THE UNIVERSITY OF CALGARY

On Vertex-Vertex Systems and Their Use in Geometric and Biological Modelling

by

Colin Smith

A DISSERTATION
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

April, 2006

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a dissertation entitled "On Vertex-Vertex Systems and Their Use in Geometric and Biological Modelling" submitted by Colin Smith in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY.

Dr. Przemyslaw Prusinkiewicz
Department of Computer Science

Dr. Faramarz Samavati
Department of Computer Science

Dr. Michael Surette
Department of Microbiology & Infections Diseases
Department of Biochemistry & Molecular Biology

Dr. Lawrence Harder
Department of Biological Sciences

Dr. Alla Sheffer
Department of Computer Science
University of British Columbia

Date

Abstract

In the areas of geometry and biology, there are a number of modelling problems that require the creation and manipulation of discrete surfaces that behave dynamically. For example, in geometric modelling there are surface subdivision algorithms that require the repeated insertion of vertices into a polygon mesh. In biological modelling there is the question of modelling growing surfaces, such as a growing flower or a growing tissue of cells. In these cases, there is the open question of how to model *dynamical systems with a dynamical structure* of a 2-manifold topology, discrete surfaces that have components that change in character, connectivity and number over time.

However, the selection of available tools for modelling dynamical surfaces is limited. There have been some proposed solutions for limited cases, such as *cell systems* for modelling cells. But there is still a need for a methodology and tools for dealing with dynamical surfaces in general.

In this dissertation, I present a methodology for modelling dynamical systems with a dynamical structure of a 2-manifold topology. This methodology is comprised of the *vertex-vertex data structure* and *algebra* and is implemented in the *vertex-vertex software environment*. I also demonstrate its application with examples in the domains of geometric and biological modelling.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Part I

An Introduction to Vertex-Vertex Systems

Chapter 1

The Philosophy of Local Modelling

1.1 Global and Local Modelling Methods

A model can be designed such that it represents either a *global* or *local* perspective of a system. A global model is one where the data and simulation of a system is characterised as a whole. Any component in the system can be related to every other component by an index. For example, data may be characterised by a matrix and transformations upon it are done by a system of equations.

By contrast, a local model is one where relations between components are described without an indexing scheme (*i.e.* each component can directly reference the neighbouring components). A result can be obtained by considering how each component changes over time and how neighbouring components interact. When a system can be well-described by its components, a local modelling methodology is often preferable.

This distinction between local and global methodologies is not a hard categorical dividing line. Rather, it is a heuristic question that can help formulate a method for modelling a particular system. Given a particular modelling problem, are the processes easier to describe as phenomena over the whole of the data or over subsets of the data? In practice, the answer depends on whether an indexing scheme is a useful way of referencing the components in a model.

1.2 The Advantages of Local Modelling for Geometric and Biological Modelling

Local modelling methods are particularly useful for many classes of problems in geometric and biological modelling.

In the case of geometric modelling, many algorithms operate on polygon meshes that use a local methodology in their implementation. For example, an algorithm may require a measure or a transformation of a mesh based on a vertex in the mesh and the immediately connected vertices (the *k-ring* around a vertex). An example of a geometric modelling algorithm that uses k-rings is that of *incremental adaptive subdivision*, covered in §5.5.

Similarly, biological models often deal with how individual components change over time. For example, the growth of a plant can be described as the result of the growth of the stem, branches, leaves and flowers. Or the growth of a tissue can be described as the result of the growth and division of each cell in the tissue.

The above examples belong to a class of systems called *dynamical systems with a dynamical structure* ($(\mathrm{DS})^2$) [18, 19]. A *dynamical system* includes components with properties that change over time. In a *dynamical structure*, the connectivity or the number of components change over time. So, a $(\mathrm{DS})^2$ describes a model with components that change in nature, number and connectivity over time.

The dynamical structure aspect of the $(\mathrm{DS})^2$ description makes a local modelling method important. If the number of components or if the relations change, an indexing scheme becomes burdensome. For example, a linear structure could be contained in an array with items indexed sequentially (say indices 1, 2, 3 & 4).

However, if an item in the middle is removed (*e.g.* the item indexed as 3), then either all subsequent items have to get a new index (the item indexed as by 4 is now indexed as by 3) or the indexing will not indicate to the ordering of the items (the index sequence is now 1, 2 & 4, but is there something between 2 and 4?).

Also, an intuitive indexing scheme may be elusive if the structure is irregular. Arrays and matrices have obvious indexing schemes (integers on the rows and columns), but what is a good indexing scheme for a polygon mesh? There are no rows and columns to follow. One possible indexing scheme, proposed by Zorin *et al.* [88], illustrated in Figure 1.1, requires three components: one superscript and a two subscripts. It is not obvious how the numbering proceeds along the mesh; without examining the figure, it would be difficult to appreciate that vertex $p^j_{1,4}$ is adjacent to vertex $p^j_{i-1,6}$.

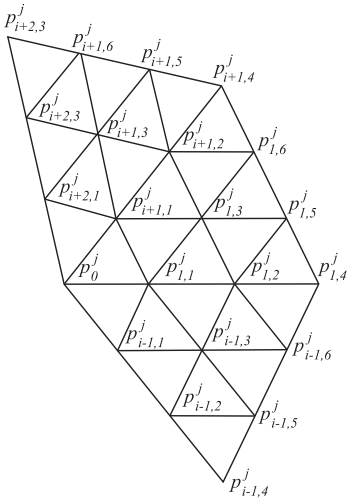


Figure 1.1: An example of an indexing scheme for a polygon mesh (adapted from Figure 4.2 in [88])

1.3 A Methodology for Modelling Dynamical Surfaces

While there are some well-known systems for modelling $(DS)^2$, these systems are typically only appropriate for particular topologies or particular modelling problems. For example, *L-systems* [32] are useful for modelling $(DS)^2$ with linear and branching topologies and *cell systems* can be used to model cellular tissues. However, there is no general modelling system that is appropriate for $(DS)^2$ with a discrete 2-manifold.

This is an important topology to consider in the areas of geometric and biological modelling. In geometric modelling, there are important problems in modelling dynamical polygon meshes. Surface subdivision algorithms and procedural surface creation are examples of often used dynamical surface algorithms. In biological modelling, there are problems of modelling growing surfaces, such as growing flowers or cellular tissues. A system to model a wide variety of situations, including the aforementioned problems, on discrete 2-manifold topologies would be a valuable tool.

In this dissertation, I present a methodology for implementing models that are a $(DS)^2$ on a discrete 2-manifold topology. At the core of this are *vertex-vertex systems* (vv), comprised of the vv data structure, the vv algebra and the vv language and software environment. VV fills the need for a general system for modelling $(DS)^2$ in a wide variety situations. The use of vv is demonstrated in this dissertation through a series of examples. The examples were chosen on two criteria: to show the diverse domain of problems that can be modelled with vv and to show diverse techniques and patterns that can be used in vvprograms.

In Part 1 of this dissertation, vv and its implementation in the vv software environment is presented. In Part 2, it is demonstrated how to apply vv to problems

in geometric modelling and in Part 3, to problems biological modelling. In Part 4 vv is compared to other modelling systems and data structures, followed by some concluding remarks. Finally, the Appendices contain the technical reference for the vv software environment.

Chapter 2

A Review of Methods for Modelling Dynamical Systems with a Dynamical Structure

Several systems for creating $(DS)^2$ models have been proposed which are appropriate for various classes of problems. In the following review of existing modelling systems for $(DS)^2$, the advantages and limitations of each are highlighted. The purpose is to provide a feature list that was considered in the design of vv.

2.1 Cellular Automata

An early example of modelling of discrete dynamical systems is the use of *cellular automata* [78, 79]. Cellular automata are components on a set topology, usually a chain or a grid, where each component is an automaton with a set of data and procedures. Each automaton can exchange information only with its immediate neighbours.

Cellular automata are inherently local: indices are not needed to refer to relations between cells. In addition, the data and procedures associated to each automaton are arbitrary and the topology of the structure is fixed. So, if a model describes as a dynamical system with a fixed structure, cellular automata may be a good choice. But, this is not the case when modelling a $(DS)^2$ on a discrete 2-manifold, where the structure is also dynamical.

2.2 L-Systems

The other widely-used modelling paradigm for $(DS)^2$ is *L-systems* [32, 55]. An L-system is a string of *modules*, a symbol, possibly with a set of data, and a set of *productions*, rewriting rules based on the symbol of each module. The string structure allows for the representation of linear and branching topologies.

The productions allow for the alteration, replacement, addition and removal of modules or branching points and so can change the topology of the structure at each rewriting step. Moreover, they are in a form that readily allows changes to the structure. Finally, all productions are applied to the string synchronously; the entire string develops in stages. This is all very convenient for modelling linear structures that develops over time.

The predecessor and successor structure of the productions provide some useful features. The prior state of the string, in the predecessor, and its current state, in the successor, are both available at the moment of the production application. Also, the context portion of the predecessor gives access to the data in neighbouring modules in a convenient, local manner.

The *turtle geometry* interpretation of L-systems [73, 52, 53] provides a natural and convenient method of relating an L-system to a graphical representation. An *affine geometry* interpretation of an L-system can also be easily specified [56].

A recent and widely used [29] implementation of L-systems, *LPFG* [25, 26], uses the *L+C* language used to specify L-systems. The L+C language combines L-system constructs with the C++ language [72]. This approach is advantageous as it allows the modeller to access all the features and libraries already available in C++ in con-

junction with the features of L-systems. For example, the modeller can include an existing numerical library, such as the well-known *Numerical Recipes in C* [51], or to supplement the graphics primitives provided in LPFG with calls directly to OpenGL [63].

2.3 Map L-systems

Map L-systems [33, 55] attempt to apply the design of L-systems to a graph structure for the of modelling cells. Like L-systems, map L-systems have a structure labelled with symbols and a set of productions to transform the structure.

The alphabet is a set of *wall symbols*, and in Lindenmayer's original formulation [33] also *cell symbols*. The symbols are connected as a *map* that defines the topology of the structure. The productions replace a wall segment with other wall segments. If there is a cell alphabet, then cell productions define where new walls are added. If there is no cell alphabet, as is the case in [55], then new walls are implied by branches in the productions.

If a map L-system is embedded into some space, the structure is a *polygon mesh*; but, the geometry of the structure must be supplied by some external description or simulation. The geometry of the structure cannot be provided conveniently as part of a map L-system. This is unlike L-systems, where special symbols can be added to control the geometry.

2.4 Cell Systems

Cell systems [10] are somewhat similar to map L-systems in that they are designed to model cells and they share the notion of transforming a labelled map structure.

Cell systems have an alphabet to label cells. The structure is a polygon mesh overlaid with a *vector field*. The vector of the vector field in the position of a cell's centre is that cell's *reference vector*. A production of a cell system replaces a cell with one or two cells, the plane of division at an angle to the reference vector, as specified as part of the productions.

The productions do not alone determine the topology of the structure. The geometry of the polygon mesh is controlled by a physical simulation and the positions of the walls depends on the geometry at the time of insertion. The implication is that a change in the physical simulation can produce a different topology, even though the productions do not different. Thus, the result depends on the particular implementation of the physics engine of the cell systems software.

Although cell systems provide a method for modelling a $(DS)^2$ on a discrete 2-manifold, they are not appropriate for modelling arbitrary 2-manifold transformations. Firstly, the only available topological operation is cell division; walls can be created, but not split, removed or reconnected. Therefore, structures can be made more complex, but not reduced. Secondly, the only geometric interpretation available is physical; cell systems cannot be used for models that require explicit geometric transformations.

2.5 MGS

MGS, (encore) un Modèle Géneral de Simulation (de système dynamique), [18, 19] is a functional language that operates on monoidal collections group-based fields that can be used to implement a $(DS)^2$ model in an arbitrary topology. MGS is a flexible language that allows for the implementation of a wide variety of models.

MGS was still a nascent language and not widely known when vv was first conceived, it was not considered as an option when looking for a system for modelling a $(DS)^2$ on a discrete 2-manifold, although it is capable of fulfilling many of the desired requirements.

In some ways MGS is not the ideal system for implementing the classes of models here considered. The notions of monoidal collections and group-based fields encompass a wide range of topological structures, but are many topologies are not regular and so cannot be described readily in these terms. Group based-fields are also quite general; they can represent structures of arbitrary dimension. At this level of generality, notions specific to 2-manifolds are not directly supported. The modeller may also find the semantic gap between 2-manifolds and group-based fields unintuitive.

MGS also has two drawbacks in practice. MGS is a custom-built language and so a modeller's experience and resources with other languages, such as the use of program code and libraries from other sources, cannot be immediately incorporated into a model implemented in MGS. Experience with L+C demonstrates that having a modelling language that is an extension of an existing and well-established language is extremely advantageous. Also, many programmers, including this author, simply prefer imperative languages over functional ones.

2.6 A Summary of Desired Features for a VV

All the aforementioned modelling systems have a mechanism for conveniently accessing the neighbours of a component. All the systems introduced in this chapter, except cellular automata, provide a convenient method for the transformation of the structure and topology. Clearly these are desirable features in a modelling system for $(DS)^2$.

Some systems, allow only limited ways to interpret the structure. For example, with map L-systems and cell systems, only interpretations by means of physical simulation are available; they lack geometric interpretations. In contrast, L-systems can be interpreted with turtle geometry, affine geometry and physical simulation, so they allow a much wider range interpretations. This range of available interpretations broadens the variety of models that can handled by a system, and so such a wide range is considered a desirable feature. Ideally, a modelling system should be able to use any interpretation that is appropriate for a particular situation.

The L+C programming language, there is the successful design of using C++ extended with L-system specific constructs. Quite simply, the design of a modelling system cannot anticipate all the needs of the modeller, so it makes sense to augment the available tools by using an existing, general purpose programming language.

Chapter 3

VV Systems

In this chapter, *vertex-vertex systems* (vv) are introduced. VV is a methodology for modelling dyamical systems with a dynamical structure on a 2-manifold topology.*

3.1 Definitions

Let U be the *universe* of elements called *abstract vertices* such that it is an enumerable set ordered by a relation $<$. Next, let $N : U \mapsto 2^U$ be a function that takes every vertex $v \in U$ to a finite subset $v^\star \subset U$ of other vertices such that $v \notin v^\star$. The subset $v^\star$ is called a *neighbourhood*, and its elements are the *neighbours*† of v. Note that the neighbourhood around the vertex v is also an open disc on the structure around v. Finally, let the *vertex set* $S \subset U$ be a finite subset of the universe U, and N_S be the restriction of the neighbourhood function N to the domain S; thus $N_S(v) = v^\star$ if $N(v) = v^\star$ and $v \in S$ (the elements of $v^\star$ may lay outside S). The pair $\langle S, N_S \rangle$ is called a *vertex-vertex structure* over the set S with neighbourhood N_S.

An *undirected graph* over a vertex set S is a vertex-vertex structure over S, in which all neighbourhoods are included in S (the vertex set S is *closed* with respect to the function N) and vertex u is in the neighbourhood of v if and only if vertex v is in the neighbourhood of u ($u \in v^\star$ if and only if $v \in u^\star$, the *symmetry condition*).

*The text and figures of §3.1 and 3.2 are adapted from the original publication on vv [69].

†The terminology is motivated by the practice of referring to adjacent cells in a grid as neighbours.

The pairs (u, v) of vertices that are in the neighbourhood of each other are called *edges* of the graph. An edge is *oriented* if the pair (u, v) is considered different from (v, u).

A *vertex-vertex rotation system*, or *vertex-vertex system* for short, is a vertex-vertex structure in which the vertices in each neighbourhood form a cyclic permutation (i.e., are arranged into a circular list). A *graph rotation system* is a vertex-vertex system that is both a graph and a vertex-vertex rotation system.

A *polygon mesh* is a collection of vertices, edges bound by vertex pairs, and polygons bound by sequences of edges and vertices. A mesh is a *closed 2-manifold* if it is everywhere locally homeomorphic to an open disk [86].

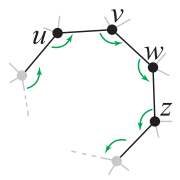


Figure 3.1: A polygon identification in a graph rotation system.

A *polygonal interpretation* of a vertex-vertex system maps it into a polygon mesh. Such an interpretation is a variant of the *Edmonds' permutation technique* [15, 86, 2], which is defined for connected graph rotation systems. It defines polygons of the mesh using the following algorithm (Figure 3.1). Given an oriented edge (u, v) in $S \times S$, the next oriented edge is (v, w) such that w immediately follows u in the *cyclic* neighbourhood of v. Next, the oriented edge (w, z) is found such that z immediately follows v in the neighbourhood of w. This process is continued until it returns to the starting point u. The resulting *orbit* (cyclic permutation) of vertices $u, v, w, z,$ &c. is the boundary of a polygon. By considering all such orbits in S, a polygon

mesh is obtained with polygons on both sides of each (unoriented) edge. From this construction it follows that the resulting mesh is a uniquely defined, orientable, closed 2-manifold (see [86] for a proof).

Vertex *positions* are a crucial aspect of the *geometric interpretation* of vv. The geometric interpretation considered here is such that edges are drawn as straight lines between vertices, and polygons are properly defined if their vertices and edges are coplanar.

The above progression of notions is summarised in Figure 3.2. It suggests that polygon meshes can be manipulated using three types of operations: set-theoretic, topological, and geometric operations. The most difficult problem is the manipulation of topology. This matter is addressed by introduction of a set of operations that modify at most one neighbourhood at a time, and transform a vertex-vertex system into another vertex-vertex system. The individual operations do not necessarily transform graphs into graphs, because they may create *incomplete neighbours* that violate the symmetry condition ($u \in v^\star$ but $v \notin u^\star$).

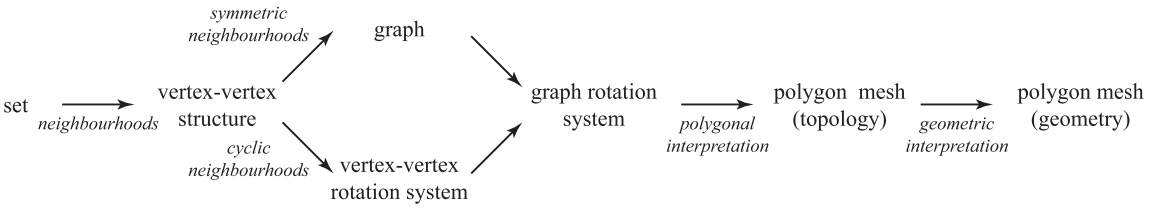


Figure 3.2: Relations between notions pertinent to vv

3.2 The VV Algebra

The *vv algebra* consists of the class of vertex-vertex rotation systems with a set of operations defined on them. These operations are introduced using a mathematical notation that combines standard and new mathematical symbols. The equivalent expressions and statements of the *vv language* are also presented. A description of this language and its implementation is given in Section 3.3. A complete specification of the vv language is presented in Appendix A.

In the vv language, vertex sets are a predefined data type. A set S is created using the declaration mesh S, and is in existence according to the standard scoping rules of C++. The vv language supports a subset of the standard set operations, listed in Table 3.1. In addition to operations that return a set as the result, vv includes iteration operators for flow control in vv programs.

Name	Math. notation	VV Language
set creation	let $S \subset U$	mesh S
assignment	$S = T$	$S = T$
union	$S = S \cup T$	merge S with T
addition of an element	$S = S \cup \{v\}$	add v to S
removal of an element	$S = S - \{v\}$	remove v from S
iteration over a set	$\forall v \in S$	forall v in S
iteration over neighbours	$\forall x \in v^\star$	forall x in v

Table 3.1: Set-theoretic operations supported by the vv language

Topological operations are the core of the vv algebra. They are divided into three groups: *query*, *selection*, and *editing* operations. Query operations return information about vertices. Selection operations return an element of a vertex neighbourhood. Editing operations locally modify a vertex-vertex system. Definitions of these

operations are given in Table 3.2.

The standard functional notation $f(v)$ or vv expression v\$f associates a property f with a vertex v. For the examples presented in this thesis, it is always assumed that there is a property, pos, accessed as v\$pos, that is a position in space of type util::Point<double>[‡] (see §B.2.1 for details of this type). The standard C++ operator overloading mechanisms are used to extend arithmetic operators to positions and vectors. However, because the properties of a vertex are defined on a per model basis, another definition for the vertex's position could be used.

Operations of the vv algebra are commonly iterated over vertex sets. This raises important questions concerning the sequencing of these individual operations. For example, if the same operation is to be performed on a pair of neighbouring vertices u and v, the results may differ depending on whether u is modified first, v is modified first, or both vertices are modified simultaneously. To eliminate the unwanted dependence on the execution sequence, the *coordination operation* synchronise S, which creates a copy $`v$ of each vertex $v$ in the set $S$. All subsequent operations on the vertices $v \in S$ (until the next synchronise statement) do not affect the vertices $`v$, which continue to store the "old" values of vertex attributes. For example, $`v$\$pos denotes the position of vertex $v$ at the time when the synchronise statement was last issued, whereas $v$\$pos denotes the current position of $v$. Similarly, $`v^\star$ and $v^\star$ denote the old and current neighbourhoods of v. The use of old attributes instead of the current ones makes it possible to iterate over the elements of a set in any order without affecting the iteration results.

[‡]For the sake of brevity in the examples, the type util::Point<double> is aliased as Pt. This is done simply using the C++ statement typedef util::Point<double> Pt.

Name	Math. notation	VV Language	Description	Note	Fig
		Query operations			
membership	$x \in v^\star$	is x in v	true iff x is in the neighbourhood of v		
order	$x < v$	$x < v$	true iff x precedes v in the universe U		
valence	$\lvert v^\star \rvert$	valence v	returns the number of neighbours of v		
		Selection operations			
any	let $x \in v^\star$	any in v	returns a neighbour of v	1	
next	$v^\star \uparrow x$	nextto x in v	returns the vertex following x in the neighbourhood of v	2	b
previous	$v^\star \downarrow x$	prevto x in v	returns the vertex preceding x in the neighbourhood of v	2	c
		Editing operations			
create	let $v \in U$	vertex v	create a vertex		
set neighbours	$v^\star = \{a, b, c\}$	make {a, b, c} nb_of v	set the neighbourhood of v to the given circular list	3	a
erase	$v^\star = v^\star - x$	erase x from v	removes x from the neighbourhood of v if $v \in x^\star$	4	b
replace	$v^\star = v^\star - a + x$	replace a with x in v	substitute x for a in the neighbourhood of v	5	c
splice after	$v^\star + x \succ a$	splice x after a in v	insert x after a in the neighbourhood of v	5	d
splice before	$v^\star + x \prec a$	splice x before a in v	insert x before a in the neighbourhood of v	5	e

1. Returns the null vertex if $v^\star$ is empty.
2. Returns the null vertex if $x \notin v^\star$.
3. Not defined if v appears in the list, or the same vertex is listed twice.
4. No effect if $x \notin v^\star$.
5. No effect if $a \notin v^\star$; not defined if $x = v$ or $x \in v^\star$.

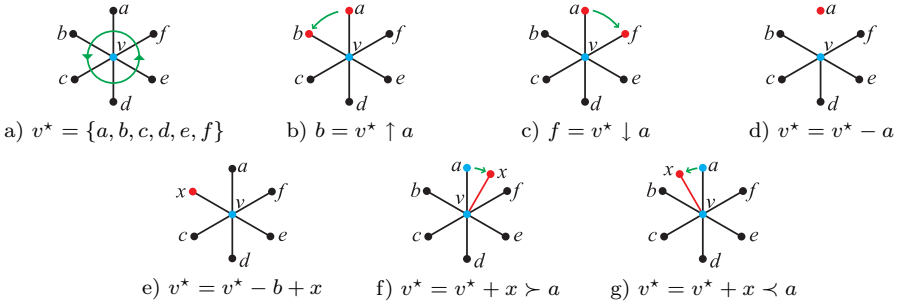


Table 3.2: Top: definition of the topological operations of the VV algebra. Bottom: graphical interpretation of the selection and editing operations. a) Setting the initial neighbourhood of vertex v. b-g) The results of various operations applied to v.

3.3 The Implementation of VV Systems

VV is implemented as a set of programs and libraries collectively called the vv software environment. The central component of this environment is libvv, a C++ library containing data structures and functions implementing the vv polygon mesh representation and algebra. The user can refer to these structures and functions directly from a program written in C++, or from a program written in the vv language.

The vv language extends C++ with keywords and expressions implementing the vv algebra. They are listed under the column 'vv statement' in Tables 3.1 and 3.2.

All of the algorithms presented in this dissertation are code fragments in the vv language and can be used directly in version 1.1 of the vv software environment[§]. To aide the reader, the algorithms have been typeset such that all the variables of types particular to vv (*i.e.* vertices and vertex sets) are in italics and commands of the vv language are in sans-serif. For clarity, vertex variables are always lower case and vertex sets are in upper case. Regular C++ code is not specially typeset, except for comments which are typeset in small caps and not given line numbers to set them apart.

To be executed, a vv program is first translated into a C++ program, with the keywords and expressions specific to vv translated into calls to the libvv library. This C++ program is then compiled into a dynamically linked library (DLL). The modeling program, called vvinterpreter, loads this DLL, runs, and produces the graphical output. This whole processing sequence is automated: from the user's perspective, the vvinterpreter treats the vv program as an input and runs accordingly. Figure 3.3

[§]There is a version 2.0 of the vv software environment. It is discussed in §13.2.4.

depicts the organisation of the software and how the data flows between components. This approach is based on that introduced by Karwowski and Prusinkiewicz [25, 26] to translate and execute L-system-based programs in the L+C language. An example of how to use the vv software is provided in Appendix C.

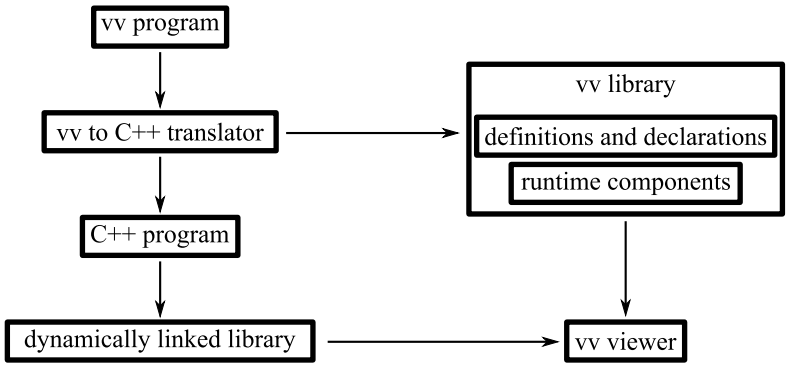


Figure 3.3: The organisation and data flow of the vv software environment

Most of the work is done in the translation phase. The translator program must do two things: generate C++ definitions of the vertices, edges and vertex sets based on the properties declared in the vv program and translate expressions in the vv language into expressions in C++.

The C++ definitions of the vertices and edges are created from templates [72, Chapter 13]. The use of templates has the advantage that parts of the vv language that are declared but never used are not compiled into the output binary. Thus, features that are not used do not incur a penalty. Furthermore, the use of templates allows the inclusion of arbitrary parameters without recourse to inheritance. Since inheritance requires that extra pointer indirections through a virtual table, inherited types perform slower than those that are not inherited. Therefore, avoiding inheritance improves the run-time performance of vv programs.

3.4 A Short VV Example: Vertex Insertion

Insertion of a new vertex on an edge is a commonly used transformation. The vertex insertion algorithm is illustrated in Figure 3.4 and its vv implementation is shown in Algorithm 3.1. The algorithm works by first creating a new vertex, x, (line 2) and sets its neighbourhood to be the two vertices p and q (line 3). Then, p and q are connected to x using the replace statement (lines 4 and 5). This algorithm is used in many of the examples presented later in this dissertation.

Figure 3.4: Illustration of vertex insertion

Algorithm 3.1: Vertex insertion

```
1   vertex insert(vertex p, vertex q)  {
–       // Create a vertex
2       vertex x;
–       // Assign the two supplied vertices to the
–       // neighbourhood of the new one
3       make { p, q } nb_of x;
–       // Put the new vertex into the neighbourhoods
–       // of the supplied ones
4       replace p with x in q;
5       replace q with x in p;
–       // Return the new vertex
6       return x;
7   }
```

3.5 Extensions to the VV Formalism

Several new features that have been added to vv since it was first introduced in [69]. These features have been designed to increase the convenience of implementing mod-

els in vv. They are introduced here and their use is demonstrated in the algorithms that follow.

The first two extensions (§3.5.1 and §3.5.2) are constructs that simplify compound expressions of next and previous operations. The third extension (§3.5.3) is an additional method to provide access to the neighbourhood. Finally, (§3.5.4) an extension of vv that allows information to be stored on the edges of a polygon mesh is given.

3.5.1 Indexed Next and Previous Operations

Given some vertex a with a neighbourhood that contains a vertex b, the vertices in that neighbourhood adjacent to b can be found with the expressions

$$a^\star \uparrow b \qquad \text{and} \qquad a^\star \downarrow b.$$

However, it may be that the vertex several positions away in a neighbourhood is desired. Compounded next and previous operations can be used for this. For example, the expression to find the vertex three positions following b in the neighbourhood of a is

$$a^\star \uparrow (a^\star \uparrow (a^\star \uparrow b)),$$

which is an unfortunately unwieldly composition. To simplify these sorts of expressions, the *indexed next* and *previous* operations are here introduced and are defined with the recursive equations

$$a^\star \uparrow^{(i)} b = \begin{cases} i = 1 & a^\star \uparrow b \\ i > 1 & a^\star \uparrow (a^\star \uparrow^{(i-1)} b) \end{cases} \quad \text{and} \quad a^\star \downarrow^{(i)} b = \begin{cases} i = 1 & a^\star \downarrow b \\ i > 1 & a^\star \downarrow (a^\star \downarrow^{(i-1)} b) \end{cases}$$

for some positive integer i. In the vv language, these operations are specified as

$$\mathsf{next}(i)\mathsf{to}\ b\ \mathsf{in}\ a \qquad \text{and} \qquad \mathsf{prev}(i)\mathsf{to}\ b\ \mathsf{in}\ a.$$

In the butterfly subdivision algorithm (Algorithm 5.3) the use of indexed next and previous operations is illustrated.

3.5.2 Path Statements

Indexed next and previous operations are extremely useful for simplifying expressions in the vv algebra, but they apply only when a compound expression can be structured in the aforementioned recursive form. Since not all compound expressions can be thus structured, there are *path statements* as a shortened form for the general case.

A path statement begins with a pair of adjacent vertices and a list of *path operations*, executed in sequentially. The path operations are given in Table 3.3 and the path commands in the vv language are given in Table 3.4.

Operation	Effect
next	Sets $\langle a, b\rangle$ to $\langle a, a^\star \uparrow b\rangle$
prev	Sets $\langle a, b\rangle$ to $\langle a, a^\star \downarrow b\rangle$
swap	Sets $\langle a, b\rangle$ to $\langle b, a\rangle$

Table 3.3: Path operations

Using the path constructs, a compound expression such as

$$x = (b^\star \uparrow (b^\star \uparrow a))^\star \uparrow b$$

or equivenlently, in the vv language,

$$\textbf{vertex}\ x = \textbf{nextto}\ b\ \textbf{in nextto nextto}\ a\ \textbf{in}\ b\ \textbf{in}\ b;$$

Statement	Effect
@(a, b) ... *path ops* ... @	Returns the first vertex in the pair resulting from the operations
'@(a, b) ... *path ops* ... @	Returns the first vertex in the pair resulting from the operations on the synchronised neighbourhoods
@&(a, b) ... *path ops* ... @	Sets a and b to the pair resulting from the operations
'@&(a, b) ... *path ops* ... @	Sets a and b to the pair resulting from the operations on the synchronised neighbourhoods

Table 3.4: Path commands in the vv language. The *path ops* can be any if the path operations given in Table 3.3.

can be written as a path statement

vertex x = **@**(a, b) **swap next next swap next @**;.

Though the example is admittedly contrived and does not demonstrate the usefulness of path statements fully, it shows that a long statement involving next and previous operations can be made shorter and easier to read. The advantage of path statements is better demonstrated in Algorithm 5.6, where a short path statement in a loop is used to traverse a polygon of arbitrary size.

3.5.3 Neighbourhood Flags

In some cases, it can be useful to track a particular vertex in a neighbourhood. For example, given vertex a with the neighbourhood $\{b, c, d, e, f\}$, it may be desirable to place some sort of mark on the vertex d such that at some later time, it is possible to ask "What is the vertex that was marked in the neighbourhood of a?" and get the vertex d in return.

In practice, this could be done by storing a pointer associated with a that points to d. However, a pointer is not a sufficient representation for this relation. If the neighbourhood is altered, there is a risk that a pointer could be corrupted. In par-

ticular, the splice, erase and replace operations may change how the neighbourhood is arranged in memory. The solution is to instead use an iterator that refers to the position of d in the neighbourhood of a. If the neighbourhood is altered, an iterator will always refer to the same relative location in the neighbourhood. Unless the neighbourhood is completely changed by an assignment or d is explicitly removed with the erase operation, the iterator will always refer to d. Thus, an iterator can be used as an identifier to a particular vertex and is stable with respect to the splice, replace and erase operations. This iterator is the *flag*. Operations on the flag are given in Table 3.5.

Algebraic Notation	VV Language	Description
$a^{\bowtie}$	flagged in a	Retrieves the vertex in the position flagged in a.
$a \bowtie b$	flag b in a	Sets the flag of a to the position where b is. b may be a null vertex.†
$a^{\star} \uparrow \bowtie$	nextto flag in a	Returns the vertex after the flagged position in a.‡
$a^{\star} \downarrow \bowtie$	prevto flag in a	Returns the vertex before the flagged position in a.‡
$a^{\star} + b \succ \bowtie$	splice b after flag in a	Inserts b in the neighbourhood of a after the flagged position.‡
$a^{\star} + b \prec \bowtie$	splice b before flag in a	Inserts b in the neighbourhood of a before the flagged position.‡
$a^{\star} - \bowtie + b$	replace flag with b in a	Replaces the vertex in flagged position in a with b.‡
$a^{\star} - \bowtie$	erase flag from a	Removes the vertex in the flagged position from the neighbourhood of a and sets the flag to null.‡

† If b is not in the neighbourhood of a, then the statement has no effect.
‡ If the flag is null then the statement produces an error.

Table 3.5: Operations on the flag

Whenever the flag is set to null, it is semantically considered *unset* and does not refer to any position in the neighbourhood. Each vertex is initialised with a flag set to null. The flag of each vertex is synchronised in the same manner as the other

vertex properties. See the sea shell and cell system models (Algorithms 8.1 & 12.6) for examples that use the flag feature.

3.5.4 Edge Information

Information can be added to the edges of a mesh by only a small modification to the vv data structure and additions to the algebra. Existing operations of the algebra do not require any changes.

The vv data structure defines the neighbourhood of some vertex v as the ordered, circular list of vertices

$$v^\star = \{v_1, v_2, \ldots, v_n\}.$$

To add in the edges, the elements are changed from vertices to pairs, consisting of vertices and an arbitrary block of information. This alteration gives

$$v^\star = \{\langle v_1, e_{v,v_1}\rangle, \langle v_2, e_{v,v_2}\rangle, \ldots, \langle v_n, e_{v,v_n}\rangle\},$$

where each e_{v,v_i} is the information on the directed half-edge that goes from vertex v to v_i. Notationally, for vertices a and b, $e_{a,b}$ is the information for the half-edge from a to b.

Edge information nodes are synchronised when the synchronise command is issued and can be retrieved using the backquote notation on a vertex, as with all other synchronised information of a vertex. However, because of operator precedence in C++, the synchronised vertex expression must be in parentheses inside the edge expression (see Table 3.6).

An edge information structure can be accessed *symmetrically* or *asymmetrically*. The asymmetric access, denoted $\langle a|b\rangle$, allows reading and writing to the half-edge

Algebraic Notation	VV Language	Description
$e_{i,j}$	edge e;	An arbitrary edge structure between vertices. It behaves similar to a C++ struct.
$\langle a\|b\rangle$	(a^b)	Asymmetric access.
$\langle a\|\|b\rangle$	(a|b)	Symmetric access.
$\langle(`a)\|b\rangle$	((`a)^b)	Asymmetric access to the synchronised edge.
$\langle(`a)\|\|b\rangle$	((`a)|b)	Symmetric access to the synchronised edge.

Table 3.6: Operations on edges

information structure $e_{a,b}$. Thus, $e_{a,b} \neq e_{b,a}$ if asymmetric access is used. When a symmetric access is used, denoted $\langle a\|\|b\rangle$, the corresponding edge information is always synchronised to be the same in both directions. It is symmetric in the sense that the two half-edges contain the same information. That is, if $e_{a,b}$ is accessed, $e_{b,a}$ is made equal to $e_{a,b}$.[¶]

The edge feature is used quite frequently in the more advanced modelling cases; the subdivision surfaces with boundaries (Algorithms 5.9 & 5.10), the Penrose tiling (Algorithm 6.2), the Korn-Spalding cell division (Algorithm 7.2), the daffodil corona (§8.2) and the implementation of L-systems as vv programs (§12.1.1) are all examples that use the edge feature.

[¶]Using the above notation we get $\langle a|b\rangle = e_{a,b}$ and $\langle a\|b\rangle = e_{a,b} = e_{b,a}$.

Part II

Geometric Modelling

Chapter 4

Curve Subdivision Algorithms

A simple and useful set of geometric algorithms are *subdivision curves* (initially presented in [6]; see [71, 85] for an overview). A subdivision curve is a curve, composed of vertices connected with line segments, on which a refinement process can be iteratively applied. The refinement process adds vertices and alters the positions of existing vertices by affine transformations such that, at the limit, the curve is continuous and smooth. The curve subdivision algorithms presented here are chosen to illustrate some basic idioms used vv programs, and so the subdivision algorithms chosen are presented in their simplest forms.

The rules used to subdivide the curve are often depcited as a *mask*. A mask is a graphical representation of a transformation to the curve that depicts some segment of the curve and the weights applied to the vertices in an affine combination of the vertex positions to produce the new geometry. But the mask is just a graphical representation, not a programming construct.

A curve subdivision is traditionally implemented by a matrix transformation. For a coarse curve of n vertices, and a refinement that results in a curve with $2n$ vertices, the vertices of the original curve are given in a column matrix of n elements, the transformation is given an n by $2n$ matrix and the resulting refined curve is column matrix of $2n$ elements. For example, the cubic B-spline subdivision algorithm [16] (see §4.1), depicted by the masks in Figure 4.1, applied to a four-point curve is described by the equation

(a) The mask for existing vertices (b) The mask for new vertices

Figure 4.1: The masks for cubic B-spline subdivision

$$\begin{bmatrix} P_1^2 \\ P_2^2 \\ P_3^2 \\ P_4^2 \\ P_5^2 \\ P_6^2 \\ P_7^2 \\ P_8^2 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ \frac{3}{4} & \frac{1}{8} & 0 & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{8} & 0 & \frac{1}{8} & \frac{3}{4} \end{bmatrix} \begin{bmatrix} P_1^1 \\ P_2^1 \\ P_3^1 \\ P_4^1 \end{bmatrix},$$

or in general, for a curve of n points,

$$\begin{bmatrix} P_1^{k+1} \\ P_2^{k+1} \\ P_3^{k+1} \\ P_4^{k+1} \\ P_5^{k+1} \\ \vdots \\ P_{2n-2}^{k+1} \\ P_{2n-1}^{k+1} \\ P_{2n}^{k+1} \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & \cdots & 0 & 0 & \frac{1}{2} \\ \frac{3}{4} & \frac{1}{8} & 0 & 0 & \cdots & 0 & 0 & \frac{1}{8} \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & \cdots & 0 & 0 & 0 \\ \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & 0 & \cdots & 0 & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} & 0 & \cdots & 0 & 0 & 0 \\ 0 & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{1}{8} & \frac{3}{4} & \frac{1}{8} \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{8} & 0 & 0 & 0 & \cdots & 0 & \frac{1}{8} & \frac{3}{4} \end{bmatrix} \begin{bmatrix} P_1^k \\ P_2^k \\ P_3^k \\ P_4^k \\ \vdots \\ P_{n-1}^k \\ P_n^k \end{bmatrix}.$$

This notation is manageable for a small number of points, but as the number of points increases, the matrices also become large and unwieldly. Moreover, a new transformation matrix of size $2n$ by n must be generated for each n that is used.

This example illustrates why a global description of the data is not always ideal for implementing a geometric model. It is especially true in this case because the transformations of the curve subdivision are inherently local. Each transformation is typically applied to a vertex, or pair of vertices, and considers the geometry of those vertices and the neighbouring vertices only. This locality is evident when the mask representation is used.

As illustrated in [56], L-systems can be used to define a curve subdivision algorithm. An L-system production is a concise form to express a subdivision rule using only local relations, similar to the mask representation.

For example, in the L+C language, the L-system that implements cubic B-spline subdivision* is given in Algorithm 4.1. Unlike the matrix representation, the locality of the transformation is evident: the production is applied to each vertex and the context provides access to geometry of the neighbouring vertices.

Algorithm 4.1: Cubic B-spline subdivision implemented as an L-system

```
1  #include <lpfgall.h>
2  V2f v1(0, 0), v2(0, 1), v3(1, 1), v4(1, 0);
3  module P(V2f);
4  ring L-system: 1;
5  derivation length: 3;
6  Axiom: P(v1) P(v2) P(v3) P(v4);
7  P(vl) < P(v) > P(vr) :  {
8      produce P(0.125 * vl + 0.75 * v + 0.125 * vr) P(0.5 * vl + 0.5 * v);
9  }
10 interpretation:
```

*The L-system in Algorithm 4.1 is adapted from that on p. 47 of [56].

```
11  P(v) :  {
12      produce LineTo2f(v) Circle(0.01);
13  }
```

Like L-systems, vv lends itself well to the implementation of subdivision curves. The notion of the predecessor context in L-systems is analogous to the notion of neighbourhoods in vv and the production application is analogous to the neighbourhood editing operations. Therefore, a subdivision curve algorithm can be specified using vv in a similar spirit as an L-system specification, in terms of a vertex and its neighbours.

4.1 Cubic B-Spline Subdivision

One of the simplest curve subdivision algorithms is the cubic B-spline subdivision algorithm [16]. New vertices are added into the curve at the mid-points between existing vertices and the positions of existing vertices are modified according to a weighted average of its own position and of those of its neighbours.

Algorithm 4.2 is a vv implementation of the cubic B-spline subdivision algorithm. Lines 5 and 7 set the new position of an existing vertex as a weighted average of its own position and those of its neighbours. Lines 9 and 10 create a new vertex, insert it between two existing ones, using Algorithm 3.1, and set its position to the midpoint of existing vertices. The comparison in Line ensures that only one vertex is inserted between every pair of neighbouring vertices. The results of the algorithm are shown in Figure 4.2.

Algorithm 4.2: Cubic b-spline subdivision

```
1  void bspline(mesh & V) {
–      // Create a copy of each vertex
2      synchronise V;
3      mesh N;
–      // Iterate all the vertices
4      forall v in V {
5          v$pos *= 0.75;
6          forall p in ‘v {
7              v$pos += ‘p$pos * 0.125;
–              // Make sure the next transformation is applied to each
–              // adjacent pair of vertices exactly once
8              if (p < v) continue;
–              // Insert a new vertex between two existing ones
9              vertex x = insert(v, p);
10             x$pos = ‘v$pos * 0.5 + ‘p$pos * 0.5;
11             add x to N;
12         }
13     }
14     merge V with N;
15 }
```

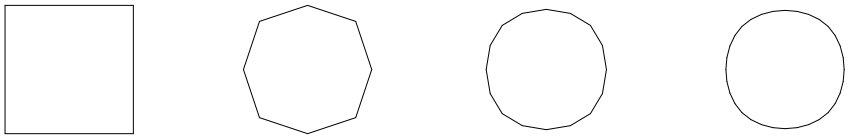


Figure 4.2: B-spline subdivision applied four times

4.2 Dyn-Levin-Gregory Subdivision

With only a small modification to Algorithm 4.2, it is possible to get the Dyn-Levin-Gregory subdivision algorithm [13] (see Figure 4.3 and Algorithm 4.3). The only difference between these algorithms is the mask that defines geometric transformation applied to the vertices. The mask for Dyn-Levin-Gregory subdivision involves two more vertices than that of the cubic B-spline. In particular, lines 5 and 7 of Algorithm

4.2 are removed and line 10 is replaced. Lines 8 and 9 of Algorithm 4.3 implement the new geometric transformation. The results of the algorithm are shown in Figure 4.4.

Algorithm 4.3: Dyn-Levin-Gregory subdivision

```
1  void dlg(mesh & V) {
2      synchronise V;
3      mesh N;
–      // Iterate over the vertices in the curve
4      forall v in V {
5          forall p in ‘v {
6              if (p < v) continue;
–              // Insert a new vertex between each adjacent pair of vertices
7              vertex x = insert(p, v);
8              x$pos = (nextto p in ‘v)$pos * -0.0625 + (nextto v in ‘p)$pos * -0.0625
9                  + p$pos * 0.5625 + v$pos * 0.5625;
10             add x to N;
11         }
12     }
13     merge V with N;
14 }
```

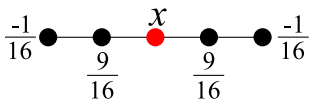


Figure 4.3: The Dyn-Levin-Gregory subdivision mask

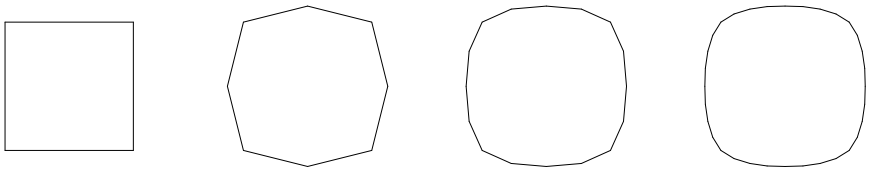


Figure 4.4: Dyn-Levin-Gregory subdivision applied four times

4.3 Chaikin Subdivision

Another commonly used approximating curve subdivision algorithm is the Chaikin corner-cutting algorithm [6] (see Figure 4.5). Unlike the previous two subdivision schemes, the Chaikin subdivision scheme does not retain the existing vertices of the curve, but "cuts" them from the curve. This is analogous to cutting the corners off of a sheet of paper to make rounded corners.[†]

The vv implementation (Algorithm 4.4) uses two passes over the curve. The first pass (lines 5 – 14) inserts two new vertices between each pair of neighbouring vertices. The second pass (lines 15 – 20) removes the old vertices from the neighbourhoods of the new ones. The results of the algorithm are shown in Figure 4.6.

Algorithm 4.4: Chaikin subdivision

```
1  void chaikin(mesh & V) {
2      synchronise V;
3      mesh N;
-      // Insert two new vertices one each edge
4      forall v in V {
5          forall p in ‘v {
6              if (p < v) continue;
7              vertex a = insert(p, v);
8              vertex b = insert(a, p);
9              a$pos = v$pos * 0.75 + p$pos * 0.25;
10             b$pos = v$pos * 0.25 + p$pos * 0.75;
11             add a to N;
12             add b to N;
13         }
14     }
-      // Remove the old vertices from the neighbourhoods of the new ones
15     forall v in V {
16         vertex a = any in v;
17         vertex b = nextto a in v;
18         replace v with b in a;
```

[†]It is possible to also consider the Chaikin algorithm as one that inserts new vertices and repositions old ones, much the same as the previous two algorithms. It is here presented as a corner cutting algorithm to demonstrate a different technique in altering a curve using vv.

```
19          replace v with a in b;
20      }
21      V = N;
22  }
```

(a) The mask for the first new vertex

(b) The mask for the second new vertex

Figure 4.5: The masks for Chaikin subdivision

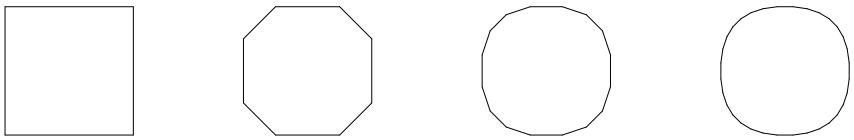


Figure 4.6: Chaikin subdivision applied four times

4.4 On the Use of VV for Implementing Subdivision Curves

The preceding examples illustrate that vv is well-suited to implement the transformations of a subdivision curve while retaining their local nature. The neighbourhood is traversed iteratively and the next and previous operations are used to gather the information about the neighbouring vertices. The editing operations of the algebra are convenient for replacing or inserting vertices into the curve.

However, in comparing the implementations of the cubic B-spline subdivision curve (Algorithms 4.1 & 4.2), it can be seen that the vv implementation is longer than the L+C implementation and with no significant improvement in simplicity or readability. It should not be surprising: L-systems were designed to model linear and branching structures.

Instead, the examples demonstrate some basic techniques of using vv to gather some data in the structure and effecting transformations using a simple class of geometric modelling algorithms. These basic techniques illustrated in these simple examples are used often in examples that follow in this thesis. In the next chapter, it will be seen that vv programs, not significantly more complex, can be used to implement *subdivision surfaces* using the same principles.

Chapter 5

Surface Subdivision Algorithms

Subdivision surfaces (initially presented in [5, 11]; see [71, 85, 88] for an overview), akin to subdivision curves, are a method for refining polygon meshes. Like subdivision curves, a subdivision surface is refined by an affine transformation that uses a vertex and its surrounding neighbouring vertices as input.*

5.1 Subdivision Algorithms on Triangular Meshes

5.1.1 Polyhedral subdivision

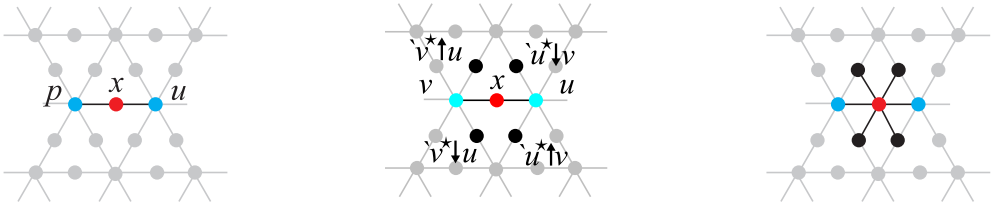
One of the simplest subdivision algorithms is the polyhedral subdivision of triangular meshes [71]. The algorithm inserts a new vertex at the midpoint of each edge, and divides each triangle of the mesh into four co-planar triangles. The overall shape of the initial polyhedron does not change, but the faces are subdivided.

Algorithm 5.1: Polyhedral Subdivision

```
1  void polyhedral(mesh & V) {
2      mesh N;
3      synchronise V;
-      // Insert a new vertex at the midpoint between each
-      // pair of adjacent vertices
4      forall v in V {
5          forall u in ‘v {
6              if (u < v) continue;
7              vertex x = insert(u, v);
8              x$pos = 0.5 * (u$pos + v$pos);
9              add x to N;
```

*Parts of the text in §5.1.1 – §5.1.4 are adapted from [69].

```
10          }
11      }
  –     // Assign a neighbourhood to each new vertex
12      forall v in N  {
  –         // Get the two vertices adjacent to the new vertex
13          vertex a = any in v;
14          vertex b = nextto a in v;
  –         // Assign the final neighbourhood to the new vertex
15          make { nextto v in a, a, prevto v in a, nextto v in b, b, prevto v in b } nb_of v;
16      }
17      merge V with N;
18  }
```



(a) The vertices p and q are used to create the vertex x

(b) The identification of vertices that will become neighbours of x

(c) The updated neighbourhood of the new vertex x

Figure 5.1: The polyhedral subdivision process illustrated. The variable names correspond to those in Algorithm 5.1.

The vv program that implements polyhedral subdivision consists of two loops (Algorithm 5.1). The first loop (lines 4 – 11) iterates over pairs of neighbouring vertices in the vertex set V. The condition $u < v$ in line 6 assures that each vertex pair (*i.e.* edge of the polygon mesh) is considered only once. New vertices are inserted at the midpoint of each edge (line 7) and added to the set N (line 9). The second loop (lines 12 – 16) inserts new edges by redefining the neighbourhoods of the new points. The intervening neighbourhoods and the result of insertion are shown in Figure 5.1. Finally, the set of vertices V is merged with the set of new vertices,

N, to complete the refinement of the polygon mesh. An example of a polygon mesh and the results of its polyhedral subdivision are shown in Figure 5.2.

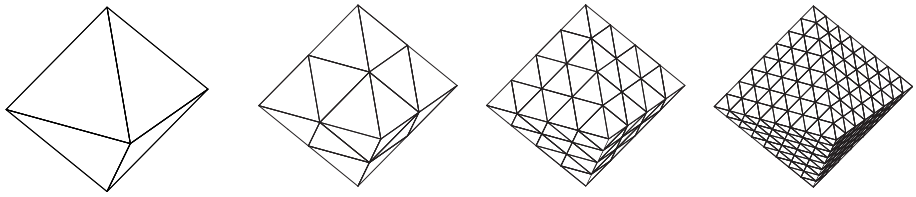


Figure 5.2: Polyhedral subdivision applied to a polygon mesh three times

5.1.2 Loop algorithm

The Loop subdivision scheme [35] is topologically equivalent to the polyhedral subdivision scheme, in the sense that both operate on triangular meshes and subdivide a triangular face into four triangles in every iteration step. The vv implementations of both schemes therefore have a similar structure. The difference is in the positioning of vertices; the Loop subdivision scheme aims at constructing a smooth surface that approximates the initial polyhedron (Figure 5.3). The Loop algorithm places new vertices using a mask involving four old vertices. The mask for new vertices is shown in Figure 5.4a and the expressions of the vv algebra used to find the vertices for that mask are shown in Figure 5.4b. Existing vertices are repositioned using a mask that incorporates all of their immediate neighbours (Figure 5.4c). A vv implementation of the Loop subdivision algorithm for closed surfaces is given in Algorithm 5.2 and the corresponding masks are given in Figure 5.3.

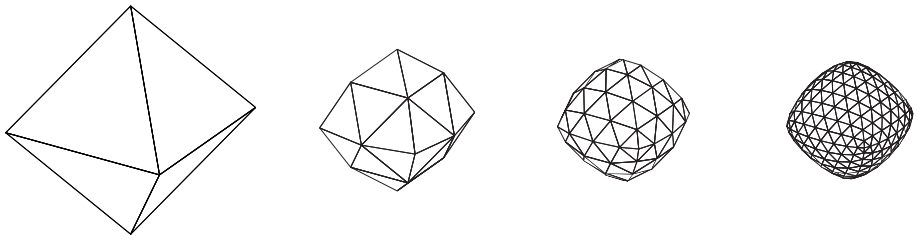
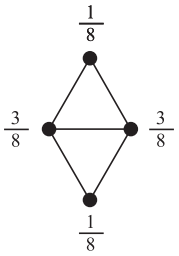
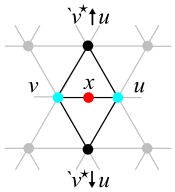


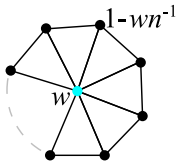
Figure 5.3: Loop subdivision applied to a polygon mesh three times



(a) The mask for a new vertex



(b) The identifications of vertices for the application of the mask in (a)



(c) The Loop subdivision mask for old vertices

Figure 5.4: The Loop subdivision masks

Algorithm 5.2: Loop subdivision

```
1   void loop(mesh & V)  {
2       mesh N;
3       synchronise V;
–       // Insert a new vertex on each edge and calculate the new positions
–       // of existing vertices
4       forall v in V  {
–           // Calculate the weight to apply to the position of neighbouring
–           // vertices for the affine transformation to the position of v
5           unsigned int n = valence v;
6           double w = std::pow(0.375 + 0.25 * std::cos(2.0 * util::pi / double(n)), 2.0) + 0.375;
–           // Declare the point Q, see §3.2 and §B.2.1
7           Pt Q;
8           forall u in `v  {
9               Q += `u$pos;
10              if (u < v) continue;
11              vertex x = insert(u, v);
12              x$pos = `v$pos * 0.375 + `u$pos * 0.375
```

```
13                  + ‘(nextto u in ‘v)$pos * 0.125 + ‘(prevto u in ‘v)$pos * 0.125;
14              add x to N;
15          }
16          Q /= double(n);
17          v$pos = w * v$pos + (1.0 - w) * Q;
18      }
 –      // Assign the neighbourhoods to the new vertices
19      forall v in N  {
20          vertex a = any in v;
21          vertex b = nextto a in v;
22          make { nextto v in a, a, prevto v in a, nextto v in b, b, prevto v in b } nb_of v;
23      }
24      merge V with N;
25  }
```

5.1.3 Butterfly algorithm

The butterfly subdivision algorithm [14], like that for Loop subdivision, is topologically equivalent to polyhedral subdivision. In contrast to Loop subdivision, which approximates the shape of the initial polyhedron, the butterfly algorithm is an interpolating scheme. Consequently, the old vertex positions are not adjusted in the course of the algorithm.

The butterfly algorithm uses a more extensive mask for the new vertices, which includes points outside the immediate neighbourhood of the subdivided edge. This mask and the complete vv implementation of the butterfly algorithm for closed surfaces are presented in Algorithm 5.3 and the masks in Figure 5.5. An example application of the algorithm is illustrated in Figure 5.6.

The use of the indexed next and previous operations (see §3.5.1) simplifies the expression of the subdivision mask (lines 10 and 11 of Algorithm 5.3). For example, the expression

next(2)to u **in** ‘v

from line 10 in Algorithm 5.3 would have to be written as

nextto nextto u **in** v **in** v

if the indexed notation were not used.

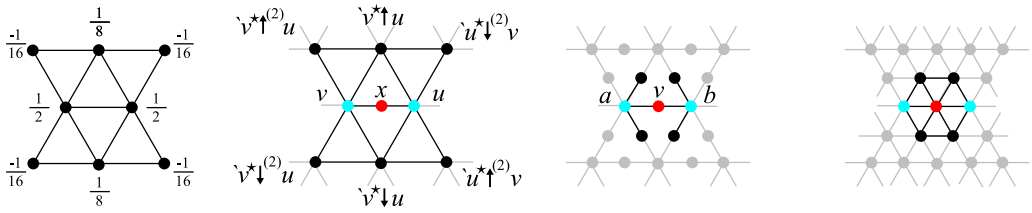


Figure 5.5: The butterfly subdivision masks. From left to right: The butterfly mask, the vv expressions to get the vertices for the mask, the resulting new vertices and the resulting mesh after the new vertices are connected.

Algorithm 5.3: Butterfly subdivision

```
1  void butterfly(mesh & V) {
2      mesh N;
3      synchronise V;
       // Insert a new vertex on each edge
4      forall v in V {
5          forall u in `v {
6              if (u < v) continue;
7              vertex x = insert(u, v);
8              x$pos = v$pos * 0.5 + u$pos * 0.5
9                  + (prevto u in `v)$pos * 0.125 + (nextto u in `v)$pos * 0.125
10                 - (next(2)to u in `v)$pos * 0.0625 - (prev(2)to u in `v)$pos * 0.0625
11                 - (next(2)to v in `u)$pos * 0.0625 - (prev(2)to v in `u)$pos * 0.0625;
12             add x to N;
13         }
14     }
       // Assign the neighbourhood to each new vertex
15     forall v in N {
16         vertex a = any in v;
17         vertex b = nextto a in v;
18         make { nextto v in a, a, prevto v in a, nextto v in b, b, prevto v in b } nb_of v;
19     }
20     merge V with N;
21 }
```

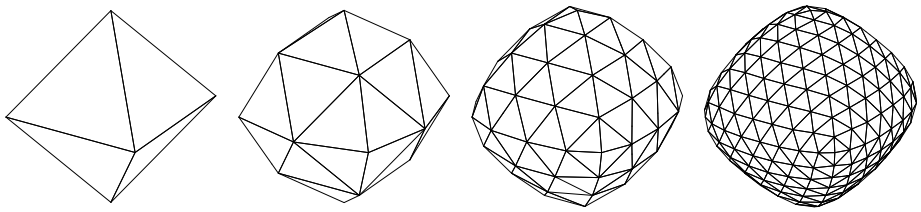


Figure 5.6: Three applications of butterfly subdivision to a triangular mesh

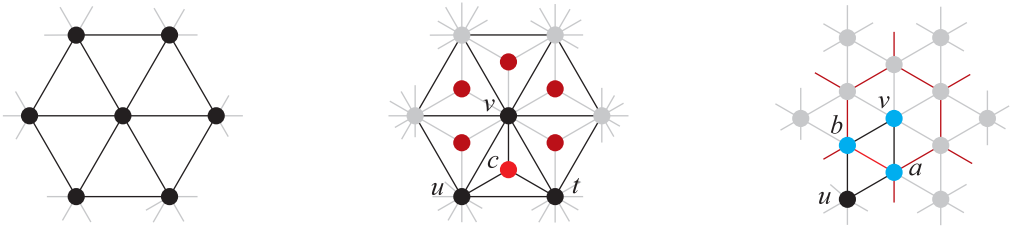
5.1.4 $\sqrt{3}$ Subdivision

Kobbelt's $\sqrt{3}$-subdivision algorithm [27] is an example of a scheme that changes the topology of a triangular mesh in a manner different from polyhedral subdivision. The vv specification of the $\sqrt{3}$-subdivision algorithm is given in Algorithm 5.4 and the transformations are illustrated in Figure 5.7. In the first loop (lines 7 – 17), a new vertex c is created at the centroid of each triangle. The neighbourhoods are then updated such that each triangle is divided into three, that is each vertex v, u, t of the original triangle is connected to c, and the vertices v, u, t form the neighbourhood of c (lines 12 – 16). In the second loop (lines 19 – 30), the topology is updated by "flipping" all the edges between pairs of old vertices. An example of the operation of the algorithm is shown in Figure 5.8.

Algorithm 5.4: $\sqrt{3}$ Subdivision

```
1  void sqrt3(mesh & V) {
2      mesh N;
3      synchronise V;
4      forall v in V {
5          double w = (4.0 - 2.0 * std::cos(util::two_pi / double(valence ‘v))) / 9.0;
6          v$pos *= (1.0 - w);
–          // Place a new vertex at the centre of each triangle
7          forall u in ‘v {
8              v$pos += ‘u$pos * w / double(valence ‘v);
9              vertex t = nextto u in ‘v;
10             if (u < v || t < v) continue;
11             vertex c((‘v$pos + ‘u$pos + ‘t$pos) / 3.0);
```

```
12              make { v, u, t } nb_of c;
13              splice c after u in v;
14              splice c after t in u;
15              splice c after v in t;
16              add c to N;
17          }
18      }
  –     // Flip all the edges
19      forall v in V  {
20          forall u in ‘v  {
21              if (u < v) continue;
22              vertex a = nextto u in v;
23              vertex b = prevto u in v;
24              splice a after u in b;
25              splice b after v in a;
26              erase u from v;
27              erase v from u;
28          }
29      }
30      merge V with N;
31  }
```



(a) A portion of the original mesh

(b) New vertices are inserted at the centre of each triangle

(c) The edges are flipped

Figure 5.7: The $\sqrt{3}$ subdivision process illustrated. The variable names correspond to those in Algorithm 5.4.

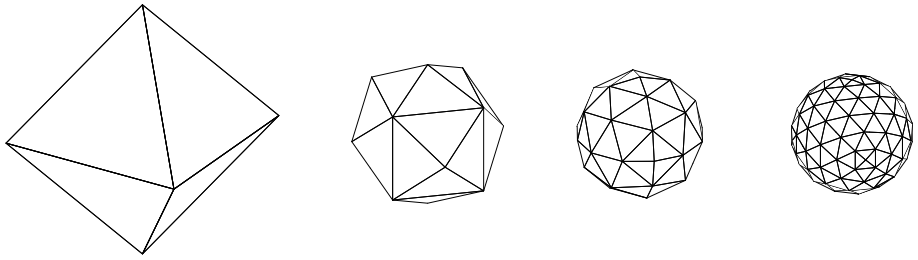


Figure 5.8: $\sqrt{3}$ subdivision applied to a polygon mesh three times

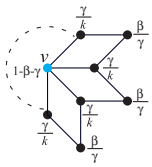
5.2 Subdivision Algorithms on Quadrilateral and Mixed-Polygon Meshes

Of course, not all polygon meshes are composed of triangles. It is also common to have polygon meshes of quadrilaterals and sometimes polygons of other sizes.

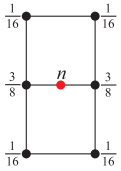
The Catmull-Clark subdivision algorithm [5] can be used to subdivide meshes composed of quadrilaterals and the Doo-Sabin algorithm [11] can be used to subdivide meshes that are primarily quadrilaterals, but may also contain polygons of arbitrary size.

5.2.1 Catmull-Clark Subdivision

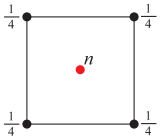
Catmull-Clark subdivision (Figure 5.9 and Algorithm 5.5) is a face-splitting algorithm that modifies the positions of existing vertices (lines 4 – 15) as an average of the surrounding quadrilaterals and inserts new vertices on each edge (lines 16 – 27) and at the centre of each face (lines 29 – 49). The results of the algorithm are shown if Figure 5.10.



(a) The masks for existing vertices where k is the valence of the vertex, $\beta = \frac{3}{2k}$ and $\gamma = \frac{1}{4k}$.



(b) The masks for new vertices on the edges



(c) The masks for new vertices on the faces

Figure 5.9: The Catmull-Clark subdivision masks

Algorithm 5.5: Catmull-Clark subdivision

```
1  void catmull_clark(mesh & V) {
2      synchronise V;
3      mesh N;
       // Set the new position of existing vertices
4      forall v in V {
5          double k = valence v;
6          double beta = 3.0 / (2.0 * k);
7          double gamma = 1.0 / (4.0 * k);
8          v$pos = `v$pos * (1.0 - beta - gamma);
9          double b_k = beta / k;
10         double g_k = gamma / k;
11         forall p in v {
12             v$pos += `p$pos * b_k;
13             v$pos += `(nextto v in p)$pos * g_k;
14         }
15     }
       // Insert new vertices on each edge
16     forall v in V {
17         forall p in v {
18             if (v < p) continue;
19             vertex a = nextto p in `v;
20             vertex b = prevto p in `v;
21             vertex c = nextto v in `p;
22             vertex d = prevto v in `p;
23             vertex n = insert(v, p);
24             n$pos = (`v$pos + `p$pos) * 0.375
25                 + (`a$pos + `b$pos + `c$pos + `d$pos) * 0.0625;
26             add n to N;
27         }
28     }
       // Connect the new vertices to get quadrilaterals
```

```
29      forall v in V  {
30          forall p in v  {
  –             // Find the vertices that complete the quadrilaterals, but
  –             // since these paths can fit on the mesh in several ways
  –             // test to make sure that the paths terminate at the correct
  –             // vertices
31              vertex x = prevto p in v;
32              vertex a = prevto v in x;
33              if (v < a) continue;
34              vertex y = prevto x in a;
35              vertex b = prevto a in y;
36              if (v < b) continue;
37              vertex z = prevto y in b;
38              vertex c = prevto b in z;
39              if (v < c) continue;
40              vertex n;
41              add n to N;
42              n$pos = (`v$pos + `a$pos + `b$pos + `c$pos) * 0.25;
43              make { p, x, y, z } nb_of n;
44              splice n after a in x;
45              splice n after b in y;
46              splice n after c in z;
47              splice n after v in p;
48          }
49      }
50      merge V with N;
51  }
```

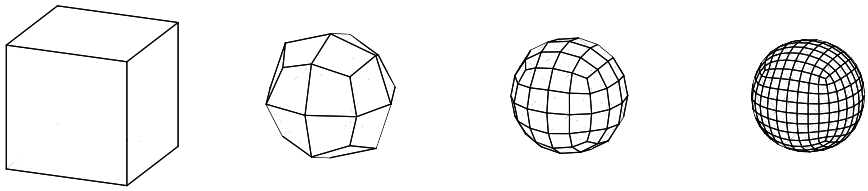


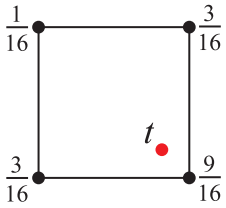
Figure 5.10: Catmull-Clark subdivision applied three times

5.2.2 Doo-Sabin Subdivision

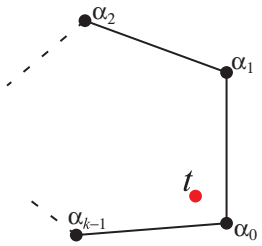
Doo-Sabin subdivision (Figure 5.11 and Algorithm 5.6) is a corner-cutting surface subdivision, similar to the corner cutting that is found in the Chaikin curve subdivision (see 4.3).

The vv implementation operates in two phases. Firstly, a new vertex is added to each face for each of the existing vertices (lines 4 – 25). Different masks need to be applied for the case of a new vertex on a quadrilateral face and on faces of any other size. As there is no direct storage of faces in the vv data structure, the first important step is to determine how many vertices are in the current face. This is determined by walking around the vertices and incrementing a counter (lines 12 – 18). The actual walking is done with the statement on line 17. The use of a path statement (see §3.5.2) allows this to be done in a single statement. A second walk around the face (lines 21 – 31) is used to apply the subdivision mask. These walks are polygon orbits, as described in §3.1.

Secondly, after the new vertices are in place, the neighbourhoods of the new vertices are set (lines 36 – 44) and the old vertices are discarded (line 45). The results of the algorithm are shown in Figure 5.12.



(a) The mask for a new vertex on a face with four vertices



(b) The mask for a new vertex on a face with k vertices, where $\alpha_0 = {}^1\!/_4 + {}^1\!/_{4k}$ and $\alpha_i = (3 + 2\cos({}^{2i\pi}\!/_k))/4k$

Figure 5.11: The Doo-Sabin subdivision masks

Algorithm 5.6: Doo-Sabin subdivision

```
1  void doo_sabin(mesh & V) {
2      synchronise V;
3      mesh N;
```

```
–   // Create the new vertices
4   forall v in V {
5       forall p in ‘v {
6           vertex t;
7           vertex q = nextto p in v;
8           vertex r = nextto v in ‘q;
9           if (r == prevto v in ‘p)
–               // The orbit is a quadrilateral
10              t$pos = v$pos * 0.5625 + p$pos * 0.1875 + q$pos * 0.1875 + r$pos * 0.0625;
11          else {
–               // The orbit is an arbitrary polygon
12              unsigned int k = 0;
13              vertex a = v;
14              vertex b = q;
–               // Count the number of vertices in the orbit
15              do {
16                  ++k;
17                  ‘@&(a, b) swap next @;
18              }
19              while (a != v);
20              unsigned int i = 0;
–               // Walk over the orbit to find the new position
–               // of the new vertex
21              do {
22                  if (i == 0)
23                      t$pos += a$pos * (0.25 + 1.25 / double(k));
24                  else
25                      t$pos += a$pos * ((3.0 + 2.0
26                          * std::cos(util::two_pi * double(i) / double(k))) / double(4.0 * k));
27                  ‘@&(a, b) swap next @;
28                  ++i;
29              }
30              while (a != v);
31          }
–           // Connect the new vertex to the mesh
32          splice t after p in v;
33          add t to N;
34      }
35  }
–   // Complete the neighbourhoods of the new vertices
36  forall v in V {
37      forall p in ‘v {
38          vertex n = nextto p in v;
39          make {
40              prevto v in p, nextto v in nextto n in v,
41              nextto nextto n in v in v, prevto p in v
42          } nb_of n;
43      }
```

```
44      }
45      V = N;
46  }
```

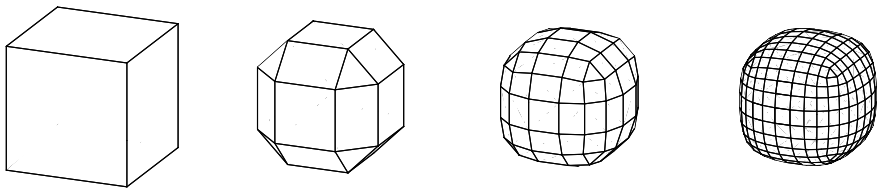


Figure 5.12: Doo-Sabin subdivision applied three times

5.3 Surfaces with Boundaries and Creases

In subdivision surface algorithms, there is typically an alternate mask to deal with vertices that are on the boundaries and creases of a polygon mesh. The boundary mask is typically the subdivision curve mask corresponding to the subdivision scheme. For example, the Catmull-Clark subdivision scheme is based on cubic B-spline patches and so uses the cubic B-spline subdivision curve (see §4.1) for the boundaries. Creases and other sharp features can similarly be subdivided using a subdivision curve mask. These were first introduced by Hoppe *et al.* [24] for the Loop subdivision scheme.

One method to deal easily with the crease case in vv is to use the edge information structures (see §3.5.4). A simple boolean flag in the edge structure can be used to flag where the creases are. When a crease is encountered, the appropriate special subdivision mask can be applied instead of the regular mask.

However, handling boundaries on the surface requires a bit more care. Recall from the definition of the vv data structure that faces are not stored explicitly and

therefore the edges that form a boundary of a mesh is also an orbit and is thus also a polygon. To differentiate a boundary of a mesh from a face, some application-specific criteria must be included. Two methods for defining a boundary of a mesh follow. Both methods are appropriate for use with subdivision surfaces and many other modelling applications.

5.3.1 Boundary Inference from Topology

With subdivision surfaces, it can often be assumed that a mesh is composed entirely of triangles or of triangles and quadrilaterals. Therefore, any orbit of vertices that is not a triangle or a quadrilateral can be assumed to be a boundary. In general, an algorithm can assume that no polygon in the mesh has no more than n sides. Then, any orbit of vertices of more than n vertices is not a polygon to be considered by the algorithm.

When all the polygons in the mesh are assumed to be triangles, the situation is quite simple. If a pair of vertices share a neighbour on one side, then there is a triangle there (see Figure 5.13). Algorithm 5.7 implements this test to check for a face on either side of an adjacent pair of vertices a and b. The algorithm tests each side of the pair of vertices and if it finds a face it executes the function `f`, which can be any arbitrary function object that takes the vertices of a triangle as arguments. If one of the tests in the algorithm fails, then the pair of vertices lie on a boundary of the mesh.

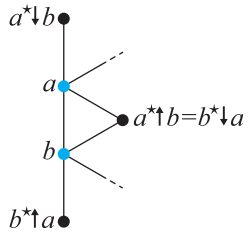


Figure 5.13: On the right of the vertices a and b, the triangle check succeeds because $a^\star \uparrow b = b^\star \downarrow a$, but on the left, it fails because $a^\star \downarrow b \neq b^\star \uparrow a$

Algorithm 5.7: Triangle Test for an Adjacent Pair of Vertices

```
1  bool forEachFace(vertex a, vertex b, function f) {
2      bool on_boundary = false;
3      vertex x = nextto b in a;
4      if (x == prevto a in b) f(a, b, x);
5      else on_boundary = true;
6      vertex y = prevto b in a;
7      if (y == nextto a in b) f(b, a, y);
8      else on_boundary = true;
9      return on_boundary;
10 }
```

An implementation of Loop subdivision with boundaries is given in Algorithm 5.8 and the results are illustrated in Figure 5.14. The boundaries of a Loop subdivision surface are cubic B-spline subdivision curves; therefore the cubic B-spline masks, illustrated in Figure 4.1, are used for vertices on the boundaries.

In the algorithm, the triangle check is used in three places. Firstly, it is used to decide which mask to apply to the existing vertices of the mesh. If the test in lines 13 or 18 fail, then the mask for the boundary is used along the appropriate pair of edges. Secondly, in line 33, the test is used to determine if the boundary mask for new vertices is used. Finally, in lines 43 and 47, if the test succeeds, the new edges for the interior are connected.

Algorithm 5.8: Loop subdivision with boundaries

```
1  void loop(mesh & V) {
2      mesh N;
3      synchronise V;
       // Insert new vertices on each edge and adjust the positions
       // of existing vertices
4      forall v in V {
5          unsigned int n = valence v;
6          double w = std::pow(0.375 + 0.25 * std::cos(2.0 * util::pi / double(n)), 2.0) + 0.375;
7          Pt Q;
8          bool sharp = false;
           // Set the new positions of the existing vertices
9          forall u in `v {
10             Q += `u$pos;
               // Discover if the edge is part of a boundary
11             vertex l = prevto u in `v;
12             vertex r = nextto u in `v;
13             if (l != nextto v in `u) {
14                 v$pos = 0.75 * `v$pos + 0.125 * `u$pos + 0.125 * `l$pos;
15                 sharp = true;
16                 break;
17             }
18             else if (r != prevto v in `u) {
19                 v$pos = 0.75 * `v$pos + 0.125 * `u$pos + 0.125 * `r$pos;
20                 sharp = true;
21                 break;
22             }
23         }
           // If it is an interior edge, apply the regular case
24         if (!sharp) {
25             Q /= double(n);
26             v$pos = w * v$pos + (1.0 - w) * Q;
27         }
           // Insert a new vertex on each edge, with the position
           // dependent on whether it is on a boundary
28         forall u in `v {
29             if (u < v) continue;
30             vertex x = insert(u, v);
31             vertex l = prevto u in `v;
32             vertex r = nextto u in `v;
33             if (l == nextto v in `u && r == prevto v in `u)
34                 x$pos = `v$pos * 0.375 + `u$pos * 0.375 + `l$pos * 0.125 + `r$pos * 0.125;
35             else
36                 x$pos = `v$pos * 0.5 + `u$pos * 0.5;
37             add x to N;
38         }
```

```
39      }
  –     // Complete the neighbourhoods of existing vertices
  –     // dependent on whether it is on a boundary
40      forall v in N  {
41          vertex a = any in v;
42          vertex b = nextto a in v;
43          if (nextto b in `a == prevto a in `b)  {
44              splice nextto v in a before a in v;
45              splice prevto v in b after b in v;
46          }
47          if (prevto b in `a == nextto a in `b)  {
48              splice prevto v in a after a in v;
49              splice nextto v in b before b in v;
50          }
51      }
52      merge V with N;
53  }
```

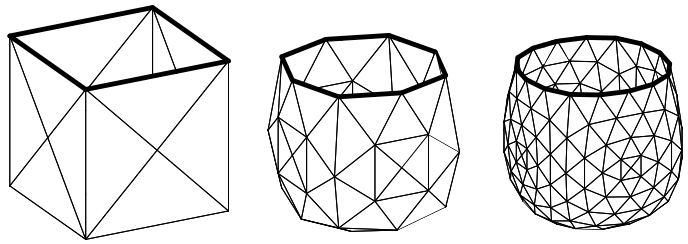


Figure 5.14: A surface with a boundary, marked with a thick line, subdivided twice

This method can easily be extended to check for any other polygon of a specific number of vertices. Alternately, a walk around the face, as was done in the Doo-Sabin subdivision algorithm (Algorithm 5.6, lines 15 – 18), could be used.

Note that this method is simplest when there is a mesh composed entirely of triangles. With polygons of more vertices or with meshes of mixed polygons, the method becomes more complex. Also, this method is not compatible with boundaries that are the same size as a face in the mesh. However, this method has the advantage that it does not require any special marking on the boundaries; it can operate on

any supplied mesh.

5.3.2 Explicit Boundary Demarcation

The second method to handle boundaries is to specially marked boundaries on the edge information structures (see §3.5.4). Using asymmetric accesses, the outside half-edge is marked as a boundary and half-edges on the interior of the mesh are marked differently (see Figure 5.15). Algorithm 5.9 demonstrates how a boolean flag on the edges can be used.

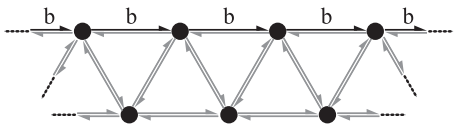


Figure 5.15: The half-edges marked with a 'b' determine the boundary of the mesh (marks on the interior edges are not shown)

Algorithm 5.9: The Use of Edge Information for Boundaries

```
1  bool forEachFace(vertex a, vertex b, function f)  {
2      bool on_boundary = false;
3      if ((a^b).boundary) f(a, b, nextto b in a);
4      else on_boundary = true;
5      if ((b^a).boundary) f(b, a, nextto a in b);
6      else on_boundary = true;
7      return on_boundary;
8  }
```

An implementation of Loop subdivision with both boundaries and creases marked on the edges is given in Algorithm 5.10 and the results are illustrated in Figure 5.16.

The use of explicit information in the edge structures to determine the boundaries can be compared to inferring the boundaries by comparing the statements for edge accesses to the statements for the triangle tests. For example, the use of edge

information in lines 29 and 33 of Algorithm 5.10 correspond to the triangle tests in lines 13 and 18 in Algorithm 5.8.

Also in this algorithm is the application of the crease masks from [24]. Because the edge structures are already used to mark the boundaries, an edge can be marked as a crease simply by adding an additional boolean flag to the edge structure. The application of a crease mask to a new vertex is exactly the same as for the boundary case (lines 47 and 48). For an existing vertex on a crease, the application is a little trickier. To determine which mask to apply, the number of edges marked as creases that are incident to a particular vertex must be counted (lines 14 – 22). If there are two edges, then same mask as the boundary is applied (line 18). If the vertex is at a *dart*, only one edge marked as a crease at the vertex, the regular subdivision mask is applied (line 40). Finally, if three or more incident edges are marked as creases, the vertex is at a *corner* and its position is not changed (lines 19 – 20).

The other major difference from Algorithm 5.8 is that now the edge information must be maintained for new edges. Edge information is propagated in lines 52 and 53 by copying the old edge structures to the new edges. Note that in 60 – 67, new edges are created, but these edges will never be boundaries or creases, so the default edge structure is sufficient.

Algorithm 5.10: Loop subdivision with boundaries and creases

```
1  void loop(mesh & V)  {
2      mesh N;
3      synchronise V;
-      // Adjust the positions of existing vertices
4      forall v in V  {
5          unsigned int n = valence v;
6          double w = std::pow(0.375 + 0.25 * std::cos(2.0 * util::pi / double(n)), 2.0) + 0.375;
7          Pt Q;
```

```
8         bool sharp = false;
9         forall u in v  {
10            Q += `u$pos;
11            if ((v^u).crease)  {
–                 // Discover the number of edge features that are at this vertex
12                vertex a = nextto u in v;
13                unsigned int creases = 1;
14                while (a != u && creases < 3)  {
15                    if ((v^a).crease || (v^a).boundary || (a^v).boundary)  {
16                        creases++;
17                        if (creases == 2)
18                            v$pos = 0.75 * `v$pos + 0.125 * `u$pos + 0.125 * `a$pos;
19                        else
20                            v$pos = `v$pos;
21                    }
22                    a = nextto a in v;
23                }
24                if (creases > 1)  {
25                    sharp = true;
26                    break;
27                }
28            }
–             // Check if the edge is on a boundary
29            else if ((v^u).boundary)  {
30                v$pos = 0.75 * `v$pos + 0.125 * `u$pos + 0.125 * `(nextto u in v)$pos;
31                sharp = true;
32            }
33            else if ((u^v).boundary)  {
34                v$pos = 0.75 * `v$pos + 0.125 * `u$pos + 0.125 * `(prevto u in v)$pos;
35                sharp = true;
36            }
37        }
–         // If the vertex is not on a boundary or a crease, apply
–         // the regular Loop mask
38        if (!sharp)  {
39            Q /= double(n);
40            v$pos = w * v$pos + (1.0 - w) * Q;
41        }
42    }
–     // Insert new vertices on each edge
43    forall v in V  {
44        forall u in `v  {
45            if (u < v) continue;
46            vertex x = insert(u, v);
–             // If the vertex in on a boundary or crease, set its
–             // position to be the midpoint. Otherwise use the
–             // regular Loop mask
47            if ((((`v)|u).crease || ((`v)^u).boundary || ((`u)^v).boundary)
```

```
48              x$pos = ‘v$pos * 0.5 + ‘u$pos * 0.5;
49          else
50              x$pos = ‘v$pos * 0.375 + ‘u$pos * 0.375
51                  + ‘(nextto u in ‘v)$pos * 0.125 + ‘(prevto u in ‘v)$pos * 0.125;
52          (v^x) = (x^u) = ((‘v)^u);
53          (u^x) = (x^v) = ((‘u)^v);
54          add x to N;
55      }
56  }
-   // Complete the neighbourhoods of the new vertices
57  forall v in N  {
58      vertex a = any in v;
59      vertex b = nextto a in v;
60      if (!((‘a)|b).boundary)  {
61          splice nextto v in a before a in v;
62          splice prevto v in b after b in v;
63      }
64      if (!((‘b)|a).boundary)  {
65          splice prevto v in a after a in v;
66          splice nextto v in b before b in v;
67      }
68  }
69  merge V with N;
70 }
```

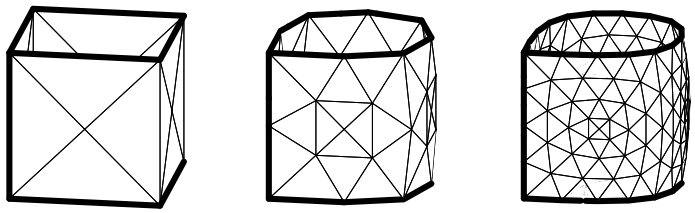


Figure 5.16: A surface with a boundary and creases, marked with the thick lines, subdivided twice

Using explicitly marked boundaries allows for more flexibility than in §5.3.1, as any minimal cycle of vertices in the mesh can be explicitly marked as either a boundary or as part of the interior. Also, since this method already uses edge structures, it is a trivial to add other edge properties, such as marking edges as creases. However, this method requires that the mesh be properly preprocessed by

having the boundaries explicitly encoded in the edge information structures, and that information must be properly maintained at each application of the algorithm.

5.4 Subdivision of Non-Orientable Manifold Surfaces

Up to this point, all the subdivision surface algorithms considered have operated on surfaces that are orientable. In general, it is usually the case that the surfaces used for geometric modelling are orientable. But, it is possible to use vv to create algorithms that operate on non-orientable surfaces. To demonstrate this, an algorithm that subdivides a Möbius strip [45, 41] follows.

The surface of a Möbius strip is not orientable; however, the neighbourhood of a vertex in vv is always oriented because the vertices are listed in counter-clockwise order. That is, the ordering of the vertices locally defines the orientation of the surface. Therefore, a surface in vv that is visually similar to a Möbius strip is oriented everywhere locally, but it may have inconsistent orientation globally. A surface can be inconsistently oriented globally because the surface is discrete.

An implementation of the Warren variation of Loop subdivision [85, §7.3.2] for a Möbius strip is given in Algorithm 5.11. The results are illustrated in Figure 5.18.

This algorithm operates like the regular subdivision with boundaries (see §5.3), except that prior to the application of the subdivision mask, the algorithm must check to ensure that the vertices considered are consistently oriented. Note that because the interior subdivision masks are symmetric, only the boundary cases and the assembly of neighbourhoods around new vertices require special consideration (see Figure 5.17). The largest difference in the implementation can be seen by

comparing lines 47 – 62 of Algorithm 5.11 to lines 43 – 51 of Algorithm 5.8. Here, it is necessary to include two extra cases to create the neighbourhood around the new vertex when the adjacent old vertices have neighbourhoods oriented oppositely. Otherwise, there are only minor differences between the two algorithms.

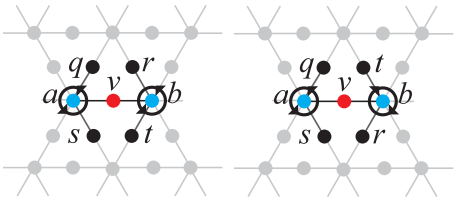


Figure 5.17: Illustrations of when the neighbourhoods of adjacent vertices are consistently oriented (left) and inconsistently oriented (right)

Algorithm 5.11: Loop subdivision for a Möbius strip

```
1   void loop_mobius(mesh & V) {
2       mesh N;
3       synchronise V;
–       // Adjust the position of existing vertices
4       forall v in V {
5           bool sharp = false;
6           Pt Q;
7           forall u in v {
8               vertex l = nextto u in v;
9               vertex r = prevto u in v;
–               // Check if this edge is on a boundary
10              if (!is l in u) {
11                  v$pos = 0.75 * v$pos + 0.125 * `u$pos + 0.125 * `l$pos;
12                  sharp = true;
13                  break;
14              }
15              else if (!is r in u) {
16                  v$pos = 0.75 * v$pos + 0.125 * `u$pos + 0.125 * `r$pos;
17                  sharp = true;
18                  break;
19              }
20              else Q += `u$pos;
21          }
22          if (!sharp) {
23              Q /= double(valence v);
24              v$pos = 0.625 * v$pos + 0.375 * Q;
```

```
25          }
26      }
-       // Insert a new vertex on each edge
27      forall v in V  {
28          forall u in ‘v  {
29              if (u < v) continue;
30              vertex x = insert(u, v);
31              vertex l = nextto u in ‘v;
32              vertex r = prevto u in ‘v;
33              if (is l in ‘u && is r in ‘u)
34                  x$pos = 0.375 * ‘v$pos + 0.375 * ‘u$pos + 0.125 * ‘l$pos + 0.125 * ‘r$pos;
35              else
36                  x$pos = 0.5 * ‘v$pos + 0.5 * ‘u$pos;
37              add x to N;
38          }
39      }
-       // Complete the neighbourhoods of each new vertex
40      forall v in N  {
41          vertex a = any in v;
42          vertex b = nextto a in v;
43          vertex w = nextto v in a;
44          vertex x = prevto v in a;
45          vertex y = nextto v in b;
46          vertex z = prevto v in b;
-           // Check for which combination of neighbourhood orientations are
-           // around a and b and complete the new neighbourhoods accordingly
47          if (nextto b in ‘a == prevto a in ‘b)  {
48              splice w after a in v;
49              splice z before b in v;
50          }
51          else if (nextto b in ‘a == nextto a in ‘b)  {
52              splice w after a in v;
53              splice y before b in v;
54          }
55          if (prevto b in ‘a == nextto a in ‘b)  {
56              splice x before a in v;
57              splice y after b in v;
58          }
59          else if (prevto b in ‘a == prevto a in ‘b)  {
60              splice x before a in v;
61              splice z after b in v;
62          }
63      }
64      merge V with N;
65  }
```

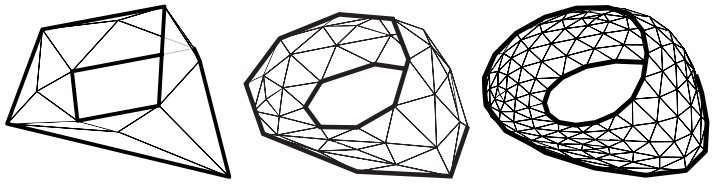


Figure 5.18: A Möbius strip subdivided twice

5.5 Incremental Subdivision

In addition to the regular surface subdivision algorithms, vv has been useful for implementations of advanced subdivision surface techniques, as demonstrated in the works of Pakdel and Samavati. The *incremental adaptive subdivision* [47, 48, 49] methods that they have developed using vv are here outlined. Details of the methods can be found in the respective original publications.

Adaptive subdivision is the subdivision of a selected regions of a polygon mesh, as opposed to the entire mesh. The essential challenge to the design of an adaptive subdivision algorithm is how to create a transition from the subdivided regions to the un-subdivided regions of the mesh while maintaining a conforming geometry (*i.e.* without introducing of cracks).

The incremental subdivision method addresses this problem by expanding the region of selected vertices to a k-ring of neighbouring vertices outside of the selected vertices. For example, if $k = 1$, then the selected region is expanded to include any vertex that is immediately connected to a vertex in the selected region. Then, it is the expanded region that is subdivided and any cracks are repaired by adding edges as necessary. Because the expansion of the selected region in the produced mesh is entirely contained in the subdivided region, there is no overlap in the regions where new edges need to be inserted to repair the cracks.

This method produces a gradual transition in the resolution of the mesh from the subdivided regions to the un-subdivided regions, avoiding undesired features, such as sliver triangles and high-valence vertices, that occur in naïve adaptive subdivision methods. Additionally, this method has the advantage is simple and the algorithms lend themselves well to implementation in vv.

The simple case of incremental Loop subdivision using a 1-ring is given in Algorithm 5.12 and the results of the subdivision are given in Figure 5.19, where it can be seen that the top and bottom of the polyhedron are subdivided more than the middle.

The algorithm, first proceeds by finding the vertices selected for subdivision and those adjacent to those selected and groups them together in the set R (lines 2 – 9). That set contains the vertices in the regions of surface to be subdivided. New vertices are inserted between existing pairs of vertices that are inside the selected region (lines 12 – 28) in exactly the same manner as the regular Loop subdivision. Then, the neighbourhoods of the new vertices are completed (lines 29 – 50). When the new vertex is in the interior of the selected region (*i.e.* it is entierly surrounded by other vertices in the selected region), then the neighbourhoods are constructed normally. However, when the new vertex is on the edge of the selected region, it is necessary to connect it to a vertex that is outside the selected region. This connection is a $T - junction$.

Algorithm 5.12: Incremental Loop Subdivision

```
1  void loop(mesh & V) {
-      // Add the selected vertices to a set
2      mesh R;
3      forall v in V {
```

```
4         if (!v$selected) continue;
5         add v to R;
6         forall u in v  {
7             add u to R;
8         }
9     }
10    mesh N;
11    synchronise V;
–     // Insert new vertices and adjust the position of existing vertices
12    forall v in R  {
13        unsigned int n = valence v;
14        double w = std::pow(0.375 + 0.25 * std::cos(2.0 * util::pi / double(n)), 2.0) + 0.375;
15        Pt Q;
16        forall u in ‘v  {
17            Q += ‘u$pos;
–             // Insert new vertices only in the selected regions
18            if (!is u in R) continue;
19            if (u < v) continue;
20            vertex x = insert(u, v);
21            x$pos = ‘v$pos * 0.375 + ‘u$pos * 0.375
22                + ‘(nextto u in ‘v)$pos * 0.125 + ‘(prevto u in ‘v)$pos * 0.125;
23            x$selected = false;
24            add x to N;
25        }
26        Q /= double(n);
27        vpos = w * vpos + (1.0 - w) * Q;
28    }
–     // Complete the neighbourhoods of existing vertices
29    forall v in N  {
30        vertex a = any in v;
31        vertex b = nextto a in v;
32        vertex c = nextto b in ‘a;
–         // Complete the neighbourhoods of the new vertices, such that
–         // when a new vertex is at the edge of the selected region, a
–         // T-junction is inserted
33        if (is c in R)  {
34            splice nextto v in a before a in v;
35            splice prevto v in b after b in v;
36        }
37        else  {
38            splice v after a in c;
39            splice c after b in v;
40        }
41        vertex d = prevto b in ‘a;
42        if (is d in R)  {
43            splice prevto v in a after a in v;
44            splice nextto v in b before b in v;
45        }
```

```
46          else {
47              splice v after b in d;
48              splice d after a in v;
49          }
50      }
51      merge V with N;
52  }
```

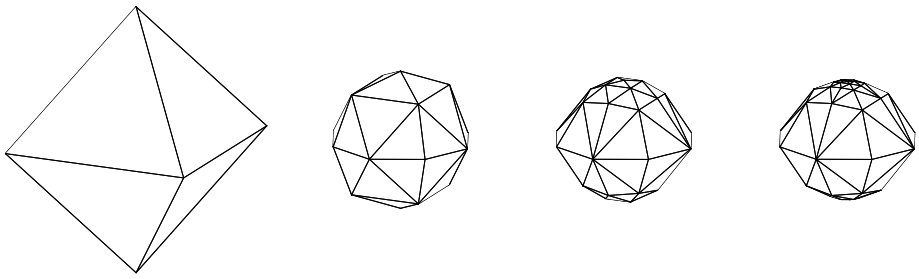


Figure 5.19: Incremental Loop subdivision applied three times to a polygon mesh where the top and bottom vertices are selected for subdivision

5.6 On the Use of VV for Implementing Subdivision Surfaces

Surface subdivision algorithms, like curve subdivision algorithms, lend themselves well to implementations in vv: the transformations are local processes that can be easily described with the vv algebra. Unlike the curve subdivision algorithms of Chapter 4, the algorithms in this chapter do not operate on a linear or branching structure and so cannot be implemented simply with an L-system.

Note that there is only a small increase in complexity from the implementations of subdivision curves to simple subdivision surfaces. This small increment of complexity is particularly evident when comparing the implementations of the B-spline and Loop subdivision algorithms (Algorithms 4.2 & 5.2) and the Dyn-Levin-Gregory and

butterfly algorithms (Algorithms 4.3 & 5.3). In both cases, the subdivision surface algorithms use a larger expression to apply the subdivision masks and there is an additional statement to set the neighbourhoods of the new vertices. Otherwise, the differences are minimal.

A comparison of the various implementations of Loop subdivision in this chapter (Algorithms 5.2, 5.8, 5.10, 5.11 & 5.12) also reveals that the addition of advanced subdivision features and techniques is reasonably straight forward.

Additionally, the example of incremental subdivision (§5.5) demonstrates that vv is suitable for researching and developing new surface subdivision techniques.

Chapter 6

Pattern Generation

Pattern generation, such as fractals and tilings, is often well-described by a set of iterative and local transformations. As was seen in the subdivision algorithm examples (see Chapters 4 & 5), vv lends itself well to the implementation of iterative algorithms and so can be used equally well to model fractals calculated by iterative algorithms.

6.1 The Sierpinski Gasket

A common example of a fractal that can be calculated iteratively is the Sierpinski gasket [67]. The Sierpinski gasket algorithm takes a triangle and divides it into three triangles with an empty triangular region at the centre. The division is applied recursively to the new triangles (see Figure 6.2).

Immediately, there is a notable feature in this fractal that was not seen the subdivision algorithms: every iteration creates holes in the surface. Also, unlike subdivision surface algorithms, the holes are also triangles, so a different strategy for determining what belongs to the surface is required than that presented in §5.3.

For this purpose, multiple interpretations of vertices can be used. A vertex in vv is a node in a graph with a geometric interpretation applied to it. However, the definition of vv does not require that the same geometric interpretation be applied to every vertex. For the case of the Sierpinski Gasket, two interpretations are used:

vertices that belongs to the corners of triangles and vertices that mark the insides of triangles.

The Sierpinski gasket algorithm (Algorithm 6.1 and Figure 6.1) first adds new vertices at the mid-point of each edge (lines 4 – 12) and then subdivides each triangle by using the vertices that mark the the triangles centres (lines 15 – 29). But, only the vertices that mark the triangle corners are used for rendering.

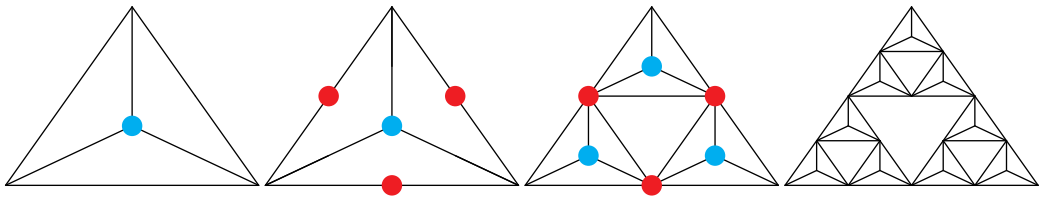


Figure 6.1: Illustration of the Sierpinski gasket algorithm. From left to right: A triangle starts with a special vertex at the centre (coloured blue). New vertices are added on each edge (coloured red). The centre vertex is replaced with three new centres. The process is repeated.

Algorithm 6.1: Sierpinski gasket

```
1   void sierpinski(mesh & V) {
2       mesh N;
3       synchronise V;
–       // Insert new vertices on the edges
4       forall v in V {
5           if (v$type != 'v') continue;
6           forall u in ‘v {
7               if (u < v || u$type == 'c') continue;
8               vertex x = insert(u, v);
9               x$pos = 0.5 * (u$pos + v$pos);
10              x$type = 'v';
11              add x to N;
12          }
13      }
14      merge V with N;
15      clear N;
16      forall c in V {
17          if (c$type != 'c') continue;
18          forall v in c {
```

```
–           // Create the edges to get a new triangle
19          vertex a = nextto c in v;
20          vertex b = prevto c in v;
21          splice b after v in a;
22          splice a before v in b;
–           // Create a new vertex at the centre of the new triangle
23          vertex n;
24          n$type = 'c';
25          make { v, b, a } nb_of n;
26          add n to N;
27          replace c with n in v;
28          splice n after v in a;
29          splice n before v in b;
30        }
31        remove c from V;
32    }
33    merge V with N;
34  }
```

Figure 6.2: The Sierpinski gasket algorithm applied six times

The idiom of having vertices with different interpretations illustrated in this example is often useful when the structure of the modelled system contains heterogeneous components. It can be considered to be a simple typing system. This technique is used frequently in the biological modelling examples shown later in this thesis.

6.2 Penrose Tiles

Tiling patterns can also be implemented using VV systems. One interesting case is the *Penrose tilings* [50, 64], aperiodic tilings of the plane. Algorithm 6.2 implements the *rhomb Penrose tiling* as a recursive tile subdivision based on the masks in Figures 6.3 and 6.4.

The algorithm operates by first replacing each edge in the mesh according to the edge masks specified in Figure 6.3 (lines 6 – 46). Then, for the two edge masks that insert new vertices, additional edges are inserted to divide the tiles, as specified in Figure 6.4 (lines 47 – 110). The results of tiling can be seen in Figure 6.5.

Note that the edges in the masks are oriented and have differing labels (the direction and number of arrow-heads on each edge). This information is coded in an edge structure using two variables. Firstly, the number of arrow-heads is coded with the variable, t, using a value from one to four. Secondly, the direction is coded as a boolean variable, d, which is set to true when the direction is in the same direction as the relation of the adjacent vertices' ordering (*i.e.* if the direction is from vertex u to v and $u < v$, then $(u|v).\mathsf{d} = \mathsf{true}$, otherwise $(u|v).\mathsf{d} = \mathsf{false}$).

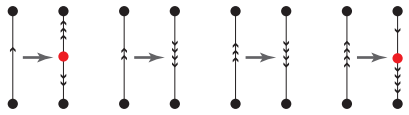


Figure 6.3: The edge masks for Penrose tiling

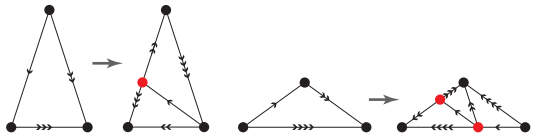


Figure 6.4: The tile masks for Penrose tiling, there are two additional symmetrical cases

Algorithm 6.2: Penrose tiling

```
1  void penrose(mesh & V) {
2      mesh N1;
3      mesh N2;
4      mesh O;
5      synchronise V;
6      forall v in V {
```

```
7         forall u in ‘v  {
8             if (u < v) continue;
–             // Apply the edge masks
9             switch((u|v).t)  {
–                 // Apply the first edge mask
10                case 1:
11                    {
12                        bool d = (u|v).d;
13                        vertex n = insert(u, v);
14                        if (d) n$pos = 0.618 * u$pos + 0.382 * v$pos;
15                        else n$pos = 0.618 * v$pos + 0.382 * u$pos;
16                        (v|n).t = d ? 2 : 3;
17                        (u|n).t = d ? 3 : 2;
18                        (v|n).d = (n < v);
19                        (u|n).d = (n < u);
20                        add n to N1;
21                    }
22                    break;
–                 // Apply the second edge mask
23                case 2:
24                    (u|v).t = 4;
25                    (u|v).d = !((u|v).d);
26                    break;
–                 // Apply the third edge mask
27                case 3:
28                    (u|v).t = 2;
29                    (u|v).d = !((u|v).d);
30                    break;
–                 // Apply the fourth edge mask
31                case 4:
32                {
33                    bool d = (u|v).d;
34                    vertex n = insert(u, v);
35                    if (d) n$pos = 0.618 * u$pos + 0.382 * v$pos;
36                    else n$pos = 0.618 * v$pos + 0.382 * u$pos;
37                    (v|n).t = d ? 4 : 1;
38                    (u|n).t = d ? 1 : 4;
39                    (v|n).d = d ? (n < v) : (v < n);
40                    (u|n).d = d ? (u < n) : (n < u);
41                    add n to N2;
42                }
43                break;
44            }
45        }
46    }
–     // Apply the first face mask, the else cases handle
–     // the semetrical variation of the mask
47    forall n in N1  {
```

```
48          vertex a = any in n;
49          if ((a|n).t != 2) a = nextto a in n;
50          vertex x = nextto n in a;
51          if ((a|x).t == 0) {
52              vertex o(n$pos);
53              make { n } nb_of o;
54              splice o before a in n;
55              (n|o).t = 0;
56              add o to O;
57          }
58          else {
59              splice x before a in n;
60              splice n after a in x;
61              (n|x).t = 1;
62              (n|x).d = (x < n);
63          }
64          x = prevto n in a;
65          if ((a|x).t == 0) {
66              vertex o(n$pos);
67              make { n } nb_of o;
68              splice o after a in n;
69              (n|o).t = 0;
70              add o to O;
71          }
72          else {
73              splice x after a in n;
74              splice n before a in x;
75              (n|x).t = 1;
76              (n|x).d = (x < n);
77          }
78      }
-       // Apply the second face mask, the else cases handle
-       // the semetrical variation of the mask
79      forall n in N2 {
80          vertex a = any in n;
81          while (!((a|n).t == 1 && (a|n).d == (a < n))) a = nextto a in n;
82          vertex x = nextto n in a;
83          if ((a|x).t == 0) {
84              vertex o(n$pos);
85              make { n } nb_of o;
86              splice o before a in n;
87              (n|o).t = 0;
88              add o to O;
89          }
90          else {
91              splice x before a in n;
92              splice n after a in x;
93              (n|x).t = 2;
```

```
94              (n|x).d = (n < x);
95          }
96          x = prevto n in a;
97          if ((a|x).t == 0)  {
98              vertex o(n$pos);
99              make { n } nb_of o;
100             splice o after a in n;
101             (n|o).t = 0;
102             add o to O;
103         }
104         else  {
105             splice x after a in n;
106             splice n before a in x;
107             (n|x).t = 2;
108             (n|x).d = (n < x);
109         }
110     }
111     merge V with N1;
112     merge V with N2;
113     merge V with O;
114 }
```

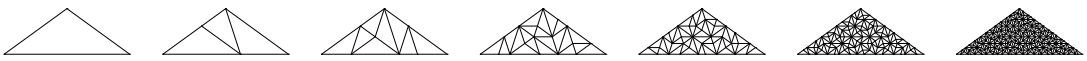


Figure 6.5: The Penrose tiling applied recursively six times

The use of directed edges is not often used in the examples presented in this thesis; however, there are many algorithms in modelling that make use of directed edges and so can be modelled with vv using the technique illustrated in this example.

6.3 Terrain Generation

A simple and often used example that demonstrates the use of fractals in modelling is the *fractal mountain algorithm* [17, 54]. By the iterative application of a generative rule, a terrain-like object can be created. The complete algorithm that has the features discussed in §6.3.1 – 6.3.3 is given in Algorithm 6.3.

6.3.1 Fractal Mountains

The topological transformations used in the fractal mountain algorithms are exactly those used in the polyhedral subdivision surface algorithm (§5.1.1). Thus the topological transformation portion of the vv program for Loop or butterfly subdivision can be used to implement the fractal mountain algorithm.

However, there subdivision algorithms and the fractal mountain algorithms differ in the geometric transformations applied. Whereas subdivision algorithms use only affine transformations, the fractal mountain algorithm also uses vector algebra. Part of the terrain specification is a vector to define the up direction. Whenever a new vertex is added to the terrain, it is first set to the midpoint between two existing vertices, an operation of affine geometry, and then it is displaced by a random amount in the up direction, an operation of vector algebra, (lines 17 – 22). At each step the range of the random displacement is reduced to get progressively finer features. The addition of the vertices with random displacements gives a terrain a jagged appearance reminiscent of mountains. The development of the terrain is shown in Figure 6.6.

6.3.2 Fractal Foothills

The terrain generated by the fractal mountain algorithm has sharp features everywhere, but the base of mountain ranges is often composed of smoother foothills. A terrain that is a mix of mountains and foothills can be created by a small modification to that used for fractal mountains.

The algorithm first calculates the positions of new vertices in the terrain as described in §6.3.1, but then an alternate position is calculated using the Warren

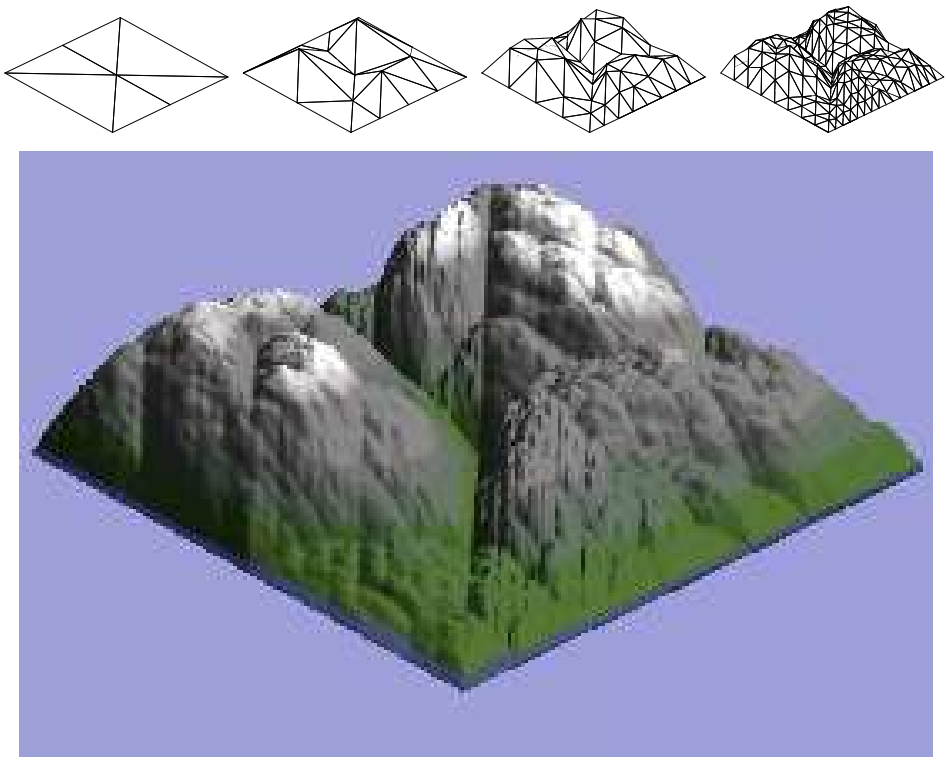


Figure 6.6: The first four and seventh steps of the fractal mountain

variation of the Loop surface subdivision algorithm (lines 6 – 10 and 16). Thus, a jagged feature created from the fractal part of the algorithm and a smooth feature created by the subdivision algorithm are set at the same vertex in the terrain. Mountain peaks should be jagged and the hills at the base smooth, so a bias function is used to weight the two positions and assign a blended position to the vertex. The bias function allows an artistic decision of where in the terrain the features should be jagged, smooth or between the two extremes. The results of the algorithm are shown in Figure 6.7.

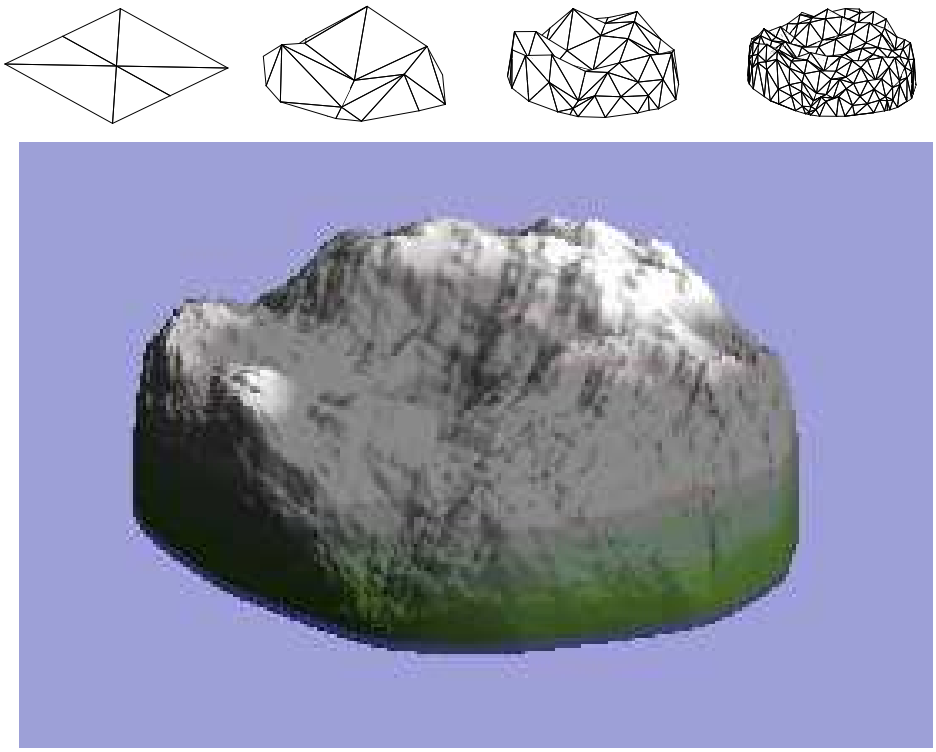


Figure 6.7: The first four and seventh steps of the fractal mountain with smoothing

6.3.3 Fractal River

A terrain feature that often accompanies a fractal mountain terrain is a fractal river modelled by a *squig curve* [36, 37]. The river is composed of segments, and as the terrain divides, the river segments are subdivided.

The river segments are marked vertices on the terrain mesh. When new vertices are inserted into the terrain, some of those new vertices are around those marked as river segments. A random path is found between the two old vertices along the new ones (lines 29 – 36). The new vertices along that path are marked as part of the river. Each iteration of terrain development, adds more twists and turns to the river

(see Figure 6.8). After the new river path is found, all vertices that are part of the river are set to a height of zero* (lines 90 – 92). When the river is combined with smoothing, as described in §6.3.2, a wide, winding river can be achieved, as is seen in the last image of Figure 6.8.

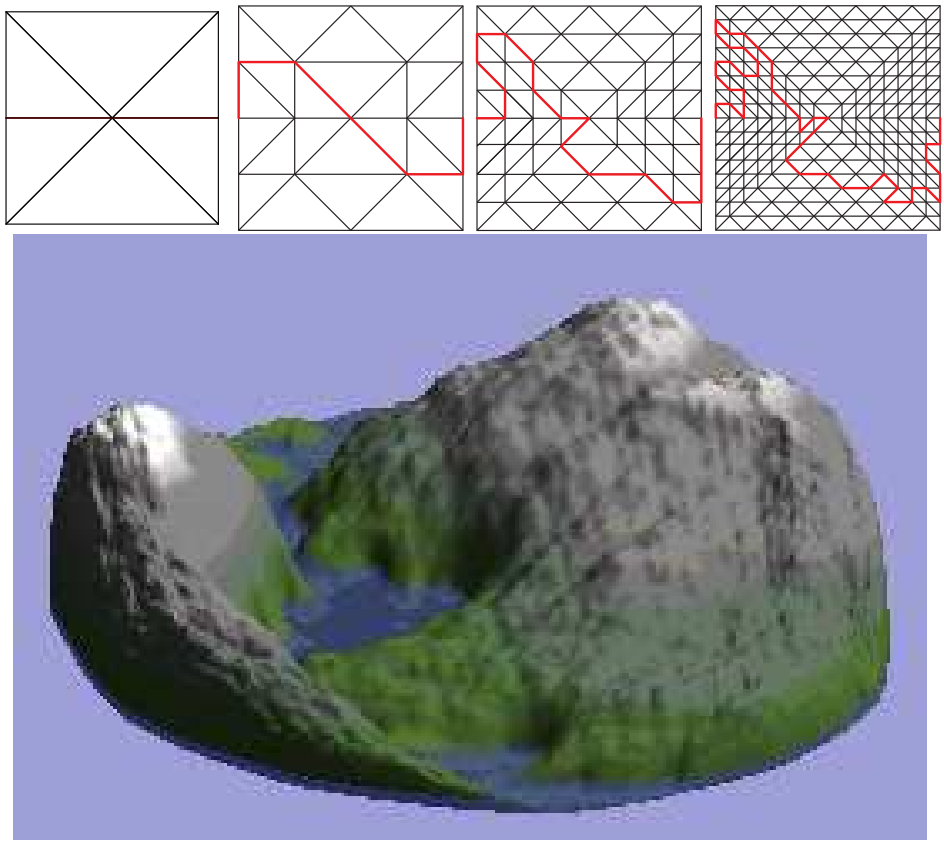


Figure 6.8: The first four (overhead view) and seventh steps with the mountains with a river

Algorithm 6.3: The Fractal Terrain

```
1  void terrain(mesh & V, mesh & R)  {
2      ++step;
3      synchronise V;
```

*To keep the model simple, the gradient of the river is not considered. This is consistent with the squig river model proposed by Mandelbrot.

```
4     mesh N;
5     forall v in V  {
6         double w = 0.0;
7         unsigned int n = valence v;
8         if (n == 3) w = 0.1875;
9         else w = (3.0 / (8.0 * double(n)));
10        v$pos *= (1.0 - (double(n) * w));
–         // Insert new vertices on each edge
11        forall p in `v  {
12            v$pos += `p$pos * w;
13            if (p < v)  {
14                vertex n = insert(p, v);
15                n$river = false;
–                 // Set the position of the new vertex using surface subdivision
16                n$pos = `v$pos * 0.375 + `p$pos * 0.375
17                    + `(nextto p in `v)$pos * 0.125 + `(prevto p in `v)$pos * 0.125;
–                 // Find a random offset to the position, uni() returns a
–                 // random value between zero and one
18                double height = uni() * pow(0.5, step);
19                if (!first)  {
–                     // Find the final position as a blend of subdivision
–                     // and vector geometry
20                    height *= bias(n$pos.y() / max_y);
21                    height += n$pos.y();
22                }
23                n$pos.y(height);
24                add n to N;
25            }
26        }
27    }
28    mesh NR;
–     // Traverse the river vertices and select new vertices
–     // to fill in the squig curve
29    forall v in R  {
30        forall p in v  {
31            vertex r = nextto v in p;
32            if (r$river && r < v)  {
–                 // Randomly select one of three adjacent paths
–                 // from r to v to add to the river
33                double path = uni();
34                if (path < 0.33)  {
35                    p$river = true;
36                    add p to NR;
37                }
38                else if (path < 0.67)  {
39                    vertex a = prevto p in v;
40                    vertex b = nextto p in r;
41                    a$river = true;
```

```
42                  b$river = true;
43                  add a to NR;
44                  add b to NR;
45              }
46              else {
47                  vertex a = nextto p in v;
48                  vertex b = prevto p in r;
49                  a$river = true;
50                  b$river = true;
51                  add a to NR;
52                  add b to NR;
53              }
54          }
55      }
56  }
  –   // Set the vertices in the river to a height of zero
57  merge R with NR;
58  forall v in R {
59      v$pos.y(0);
60  }
  –   // Complete the neighbourhoods of the new vertices. Edges that are
  –   // on the boundary of the terrain are set to a height of zero.
61  synchronise N;
62  forall v in N {
63      vertex a = any in v;
64      vertex b = nextto a in v;
65      vertex q = nextto v in a;
66      vertex r = prevto v in b;
67      vertex s = prevto v in a;
68      vertex t = nextto v in b;
  –       // Complete the neighbourhood depending on if the vertex
  –       // is on a boundary
69      if (nextto a in `q == prevto b in `r) {
70          splice q before a in v;
71          splice r after b in v;
72      }
73      else {
74          v$pos.y(0);
75          a$pos.y(0);
76          b$pos.y(0);
77      }
78      if (prevto a in `s == nextto b in `t) {
79          splice s after a in v;
80          splice t before b in v;
81      }
82      else {
83          v$pos.y(0);
84          a$pos.y(0);
```

```
85              b$pos.y(0);
86          }
87      }
88      merge V with N;
89      first = false;
90  }
```

Part III

Biological Modelling

Chapter 7

Physically-Based Models of Growth

7.1 Descriptions of Growth

In addition to geometric models, such as those presented in Chapters 4 – 6, vv is a useful tool for modelling some biological subjects. VV lends itself particularly well to modelling growth. A growing structure frequently adds components, which is easily done using the neighbourhood editing operations of the vv algebra. Biological systems also exchange of information between neighbouring components, such as a chemical that diffuses between adjacent cells. Such information exchange is easily modelled using the neighbourhood query operations of vv.

Several strategies for modelling biological systems can be implemented using vv, depending on how the growth of the structure is described and where the growth takes place. The description of growth can be separated into three categories. Firstly, there are models with *descriptive growth*, where the growth follows a supplied geometric description, such as a profile curve. Secondly, there are models with *physically simulated growth*, where the form is an emergent property of a physics system (*e.g.* a *mass-spring system* [74]). And thirdly, growth can be described as a *growing canvas* [7], a transformation of the space containing the structure.

Growth can take place at many locations on the structure. The examples in the following chapters model growth that occurs on a growing boundary, in a locally

contained region, as cell division and everywhere on a surface.*

7.2 On the Physical Simulation of Tissue Growth

An organism's form results from a *morphogenetic process*. The development and growth of its components contribute to its shape. The growth of each part of the organism exerts tensions and stresses on the rest of the body. Form can be considered an emergent property of the tensions induced by growth [75, 7, 40].† According to this interpretation of form, a physical simulation of the forces induced by growth can be used to determine the emergent shape of a biological model. To implement such a physical simulation using vv, a physical interpretation of a graph is required.

One simple physical interpretation of a graph is to treat it as a *mass-spring system* [74], with the edges in the graph representing springs that act on masses located at the nodes. At each time step, the force acting on a vertex is the sum of the forces exerted by the springs connecting it to its neighbours. The resulting accelerations and velocities are updated by forward Euler integration until an equilibrium is met. A function to visit each vertex, calculate the forces exerted by the springs and update the positions and velocities of each vertex is given in Algorithm 7.1. In this function, the spring equation is simplified by assuming that each vertex has a mass of one unit so that the force on the vertex is also its acceleration. Other sources of forces may also be considered. For example, a force in normal direction to a vertex can be used to represent the pressure of internal tissues on the external layer of a volumetric

*Parts of this chapter are based on [68].

†In the context of classical biology, the work of Thompson [75] is an excellent and thorough investigation on the topic. The use of physics for the simulation of growth in a modern context is covered in [7, 40].

structure.

Algorithm 7.1: Sample vv code for calculating the force at each vertex

```
1  forall v in V  {
2      v$acceleration.set(0.0, 0.0, 0.0);
3      forall x in v  {
4          float dist = v$pos.distance(x$pos);
5          float k = (rest_length - dist) / dist;
6          v$acceleration += k * (v$pos - x$pos);
7      }
8      v$velocity += (v$acceleration - v$velocity * drag) * dt;
9      v$pos += v$velocity * dt;
10 }
```

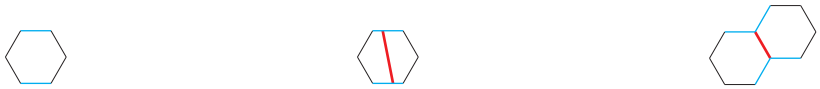
Growth and development are simulated by changing the parameters (*e.g.* the rest length) and configuration of selected springs and masses [55]. These changes can be induced and controlled by external factors, such as the orientation of springs and polygons with respect to gravity, and internal factors, such as the concentration of morphogens that diffuse and react in the simulated tissue.

7.3 Modelling the Korn-Spalding Cell Division Pattern

Korn and Spalding proposed a simple algorithm to mimic cell division patterns in plant epidermal tissues [28]. This pattern was previously used as a test case for simulating developing tissues [10] and is here used again for the same purpose.

The Korn-Spalding pattern begins with a single hexagonal cell (Figure 7.1a). Its division is simulated by inserting a cell wall between two opposite sides of the cell, at one third the distance along each edge (Figure 7.1b). The daughter cells then return to the regular hexagonal shape (Figure 7.1c). In subsequent iterations, the process of cell division is repeated with the dividing walls rotated by $^2/_3\pi$ with respect to

their previous orientation each time.



(a) The initial cell with the bottom and top edges selected

(b) The insertion of an edge to divide the cell into two

(c) The cells after the physical simulation

Figure 7.1: The Korn-Spalding cell division process

To model this process in vv, it is assumed that all springs of the developing mass-spring system have the same rest length (*i.e.* a rest length of one unit) and that each cell exerts some pressure outwards on its walls. Consequently, after division, the cells assume an approximately regular hexagonal tessellation. The first steps in the simulation of a growing tissue according to the Korn-Spalding algorithm are shown in Figure 7.2.

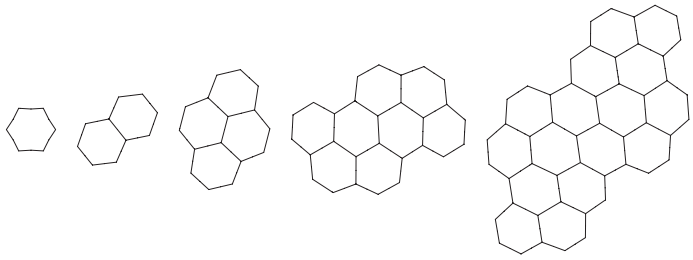


Figure 7.2: Development of the Korn-Spalding pattern

The vv program for modelling the Korn-Spalding cell division pattern is given in Algorithm 7.2. The boundary of the tissue is marked using the half-edge marking scheme as described in §5.3.2. The algorithm starts with a single hexagonal cell with the outside half-edges are marked with a '0' and the inside edges are marked as '1', '2' and '3', such that the opposite inside edges have the same mark and the marks

increase in clockwise order.

The first part of the algorithm (lines 4 – 22) is the physical simulation. The physical simulation is composed of a mass-spring system, after the pattern presented Algorithm 7.1 and an approximation of a pressure simulation. The forces from the masses in springs are calculated in lines 7 – 10. Where there is a half-edge on the border of the tissue, the half-edges marked with '0', a force in the direction normal to the edge is added to each vertex of the edge. This force, calculated in lines 12 – 14, is an approximation of the pressure exerted by the cells outwards on the tissue. This force keeps the the cells hexagonal in shape.

When the forces in the system have reached equilibrium, checked in line 19, the algorithm proceeds to the cell division phase (lines 29 – 62). The division occurs by dividing in two stages. Firstly, all edges that are marked with the current mark, for example, all edges marked with a '1', are divided into three edges (lines 31 – 36) and marked with new labels (lines 39 – 50). Secondly, each new vertex is connected to another new vertex found by walking around the inside cell (lines 56 – 59) and the new edge is marked (line 60) to form the new cell walls.

Algorithm 7.2: Korn-Spalding Cell Division

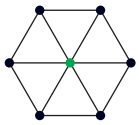
```
1  void ks(mesh & S) {
2      synchronise S;
-      // Perform the physics simulation
3      bool settled = true;
4      forall v in S {
5          v$acceleration.zero();
6          forall u in v {
7              double d = v$pos.distance('u$pos);
8              double k = (rest_length - d) / d;
9              Pt w = 'u$pos - v$pos;
10             v$acceleration -= k * w;
11             if ((u^v).mark == 0) {
```

```
12                  Pt n(-w.y(), w.x());
13                  n.normalise();
14                  n *= pressure;
15                  v$acceleration += 0.5 * n;
16                  u$acceleration += 0.5 * n;
17              }
18          }
19          if (settled && v$acceleration.length() > 0.01) settled = false;
20          v$velocity += (v$acceleration - v$velocity * drag) * dt;
21          v$pos += v$velocity * dt;
22      }
23      if (!settled) return;
        // Change the pair of edges that are divided
24      switch (mark)  {
25          case 1: next_mark = 2; prev_mark = 3; break;
26          case 2: next_mark = 3; prev_mark = 1; break;
27          case 3: next_mark = 1; prev_mark = 2; break;
28      }
29      mesh N;
        // Add two new vertices to each cell wall marked for division
30      forall v in S  {
31          forall u in v  {
32              if (!((v^u).mark != mark && (u^v).mark != mark)) continue;
33              vertex a = insert(u, v);
34              vertex b = insert(a, v);
35              a$pos = (u$pos + u$pos + v$pos) / 3.0;
36              b$pos = (u$pos + v$pos + v$pos) / 3.0;
37              add a to N;
38              add b to N;
                // Assign marks to the new edges
39              {
40                  unsigned int old_mark = ((`u)^v).mark;
41                  (u^a).mark = old_mark;
42                  (a^b).mark = (old_mark) ? prev_mark : 0;
43                  (b^v).mark = old_mark;
44              }
45              {
46                  unsigned int old_mark = ((`v)^u).mark;
47                  (a^u).mark = old_mark;
48                  (b^a).mark = (old_mark) ? prev_mark : 0;
49                  (v^b).mark = old_mark;
50              }
51          }
52      }
        // Add in the new cell walls
53      forall v in N  {
54          forall u in v  {
55              if ((v^u).mark == mark)  vertex a = prevto v in u;
```

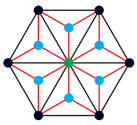
```
56              vertex b = prevto u in a;
57              vertex c = prevto a in b;
58              vertex d = prevto b in c;
59              splice v before c in d;
60              (d^v).mark = next_mark;
61          }
62      }
63      S += N;
64      mark = next_mark;
65  }
```

7.4 Modelling a Root Apical Meristem

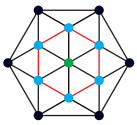
The next example is a simple model of a growing *root apical meristem*, or *root apex*. This example differs in three respects from the Korn-Spalding cell division example. First, growth occurs in just a small portion of the tissue. Secondly, instead of considering individual cells, the tissue is modelled as a continuous material, discretised as a polygon mesh. And thirdly, the model represents the surface of a three-dimensional structure, whereas the cell division model was in a plane.



(a) The initial configuration; the apical vertex is in the center



(b) The configuration of the root tip after the insertion of new vertices



(c) The edges are flipped such that the valence of the apical vertex returns to six

Figure 7.3: Simulation of growth at the root tip

The root is represented as a tubular polygonised surface, growing near the tip. To extend the root, new vertices are inserted into the mesh uniformly around the apical vertex (Figures 7.3a and b). The arrangement of the edges around the apical vertex

is then modified in a manner similar to Kobbelt's $\sqrt{3}$ surface subdivision algorithm (see §5.1.4), so that the valence of the apical vertex returns to six (Figure 7.3c). The newly created edges have the same rest length as the older edges, but when they are newly created they are shorter than their rest length; afterwhich, they expand, causing the structure grow at the apex. In subsequent steps, the process illustrated in Figure 7.3 is repeated on the newly formed rings around the new apical vertex. Algorithm 7.3 specifies these operations in vv.

Algorithm 7.3: Sample vv code for the insertion of vertices around a vertex

```
–  // Insert a vertex at the centre of each triangle adjacent
–  // apical vertex
1  forall x in ‘apex {
2      vertex y = nextto x in apex;
3      vertex n;
4      n$pos = 0.33 * (apex$pos + x$pos + y$pos);
5      make { apex, x, y } nb_of n;
6      splice n after x in apex;
7      splice n after y in x;
8      splice n after apex in y;
9  }
–  // Flip all the edges around the new vertices
10 forall x in ‘apex {
11     vertex a = nextto apex in x;
12     vertex b = prevto apex in x;
13     erase x from apex;
14     erase apex from x;
15     splice a after apex in b;
16     splice b after x in a;
17 }
```

The remainder of the root model code is quite simple and so is not given in complete here. Like the Korn-Spalding model, the physical simulation is again a mass-spring system with a simplified pressure, modelled by a force in the normal direction, though normal to the surface in this case. The only other difference is that some of the initial vertices are kept in a fixed position during simulation to

anchor the root in place. Thus, the code for the physical simulation is largely the same as before.

A sample structure generated by this algorithm is shown in Figure 7.4a. A simple model of root gravitropism is obtained by modulating the rest length of the springs as a function of the orientation of their incident polygons with respect to gravity, as shown in Figure 7.4b.

(a) A growing root tip where all the springs have the same rest length

(b) Gravitropism is simulated by having springs on the top side of the root are longer than those on the bottom

Figure 7.4: Two growing root tips

7.5 Remarks on Physically-Based Models of Growth

The examples in this chapter illustrate that simple physical simulations combined with simple vertex insertion patterns can be used to create developmental models.

Mass-spring systems solved with an explicit integration method lend themselves particularly well to creating physical simulations using vv. The forces acting on each mass are from the springs immediately attached to it; parts of the structure that are not immediately neighbouring need not get visited to get the needed information for the the physical simulation. The addition of other local forces, such as the

approximation of pressure here used, is a likewise simple matter. When the structure is changed, either in configuration or by the addition of a vertex, the mass-spring system naturally adapts. The forces and positions for each vertex are calculated separately, so a new spring or a new mass does not incur a large computational overhead.

The modelling principles demonstrated in this chapter, the use of mass-spring systems, patterns for cell division and local addition of vertices, are used in the following chapters to create more complex models of growing organisms. Similar physics are used to create a daffodil corona (§8.2) and a shoot apical meristem (§9). The model of a shoot apical meristem also makes use of local vertex insertions similar to that of the root model. Cell division patterns are revisited in §10 and §12.2.

Chapter 8

Growth on a Boundary

8.1 A Concho-Spiral Sea Shell Model

One type of growth that occurs in a number of organisms is the addition of new material at a surface's boundary. One example is the formation of a sea shell. As a shell grows, new material is added to the margin of the shell opening, but the remainder of the shell is a static structure* (see [82] for an overview of sea shell biology).

Sea shell growth is modelled with two processes (Algorithm 8.1). Firstly, growth at the margin of the shell is performed by first inserting new vertices near the growing boundary (lines 4 – 11) and the vertices on the growing boundary are advanced along a concho-spiral (lines 12 – 15), as described by Coxeter [9] and used in the sea shell models in [43, Chapter 10]. This spiral in parametric form, for cylindrical coordinates (r, θ, z) and a rotation around the point $(a, 0, c)$, is

$$r = \mu^u a, \theta = u, z = \mu^u c.$$

An illustration of a concho-spiral is shown in Figure 8.1. The explicit advancement of points along that curve in three-dimensional spatial coordinates (x, y, z) is

*A sea shell is not entirely static. Interior sections and features on the outer section of a shell may be dissolved by the organism living in the shell. However, the model here only considers a shell's exterior where material is added to the growing margin.

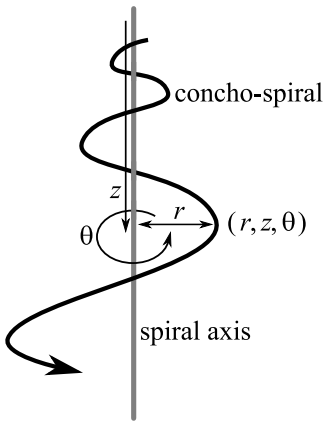


Figure 8.1: A concho-spiral with coordinates (r, θ, z). The values of the coordinates increase regularly over the length of the spiral.

$$\begin{aligned} x_{i+1} &= x_i \cos(\mathrm{d}r) + z_i \sin(\mathrm{d}r), \\ y_{i+1} &= y_i - \mathrm{d}t, \\ z_{i+1} &= -x_i \sin(\mathrm{d}r) + z_i \cos(\mathrm{d}r). \end{aligned}$$

When the algorithm adds the connectivity for the new vertices, the flag feature (see §3.5.3) is used to orient the connections. In the neighbourhood of a new vertex, the flag is set to vertex that is on the outermost ring of the surface. Thus the flagged vertex can be used as a guide to find the vertices that are used to complete the neighbourhood of the new vertex.

Secondly, after some number of growth steps, new vertices are added onto the boundary to increase the shell's resolution as it grows (lines 25 – 52). This increase is done by considering the boundary as a contour and applying the Chaikin subdivision curve algorithm. The curve subdivision differs from Algorithm 4.4 only in that it must consider the connections of the boundary vertices to the rest of the shell model. The use of the boundary property on the vertices simplifies this process.

The resulting shell model is shown in Figure 8.2. Note that near the margin of

the shell, the increase in resolution from the subdivision of the growing margin is evident in the figure.

Algorithm 8.1: Sea shell growth

```
1  void sea_shell(mesh & V, mesh & B) {
2      mesh N;
       // Extend the boundary of the shell
3      forall v in B {
4          forall u in v {
5              if (u$boundary) continue;
               // Insert a new vertex behind the vertex on the boundary
6              vertex n = insert(u, v);
7              n$boundary = false;
8              n$pos = v$pos;
9              flag v in n;
10             add n to N;
11         }
           // Advance the vertex on the boundary along the concho-spiral
12         double x = v$pos.x() * ct + v$pos.z() * st;
13         double z = v$pos.x() * -st + v$pos.z() * ct;
14         v$pos.set(x, v$pos.y() - dt, z);
15         v$pos *= s;
16     }
       // Complete the neighbourhoods of the new vertices
17     forall v in N {
18         vertex a = flagged in v;
19         splice prevto a in prevto v in a after a in v;
20         splice nextto a in nextto v in a before a in v;
21     }
22     merge V with N; clear N;
23     if (--subdivision_counter) return;
24     subdivision_counter = count;
       // Subdivide the boundary
25     synchronise B;
       // Insert two new vertices on each edge of the boundary
26     forall v in B {
27         forall p in ‘v {
28             if (!p$boundary ||p < v) continue;
29             vertex a = insert(v, p);
30             vertex b = insert(a, p);
31             a$boundary = true;
32             b$boundary = true;
33             a$pos = v$pos * 0.75 + p$pos * 0.25;
34             b$pos = v$pos * 0.25 + p$pos * 0.75;
35             add a to N;
```

```
36              add b to N;
37          }
38      }
 –      // Connect the new vertices to the ring behind the
 –      // growing boundary
39      forall v in B  {
40          vertex x = any in v;
41          while (x$boundary) x = nextto x in v;
42          vertex a = prevto x in v;
43          vertex b = nextto x in v;
44          replace v with b in a;
45          replace v with a in b;
46          splice x after b in a;
47          splice x before a in b;
48          splice a after v in x;
49          splice b before v in x;
50          erase v from x;
51          remove v from V;
52      }
53      B = N;
54      merge V with N;
55  }
```

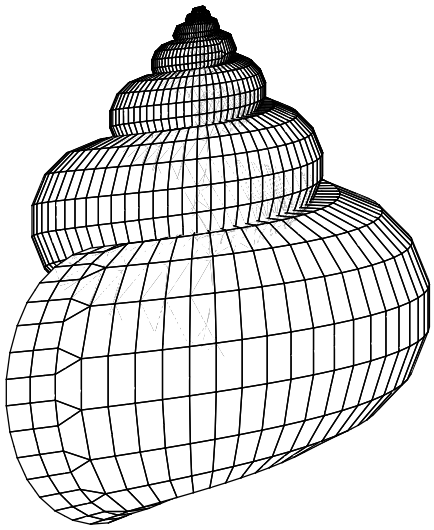


Figure 8.2: A sea shell model

8.2 A Wrinkled Daffodil Corona

As demonstrated in §8.1, growth on a boundary can be a useful strategy for modelling certain growing structures. An interesting question is what happens with *hyperbolic growth*, when the growing margin grows faster radially than the interior surface. A mathematical description of which can be found in [23]. Hyperbolic growth can be a good description for wrinkled and buckling structures [39, 66, 40, 65]. Of particular interest, Sharon *et al.* [65] postulated that hyperbolic growth is the cause of the wrinkling that is seen in the corona of a daffodil, as is seen in Figure 8.3.

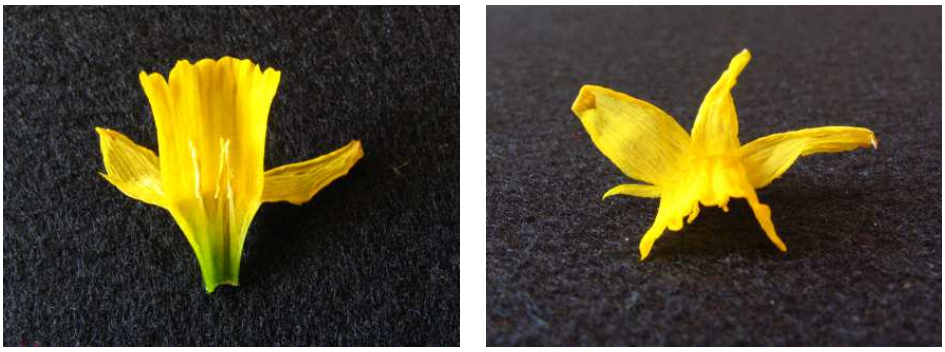


Figure 8.3: A cross-section of a mini-daffodil

Hyperbolic growth can be modelled using a physically-based approach, such as a mass-spring system, as described in §7.2. However this case, also requires that the surface buckles instead of folding; therefore, a thin-sheet flexion model, a simplified version of that found in [20], is used.

A daffodil's corona can be modelled as a surface that is topologically cylindrical and growing at the top margin. The boundary conditions of the cylinder are handled using the method described in §5.3.1. To produce wrinkling at the top margin, the margin grows faster horizontally than the region below. This growth is achieved by

both the lengthening of the springs and by subdividing the growing margin to insert more vertices, similar to the case of the sea shell model.

A subdivided margin with lengthened springs is equivalent to the discrete construction of a hyperbolic surface as found in [23]. An example of a discrete hyperbolic construction is given in Figure 8.4, in which the line segments in the horizontal lines across the mesh are half the size of the row below.

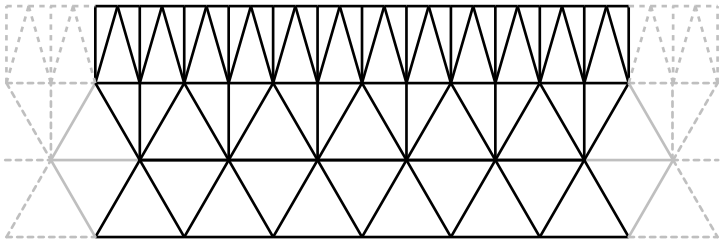
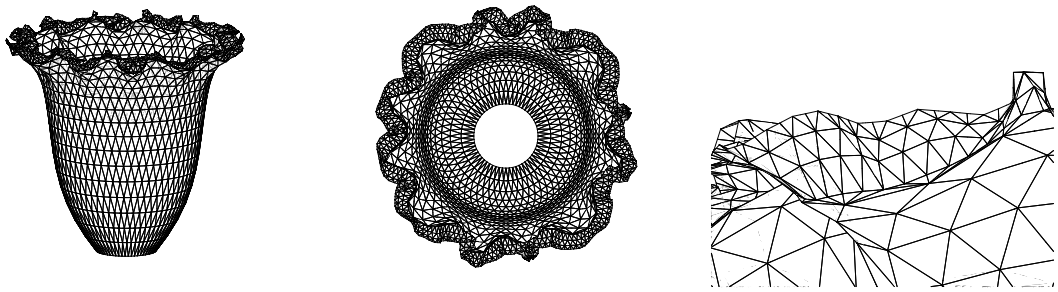


Figure 8.4: A discrete hyperbolic construction

If instead the line segments in subsequent rows were not of one-half length, the overall shape would have to bend to hold together. With some constraint of the position of some of the vertices (*e.g.* the vertices on the bottom row must be kept in place), the shape will buckle out of plane, resulting in the ripples and waves like those seen in the photographs of the daffodil (Figure 8.3).

In this model, the addition of new vertices to the top of a cylinder matches the construction shown in Figure 8.4, and the amount variation of the springs' rest lengths lengths from the one-half ratio from one row to the next allows control over how much the corona buckles. In the middle and lower parts of the corona, the springs' rest lengths increase slowly, resulting in a trumpet shape. And near the top, the rest lengths increase rapidly, resulting in the buckling at the top rim. The results of this model are shown in Figure 8.5. As an additional degree of control, not

every row of vertices added to the cylinder has more vertices than that below it; the hyperbolic construction is only used in certain regions to increase the buckling as it grows.



(a) Profile view (b) Top view (c) Detail view

Figure 8.5: The resulting daffodil corona model viewed in profile, from the top and in detail. In (a) and (b), the top buckling can be seen. In (c), the increase in the the mesh resolution can be seen.

The main function for producing the daffodil corona is given is Algorithm 8.2. There are separate functions for adding new vertices to the top of the cylinder (Algorithm 8.3), running the physics simulation (Algorithm 8.4) and performing an adaptive subdivision around the top ring to create the hyperbolic construction (Algorithm 8.5). Here, there are a few global variables: C, the set of all vertices, M, the vertices in the top-most ring of the cylinder, segments, the number of rings in the cylinder and max_segments, the maximum number of rings that the simulation should add.

Algorithm 8.2: Daffodil Corona

```
1  void daffodil() {
2      synchronise C;
–      // Calculate the normal at each vertex
3      set_normals(C);
–      // Run the physics simulation. If the system is stable
```

```
–     // after the simulation, execute the adaptive subdivision.
4     if (physics(C)) {
–         // Run the adaptive subdivision, if no subdivision is
–         // necessary, grow the surface.
5         if (!adaptive_subdivision(M)) {
–             // Check that the maximum size of the surface has not been reached
6             if (segments > max_segments) return;
7             mesh N = add_ring(M);
8             merge C with M;
9             M = N;
–             // Recalculate the normal at each vertex
10            set_normals(C);
–             // Set the rest angles between on the new edges
11            forall a in M {
12                forall b in a {
13                    if (a < b) {
14                        if (a$r_no == b$r_no)
–                             // Set the rest angle on the edges in the topmost ring
15                            (a|b).rest_angle =
16                                r_rest_angle(double(a$r_no) / double(max_segments));
17                        else
–                             // Set the rest angle on the edges between rings
18                            (a|b).rest_angle = rest_angle(a, b);
19                    }
20                }
21            }
22        }
23    }
24 }
```

Algorithm 8.3: Addition of vertices to the daffodil corona

```
1  mesh add_ring(mesh & R) {
2      mesh N;
–      // Traverse the vertices in the topmost ring
3      forall v in R {
–          // Find the vertex to the right of v
4          vertex r = any in v;
5          while (!(r$r_no == v$r_no && (prevto r in v)$r_no == v$r_no - 1))
6              r = nextto r in v;
–          // Create a new vertex and attach it to the
–          // current vertex on the ring
7          unsigned int r_no = v$r_no + 1;
8          Pt pos = 0.5 * (v$pos + r$pos);
9          Pt offset = pos - (prevto r in v)$pos;
```

```
10          pos += offset * 0.5;
11          pos += 0.001 * (v$normal + r$normal);
12          vertex n(r_no, pos, Pt(), Pt(), Pt(), false);
13          make { v } nb_of n;
14          splice n after r in v;
15          add n to N;
16      }
17      synchronise R;
-       // Traverse the new vertices and set their neighbourhoods
18      forall a in N  {
19          vertex b = any in a;
20          vertex br = prevto a in b;
21          vertex bl = nextto a in b;
22          vertex ar = prevto b in `br;
23          vertex al = nextto b in `bl;
24          make { al, b, br, ar } nb_of a;
25          splice al after a in b;
26      }
-       // Set the rest lengths of the new edges
27      forall a in N  {
28          forall b in a  {
29              if (a$r_no == b$r_no)
-                   // Set the rest length on the edges on topmost ring
30                  (a|b).rest_length =
31                      r_rest_length(double(a$r_no) / double(max_segments), M.vertexCount());
32              else
-                   // Set the rest length on the edges between the topmost ring
-                   // and the one below
33                  (a|b).rest_length =
34                      l_rest_length(0.5 * double(a$r_no + b$r_no) / double(max_segments));
35          }
36      }
37      return N;
38  }
```

Algorithm 8.4: Physics of the daffodil corona

```
1   bool physics(mesh & M)  {
-       // Reset the accelerations on all the vertices
2       forall a in M  {
3           a$accel.zero();
4       }
-       // Calculate the force acting on each vertex
5       forall a in M  {
6           forall b in a  {
```

```
7             if (a < b)  {
–                 // Calculate the force exerted by the compression
–                 // or strectch on the spring between a and b
8                 Pt s = a$pos - b$pos;
9                 double d = s.length();
10                Pt force = (((a|b).rest_length - d) / d) * s;
11                a$accel += force;
12                b$accel -= force;
–                 // Skip the rest of the loop if the current edge
–                 // is on the topmost or bottomost ring
13                if (a$r_no == 0 && b$r_no == 0) continue;
14                if (a$r_no == segments && b$r_no == segments) continue;
–                 // Find the two vertices that complete the triangles
–                 // that complete the triangles that share the edge
–                 // from a to b
15                vertex l = nextto a in b;
16                vertex r = prevto a in b;
17                Pt al = a$pos - l$pos; al.normalise();
18                Pt bl = b$pos - l$pos; bl.normalise();
19                Pt ar = a$pos - r$pos; ar.normalise();
20                Pt br = b$pos - r$pos; br.normalise();
21                Pt nl = al.cross(bl);
22                Pt nr = br.cross(ar);
23                nl.normalise();
24                nr.normalise();
–                 // Find the angle between the two triangles
25                double cos_alpha = util::clamp(nl * nr, -1.0, 1.0);
26                double alpha = acos(cos_alpha);
27                Pt lab = 0.5 * (al + bl);
28                double cos_lab_nr = lab * nr;
29                if (cos_lab_nr > 0.0) alpha = -alpha;
–                 // Apply a force to the vertices relative to the difference
–                 // between the actual angle between the triangles and the
–                 // rest angle across the edge
30                double da = (a|b).rest_angle - alpha;
31                double mag = flex * da * resistance(0.5 * double(a$r_no + b$r_no)
32                    / double(max_segments));
33                Pt f = 0.5 * 0.5 * (nr + nl) * mag;
34                l$accel += f;
35                r$accel += f;
36                a$accel -= f;
37                b$accel -= f;
38            }
39        }
40    }
–     // Adjust the position of the vertices due to the forces acting on them
41    bool settled = true;
42    forall a in M  {
```

```
43          a$vel += a$accel * dt;
44          a$vel -= ‘a$vel * drag * dt;
45          a$pos += a$vel * dt;
46          if (settled && a$accel.length() > 0.01) settled = false;
47      }
48      return settled;
49  }
```

Algorithm 8.5: Adaptive subdivision on the daffodil

```
1   bool adaptive_subdivision(mesh & S)  {
–       // Clear the subdivision flag before determining the new
–       // region to be subdivided
2       bool subdivided = false;
3       forall v in S  {
4           v$selected = false;
5       }
–       // Determine the length of the maximum edge length based on the
–       // number of rings in the surface
6       double s_frac = double(segments) / double(max_segments);
7       double d = std::max(1.0 - s_frac, 0.1);
–       // Select the vertices that are along the topmost ring that are
–       // adjacent to an edge longer than above calculated length
8       forall a in S  {
9           if (subdivided) break;
10          forall b in a  {
11              if (a < b && b$r_no == segments)  {
12                  if ((a|b).rest_length > d && a$pos.distance(b$pos) > d)  {
13                      subdivided = true;
14                      break;
15                  }
16              }
17          }
18      }
–       // Expand the selected region to the vertices adjacent
–       // to those already selected
19      if (subdivided)  {
20          forall a in S  {
21              a$selected = true;
22          }
23      }
–       // Perform an adaptive polygonal subdivision on the
–       // selected vertices
24      mesh N;
25      synchronise S;
```

```
–     // Insert new vertices between the adjacent pairs of
–     // selected vertices
26    forall a in m {
27        forall b in `a {
28            if (b$selected && a < b) {
29                double rl = 0.5 * (a|b).rest_length;
30                double ra = (a|b).rest_angle;
31                vertex v = insert(a, b);
32                v$pos = 0.5 * (a$pos + b$pos);
33                v$vel = 0.5 * (a$vel + b$vel);
34                v$selected = true;
35                v$r_no = std::min(a$r_no, b$r_no);
36                if (v$r_no == segments) {
37                    add v to M;
38                }
39                (a|v).rest_length = rl;
40                (v|b).rest_length = rl;
41                (a|v).rest_angle = ra;
42                (v|b).rest_angle = ra;
43                add v to N;
44            }
45        }
46    }
–     // Assign the neighbourhoods to the new vertices, depending on if
–     // these vertices are at the edge of the selected subdivision region.
–     // The physical properties, the rest length and anlges of the new
–     // edges are also set.
47    synchronise N;
48    forall v in N {
49        vertex a = any in v;
50        vertex b = nextto a in v;
51        {
52            vertex x = nextto v in a;
53            vertex y = prevto v in b;
54            if (x == y) {
55                splice x before a in v;
56                splice v after a in x;
57                (x|v).rest_length = x$pos.distance(v$pos);
58                (x|v).rest_angle = 0.5 * ((a|x).rest_angle + (b|x).rest_angle);
59            }
60            else if (nextto a in `x == prevto b in `y) {
61                splice x before a in v;
62                splice y after b in v;
63                (v^x).rest_length = x$pos.distance(v$pos);
64                (v^y).rest_length = y$pos.distance(v$pos);
65                (v^x).rest_angle = 0.5 * ((a|v).rest_angle + (a|x).rest_angle);
66                (v^y).rest_angle = 0.5 * ((b|v).rest_angle + (b|y).rest_angle);
67            }
```

```
68          }
69          {
70              vertex x = prevto v in a;
71              vertex y = nextto v in b;
72              if (x == y)  {
73                  splice x after a in v;
74                  splice v before a in x;
75                  (x|v).rest_length = x$pos.distance(v$pos);
76                  (x|v).rest_angle = 0.5 * ((a|x).rest_angle + (b|x).rest_angle);
77              }
78              else if (prevto a in `x == nextto b in `y)  {
79                  splice x after a in v;
80                  splice y before b in v;
81                  (v^x).rest_length = x$pos.distance(v$pos);
82                  (v^y).rest_length = y$pos.distance(v$pos);
83                  (v^x).rest_angle = 0.5 * ((a|v).rest_angle + (a|x).rest_angle);
84                  (v^y).rest_angle = 0.5 * ((b|v).rest_angle + (b|y).rest_angle);
85              }
86          }
87      }
88      merge C with N;
89      set_normals(m);
90      return subdivided;
91  }
```

8.3 Remarks on Modelling Growing Boundaries

The examples in this chapter illustrate that vv can be used effectively produce models of complex phenomena with a simple growth pattern. In each case, the connectivity of the polygon meshes were only altered by the addition of vertices to one boundary. In combination with other rules, be it a description of shape as with the sea shell (§8.1) or a physical simulation as with the daffodil corona (§8.2), a wide variety of biological phenomena can be modelled easily using vv.

Chapter 9

Phyllotaxis and the Apex

By combining the modelling strategies described in Chapters 5, 7 & 8 more complex biological models can be constructed where the development is the result of combining the principles demonstrated in the previous chapters. That is, the development is not a single growth pattern driven by a single driving principle. In this chapter, a model of a shoot apical meristem with growing primordia is presented. This model combines adaptive subdivision, physics, patterns of local and boundary growth and pseudo-chemical simulations all in conjunction to get a complex, growing shape.

9.1 Biological Principles of Phyllotaxis and the Shoot Apical Meristem

Phyllotaxis, in its broadest sense, is the arrangement of leaves, flowers, branches and other organs on a plant [1]. Phyllotaxis reflects the arrangement of *primordia*, the initial cells clusters that become the leaves, flowers and branches, on the *shoot apical meristem*, or the *shoot apex*, the topmost part of a plant's shoot. For an overview of the shoot apex's structure and development see [70, 42]; a short description of the pertinent information about the structure and development follows.

9.1.1 Spiral Phyllotaxis

The model described in this chapter produces a *spiral* phyllotaxy, whereby each consecutive primordium is produced at approximately the *golden angle** from the previous primordium. Figure 9.1 shows a schematic representation of how primordia can be arranged in a spiral phyllotaxis on an apex.

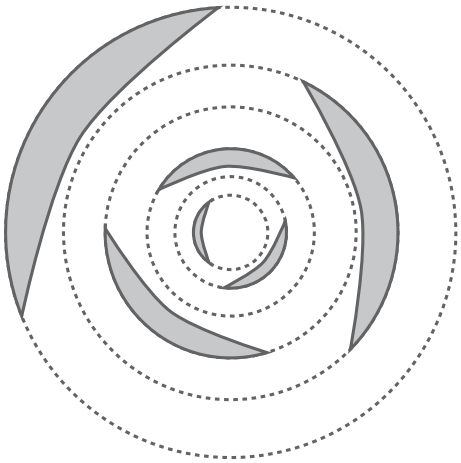


Figure 9.1: A schematic representation of primordia on an apex in a spiral phyllotaxis; each primordium occurs at at the golden angle from the previous

9.1.2 Structure of the Shoot Apex

The parts of the shoot apex can first be categorised into the *tunica*, the outermost layer of cells, and the *corpus*, the layers of cells under the tunica, distinguishing the exterior and interior of the apex. This model considers the activity on the tunica that is considered, whereas the corpus is only considered to the extent that it is the structural support of the apex.

*The golden angle, ϕ, is based on the golden ratio, τ, introduced in Euclid's *Elements of Geometry*, such that $\phi = 360^\circ - \tau 360^\circ \approx 137.5^\circ$ where $1 : 1 + \tau = \tau : 1$. For a history of the golden ratio and the golden angle see [34].

Two regions of the tunica warrant special consideration: the *quiescent centre* and the *active ring* (Figure 9.2). The quiescent centre is the few topmost cells of the apex. These cells are the stem cells for apex growth, which rarely divide. The quiescent centre is the source of stem cells for the shoot. The active ring is a narrow band of cells at that encircles the quiescent centre. Cells in the active ring that cells divide more frequently produce the phyllotactic patterning. As a part of this patterning, cells differentiate and become the first cells of the primordia. Below the active ring, the primordia grow and develop into the various plant organs.

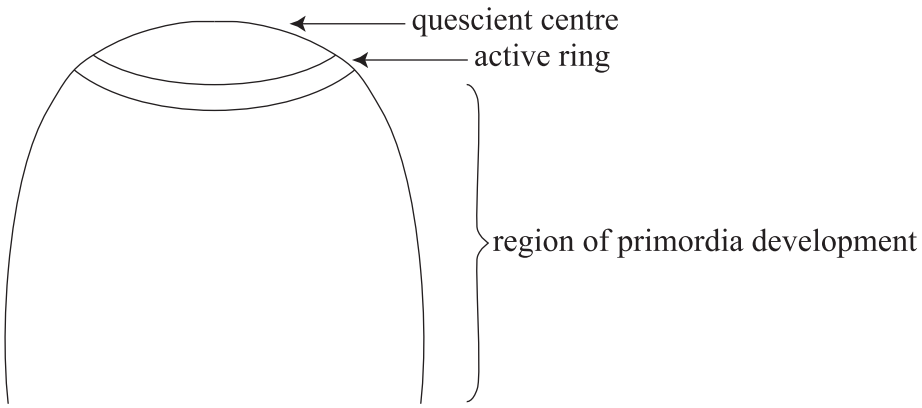


Figure 9.2: Major zones of the tunica of a shoot apex

9.2 A VV Model of a Growing Shoot Apex

The main function of this apex model is given in Algorithm 9.1. The model's functions are in two parts: the chemical and physical simulations (§9.2.2) and growth which involves the addition of new vertices at the active ring and the primordia (§9.2.3).

Algorithm 9.1: Main function of the phyllotaxis model

```
1  void phyllotaxis()  {
2      t += dt;
3      delay -= dt;
4      synchronise C;
5      chemical_simulation();
6      physics_simulation();
7      check_active_ring();
8      if (delay > 0.0) return;
9      delay = delay_interval;
10     find_subdivision_regions();
11     adaptive_subdivision();
12 }
```

9.2.1 An Overview of the Model's Structure

The apex model depicts a cylinder composed of rings linked with triangular polygons, somewhat similar to that of the daffodil corona model in §8.2. The physics simulation shapes this cylinder like a dome. The topmost ring of this cylinder is the active ring. Because this is a cylinder, there is a hole at the top, but it is small. That there is no structure at this hole is not of concern as it corresponds to the quiescent centre, a region where little relevant activity occurs.

There are four vertex sets globally declared in this vv program: C, the set of all vertices, R, the set of vertices in the active ring, P, the set of vertices around which primordia grow, and S, a set used to track regions around the vertices in P.

9.2.2 The Simulations of the Shoot Apex Model

The Inhibitor-Diffusion Simulation

To a model a growing shoot apical meristem, a very simple inhibitor-diffusion model of phyllotaxis is here used. Although this model is inconsistent with the current

understanding of the genetic and hormonal controls in the shoot apex, it is simple to implement and produces a realistic spiral phyllotaxis pattern. This model assumes that each primordium produces a chemical that inhibits the differentiation of cells into primordia, similar to the model described in [76], and that this chemical diffuses around the active ring of the apex.

The simulation incorporates the assumption that each cell on the active ring produces an inhibitor and this inhibitor diffuses on the ring. Cells, represented by vertices in the active ring, with a high primordia "identity", characterised by the prim_effect property, produce more of the inhibitor than other cells. At each step, the amount of inhibitor produced by each cell decreases as the prim_effect parameter decreases. When the amount of inhibitor in a cell is sufficiently low, the prim_effect is set to the maximum again and the cell behaves like a newly formed primordium on the active ring. This is all implemented in Algorithm 9.2.†

Algorithm 9.2: The pseudo-chemical simulation

```
1  void chemical_simulation() {
2      forall c in R {
3          if (c$pos.y() > min_y) min_y = c$pos.y();
4          double diff = 0.0;
   –          // Remove some of the inhibitor due to decay
5          diff -= nu * c$inhibitor;
6          forall v in c {
   –              // Diffuse the inhibitor between vertices on the active ring
7              if (!v$active) continue;
8              diff -= Dh * (c$inhibitor - 'v$inhibitor);
9          }
   –          // Create more inhibtor based on the primordia identity factor
10         diff += (64.0 - c$inhibitor) * rate_of_age(c$prim_effect / 150.0);
11         c$inhibitor += diff * dt;
   –          // Decrease the primordia identity factor
12         c$prim_effect -= dt;
13     }
```

†This part of the model is based on an unpublished L-system by Prusinkiewicz.

```
14 }
```

The Physical Simulation

The physical simulation of the apex uses a combination of a mass-spring system combined with an approximation of pressure much like was used in the examples in Chapter 7; however, this model includes some features that were not in the previous examples.

One large difference in the mass-spring system is that the spring lengths are adjusted according to the normal vectors (line 7), such that regions with normals pointed downwards have longer spring lengths. As a result is that the abaxial sides of the primordia expand more than the adaxial sides, so primordia grow upwards. This is a simple implementation of the tropism seen in many primordia [87].

Likewise, pressure is reduced at the growing parts of the primordia (lines 20 – 22). This feature smoothes out the growth in the regions where new primordia have just been added. If this is not done, new vertices tend to "pop" out as they would be under too high a pressure.

Algorithm 9.3: The physics simulation

```
1  void physics_simulation() {
2      forall c in C {
3          c$age += dt;
4          c$acceleration = Pt();
5          Pt normal;
–          // Calculate the force exerted by the springs
6          forall v in c {
–              // Adjust the rest length such that springs on the
–              // abaxial side are longer
7              double length = (c$normal * up >= -0.5 || v$normal * up >= -0.5) ? 0.1 : 0.2;
–              // Calculate the force exerted by the spring
8              double dist = c$pos.distance(v$pos);
```

```
 9              double k = (length - dist) / dist;
10              c$acceleration += k * (c$pos - v$pos);
 –              // Calculate the vector normal to the adjacent triangle
 –              // and add it to the vertices so that an average normal
 –              // vector can be found at each vertex
11              Pt vc = v$pos - c$pos;
12              vc.normalise();
13              Pt uc = (nextto v in c)$pos - c$pos;
14              uc.normalise();
15              normal += vc.cross(uc);
16          }
17          normal.normalise();
18          c$normal = normal;
 –          // Apply a force in the normal direction. This force is smaller
 –          // in the regions where the primordia are growing.
19          if (c$visited < 2)
20              c$acceleration += 0.1 * normal;
21          else
22              c$acceleration += 0.05 * c$visited * normal;
23          double local_drag = (c$visited < 1) ? drag : 0.5;
 –          // Find the new positions of the vertices
24          c$velocity += (c$acceleration - c$velocity * local_drag) * dt;
25          c$pos += c$velocity * dt;
 –          // Set all the vertices in the active ring to the same height
26          if (c$active)  {
27              c$pos.x(‘c$pos.x());
28              c$pos.z(‘c$pos.z());
29              if (c$pos.y() < min_y) c$pos.y(min_y);
30          }
 –          // Set all the vertices in the lowermost ring to the same height
31          else if (c$base)  {
32              c$pos.y(0.0);
33          }
34          if (c$pos.y() < 0.0) c$pos.y(0.0);
35      }
36  }
```

9.2.3 Growth of the Shoot Apex Model

Growth Near the Active Ring

Growth near the active ring is done by the addition of a new ring of vertices at the top of the cylinder. This technique has already been seen in examples in Chapter

8. The first function (Algorithm 9.4) scans the active ring (the set R) and checks for any vertices where the inhibitor is below the threshold. When such a vertex is found, it is added to the set P and its prim_effect property is set to the maximum value.

Algorithm 9.4: Check the active ring for new primordia

```
1  void check_active_ring() {
2      bool addring = false;
–      // Check to see if any of the vertices in the active
–      // ring have an inhibitor level below the threshold
3      forall c in R {
4          if (c$inhibitor > thr) continue;
5          addring = true;
6          c$suppress = false;
7          add c to P;
8          c$inhibitor = 64.0;
9          c$prim_effect = 150.0;
10     }
11     if (addring) {
–          // Increase the age of the vertices in the active ring
12         for (unsigned int i = 0; i < ring_space; ++i) {
13             synchronise C;
14             forall c in C {
15                 c$age += delay;
16             }
17             add_ring();
18         }
19     }
20 }
```

Afterwards, if vertices with inhibitor levels below the threshold were found, a new ring of vertices is added to the top of the cylinder (Algorithm 9.5).

Algorithm 9.5: Add a new ring to the top of the mesh

```
1  void add_ring() {
2      static bool d_switch = false;
3      d_switch = !d_switch;
4      mesh N;
–      // Create the new vertices that will form the new active ring
```

```
5     forall c in R  {
6         vertex r = any in ‘c;
7         while (!r$active) r = nextto r in ‘c;
8         if ((prevto r in ‘c)$active) r = prevto r in ‘c;
9         Pt pos = 0.5 * (c$pos + r$pos) + ring_offset;
10        double inh = d_switch ? c$inhibitor : r$inhibitor;
11        double p_eff = d_switch ? c$prim_effect : r$prim_effect;
12        vertex n(pos, Pt(), Pt(), Pt(), inh, 0.0, p_eff, true, false, false, true, -1);
13        make { c, r } nb_of n;
14        splice n after r in c;
15        splice n before c in r;
16        add n to N;
17    }
  –   // Assign the new vertices their complete neighbourhoods
18    forall c in N  {
19        vertex c = any in c;
20        if (‘(next(2)to c in c)$active) c = nextto c in c;
21        vertex d = nextto c in c;
22        make { nextto c in d, d, c, prevto c in c } nb_of c;
23        c$active = false;
24        d$active = false;
25    }
  –   // Calculate the normals for the new vertices
26    forall v in N  {
27        Pt normal;
28        forall x in v  {
29            Pt xv = x$pos - v$pos;
30            xv.normalise();
31            Pt yv = (nextto x in v)$pos - v$pos;
32            yv.normalise();
33            normal += xv.cross(yv);
34        }
35        normal.normalise();
36        v$normal = normal;
37    }
38    merge C with N;
39    R = N;
40 }
```

Growth of the Primordia

Growth around the primordia uses a localised vertex insertion that first selects a region around each vertex in set P and then uses a simple adaptive surface polyhedral subdivision on those regions. The region selection is done using two functions

(Algorithms 9.6 & 9.7). The first function (Algorithm 9.6) prepares the model by setting all the vertices as "unvisited" and none are selected for subdivision (lines 2 – 5). The selected region is the vertex from p, with the vertices on the 2-ring that are on the rows of the cylinder above, below and the same as that vertex. The age property of each vertex is a measurement of how far it is along the cylinder away from the top and it is used to limit the region to the three rows. The selection out to the 2-ring is done by recursing away from the vertex.

Algorithm 9.6: Find the regions around the primordia for adaptive subdivision

```
1  find_subdivision_regions() {
   –    // Reset the counter walking counter
2      forall c in C {
3          c$visited = -1;
4      }
5      clear S;
   –    // Execute the recursive walk around vertices that are marked as
   –    // primordia, but not too close to the active ring
6      forall p in P {
7          if (p$age < divide || p$suppress) continue;
8          p$visited = k;
9          double min_age = p$age;
10         double max_age = p$age;
   –        // Determine the ages of the vertices in the 1-ring
11         forall x in p {
12             min_age = std::min(min_age, x$age);
13             max_age = std::max(max_age, x$age);
14         }
15         find_candidates(p, S, min_age, max_age);
16     }
17 }
```

Algorithm 9.7: Select the vertices for the adaptive subdivision

```
1  void find_candidates(vertex v, mesh & V, double min_age, double max_age) {
2      add v to V;
3      forall x in v {
4          if (!util::range_closed(min_age, max_age, x$age)) continue;
```

```
5           if (x$visited < v$visited) x$visited = v$visited - 1;
6       }
7       forall x in v  {
8           if (!util::range_closed(min_age, max_age, x$age)) continue;
9           if (x$visited >= v$visited || x$visited == 0 || x$base || x$active) continue;
10          find_candidates(x, V, min_age, max_age);
11      }
12  }
```

When the regions are selected, they are subdivided adaptively (Algorithm 9.8). This algorithm largely resembles and functions similar to the adaptive subdivision algorithm given in Algorithm 5.12, save that each vertex here has many more parameters that need to be initialised on creation.

Algorithm 9.8: Adaptive subdivision

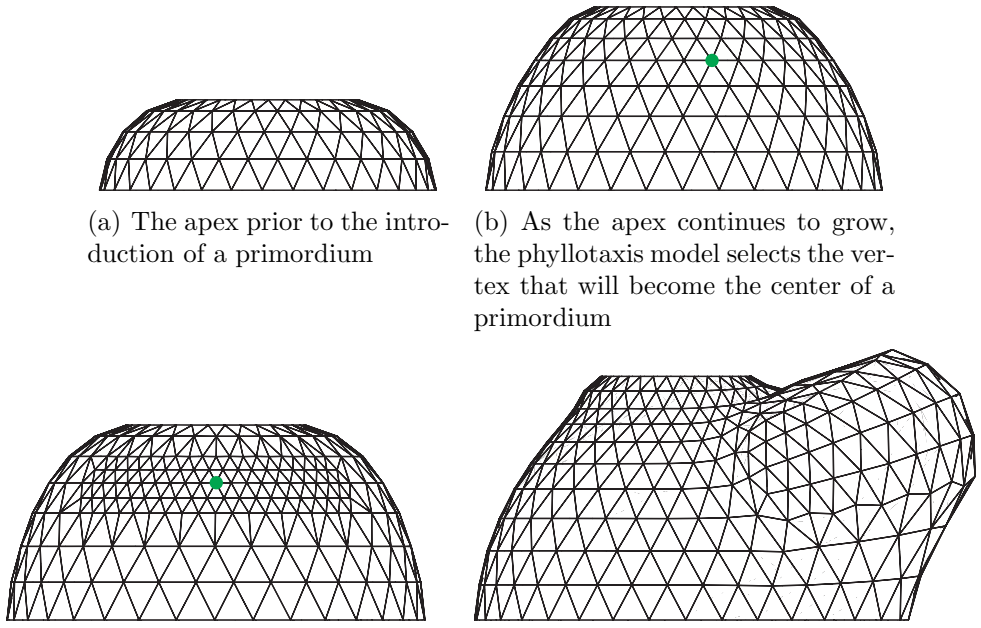
```
1   void adaptive_subdivision()  {
2       mesh N;
-       // Create a new vertex between pairs of existing vertices
-       // in the selected region
3       forall v in S  {
4           forall x in ‘v  {
5               if (x < v) continue;
6               if (x$visited < 1) continue;
7               Pt pos = 0.5 * (v$pos + x$pos);
8               double inh = 0.5 * (v$inhibitor + x$inhibitor);
9               double age = 0.5 * (v$age + x$age);
10              int visited = (v$visited + x$visited) / 2;
11              vertex n(pos, Pt(), Pt(), Pt(), inh, age, 0.0, false, false,
12                  x$base && v$base, false, visited);
13              add n to N;
14              make { v, x } nb_of n;
15              replace x with n in v;
16              replace v with n in x;
17          }
18      }
-       // Complete the neighbourhoods of the new vertices
19      forall v in N  {
20          vertex a = any in v;
21          vertex b = nextto a in v;
22          vertex w = nextto v in a;
23          vertex x = prevto v in a;
24          vertex y = nextto v in b;
```

```
25          vertex z = prevto v in b;
26          splice w before a in v;
27          if (w == z) splice v after a in w;
28          else splice z after b in v;
29          splice y before b in v;
30          if (y == x) splice v after b in y;
31          else splice x after a in v;
32      }
-       // Calculate the normals of the new vertices
33      forall v in N  {
34          Pt normal;
35          forall x in v  {
36              Pt xv = x$pos - v$pos;
37              xv.normalise();
38              Pt yv = (nextto x in v)$pos - v$pos;
39              yv.normalise();
40              normal += xv.cross(yv);
41          }
42          normal.normalise();
43          v$normal = normal;
44      }
45      merge C with N;
46  }
```

9.3 Results of the Shoot Apex Model

The first stages of growth of this apex model are shown in Figure 9.3 and the model after a longer simulation time is shown in Figure 9.4.

This model succeeds in producing a model of a growing apex with growing primordia, although admittedly it is not a highly-realistic representation of the apex shape. Nevertheless, phyllotaxis being the subject of many previous models [22, 76, 77, 80, 83, 58, 62, 12, 44], no other model simulated a growing apex with growing apices. Therefore, this model represents a step forwards in modelling the physical aspects of the shoot apical meristem.



(a) The apex prior to the introduction of a primordium

(b) As the apex continues to grow, the phyllotaxis model selects the vertex that will become the center of a primordium

(c) The region around the selected vertex is adaptively subdivided

(d) The apex has been rotated to best show the resulting shape

Figure 9.3: The first stages of growth of a model of the shoot apical meristem

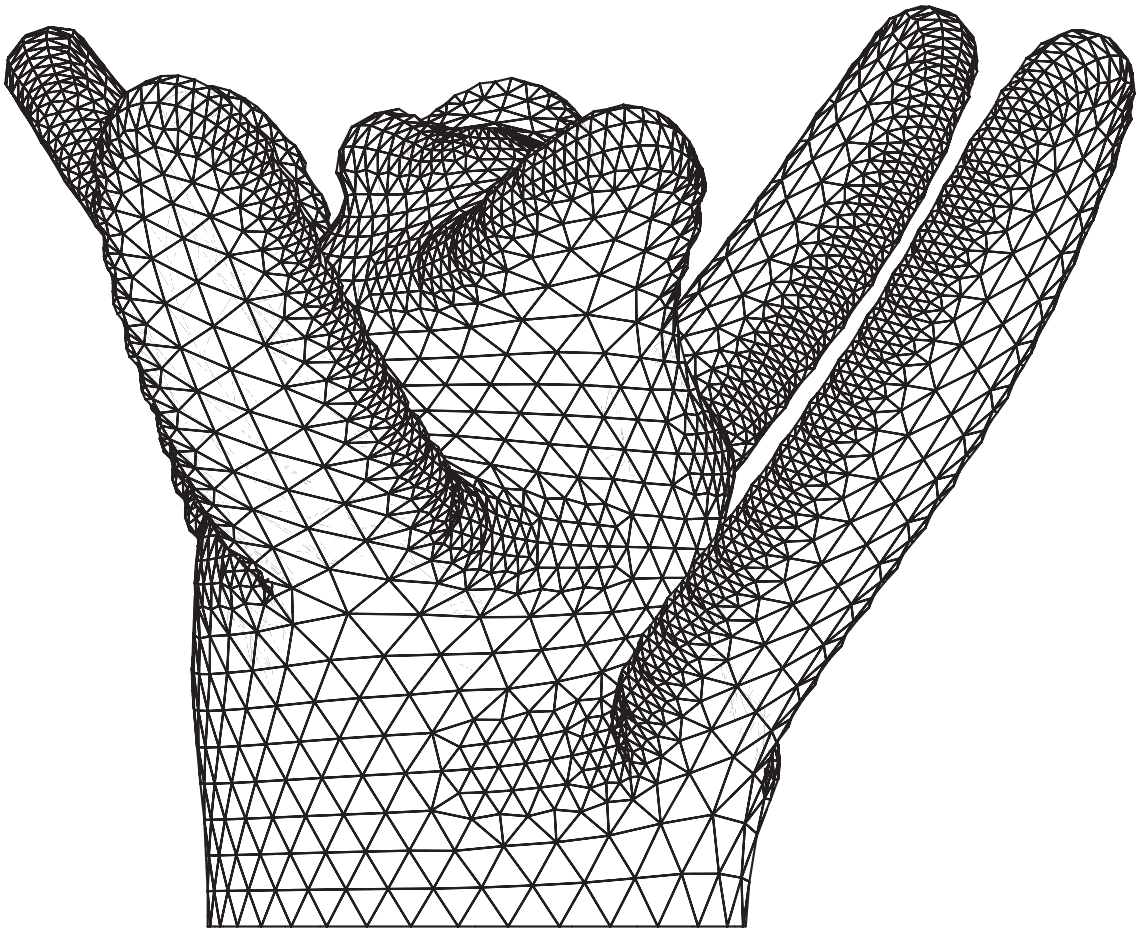


Figure 9.4: An apex with many primordia

Chapter 10

Canvas-Coordinated Growth

A strategy that has not yet been explored in this thesis is the possibility of using information from large scale processes as the parameters for the processes at small scales that drive the development of a $(DS)^2$. In some situations, this allows interaction of processes at global and local scales. This multi-scale principle is here demonstrated using the principle of a *canvas* [7, 8].

A canvas is a space on which transformations can be applied. As the canvas is transformed, the components of any contained structures are likewise transformed. For example, a canvas could be a grid that is marked with points. As the grid is stretched, the points on the grid move apart (see Figure 10.1).

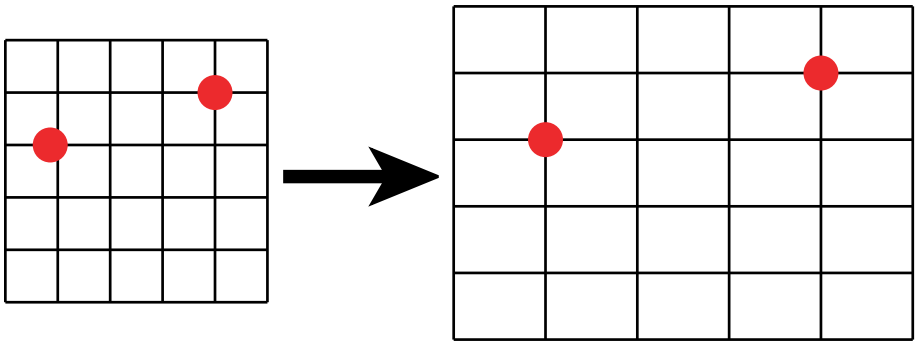


Figure 10.1: As the grid is stretched, the two points grow apart

One biological application of a growing canvas is to model a growing tissue. For example, the radial growth of a tissue can be considered as a global process. This sort of canvas growth was considered and modelled by Nakielski [46] and is here reproduced as a vv program. An example of a tissue with a radial growth is the

outermost layer of cells on a shoot meristem.

As the tissue grows, its cells divide when they when they reach a threshold size. For this model, a cell is composed as a polygonal cell wall and a vertex at the cell wall's barycentre, as shown in Figure 10.2. A vertex at the centre of each cell allows for an entry point in the structure for each cell; therefore, it is possible to visit each cell by iterating over the set of cell-centre vertices.

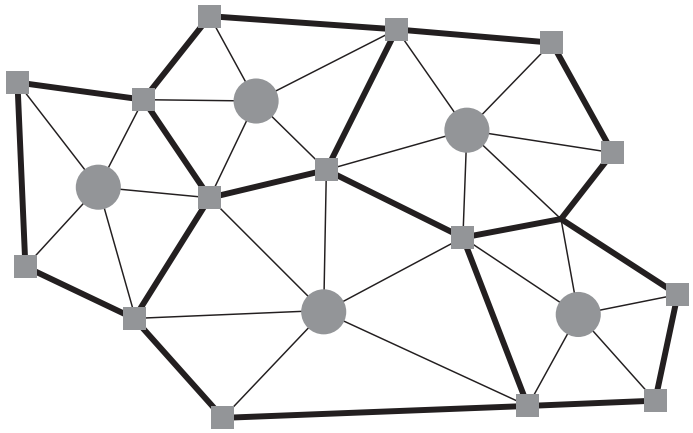


Figure 10.2: A cell structure with the vertices on the cell walls depicted as squares and the vertices at the cell centres as dots

The growth of the tissue is then defined as the velocity of the vertices in cell walls from the tissue's centre. This can be defined as a function of speed versus the distance from the centre. When the area of the polygon defined by the cell wall surpasses a threshold, the cell divides. Thus, the global process of tissue growth signals the local process of cell division (Figure 10.3).

The vv program for this model is given in Algorithm 10.1. In this program, there are three sets of vertices: S, the set of all vertices, C, the vertices at the cell centres (those marked as dots in Figure 10.2) and J, the vertices at the junction in the cell walls (those marked by squares in Figure 10.2). In lines 2 – 6, the vertices on the

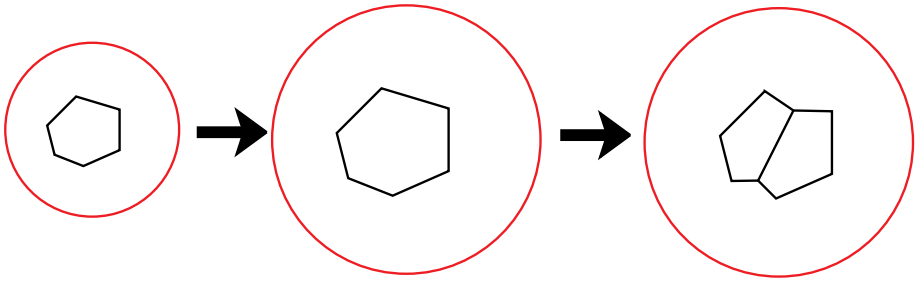


Figure 10.3: (a) A cell on the canvas (b) The cell grows and reaches a maximum area (c) The cell divides

cell walls are moved outwards from the centre of the tissue (*i.e.* the transformation on the canvas) at a constant speed. In lines 15 – 19, the area of each cell is found and when the cell's area is sufficiently large (line 20), the cell is divided (lines 21 – 117).

The cell division starts by first finding the point on the cell wall closest to the cell's centre, found by projecting the cell centre onto each segment of the cell wall (lines 26 – 39). Then, a new vertex is inserted into the cell wall at that point and connected to the cell's centre (lines 40 – 46) and, if necessary, it is connected to the centre of the adjacent cell (lines 48 – 52). Then, another vertex is inserted on the opposite side of the cell, found by intersecting the line from the cell centre to the closest point on the wall with the the rest of the cell wall (lines 54 – 69). Again, if necessary, this second new vertex is connected to the centre of an adjacent cell (lines 70 – 76). Next, two new cell centres are created (lines 77 – 84) and connected to the new vertices on the cell wall (lines 85 – 92). These new cell centres are connected to the existing cell walls by walking over each half of the old cell starting and ending at the new vertices on it (lines 93 – 108). Finally the old cell centre is marked for removal (line 117). After all the cells have been processed and possibly divided, cell

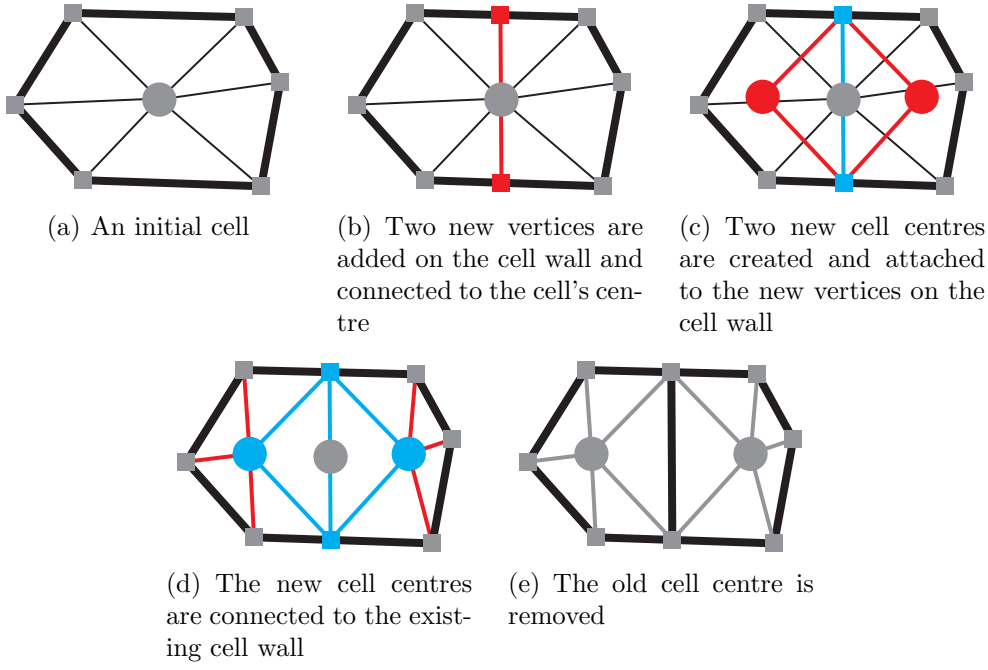
centres marked for removal are removed (lines 120 – 123). This cell division process is illustrated in Figure 10.4.

Algorithm 10.1: A growing radial canvas

```
1   void cells_on_radial_canvas(mesh & S, mesh & C, mesh & J) {
-       // Displace the vertices of the cell walls according
-       // to the canvas transformation
2       forall j in J {
3           Pt direction = j$pos;
4           direction.normalise();
5           j$pos += direction * speed * dt;
6       }
7       mesh N;
8       mesh D;
9       forall c in C {
-           // Place the centre vertex at the barycentre of the cell's wall
10          c$pos.zero();
11          forall j in c {
12              c$pos += j$pos;
13          }
14          c$pos /= double(valence c);
-           // Calculate the cell's area
15          double area = 0.0;
16          forall a in c {
17              vertex b = nextto a in c;
18              area += util::planar_triangle_area(a$pos, b$pos, c$pos);
19          }
20          if (area < max_area) continue;
-           // Find the position on the cell wall closest to the barycentre
21          double shortest = DBL_MAX;
22          vertex j_short = 0;
23          vertex j_next = 0;
24          Pt pos;
25          bool found_proj = false;
26          forall j in c {
27              vertex jn = nextto j in c;
28              Pt a = c$pos - j$pos;
29              Pt b = jn$pos - j$pos;
30              Pt proj = (a * b / b.length_sq()) * b;
31              double distance = a.distance(proj);
32              if (assert_between_points(Pt(), proj, b) && distance < shortest) {
33                  found_proj = true;
34                  j_short = j;
35                  j_next = jn;
36                  shortest = distance;
```

```
37              pos = j$pos + proj;
38          }
39      }
-       // Insert a vertex on the cell wall at the point
-       // closest to the barycentre
40      vertex n = insert(j_short, j_next);
41      n$pos = pos * shortening + c$pos * (1.0 - shortening);
42      n$type = 'j';
43      add n to S;
44      add n to J;
45      splice c before j_short in n;
46      splice n after j_short in c;
47      {
48          vertex opp = prevto n in j_short;
49          if (opp == nextto n in j_next)  {
50              splice opp after j_short in n;
51              splice n after j_next in opp;
52          }
53      }
-       // Find the point on the cell wall opposite to the
-       // newly inserted vertex and insert a vertex there
54      vertex x = n;
55      vertex y = nextto x in c;
56      bool found = false;
57      while (!found && y != n)  {
58          x = y;
59          y = nextto x in c;
60          bool ib = false;
61          pos = util::planar_line_intersection(x$pos, y$pos, c$pos, n$pos, found, ib);
62      }
63      vertex o = insert(x, y);
64      o$pos = pos * shortening + c$pos * (1.0 - shortening);
65      o$type = 'j';
66      add o to J;
67      add o to S;
68      splice c after y in o;
69      splice o after x in c;
70      {
71          vertex o = nextto o in y;
72          if (o == prevto o in x)  {
73              splice opp after x in o;
74              splice o after y in opp;
75          }
76      }
-       // Create two vertices that are the two new cell centres
77      vertex cl;
78      vertex cr;
79      add cl to S;
```

```
80          add cr to S;
81          add cl to N;
82          add cr to N;
83          cl$type = 'c';
84          cr$type = 'c';
-           // Attach the two new cell centres to the newly inserted
-           // vertices on the cell wall
85          make { o, n } nb_of cl;
86          make { n, o } nb_of cr;
87          splice cl after c in n;
88          splice cr before c in n;
89          splice cl before c in o;
90          splice cr after c in o;
91          replace c with o in n;
92          replace c with n in o;
-           // Walk around the cell wall and attach the wall to the new
-           // cell centres where it was connected to the cell centres
93          {
94              vertex x = nextto n in c;
95              while (x != o) {
96                  splice x before o in cr;
97                  replace c with cr in x;
98                  x = nextto x in c;
99              }
100         }
101         {
102             vertex x = nextto o in c;
103             while (x != n) {
104                 splice x before n in cl;
105                 replace c with cl in x;
106                 x = nextto x in c;
107             }
108         }
109         forall j in cl {
110             cl$pos += j$pos;
111         }
112         cl$pos /= double(valence cl);
113         forall j in cr {
114             cr$pos += j$pos;
115         }
116         cr$pos /= double(valence cr);
117         add c to D;
118     }
119     merge C with N;
120     forall c in D {
121         remove c from C;
122         remove c from S;
123     }
```



(a) An initial cell

(b) Two new vertices are added on the cell wall and connected to the cell's centre

(c) Two new cell centres are created and attached to the new vertices on the cell wall

(d) The new cell centres are connected to the existing cell wall

(e) The old cell centre is removed

Figure 10.4: The division of a cell as done in Algorithm 10.1. The vertices on the cell wall are depicted as squares and the vertices at the cell centres as dots. New components at each stage are shown in red.

The results of Algorithm 10.1 are shown in Figure 10.5. If the positions of the vertices are mapped to polar coordinates, it is a trivial matter to place the cells on a surface of revolution characterised by a profile curve.

A variable speed, instead of a constant speed, is desirable for modelling the shoot apical meristem because cells divide much slower near the centre than in other regions of the apex. The region where cells divide slower is the *quiescent centre* [70]. In Figure 10.6, the cells have been placed on a surface that is shaped like the shoot apical meristem of a plant and uses a variable speed function.

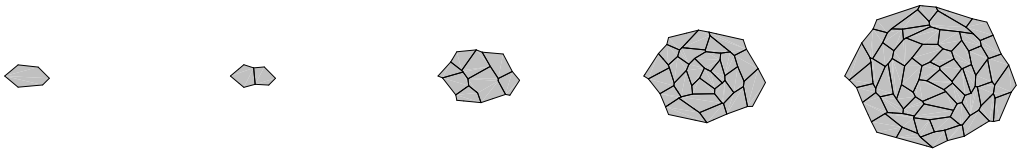


Figure 10.5: Cells growing on a radial canvas

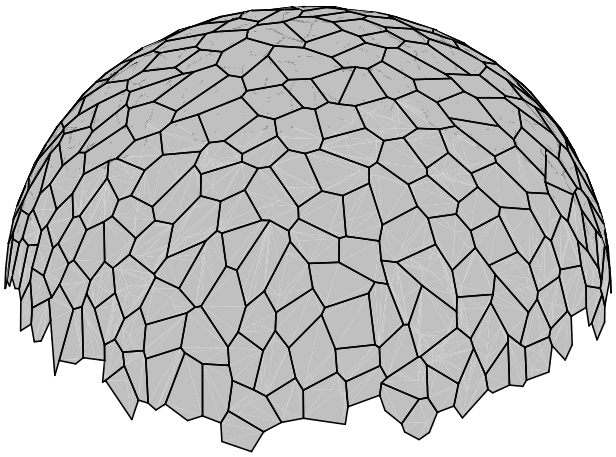


Figure 10.6: Cells on a canvas mapped to an apex-shaped surface and with cells that divide slower in the quiescent centre

Part IV

Evaluation and Conclusions

Chapter 11

Comparisons of VV to Other Polygon Mesh Structures

The examples presented in this thesis illustrate that vv is well suited for applications with polygon meshes. But, how does vv compare to other existing data structures for polygon meshes?

VV is designed to handle dynamical structures; polygon meshes that are static may be better handled by other data structures. For example, to render a polygon mesh in vv, it is necessary to walk around each vertex's neighbourhood; but, using a triangle strip structure, rendering in OpenGL can be performed much faster as the vertices and faces can be passed to the hardware much more efficiently. However, the cost of altering the connectivity of a mesh stored as a triangle strip is prohibitively high. There is a trade between high rendering speed and ease of transforming the mesh.

As part of the design that facilitates handling dynamical structure, vv has an algebra that defines quite clearly the transformations that can be applied to a structure. Most polygon mesh data structures, such as the half-edge [38] and winged-edge [3] data structures, do not have an algebra to accompany them. Consequently, the user must manipulate a collection of pointers to access and alter the structure, which can be an error-prone process. For example, a change in the pointers raises the question "Was that a valid transformation?". Without an algebra, a data structure is just a

schema for organising pointers; it lacks a proper interface to interact with the data it contains. Of course, these data structures are widely-used in the implementation of geometric models, and so in §11.1.1, comparisons of the vv data structure to these data structures is made.

Of course, some data structures have an algebra as part of their definition, of particular note is the quad-edge data structure [21]. So, what differentiates vv from other polygon mesh data structures that include an algebra? All the aforementioned data structures are index-free and so solve the problem of indices stated in §1.1. Beyond that, there are some significant features of vv that make it more useful than these other data structures.

11.1 On the Simplicity of the VV Data Structure

One key trait of vv is its very simple and "light-weight" data structure, especially when compared to other data structures. This simplicity has two definite advantages. Firstly, the amount of memory required to store a polygon mesh in memory using vv is less than that required by other data structures. Secondly, the simpler the data structure, the easier it is for a user to understand and figure out how to apply transformations.

The complexity of the vv data structure can be compared to other data structures using the notation proposed by Rossignac [60]. In this notation, the relations between structures are shown with arrows; a regular arrow denotes a set of pointers and a double arrow denotes an ordered set of pointers. A number over the arrow indicates a specific number of pointers. For example, an edge structure that contains a pointer

to another edge is notated as $E \rightarrow E$ or a face that points to an ordered list of four vertices uses the notation $F \stackrel{4}{\Rightarrow} V$. By listing the various pointer relations of a data structure using this notation, it is possible to describe most polygon mesh data structures. Moreover, this notation reveals the complexity of any particular data structure: the more arrows required to describe the data structure, the more pointers are involved. With more pointers, the storage requirements increase and the number of changes to the data to effect a transformation increase. Also from this notation, it is apparent what sort of traversals can be made over a polygon mesh using each data structure.

11.1.1 The Complexity of Various Polygon Mesh Data Structures

To show the differences in data structures, the complexity of the representation of a cube (six faces and eight vertices) is considered and the possible traversals are using each data structure. The results are summarised in Table 11.1.

Face-Vertex

The face-vertex structure is a simple and popular data structure, which is used in the widely used OBJ file format. Rossignac gave the notation for this data structure as

$$\{R, F, V : R \rightarrow F \Rightarrow V\},$$

which is read as a data structure where a region is composed of faces and each face is points to an ordered list of vertices. It can be seen that the face-vertex data structure has the advantage of being simple, but the arrows indicate that it is only possible to traverse from faces to vertices. It is not possible to traverse from vertices to faces,

there is no concept of neighbouring vertices across faces. Consequently, the face-vertex data structure is an awkward data structure to use in many situations. With pointers to six faces and four vertices per face, twenty-four pointers are required to store a cube using the face-vertex data structure.

Winged-Edge

The notation for the winged-edge data structure is given by Rossignac as

$$\{F, E, V : F \to E \overset{2}{\Rightarrow} V, E \overset{4}{\Rightarrow} E, V \to E \overset{2}{\Rightarrow} (F, E)\}.$$

Every face points to its edges and each edge points to two vertices. Each edge also points to its four neighbouring vertices. Each vertex points to the edges it is part of and each edge also points to two pairs consisting of the neighbouring face oriented by the edge on the counter-clockwise rotation of each face. To represent a cube with all those relations, one hundred and ninety-two pointers are required. Although this data structure does have the advantage that it is possible to traverse from any element of the representation to another, the storage requirement is much more than any other data structure shown here.

Half-Edge

The half-edge data structure uses a combination of vertices, half-edges, edges and faces to represent a polygon mesh. A half-edge is notated as fe. The L is a loop, an ordered cycle of edges in the mesh. The notation, as given by Rossignac, for the half-edge data structure is

$$\{R, F, fe, E, V : R \to F \to L \to fe \overset{1}{\to} (V, E), E \overset{2}{\to} fe, V \overset{1}{\to} fe \overset{1}{\to} L \to F \overset{1}{\to} R\}.$$

Adding together the above relations, one hundred and forty-four pointers are required to represent a cube.

In practice, the half-edge data structure is sometimes implemented without the explicit storage of faces. This gives

$$\{R, fe, E, V : R \to L \to fe \xrightarrow{1} (V, E), E \xrightarrow{2} fe, V \xrightarrow{1} fe \xrightarrow{1} L \xrightarrow{1} R\}$$

as an alternate version of the half-edge data structure. With this version, the pointers to and from the faces are removed, so one hundred and thirty-six pointers are required to represent a cube.

Quad-Edge

Unlike the previous data structures, the quad-edge structure does not require faces as part of its definition; the representation relies entirely on how the edges and half-edges are connected. In Rossignac's notation, the quad-edge data structure is

$$\{R, E, fe, V : E \xrightarrow{2} fe, E \overset{4}{\Rightarrow} fe, E \xrightarrow{2} V, V \to E\}.$$

One hundred and twenty-eight pointers are required to represent a cube using the quad-edge data structure, a memory requirement similar to that of the half-edge data structure.

Corner-Table

The *corner-table* data structure [61] was developed for the storage and compression of triangle meshes stored as triangle strips. It is structured such that the vertices are stored in an array and each vertex has indices, stored in more arrays, that correspond to the the vertices that are near to it. In Rossignac's notation, the corner-table data

structure is

$$\{F, V : V \xrightarrow{5} V, V \xrightarrow{1} F\}.$$

The corner-table data structure is limited as it can only represent triangle meshes: the cube used here for comparison is not representable. Moreover, the vertices are stored in arrays and the indices into these arrays describe the relations between the vertices; therefore, the cost of modifying a mesh is high. If a new vertex is inserted or removed, all the arrays must be reallocated and the indices recomputed. The compression algorithms presented in [61] do not modify the meshes, but instead create new meshes as an output. The corner-table data structure is not appropriate for dealing with $(\mathrm{DS})^2$; it is intended for use with static structures.

However, the corner-table data structure is interesting as its definition includes an algebra for navigating a triangle mesh in a local manner. Given a triangle t and a vertex v, then in this algebra, the other two vertices of t are $c.n$ and $c.p$, such that, in counter-clockwise order, $c.p$, c & $c.n$ form t. Three other vertex relations are defined by the three adjacent triangles. The vertex $c.l$ forms a triangle with c and $c.n$, $c.r$ forms a triangle with c and $c.r$ and $c.o$ forms the triangle with $c.p$ and $c.n$. These expressions are evaluated by operations on indices or table look-ups. From a triangle and a vertex, these expressions allow the exploration of the local structure. These algebraic expressions are reminiscent of the next and previous operations of the vv algebra, but they cannot be easily chained together to make compound expressions as can be done in the vv algebra. The navigation that is possible with the corner-table algebra is a subset of what can be done with the vv algebra.

An interesting remark made in the conclusion of [61] is that the simplicity of the corner-table data structure makes it easy to implement and amenable to many

applications that require fast data access and small memory usage. Simplicity is advantageous.

vv

In its simplest form, vv is vertices that point to ordered, circular lists of vertices. In Rossinganc's notation, this is

$$\{V : V \Rightarrow V\},$$

which is a much simpler expression than those for the half-edge, winged-edge or quad-edge data structures.* With this simplicity, only a very small number of pointers are required: a cube can be represented in vv with just twenty-four pointers. Interestingly, this is as simple as the face-vertex data structure, but unlike the face-vertex data structure, it is possible to make any traversal over the mesh.

Of course, vv is rarely used in this simplest configuration, and with additional features, more pointers are added. However, the pointers are only added as the features are required. By only adding features as they are used, the data structure is kept as simple as it can be in each vv program.

When edges are added, the expression becomes

$$\{V, E : V \Rightarrow (V, E)\}.$$

The only change is that each vertex now points to an ordered circular list of pairs, instead of vertices. The complexity of the data structure has increased by the introduction of edges, forty-eight pointers are required to represent a cube.

*This expression inspired the name "vertex-vertex systems" as it is plainly visible in this notation that it is just vertices that relate to vertices in its simplest form.

Use of the flag feature requires one additional pointer per vertex. This results in the expression

$$\{V, E : V \Rightarrow (V, E), V \xrightarrow{1} V\}$$

and a total of fifty-six pointers to represent a cube.

If the vertices are synchronised, all the data is copied, doubling the storage requirements. This can be notated as

$$\{V, E : V \Rightarrow (V, E), V \Rightarrow (V, E), V \xrightarrow{1} V, V \xrightarrow{1} V\}$$

So, one hundred and twelve pointers are required to represent a cube when synchronised. So, even when all the extended features of vv is used, the memory requirements are still less than that of the half-edge, winged-edge and quad-edge data structures.

11.2 Unique Features of the VV Algebra

VV includes features that are not found in other polygon mesh data structures and algebras. These features can make modelling much easier. As was described in §3.1, a structure in vv need not satisfies the symmetry condition. For example, two vertices, a and b, can be related such that $a^\star = \{b\}$ and $b^\star = \{\}$. With the half-edge, winged-edge and quad-edge data structures, this asymmetric case would be an error. Another feature unique to vv is synchronisation, which provides copies the structure in such a way that is tied to the current structure.

Together, these two features allow for writing algorithms that transform a polygon mesh with a sort of "scaffolding" in the intermediary steps. Many of the algorithms described in this thesis use the synchronised state to traverse the stucture and collect information to build the new state and asymmetric conditions allow transitory states.

Representation	Pointers for a Cube	Possible Traversals
face-vertex	24	region to face face to vertex
winged-edge	192	face to edge edge to face edge to vertex vertex to edge
half-edge	144	face to half-edge half-edge to edge half-edge to vertex half-edge to face edge to half-edge vertex to half-edge
half-edge, no faces	136	"
quad-edge	128	edge to half-edge half-edge to edge edge to vertex vertex to edge
corner-table	–	vertex to vertex vertex to face
vv	24	vertex to vertex
add edges	48	vertex to vertex vertex to half-edge
add flag	56	"
synchronised	112	"

Table 11.1: A comparison of different polygon mesh data structures

Also, unlike the half-edge, winged-edge and quad-edge data structures, the neighbourhood is a representation that stores all the vertices surrounding a vertex in a single structure. Thus, the traversal of a neighbourhood using the **forall** construct is the traversal of the open disc around each vertex. With the half-edge, winged-edge and quad-edge data structures, it is necessary to traverse across several edge or half-edge structures to visit all the vertices in the open disc surrounding a vertex.

Chapter 12

Conversions from Other Paradigms

One measure of vv is to demonstrate that it can be used to model objects that have also been modelled with other paradigms. To this end, it is here demonstrated that vv can be used to model objects that can be modelled using L-systems and cell systems. This is done by showing how to construct vv structures and programs equivalent to structures and models of those other systems.

12.1 From L-systems

As demonstrated in Chapter 4, vv can be used to implement algorithms on linear structures in a similar spirit to an implementation using L-systems. However, vv is not limited to implementing a subset of algorithms that are possible using L-systems. The following section demonstrates how to transform an L-system with context and parameters into a vv program.

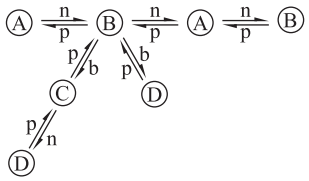
12.1.1 A VV Construction Corresponding to the L-string

The first step of the transformation is to define a vv construction that corresponds to the L-string of L-systems. The L-string is composed of modules, each identified by a symbol and optionally, some parameters.

Each module in the string can be represented by a vertex of the vv data structure. The vertex contains at least one property for the module's symbol and more prop-

erties to represent the module parameters as required. The order of the symbols in the L-string is represented in the vv data structure using neighbourhoods and asymmetric edge information. A module in the L-string can have modules preceding and following it, and also a branching symbol following it. This connectivity ordering to the neighbouring vertices is maintained as a property of the edge information. The half-edge to the previous vertex is marked with 'p', the next with 'n' and the leading module in a branch with 'b'. The order of branches in the L-string is maintained in a vertex neighbourhood as the counter-clockwise ordering. An example of a vv construction derived from an L-string is shown in Figure 12.1.

AB[CD][D]AB



(a) An example L-string (branches are indicated by square brackets)

(b) A vv construction equivalent to the example L-string

Figure 12.1: An L-string and its equivalent vv construction

It is also necessary to keep track of the root vertex of the tree structure; the root vertex is equivalent to the first symbol in the L-string. This is done quite simply by keeping a pointer to the root vertex in the vv program.

The resulting vv structure is a tree structure equivalent to an L-string. One consequence of using a tree-structure instead of a string is that it is not necessary to indicate the branch starts and terminals with extra symbols. To have a serialised tree stored in the L-string, special symbols must be inserted to indicate the start and end of branches, the [and] symbols in the cpfg language.

12.1.2 Tracing the L-string and Turtle Geometry

One of the most important features of L-systems in practical applications is that they are easily rendered graphically using turtle geometry [52]. The turtle begins at the first symbol in the L-string and is moved by the instructions encoded by symbols in the L-string, or in a homomorphism of the L-system, in the sequence that they appear in the string. These instructions are special symbols in the L-string alphabet. Because the branches are serialised in the string, the turtle instructions are in a depth-first sequence on the tree.

To implement a turtle geometry interpretation on the vv tree structure, it is then necessary to walk on the tree in a depth-first manner. A depth-first walk can be implemented by starting at the root vertex and proceeding first along the half-edges marked with 'b', in counter-clockwise order, and then to the half-edge marked with 'n'. To keep track of where to return to when walking down a branch, a stack holding the turtle state is kept. An algorithm that implements the depth-first walk is given in Algorithm 12.2. The algorithm begins at the root vertex r and each symbol is passed to an auxiliary function to handle the graphical interpretation. Then, the walk uses the function given in Algorithm 12.1 to start a traversal down each subtree.

Algorithm 12.1: Find the first vertex in a neighbourhood for a depth-first walk

```
1  vertex find_first(vertex v)  {
2      if (valence v == 0) return 0;
3      else if (valence v == 1)  {
4          vertex u = any in v;
5          if ((v^u).mark == 'n') return u;
6          else return 0;
7      }
8      else  {
9          forall u in v  {
10             if ((v^u).mark == 'p') return nextto u in v;
```

```
11          }
12      }
13  }
```

Algorithm 12.2: Depth-first walk on a tree

```
1   void walk(vertex v, turtle_stack& s, turtle& t)  {
2       interpret_symbol(v, s, t);
3       vertex u = find_first(v);
4       if (!u)  {
5           t = s.pop();
6           return;
7       }
8       while ((v^u).mark != 'p')  {
9           if ((v^u).mark == 'b') s.push(t);
10          walk(u, s, t);
11          u = nextto u in v;
12      }
13  }
```

The symbol interpretation function used here (Algorithm 12.3) includes interpretations for drawing a line forward by a distance d, $\mathsf{F}(d)$ and turns by an angle a in the counter-clockwise, $+(a)$, and clockwise, $-(a)$, directions. Although this is only a small set of possible turtle instructions, it is sufficient for implementing turtle geometry in a plane.

Algorithm 12.3: A turtle geometry interpretation function

```
1   void interpret_symbol(vertex v, turtle& t)  {
2       switch (v$symbol)  {
3           case 'F':
4           {
5               Pt a = t.pos;
6               Pt b = a + t.heading * v$value;
7               draw_line(a, b);
8               t.pos = b;
9           }
10          break;
11          case '+': t.heading = rotate(t.heading, v$value); break;
```

```
12            case '-': t.heading = rotate(t.heading, -v$value); break;
13        }
14    }
```

12.1.3 Productions

An L-system production can be implemented in vv simply as a replacement of one vertex with another or several vertices while maintaining the half-edges to the adjacent vertices. This can be viewed as a graph transformation replacing a node with a sub-graph. So, the successor of the L-system production is supplied as a pattern of vertices for the implementation in vv.

The pattern for the successor can be specified like any other input polygon mesh and stored as a vertex set or, as is used in the example in §12.1.4, a function that generates a set of vertices. The function for applying a production is given in Algorithm 12.4. This function first calls another function to generate the subgraph required for the successor for the production matching the vertex, this is an application-specific function (line 5). The subgraph G is generated and the starting and ending vertices, s and e, in this subgraph are also supplied. The vertex is replaced with the subgraph such that the preceding vertex in the tree is connected to s (lines 11 – 13) and the succeeding and branching vertices in the tree are connected to e (lines 16 – 22). If the vertex was the root vertex in the tree, the root is assigned to s (line 15). Finally, the neighbourhood of the replaced vertex is destroyed (line 23), thus removing its connections to the tree. In the case of an identity production, the function generate_sub_tree returns false and the algorithm returns early as no work needs to be done.

Algorithm 12.4: Application of a production

```
1  void apply_production(vertex v, vertex r, mesh & S) {
2      vertex s = 0;
3      vertex e = 0;
4      mesh G;
5      if (!generate_sub_tree(v, G, s, e)) return;
       // Find the vertex previous to v in the L-string
6      vertex p = 0;
7      forall x in v {
8          if ((v^x).mark == 'p') p = x;
9      }
       // If a previous vertex was found, connect it to the
       // generated sub-graph, otherwise, set the root to the
       // start of the sub-graph
10     if (p) {
11         replace v with s in p;
12         (p^s).mark = (('p)^v).mark;
13         (s^p).mark = 'p';
14     }
15     else r = s;
       // Connect the vertices following v in the L-string to
       // the generated sub-graph
16     vertex n = find_first(v);
17     if (n) vertex x = any in e;
18     while ((v^n).mark != 'p') {
19         splice n before x in e;
20         replace v with e in n;
21         (e^n).mark = (v^n).mark;
22     }
       // Remove all the remaining references to v
23     while (valence v) erase any in v from v;
24     remove v from S;
25     merge S with G;
26 }
```

The context in the predecessor of a production can be implemented by scanning the neighbourhood. To implement the context matching, the synchronised state is used, but to implement fast information transfer [25], where the context matches the current state of the L-string, the current neighbourhood can be used for matching.

Again, a depth-first walk is used to traverse the structure, but instead of a symbol interpretation function, there is the function that implements the productions.

12.1.4 A Koch Snowflake: A VV Program Derived from an L-systems

To demonstrate how the preceding algorithms fit together, a simple L-system is here transformed into a vv program. The L-system considered is the Koch Snowflake curve [84] and is specified as

$$\mathsf{F}(d) \rightarrow \mathsf{F}(d/3.0) + (60)\mathsf{F}(d/3.0) - (120)\mathsf{F}(d/3.0) + (60)\mathsf{F}(d/3.0)$$

in the cpfg syntax*.

To implement this production in vv, a generate_sub_tree function must be implemented such that it produces a chain of seven vertices: four 'F', two '+' and one − modules. This function is given in Algorithm 12.5. The result of the algorithm is shown in Figure 12.2.

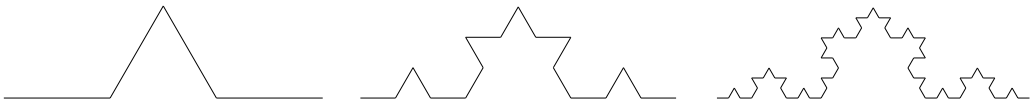


Figure 12.2: The Koch snowflake curve with two derivation steps

Algorithm 12.5: Generation of the sub-tree for the Koch snowflake

```
1  void generate_sub_tree(vertex v, mesh G, vertex s, vertex e)  {
2      if (v$symbol != 'F') return false;
-      // Create a vertex for every symbol in the antecedant of the production
3      vertex f1('F', v$value / 3.0); add f1 to G;
4      vertex a1('+', 60.0); add a1 to G;
5      vertex f2('F', v$value / 3.0); add f2 to G;
6      vertex a2('-', 120.0); add a2 to G;
7      vertex f3('F', v$value / 3.0); add f3 to G;
```

*This L-system is also shown in [55, Chapter 1].

```
 8     vertex a3('+', 60.0); add a3 to G;
 9     vertex f4('F', v$value / 3.0); add f4 to G;
 –     // Set the start and end vertices of the graph
10     s = f1;
11     e = f4;
 –     // Assign all the neighbourhoods of the graph
12     make { a1 } nb_of f1;
13     make { f1, f2 } nb_of a1;
14     make { a1, a2 } nb_of f2;
15     make { f2, f3 } nb_of a2;
16     make { a2, a3 } nb_of f3;
17     make { f3, f4 } nb_of a3;
18     make { a3 } nb_of f4;
 –     // Assign all the marks to the edges to establish the
 –     // order of the symbols in the L-string
19     (f1^a1).mark = 'n';
20     (a1^f1).mark = 'p'; (a1^f2).mark = 'n';
21     (f2^a1).mark = 'p'; (f2^a2).mark = 'n';
22     (a2^f2).mark = 'p'; (a2^f3).mark = 'n';
23     (f3^a2).mark = 'p'; (f3^a3).mark = 'n';
24     (a3^f3).mark = 'p'; (a3^f4).mark = 'n';
25     (f4^a3).mark = 'p';
26     return true;
27 }
```

Though Algorithm 12.5 is somewhat lengthy, it can be seen that its structure is reasonably simple and is proportional to that of the L-system production that it implements. For each module in the production there are a set number of statements in the function and only two additional statements (lines 9 and 10) to mark the start and end of the generated sub-tree. This is a function that can be generated mechanically from a given L-system production.

12.1.5 Remarks on Implementing L-systems Using VV

All the algorithms in this section implement the L-system machinery to which the user of L-systems is not normally exposed. Only in parts of Algorithm 12.5 is there something comparable to the L-system production that comprises an L-system pro-

gram. Other productions can be implemented as reimplementations of the generate_sub_tree function. Essentially, Algorithms 12.1 – 12.4 implement an engine for L-systems, comparable to other L-system engines such as cpfg or lpfg. Moreover, because different generate_sub_tree functions from given L-system productions can be implemented mechanically, it would be possible to add a preprocessing stage could be added to create a program that generates a vv program from existing L-systems.

However, this example does not just simply reimplement an L-system engine. Unlike regular L-systems, the underlying structure is not a tree serialised as a string, it is a graph structured as a tree. This underlying structure is exposed in the vv program; it is normally hidden in an L-system program.

12.2 From Map L-systems and Cell Systems

In *map L-systems with geometry* [33, 55] and *cell systems* [10], productions change a polygon mesh locally. This is also the strength of vv. VV can be related to map L-systems and cell systems by considering how the elements of these formalisms can be represented by graphs and transformations to graph rotations systems. Therefore, to relate map L-systems and cell systems to vv, it is necessary to express the polygon mesh in the vv data structure and the productions as algorithms in the vv language. Here, it is demonstrated how to express the mesh and productions of a cell system in vv.

A cell is again structured as a polygonal cell wall with an additional vertex at the cell's centre, similar to that described in §10; additionally, each cell now has a reference vector. The cell division algorithm is also similar to that used in §10, except

that here, the line that divides the cell is perpendicular to the reference vector. To match the physical model used in cell systems, the shape of the cells is governed by a mass-spring system, similar to that described in §7.2, and a pressure system, such that each cell has an internal pressure that forces the segments of its wall outwards. Unlike that used for the Korn-Spalding cell division pattern in §7.3, this pressure system is not simplified and uses the cell's area to properly calculate the force exerted by the pressure.

There are three forms of productions in cell systems: do nothing to a cell, replace a cell, and divide a cell. The first type of production (do nothing) is the identity production, so no work needs to be done. It is trivially handled by ignoring cells of those types when applying productions. For the second type of production (replacement), the type property of the cell is simply updated to reflect the new type.

The third type of production (cell division) changes to the structure topologically. These productions take the form of

$$A \rightarrow B \uparrow (\alpha, \gamma) C,$$

where A is a mother cell, B and C are the daughter cells to the left and right of the reference vector respectively, α is the rotation of the reference vector and γ is the proportion of the area of A allocated to B.

Like the case of moving from L-systems to vv (§12.1), it is necessary to implement all the machinery found in cell systems as a vv program. Then, some additional code that implements the productions specific to each cell system is added. In this example, a variation of the Korn-Spalding cell division pattern [28] found in [10] is modelled. As a cell system, the Korn-Spalding cell division pattern requires a single

production,

$$C \rightarrow C \uparrow (1.4118, 0.5)C.$$

That is, a cell is replaced by two cells of the same type with the reference vector rotated by 1.4118 radians. The vv program that implements the general cell system machinery and the above production is given in Algorithm 12.6.

Algorithm 12.6: The Korn-Spalding cell system as a vv program

```
1   void cell_system_engine() {
    // Reset the accelerations of the vertices in the cell walls
2   forall j in J {
3       j$accel.zero();
4   }
5   synchronise J;
    // Calculate the pressure in each cell
6   forall c in C {
        // Get the area of the cell
7       float cell_area = area(c);
        // Apply the force due to pressure on each of the cell's walls
8       forall j in c {
9           Pt a = (nextto j in c)$pos;
10          Pt b = (prevto j in c)$pos;
11          Pt aj = j$pos - a;
12          Pt jb = b - j$pos;
13          Pt paj(-aj.y(), aj.x(), 0.0);
14          Pt pjb(-jb.y(), jb.x(), 0.0);
15          j$accel += (1.0f / cell_area) * (paj + pjb);
16      }
17  }
18  forall j in J {
        // Calculate the force from the compression of stretch of the
        // cell walls
19      forall v in j {
20          if (v$type != 'j') continue;
21          Pt l = j$pos - `v$pos;
22          Pt l_rest = l;
23          l_rest.normalise();
24          l_rest *= 2.0;
25          j$accel += 0.5f * (l_rest - l);
26      }
        // Calculate the new position of the cell walls
27      j$vel += j$accel * 0.05f;
```

```
28          j$vel -= 0.2f * ‘j$vel;
29          j$pos += j$vel * 0.05f;
30          if (settled && j$accel.length() > 0.01f) settled = false;
31      }
        // Find the barycentre of each cell and place the centre cell there
32      forall c in C  {
33          c$pos.zero();
34          forall j in c  {
35              c$pos += j$pos;
36          }
37          c$pos /= float(valence c);
38      }
        // If forces in the system are not settled, return so that another
        // iteration of the physics can be performed
39      if (!settled) return;
        // Find the reference vector for each cell
40      forall c in C  {
41          Pt e = (flagged in c)$pos - (prevto flag in c)$pos;
42          e.normalise();
43          c$reference.set(-e.y(), e.x(), 0.0);
44      }
45      mesh NJ;
46      mesh NC;
47      forall c in C  {
48          vertex j0 = 0;
49          vertex j1 = 0;
50          vertex j2 = 0;
51          vertex j3 = 0;
52          Pt pos01;
53          Pt pos23;
            // Rotate the reference vector by 1.4118 radians
54          Pt w = c$reference;
55          {
56              float r = w.length();
57              float a = atan2(w.y(), w.x());
58              a += 1.4118;
59              w.set(r * cos(a), r * sin(a), 0.0);
60          }
            // Find the intersections of the cell wall and the line
            // of fissure, the line perpendicular to the new reference vector
61          Pt l(-w.y(), w.x(), 0.0);
62          l += c$pos;
63          bool firstpair = true;
            // The intersection case for a vertical fissure line (x == 0)
64          if (fabs(l.x() - c$pos.x()) < 0.001)  {
65              forall j in c  {
66                  vertex k = nextto j in c;
67                  if ((l.x() >= j$pos.x() && l.x() < k$pos.x())
```

```
68                  || (l.x() <= j$pos.x() && l.x() > k$pos.x()))  {
69                  float jk_slope = (j$pos.y() - k$pos.y()) / (j$pos.x() - k$pos.x());
70                  float jk_constant = j$pos.y() - jk_slope * j$pos.x();
71                  float y = jk_slope * l.x() + jk_constant;
72                  if (firstpair)  {
73                      j0 = j;
74                      j1 = k;
75                      pos01.set(l.x(), y, 0.0);
76                      firstpair = false;
77                  }
78                  else  {
79                      j2 = j;
80                      j3 = k;
81                      pos23.set(l.x(), y, 0.0);
82                  }
83              }
84          }
85      }
-       // Intersection tests for regular fissure lines
86      else  {
87          float lc_slope = (l.y() - c$pos.y()) / (l.x() - c$pos.x());
88          float lc_constant = l.y() - lc_slope * l.x();
89          forall j in c  {
90              vertex k = nextto j in c;
-               // The case that the cell wall segment is vertical (x == 0)
91              if (fabs(j$pos.x() - k$pos.x()) < 0.001)  {
92                  if ((l.y() >= j$pos.y() && l.y() < k$pos.y())
93                      || (l.y() <= j$pos.y() && l.y() > k$pos.y()))  {
94                      if (firstpair)  {
95                          j0 = j;
96                          j1 = k;
97                          pos01.set(j$pos.x(), l.y(), 0.0);
98                          firstpair = false;
99                      }
100                     else  {
101                         j2 = j;
102                         j3 = k;
103                         pos23.set(j$pos.x(), l.y(), 0.0);
104                     }
105                 }
106             }
-               // The case that the cell wall segment is not vertical
107             else  {
108                 float jk_slope = (j$pos.y() - k$pos.y()) / (j$pos.x() - k$pos.x());
109                 float jk_constant = j$pos.y() - jk_slope * j$pos.x();
110                 float x = (jk_constant - lc_constant) / (lc_slope - jk_slope);
111                 float y = (lc_constant * jk_slope - jk_constant * lc_slope)
112                     / (jk_slope - lc_slope);
```

```
113                 if ((fabs(jk_slope - lc_slope) > 0.001) && ((x >= j$pos.x()
114                     && x < k$pos.x())
115                     || (x <= j$pos.x() && x > k$pos.x()))) {
116                     if (firstpair) {
117                         j0 = j;
118                         j1 = k;
119                         pos01.set(x, y, 0.0);
120                         firstpair = false;
121                     }
122                     else {
123                         j2 = j;
124                         j3 = k;
125                         pos23.set(x, y, 0.0);
126                     }
127                 }
128             }
129         }
130     }
131     {
132         Pt mid12 = 0.5f * (j1$pos + j2$pos);
133         Pt mid03 = 0.5f * (j0$pos + j3$pos);
134         if (mid12.distance(w + c$pos) < mid03.distance(w + c$pos)) {
135             vertex t = 0;
136             t = j0; j0 = j2; j2 = t;
137             t = j1; j1 = j3; j3 = t;
138             Pt pt = pos23;
139             pos23 = pos01;
140             pos01 = pt;
141         }
142     }
143     if (pos01.distance(j0$pos) < 0.01) pos01 *= 1.01;
144     if (pos23.distance(j0$pos) < 0.01) pos23 *= 1.01;
      – // Insert two new vertices into the cell wall
145     vertex j01 = insert(j0, j1); j01$type = 'j'; j01$pos = pos01;
146     vertex j23 = insert(j2, j3); j23$type = 'j'; j23$pos = pos23;
147     add j01 to NJ;
148     add j23 to NJ;
149     char symleft = 'c', symright = 'c';
150     Pt ref = c$reference;
      – // Create two new cell centres
151     vertex cleft(symleft, 'X', Pt(), ref, Pt(), Pt());
152     vertex cright(symright, 'X', Pt(), ref, Pt(), Pt());
      – // Attach the new cell centres to the cell wall
153     splice cleft after j1 in j01;
154     splice j23 after cleft in j01;
155     splice cright before j0 in j01;
156     splice cleft before j2 in j23;
157     splice j01 before cleft in j23;
```

```
158         splice cright after j3 in j23;
159         add cleft to NC;
160         add cright to NC;
161         make { j01, j2, j23 } nb_of cleft;
162         replace c with cleft in j2;
163         vertex x = j1;
–           // Walk around the cell wall to complete the neighbourhoods
–           // of the new cell centres
164         while (x != j2) {
165             splice x before j2 in cleft;
166             replace cw with cleft in x;
167             x = nextto x in c;
168         }
169         make { j23, j0, j01 } nb_of cright;
170         replace c with cright in j0;
171         x = j3;
172         while (x != j0) {
173             splice x before j0 in cright;
174             replace c with cright in x;
175             x = nextto x in c;
176         }
177         flag j01 in cleft;
178         flag j23 in cright;
–           // If the new vertices on the cell wall are not on the edge,
–           // connect them to the cell centres of the adjacent cells
179         vertex o01 = nextto j01 in j1;
180         if (o01$type != 'j') {
181             splice j01 after j1 in o01;
182             splice o01 before j1 in j01;
183         }
184         vertex o23 = nextto j23 in j3;
185         if (o23$type != 'j') {
186             splice j23 after j3 in o23;
187             splice o23 before j3d in j23;
188         }
189     }
190     merge J with NJ;
191     C = NC;
–       // Calulcate the new barycentres of the cells
192     forall c in C {
193         c$pos.zero();
194         forall j in c {
195             c$pos += j$pos;
196         }
197         c$pos /= float(valence c);
198     }
199 }
```

The logic of Algorithm 12.6 can be broken into three sections: the physics engine, the cell division engine and the production logic. The physics (lines 2 – 31) has two steps; the pressure exerted by each cell on its cell walls is calculated (lines 6 – 17) and a mass-spring system is used (lines 19 – 26), much like in the initial Korn-Spalding implementation in §7.3.

Lines 66 and 157 encode the logic of the production. In line 66, the reference vector is rotated by 1.4118 radians. In line 157, the symbols are assigned to the daughter cells in the production. In this example, there is only one production; but, were there more productions, those lines could be inside conditional statements based on the parent cell type. For example, for a different set of example productions,

$$\begin{aligned} A &\rightarrow A \uparrow (1.41, 0.5)B \\ B &\rightarrow B \uparrow (-1.41, 0.5)A, \end{aligned}$$

line 66 would become

```
switch(c$type) {
    case 'A': a += 1.41; break;
    case 'B': a += -1.41; break;
}
```

and line 157 would become

```
switch(c$type) {
    case 'A': symleft = 'A'; symright = 'B'; break;
    case 'B': symleft = 'B'; symright = 'A'; break;
}
```

to implement those productions. As can be seen, the structure of those code fragments follows the structure of the cell systems production quite closely and like the L-system productions in §12.1.3, the vv program code that implements cell system

production could be produced mechanically.

The remainder of Algorithm 12.6 implements the cell division and maintenance of cell structures. In lines 62 – 68, the reference vector is rotated. Then, using the reference vector, the intersections of the line parallel to the reference vector and the cell wall are found (69 – 152). Following this, the cell division algorithm follows the pattern seen in Algorithm 10.1. Two new vertices are added to the cell wall (lines 153 & 154), two new cell centres are created (lines 157 – 160), connected to the new vertices on the wall (lines 161 – 166) and connected to the existing cell wall (lines 169 – 197).

The results of Algorithm 12.6 are shown in Figure 12.3. This result can be compared to those shown in Figure 7.2. The difference in the results is due to differences in how the lines of fissure are found. In the first presentation of the Korn-Spalding model (§7.3) the division always divides an opposite pair of walls at one-third along its length and the pair of divided walls is always rotated. Thus, no matter what the parameters of the physical simulation, the divisions will always have the same topological result. In contrast, here the line of division is always perpendicular to the reference vector and the direction of the reference vector is greatly influenced by the physical simulation; which can result in different topological results. Here, physical parameters were chosen to produce a result that is noticeably different than that in Figure 7.2.

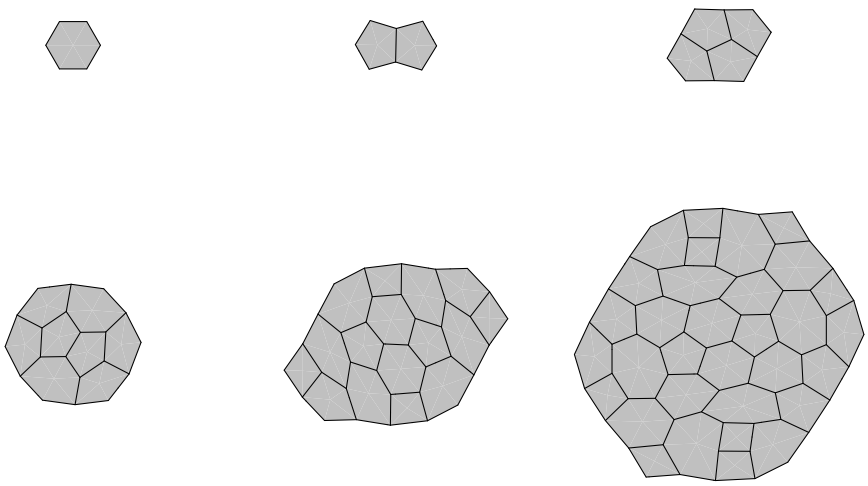


Figure 12.3: The development of the Korn-Spalding cell system

Chapter 13

Limitations of VV

Although vv has many advantages as a modelling system, it is obvious that it is not a catch-all solution for all modelling situations. There are limitations to vv and it is beneficial to a user of vv to appreciate these limitations. When a system can be modelled inside this set of limitations, then vv is likely a suitable choice for its implementation.

13.1 Limitations on the Topological Domain

VV can only be used for modelling structures with a discrete 2-manifold topology (*e.g.* graphs and polygon meshes). However, a question users of software often whether vv can be extended to handle a wider class of topologies. As the topologies discussed in the following sections are not discrete 2-manifolds, methods for modelling structures of these topologies are not considered in the scope of this research.

13.1.1 Modelling Volumetric Structures

The first topological domain that is asked about is usually volumetric structures, such as tetrahedral or cubic vertex sets or voxel structures. VV requires that the neighbourhood can be defined in a rotational order; but, in a volumetric structure, a vertex would have a neighbourhood for which there is no obvious rotational order. In general, there is not an obvious local ordering around a vertex in the non-planar

case. Some specific cases that can be handled easily. For example, a voxel structure always has six neighbours in the up, down and four adjacent positions. Only when there is always a fixed numbers of neighbours in a fixed orientation can an obvious, simple neighbourhood ordering be used.

Some data structures have been proposed for arbitrary discrete volumetric structures. Two examples are G-maps [31] and the group-based fields used in MGS [18, 19]. In the case of G-maps, the connectivity of each dimension of the structure is maintained with arrays of arrays, each array containing arrays representing the next lower dimension down to a set of labels for dimension zero. At each modification of the connectivity, each level of arrays must be updated. For objects with more than one or two dimensions, this becomes quite cumbersome. Moreover, G-maps are just a data structure, lacking operations to describe transformations to the structure. Group-based fields in MGS can be used to describe any discrete structure of arbitrary dimension, provided that the user can come up with a group that describes the connectivity. It may be technically possible to always find a group that describes the connectivity of a structure; however, doing so in practice is often quite difficult. It should be noted that the examples provided in [18, 19] mostly deal with planar cases.

In the end, there is no obvious and simple way to handle discrete volumetric structures with arbitrary connectivity using a vv-like approach has yet been discovered.

13.1.2 Multiresolution, Hierarchical, Ramified & Layered Structures

In the definition of a vertex structure (see §3.1), a vertex has exactly one neighbourhood. What could be done if a vertex had more than one neighbourhood. Vertices that each have several neighbourhoods would allow for several interesting modelling strategies and new structures that are currently not possible with vv.

One possibility is the use multi-resolution or hierarchical structures, describing a structure with more than one connectivity. In practice, this would be a mechanism for describing different aspects of a model. For example, a cell-tissue model there could include two sorts of relations: the connections of the cell walls and the connections of neighbouring cells. Both sorts of connections could be modelled by a set of neighbourhoods that describe the cell walls and a set that describes the cell neighbours.

Another possibility is that of modelling a limited set of non-2-manifold topologies. Two examples of possible topological classes that could be considered are ramified structures and layered surfaces.

At this point, no satisfactory definition and operations have been found to describe a vertex with several neighbourhoods and the research necessary to find them requires exploration of a large set of models and so falls outside the scope of this work. Moreover, the technical limitations imposed by the implementation of version 1.1 of the vv software environment make experimentation with extensions to the graph rotation system prohibitively difficult.

13.2 Limitations of the VV Software Environment

Many of the limitations encountered in creating models with vv are technical limitations of the vv software. In each case, the problem is the result of a design decision. VV is a new technology and the design is based on the relatively new LPFG language, so some problems with the first version of the software expected.

All of the following refer to version 1.1 of the software, they do not apply to version 2.0. The solutions that are used in version 2.0 to address these problems are discussed at the end of this section.

13.2.1 The Execution Model

To have a standard viewing application for vv models, the vvinterpreter program requires a standard sequence of events in a vv model. Each model can implement a fixed set of functions that are called by the viewing application and this limits how the user can interact with a model.

This restriction leads to the request of many users to have more functions in the viewer to interact with their models or to have alternate sequences in the sequence of events. The requested methods of interactions vary considerably between users. Version 1.1 included no method for users to introduce new methods of interaction or new event sequences.

13.2.2 Lengthy Compilation Times

To provide vertex and edge types with an arbitrary set of user-defined properties, the library code uses many templates that can be instantiated with the property

types. The translator program vvp2cpp generates C++ code that is the vertex class with the parameter types filled in. For flexibility, many of the classes in the utility portion of the library also use templates.

Templates allow library code to be written such that definitions created by the compiler from the templates are type-safe and fast at run-time, as no dynamic casting is required.

Unfortunately, most compilers instantiate classes with templates slowly and so the compile time can be lengthy. The effect to the user is that it may take ten seconds or more to compile a vv program when other programs of similar complexity can be compiled in a few seconds. In a rapid modelling situation, where the user is making many small iterative changes to a program and testing often, the compile time should be as short as possible.

13.2.3 The VV Language

The largest frustrations that users find with the vv language come from limitations in the translator program vvp2cpp. The translator program serves two rôles: generate the vertex, edge and vertex set definitions in C++ from the user's definitions and translate the vv language statements into C++ statements.

During the creation of the definitions, no forward declarations are supplied, therefore the user's declaration of vertex, edge and vertex set properties cannot in any way refer to vertices, edges or vertex sets. This possibility was not anticipated when vv was designed and for that reason it is not supported. Aside from that, the generation of vertex, edge and vertex set definitions works quite well.

The larger problems stem from the translation of vv statements into C++ state-

ments. The root of the problems is that the translator must correctly identify all existing C++ statements and vv statements in a vv program and separate and treat them accordingly. However, the C++ and vv portions can be mixed quite freely and C++ is already an extremely difficult language to parse. Currently, all the known bugs in vv stem from the parsing problem.

This difficulty has resulted in a translator program that only works most of the time, but can unexpectantly fail under cryptic circumstances. For example, the statement

nextto _nextto(a) **in** f(a, b, **prevto** b **in** c) **in** a

should be a valid statement in the vv language; but, the translator could fail at two points in that statement. Firstly, there is a recognition problem in separating the vv keyword **nextto** and the C++ function _nextto. Secondly, the second argument of the **nextto** expression is a C++ expression that in turn contains a vv expression. Having an arbitrary string inside the expression that must be parsed that recursively contains other arbitrary statements that require parsing recursively to any arbitrary depth is a difficult problem.

Another problem that can occur in the translation of a vv program is correctly placing parentheses around expressions that have prefixed or postfixed operators. Particularly in vv, the prefix operators are ‘, **any** and **valence** and the postfix operator $ must be properly considered. These work fine as long as the expression to which they are attached is already in parenthesis or is just a simple identifier (*i.e.* a variable name). To work correctly, the translator must correctly identify any possible statement and properly put parenthesis around it to prepend or append the operator.

For example, seemingly simple expressions such as

‘q[8]

and

f(a)$x

cause translation to fail.

The translator program, like many translation and parsing programs is implemented using *LALR (Look Ahead Left-Right)* parsing (see [30] for an overview of LALR parsing). Specifically it uses the programs flex and bison to generate a parser. The limitations of LALR parsing cause some of the difficulties in the translator design. For example, there is no easy way to express the C++ grammar, because it requires knowledge of the types used, not just the tokens that are found while parsing. To correctly handle C++, a parser is not sufficient. In LALR parsing, scanning can only occur from left to right. This is fine for the vast majority of situations, but it means that it is impossible to correctly implement a postfix operation, which requires that the operator be found, followed by a scan from right to left to find the expression that to which it is attached. Therefore, a LALR parser, cannot be used to correctly implement the $ operator.

Unfortunately, it the available parser generators, including popular choices such as lex and yacc, PCCTS and ANTLR are all variants of LALR parsing. No widely available choice for a parser generator is suitable for the vv language.

13.2.4 VV 2.0

The listed difficulties with version 1.1 of the software have been addressed in the design of vv 2.0; however, some of the changes in the design have resulted in a software suite that is in some respects quite different. The result is a software package that is used as a library with a few extra tools, rather than a specialized language.

The first major difference is that the vv language has been largely removed. This was the largest source of problems in vv 1.1, so its removal has resulted in a code base that is much easier to debug and maintain. Therefore, instead of a vv language, the vv data structure and algebra are available as a C++ library. For example, instead of the vv language statement

nextto u **in** v

there is the C++ statement

$$v\text{->next}(u).$$

A few language constructs remain that are not native to C++. The first is the declaration of the vertex and edge properties, which is largely unchanged from vv 1.1 and there is still a tool to generate C++ definitions for the vertices and edges. These definitions are easily separated from C++ statements, so writing a parser for this is a simple task and beneficial as it saves the user considerable work. Also, the ‘ and $ operators are kept, as they are easy to recognise and convenient to have. However, instead of implementing the recognition of these operators with an LALR parser, a bi-directional scanner was written to handle only these two cases. With this bi-directional parser, translation of these two operators into C++ is relatively

simple. Also retained from the vv language is the **forall** statement for iteration over a vertex neighbourhood or a vertex set. Since **forall** statements are mixed with C++ in a very limited way, a LALR parser has been written to deal specifically with those statements.

Without the dependency of the vv language, vv can be used strictly as a C++ library. This allows users to create any sort of execution model that they need. A standard execution model is still available, but this is now supplied as a separate C++ library that can be extended as the user needs. The responsibilities of the viewing program have been delegated a simulator library that contains classes that the user can inherit from to add functionality. This flexibility eliminates the constraints of the fixed execution model of vv 1.1.

Finally, the templates used in vv 1.1 have been replaced with inheritance. This reduces performance at run-time slightly, as now vertices must be dynamically cast at run-time, but the compile times are shortened drastically. This satisfies the need for rapid prototyping of models using vv.

Chapter 14

Final Remarks on VV

The examples in this thesis illustrate that concept of using a graph rotation system with an algebra, as implemented with vv is well-suited for implementing $(DS)^2$ models of discrete 2-manifold topologies. Characterizing discrete structures in local terms (*i.e.* using a graph rotation system to represent vertices and their neighbourhoods) and applying transformations to the structure as algorithms that edit and traverse the local relations is a sound methodology. The next and previous operations and the iteration with the **forall** construction provide easy navigation across vertices and their neighbourhoods. The neighbourhood editing operations (neighbourhood assignment, splice, replace and erase operations) allow easy modifications to the connectivity of the vertices.

With these mechanisms, $(DS)^2$ with a discrete 2-manifold topology can be elegantly modelled using a local approach; there is no need to impose a global indexing schema on the components. Thus, the problems associated with indexing dynamical structures, as discussed in §1.2, are avoided.

Unlike map L-systems and cell systems, vv is not declarative. Programs written in the vv language are imperative. This allows the modeller to explore and transform the structure directly. Where the productions of declarative systems replace whole sections of a structure in one step, the editing operations of the vv algebra effect atomic changes to the structure. Therefore, the modeller can create smaller, intermediary steps in transformation algorithms, which can often be more intuitive

then trying to determine the correct expressions to effect large transformations.

The synchronisation feature of the vv algebra has also proven to be a useful feature that is not found in other modelling systems. The synchronised structure can be used as a static image for navigating the structure while effecting changes to it. The modeller has direct access to both a changing and an unchanging version of the structure simultaneously. Synchronisation together with editing operations can be thought of as analogous to using a scaffolding while making small constructions, additions and repairs in building construction. While synchronisation is similar to the accessibility of the old states in declarative systems, it is different in that it can be controlled explicitly by the modeller. In declarative systems, the state is always copied in its entirety at every derivation step, even when the copied state is not used. This imposes a penalty in both computation time and storage in memory. In vv, synchronisation is only used when the modeller wants it and only on the vertices requested. Thus, in vv, there is no penalty for copying when synchronisation is not used.

The wide variety of examples in this thesis demonstrate that, unlike other polygon mesh or surface description paradigms, vv is well-suited for a very wide range of modelling problems. In Chapters 4 & 5, it was seen how the easy insertion of vertices and neighbourhood traversals can be used to implement subdivision algorithms with concise vv programs. In Chapter 6, it was seen how additional information can be added to the structure, such as alternate geometric interpretations for vertices as in the example of the Sierpinski gasket (§6.1) or the edge information in the Penrose tiling (§6.2) and the fractal terrain (§6.3.1). Through the examples in Chapter 7, it was demonstrated how to use simple physics systems to achieve emergent forms

by simulation. Different patterns of growth by the addition of vertices to polygon meshes were demonstrated in §7.4 and Chapters 8 & 9 and growth by dividing cells was seen in §7.3, §10 & §12.2. Finally, in Chapter 12, it was seen that it is possible to reimplement other modelling systems in vv, demonstrating that the modelling domain of vv encompasses that of L-systems and cell systems.

Experience from developing a wide variety of models using vvhas lead to the inclusion of new features (§3.5) to support development that were not included in vv as it was initially published in [69]. Of particular importance was the introduction of edge information (§3.5.4). This extension was used in a large number of the examples in this thesis as it aides in the expression of many algorithms. The other extensions, though less used, also aide the expression of algorithms as vv programs. It is important to note that these extensions do not have a negative impact when they are not used. For example, if edges are not used, there is no memory allocation for edges. Thus, the extensions respond to the needs of modellers without an impact on the cases when they are not used.

This same experience has also shown that vv has limitations (discussed in Chapter 13), but many of those that are technical in nature have been solved by further software development. The development from version 1.1 to 2.0 of the vv software environment is a large step in solving many of the limitations of using vv that have been found while using vv in the development of models. Future releases of the vv software environment will continue to improve on the tools based on the experience gained from modelling.

Future research into vv should follow two lines. Firstly, there is the ongoing question of what conceptual features and extensions, beyond the technical issues already

discussed, can be added modelling discrete 2-manifold surfaces. For example, there is the possibility that richer typing mechanisms for vertices or the use of functional or declarative paradigms to effect transformations to the graph rotation systems may make modelling easier. Secondly, there is the open question of how to extend the concepts used in vv to other topologies (as discussed in §13.1).

Part V

Appendices

Appendix A

The VV Language Specification

The following is a formal description of the operations and grammar of the vv language (version 1.1). Any of the statements described can appear in a vv program and are translated into C++ statements by the program vvp2cpp. The information here can serve as both a reference to the vv language and as a specification for its implementation.

A.1 Properties

Each vv program requires definitions for the edge, vertex and mesh properties used in the program. Any C++ type with implemented streaming and assignment operators can be included as property. Edge properties also require that an inequality operator by implemented. Properties are read or written to files using the streaming operators implemented for it. If a type that does not have implemented streaming operators is used, compile errors will occur. Note that all the plain data types in C++ satisfy these requirements and complex types can have these operators supplied to them. The names of the data properties are matched to property fields in the data file automatically.

It is also possibly to include functions in the property declarations. Functions are ignored for the purposes of reading and writing to files, but they can be used in the vv program like member functions.

The syntax for property definitions is

```
edge {
    property_type property_name;
};
vertex {
    property_type property_name;
};
mesh {
    property_type property_name;
};
```

where there may be an arbitrary number of properties in each declaration. The declarations must appear in the above order. Edge, vertex and mesh definitions are required in a vv program prior to any other use of the edge, vertex or mesh keywords.

Any permissible C++ identifier can be used as a property name except for *nb*, which is reserved as the property name for the neighbourhood data in the data file.

To use the rendering algorithms, a function void glRender() must be defined in the vertex definition. If the rendering algorithms are not used, this function does not need to be present. The syntax to get a property from an edge e is

e.*property_name*,

from a property from vertex v is

v$*property_name*

and for a property from the vertex set m, the syntax is

m.*property_name*.

A.2 Execution Blocks

The interpreter program executes a vv program according to defined execution blocks. A vv program may include at most one instance of each block, though none of them are mandatory.

The syntax for each block is the block name followed by a pair of curly braces. The curly braces may contain any amount of code to be executed in that block. The block names cannot be preceded with qualifiers. The blocks and how they are used are specified in Table A.1.

Block name	Use
start	Executed when the vv program is loaded.
step	Executed as the program's iterative step.
end	Executed as an auxiliary program path for arbitrary use.
close	Executed at the program terminal.
render_init	Executed when the rendering context is created.
render	Executed when the rendering context is redrawn. Rendering is done in xyz-coordinates.
render_screen	Executed immediately following the render block. Rendering is done in the xy-coordinates of the screen space.

Table A.1: The execution blocks and their uses

A.3 Keywords and expressions

The following terms are used:

- *ident* is any valid C++ identifier.
- *mexpr* is any vv expression designated as a mesh expression.
- *vexpr* is any vv expression designated as a vertex expression.

If more than one of the above terms is used in one of the following statements, they are suffixed with numbers to distinguish them.

A.3.1 Edge objects

Edges are structures that contain the data of how one vertex relates to a neighbouring vertex. Every pair of neighbouring vertices has two edge structure. For a pair of vertices a and b, there is an edge structure that contains the information of how a relates to b and a second edge structure for how b relates to a.

Edge structures may be accessed symmetrically or asymmetrically. If an edge is accessed asymmetrically, only one edge structure in the vertex pair is updated. When accessed symmetrically, the second vertex structure is overwritten by the information contained in the first.

(*vexpr1*|*vexpr2*)

Symmetric access to the edge between vertices *vexpr1* and *vexpr2*.

(*vexpr1*^*vexpr2*)

Assymmetric access to to the edge from *vexpr1* to *vexpr2*.

edge *ident*;

Declare an edge structure.

A.3.2 Vertex objects

Vertices are dynamically allocated and referenced counted automatically. The program does not need to take care of vertex allocation and deallocation. A vertex

object in the program is actually a pointer to this allocated vertex. It is illegal to create a vertex in the global scope. A vertex may have a null value.

vertex *ident*;

Allocates a new vertex referred to by the pointer *ident*.

vertex *ident* = *vexpr*;

Creates a new reference to the vertex pointed to by the expression *vexpr*.

vertex *ident* = 0;

Creates a null vertex.

A.3.3 Mesh objects

The mesh objects in vv programs are regular objects, following the existing C++ rules; mesh can be used just as any other C++ class name.

A.3.4 Vertex expressions

Any of the following expressions can be used in place of *vexpr*.

ident

Any C++ identifier referring to a vertex variable may be vertex expression.

‘*vexpr*

Returns a vertex representing the old state of the vertex referred to by *vexpr*. If *vexpr* is a null vertex then an error occurs.

any in *vexpr*

Returns a vertex from the neighbourhood of the vertex pointed to by *vexpr*. If the neighbourhood is empty a null vertex is returned.

nextto *vexpr1* in *vexpr2*

Returns the vertex following *vexpr1* in *vexpr2*. If *vexpr1* is not in the neighbourhood of *vexpr2*, then the expression returns a null vertex. If *vexpr2* refers to a null vertex then an error occurs.

prevto *vexpr1* in *vexpr2*

Returns the vertex preceding *vexpr1* in *vexpr2*. If *vexpr1* is not in the neighbourhood of *vexpr2*, then the expression returns a null vertex. If *vexpr2* refers to a null vertex then an error occurs.

next(*i*)to *vexpr1* in *vexpr2*

Returns the *ith* vertex following *vexpr1* in *vexpr2*. If *vexpr1* is not in the neighbourhood of *vexpr2*, then the expression returns a null vertex. If *vexpr2* refers to a null vertex then an error occurs.

prev(*i*)to *vexpr1* in *vexpr2*

Returns the *ith* vertex preceding *vexpr1* in *vexpr2*. If *vexpr1* is not in the neighbourhood of *vexpr2*, then the expression returns a null vertex. If *vexpr2* refers to a null vertex then an error occurs.

@(*vexpr*, *vexpr*) *pOp1*, *pOp2...*, *pOpN*@

Executes the path specified by *pOp1*, *pOp2...*, *pOpN*. Each *pOp*i may be one of next, prev or swap, each has the following cumulative effect:

- next: The pair (a, b) becomes (a, nextto b in a)
- prev: The pair (a, b) becomes (a, prevto b in a)
- swap: The pair (a, b) becomes (b, a)

The expression returns the second vertex in the resulting pair. The path statement as above uses the current neighbourhoods of the vertices. To use the old neighbourhoods, use a prefixed backquote: ‘@(*vexpr*, *vexpr*) *pOp1*, *pOp2...*, *pOpN*@.

@&(*vexpr*, *vexpr*) *pOp1*, *pOp2...*, *pOpN*@

A variation on the path expression where *vexpr1* and *vexpr2* are passed by reference. This is useful when access to both members of the resulting pair are desired. Again, a prefixed backquote can be used to act on the old neighbourhoods.

A.3.5 Vertex query statements

The following are used to query values regarding the state of a vertex.

labelof *vexpr*

Returns the unique integer representation of the vertex as an unsigned int. If *vexpr* refers to a null vertex, then an error occurs.

valence *vexpr*

Returns the number of neighbours of *vexpr*. If *vexpr* refers to a null vertex, then an error occurs.

is *vexpr1* in *vexpr2*

If *vexpr1* exists in the neighbourhood of *vexpr2*, then the statement returns true, otherwise false. If *vexpr2* is a null vertex, then an error occurs.

A.3.6 Vertex neighbourhood edit expressions

The following are used to edit the neighbourhood of a vertex. These expressions always modify the current state of the vertex referred to by the last *vexpr* in the expression, even if the vexpr evaluates to the old vertex representation.

erase *vexpr1* from *vexpr2*;

Modifies the neighbourhood of *vexpr2* such that *vexpr1* is not in it. If *vexpr2* is a null vertex, then an error occurs.

replace *vexpr1* with *vexpr2* in *vexpr3*;

Modifies the neighbourhood of *vexpr3* such that *vexpr1* is replaced with *vexpr2*. If *vexpr3* is null, then an error occurs.

splice *vexpr1* after *vexpr2* in *vexpr3*;

Modifies the neighbourhood of *vexpr3* such that *vexpr1* follows *vexpr2*. If *vexpr3* is null, then an error occurs.

splice *vexpr1* before *vexpr2* in *vexpr3*;

Modifies the neighbourhood of *vexpr3* such that *vexpr1* precedes *vexpr2*. If *vexpr3* is null, then an error occurs.

make {*vexpr1*, *vexpr2*, ..., *vexprN*} nb_of *vexpr*;

Assigns the neighbourhood of *vexpr* to be the list of N vertices listed. Any neighbourhood that was previously there is overwritten. If *vexpr* is a null vertex, then an error occurs.

A.3.7 Vertex comparisons

Vertices can be compared using ordered relations with boolean tests. The tests use the integer representation of a vertex, available using the labelof *vexpr* statement, as described above. The implemented tests are ==, !=, <, >, <=, >=.

There is also an automatic cast to bool to test if the vertex is a null vertex. This is useful for use in if statements. The unary ! operator is also available with this cast.

A.3.8 Mesh expressions

Any of the following expressions can be used in place of *mexpr*.

ident

Any C++ identifier referring to a mesh variable may be mesh expression.

A.3.9 Mesh query statements

is *vexpr* in *mexpr*;

Check if the vertex referred to by *vexpr* exists in *mexpr*.

A.3.10 Mesh edit statements

clear *mexpr*;

Removes all the vertices from *mexpr*.

add *vexpr* to *mexpr*;

Adds the vertex referred to by *vexpr* to *mexpr*.

remove *vexpr* from *mexpr*;

Removes the vertex referred to by *vexpr* from *mexpr*.

merge *mexpr1* with *mexpr2*;

Adds all of the vertices contained in *mexpr2* into *mexpr1*.

mexpr1* = *mexpr2

Removes all the vertices from *mexpr1* and adds to it all the vertices from *mexpr2*.

synchronise *mexpr*

Assigns the current vertex state to the old state of each vertex contained in *mexpr*.

A.3.11 Iteration

Iteration can be done over both vertex neighbourhoods and meshes. There is no guarantee with regard to the order of the vertices in the iteration. In both cases below, the ellipsis is replaced by any amount of arbitrary code. Also, in both cases, the expression, either *vexpr* or *mexpr* is evaluated once, just prior to entering the code in the curly braces for the first time. If the neighbourhood or set being iterated over is altered before iteration is complete, the behaviour of the program becomes undefined.

forall *ident* in *mexpr* {...}

iterate over each vertex, accessible with the name *ident*, in the mesh referred to by *mexpr*.

forall *ident* in *vexpr* {...}

iterate over each vertex, accessible with the name *ident*, in the neighbourhood of the vertex referred to by *vexpr*.

A.4 The proxy object

The proxy object provides an intermediary repository for data that needs to be communicated between the program and the interpreter. The following members of the proxy object are available for use in a vv program.

unsigned int steps

The number of steps to execute when the *run* command is issued by viewing program can be assigned to this variable. By default, this value is set to zero.

unsigned int delay_msec

A delay between each step, in milliseconds, can be assigned to this variable. By default, this value is set to zero.

bool no_animate

If this variable is set to true, then the *animate* command from the viewing program is ignored. By default, this value is set to false.

bool record_frame

If this variable is set to true, then a screenshot is captured and written to a numbered file after each step in the PNG format* [59]. By default, this value is set to false.

std::list<util::Materials*> materials

A pointer a util::Materials object can be added to this list so that when the viewing program issues the *reread files* command, the data file for that object is reread.

std::list<util::Palette*> palettes

A pointer a util::Palette object can be added to this list so that when the viewing program issues the *reread files* command, the data file for that object is reread.

std::list<util::Function*> functions

A pointer a util::Function object can be added to this list so that when the viewing program issues the *reread files* command, the data file for that object is reread.

std::list<util::Contour*> contours

A pointer a util::Contour object can be added to this list so that when the viewing program issues the *reread files* command, the data file for that object is reread.

VVPViewer* viewer

A pointer to the viewer programs' rendering canvas. This canvas is inherited from the Qt class QGLWidget and so any method from that class may be used on this pointer. Refer to the Qt documentation for details. This pointer should never be modified by the vv program.

*Under Linux, this file is /scratch/f*N*.png and under Windows it is c:\temp\frames\f*N*.png, where *N* is the frame number.

void setViewVolume(float maxx, float maxy, float maxz, float minx, float miny, float minz)

Sets the view volume of the rendering canvas to be no smaller than the size given in the arguments. The view volume size may be larger than asked for.

A.5 The VVM File Format

The file format used by vv to store polygon meshes is the vvm (*vv model*) format. The vvm format is an *XML* (*Extensible Markup Language*) [57, 4] specification and so follows the XML grammar. A *DTD* (*Document Type Declaration*) is not required to be a well-formed vvm file.[†]

There are three tags in a vvm file: mesh to represent a vertex set, v to represent a vertex and e to represent an edge. The top-level tag is the mesh tag and it has no required attributes. Inside the mesh tag, there may be any number of v and e tags. The v tag has one required attribute, "nb" and the e tag has three required attributes, "symmetric", "first" and "second". Each tag may also have an arbitrary set of additional attributes. The additional attributes that match the properties in the definitions part of the vv program will have their values read into program.

In the vvm file, each vertex is assigned an index in the order that they appear in the file, starting with zero. The neighbourhood of each vertex is then defined in the "nb" attribute of a v tag as a list of indices. These indices are also used to indicate to which vertices an edge belongs to using the "first" and "second" attributes in an

[†]A DTD would be cumbersome for vvm files as each model would require a different DTD. Since DTDs are usually only used for validation of an XML file in practice, not having a DTD does not impact the usability of the vvm format.

e tag. If the "symmetric" attribute is set to zero, then the information is assigned to the half-edge going from the vertex at the index indicated in the "first" attribute to that indicated by "second". If it is set to one, the information is set to both half-edges.

Considering an arbitrary declaration in a vv program,

```
edge {
    bool b;
};
vertex {
    double t;
};
mesh {
    int x;
    char y;
};
```

a matching vvm file would be

```
<mesh x="10" y="d">
    <v t="0.1" nb="1 5 3 4" />
    <v t="0.5" nb="2 5 0 4" />
    <v t="1.1" nb="3 5 1 4" />
    <v t="0.9" nb="0 5 2 4" />
    <v t="1.2" nb="3 2 1 0" />
    <v t="0.1" nb="0 1 2 3" />
    <e symmetric="0" first="0" second="1" b="1" />
    <e symmetric="0" first="1" second="0" b="0" />
    <e symmetric="1" first="2" second="3" b="1" />
</mesh>
```

Appendix B

The VV Software Environment Libraries

To facilitate the creation of vv programs, the vv software environment also includes some support libraries and algorithms which are here presented. Since these are not part of the core vv implementation, only summaries of them are given. Complete documentation of these libraries can be found in the vv software environment's documentation.

B.1 Algorithms

The algorithms library is a set of routines that operate on vertices as function objects. In each case, an algorithm is instantiated as an object in the vv program with the parameter V set to vertex.

B.1.1 Rendering

The rendering objects iterate over a set and evaluate the connectivity as needed to render the vertices as some sort of graphical object. These algorithms rely on having the function glRender() defined in the vertex properties.

template <class V> void Draw(algebra::AbstractMesh<V>& mesh, DrawFunc<V>& func)

A wrapper function that takes a DrawFunc function object and applies it to a vertex set.

template <class V> class DrawPoints : public DrawFunc<V>

Iterate over each vertex in a set while in the GL_POINTS mode.

template <class V> class DrawWireframe : public DrawFunc<V>

Iterate over each adjacent pair of vertices in a set while in the GL_LINES mode.

template <class V> class DrawTriangles : public DrawFunc<V>

Iterate over each adjacent triplet of vertices in a set while in the GL_TRIANGLES mode.

template <class V> class DrawTrianglesChecked : public DrawFunc<V>

Iterate over each adjacent triplet of vertices, ignoring triplets that are not a minimal cycle, in a set while in the GL_TRIANGLES mode.

template <class V> class DrawQuads : public DrawFunc<V>

Iterate over each adjacent quadruplet of vertices in a set while in the GL_QUADS mode.

template <class V> class DrawQuadsChecked : public DrawFunc<V>

Iterate over each adjacent quadruplet of vertices, ignoring quadruplets that are not a minimal cycle, in a set while in the GL_QUADS mode.

template <class V> class DrawTriAndQuds : public DrawFunc<V>

Iterate over minimal cycles of these and four vertices in the vertex set while in GL_POLYGON mode.

B.1.2 Stellar Operations

The stellar operations are polygon mesh modification operations that always take a conforming polygon mesh to another conforming polygon mesh [81].

template <class V> class Remove

Remove a vertex from all of its neighbouring vertices' neighbourhoods.

template <class V> class Flip

An edge between two vertices a and b is removed and a new edge is inserted between $a^\star \uparrow b$ and $b^\star \uparrow a$.

template <class V> class Centroid

Places a vertex at the centre of a face and connects it to all the vertices of that face.

template <class V> class EdgeSplit

Inserts a new vertex on an edge and connects it to the vertices opposite that edge. For an edge that connects vertices a and b, the new vertex has the neighbourhood $\{a, b^\star \uparrow a, b, a^\star \uparrow b\}$.

B.1.3 Miscellaneous

template <class V> class Insert

Inserts a new vertex onto an edge, this essentially the same as Algorithm 3.1.

template <class V> class Symmetric

Iterate over a set of vertices and ensure that each adjacent pair of vertices satisfies the symmetry condition (see §3.1).

B.2 Utility

There are a number of utility classes that have nothing to do with the vv data structure or algebra, but are convenient to have.

B.2.1 Geometry

template <class T> class Point

A class that encapsulates point and vector operations. In the examples presented in this thesis, this class was presented using the typedef

```
typedef util::Point<double> Pt;
```

for the sake of notational brevity.

The Point class contains a full set of operator overloads and vector operations such as cross product, distance and normalisation.

template <class T> T angle(util::Point<T> a, util::Point<T> b)

Find the angle between two Point objects

template <class T> T planar_triangle_area(util::Point<T> a, util::Point<T> b, util::Point<T> c)

Find the area of the triangle defined by a, b and c. It is assumed that the triangle lies in the xy-plane.

template <class T> util::Point<T> planar_line_intersection

(util::Point<T> a1, util::Point<T> a2, util::Point<T> b1,

util::Point<T> b2, bool& ia, bool& ib)

Find the intersection of two lines or line segments in the xy-plane.

template <class T> util::Point<T> planar_rotation

(const util::Point<T>& v, T angle)

Rotation of a vector in the xy-plane.

B.2.2 Graphics

template<class T> class Colour : public Point<T>

A class that encapsulates an RGBA-colour. It is used for the definitions of the Palette and Material classes.

class Function

An encapsulation of a spline function in the VLAB function file format.

class Contour

An encapsulation of spline curve in the VLAB curve format.

class Palette

An encapsulation of a colour palette in the VLAB palette format.

class Materials

An encapsulation of a palette of OpenGL materials (ambient, diffuse, emmisive and specular lighting) in the VLAB materials format.

Appendix C

A Complete Example VV Program

All of the vv programs in this thesis so far have been fragments of complete vv programs. For each to compile, it is necessary to add some declarations, and to see the results, some sort of rendering code needs to be added. In this appendix, a complete example vv program with instructions of how to compile and execute it are given.

For this example, it is assumed that Qt 3.x is installed and that g++ 3.x or Visual C++ 7.1*† is installed. There should also be three environment variables defined: QTDIR, the location of the Qt directory, VVDIR, the location of vv directory and VVLIBDIR, the subdirectory 'vvlib' found in the vv directory.‡

In Algorithm C.1, the complete program for the simple case of butterfly subdivision is given. The function butterfly in this algorithm is the same as that provided in Algorithm 5.3. This vv program should be in a file, for the sake of this example, called "program.vvp".

Algorithm C.1: Butterfly subdivision program

```
1 #include <algorithms/insert.hpp>
2 #include <algorithms/render.hpp>
3 #include <util/point.hpp>
4 typedef util::Point<double> Pt;
5 edge {};
```

*Not all the library features in vv cannot be compiled with Visual C++. It is recommended to use the g++ compiler.

†On Windows platforms, Qt 3.x is only compatible with Visual C++.

‡VVLIBDIR can usually be defined as VVDIR/vvlib.

```
6  vertex {
7      Pt pos;
8      void glRender() {
9          glVertex3dv(pos.c_data());
10     }
11 };
12 mesh {};
13 mesh V;
14 algorithms::DrawTriangles<vertex> draw;
15 algorithms::Insert<vertex> insert;
16 void butterfly(mesh & V) {
17     mesh N;
18     synchronise V;
19     forall v in V {
20         forall u in ‘v {
21             if (u < v) continue;
22             vertex x = insert(u, v);
23             x$pos = v$pos * 0.5 + u$pos * 0.5
24                 + (prevto u in ‘v)$pos * 0.125 + (nextto u in ‘v)$pos * 0.125
25                 - (next(2)to u in ‘v)$pos * 0.0625 - (next(2)to v in ‘u)$pos * 0.0625
26                 - (prev(2)to u in ‘v)$pos * 0.0625 - (prev(2)to v in ‘u)$pos * 0.0625;
27             add x to N;
28         }
29     }
30     forall v in N {
31         vertex a = any in v;
32         vertex b = nextto a in v;
33         make { nextto v in a, a, prevto v in a, nextto v in b, b, prevto v in b } nb_of v;
34     }
35     merge V with N;
36 }
37 start {
38     V.readXMLFile(”surface.vvm”);
39 }
40 step {
41     butterfly(V);
42 }
43 render_init {
44     glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
45     glColor3d(0.0, 0.0, 0.0);
46     glLineWidth(2.0f);
47 }
48 render {
49     glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
50     glEnable(GL_POLYGON_OFFSET_FILL);
51     glPolygonOffset(1.0, 1.0);
52     glColor3d(1.0, 1.0, 1.0);
53     algorithms::Draw(V, draw);
```

```
54      glDisable(GL_POLYGON_OFFSET_FILL);
55      glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
56      glColor3d(0.0, 0.0, 0.0);
57      algorithms::Draw(V, draw);
58  }
```

Along with this program, a data file, called "surface.vvm" for this case, in the vvm format is needed (see §A.5 for the vvm specification). This vvm file contains the specification for the double-pyramid shaped polygon mesh used in some of the subdivision surface examples (see Figures 5.2, 5.3, 5.6 & 5.8). The contents of this file are

```
<mesh>
    <v pos="-1 0 -1" nb="1 5 3 4" />
    <v pos="1 0 -1" nb="2 5 0 4" />
    <v pos="1 0 1" nb="3 5 1 4" />
    <v pos="1 0 1" nb="0 5 2 4" />
    <v pos="0 1.4142 0" nb="3 2 1 0" />
    <v pos="0 -1.4142 0" nb="0 1 2 3" />
</mesh>
```

and this file should be in the same directory that the program is executed in.

The program file then needs to be transformed into a regular C++ file using the command

```
vvp2cpp program.vvp program.cpp
```

and that file can then be compiled to a dynamic library with the command

```
g++ program.cpp -c -fPIC -lqt-mt -Werror -Wall
    -I$(QTDIR)/include -I$(VVLIBDIR) -I$(VVDIR)
```

followed by the command

```
g++ program.o -fPIC -lqt-mt -Werror -Wall
    -I$(QTDIR)/include -I$(VVLIBDIR) -I$(VVDIR)
    -shared -Wl,-soname,program.so -o program.so –use-cxa-atexit
```

when using g++ or

```
cl /D "WIN32" /D "_WINDOWS" /D "_USRDLL" /D "_WINDLL"
    /D "_MBCS" /GF /FD /EHsc /GS /Gy /W3 /nologo /c /Wp64
    /Zi /TP /MD /GR /O2 /Og /Ob2 /Oi /Ot /G6 /GA
    /I %VVDIR% /I %VVLIBDIR% /I "c:\Qt\3.2.0Educational\include"
    program.cpp
```

and

```
link /OUT:"program.dll" /INCREMENTAL:NO /NOLOGO /DLL
    /SUBSYSTEM:WINDOWS /OPT:REF /OPT:ICF /IMPLIB:"model.lib"
    model.obj "c:\Qt\3.2.0Educational\lib\qt-mtedu320.lib"
    "c:\Qt\3.2.0Educational\lib\qtmain.lib" opengl32.lib %VVDIR%lib\vvlib.lib
```

when using Visual C++. It is recommended that these commands are put into a makefile. Then, to execute the vv program, the produced dynamic library is loaded into the vvinterpreter program using the command

```
vvinterpreter program.so
```

or with 'program.dll' substituted in when that is the produced file.

Bibliography

[1] I. Adler, D. Barabe, and R. Jean. A History of the Study of Phyllotaxis. *Annals of Botany*, 80(3):231–244, September 1997.

[2] E. Akleman, J. Chen, and V. Srinivasan. A new paradigm for changing topology during subdivision modeling. In *Proceedings of Pacific Graphics*, pages 192–201, October 2000.

[3] B. Baumgart. A polygonhedron representation for computer vision. In *AFIPS Conference Proceedings (Proceedings of the National Computer Conference, May 19–22, 1975. Anaheim, California)*, volume 44, pages 589–596, Montvale, New Jersey, 1975. AFIPS, AFIPS Press.

[4] T. Bray, J. Paoli, C. Sperberg-McQueen, and F. Yergau. Extensible Markup Language (XML) 1.0 (Third Edition) [online]. 2004 [cited November 4, 2005]. Available from: `http://www.w3.org/TR/2004/REC-xml-20040204`.

[5] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-Aided Design*, 10(6):350–355, November 1978.

[6] G. Chaikin. An algorithm for high speed curve generation. *Computer Graphics and Image Processing*, 3:346–349, 1974.

[7] E. Coen. *The Art of Genes: How Organisms Make Themselves*. Oxford University Press, 1999.

[8] E. Coen, A. Rolland-Lagan, M. Matthews, A. Bangham, and P. Prusinkiewicz. The genetics of geometry. *PNAS*, 101(14):4728–4735, 2004.

[9] H. Coxeter. *Introduction to Geometry.* John Wiley & Sons, Inc., United States of America, 2nd edition, 1969.

[10] M. de Boer, F. Fracchia, and P. Prusinkiewicz. A model for cellular development in morphogenetic fields. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 351–370. Springer-Verlag, Berlin, 1992.

[11] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6):356–360, November 1978.

[12] S. Douady and Y. Couder. Phyllotaxis as a Dynamical Self Organizing Process Part I: The Spiral Modes Resulting from Time-Periodic Iterations. *Journal of theoretical biology*, 178(3):255–274, February 1996.

[13] N. Dyn, J. Gregory, and D. Levin. A four-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design*, 4:257–268, 1987.

[14] N. Dyn, D. Levin, and J. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Transactions on Graphics*, 9(2):160–169, 1990.

[15] J. Edmonds. A combinatorial representation of polyhedral surfaces (abstract). *Notices of the American Mathematical Society*, 7:646, 1960.

[16] G. Farin. *Curves and surfaces for CAGD. A practical guide.* Morgan Kaufmann, San Francisco, fifth edition edition, 2002.

[17] A. Fournier, D. Fussell, and L. Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, June 1982.

[18] J.-L. Giavitto and O. Michel. MGS: a Programming Language for the Transformations of Topological Collections. Technical report, Laboratoire de Méthodes Informatiques, CNRS – Université d'Evry Val d'Essonne, Evry, France, Mai 2001.

[19] J.-L. Giavitto and O. Michel. Modeling the topological organization of cellular processes. *BioSystems*, 70(2):149–163, July 2003.

[20] E. Grinspun, A. Hirani, M. Desbrun, and P. Schrder. Discrete shells. In *Proceedings of SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 62–67, July 2003.

[21] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[22] P. Hellendoorn and A. Lindenmayer. Phyllotaxis in *Bryophyllum tubiflorum*: Morphogenetic Studies and Computer Simulations. *Acta Biol. Neerl*, 4:473–492, August 1974.

[23] D. Henderson and D. Taimina. Crocheting the Hyperbolic Plane. *The Mathematical Intelligencer*, 23(2):17–28, 2001.

[24] H. Hoppe, T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 295–302, July 1994.

[25] R. Karwowski. *Improving the process of plant modeling: the L+C modeling language*. PhD thesis, University of Calgary, August 2002.

[26] R. Karwowski and P. Prusinkiewicz. Design and Implementation of the L+C Modeling Language. *Electronic Notes in Theoretical Computer Science*, 86(2):1–19, September 2003.

[27] L. Kobbelt. $\sqrt{3}$-subdivision. In *Proceedings of SIGGRAPH*, pages 103–112. ACM, 2000.

[28] R. Korn and R. Spalding. The geometry of plant epidermal cells. *New Phytologist*, 72(6):1357–1365, 1973.

[29] B. Lane and P. Prusinkiewicz. Algorithmic Botany – The Virtual Laboratory [online]. 2005 [cited July 17, 2005]. Available from: `http://algorithmicbotany.org/virtual_laboratory/`.

[30] J. Levine, T. Mason, and D. Brown. *lex & yacc*. UNIX Programming Tools. O'Reilly & Associates, Inc., USA, second edition with minor corrections edition, 1995.

[31] P. Lienhardt. Topological models for boundary representation: a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1):59–92,

January - February 1991.

[32] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theretical Biology*, 18(3):280–315, March 1968.

[33] A. Lindenmayer and G. Rozenberg. Parallel generation of maps: Developmental systems for cell layers. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Graph grammars and their application to computer science; First International Workshop*, Lecture Notes in Computer Science 73, pages 301–316. Springer-Verlag, Berlin, 1978.

[34] M. Livio. *The Golden Ratio: The Story of Phi, the World's Most Astonishing Number*. Broadway Books, New York, New York, October 2002.

[35] C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, The University of Utah, August 1987.

[36] B. Mandelbrot. Les objet fractals. *La Recherche*, 9(85):1–13, January 1978.

[37] B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, San Francisco, 1982.

[38] M. Mäntylä. *An Introduction to Solid Modeling*, chapter 10, pages 161–174. Principles of Computer Science Series. Computer Science Press, Rockville, Maryland, 1988.

[39] M. Marder. The Shape of the Edge of a Leaf, November 2002.

[40] M. Marder, E. Sharon, S. Smith, and B. Roman. Theory of edges of leaves. *Europhysics Letters*, 62(4):498–504, May 2003.

[41] H. Maschke. Note on the unilateral surface of moebius. *Transactions of the American Mathematical Society*, 1(1):39, 1900.

[42] M. McManus and B. Veit, editors. *Meristematic Tissues in Plant Growth and Development*. Sheffield Biological Sciences. Sheffield Academic Press Ltd., 2002.

[43] H. Meinhardt. *The Algorithmic Beauty of Sea Shells*. The Virtual Laboratory. Springer-Verlag, Berlin, 1995.

[44] H. Meinhardt. Complex pattern formation by a self-destabilization of established patterns: chemotactic orientation and phyllotaxis as examples. *Comptes Rendus Biologies*, 326:223–237, 2003.

[45] A. Möbius. über die Bestimmung des Inhalts eines Polyeders. *Berichte über die Verhandlungen der Königlich sächsischen Gesellschaft der Wissenschaften zu Leipzig*, 17:31–68, 1864.

[46] J. Nakielski. Tensorial model for growth and cell division in the shoot apex. In A. Carbone, M. Gromov, and P. Prusinkiewicz, editors, *Pattern Formation in Biology, Vision and Dynamics*, pages 252–267. World Scientific Publishing Co. Pte. Ltd., Signapore, 2000.

[47] H.-R. Pakdel and F. Samavati. Incremental adaptive loop subdivision. In A. Lagan, M. Gavrilova, and V. Kumar, editors, *Computational Science and Its Applications – ICCSA 2004: International Conference, Assisi, Italy, May 14-17, 2004, Proceedings, Part III*, volume 3045 of *Lecture Notes in Computer Science*, pages 237–246, Berlin Heidelberg, April 2004. Springer-Verlag.

[48] H.-R. Pakdel and F. Samavati. Incremental catmull-clark subdivision. In *Proceedings of the 5th International Conference on 3-D Digital Imaging and Modeling*, 3DIM 2005, pages 95–102. IEEE Computer Society Press, June 2005.

[49] H.-R. Pakdel and F. Samavati. Incremental subdivision for triangle meshes. Accepted for publication in the International Journal of Computational Science and Engineering, 2005.

[50] R. Penrose. The rôle of aesthetics in pure and applied mathematical research. *The Institute of Mathematics and its Applications Bulletin*, 10(7/8):266–271, July/August 1974.

[51] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, 2nd edition, 1992.

[52] P. Prusinkiewicz. Graphical applications of l-systems. In *Proceedings of Graphical Interface '86 – Vision Interface '86*, pages 247–253. CIPS, 1986.

[53] P. Prusinkiewicz. Applications of L-systems to computer imagery. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Graph-Grammars and Their Application to Computer Science (3rd International Workshop, Warrenton, Virginia, USA, December 1986)*, volume 291 of *Lecture Notes in Comuter Science*, pages 534–548, Heidelberg, 1987. Springer-Verlag.

[54] P. Prusinkiewicz and M. Hammel. A fractal model of mountains with rivers. In *Proceeding of Graphics Interface '93*, pages 174–180, May 1993.

[55] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990. With J. Hanan, F. Fracchia, D. Fowler, M. de Boer, and L. Mercer.

[56] P. Prusinkiewicz, F. Samavati, C. Smith, and R. Karwowski. L-System Description of Subdivision Curves. *Internation Journal on Shape Modeling*, 9(1):41–59, June 2003.

[57] E. Ray. *Learning XML*. O'Reilly & Associates, Inc., Sebastopol, California, 2nd edition, 2003.

[58] J. Ridley. Computer simulation of contact pressure in capitula. *Journal of theoretical biology*, 95:1–11, 1982.

[59] G. Roelofs. Portable network graphics (png) specification and extensions [online]. 2004 [cited September 27, 2005]. Available from: `http://www.libpng.org/pub/png/spec/`.

[60] J. Rossignac. Specification, representation, and construction of non-manifold geometric structures. In *Siggraph '94 Course Notes*, 1994.

[61] J. Rossignac, A. Safonova, and A. Szymczak. 3d compression made simple: Edgebreaker on a corner-table. In *Shape Modeling International Conference*, pages 278–283, 2001.

[62] W. Schwabe and A. Clewer. Phyllotaxis – a Simple Computer Model Based on the Theory of a Polarly-Translocated Inhibitor. *Journal of theoretical Biology*, 109:595–619, 1984.

[63] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., April 1999. Available at `http://www.opengl.org/documentation/specs/version1.2/OpenGL_spec_1.2.1.pdf`.

[64] M. Senechal. *Quasicrystals and Geometry*. Cambridge University Press, 1995.

[65] E. Sharon, M. Marder, and H. Swinney. Leaves, flowers and garbage bags: Making waves. *American Scientist*, 92(3):254–261, May–June 2004.

[66] E. Sharon, B. Roman, M. Marder, G.-S. Shin, and H. Swinney. Mechanics: Buckling cascades in free sheets. *Nature*, 419:579–579, October 2002.

[67] M. Sierpinski. Sur une courbe dont tout point est un point de ramification. *Compte Rendus hebdomadaires des séance de l'Académie des Science de Paris*, 160:302–305, 1915.

[68] C. Smith and P. Prusinkiewicz. Simulation modeling of growing tissues. In C. Godin, J. Hanan, W. Kurth, A. Lacointe, A. Takenaka, P. Prusinkiewicz, T. DeJong, C. Beveridge, and B. Andrieu, editors, *4th International Workshop on Functional-Structural Plant Models*, pages 365–370, Montpellier, France, June 2004.

[69] C. Smith, P. Prusinkiewicz, and F. Samavati. Local specification of surface subdivision algorithms. In J. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003 Revised Selected and Invited Papers)*, volume 3062 of *Lecture Notes in Computer Science*, pages 313–327, Heidelberg, 2004. Springer-Verlag.

[70] T. Steeves and I. Sussex. *Patterns in Plant Development*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[71] E. Stollnitz, T. DeRose, and D. Salesin. *Wavelets for computer graphics*. Morgan Kaufman, San Francisco, 1996.

[72] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3rd edition, 1997.

[73] A. Szilard and R. Quinton. An interpretation for D0L-systems by computer graphics. *The Science Terrapin*, 4:8–13, 1979.

[74] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. In Maureen C. Stone, editor, *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 205–214. ACM Press, 1987.

[75] D. Thompson. *On Growth and Form*. Cambridge University Press, Cambridge, second edition, 1942.

[76] J. Thornley. Phyllotaxis. I. A Mechanistic Model. *Annals of Botany*, 39:491–507, 1975.

[77] J. Thornley. Phyllotaxis. II. A Description in Terms of Intersecting Logaritmic Spirals. *Annals of Botany*, 39:509–524, 1975.

[78] S. Ulam. On some mathematical problems connected with patterns of growth and figures. In *American Mathematical Society Proceedings of Symposia in Ap-*

plied Mathematics, volume 14, pages 215–224. American Mathematical Society, 1962.

[79] S. Ulam. Patterns of growth of figures: Mathematical aspects. In G. Kepes, editor, *Module, Proportion, Symmetry, Rhythm*, pages 64–74. Braziller, 1966.

[80] A. Veen and A. Lindenmayer. Diffusion Mechanism for Phyllotaxis. *Plant Physiology*, 60:127–139, 1977.

[81] L. Velho. Stellar subdivision grammars. In L. Kobbelt, P. Schrder, and H. Hoppe, editors, *Eurographics Symposium on Geometry Processing*, 2003.

[82] G. Vermeij. *A Natural History of Shells*. Princeton Science Library. Princeton University Press, United States of America, 1993.

[83] H. Vogel. A Better Way to Construct the Sunflower Head. *Mathematical Biosciences*, 44(3):179–189, June 1978.

[84] H. von Koch. Une méthode géométrique pour l'étude de certaines questions de la théorie des courbes planes. *Acta Mathematica*, 30:145–174, 1906.

[85] J. Warren and H. Weimer. *Subdivision Methods for Geometric Design : A Constructive Approach*. Morgan Kaufmann series in computer graphics and geometric modeling. Morgan Kaufmann, San Francisco, 2002.

[86] A. White. *Graphs, groups and surfaces*. North-Holland, Amsterdam, 1973.

[87] R. Williams. *The Shoot Apex and Leaf Growth*. Cambridge University Press, 1974.

[88] D. Zorin, P. Schröder, T. DeRose, L. Kobbelt, A. Levin, and W. Sweldens. Subdivision for modeling and animation. In *SIGGRAPH Course Notes*, New York, 2000. ACM.