operations which they offer. The underling data structure, the *flip tower*, is also used in implementing other basic cell complex operations, such as finding relative orientations (Section 4.3) and testing for incidence (Section 4.5).

Recall that cell tuples are the software realization of darts and are defined by a sequence of cells c_0 through c_n:

```
struct CellTuple {
  Cell *c_0;
  ⋮
  Cell *c_n; };
```

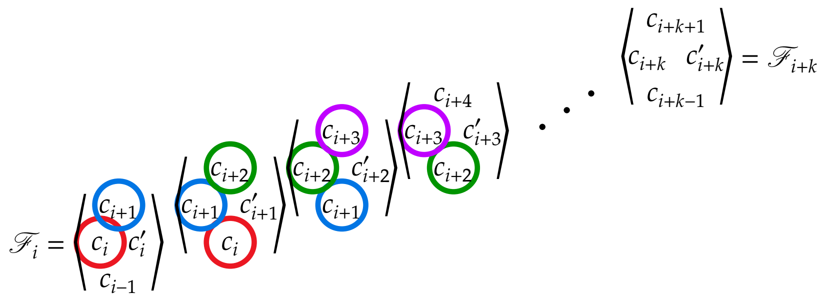
The cells defining a cell tuple τ can be accessed by the user as the indices of τ; thus, $\tau[0]$ is the cell tuple's vertex, while $\tau[k]$ is the tuple's k-cell. The flip operation $\tau.\mathbf{flip}(k)$ returns the result of applying σ_k to τ. This operation changes the cell c_k in the tuple to the unique cell $c'_k \neq c_k$ with $c_{k-1} \prec c'_k \prec c_{k+1}$. This is, of course, exactly the relationship stored in an entry in the flip table. The operation is performed by finding the corresponding flip in the flip table; as described in Section 4.2, this can be done by searching the flip table for the flip matching the template $\langle c_{k-1}, c_k \leftrightarrow ⓘ, c_{k+1}\rangle$. If an entire new cell tuple is not needed, the cell c'_k can be accessed as $\tau.\mathbf{other}(k)$.

The Framework lets a cell tuple be created given any subset of the cells it contains. If the cells do not uniquely define a single tuple, then an arbitrary tuple containing all of those cells is created. Thus, for instance, the operation **tuple containing** $\{v_0\}$ will return one of the many tuples containing v_0, while **tuple containing** $\{\}$ will return one arbitrary tuple from the entire cell complex. A cell tuple cannot be made from just any set of cells, however: the cells must all be incident. In order to ensure the correctness of cell tuples, the Cell Complex Framework builds a cell tuple out of a *flip tower* which supports it as a *cell sequence*.

A *cell sequence* is a sequence of cells $c_i \prec c_{i+1} \prec \ldots \prec c_{i+k}$, where c_ℓ is an ℓ-cell. For example, a cell tuple is a cell sequence $c_0 \prec \ldots \prec c_n$ with $n+1$ elements. We can define a sequence of flips $\{\mathscr{F}_i\}$ associated with a particular cell sequence as follows:

- $\mathscr{F}_i$ is one of the flips matching $\langle ⓘ, c_i \leftrightarrow ⓘ, c_{i+1}\rangle$;
- $\mathscr{F}_j$, where $i < j < i+k$, is the unique flip $\langle c_{j-1}, c_j \leftrightarrow c'_j, c_{j+1}\rangle$;
- $\mathscr{F}_{i+k}$ is one of the flips matching $\langle c_{i+k-1}, c_{i+k} \leftrightarrow ⓘ, ⓘ\rangle$.

The result is a sequence of flips which share cells in an overlapping manner:



I call such a sequence of overlapping flips a *flip tower*, and say that it *supports* the cell sequence $c_i \prec c_{i+1} \prec \ldots \prec c_{i+k}$. Note that all of the flips, except the first and last, are determined uniquely. In addition, for a cell tuple, we note that the flips $\mathscr{F}_0$ and $\mathscr{F}_n$ are determined too, as there is only one flip with interior c_1 and facet c_0 ($\langle ⊥, c_0 \leftrightarrow c'_0, c_1\rangle$) and only one flip with facet c_n and joint c_{n-1} ($\langle c_{n-1}, c_n \leftrightarrow c'_n, ⊤\rangle$). Thus a cell tuple is supported by a unique flip tower, and in fact $\mathscr{F}_i$ is exactly the flip which defines the operation $\mathbf{flip}(i)$.

The operation **tuple containing** attempts to construct a complete flip tower joining the given cells. The construction process naturally produces the incident cells of all of the missing dimensions. The flips which connect the cells are determined in different ways, depending where in the tower they lie.

$$\begin{array}{lcl}
\textcircled{$\top$} & & \\
c_2 = \textcircled{?} & \left\langle\begin{matrix} & \textcircled{$\top$} & \\ \textcircled{?} & & \textcircled{?} \\ & e_{12} & \end{matrix}\right\rangle & = \mathscr{F}_2 \\
c_1 = e_{12} & & \\
c_0 = \textcircled{?} & \left\langle\begin{matrix} & e_{12} & \\ \textcircled{?} & & \textcircled{?} \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle & = \mathscr{F}_0 \\
\textcircled{$\perp$} & &
\end{array}$$

A run of a single missing cell is the easiest to fill. For instance, suppose a cell tuple is to be created from the cell complex shown in Figure 4.4, containing the specified cell e_{12}. The cells c_0 and c_2 are unknown. The flip tower entry $\mathscr{F}_0$ has interior cell e_{12} and joint $\textcircled{$\perp$}$; similarly, $\mathscr{F}_2$ has interior cell $\textcircled{$\top$}$ and joint e_{12}. The unknown cells are replaced by $\textcircled{?}$ and the flips are used as search templates.

The $\mathscr{F}_0$ template matches the flip $\langle \textcircled{$\perp$}, v_1 \leftrightarrow v_2, e_{12}\rangle$, while the $\mathscr{F}_2$ template matches $\langle e_{12}, f_A \leftrightarrow f_B, \textcircled{$\top$}\rangle$. The unknown tuple entries are filled with the facet cells v_1 and f_A. (Which of the facet cells is picked is arbitrary.) The flip $\mathscr{F}_1$ then has joint v_1, interior cell f_A, and one facet e_{12}; adding a $\textcircled{?}$ for the unknown other facet gives us a search template to find the final entry in the flip tower.

$$\begin{array}{lcl}
c_2 = f_A & \left\langle\begin{matrix} & \textcircled{$\top$} & \\ f_A & & f_B \\ & e_{12} & \end{matrix}\right\rangle & = \mathscr{F}_2 \\
c_1 = e_{12} & \left\langle\begin{matrix} & f_A & \\ e_{12} & & \textcircled{?} \\ & v_1 & \end{matrix}\right\rangle & = \mathscr{F}_1 \\
c_0 = v_1 & \left\langle\begin{matrix} & e_{12} & \\ v_1 & & v_2 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle & = \mathscr{F}_0
\end{array}$$

The template matches exactly one flip (as it must), and we are left with both the tuple and its supporting flip tower:

$$\tau = \left(v_1, e_{12}, f_A\right), \qquad \mathscr{F}_i = \left(\left\langle\begin{matrix} & e_{12} & \\ v_1 & & v_2 \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle, \left\langle\begin{matrix} & f_A & \\ e_{12} & & e_{01} \\ & v_1 & \end{matrix}\right\rangle, \left\langle\begin{matrix} & \textcircled{$\top$} & \\ f_A & & f_B \\ & e_{12} & \end{matrix}\right\rangle\right).$$

$$\begin{array}{lcl}
c_2 = f_B & & \\
c_1 = \textcircled{?} & \left\langle\begin{matrix} & f_B & \\ \textcircled{?} & & \textcircled{?} \\ & \textcircled{?} & \end{matrix}\right\rangle & = \mathscr{F}_1 \\
c_0 = \textcircled{?} & &
\end{array}$$

A longer run of unknown cells can be filled iteratively. Suppose the cell tuple is to be created from f_B, with c_0 and c_1 initially unknown. Only the interior cell in $\mathscr{F}_1$ is then known; the other cells are filled by $\textcircled{?}$ and the flip is used as a search template.

The $\mathscr{F}_1$ template matches several flips; we arbitrarily select $\langle v_3, e_{13} \leftrightarrow e_{23}, f_B\rangle$. We can then set $c_0 = v_3$ and $c_1 = e_{13}$. The remaining entries in the flip tower will then match the templates $\mathscr{F}_0 = \langle \textcircled{$\perp$}, v_3 \leftrightarrow \textcircled{?}, e_{13}\rangle$ and $\mathscr{F}_2 = \langle e_{13}, f_B \leftrightarrow \textcircled{?}, \textcircled{$\top$}\rangle$.

$$\begin{array}{lcl}
c_2 = f_B & \left\langle\begin{matrix} & \textcircled{$\top$} & \\ f_B & & \textcircled{?} \\ & e_{13} & \end{matrix}\right\rangle & = \mathscr{F}_2 \\
c_1 = e_{13} & \left\langle\begin{matrix} & f_B & \\ e_{13} & & e_{23} \\ & v_3 & \end{matrix}\right\rangle & = \mathscr{F}_1 \\
c_0 = v_3 & \left\langle\begin{matrix} & e_{13} & \\ v_3 & & \textcircled{?} \\ & \textcircled{$\perp$} & \end{matrix}\right\rangle & = \mathscr{F}_0
\end{array}$$

We find the unique flips matching these templates to see that the cell tuple and supporting flip tower are

$$\tau = \left(v_3, e_{13}, f_B\right), \qquad \mathscr{F}_i = \left(\left\langle \begin{matrix} & e_{13} & \\ v_3 & & v_1 \\ & \textcircled{$\bot$} & \end{matrix} \right\rangle, \left\langle \begin{matrix} & f_B & \\ e_{13} & & e_{23} \\ & v_3 & \end{matrix} \right\rangle, \left\langle \begin{matrix} & \textcircled{$\top$} & \\ f_B & & \textcircled{∞} \\ & e_{13} & \end{matrix} \right\rangle \right).$$

4.5 Operations on cells

As mentioned in Section 4.1, the user-facing operations of the Cell Complex Framework operate on cells and cell chains. They notably do *not* operate on flips; these operations are meant to be atomic and should be independent of the underlying representation of the cell complex. In this section, I will cover how these operations are carried out under the flip table representation.

Query operators First are the topological queries **boundary**, **coboundary**, and **neighbours**. The first two give the boundary and coboundary chain, respectively. These are implemented by simply collating information from flips involving the queried cell. For instance, to find the boundary of cell f_A in Figure 4.4, we find all flips with f_A in interior position:

$$\left\langle \begin{matrix} & f_A & \\ e_{01} & & e_{02} \\ & v_0 & \end{matrix} \right\rangle, \qquad \left\langle \begin{matrix} & f_A & \\ e_{01} & & e_{12} \\ & v_1 & \end{matrix} \right\rangle, \qquad \left\langle \begin{matrix} & f_A & \\ e_{02} & & e_{12} \\ & v_2 & \end{matrix} \right\rangle$$

and collect a set of the facets from these flips: $\{e_{01}, e_{02}, e_{12}\}$. By building flip towers downward we can then find the relative orientations between these facets and f_A and combine all of this information into the answer:

$$\mathbf{boundary}(f_A) = e_{01} + e_{02} - e_{12}.$$

The **neighbours** operation finds all cells which share both a bounding cell and a cobounding cell with the given cell. The neighbours of a vertex are those vertices on the other end of edges incident on the given vertex; this is exactly the vv definition of neighbouring vertices. The neighbours of a cell of maximal dimension are the cells it is adjacent to; again, this is a natural definition of neighbouring cells. For cells of intermediate dimension the **neighbours** operation is less useful. The general definition is used because the cells which match are in fact exactly *those which the given cell can flip to*. For example, the neighbours of the vertex v_1 in Figure 4.4 are found by retrieving all flips with v_1 in facet position:

$$\left\langle \begin{matrix} & e_{01} & \\ v_1 & & v_0 \\ & \textcircled{$\bot$} & \end{matrix} \right\rangle, \qquad \left\langle \begin{matrix} & e_{12} & \\ v_1 & & v_2 \\ & \textcircled{$\bot$} & \end{matrix} \right\rangle, \qquad \left\langle \begin{matrix} & e_{13} & \\ v_1 & & v_3 \\ & \textcircled{$\bot$} & \end{matrix} \right\rangle$$

then collecting all of the other facets: $\{v_0, v_2, v_3\}$.

The next query operations are **incident** and **border**. The **incident** test determines whether the two cells it is given are incident. This is done by attempting to construct a partial flip tower (Section 4.4) in exactly the same way as cells are interpolated when constructing a cell tuple. The cells are incident only if the partial cell tower can be built. For example, to determine whether cells v_0 and f_A are incident, we must find the 1-cell between them. This is done by finding a flip which matches the template $\langle v_0, \textcircled{?} \leftrightarrow \textcircled{?}, f_A\rangle$. Since $\langle v_0, e_{01} \leftrightarrow e_{02}, f_A\rangle$ matches, we can conclude that

$$\textbf{incident}(v_0, f_A) = \textbf{true}.$$

We only have to find this single flip; this is enough of the flip tower to confirm the incidence.

The **border** test determines if the given cell is on the border of the complex. For cells below maximal dimension, this means that the cell is part of the boundary manifold of the entire cell complex. A cell of maximal dimension is considered to be on the border if the outside of the complex is across one of its walls. Both of these cases are handled using the $\textcircled{\infty}$ pseudocell. Then a cell of maximal dimension is on the border if it flips to $\textcircled{\infty}$, while a cell of below maximal dimension is on the border if it is incident to $\textcircled{\infty}$. For example, for the cell complex in Figure 4.4 the flip $\langle e_{01}, f_A \leftrightarrow \textcircled{\infty}, \textcircled{\top}\rangle$ shows both that

$$\textbf{border}(f_A) = \textbf{true}$$

and, as part of the flip tower joining e_{01} with $\textcircled{\infty}$, that

$$\textbf{border}(e_{01}) = \textbf{incident}(e_{01}, \textcircled{\infty}) = \textbf{true}.$$

meet and join The last two query operations are **meet** and **join**. Their implementations are somewhat more complex than the other queries because they require us to find paths from both cells to an unknown common cell, out of all possible paths in the incidence graph. In the current implementation this is done by constructing *all possible flip towers* from the query cells, then matching them at the first opportunity. In effect, this reconstructs the incidence graph starting from the query cells. As an example, consider finding the **meet** of the cells f_B and e_{01} from the cell complex in Figure 4.4.

$$\begin{tikzcd} & f_B \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & & \\ e_{12} \arrow[d, dash] \arrow[dr, dash] & e_{13} \arrow[dl, dash] \arrow[dr, dash] & e_{23} \arrow[dl, dash] \arrow[d, dash] & e_{01} \\ v_1 & v_2 & v_3 & \end{tikzcd}$$

We start by working down from f_B. There are three flips with f_B in interior position:

$$\left\langle \begin{matrix} & f_B & \\ e_{13} & & e_{23} \\ & v_3 & \end{matrix} \right\rangle, \quad \left\langle \begin{matrix} & f_B & \\ e_{12} & & e_{13} \\ & v_1 & \end{matrix} \right\rangle, \quad \text{and} \quad \left\langle \begin{matrix} & f_B & \\ e_{12} & & e_{23} \\ & v_2 & \end{matrix} \right\rangle.$$

We can thus connect three edges and three vertices to f_B. None of the edges is e_{01}, so we must expand downwards from there next.

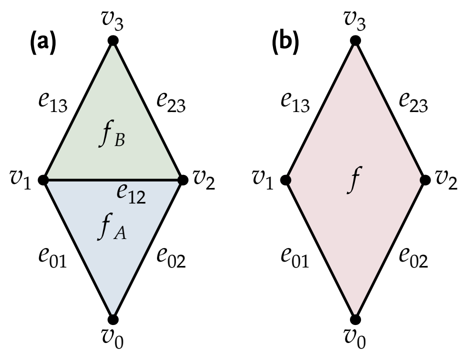
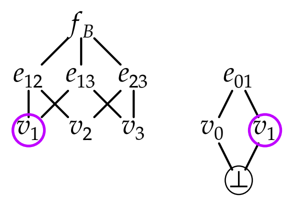


Figure 4.6: The **join** of two cells on the boundary of a cell complex. (a) As no real cell joins them, **join**$(v_0, v_3) = \textcircled{\infty}$. (b) We prefer to return a real cell from a query, if possible. Since they are joined by a real cell, **join**$(v_0, v_3) = f$, even though $\textcircled{\infty}$ still cobounds them both.

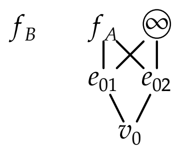
There is only one flip with e_{01} in interior position:

$$\left\langle \begin{array}{c} e_{01} \\ v_1 \quad v_0 \\ \textcircled{\perp} \end{array} \right\rangle.$$



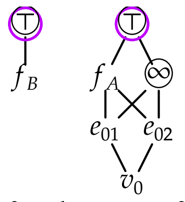
This connects two vertices to e_{01}, and we compare the vertices to see that v_1 is present in paths from both f_B and e_{01}. We can therefore conclude that **meet**$(e_{01}, f_B) = v_1$.

As another example, consider finding the **join** of the cells f_B and v_0. We work upward in this case, starting from the cell of lower dimension. There are two flips with v_0 in interior position:



$$\left\langle \begin{array}{c} f_A \\ e_{01} \quad e_{02} \\ v_0 \end{array} \right\rangle \quad \text{and} \quad \left\langle \begin{array}{c} \textcircled{\infty} \\ e_{01} \quad e_{02} \\ v_0 \end{array} \right\rangle.$$

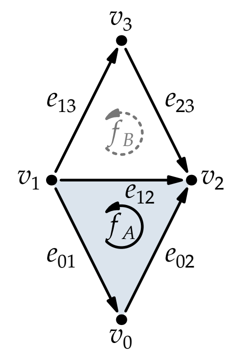
Two edges, one face, and the pseudocell $\textcircled{\infty}$ are thus connected to v_0. f_B is not the face, so we expand further.



If this was a three-dimensional cell complex, we would find the flips with f_B, f_A, and $\textcircled{\infty}$ in joint position to continue building the graphs upward. In two dimensions, however, the only coboundary of a 2-cell is $\textcircled{\top}$, so the graphs end there. We can therefore conclude that **join**$(v_0, f_B) = \textcircled{\top}$.

Note the presence of $\textcircled{\infty}$ in the previous example. It is possible for the **join** of two cells to be $\textcircled{\infty}$; for instance, both v_0 and v_3 in Figure 4.4 are on the boundary of the cell complex, so **join**$(v_0, v_3) = \textcircled{\infty}$. We would prefer to return a real cell from any query, if possible; we therefore return $\textcircled{\infty}$ only if there is no real cell which joins the input cells (Figure 4.6).

Adding cells The first operation which modifies the cell complex is **addCell**, which creates a new cell, given its boundary chain. This operation is performed by creating all of the flips which involve the new cell, then adding them to the cell complex's flip table. The new cell may also be filling in space formerly occupied by $\textcircled{\infty}$; in this case, flips involving $\textcircled{\infty}$ may need to be deleted. In the process of creating the flips, the validity of



$$
\left\langle\begin{matrix} e_{01} \\ v_1\ \ v_0 \\ \textcircled{\perp} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{02} \\ v_0\ \ v_2 \\ \textcircled{\perp} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{12} \\ v_1\ \ v_2 \\ \textcircled{\perp} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{13} \\ v_1\ \ v_3 \\ \textcircled{\perp} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{23} \\ v_3\ \ v_2 \\ \textcircled{\perp} \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} f_A \\ e_{02}\ \ e_{01} \\ v_0 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_A \\ e_{01}\ \ e_{12} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_A \\ e_{12}\ \ e_{02} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{\infty} \\ e_{02}\ \ e_{01} \\ v_0 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{\infty} \\ e_{01}\ \ e_{12} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{\infty} \\ e_{12}\ \ e_{02} \\ v_2 \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} \textcircled{\top} \\ f_A\ \ \textcircled{\infty} \\ e_{12} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{\top} \\ \textcircled{\infty}\ \ f_A \\ e_{01} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{\top} \\ \textcircled{\infty}\ \ f_A \\ e_{02} \end{matrix}\right\rangle.
$$

Figure 4.7: A cell complex and the flips which define it. The new face f_B will be added with boundary $e_{12} - e_{13} - e_{23}$.

the supplied boundary chain can be verified: it must be closed, it must be in one segment, and the stated orientations must be consistent.

As an example, consider adding a face to the cell complex depicted in Figure 4.7. We wish to add the new face f_B, bounded by edges e_{12}, e_{13}, and e_{23}. The face is to be oriented counterclockwise, so its boundary chain is $e_{12} - e_{13} - e_{23}$. We thus want to perform the operation $f_B \leftarrow$ **addCell**$(e_{12} - e_{13} - e_{23})$.

First among the new flips to be added are those with f_B in interior position (Figure 4.8a). Their facets will be cells from f_B's boundary, that is from $\{e_{12}, e_{13}, e_{23}\}$. The joints of the new flips will be from the boundary of the boundary cells. We therefore look at the flips with these edges in interior position:

$$
\left\langle\begin{matrix} e_{12} \\ v_1\ \ v_2 \\ \textcircled{\perp} \end{matrix}\right\rangle, \qquad
\left\langle\begin{matrix} e_{13} \\ v_1\ \ v_3 \\ \textcircled{\perp} \end{matrix}\right\rangle, \qquad \text{and} \qquad
\left\langle\begin{matrix} e_{23} \\ v_2\ \ v_3 \\ \textcircled{\perp} \end{matrix}\right\rangle.
$$

There are three different vertices in these flips: v_1, v_2, and v_3. Importantly, each is in the boundary of exactly two edges; this means that flips can be created, one for each vertex:

$$
\left\langle\begin{matrix} f_B \\ e_{12}\ \ e_{13} \\ v_1 \end{matrix}\right\rangle, \qquad
\left\langle\begin{matrix} f_B \\ e_{23}\ \ e_{12} \\ v_2 \end{matrix}\right\rangle, \qquad \text{and} \qquad
\left\langle\begin{matrix} f_B \\ e_{13}\ \ e_{23} \\ v_3 \end{matrix}\right\rangle.
$$

If a vertex is encountered in relation to only one edge, or to more than two edges, then the supplied boundary is invalid. The boundary is also invalid if the orientations of the

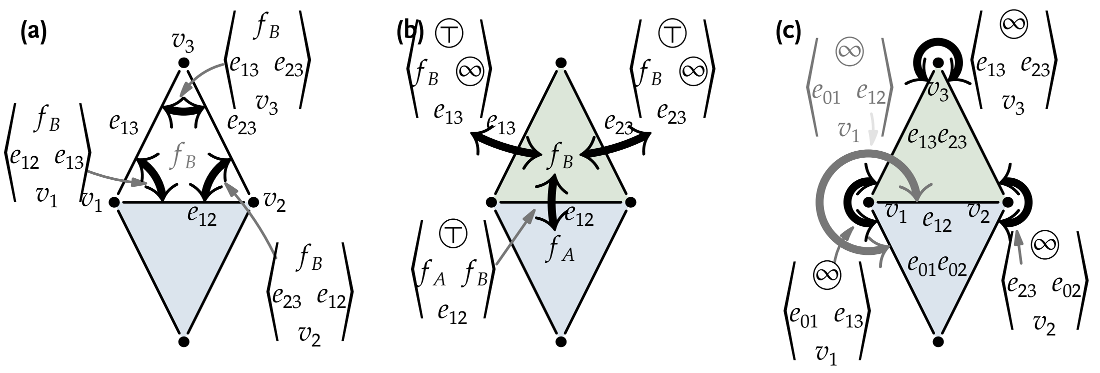


Figure 4.8: Some of the flips added in the execution of $f_B \leftarrow$ **addCell**$(e_{12} - e_{13} - e_{23})$. (a) Flips with f_B in interior position; (b) Flips with f_B in facet position; (c) Flips with $\textcircled{\scriptsize$\infty$}$ in interior position. Existing flips across $\textcircled{\scriptsize$\infty$}$ are altered by replacing new interior edges by the corresponding new exterior edge. For example, in the flip shown in grey ($\langle v_1, e_{01} \leftrightarrow e_{12}, \textcircled{\scriptsize$\infty$}\rangle$) the new interior edge e_{12} is replaced by its corresponding new exterior edge e_{13}.

vertices with respect to the edges are inconsistent. We use a disjoint-set data structure (Cormen, Leiserson, and Rivest 1990) to count how many disjoint segments the boundary is divided into; if we have more than one segment, then the boundary is invalid.

The next group of flips that are to be affected are 2-flips across the new face's edges (Figure 4.8b). There is only one existing flip with one of these edges in joint position: $\langle e_{12}, f_A \leftrightarrow \textcircled{\scriptsize$\infty$}, \textcircled{\scriptsize$\top$}\rangle$. This flip asserts that f_A and $\textcircled{\scriptsize$\infty$}$ are adjacent across e_{12}, and that e_{12} is on the boundary of the cell complex. Now that f_B has been added, the adjacency is between f_A and f_B, so we replace $\textcircled{\scriptsize$\infty$}$ with f_B. Edges e_{13} and e_{23} are on the boundary of the cell complex, so we also add one flip for adjacency across each of them:

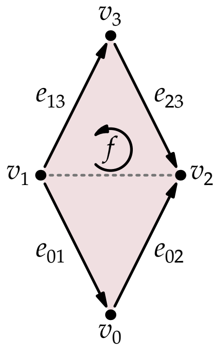
$$\left\langle\begin{matrix} & \textcircled{\scriptsize$\top$} & \\ f_A & & f_B \\ & e_{12} & \end{matrix}\right\rangle, \qquad \left\langle\begin{matrix} & \textcircled{\scriptsize$\top$} & \\ f_B & & \textcircled{\scriptsize$\infty$} \\ & e_{13} & \end{matrix}\right\rangle, \qquad \text{and} \qquad \left\langle\begin{matrix} & \textcircled{\scriptsize$\top$} & \\ f_B & & \textcircled{\scriptsize$\infty$} \\ & e_{23} & \end{matrix}\right\rangle.$$

Finally, the edges that are now towards the outside of the cell complex must be connected by flips across $\textcircled{\scriptsize$\infty$}$ (Figure 4.8c). Now that e_{12} is not on the boundary of the cell complex, we identify the flips involving it and $\textcircled{\scriptsize$\infty$}$:

$$\left\langle\begin{matrix} & \textcircled{\scriptsize$\infty$} & \\ e_{01} & & e_{12} \\ & v_1 & \end{matrix}\right\rangle \qquad \text{and} \qquad \left\langle\begin{matrix} & \textcircled{\scriptsize$\infty$} & \\ e_{12} & & e_{02} \\ & v_2 & \end{matrix}\right\rangle.$$

The boundary edges involving v_1 are now e_{01} and e_{13}; the entry for e_{12} in the first flip is thus replaced by e_{13}. Similarly, the entry for e_{12} in the second flip is replaced by e_{23}. Finally, there is no existing flip for v_3, so a new flip is created. This gives three new flips with $\textcircled{\scriptsize$\infty$}$ in the interior position:

$$\left\langle\begin{matrix} & \textcircled{\scriptsize$\infty$} & \\ e_{01} & & e_{13} \\ & v_1 & \end{matrix}\right\rangle, \qquad \left\langle\begin{matrix} & \textcircled{\scriptsize$\infty$} & \\ e_{23} & & e_{02} \\ & v_2 & \end{matrix}\right\rangle, \qquad \text{and} \qquad \left\langle\begin{matrix} & \textcircled{\scriptsize$\infty$} & \\ e_{13} & & e_{23} \\ & v_3 & \end{matrix}\right\rangle.$$



$$
\left\langle\begin{matrix} e_{01} \\ v_1\;\, v_0 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{02} \\ v_0\;\, v_2 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{13} \\ v_1\;\, v_3 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
\left\langle\begin{matrix} e_{23} \\ v_3\;\, v_2 \\ \textcircled{$\perp$} \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} f \\ e_{02}\;\, e_{01} \\ v_0 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f \\ e_{01}\;\, e_{13} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f \\ e_{23}\;\, e_{02} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f \\ e_{13}\;\, e_{23} \\ v_3 \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} \textcircled{∞} \\ e_{02}\;\, e_{01} \\ v_0 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{01}\;\, e_{13} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{23}\;\, e_{02} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{13}\;\, e_{23} \\ v_3 \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} \textcircled{$\top$} \\ \textcircled{∞}\;\, f \\ e_{01} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ \textcircled{∞}\;\, f \\ e_{02} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ f\;\, \textcircled{∞} \\ e_{13} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ f\;\, \textcircled{∞} \\ e_{23} \end{matrix}\right\rangle.
$$

Figure 4.9: A cell complex and the flips which define it. The face f will be split by the edge with boundary $v_1 - v_2$.

The addition of the cell f_B therefore requires removing three flips

$$
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_A\;\, \textcircled{∞} \\ e_{12} \end{matrix}\right\rangle,
\qquad
\left\langle\begin{matrix} \textcircled{∞} \\ e_{01}\;\, e_{12} \\ v_1 \end{matrix}\right\rangle,
\qquad
\left\langle\begin{matrix} \textcircled{∞} \\ e_{12}\;\, e_{02} \\ v_2 \end{matrix}\right\rangle
$$

and adding nine flips

$$
\left\langle\begin{matrix} f_B \\ e_{12}\;\, e_{13} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_B \\ e_{23}\;\, e_{12} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} f_B \\ e_{13}\;\, e_{23} \\ v_3 \end{matrix}\right\rangle,
\qquad
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_A\;\, f_B \\ e_{12} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_B\;\, \textcircled{∞} \\ e_{13} \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{$\top$} \\ f_B\;\, \textcircled{∞} \\ e_{23} \end{matrix}\right\rangle,
$$

$$
\left\langle\begin{matrix} \textcircled{∞} \\ e_{01}\;\, e_{13} \\ v_1 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{23}\;\, e_{02} \\ v_2 \end{matrix}\right\rangle,
\left\langle\begin{matrix} \textcircled{∞} \\ e_{13}\;\, e_{23} \\ v_3 \end{matrix}\right\rangle
$$

to the flip table. One important feature of this operation is that it is not only *spatially local* (that is, that only flips involving cells incident to the new cell are affected) but also *dimensionally local* — only flips in dimensions near the dimension of the new cell are affected. In the example, a 2-cell is added and the only modifications are to 1- and 2-flips. This property holds in higher dimensions; adding a k-cell to an n-dimensional complex involves only modifications to k-flips and $(k-1)$-flips. This is an important property which limits the implementation complexity of these operations.

There is also a **deleteCell** operation which deletes a cell. It requires that the cell to be deleted does not lie in the boundary of any other cell; once this is confirmed to be the case, the operation essentially does the reverse of **addCell**, removing flips involving the deleted cell and adding flips involving $\textcircled{$\infty$}$.

Splitting cells The second modification operation is **splitCell**, which takes as arguments the cell to split as well as the boundary chain of the new cell introduced between

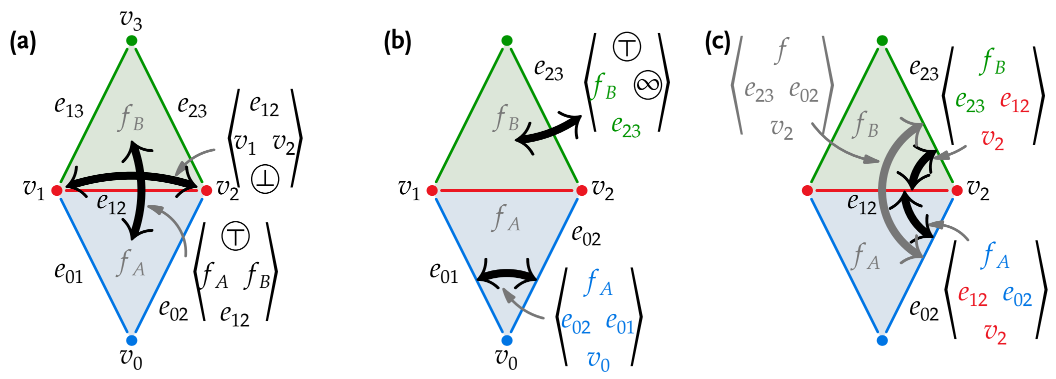


Figure 4.10: Some of the flips modified in the execution of $(f_B, e_{12}, f_A) \leftarrow$ **splitCell**$(f, v_1 - v_2)$. (a) Flips added involving the membrane e_{12} and the left and right children f_B and f_A. (b) Flips involving only the right (blue) or left (green) boundaries have the face f replaced by the corresponding child face, f_A or f_B. (c) Flips involving both right and left boundaries are replaced by two flips. Each of these flips contains only cells from the membrane and either the left or right boundaries. Here, the single flip $\langle v_2, e_{23} \leftrightarrow e_{02}, f\rangle$ is replaced by $\langle v_2, e_{12} \leftrightarrow e_{02}, f_A\rangle$ and $\langle v_2, e_{23} \leftrightarrow e_{12}, f_B\rangle$.

the two child cells. The flips involving the new cell are created in much the same way as in **addCell**, including the same consistency checks. In the case of **splitCell**, however, different flips must be added and modified. As an example, consider splitting the face f in the cell complex shown in Figure 4.9 by an edge from v_1 to v_2. The new edge, or "membrane", will be e_{12}, and it will split f into two faces f_A and f_B. We thus want to perform the operation $(f_B, e_{12}, f_A) \leftarrow$ **splitCell**$(f, v_1 - v_2)$.

The new edge e_{12} does not lie on the boundary of this cell complex, so only two flips must be added to the flip table: the flip with e_{12} interior position $\langle \textcircled{\perp}, v_1 \leftrightarrow v_2, e_{12}\rangle$ (Figure 4.10a) and a flip between the children across the membrane: $\langle e_{12}, f_A \leftrightarrow f_B, \textcircled{\top}\rangle$. The rest of the changes will involve flips where the old face is replaced by its child faces.

The first task is to partition the boundary of the old face into three pieces: vertices in the boundary of the membrane; edges and vertices only in the boundary of the left child; and edges and vertices only in the boundary of the right child. This partitioning is again performed by a disjoint-set algorithm. The flips involving only the left- or right-hand boundaries are then updated by replacing the cell f by the corresponding child (Figure 4.10b). This means that

$$\left\langle\begin{array}{cc}\multicolumn{2}{c}{f}\\ e_{02} & e_{01}\\ \multicolumn{2}{c}{v_0}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ \textcircled{\infty} & f\\ \multicolumn{2}{c}{e_{01}}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ \textcircled{\infty} & f\\ \multicolumn{2}{c}{e_{02}}\end{array}\right\rangle \quad \text{become} \quad \left\langle\begin{array}{cc}\multicolumn{2}{c}{f_A}\\ e_{02} & e_{01}\\ \multicolumn{2}{c}{v_0}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ \textcircled{\infty} & f_A\\ \multicolumn{2}{c}{e_{01}}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ \textcircled{\infty} & f_A\\ \multicolumn{2}{c}{e_{02}}\end{array}\right\rangle$$

$$\text{and} \quad \left\langle\begin{array}{cc}\multicolumn{2}{c}{f}\\ e_{13} & e_{23}\\ \multicolumn{2}{c}{v_3}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ f & \textcircled{\infty}\\ \multicolumn{2}{c}{e_{13}}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ f & \textcircled{\infty}\\ \multicolumn{2}{c}{e_{23}}\end{array}\right\rangle \quad \text{become} \quad \left\langle\begin{array}{cc}\multicolumn{2}{c}{f_B}\\ e_{13} & e_{23}\\ \multicolumn{2}{c}{v_3}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ f_B & \textcircled{\infty}\\ \multicolumn{2}{c}{e_{13}}\end{array}\right\rangle, \left\langle\begin{array}{cc}\multicolumn{2}{c}{\textcircled{\top}}\\ f_B & \textcircled{\infty}\\ \multicolumn{2}{c}{e_{23}}\end{array}\right\rangle.$$

Next, the flips involving both left- and right-hand boundaries are replaced. These are flips from an edge from the left-hand boundary to an edge from the right-hand boundary, with f in interior position and a vertex from the boundary of the membrane in joint position:

$$\left\langle \begin{array}{c} f \\ e_{01}\;\; e_{13} \\ v_1 \end{array} \right\rangle \qquad \text{and} \qquad \left\langle \begin{array}{c} f \\ e_{23}\;\; e_{02} \\ v_2 \end{array} \right\rangle.$$

These are each replaced by two flips, one in which f is replaced by the left-hand child and the facet from the right-hand boundary is replaced by the membrane, and one in which f is replaced by the right-hand child and the facet from the left-hand boundary is replaced by the membrane (Figure 4.10c):

$$\left\langle \begin{array}{c} f_A \\ e_{01}\;\; e_{12} \\ v_1 \end{array} \right\rangle, \left\langle \begin{array}{c} f_B \\ e_{12}\;\; e_{13} \\ v_1 \end{array} \right\rangle \qquad \text{and} \qquad \left\langle \begin{array}{c} f_A \\ e_{12}\;\; e_{02} \\ v_2 \end{array} \right\rangle, \left\langle \begin{array}{c} f_B \\ e_{23}\;\; e_{12} \\ v_2 \end{array} \right\rangle.$$

In the end, the **splitCell** operation modifies the flip table by removing the flips

$$\left\langle \begin{array}{c} f \\ e_{02}\;\; e_{01} \\ v_0 \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ \textcircled{\infty}\;\; f \\ e_{01} \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ \textcircled{\infty}\;\; f \\ e_{02} \end{array} \right\rangle, \left\langle \begin{array}{c} f \\ e_{13}\;\; e_{23} \\ v_3 \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ f\;\; \textcircled{\infty} \\ e_{13} \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ f\;\; \textcircled{\infty} \\ e_{23} \end{array} \right\rangle, \left\langle \begin{array}{c} f \\ e_{01}\;\; e_{13} \\ v_1 \end{array} \right\rangle, \text{ and } \left\langle \begin{array}{c} f \\ e_{23}\;\; e_{02} \\ v_2 \end{array} \right\rangle$$

and adding

$$\left\langle \begin{array}{c} e_{12} \\ v_1\;\; v_2 \\ \textcircled{\bot} \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ f_A\;\; f_B \\ e_{12} \end{array} \right\rangle, \left\langle \begin{array}{c} f_A \\ e_{02}\;\; e_{01} \\ v_0 \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ \textcircled{\infty}\;\; f_A \\ e_{01} \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ \textcircled{\infty}\;\; f_A \\ e_{02} \end{array} \right\rangle, \left\langle \begin{array}{c} f_B \\ e_{13}\;\; e_{23} \\ v_3 \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ f_B\;\; \textcircled{\infty} \\ e_{13} \end{array} \right\rangle, \left\langle \begin{array}{c} \textcircled{\top} \\ f_B\;\; \textcircled{\infty} \\ e_{23} \end{array} \right\rangle,$$

$$\left\langle \begin{array}{c} f_A \\ e_{01}\;\; e_{12} \\ v_1 \end{array} \right\rangle, \left\langle \begin{array}{c} f_B \\ e_{12}\;\; e_{13} \\ v_1 \end{array} \right\rangle, \qquad \left\langle \begin{array}{c} f_A \\ e_{12}\;\; e_{02} \\ v_2 \end{array} \right\rangle, \text{ and } \left\langle \begin{array}{c} f_B \\ e_{23}\;\; e_{12} \\ v_2 \end{array} \right\rangle.$$

Just as for **addCell**, the **splitCell** operation has an inverse **mergeCells**, which removes a membrane and joins two formerly separate cells into one. It requires that the two cells to be joined are the only cells with the membrane in their boundaries, and that their relative orientations be consistent, exactly as if the membrane had been created by a **splitCell** operation. Given this requirement, **mergeCells** reverses the steps taken by **splitCell**, removing flips involving the membrane and the cells to be merged, and inserting flips involving the cell resulting from the merge.

4.6 Implementation in C⁺⁺

The Cell Complex Framework has been implemented as a C⁺⁺ library. In this section I describe this implementation, including the fundamental data structures, the API, and the optimizations used.

Data structures The principal data structure representing the state of the cell complex is `CellStructure`. A `CellStructure` maintains the flip table and all of the higher-level operations to be used on it. Operations like **boundary**, **meet**, **addCell**, and **mergeCells** are implemented as methods of the `CellStructure` class. A new `CellStructure` must be created with a given maximal dimension:

```
CellStructure<Index> cs(2);
```

`CellStructure` is templated over `Index`, the type of a cell reference. There are a few requirements on a cell reference for the CCF: individual `Index` objects must be comparable by `==`, `!=`, and `<` (for use in STL containers); the pseudocells $\ominus$, $\textcircled{\top}$, $\textcircled{\perp}$, $\textcircled{\infty}$, and $\textcircled{?}$ must be available as `Index::UNDEF`, `Index::TOP`, `Index::BOTTOM`, `Index::INFTY`, and `Index::Q`; unary + and − operators which turn an `Index` into an oriented index; and the methods `isPseudocell()`, which reports if an `Index` represents a pseudocell, and `match()`, which acts like `==` but also matches any cell to $\textcircled{?}$, must be defined.

Two kinds of cell reference are included with the CCF. The `UIntIndex` encapsulates an unsigned integer, while the `PtrIndex` encapsulates a pointer. The `CellStructure` treats the two identically; the primary difference between them is in how data is stored on the cell complex. For integer indexes, data is stored in some associative array: for example, an STL `map`:

```
std::map<UIntIndex , Vector3D> vertexPosition;
vertexPosition[v0] = Vector3D(1,0,0);
vertexPosition[v1] = vertexPosition[v0] + Vector3D(0,1,0);
```

The `PtrIndex` lets the user store data directly accessible from the index. It encapsulates a pointer to a data structure specific to the cell it represents. The entire `PtrIndex` is templated with the types of these data structures for each dimension of cell. Thus, if vertices have a data type `VertexT`, edges a data type `EdgeT`, and faces a data type `FaceT`, the corresponding `PtrIndex` in a two-dimensional cell complex will be

```
typedef PtrIndex2D<VertexT , EdgeT , FaceT> Index;
```

The user can then cast an `Index` of known dimension into a subclass which can be dereferenced to the proper class:

```
Index::Index0 v0, v1;
v0->position = Vector3D(0,0,0);
v1->position = Vector3D(1,0,0);
Index::Index1 edge = cs.join(v0,v1);
edge->restLength = 1;
```

The `CellStructure` does not handle the creation of new `Index`es; instead, the user is responsible for creating and deleting them as required. The `UIntIndex` and `PtrIndex` each have associated *index managers* `UIntIndexManager` and `PtrIndexManager` which create and release indexes on request:

```
UIntIndexManager manager;
UIntIndex v0 = manager.create(0), v1 = manager.create(0);
UIntIndex edge = manager.create(1);
```

```
manager.release(edge);
```

Positive and negative orientations are `POS` and `NEG`, of type `RO`. A cell reference can be assigned an orientation by multiplying by `POS` or `NEG`, or with the unary operator + or −, and these oriented cells can be combined to create a `BoundaryChain`:

```
CellStructure<Index>::BoundaryChain chain1 = POS * v0 + NEG * v1;
CellStructure<Index>::BoundaryChain chain2 = +e0 -e1 -e2 +e3;
```

The orientation can be removed from an oriented cell with the ~ operator:

```
CellStructure<Index>::OrientedCell ocell = +v0;
Index unoriented = ~ocell;
```

`CellTuple` is a nested class of `CellStructure`. A new `CellTuple` must be given the `CellStructure` it is operating over. In addition, a number of cell references can be passed, creating a `CellTuple` on those cells; with no cells passed, an arbitrary `CellTuple` on the structure is created:

```
CellStructure<Index>::CellTuple tuple1(cs);
CellStructure<Index>::CellTuple tuple2(cs,v0);
CellStructure<Index>::CellTuple tuple3(cs,e0,f0);
```

Operations The operations introduced in Section 4.1 are all methods of the `CellStructure` class. **boundary** and **coboundary** take a single cell reference as an argument and return a `BoundaryChain` representing the boundary or coboundary:

```
CellStructure<Index>::BoundaryChain e0Boundary = cs.boundary(e0);
CellStructure<Index>::BoundaryChain v0Coboundary = cs.coboundary(v0);
```

There is a special operation to return the boundary of an edge. The `edgeBounds` method returns an STL `pair`; the first element is the vertex with positive relative orientation, the second element the negatively oriented vertex.

```
std::pair<Index,Index> endpoints = cs.edgeBounds(edge);
```

The boundary and coboundary can also be returned as a set of cells. Along with **neighbours**, these methods return the cells in an STL `set`:

```
std::set<Index> f0Neighbours = cs.neighbours(f0);
std::set<Index> e0BoundingCells = cs.bounds(e0);
std::set<Index> v0CoboundingCells = cs.cobounds(v0);
```

Tests for incidence or being on the border take cell references as argument, and return a boolean value:

```
if( cs.border(e0) ) …;
if( cs.incident(e0,Index::INFTY) ) …;
```

meet and **join** take a number of cell references and return a single `index`, which might be a pseudocell:

```
Index faceMeet = cs.meet(f0,f1);
Index vertexJoin = cs.join(v0,v1,v2);
```

The user can query whether a cell is actually in the complex is with `hasCell`, while the dimension of a cell is returned by `dimensionOf`. All cells in the complex of a given dimension can be retrieved with `cellsOfDimension`.

```
if( cs.hasCell(v0) ) …;
unsigned int dim = cs.dimensionOf(v0);
std::set<Index> allVertices = cs.cellsOfDimension(0);
```

The relative orientation between two incident cells is returned by `relativeOrientation`.

```
if( cs.incident(v0,e0) )
  RO orientation = cs.relativeOrientation(v0,e0);
```

The modification operation **addCell** takes as argument the `Index` of the new cell and the `BoundaryChain` defining its boundary. (The boundary may be omitted if we are adding a vertex.) The **deleteCell** operation needs only the cell reference. These operations return a boolean value reporting whether the operation was successful; if the operation was not successful, no change is made to the cell complex. (A common reason for failure is that the provided boundary is invalid.)

```
Index v0 = manager.create(0), v1 = manager.create(0);
cs.addCell(v0);
cs.addCell(v1);
Index edge = manager.create(1);
cs.addCell(edge , +v0 -v1);
if( cs.deleteCell(edge) ) manager.release(edge);
```

There are four cells involved in a **splitCell** operation: the parent n-cell, the two child n-cells, and the new $(n-1)$-cell separating the child cells (the *membrane*). These cells are provided to the operation in a structure of type `SplitStruct`:

```
struct CellStructure<Index>::SplitStruct {
  Index parent;
  Index childP , childN;
  Index membrane; };
```

The child cells are distinguished by their relative orientation with respect to the membrane: `childP` is positively oriented, while `childN` is negatively oriented. The second argument of the operation is the `BoundaryChain` defining the boundary of the membrane. Again, the operation returns `false` and makes no changes if the split operation fails.

```
Index quad = cs.join(v0,v1,v2,v3);
CellStructure<Index>::SplitStruct ss;
ss.parent = quad;
ss.childP = manager.create(2);
ss.childN = manager.create(2);
ss.membrane = manager.create(1);
if( cs.splitCell(ss , +v0 -v2) ) manager.release(quad);
```

mergeCells also requires a `SplitStruct`, though in this case the children and membrane will be removed from the cell complex, while the `parent` will be added.

```
CellStructure<Index>::SplitStruct ss;
ss.parent = manager.create(1);
ss.childP = cs.join(v0,v1);
ss.childN = cs.join(v1,v2);
ss.membrane = v1;
if( cs.mergeCells(ss) ) {
  manager.release(ss.membrane);
  manager.release(ss.childP);
  manager.release(ss.childN); }
```

Cell tuples also have several operations defined. The *k*-cell of a tuple is returned by the indexing operator [*k*], while the tuple resulting from a **flip** operation is returned by `flip`. The **other** operation returns the *k*-cell of the result of **flip**(*k*).

```
CellTuple tuple(cs,v0,e0);
do
  tuple = tuple.flip(1,2);
  Index neighbouringVertex = tuple.other(0);
while ( tuple[1] != e0 );
```

The relative orientation between the *k*-cell and the $k + 1$-cell of the tuple is returned by `relativeOrientation`.

```
RO vertexEdgeOrientation = tuple.relativeOrientation(0);
RO faceVolumeOrientation = tuple.relativeOrientation(2);
```

Complete example Algorithm 4.1 is a complete example of using the Cell Complex Framework. It creates a Sierpinski triangle using the polyhedral subdivision method of Algorithm 5.4. The program starts by creating useful type aliases for `Index`, `SplitStruct`, and `CellTuple` (lines 2–4); cell references in this example will be of type `UIntIndex`. We create the index manager (line 7) and a two-dimensional cell complex (line 8). Vertex positions will be kept in an STL `map` (line 11).

We next create the initial triangle. The three vertices are `v0`, `v1`, and `v2`. For each vertex, we first create a new index with `manager.create(0)` (line 15). We then add the vertex to the cell complex with `addCell` (line 16). Finally, we set the position by setting the appropriate entry in `vpos` (line 17). Next are the edges; the triangle has three, `e0`, `e1`, and `e2`. We create the new indexes (line 19), then add the edges to the cell complex. Each edge goes from its corresponding vertex to the next vertex in the triangle; thus, `e0` goes from `v0` to `v1`, and so has boundary chain `+v0 -v1` (line 20). Finally, we add the face. We create the index with `manager.create(2)` and add the face to the cell complex; the three edges are uniformly oriented, so the face's boundary is `+e0 +e1 +e2` (line 22).

Now we subdivide the triangle indefinitely (lines 25–69). The set `newVertices` will hold all of the new vertices created in this step (line 26). We retrieve all of the edges

Algorithm 4.1 Creation of a Sierpinski triangle using the Cell Complex Framework (part 1)

```
// useful type aliases
typedef UIntIndex Index;
typedef CellStructure<Index>::SplitStruct SplitStruct;
typedef CellStructure<Index>::CellTuple CellTuple;

// new index manager and cell structure
UIntIndexManager manager;
CellStructure<Index> cs(2);

// map containing vertex positions
std::map<Index , Vector2D> vpos;

// create an initial triangle
// vertices
Index v0 = manager.create(0), v1 = manager.create(0), v2 = manager.create(0);
cs.addCell(v0); cs.addCell(v1); cs.addCell(v2);
vpos[v0] = Vector2D(0,0); vpos[v1] = Vector2D(1,0); vpos[v2] = Vector2D(0,1);
// edges
Index e0 = manager.create(1), e1 = manager.create(1), e2 = manager.create(1);
cs.addCell(e0, +v0 -v1); cs.addCell(e1, +v1 -v2); cs.addCell(e2, +v2 -v0);
// face
Index face = manager.create(2); cs.addCell(face, +e0 +e1 +e2);

// subdivide indefinitely
while( true ) {
  std::set<Index> newVertices;

  // subdivide edges
  std::set<Index> edges = cs.cellsOfDimension(1);
  for(auto edgeIter = edges.begin() ; edgeIter != edges.end() ; edgeIter++) {
    std::pair<Index,Index> endpoints = cs.edgeBounds(*edgeIter);
    SplitStruct ss;
      ss.parent = *edgeIter;
      ss.childP = manager.create(1);
      ss.childN = manager.create(1);
      ss.membrane = manager.create(0);
    cs.splitCell(ss);
    vpos[ss.membrane] = 0.5 * (vpos[endpoints.first] + vpos[endpoints.second]);
    newVertices.insert(ss.membrane);
  }
```

Algorithm 4.1 Sierpinski example (part 2)

```
  // subdivide faces
  std::set<Index> faces = cs.cellsOfDimension(2);
  for(auto faceIter = faces.begin() ; faceIter != faces.end() ; faceIter++) {
    CellTuple tau(cs,*faceIter);
    if( newVertices.count(tau[0]) == 0 ) tau = tau.flip(0);
    Index v0 = tau[0];
    tau = tau.flip(0,1,0,1);
    Index v1 = tau[0];
    tau = tau.flip(0,1,0,1);
    Index v2 = tau[0];
    SplitStruct ss;
      ss.parent = *faceIter;
      ss.childP = manager.create(2);
      ss.childN = manager.create(2);
      ss.membrane = manager.create(1);
    cs.splitCell(ss, +v0 -v1);
      ss.parent = cs.join(v1,v2);
      ss.childP = manager.create(2);
      ss.childN = manager.create(2);
      ss.membrane = manager.create(1);
    cs.splitCell(ss, +v1 -v2);
      ss.parent = cs.join(v2,v0);
      ss.childP = manager.create(2);
      ss.childN = manager.create(2);
      ss.membrane = manager.create(1);
    cs.splitCell(ss, +v2 -v0);
    cs.deleteCell(cs.join(v0,v1,v2));
  }
}
```

with the method `cellsOfDimension(1)` (line 29), then iterate over them (lines 30–40). We find the endpoints of the edge with `edgeBounds` (line 31); these are used later to find the position of the vertex midway between them (line 38). The `SplitStruct` for splitting the edge is created and initialized (lines 32–36), and the edge is split (line 37). The new vertex is added to the set (line 39). Note that we do not release the cell index of the old edge; releasing `UIntIndexes` has no effect. If we were using `PtrIndexes`, however, we would have to add the line

```
manager.release(ss.parent);
```

after the split to deallocate the cell data and avoid a memory leak.

After splitting every edge, we must split every face (lines 42–43). We create a new cell tuple `tau` with the current face (line 44) and ensure that its vertex is a new vertex: if `tau[0]` is not in `newVertices`, then we **flip** the tuple to a new vertex (line 45). We then read the three new vertices `v0`, `v1`, and `v2`, advancing the tuple after each (lines 46–50). Now we have to split the face three times (lines 56, 61, and 56). Before each split, we specify the cell to be divided (lines 52, 57, and 52) and allocate the new cells. (Note again that we do not release any of these indexes.) Finally, in order to create the Sierpinski triangle, we find the central triangle with a **join** operation then delete it with `deleteCell` (line 67).

The flip table The methods of the `CellStructure` class are intended to be complete for most users. Descending to the level of flips should only be necessary for special topological manipulations. At this level, the `CellStructure` has two principal member variables which record the state of the cell complex at this level: the map `dimension` records the dimension of each cell, while the flip relations are kept in the flip table `flips`.

The flip table is a container holding all of the flips defining all of the involutions of the combinatorial map. The flips in turn are data structures containing four cell references:

```
struct Flip<Index> {
  Index joint;
  Index facet[2];
  Index interior; };
```

The flip class defines the `match` method, which compares two flips and returns `true` if they match, that is, if the cell references in corresponding positions are equal (up to facet order) or one of the references is ⓘ.

The simplest flip table is just a flip *set*, which is implemented using STL's `set`. Queries are resolved by checking the flips against the template with `match`, one by one. Using this structure, every query becomes a linear-time operation, and flip operations of greater complexity are extremely inefficient. The container used by the Cell Complex Framework is therefore a flip *table*. This table takes advantages of a time-space tradeoff and stores each flip four times, indexed individually by the cell in each of the positions:

```
struct FlipTable<Index> {
  std::map<Index,FlipSet<Index>> byJoint;
  std::map<Index,FlipSet<Index>> byFacet;
  std::map<Index,FlipSet<Index>> byInterior; };
```

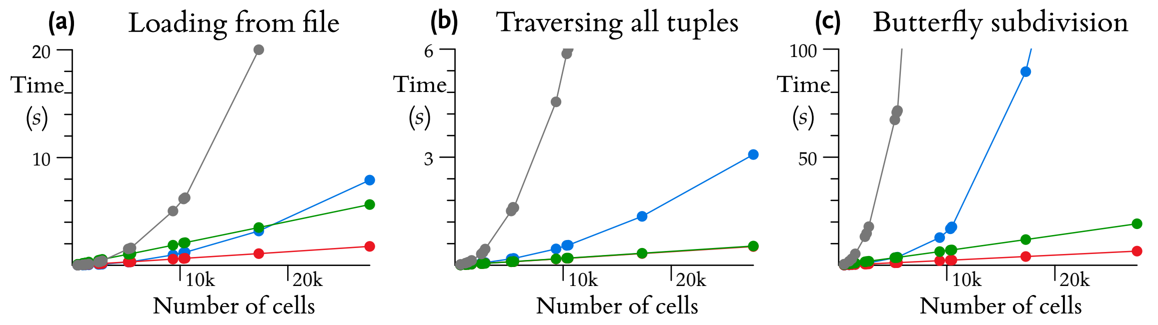


Figure 4.11: Timings of common tasks using a flat flip set (grey), a flip table with one layer of indirection (blue), with two layers of indirection (red), and with three layers of indirection (green). All of the flip tables are significantly more efficient than the flip set, and the double-indirection flip table is always better than the single-indirection flip table. The triple-indirection flip table is worse than the double-indirection table at the tasks involving creating new flips, and about the same at traversal, which involves only looking up flips.

Each of the STL `maps` contains all of the flips in the table, indexed by the cell in the corresponding location. `byJoint`, for example, is a `map` which relates a cell reference c to the set of all flips with c in joint position, while `byFacet` relates c to the set of all flips with c in *either* facet position.

Matching a flip template is now a two-stage process. For example, recall that performing a flip operation in the kth dimension requires finding the unique flip matching $\langle c_{k-1}, c_k \leftrightarrow ⓘ, c_{k+1}\rangle$. To look this up in the flip table, we first get the flip subset corresponding to one of the cells which is not ⓘ. In this case, we look up the flip set corresponding to c_{k-1} in `byJoint`; this operation is a binary search and takes time $O(\log\ n)$. This $O(1)$-sized flip set is then searched linearly for an exact match to the template. The entire query operation thus takes logarithmic time. We could also start the search by looking at `byFacet` or `byInterior`; the difference in expected run time in these cases depends on the particular cell complex. I found in implementing the example models, however, that volumes with many faces and polygons with many sides were more common than vertices with high valence. In this situation, the average number of flips with a given cell in interior position is larger than the average number of flips with a cell in joint position. The default behaviour of the query function is therefore to check `byJoint` unless the template has ⓘ in that position.

It is reasonable to ask whether it would be faster to add a second layer of indirection; that is, to have `byJoint`, for instance, match a cell c not to a flat set of the flips with c in joint position, but to have those flips arranged into a table indexed by the cells in facet and interior position. The linear-time search would then only have to be run against a smaller number of flips. In the example above, matching $\langle c_{k-1}, c_k \leftrightarrow ⓘ, c_{k+1}\rangle$ would be done by searching the smaller flip set `byJoint[`c_{k-1}`].byFacet[`c_k`]` or even `byJoint[`c_{k-1}`].byFacet[`c_k`].byInterior[`c_{k+1}`]`; the latter, a triple indirection, should return exactly the required flip. Indeed, in timing tests I found that adding a second layer of indirection significantly speeds up flip table operations (Figure 4.11). Adding a third layer of indirection is almost as fast as two layers in traversing all tuples (Figure 4.11b), but somewhat slower in tasks in which we create new flips (Figures 4.11ac). The Cell Complex Framework thus uses by default a flip table with two layers of indirection.

Part II

Developmental Modeling with the Cell Complex Framework

5 Techniques for Modeling with the Cell Complex Framework

In Part I, I introduced and developed the Cell Complex Framework, a representation for cell complexes based on the combinatorial map and the Cell-Tuple system of Brisson (1990). In Part II, I will discuss modeling with the Cell Complex Framework. Chapter 6 covers geometric modeling with cell complexes, while Chapter 7 treats biological modeling. First, though, I discuss some general techniques which apply to many different kinds of models: navigating through the cell complex using sequences of flips, subdividing cells, and incorporating geometric information into models. These examples serve as an introduction to modeling with cell complexes.

5.1 Flip navigation

One of the powerful advantages of the flip table representation is that flips can be chained together to move between any connected tuples. Paths between tuples can be used to encode the topological relationship between two cells, or to visit a number of tuples in some order.

Tuple paths We can use a tuple path to describe how to find a target cell with a given topological relationship with respect to a given cell. One application in which this is useful is in specifying subdivision masks in geometric modeling, such as for the butterfly subdivision scheme (Dyn, Levin, and Gregory 1990). In butterfly subdivision, each edge is subdivided, and the position of each new vertex is an affine combination of the positions of existing nearby vertices; the particular weights used depend on the topological relationship between the vertex and the edge to be subdivided. Given a tuple τ containing the edge, a fixed sequence of flips moves across the cell complex to the required vertices. For example, the red vertex in Figure 5.1 is on the tuple τ_R, which can be reached from τ by the sequence of flips $\sigma_0 \circ \sigma_1 \circ \sigma_2$, or[1]

τ.**flip**(2,1,0)

Similarly, the blue vertex is on τ_B and can be reached by

τ.**flip**(1,2,1,0)

[1]The functional composition notation has the *last* operation applied in the leftmost position, while the **flip** operation lists the dimensions to flip in order of application; the leftmost is thus the *first* applied. Because of this, the operations appear in the opposite order in the two notations.

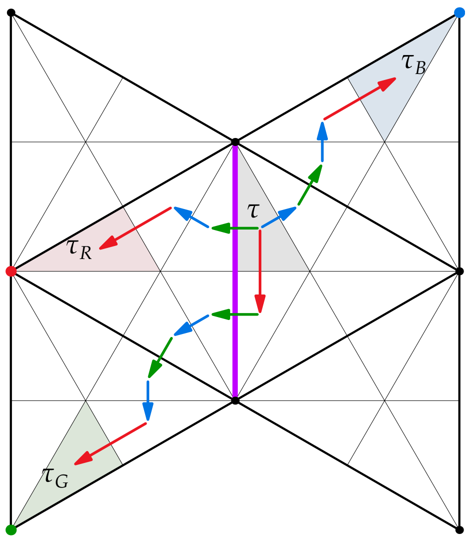


Figure 5.1: The position of a new point (on the purple edge) in the butterfly subdivision scheme depends on the positions of the eight existing points at fixed relative positions. Traveling from the grey tuple in the center to the shaded tuples incident to the red, blue, and green vertices can be accomplished by fixed sequences of flips.

$$\tau_R = \sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \tau,$$
$$\tau_B = \sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_1 \circ \tau,$$
$$\tau_G = \sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_1 \circ \sigma_2 \circ \sigma_0 \circ \tau.$$

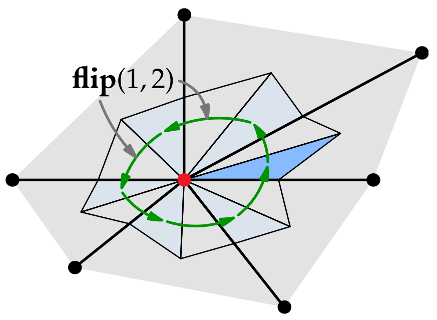


Figure 5.2: Composing flips in 1- and 2-cells to iterate around a vertex (red). Starting from the blue cell tuple, successive applications of **flip**(1, 2) (green arrows) move the tuple counterclockwise around the vertex.

and τ_G, containing the green vertex, can be reached by

τ.**flip**(0,2,1,2,1,0)

The entire subdivision mask, in fact, can be represented by a table of tuple paths and corresponding weights. In Section 5.2 I show this table and describe a complete butterfly subdivision model.

Tuple rotation Another type of traversal is the iteration, which lets the model visit a number of tuples in some order. As mentioned in Section 3.3, flipping about adjacent dimensions performs a *rotation*, and repeating this operation lets the model iterate over a cycle of tuples: around the edges and vertices of a polygon, around the edges and faces adjoining a vertex, around the faces and volumes sharing an edge. For example, Figure 5.2 shows the rotation of a tuple around a central vertex (red). Starting from the blue cell tuple, we can flip the edge then the face to arrive at the corresponding tuple in the next face. This is an application of **flip**(1, 2), and successive applications (green arrows) rotate us counterclockwise around the vertex.

Algorithm 5.1 uses such a vertex iteration to compute the *vertex normal* on a polygon mesh. This normal is an estimate of the normal of a smooth surface interpolated by the mesh (Figure 5.3) and is the weighted average of the normals of the faces adjoining the vertex. Several possible weightings have been suggested; the particular weightings used

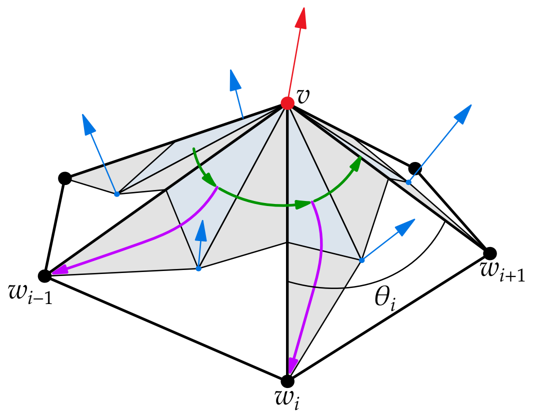


Figure 5.3: The vertex normal (red) at v is an estimate of the normal of an interpolated smooth surface. It is the weighted average of the normals of the adjoining faces (blue). The normal can be computed by iterating around v with the operation **flip**(1, 2) (green arrows); at each step, the neighbouring vertex w_i is just **other**(0) (purple arrows).

in Algorithm 5.1 are those derived by Max (1999):

$$\vec{n} = \sum_i \frac{\sin\theta_i}{|\vec{v}_i||\vec{v}_{i+1}|}\hat{n}_i,$$

where θ_i is the angle subtended by face i at the vertex and $\vec{v}_i$ is the vector from vertex v to its neighbour w_i. This weighting means that faces which take up a lot of v's neighbourhood contribute more to the vertex normal, while faces which extend a long way from v have less effect on the local curvature. Note that

$$\sin\theta_i\,\hat{n}_i = \frac{\vec{v}_i \times \vec{v}_{i+1}}{|\vec{v}_i||\vec{v}_{i+1}|}$$

$$\text{so}\quad \vec{n} = \sum_i \frac{\vec{v}_i \times \vec{v}_{i+1}}{|\vec{v}_i|^2|\vec{v}_{i+1}|^2}. \tag{5.1}$$

This is the equation Algorithm 5.1 uses to compute the normal.

The vertex normal is computed with an iteration of a cell tuple about the vertex v. First (line 2) a tuple τ containing v is created. At each step of the iteration, we will update the normal $\vec{n}$ according to Equation 5.1; this accumulator is set to zero in line 3. Finally, before we begin the iteration, we must set the endpoint of the iteration by saving the current tuple (line 4); when we reach this tuple again, the loop will end (line 10).

For each tuple in the iteration, we compute the contribution of its face according to Equation 5.1. In line 6 we find the vector $\vec{v}_i$; it is the displacement between v and the vertex w_i at the other end of the current edge. w_i is thus the 0-cell of $\sigma_0\tau$; this is exactly the cell τ.**other**(0) (purple arrow, Figure 5.3). The position of each vertex w is stored in $\boldsymbol{P}(w)$, and we subtract the positions to find the vector.

In line 7 we advance the tuple to the next face (green arrow, Figure 5.3). The displacement w_{i+1} along this edge is computed in exactly the same way as w_i (line 8). Finally, the accumulator $\vec{n}$ is updated according to Equation 5.1 (line 9). Once the iteration has finished by circling back to the original tuple, the accumulated normal is normalized and returned (line 11).

The visual effect of rendering with a vertex normal is striking (Figure 5.4). In the default rendering (a) each face is a uniform colour which depends on its orientation; this emphasizes the nature of the mesh as a collection of flat polygons. In drawing with vertex

Algorithm 5.1 Compute the vertex normal according to the method of Max (1999).

```
Input: A vertex $v$
Require: Each vertex $w$ in the cell complex has a position $P(w)$
Output: The estimated normal at $v$
 1: procedure VertexNormal($v$)
 2:     $\tau \leftarrow$ tuple containing $v$
 3:     $\vec{n} \leftarrow 0$                                  ▷ normal accumulator
 4:     $\tau_{\text{start}} \leftarrow \tau$                   ▷ record starting point
 5:     repeat
 6:         $\vec{v}_i \leftarrow P(\tau.\text{other}(0)) - P(v)$
 7:         $\tau.\text{flip}(1,2)$                             ▷ advance to the next tuple
 8:         $\vec{v}_{i+1} \leftarrow P(\tau.\text{other}(0)) - P(v)$
 9:         $\vec{n} \leftarrow \vec{n} + \frac{\vec{v}_{i+1} \times \vec{v}_i}{|\vec{v}_i|^2 |\vec{v}_{i+1}|^2}$
10:     until $\tau = \tau_{\text{start}}$                      ▷ we've come full circle
11:     return $\vec{n} \,/\, |\vec{n}|$
12: end procedure
```

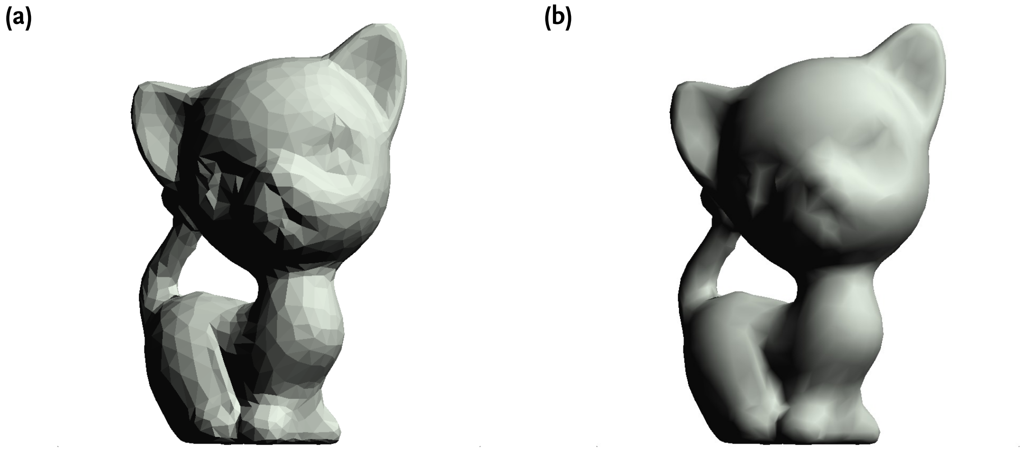


Figure 5.4: The effect of rendering with vertex normals. (a) The mesh has been rendered with flat faces; (b) the mesh has been rendered with vertex normals computed by Algorithm 5.1.

normals (b) the colour can be computed independently for each vertex then interpolated across the faces (Gouraud 1971), giving the illusion that the surface is smooth.

General traversal Traversals by alternating flips are useful in many situations, but can only reach a one-dimensional cycle of cells. In many situations, however, the set of cells that must be reached do not have such a ring structure. For instance, iterating through the faces (2-cells) of a 3-cell can not be performed by simply repeating the application of a single operation. In this case, we can recourse to general graph traversal methods. A breadth-first search, for example, can be used to iterate through all tuples which can be reached using a given set of flips. The faces of a 3-cell can be found by iterating through the tuples reachable by combinations of σ_0, σ_1, and σ_2, holding the 3-cell itself constant. This iteration will not be cyclic (in the general case, cannot be), but is guaranteed to find all of the tuples adjoining the 3-cell.

One application of a graph traversal is in determining the *orientability* of a subdivided manifold. A manifold is considered orientable if an *orientation* can be consistently chosen over the entire manifold. The topological definition of "orientation" in this sense is complex (Hatcher 2002), but it is related to the *total orientation* of a cell tuple.

Definition 2. *The* total orientation $\vartheta(\tau)$ *of a cell tuple* $\tau = (c_0, c_1, \dots, c_n)$ *is the product of the successive relative orientations between its cells:*

$$\vartheta(\tau) = \rho(c_0, c_1)\, \rho(c_1, c_2)\, \cdots\, \rho(c_{n-1}, c_n)\, \rho(c_n, \textcircled{\top}).$$

The total orientation of a tuple is related to the handedness of its associated tuple coordinate system (Section 5.3). The two possible orientations correspond to left-handed and right-handed coordinates, though which is which depends on the particular embedding of the cell complex.

When we perform an involution σ_i on a tuple τ, we change exactly one cell: c_i becomes c_i'. The only relative orientations which might differ are those between c_{i+1} and c_i, or between c_i and c_{i-1}. Remember, though, that this change is produced by the application of the flip

$$\left\langle \begin{array}{c} c_{i+1} \\ c_i \;\; c_i' \\ c_{i-1} \end{array} \right\rangle;$$

the old relative orientations (with respect to c_i) run down the left side of this flip, while the new relative orientations (with respect to c_i') run down the right side. But this means that, by Equation 4.2, their products must be of opposite sign. Thus, the total orientation of τ is the *opposite* of the total orientation of $\sigma_i \circ \tau$.

All neighbours of a positively-oriented tuple are thus negatively oriented, while all neighbours of a negatively-oriented tuple are positively oriented. If a manifold is orientable, then these orientations are consistent across all tuples, and we can divide the set of tuples into two disjoint sets: positively and negatively oriented. If the manifold is not orientable, however, then orientations cannot be assigned consistently, so we will not be able to divide the cell tuples into two disjoint sets. Algorithm 5.2 uses this fact to determine the orientability of the manifold represented by a cell complex.

Algorithm 5.2 Determine whether a connected manifold component represented by the cell complex is orientable by checking that its tuples form a bipartite graph

Input: a tuple τ_0 on the manifold component in question
Output: **true** if the component is orientable; **false** otherwise

```
 1: procedure IsOrientable($\tau_0$)
 2:     $pending \leftarrow \{\tau_0\}$
 3:     $visited \leftarrow \{\tau_0\}$
 4:     $\vartheta(\tau_0) \leftarrow +$
 5:     while $pending \neq \{\}$ do
 6:         pick $\tau \in pending$
 7:         $pending \leftarrow pending \setminus \{\tau\}$
 8:         $\rho \leftarrow \vartheta(\tau)$
 9:         for $d \in \{0, \ldots, n\}$ do                ▷ find neighbours in each dimension
10:             $\tau' \leftarrow \tau.\mathbf{flip}(d)$
11:             if $\tau' \notin visited$ then
12:                 $pending \leftarrow pending \cup \{\tau'\}$
13:                 $visited \leftarrow visited \cup \{\tau'\}$
14:                 $\vartheta(\tau') \leftarrow -\rho$
15:             else if $\vartheta(\tau') \neq -\rho$ then           ▷ inconsistent orientation
16:                 return false
17:             end if
18:         end for
19:     end while
20:     return true                              ▷ all tuples are consistently oriented
21: end procedure
```

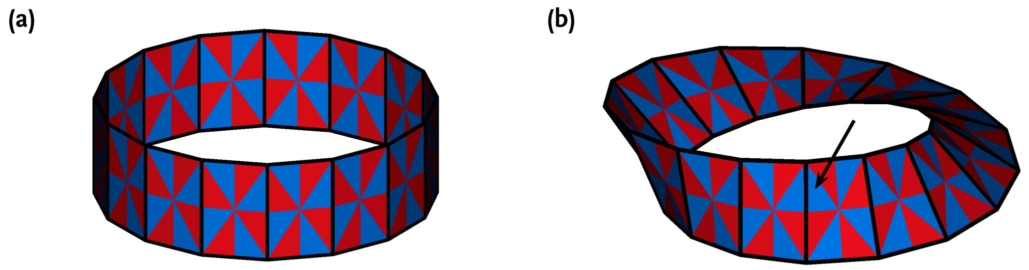


Figure 5.5: Cylinder (a) and Möbius strip (b) made of square faces (outlined in black). Tuples with positive orientation are coloured blue, while tuples with negative orientation are red. (a) Tuples on the cylinder are adjacent to tuples of the opposite orientation, meaning that the cylinder is an orientable manifold. (b) On the Möbius strip, one pair of blue tuples (indicated by an arrow) and one pair of red tuples are adjacent; this demonstrates that the Möbius strip is not orientable.

The algorithm takes as input a single tuple; its search will reach all connected tuples and thus determine whether one particular connected component of the manifold is orientable. The value returned is either **true** (if the manifold component is orientable) or **false** (if the manifold component is not orientable). The algorithm performs a graph traversal of the tuples connected to the input tuple and ensures that their orientations are consistent. The traversal relies on two sets: *visited* is a set of all tuples whose orientation has been computed, while *pending* is a set of all tuples whose neighbours have not yet been examined. Both sets initially contain only the input tuple τ_0, whose orientation is (arbitrarily) set to positive (line 4).

The algorithm then examines one pending tuple τ each iteration until no tuples are left in the *pending* set (lines 5–19). Each of τ's neighbours $\tau' \in \{\sigma_0 \circ \tau, \dots, \sigma_n \circ \tau\}$ is examined in turn (lines 9–18). If τ' has not been visited, then it is added to the *pending* and *visited* sets (lines 12–13) so its neighbours will be examined in a future iteration. Its orientation is also set to the reverse of the orientation of τ (line 14).

If, on the other hand, τ' has already been visited, then its previously computed orientation $\vartheta(\tau')$ should be the reverse of the orientation of τ. If this is not the case, then the orientations have not been assigned consistently and the algorithm indicates that the manifold component is non-orientable (line 16). If every tuple is traversed without finding an inconsistency, however, then the manifold component must be orientable, and the algorithm returns **true** (line 20).

This algorithm is visualized in Figure 5.5. Positively-oriented tuples are coloured blue, while negatively-oriented tuples are coloured red. A cylinder is an orientable manifold, and Figure 5.5a shows the expected distribution of orientations: the cell tuples alternate between positive and negative. A Möbius strip (Figure 5.5b) is not orientable, and the algorithm reports this when it finds adjacent tuples of the same orientation (two adjacent red tuples and two adjacent blue tuples, indicated by an arrow).

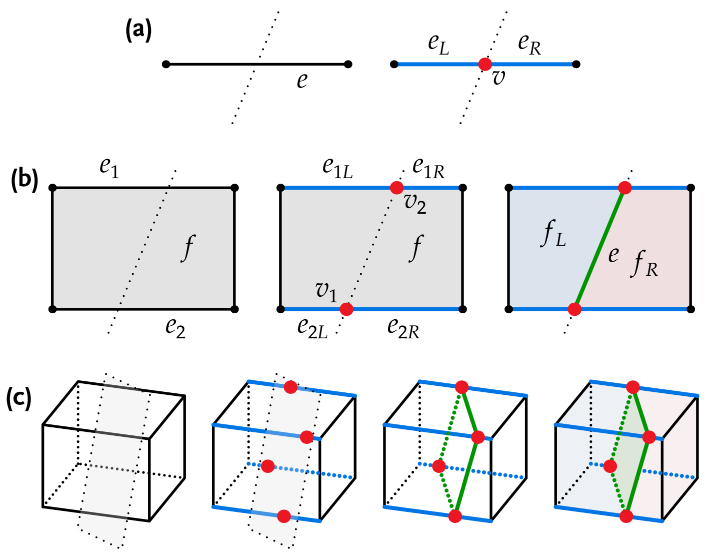


Figure 5.6: (a) Splitting an edge e into two edges e_L and e_R at its intersection with line ℓ is performed simply by the CCF operation $(e_L, v, e_R) \leftarrow$ **splitCell**(e). (b) Splitting a face f into two faces f_L and f_R along the intersection with ℓ is a two-stage process. First, the edges e_1 and e_2 intersecting ℓ are subdivided themselves; the new vertices v_1 and v_2, lying on ℓ, are then the endpoints of the new edge e subdividing f: $(f_L, e, f_R) \leftarrow$ **splitCell**$(f, +v_1\ -v_2)$. (c) Splitting a volume in two is a three-stage process: first edges are split and new vertices created, then faces are split and new edges created, then the volume is split by a new face.

5.2 Subdividing cells

Splitting along a line The most common way in which the cell complex is changed is through the splitting of cells. Splitting an edge at its intersection point with a line (Figure 5.6a) is simple; the split operation $(e_L, v, e_R) \leftarrow$ **splitCell**(e) makes all of the topological changes. But how do we split a face at its intersection with a line (Figure 5.6b)? Two of the vertices of the face will become the endpoints of a new edge, splitting the face in two, but what do we do if there are no vertices on the splitting line? The answer is to split a face as a two-stage process: first, the edges intersecting the line are split, creating new vertices which lie on the splitting line; then these vertices become the endpoints of the splitting edge.[2] Similarly, splitting a volume is a three-stage process (Figure 5.6c): first, edges are split, then faces, then the volume itself.

Algorithm 5.3 splits a face along its intersection with a plane ℓ. The set *endpoints* stores the two intersection points of the plane with the face's edges. These points are found in an iteration over the edges of the face (lines 5–17); this is similar to the iteration over faces in Algorithm 5.1, but the tuple is advanced by flips of vertex and edge (line 7). For each edge, its endpoints p_0 and p_1 are found (lines 6 and 8), then used to find the intersections of the edge with the plane (line 9).

The function LineSegmentIntersect returns the parameter of the intersection point

[2]This assumes that the face is convex; if the face is concave, the splitting line might intersect its edges in many points, and it becomes ambiguous how we are to split the face in two. The rest of this section assumes that cells are convex.

Algorithm 5.3 Split a face f along its intersection with plane ℓ

```
Input: A face $f$ and a plane $\ell$
Require: Each vertex $v$ in the cell complex has a position $\boldsymbol{P}(v)$
 1: procedure SplitFaceByPlane($f$,$\ell$)
 2:     $\tau \leftarrow$ tuple containing $f$
 3:     $endpoints \leftarrow \{\}$
 4:     $\tau_{\text{start}} \leftarrow \tau$                              ▷ starting point for iteration
 5:     repeat
 6:         $\boldsymbol{p}_0 \leftarrow \boldsymbol{P}(\tau[0])$
 7:         $\tau$.flip$(0, 1)$                                  ▷ advance to the next tuple
 8:         $\boldsymbol{p}_1 \leftarrow \boldsymbol{P}(\tau[0])$
 9:         $t \leftarrow$ LineSegmentIntersect($\boldsymbol{p}_0$,$\boldsymbol{p}_1$,$\ell$)
10:         if $t = 1$ then                                ▷ plane passes through vertex
11:             $endpoints \leftarrow endpoints \cup \{\tau[0]\}$
12:         else if $0 < t < 1$ then                    ▷ split edge where the plane cuts it
13:             $(e_L, v, e_R) \leftarrow$ splitCell($\tau$.other$(1)$)
14:             $\boldsymbol{P}(v) \leftarrow \boldsymbol{p}_0 + t(\boldsymbol{p}_1 - \boldsymbol{p}_0)$
15:             $endpoints \leftarrow endpoints \cup \{v\}$
16:         end if
17:     until $\tau = \tau_{\text{start}}$
18:     if $endpoints = \{v_0, v_1\}$ and join$(v_0, v_1) = f$ then
19:         $(f_L, e, f_R) \leftarrow$ splitCell$(f, +v_0\ - v_1)$
20:     end if
21: end procedure
```

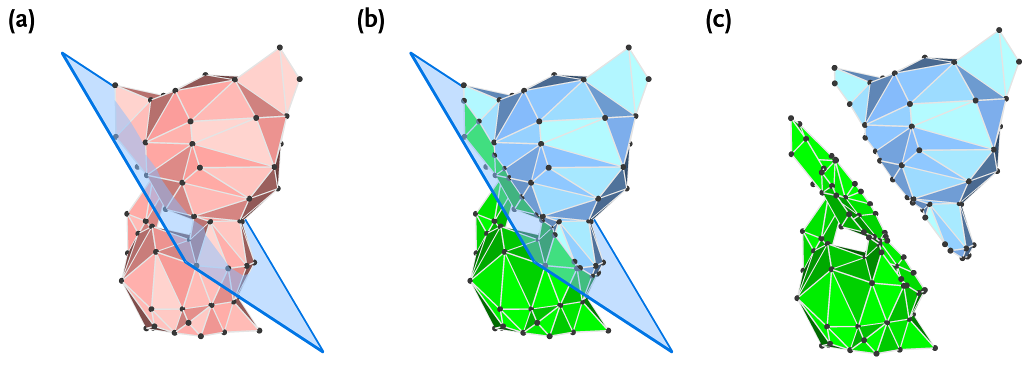


Figure 5.7: Splitting a mesh along a plane. (a) The initial mesh and the splitting plane (blue). (b) Faces on the mesh intersecting the plane have been split; faces above the plane are drawn in blue, faces below are drawn in green. (c) The mesh has been split in two along the plane, and the two parts moved apart for illustration.

of the line segment through $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ with the plane ℓ. If $t = 1$ (line 10), then $\boldsymbol{p}_1$ lies on the plane and this point will be one of the endpoints of the splitting edge (line 11). If the parameter t is between 0 and 1 (line 12) then the plane intersects the edge proper. In this case, we split the edge (line 13), move the newly created vertex to the intersection point (line 14), and add the vertex to the set of endpoints (line 15). If the parameter is outside the interval $(0, 1]$, then we do nothing; the case $t = 0$ ($\boldsymbol{p}_0$ lies on the plane) is caught by the previous stage of the iteration (or the final stage, if this is the first edge in the loop).

When we have iterated around the entire face, we can split the face if two conditions are met (line 18). First, exactly two endpoints v_0 and v_1 must have been found; second, the **join** of these vertices must be the face itself. This second condition catches the possibility that the face intersects the plane along an existing edge; if that is the case, their **join** will be that edge. If the conditions are met, then the face is split from v_0 to v_1 (line 19).

In Figure 5.7 I illustrate a model which divides an entire mesh in two along a plane. Each face is divided by Algorithm 5.3, then the faces are categorized by centroid (Section 5.3); those faces above the plane are coloured blue and moved up, those below the plane are coloured green and moved down.

Polyhedral subdivision Another common use for the **splitCell** operation is *subdivision*. By subdividing all of the cells in a model or a localized region, we can geometrically smooth a surface (Chapter 6) or refine mathematical computations done on the cell complex. The *polyhedral* subdivision, for example, subdivides a mesh of triangles by splitting each edge into two, and each triangle into four (Figure 5.8). A new vertex is placed at the midpoint of the edge it subdivides. Algorithm 5.4 subdivides a triangular mesh in this way.

The mesh is subdivided in two stages. In the first stage (lines 3–7), every edge is subdivided and the splitting vertex is placed at its midpoint. In the second stage (lines 8–21), each face is split into four faces by new edges whose endpoints are the new vertices created in the first stage. In order to distinguish these from the original vertices, the new vertices are saved in the set *newV* (line 6).

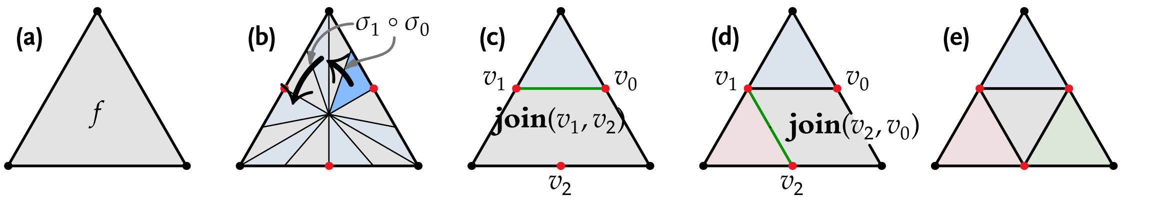


Figure 5.8: Polyhedral subdivision of a triangle. (a) The initial triangle f. (b) After subdividing the edges, the shape has six vertices and twelve tuples. We proceed from the shaded tuple to the next *new* vertex by flipping four times. (c) After splitting along the edge from v_0 to v_1, the next face to be split is **join**(v_1, v_2). (d) The next face to be split is **join**(v_2, v_0). (e) The original triangle has been split into four triangles.

Algorithm 5.4 Polyhedral subdivision

Require: Each vertex v in the cell complex has a position $\boldsymbol{P}(v)$

```
1:  procedure PolyhedralSubdivision
2:      $newV \leftarrow \{\}$
3:      for all edge $e$ do                                ▷ split each edge at its midpoint
4:          $(e_L, v, e_R) \leftarrow$ splitCell($e$)
5:          $\boldsymbol{P}(v) \leftarrow$ Centroid($e$)
6:          $newV \leftarrow newV \cup \{v\}$
7:      end for
8:      for all face $f$ do                                ▷ split each face in four
9:          $\tau \leftarrow$ tuple containing $f$
10:         if $\tau[0] \notin newV$ then                   ▷ ensure that $\tau[0]$ is a new vertex
11:             $\tau$.flip(0)
12:         end if
13:         $v_0 \leftarrow \tau[0]$
14:         $\tau$.flip(0, 1, 0, 1)                        ▷ advance to the next new vertex
15:         $v_1 \leftarrow \tau[0]$
16:         $\tau$.flip(0, 1, 0, 1)
17:         $v_2 \leftarrow \tau[0]$
18:         splitCell($f, +v_0 - v_1$)
19:         splitCell(join($v_1, v_2$), $+v_1 - v_2$)
20:         splitCell(join($v_2, v_0$), $+v_2 - v_0$)
21:     end for
22: end procedure
```

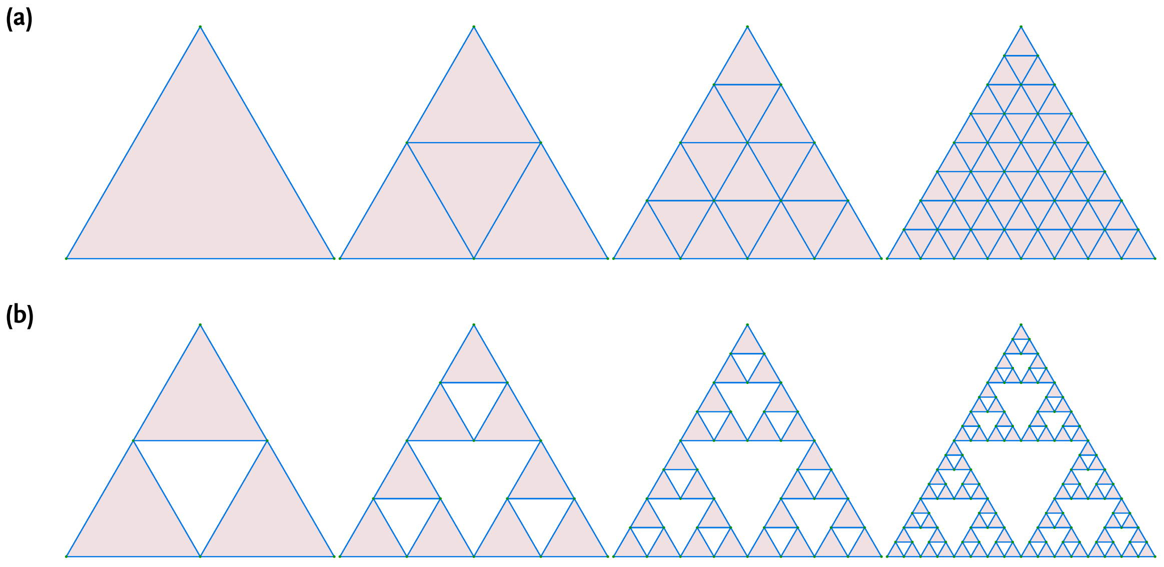


Figure 5.9: (a) Successive application of the polyhedral subdivision algorithm turns a single triangle into a tiled array of triangles. (b) Removing the central triangle after each subdivision step results in the construction of a Sierpinski gasket.

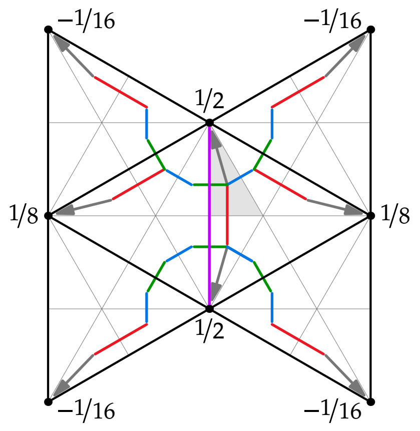
Edges are split much as in the previous Algorithm. For splitting each face, we first create a cell tuple τ (line 9) and ensure that its vertex is new by flipping the vertex if it is old (lines 10–12). We then collect the three new vertices (lines 13–17). Between each vertex, we advance the tuple *two* vertices around the face, bypassing the old vertex (Figure 5.8b). Finally, the face is split three times (lines 18–20); as f itself will not exist after the first split (Figure 5.8c), the face connecting vertices v_1 and v_2 is found as $\mathbf{join}(v_1, v_2)$ (line 19), and the same for v_0 and v_2 (line 20).

Successive application of the polyhedral subdivision algorithm turns a single triangle into a tiled array of triangles (Figure 5.9a). We can also slightly alter the polyhedral subdivision algorithm to create a Sierpinski gasket. In this case, the internal of the four subdivided triangles is removed from the cell complex at each subdivision step, leaving a self-similar structure with holes (Figure 5.9b). The change requires only one alteration to Algorithm 5.4: the insertion of a new line

$$\mathbf{deleteCell}(\mathbf{join}(v_0, v_1, v_2))$$

at the end of the face loop, after line 20. The **join** of the three new vertices is their common triangle, i.e. the central triangle of the subdivision.

The new vertices do not have to be placed at the midpoints of the edge they subdivide. In the butterfly subdivision scheme (Dyn, Levin, and Gregory 1990), the position of new points is an affine combination of the positions of nearby points. In Section 5.1 I showed how a specific sequence of flips could be used to travel from the edge to be subdivided to each vertex whose position must be considered to place the new vertex. Figure 5.10 shows the weights of each of these vertices and their corresponding flip sequences. Using this table, we can turn the polyhedral subdivision algorithm into a butterfly subdivision



1/2	$\tau[0]$
1/2	$(\sigma_0 \circ \tau)[0]$
1/8	$(\sigma_0 \circ \sigma_1 \circ \tau)[0]$
1/8	$(\sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \tau)[0]$
−1/16	$(\sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_1 \circ \tau)[0]$
−1/16	$(\sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_1 \circ \sigma_0 \circ \tau)[0]$
−1/16	$(\sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_1 \circ \sigma_2 \circ \tau)[0]$
−1/16	$(\sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \sigma_1 \circ \sigma_2 \circ \sigma_0 \circ \tau)[0]$

Figure 5.10: Tuple paths are used to find nearby vertices in the Butterfly subdivision scheme. When the purple edge is subdivided, the position of the new vertex is an affine combination of the positions of its eight neighbouring vertices. Starting from the shaded tuple containing the edge to be subdivided, these vertices can be reached through specific flip sequences.

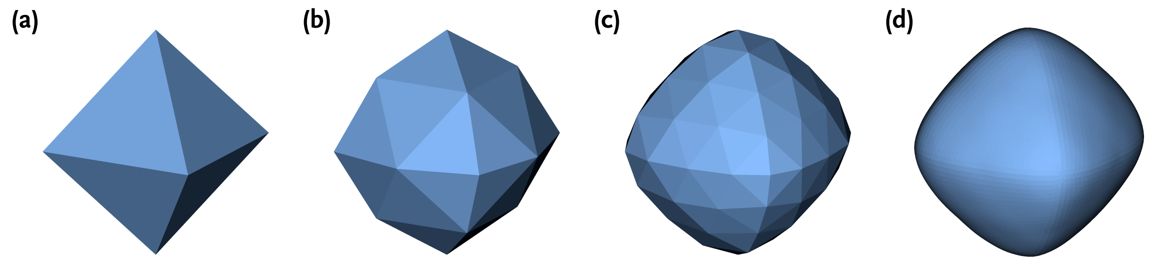


Figure 5.11: Butterfly subdivision applied to an octahedral mesh. (a) The initial mesh; (b) after one subdivision step; (c) after two subdivision steps; (d) after five subdivision steps.

algorithm (Algorithm 5.5).

The key difference in Algorithm 5.5 is line 16, in which a new vertex's position is set. Rather than using the midpoint of the edge, point v is placed at a point $\boldsymbol{P}'(e)$ which depends on the edge it splits. This position is computed in the first loop over all edges (lines 2–12). First, a tuple τ containing the edge is created (line 3); this is the shaded tuple in Figure 5.10. In line 4, the point $\boldsymbol{P}'(e)$ is set to the position of τ's vertex, scaled by 1/2; this is the first of eight points whose positions contribute to the position of the vertex which will split e. In each of the next seven lines, one neighbouring vertex is found by following the corresponding flip sequence (giving tuple τ') and its position is scaled and added to $\boldsymbol{P}'(e)$. The scaled positions of these eight vertices combine to give the position we need.

Figure 5.11 illustrates the application of butterfly subdivision. Starting from a simple shape (Figure 5.11a), the subdivision reduces the size of the triangles and smooths the surface of the mesh, while interpolating the vertices of the original mesh.

Algorithm 5.5 Butterfly subdivision

```
Require: Each vertex $v$ in the cell complex $C$ has a position $P(v)$
 1: procedure ButterflySubdivision
 2:     for all edge $e$ do                                  ▷ find positions of new vertices
 3:         $\tau \leftarrow$ tuple containing $e$
 4:         $P'(e) \leftarrow 1/2\, P(\tau[0])$
 5:         $\tau' \leftarrow \tau.\mathbf{flip}(0)$;  $P'(e) \leftarrow P'(e) + 1/2\, P(\tau'[0])$
 6:         $\tau' \leftarrow \tau.\mathbf{flip}(1,0)$;  $P'(e) \leftarrow P'(e) + 1/8\, P(\tau'[0])$
 7:         $\tau' \leftarrow \tau.\mathbf{flip}(2,1,0)$;  $P'(e) \leftarrow P'(e) + 1/8\, P(\tau'[0])$
 8:         $\tau' \leftarrow \tau.\mathbf{flip}(1,2,1,0)$;  $P'(e) \leftarrow P'(e) - 1/16\, P(\tau'[0])$
 9:         $\tau' \leftarrow \tau.\mathbf{flip}(0,1,2,1,0)$;  $P'(e) \leftarrow P'(e) - 1/16\, P(\tau'[0])$
10:         $\tau' \leftarrow \tau.\mathbf{flip}(2,1,2,1,0)$;  $P'(e) \leftarrow P'(e) - 1/16\, P(\tau'[0])$
11:         $\tau' \leftarrow \tau.\mathbf{flip}(0,2,1,2,1,0)$;  $P'(e) \leftarrow P'(e) - 1/16\, P(\tau'[0])$
12:     end for
13:     $newV \leftarrow \{\}$
14:     for all edge $e$ do                                  ▷ split each edge at position $P'(e)$
15:         $(e_L, v, e_R) \leftarrow$ splitCell($e$)
16:         $P(v) \leftarrow P'(e)$
17:         $newV \leftarrow newV \cup \{v\}$
18:     end for
19:     for all face $f$ do                                  ▷ split each face in four
20:         $\tau \leftarrow$ tuple containing $f$
21:         if $\tau[0] \notin newV$ then                     ▷ ensure that $\tau[0]$ is a new vertex
22:             $\tau.\mathbf{flip}(0)$
23:         end if
24:         $v_0 \leftarrow \tau[0]$
25:         $\tau.\mathbf{flip}(0,1,0,1)$                        ▷ advance to the next new vertex
26:         $v_1 \leftarrow \tau[0]$
27:         $\tau.\mathbf{flip}(0,1,0,1)$
28:         $v_2 \leftarrow \tau[0]$
29:         splitCell($f, +v_0 - v_1$)
30:         splitCell(join($v_1, v_2$), $+v_1 - v_2$)
31:         splitCell(join($v_2, v_0$), $+v_2 - v_0$)
32:     end for
33: end procedure
```

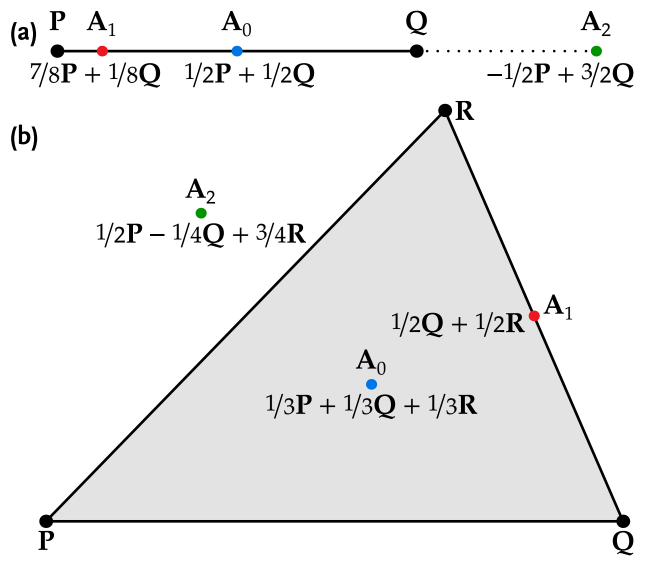


Figure 5.12: Barycentric coordinates label points in simplexes of one and two dimensions.

5.3 Tuple coordinates

In this section, I introduce *tuple coordinates*, a set of intrinsic coordinate systems defined on the cell complex. These coordinates are helpful in performing geometric operations on the cell complex, and I provide algorithms using tuple coordinates for computing the *measure* and *centroid* of a cell. The measure is the size of the cell, and is a generalization of one-dimensional length, two-dimensional area, and three-dimensional volume. The centroid is the geometric center of the cell, the average position of all of its points. Tuple coordinates are also used in Chapter 6 to propagate geodesics across a mesh.

The tuple frame As discussed in Section 3.2.1, the way in which cells are assigned positions in a manifold is called the cell complex's embedding. Most of the models described in this thesis use the *default embedding*, which assumes that vertices v_i are placed at positions p_i, and that all k-cells are the flat k-dimensional polytopes defined by their vertices. Of course, a k- cell with more than $k+1$ vertices will not in general be flat (e.g. a quadrilateral in three-dimensional space (a 2-cell with 4 vertices) is not necessarily flat). In many models, this is not a problem: we can restrict the cells to simplexes or guarantee that non-simplicial cells will nonetheless be flat. Even when these guarantees are not possible, cells are often very close to flat in practice and this approximation makes modeling significantly easier.

Note that in any embedding, the only points on that manifold that are explicitly represented are the vertices of the complex. In order to relate other points on the manifold to the cell complex, we can introduce a coordinate system to relate any point in the manifold to the positions of the vertices. In the default embedding, this is particularly easy, and in this section I define *tuple coordinates*, a family of coordinate systems, each defined by a single cell tuple.

Tuple coordinates are based on *barycentric coordinates* (Coxeter 1961), which let us define a point in a simplex as a weighted sum of the positions of the vertices. For example (Figure 5.12a), the midpoint of a line $\overline{PQ}$ is at the average position of the vertices, $A_0 =$

$1/2\boldsymbol{P} + 1/2\boldsymbol{Q}$. In fact, we can take any weights instead of $1/2$; as long as the sum of the weights is 1, the points will lie on the line $\overline{PQ}$. Thus, $\boldsymbol{A}_1 = 7/8\boldsymbol{P} + 1/8\boldsymbol{Q}$, in which the point P is weighted much more than Q, is much closer to P. The weights need not even be positive; $\boldsymbol{A}_2 = -1/2\boldsymbol{P} + 3/2\boldsymbol{Q}$ still lies on the line $\overline{PQ}$, though points with negative coefficients do not lie between the endpoints. Given the constraint that the coefficients sum to 1, each of these sums uniquely defines a point on the line.

The same is true for higher-dimensional simplexes; a point on a triangle ΔPQR is uniquely defined by a weighted average of the triangle's vertices, where the weights sum to 1 (Figure 5.12b). In this case, if all three of the coefficients are between 0 and 1, the point lies within the triangle, like $\boldsymbol{A}_0 = 1/3\boldsymbol{P} + 1/3\boldsymbol{Q} + 1/3\boldsymbol{R}$. If one of the coefficients is zero, then the point lies on one of the triangle's edges like $\boldsymbol{A}_1 = 1/2\boldsymbol{Q} + 1/2\boldsymbol{R}$. Finally, if a coefficient is negative, the point lies in the plane of the triangle but outside its edges, like $\boldsymbol{A}_2 = 1/4\boldsymbol{P} - 1/4\boldsymbol{Q} + 3/4\boldsymbol{R}$.

We can remove the constraint that the weights sum to 1 by expressing these points in a coordinate frame; that is, a coordinate system with a specified origin point. A point $\boldsymbol{p}$ on an n-simplex is uniquely defined by

$$\boldsymbol{p} = \alpha_0\boldsymbol{P}_0 + \alpha_1\boldsymbol{P}_1 + \cdots + \alpha_n\boldsymbol{P}_n;$$

since $\sum_i \alpha_i = 1$, we see that $\alpha_0 = 1 - \alpha_1 - \cdots - \alpha_n$ and

$$\begin{aligned}\boldsymbol{p} &= (1 - \alpha_1 - \cdots - \alpha_n)\boldsymbol{P}_0 + \alpha_1\boldsymbol{P}_1 + \cdots + \alpha_n\boldsymbol{P}_n \\ &= \boldsymbol{P}_0 + \alpha_1(\boldsymbol{P}_1 - \boldsymbol{P}_0) + \cdots + \alpha_n(\boldsymbol{P}_n - \boldsymbol{P}_0) \\ &= \boldsymbol{P}_0 + \alpha_1\vec{e}_1 + \cdots + \alpha_n\vec{e}_n\end{aligned}$$

where $\vec{e}_i = \boldsymbol{P}_i - \boldsymbol{P}_0$ are the coordinate axes and $\boldsymbol{P}_0$ is the origin point.

Even though the cells are not necessarily simplexes, we can use this same idea to define a unique reference frame for every tuple (Figure 5.13):

Definition 3. *The* tuple frame *corresponding to an n-dimensional tuple τ is $\{\boldsymbol{p}, \vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n\}$, where $\boldsymbol{p}$ is the position of the vertex $\tau[0]$ and $\vec{e}_k$ is the vector $\boldsymbol{p}_k - \boldsymbol{p}$, where $\boldsymbol{p}_k$ is the position of the vertex of the tuple $\kappa = \sigma_0 \circ \sigma_1 \circ \cdots \circ \sigma_{k-1} \circ \tau$.*

The definition of $\vec{e}_k$ means that each $\vec{e}_k$ is a vector with the direction and length of an edge incident on $\tau[0]$. $\vec{e}_1$ is in the direction of the tuple's own edge; $\vec{e}_2$ is in the direction of the edge which shares the tuple's face; $\vec{e}_3$ is in the direction of the edge which shares the tuple's volume but not its face, and so on.

The tuple frame defines a set of tuple coordinates for any point within the n-cell. If the cell is a simplex, then the coordinates are exactly equivalent to the barycentric coordinates on the cell. Even if the cell is not a simplex, the tuple coordinates still uniquely define points within it, though some points in the cell might have coordinates greater than one or less than zero. Moreover, each tuple in an n-cell defines a different set of tuple coordinates; thus, there are many possible coordinate systems which define points in each cell.

Each set of tuple coordinates provides a map from a Euclidean space on that cell to the embedding space; the totality of these maps (one for each tuple in the divided manifold)

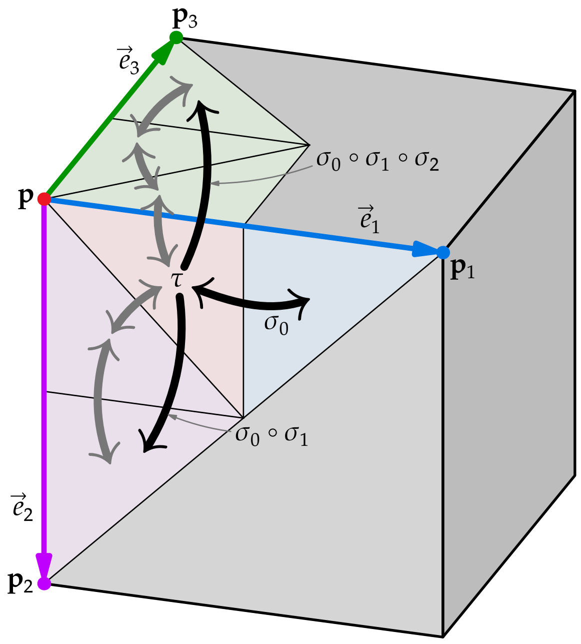


Figure 5.13: The construction of the tuple frame $\{\boldsymbol{p}, \vec{e}_1, \vec{e}_2, \vec{e}_3\}$ corresponding to the red tuple τ. The coordinate origin $\boldsymbol{p}$ is the position of τ's vertex $\tau[0]$; the first coordinate axis $\vec{e}_1$ is in the direction of τ's edge, which lies between $\tau[0]$ and $(\sigma_0 \circ \tau)[0]$. The second axis $\vec{e}_2$ is in the direction of the other edge on τ's face: the other endpoint of this edge is $(\sigma_0 \circ \sigma_1 \circ \tau)[0]$. Finally, the third axis $\vec{e}_3$ is in the direction of the other edge on the other face on τ's volume; that makes its other endpoint $(\sigma_0 \circ \sigma_1 \circ \sigma_2 \circ \tau)[0]$.

thus forms a complete atlas covering the manifold. In other words, the tuple coordinates provide us with an *intrinsic* coordinate system for the divided manifold. When we traverse the cell complex, each flip function σ_i corresponds to a transition function between tuple frames:

$$\{\boldsymbol{p}, \vec{e}_1, \dots, \vec{e}_n\} \mapsto \{\boldsymbol{p}', \vec{e}_1', \dots, \vec{e}_n'\}$$

Examples of these transition functions are derived in Section 6.3, where they are used to propagate curves across a polygonized surface.

The tuple frame might be left- or right-handed, depending on the particular embedding. However, the transition functions always *reverse the orientation of the coordinate frame* (Figure 5.14). We thus see that the orientation of the coordinate frame is related to the total orientation of a cell tuple as discussed in Section 5.1.

The vectors in the tuple frame are not in general perpendicular or of unit length, but we can define a frame whose coordinate axes are orthonormal (Figure 5.15):

Definition 4. *The* orthonormal tuple frame *corresponding to* τ *is* $\{\boldsymbol{p}, \hat{\xi}_1, \hat{\xi}_2, \dots, \hat{\xi}_n\}$, *where* $\boldsymbol{p}$ *is as above,* $\hat{\xi}_1 = \vec{e}_1/|\vec{e}_1|$, *and* $\hat{\xi}_k$ *is the inward-pointing normal to the cell* $\tau[k-1]$ *lying in the subspace of the cell* $\tau[k]$.

Thus, $\hat{\xi}_1$ points inward from $\boldsymbol{p}$ along the edge $\tau[1]$; $\hat{\xi}_2$ is orthogonal to the edge and points into the face $\tau[2]$; $\hat{\xi}_2$ is normal to this face and points into the volume $\tau[3]$, and so on. Whenever the cells are not geometrically degenerate, we can derive the orthonormal

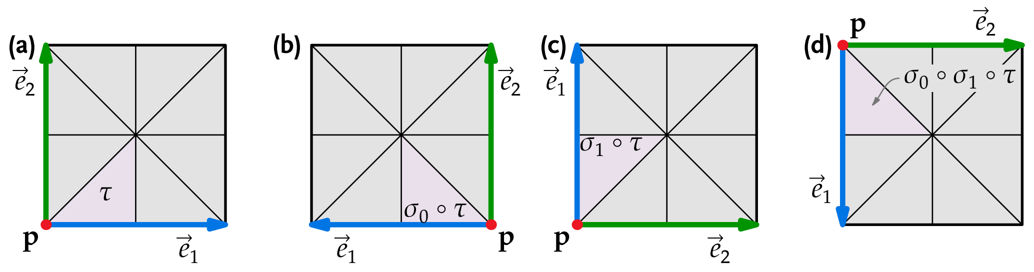


Figure 5.14: Flips reverse the orientation of tuple coordinates. The coordinate system of τ (a) is right-handed, while the coordinate systems of $\sigma_0 \circ \tau$ (b) and $\sigma_1 \circ \tau$ (c) are left-handed; flipping once more, the coordinate system of $\sigma_0 \circ \sigma_1 \circ \tau$ (d) is again right-handed.

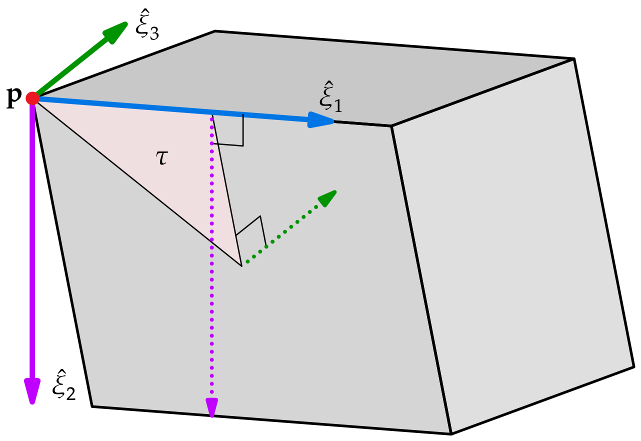


Figure 5.15: The construction of the orthonormal tuple frame $\{p, \hat{\xi}_1, \hat{\xi}_2, \hat{\xi}_3\}$ corresponding to the pink tuple τ. Again, the coordinate origin p is the position of τ's vertex $\tau[0]$, and the first coordinate axis $\hat{\xi}_1$ is in the direction of τ's edge. Now, however, the second axis $\hat{\xi}_2$ is the vector in the plane of τ's face pointing inward from $\hat{\xi}_1$ (purple), and the third axis $\vec{e}_3$ is the vector normal to τ's face pointing into τ's volume (green).

tuple frame from the non-orthogonal frame through Gram-Schmidt orthogonalization. This is the purpose of Algorithm 5.6.

Algorithm 5.6 Calculate the orthonormal tuple frame through Gram-Schmidt orthogonalization of the tuple edge directions

```
Input: A tuple $\tau$
Require: Each vertex $v$ in the cell complex has a position $\boldsymbol{P}(v)$
Output: The orthonormal tuple frame $(\boldsymbol{p}, \hat{\xi}_1, \dots, \hat{\xi}_n)$ corresponding to $\tau$
 1: procedure OrthonormalTupleFrame($\tau$)
 2:     $\boldsymbol{p} \leftarrow \boldsymbol{P}(\tau[0])$
 3:     for $i \in \{1, \dots, n\}$ do
 4:         $\tau' \leftarrow \tau.\mathbf{flip}(i-1, \dots, 1, 0)$
 5:         $\vec{e}_i \leftarrow \boldsymbol{P}(\tau'[0]) - \boldsymbol{p}$
 6:         for $j \in \{1, \dots, i-1\}$ do          ▷ Gram-Schmidt orthogonalization
 7:             $\vec{e}_i \leftarrow \vec{e}_i - (\vec{e}_i \cdot \hat{\xi}_j)\hat{\xi}_j$
 8:         end for
 9:         $\hat{\xi}_i \leftarrow \vec{e}_i / |\vec{e}_i|$
10:     end for
11:     return $(\boldsymbol{p}, \hat{\xi}_1, \dots, \hat{\xi}_n)$
12: end procedure
```

The point $\boldsymbol{p}$ is just the position of the input tuple τ's vertex (line 2). The basis vectors $\hat{\xi}_i$ are found one at a time (lines 3–10). In order to find the ith vector, we first find the tuple τ' which contains the edge and vertex which lie on τ's i-cell but not on its $(i-1)$-cell (line 4). The vector from $\boldsymbol{p}$ to τ''s vertex is the non-orthonormal tuple frame axis $\vec{e}_i$ (line 5). This is orthogonalized against the already-found axes by the Gram-Schmidt technique (lines 6–8), then normalized (line 9). Finally, once all of the basis vectors have been computed, the frame is returned (line 11).

Measure The orthonormal tuple frame can be used to compute the measure of a cell. We state arbitrarily that the measure of a vertex is 1. The algorithm for computing the measure of a cell of higher dimension is stated, with proof in two dimensions only, by Franklin (1992). Here I prove the general multidimensional case:

Theorem 5.1. *Let τ be a cell tuple in a d-dimensional cell complex with a default embedding. Let $\nu(\tau) = \prod(\vec{p} \cdot \hat{\xi}_i)$, where $\{\boldsymbol{p}, \hat{\xi}_1, \hat{\xi}_2, \dots, \hat{\xi}_n\}$ is the orthonormal tuple frame associated with τ. If C is a k-cell with measure $\|C\|$, then*

$$\|C\| = \frac{(-1)^k}{k!} \sum_{\substack{\tau \\ C \in \tau}} \nu(\tau) \tag{5.2}$$

where the sum is taken over all tuples τ with k-cell C.

Proof. The measure of a closed space is the integral of a function with constant value 1 over the space:

$$\|C\| = \int_C 1 \, dC.$$

We introduce the function $\nabla \cdot \vec{x}$; in k dimensions, this has the constant value k:

$$\|C\| = \frac{1}{k} \int_C \nabla \cdot \vec{x}\, dC.$$

By the Divergence Theorem, this becomes

$$\|C\| = \frac{1}{k} \int_{\partial C} \vec{x} \cdot d\hat{n},$$

where $\hat{n}$ is the outward-facing normal to the boundary of C. Since the boundary of C is made up of flat $(k-1)$-cells $\phi \prec C$, we see that

$$\|C\| = \frac{1}{k} \sum_{\phi \prec C} \int_\phi \vec{x} \cdot \hat{n}_\phi \, d\phi;$$

but the *inward*-facing normal to ϕ is $\hat{\xi}_k$; that is, $\hat{n}_\phi = -\hat{\xi}_k$, so:

$$\|C\| = -\frac{1}{k} \sum_{\phi \prec C} \int_\phi \vec{x} \cdot \hat{\xi}_k \, d\phi.$$

Since $\hat{\xi}_k$ is perpendicular to ϕ, its dot product with any point in ϕ is the same. ϕ contains $\boldsymbol{p}$, so

$$\|C\| = -\frac{1}{k} \sum_{\phi \prec C} \int_\phi \vec{p} \cdot \hat{\xi}_k \, d\phi = -\frac{1}{k} \sum_{\phi \prec C} (\vec{p} \cdot \hat{\xi}_k) \int_\phi d\phi.$$

We can recursively perform the same derivation to find that

$$\begin{aligned}
\|C\| &= -\frac{1}{k} \sum_{\phi \prec C} (\vec{p} \cdot \hat{\xi}_k) \frac{-1}{k-1} \sum_{\chi \prec \phi} (\vec{p} \cdot \hat{\xi}_{k-1}) \int_\chi d\chi \\
&= -\frac{1}{k} \sum_{\phi \prec C} (\vec{p} \cdot \hat{\xi}_k) \frac{-1}{k-1} \sum_{\chi \prec \phi} (\vec{p} \cdot \hat{\xi}_{k-1}) \cdots \frac{-1}{2} \sum_e (\vec{p} \cdot \hat{\xi}_2) \int_e de
\end{aligned}$$

where e is an edge of the cell C. If e's endpoints are at $\vec{p}$ and $\vec{p}'$, then

$$\begin{aligned}
\int_e de = \|\vec{e}\| &= \vec{e} \cdot \hat{e} \\
&= (\vec{p}' - \vec{p}) \cdot \hat{e} \\
&= -(\vec{p} \cdot \hat{\xi}_1 + \vec{p}' \cdot (-\hat{\xi}_1))
\end{aligned}$$

because the inward-facing normal from $\vec{p}$ is opposite to the inward-facing normal from $\vec{p}'$. But then

$$\begin{aligned}
\|C\| &= -\frac{1}{k} \sum_{\phi \prec C} (\vec{p} \cdot \hat{\xi}_k) \frac{-1}{k-1} \sum_{\chi \prec \phi} (\vec{p} \cdot \hat{\xi}_{k-1}) \cdots \frac{-1}{2} \sum_e (\vec{p} \cdot \hat{\xi}_2) \frac{-1}{1} \sum_p \vec{p} \cdot \hat{\xi}_1 \\
&= \sum_{\phi \prec C} \sum_{\chi \prec \phi} \cdots \sum_{p \prec e} \left(\frac{-1}{k} \frac{-1}{k-1} \cdots \frac{-1}{1} \right) (\vec{p} \cdot \hat{\xi}_k)(\vec{p} \cdot \hat{\xi}_{k-1}) \cdots (\vec{p} \cdot \hat{\xi}_1) \\
&= \sum_{\phi \prec C} \sum_{\chi \prec \phi} \cdots \sum_{p \prec e} \frac{(-1)^k}{k!} \prod_i (\vec{p} \cdot \hat{\xi}_i).
\end{aligned}$$

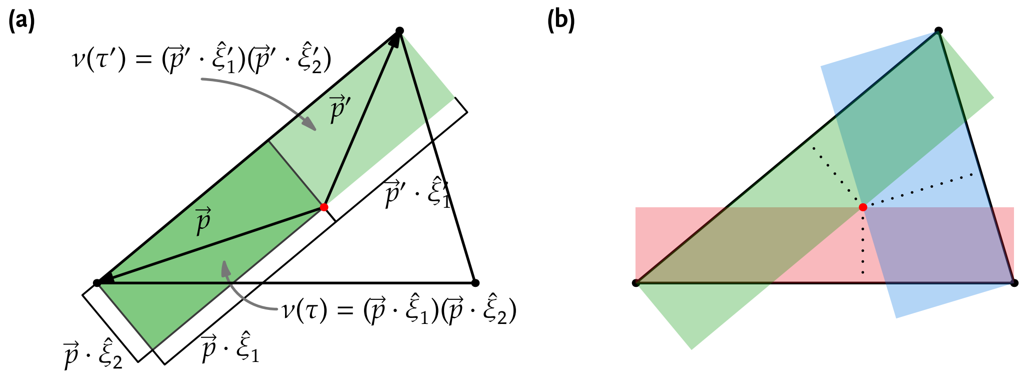

(a)
$\nu(\tau') = (\vec{p}' \cdot \hat{\xi}'_1)(\vec{p}' \cdot \hat{\xi}'_2)$
$\vec{p}'$
$\vec{p}' \cdot \hat{\xi}'_1$
$\vec{p}$
$\nu(\tau) = (\vec{p} \cdot \hat{\xi}_1)(\vec{p} \cdot \hat{\xi}_2)$
$\vec{p} \cdot \hat{\xi}_2$
$\vec{p} \cdot \hat{\xi}_1$
(b)

Figure 5.16: Calculating the measure in two dimensions using Equation 5.2. (a) The value $\nu(\tau) = (\vec{p} \cdot \hat{\xi}_1)(\vec{p} \cdot \hat{\xi}_2)$ is the area of a rectangle aligned with the edge. The sum $\nu(\tau) + \nu(\tau')$ of the areas of both rectangles on an edge is the area of the rectangle whose base is the edge. (b) The sum of ν for all tuples on the triangle is equal to the sum of the areas of three rectangles. The entire triangle is covered twice by the sum of the rectangles, so $\|C\| = 1/2 \sum \nu(\tau)$.

All of the sums mean that there is one term for each cell sequence $p \prec e \prec \cdots \prec \chi \prec \phi \prec C$; that is, one term for each cell tuple containing C:

$$\|C\| = \frac{(-1)^k}{k!} \sum_{\substack{\tau \\ C \in \tau}} \nu(\tau)$$

as expected. □

Some intuition to how Equation 5.2 produces the measure is shown in Figure 5.16. In two dimensions, each value $\nu(\tau)$ is the area of a rectangle aligned with the edge $\tau[1]$ whose opposite vertices are $\tau[0]$ and the coordinate origin; the vector $\vec{p}$ is thus its diagonal. The sum of ν for both tuples which share that edge, i.e. $\nu(\tau) + \nu(\sigma_1 \circ \tau)$, is the area of the rectangle whose base is that edge and whose altitude is the distance to the origin (Figure 5.16a). Overlaying the rectangles corresponding to all of the tuples (Figure 5.16b), we see that some parts of the triangle are covered by two rectangles. The parts of the triangle covered by only one rectangle, however, correspond exactly to the parts of that rectangle which lie outside the triangle. This means that the triangle is covered exactly twice by the sum of the rectangles, or $\|C\| = 1/2 \sum \nu(\tau)$.

Algorithm 5.7 finds the measure of a cell C by performing a traversal of its tuples. The measure will be held in accumulator μ, which is initialized to zero in line 3. We start our traversal at an arbitrary tuple τ_0 containing C (line 4). The traversal is performed in exactly the same way as that in Algorithm 5.2, with the difference that the traversal only uses flips in dimensions less than the dimension of C (line 16). This dimension is retrieved as k in line 2. Since there are no flips in dimension k, every tuple we visit must include C.

For each tuple τ we visit, we first compute its orthonormal tuple frame (line 10). We then find the value of $\nu(\tau)$ (lines 11–14) and add it to the accumulator μ (line 15). Finally, once all of the tuples have been visited, we return the total measure (line 24).

Algorithm 5.7 Find the measure of a cell C by performing a traversal of its tuples and applying Equation 5.2

```
Input: A cell $C$
Require: Each vertex $v$ in the cell complex has a position $\boldsymbol{P}(v)$
Output: The measure $\mu$ of $C$
 1: procedure MEASURE($C$)
 2:     $k \leftarrow$ dimension of $C$
 3:     $\mu \leftarrow 0$                                      ▷ accumulator for measure
 4:     $\tau_0 \leftarrow$ tuple containing $C$
 5:     $pending \leftarrow \{\tau_0\}$
 6:     $visited \leftarrow \{\tau_0\}$
 7:     while $pending \neq \{\}$ do                             ▷ traversal of $C$'s tuples
 8:         pick $\tau \in pending$
 9:         $pending \leftarrow pending \setminus \{\tau\}$
10:         $(\vec{p}, \hat{\xi}_1, \dots, \hat{\xi}_n) \leftarrow$ ORTHONORMALTUPLEFRAME($\tau$)
11:         $\nu \leftarrow 1$
12:         for $i \in \{1, \dots, k-1\}$ do                    ▷ compute $\nu = \prod(\vec{p} \cdot \hat{\xi}_i)$
13:             $\nu \leftarrow \nu * (\vec{p} \cdot \hat{\xi}_i)$
14:         end for
15:         $\mu \leftarrow \mu + \nu$
16:         for $d \in \{0, \dots, k-1\}$ do              ▷ only flip in dimensions less than $k$
17:             $\tau' \leftarrow \tau.\mathbf{flip}(d)$
18:             if $\tau' \notin visited$ then
19:                 $pending \leftarrow pending \cup \{\tau'\}$
20:                 $visited \leftarrow visited \cup \{\tau'\}$
21:             end if
22:         end for
23:     end while
24:     return $\frac{(-1)^k}{k!}\mu$
25: end procedure
```

Centroid An algorithm for computing the centroid of a cell is given for three-dimensional cells with triangular faces by Nürnberg (2013). A generalization is more complicated than the expression for computing measure, but takes much the same form: the centroid of C is the sum, over every tuple containing C, of a function of that tuple's orthonormal tuple frame:

Theorem 5.2. *Let τ be a cell tuple in a d-dimensional cell complex with a default embedding. Let*

$$\eta(\hat{\epsilon},\tau)=\begin{cases}\dfrac{(\vec{p}\cdot\hat{\epsilon})^{k+1}}{\hat{\xi}_1\cdot\hat{\epsilon}}\displaystyle\prod_{2\le i\le k}(\hat{\epsilon}\cdot\hat{\xi}_i) & \text{if } \hat{\xi}_1\cdot\hat{\epsilon}\neq 0\\ (k+1)\,(\vec{p}\cdot\hat{\epsilon})^k\,(\vec{p}\cdot\hat{\xi}_1)\displaystyle\prod_{2\le i\le k}(\hat{\epsilon}\cdot\hat{\xi}_i) & \text{if } \hat{\xi}_1\cdot\hat{\epsilon}=0,\end{cases}$$

where $\{\boldsymbol{p},\hat{\xi}_1,\hat{\xi}_2,\dots,\hat{\xi}_n\}$ is the orthonormal tuple frame associated with τ, and $\hat{\epsilon}$ is any unit vector. If C is a k-cell with measure $\|C\|$ and centroid $\vec{Z}$, then

$$\|C\|\,\vec{Z}\cdot\hat{\epsilon}=\frac{(-1)^k}{(k+1)!}\sum_{\substack{\tau\\ C\in\tau}}\eta(\hat{\epsilon},\tau) \tag{5.3}$$

where the sum is taken over all tuples τ with k-cell C.

Proof. The centroid $\vec{Z}$ of C is average of every point in C, i.e.

$$\vec{Z}=\frac{\int_C \vec{x}\,dC}{\|C\|},$$

so

$$\|C\|\,\vec{Z}\cdot\hat{\epsilon}=\int_C \vec{x}\cdot\hat{\epsilon}\,dC.$$

We want to apply the Divergence Theorem to this integral:

$$\begin{aligned}\|C\|\,\vec{Z}\cdot\hat{\epsilon}=\int_C \vec{x}\cdot\hat{\epsilon}\,dC&=\int_C \nabla\cdot\left(\frac{(\vec{x}\cdot\hat{\epsilon})^2}{2}\,\hat{\epsilon}\right)dC\\&=\int_{\partial C}\left(\frac{(\vec{x}\cdot\hat{\epsilon})^2}{2}\,\hat{\epsilon}\right)\cdot d\hat{n}\\&=\frac{1}{2}\sum_{\phi<C}\int_\phi(\vec{x}\cdot\hat{\epsilon})^2\,\hat{\epsilon}\cdot\hat{n}_\phi\,d\phi\end{aligned}$$

where $\hat{n}_\phi$ is the outward-pointing normal to ϕ. We replace this with the inward-pointing normal $\hat{\xi}_k=-\hat{n}_\phi$:

$$\int_C \vec{x}\cdot\hat{\epsilon}\,dC=\sum_{\phi<C}\frac{-\hat{\epsilon}\cdot\hat{\xi}_k}{2}\int_\phi(\vec{x}\cdot\hat{\epsilon})^2\,d\phi.$$

By a similar argument,

$$\int_\phi (\vec{x}\cdot\hat{\epsilon})^2\, d\phi = \sum_{\chi<\phi} \frac{-\hat{\epsilon}\cdot\hat{\xi}_{k-1}}{3} \int_\chi (\vec{x}\cdot\hat{\epsilon})^3\, d\chi$$

and, in general,

$$\int_{C_i} (\vec{x}\cdot\hat{\epsilon})^{k+1-i}\, dC_i = \sum_{C_{i-1}<C_i} \frac{-\hat{\epsilon}\cdot\hat{\xi}_i}{k+2-i} \int_{C_{i-1}} (\vec{x}\cdot\hat{\epsilon})^{k+2-i}\, dC_{i-1}$$

until we reach an edge e. Here we must be careful: in the general case, we find that

$$\begin{aligned}\int_e (\vec{x}\cdot\hat{\epsilon})^k\, de &= \int_0^1 \left((\vec{p}_0 + t(\vec{p}_1-\vec{p}_0))\cdot\hat{\epsilon}\right)^k \|\vec{p}_1-\vec{p}_0\|\, dt \\ &= \frac{\|\vec{p}_1-\vec{p}_0\|}{(k+1)\,(\vec{p}_1-\vec{p}_0)\cdot\hat{\epsilon}} \left(\left((\vec{p}_0 + t(\vec{p}_1-\vec{p}_0))\cdot\hat{\epsilon}\right)^{k+1}\Big|_0^1 \right. \\ &= \frac{1}{(k+1)\,\hat{\xi}_1\cdot\hat{\epsilon}} \left((\vec{p}_1\cdot\hat{\epsilon})^{k+1} - (\vec{p}_0\cdot\hat{\epsilon})^{k+1}\right).\end{aligned}$$

This cannot be done, of course, if $\hat{\xi}_1\cdot\hat{\epsilon}=0$. In that case, however, the edge is perpendicular to $\hat{\epsilon}$, so the value $(\vec{x}\cdot\hat{\epsilon})^k$ is the same at all points; then

$$\begin{aligned}\int_e (\vec{x}\cdot\hat{\epsilon})^k\, de &= (\vec{p}\cdot\hat{\epsilon})^k \int_e de \\ &= (\vec{p}\cdot\hat{\epsilon})^k \left(-\vec{p}_0\cdot\hat{\xi}_1 + \vec{p}_1\cdot(-\hat{\xi}_1)\right) \\ &= -\left((\vec{p}_0\cdot\hat{\epsilon})^k\,(\vec{p}_0\cdot\hat{\xi}_1) + (\vec{p}_1\cdot\hat{\epsilon})^k\,(\vec{p}_1\cdot -\hat{\xi}_1)\right).\end{aligned}$$

We can thus write the entire sum as either

$$\|C\|\,\vec{Z}\cdot\hat{\epsilon} = \sum_{\phi<C}\frac{-\hat{\epsilon}\cdot\hat{\xi}_k}{2}\sum_{\chi<\phi}\frac{-\hat{\epsilon}\cdot\hat{\xi}_{k-1}}{3}\cdots\sum_{e<f}\frac{-\hat{\epsilon}\cdot\hat{\xi}_2}{k}\sum_{p<e}\frac{-1}{(k+1)\,\hat{\xi}_1\cdot\hat{\epsilon}}(\vec{p}\cdot\hat{\epsilon})^{k+1}$$

$$\text{or}\quad \|C\|\,\vec{Z}\cdot\hat{\epsilon} = \sum_{\phi<C}\frac{-\hat{\epsilon}\cdot\hat{\xi}_k}{2}\sum_{\chi<\phi}\frac{-\hat{\epsilon}\cdot\hat{\xi}_{k-1}}{3}\cdots\sum_{e<f}\frac{-\hat{\epsilon}\cdot\hat{\xi}_2}{k}\sum_{p<e}-(\vec{p}\cdot\hat{\xi}_1)\,(\vec{p}\cdot\hat{\epsilon})^k,$$

i.e.

$$\|C\|\,\vec{Z}\cdot\hat{\epsilon} = \frac{(-1)^k}{(k+1)!}\sum_{\substack{\tau\\ C\in\tau}}\eta(\tau). \qquad \square$$

Algorithm 5.8 uses this slightly more complex function to compute the centroid of a cell C, again using a traversal. The traversal works exactly the same as in Algorithm 5.7, so I will concentrate on how each tuple τ' is processed. First (line 6) the orthonormal tuple frame for τ' is calculated. The centroid term corresponding to τ' will be accumulated in variable ω. First, we multiply in the terms $-\hat{\epsilon}\cdot\hat{\xi}_m$ for $2\le m\le k$ (lines 8–8); we also accumulate the factorial at this time, though, since the same indexes are involved, we divide each term involving $\hat{\xi}_m$ by m, not $(k+2-m)$ as suggested by the derivation.

Algorithm 5.8 Find the component of the centroid of a cell C parallel to $\hat{\epsilon}$ by performing a traversal of its tuples.

Input: A cell C and a basis vector $\hat{\epsilon}$
Require: Each vertex v in the cell complex has a position $\boldsymbol{P}(v)$
Output: The component of the centroid of C parallel to $\hat{\epsilon}$

```
1:  procedure CentroidComponent($C$,$\hat{\epsilon}$)
2:      $k \leftarrow$ dimension of $C$
3:      $z \leftarrow 0$
4:      $\tau_0 \leftarrow$ tuple containing $C$
5:      for $\tau'$ from traversal starting from $\tau_0$ do
6:          $(\vec{p}, \hat{\xi}_1, \dots, \hat{\xi}_n) \leftarrow$ OrthonormalTupleFrame($\tau'$)
7:          $\omega \leftarrow 1$
8:          for m from 2 to k do
9:              $\omega \leftarrow \omega * -\frac{\hat{\epsilon} \cdot \hat{\xi}_m}{m}$
10:         end for
11:         if $\hat{\epsilon} \cdot \hat{\xi}_1 = 0$ then
12:             $\omega \leftarrow \omega * (\vec{p} \cdot \hat{\xi}_1) * (\vec{p} \cdot \hat{\epsilon})^k$
13:         else
14:             $\omega \leftarrow \omega * \frac{(\vec{p} \cdot \hat{\epsilon})^{k+1}}{\hat{\epsilon} \cdot \hat{\xi}_1}$
15:         end if
16:         $z \leftarrow z + \omega$
17:     end for
18:     $\mu \leftarrow$ Measure($C$)
19:     return $z/\mu$
20: end procedure
```

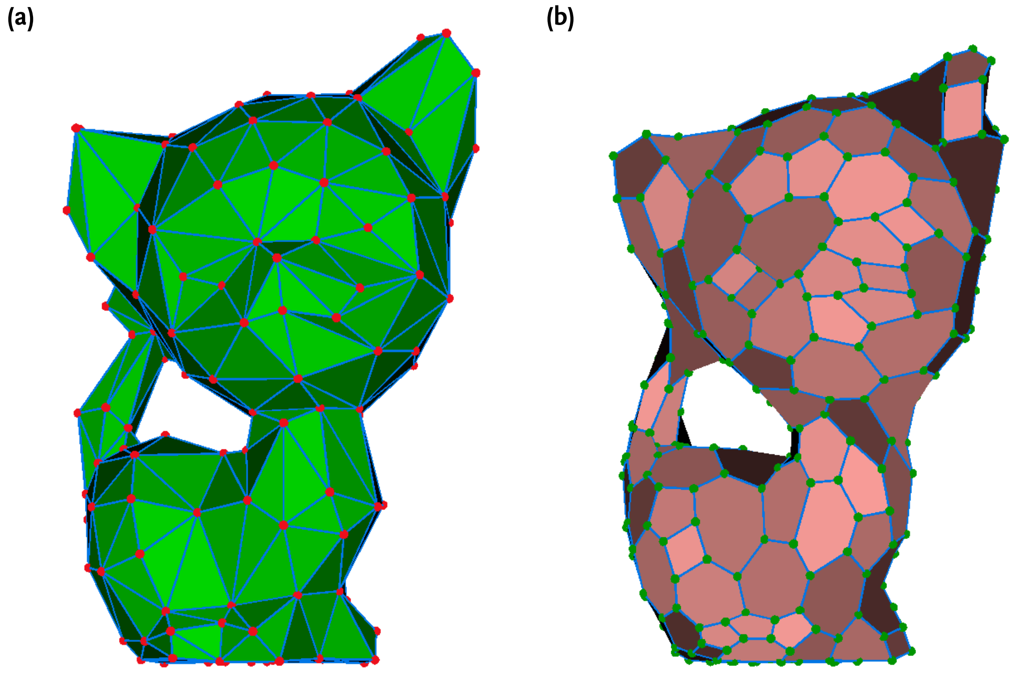


Figure 5.17: A two-dimensional cell complex (a) and its dual complex (b). Vertices (red) in the original complex correspond to faces (red) in the dual complex. Edges connecting vertices correspond to edges between faces. Faces (green) in the original complex correspond to vertices (green) in the dual; these have been placed at the centroid of their corresponding face.

Then we check whether $\hat{e}$ is perpendicular to $\hat{\xi}_1$, and multiply by the appropriate term (lines 11–15). Finally, we add the accumulated term ω to z (line 16).

After visiting all of the tuples, we must still divide z by the measure of C; this is done in lines 18 and 19.

One application of finding the centroid of the cells in the complex is in creating the complex's *dual*. This is the cell complex which has a vertex corresponding to each n-cell of the original complex, connected by edges which correspond to the $(n-1)$-cells of the original, and so on, up to n-cells which correspond to the original vertices, separated by $(n-1)$-cells which correspond to the original edges (Figure 5.17). We define the *dual* of a k-cell C as the $(n-k)$-cell **DUAL**(C). The dual of the dual complex is the original complex; thus, the **DUAL** operation is an involution, or

$$\mathbf{DUAL}(\mathbf{DUAL}(C)) = C.$$

The dual of $\textcircled{\perp}$ is $\textcircled{\top}$, and vice versa.

Because the adjacencies correspond so closely, the flip structure is mostly intact. Vertices which used to be adjacent over an edge correspond to n-cells which are adjacent over an $(n-1)$-cell, except that the $(i+1)$ and $(i-1)$ positions of the flip have changed places:

$$\left\langle \begin{array}{c} c_{i+1} \\ c_i \;\; c'_i \\ c_{i-1} \end{array} \right\rangle \mapsto \left\langle \begin{array}{c} \mathbf{DUAL}(c_{i-1}) \\ \mathbf{DUAL}(c_i) \;\; \mathbf{DUAL}(c'_i) \\ \mathbf{DUAL}(c_{i+1}) \end{array} \right\rangle.$$

The positions of vertices in the dual complex is ambiguous. However, a common choice is to place dual vertices at the *centroids* of the corresponding n-cells; this is what has been done in Figure 5.17.

One problem to note is that if the complex has a boundary, then cells on the boundary are dual to unbounded cells. This can be easily seen in the flip structure representation, as cells on the boundary are adjacent to the pseudocell $\textcircled{\infty}$, or "the space outside the manifold", whose dual would be a "point at infinity". While this makes the embedding of the dual of a manifold with boundary problematic, the dual of a manifold without boundary has a clear embedding.

6 Geometric modeling

In this chapter, I present Cell Complex Framework models in several areas related to geometric modeling. In Section 6.1 I talk about global subdivision algorithms for mesh smoothing: the Catmull-Clark and Kobbelt $\sqrt{3}$ methods in two dimensions, and three-dimensional extensions to both. Section 6.2 covers the shrink-wrap algorithm for polygonizing an implicit surface. Finally, in Section 6.3 I go into detail on a model which draws *geodesics*, the manifold counterpart to straight lines, on a subdivided manifold surface.

6.1 Smoothing by global subdivision

In Section 5.2 I demonstrated the butterfly global subdivision method. There are a number of such methods which refine a mesh by subdividing faces and edges, and positioning vertices as affine combinations of the positions of nearby vertices. If the weights used to find vertex positions are chosen carefully, then successive refinements of the mesh asymptotically approach a smooth surface. The butterfly subdivision, which does not move existing vertices, is an *interpolating* subdivision; the limit surface passes through all of the vertices in the original mesh. In this section, I will cover CCF implementations of two non-interpolating subdivision schemes: Catmull-Clark subdivision (Catmull and Clark 1978) and Kobbelt's $\sqrt{3}$ subdivision (Kobbelt 2000).

6.1.1 Catmull-Clark subdivision in two dimensions

The butterfly subdivision described in Section 5.2 inserts a new vertex on each edge, then subdivides each triangular face into four triangular faces (Figure 6.1a). The Catmull-Clark subdivision, on the other hand, inserts a new vertex on each edge as well as in the interior of each face, then splits each n-gon into n quadrilateral faces (Figures 6.1b–d). Consequently, every face is a quadrilateral after the first refinement step.

The new vertex in the middle of face f (a *face*-vertex) is placed at the average position of the vertices of f:

$$\boldsymbol{P}'(f) = \sum_{i=1}^{n} \frac{1}{n} \boldsymbol{P}(v_i).$$

The position of the vertex splitting edge e (an *edge*-vertex) is best described in terms of the

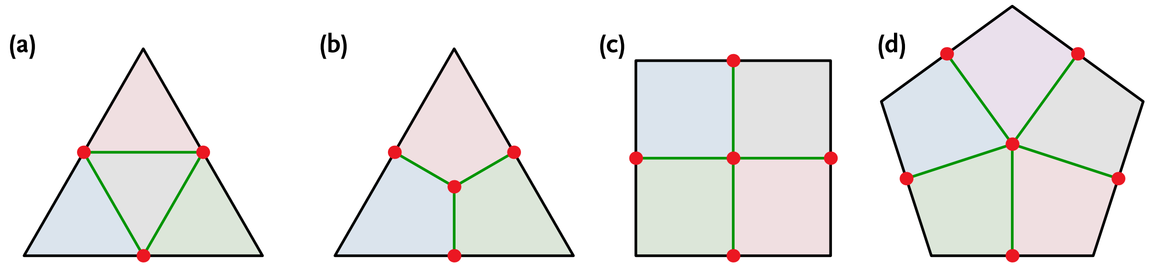


Figure 6.1: (a) In butterfly subdivision, new vertices (red) are inserted on each edge and each triangular face is split into four new faces. (b)-(d) In Catmull-Clark subdivision, new vertices are inserted on each edge and in the interior of each face, and each n-gonal face is split into n quadrilateral faces.

(newly computed) positions of the face-vertices on the faces adjoining e and the (already known) positions of the *vertex*-vertices which are e's endpoints. If e has endpoints v_0 and v_1, and adjoins faces f_a and f_b, then the position of the new vertex is

$$\boldsymbol{P}'(e) = \frac{\boldsymbol{P}'(f_a) + \boldsymbol{P}'(f_b) + \boldsymbol{P}(v_0) + \boldsymbol{P}(v_1)}{4}. \tag{6.1}$$

Finally, the new position of a vertex-vertex is a combination of its old position and that of the adjoining edge and face vertices. If vertex v adjoins n edges and faces, then its new position is

$$\boldsymbol{P}'(v) = \frac{n-2}{n}\boldsymbol{P}(v) + \frac{1}{n}\sum_i \frac{\boldsymbol{P}'(e_i)}{n} + \frac{1}{n}\sum_i \frac{\boldsymbol{P}'(f_i)}{n}. \tag{6.2}$$

Algorithm 6.1 performs Catmull-Clark refinement on an input mesh.

The new vertex positions are computed first (lines 2–29). The position of each face-vertex is simply the average of the face's incident vertices. We find these vertices by iterating a tuple τ around the face by repeated application of **flip**(0, 1) (lines 7–11). At each step, we add $\tau[0]$'s position to the accumulating sum of vertex positions (line 8) and increment the number of incident vertices (line 9). The position of the face-vertex is then the sum of vertex positions divided by the number of incident vertices (line 12).

The position of an edge-vertex is the average of the positions of its incident face- and vertex-vertices (Equation 6.1). Any tuple containing e (line 15) has one of these vertices as 0-cell and one of these faces as 2-cell. The other vertex is then **other**(0), and the other face is **other**(2). The four positions are averaged in line 16 and the result saved as the position of e's edge-vertex.

The new position of a vertex-vertex v is a combination of its old position and the average positions of the edge- and face-vertices incident to v (Equation 6.2). These positions are accumulated while iterating a tuple around the vertex with **flip**(1, 2) (lines 23–27). During this iteration, we also find n, the valence of v (line 24). Finally, we compute the new position of v in line 28.

Once all of the new positions have been calculated, we proceed to subdividing the mesh. Each edge is split (line 32) and the new vertex's position is set to the computed position assigned to the edge (line 33). Because of the introduction of a face-vertex, face splitting is somewhat different from midpoint subdivision. However, it begins in

Algorithm 6.1 Catmull-Clark mesh subdivision (part 1)

```
Require: Each vertex $v$ has a position $\boldsymbol{P}(v)$
 1: procedure CatmullClarkSubdivision
 2:     for all face $f$ do                                  ▷ calculate face-vertex positions
 3:         $\boldsymbol{P}'(f) \leftarrow \mathbf{0}$
 4:         $n \leftarrow 0$
 5:         $\tau \leftarrow$ tuple containing $f$
 6:         $\tau_{\text{start}} \leftarrow \tau$
 7:         repeat
 8:             $\boldsymbol{P}'(f) \leftarrow \boldsymbol{P}'(f) + \boldsymbol{P}(\tau[0])$
 9:             $n \leftarrow n + 1$
10:             $\tau.\mathbf{flip}(0, 1)$
11:         until $\tau = \tau_{\text{start}}$
12:         $\boldsymbol{P}'(f) \leftarrow 1/n\, \boldsymbol{P}'(f)$
13:     end for
14:     for all edge $e$ do                                  ▷ calculate edge-vertex positions
15:         $\tau \leftarrow$ tuple containing $e$
16:         $\boldsymbol{P}'(e) \leftarrow 1/4\big(\boldsymbol{P}'(\tau[2]) + \boldsymbol{P}'(\tau.\mathbf{other}(2)) + \boldsymbol{P}(\tau[0]) + \boldsymbol{P}(\tau.\mathbf{other}(0))\big)$
17:     end for
18:     for all vertex $v$ do                                ▷ calculate vertex-vertex positions
19:         $\boldsymbol{P}'(v) \leftarrow \mathbf{0}$
20:         $n \leftarrow 0$
21:         $\tau \leftarrow$ tuple containing $v$
22:         $\tau_{\text{start}} \leftarrow \tau$
23:         repeat
24:             $n \leftarrow n + 1$
25:             $\boldsymbol{P}'(v) \leftarrow \boldsymbol{P}'(v) + \boldsymbol{P}'(\tau[1]) + \boldsymbol{P}'(\tau[2])$
26:             $\tau.\mathbf{flip}(1, 2)$
27:         until $\tau = \tau_{\text{start}}$
28:         $\boldsymbol{P}(v) \leftarrow \frac{n-2}{n}\boldsymbol{P}(v) + \frac{1}{n^2}\boldsymbol{P}'(v)$
29:     end for
```

Algorithm 6.1 Catmull-Clark mesh subdivision (part 2)

```
30:     $edge\text{-}vertices \leftarrow \{\}$
31:     for all edge $e$ do                                    ▷ split each edge
32:         $(e_L, v, e_R) \leftarrow$ splitCell($e$)
33:         $P(v) \leftarrow P'(e)$
34:         $edge\text{-}vertices \leftarrow edge\text{-}vertices \cup \{v\}$
35:     end for
36:     for all face $f$ do                                    ▷ split each face
37:         $\tau \leftarrow$ tuple containing $f$
38:         if $\tau[0] \notin edge\text{-}vertices$ then                ▷ start on an edge-vertex
39:             $\tau$.flip(0)
40:         end if
41:         $n \leftarrow 0$
42:         repeat                                          ▷ find all edge-vertices
43:             $n \leftarrow n + 1$
44:             $v_n \leftarrow \tau[0]$
45:             $\tau$.flip(0, 1, 0, 1)
46:         until $\tau[0] = v_1$
47:         $(f_L, e, f_R) \leftarrow$ splitCell($f, +v_1 - v_2$)                ▷ split face once
48:         $(e_L, v_f, e_R) \leftarrow$ splitCell($e$)          ▷ split edge to create face-vertex
49:         $P(v_f) \leftarrow P'(f)$
50:         for $i \in \{3, \dots, n\}$ do                   ▷ split the face into $n$ quadrilaterals
51:             splitCell(join($v_f, v_i$), $+v_i - v_f$)
52:         end for
53:     end for
54: end procedure
```

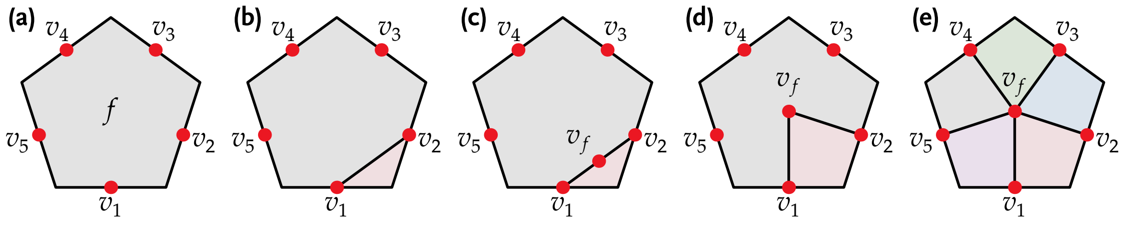


Figure 6.2: Splitting a face in the Catmull-Clark subdivision. (a) A pentagonal face's edges have been subdivided; the new edge-vertices (red) have been labelled v_1, v_2, v_3, v_4, v_5. (b) The face is split along the edge between v_1 and v_2. (c) The new edge is split; the splitting vertex is the face-vertex v_f. (d) The vertex v_f is moved to the center of the face. (e) The face is further split by edges between the remaining edge-vertices and the face-vertex.

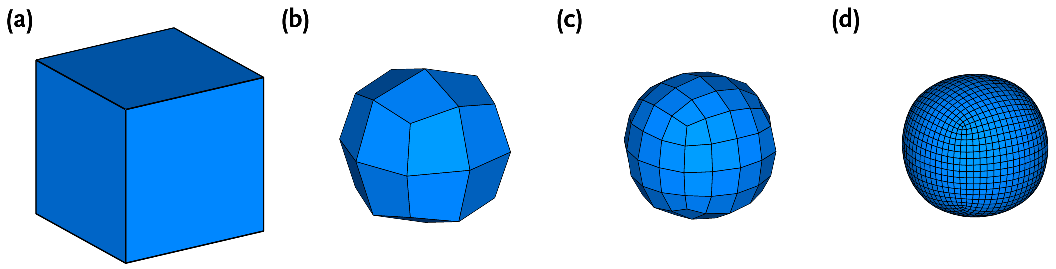


Figure 6.3: A coarse mesh (a) and refinements using a Catmull-Clark subdivision. (b) One subdivision step; (c) two subdivision steps; (d) four subdivision steps.

the same way: finding all of the new vertices of the face (lines 38–46). A tuple is chosen (line 37) and we ensure that its vertex is an edge-vertex (lines 38–40); the vertex is checked against the set *edge-vertices*, to which each edge vertex was added during the edge loop (line 34). In lines 42–46, the edge-vertices $v_1, v_2, \dots, v_n$ are found by continuing the iteration around the face.

Once the new vertices have been found, we can proceed to splitting the face (lines 47–52). This process is illustrated in Figure 6.2. We have to create a vertex interior to the face; simply adding the vertex into the face isn't possible, however, as the resulting structure would no longer be a cell complex. We therefore have to split the face first, creating an edge through it (line 47); this edge is then itself split, and the new vertex v_f is placed in the middle of the face (lines 48–49). Finally, the face is split between the remaining edge-vertices and the face-vertex (lines 50–52).

The results of applying Algorithm 6.1 to an input mesh are shown in Figure 6.3.

6.1.2 Catmull-Clark subdivision in three dimensions

Joy and MacCracken (1996) derive a three-dimensional extension to Catmull-Clark subdivision. On the boundary of the mesh, this subdivision works just like two-dimensional Catmull-Clark; in the interior, however, it introduces *volume*-vertices along with face and edge-vertices. The new positions of interior n-vertices are computed in a similar way as $(n-1)$-vertices in the two-dimensional refinement; for instance, volume-vertices are placed at the average position of the volume's vertices, just like face-vertices in the two-dimensional method.

Algorithm 6.2 Computing the position of edge-vertices in three-dimensional Catmull-Clark subdivision

```
 1: for all edge $e$ do
 2:     $\tau \leftarrow$ tuple containing $e$
 3:     $\boldsymbol{p}_m \leftarrow 1/2(\boldsymbol{P}(\tau[0]) + \boldsymbol{P}(\tau.\mathbf{other}(0))$
 4:     if border($e$) then
 5:         $\boldsymbol{P}'(e) \leftarrow 1/4\left(2\boldsymbol{p}_m + \boldsymbol{P}'(\tau[2]) + \boldsymbol{P}'(\tau.\mathbf{other}(2)\right)$
 6:     else
 7:         $\boldsymbol{P}'(e) \leftarrow \mathbf{0}$
 8:         $n \leftarrow 0$
 9:         $\tau_{\text{start}} \leftarrow \tau$
10:         repeat
11:             $n \leftarrow n + 1$
12:             $\boldsymbol{P}'(e) \leftarrow \boldsymbol{P}'(e) + 2\boldsymbol{P}'(\tau[2]) + \boldsymbol{P}'(\tau[3])$
13:             $\tau.\mathbf{flip}(2, 3)$
14:         until $\tau = \tau_{\text{start}}$
15:         $\boldsymbol{P}(e) \leftarrow \frac{n-3}{n}\boldsymbol{p}_m + \frac{1}{n^2}\boldsymbol{P}'(e)$
16:     end if
17: end for
```

Algorithm 6.2 shows how the positions of edge-vertices are computed in the three-dimensional extension to Catmull-Clark. On line 4, we use the **border** operation to check if the edge *e* is on the boundary of the mesh; if so, then the edge-vertex is computed using Equation 6.1 (line 5). If the edge is not external, then the edge-vertex position is an affine combination of the edge's midpoint and the positions of the incident face and volume-vertices, similar to Equation 6.2 for vertex-vertices in the 2D method. An iteration through the incident faces and volumes (lines 10–14) is used to accumulate the face-vertex and volume-vertex positions (line 12); note that the subdivision requires that face-vertices are weighted twice as much as volume-vertices. The position of the edge-vertex is then computed as the average of these positions and the midpoint of the edge (line 15).

Once all of the new vertex positions have been computed, the 2-cells are subdivided just as in the two-dimensional method: first edges are split by edge-vertices (Figure 6.4a), then face-vertices are created and used to split faces, in the same way as Figure 6.2. The volume splitting is completed by the method described in Algorithm 6.3 (Figures 6.4c–f).

We have to split the volume into eight volumes; this is done by splitting it in half (Figure 6.4c, lines 2–15), then further splitting each half (Figure 6.4e, lines 17–31) and finally each resulting quarter (Figure 6.4f, lines 32–40). Recall that the **splitCell** operation requires the boundary of the new face dividing the two volumes; this is stored in the chain *meridian* (line 9). We create this boundary chain by following a path of edges which connect face-vertices and edge-vertices; lines 3–8 ensure that the iteration starts

Algorithm 6.3 Split hexahedral volumes for the extended Catmull-Clark subdivision

```
1: for all volume $\ell$ do
2:     $\tau \leftarrow$ tuple containing $\ell$
3:     if $\tau[0] \in$ edge-vertices then
4:         $\tau$.flip(0)
5:     end if
6:     if $\tau[0] \in$ vertex-vertices then
7:         $\tau$.flip(0, 1, 0)
8:     end if
9:     $meridian \leftarrow 0$
10:    $\tau_{\text{start}} \leftarrow \tau$
11:    repeat
12:        $meridian \leftarrow meridian + \rho(\tau[0], \tau[1])\ \tau[1]$
13:        $\tau$.flip(0, 1, 2, 1)
14:    until $\tau = \tau_{\text{start}}$
15:    $(\ell_1, f, \ell_2) \leftarrow$ splitCell($\ell$, $meridian$)
16:    split face $f$ as in Algorithm 6.1 to create volume-vertex $v_\ell$
17:    for $j \in \{1, 2\}$ do
18:        $\tau \leftarrow$ tuple containing $\{v_\ell, \ell_j\}$
19:        $meridian \leftarrow 0$
20:        $n \leftarrow 0$
21:        repeat
22:            if $n = 3$ then
23:                $v_j \leftarrow \tau[0]$
24:            end if
25:            $meridian \leftarrow meridian + \rho(\tau[0], \tau[1])\ \tau[1]$
26:            $\tau$.flip(0, 1, 2, 1)
27:            $n \leftarrow n + 1$
28:        until $\tau[0] = v_\ell$
29:        $(\ell_{j0}, f_j, \ell_{j1}) \leftarrow$ splitCell($\ell_j$, $meridian$)
30:        $(f_L, e_j, f_R) \leftarrow$ splitCell($f_j$, $+v_j - v_\ell$)
31:    end for
32:    for $(j, k) \in \{(0,0), (0,1), (1,0), (1,1)\}$ do
33:        $\tau \leftarrow$ tuple containing $\{v_\ell, e_j, \ell_{jk}\}$
34:        $meridian \leftarrow 0$
35:        repeat
36:            $meridian \leftarrow meridian + \rho(\tau[0], \tau[1])\ \tau[1]$
37:            $\tau$.flip(0, 1, 2, 1)
38:        until $\tau[0] = v_\ell$
39:        $(\ell_{jkL}, f_{jk}, \ell_{jkR}) \leftarrow$ splitCell($\ell_{jk}$, $meridian$)
40:    end for
41: end for
```

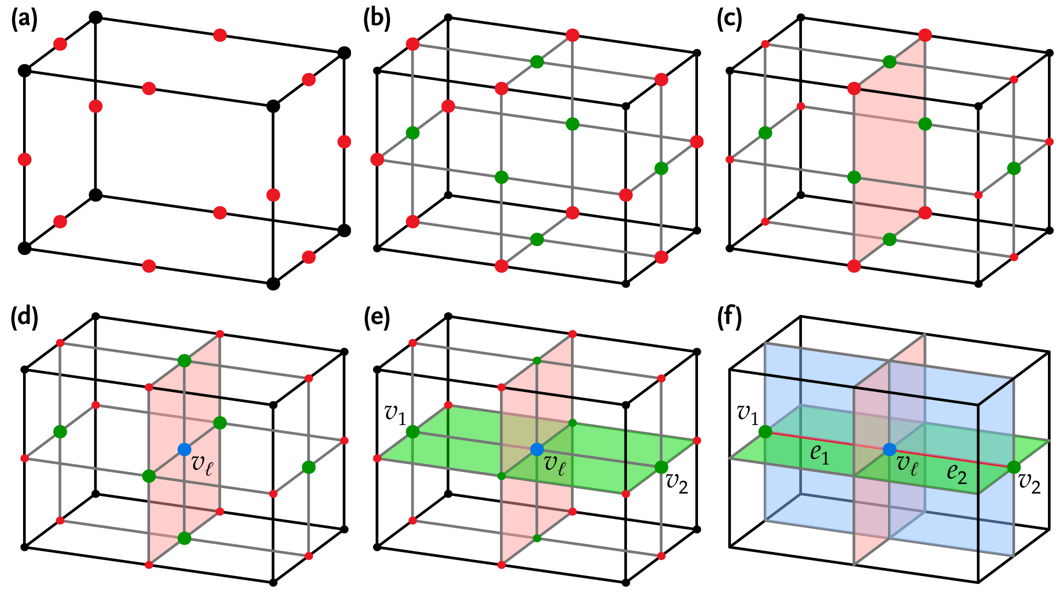


Figure 6.4: A hexahedral volume element in a mesh is split into eight subvolumes. (a) Edges are split by edge-vertices (red). (b) Faces are split in four at face-vertices (green). (c) One meridian of edges becomes the boundary of the first split face (light red). (d) The first split face is itself divided in four at a volume-vertex v_ℓ (blue). (e) Meridians in left and right child volumes are used as the boundaries for faces (light green) splitting their respective volumes. During this traversal, the remaining two face-vertices v_1 and v_2 are identified. (f) Starting from the edge between the volume-vertex and the face-vertex (v_1 or v_2) in each of the four volumes, meridians are traced and used as boundaries for the final splitting planes (light blue).

on a tuple with a face-vertex, then lines 11–14 perform the iteration; the tuple is advanced at each step by the operation **flip**$(0,1,2,1)$ (line 13). In line 12 the edge is added to the boundary chain. We require all of the orientations of the edges to be consistent; to this end, the orientation of the edge $\tau[1]$ added to the boundary chain is its relative orientation with respect to the vertex $\tau[0]$. The orientation of each edge in *meridian* is thus the direction of the iteration.

The *meridian* chain is then used by **splitCell** as the boundary of the face f which splits the volume ℓ into ℓ_1 and ℓ_2 (line 15). We use the same method shown in Figure 6.2 and described in Algorithm 6.1 to create a vertex on this face and split it in four. This vertex is the volume-vertex v_ℓ (Figure 6.4d).

We now split each of the volumes ℓ_1 and ℓ_2. (lines 17–31). For each volume, we find a boundary chain *meridian* in exactly the same way as for the entire volume ℓ, starting from a tuple on the volume-vertex v_ℓ. The only addition is that we must also record the vertex v_j (line 23); this is the face vertex incident with ℓ_j we have not encountered yet, and is always the third vertex encountered (Figure 6.4e). After splitting ℓ_j with face f_j (line 29), we further split the face by an edge e_j between the volume-vertex v_ℓ and the face-vertex v_j which we recorded earlier (line 30).

The final splits (lines 32–40) start from a tuple including the edge e_j (line 33) then iterate as before to create the boundary chain *meridian* (lines 34–38) which is used to split the quarter-volume ℓ_{jk} (line 39).

The results of applying extended Catmull-Clark subdivision are shown in Figure 6.5.

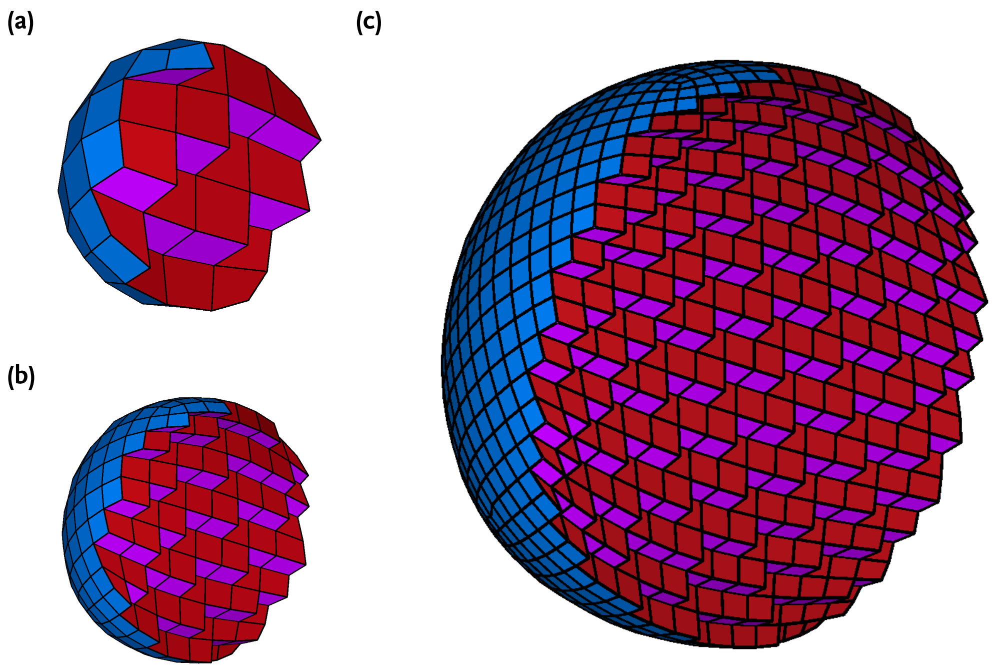


Figure 6.5: Section of a a 3D mesh after (a) two steps of refinement with the extended Catmull-Clark method; (b) three steps of refinement; (c) four steps of refinement. The internal structure is revealed by drawing only those 3-cells to one side of a plane (not visualized). External faces are coloured blue, while the colour of internal faces depends on their orientation.

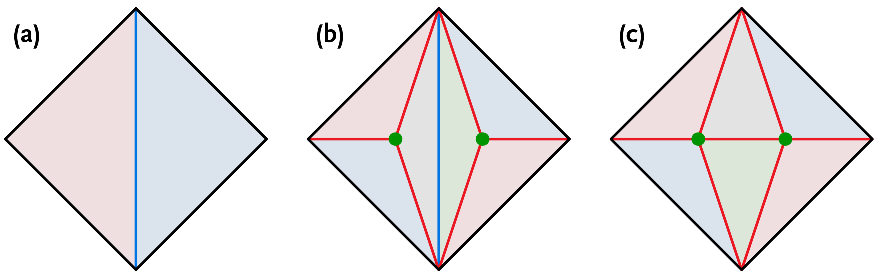


Figure 6.6: The two stages of the $\sqrt{3}$ subdivision scheme. (a) The initial mesh of two triangles. The blue edge is internal, while the black edges are external. (b) A new vertex (green) is placed at the center of each triangle, and new edges (red) connecting it to the triangle's vertices split the triangle into three. (c) The internal edge is "flipped"; it is removed and the resulting quadrilateral is split by a new edge between the green vertices.

The initial mesh is a cube, and the boundary of the shape is the same as in Figure 6.3; the internal structure is divided into roughly uniform cubelike cells.

6.1.3 Kobbelt's $\sqrt{3}$ subdivision in two dimensions

Another two-dimensional subdivision scheme is Kobbelt's $\sqrt{3}$ method (Kobbelt 2000). Not only does it require moving existing vertices and adding vertices in the interior of faces, but also removing edges and adding new ones (Figure 6.6). The subdivision is a two-step process. In the first step, each triangle is subdivided into three, which share a new vertex at the middle of the triangle. In the second step, the old edges between triangles are "flipped" so they run between the new midpoint vertices.

Algorithm 6.4 implements one step of the $\sqrt{3}$ subdivision. It consists of three loops. In the first loop (lines 2–10), the updated positions of the preexisting vertices are found. The new position of a vertex v is a linear combination of its old position and the average position of its neighbouring vertices (line 9). Line 3 retrieves the neighbours of v, and the sum of their positions is computed in the loop in lines 5–7. The exact interpolant α is a function of the number of neighbours, $||nbs||$; it is computed in line 8.

The second loop (lines 12–22) splits all the triangles into three. Like the Catmull-Clark subdivision, one endpoint of all three of the edges is in the middle of the face; we therefore use a similar method to subdivide the triangle (Figure 6.7). First, the vertices of the triangle f are found by creating a tuple and finding its vertex, the **other** vertex on the same edge, and the **other** vertex after flipping the edge (lines 13–17). Then f is split by an edge *with the same endpoints as an existing edge* (line 18). This operation is topologically valid, and the structure is still a cell complex. The new edge is split, and the resulting vertex is moved to the center of the triangle (lines 19–20). Finally, the quadrilateral remainder of the original triangle is split into two triangles (line 21).

The third loop (lines 26–38) flips the preexisting edges, which were saved in line 11. Only edges between two triangles can be flipped; external edges are excluded by the test in line 27. For each remaining edge, its endpoints are retrieved with the **boundary** operation (line 30), then the edge itself is deleted by the **mergeCell** operation (line 31). The new quadrilateral face is split by an edge through the other two vertices (line 36); these vertices are found by creating a new tuple at v_1 and finding its **other** cells (lines 32–

Algorithm 6.4 $\sqrt{3}$ subdivision

```
Require: Each vertex $v$ has a position $\boldsymbol{P}(v)$
1: procedure Root3Subdivision
2:     for all vertex $v$ do                              ▷ Set new positions of old vertices
3:         $nbs \leftarrow$ neighbours($v$)
4:         $\boldsymbol{W} \leftarrow 0$
5:         for all $w \in nbs$ do
6:             $\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{P}(w)$
7:         end for
8:         $\alpha \leftarrow \left(4 - 2\cos\frac{2\pi}{||nbs||}\right) / 9$
9:         $\boldsymbol{P}'(v) \leftarrow (1-\alpha)\,\boldsymbol{P}(v) + \alpha\frac{\boldsymbol{W}}{||nbs||}$
10:    end for
11:    $oldEdges \leftarrow$ cells of dimension 1
12:    for all face $f$ do                                 ▷ Split all triangles into three
13:        $\tau \leftarrow$ tuple containing $f$
14:        $v_1 \leftarrow \tau[0]$
15:        $v_2 \leftarrow \tau$.other(0)
16:        $\tau$.flip(1)
17:        $v_3 \leftarrow \tau$.other(0)
18:        $(f_L, e, f_R) \leftarrow$ splitCell($f, +v_1 \; - v_2$)
19:        $(e_L, v_m, e_R) \leftarrow$ splitCell($e$)
20:        $\boldsymbol{P}'(v_m) \leftarrow 1/3\,(\boldsymbol{P}(v_1) + \boldsymbol{P}(v_2) + \boldsymbol{P}(v_3))$
21:        $(f_L, e, f_R) \leftarrow$ splitCell(join($v_3, v_m$), $-v_m \; + v_3$)
22:    end for
23:    for all vertex $v$ do
24:        $\boldsymbol{P}(v) \leftarrow \boldsymbol{P}'(v)$
25:    end for
26:    for all $e \in oldEdges$ do                         ▷ Flip internal edges
27:        if border($e$) then
28:            skip this edge
29:        else
30:            $\{+v_1, -v_2\} \leftarrow$ boundary($e$)
31:            $f \leftarrow$ mergeCell($e$)
32:            $\tau \leftarrow$ tuple containing $\{v_1, f\}$
33:            $v_3 \leftarrow \tau$.other(0)
34:            $\tau$.flip(1)
35:            $v_4 \leftarrow \tau$.other(0)
36:            $(f_L, e', f_R) \leftarrow$ splitCell($f, -v_3 \; + v_4$)
37:        end if
38:    end for
39: end procedure
```

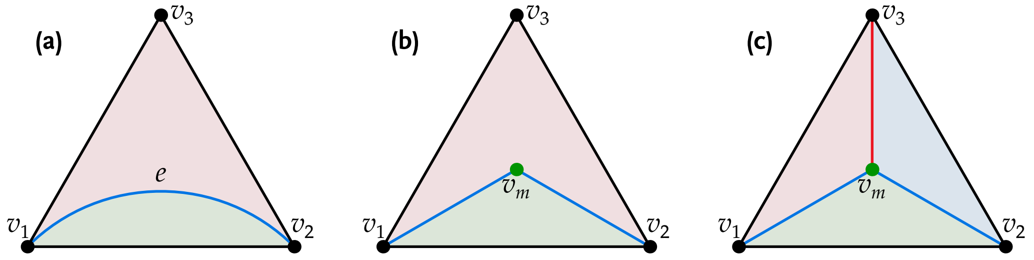


Figure 6.7: Splitting a single triangular face from its center is a three-step process. (a) The triangle is split by a new edge (visualized as a blue arc) parallel to an existing edge. (b) The new edge is split and the new vertex (green) is moved to the center of the triangle. (c) The quadrilateral face is split into two triangles.

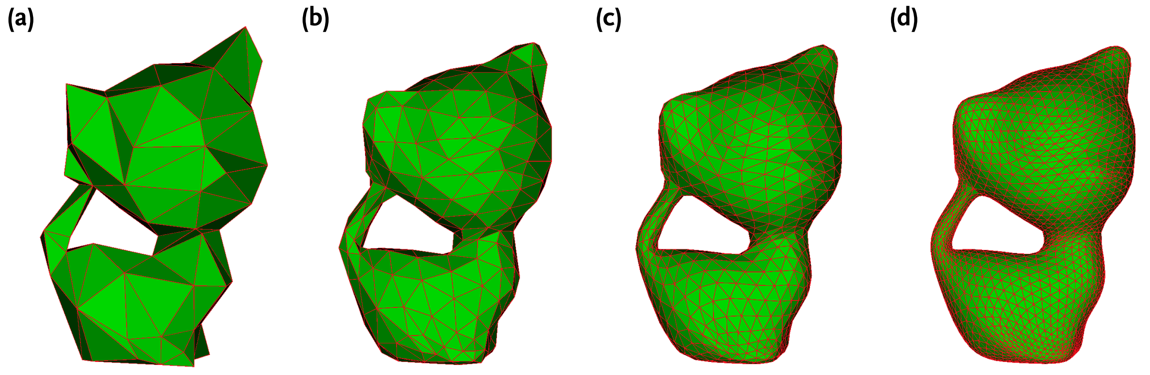


Figure 6.8: Successive applications of the $\sqrt{3}$ subdivision algorithm to an input mesh. (a) Original mesh; (b) after one refinement step; (c) after two steps; (d) after three steps.

35).

The results of successively applying the $\sqrt{3}$ subdivision to an input mesh can be seen in Figure 6.8.

6.1.4 Kobbelt's $\sqrt{3}$ subdivision in three dimensions

A three-dimensional extension to the $\sqrt{3}$ algorithm, applicable to tetrahedral meshes, is described by Burkhart, Hamann, and Umlauf (2010). Tetrahedra are divided by a method analogous to triangles in the $\sqrt{3}$ method. First, a new vertex is inserted in the interior of each tetrahedron and used to divide each tetrahedron into four (Figure 6.9). Next, the pre-existing internal faces are removed and replaced by edges between the new vertices; this turns each pair of adjoining tetrahedra into three tetrahedra (Figure 6.10). The original $\sqrt{3}$ method is then applied to triangles on the surface of the complex. The division of each surface triangle into three is carried to its incident tetrahedron, dividing it as well (Figure 6.11). Finally, the edge-flip applied to surface edges must alter the incident tetrahedra (Figure 6.12).

Splitting a tetrahedron in four is performed by a method analogous to the method used to split a triangle in three. The tetrahedron is split by a new face parallel to an existing face, which is itself split in three, creating one tetrahedron and one volume with

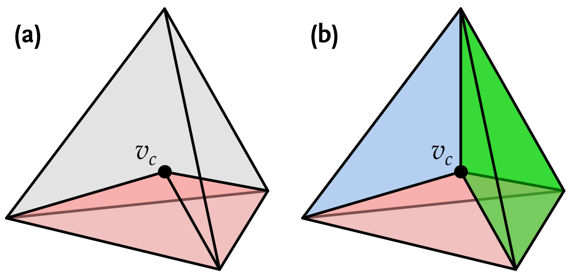


Figure 6.9: Splitting a tetrahedron into four tetrahedra which meet at a vertex v_c. (a) The tetrahedron is split once, into a new tetrahedron (red) and a six-sided volume (grey). (b) The six-sided volume is split into three tetrahedra (the front tetrahedron is not rendered).

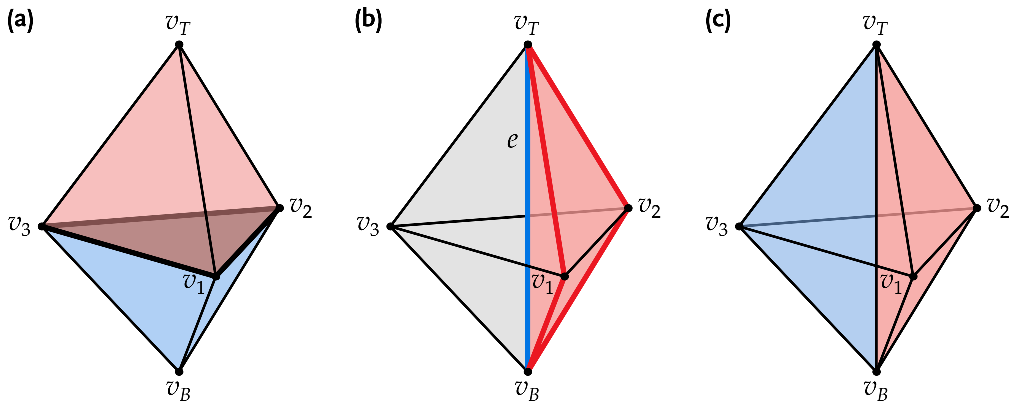


Figure 6.10: Turning an internal face separating two tetrahedra into an edge incident with three tetrahedra. (a) Two tetrahedra (red and blue) are separated by the shaded face. The vertices $\{v_1, v_2, v_3\}$ incident with the face are identified, as are the other vertices involved: v_T on the red tetrahedron and v_B on the blue tetrahedron. (b) The face has been removed, resulting in a volume with six triangular faces, which is then split by the nonplanar face whose boundary is the cycle through the vertices $[v_T, v_1, v_B, v_2]$ (red). This face is then itself split by e (blue), between v_T and v_B, into two triangular faces. (c) Two more tetrahedra sharing e are created: one with vertices $\{v_T, v_2, v_B, v_1\}$ (blue) and one with vertices $\{v_T, v_1, v_B, v_3\}$ (not rendered).

six triangular faces (Figure 6.9a). The six-sided shape is then split into the other three tetrahedra (Figure 6.9b).

Algorithm 6.5 performs the next step, flipping internal faces into edges (Figure 6.10). First, a tuple iteration is used to find the face's vertices $\{v_1, v_2, v_3\}$ (lines 2–8). We also use flip paths involving 2-flips and 3-flips to identify the other two vertices in the two tetrahedra, v_T and v_B (lines 9 and 10). We use **mergeCell** to remove the face f (line 11) then split the combined volume by a single face along four of its edges. These edges are those which form a circuit connecting the vertices $[v_T, v_1, v_B, v_2]$, and are found using **join** (lines 12–15); each is multiplied by the relative orientation with respect to its starting point to ensure that the orientations are consistent, then they are combined into the boundary chain *faceBound* (line 16), which is used to split the volume (line 17). The new quadrilateral face is itself split by an edge e between v_T and v_B (line 18). Finally, the new edge is used to split the larger volume into two more tetrahedra (lines 14–21).

The third step of the extended $\sqrt{3}$ subdivision method splits the triangular faces on the boundary of the manifold into three; this is done as in the two-dimensional method. Once the face has been split, its incident tetrahedron is split along the same edges, producing three tetrahedra (Figure 6.11).

Algorithm 6.5 Flip internal faces into edges in the extended $\sqrt{3}$ subdivision, turning two adjacent tetrahedra into three

```
procedure FaceEdgeFlip($f$)
    $\tau \leftarrow$ tuple containing $f$
    $n \leftarrow 0$
    repeat
        $n \leftarrow n + 1$
        $v_n \leftarrow \tau[0]$
        $\tau$.flip$(0, 1)$
    until $\tau[0] = v_1$
    $v_T \leftarrow \tau$.flip$(2, 1)$.other$(0)$
    $v_B \leftarrow \tau$.flip$(3, 2, 1)$.other$(0)$
    $\ell \leftarrow$ mergeCell$(f)$
    $e_{T1} \leftarrow$ join$(v_T, v_1)$
    $e_{1B} \leftarrow$ join$(v_1, v_B)$
    $e_{B2} \leftarrow$ join$(v_B, v_2)$
    $e_{2T} \leftarrow$ join$(v_2, v_T)$
    $faceBound \leftarrow \rho(v_T, e_{T1})\, e_{T1} + \rho(v_1, e_{1B})\, e_{1B} + \rho(v_B, e_{B2})\, e_{B2} + \rho(v_2, e_{2T})\, e_{2T}$
    $(\ell_L, f', \ell_R) \leftarrow$ splitCell$(\ell, faceBound)$
    $(f_L, e, f_R) \leftarrow$ splitCell$(f', +v_T \; - v_B)$
    $e_{B3} \leftarrow$ join$(v_B, v_3)$
    $e_{3T} \leftarrow$ join$(v_3, v_T)$
    splitCell(join$(e, v_3)$, $+e + \rho(v_B, e_{B3})\, e_{B3} + \rho(v_3, e_{3T})\, e_{3T}$)
end procedure
```

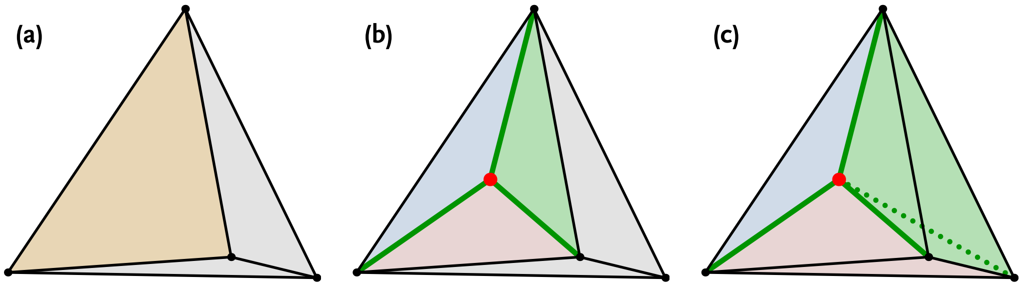


Figure 6.11: Splitting a tetrahedron on the border of the cell complex. (a) Before splitting: the brown face is on the border of the cell complex. (b) The face has been split into three triangles, which all meet at the face-vertex (red); the volume now has six triangular faces. (c) The volume has been split into three tetrahedra, each with one border face, and all meeting at the edge (dotted) which joins the face-vertex to the internal vertex of the original tetrahedron.

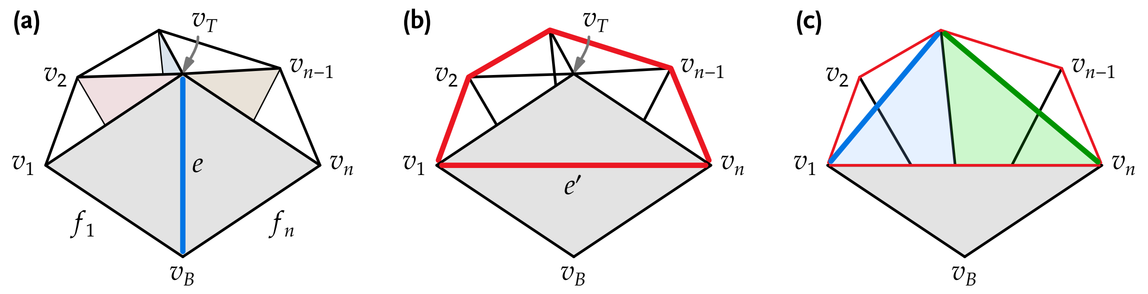


Figure 6.12: Performing an edge flip on an external edge in a 3D complex requires altering all incident tetrahedra. (a) Before flipping, the edge e (blue) is incident to five faces: two (f_1 and f_n) are on the border of the complex, while the other three are internal. (b) All of the internal faces incident with e are deleted and e itself is flipped into e', leaving a volume with ten triangular faces. The equator (red) divides the cell into two pyramids. (c) The volume is split by an equatorial face, which is itself subdivided into triangles. The new edges (blue and green), together with v_B, define new faces which split the bottom pyramid into three tetrahedra. The top pyramid (not shown) is divided in the same way.

The final step is to flip edges on the boundary. This is complicated in a 3D complex because such an edge may be incident to many faces (Figure 6.12a). We have to remove all of the internal faces, producing a diamond-shaped volume, then split this volume into two pyramidal cells (Figure 6.12b). The shared face is then triangulated, and each of the new edges used to split the upper and lower pyramids (Figure 6.12c). Algorithm 6.6 performs this flip operation.

We first collect the important cells incident to the edge e. The top and bottom vertices of the diamond-shaped volume, v_T and v_B, are the endpoints of e (line 2). We perform an iteration around e (lines 5–10) to find all of the incident faces f_i (line 7) and the third vertex on each face, v_i; the flip operation in line 8 accesses this vertex. Because we start and end on a tuple containing the external volume $\circledast$, the faces f_1 and f_n are on the boundary of the complex, and the other faces are on the inside.

We first delete these internal faces (lines 11–13) then perform the 2D edge-flip operation (lines 14–15). Next we split the resulting diamond-shaped cell equatorially (lines 16–21). The boundary of the face is the boundary chain *equator*, which contains the flipped edges $e_\perp$ as well as the edges connecting each v_i to v_{i+1} (line 18); once again, each edge is multiplied by its relative orientation with respect to its start vertex (line 19) in order to ensure the proper orientation of the boundary.

Once split, the equatorial face is triangulated (line 22). Following Burkhart, Hamann, and Umlauf, the procedure TRIANGULATEPOLYGON uses the algorithm of Klincsek (1980) to triangulate the face in a way that the sum of the length of the edges is minimized. The procedure returns a list of the new edges inserted in the face $f_\perp$. The final loop (lines 23–31) introduces faces splitting each of the upper and lower pyramids along each edge. For each new edge e' the boundary chain *faceBound* is created containing the edge e' and the edges e_a and e_b which connect its endpoints v_a and v_b with the tip of the pyramid, either v_T or v_B (line 28). Finally, the pyramid is split by this face (line 29).

The results of applying the extended $\sqrt{3}$ subdivision to an input mesh are shown in Figure 6.13.

Algorithm 6.6 Applying a $\sqrt{3}$ edge-flip operation to an edge and all of its incident tetrahedra

```
procedure ExternalEdgeFlip($e$)
    $\{+v_T, -v_B\} \leftarrow$ boundary($e$)
    $\tau \leftarrow$ tuple containing $\{e, \otimes\}$
    $n \leftarrow 0$
    repeat
        $n \leftarrow n + 1$
        $f_n \leftarrow \tau[3]$
        $v_n \leftarrow \tau$.flip(1).other(0)
        $\tau$.flip(3, 2)
    until $\tau[3] = \otimes$
    for all $f \in \{f_2, \dots, f_{n-1}\}$ do                ▷ delete all internal faces
        mergeCell($f$)
    end for
    $f_{EXT} \leftarrow$ mergeCell($e$)                        ▷ flip edge
    $(f_T, e_\perp, f_B) \leftarrow$ splitCell($f_{EXT}, +v_n - v_1$)
    $equator \leftarrow +e_\perp$                              ▷ split volume into two pyramids
    for all $i \in \{1, \dots, n-1\}$ do
        $e' \leftarrow$ join($v_i, v_{i+1}$)
        $equator \leftarrow equator + \rho(v_i, e')\, e'$
    end for
    $(\ell_T, f_\perp, \ell_B) \leftarrow$ splitCell(join($v_T, v_B$), $equator$)
    $newEdges \leftarrow$ TriangulatePolygon($f_\perp$)
    for all $e' \in newEdges$ do                               ▷ split pyramids into tetrahedra
        $\{+v_a, -v_b\} \leftarrow$ boundary($e'$)
        for $v \in \{v_T, v_B\}$ do
            $e_a \leftarrow$ join($v_a, v$)
            $e_b \leftarrow$ join($v_b, v$)
            $faceBound \leftarrow +e' + \rho(v_b, e_b)\, e_b + \rho(v, v_a)\, e_a$
            $(\ell_L, f', \ell_R) \leftarrow$ splitCell(join($e', v$), $faceBound$)
        end for
    end for
end procedure
```

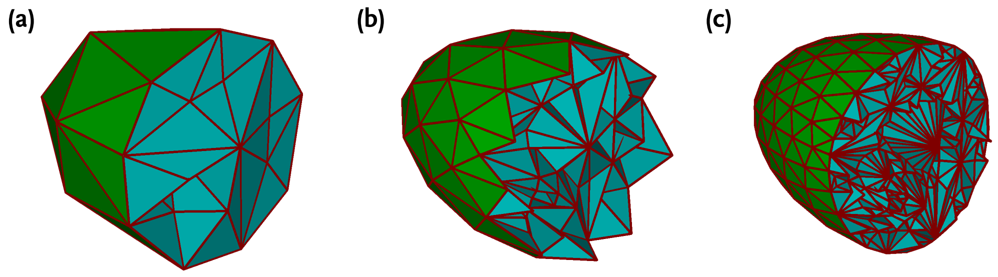


Figure 6.13: Successive applications of the extended three-dimensional $\sqrt{3}$ subdivision algorithm to an input mesh. (a) Mesh after two refinement steps; (b) after three steps; (c) after four steps.

6.2 Polygonizing implicit surfaces

Another important problem in geometric modeling is polygonizing implicit surfaces: creating a mesh which approximates a mathematically-defined surface of the form $F(x) = c$. The method I will implement in this section is the shrink-wrap algorithm (van Overveld and Wyvill 2004). This algorithm finds a triangle mesh which does not lie more than a distance ϵ from the mathematical surface. Other polygonization methods, such as the marching cubes algorithm (Lorensen and Cline 1987), produce polygons of approximately uniform size. The shrink-wrap algorithm, on the other hand, produces triangles whose sizes match the surface: large triangles where the curvature is low and the surface does not change rapidly, and smaller triangles where the curvature is higher and the surface changes rapidly. The triangles are produced by subdividing an initial coarse mesh. Unlike the global subdivision methods discussed in Section 6.1, acceptable triangles are *not* subdivided; the shrink-wrap algorithm only divides triangles where necessary to match the surface. It is thus an example of an *adaptive* subdivision.

The shrink-wrap algorithm starts with a coarse mesh far outside the surface, with vertices on a higher isosurface $F(x) = \theta_0$. The mesh is then shrunk towards the desired isosurface, and each edge is checked for two conditions. The first condition ensures that the distance between the edge and the surface is not too high. The second condition ensures that the mesh captures even small regions of high curvature. If an edge fails either of these conditions, it is deemed *unacceptable* and is split in two. Faces whose edges are split are themselves subdivided so that all faces are triangular. The loop then repeats, with the mesh shrunk and subdivided further, until all of the points are close enough to the desired isosurface.

Algorithm 6.7 performs one step of this algorithm. It takes as argument the isovalue θ that the mesh is to be shrunk to, and moves all vertices along the gradient of the field ∇F so that they are on the isosurface $F(x) = \theta$ (lines 2–4). Each edge is then checked to see if it is acceptable (lines 5–14). An unacceptable edge will be split (line 8). The new vertex is moved to the point SplineMidpoint (line 9), the midpoint of the spline fitted to the position of the endpoints and the gradient of the field at those points. This point is chosen by van Overveld and Wyvill to improve the convergence of the polygonization.

If an edge is split, then the splitting vertex is added to a list associated with each incident face (lines 10–12). This list is checked to see if each face must be split (lines 15–

Algorithm 6.7 Perform one step of the shrink-wrap polygonization method

```
Input: The new isovalue the mesh will interpolate
 1: procedure ShrinkWrap($\theta$)
 2:     for all vertex $v$ do
 3:         move $v$ along $\nabla F$ to the point where $F(\boldsymbol{P}(v)) = \theta$
 4:     end for
 5:     for all edge $e$ do
 6:         if EdgeUnacceptable($e$) then
 7:             $\{+v_L, -v_R\} \leftarrow$ boundary($e$)
 8:             $(e_L, v, e_R) \leftarrow$ splitCell($e$)
 9:             $\boldsymbol{P}(v) \leftarrow$ SplineMidpoint($v_L, v_R, \theta$)
10:             $\{+f_L, -f_R\} \leftarrow$ coboundary($e$)
11:             $splitVs[f_L] \leftarrow splitVs[f_L] \cup \{v\}$
12:             $splitVs[f_R] \leftarrow splitVs[f_R] \cup \{v\}$
13:         end if
14:     end for
15:     for all face $f$ do
16:         if $\|splitVs[f]\| = 1$ then
17:             $\{v\} \leftarrow splitVs[f]$
18:             $\tau \leftarrow$ tuple containing $\{f, v\}$
19:             $\tau$.flip$(0, 1, 0)$
20:             splitCell($f, +v\ - \tau[0]$)
21:         else if $\|splitVs[f]\| = 2$ then
22:             $\{v_1, v_2\} \leftarrow splitVs[f]$
23:             $\tau \leftarrow$ tuple containing $\{f, v_1\}$
24:             $\tau$.flip$(0, 1)$
25:             if $\tau$.other$(0) = v_2$ then
26:                 $\tau$.flip$(1, 0, 1, 0, 1)$
27:             end if
28:             $v'_1 \leftarrow \tau[0]$
29:             $v'_2 \leftarrow \tau$.other$(0)$
30:             $(f_L, e, f_R) \leftarrow$ splitCell($f, +v_1\ - v_2$)
31:             if $\|\boldsymbol{P}(v_1) - \boldsymbol{P}(v'_2)\| < \|\boldsymbol{P}(v_2) - \boldsymbol{P}(v'_1)\|$ then
32:                 splitCell(join($v_1, v'_2$), $+v_1\ - v'_2$)
33:             else
34:                 splitCell(join($v_2, v'_1$), $+v_2\ - v'_1$)
35:             end if
36:         else if $\|splitVs[f]\| = 3$ then
37:             $\{v_1, v_2, v_3\} \leftarrow splitVs[f]$
38:             splitCell(join($v_1, v_2$), $+v_1\ - v_2$)
39:             splitCell(join($v_2, v_3$), $+v_2\ - v_3$)
40:             splitCell(join($v_3, v_1$), $+v_3\ - v_1$)
41:         end if
42:     end for
43: end procedure
```

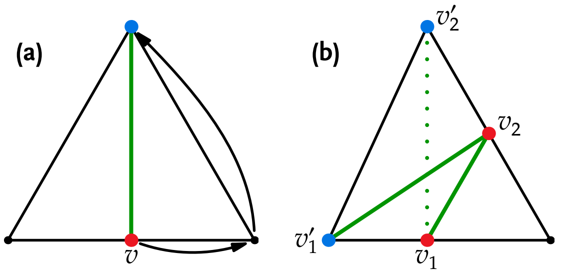


Figure 6.14: Splitting faces in the shrink-wrap polygonization algorithm. (a) If one edge has been split by a vertex v, the face is split across to the opposite vertex, which is found by iterating two vertices from v. (b) If two edges have been split by vertices v_1 and v_2, the face is split between v_1 and v_2, and between the shorter of v_1v_2' and v_2v_1'.

42). The face will be subdivided differently depending on how many of its edges were split. If one edge was split (lines 16–20), the face is split from the new vertex to the opposite vertex; this vertex is found by rotating a tuple two vertices around the triangle (Figure 6.14a). If two edges were split (lines 21–35), the triangle will be split twice: once along an edge between the two new vertices (line 30), and once between a new vertex and its opposite vertex (lines 32 or 34). Of the two possibilities, the shortest splitting edge is used (line 31). The opposite vertices are found through tuple rotation in lines 23–29; the tuple is rotated by one vertex starting from v_1 (line 24) then checked to see if it has been rotated towards v_2 or away; if the former, it is rotated in the opposite direction (line 26) so it ends in the correct place to find v_1' and v_2'. Finally, if all three edges of a triangle have been split, the face is split between each pair of new vertices (lines 36–41).

Figure 6.15 shows a use of the shrink-wrap algorithm to find a triangle mesh approximating an isosurface of the form

$$F(\boldsymbol{p}) = f(\|\boldsymbol{p} - \boldsymbol{p}_0\|) + f(\|\boldsymbol{p} - \boldsymbol{p}_1\|) = 0.5,$$

where $f(r) = {}^1\!/\!_9\,(1 - 22r^2 + 17r^4 - 4r^6)$ is Wyvill's *soft object function* (G. Wyvill, McPheeters, and B. Wyvill 1986). This isosurface has the form of two balls which join smoothly around the axis joining their centers. The initial surface is an icosahedron (Figure 6.15a) whose vertices are moved along the gradient ∇F to the isosurface $F(\boldsymbol{p}) = 0.2$ (Figure 6.15b). After a few applications of Algorithm 6.7, the triangles on the outside of the balls are much larger than those on the join section, where the surface changes more rapidly (Figures 6.15c–e). The larger polygons still approximate the surface well, as shown by the smooth rendering (Figure 6.15f) in which each vertex is rendered with a normal defined by the gradient to the field at that point.

6.3 Propagating geodesics

The next model is original to this thesis and illustrates the use of *tuple coordinates*, which were introduced in Section 5.3. A tuple $\tau = (v, e, f, \ldots)$ defines a coordinate frame: the origin is the position of v, the x-axis is the direction of e, the xy-plane is the same as the plane of f, and so on. Particular choices for the axes lead to different coordinate frames: the *tuple frame* chooses each axis to lie along an edge incident to v, while the *orthogonal tuple frame* instead has the axes orthogonal and pointing inward (Figure 6.16).

The tuple frame provides an intrinsic coordinate system for a cell, and the totality of these coordinate frames defines an atlas for the entire subdivided manifold. Each flip

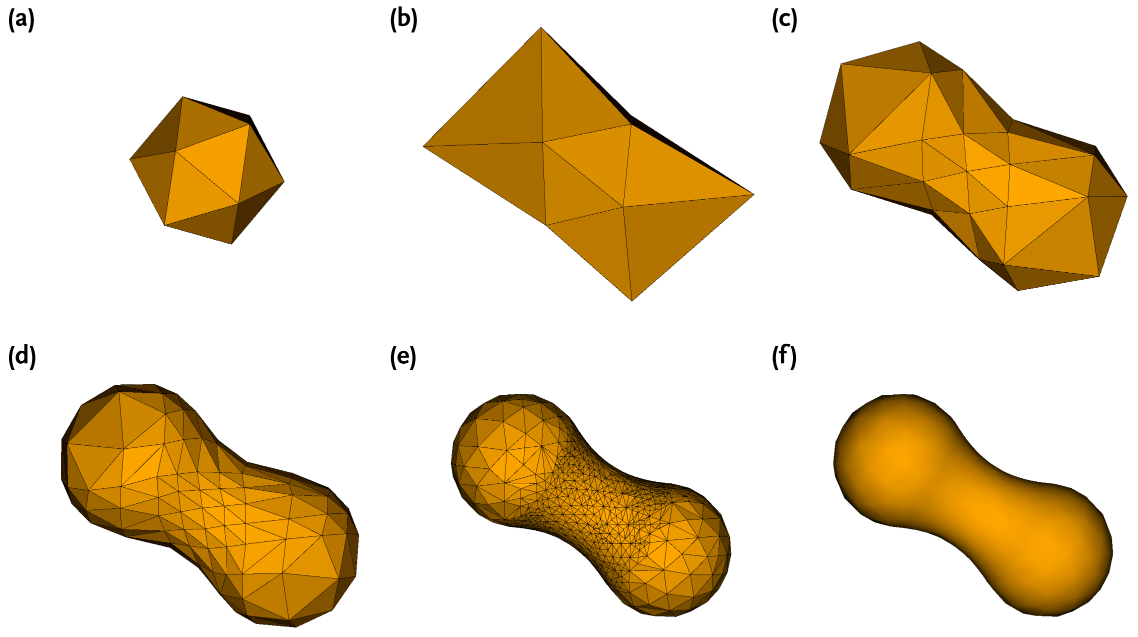


Figure 6.15: A run of the shrink-wrap algorithm to polygonize an implicit surface $F(x) = 0.5$. (a) The algorithm starts with an icosahedron; (b) the vertices are moved to the isosurface $F(x) = 0.2$. At each step of the algorithm, the vertices are moved to a lower isosurface and unacceptable edges are split. (c) After one step (isovalue 0.3); (d) after two steps (isovalue 0.36); (e) after eight steps (isovalue 0.48). (f) The mesh shown in (e), rendered with Gouraud shading.

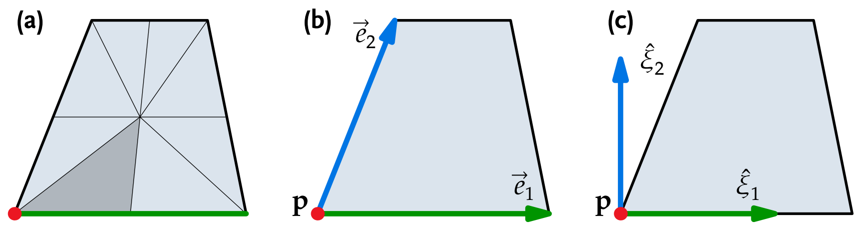


Figure 6.16: The tuple frame of reference. (a) The shaded tuple has the red vertex, the green edge, and the blue face. (b) The tuple frame has position $\vec{p}$, first coordinate axis $\vec{e}_1$ along the green edge, and second coordinate axis $\vec{e}_2$ along the other edge shared by the vertex. (c) The orthogonal tuple frame has the same position, and the first coordinate axis is the unit-length version of $\vec{e}_1$, while the second coordinate axis is orthogonal to the first within the plane of the face.

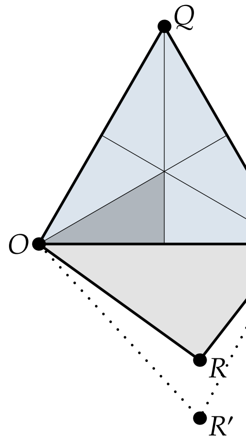


Figure 6.17: Naming the vertices for the derivation of the tuple coordinate transition functions. The cell tuple $\tau = (O, \overline{OP}, \triangle OPQ)$ is shaded. R' is the point R rotated about $\overline{OP}$ into the plane of $\triangle OPQ$.

operation σ_i takes a tuple to an adjacent tuple, and so defines a coordinate system transformation between the respective tuple frames. Using these coordinate transformations, we can propagate geometric information across the manifold in a way that does not depend on its particular embedding. The model described in this section propagates a *geodesic*: a curve that is locally straight. On the surface of a mesh, this means that a geodesic is a piecewise straight line, running straight across faces and turning only on edges or vertices.

This propagation depends on knowing the coordinate transition functions corresponding to a flip in each dimension; these are derived in Section 6.3.1. The algorithm itself is then described in Section 6.3.2.

6.3.1 Tuple coordinate transition functions

Consider the cell complex in a default embedding (Section 3.2.1); that is, an embedding in which all cells are linear subspaces. As the transitions depend only on intrinsic information (such as the sizes of cells and the angles between them), they will be invariant in any isometric embedding. Since all tuple frame axes are edges, and thus straight in a default embedding, they must be related to the canonical basis of the embedding space by an affine transformation. In turn, this means that the transition functions must also be affine transformations. Thus, any of the transition functions consists of a *linear* transformation of the coordinate directions followed by a *translation* of the origin:

$$\Phi : \boldsymbol{p} \to T\boldsymbol{p} + \boldsymbol{o}.$$

(Vectors are independent of origin, so $\Phi : \vec{v} \to T\vec{v}$.)

We are particularly interested in triangular meshes. In this case, we label the three vertices of the triangle O, P, and Q, where $\tau = (O, \overline{OP}, \triangle OPQ)$; the adjoining triangle τ.**other**(2) is $\triangle OPR$ (Figure 6.17).

Transformation under σ_0 We will find the particular transformation by reducing a point in tuple coordinates to barycentric coordinates (Section 5.3). Recall that a point in a simplex can be uniquely represented as an affine combination of the vertices; for our example triangle $\triangle OPQ$, this means that a point can be defined as

$$\boldsymbol{p} = \alpha_O \boldsymbol{O} + \alpha_P \boldsymbol{P} + \alpha_Q \boldsymbol{Q},$$

where $\alpha_O + \alpha_P + \alpha_Q = 1$. A point with tuple frame coordinates (u, v) is at

$$\begin{aligned} \boldsymbol{p} &= \boldsymbol{O} + u\,\vec{e}_1 + v\,\vec{e}_2 \\ &= \boldsymbol{O} + u(\boldsymbol{P} - \boldsymbol{O}) + v(\boldsymbol{Q} - \boldsymbol{O}) \\ &= (1 - u - v)\boldsymbol{O} + u\boldsymbol{P} + v\boldsymbol{Q}, \end{aligned}$$

i.e. it has barycentric coordinates $(1 - u - v, u, v)$.

Applying the flip σ_0 effectively switches O with P; the point $\boldsymbol{p}$ is therefore

$$\begin{aligned} \boldsymbol{p} &= u\boldsymbol{O}' + (1 - u - v)\boldsymbol{P}' + v\boldsymbol{Q} \\ &= (1 + (1 - u - v) + v)\boldsymbol{O}' + (1 - u - v)(\boldsymbol{P}' - \boldsymbol{O}') + v(\boldsymbol{Q} - \boldsymbol{O}') \\ &= \boldsymbol{O}' + (1 - u - v)\,\vec{e}_1' + v\,\vec{e}_2', \end{aligned}$$

which is the point with tuple coordinates $(1 - u - v, v)$. Thus, we see that

$$\Phi_0 : (u, v) \to (-(u + v), v) + (1, 0). \tag{6.3}$$

(A vector $\vec{w} = (w_u, w_v)$ is thus transformed to $\vec{w}' = (-(w_u + w_v), w_v)$.)

Transformation under σ_1 As before, a point $\boldsymbol{p}$ with tuple coordinates (u, v) has barycentric coordinates

$$\boldsymbol{p} = (1 - u - v)\boldsymbol{O} + u\boldsymbol{P} + v\boldsymbol{Q}.$$

The flip σ_1 maintains the vertex $\boldsymbol{O}$, but interchanges the edges e_0 and e_1. This effectively switches $\boldsymbol{P}$ and $\boldsymbol{Q}$:

$$\begin{aligned} \boldsymbol{p} &= (1 - u - v)\boldsymbol{O} + v\boldsymbol{P}' + u\boldsymbol{Q}' \\ &= ((1 - u - v) + u + v)\boldsymbol{O} + v(\boldsymbol{P}' - \boldsymbol{O}) + u(\boldsymbol{Q}' - \boldsymbol{O}) \\ &= \boldsymbol{O} + v\,\vec{e}_1' + u\,\vec{e}_2', \end{aligned}$$

so we see that

$$\Phi_1 : (u, v) \to (v, u). \tag{6.4}$$

Transformation under σ_2 The transition function corresponding to σ_2 is more complicated than those corresponding to σ_0 and σ_1 because of the involvement of a new triangle $\triangle OPR$. This triangle might not even be in the same plane as $\triangle OPQ$; we therefore want to rotate it into this plane, bringing R to R'. We can then write $\boldsymbol{R}'$ in tuple coordinates:

$$\begin{aligned} \boldsymbol{R}' &= \boldsymbol{O} + r_u\,\vec{e}_1 + r_v\,\vec{e}_2 \\ &= (1 - r_u - r_v)\boldsymbol{O} + r_u\boldsymbol{P} + r_v\boldsymbol{Q}. \end{aligned}$$

We want to express $\boldsymbol{p}$ in terms of $\boldsymbol{O}$, $\boldsymbol{P}$, and $\boldsymbol{R}'$, so we solve for $\boldsymbol{Q}$,

$$\boldsymbol{Q} = \frac{1}{r_v}\left(\boldsymbol{R}' - (1 - r_u - r_v)\boldsymbol{O} - r_u\boldsymbol{P}\right),$$

and substitute: any point with tuple coordinates (u, v) has barycentric coordinates

$$\begin{aligned}
\boldsymbol{p} &= (1 - u - v)\boldsymbol{O} + u\boldsymbol{P} + v\boldsymbol{Q} \\
&= (1 - u - v)\boldsymbol{O} + u\boldsymbol{P} + \frac{v}{r_v}\left(\boldsymbol{R}' - (1 - r_u - r_v)\boldsymbol{O} - r_u\boldsymbol{P}\right) \\
&= \left((1 - u - v) - v\frac{(1 - r_u - r_v)}{r_v}\right)\boldsymbol{O} + \left(u - v\frac{r_u}{r_v}\right)\boldsymbol{P} + \frac{v}{r_v}\boldsymbol{R}'.
\end{aligned}$$

Since the rotation is an isometric transformation, the barycentric coordinate frame in $\triangle OPR'$ must be the same as that in $\triangle OPR$:

$$\begin{aligned}
\boldsymbol{p} &= \left((1 - u - v) - v\frac{(1 - r_u - r_v)}{r_v}\right)\boldsymbol{O} + \left(u - v\frac{r_u}{r_v}\right)\boldsymbol{P} + \frac{v}{r_v}\boldsymbol{R} \\
&= \left((1 - u - v) - v\frac{(1 - r_u - r_v)}{r_v} + \left(u - v\frac{r_u}{r_v}\right) + \frac{v}{r_v}\right)\boldsymbol{O} + \left(u - v\frac{r_u}{r_v}\right)(\boldsymbol{P} - \boldsymbol{O}) + \frac{v}{r_v}(\boldsymbol{R} - \boldsymbol{O}) \\
&= \boldsymbol{O} + \left(u - v\frac{r_u}{r_v}\right)\vec{e}_1' + \frac{v}{r_v}\vec{e}_2',
\end{aligned}$$

where $\vec{e}_1' = \vec{e}_1 = \boldsymbol{P} - \boldsymbol{O}$, and $\vec{e}_2' = \boldsymbol{R} - \boldsymbol{O}$. Therefore,

$$\Phi_2 : (u, v) \rightarrow \left(u - v\, {}^{r_u}\!/_{r_v}, v\, {}^{1}\!/_{r_v}\right). \tag{6.5}$$

All that remains is to find (r_u, r_v), the tuple frame coordinates of R'.

Consider the projection of the vector $(\boldsymbol{R}' - \boldsymbol{O})$ onto $\overline{OP}$:

$$\vec{R}_{\parallel}' = \frac{(\boldsymbol{R}' - \boldsymbol{O}) \cdot (\boldsymbol{P} - \boldsymbol{O})}{\|\boldsymbol{P} - \boldsymbol{O}\|^2}(\boldsymbol{P} - \boldsymbol{O}).$$

Because vertex R' is the result of rotating R around $\overline{OP}$, this should be the same as the projection of $(\boldsymbol{R} - \boldsymbol{O})$ onto the same line:

$$\vec{R}_{\parallel}' = \vec{R}_{\parallel} = \frac{(\boldsymbol{R} - \boldsymbol{O}) \cdot (\boldsymbol{P} - \boldsymbol{O})}{\|\boldsymbol{P} - \boldsymbol{O}\|^2}(\boldsymbol{P} - \boldsymbol{O}).$$

The only difference between the two is therefore in the parts perpendicular to $\overline{OP}$:

$$\vec{R}_{\perp} = (\boldsymbol{R} - \boldsymbol{O}) - \vec{R}_{\parallel} \qquad \text{and} \qquad \vec{R}_{\perp}' = (\boldsymbol{R}' - \boldsymbol{O}) - \vec{R}_{\parallel}'.$$

Both vertices R and R' are the same distance from the axis of rotation $\overline{OP}$, so the two vectors $\vec{R}_{\perp}$ and $\vec{R}_{\perp}'$ must be of the same length. In addition, the direction of $\vec{R}_{\perp}'$ must be *opposite* that of the part of $(\boldsymbol{Q} - \boldsymbol{O})$ perpendicular to $\overline{OP}$:

$$\vec{R}_{\perp}' = -\frac{\|\vec{R}_{\perp}\|}{\|\vec{Q}_{\perp}\|}\vec{Q}_{\perp}.$$

Since

$$\vec{Q}_\perp = (\boldsymbol{Q}-\boldsymbol{O}) - \frac{(\boldsymbol{Q}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}(\boldsymbol{P}-\boldsymbol{O}),$$

we can substitute into $\vec{R}'_\perp = (\boldsymbol{R}'-\boldsymbol{O}) - \vec{R}'_\parallel$ to see that

$$\boldsymbol{R}'-\boldsymbol{O} = \frac{(\boldsymbol{R}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}(\boldsymbol{P}-\boldsymbol{O}) - \frac{||\vec{R}_\perp||}{||\vec{Q}_\perp||}\left((\boldsymbol{Q}-\boldsymbol{O}) - \frac{(\boldsymbol{Q}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}(\boldsymbol{P}-\boldsymbol{O})\right).$$

This vector is $\vec{e}_2$; if we can express it in ΔOPQ's tuple coordinates, we can relate them to the tuple coordinates of ΔOPR. Recall that a vector $\vec{p} = (u, v)$ in coordinates $\{\vec{e}_1, \vec{e}_2\}$ is represented in $\{x, y, z\}$ coordinates as

$$\vec{p} = u\,\vec{e}_1 + v\,\vec{e}_2 = (u\,e_{1x} + v\,e_{2x}, u\,e_{1y} + v\,e_{2y}, u\,e_{1z} + v\,e_{2z}),$$

or

$$\begin{pmatrix} e_{1x} & e_{2x} \\ e_{1y} & e_{2y} \\ e_{1z} & e_{2z} \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}.$$

As we want to solve for (u, v), we add a third vector $\vec{e}_3$ to the coordinate frame:

$$\begin{pmatrix} e_{1x} & e_{2x} & e_{3x} \\ e_{1y} & e_{2y} & e_{3y} \\ e_{1z} & e_{2z} & e_{3z} \end{pmatrix}\begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}.$$

This system can now be solved with Cramer's rule:

$$u = \frac{\vec{e}_3\cdot(\vec{p}\times\vec{e}_2)}{\vec{e}_3\cdot(\vec{e}_1\times\vec{e}_2)}, \qquad v = \frac{\vec{e}_3\cdot(\vec{e}_1\times\vec{p})}{\vec{e}_3\cdot(\vec{e}_1\times\vec{e}_2)}.$$

After substituting $\vec{e}_1 = (\boldsymbol{P}-\boldsymbol{O}), \vec{e}_2 = (\boldsymbol{Q}-\boldsymbol{O}), \vec{e}_3 = \vec{e}_1\times\vec{e}_2$, and $\vec{p} = (\boldsymbol{R}'-\boldsymbol{O})$ we see that

$$\begin{aligned} r_u &= \frac{((\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}))\cdot((\boldsymbol{R}'-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}))}{((\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}))^2} \quad \text{and} \\ r_v &= \frac{((\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}))\cdot((\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{R}'-\boldsymbol{O}))}{((\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}))^2}. \end{aligned}$$

The cross product of $(\boldsymbol{R}'-\boldsymbol{O})$ with $(\boldsymbol{Q}-\boldsymbol{O})$ is

$$\begin{aligned} (\boldsymbol{R}'-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}) = {} & \frac{(\boldsymbol{R}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}) - \\ & \frac{||\vec{R}_\perp||}{||\vec{Q}_\perp||}\left((\boldsymbol{Q}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}) - \frac{(\boldsymbol{Q}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O})\right) \\ = {} & \left(\frac{(\boldsymbol{R}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2} - \frac{||\vec{R}_\perp||}{||\vec{Q}_\perp||}\left(0 - \frac{(\boldsymbol{Q}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}\right)\right)(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}) \end{aligned}$$

so

$$r_u = \frac{(\boldsymbol{R}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2} + \frac{||\vec{R}_\perp||}{||\vec{Q}_\perp||}\frac{(\boldsymbol{Q}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{||\boldsymbol{P}-\boldsymbol{O}||^2}. \qquad (6.6)$$

The cross product of $(\boldsymbol{P}-\boldsymbol{O})$ with $(\boldsymbol{R}'-\boldsymbol{O})$ is

$$\begin{aligned}(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{R}'-\boldsymbol{O}) &= \frac{(\boldsymbol{R}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{\|\boldsymbol{P}-\boldsymbol{O}\|^2}(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{P}-\boldsymbol{O})- \\ &\quad \frac{\|\vec{R}_\perp\|}{\|\vec{Q}_\perp\|}\left((\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O})-\frac{(\boldsymbol{Q}-\boldsymbol{O})\cdot(\boldsymbol{P}-\boldsymbol{O})}{\|\boldsymbol{P}-\boldsymbol{O}\|^2}(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{P}-\boldsymbol{O})\right) \\ &= -\frac{\|\vec{R}_\perp\|}{\|\vec{Q}_\perp\|}(\boldsymbol{P}-\boldsymbol{O})\times(\boldsymbol{Q}-\boldsymbol{O}).\end{aligned}$$

so

$$r_v = -\frac{\|\vec{R}_\perp\|}{\|\vec{Q}_\perp\|}. \tag{6.7}$$

Equations 6.6 and 6.7 give us the values of r_u and r_v; we can substitute these into Equation 6.5 to find the transition function corresponding to σ_2:

$$\Phi_2 : (u,v) \rightarrow (u - v\, {r_u}/{r_v}, v\, {1}/{r_v}).$$

6.3.2 The geodesic propagation algorithm

Algorithm 6.8 propagates a geodesic across a triangle mesh. It takes as arguments a cell tuple τ, a position $\boldsymbol{q}$ and direction $\vec{x}$ defined in the tuple frame, and a distance d, then follows the geodesic across triangles until the total travelled distance equals d.

The entire procedure is a loop which continues until the distance left to travel is zero. In each iteration, the geodesic is propagated across one face of the mesh. For each triangle, the first step taken is to find the intersection points of the geodesic with the triangle's edges (lines 3–15). We are looking for the next edge intersected by the geodesic; this is the edge whose intersection point is in the direction of $\vec{x}$ (lines 6–15). Which edge in particular is stored in e; the edge of the current tuple is 0, the **other** edge is 1, and the third edge of the triangle is 2. The point the geodesic leaves the triangle, in barycentric coordinates, is recorded as ℓ.

We know the distance to ℓ in barycentric coordinates, but this is not necessarily the same as the distance in the embedding space. We thus determine the scale d_s between the magnitude of a barycentric vector and its embedded length. After retrieving the points $\boldsymbol{O}$, $\boldsymbol{P}$, and $\boldsymbol{Q}$ (lines 16–18), we divide the world length of $\vec{x}$ by its barycentric length to find the scale factor (line 19). The length d_t (line 20) is the distance left within the triangle; if this is greater than the total distance remaining (line 21), then the geodesic will end within this triangle. In this case, the geodesic is advanced by the remaining distance, and the callback function PolylinePoint is called to place the point (line 24).

If the distance remaining is greater than the distance within the triangle, then the geodesic must be advanced to the exit point and transformed into the next face. First, the exit point is recorded as the next point of the geodesic (line 28). We then have to apply flips to move the tuple to the exit edge. If this is the current edge (if $e = 0$), then nothing needs be done. If the exit edge is the **other** edge of the tuple ($e = 1$), then we must perform a **flip**(1) operation on the tuple and the geodesic (lines 29–32). The point $\boldsymbol{q}$ and vector $\vec{x}$ are thus modified by the transition function corresponding to this flip

Algorithm 6.8 Geodesic propagation (part 1)

Input: a cell tuple τ, a position $\boldsymbol{q}$ and direction $\vec{x}$ in τ's tuple frame, and a distance d

```
1:  procedure PropagateGeodesic($\tau$, $\boldsymbol{q}$, $\vec{x}$, $d$)
2:      while $d > 0$ do
3:          $u_0 \leftarrow -\frac{x_u}{x_v} q_v + q_u$                      ▷ $u$-crossing of geodesic
4:          $v_0 \leftarrow -\frac{x_v}{x_u} q_u + q_v$                      ▷ $v$-crossing of geodesic
5:          $w_0 \leftarrow q_u + x_u \frac{1 - q_u - q_v}{x_u + x_v}$       ▷ $(u + v = 1)$-crossing of geodesic
6:          if $0 < u_0 < 1$ and $((u_0, 0) - \boldsymbol{q}) \cdot \vec{x} > 0$ then
7:              $e \leftarrow 0$
8:              $\ell \leftarrow (u_0, 0)$
9:          else if $0 < v_0 < 1$ and $((0, v_0) - \boldsymbol{q}) \cdot \vec{x} > 0$ then
10:             $e \leftarrow 1$
11:             $\ell \leftarrow (0, v_0)$
12:         else if $0 < w_0 < 1$ and $((w_0, 1 - w_0) - \boldsymbol{q}) \cdot \vec{x} > 0$ then
13:             $e \leftarrow 2$
14:             $\ell \leftarrow (w_0, 1 - w_0)$
15:         end if
16:         $\boldsymbol{O} \leftarrow \boldsymbol{P}(\tau[0])$
17:         $\boldsymbol{P} \leftarrow \boldsymbol{P}(\tau.\textbf{other}(0))$
18:         $\boldsymbol{Q} \leftarrow \boldsymbol{P}(\tau.\textbf{flip}(1).\textbf{other}(0))$
19:         $d_s \leftarrow ((\boldsymbol{P} - \boldsymbol{O})x_u + (\boldsymbol{Q} - \boldsymbol{O})x_v) \ / \ \|\vec{x}\|$       ▷ distance scale
```

Algorithm 6.8 Geodesic propagation (part 2)

```
20:             $d_t \leftarrow d_s \|\boldsymbol{\ell} - \boldsymbol{q}\|$
21:             if $d_t \geq d$ then                         ▷ geodesic ends in this triangle
22:                 $\boldsymbol{q} \leftarrow \boldsymbol{q} + (d/d_s)\vec{x}$
23:                 $d \leftarrow 0$
24:                 PolylinePoint($q$)
25:             else                                         ▷ go to the next triangle
26:                 $d \leftarrow d - d_t$
27:                 $\boldsymbol{q} \leftarrow \boldsymbol{\ell}$
28:                 PolylinePoint($q$)
29:                 if $e = 1$ then                          ▷ move to edge where geodesic departs
30:                     $\tau$.flip(1)
31:                     $\boldsymbol{q} \leftarrow (q_v, q_u)$
32:                     $\vec{x} \leftarrow (x_v, x_u)$
33:                 else if $e = 2$ then
34:                     $\tau$.flip(0)
35:                     $\boldsymbol{q} \leftarrow (1 - (q_u + q_v), q_v)$
36:                     $\vec{x} \leftarrow (-(x_u + x_v), x_v)$
37:                     $\tau$.flip(1)
38:                     $\boldsymbol{q} \leftarrow (q_v, q_u)$
39:                     $\vec{x} \leftarrow (x_v, x_u)$
40:                 end if
41:                 $\tau$.flip(2)                           ▷ transition into next face
42:                 $\boldsymbol{R} \leftarrow \boldsymbol{P}(\tau.\mathbf{flip}(1).\mathbf{other}(0))$
43:                 $Q_{dot} \leftarrow \dfrac{(\boldsymbol{Q} - \boldsymbol{O}) \cdot (\boldsymbol{P} - \boldsymbol{O})}{(\boldsymbol{P} - \boldsymbol{O})^2}$
44:                 $\vec{Q}_{\perp} \leftarrow (\boldsymbol{Q} - \boldsymbol{O}) - Q_{dot}(\boldsymbol{P} - \boldsymbol{O})$
45:                 $R_{dot} \leftarrow \dfrac{(\boldsymbol{R} - \boldsymbol{O}) \cdot (\boldsymbol{P} - \boldsymbol{O})}{(\boldsymbol{P} - \boldsymbol{O})^2}$
46:                 $\vec{R}_{\perp} \leftarrow (\boldsymbol{R} - \boldsymbol{O}) - R_{dot}(\boldsymbol{P} - \boldsymbol{O})$
47:                 $r_u \leftarrow R_{dot} + \dfrac{\|\vec{R}_{\perp}\|}{\|\vec{Q}_{\perp}\|} Q_{dot}$
48:                 $r_v \leftarrow -\dfrac{\|\vec{R}_{\perp}\|}{\|\vec{Q}_{\perp}\|}$
49:                 $\boldsymbol{q} \leftarrow (q_u - q_v\ {}^{r_u}/_{r_v}, q_v\ {}^{1}/_{r_v})$
50:                 $\vec{x} \leftarrow (x_u - x_v\ {}^{r_u}/_{r_v}, x_v\ {}^{1}/_{r_v})$
51:             end if
52:         end while
53:     end procedure
```

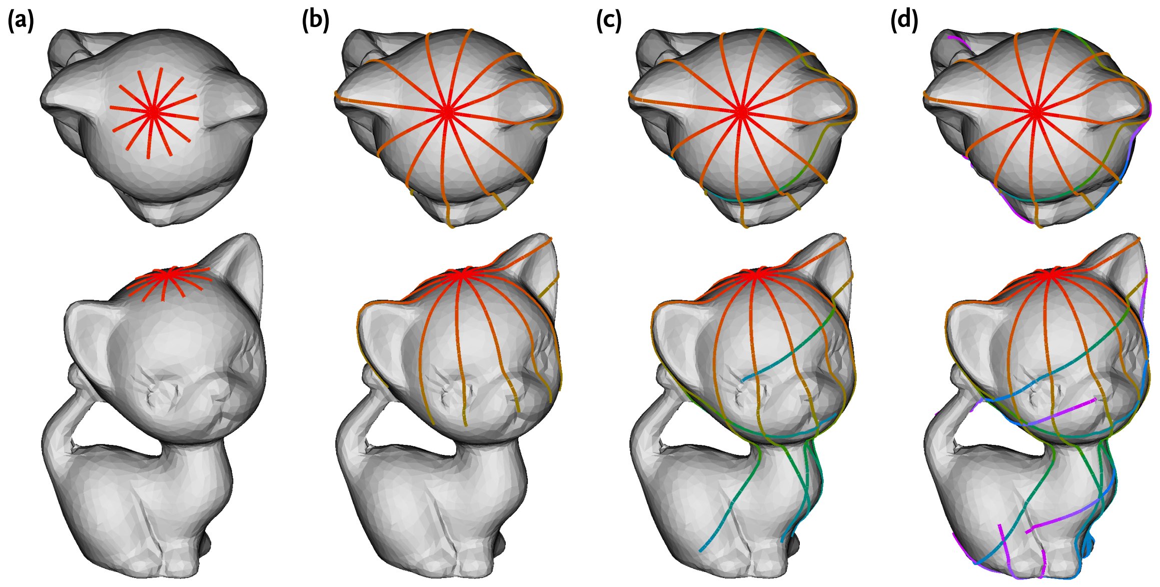


Figure 6.18: Several geodesics propagate across a mesh.

operation (Equation 6.4). If the exit edge is the third edge of the triangle, we must get there by the flip sequence **flip**$(0, 1)$ (lines 33–40). This means that we must first apply the transition corresponding to σ_0 (Equation 6.3), then the transition corresponding to σ_1 (Equation 6.4).

Once the tuple has reached the exit edge, we must flip it into the next face. We find r_u according to Equation 6.6 (line 47) and r_v according to Equation 6.7 (line 48), then use Equation 6.5 to find the corresponding values of $\boldsymbol{q}$ and $\vec{x}$ (lines 49–50). The iteration then continues with the next face.

Figure 6.18 shows a result of using the geodesic propagation algorithm. Several geodesics are started at the top of the kitten mesh, and propagated in all directions.

This mechanism for producing straight lines can be used to define *turtle geometry* (Abelson and diSessa 1986) on an arbitrary subdivided manifold. A common geometric interpretation of L-systems, for instance, is defined through turtle geometry (Prusinkiewicz and Lindenmayer 1990), and by using Algorithm 6.8 many L-systems can be run on an arbitrary surface defined by a triangular mesh. For example, Figure 6.19 shows the result of using the geodesic propagation algorithm to create an Ulam branching pattern (Ulam 1962) on the surface of a mesh.

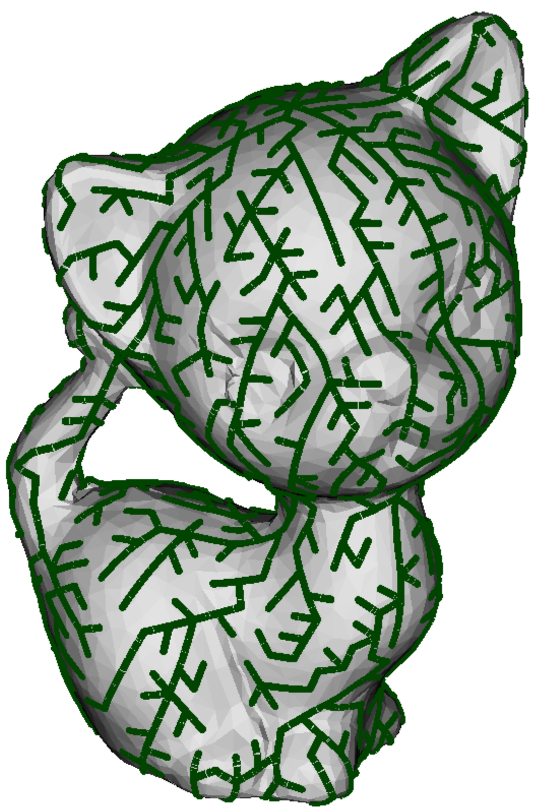


Figure 6.19: An Ulam branching pattern grown on the surface of a mesh.

7 Biological modeling

The original inspiration for developmental modeling was models of the growth and development of plants. In this chapter, I demonstrate how cell complexes can be applied to this area. In Section 7.1 I discuss modeling plant form with simulation of the mechanics of elastic sheets. In Section 7.2 I talk about dividing cells and simulating chemical processes in expanding tissues. Finally, in Section 7.3 I describe two models which seek to reproduce the growth and form of a specific plant: the moss *Physcomitrella patens*.

7.1 Mechanical simulation

One of the advantages of modeling with cell complexes put forward in Chapter 1 was that physical quantities are associated with cells of the appropriate dimension. A good illustration of this advantage is in simulating Newtonian mechanics. In the thin-shell model described later in this section, for example, points have positions and velocities, edges have flexural energy, faces have stretching energy, and volumes have pressure. In this section I will cover three cell complex models of plantlike forms which are created by resolving the Newtonian mechanics of elastic sheets.

Mass-spring systems A simple representation of an elastic body is the *mass-spring system* (Figure 7.1a). The body is modeled as a set of point masses connected by massless springs. It is often assumed that the springs apply forces to the masses according to Hooke's law. Each spring has a *rest length* ℓ_0, and the applied force is in opposition to a

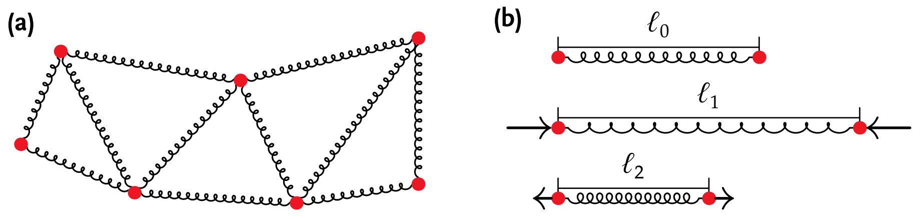


Figure 7.1: (a) A mass-spring system. Point masses (red) are joined by springs (black). (b) Each spring has a rest length ℓ_0; if the spring is stretched to a length $\ell_1 > \ell_0$, it applies an inward force on the masses, while if the spring is compressed to a length $\ell_2 < \ell_0$, the applied force is directed outward.

change in length with respect to ℓ_0, with the magnitude proportional to the difference between the current length and the rest length. Thus, if the spring is stretched to a length $\ell_1 > \ell_0$, the force applied to each mass will be of magnitude $k\ (\ell_1 - \ell_0)$ (where the constant of proportionality k is called the *spring constant*) and will be directed inward, while if the spring is compressed to a length $\ell_2 < \ell_0$, the force applied to each mass will be of magnitude $k\ (\ell_0 - \ell_2)$ and will be directed outward.

Suppose the masses m_i and m_j are at positions $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, respectively, and the spring joining them has rest length ℓ_{0ij}. The current length of the spring is $\ell = \|\boldsymbol{x}_j - \boldsymbol{x}_i\|$, so the magnitude of the applied force is $k_{ij}\ \|\ell - \ell_{0ij}\|$. If the spring has been stretched ($\ell > \ell_{0ij}$), then the force applied to m_i will be towards m_j; that is, it will be in the direction of $\vec{e}_{ij}$. On the other hand, if the spring has been compressed ($\ell < \ell_{ij}$), then the force applied to m_i will be away from m_j, in the direction of $-\vec{e}_{ij}$. In either case, then, we see that the force applied by the spring on m_i is

$$\vec{F}_i = k_{ij}\ \left(\|\vec{e}_{ij}\| - \ell_{ij}\right)\ \frac{e_{ij}}{\|\vec{e}_{ij}\|} = k_{ij}\ \left(1 - \frac{\ell_{ij}}{\|\vec{e}_{ij}\|}\right)\ \vec{e}_{ij}.$$

Similarly, the force applied by the spring on m_j is

$$\vec{F}_j = -\vec{F}_i = -k_{ij}\ \left(1 - \frac{\ell_{ij}}{\|\vec{e}_{ij}\|}\right)\ \vec{e}_{ij}.$$

The masses in a mass-spring system can in general be connected in any way, but in most cases we wish to model a subdivided manifold. In a cell complex, we can model a mass-spring system by representing point masses by vertices and springs by edges connecting the vertices. Higher-dimensional cells are not involved at all in the simulation It is then clear that the force applied by a spring is a property of an edge, while the position or mass of a point mass is a property of a vertex. We can compute all of the forces in the system by visiting every edge in the cell complex. Algorithm 7.1 performs one timestep of the simulation of a mass-spring system.

The forces for each vertex v are accumulated in $\vec{F}(v)$. Lines 2–4 initialize all of these forces to zero. The loop between lines 5 and 11 computes all of the spring forces in the system. Line 6 retrieves the endpoints of the edge e, and line 7 finds the vector $\vec{e}$ between them. The force is computed in line 8 using e's rest length $L(e)$ and the length of $\vec{e}$, then applied to the endpoints in lines 9 and 10. In the last loop (lines 12–15) the positions and velocities of all of the vertices are updated using a symplectic Euler integration with timestep Δt (Stern and Desbrun 2006).

Mass-spring systems are useful and have been used in a wide variety of developmental models of plant form. As an example, I will present cell complex implementations of two models of organ shape. The first is the model of leaf shape presented by Prusinkiewicz and Barbier de Reuille (2010), originally implemented in the vertex-vertex modeling system VVE. This model produces a surface exhibiting a fractal cascade of waves of increasing frequency, reminiscent of the leaves of *Brassica oleracea* varieties such as kale (Figure 7.3a).

A sheet representing the leaf is divided into rows of rectangles, each of which is further divided into three triangles (Figure 7.3). The number of rectangles doubles in each successive row, while the lengths of the edges is reduced by a constant factor r. If $r = 1/2$,

Algorithm 7.1 Model of a mass-spring system

```
Require: Vertices $v$ have mass $M(v)$, position $\boldsymbol{P}(v)$ and velocity $\vec{V}(v)$; edges $e$ have rest length $L(e)$
1: procedure MassSpringTimeStep
2:     for all vertex $v$ do                                ▷ initialize forces to zero
3:         $\vec{F}(v) \leftarrow 0$
4:     end for
5:     for all edge $e$ do                                  ▷ compute spring forces
6:         $\{+v_0, -v_1\} \leftarrow$ boundary($e$)
7:         $\vec{e} \leftarrow \boldsymbol{P}(v_1) - \boldsymbol{P}(v_0)$
8:         $\vec{f} \leftarrow k\left(1 - (L(e) / \|\vec{e}\|)\right) \vec{e}$
9:         $\vec{F}(v_0) \leftarrow \vec{F}(v_0) + \vec{f}$
10:        $\vec{F}(v_1) \leftarrow \vec{F}(v_1) - \vec{f}$
11:    end for
12:    for all vertex $v$ do                                ▷ update positions and velocities
13:        $\vec{V}(v) \leftarrow \vec{V}(v) + \left(\vec{F}(v) / \vec{M}(v)\right) \Delta t$
14:        $\boldsymbol{P}(v) \leftarrow \boldsymbol{P}(v) + \vec{V}(v)\, \Delta t$
15:    end for
16: end procedure
```

as in Figure 7.3, then this formation can be realized in two dimensions. For larger values of r, the formation can only be realized by buckling into the third dimension. The leaf model creates such a realization by making each edge a spring whose rest length is s_i, d_i, or h_i, then letting the mass-spring system relax into proper shape.

In the cell-complex implementation of this model, the surface is produced one row at a time. Algorithm 7.2 is used to produce the new row. The algorithm maintains an ordered list *leading* of the vertices in the leading row (Figure 7.2a). The first loop of the algorithm (lines 2–18) creates new edges and vertices which connect back to one of the leading vertices v_i (Figure 7.2b). The first step is to find the direction of the new edges; this is done by finding the position on the *previous* leading row corresponding to v_i and subtracting this from v_i's position. The tuple τ is used to find the vertex; initially it is set to any vertex containing v_i (line 3), then lines 4–6 ensure that the tuple's edge is one that connects to the previous row.

Even-indexed vertices connect to only one vertex in the previous boundary; these are also exactly those vertices which have two (for corner vertices) or three neighbouring vertices. We check this in line 7; if the vertex connects back to one corresponding vertex τ.**other**(0), then we set $\vec{u}$ to the offset between the two. If, on the other hand, the vertex v_i has *four* neighbouring vertices, then it must connect back to two vertices in the previous row. These are τ.**other**(0) and τ.**flip**(1).**other**(0), and the position corresponding to v_i is halfway between them (line 10). In either case, once the offset $\vec{u}$ is known, a new vertex is created (line 13) and given a position offset from the position of v_i in the direction of $\vec{u}$ at distance h (line 14). A new edge is created from the old vertex to the

Algorithm 7.2 Production of a new row of triangles in the kale-leaf model

```
procedure AddOneRow($leading$,$s$,$d$,$h$)
    for vertex $v_i \in leading$ do
        $\tau \leftarrow$ tuple containing $v_i$        ▷ find direction $\vec{r}$ of next vertex
        if $\tau$.other(0) $\in leading$ then
            $\tau \leftarrow \tau$.flip(1, 2)
        end if
        if $\|$neighbours$(v_i)\| = 2$ or 3 then
            $\vec{u} \leftarrow P(v_i) - P(\tau.\mathbf{other}(0))$
        else if $\|$neighbours$(v_i)\| = 4$ then
            $a \leftarrow 1/2\, P(\tau.\mathbf{other}(0)) + 1/2\, P(\tau.\mathbf{flip}(1).\mathbf{other}(0))$
            $\vec{u} \leftarrow P(v_i) - a$
        end if
        $w \leftarrow$ addCell()        ▷ create new vertex and edge
        $P(w) \leftarrow P(v_i) + h\hat{u}$
        $e \leftarrow$ addCell($+v_i\ - w$)
        $L(e) \leftarrow h$
        $newLeading[2i] \leftarrow w$
    end for
    for $v_i, v_{i+1} \in leading$ do
        $w_i \leftarrow newLeading[2i]$
        $w_{i+1} \leftarrow newLeading[2i+2]$
        $e \leftarrow$ addCell($+w_i\ - w_{i+1}$)
        addCell($+$join$(v_i, w_i)\ + e\ -$join$(v_{i+1}, w_{i+1})\ -$join$(v_i, v_{i-1})$)
        $(e_L, w', e_R) \leftarrow$ splitCell($e$)
        $L(e_L) \leftarrow s$
        $L(e_R) \leftarrow s$
        $newLeading[2i+1] \leftarrow w'$
        $P(w) \leftarrow 1/2\,(P(w_i) + P(w_{i+1})) + \vec{o}$
        $(f_L, e_i, f_R) \leftarrow$ splitCell(join$(v_i, w')$, $+v_i\ - w'$)
        $(f'_L, e_{i+1}, f'_R) \leftarrow$ splitCell(join$(v_{i+1}, w')$, $+v_{i+1}\ - w'$)
        $L(e_i) \leftarrow d$
        $L(e_{i+1}) \leftarrow d$
    end for
    $leading \leftarrow newLeading$
end procedure
```

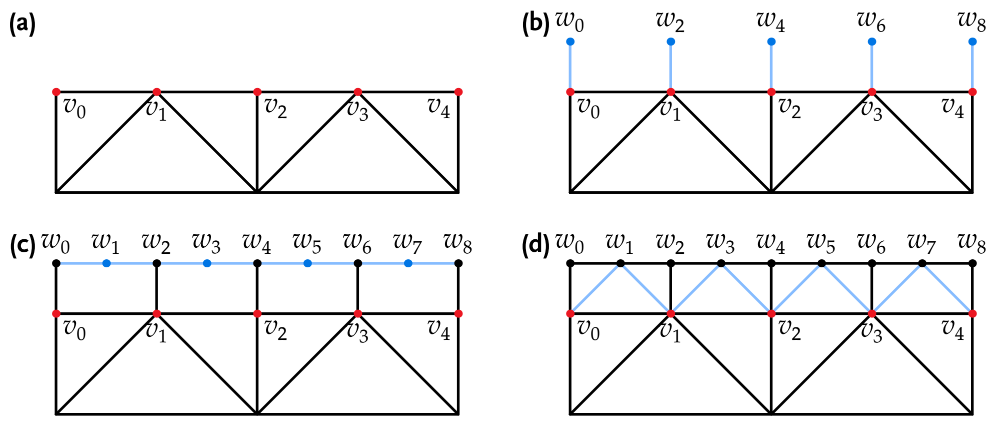

(a)
(b)
(c)
(d)
v_0 v_1 v_2 v_3 v_4
w_0 w_1 w_2 w_3 w_4 w_5 w_6 w_7 w_8

Figure 7.2: Adding a new row to the fractal leaf model. (a) Initial configuration; the vertices $\{v_0, \dots, v_4\}$ are stored in the ordered list *leading*. (b) Even-indexed new vertices w_{2i} are created in front of each leading vertex v_i and attached by new edges. (c) The new vertices are joined by new edges, which are split to create the odd-indexed new vertices w_{2i+1}. (d) The five-sided faces are split into three triangles by diagonal edges between w_{2i+1} and v_i and v_{i+1}.

new (line 15) and given the rest length h (line 16). Finally, the new vertex is added to the list *newLeading* of new leading vertices. Since there will be more vertices added between these, the new vertex corresponding to v_i is w_{2i}.

In the second loop (lines 19–33), the rest of the new cells are created. The iteration is over adjacent old vertices v_i, v_{i+1}; we then recall the corresponding new vertices w_i, w_{i+1} (lines 20–21). A new edge e is created between the new vertices (line 22), then a new face bounded by the edges between v_i, v_{i+1}, w_{i+1}, and w_i (line 23). The edge e is split (line 24) and the new edges have their rest lengths set to s (lines 25–26). The new vertex is placed into the *newLeading* list between w_i and w_{i+1} (line 27) and its position is set halfway between those vertices (line 28). The position is altered by an offset $\vec{o}$; this offset determines which way the surface bends in three dimensions. Following Prusinkiewicz and Barbier de Reuille (2010), this model alternates placing the vertex to the left and to the right of the plane. This produces a margin in the form of the space-filling *dragon curve*, so the margin develops without self-intersection (Figure 7.3b).

After the new vertex w' is placed, it is used to split the face twice (Figure 7.2d). The first split is between v_i and w' and creates the diagonal edge e_i (line 29), while the second is between v_{i+1} and w' and creates the diagonal edge e_{i+1}. Both edges have their rest lengths set to d (lines 31–32). Finally, once all faces have been created, the list *leading* is updated to the list of new leading vertices *newLeading* (line 34).

After each row is added, we iterate the mass-spring relaxation of Algorithm 7.1. In order to minimize oscillation, we add a damping force to each vertex, opposed to the vertex's motion:

$$\vec{F}(v) \leftarrow \vec{F}(v) - \nu \vec{V}(v),$$

where ν is a small damping coefficient. The iteration is ended when the total force drops below a small constant ϵ; the system is then considered to be at equilibrium.

Choosing $r = \sqrt{1/2}$ (after Prusinkiewicz and Barbier de Reuille (2010)), the model produces a surface exhibiting the wavy nature we expect (Figure 7.3).

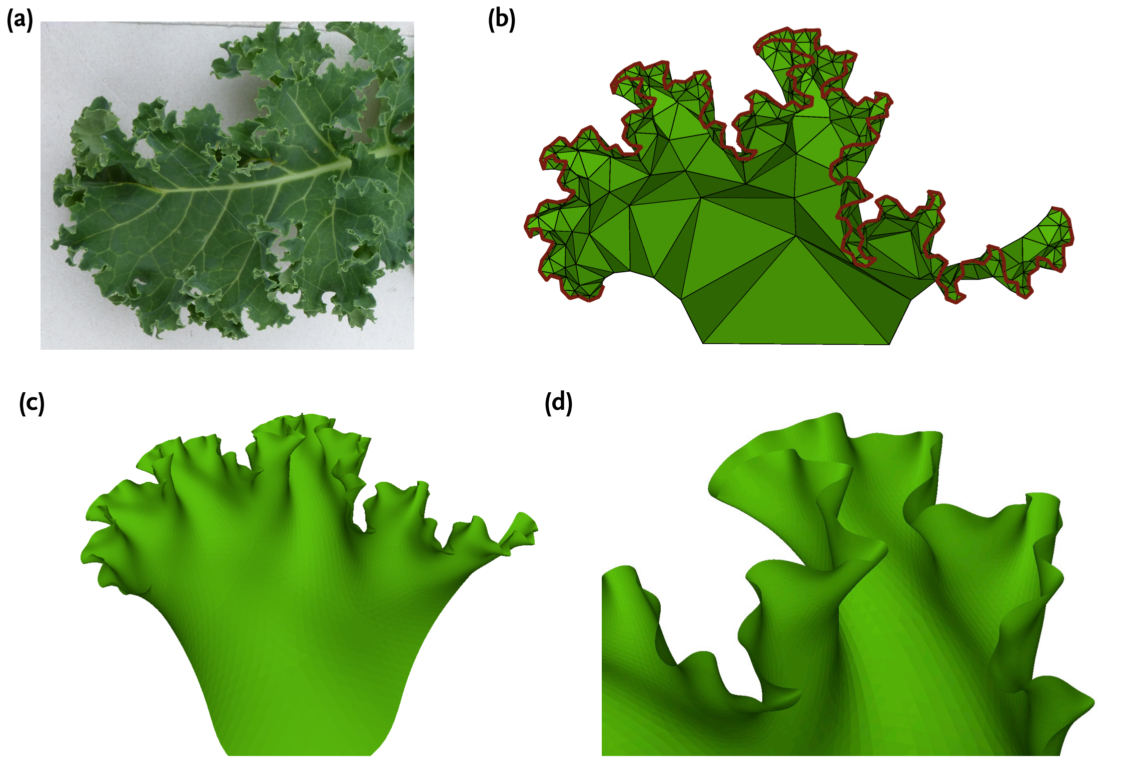


Figure 7.3: (a) Photograph of a kale leaf. (b) A simulated leaf surface resembling kale. (c) The kale-leaf model, with ten rows added, followed by several rounds of subdivision for smoothing. (d) Detail of the model.

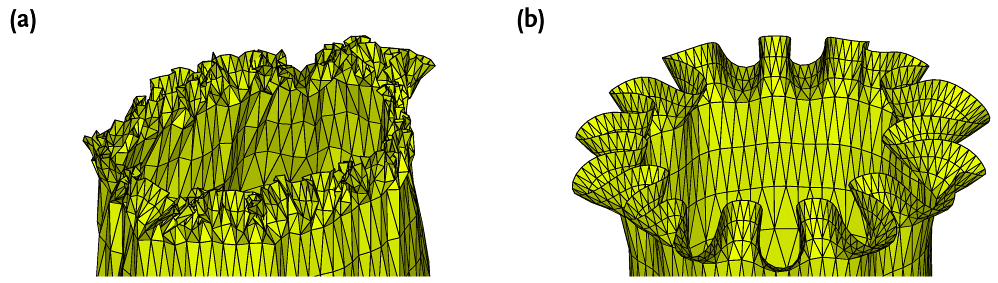


Figure 7.4: The equilibrium state of a mass-spring system. (a) The rest circumference of the shape increases in the higher rings, and the only forces resolved are those which govern the edge lengths. The result is a surface which bends severely on many edges. (b) The rest lengths are the same as in the left figure, but there are also angular springs ensuring the surface bends more gradually and periodically.

Angular springs The mass-spring system in the kale-leaf model finds a configuration of the surface in which all of the edges are of the required lengths. More generally, though, a mass-spring system can be used to find an equilibrium between opposing forces. This is the case with the next model, originally presented by Smith (2006) in the vv language, of wavy tubular plant organs, such as in the flowers of daffodil. Here the forces seeking to maintain edge lengths act in opposition to forces that try to avoid folding of the surface. The effect of this is that the surface bends gradually and less frequently than otherwise (Figure 7.4).

The forces that seek to prevent the surface from folding are provided by *angular springs* acting at each edge. These springs work in a manner analogous to the linear springs which govern edge length, but the magnitude of the applied force is proportional to the difference between the edge's *rest angle* Θ, and θ, the angle between the faces separated by the edge:

$$||\vec{F}|| = \kappa\,(\theta - \Theta).$$

This is a reasonable approximation to the torque created by a physical spring on the edge if the triangles are of similar size and the angular difference is small. The force is applied to four vertices: the endpoints of the edge are pushed parallel to the average of the normals of the faces, while the opposite vertices on the triangles are pushed in the other direction (Figure 7.5b); this force thus opens or closes the angle as required.

Algorithm 7.3 computes this bending spring force. It modifies the accumulated force $\vec{F}(v)$ associated with a vertex and could thus be inserted, for instance, after line 11 in Algorithm 7.1. The forces are accumulated one edge at a time (lines 2–22). Edges on the border of the cell complex have no bending force, so they are skipped (lines 3–5).

The two triangles adjoining edge e are f_ℓ and f_r (Figure 7.5a), and have a total of four vertices: v_a and v_b are the endpoints of e (line 6), while v_ℓ and v_r are the third vertices of f_ℓ and f_r, respectively (lines 8–9). The face normals are computed in lines 10 and 11; the angle θ is then the inverse cosine of their dot product (line 12). We then check the dot product of $\vec{n}_r$ with the vector from v_ℓ to v_a (line 13). If the dot product is negative, then the angle is concave (Figure 7.5c) and we negate θ to reflect this (line 14). The direction of the force is the average of $\vec{n}_\ell$ and $\vec{n}_r$ (line 16), and the magnitude is applied

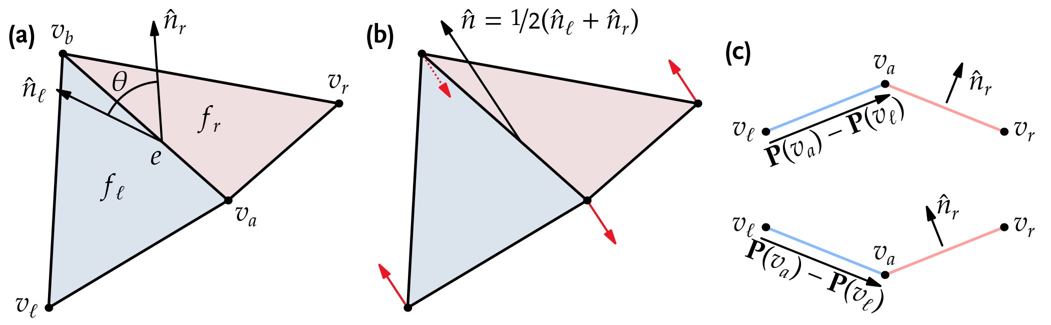


Figure 7.5: The angular spring model of Smith (2006). (a) The angle between two faces f_ℓ and f_r separated by edge e is the angle between the faces' respective normals, $\hat{n}_\ell$ and $\hat{n}_r$. (b) The force applied to resist bending is parallel to the average of the normals, $\vec{n} = 1/2\ (\hat{n}_\ell + \hat{n}_r)$. On v_a and v_b, the endpoints of edge e, the force is in the same direction as $\vec{n}$ when $\theta < \Theta$, and in the opposite direction otherwise; on v_ℓ and v_r, the force is in the opposite direction to $\vec{n}$ when $\theta < \Theta$, and in the same direction otherwise. (c) The sign of the angle θ can be found by calculating the dot product between $\hat{n}_r$ and $(P(v_a) - P(v_\ell))$. This value is positive if the angle is convex (top), negative if the angle is concave (bottom).

Algorithm 7.3 Computation of bending spring forces

Require: Vertex v has position $P(v)$ and accumulates force $\vec{F}(v)$; edge e has rest angle $\Theta(e)$

```
 1: procedure BendingSpringForce
 2:     for all edge $e$ do
 3:         if border(e) then                          ▷ no bending force on border of complex
 4:             continue to next edge
 5:         end if
 6:         $\{+v_a, -v_b\} \leftarrow$ boundary($e$)          ▷ find vertices $\{v_a, v_b, v_\ell, v_r\}$
 7:         $\tau \leftarrow$ tuple containing $\{e, v_a\}$
 8:         $v_\ell \leftarrow \tau$.flip(1).other(0)
 9:         $v_r \leftarrow \tau$.flip(2, 1).other(0)
10:         $\vec{n}_\ell \leftarrow (P(v_a) - P(v_\ell)) \times (P(v_b) - P(v_\ell))$          ▷ face normals
11:         $\vec{n}_r \leftarrow (P(v_b) - P(v_r)) \times (P(v_a) - P(v_r))$
12:         $\theta \leftarrow \cos^{-1}(\hat{n}_\ell \cdot \hat{n}_r)$          ▷ angle between faces
13:         if $\vec{n}_r \cdot (P(v_a) - P(v_\ell)) < 0$ then
14:             $\theta = -\theta$
15:         end if
16:         $\vec{n} \leftarrow 1/2\ (\vec{n}_\ell + \vec{n}_r)$
17:         $\vec{f} \leftarrow \kappa\ (\theta - \Theta(e))\ \hat{n}$
18:         $\vec{F}(v_a) \leftarrow \vec{F}(v_a) - \vec{f}$
19:         $\vec{F}(v_b) \leftarrow \vec{F}(v_b) - \vec{f}$
20:         $\vec{F}(v_\ell) \leftarrow \vec{F}(v_\ell) + \vec{f}$
21:         $\vec{F}(v_r) \leftarrow \vec{F}(v_r) + \vec{f}$
22:     end for
23: end procedure
```

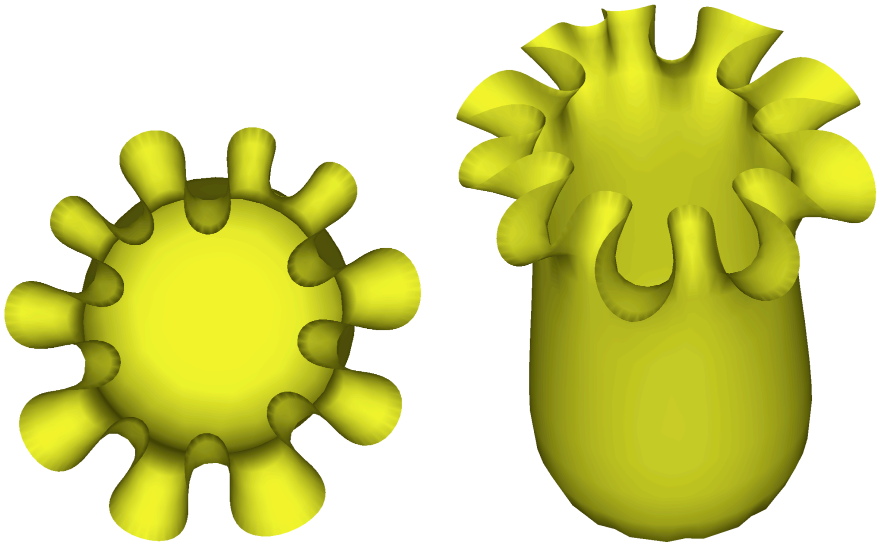


Figure 7.6: The model creates a tubular flower with smooth, roughly periodic wrinkles.

in line 17. Finally, the force vector is subtracted from the accumulated force for v_a and v_b, and added to the accumulated force for v_ℓ and v_r (lines 18–21).

As with the leaf model, the tube model brings the mechanical simulation to an equilibrium state before adding a new row of triangles. We again rapidly increase the rest length of the springs in successive rows. Unlike the leaf model, however, the triangles in any row are arranged in a ring. This forces the buckling into roughly periodic waves (Figure 7.6).

Thin shells The leaf and flower models use mass-spring systems to find a shape determined by the spring forces: in the former case, a shape in which the edges achieve their assigned rest lengths; in the latter, a shape created as a compromise between rest lengths and rest angles. In both of these cases, we are seeking a shape fulfilling certain constraints and using spring forces to find this shape. We can also use spring forces to find a physically accurate shape given the physical properties of a system. This goal, however, is more problematic, as mass-spring networks are not the approximation to any continuous surface.

Therefore, in order to model physically accurate shapes, I now move to a cell complex in which all cells are taken into account in the simulation. The next model I will present tries to find the equilibrium shape of a single plant cell, modeled as a pressure vessel whose walls are represented as *thin shells*; that is, as (nominally) three-dimensional objects whose thickness is negligible in relation to the other dimensions. Thin shells can therefore be modeled as two-dimensional membranes with resistance to bending (Destuynder 1985). This is similar to the other models in this section, but here both the forces within the surface and the forces opposing bending are more physically accurate than the mass-spring network and ad-hoc angular springs described previously.

Another difference between this model and the previous models is that forces are not computed directly, but are instead derived as the gradient of the total energy of the

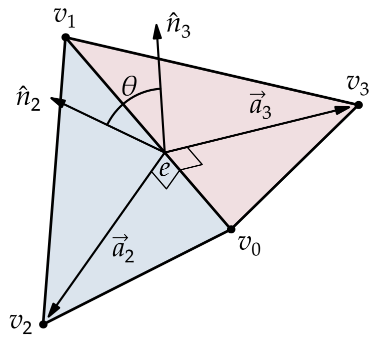


Figure 7.7: The flexural energy along edge e depends on the deformed and undeformed angle θ between the two triangles, the length $||e||$ of the edge, and the average of the heights of the faces ($h_e = 1/6\left(||\vec{a}_2|| + ||\vec{a}_3||\right)$).

system. This simplifies the computation of the forces, especially those due to pressure. The potential energy due to pressure within a volume $\mathscr{V}$ is $U = -P\mathscr{V}$ (Stowe 2007).[1]

Algorithm 7.4 computes the membrane and flexural energies of the thin shell. After initializing the potential energy U to zero in line 2, we compute the flexural energy in a loop over all edges (lines 3–16) and the membrane energy in a loop over all faces (lines 17–30). The flexural energy is computed with the formula derived by Grinspun et al. (2003); their expression for the energy at an edge e is

$$U_e = \kappa\,(\theta_e - \bar{\theta}_e)^2 ||\bar{e}|| \,/\, \bar{h}_e, \tag{7.1}$$

where θ_e is the current angle formed by the two faces adjoining e, $\bar{\theta}_e$ is the rest angle associated with e, $||\bar{e}||$ is the rest length of the edge e, and $\bar{h}_e$ is one third of the average of the heights of the faces (Figure 7.7). To compute this measure, we first find the vertices of the triangles adjoining the edge; v_0 and v_1 are the endpoints of the edge (line 4, while simple flip paths bring us to the other two vertices v_2 and v_3 (lines 5–7). We find the vector $\vec{a}_2$, the altitude of the triangle $\Delta v_0 v_1 v_2$, by projecting the vector $\vec{e}_{02}$ along the side of the triangle onto the vector $\vec{e}_{01}$ along the edge e, then subtracting from $\vec{e}_{02}$ (line 11); $\vec{a}_3$ is found in a similar manner (line 12). We calculate $\bar{h}_e$ (line 13) and the current angle θ (line 14) from these vectors, then increment U by the flexural energy, computed by Equation 7.1 (line 15).

The membrane energy is computed with a method derived by Delingette (2008) based on "biquadratic springs"; the form of the energy is similar to that of a linear spring following Hooke's law, the difference is taken between the *squares* of length and rest length. Delingette (2008) shows that the membrane energy of a flat, deformed triangle is

$$U_f = \sum_{i=1}^{3} k_{fi}\,(\ell_i^2 - \ell_{i0}^2)^2 + \sum_{j \neq i} c_{fij}\,(\ell_i^2 - \ell_{i0}^2)\,(\ell_j^2 - \ell_{j0}^2). \tag{7.2}$$

The constants k_{fi} and c_{fij} depend on the triangle's undeformed shape:

$$k_{fi} = \frac{E}{64\,(1 - \nu^2)}\;\frac{2\cot^2\alpha_i + 1 - \nu}{\mathscr{A}_f} \tag{7.3}$$

$$c_{fij} = \frac{E}{32\,(1 - \nu^2)}\;\frac{2\cot\alpha_i \cot\alpha_j + \nu - 1}{\mathscr{A}_f}, \tag{7.4}$$

[1]This assumes that temperature and pressure remain constant, which is plausible for a plant cell exchanging material with the environment.

Algorithm 7.4 Calculation of thin-shell flexural and membrane energy

```
Require: Vertex $v$ has position $\boldsymbol{P}(v)$; edge $e$ has rest length $L(e)$ and rest angle $\Theta(e)$; face $f$
    has rest area $\mathscr{A}(f)$; the rest angle of face $f$ at vertex $v$ has cotangent $C(f,v)$
 1: procedure ThinShellEnergy
 2:     $U \leftarrow 0$                                                    ▷ initial energy
 3:     for all edge $e$ do                                                 ▷ compute flexural energy
 4:         $\{+v_0, -v_1\} \leftarrow$ boundary($e$)                        ▷ find vertices $\{v_0, v_1, v_2, v_3\}$ near $e$
 5:         $\tau \leftarrow$ tuple containing $e$
 6:         $v_2 \leftarrow \tau$.flip(1).other(0)
 7:         $v_3 \leftarrow \tau$.flip(2, 1).other(0)
 8:         $\vec{e}_{01} \leftarrow \boldsymbol{P}(v_1) - \boldsymbol{P}(v_0)$
 9:         $\vec{e}_{02} \leftarrow \boldsymbol{P}(v_2) - \boldsymbol{P}(v_0)$
10:         $\vec{e}_{03} \leftarrow \boldsymbol{P}(v_3) - \boldsymbol{P}(v_0)$
11:         $\vec{a}_2 \leftarrow \vec{e}_{02}$ − projection of $\vec{e}_{02}$ onto $\vec{e}_{01}$          ▷ triangle altitudes
12:         $\vec{a}_3 \leftarrow \vec{e}_{03}$ − projection of $\vec{e}_{03}$ onto $\vec{e}_{01}$
13:         $\bar{h}_e \leftarrow 1/6 \left(\|\vec{a}_2\| + \|\vec{a}_3\|\right)$
14:         $\theta \leftarrow \pi - \cos^{-1}\left(\frac{\vec{a}_2 \cdot \vec{a}_3}{\|\vec{a}_2\| \, \|\vec{a}_3\|}\right)$
15:         $U \leftarrow U + \kappa \, (\theta - \Theta(e))^2 L(e) / \bar{h}_e$                ▷ flexural energy of $e$
16:     end for
17:     for all triangle $f$ do                                             ▷ compute membrane energy
18:         $\tau \leftarrow$ tuple containing $f$
19:         $\tau_0 \leftarrow \tau$
20:         repeat
21:             $v \leftarrow \tau$.flip(1).other(0)                         ▷ vertex opposite $\tau[1]$
22:             $k \leftarrow \frac{E}{64\,(1-\nu^2)} \, \frac{2\,\left(C(f,v)\right)^2 + 1 - \nu}{\mathscr{A}(f)}$
23:             $\ell \leftarrow$ Measure($\tau[1]$)
24:             $U \leftarrow U + k \, \left(\ell^2 - L(\tau[1])^2\right)^2$
25:             $c \leftarrow \frac{E}{32\,(1-\nu^2)} \, \frac{2\,C(f,v)\,C(f,\tau.\mathbf{other}(0)) + \nu - 1}{\mathscr{A}(f)}$
26:             $\ell' \leftarrow$ Measure($\tau$.other(1))
27:             $U \leftarrow U + c \, \left(\ell^2 - L(\tau[1])^2\right) \left(\ell'^2 - L(\tau.\mathbf{other}(1))^2\right)$
28:             $\tau \leftarrow \tau$.flip(0, 1)
29:         until $\tau = \tau_0$
30:     end for
31:     return U
32: end procedure
```

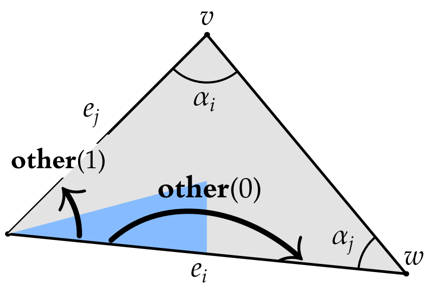


Figure 7.8: The membrane energy of the triangle depends on the rest and deformed lengths of the edges and the undeformed opposite angles. In iteration, the shaded tuple τ has edge e_i; edge e_j is then τ.**other**(1). v, with undeformed angle α_i, is opposite e_i, and is at τ.**flip**(1).**other**(0). w has undeformed angle α_j, is opposite e_j, and is found at τ.**other**(0).

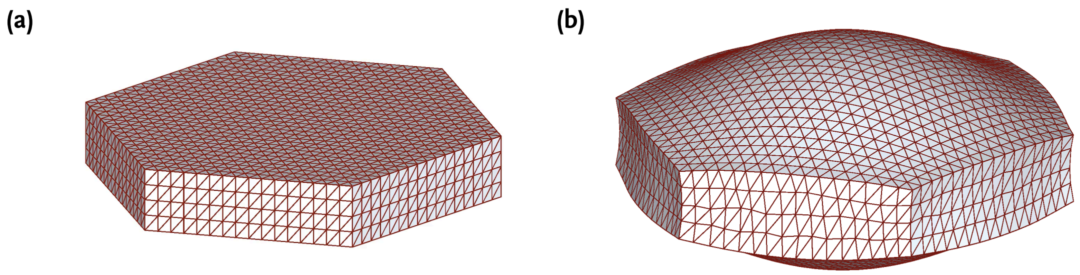


Figure 7.9: Undeformed and equilibrium configuration of a three-dimensional cell-like body. Pressure from within the cell is balanced by forces opposing the deformation of the triangles and the bending of the membrane at the edges.

where $\mathscr{A}_f$ is the undeformed area of the triangle and α_i is the rest angle opposite edge i (Figure 7.8), while E and ν are the Young's modulus and Poisson coefficient of the material, respectively.

The sum over all edges is carried out by iterating a tuple around the triangle (lines 18–29). The edge e_i is the edge of the tuple; the opposite vertex is $v = \tau$.**flip**(1).**other**(0) (line 21). The undeformed area of face f is stored in $\mathscr{A}(f)$, while the cotangent of the undeformed internal angle at v is stored in $C(f, v)$; these are used to compute k according to Equation 7.3 (line 22). The first term of Equation 7.2 is then added to the total potential energy U (line 24). The edge e_j is the **other** edge of the tuple; its opposite vertex is thus the **other** vertex (Figure 7.8). Line 25 computes the coefficient c, line 26 the length of e_j, and line 27 adds the second term of Equation 7.2 to the potential energy.

To the potential energy computed by Algorithm 7.4 we add the energy due to pressure internal to the volume ω,

$$U \leftarrow U - P\ \textsc{Measure}(\omega).$$

We then minimize U to find the equilibrium configuration of the cell; this is done using an external solver from the SUNDIALS suite (Hindmarsh et al. 2005). Figure 7.9a shows the undeformed configuration of a hypothetical hexagonal prism cell; Figure 7.9b shows the minimal energy configuration of this cell.

7.2 Cell expansion and division

A fundamental problem in the modeling of developing organisms is the simulation of cell divisions and growth in a tissue. In the models described in Section 7.1, new mathemati-

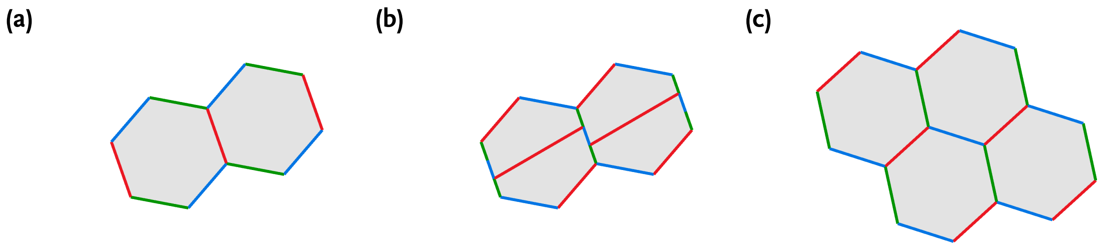


Figure 7.10: Creating the Korn-Spalding pattern with the algorithm of Lindenmayer and Rozenberg (1979). (a) The walls of each hexagonal cell are coloured alternately red, green, and blue. (b) The red walls have each been divided into three walls, coloured green, blue, and green. Other walls have been advanced in colour: green walls become blue, blue walls become red. Each cell has been divided by a red wall. (c) After relaxation, the cells are once more hexagonal.

cal cells were either added marginally (as in the leaf and flower models) or not introduced at all (as in the thin-shell model). In the models described in this section, on the other hand, new cells are created by dividing existing cells. In the first model, division adds new mechanical stresses which result in growth; in the second model, externally-driven growth causes cells to divide when they become too large.

Expansion driven by division The growth pattern of the onion epidermis modeled by Korn and Spalding (1973) has been previously implemented in map L-systems in two and three dimensions (Lindenmayer and Rozenberg 1979; Lindenmayer 1984), cell systems (de Boer, Fracchia, and Prusinkiewicz 1992), and vv (Smith 2006). The CCF version described here follows the algorithm used in the implementation of Lindenmayer and Rozenberg (1979).

The onion epidermis is composed of hexagonal cells. For the purposes of the model, the walls of the cells are coloured either red, green, or blue; the colours are assigned clockwise around each cell (Figure 7.10a). This means that walls of the same colour are opposite each other. At each division step, all red walls are divided into three new walls which are coloured so as to maintain the ordering. Moving clockwise around the cell, the *first* new vertex is one of the endpoints for the wall which splits the cell in two. The colours of the walls are advanced: green walls become blue, while blue walls become red (Figure 7.10b).

Algorithm 7.5 carries out one step of this division. The main body of the procedure is a loop over all red edges (lines 5–27). For each red edge, we start with a tuple τ on the edge (line 7). We want to process all edges in clockwise order, so we make sure the tuple starts at the vertex of e adjoining a blue edge (lines 8–10). We move the tuple off of the edge (line 13) then divide the edge twice, creating three new edges (lines 14–15). We process each of these edges in turn (lines 16–25). For each edge, we advance the tuple (lines 16, 20, and 24) and set its colour (lines 17, 21, and 25). For each of the two new vertices, we must set the position (lines 18 and 22) and register it as the endpoint of a splitting edge. Each vertex is placed in the set *split*(c) corresponding to the cell it splits; the first vertex found while moving clockwise is the endpoint of the edge which splits the **other** cell (line 19), while the second vertex is the endpoint of the edge which will

Algorithm 7.5 Cell division for Korn-Spalding model

```
Require: Vertex $v$ has position $P(v)$; edge $e$ has colour $colour(e)$
procedure KornSpaldingDivision
    for all face $c$ do                                  ▷ clear split vertices
        $split(c) \leftarrow \{\}$
    end for
    for all edge $e$ do                                  ▷ split red edges
        if $colour(e) = red$ then
            $\tau \leftarrow$ tuple containing $e$
            if $colour(\tau.\mathbf{other}(1)) = green$ then
                $\tau \leftarrow \tau.\mathbf{flip}(0)$
            end if
            $p_0 \leftarrow P(\tau[0])$
            $p_1 \leftarrow P(\tau.\mathbf{other}(0))$
            $\tau \leftarrow \tau.\mathbf{flip}(1)$
            $(e_1, v_1, e') \leftarrow \mathbf{splitCell}(e)$
            $(e_2, v_2, e_3) \leftarrow \mathbf{splitCell}(e')$
            $\tau \leftarrow \tau.\mathbf{flip}(1, 0)$
            $colour(\tau[1]) \leftarrow red$
            $P(\tau[0]) \leftarrow 2/3 p_0 + 1/3 p_1$
            $split(\tau.\mathbf{other}(2)) \leftarrow split(\tau.\mathbf{other}(2)) \cup \{\tau[0]\}$
            $\tau \leftarrow \tau.\mathbf{flip}(1, 0)$
            $colour(\tau[1]) \leftarrow green$
            $P(\tau[0]) \leftarrow 1/3 p_0 + 2/3 p_1$
            $split(\tau[2]) \leftarrow split(\tau[2]) \cup \{\tau[0]\}$
            $\tau \leftarrow \tau.\mathbf{flip}(1, 0)$
            $colour(\tau[1]) \leftarrow red$
        end if
    end for
    for all edge $e$ do                                  ▷ advance edge colour
        if $colour(e) = red$ then
            $colour(e) \leftarrow green$
        else if $colour(e) = green$ then
            $colour(e) \leftarrow blue$
        else if $colour(e) = blue$ then
            $colour(e) \leftarrow red$
        end if
    end for
    for all face $c$ do                                  ▷ split faces
        $\{v_0, v_1\} \leftarrow split(c)$
        $(c_1, e, c_2) \leftarrow \mathbf{splitCell}(c, +v_0 \; -v_1)$
        $colour(e) \leftarrow red$
    end for
end procedure
```

split the current cell (line 23).

After splitting each red edge, we recolour the edges (lines 28–36); red edges become green, green edges become blue, and blue edges become red. Finally, we split each face (lines 37–41). The splitting edge runs between the two vertices in *split*(c) (lines 38 and 39) and is coloured red (line 40).

After splitting the old faces all of the new faces have six sides but are no longer regular hexagons. The new regular shape is found using a mass-spring system. Each edge is modeled by a spring with a rest length of 1. To maintain the regular shape, extra forces are added as if there were a spring of rest length 1 between the center of each face and each of its vertices. Algorithm 7.6 performs one timestep of the mass-spring simulation.

Algorithm 7.6 One timestep of mass-spring simulation for the Korn-Spalding pattern

```
Require: Vertices $v$ have position $\boldsymbol{P}(v)$ and velocity $\vec{V}(v)$
 1: procedure KornSpaldingTimeStep
 2:     for all vertex $v$ do                                   ▷ initialize forces to zero
 3:         $\vec{F}(v) \leftarrow 0$
 4:     end for
 5:     for all edge $e$ do                                     ▷ compute edge forces
 6:         $\{+v_0, -v_1\} \leftarrow$ boundary($e$)
 7:         $\vec{e} \leftarrow \boldsymbol{P}(v_1) - \boldsymbol{P}(v_0)$
 8:         $\vec{f} \leftarrow \left(1 - (1 / \|\vec{e}\|)\right) \vec{e}$
 9:         $\vec{F}(v_0) \leftarrow \vec{F}(v_0) + \vec{f}$
10:         $\vec{F}(v_1) \leftarrow \vec{F}(v_1) - \vec{f}$
11:     end for
12:     for all face $f$ do                                     ▷ compute face forces
13:         $\boldsymbol{p}_f \leftarrow$ Centroid($f$)
14:         $\tau \leftarrow$ tuple containing $f$
15:         $\tau_0 \leftarrow \tau$
16:         repeat
17:             $\vec{e} \leftarrow \boldsymbol{P}(\tau[0]) - \boldsymbol{p}_f$
18:             $\vec{F}(\tau[0]) \leftarrow \vec{F}(\tau[0]) + \left(1 - (1 / \|\vec{e}\|)\right) \vec{e}$
19:             $\tau \leftarrow \tau$.flip$(0, 1)$
20:         until $\tau = \tau_0$
21:     end for
22:     for all vertex $v$ do                                   ▷ update positions and velocities
23:         $\vec{V}(v) \leftarrow \vec{V}(v) + \vec{F}(v)\,\Delta t$
24:         $\boldsymbol{P}(v) \leftarrow \boldsymbol{P}(v) + \vec{V}(v)\,\Delta t$
25:     end for
26: end procedure
```

Edge spring forces are computed exactly as in Algorithm 7.1 (lines 5–11). The face spring forces are computed in lines 12–21). The centroid of the cell is found (line 13); this is one of the endpoints of the springs whose forces we are computing. We then iterate

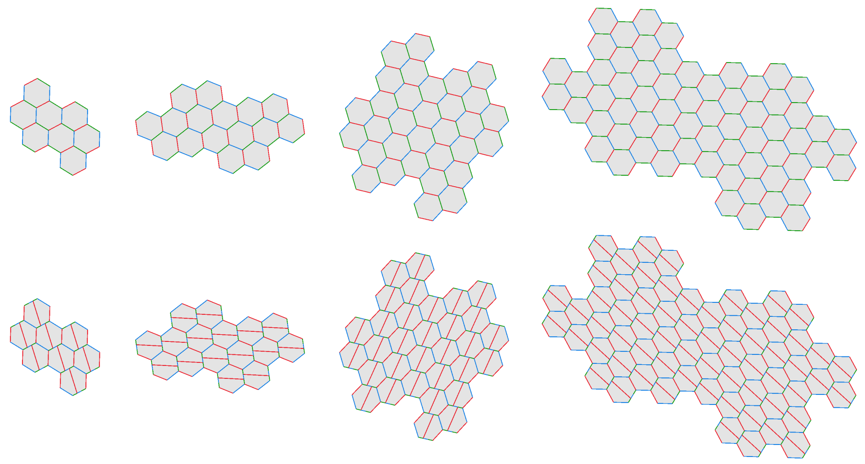


Figure 7.11: Several steps of the development of the Korn-Spalding pattern: from left to right, with eight, sixteen, thirty-two, and sixty-four cells.

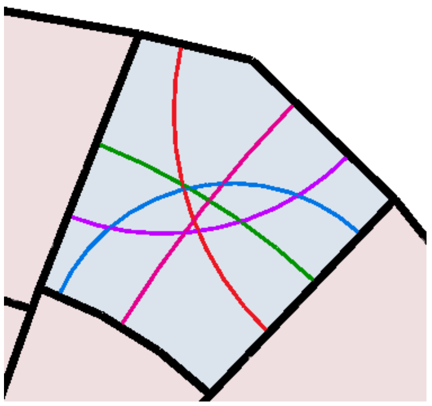


Figure 7.12: A cell and all minimal division walls which divide it into two cells of equal sizes. All of the minimal division walls are circular sections and meet the cell's existing walls at right angles.

over the vertices of the face (lines 14–20). For each vertex, we compute a force for the spring between it and the centroid in exactly the same way as for the edge forces (lines 17 and 18). Finally, once all edge and face forces are computed, the time is advanced (lines 22–25).

Figure 7.11 shows a few steps in the development of the Korn-Spalding pattern implemented using Algorithms 7.5 and 7.6.

Division driven by expansion In the Korn-Spalding model, cells expand because division changes the rest lengths of their walls; that is, division drives expansion. It is also possible to model expansion in a tissue independently of division, and have cell division be driven by this expansion; for example, cells might divide once they exceed some threshold size. The models described in this subsection work in this way.

There are several different models for the shape and positioning of division walls (Besson and Dumais 2011). One of these is the model of Errera (1886), which posits that the division wall takes the same form as a soap film would under the same conditions. In a two dimensional model of a cell tissue, this means that the division wall is a circular section which is perpendicular to the walls it attaches to, and that it is of locally minimal

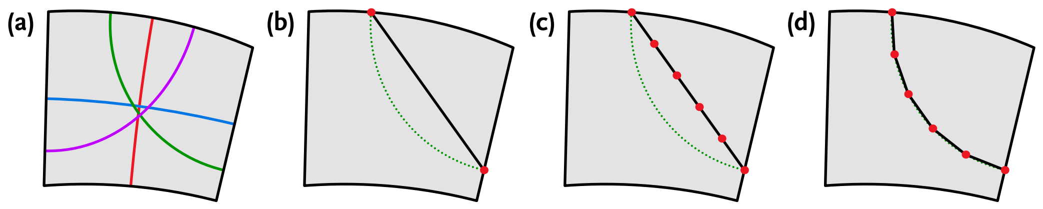


Figure 7.13: A cell is split by a polyline approximation to a circular splitting wall. (a) The candidate dividing walls. (b) A dividing wall (green) has been chosen. The cell is split by a straight edge with the same endpoints. (c) The straight edge is divided into five pieces. (d) The new vertices (red) are moved onto the desired geometry.

length (Besson and Dumais 2011) (Figure 7.12). In order to capture the geometry of a circular section while using a geometric interpretation of cell complexes in which all edges are straight lines, we subdivide each wall into several edges to better fit the circular shape (Figure 7.13).

The distance s between an arc with subtending angle θ and radius r and the chord between its endpoints is $s = r\left(1 - \cos\frac{\theta}{2}\right)$. If we then require that the distance from the true arc to the subdivided edge is less than some ϵ, then we see that the edge must be divided into at least

$$N = \frac{\theta}{2\cos^{-1}\left(1 - \frac{\epsilon}{r}\right)} \tag{7.5}$$

segments. To divide the cell, then, we find all of the circular sections of locally minimal length (Figure 7.13a). Besson and Dumais (2011) discuss how to choose *which* of these curves will form the division wall; it could just be the shortest of the candidate curves, or could be chosen by some stochastic process weighted by the curve lengths. Once the curve has been chosen, the cell is divided by a single edge e with the same endpoints. (Figure 7.13b). The edge e is then subdivided into N parts, where N is computed with Equation 7.5 (Figure 7.13c). Finally, the new vertices are moved onto the desired splitting curve (Figure 7.13d).

Figure 7.14 shows the result of repeated cell division in a tissue. The initial state is a single hexagonal cell, and the largest cell is divided in each step; the shortest candidate curve is used for every division. The figure shows the state of the tissue when there are two, three, four, eight, sixteen, and thirty-two cells.

Many models of plant processes rely on an underlying tissue geometry. I next show an application of this subdividing tissue as an underlying space for the model of auxin transport described by Cieslak, Runions, and Prusinkiewicz (2015). The plant hormone auxin moves between plant cells largely by the action of the protein PIN, which is associated with a cell's membrane and acts to export auxin from the cell to its neighbour. However, PIN is not uniformly distributed across the cell membrane; instead, the concentration on each wall changes over time depending on influx and efflux across that wall. The flow of auxin and the redistribution of PIN can thus form patterns on the tissue, from flow channels to evenly-spaced peaks.

Recall the discussion of diffusion from Chapter 1. The change of auxin concentration a in a cell i is the sum of the *influxes* to the cell, minus the sum of the *effluxes* from

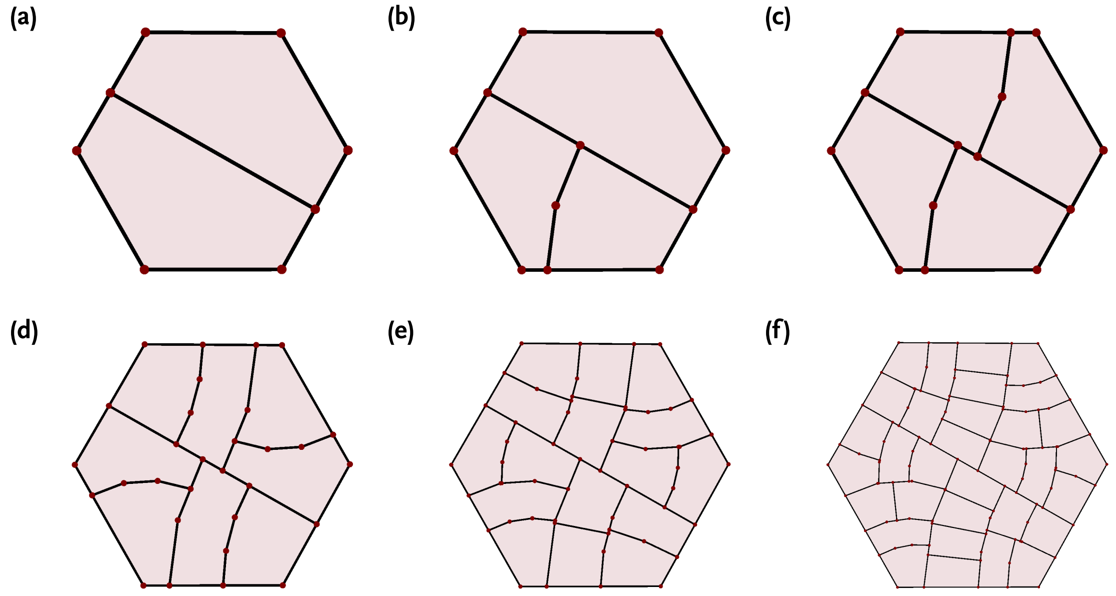


Figure 7.14: Simulation of cell division by the Errera criterion. In each step, the largest cell is divided into two cells of equal sizes. (a) The initial hexagonal cell is dived into two. (b) One of the child cells is divided. (c) The other initial child cell is divided, producing a tissue of four cells. (d)–(f) The state of the tissue with eight, sixteen, and thirty-two cells.

the cell. In one dimension, the cell has two neighbours with indexes $i-1$ and $i+1$, and the change of auxin concentration a_i is

$$\begin{aligned}\frac{da_i}{dt} &= E_{i-1\to i} + E_{i+1\to i} - E_{i\to i-1} - E_{i\to i+1}\\ &= Ta_{i-1}P_{i-1\to i} + Ta_{i+1}P_{i+1\to i} - Ta_iP_{i\to i-1} - Ta_iP_{i\to i+1},\end{aligned}$$

where T is the *transport* coefficient and $P_{i\to j}$ is the concentration of PIN on the membrane of cell i which separates it from cell j. In the case of diffusion, we could assign the two variables *concentration* and *flux* to cells and walls, respectively; but where does the PIN concentration P fit? P is not a property of a cell, as it may differ on different membranes; nor is it the property of a wall, as it may differ in the cells on either side.

We can still find a place to record the PIN concentration P, but we have to change the topology of the cellular representation. Instead of a string of cells separated by walls

$$\cdots \textbf{\textit{Wall Cell Wall Cell Wall}} \cdots,$$

we have a string of cells and their associated membranes

Membrane Cell Membrane

separated by *intercellular spaces*:

$$\cdots \textbf{\textit{Intercell Membrane Cell Membrane Intercell Membrane Cell Membrane Intercell}} \cdots.$$

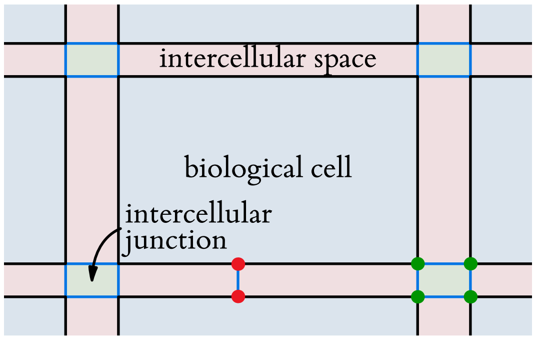


Figure 7.15: The structure of a two-dimensional tissue with intercellular space. Biological cells (blue) are bounded by cell membranes (black), which separate them from intercellular space (pink). Intercellular spaces meet at intercellular junctions (green). Vertices joined by intercellular edges (blue) have the same position but have been moved apart for visualization; thus, the position of the red vertices is the same, while all four green vertices have the same position.

Then the PIN concentrations are the property of the *membranes*, while the intercellular spaces take the place of walls in the previous model, carrying the net flux between two cells:

$$\cdots \textbf{\textit{Intercell}}(J)\ \textbf{\textit{Membrane}}(P)\ \textbf{\textit{Cell}}(a)\ \textbf{\textit{Membrane}}(P)\ \textbf{\textit{Intercell}}(J) \cdots .$$

The productions corresponding to 1.8a and 1.8b are then clearly

$$\textbf{\textit{Cell}}(a_L)\ \textbf{\textit{Membrane}}(P_L) < \textbf{\textit{Intercell}}(J) > \textbf{\textit{Membrane}}(P_R)\ \textbf{\textit{Cell}}(c_R) \rightarrow \\ \textbf{\textit{Intercell}}\,(T\,(a_L P_L - a_R P_R)) \quad (7.6a)$$

$$\textbf{\textit{Intercell}}(J_L)\ \textbf{\textit{Membrane}}(P_L) < \textbf{\textit{Cell}}(a) > \textbf{\textit{Membrane}}(P_R)\ \textbf{\textit{Intercell}}(J_R) \rightarrow \\ \textbf{\textit{Cell}}\,(a + \Delta t\,(J_L - J_R))\,. \quad (7.6b)$$

There is also a production for ***Membrane*** which alters the PIN concentration based on the flux and total PIN in the cell:

$$\textbf{\textit{Intercell}}(J) < \textbf{\textit{Membrane}}(P_L) > \textbf{\textit{Cell}}(a)\ \textbf{\textit{Membrane}}(P_R) \rightarrow \\ \textbf{\textit{Membrane}}\left(P_L + \Delta t\left((\sigma_{PQ} J^2 + \sigma_P)(1 - (P_R + P_L)) - \mu_P P_L\right)\right), \quad (7.6c)$$

where $(1 - (P_R + P_L))$ is the amount of PIN in the cell unassociated with either membrane. (There is a nearly identical production for the *right* membranes. This is once again a problem with the inherent left-right orientation of a string.)

We can extend this structure to two dimensions. In this case, the intercellular spaces are 2-cells; we also have a new kind of space, the *intercellular junction*, where more than two intercellular spaces meet (Figure 7.15). 1-cells are either cell membranes or *intercellular edges* between intercellular space and intercellular junctions. 0-cells are all on the boundaries of biological cells and still carry position information. We can choose how large the intercellular spaces are: for the purposes of the simulation described in this section, the spaces are of *zero thickness*. This means that vertices joined by intercellular edges have the same position.

In order to divide cells in a cellular structure with intercellular space, we have to change how splitting operations work. The operation which splits a 1-cell must now also split the adjacent intercellular space and the 1-cell on the other side of that space

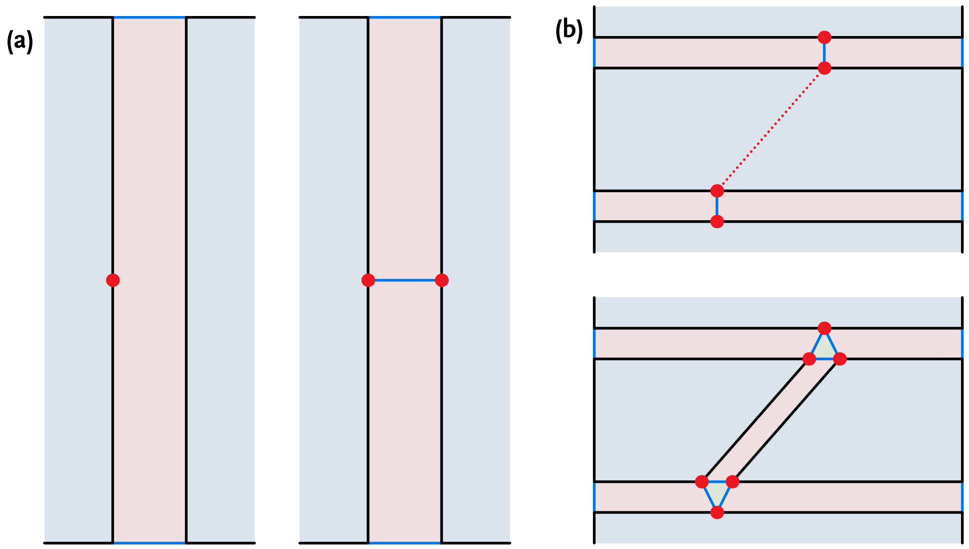


Figure 7.16: Splitting operations with intercellular space. (a) When splitting an edge, we also divide the adjacent intercellular space and the edge on its opposite side. (b) When splitting a face, we insert intercellular space between the two child faces and create intercellular junctions at the endpoints of the splitting wall.

(Figure 7.16a). The operation to split a 2-cell must insert intercellular space between the two child cells and add intercellular junctions to the intercellular spaces at the endpoints (Figure 7.16b). Algorithms 7.7 and 7.8 implement these augmented operations.

To split an edge and its adjoining intercellular space (Algorithm 7.7), we start with a cell tuple τ on that edge (line 2) and ensure that its 2-cell is the intercellular space (lines 3–5). We can then iterate around the rectangular intercellular space (line 7) to reach the opposite edge.

We now split both the original edge e (line 8) and the opposite edge (line 11). Each of the new vertices is placed at the supplied position $\vec{p}$ (lines 9 and 12) and its vertex type is set to INTERCELL; this means that the vertex is adjacent only to intercellular spaces, and not to intercellular junctions. We assume that the children after a **splitCell** operation are set to the same type as the parent.

Finally, we split the intercellular space itself between v and v' (line 14). The splitting edge is of type INTERCELL as well; this means that it separates intercellular spaces and junctions, and is not adjacent to an actual cell.

Splitting a 2-cell adjacent to intercellular space (Algorithm 7.8) is illustrated in Figure 7.17. We first split the face by an edge with the indicated endpoints (line 2, Figure 7.17a); the splitting edge will become a cell MEMBRANE (line 3). The endpoints must then be doubled; we do this by splitting the edges incident to the vertices and to the child face f' (line 6, Figure 7.17b). These new vertices will be on the other side of the new intercellular space from the old vertices, so their positions will be the same (line 7). The face f' is then split between the new vertices (line 23, Figure 7.17d). and the cell types are set: the splitting edge is the MEMBRANE of cell f_R (line 24), while the space between f_L and f_R is an INTERCELL (line 25).

Most of Algorithm 7.8 deals with the changes to the intercellular space required

Algorithm 7.7 Splitting an edge in a structure with intercellular space

```
Require: Vertices $v$ have position $P(v)$; edges and faces have type $kind$
 1: procedure IntercellEdgeSplit($e$,$\vec{p}$)
 2:     $\tau \leftarrow$ tuple containing $e$
 3:     if $kind(\tau[2]) \neq$ INTERCELL then
 4:         $\tau \leftarrow \tau$.flip(2)
 5:     end if
 6:     $c \leftarrow \tau[2]$
 7:     $\tau \leftarrow \tau$.flip(1,0,1)
 8:     $(e_L, v, e_R) \leftarrow$ splitCell($e$)
 9:     $P(v) \leftarrow \vec{p}$
10:     $kind(v) \leftarrow$ INTERCELL
11:     $(e'_L, v', e'_R) \leftarrow$ splitCell($\tau[1]$)
12:     $P(v') \leftarrow \vec{p}$
13:     $kind(v') \leftarrow$ INTERCELL
14:     $(c_L, \epsilon, c_R) \leftarrow$ splitCell($c, +v\ -v'$)
15:     $kind(\epsilon) \leftarrow$ INTERCELL
16: end procedure
```

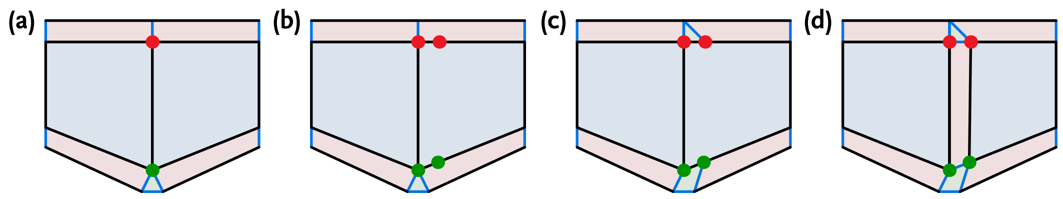


Figure 7.17: The steps taken to split a face surrounded by intercellular space. (a) The face is split once between the indicated endpoints. (b) The edges incident to one side of the splitting edge are split again to create new split vertices (red and green). (c) Intercellular junctions are created (top) or widened (bottom) to accommodate the new vertices. (d) The right-hand face is split again between the new vertices and the area to the left is designated intercellular space.

Algorithm 7.8 Splitting a face in a structure with intercellular space

```
Require: Vertices $v$ have position $\boldsymbol{P}(v)$; edges and faces have type $kind$
 1: procedure IntercellFaceSplit($f$,$+v_0 \;\; - v_1$)
 2:     $(f_L, e, f') \leftarrow$ splitCell($f$, $+v_0 \;\; - v_1$)
 3:     $kind(e) \leftarrow$ MEMBRANE
 4:     for $v \in \{v_0, v_1\}$ do
 5:         $\tau \leftarrow$ tuple containing $\{v, e, f'\}$
 6:         $(\ell_L, v', \ell_R) \leftarrow$ splitCell($\tau$.other(1))
 7:         $\boldsymbol{P}(v') \leftarrow \boldsymbol{P}(v)$
 8:         $\tau \leftarrow \tau$.flip(1, 2)
 9:         $kind(\tau[1]) \leftarrow$ INTERCELL
10:         $w \leftarrow \tau$.flip(1).other(0)
11:         $(c_L, \epsilon, c_R) \leftarrow$ splitCell($\tau[2]$, $+v' \;\; - w$)
12:         $kind(\epsilon) \leftarrow$ INTERCELL
13:         if $kind$(v) = INTERCELL then
14:             $kind$(join($v, \epsilon$)) $\leftarrow$ ICJUNCTION
15:         else if $kind$(v) = ICJUNCTION then
16:             $j \leftarrow$ mergeCell(join($v, w$))
17:             $kind(j) \leftarrow$ ICJUNCTION
18:         end if
19:         $kind(v) \leftarrow$ ICJUNCTION
20:         $kind(v') \leftarrow$ ICJUNCTION
21:         $kind(w) \leftarrow$ ICJUNCTION
22:     end for
23:     $(c, e', f_R) \leftarrow$ splitCell($f'$, $+v'_0 \;\; - v'_1$)
24:     $kind(e') \leftarrow$ MEMBRANE
25:     $kind(c) \leftarrow$ INTERCELL
26: end procedure
```

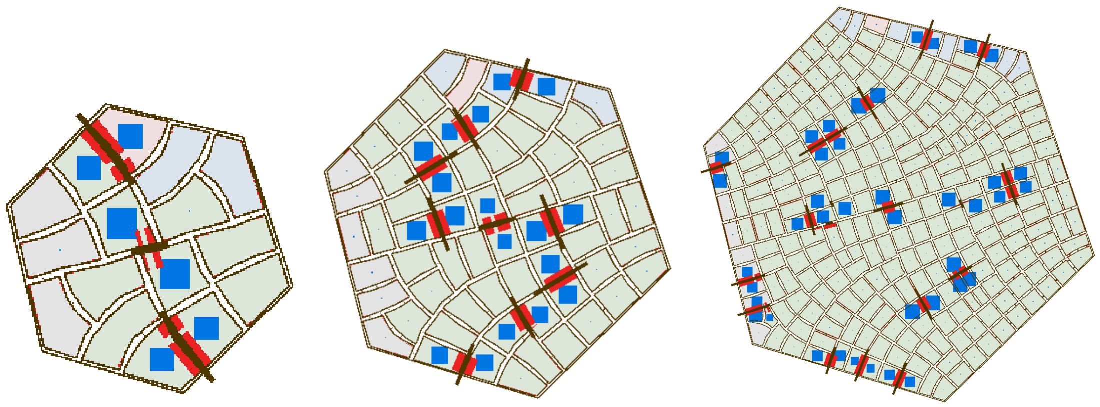


Figure 7.18: Steps from an auxin transport simulation on a growing tissue. Auxin concentration in cells is represented by a proportionally-sized blue square, while pin concentration is represented by the thickness of a red bar along the membrane. Auxin peaks form early and are distributed throughout the tissue as it grows.

when we double the endpoints. The loop in lines 4 to 22 handles each of the endpoints in turn. After creating the new vertex v', we set the type of the edge between it and v to INTERCELL (line 9). We then find w, the vertex corresponding to v on the other side of the old intercellular space (line 10). The intercellular space is then split between v' and w (line 11); the space shared by v, v', and w will become an intercellular junction. If there was no pre-existing junction at v (in which case v's type was INTERCELL), then we simply set the type of the new triangular face to ICJUNCTION (lines 13–14). On the other hand, if v was already on an intercellular junction, the new junction must be merged into it. This is done by mergin along the edge between v and w (line 16); the newly merged cell is then set to type ICJUNCTION (line 17). Finally, the three vertices v, v' and w are set to type ICJUNCTION (lines 19–21).

We bring all of these pieces together in the simulation shown in Figure 7.18. A tissue with intercellular space is grown by scaling all positions uniformly at each time step. We use the MEASURE procedure (Algorithm 5.7) to calculate each cell's area, and if it reaches a threshold area, the cell is divided according to Errera's criterion along a polyline approximating a circular section; the division uses INTERCELLEDGESPLIT and INTERCELLFACESPLIT in place of default **splitCell** operations. During expansion, a chemical system, the two-dimensional equivalent of productions 7.6, is simulated on the growing tissue, using differential equation solvers from the SUNDIALS suite (Hindmarsh et al. 2005). As a cell expands, the concentration of auxin in each cell and of PIN on each wall remains constant. When a wall divides, the concentration of PIN on the two child walls is the same as on the parent; when a cell divides, the concentration of auxin in the children is the same as in the parent, while the initial concentration of PIN on the new wall is assumed to be zero.

In Figure 7.18, the auxin concentration in a cell is visualized by a proportionally-sized blue square, while the PIN concentration in a membrane is visualized by the thickness of a red line along that edge. Auxin peaks initially form in the top left, center, and bottom right of the tissue, and more peaks join them as the tissue grows.

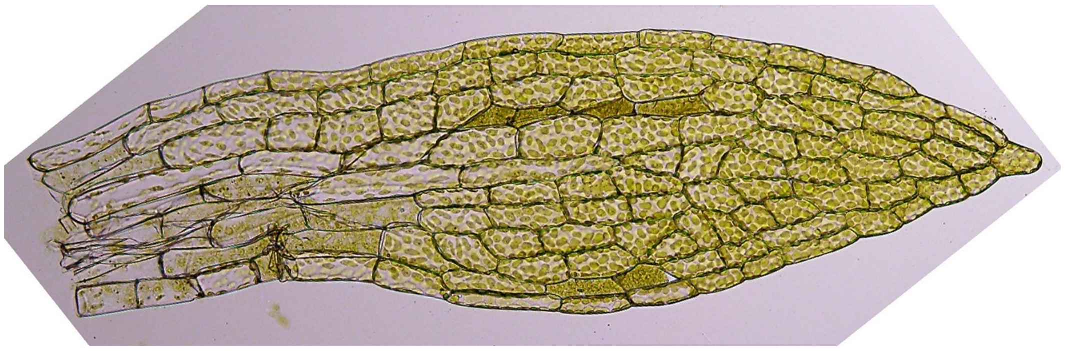


Figure 7.19: Microphotograph of a leaf of *Physcomitrella patens*. Note the rectangular cells in staggered files, narrower cells on the margin, and long, straight cells at the middle of the cell. (Courtesy of E. Barker)

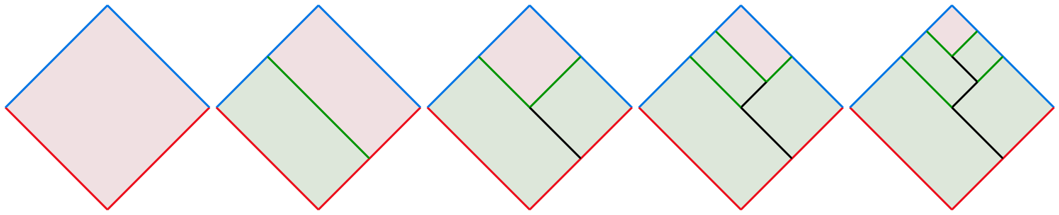


Figure 7.20: The first cell divisions of the leaf of *P. patens*. The apical cell (pink) divides alternately to the left and right, producing a herringbone pattern in the initial cells.

7.3 The leaf and apex of *Physcomitrella patens*

The last family of models I will discuss in this section aims to capture the growth and form of a specific plant: the moss *Physcomitrella patens*. The first of these is a model of the growth of leaves, with focus on the cell division patterns that lead to the emergence of parallel cell files in the mature leaf. This model is based on the work of Chelladurai (2012), which was originally done in the vertex-vertex modeling system vve. The second model in this chapter is a geometric model of the growing shoot apex of *P. patens*. The apex displays a very interesting three-dimensional structure which naturally lends itself to, and thus provides a good example of, modeling using three-dimensional cell complexes.

Leaf model The *Physcomitrella* leaf model is based on the work of Chelladurai (2012), which aimed to reproduce the cell patterns seen in microphotographs (Figure 7.19) by growing the leaf from a single cell, expanding cells through a mechanical (mass-spring) simulation. The cells seen in Figure 7.19 are rectangular, and appear in staggered files. This is puzzling, because early division patterns produce cell walls at 45° to the leaf axis (Figure 7.20). Further divisions then produce four-way cellular junctions (Figure 7.21a).

Chelladurai showed that the four-way junctions could be manipulated to create exactly the rectangular cell patterns needed. This is done by *expanding* these junctions, inserting a new horizontal edge to divide them into two three-way junctions and turn the internal cells into hexagons (Figure 7.21b). We add bending springs with 90° rest angles at the top and bottom vertices of these cells, and springs with 180° rest angles at the

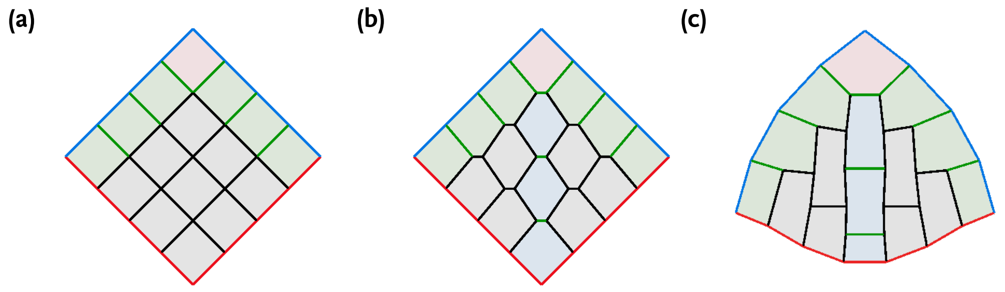


Figure 7.21: The model of early cell divisions in *Physcomitrella*. (a) Dividing the initial apex many times creates cells which meet at four-way junctions. (b) The cells can be made into hexagonal cells by inserting new horizontal edges at each of the four-way junctions. (c) By adding bending springs and relaxing the mass-spring system, a leaf blade with staggered files of rectangular cells is created.

side vertices. Relaxing the mass-spring system then pushes the leaf shape into staggered files of rectangular cells (Figure 7.21c).

Now the leaf blade expands. This expansion is driven by an increase in the rest length of the cell walls. We want the leaf to expand more in the apical-distal (vertical) direction than in the medio-lateral (horizontal) direction so cell walls aligned vertically see a greater increase in rest length than cell walls aligned horizontally. The vertical expansion factor is uniform, but the horizontal expansion factor is greater about halfway up the leaf (Chelladurai 2012). The growth rates are interpolated between the horizontal and vertical extremes, depending on the wall's orientation.

Two kinds of wall do not expand. First, the leaf base (red edges in Figure 7.21c) will not expand, as in a real plant it is attached to the stem. Second, the horizontal edges adjacent to the margin and in the emergent midrib have a fixed, relatively small rest length. This fixed length results in cells on the margin and at the midrib being narrower than other cells, just as in the real cell pattern (Figure 7.19).

As the cells expand, they eventually reach a threshold area and divide. This threshold area depends on the vertical position of the cell: cells near the bottom of the leaf blade are much larger, which indicates that their division area must have been relatively larger; the cells near the leaf apex are smaller and have a lower threshold division area. When a cell divides and creates four-way junctions, an edge is inserted into them. In contrast to the apical part shown in Figure 7.21b, the inserted edges in the lower areas of the leaf are oriented vertically (pink walls in Figure 7.22cd). These vertical edges result in staggered files of cells in the leaf.

Figure 7.22 shows the output of a Cell Complex Framework implementation of Chelladurai's *Physcomitrella* leaf model. This model uses the same growth pattern as that described by Chelladurai (2015). The CCF implementation combines the mass-spring dynamics described in Section 7.1, using angular springs which are attached to faces instead of edges, with cell divisions determined by the method discussed in Section 7.2. Cell sizes are calculated using the Measure algorithm shown in Chapter 5. Expansion is simulated with the help of an external solver from the SUNDIALS suite (Hindmarsh et al. 2005).

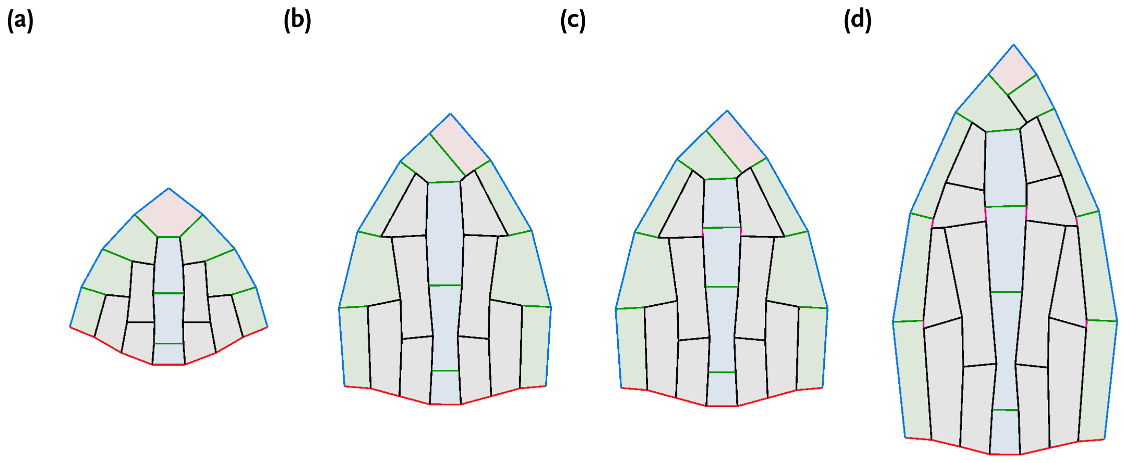


Figure 7.22: The development of the model of the *Physcomitrella* leaf. Wall divisions which result in four-way junctions are split by a vertical edge (pink), maintaining the staggered files of cells. Cells on the margin (green) and at the midleaf (blue) are narrower, while cells near the top of the leaf are still dominated by the apex's diagonal division pattern.

Apex model The shoot apex of *Physcomitrella* also displays an interesting growth pattern (Harrison et al. 2009). A single apical cell undergoes successive divisions in a spiral pattern, cleaving off new leaf initials surrounding it. The spiral division pattern leaves the apical cell tetrahedral in shape (Figure 7.23). This three-dimensional division pattern is difficult to visualize, so the Cell Complex Framework was used to create a descriptive model replicating the pattern.

The bud is modeled as a dome-shaped volume (Figure 7.24a), created by Catmull-Clark subdivision (Section 6.1) of a hexagonal prism. Divisions are performed by splitting the apical cell at its intersection with a plane (Section 5.2). All of the splitting planes are inclined from the vertical, though this is only visible for the first two splits (Figures 7.24ab). Successive splitting planes are rotated by a phyllotactic angle of 137.5° about

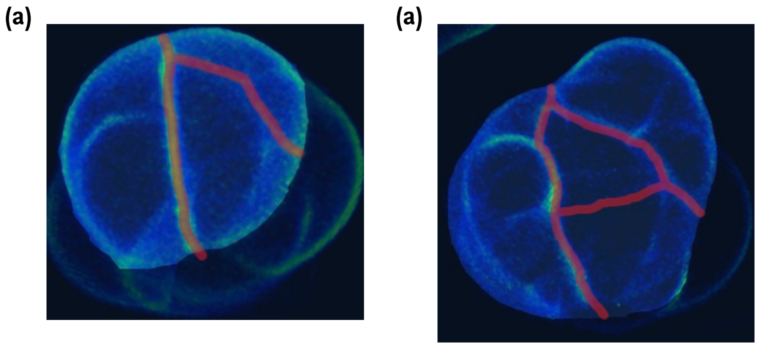


Figure 7.23: The development of the shoot apex of *P. patens*. (a) The apical cell (lower right) has divided to create two leaf cells (left and upper right). (b) After a third division, the apical call (center) is tetrahedral in shape and surrounded by initial leaf cells. (Courtesy of J. Harrison)

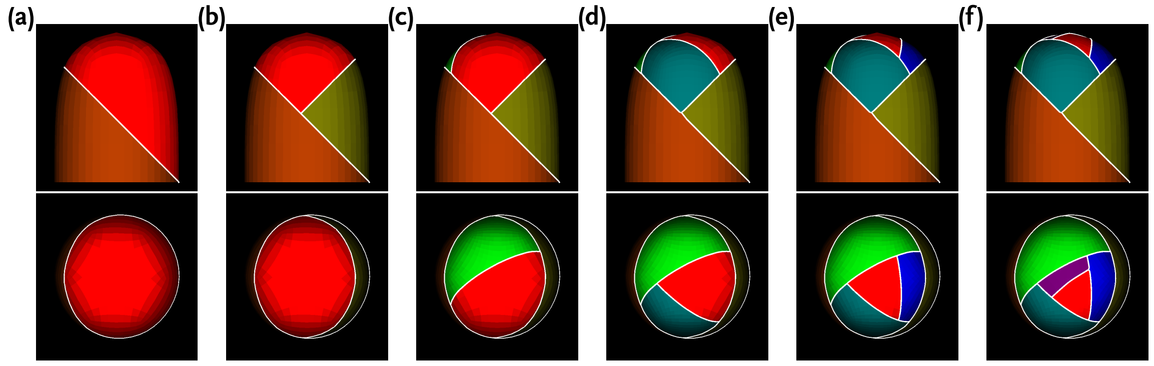


Figure 7.24: Division of the apical cell (red) in the model of the shoot apex of *Physcomitrella*. All of the division planes are inclined from the vertical, and successive divisions are rotated by 137.5°, leaving a tetrahedral apical cell surrounded by leaf initials.

the vertical axis, creating a spiral arrangement of leaf initials and a tetrahedral apical cell (Figures 7.24c–f).

We can better appreciate the internal structure of the apex if we render some of the cells as partly transparent (Figure 7.25). The oblique splitting planes that create wedge-shaped leaf initials and a tetrahedral apical cell are then clearly visible.

The presented apex model can serve as a foundation for other models. For instance, Figure 7.26 shows the result of adding a descriptive model of the early development of *Physcomitrella* leaves. The upper surfaces of each leaf initial bulges upward as the leaf apical cell divides in the characteristic herringbone pattern (Figure 7.20). As the leaves expand, the apex becomes surrounded by young growing leaves.

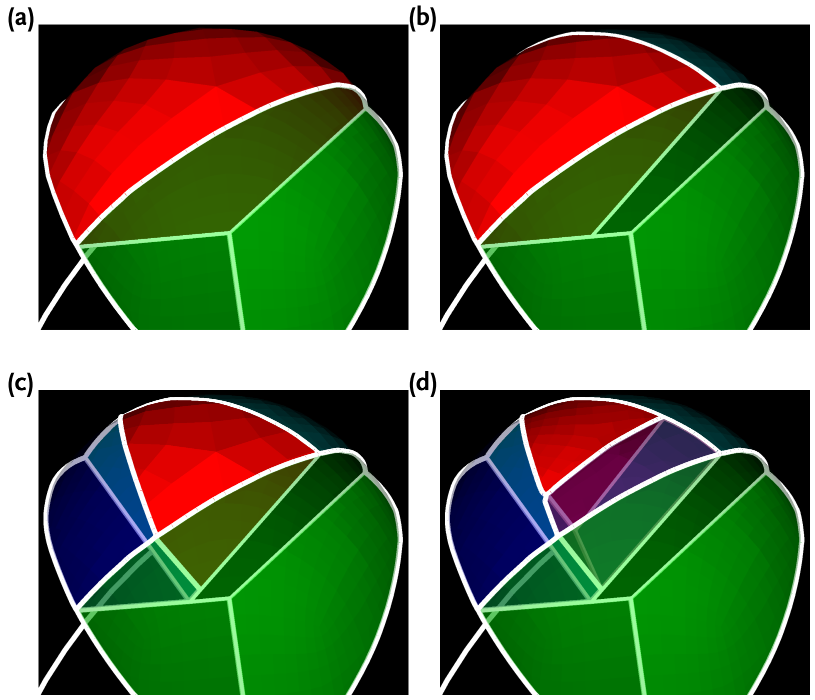


Figure 7.25: Development of the *Physcomitrella* apex model, rendered with transparent cells to reveal internal structure. Panel (a) corresponds to Figure 7.24c; (b) to 7.24d; (c) to 7.24e; and (d) to 7.24f. Note the progression of the apical cell (red) to a tetrahedral cell surrounded by leaf initials.

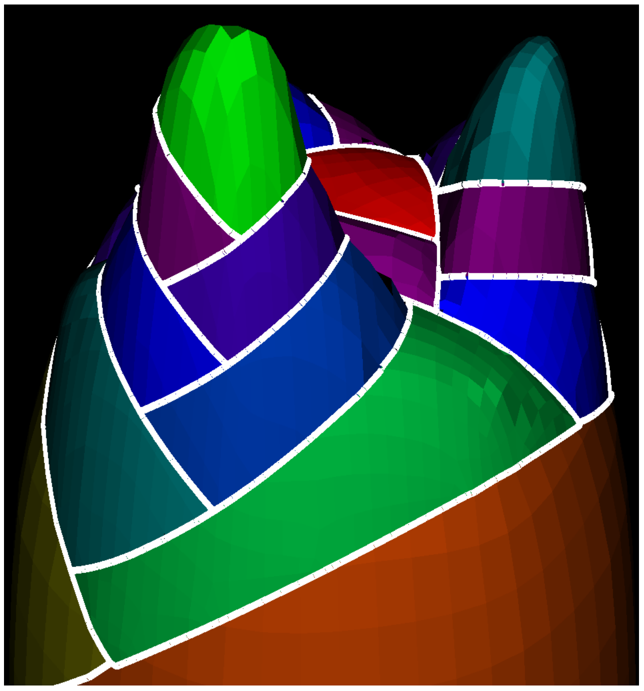


Figure 7.26: Early leaf development on the *Physcomitrella* apex. The leaf initials have divided in a herringbone pattern and surround the apex.

8 Conclusions

Embracing cell complexes as an underlying representation for a modeled problem has many advantages. They are a natural representation for a subdivided manifold, and supply a discrete topological structure to space. This structure includes cells of all dimensions, which provides placeholders for physical quantities of different inherent dimensionality. The cell complex structure is locally defined: a cell is defined in terms of the cells in its boundary. This means that operations which modify the topology are local in nature. In addition, the topological relations in the cell complex provide a local context for changing the parameters of individual cells. This locality of both topological operations and parameter modifications mans that the behaviour of any one cell depends only on its own history and its interaction with its neighbours. In other words, cells are *modules*, so problems can be modeled taking cells into account in a modular fashion.

I have systematically investigated the use of cell complexes as a framework for developmental modeling in one, two, and three dimensions. I created a new computational representation for a cell complex — the *flip table*. This single representation provides information on incidence relations between cells, adjacency relations between cells, and relative orientations between cells. I used this representation to design and implement the Cell Complex Framework, a modeling system built on cell complexes. I used the CCF to construct models illustrating both programming techniques and fundamental applications of cell complexes in both geometric modeling and biological modeling. The CCF is now ready for practical applications.

The Cell Complex Framework The Cell Complex Framework is implemented as a C++ API, offering the user access to the cell complex both through the low-level representation as flips, and through higher-level operations. The CCF incorporates the basic operations described in Chapter 4, chosen because, while by no means exhaustive, they allowed me to implement the wide range of models described in later chapters.

Operations which modify the cell complex work by modifying the flips which include the cells which are altered. Since a flip only refers to adjacent and incident cells, the operations are thus local. Moreover, since each flip only refers to cells of nearby dimension, operations are also *dimensionally* local; the splitting of an n-cell only impacts flips involving cells down to dimension $(n-3)$.

Also vital to the locality of CCF operations is the *cell tuple*, which provides a "handle" to the cell complex. Modifying cell tuples with **flip** operations is a useful way to move across the cell complex or to rotate through a number of cells in order.

Tuple coordinates Another concept introduced in this thesis that I think is extremely useful is *tuple coordinates* (Chapter 5). Each cell tuple in the cell complex has an associated coordinate frame $(\boldsymbol{p}, \vec{e}_1, \vec{e}_2, \ldots)$, where $\boldsymbol{p}$ is the position of the tuple's vertex, $\vec{e}_1$ is the direction of the tuple's edge, $\vec{e}_2$ is the direction of the other edge in the tuple's face, and so on. There is also an orthonormal coordinate frame $(\boldsymbol{p}, \hat{e}_1, \hat{e}_2, \ldots)$ which can be derived from the tuple frame.

Tuple coordinates are used to great effect in Algorithms 5.7, 5.8, and 6.8. The first two algorithms use the orthonormal tuple frame to compute the measure and centroid of a cell as the sum of a simpler expression over all of the tuples containing that cell. This is a general solution to finding the measure and centroid of arbitrary (non-intersecting) n-dimensional polytopes which I have not been able to find in the published literature. While the expression for the centroid is somewhat more complex than that for measure, both operations are extremely useful in further modeling.

Algorithm 6.8, on the other hand, propagates a geodesic across an arbitrary triangulated surface by associating each flip operation with a transformation between tuple coordinate frames. In this way, a sequence of flips which follows the geodesic is coupled with the coordinate transformation relating the original direction of the geodesic to the coordinates in the current cell. There is nothing inherently two-dimensional about this result, which could equally well be applied to 3-cells or 4-cells in a curved spacetime.

Comparing the CCF to vv The inspiration for creating a framework based on cell complexes was the modeling system vv (Smith, Prusinkiewicz, and Samavati 2003) and its extension VVE (Barbier de Reuille 2008). Both systems have been widely used to create published models of multicellular structures. When compared to the CCF, vv has several advantages. Without so many cells to take into account, operations are simpler and faster to execute. Further, vv exists as a language extension to C++. This means that it includes language constructs that allow some topological operations, such as iterating around a neighbourhood, to be expressed more intuitively than calls to an API.

There are also shortcomings of vv which the CCF overcomes. The first of these is the representation of cells other than vertices. In vv, only vertices have a true topological representation. Data stored on edges is associated instead with a pair of vertices, and faces have no representation whatever in the basic system. A representation for faces can be added by, for instance, adding in a new "face" vertex which is in the neighbourhoods of all of the vertices of the face, but this ad-hoc solution confounds the topology and increases the complexity of operations. The lack of representation for faces does not critically hamper vv in all cases, of course: some of the models described in this thesis only use the vertex-edge structure and store no parameters on faces. For many models, however, the lack of faces is a serious problem.

The second problem is that vv inherently relies on an orientation of two-dimensional space, in the same way as L-systems have an inherent difference between "left" and "right". This reliance on absolute orientations is the reason for vv's greatest limitation:

it is *inherently two-dimensional*. The CCF was explicitly designed to create an alternative framework that works in any number of dimensions.

Comparing the CCF to MGS A developmental modeling system that operates on three-dimensional cell complexes is MGS (Giavitto and Michel 2001). MGS is a language with declarative semantics: a rule is defined by a pattern of incidences and adjacencies, and a specification of how cells that match should be altered. The declarative style makes it possible to specify some manipulations more cleanly than the CCF's imperative style. However, the pattern specification in MGS quickly grows very verbose. Furthermore, searching the cell complex for a match to the pattern can be quite slow. It is unclear whether a more expressive pattern syntax would make declarative semantics more attractive; it is my hope that the CCF may be extended in this direction efficiently.

Comparing the CCF to G-maps The recent book by Damiand and Lienhardt (2014) describes in detail the implementation of a modeling framework built on G-maps. It is likely that low-level CCF and G-map operations are of similar time and space complexity, as there seem to be similar numbers of flips and darts in a cell complex. One advantage of the flip table representation is that operations are dimensionally local; however, this advantage is only prominent in higher dimensions. It would be enlightening to attempt a direct comparison between the formalisms.

8.1 Future directions

My goal is to make the Cell Complex Framework as useful for modelers as possible. I can see a few key directions for improvement with this in mind. The first direction is to investigate improvements to the speed of the system. The second would be to enhance the set of basic topological operations. The third is to simplify and make more intuitive the construction of models by introducing new predefined functions or language constructs.

Efficiency Except for the most basic time profiling (Chapter 4), little work has been done investigating the speed of the CCF. Faster implementations of the basic data structures are only a start. Can CCF operations be readily parallelized, using multicore machines or GPUs? Rather than offloading differential equations to an external solver, might more efficient solvers be created which use the topology of the existing cell complex? How does the CCF compare in terms of efficiency to other similarly powerful systems, particularly G-maps?

New topological operations The current set of topological operations, while sufficient for almost all of the models described in this thesis, is not complete. The standard topological operations **glue** and **cut** are not currently included. These operations are very useful for changing the genus of a manifold: for turning a plane into a cylinder, for example, or, in biology, for the process of gastrulation.

The model of the leaf of *Physcomitrella patens* (Chapter 7) used an operation **insertEdge** which separates a vertex into two vertices by inserting an edge between them.

This operation is also not in the CCF, and had to be written in terms of low-level flips. However, this operation is *dual* to the **splitCell** operation applied to a face, so the flip implementation of **insertEdge** was simply a matter of copying the implementation of **splitCell**, then reversing each flip into its dual. Presumably the same could be done to **mergeCell** to create **collapseEdge**, but would it also be useful to allow access to the dual complex more generally? That is, would a "filter" of sorts, which allows the dual complex to be accessed and modified as if it were the true complex, be useful as a modeling tool?

Higher-level constructs Language-level abstractions could also be useful. Looking at the models implemented in this thesis, one sees that the pattern of iterating around a loop of tuples by flipping in adjacent dimensions is used frequently, to iterate through all neighbours of a vertex, all vertices of a face, or all faces incident to an edge. This iteration always has a form similar to:

```
$\tau \leftarrow$ some tuple
$\tau_0 \leftarrow \tau$
repeat
    do something with the tuple
    $\tau \leftarrow \tau.\mathbf{flip}(0,1)$
until $\tau = \tau_0$
```

It may be possible to abstract out this loop into higher-level syntax, such as something of the form

```
perform_iteration(τ,flip(0,1)) {
  do something with τ
}
```

Broader patterns might also be used as higher-level abstractions. For example, manipulating a two-dimensional tissue involves many of the same operations: expanding uniformly, or according to some growth tensor; dividing a cell with a wall determined by Errera's rule; or diffusing a substance to neighbouring cells. These could be abstracted into a library for two-dimensional tissues. If this were done, the tissue itself might be replaced at some point by a tissue with intercellular space (Chapter 7), while the model code, using the library functions, would remain virtually identical.

At a higher level still, it may be desirable to turn the API into an embedded language like VV. This would let some of the more cumbersome syntax, like the face iteration above, become easier to use. A more expressive language might even replace the imperative semantics with declarative semantics. There are still problems with defining neighbourhoods declaratively, as the example of MGS shows, but it is worth pursuing if it would let local operations be performed even more easily and the semantics of operations be more clearly defined.

Even though the CCF clearly has room for extension, however, this thesis has shown that in its current state the CCF is functional, and has already opened the door to modeling with cell complexes in one, two, and three dimensions. I look forward to seeing the many interesting models the Cell Complex Framework will be used to construct.

Bibliography

Abelson, Harold and Andrea diSessa (1986). *Turtle Geometry. The Computer as a Medium for Exploring Mathematics*. MIT Press. ISBN: 9780262510370.

Barbier de Reuille, Pierre (2008). *VVE documentation*. URL: http://pages.cpsc.ucalgary.ca/~pbarbier/vve_win/doc/html/index.html.

Baumgart, Bruce G. (1975). "A polyhedron representation for computer vision". In: *AFIPS '75: Proceedings of the National Computer Conference and Exposition*, pp. 589–596. DOI: 10.1145/1499949.1500071.

Besson, Sébastien and Jacques Dumais (2011). "Universal rule for the symmetric division of plant cells". In: *Proceedings of the National Academy of Sciences* 108.15, pp. 6294–6299. DOI: 10.1073/pnas.1011866108.

Botsch, Mario et al. (2010). *Polygon Mesh Processing*. A K Peters. ISBN: 978-1-56881-426-1.

Braess, Dietrich (2007). *Finite elements: Theory, fast solvers, and applications in solid mechanics*. 3rd ed. Cambridge University Press. DOI: 10.1017/CBO9780511618635.

Brisson, Erik (1990). "Representation of d-dimensional geometric objects". PhD thesis. University of Washington.

Burkhart, Daniel, Bernd Hamann, and Georg Umlauf (2010). "Adaptive and feature-preserving subdivision for high-quality tetrahedral meshes". In: *Computer Graphics Forum* 29.1, pp. 117–127. DOI: 10.1111/j.1467-8659.2009.01581.x.

Catmull, Edward and James Clark (1978). "Recursively generated B-spline surfaces on arbitrary topological meshes". In: *Computer Aided Design* 10.6, pp. 350–355. DOI: 10.1016/0010-4485(78)90110-0.

Chelladurai, Jeyaprakash (2012). "Simulation Modeling of Plant Tissue with an emphasis on the moss *Physcomitrella patens* Leaf Development". PhD thesis. University of Calgary.

— (2015). "Unpublished manuscript".

Chen, Jianer and Ergun Akleman (2003). *Topologically robust mesh modeling: Concepts, data structures, and operations*. Tech. rep. Texas A&M University. URL: http://www.viz.tamu.edu/faculty/ergun/research/topology/papers/tr03a.pdf.

Chomsky, Noam (1956). "Three models for the description of language". In: *IRE Transactions on Information Theory* 2 (3), pp. 113–124. DOI: 10.1109/TIT.1956.1056813.

Cieslak, Mikolaj, Adam Runions, and Przemyslaw Prusinkiewicz (2015). "Auxin-driven patterning with unidirectional fluxes". In: *Journal of Experimental Botany* 66.16, pp. 5083–5102. DOI: 10.1093/jxb/erv262.

Cieslak, Mikolaj et al. (2011). "Towards aspect-oriented functional-structural plant modelling". In: *Annals of Botany* 108.6, pp. 1025–1041. DOI: 10.1093/aob/mcr121.

Cormen, Thomas H, Charles E Leiserson, and Ronald L Rivest (1990). *Introduction to Algorithms*. MIT Press.

Coxeter, H. S. M. (1961). *Introduction to Geometry*. New York: Wiley.

Crank, John (1975). *The Mathematics of Diffusion*. 2nd ed. Oxford: Clarendon Press. ISBN: 0-19-853344-6.

Damiand, Guillaume and Pascal Lienhardt (2014). *Combinatorial Maps: Efficient Data Structures for Computer Graphics and Image Processing*. A K Peters. ISBN: 978-1482206524.

de Boer, Martin J. M., F. David Fracchia, and Przemyslaw Prusinkiewicz (1992). "A model for cellular development in morphogenetic fields". In: *Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology*. Springer-Verlag, pp. 351–370.

Delingette, Hervé (2008). "Triangular springs for modeling nonlinear membranes". In: *IEEE Transactions on Visualization and Computer Graphics* 14.2, pp. 329–341. DOI: 10.1109/TVCG.2007.70431.

Destuynder, Philippe (1985). "A classification of thin shell theories". In: *Acta Applicandae Mathematicae* 4 (1), pp. 15–63. DOI: 10.1007/BF02293490.

Dobkin, David P. and Michael J. Laszlo (1987). "Primitives for the manipulation of three-dimensional subdivisons". In: *SCG '87: Proceedings of the third annual symposium on computational geometry*, pp. 86–99. DOI: 10.1145/41958.41967.

Dupuy, Lionel et al. (2008). "A system for modelling cell-cell interactions during plant morphogenesis". In: *Annals of Botany* 101.8, pp. 1255–1265. DOI: 10.1093/aob/mcm235.

Dyn, Nira, David Levin, and John A. Gregory (1990). "A butterfly subdivision scheme for surface interpolation with tension control". In: *ACM Transactions on Graphics* 9 (2), pp. 160–169. DOI: 10.1145/78956.78958.

Edmonds, Jack R. (1960). "A combinatorial representation for polyhedral surfaces". In: *Notices of the American Mathematical Society*. Vol. 7, p. 646.

Egli, Richard and Neil F. Stewart (1999). "A framework for system specification using chains on cell complexes". In: *Computer-Aided Design* 31.11, pp. 669–681. DOI: 10.1016/S0010-4485(99)00064-0.

Errera, Léo (1886). "Sur une condition fondamentale d'équilibre des cellules vivantes". French. In: *Comptes rendus hebdomadaires des séances de l'Académie des sciences* 103, pp. 822–824.

Federl, Pavol (2002). "Modeling fracture formation on growing surfaces". PhD thesis. University of Calgary.

Franklin, W. Randolph (1992). "Local topological properties of polyhedra". URL: http://www.ecse.rpi.edu/~wrf/wiki/Research/unpub/localtopo.pdf (visited on 08/12/2015).

Giavitto, Jean-Louis and Olivier Michel (2001). "MGS: A rule-based programming language for complex objects and collections". In: *Electronic Notes in Theoretical Computer Science* 59.4, pp. 286–304. DOI: 10.1016/S1571-0661(04)00293-2.

Giblin, Peter J. (1977). *Graphs, Surfaces and Homology*. Chapman and Hall.

Gillespie, Daniel T. (1976). “A general method for numerically simulating the stochastic time evolution of coupled chemical reactions”. In: *Journal of Computational Physics* 22 (4), pp. 403–434. DOI: 10.1016/0021-9991(76)90041-3.

Ginsburg, Seymour (1960). “Some remarks on abstract machines”. In: *Transactions of the American Mathematical Society* 96.3, pp. 400–444. DOI: 10.2307/1993532.

Gouraud, Henri (1971). “Continuous shading of curved surfaces”. In: *IEEE Transactions on Computers* C-20 (6), pp. 623–629. DOI: 10.1109/T-C.1971.223313.

Grieneisen, Verônica A. et al. (2007). “Auxin transport is sufficient to generate a maximum and gradient guiding root growth”. In: *Nature* 449.7165, pp. 1008–1013. DOI: 10.1038/nature06215.

Grinspun, Eitan et al. (2003). “Discrete shells”. In: *Eurographics/SIGGRAPH Symposium on Computer Animation*.

Guibas, Leonidas and Jorge Stolfi (1985). “Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams”. In: *ACM Transactions on Graphics* 4.2, pp. 74–123. DOI: 10.1145/282918.282923.

Hanan, James (1992). “Parametric L-systems and their application to the modelling and visualization of plants”. PhD thesis. University of Regina.

Harrison, C. Jill et al. (2009). “Local cues and asymmetric cell divisions underpin body plan transitions in the moss *Physcomitrella patens*”. In: *Current Biology* 19, pp. 461–471. DOI: 10.1016/j.cub.2009.02.050.

Haselkorn, Robert (1978). “Heterocysts”. In: *Annual Review of Plant Physiology* 29, pp. 319–344. DOI: 10.1146/annurev.pp.29.060178.001535.

Hatcher, Allen (2002). *Algebraic Topology*. Cambridge University Press.

Hindmarsh, Alan C. et al. (2005). “SUNDIALS: Suite of Nonlinear and Differential/Algebraic Equation Solvers”. In: *ACM Transactions on Mathematical Software* 31.3, pp. 363–396. DOI: 10.1145/1089014.1089020.

Joy, Ken and Ron MacCracken (1996). *The refinement rules for Catmull-Clark solids*. Tech. rep. CSE-91-1. Computer Science Department, University of California, Davis.

Jullian, Yann (2005). “Conception et développement d’un éditeur graphique de filtre pour MGS”. MA thesis. ENSIIE.

Klincsek, Gheza Thomas (1980). “Minimal triangulations of polygonal domains”. In: *Annals of Discrete Mathematics* 9, pp. 121–123. DOI: 10.1016/S0167-5060(08)70044-X.

Kobbelt, Leif (2000). “$\sqrt{3}$-Subdivision”. In: *Proceedings of SIGGRAPH 2000*, pp. 103–112. DOI: 10.1145/344779.344835.

Korn, Robert W. and Richard M. Spalding (1973). “The geometry of plant epidermal cells”. In: *New Phytologist* 72, pp. 1357–1365. DOI: 10.1111/j.1469-8137.1973.tb02114.x.

Lévy, Bruno and Jean-Laurent Mallet (1999). *Cellular modeling in arbitrary dimension using generalized maps*. Tech. rep. GOCAD Consortium. URL: http://alice.loria.fr/index.php/publications.html?redirect=0&Paper=g_maps@1999&Author=Levy.

Lienhardt, Pascal (1991). “Topological models for boundary representation: a comparison with n-dimensional generalized maps”. In: *Computer-Aided Design* 23.1, pp. 59–82. DOI: 10.1016/0010-4485(91)90082-8.

— (1994). “N-dimensional generalized combinatorial maps and cellular quasi-manifolds”. In: *International Journal of Computational Geometry and Applications* 4.3, pp. 275–324. DOI: 10.1142/S0218195994000173.

Lindenmayer, Aristid (1968). “Mathematical models for cellular interactions in development: Parts I & II”. In: *Journal of Theoretical Biology* 18.3, pp. 280–315. DOI: 10.1016/0022-5193(68)90079-9.

— (1984). “Models for plant tissue development with cell division orientation regulated by preprophase bands of microtubules”. In: *Differentiation* 26, pp. 1–10. DOI: 10.1111/j.1432-0436.1984.tb01366.x.

Lindenmayer, Aristid and Grzegorz Rozenberg (1979). “Parallel generation of maps: Developmental systems for cell layers”. In: *Graph Grammars and Their Application to Computer Science and Biology*. Ed. by Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg. Lecture Notes in Computer Science 73. Springer, pp. 301–316. DOI: 10.1007/BFb0025728.

Lorensen, William E. and Harvey E. Cline (1987). “Marching cubes: A high resolution 3D surface construction algorithm”. In: *Computer Graphics* 21 (4), pp. 163–169. DOI: 10.1145/37402.37422.

Markov, Andrey Andreyevich (1958). “Insolubility of the problem of homeomorphy”. In: *Proceedings of the International Congress of Mathematicians*. Ed. by J. A. Todd. Trans. by Rolf Herken and Afra Zomorodian.

Max, Nelson (1999). “Weights for computing vertex normals from facet normals”. In: *Journal of Graphics Tools* 4 (2), pp. 1–6. DOI: 10.1080/10867651.1999.10487501.

Mayoh, Brian H. (1974). “Multidimensional Lindenmayer organisms”. In: *L-systems*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Lecture Notes in Computer Science 15. Springer, pp. 302–326. DOI: 10.1007/3-540-06867-8_24.

Měch, Radomír (2005). *CPFG Version 4.0 User's Manual*. URL: http://algorithmicbotany.org/lstudio/CPFGman.pdf.

Merks, Roeland M.H. et al. (2011). “VirtualLeaf: An open-source framework for cell-based modeling of plant tissue growth and development”. In: *Plant Physiology* 155.2, pp. 656–666. DOI: 10.1104/pp.110.167619.

Muller, David E. and Franco P. Preparata (1978). “Finding the intersection of two convex polyhedra”. In: *Theoretical Computer Science* 7 (2), pp. 217–236. DOI: 10.1016/0304-3975(78)90051-8.

Nakamura, Akira, Aristid Lindenmayer, and Kunio Aizawa (1986). “Some systems for map generation”. In: *The Book of L*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Springer, pp. 323–332. DOI: 10.1007/978-3-642-95486-3_26.

Nürnberg, Robert (2013). “Calculating the volume and centroid of a polyhedron in 3d”. URL: http://www2.imperial.ac.uk/~rn/centroid.pdf (visited on 08/12/2015).

Palmer, Richard S. and Vadim Shapiro (1993). “Chain models of physical behavior for engineering analysis and design”. In: *Research in Engineering Design* 5.3–4, pp. 161–184. DOI: 10.1007/BF01608361.

Paoluzzi, Alberto et al. (1993). “Dimension-independent modeling with simplicial complexes”. In: *ACM Transactions on Graphics* 12.1, pp. 56–102. DOI: 10.1145/169728.169719.

Pfaltz, John L. and Azriel Rosenfeld (1969). “Web grammars”. In: *IJCAI’69 Proceedings of the 1st international joint conference on Artificial intelligence*, pp. 609–619.

Prusinkiewicz, Przemyslaw (2009). “Developmental computing”. In: *Proceedings of Unconventional Computation 2009*. Lecture Notes in Computer Science 5715, pp. 16–23. DOI: 10.1007/978-3-642-03745-0_9.

Prusinkiewicz, Przemyslaw and Pierre Barbier de Reuille (2010). “Constraints of space in plant development”. In: *Journal of Experimental Botany* 61.8, pp. 2117–2129. DOI: 10.1093/jxb/erq081.

Prusinkiewicz, Przemyslaw, Mark Hammel, and Eric Mjolsness (1993). “Animation of plant development”. In: *Proceedings of SIGGRAPH 1993*, pp. 351–360. DOI: 10.1145/166117.166161.

Prusinkiewicz, Przemyslaw and Aristid Lindenmayer (1990). *The algorithmic beauty of plants*. The Virtual Laboratory. New York: Springer-Verlag.

Rosenfeld, Azriel and James P. Strong (1969). *A grammar for maps*. Tech. rep. 69-102. College Park, Maryland: University of Maryland, pp. 1–18.

Rudge, Timothy J. et al. (2012). “Computational modeling of synthetic microbial biofilms”. In: *ACS Synthetic Biology* 1.8, pp. 345–352. DOI: 10.1021/sb300031n.

Smith, Colin (2006). “On vertex-vertex systems and their use in geometric and biological modelling”. PhD thesis. University of Calgary.

Smith, Colin, Przemyslaw Prusinkiewicz, and Faramarz Samavati (2003). “Local specification of surface subdivision algorithms”. In: *Applications of Graph Transformations with Industrial Relevance*, pp. 313–327. DOI: 10.1007/b98116.

Spicher, Antoine and Olivier Michel (2007). “Représentation et manipulation de structures topologiques dans un langage fonctionnel”. French. In: *Technique et science informatiques* 26.9, pp. 1169–1194. DOI: 10.3166/tsi.26.1169-1194.

Stern, Ari and Mathieu Desbrun (2006). “Discrete geometric mechanics for variational time integrators”. In: *SIGGRAPH ’06: ACM SIGGRAPH 2006 Courses*. ACM SIGGRAPH 2006 Courses. Chap. 1, pp. 75–80. DOI: 10.1145/1185657.1185669.

Stowe, Keith (2007). *An Introduction to Thermodynamics and Statistical Mechanics*. Cambr. ISBN: 978-0-521-86557-9.

Terraz, Olivier et al. (2009). “3Gmap L-systems: an application to the modelling of wood”. In: *The Visual Computer* 25, pp. 165–180. DOI: 10.1007/s00371-008-0212-5.

Ulam, Stanislaw (1962). “On some mathematical problems connected with patterns of growth and figures”. In: *Mathematical Problems in the Biological Sciences*. Ed. by Richard E. Bellman. Vol. 14. Proceedings of Symposia in Applied Mathematics, pp. 215–224. DOI: 10.1090/psapm/014/9947.

van Overveld, Kees and Brian Wyvill (2004). "Shrinkwrap: An efficient adaptive algorithm for triangulating an iso-surface". In: *The Visual Computer* 20 (6), pp. 362–379. DOI: 10.1007/s00371-002-0197-4.

Vince, Andrew (1983). "Combinatorial maps". In: *Journal of Combinatorial Theory*. B 34 (1), pp. 1–21. DOI: 10.1016/0095-8956(83)90002-3.

Weiler, Kevin (1985). "Edge-based data structures for solid modeling in curved-surface environments". In: *IEEE Computer Graphics and Applications* 5 (1), pp. 21–40. DOI: 10.1109/MCG.1985.276271.

Wyvill, Geoff, Craig McPheeters, and Brian Wyvill (1986). "Data structures for soft objects". In: *The Visual Computer* 2 (4), pp. 227–234. DOI: 10.1007/BF01900346.

Yoon, Ho-Sung and James W. Golden (1998). "Heterocyst pattern formation controlled by a diffusible peptide". In: *Science* 282.5390, pp. 935–938. DOI: 10.1126/science.282.5390.935.