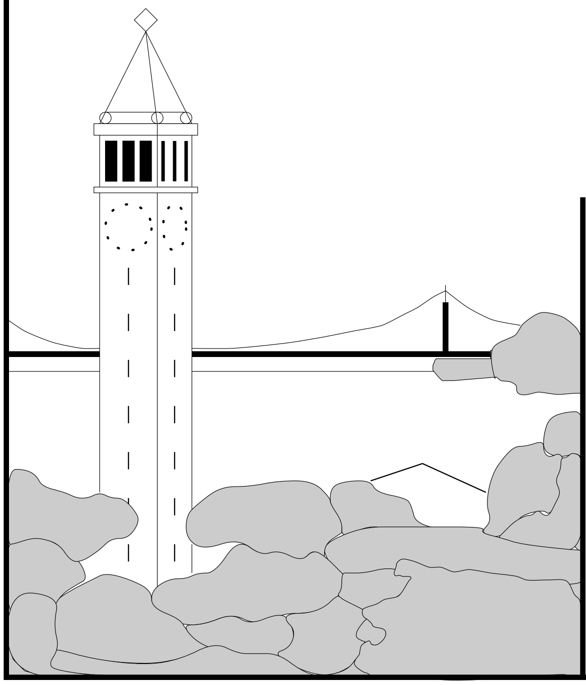
Geometric Algorithms and Data Representation for Solid Freeform Fabrication

Sara Anne McMains



Report No. UCB/CSD-00-1119

October 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Geometric Algorithms and Data Representation for Solid Freeform Fabrication

by

Sara Anne McMains

A.B. (Harvard College) 1991
M.S. (University of California, Berkeley) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Carlo H. Séquin, Chair
Professor Joseph M. Hellerstein
Professor Paul K. Wright

Fall 2000

The dissertation of Sara Anne McMains is approved:

Carlo H. Séquin 6/29/00

Chair Date

Paul Wright 7/11/00

Date

[illegible] M. [illegible] 8/21/2000

Date

University of California, Berkeley

Fall 2000

Geometric Algorithms and Data Representation for Solid Freeform Fabrication

by

Sara Anne McMains

Abstract

Geometric Algorithms and Data Representation for Solid Freeform Fabrication

by

Sara Anne McMains

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Carlo H. Séquin, Chair

Solid freeform fabrication (SFF) refers to a class of technologies used for making rapid prototypes of 3-D parts. With these processes, a triangulated boundary representation of the CAD model of the part is "sliced" into horizontal 2.5-D layers of uniform thickness that are successively deposited, hardened, fused, or cut, depending on the particular process, and attached to the layer beneath. The stacked layers form the final part.

The current de facto standard interface to these machines, STL, has many shortcomings. We have developed a new "Solid Interchange Format" (SIF) for use as a digital interface to SFF machines. SIF includes constructs for specifying surface and volume properties, precision information, and transmitting unevaluated Boolean trees. We have also developed a 2-D variant, LSIF (Layered SIF), for describing the sliced layers.

Many solid modeling applications require information not only about the geometry of an object but also about its "topology" – the connectivity of its faces, edges, and vertices. We have designed a new topological data structure, the loop edge data structure (LEDS), specifically targeted at supporting SFF software. For very large data sets, the topological data structure itself can be bigger than core memory, and a naive algorithm for building it becomes prohibitively slow due to memory thrashing. We have developed an algorithm for building the LEDS efficiently from a boundary representation, even when it doesn't fit in

main memory, improving software performance for highly tessellated parts by two orders of magnitude.

We have implemented an analysis and cleanup tool for faceted B-reps on top of this data structure. The analysis module reports basic topological information about the part and checks the boundary for cracks. The cleanup module closes cracks via intelligent vertex merging.

We have developed a new sweep-plane slicing algorithm that also operates on the LEDS. Our slicer outputs LSIF, passing along the unevaluated booleans to be resolved at the 2-D level. This slicer exploits the coherence between the finely spaced parallel slices needed for SFF process planning and fabrication. Its performance is two to three times faster than that of a comparison commercial STL slicer on typical parts.

Carlo H. Séquin

Professor Carlo H. Séquin
Dissertation Committee Chair

In memory of Dr. Sara Anne Cassaday

Contents

List of Figures

List of Tables

Acknowledgements

I have been very fortunate to have Carlo Séquin as my advisor at Berkeley. He has provided me with expert guidance, extensive and timely feedback on my research and writing, and his enthusiasm and dedication have never lagged. Paul Wright taught me about manufacturing, welcomed me as an auxilliary member of his research group, and offered valuable comments on this dissertation. Joe Hellerstein introduced me to fundamental database research and techniques in his course, then patiently learned about geometric modeling in order to better guide my research, from qualifying exam proposal through final draft of my thesis. John Canny also provided useful suggestions during my qualifying exam.

Maryann Simmons has been the source of invaluable technical and moral support, from maintaining hardware and software for our whole research group, to faithfully attending and giving feedback on every single practice talk I've ever given at Berkeley. Jordan Smith implemented the original versions of many of the SIF support libraries, patiently answered all of my questions during my "conversion" from a C to a C++ programmer, and shared countless hours of discussion about how to best transmit, analyze, and slice parts for solid freeform fabrication. We couldn't have done it without the rest of the SIF team, Jane Yen and Jianlin Wang, who have ensured that I had interesting input files and written the software to process my output files, respectively. My officemates during my years at Berkeley, including Jordan, Jane, Serge Belongie, Mark Brunkhart, Steve Chenney, Frank Cho, Joni Fazo, and Adam Sah, have provided friendship, advice, phone messages, reference books, and other distractions. I can always count on Greg Ward for answers to obscure Unix questions or an opportunity to dust off my rollerblades. I've enjoyed class projects shared with Maryann, Ajay Sreekanth, Randy Keller, Taku Tokuyasu, Tina Wong, and Jacqueline Chang. I've learned a tremendous amount about manufacturing from all of the students in Professor Wright's Integrated Manufacturing Lab over the years. I couldn't have gotten through prelims without my study partner Ketan Patel. I had a great time teaching sections of undergraduate computer graphics with Laura Downs and Dan Garcia, under Brian Barsky, Kevin Weiler, and David Forsyth. For systems advice I could always turn to Remzi Arpaci-Dusseau. Mike Wittman has cheerfully served as my Linux guru. We have a great staff in Soda Hall, and I would particularly like to thank Catherine Crabtree, Ann Fuller, Terry Lessard-Smith, Phil Loarie, and Jon Forrest, who have always given me cheerful assistance over the years.

I've also received valuable assistance from outside of Berkeley. Professor Rich Crawford and David Thompson from the University of Texas, Austin gave me specifications and code for reading STL files. Professor Deba Dutta and Anne Marsan from the University of Michigan, Ann Arbor, provided advice on Fused Deposition Modeling machines and interfaces. Professor Marc Levoy of Stanford gave me access to models from the Digital Michaelangelo project and the 3D Scanning Repository, and Simon Rusinkiewicz and Henrik van Jensen helped me interpret the file format. Mike Hohmeyer and the folks at ICEM CFD gave me the opportunity to work with descriptions of complex, real-world parts.

This work was supported in part by an NSF graduate fellowship and grants MIP-96-32345 and EIA-99-05140 from the National Science Foundation. Grants from the NSF (EIA-96-17995), Visteon Systems/Ford Motor Company, and a Cal Micro grant funded the purchase of our FDM machine, without which my algorithms would not have been used to produce actual physical models.

And last but not least, a special thank you to my husband Phillippe and my family for all their support through the years.

Chapter 1

Introduction

1.1 Solid Freeform Fabrication

Several technologies collectively referred to as layered manufacturing or solid freeform fabrication (SFF) have been developed recently for making prototypes of solid three-dimensional parts directly from CAD descriptions. These include stereolithography (SLA) [33], 3-D printing [63], selective laser sintering (SLS) [9], fused deposition modeling (FDM) and laminated object manufacturing (LOM) [2, 3, 42, 48, 52, 82]. The prototypes they produce can serve as physical models during design review, allow engineers to perform functional testing of parts, or be used as mold patterns or positives for secondary tooling. These technologies promise to play an increasingly important role in fully automated rapid prototyping of mechanical parts of arbitrarily complex shape with only a minimal amount of process planning.

In all these processes, a triangulated boundary representation (b-rep) of the CAD model of the part in STL format [1] is "sliced" into horizontal 2.5-D layers of uniform thickness. Each cross sectional layer is successively deposited, hardened, fused, or cut, depending on the particular process, and attached to the layer beneath it. The stacked layers form the final part.

1.2 Motivation

The introduction of the first commercial SFF technology, stereolithography, was accompanied by the introduction of STL, a descriptive format to specify the solid shape to be produced. Unfortunately, although STL has become the de facto standard exchange format for the SFF industry, it is inadequate for many reasons. STL is a faceted, non-connected, boundary description of a solid. The facets are triangles, transmitted as three 3-D vertex coordinates and a normal vector. Redundant information at the lowest level is inherently required, since each vertex must appear in at least three triangular facets on the boundary of a valid closed solid, and its coordinates must be specified explicitly for each facet that shares that vertex. The inside and outside of the solid relative to each facet is specified by the ordering of the facet vertices as well as by a surface normal. This constitutes redundant information, yet there is no semantic resolution if this information is inconsistent. Since triangle vertices are not explicitly shared, round-off errors may produce small cracks in the described boundary. This lack of connectivity also makes it more difficult to check whether a model describes a closed solid.

After receiving a part description, whether in STL or some other format, an application must build an internal representation to process it. Ideally, the internal representation will record not only geometry, but also topological information about adjacency relationships. Adjacency information is useful for a variety of applications, including analysis, file repair, slicing, and offsetting.

Building such a topological data structure efficiently presents a challenge for large data sets. When the data is small enough, memory-resident auxiliary intermediate data structures can be used to derive the connectivity. But for very large data sets, where these intermediate structures will not fit in memory, random accesses to read them can lead to thrashing. In addition, random accesses to update the existing data structure with connectivity information to newly processed geometry can cause additional thrashing.

The first step in processing an SFF input file is to verify that it indeed represents the boundary of a valid, closed solid (or in the case of SIF input with Booleans, that the primitives are valid, closed solids). Often there will be gaps or cracks in the boundary, particularly for STL input where vertices are not shared, caused by round-off errors that occur as the result of instancing or trimming operations on the design side. An unambiguous, closed boundary is necessary in order to determine what volume is inside of and what is outside of the part, because the SFF machine needs to know where to deposit/fuse/harden/etc. the build material.

The next step that is required to build the SFF part is to slice it into parallel layers. Previous algorithms for slicing the 3-D b-rep into the layers that form the process plan for these machines have treated each slice operation as an individual intersection with a plane. But for a typical stereolithography build with .005" layers, a 5" high part will be made from one thousand parallel slices with significant coherence between slices. An additional shortcoming of many existing slicers is a lack of robustness when dealing with non-2-manifold geometry.

1.3 Contribution

We have developed a simple and clean interchange format for SFF called “SIF” for “Solid Interchange Format,” in close analogy to CIF, the Caltech Intermediate Form, which serves a similar function for the exchange of LSI layout descriptions [49]. With SIF, our goal was a simple and easy-to-describe interchange format powerful enough to describe a wide variety of parts unambiguously. We have fixed some of the obvious shortcomings of STL by giving vertices identifiers and sharing the vertices between triangles, eliminating redundant surface normal information, and specifying units and version information. We have introduced constructs for specifying surface and volume properties, precision information, and the ability to describe parts with unevaluated Boolean constructive solid geometry (CSG) operations.

We have implemented a new variation of a radial edge data structure, the loop edge data structure (LEDS), specifically targeted at supporting SFF software. LEDS is designed to record connectivity information for all valid solids, including non-2-manifold solids. For efficient memory allocation, we use constant space storage for each vertex and each edge use, at the same time answering the majority of topological adjacency queries in time linear in the number of responses, independent of the size of the input. We also study trade-offs between time and space efficiency.

For building the LEDS, we implemented and optimized two different algorithms. The first is designed for an in-memory build, using hash tables for efficient derivation of connectivity information. Unfortunately, using this simple algorithm with large files causes virtual memory thrashing. The second algorithm is an out-of-core algorithm, designed such that the only random accesses are to structures that fit in main memory, limiting disk accesses to sequential reads and writes. This algorithm exhibits run times that only increase linearly, not exponentially, when main memory size is exceeded. One of the goals of this work was to demonstrate that a full topological data structure can be constructed efficiently from a bucket of facets, obviating the need to include the topological data structure explicitly in the interchange format.

After building the LEDS, we use it to analyze the input. We verify that there are no gaps in the boundary, or if there are gaps, we make a limited attempt to repair the file automatically (we only attempt to fix problems caused by round-off errors in the vertices). If the data still has problems, other file repair programs, possibly involving user interaction, must be used to fix the file.

Our slicing algorithm exploits both geometric and topological inter-slice coherence to output clean slices with no self-intersections and explicit nesting of contours. After building the initial LEDS, we logically separate non-2-manifold edges and vertices into coincident "pseudo-2-manifold" edges and vertices, so that they will be handled correctly. The main body of the algorithm uses a sweep plane approach, using the connectivity information for the 3-D solid to derive and update the connectivity of the 2-D slices. Actual intersection calculations on the edges are performed quickly and efficiently through incremental updates between slices. The resulting slice descriptions are topologically consistent, connected, nested contours, rather than simply unordered collections of edges.

Chapter 2

Background and Previous Work

2.1 Solid Representation Schemes

Several representations are commonly used to define solid geometry, including parameterized sweeps and generalized cylinders, spatial partitioning schemes such as voxels, octrees, or binary space partitioning trees, constructive solid geometry (CSG), and boundary representations (b-reps) [20, 29, 50]. For manufacturing applications, CSG and b-reps are most common.

2.1.1 CSG

With CSG, solid primitives such as cones, spheres, cubes, and half spaces are scaled, translated, and/or rotated using geometric transforms and then combined via Boolean set operations (union, intersection, and difference) to describe more complicated shapes. Since the domain of solid objects is not closed under standard Boolean operations (for example, the intersection of two 3-D solids can result in a 2-D face or a 1-D edge as shown in Figure 2.1), a variation called regularized Boolean set operations are used for solid modeling with CSG [57].

A regularized Boolean set operation is carried out by first performing the standard Boolean operation, then taking the result and finding the closure of its interior (see Figure 2.2). The process of taking the closure of the interior is called "regularization." The regularized Boolean operators are denoted as $\bigcap$*, $\bigcup$*, and $-$*.

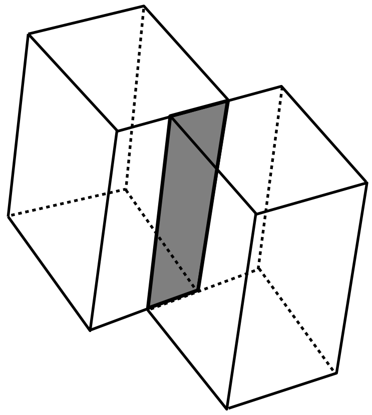
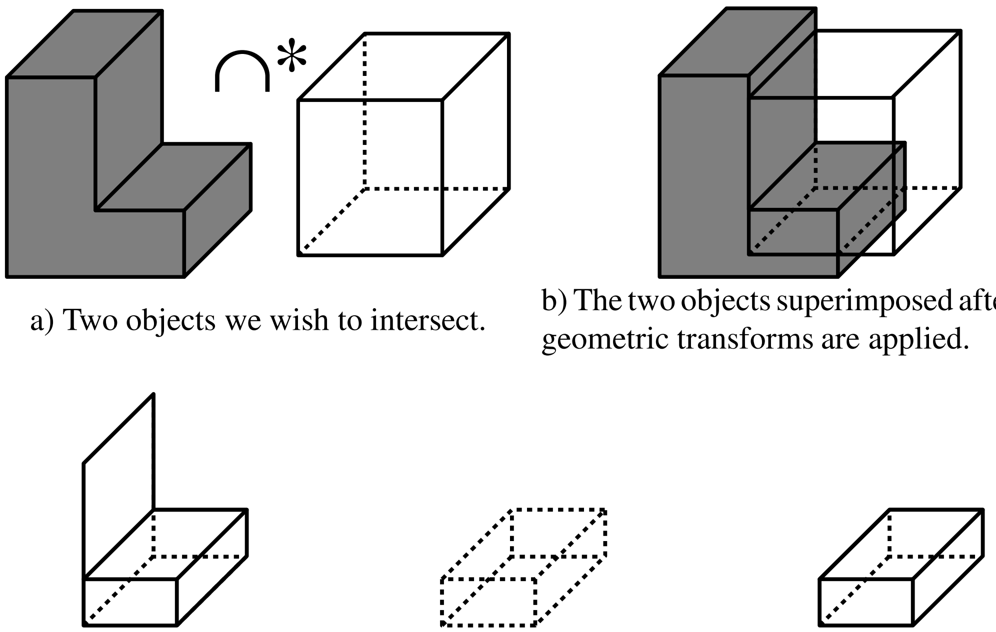


Figure 2.1: *Examples of Boolean operations on manifold solids that do not yield solids. The intersection of the two cubes on the left is the gray 2-D face. The intersection of the two cubes on the right is the gray 1-D edge.*



a) Two objects we wish to intersect.

b) The two objects superimposed after their geometric transforms are applied.

c) Their intersection.

d) The interior of their intersection.

e) The closure of the interior of their intersection.

Figure 2.2: *A regularized Boolean intersection operation ($\bigcap$*).*

An unevaluated CSG representation is in the form of a tree with the operands (the transforms and Boolean operators) at the nodes and the solid primitives at the leaves (see Figure 2.3). Although some display algorithms operate on unevaluated CSG trees directly [56], for analysis purposes it is generally necessary to convert the CSG tree to an evaluated representation such as a b-rep.

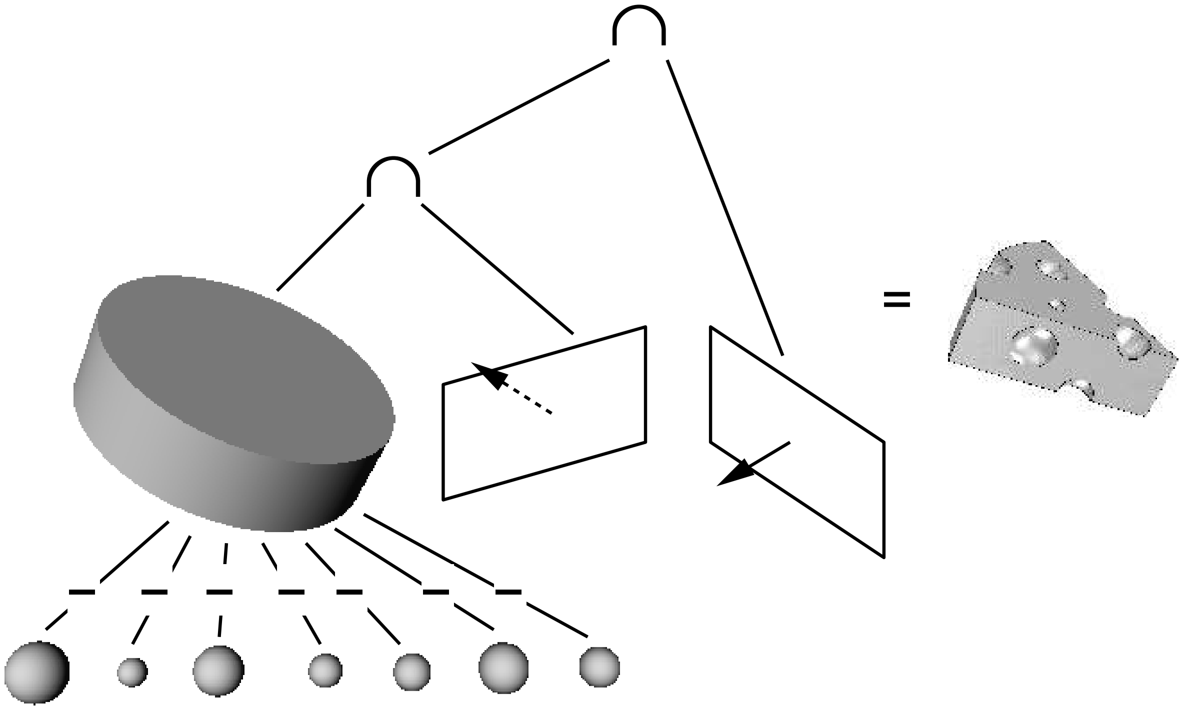


Figure 2.3: *A CSG tree describing a wedge of Swiss cheese as the intersection of a cylinder and two half spaces, minus several spheres. The spheres are illustrated after the application of scaling transforms; translation transforms applied to each of the leaves of the tree are not pictured.*

2.1.2 B-reps

With a b-rep, only the geometry of the surface that forms the boundary between the interior and exterior of the solid is represented explicitly. The boundary surface may be described by polygonal faces, by trimmed and untrimmed tensor-product spline patches [7, 31], or by subdivision surfaces [30, 58, 65].

These faces are oriented, having a front and a back. By convention, the front face of a polygon in a b-rep is the side that is visible when viewed from outside the solid. In a right-handed coordinate system, a simple polygonal face's boundary is defined by a list of vertices ordered so that if we walk the oriented edges between them, while standing on the outside of the polyhedron, the face will always be to our left. For a non-simple polygonal face defined by multiple contours, the same will be true if we walk the oriented edges of the inner hole contours. The outer contour of a face is oriented counter-clockwise as seen from the front of the face; any inner hole contours are oriented clockwise. The normal of a face is found by taking the cross product of two consecutive edges of its outer contour that form a convex corner. This surface normal points out of the front of the face toward the exterior of the part (see Figure 2.4). Thus by looking at the orientation of a face we can tell on which side solid material lies – which side is the interior of the part and which the exterior.

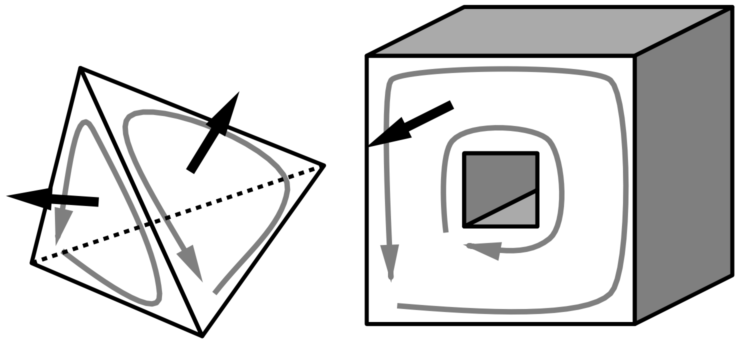


Figure 2.4: *In a right handed coordinate system, the outer contour of each face is oriented counter-clockwise, while any hole contours are oriented clockwise (gray arrows indicate contour orientations). Surface normals point toward the outside of the solid (black arrows indicate surface normals).*

It is in principle straightforward to evaluate a CSG tree to obtain a b-rep, but rounding errors are a common problem. Such a conversion operation is needed when a modeling system provides a CSG interface but the exchange format is a b-rep. Deriving a compact CSG tree from an initial b-rep is conceptually more complicated, as there are many different possible CSG trees to describe the same geometry. This is one reason that b-reps, or a combination of b-rep and CSG, are favored over CSG-only for exchange formats. Another advantage of b-reps is their compatibility with data structures providing adjacency information, which is useful for analysis operations, such as the determination of connectedness and counting components [83, 77].

2.1.3 Manifold Properties

Many solid modeling kernels are restricted to 2-manifold geometry. On a 2-manifold boundary, the mathematical neighborhood of each point is topologically equivalent to a 2-D disk. Any geometry that does not have this property is *non-manifold*. In the b-rep of a 2-manifold, each edge is used by exactly two faces, in opposite directions (see Figure 2.5). (Note that this is a necessary, but not a sufficient, condition for manifoldness.)

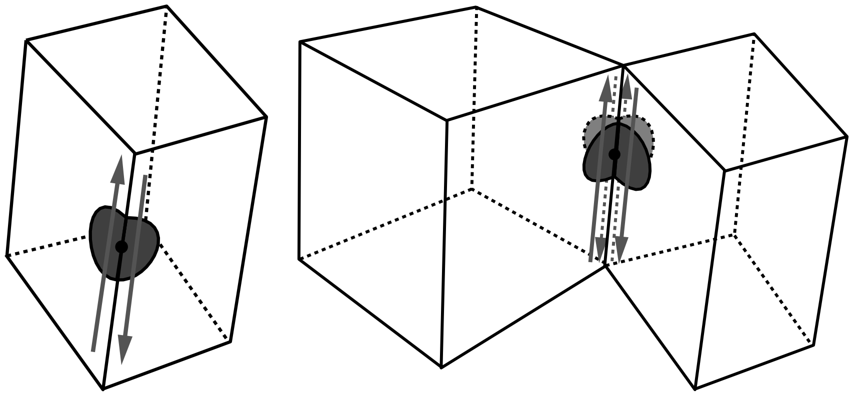


Figure 2.5: *The cube on the left is 2-manifold. The shaded neighborhood of the point indicated on the front edge is topologically equivalent to a disk, and the edge is used once in both directions, as indicated by the gray arrows on the faces that use it. The two cubes on the right share the central, non-manifold edge. The shaded neighborhood of the point indicated on the non-manifold edge is topologically equivalent to two disks, and the edge is used twice in each direction.*

Merely supporting 2-manifold geometry is inadequate for robust solid modeling systems. The set of 2-manifold solids is not closed even under *regularized* Boolean operations – the two touching cubes in Figure 2.5 are one example: a non-manifold edge appears in the union of two manifold cubes. It is not physically possible to manufacture such an infinitely thin edge, but the ideal shape the designer wants could still be a connected solid with a non-manifold edge (see Figure 2.6). Non-homogeneous parts where multiple materials abut are also non-2-manifold.

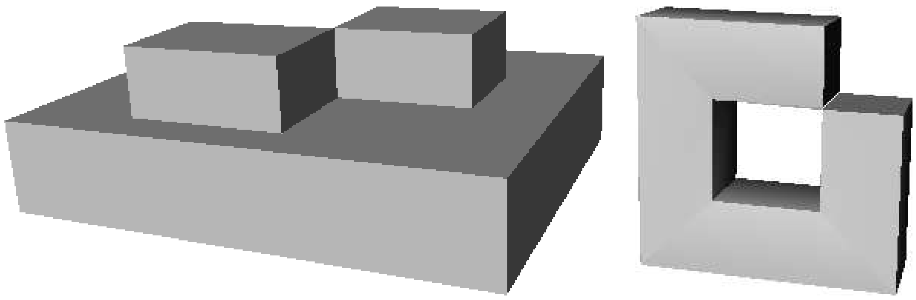


Figure 2.6: *Two examples of rigidly connected parts that contain non-manifold edges.*

Ideally, a design system should support other non-manifolds so that mixed dimension geometry can be represented, not just solids. Even if a designer ultimately wants a solid part, the intermediate geometry that arises in the process of designing that part may not be closed. Non-manifold entities are also useful for partitioning an object for analysis, to identify features for machining, or to establish reference points or planes for measurement and tolerancing.

2.2 Topological Data Structures for Solid Models

To analyze a b-rep (for example, to determine whether each edge is used in opposite directions by two faces in a valid 2-manifold), a data structure that includes adjacency information, or *topology*, is useful. A popular topological data structure for b-reps of 2-manifolds is Baumgart's winged edge data structure [8], illustrated in Figure 2.7. Each edge points to its endpoints, to the two adjacent faces that share it, and to the next and previous edges around those two faces. Faces and vertices each point to one of their edges. This data structure allows us to answer many questions about topological adjacency relationships efficiently, such as the two faces coincident to an edge, or all the edges adjacent to a vertex. Mäntylä's half-edge data structure [47] is a variation on the winged-edge data structure, also limited to 2-manifolds, that divides the information stored with a winged edge into two half-edges, one associated with each adjacent face. Kalay, in a 2-D architectural floor plan editor, has also used a similar encoding [37].

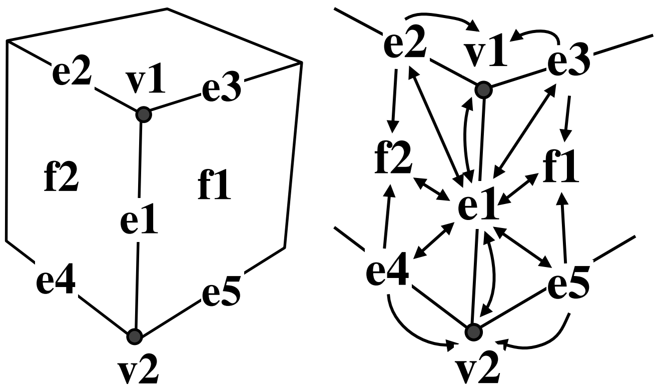


Figure 2.7: *In a winged edge data structure, connectivity information is stored primarily with the edges. The front edge of the cube on the left points to its vertices, adjacent faces, and neighboring edges around these faces, as pictured on the right. Vertices and faces point back to a single adjacent edge.*

Valid non-2-manifold b-rep solids, such as the ones pictured in Figure 2.6, can be represented using *pseudo-2-manifold* representations, which have 2-manifold connectivity but non-2-manifold geometry [46, 62]. For these valid solids, the neighborhood of each point is topologically equivalent to n disks, and each edge in the b-rep is used an equal number of times in both directions. To make a pseudo-2-manifold representation, we must virtually separate these disks (see Figure 2.8). For non-manifold edges, this is accomplished by pairing up edges with opposite orientation so that each pair corresponds to one of n abutting but non-intersecting disks. For non-manifold vertices, this is accomplished by making duplicate coincident vertices for each disk, connected to the edges on that disk. With a pseudo-2-manifold representation, adjacency information is lost at non-manifold points on the boundary.

Weiler's radial edge data structure is a generalization of the winged-edge data structure that supports adjacency queries on non-2-manifolds [78]. An important concept from Weiler's work is the distinction between an abstract, unoriented geometric entity, such as an edge, and an oriented use of that entity, such as a directed *edge-use* that describes part of the boundary of a face. For example, the geometry of an edge between two points is stored once and then referred to by as many edge-uses as faces share that edge. In the winged edge structure, the implicit edge-uses are limited by the two pointers that pointed to the two

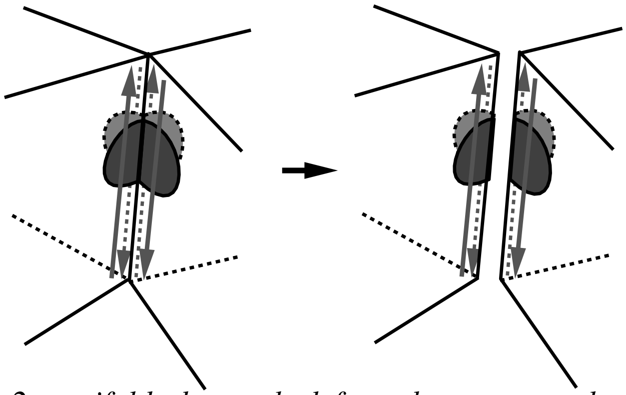


Figure 2.8: *The non-2-manifold edge on the left can be represented as pseudo-2-manifold by duplicating both end points and connecting pairs of edges with opposite orientation. The edges and duplicated vertices will be geometrically coincident, but their connectivity will be as for the 2-manifold geometry shown on the right.*

faces that used the edge in opposite directions. In the radial edge data structure, both the unoriented geometry and its uses are stored explicitly. Weiler's data structure also records the radial ordering of faces around non-manifold edges (hence its name). The internal data structure that we build is a stripped-down version of the radial edge data structure.

Another pioneering non-manifold representation forms the basis for the NOODLES geometric modeling system developed at Carnegie Mellon [27, 28]. In addition to the radial ordering of faces around non-manifold edges, Noodles divides the distinct areas around non-manifold vertices into explicit *zones*. The manifold sheets that form the boundaries between zones are called *disks* (see Figure 2.9).

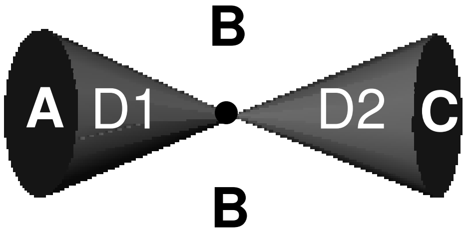


Figure 2.9: *The vertex in the center is surrounded by three zones, A, B, and C. They are bounded by two disks, D1 and D2.*

Each disk records a cyclically ordered list of edge-uses incident to the vertex and on the surface of the disk. This is called a disk cycle (see Figure 2.10). Weiler subsequently described a modification to his original radial edge structure to represent this vertex neighborhood information explicitly [79].

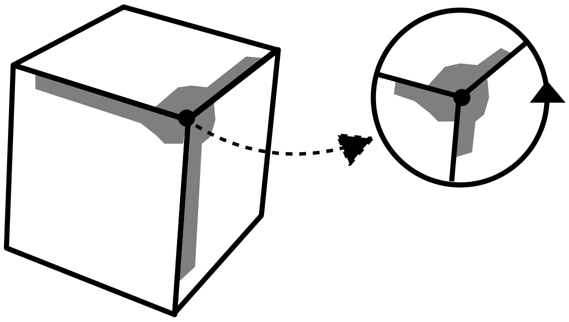


Figure 2.10: *This vertex has a single disk cycle made up of three edge-uses. The edge-uses are depicted as shaded regions on the face_uses that they bound (figure adapted from Gursoz et al).*

Rock and Wozny [61] designed a simple topological data structure specifically for 2-manifold STL input. Each triangular face points to its three vertices, its three adjacent faces, and the three edges shared with the adjacent faces. Edges point to their two endpoint vertices and the two adjacent faces. Vertices point to a list of the adjacent faces. In the first pass of their algorithm to build this data structure from STL input, they allocate the vertices and faces and fill in the pointers between them. During this stage, they must identify the vertices shared between different triangles. To do so, they incrementally build a balanced binary search tree of the vertex entities they create, searching the binary tree for the vertices of each new triangle read in order to determine if the vertices are new, or if they match up with identical or nearly identical existing vertices. In the second pass of their algorithm, they use the lists of adjacent triangles stored with each vertex to limit the search for adjacent faces for each triangle edge, and fill in the edge/face adjacencies.

2.3 Exchange Formats

2.3.1 STL

The industry de facto standard for exchanging part descriptions for layered manufacturing is the STL format. STL is a boundary representation that consists of a simple list of triangular facets. The vertex coordinates are specified explicitly for each triangle in which the vertex appears. The vertices are enumerated in counter clockwise order as seen from the exterior of the part. In addition, for each triangle, a surface normal that points to the exterior of the part is specified. Figure 2.11 shows an example of an STL file for a cube centered at the origin.

STL files come in two types, ASCII and binary. Although the ASCII version uses an organization and keyword choice that suggests the possibility of defining non-triangular facets with multiple loops and grouping them into multiple solids, the binary format has no way of capturing this information. The binary format consists of only a header string, the number of triangles to follow, and a list of four 3-D coordinates for each triangular facet, defining the normal and the three vertices.

STL has many shortcomings. It is very low level and verbose, making it unwieldy for data exchange. It is redundant, both in repeating the coordinates of shared vertices in each triangle in which they appear, and in the specification of where the exterior of the part lies (the surface normal, as well as the ordering of the vertices, gives this information). This redundancy not only makes files unnecessarily verbose, it also allows for inconsistency if the two methods of specifying the exterior do not agree. There are no rules associated with STL for resolving such inconsistencies if they arise. Because the vertices are not shared between triangles, gaps can be introduced between vertices that should be coincident. With no information about topology or connectivity included, it is impossible to communicate the designer's original intent. As a strictly faceted, triangular representation, curved surfaces can only be approximated, and any approximation must make process dependent assumptions about the resolution of the fabrication method. Units are unspecified in STL, leading some manufacturers to guess the intended units based on the bounding box of the part compared to the machine build volume. There is no way to specify solid or surface properties. And because there is no version specification in the format, it is difficult to update it.

2.3.2 The ACIS .sat Format

Another popular exchange format is the .sat save file format [69] used by the ACIS geometric modeling kernel. The .sat format is closely tied to ACIS's internal topological

```
solid ascii
facet normal 0.000000 1.000000 0.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex 1.000000 1.000000 -1.000000
vertex -1.000000 1.000000 -1.000000
endloop
endfacet
facet normal 0.000000 1.000000 0.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex -1.000000 1.000000 -1.000000
vertex -1.000000 1.000000 1.000000
endloop
endfacet
facet normal 0.000000 0.000000 1.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex -1.000000 1.000000 1.000000
vertex -1.000000 -1.000000 1.000000
endloop
endfacet
facet normal 0.000000 0.000000 1.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex -1.000000 -1.000000 1.000000
vertex 1.000000 -1.000000 1.000000
endloop
endfacet
facet normal 1.000000 0.000000 0.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex 1.000000 -1.000000 1.000000
vertex 1.000000 -1.000000 -1.000000
endloop
endfacet
facet normal 1.000000 0.000000 0.000000
outer loop
vertex 1.000000 1.000000 1.000000
vertex 1.000000 -1.000000 -1.000000
vertex 1.000000 1.000000 -1.000000
endloop
endfacet
facet normal 0.000000 0.000000 -1.000000
outer loop
vertex -1.000000 -1.000000 -1.000000
vertex 1.000000 1.000000 -1.000000
vertex 1.000000 -1.000000 -1.000000
endloop
endfacet
facet normal 0.000000 -1.000000 0.000000
outer loop
vertex -1.000000 -1.000000 -1.000000
vertex 1.000000 -1.000000 -1.000000
vertex 1.000000 -1.000000 1.000000
endloop
endfacet
facet normal 0.000000 -1.000000 0.000000
outer loop
vertex -1.000000 -1.000000 -1.000000
vertex 1.000000 -1.000000 1.000000
vertex -1.000000 -1.000000 1.000000
endloop
endfacet
facet normal -1.000000 0.000000 0.000000
outer loop
vertex -1.000000 -1.000000 -1.000000
vertex -1.000000 -1.000000 1.000000
vertex -1.000000 1.000000 1.000000
endloop
endfacet
facet normal -1.000000 0.000000 0.000000
outer loop
vertex -1.000000 -1.000000 -1.000000
vertex -1.000000 1.000000 1.000000
vertex -1.000000 1.000000 -1.000000
endloop
endfacet
facet normal 0.000000 0.000000 -1.000000
outer loop
vertex -1.000000 -1.000000 -1.000000
vertex -1.000000 1.000000 -1.000000
vertex 1.000000 1.000000 -1.000000
endloop
endfacet
end solid
```

Figure 2.11: *An ASCII STL file for a cube.*

data structure, allowing the kernel to quickly rebuild the data structure from saved files. While it makes sense to use this type of "data structure dump" as an internal save file format for applications which use ACIS, it is inappropriate for an SFF exchange format. An exchange format should be independent of the internal data structure of any one modeler. Furthermore, since the process of exchanging geometric information consists primarily of writing, transmitting, and reading back in data files, an exchange format should be compact.

An internal data structure for a modeler, on the other hand, generally contains redundant information to facilitate real-time analysis and modeling operations; space-time tradeoff considerations will sacrifice a compact data structure for interactivity.

Therefore, it is not surprising that ACIS .sat files are generally even more bloated than STL files. Yet some of the additional information they contain would in fact be useful for SFF applications. For example, the faces that define each connected boundary surface are grouped together into "shells," and the shells that bound a connected piece of solid material are grouped together into "lumps." Surface and solid properties could conceivably be attached to the shells and lumps, respectively. Another improvement over STL is that faces can be embedded in curved surfaces, not just flat planes.

Other information is redundant and not always useful for SFF. For a faceted model, the plane equation of the plane containing each facet and the line equation of the line containing each edge must still be specified explicitly. Each vertex is specified separately from the 3-D point that determines its location. Every edge-use, or "coedge," is transmitted separately from its edge and the loop that contains it. Each entity is assigned an index corresponding to its line number in the ASCII version of the .sat file, and the full radial-edge connectivity of the entities is recorded using these indices. Figure 2.12 shows an excerpt from an ACIS .sat file representing a simple cube.

2.3.3 Alternate SFF Interchange Formats

Several alternate interchange formats have been proposed specifically for SFF. Stroud and Xirouchakis [71] proposed extending an STL file with an extra section at the end that lists the faces in the original CAD model and indicates which triangles in the STL file came from which faces. This information about the designer's intent could then be used to clean up inconsistencies in the generated STL. They do not specify the details of how the original face information should be communicated; they used ACIS in their example implementation, but ideally the original faces would be described in a CAD-system independent manner.

Other proposed exchange formats encode their own full topological data structure as well as geometric information. Rock and Wozny [59] developed the "RPI Format" shortly

```
201 0 1 0
7 Unknown 16 ACIS 2.1 Solaris 24 Wed Jul 30 16:51:37 1997
-1 9.9999999999999995470e-07 1.00000000000000036e-10
body $-1 $1 $-1 $-1 #
lump $-1 $-1 $2 $0 #
shell $-1 $-1 $-1 $3 $-1 $1 #
face $-1 $4 $5 $2 $-1 $6 reversed single #
face $-1 $7 $8 $2 $-1 $9 reversed single #
loop $-1 $-1 $10 $3 #
plane-surface $-1 0 0 5 0 0 -1 -1 0 0 forward_v I I I I #

...

vertex $-1 $18 $62 #
straight-curve $-1 5 -5 5 -1 0 0 I I #
face $-1 $-1 $50 $2 $-1 $63 reversed single #
plane-surface $-1 0 -5 0 0 1 -0 -0 0 1 forward_v I I I I #
coedge $-1 $30 $64 $32 $60 1 $20 $-1 #
coedge $-1 $45 $22 $27 $54 1 $12 $-1 #
coedge $-1 $22 $45 $49 $65 1 $12 $-1 #
coedge $-1 $64 $30 $22 $44 0 $20 $-1 #
edge $-1 $57 $66 $43 $67 forward #
coedge $-1 $42 $41 $23 $46 0 $12 $-1 #
edge $-1 $55 $51 $45 $68 forward #
edge $-1 $52 $35 $33 $69 forward #
coedge $-1 $70 $25 $59 $71 0 $50 $-1 #
coedge $-1 $25 $70 $42 $65 0 $50 $-1 #
loop $-1 $-1 $70 $38 #

...

point $-1 -5 5 -5 #
straight-curve $-1 5 -5 -5 -1 0 0 I I #
point $-1 -5 -5 -5 #
straight-curve $-1 -5 -5 -5 0 1 0 I I #
End-of-ACIS-data
```

Figure 2.12: *An excerpt from an ACIS .sat file for a cube.*

after STL was introduced. A header section specifies the intended manufacturing process, scanning methodology, material, and part name, followed by sections that define the vertices, straight line edges, and triangular faces of the part. Vertices, edges, and faces are stored in indexed array-like lists, allowing triangles that share vertices to reference the same vertex index. In addition to the implicit connectivity information provided by the shared vertex indices, each triangular face explicitly records the indices of the adjacent faces and the edges shared between them. Faces also record outward facing surface normals. Edges record the indices of their endpoints and the adjacent faces. CSG trees can also be built from cuboid, cylinder, cone, sphere and tori primitives using geometric transforms. Like STL, the RPI Format includes redundant surface normal information. In addition, the face connectivity information and the edges are redundant and directly tied to their data structure representation.

Similarly, Jacob et al.'s LMI (Layered Manufacturing Interface) format [32] is organized to match their own topological data structure choice. It is restricted to 2-manifold objects with a single boundary shell. The faceted version contains triangular facets defined by one loop consisting of three edges. Connectivity information is recorded in the edges, which reference their two endpoints and two adjacent faces. Each facet points to a plane and each edge to a line. In the precise version, edges can point to curves and facets to surfaces. Loops can have any number of edges. Connectivity is again recorded in edges, but this time they are divided into half-edges. Each half edge records one adjacent face, the next half-edge in the loop around that face, and the previous half-edge in the loop around the other adjacent face.

The differences between the organization of ACIS .sat, the RPI format, and LMI, all of which include similar topological information, reflect the many variations on topological data structures that can represent that information. Clearly a neutral interchange format should not favor a particular data structure. The fact that connectivity shows up in so many different interchange formats, however, reflects the fact that deriving the connectivity is non-trivial.

Standard algorithms for deriving connectivity information break down when memory is not big enough to hold all the data at once. One obstacle to building a topological data structure efficiently is that each new element created may be connected to elements that appear anywhere in the input, and pointers need to be set up from those elements (assuming they have been created yet) back to the new element. Unfortunately, when the data structure does not all fit in physical memory, this can cause thrashing as the same blocks of data are repeatedly read into memory from disk, modified slightly, and then written out to make room for other blocks, only to be read back in again later. We also need auxiliary data structures, such as the binary balanced search trees Rock and Wozny use, to derive the connectivity information. Binary trees will be inefficient if they do not fit in memory because a single traversal might cause a page fault at each level of the tree. If, instead, we build hash tables to look up the connectivity information, the hash table lookups will also contribute to thrashing, since hash tables by their very nature destroy locality. Because it is extremely slow to read and write information to and from the disk compared to reading and writing main memory, we need to consider our memory access patterns carefully when building topological data structures for large input files. Algorithms that take these considerations into account are called external-memory or out-of-core algorithms.

2.4 Out-of-Core Geometric Algorithms

Numerous out-of-core techniques have been developed for other geometric applications. In the graphics domain, these applications include large building walk-throughs, radiosity, and ray tracing [22, 73, 54]. In the visualization domain, several researchers have addressed out-of-core isosurface extraction [13, 12, 72, 4]; others have looked at visualization of terrain and computational fluid dynamics, including streamlines on meshes [16, 15, 76].

For interactive walk-throughs of large, densely occluded building environments, Teller developed efficient pre-computation visibility algorithms and real-time culling algorithms to speed up hidden surface calculations [74, 75], and Funkhouser implemented a predictive algorithm for adaptively selecting a model level-of-detail and type of rendering algorithm to achieve constant frame rates [22, 23]. For out-of-core radiosity calculation, Teller et al. [73] used both partitioning, to reduce the working set size, and reordering sub-computations, balancing faster convergence against memory coherent access patterns for reduced external I/O. For out-of-core ray tracing, Pharr et al. [54] reorder rendering computations using a voxelized scheduling grid that holds queued rays and a voxelized geometry grid that partitions the geometry. They process rays contained in one scheduling voxel at a time, intersecting the contained rays with the contents of the overlapping geometry voxels, continuing to trace each contained ray and its spawned rays until they leave the scheduling voxel, which causes the rays to be queued in their new scheduling voxels.

For isosurface extraction, Chiang and Silva [13] introduced a new preprocessing algorithm on external memory interval trees to speed up finding the cells intersected by the isosurface. Chiang et al.'s revised algorithm [13, 12] reduces the external memory requirements, partitions the data into spatially coherent meta-cells through a few external sorts, and introduces a more compact version of the out-of-core interval tree data structure. Sulatycke and Ghose describe an out-of-core, parallel isosurface renderer that also uses an interval tree, focusing on reducing seek times as well as the number of disk I/Os, and overlapping rendering computations with disk I/O [72]. Bajaj et al. [4] only index a small set of seed cells that are guaranteed to intersect each connected component of the desired isosurface.

Davis et al. discuss out-of-core terrain visualization [16], addressing the problem of interactive visualization of small subsets of a data set that is much larger than memory. The major issues they address are scheduling and moving between different levels of detail. Cox and Ellworth describe disk storage and paging approaches to support a wide variety of computational fluid dynamics (CFD) visualizations [15]. Noting that most CFD visualizations only use a subset of the total data, they first obtained speedups by using using the Unix operator `mmap()` to support demand paging rather than loading the full data set. They achieved even better performance by instead implementing demand paging with smaller than the default system page size and storing the data in hierarchical cubes rather than flat planes for improved locality of reference. For their data, however, the storage of the hierarchical cubes required substantial padding for alignment, resulting in files up to two times larger. To avoid this "file bloat," they packed the data on disk and unpacked it in memory, which required custom memory management, as did the smaller page sizes. They note that using their custom memory management with the same default page size and flat plane layout as their `mmap()` version uses, performance is often considerably worse. This illustrates the potential performance hit with custom memory management that does not take advantage of the hardware and operating system. Requiring custom memory management also complicates implementation and testing.

Ueng et al. describe an out-of-core algorithm for constructing streamlines for visualization of large, unstructured grid data sets [76]. Their algorithm sorts the data into an octree structure and then processes the data within a group of octree cells that fit in memory. Calculations that require additional octree cells are queued to be performed later, when no more calculations are possible with the data currently in memory. One disadvantage of this approach is data-redundancy, since the same grid element needs to be stored in as many octree cells as it intersects. Another disadvantage is non-uniform data sizes in the octree cells (differing by one or two orders of magnitude) because some areas of the input have greater detail; they allocate memory blocks of different sizes to address this problem, which complicates memory management bookkeeping. As with Cox and Ellsworth, this means they cannot use the operating system's memory management efficiently, and thus they implement their own memory manager, with the drawbacks discussed above.

2.5 Wiener's Algorithm

Although our input is geometric, the problem of updating connectivity pointers in a topological data structure actually has a closer analogy in building object oriented databases (OODBs). In these databases, the presence of inverse relationships in the data means that inserting one object in the database requires the system to update its inverses as well. A related issue is resolving "forward references," where an object includes a pointer to another object that has not yet been created (because it appears later in the input file). Wiener and Naughton have proposed a solution for efficient bulk-loading of OODBs [80] that provides the inspiration for our approach.

Their basic algorithm makes a first pass through the input to assign each object a sequential logical identifier, hashes the object name and records the identifier assignment in a hash table called the ID map, and records any inverse relationships that need to be assigned referring back to the object in a sequential file called the "inverse todo list." The entries in this list will consist of a logical identifier for the object just read, but only the "surrogate" input identifier – the object name – for the object that needs to point to it. After the input is completely read, they use the hash table ID map to translate each surrogate identifier in the inverse todo list into the corresponding logical identifier. Next Wiener does an external sort of the inverse todo list by the (newly-translated) logical identifier of the entity with which the inverse pointer will be stored, so that all updates to the same object will be grouped together, and since the identifiers were assigned in the order of the objects in the input, they will also be in creation order. Finally, on a second pass through the input, coordinated with a sequential read of the sorted, translated inverse todo list, they build the real data structure. They store all inverse relationships with the objects at creation time, and use the correct logical identifiers for their pointers.

The basic algorithm relies on the hash table for the ID map (the only structure that is accessed randomly) fitting into main memory. In their revised partitioned-list version of the algorithm, instead of building a hash table for the ID map immediately, the assignments from surrogate to logical identifier are divided up into hash partitions based on the hash value of the surrogate. This just requires a sequential write of the pair onto the end of the appropriate partition. While reading the input file, they use the same hash function to partition the inverse todo list by the hash value of the surrogate for the object to update. In addition, the forward references in each object are recorded in a separate todo list which they also partition, in this case based on the hash value of the surrogate for the referenced object rather than of the referring object. After the input file has been read, they ensure that each partition of the ID map fits in memory, repartitioning it and its corresponding inverse todo list and todo list partitions if necessary. Then they take one partition at a time, build the hash table for its ID map, and use it to translate the appropriate partition of first the todo list and then the inverse todo list, which are read sequentially and then written sequentially after translation to an update list. This is equivalent to a partitioned hash-join [24]. Next they sort this update list, again using an external merge sort. Finally, they re-read the input file and update list simultaneously to create each object.

2.6 File Repair

A topological data structure is often used to aid in repairing input files which do not describe closed, "water-tight" boundaries. One approach is to repair faceted files by matching up pairs of facet edges, identifying those without matches, and adding triangles to fill the cracks. Bohn and Wozny [11, 10] identify closed curves along unmatched boundaries, and triangulate to fill in one hole at a time. Barequet and Sharir [6] locate unmatched borders and find mates using partial curve matching techniques from computer vision, then triangulate to fill the holes. Both approaches were able to repair real files with thousands to tens of thousands of triangles in under a minute on a SparcStation2, but there is no guarantee that the repaired files will not have self-intersections or that these connections are formed in the "correct" or intended way. In later work Barequet performed with Kumar [5], they first try to close small cracks by merging edges, and then triangulate to fill remaining larger holes. Then they visualize the repairs they have made, allowing the user to over-ride any incorrect decisions made during automatic repair. Running on an unspecified SGI workstation, they were able to automatically repair a 10,000 facet file in 8 seconds.

Another approach is to find and close gaps in the original curved surface representations to produce an STL file without gaps. Dolenc [18] fills gaps by matching trimming curves, discretized with a user-supplied parameter, if they are within a user-supplied tolerance. Sheng and Meier [67] also rely on a user-supplied tolerance and try to merge boundary curves, which they discretize using a smaller tolerance. They also try to prevent intersections between geometry while triangulating the curved surfaces.

Murali and Funkhouser have a completely different approach to file repair that is global instead of local [51]. They build a BSP (binary space-partioning) tree from the input polygons, identify solid regions, and output their boundaries. A BSP has size $\Theta(n^2)$ [53] and requires numerous floating point intersection calculations even for parts of the geometry that were clean to begin with. The algorithm's strength is its ability to produce consistent solid output from almost any polygonal input. It is slow, however; files with only 1-2K polygons took 1-3 minutes to repair on an Indigo2 with a 200MHz processor (one or two orders of magnitude slower than the STL repair results cited above for runs on SparcStation2s with slower processors). The global nature of this approach presents a challenge to efficiently scaling it to work with the larger files that are typical in CAD/CAM environments.

2.7 Slicing

After making any necessary repairs to the input, the next step in processing a file for layered manufacturing is to calculate the closely spaced parallel slices. One of the simplest slicing algorithm for STL files is to intersect all triangles with each z-plane and connect the resulting line segments into closed polygons, one slice at a time. This is the approach that is used by Kirschman, et al. [39], who also parallelized this algorithm. In a different context, Mäntylä [47] developed a "splitting algorithm," which uses his half-edge data structure. His algorithm, like ours, is topology based, but since he is making only a single slice through the part, the technique is quite different; there is no coherence to exploit. Another difference is that he builds and outputs solid models (one on each side of the slice plane), not 2.5-D slices. Rock and Wozny [60] also build a topological model before slicing and use it to derive connectivity on a slice by slice basis by marching from one intersected face to the next based on their connectivity. To find multiple contours in a single slice, they search for unvisited edges. They do not exploit topological coherence between slices, but they do use geometric coherence to calculate intersection coordinates incrementally. Dolenc and Mäkelä [19] concentrate on where to slice a part to obtain the greatest accuracy with the fewest number of 2.5-D slices when adaptive slice thicknesses are an option.

Other slicing literature focuses on slicing higher-order descriptions of a part. Luo and Ma [44] slice parametric, C^2 continuous surfaces and fit bi-arcs to the resulting curves. Guduri et al. [25, 26] slice curved primitives prior to computing CSG operations on them. Mani, et al. and Kulkarni and Dutta [45, 43] address adaptive slicing for curved surfaces.

Chapter 3

SIF: A New Solid Interchange Format for SFF

The STL format has remained popular because of its simplicity. On the input side, the SFF machine vendors need only write a simple program to read the single triangle primitive. On the output side, virtually every modeling application that renders a shaded solid model on the screen has already triangulated the model to send it down the graphics pipeline; thus outputting those same triangles in STL format is a trivial change. Unfortunately, triangles that appear to bound a watertight solid when rendered on the screen can easily contain hidden intersections or gaps that are not visible at this low resolution. These problems may not be discovered until the SFF fabricator slices the STL file.

Our goal with SIF was a format that would help to make the part submission process a one-way process, instead of a back-and-forth between designer and manufacturer trying to reach an understanding of what the designer really meant and what the manufacturer was capable of producing. We have tried to keep our format relatively simple, only introducing features that help us to achieve this goal.

In the course of developing the SIF format, we introduced three different dialects. The "pure" SIF dialect, for transmitting solid descriptions for SFF, is formally known as **SIF_SFF**, but we often refer to it simply as SIF. Layered SIF, or **LSIF**, describes the 2.5-D layers that will be built by the SFF machine. We also developed a third dialect, for the Cybercut system [84], to serve as an interchange format between a CAD tool that restricts the user to a design space of parts that can be expressed using Destructive Solid Geometry (DSG) operations [64] and automated process planning tools for 3-axis machining. This dialect, **SIF_DSG**, is outside the scope of this thesis.

3.1 The SIF (SIF_SFF) Format

Like STL, SIF_SFF remains triangle based, but instead of repeating vertex coordinates in each triangle, a vertex array containing each unique vertex is transmitted for each connected shell. The vertices of triangles are specified as indices into this array. The connectivity of the triangles in each shell, specified implicitly via these shared vertices, must correspond to a single, closed, “water-tight” boundary surface, with outward pointing surface normals. Requiring closed boundaries and vertex sharing in the interchange format puts more of a burden on the modeling application that produces it, but this serves to highlight problems and ambiguities on the design end, before the part description has been transmitted to the manufacturer.

One common sources of cracks in triangulated models is instancing. For example, a gear wheel may be described as a central hub containing holes where each tooth will be placed, along with the geometry of a single involute-profile gear tooth, open at the base, instanced with appropriate rotations and transformations for each tooth of the final gear. But where the instanced gear teeth meet up with the holes in the hub, round-off errors in the finite precision floating point operations that implement the rotations and transformations may leave the transformed gear tooth vertices slightly apart from the vertices on the hub that they were meant to coincide with. This type of crack arising from simple round-off errors where connected pieces of geometry meet along boundaries with identical topology can be fixed using the vertex merging approach described in Chapter 6.

Problems also arise when Boolean CSG trees with curved primitives are converted into boundary representations composed of trimmed spline patches. Cracks between these trimmed spline patches cannot be closed simply by vertex merging. Trim curves calculated where patches intersect are approximated, since the intersection is generally of higher order than the original surfaces and is difficult to calculate exactly if the surfaces are both expressed parametrically. This approximate intersection curve is projected back onto the surfaces of the two original patches to “trim” away the part of the patch that is not wanted. Then the two trimmed patches are triangulated separately. This causes gaps in the triangulation along the trim curve, often with different vertex distributions on the two adjacent patches. Models constructed from these patches do not need to be water-tight to render views for manual process planning, or even for finite element analyses such as computational fluid dynamics. Some CSG rendering algorithms, such as the ray tracing algorithm described in Foley et al. [20] or constrained difference aggregates [56], do not even need to calculate the intersections explicitly. Sometimes implicit unions occur where separate shells intersect. Such implicit unions look correct on the screen, but SFF software may process them inconsistently. The QuickSlice 6.2 software [70] that ships with fused deposition modeling (FDM) machines, for example, will treat them as differences in some slices and unions in others, depending on whether the slice contours of the two shells are nested or intersect in a particular slice.

Because it is burdensome for the designer to convert a Boolean description into a water-tight mesh with shared vertices, we allow unevaluated Boolean expressions in SIF. We still require that the leaves of the Boolean tree be closed shells with outward facing surface normals, so that the result of applying the Booleans will be finite and unambiguous. Transmitting unevaluated Booleans puts the burden of evaluating them on the manufacturer, but the manufacturer can slice each shell separately and only evaluate the Booleans in 2-D, a far easier operation than resolving 3-D Booleans. The 2-D Booleans can be evaluated efficiently using readily available OpenGL software, often accelerated by inexpensive graphics cards. Booleans are implemented in OpenGL by using the polygon tessellator and setting the winding number rules appropriately [81]. The OpenGL tesselator can return either filled polygons, which would be appropriate input for a raster scan technology such as 3-D printing, or the polygons’ oriented boundary contours, which would be appropriate input for a calligraphic scan technology such as fused deposition modeling.

Earlier versions of SIF_SFF included tensor product spline patches, allowing curved surfaces to be transmitted in a resolution-independent form. We have removed them for two reasons: they did not generally describe water-tight boundaries, and they required a large software infrastructure to represent and process this information on the manufacturing side. Even after we had implemented this infrastructure, the effort to maintain it was considerable. We believe that by keeping SIF simple, there is a much better chance that it will actually be accepted by the SFF machine manufacturers.

We also believe that as more mechanical engineers discover subdivision surfaces, they will replace spline patches as the representation of choice for curved surface geometry. A subdivision surface is defined by a polyhedral control mesh and a subdivision scheme that specifies how to divide each polygon into sub-polygons and adjust their vertices at each iteration. The scheme is designed so that the limit surface, obtained by applying the subdivision rule an infinite number of times, will be smooth. Sharp corners and fillets can be defined by tagging edges of the original control mesh to control the amount of smoothing around that edge. Schemes based on triangular meshes simplify defining parts of arbitrary topology. The advantage of subdivision surfaces for SFF is that smooth parts of arbitrary topology can be defined by a simple control mesh and still be guaranteed to be water-tight at each level of refinement. Triangular control meshes can be specified using the exact same syntax as for SIF_SFF shells.

Table 3.1 contains the context-free grammar for the current version of SIF_SFF, version 2.0. We specify the grammar by listing its production rules, with the start symbol ⟨part⟩ at the top of the table. Non-terminals appear in ⟨angle brackets⟩ and terminals appear in **bold**. We use {curly braces} for grouping subexpressions and [square brackets] to denote optional subexpressions. Comments describing some of the fields appear in *italics*. The symbol | indicates "or," + indicates "one or more," and * indicates "zero or more." Table 3.2 contains a summary of the semantics of the tokens. A sample SIF_SFF file is shown in Table 3.3.

All dialects of SIF use a simple LISP-like syntax, enclosing each construct in parentheses and following each open parenthesis by a keyword. This syntax allows for limited forward compatibility, since a parser that encounters a keyword it does not recognize can ignore everything up to the matching parenthesis and then continue parsing with a warning message about the construct it skipped.

All SIF files start by specifying the dialect, a major version number and minor release number, and the units for the file. There are no default units; they must be specified explicitly. For a SIF_SFF part, the file also optionally specifies the desired accuracy for the

<table>
<tr><th>Nonterminal</th><th></th><th>Production Rule</th></tr>
<tr><td>⟨part⟩</td><td>→</td><td>(SIF_SFF version:⟨unsigned_int⟩
release:⟨unsigned_int⟩ ⟨units⟩ ⟨accuracy⟩
⟨constellation⟩+)</td></tr>
<tr><td>⟨units⟩</td><td>→</td><td>(units ⟨units_setting⟩)</td></tr>
<tr><td>⟨units_setting⟩</td><td>→</td><td>mm | inches</td></tr>
<tr><td>⟨accuracy⟩</td><td>→</td><td>[(desired_accuracy ⟨float⟩)]</td></tr>
<tr><td>⟨constellation⟩</td><td>→</td><td>(constellation ⟨solid⟩+)</td></tr>
<tr><td>⟨solid⟩</td><td>→</td><td>(solid ⟨volume_property⟩* ⟨shell_set⟩+)</td></tr>
<tr><td>⟨shell_set⟩</td><td>→</td><td>⟨shell⟩ | ⟨boolean⟩</td></tr>
<tr><td>⟨shell⟩</td><td>→</td><td>(shell ⟨vertex_array⟩ ⟨open_2d_list⟩)</td></tr>
<tr><td>⟨vertex_array⟩</td><td>→</td><td>(vertices number:⟨unsigned_int⟩ ⟨vertex⟩+)</td></tr>
<tr><td>⟨vertex⟩</td><td>→</td><td>(v x:⟨float⟩ y:⟨float⟩ z:⟨float⟩)</td></tr>
<tr><td>⟨open_2d_list⟩</td><td>→</td><td>(triangles number:⟨unsigned_int⟩ ⟨open_2d⟩+)</td></tr>
<tr><td>⟨open_2d⟩</td><td>→</td><td>⟨triangle⟩ | ⟨surface⟩</td></tr>
<tr><td>⟨triangle⟩</td><td>→</td><td>(t vertex index:⟨unsigned_int⟩
vertex index:⟨unsigned_int⟩
vertex index:⟨unsigned_int⟩)</td></tr>
<tr><td>⟨surface⟩</td><td>→</td><td>(surface ⟨surface_property⟩* ⟨open_2d⟩+)</td></tr>
<tr><td>⟨boolean⟩</td><td>→</td><td>⟨intersection⟩ | ⟨union⟩ | ⟨difference⟩</td></tr>
<tr><td>⟨union⟩</td><td>→</td><td>(union ⟨shell_set⟩+)</td></tr>
<tr><td>⟨intersection⟩</td><td>→</td><td>(intersection ⟨shell_set⟩+)</td></tr>
<tr><td>⟨difference⟩</td><td>→</td><td>(difference ⟨shell_set⟩+)</td></tr>
<tr><td>⟨surface_property⟩</td><td>→</td><td>(color ⟨color_setting⟩)
| (user extension)</td></tr>
<tr><td>⟨volume_property⟩</td><td>→</td><td>(color ⟨color_setting⟩)
| (user extension)</td></tr>
<tr><td>⟨color_setting⟩</td><td>→</td><td>(rgb R:⟨float⟩ G:⟨float⟩ B:⟨float⟩)</td></tr>
<tr><td>⟨float⟩</td><td>→</td><td>[-]⟨digit⟩*[.]⟨digit⟩+
| (e mantissa:⟨int⟩ exponent:⟨int⟩)</td></tr>
<tr><td>⟨unsigned_int⟩</td><td>→</td><td>⟨digit⟩+</td></tr>
<tr><td>⟨int⟩</td><td>→</td><td>[-]⟨digit⟩+</td></tr>
<tr><td>⟨digit⟩</td><td>→</td><td>0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9</td></tr>
</table>

Table 3.1: *The context-free grammar for SIF 2.0.*

part. This field specifies the maximum acceptable deviation from the geometry described in the file when the part is manufactured on a real world, finite precision machine. This information is also useful input for a networked brokering service which may advise a designer on an appropriate SFF manufacturing technology for a part, in the spirit of

SIF tokens and lexical conventions	
{" ", "\t", "\n"}	White space is used as separators in SIF
(* . . . *)	Multiline comments; can be nested
# . . . \n	Single line comments; can be used inside multiline comments
( )	Delimit expression as in LISP
color	A volume or surface property
constellation	A group of solids whose positions relative to each other and to the coordinate system should be maintained when the part is built
desired_accuracy	The accuracy with which the designer wants the part manufactured
difference	A Boolean difference operation (returns first operand minus subsequent operands, if any)
e	Exponential (a.k.a. scientific) notation for a value
inches	Sets units to inches
intersection	A Boolean intersection operation
mm	Sets units to millimeters
rgb	Specification of color as an rgb triple
shell	A closed surface with outward pointing surface normals (by the right-hand rule)
SIF_SFF	Header stating that this is a SIF_SFF file
solid	A set of shell(s) enclosing volume(s)
surface	A set of 2-D elements
triangles	A list of triangles (possibly contained in surfaces)
t	A triangle, specified by indices into the 0-indexed vertex list
union	A Boolean union operation
units	Sets the units for the file
vertices	A list of vertices
v	A vertex

Table 3.2: *SIF lexical conventions, tokens and their semantics*

Smith's Manufacturing Advisory Service [68].

The remainder of the file specifies the geometry of the part, with optional constructs for specifying material and/or surface properties. Each connected 2-manifold boundary must be grouped in a separate shell, with the shared vertices for that shell specified at the start of the shell. (If the solid's boundary is not 2-manifold, vertices must be duplicated and referenced as for a pseudo-2-manifold.) For ease of memory allocation on the receiving

```
(* file for a yellow cube centered at the origin *)
(SIF_SFF 2 0
  (units inches)
  (desired_accuracy .05)
  (constellation
    (solid (color (rgb 1 1 0))
      (shell
        (vertices 8
          (v 1 1 1)
          (v 1 1 -1)
          (v -1 1 -1)
          (v -1 1 1)
          (v -1 -1 1)
          (v 1 -1 1)
          (v 1 -1 -1)
          (v -1 -1 -1)
        )
        (triangles 12
          (t 7 1 6)
          (t 6 1 5)
          (t 6 5 4)
          (t 4 5 3)
          (t 4 3 2)
          (t 2 3 1)
          (t 7 2 1)
          (t 5 1 0)
          (t 7 6 4)
          (t 5 0 3)
          (t 4 2 7)
          (t 0 1 3)
        )
      )
    )
  )
)
```

Table 3.3: *Sample SIF file describing a yellow cube.*

end, the number of vertices is specified at the beginning of the vertex block. The vertex block is followed by a triangle block, with the number of triangles similarly specified at the beginning of the block. The triangles treat the vertex block as a zero-indexed array for the purposes of referencing vertices.

In ACIS and other formats based on Weiler's taxonomy, shells that bound a connected region of solid material are grouped into lumps. In such a scheme, a hollow sphere would be described as a single lump with an outer spherical shell and a smaller inner spherical shell with opposite orientation (surface normals pointing inward) defining the hollow center. In earlier versions of SIF_SFF, we specified lumps by an outer shell and any number of optional inner shells, with the syntactic nesting of the parentheses enclosing the shell descriptions mirroring the intended nesting of the shells. We abandoned the explicit specification of nesting because it is another instance of redundant information that could be contradictory: the topological nesting of the shells specified by the nesting of parentheses may disagree with the nesting implied by the actual geometry of the shell boundaries. Thus in SIF_SFF 2.0, a hollow sphere is described as a Boolean difference between two boundary shells; if the shells intersect slightly due to round-off errors, this will still be an unambiguous description. As mentioned above, all shells must be oriented with outward facing surface normals in SIF_SFF; differences must be specified explicitly, not just implied by shell orientation. We also abandoned the specification of individual lumps when we introduced unevaluated Booleans. A Boolean expression or tree may yield zero, one, or more lumps, and the individual lumps cannot be identified until after the Booleans are evaluated.

We have included constructs for specifying solid and surface properties in SIF_SFF, even though current systems make no use of this information, because we believe that the capabilities of commercial systems in the near future will require this information. Already, high-end FDM machines have two nozzles for depositing a different build material and support material in the same layer; additional nozzles would allow different materials to be used in the part as well as the support. Z-corporation is currently developing new 3-D printers (which use ink-jet printing technology to deposit binder) that use four-color printing technology to make multi-colored parts. Research in controlling the surface texture of parts produced with 3-D printing is also underway [35].

In SIF_SFF 2.0, surface properties can be associated with sets of triangles within a shell. The set of surface properties is extensible beyond the immediately foreseeable ability to specify color or finish quality. When shells are combined via Boolean operations, the individual surface pieces retain their original surface properties. Thus a part with a rough finish overall except for some finely finished screw holes can be described by subtracting a screw hole geometry specified with a finer surface finish. In the case of an ambiguity where shells with conflicting surface properties have overlapping faces, the last one read in will over-ride earlier specifications. Some surface properties, such as color and surface finish, can be considered orthogonal: color and surface finish do not conflict with each other and can both be defined for the same surface. Manufacturers who support surface properties will have to specify how they interact.

Solid volume properties can be assigned to a collection of any number of shells and/or Boolean trees grouped together with the keyword solid. If a Boolean operation results in different lumps of material that should have different solid volume properties, the output must use intersections to separate them into distinct Boolean trees inside their own solid statements. Solid properties can only be assigned above the level of Booleans; therefore, it is impossible to perform explicit Boolean unions or intersections on geometry with different solid properties. Again, manufacturers will have to define how different solid properties, such as material and density, interact and whether they can be assigned to the same piece of geometry.

Above the level where solid volume properties are defined, geometry is grouped into one or more constellations. The constellation statement indicates that the geometric relationships between the contained objects is fixed. For example, within a constellation, touching lumps with different solid volume properties will fuse, and interlocking chain links will be constructed in the exact relationship specified, guaranteeing that they will interlock. Implicit unions can still appear in SIF_SFF files if lumps defined by different Boolean trees or non-leaf shells in the same constellation overlap. These are treated like explicit unions; if conflicting solid volume properties are defined, again the last one read in should over-ride previous specifications.

Each constellation has its own coordinate system. If the contents of different constellations overlap, the fabricator is expected to separate them. More advanced tools are free to move the separate constellations, whether they overlap or not, anywhere in the build volume. This allows an intelligent process planner to pack parts into the build volume more efficiently, rotating or translating individual constellations for faster or more accurate builds.

3.2 The LSIF Format

Layered SIF, or LSIF, provides a manufacturer-independent format for specifying SFF slice data. Table 3.4 contains the context-free grammar for LSIF version 2.0, with the start symbol $\langle$part_layers$\rangle$ at the top of the table. Table 3.5 contains a summary of the semantics of the five new tokens that appear in LSIF but not SIF_SFF. A sample LSIF file is shown in Table 3.6.

All LSIF files must specify a layer thickness before the actual layers are specified. For SFF processes that are capable of variable layer thicknesses, the global value can be overwritten on a layer-by-layer basis. The layers are listed sequentially from bottom to top of the part.

The geometry of individual layers is defined by closed 2-D contours and/or Boolean combinations of contours. We refer to the subset of LSIF without Booleans as *resolved* LSIF, since it is derived from a general LSIF by resolving the Boolean operations. In a resolved LSIF file or portion of a file, any polygon in the layer that has holes must specify the nesting of its outer contour and inner hole contour(s) inside a "nested" statement, listing first the outer contour and then any inner hole contours. The outer contour's vertices must be specified in counter-clockwise order and the hole contours vertices in clockwise order, as viewed from above the layer. Furthermore, if there are islands within the holes, their nesting must be specified as well by enclosing the hole contour and its islands in another nested statement, and so on through any additional levels of contour nesting. We require this complete nesting information to ensure that translation from resolved LSIF to the layer formats used by specific manufacturers will be trivial, because some manufacturer's formats require it (the SSL slice format used by FDM machines, for one).

For ease of parsing, properties such as color that define the area properties of a polygon can be attached to any contour, but are only semantically meaningful for the counter-clockwise outer contours, since a hole has no area. If an area property is specified for a

Nonterminal		Production Rule
⟨part_layers⟩	→	**(LSIF** *version:*⟨unsigned_int⟩ *release:*⟨unsigned_int⟩ ⟨units⟩ ⟨accuracy⟩ ⟨thickness⟩ ⟨layer⟩+ **)**
⟨units⟩	→	(**units** ⟨units_setting⟩)
⟨units_setting⟩	→	**mm** $\mid$ **inches**
⟨accuracy⟩	→	[**(desired_accuracy** ⟨float⟩ **)**]
⟨thickness⟩	→	**(thickness** ⟨float⟩**)**
⟨layer⟩	→	**(layer** [⟨thickness⟩] ⟨contour_set⟩* **)**
⟨contour_set⟩	→	⟨resolved_contour_set⟩ $\mid$ ⟨boolean⟩
⟨resolved_contour_set⟩	→	⟨nested⟩ $\mid$ ⟨contour⟩
⟨nested⟩	→	**(nested** ⟨area_property⟩* ⟨contour⟩ ⟨resolved_contour_set⟩* **)**
⟨contour⟩	→	**(contour** ⟨area_property⟩* ⟨vertex_list⟩ **)**
⟨area_property⟩	→	**(color** ⟨color_setting⟩ **)** $\mid$ *(user extension)*
⟨vertex_list⟩	→	⟨vertex⟩*
⟨vertex⟩	→	**(v** *x:*⟨float⟩ *y:* ⟨float⟩ **)**
⟨boolean⟩	→	⟨intersection⟩ $\mid$ ⟨union⟩ $\mid$ ⟨difference⟩
⟨intersection⟩	→	**(intersection** ⟨contour_set⟩+ **)**
⟨union⟩	→	**(union** ⟨contour_set⟩+ **)**
⟨difference⟩	→	**(difference** ⟨contour_set⟩+ **)**
⟨color_setting⟩	→	**(rgb** *R:*⟨float⟩ *G:*⟨float⟩ *B:*⟨float⟩ **)**
⟨float⟩	→	[**-**]⟨digit⟩*[**.**]⟨digit⟩+ $\mid$ **(e** *mantissa:*⟨int⟩ *exponent:*⟨int⟩ **)**
⟨unsigned_int⟩	→	⟨digit⟩+
⟨int⟩	→	[**-**]⟨digit⟩+
⟨digit⟩	→	**0** $\mid$ **1** $\mid$ **2** $\mid$ **3** $\mid$ **4** $\mid$ **5** $\mid$ **6** $\mid$ **7** $\mid$ **8** $\mid$ **9**

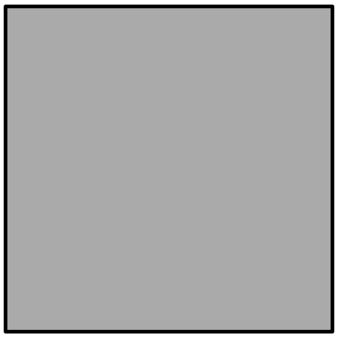
Table 3.4: *The context-free grammar for LSIF 2.0.*

Additional tokens used for LSIF	
contour	A sequential set of vertices defining a closed 1-D path
layer	A planar cross section of the part
LSIF	Header stating that this is an LSIF file
nested	A set of contours consisting of a single outer contour containing one or more inner contours or nested statements with opposite contour orientation
thickness	Sets the layer thickness

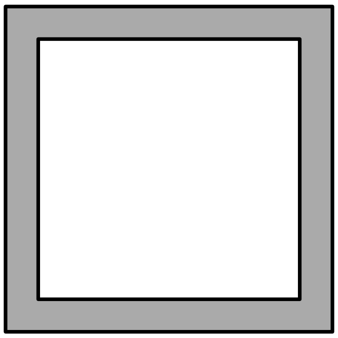
Table 3.5: *Additional tokens used for LSIF. Other tokens and lexical conventions are the same as for SIF.*

clockwise hole contour, it is ignored, even if its container did not specify an area property.

```
(* file for shallow open box *)
(LSIF 2 0
  (units inches)
  (desired_accuracy .05)
  (thickness .01)
  (layer
   (contour
    (v 1 1)
    (v -1 1)
    (v -1 -1)
    (v 1 -1)
   )
  )
  (layer
   (nested
    (contour
     (v 1 1)
     (v -1 1)
     (v -1 -1)
     (v 1 -1)
    )
    (contour
     (v .8 -.8)
     (v -.8 -.8)
     (v -.8 .8)
     (v .8 .8)
    )
   )
  )
)
```



first layer



second layer

Table 3.6: *Sample LSIF file describing a very shallow open box.*

Chapter 4

LEDS: A Topological Data Structure for Layered Manufacturing

Weiler's radial edge data structure or Gursoz's Noodles system record a fairly complete and unambiguous description of part topology, but they include overhead that is not needed for layered manufacturing part processing. Our goal was to design a general data structure that includes topological adjacency information but is also compact so that we can fit it in memory for the biggest files possible. We continually confronted tradeoffs between compactness on the one hand and the need for generality and supporting a wide variety of topological queries efficiently on the other hand. We discuss the choices we made in addressing these tradeoffs below.

4.1 The Loop Edge Data Structure

We call our variant of Weiler's radial edge structure the loop edge data structure (LEDS). Recall that Weiler represented undirected loops and faces separately from directed instances of same (the face-uses and loop-uses), allowing a face, for example, to be referenced from both sides (i.e. in both orientations) where it forms a membrane between cells. We only represent the actual directed face-uses and loop-uses, since a single face or loop is unlikely to be used more than once in SFF file descriptions; therefore, storing both the directed and undirected versions is not worth the overhead. For simplicity, we will refer to directed face-uses and loop-uses as faces and loops in the rest of this thesis.

The three main entities that we store explicitly in the LEDS are faces, edge-uses, and vertices. The queries that we want to support with our data structure, and answer in constant time or time linearly proportional to the size of the output set, include:

- What is the outer boundary of a given face (what are its edge-uses and vertices in order)?
- What are the inner holes of a given face (what are their edge-uses and vertices in order)?
- What vertices define a given edge-use?
- What edge-uses are incident to a given vertex?
- What faces are adjacent to a given edge (and in what order)?
- What are all of the vertices, edges, and faces of the geometry?

Each face (see Table 4.1) is defined by one counter-clockwise, outer loop and a (possibly empty) list of clockwise, inner hole loops. (For triangulated input, of course, we will have no inner hole loops.) Loops are not stored explicitly in the LEDS, however. Each edge-use stores a pointer to the next edge-use in its loop; we follow these pointers to traverse the loop. In a face, we store a reference to a loop as a pointer to an arbitrarily chosen edge-use in that loop. The final field in a LEDS face (as well as in LEDS edge-uses and vertices) is a *void** pointer that applications using the LEDS library can use for any additional information they need to store.

FACE	
EDGE-USE *	First Outer Loop Edge-Use
List<EDGE-USE *>	Inner Loop Edge-Use List
VOID *	Extra

Table 4.1: *The member data for a LEDS face.*

Each edge-use in the loop points back to the face whose boundary it helps to define (see Table 4.2). To make edge-uses compact, like Weiler we have chosen to store only one vertex pointer with each edge-use, a pointer to the root vertex (the vertex from which the edge-use is directed away). The vertex on the other end can be found by following the pointer to the next edge-use in the loop and getting its root vertex. While we represent each edge-use explicitly, the abstract, undirected edges are represented implicitly by circular lists of edge-uses sharing the same endpoints, linked by "sibling edge-use" pointers stored with each edge-use. The circular list of siblings is initially constructed in arbitrary order. (We have implemented radial sorting as a separate function so that only those applications that use this information will incur the overhead of sorting.) To find all of the faces incident to an edge, we can traverse the list of sibling edge-uses, each of which points to a face incident to the edge.

EDGE-USE	
FACE *	Face
VERTEX *	Root Vertex
EDGE-USE *	Next-In-Loop Edge-Use
EDGE-USE *	Sibling Edge-Use
EDGE-USE *	Next Vertex Edge-Use
VOID *	Extra

Table 4.2: *The member data for a LEDS edge-use. The Next Vertex Edge-Use field is explained below.*

We also want to be able to quickly find all of the edge-uses incident to a given vertex, even at non-manifold vertices. For manifold vertices, this could be accomplished by storing a single incident edge-use with the vertex, and then iterating through the edge-use's sibling's next-in-loop edge-uses (see Figure 4.1), since these pointers form a circular list of all the incident edge-uses at manifold vertices. But for non-manifold vertices, we could miss some incident edge-uses with this method. For example, at vertices with multiple disk cycles, such as where two cones meet at their tips, we would only find the edge-uses on one disk cycle, or for input geometry with cracks or holes, there would be gaps in the circular list due to the missing siblings (see Figure 4.2).

We rejected several potential solutions for storing the vertex to edge-use connectivity information for such non-manifold vertices. We could have stored a separate variable length list of all the incident edge-uses with the vertex, but we wanted constant space storage for each vertex (as well as for each edge-use). Constant space storage is important

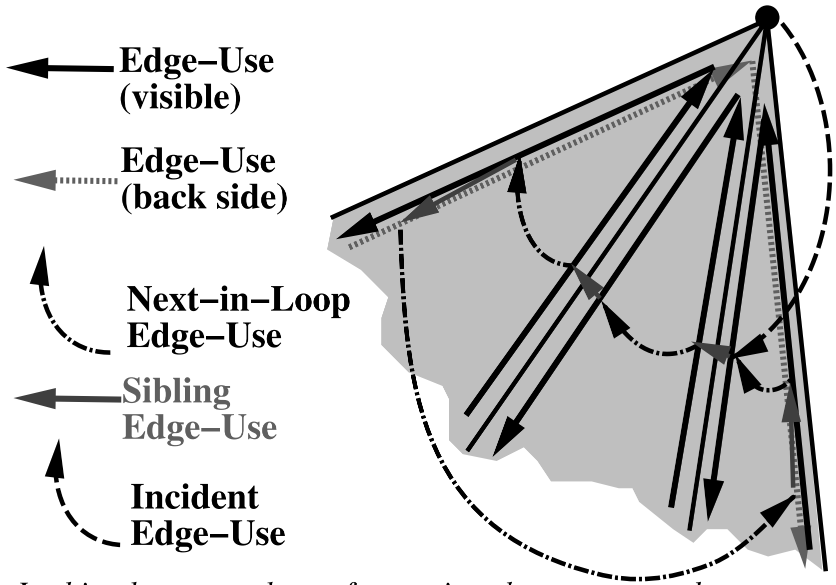


Figure 4.1: *In this alternate scheme for storing the vertex to edge-use connectivity at manifold vertices, the top vertex points to one of the incident edge-uses. From this edge-use, the others can be reached by alternately following their sibling and next-in-loop edge-use pointers.*

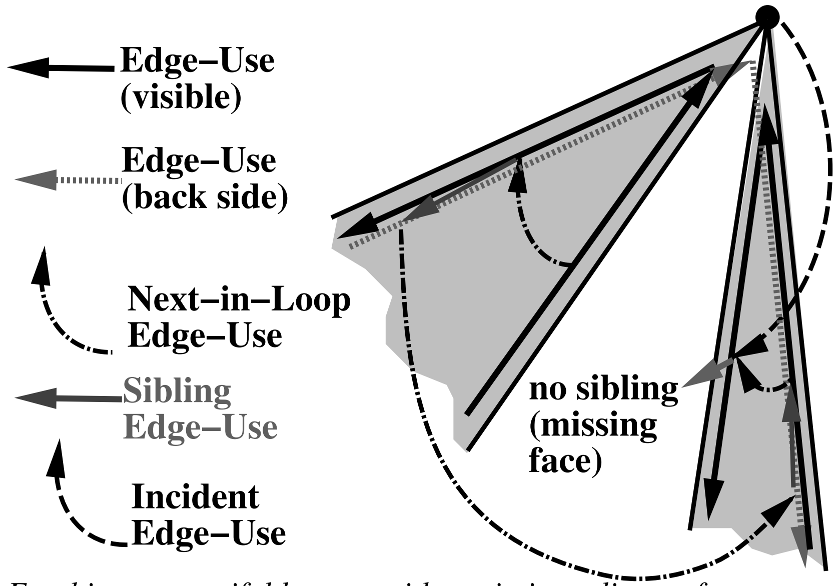


Figure 4.2: *For this non-manifold vertex with a missing adjacent face, we cannot access all of the incident edge-uses via sibling and next-in-loop edge-use pointers as illustrated in Figure 4.1, because part of the chain of faces is missing.*

for allocating memory efficiently; it allows us to pre-allocate storage in arrays. Another potential solution would be to store a list of one edge-use per disk cycle (or a list of disk cycles) with the vertex; this representation would have variable length lists only at non-manifold vertices. To build such a structure incrementally, however, requires separate lists to hold the disconnected pieces until their final connectivity is determined. Furthermore, for input data with cracks caused by round-off errors, even geometry that is intended to be manifold will require variable length lists until after we run a clean-up application.

Instead, we allocate a Next Vertex Edge-Use field in each edge-use to chain together all of the edge-uses rooted at a vertex into a circular linked list (see Figure 4.3). This circular list is initially constructed in arbitrary order. By storing one of these linked list pointers with each edge-use, we can pre-allocate the linked list storage within the edge-use array. The vertex stores a pointer to any one of these adjacent edge-uses (the First Vertex Edge-Use field in Table 4.3). The combination of these pointers allows us to iterate through all of the vertex's edge-uses efficiently, even at non-manifold vertices.

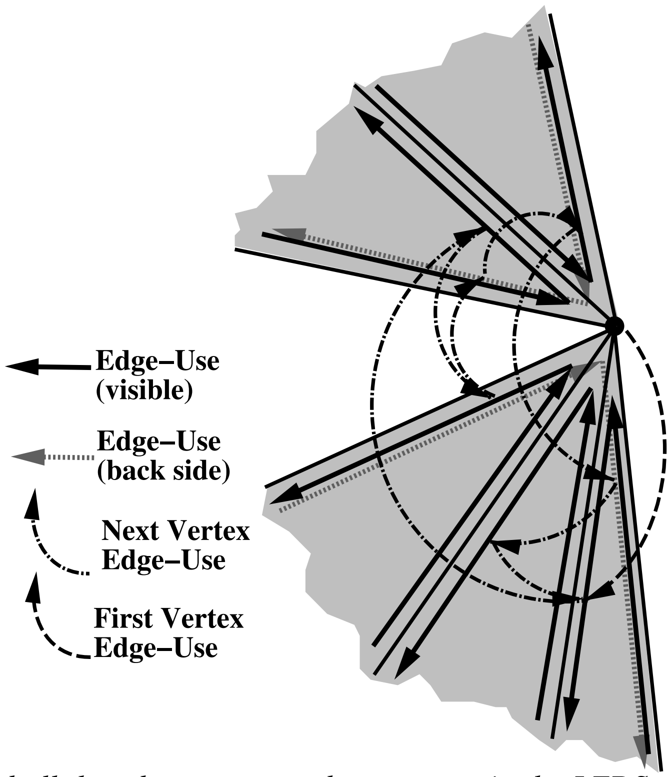


Figure 4.3: *To find all the edge-uses rooted at a vertex in the LEDS, follow the vertex's First Vertex Edge-use pointer to get the first one, and the remaining edge-uses are linked in a circular list via their Next Vertex Edge-use fields.*

The only place the LEDS includes variable length linked list storage is in faces with

VERTEX	
double	X
double	Y
double	Z
double	W
EDGE-USE *	First Vertex Edge-Use
VOID *	Extra

Table 4.3: *The member data for a vertex. We store a homogeneous vertex coordinate, W, for compatibility with our matrix libraries.*

holes. We considered an alternate design that would have made face storage constant size as well by storing loops explicitly and storing with each a pointer to the next loop for the face. But we chose to optimize for the common case of simple polygonal faces, such as triangles. Our scheme eliminates the overhead of representing loops explicitly, while still using constant space storage for faces without holes (which will merely record a null pointer for the list of inner loop edge-uses).

Although our initial allocation of the LEDS entities is in contiguous arrays, some applications that use the structures may add or delete elements. For this reason, we also store separate lists of pointers to all of the vertices, all of the faces, and all of the edge-uses, for operations that need to iterate through them.

Some operations also need to iterate through all of the undirected edges; for example, to check for 2-manifoldness we check whether each edge is used once in each direction. Iterating through all the edge-uses to check that each has a sibling with the opposite orientation, we would check each edge-use sibling pair twice. To support such operations efficiently, in addition to the list of all edge-uses, we maintain a global list of implicit "edges," actually pointers to one arbitrarily chosen sibling edge-use per edge.

4.2 Representing Mixed-Dimension Entities with the LEDS

While the LEDS was designed primarily to represent and support topological queries about the boundary representations of solids, it can also be used to represent mixed-dimension geometry. 3-D solids are represented by the connected, oriented faces that form their boundary. A disconnected 2-D polygon is represented in the same manner as a boundary face. To represent disconnected lines, or any 1-D piecewise linear path through space, we can use the edge-use class to record each segment, but record a null face pointer and a null sibling pointer. Although Kalay claims that "[using] the split edge model for representing lines [is] impossible" because each split edge records only one vertex pointer, and lines have an unequal number of vertices and edges [36], we can get around this problem by recording the final vertex in a zero-length edge use that has a null pointer for its next-in-loop edge-use pointer (see Figure 4.4). While this is not the most efficient way to represent lines, it makes it possible to represent them within the same LEDS framework.

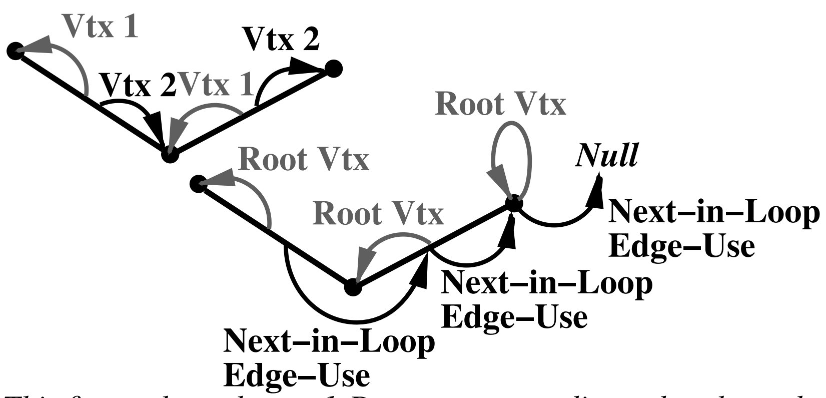


Figure 4.4: *This figure shows how a 1-D, two segment, directed path can be represented in the LEDS by using three edge-uses. The final zero-length edge-use has a null pointer for its next-in-loop edge-use.*

4.3 LEDS Extensions

Several of the tradeoffs we considered while designing the LEDS, between generality on the one hand and compactness and efficiency on the other, have been discussed above. In some cases, rather than making a decision about whether the extra storage and/or computation cost for some piece of information was justified given that not all applications will make use of it, we have included it as an optional feature or extension. This allows the user of the LEDS library to decide which level of enhancements are worthwhile for a particular application.

One such feature of the basic data structure is the option to radially order faces around non-2-manifold edges. Since the faces are already linked by their edge-uses' sibling pointers, sorting them and relinking the pointers does not change storage requirements, but we want to avoid the computation cost if the application does not need this information. The LEDS stores a flag to indicate whether this reordering has been performed for all vertices.

Many solid modeling algorithms assume their input is 2-manifold, but will also work on a pseudo-2-manifold representation of a non-manifold solid. The LEDS includes an option to transform its non-manifold representation of a valid solid into a pseudo-2-manifold representation of the same geometry. First we identify non-manifold edges and pair up their edge-uses, then after all of the non-manifold edges have been divided into pseudo-2-manifold edges, we duplicate any vertices that still have multiple disk cycles. This process is described in detail in Chapter 6.

With our basic data structure, an edge-use stores a pointer to the next edge-use in its loop, but not to the previous edge-use. Many applications do not need to look up previous edge-uses, and for triangulated input, we only need to follow one additional pointer to find the previous edge-use when the previous pointer is not stored explicitly. For those applications that use previous pointers extensively, particularly on geometry with many-sides faces, we have implemented a derived class that stores previous pointers explicitly.

The extension scheme makes ensuring adequate testing more challenging; therefore, we have weighed adding new extensions very carefully. For example, we considered an extension that would store pointers to both endpoints with an edge-use, not just the root vertex. But we only need to follow one additional pointer to the next-in-loop edge-use to get its root vertex to find this other endpoint, regardless of the geometry or topology of the input. For this reason, we have chosen not to add an extension that stores both vertices with edge-uses.

Chapter 5

Algorithms for Building the LEDS Efficiently

In this chapter we describe the two algorithms that we have developed for building the LEDS efficiently from STL input. The first is efficient when the geometry fits in main memory but naive in regards to virtual memory usage, and the second is an out-of-core algorithm that can build the LEDS without thrashing when the geometry is very large.

5.1 Efficient In-Memory LEDS Build

For an in-memory build, we make one pass through the data, constructing and updating the LEDS as we go. For a one-pass algorithm, we cannot allocate the LEDS components in the correct size arrays ahead of time, since ASCII STL contains no information about the number of triangles, vertices, or edges in the file. We use the approach of allocating the arrays in large constant-size buffers of components, filling these buffers, and allocating additional buffers when the existing ones fill up. Another alternative would be to re-allocate an array of twice the size when an array filled and copy over the existing elements, so that we would in fact end up with a contiguous array for each type of component. The amortized cost would be three times the cost of constructing an array of the correct size to begin with, and we could be using up to twice as much storage as necessary [14]. With fixed-size buffers, the size of the buffer places a constant limit on the additional overhead.

We experimented with several different buffer sizes to find a size that balanced the cost savings of allocating elements in large buffers and the cost of initializing many elements that are not used if the final buffer is too large. The results of these experiments, run on an SGI Indigo 2 with a 200MHz MIPS R4400 processor running Irix 6.5, are shown in Figure 5.1. As expected, the worst performance was for a buffer size of one. With the largest buffer sizes, the marginal speedup from allocating fewer, larger buffers was minimal and was more than canceled out by the cost of initializing unused elements. Given this data, we chose 256 elements as our standard buffer size; the exact size is not critical.

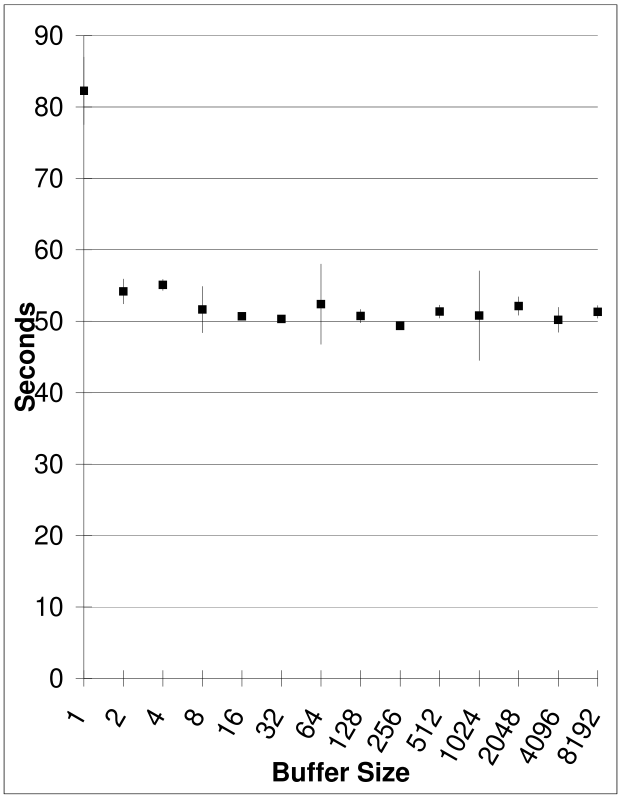


Figure 5.1: *Time, in seconds, to build the LEDS for the ring sculpture test file (pictured in Table 5.1) using different buffer sizes. Each data point shows the mean of five trials and a 90% confidence interval around that mean. A buffer size of one element showed the worst performance. We chose a buffer size of 256 elements since it showed the best performance.*

5.1.1 Hashing

The main task in building a topological data structure is to capture the connectivity of all the surface elements explicitly, which requires matching up (a.k.a. "joining" when performed on a database) all coincident vertex coordinates and edge-uses. To accomplish this, we use hash tables in both our algorithms. Because hashing takes up a significant fraction of the build time, we considered several hashing schemes and functions, and used both theoretical analysis and performance testing to optimize our implementation.

Our in-memory algorithm builds two hash tables simultaneously. The first is for looking up vertices, so that we know whether a coordinate triple in the input file refers to a new vertex that needs to be initialized, or to an existing vertex to which an edge-use needs to be added. For this vertex hash table, we use the *x,y,z* coordinate triple as the input to the hash function, and we store these coordinates (the key) and a pointer to the LEDS vertex (the data) in the hash table. The second is an edge-use hash table, so that we can match up edge-uses with their coincident siblings. To match an edge-use with its sibling(s) which may be oriented in the opposite direction, we want the hash function to return the same value regardless of the order of the edge-use's endpoints. We accomplish this by ordering the endpoints lexicographically, and use this ordered pair as the input to our hash function, which also allows us to use a standard equality check. Instead of using the coordinate triples of the two endpoints as input, however, we use the concatenation of their two addresses in the vertex hash table as the key for the edge-use hash table. Since we will have just hashed the endpoints' coordinate triples to look them up in the vertex hash table, we know their addresses there; using the addresses instead of the coordinates will be far more efficient, since a 32 bit integer is only a third of the size of three 32 bit floats. We save both space (in the hash table) and time (for computing the hash function, ordering the endpoints, and doing equality checks). The time savings is particularly pronounced on large files when the larger hash table does not fit in main memory but the smaller hash table does. The data for the edge hash table is a pointer to the first LEDS edge-use for the vertex.

We have a choice of two ways to resolve collisions in a hash table [14, 41]. In a hash table with chaining, each slot in the hash table contains a pointer to a linked list of all entries that hashed to that value. In a hash table with open addressing, if an entry hashes to a slot that is already full, we look at the next slot in some probe sequence instead. We use open addressing, which provides more slots for the same memory when hash table entries are small, due to the savings of not having to store the linked list pointers.

The simplest probe sequence with open addressing is to always probe the next slot after the one we just examined. This is called linear probing. The disadvantage of this scheme is that runs of contiguous occupied slots ("clusters") tend to grow; the larger the run, the greater the probability that we will hash to a slot in that run, and the greater the expected length of the probe sequence we must follow before we find an open slot at the end of the run (and then we insert it in the open slot, making the run even longer).

We can avoid this clustering problem by using quadratic probing, where each additional probe after a collision is offset from the original slot by $c_1 i + c_2 i^2$, an amount that grows quadratically (carefully choosing our parameters so that we will still always examine each slot in the hash table if we go through the entire probe sequence). With quadratic probing, we will only get clustering for values that initially hash to the same slot, so-called "secondary clustering."

We can get an even better probe sequence, at the cost of slightly longer computation times, with double hashing, where we use a second hash function to determine the offset for additional probes after a collision. Since we use this same offset for all additional probes, the offset must be relatively prime to the size of the hash table to assure that all slots will be probed.

We implemented both quadratic probing and double hashing to compare their performance on our data. For both methods, our hash function $h(x)$ returns a 32-bit integer much larger than m, where m is the size of our hash table. We always choose m to be prime. For quadratic probing, our i^{th} probe after a collision is to position:

$$(h(x)mod(m) + i^2)mod(m).$$

For double hashing, we can derive the second hash function from the first and guarantee a relatively prime offset by using the formula:

$$h_2(x) = 1 + h(x)mod(m - 1).$$

With this secondary hash function, our i^{th} probe after a collision for double hashing is to position:

$$(h(x)mod(m) + i(1 + h(x)mod(m-1)))mod(m).$$

In addition to a probe sequence, we need a hash function. To validate our choice of hash functions experimentally, we gathered statistics on the number of probes we made while building the hash tables. For simplicity of derivation, we compared these numbers to the expected value for the ideal case of "uniform hashing," where every probe sequence is equally likely for every input key. Of course, even if we had an ideal, completely random hash function that was equally likely to choose any position in the hash table, we would still not achieve uniform hashing. For uniform hashing, all $m!$ possible probe sequences are equally likely; with quadratic probing, the remainder of the probe sequence is fixed given the initial hash value, yielding only m different probe sequences, while with double hashing, we still have only m^2 different probe sequences.

For the ideal case of uniform hashing, given a hash table that is already fraction α filled, the expected number of probes to insert a new item will be the following sum:

1 (we must always check at least one position)
$+\alpha$ (probability of making a 2^{nd} probe because the previous position was full)
$+\alpha^2$ (probability of making a 3^{rd} probe because the previous two positions were full)
$+\alpha^3$ (probability of making a 4^{th} probe because the previous three positions were full)
$+...$
$= \frac{1}{1-\alpha}$ [14].

Of these probes, one will be a hit, and the remaining

$$\frac{1}{1-\alpha} - 1 = \frac{\alpha}{1-\alpha}$$

will be misses. For a hash table of size m, after the i^{th} item is inserted, $\alpha = \frac{i}{m}$; therefore the expected number of misses when inserting the i^{th} item will be

$$\frac{\frac{i}{m}}{1-\frac{i}{m}} = \frac{\frac{i}{m}}{\frac{m-i}{m}} = \frac{i}{m-i}$$

After inserting the n^{th} item into the hash table, we will have had n hits and an expected

$$\sum_{i=0}^{n} \frac{i}{m-i}$$

misses.

Once a hash table is very full, its performance degrades drastically, as we must search a longer and longer probe sequence to find an empty slot. Evaluating the expression computed above for various hash tables sizes m between 100 and 1,000,000, we find that when a hash table is filled to 80% capacity, averaging over all the insertions that filled the hash table, the total number of misses will be roughly equal to the total number of hits. After reaching this capacity, we rehash in a new hash table that is twice the size ("rounded" up to the next prime number, of course) to improve the hit rate. In general, a larger hash table will give better performance at the expense of using more memory. We feel that rehashing at 80% full is a reasonable time/space tradeoff.

In order to evaluate our choice of hash functions, we compared this expected value for the number of misses to the actual values we achieved building our hash tables with real data. We used double hashing for these tests so that clustering would be less of an issue. The test file geometries we used are shown in Table 5.1. Hash table performance has high variance: the actual number of misses can vary significantly from the average "expected" number of misses; thus the number of misses while building a single hash table is not very meaningful. We look at the data for a number of hash tables of varying sizes built with varying data, trying to find the point where those that underperform the expected value are roughly balanced by those that outperform the expected value. When a hash function gives these results, it is likely that it is a good approximation to a random hash function.

We want to choose hash functions that are quick to compute and that minimize collisions in the hash table. Integer computations are faster than floating point computations; therefore, our hash function treats the 32 bits that represent each floating point coordinate as a 32 bit integer.[1] As part of satisfying the second goal of minimizing collisions, we use

[1] We cannot use a straight cast to an integer to do this, however, because that merely truncates the decimal portion of the float, which would leave us with identical values for every single vertex in the instance of a part less than one unit in each dimension. While this would make computing the hash value very efficient, we would have maximized rather than minimized collisions because every vertex would get assigned the same value by the hash function! Instead we take a pointer to the float and cast it to a pointer to an int, and dereference that, allowing us to treat the 32 bits that represent the float as a 32 bit integer.

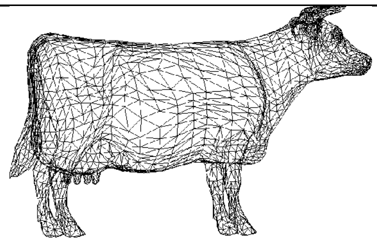
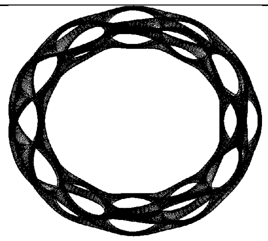
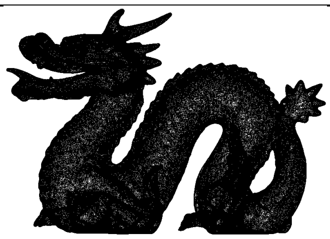
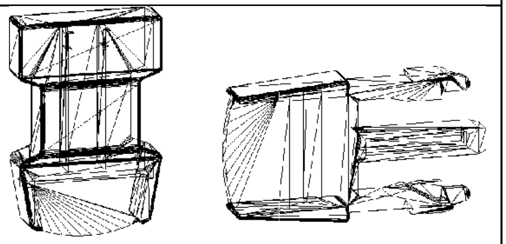
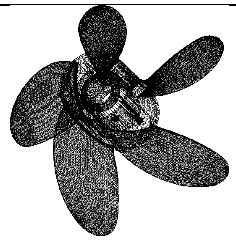
Model Name	Number of Triangles	Picture
Cow	5,804	
Ring Sculpture	107,520	
Dragon	869,898	
Buckle	4,334	
Propeller	96,040	

Table 5.1: *Models we used for testing the hash tables. The cow model was downloaded from Clemson University. The ring sculpture was generated using Séquin's sculpture generator. The dragon model was obtained from Stanford's 3D Scanning Repository. The buckle and propeller models shipped with the fused deposition modeling machine's QuickSlice 6.2 software.*

all three coordinates and both endpoints as input to our hash function, so that we will not get additional vertex collisions for files that have many vertices with the same height, for example, or additional edge-use collisions for files that have vertices of high valence.

We started with hash functions that are very simple to compute and gradually increased their complexity until we got collision behavior that was consistent with using random hash functions. For the edge hash table, we first tried simply adding together the addresses of the two endpoints, E1 and E2, and taking the result modulo the size of the hash table: $h(E1, E2) = (E1 + E2)mod(m)$. We gathered statistics on the misses to hits ratio using this hash function for building different sized hash tables filled to 80% for the cow and ring sculpture input files. The resulting miss ratio was significantly higher than the expected miss ratio, as shown in Figure 5.2. Next, we tried $h(E1, E2) = (3*E1+7*E2)mod(m)$.

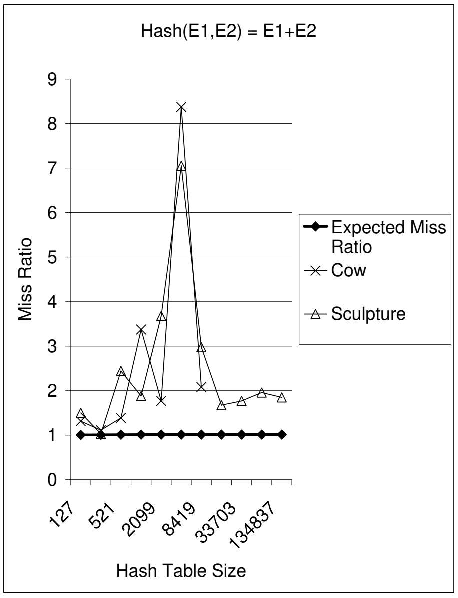


Figure 5.2: *With the edge hash function $h(E1, E2) = (E1 + E2)mod(m)$, the miss ratio for the input models was significantly higher than the expected miss ratio for many hash table sizes, indicating that the hash function was not sufficiently random.*

The miss ratio was much better, but still occasionally much higher than expected, as shown in Figure 5.3. Using the hash function $h(E1, E2) = (31 * E1 + 17 * E2)mod(m)$, we obtained miss ratios that were sometimes higher and sometimes lower than the expected miss ratio, as shown in Figure 5.4. For the largest hash table sizes, the miss ratio was fairly steady and only slightly worse than the expected miss ratio for ideal uniform hashing. Therefore we chose this as our edge hash function.

For the simplest vertex hash function, we treated the three floating point coordinate

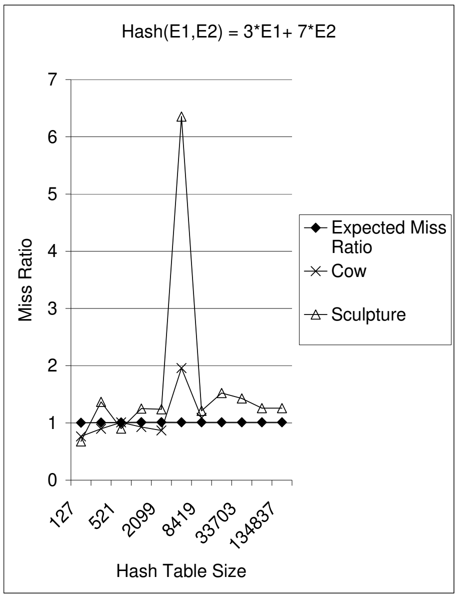


Figure 5.3: *With the edge hash function $h(E1, E2) = (3 * E1 + 7 * E2) mod(m)$, the miss ratio for the input models was occasionally significantly higher than the expected miss ratio for some hash table sizes.*

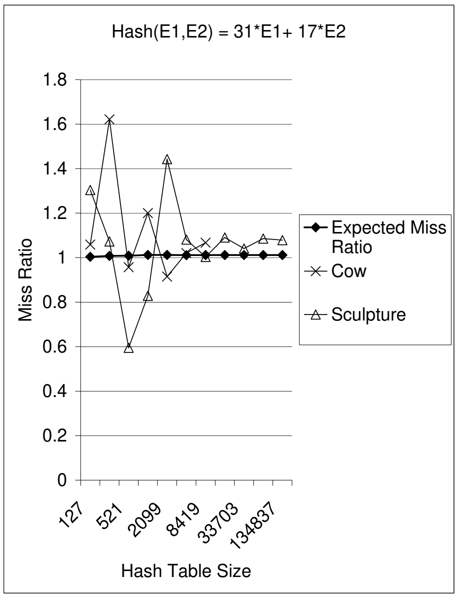


Figure 5.4: *With the edge hash function $h(E1, E2) = (31 * E1 + 17 * E2) mod(m)$, the miss ratios were sometimes higher and sometimes lower than the expected miss ratio for a random hash function, indicating a better approximation to random hashing.*

values as integers and added them together, modulo the size of the hash table. We obtained miss ratios both higher and lower than the expected value with this hash function. Again,

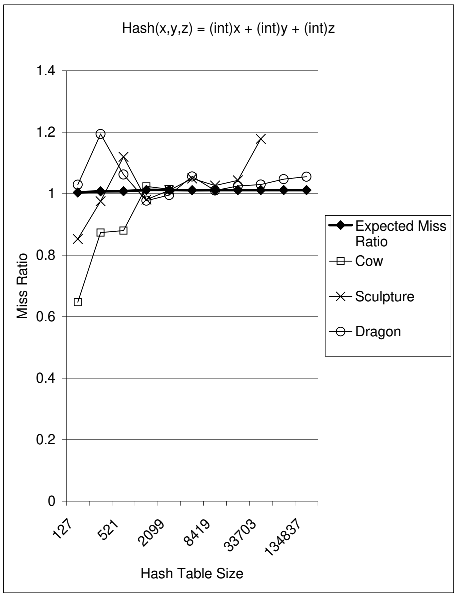


Figure 5.5: *With the vertex hash function $h(x, y, z) = x + y + z$, the miss ratios were sometimes higher and sometimes lower than the expected miss ratio for a random hash function, indicating a good approximation to random hashing.*

for the largest hash table sizes, the miss ratio was fairly steady and only slightly worse than the expected miss ratio for ideal uniform hashing, as shown in Figure 5.5. Therefore we used this simple function for vertex hashing.

Next we compared running times for quadratic probing and double hashing using these optimized hash functions. We found that quadratic probing was actually slightly faster overall with our implementation, despite the fact that it resulted in more collisions (6% faster for the buckle part, 5% faster for the cow part, and 4% faster for the propeller part, averaged over five trials). This is because it is much cheaper to compute the offsets in the quadratic series than it is to perform the expensive "modulo" operation needed to compute the first offset for double hashing. Therefore we used the quadratic probing implementation.

5.1.2 Algorithm for Memory-Resident STL Input

First, we initialize our data structures. We allocate three buffers, one each for the LEDS vertices, edge-uses, and faces. When a buffer fills, we allocate a new buffer. We allocate a small hash table for the vertices which takes a triple of floats as its key and a pointer to a

LEDS vertex as its data. We also allocate a small hash table for the edge-uses which takes a pair of vertex table addresses as its key and a pointer to a LEDS edge-use as its data. When a hash table fills to 80% capacity, we rehash its contents into a new hash table twice as big and free the old hash table.

For each triangle we read in, we take the next face in the face buffer and the next three edge-uses in the edge-use buffer for this triangle. We set the face to point to the first edge-use, and each edge-use to point to the next around the vertex.

For each vertex-use, we look it up in the vertex hash table to see if we have already allocated a vertex for it. If not, we take the next vertex in the vertex buffer for this vertex and record its address in the data field of the hash table entry. We set its coordinates, and set its first edge-use to point to the current edge-use (the edge-use in this triangle rooted at that vertex), and set the current edge-use's next-vertex-edge-use pointer back to itself. If the vertex is already in the vertex hash table, we find its address there. In this case, we look up the vertex's first edge-use, and insert the current edge-use after it (by setting the current edge-use's next-vertex-edge-use pointer to the next-vertex-edge-use pointer stored with the first edge-use, and setting the first edge-use's next-vertex-edge-use pointer to the current edge-use). This will maintain a circular list of all the edge-uses for the vertex, as shown in Figure 5.6. In either case, we also set the edge-use rooted at the vertex to point to the vertex. Note that even if we are reading in a vertex-use for a vertex we have already seen many times, we only need to access or update two existing LEDS elements: the vertex, to find its first edge-use, and that first edge-use, to insert the current edge-use after it in the next-vertex-edge-use circular list.

Then we look up each edge-use in the triangle in the edge hash table (using the lexicographically ordered pair of the addresses of its endpoints). If it is the first edge-use for the edge, we add it to the linked list of all edges and record its address in the data field of the hash table entry. If there is already an edge-use in the hash table for the edge, we check if that edge-use already has a sibling. If so, we set the sibling of the current edge-use to point to the sibling of the edge-use recorded in the hash table; otherwise the current edge-use's sibling is set to the edge-use in the hash table. The edge-use in the hash table then gets its sibling set to the current edge-use in either case. This will form a circular list of all the siblings for the edge when there are two or more edge-uses, as shown in

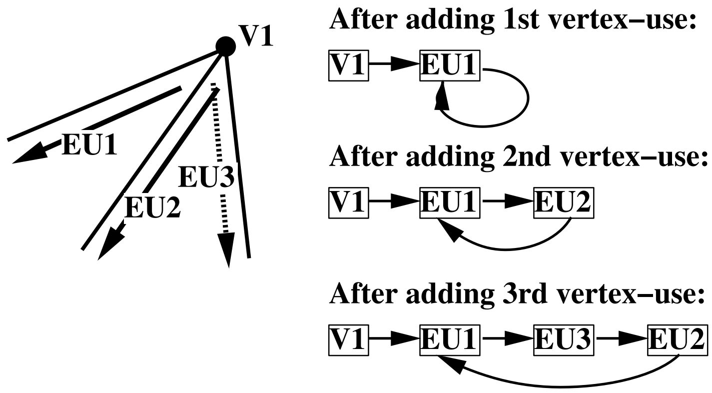


Figure 5.6: *The circular list of next-vertex-edge-uses for the vertex shown on the left is built up as illustrated on the right.*

Figure 5.7. Again, note that we limit the existing LEDS elements that must be accessed or updated: limited to one sibling edge-use for the second edge-use on an edge, or limited to two sibling edge-uses for additional edge-uses at non-manifold edges.

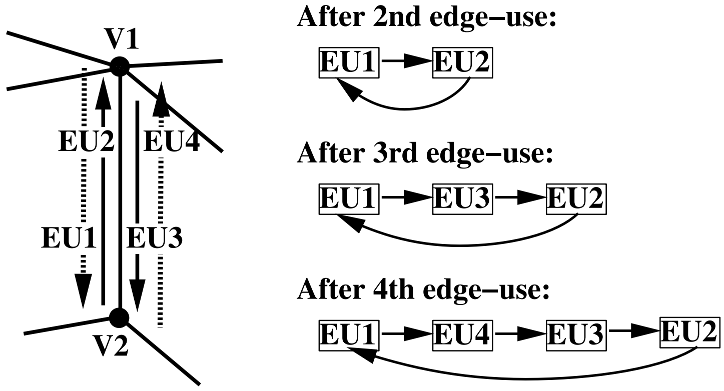


Figure 5.7: *The circular list of edge-use siblings is built as illustrated here.*

When all of the input triangles have been read, we free both hash tables.

5.1.3 Results for Memory-Resident Algorithm

For testing the effects of input size, we took a simple curved knot shape (Fig. 5.8) output from Séquin's sculpture generator [66] and varied the fineness of the tessellation to produce STL files of different sizes. All of the triangles are organized into consecutive triangle strips and output in the order in which they adjoin in the part. As such, they exhibit a great deal of topological coherence. For triangles interior to each strip, two of the triangle's three neighbors will be immediately adjacent to it in the file, and its third neighbor will be in the adjacent strip. Because of this topological coherence in the input, thrashing problems during the LEDS construction should be minimized. Thus, the results should under-estimate the need for our more sophisticated algorithm presented in the next section. To extract the effects of input coherency, we also made another version of each input file that contained the same triangles but in random order.

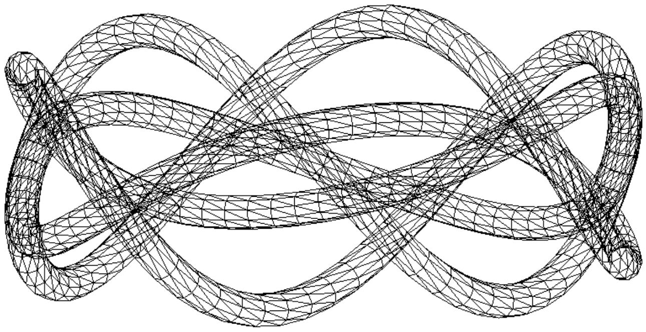


Figure 5.8: *Our canonical test part, the triangulated "knot sculpture." The version pictured here has only 4,800 triangles in order to make the organization of the triangle strips into adjacent stories clearly visible. Versions of the part with more triangles have both more stories and more triangles per story.*

Running on an SGI Indy with one 133 MHz MIPS R4600 processor, 32 MB of RAM, and 540 MB of virtual memory, we were able build the LEDS for STL test files containing 10,000 to 70,000 triangles with no page faults, since both the LEDS and the hash tables fit entirely in RAM. Running time increased proportionately with the input size. When we ran the same tests on the randomly ordered input files, the results were virtually identical up to 70,000 triangles, indicating that locality in the input is not an issue when the LEDS fits in RAM. Above this size, performance starts to slow down and the random and coherent results start to diverge. The results of these tests are shown in Figure 5.9.

For medium sized files, we start to see the effects of the slower access times for virtual memory, as shown in Figure 5.10. For files containing up to 200,000 triangles, the running times continued to grow linearly for coherent input because there was enough room in memory to hold both the hash tables and the active portion of the LEDS. For the randomly

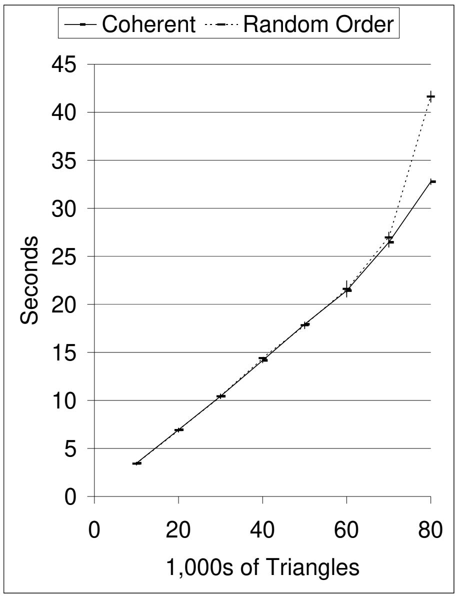


Figure 5.9: *Running times (on an SGI with 32MB of RAM) for the in-memory algorithm on STL files of the knot sculpture test part, tessellated to contain from 10,000 to 80,000 triangles. Each test was run five times, and the error bars (almost too narrow to see for some values) show a 95% confidence interval around the mean results. The running time increases proportionately with the input size up to 70,000 triangles, and is not affected by the order of the input triangles at these sizes. Above 70,000 triangles, the performance slows for the random input.*

ordered files of this size, however, the random accesses to both the LEDS and the hash tables caused thrashing and the performance worsens considerably for the random files.

For larger test parts containing over 400,000 triangles, the active portion of the LEDS no longer fits in memory simultaneously with the hash tables even for the coherent files, and the resulting memory thrashing is reflected in the run times (see Figure 5.11). Run times continue to be worse for the large randomized test files than for the large coherent test files due to the the higher probability of page faulting on each update to the LEDS. For 400,000 triangles, it took fifty times longer to process the random file compared to the coherent file. We had to stop the tests on the larger random files after they had been running for days without completing. Even with coherent input, the million triangle test part still took almost nine hours to process.

We also ran the same tests on a faster machine, a dual processor PC with two Pentium III 700 MHz processors. Using Linux, we booted this machine with only 32 MB RAM so

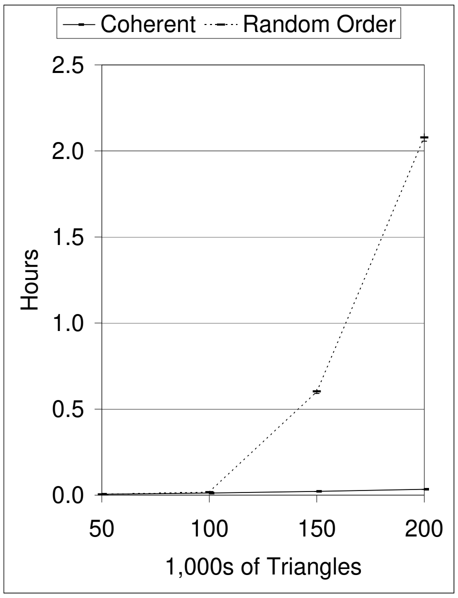


Figure 5.10: *Running times (on an SGI with 32MB of RAM) for the in-memory algorithm on STL files of the knot sculpture test part, tessellated to contain from 50,000 to 200,000 triangles. Each test was run five times, and the error bars, largely invisible due to their narrowness, show a 95% confidence interval around the mean results.*

that we could compare performance using the same amount of memory on both platforms. While the smallest sculpture test parts ran six times faster on this machine, the run times for the largest parts were almost identical (see Figure 5.12). The coherent 1,000,000 triangle test part still takes almost seven hours to process, less than a 25% improvement over the running time on the SGI Indy, and the largest randomized input file we ran on both platforms, with 400,000 triangles, shows less than a 20% improvement on the new machine.

We hypothesized that the reason the running times do not improve for the larger files on the new machine may be that recent improvements to hard drives and file systems have focused on improving sequential access times rather than random access times. To test this hypothesis, we allocated arrays containing 5,000,000 double precision floating point values on both machines (40MB arrays) and timed how long it took to write 5,000,000 entries sequentially compared to how long it took to write 5,000,000 entries to random positions. For the sequential writes, the Pentium III Linux machine was approximately five times faster, but for the random writes, it was approximately 9% slower (see Table 5.2).

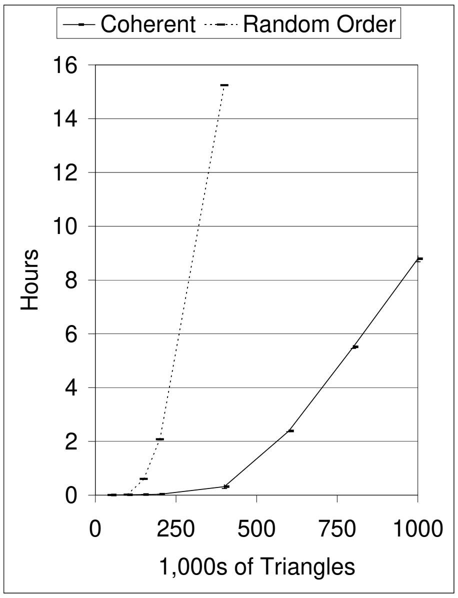


Figure 5.11: *Running times (on an SGI with 32MB of RAM) for the in-memory algorithm on STL files of the same test part, tessellated to contain up to 1,000,000 triangles. Each test was run five times, and the error bars, largely invisible due to their narrowness, show a 95% confidence interval around the mean results. The gap between the running time for the coherent and randomized input continues to widen for the larger files.*

	Sequential Writes	Random Writes
SGI MIPS Irix	0:0:11.1	8:33:35
Pentium III Linux	0:0:02.3	9:17:15

Table 5.2: *Time to write 5,000,000 items to a 40MB array sequentially and randomly on the SGI and Linux platforms with 32MB RAM.*

As described in this section, we have analyzed and optimized this in-memory algorithm extensively, except in respect to virtual memory access patterns. In the next section, we describe our algorithm for avoiding thrashing and the resulting exponential growth in build time when the data (and hash tables) are too big to fit in main memory.

5.2 Out-of-Core LEDS Algorithm

When the LEDS and the hash tables are too big to fit entirely in memory, the naive, in-memory algorithm can become prohibitively slow due to thrashing. One problem is that each new face, vertex, or edge-use that we read in could be connected to elements that have

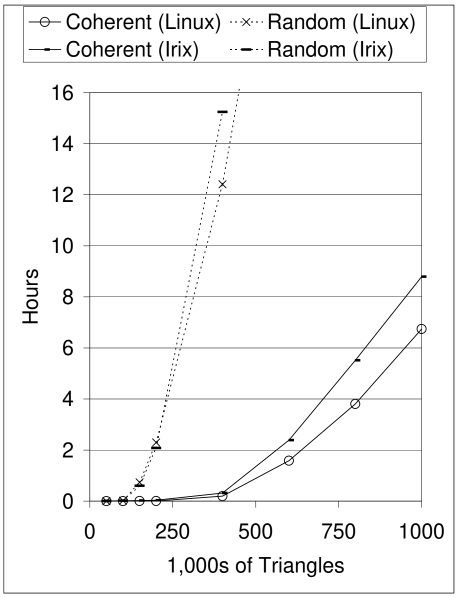


Figure 5.12: *Running times under Linux PC compared to SGI Irix Indy (both with 32MB of RAM) for the in-memory algorithm on the same knot sculpture test part STL files. The running times for the largest files only improve 20 - 25% compared to the times on the older machine.*

already been written out to disk, and those elements will now need to be paged back in to be updated. With multiple updates to the same element separated in time, each of these updates can cause a page fault. Even if the input is extremely coherent, so that updates to the same element are closely spaced in time, we still see thrashing when the vertex and edge hash tables become too large to co-exist in memory. Hash tables by their nature are accessed randomly (assuming the hash function was well chosen!) with no guarantee that the portion of the hash table we access on each look-up will still be in memory.

Our out-of-core algorithm for non-memory-resident data avoids these problems in two ways: by reordering and grouping random hash table accesses, so that we need to build and access only one memory-sized partition of a single larger hash table at a time, and by using external merge-sorts to reorder all other operations to make them sequential reads and writes. Our only out-of-order accesses are within the hash table partitions and during the sorting stage.

In addition to the partitioned hash tables, we have implemented a "dynamic array" class to store large arrays of data that we write and read sequentially but whose exact length we do not know before we begin writing. We allocate this array during the write phase, doubling its size when it becomes full. Its most important member functions, which we refer to in the detailed algorithm description below, are to support sequential accesses: appending an entry to the end of the dynamic array during the write phase, and returning the next entry during the read phase. The class is templated on the array entry type, so that we can use the same class for dynamic arrays containing many different types of objects.

Our algorithm has four stages, described in detail in the sub-sections below. In summary:

We make the only pass through the original input file during stage one. During this stage, we assign sequential edge-use and face IDs (recall that a LEDS face is really an oriented face-use) to each edge use and triangle in the input. We record the topological relationships immediately available from the input, using a separate dynamic array for each type of relationship. These relationships are recorded using the IDs just assigned. We also record the vertex-use coordinates and other information we need to derive the remaining topological relationships. In stage two, we build a partitioned vertex hash table and use it to translate vertex-use coordinates to vertex IDs, and to derive the vertex topological relationships, creating new dynamic arrays to hold them. In stage three, we build a partitioned edge hash table to match edge-uses with their siblings and record these relationships in an additional dynamic array. In stage four, we fill in the actual LEDS entities. First we sort each dynamic array so that the entries appear in the same order as they will be recorded in the final LEDS. Then we read, in parallel, from the front of all the dynamic arrays containing vertex information to create the vertices. Next, the edge-uses and then faces are constructed by reading in parallel from their corresponding arrays. This allows us to write all of the information we need to record in each LEDS entity at once, so that we do not need to go back and modify entities that have already been written out to disk.

Below, we describe these four stages in detail. Figure 5.13 shows a condensed summary of the four stages.

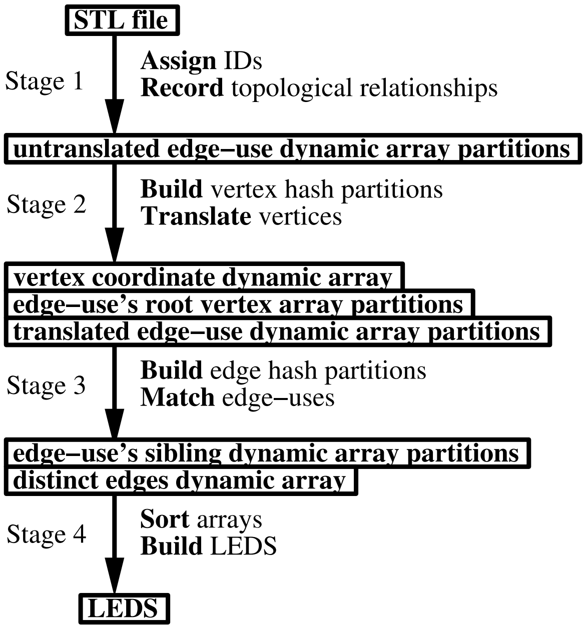


Figure 5.13: *The four main stages of the out-of-core algorithm. The arrays of intermediate data created at each stage are shown in boxes.*

5.2.1 Stage One

Recall that the two tasks in stage one are to assign IDs and record topological relationships in dynamic arrays. We use a separate counter for assigning sequential IDs to each type of LEDS entity (vertex, edge-use, and face) so that we can also use the ID as an array index for that entity's array. (Later, once we know the address for the start of each array, this allows us to translate IDs to pointers without any lookups using simple arithmetic.) For each triangle in the input, we can immediately assign new IDs for a face and its three edge-uses, since each is a unique use. We cannot immediately assign new vertex IDs, however. We do not know if each vertex is being encountered for the first time, in which case we need to assign it a new ID, or if it is a vertex that was already used in a triangle appearing earlier in the file, in which case we should use the ID already assigned to it. Therefore, in stage one, while we record IDs for faces and edge-uses, we record the coordinate triples for the vertex-uses, waiting until we have built a vertex hash table in stage two to assign vertex IDs.

In the general case, we record data in five different dynamic arrays during stage one. Four of them record ID pairs, where the first ID is that of a LEDS entity that will contain a pointer to a LEDS entity with the second ID. These are the "edge-use's face," the "edge-use's next-in-loop edge-use," the "face's outer loop edge-use," and the "face's inner loop edge-use" dynamic arrays. For triangulated input, none of the faces have inner loops; therefore, we clearly do not need this last dynamic array for STL. But for triangulated input we do not need to record these other three dynamic arrays either. The information they would contain can be derived later when we need it (as detailed in the description of stage four below) merely from knowing the total number of triangles and that the face and edge-use IDs are assigned as sequential integers.

The final dynamic array that we always create and fill during stage one will be used for deriving all of the remaining topological relationships. It contains one entry per edge-use, but unlike in the four dynamic arrays described above, each entry is not a pair of IDs that translate directly to a pointer in the final LEDS. Instead an entry contains three pieces of information: the ID of the edge-use, and the coordinates of the edge-use's two endpoints' vertex-uses. We call this the "untranslated edge-use" dynamic array because the vertex-use coordinates need to be translated to vertex IDs before we can interpret them as pointers.

5.2.2 Stage Two

Stage two is the vertex hash table building and translation stage. If the input file is large, this hash table may not fit in memory. If this is the case, we use partitioned hash tables. With partitioned hash tables, we only build one memory-sized piece (a "partition") of a larger hash table at a time. This requires estimating how many hash table partitions we will need and dividing the input into that many data partitions before we process it. We take the hash value of the vertex, modulo the number of partitions, as the index of the data partition in which to store the input. This assures that all of the input corresponding to the same entry in the hash table will be in the same input data partition. Once the data is partitioned, we read one data partition at a time and build its corresponding hash table partition.

We translate the two endpoint vertices in the input in separate steps. For the first translation step, we partition the "untranslated edge-use" dynamic array based on the hash value of the edge-use's first endpoint's coordinates. We try to predict the number of partitions that we will need from the size of the input file so that we can partition the "untranslated edge-use" dynamic array appropriately at creation time in stage one; the random hashing should make the partitions of roughly equal size, so that the hash table for each will fit in memory. If necessary we can re-partition the array when we build the hash tables.

The input to stage two consists of the "untranslated edge-use" dynamic array partitions; the output consists of three new arrays: the "vertex coordinate" dynamic array, containing the vertex coordinates corresponding to each unique vertex ID, an "edge-use's root vertex" array, containing each edge-use ID with the vertex ID for its corresponding root vertex, and a "translated edge-use" dynamic array with the same entries as the input untranslated edge-uses but with the vertex-use coordinates replaced by vertex IDs.

Some of these arrays are static and some are dynamic. We output the "edge-use's root vertex" information in one array per input partition, thus ensuring that each resulting array will also fit in memory (for later sorting). These arrays will have the same number of entries as the input partitions; therefore, we can allocate them statically. On the other hand, the "translated edge-use" information needs to be partitioned differently than the input; therefore, we do not know its partition sizes and thus cannot allocate static arrays for them. We also do not know how many distinct vertices we will have; therefore, we must use a dynamic array to hold the vertex coordinates as well.

For each "untranslated edge-use" partition, first we allocate a vertex hash table partition. It should be slightly over 25% bigger than the expected number of vertices so that it will not fill beyond 80% capacity and need not be re-hashed. We already know the total number of vertex *uses* that we will be looking up when we allocate the vertex hash table partition, unlike in the case of the naive algorithm; therefore, we can make a good guess about the number of distinct vertices that we will need to store. We assume an average vertex valence of six (i.e. that every vertex is used by six triangles.) The input key to the vertex hash table is a coordinate triple, and the data stored in the hash table along with the key is the corresponding vertex ID.

After we have allocated the vertex hash table partition for the "untranslated edge-use" dynamic array partition, we perform the first translation step. We read each untranslated entry $<$ *Edge-Use ID, Endpoint 1 Coordinates, Endpoint 2 Coordinates* $>$ from the input partition in turn, and look up the coordinates of the middle element of the entry, Endpoint 1, in the vertex hash table partition. (We use the same hash function as described above for the naive algorithm.) If the coordinates are not found in the hash table, we assign the next sequential vertex ID to these coordinates, record this vertex ID in the previously empty hash table entry, along with the coordinates, and append the coordinate triple as the ID-th element in the "vertex coordinate" dynamic array. (When we process subsequent partitions, we use the same counter for assigning vertex IDs and output them to the same, unpartitioned, "vertex coordinate" dynamic array.) Otherwise, we read the previously assigned vertex ID from the hash table. This endpoint is the root vertex for the directed edge-use; we combine its vertex ID with the edge-use ID from the original entry and append the pair $<$ *Edge-Use ID, Vertex ID* $>$ to the current "edge-use's root vertex" array partition. Finally, we replace the middle element of the entry with the vertex ID and append the semi-translated triple, $<$ *Edge-Use ID, Vertex ID, Endpoint 2 Coordinates* $>$, to the appropriate intermediate "semi-translated edge-use" array partition. This time we partition based on the hash value of the final field in the entry, the coordinates of Endpoint 2, since that is the next field we will be hashing. (The input was partitioned based on the hash value of the coordinates of Endpoint 1, which in general will be found in a different hash table partition than Endpoint 2; hence the need to re-partition.) Even though we are re-partitioning, however, we can still use static arrays, because each vertex appears the same number of times in both endpoint positions; therefore, the partitions will be the same size as last time. After processing each "untranslated triple" dynamic array partition, we free its memory, but we do not free the corresponding vertex hash table partitions yet.

In the second vertex translation step, we translate the second and final coordinate in each "semi-translated edge-use" to a vertex ID using the same hash table partitions we built in translation step number one. We process the partitions in the opposite order this time, starting with the "semi-translated edge-use" partition corresponding to the last vertex hash table partition that we built, since this hash table partition will still be in memory. We append the resulting translated triple, $<$ *Edge-Use ID, Vertex ID, Vertex ID* $>$, to the appropriate "translated edge-use" dynamic array partition (this time basing the partition choice on the hash value of the pair of vertex IDs, as explained in the next section). The two translation steps of stage two are diagramed in Figure 5.14. We can free each vertex hash table partition and its corresponding "semi-translated edge-use" array partition after we finish processing it.

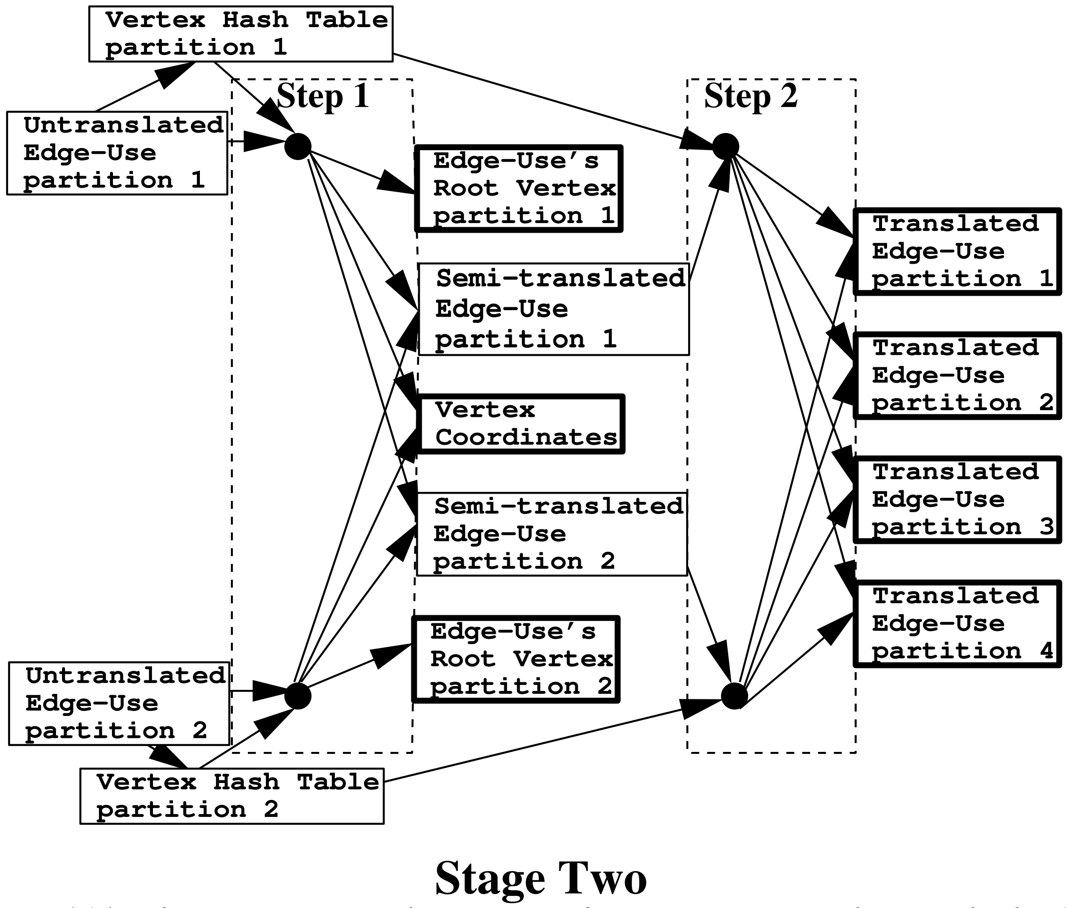


Figure 5.14: *The two vertex translation steps of stage two. We use the same hash table partitions for both steps but repartition and visit the hash table partitions in reverse order in step two. Note that there are more partitions at the end for the edge hash table we will build in stage three, since there are more edges than vertices.*

If the entire vertex hash table fits in memory and we are not partitioning, then we can perform translation step two at the same time as step one, since we'll always be looking at the same, lone hash table partition to find both vertex-use IDs for the triple. In fact, if we have not partitioned, then we could further optimize by looking up only one time each the three distinct vertex coordinates that appear as opposite endpoints of the three consecutive untranslated edge-use entries for a single triangle. This will halve the number of vertex-use lookups compared to the partitioned case. Furthermore, the "edge-use's root vertex" array for the unpartitioned case would not need to record the edge-use ID explicitly since they will be generated sequentially. Even if we have multiple partitions, some edges will still have both endpoints in the same partition. If we find upon hashing the second endpoint at the end of translation step one that it belongs in the same partition, we translate it immediately and output the fully translated triple directly instead of going through the semi-translated tables.

5.2.3 Stage Three

In stage three, we match up edge-uses that are on the same edge, outputting a dynamic array that records sibling pointer information (the "edge-use's sibling" dynamic array). We also record the IDs of one edge-use per edge in a dynamic array; later we will build the global linked list of distinct edges for the LEDS from this array.

For this stage we use a new partitioned hash table in which each entry records information about a distinct edge. For valid closed solid input, each edge has at least two edge-uses, and only occasionally more than two; therefore, we allocate an edge hash table partition whose size is 63% of the number of edge-uses we will process for the partition (the expected number of edges is 50% of the number of edge-uses; we divide by .80 = 62.5% and allocate slightly more, so that a few non-manifold edges will not fill more than 80% of the hash table and require allocating a larger hash table and rehashing). The input key to this hash table is the lexicographically ordered pair of vertex IDs of the endpoints of an edge-use (as with the naive algorithm, we use the lexicographic ordering to hide the direction of the original edge-use so that we can match it with the unoriented edge). When we partition the "translated edge-use" dynamic array that is our input at the end of stage two, we base the partition choice on the hash value of this input key. In addition to storing the input key, the edge hash table's data field entry stores up to two edge-use IDs for the edge, in the "first edge-use" field and the "most recent edge-use" field (shown in Table 5.3), as described below.

Key	Lesser Vertex ID	Greater Vertex ID
Data	First Edge-Use ID	Most Recent Edge-Use ID

Table 5.3: *An edge hash table key-data pair.*

For each partition of the "translated edge-use" dynamic array, we look up each entry's lexicographically ordered vertex IDs in the corresponding edge hash table partition. If the edge is not found, we make a new entry for it and record the edge-use ID (the first element from the input entry) in the "first edge-use" field. We also add this edge-use ID to the end of the dynamic array of distinct edges.

If there is already an entry for the edge in the hash table partition, and it has the "first edge-use" data field filled but not the "most recent edge-use" field, this is the second edge-use for the edge. We read the data from the "first edge-use" field in order to append two new entries to the "edge-use's sibling" dynamic array: one pair of edge-use IDs representing the pointer from the first to the most current edge-use, and one pair of edge-use IDs representing the pointer from the current to the first edge-use. Then we record the current triple's edge-use ID in the "most recent edge-use" field.

For input that was guaranteed to be 2-manifold, these first two edge-uses would be all the siblings for the edge; each edge-use of the pair would point to the other, its sole sibling. If this was the case for all edges, we would not need to record the most recent edge-use in the hash table. In fact, we could delete the edge's whole entry after processing the second edge-use in order to free up more space in the hash table. But for non-manifold parts, we can have more than two edge-uses per edge.

When a non-manifold edge-use hashes to an edge entry that already has both of its data fields filled by two other edge-uses for the edge, we also append two new entries to the "edge-use's sibling" dynamic array: one pair of edge-use IDs representing the pointer from the current triple's edge-use to the "first edge-use" recorded in the hash table (as before), and one pair of edge-use IDs representing the pointer from the "most recent edge-use" recorded in the hash table to the current triple's edge-use. This latter sibling pointer information will, when the actual LEDS edge-use is filled in, override the "edge-use's sibling" pair we recorded when we processed the "most recent edge-use," back when we recorded that its sibling was the first edge-use. In the LEDS, the sibling pointers of the edge-uses at each edge will thus form one circular list, though the order of the list will depend on the input file and will not necessarily be radially sorted. (We do radial sorting later and/or divide up the coincident edge-uses into pairs to make a pseudo-2-manifold representation if a particular application requires it.) Then we overwrite the "most recent edge-use" field in the hash table entry with the current input entry's edge-use ID, so that we can add additional siblings to the final circular list. Later, when we process the "edge-use's sibling" dynamic array, we will have two entries telling us what should be recorded in the sibling pointer field for some of these non-manifold edge-uses. We must be sure to take the latter one. These steps are illustrated for a sample non-manifold edge in Figure 5.15.

We repeat this process for all of the input partitions. We can free the memory for each "translated edge-use" dynamic array partition and its corresponding edge hash table partition after each has been processed, since they are not reused.

5.2.4 Stage Four

We wait until this final stage to actually allocate the LEDS vertices, edge-uses and faces. We fill in one type of LEDS entity at a time using the information in the dynamic and static arrays we have built, sorting them first (if necessary) by the ID of the entity with which the relationship each describes will be stored. The larger input arrays will be stored in multiple partitions which individually fit in memory. We sort these partitions separately and then alternate reads to the next position in each partition to find the partition containing the data for the next sequential ID being processed, performing the final "merge" stage of a merge-sort implicitly. We will only be making one sequential pass through each sorted partition during the merge; therefore, we will only need one block of each sorted partition in memory at a time.

We cannot start by filling in the edge-uses because we will not have all of the information contained in them until after we have processed the vertices. We do not start with the faces because for triangulated input there is no intermediate face data that we can free after filling them in; therefore, we want to delay allocating the faces as long as possible to minimize the total memory requirements. Therefore, we fill in the vertices first. Vertices point to edge-uses, and in order to derive pointer values, we need to know the location of the final LEDS edge-use array; therefore, we must allocate it before filling in the vertices. We do so now. Its address also gives us the information we need to build the global linked list of distinct edges for the LEDS from the array constructed in stage three of one edge-use ID per edge. We would like this list in the same order that these edge-uses will be stored, in

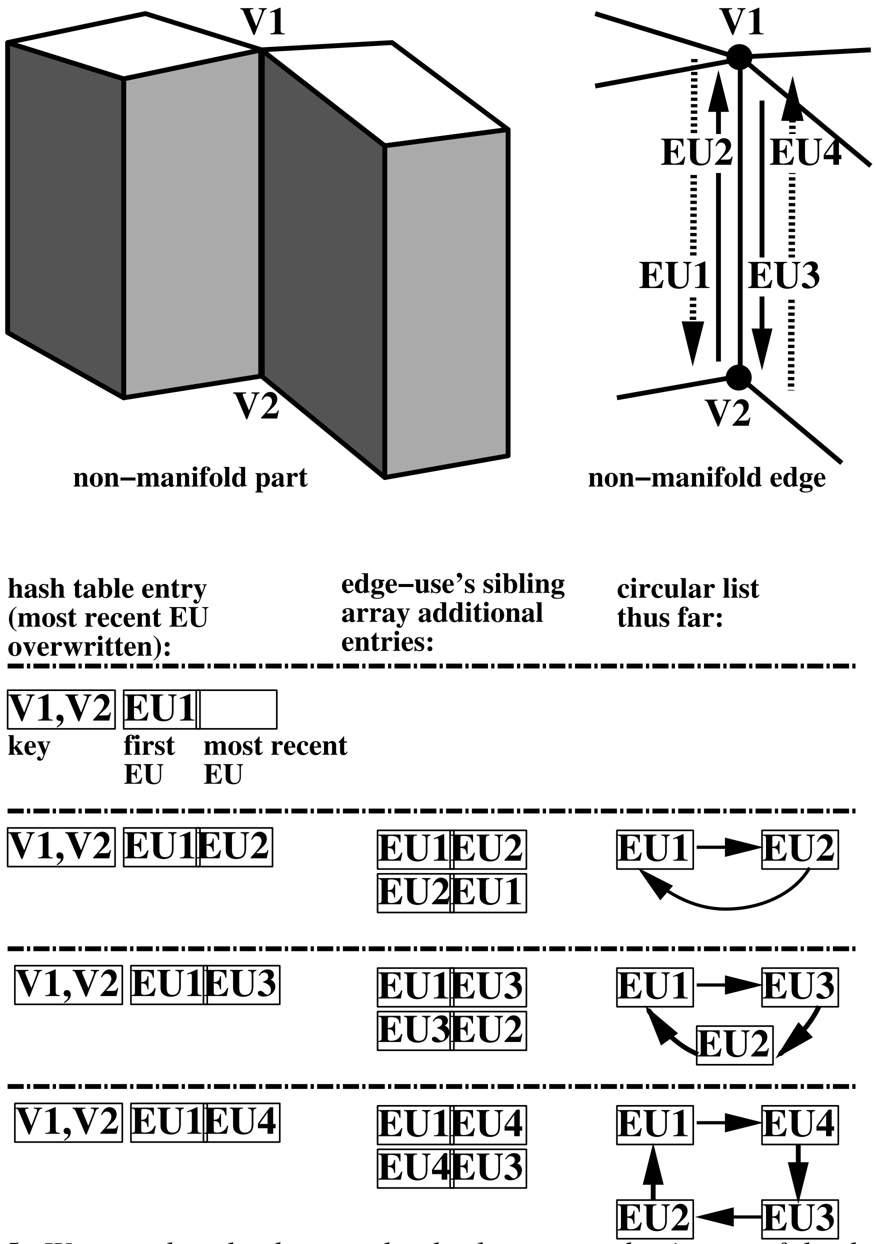


Figure 5.15: *We record each edge-use that hashes to an edge in one of the data fields in the hash entry. If the "first edge-use" data field is full, we record it in the "most recent edge-use" data field and append two new entries to the edge-use's sibling dynamic array, as illustrated. Interpreting each new sibling array entry to over-ride any previous entries for the same edge-use, we get the circular list shown on the right after each additional edge-use is added.*

order that later we can efficiently look up each edge-use in the list in sequence. Therefore we sort the array by ID (in place, if it fits in memory, otherwise with an external merge sort). Then we read through the sorted array, adding to the end of the global edge list an entry containing a pointer to each edge-use (deriving the pointer address from the address of the start of the LEDS edge-use array and the edge-use ID). At the end, we free the array.

Now we allocate and prepare to fill in the data in the LEDS vertex array. We can fill in each LEDS vertex's coordinates field directly while reading sequentially from the "vertex coordinate" dynamic array, which is indexed by vertex ID; therefore, we do not need to do any additional preparation for that field. The information we need in order to fill in each vertex's other field, the "first edge-use" field, is stored in the "edge-use's root vertex" array partitions. Each of these partitions was created with the edge-use IDs in increasing (though not consecutive) order. Although we will need the information in that order later for the edge-uses, for the vertices we sort each of these partitions by vertex ID to bring all of the edge-uses for a single vertex together. (We could make a separate copy to sort in vertex ID order, but it is actually more efficient to sort and then re-sort back to the original order.) Since the "edge-use's root vertex" array partition was built from a single hash partition, partitioned based on vertex coordinates, all of the edge-uses for a single vertex will appear in the same partition. Thus we can maintain the same partitions (which will again fit in memory) and sort within each. The final "merging" step of the merge-sort, combining the sorted partitions, is performed implicitly when we consult the next position in all these partitions to find the partition containing the information for the current LEDS vertex we are filling in.

Now we have all the data ready to fill in the vertices. We find the coordinates of each directly from the "vertex coordinate" dynamic array, and find the "vertex's edge-uses" partition containing the entries for the current vertex's ID. We record its first edge-use entry in the LEDS vertex itself, translating it to a pointer based on its ID and the address of the LEDS edge-use array. The vertex's remaining edge-uses will be stored in the edge-uses themselves.

Now that we have grouped the edge-uses together by root vertex, we can output entries for the "edge-use's next vertex edge-use" array. We read each additional sequential entry for the current vertex from its "vertex's edge-uses" input partition and append a pair of edge-use IDs to the "edge-use's next vertex edge-use" array: the ID of the prior edge-use for the current vertex just read from the input, and the ID of the edge-use in the current entry. After processing the last entry for the current vertex, we also output a pair of IDs to link its edge-use back to the first edge-use for the vertex. (See Figure 5.16.) These pairs will be translated to the pointers in the LEDS edge-uses that will form a circularly linked list of all the edge-uses rooted at the vertex. The edge-use's next vertex edge-use we output is partitioned based on the first edge-use ID in the pair.

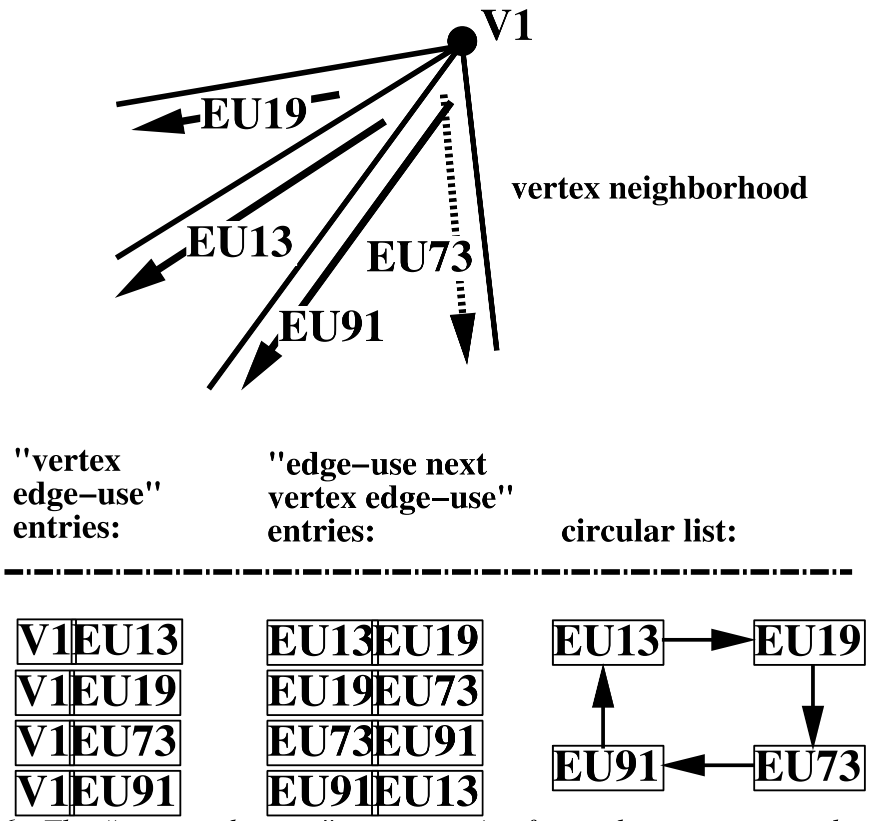


Figure 5.16: *The "vertex edge-use" array entries for each vertex are used to fill in the "edge-use next vertex edge-use" array entries for all of the edge-uses rooted at that vertex.*

After we have filled in all of the LEDS vertices, we can free the vertex coordinate dynamic array. We do not free the edge-use's root vertex partitions, but instead re-sort each by edge-use ID (back to its original order).

Next, we fill in the edge-uses. This requires some additional sorting before we begin. We must sort the "edge-use's sibling" dynamic array partitions. Recall that if the input was non-manifold, we need to maintain the order of multiple siblings listed for the same edge-use and only record the last one listed. The edge-use's siblings will be partitioned based on edges; therefore, all of the information for a single edge-use will be in the same partition, and we can maintain the partitions for sorting. We also need to sort the “edge-use's next vertex edge-use” array that we just created while filling in the LEDS vertex array, sorting it by the ID of the first field in each edge-use pair. This is another large array; therefore, when we create it, we write it out in partitions, using the same partitions the input was in, and now we sort within these partitions.

The IDs for the other two fields we need to fill in for the LEDS edge-use entities, the “edge-use's face” and the “edge-use's next-in-loop edge-use,” are derived from the ID of the edge-use we will store them in. The edge-use with ID n will point to a face with ID $\lfloor n/3 \rfloor$ and it will point to a next-in-loop edge-use with ID $n - 2$ if $(n) mod\, 3 = 0$, and ID $n + 1$ otherwise.

The only other information we need to fill the five edge-use pointer fields is the address of the LEDS face array; therefore, we allocate it now. Then we go ahead and fill in the LEDS edge-use entities sequentially, finding the appropriate partition containing this edge-use's ID for each of the three partitioned dynamic arrays, as described above for vertices. We can actually avoid having to look at multiple “edge-use's next vertex edge-use” partitions to find the one containing the current edge-use ID, because we created these partitions from the “edge-use's root vertex” array partitions without re-partitioning. Therefore, once we have found the partition index of the “edge-use's root vertex” partition containing the current edge-use ID, we know we will find the identical edge-use ID as the next item in the “edge-use's next vertex edge-use” partition with the same index. We translate the edge-use, face, and vertex IDs to pointers based on their ID value and the address of the start of the respective LEDS array. After all the edge-uses are filled in, we free the partitions for these three remaining dynamic arrays.

Finally, we fill in the LEDS face array's outer loop edge-use field. This edge-use ID is derived from the face ID: the outer loop edge-use for face with ID n will have ID $3 * n$. The pointer address is computed from the address of the edge-use array and the edge-use ID. The inner loop list pointer for each face is *null* for triangulated input.

5.3 Memory Management

For the out-of-core algorithm, the majority of reads and writes are sequential, so that all of the data on a page can be read or written while it is still in memory. For the non-sequential accesses to the hash tables, we divide the hash tables into partitions that fit in the available memory and group all reads and writes by partition. For Unix operating systems that support the `mlock()` function (including Irix and Linux, running on the machines we used for the results reported here), we use it to lock the partitions in memory while we are accessing them. If `mlock()` is not available, as an alternative we use hash tables half the size of available memory. With this ratio of hash table pages to input pages, the random nature of the hash lookups will usually mean that the hash table pages have all been accessed more recently than the oldest in-memory pages of the sequentially accessed input array, minimizing page faults in the hash table with LRU page replacement. We also tried using `mmap()` and `madvise()` to tell the operating system that the dynamic arrays would always be accessed sequentially. Unfortunately, `mmap()` requires creating a file on disk for each uninitialized array; this overhead far outweighed the savings during real accesses, since we only write and read each array one time.

5.4 Results

For comparison with the naive in-memory algorithm, we ran the out-of-core algorithm on the same knot sculpture files on the same two platforms (an SGI Indy with one 133 MHz MIPS R4600 processor and 32 MB of RAM, and a Linux PC with two Pentium III 700 MHz processors and 32 MB of RAM). These results are graphed in Figures 5.17 and 5.18.

For smaller files where some but not all data fits in memory, the out-of-core algorithm can take up to three times longer to build the LEDS than the naive algorithm, due to the time required to write the intermediate data. For our coherent input files, the break-even point comes at approximately 400,000 triangles on both platforms. For the randomized input files, the break-even point comes after 70,000 triangles on the Linux PC and after 100,000 triangles on the SGI Indy. With the million triangle test part, we more than make up for the overhead of the intermediate data with drastically reduced thrashing. For the

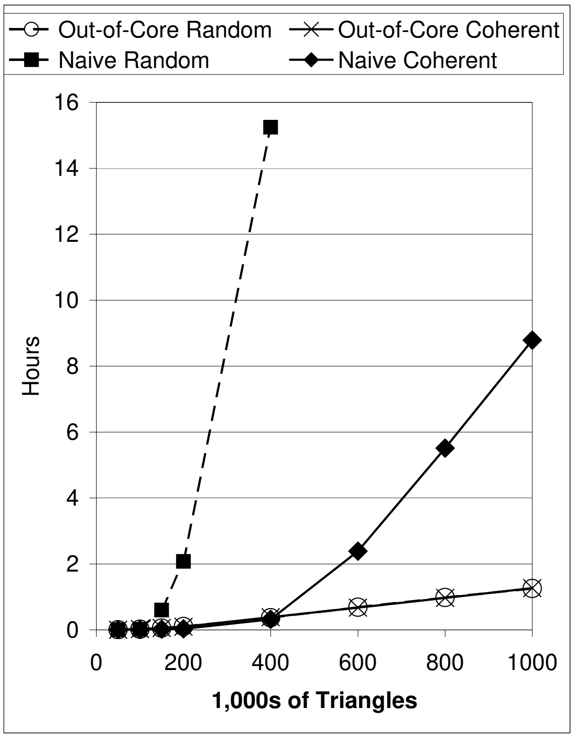


Figure 5.17: *Comparison of naive and out-of-core algorithm LEDS build times on large knot sculpture files on the SGI Indy with 32 MB RAM. The out-of-core algorithm performs equally well on the coherent and non-coherent input. On the coherent million triangle input, the naive algorithm takes 7 times as long as the out-of-core algorithm.*

coherent million triangle test part, we get a 7 times speedup on the SGI Indy and an 82 times speedup on the Linux PC, compared to the naive algorithm. For the randomized million triangle test part, the naive algorithm was so slow that we had to terminate it after running a few days, but the out-of-core algorithm performs almost identically on the coherent and randomized million triangle inputs: about an hour on the SGI Indy and about five minutes on the Linux PC. For the largest randomized file on which we successfully ran the naive algorithm, the 600,000 triangle test part, the naive algorithm took over 500 times as long as the out-of-core algorithm on the Linux PC.

As mentioned above, the coherent knot test parts were designed to have optimum spatial coherency in the ordering of their triangles, while the randomized versions have no coherency. Most real-world input will fall somewhere between these two extremes, with our algorithm showing speedups somewhere between 82 and 500 on similarly sized files when run on the Linux PC. As another test case, we ran the naive and out-of-core

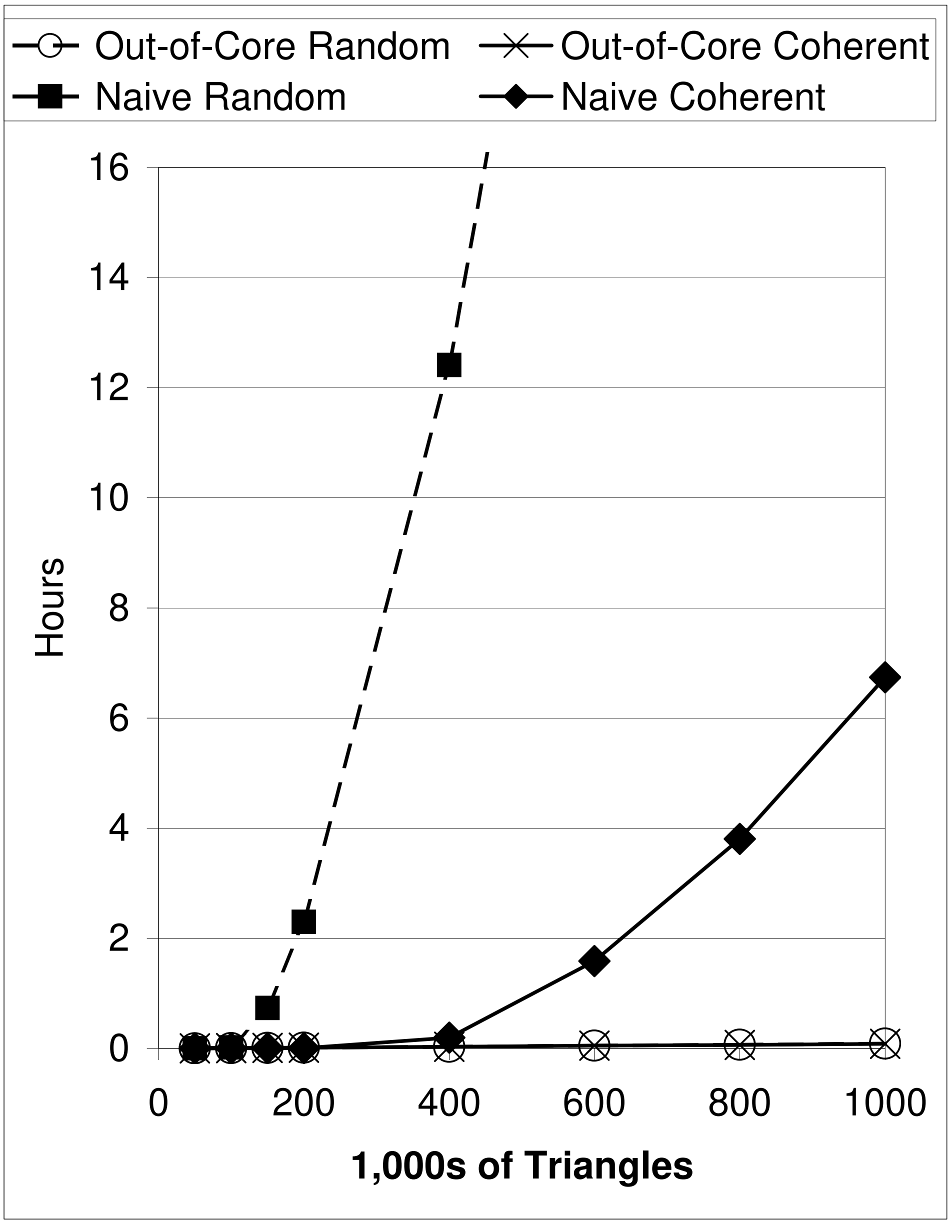


Figure 5.18: *Comparison of naive and out-of-core algorithm LEDS build times on large knot sculpture files on the Linux PC with 32 MB RAM. On the coherent million triangle input, the naive algorithm takes over 80 times as long as the out-of-core algorithm; on the 600,000 triangle random part, the naive algorithm takes over 500 times as long.*

algorithms on an STL file of the dragon model from Stanford's 3D Scanning Repository, an 870,000 triangle model reconstructed from data scanned with a laser range finder. We would expect models reconstructed from scanned data to be fairly coherent since the data is gathered in well organized blocks that cover a contiguous portion of the surface. Averaging over five trials, constructing the LEDS for the dragon took 9 hours, 15 minutes with the naive algorithm, but only 4 minutes, 11 seconds with the out-of-core algorithm, a speed-up of over 130 times.

We also measured the performance of the out-of-core algorithm on this and other large files using 128 MB of RAM. For these tests, we used an SGI Onyx Reality Engine with two 150 MHz MIPS R4400 processors and 128 MB of RAM, as well as the same Linux PC with two 700 MHz Pentium III processors, this time booted with 128 MB of RAM. These results are shown in Table 5.4. We tested different types of input files: the procedurally generated knot sculpture; a subdivision surface generated by applying nine iterations of Catmull-Clark subdivision to a single tetrahedron, followed by triangulating the resulting quadrilateral mesh; the dragon model from Stanford's 3D Scanning Repository; and the model of the head of Michelangelo's David provided by Stanford's Digital Michelangelo project, also reconstructed from laser range finder data.

Model Name	Number of Triangles	Linux Build (H:MM:SS)	Onyx Build (H:MM:SS)	Memory Usage
Dragon	869,898	0:02:48	0:17:55	287 MB
Knot Sculpture	1,000,000	0:03:12	0:23:05	279 MB
Smooth Tetrahedron	1,572,864	0:05:47	0:33:13	544 MB
David's Head	4,000,885	0:19:44	1:20:32	1,119 MB

Table 5.4: *LEDS build times (mean of five trials) with the out-of-core algorithm on the Onyx Reality Engine and Linux PC, both with 128 MB RAM.*

5.5 Memory Usage

If the out-of-core algorithm is not implemented carefully, it can require far more virtual memory than the in-memory algorithm, in order to store its intermediate data. To minimize its virtual memory requirements, we free each intermediate dynamic array and hash table partition as soon as we have finished processing it, so that we can re-use the memory. With this careful memory management, our out-of-core implementation uses almost exactly the same amount of memory as the in-memory algorithm (see Figure 5.19). For the in-memory algorithm, both hash tables are built simultaneously, and we cannot free them

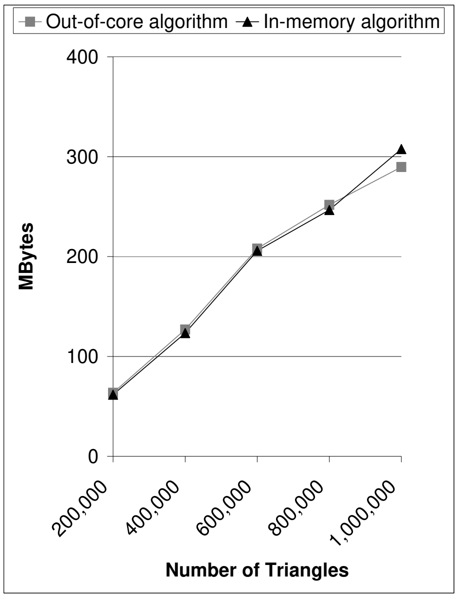


Figure 5.19: *Comparison of memory usage under Irix for the out-of-core and naive in-memory algorithms on different sized knot sculpture files.*

until the entire LEDS is built. For the out-of-core algorithm, we build the hash table partitions sequentially, freeing the vertex hash table before allocating the edge hash table, and freeing them both before allocating the LEDS. We also allocate the LEDS vertices, edge-uses, and faces in stages, allowing us to free all of the remaining intermediate data before finally allocating the LEDS faces (which are filled without intermediate data for triangulated input). An additional advantage of the out-of-core algorithm is that most of the intermediate and all of the final arrays arrays can be allocated in the exact size needed, so that less memory is wasted.

5.6 Variants of the Out-of-Core Algorithm

The general out-of-core algorithm described, while it builds a topological data structure very efficiently, may not be optimal when considered together with the running time of the application that uses the LEDS. This is because it may not organize the data within the LEDS optimally, depending on the access patterns of the particular application that will be using the data. An application can always re-sort the arrays of vertices, edge-uses, and faces after they have been constructed, but if we already know what application will be using the data structure, we might be able to build it so that its order is better tuned to the access patterns of that application in the first place. Often, a spatially coherent organization is desirable.

With the basic build algorithm, faces and edge-uses are stored in the same order that they appear in the input. If there is spatial coherence in the input, it will be preserved in the LEDS. For triangulated input, we exploit the simple numeric relationships between the face IDs and edge-use IDs to avoid having to record this information in intermediate arrays. Thus any advantages from changing the IDs at the start to induce a different ordering, rather than re-sorting at the end, would be offset by the added overhead of these new intermediate arrays. Therefore, there would be little advantage to changing the order of the faces and edge-uses during the initial build, unless the input was known to be non-coherent.

The basic build algorithm can destroy any input coherence in the case of the vertices, however. The vertex uses are randomly assigned to partitions during the initial read of the data; if there are many partitions, this will effectively shuffle them when the vertex IDs are assigned sequentially within each partition. Therefore, it might be worthwhile to wait until after the first pass through the data to partition for vertex hashing. This would allow us to gather statistics on the distribution of the vertices during the initial read, so that we could divide the vertex-uses into partitions that were spatially coherent and still had roughly equal sizes. It seems most useful to also divide our partitions with an eye on specific downstream processing needs. For example, if we were subsequently planning to run a sweep-plane slicer that looked at vertices in increasing z-coordinate order, as described in Chapter 7, we could divide the vertices into partitions based on increasing z-coordinate.

We could further sort by z-coordinate within each untranslated partition, though this overhead is unlikely to be worthwhile for slicing. During slicing, processing vertices requires one z-order read, alternating with reading the upper endpoint vertices of all active edges. With z-coordinate based partitioning, the next vertex to process would be in the same partition anyhow, and thus would likely already be in memory when accessed during slicing. The marginal gain from secondary caching from having it adjacent would be offset by the costs of sorting. A complete z-ordering might help arrange the upper endpoint vertices closer to the order in which they are accessed, but they still would not be accessed in exact z-order; therefore, the sort's effect on their access times would not be significant.

For a different application that was going to access the vertices in the same order several times, on the other hand, a complete sort at the start would make sense.

Looking ahead to the task of identifying non-manifold edges, we could combine it with the task of building the LEDS by adding a "uses" field to the edge hash table entries that recorded the number of edge-uses in each direction. After filling in each edge hash table partition and before freeing it, we would sequentially read through the entire partition to find all edges that were not used exactly twice in opposite directions. Then we would only need to look at this list to identify the non-manifold edges, rather than checking each edge in the list of all edges.

Of these possible variants, the one we have implemented is z-coordinate based vertex partitioning. We do not know the z-extents or distribution of the data before we begin, which prevents us from knowing where to place the partition boundaries for even partition sizes *a priori*. During the first pass through the triangle input data, we merely record a single dynamic array of the vertex coordinates of each sequential input triangle and find the minimum and maximum z value for the file. We still do not know the distribution in z; therefore we first evenly divide the range of input z values into small intervals, many more than the final number of partitions we need, and later combine consecutive intervals into partitions of even sizes. (Our intervals are not unlike the buckets used by Kitsuregawa et al. to tune partition sizes during hash joins [40], but we simultaneously sort and tune with our intervals, optimizing for processing that will occur after the initial spatial hash join as well.) We allocate a bin for each sequential interval, with the first bin corresponding to the lowest interval. Then we read through the array of vertex coordinates, transforming each set of three vertices defining a triangle into three "untranslated edge-use" entries, storing each entry in the bin corresponding to the interval containing the z-coordinate of its first endpoint. We also update an array that records the number of entries that have been placed in each bin.

Then we look at the bin sizes and contents to assign partition boundaries to get partitions of roughly equal sizes. The ideal partition size is equal to the total number of entries divided by the total number of partitions. For the first partition, we add up the number of entries in the first i bins until the total first reaches a number greater than or equal to the ideal partition size. If the total is less than or equal to 10% over the ideal size, bins 1 to i will be the partition. Otherwise, we subtract the number of entries in the i^{th} bin, and if this total is greater than or equal to 10% under the ideal size, bins 1 to $i-1$ will be the partition. In either case, the z-boundary of the partition is calculated and recorded (the highest bin number times the constant bin z-height for the first partition). Otherwise, we will have to divide the i^{th} bin between the first and second partitions (this will only occur if we allocated too few bins or if the vertex data is very unevenly distributed in z). Our implementation performs a quicksort on the whole bin and then finds the entry at the position for an ideal partition size; we record the z coordinate of this edge-use's first endpoint as the z-boundary of the partition. For better performance, we could modify the quicksort to terminate once we had an acceptable number of entries less than the pivot point and use the pivot point for the z-boundary. Since we rarely need to split bins, the additional complexity of modifying quicksort does not seem worthwhile.

We continue in this manner to find the z-boundaries of the remaining partitions, but rather than trying to get the size of each individual partition within 10% of the ideal size, we aim for the sum of the sizes of the partitions so far plus the current one to be within 10% of the sum of the ideal sizes. This prevents errors from building up, which could leave the final partition, consisting of all remaining entries, constrained to be much too small or large. With our scheme, individual partition sizes will, in the worst case, still be no more than 20% larger or smaller than the ideal size.

Recall that we partition vertices twice: once to translate the first endpoint in the untranslated edge-uses, then again to translate the second endpoint in the semi-translated edge-uses. For the first partitioning step, we build our hash tables and translate directly from the bins that the z-boundary table indicates belong entirely to the current partition (along with possibly a fraction of the end bin(s), if they were split). When we repartition the output of this first translation step, we actually allocate partitions, using the z-boundary table to place the output in the correct partition. The rest of the build proceeds as before.

The bins, in addition to aiding in partitioning evenly, also roughly sort the vertices within the partitions. During the first translation step, we process the bins in order; recall that it is also in the first translation step that we assign IDs to the vertices in the order that we process them. Therefore, the final vertex table will be sorted to the same granularity as the bin boundaries. More bins will result in a finer sort.

Of course, the bin partitioning scheme takes longer than random partitioning. In Figure 5.20, we compare the total times to build the LEDS followed by slicing with the sweep plane slicer described in Chapter 7 under Linux with 32 MB RAM. Our input is

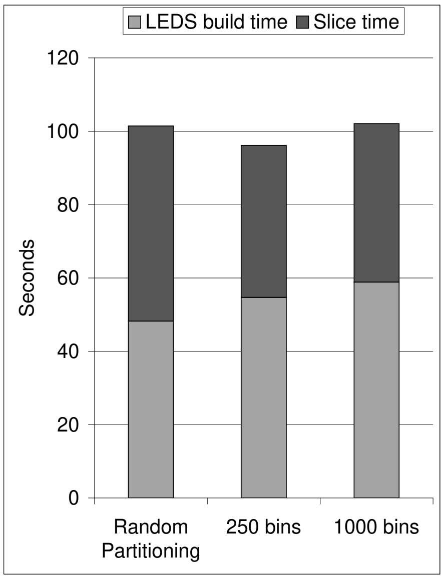


Figure 5.20: *Build times versus slice times for random partitioning and z-coordinate based vertex partitioning on the 200,000 triangle knot sculpture.*

the 200,000 triangle knot sculpture, we make a total of 402 slices through it, and we use 10 partitions. Allocating 250 bins total (25 bins per partition), the total build plus slice time was a bit faster than with random partitioning, even though the build time was longer. Using 1,000 bins (100 bins per partition), the build time increased even more, and the savings in analyzing and slicing no longer offset the increased build time.

Chapter 6

Input Analysis, Verification & Clean-up

After we have built the LEDS, we can use it to analyze the input topology and geometry of a design or part to be fabricated. If we want to manufacture the part using layered manufacturing, we must verify that the input indeed describes a closed, water-tight boundary. If there are small cracks in the boundary, we attempt simple clean-up by merging vertices. This is sufficient for repairing files whose only problems are round-off errors. For more seriously flawed input, more sophisticated techniques, such as those described in the related work chapter, can be used to fix the input file.

Even once we have assured that the LEDS describes a closed boundary, the boundary may still contain non-manifold edge-uses or vertices where different pieces of the part just touch each other. At this point in processing we may choose to transform this non-manifold representation into a pseudo-2-manifold representation of the same geometry. Algorithms that can rely on 2-manifold connectivity are generally simpler, cleaner, and more elegant than algorithms for general b-reps. By transforming a b-rep of non-manifold solid into a pseudo-2-manifold, we can still take advantage of the former class of algorithms.

6.1 Analysis

Our analysis module uses the LEDS to report basic topological and geometric information about the part. We report the total number of vertices, edges, and faces for each shell, assuming that the input file correctly separated the shells. We iterate through the list of vertices and find the valence of each (equivalent to the number of edge-uses rooted at the vertex) and report the total number of vertices of each valence. We iterate through the list of edges and count the number of edge-uses in each direction along each edge, reporting the locations of all problem edges that do not have an equal number of edge-uses in both directions. We also report the total number of edges with one incident face, two incident faces, etc. If the shell is 2-manifold, we also report its genus, calculated using a generalization of Euler's formula [20], where #Holes is the total number of inner hole contours in all faces, and #Faces, #Edges, and #Vertices are the total number of faces, edges, and vertices respectively:

Genus = (2 + #Holes - #Faces + #Edges - #Vertices)/2

We also calculate the bounding box of the part while iterating through the vertices, and find the shortest edge length while iterating through the edges. Table 6.1 shows the output from the analyzer for a sample part.

If we find problem edges with unmatched edge-uses during analysis, we also record the vertices at both their endpoints in an array of "incomplete vertex" candidates for "epsilon vertex merging."

6.2 Epsilon Vertex Merging

Because we have found that the cracks in our local, procedurally generated parts are all caused by round-off errors, we have implemented a simple routine that merges the representations of incomplete vertices located within some epsilon distance of each other as a first pass attempt to close cracks. Cracks that remain after this step must be closed using more sophisticated techniques such as described in the related work chapter; our focus is on cleaning up files with errors caused by precision limitations.

```
Part is 2-manifold.
Shell is 2-manifold with genus
37.
Bounding Box Info:
   Min X = -6.9006, Max X =
7.0833
   Min Y = -1.1627, Max Y =
1.1627
   Min Z = -7.0684, Max Z =
7.0684
53688 vertices.
   192 vertices have valence 5.
   53184 vertices have valence 6.
   192 vertices have valence 7.
   96 vertices have valence 8.
   24 vertices have valence 16.
161280 edges.
   161280 edges have 2 faces
adjacent.
107520 faces.
```

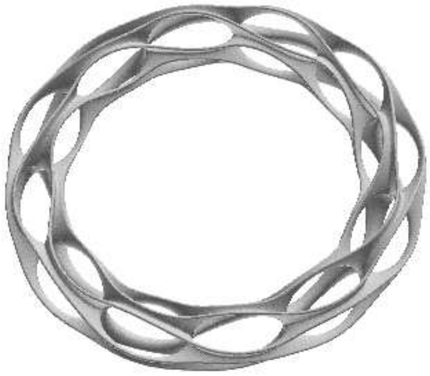


Figure 6.1: *Analyzer output for sample part pictured at right.*

6.2.1 A Comparison of Different Strategies

There are a number of schemes we could use for epsilon vertex merging. We could snap all the incomplete vertices to coordinates on an epsilon resolution grid and merge those that ended up on the same grid coordinates. The drawbacks to this scheme are that two vertices that were arbitrarily close to each other could still snap to different grid coordinates, and the merge choices are sensitive to translation or rotation of the input (see Figure 6.2). Alternately, we could process the vertices sequentially and whenever a new vertex fell within epsilon of a previously stored vertex, merge them into a new vertex at their average position. The drawbacks to this scheme are that the merge decisions and final vertex positions could vary depending on the valence of the input vertices (see Figure 6.3), and in degenerate cases vertices could drift arbitrarily far from their original positions (see Figure 6.4).

Using a weighted average, the final vertex position would be order-independent, but the

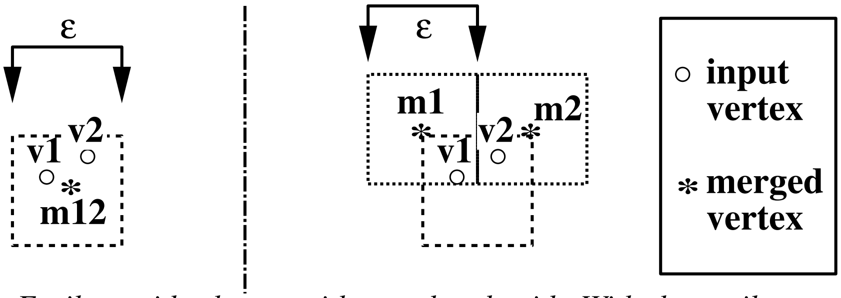


Figure 6.2: *Epsilon grid scheme, with translated grid. With the epsilon resolution grid positioned as on the left, the two vertices v1 and v2 would merge into one merged vertex, m12. If the grid was translated as shown with the finely dotted squares on the right, however, the vertices would continue to be seen as separate after the epsilon vertex merging step.*

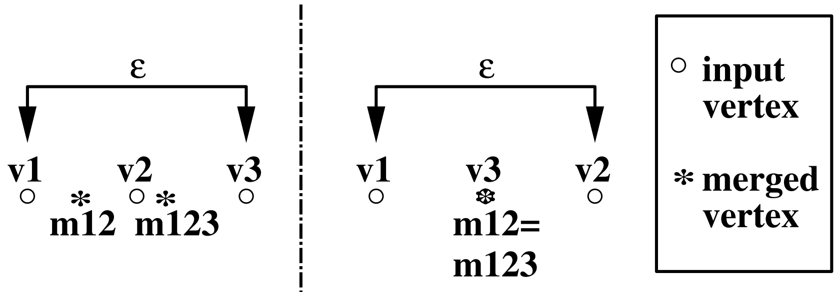


Figure 6.3: *Averaging scheme, two different input orders. With the vertices input in the order shown on the left, we would first average v1 and v2 to get m12, and then average m12 and v3 to get m123. With the vertices input in the order shown on the right, however, m12 and m123 would both be in the same position as v3.*

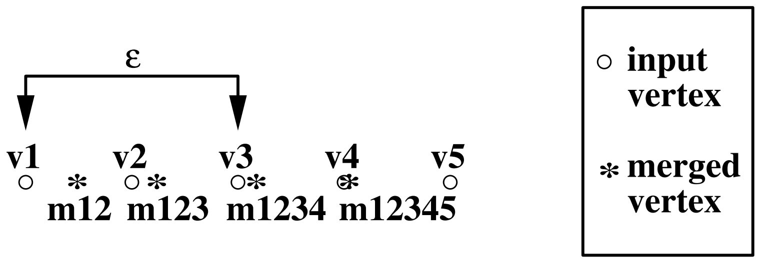


Figure 6.4: *Averaging scheme. With the vertices input in the order shown, the final merged average position is more than epsilon away from some of the merged input vertices.*

scheme would still be subject to arbitrary vertex drift and input order-dependent merging decisions. We could merge if *any* of the original vertices from a merged vertex were within epsilon of our new vertex; this would be input order-independent, but it would still be subject to arbitrary vertex drift. For example, if closely spaced vertices around a circle are merged in sequence (see Figure 6.5), the final result will lie at their centroid. If they are examined in a less regular order, the final result is more likely to be several irregularly placed merged vertices from clusters of original vertices.

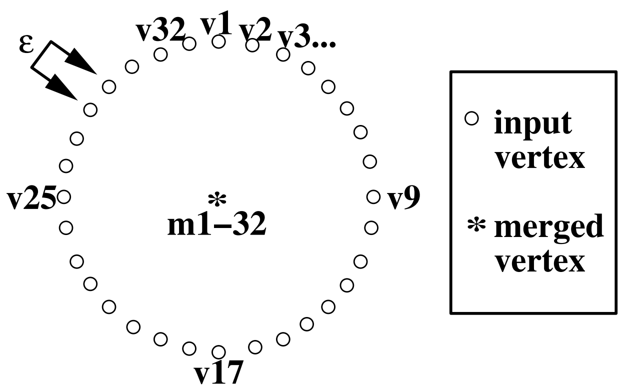


Figure 6.5: *Weighted average/any neighbor scheme. With the 32 vertices input in clockwise order as indicated, the final merged vertex is not within epsilon of any of the merged input vertices.*

6.2.2 Our Approach

The scheme we chose to implement merges a new vertex with the first previously stored vertex within epsilon, keeping the position of the old vertex. This scheme has the drawback of being input order dependent (see Figure 6.6), but it is translation invariant, the vertex drift is bounded by epsilon, and the computation and storage costs are low.

With an appropriate choice of epsilon, only the positions of the final merged vertices will be input order dependent, not the choice of which vertices to merge (see Figure 6.7). We choose an initial epsilon that is 10% of the shortest edge length unless the user specifies a different value. Using an epsilon less than the shortest edge length assures that we will not collapse edges, potentially losing details or introducing zero-area faces (see Figure 6.8).

We also want to avoid inadvertently merging two vertices that should in fact remain distinct, as shown in Figure 6.9. In most cases, nearby vertices that should not be merged with an existing vertex are connected via edges to other, nearer vertices that should be merged with it. Using an epsilon less than 50% of the shortest edge length will eliminate erroneous merges in these cases. We still might merge with the wrong vertex if the part

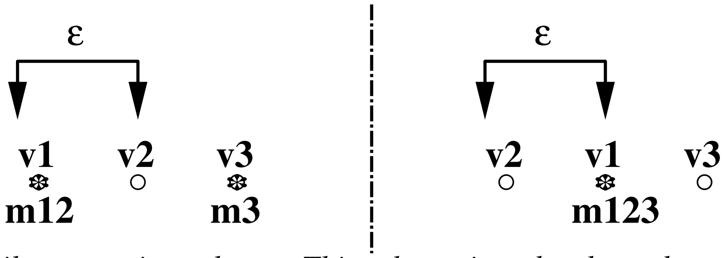


Figure 6.6: *Epsilon snapping scheme. This scheme is order dependent, as shown by the different merging results for the same data input in different order on the left and right. Final merged vertices, however, will never be more than epsilon from any of the merged input vertices.*

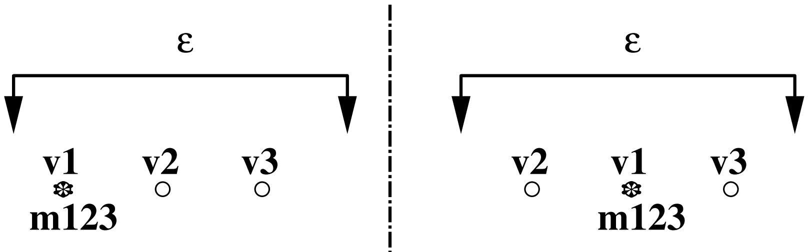


Figure 6.7: *Epsilon snapping scheme with the same vertices in the same two input orders as above, but a better choice of epsilon. Only the position of the final merged vertices, not the choice of which vertices to merge, is input order dependent.*

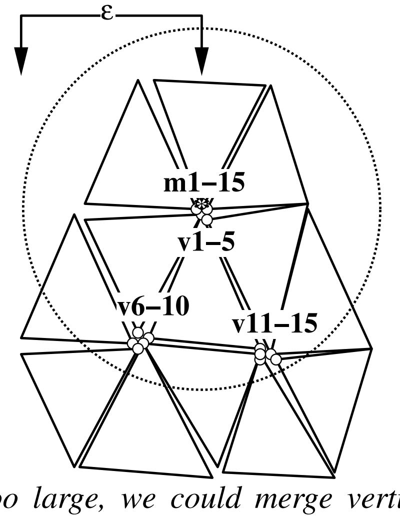


Figure 6.8: *If epsilon is too large, we could merge vertices that should be distinct, potentially collapsing edges and faces.*

had a thin neck or gap that was narrower than epsilon, as shown in Figure 6.10, but it is unlikely that a part with such a fine feature will be tessellated with triangles more than, say, ten times as large (corresponding to a 10% edge length choice for epsilon), and even more

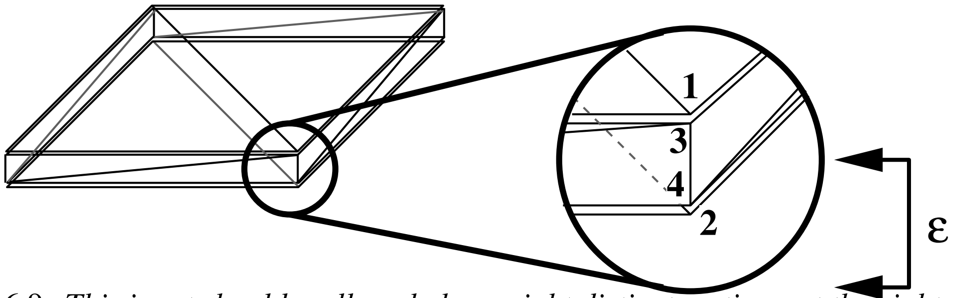


Figure 6.9: *This input should really only have eight distinct vertices, at the eight corners of the box. In the lower right corner, if we merged the vertices in the order listed, using an epsilon equal to the radius of the circle, we might mistakenly merge 1 with 2 and 3 with 4, instead of merging 1 with 3 and 2 with 4.*

unlikely that there will also be incomplete vertices on both sides of the narrow feature. By restricting our vertex merging to incomplete vertices, we minimize the risk of corrupting the clean portion of the file.

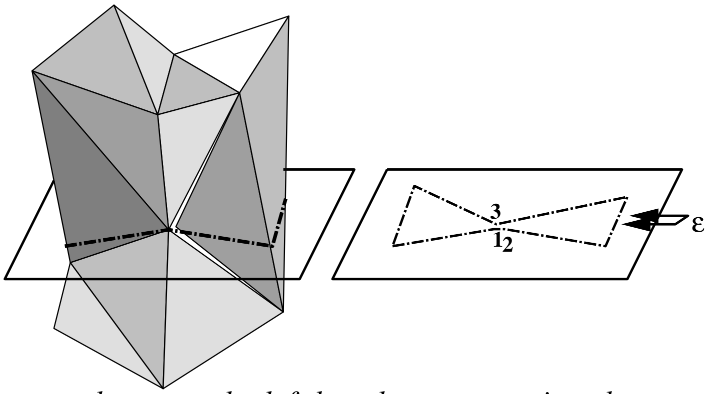


Figure 6.10: *The part shown on the left has the cross section shown on the right. Even with an epsilon smaller than the shortest edge length, we could still mistakenly merge vertex 1 with 3 instead of 2.*

Since the number of incomplete vertices is expected to be small (generally less than 1% of the total vertices), we use a simple in-memory spatial data structure, an octree, to speed the epsilon-merging operation. If the set of incomplete vertices is very small, we do not even need to bother with the octree (which essentially transforms the epsilon-merge into an Index Nested Loop join [24]). For a small number of incomplete vertices n less than about 10, the $O(n^2)$ comparisons are just as fast as building and traversing the octree.

An octree will not be a good indexing structure if it does not fit in memory, however, since the low 8-way fan-out will produce a deep index tree that will require a separate disk access at each level for each look-up, with little opportunity for caching. For a very large data set, it might be worthwhile to sort the vertices to achieve greater locality in the search for vertices within epsilon. The best global ordering is probably distance along a “space-filling” 3-D Hilbert curve (see Figure 6.11), which provides excellent locality for spatial range searches in arbitrary directions [34]. We can pack the vertices in Hilbert-sorted order into the leaf nodes of an R-tree built up on top of them, as described by Kamel and Faloutsos [38], so that the index tree will be shallow with optimal fan-out to minimize disk accesses.

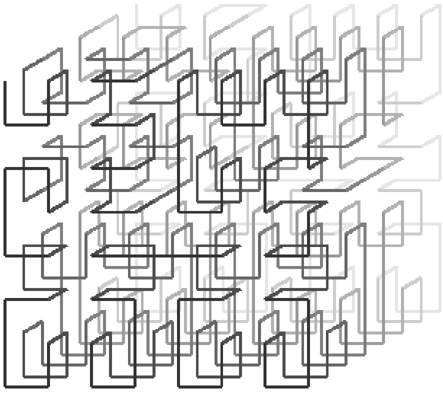


Figure 6.11: *A space-filling 3-D Hilbert curve*

In practice, however, we never expect to have more incomplete vertices than fit in physical memory from a LEDS constructed on the same machine. On today’s graphics workstations from SGI, the standard shipping configurations typically have RAM that is at least .63% of the disk size (e.g. 128 MB RAM with a 20 GB hard drive); the most powerful desktop system available from Dell Computer in June 2000 has RAM that is .67% of the disk size. For an object with average vertex valence six, the memory required to store an octree containing *all* the vertices (assuming imbalanced data makes the octree no more than two times larger than a balanced one), will be approximately 3.6% of the total virtual memory required for the build. [1] We only need to store a small fraction of these vertices,

[1]The number of pointers in a balanced octree with L leaves is $L * \sum_{k=0}^{\log_8 L} \frac{1}{8}^k < L * \sum_{k=0}^{\infty} \frac{1}{8}^k = \frac{8}{7}L = 1.14L$. If imbalanced data makes the octree twice as big, the total storage for the four-byte octree pointers will be 9.12 bytes/vertex. For each vertex, we store its three coordinates and a pointer to the LEDS vertex, an additional 16 bytes/vertex, for a total of 25.12 bytes/vertex. The total amount of memory we use during a LEDS build is approximately 700 bytes/vertex, depending on the file; 25.12/700 = 3.6%.

however – the “incomplete” vertices. If this fraction is less than 5% (and it’s typically less than 1%), then the octree will take less than .18% of the virtual memory required for the build. Thus it should fit in physical memory, even on desktop machines with high virtual to physical memory ratios. Therefore, we have not implemented Hilbert sorting for vertex merging.

We perform epsilon vertex merging to close cracks after building the topological data structure but before slicing. Others have incorporated similar techniques into the data structure building or slicing phases of processing. The QuickSlice 6.2 software [70] that ships with the fused deposition modeling (FDM) machine waits until after slicing to try to close cracks on a slice by slice basis. This may lead to inconsistencies between slices, since no underlying consistent solid is ever formed, and a crack that could have been fixed with a single vertex merge in 3-D will require as many merges as slices through all of its incident edges. Rock and Wozny’s software [61] merges all vertices within a round-off tolerance as they are read in, using a balanced binary AVL tree [41] for neighbor searching. This technique relies on choosing an appropriate epsilon before reading the data, with all of the associated pitfalls discussed above. Furthermore, they store all the vertices, not just the incomplete vertices; thus, the tree will not fit in core memory for very large files, causing the performance to degrade dramatically, since binary trees are inefficient for caching due to their low fan-out.

6.3 Making the Representation Pseudo-2-Manifold

After analysis and vertex merging, we may find that the boundary is closed but that the part is still not 2-manifold (such as the examples shown in Figure 2.6). In such cases, we can separate the non-manifold edges and vertices into multiple coincident edges and vertices in order to make a pseudo-2-manifold representation whose connectivity corresponds to that of a 2-manifold. Many algorithms that operate on topological data structures assume their input is 2-manifold or pseudo-2-manifold. Our slicer, for example, requires that all non-manifold edges be separated into pseudo-2-manifold edges.

6.3.1 Separating Non-2-Manifold Edges

First we separate non-2-manifold edges identified during the analysis stage. The goal is to transform them into coincident 2-manifold edges connected such that the boundary surface touches but does not intersect at the edge. We do this by matching the edge-uses into pairs with mutually referring sibling pointers, based on the radial ordering of the faces.

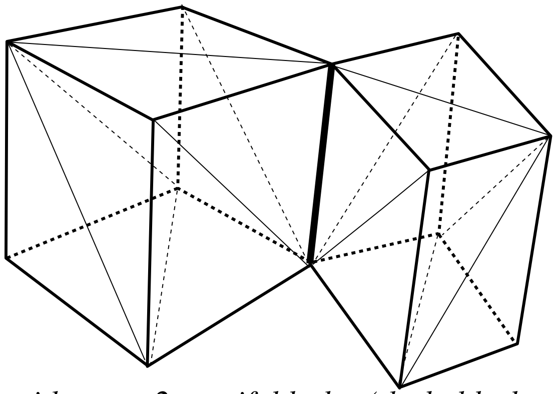


Figure 6.12: *Geometry with a non-2-manifold edge (the bold edge in the center) that must be separated into two pseudo-2-manifold edges. (The lightest lines indicate boundaries between coplanar triangles.)*

We perform the radial sort of the faces incident to the non-manifold edge, for example the one pictured in Figure 6.12, as follows. First we transform the faces into unit vectors on a projection plane perpendicular to the non-manifold edge, each vector radiating out from the point to which the edge projects. We call these "face vectors." We calculate each by taking a vector on the face pointing away from the non-manifold edge. This can be found by taking the vector of the next-in-loop edge-use after the edge-use along the non-manifold edge (continuing along the loop if it happens to be colinear), projecting it to the perpendicular plane, and normalizing it (see Figure 6.13).

Now we can radially sort the normalized face vectors. We choose one as our reference face vector, then calculate the dot product of each other face vector with the reference. The smaller the dot product, the larger the angle with the reference, ranging from a dot product of +1 for a face directly on the reference to -1 for a face directly opposite. These dot products, however, will not differentiate between vectors at 3 o'clock and at 9 o'clock with respect to the reference 12 o'clock vector. Therefore, we separate the non-reference face vectors into two lists that we sort separately, one list for those in the right hemi-circle

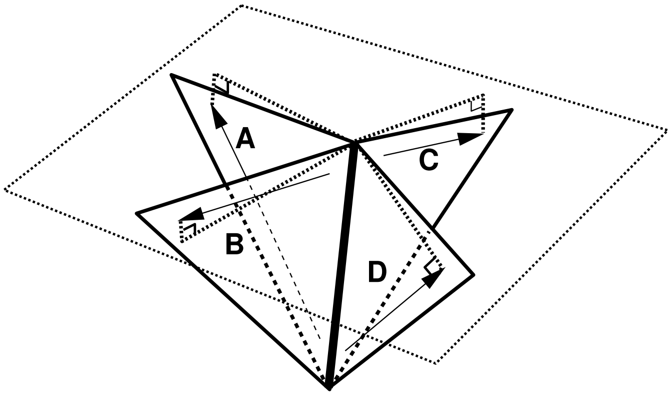


Figure 6.13: *The faces incident to the non-2-manifold edge, the vector on each face pointing away from the edge, and the projection plane for the edge.*

relative to the reference face vector, one list for those in the left hemi-circle. The appropriate list in which to place each face vector is chosen by calculating its cross product with the reference face vector. The cross products with those in the right hemi-circle relative to the reference face vector (from the point of view of an observer above the plane) will point down into the plane, while those in the left hemi-circle will point up out of the plane (see Figure 6.14). A face vector directly opposite the reference may be placed in either list. We actually do not care which is the right hemi-circle and which is the left; we just need to divide the face vectors into two lists according to which hemi-circle they fall in. Then we sort the list of vectors in one hemi-circle by increasing angle with the reference face vector and sort the list of vectors in the other hemi-circle by decreasing angle. For the complete radial ordering of all the incident faces, we order the face vectors with first the reference face vector (A in the example), followed secondly by the hemi-circle list sorted by increasing angle (C in the example), followed finally by the hemi-circle list sorted by decreasing angle (D,B in the example).

Next we look at the projected face normals that correspond to the face vectors in this radial ordering (shown in Figure 6.15). The normals point to the side of the face that is empty space, away from the side that contains material (the interior of the part). These projected face normals should alternate between being directed clockwise and counter-clockwise around the edge. If they do not, as in Figure 6.16, it indicates that the geometry was self-intersecting or that an entire shell was oriented incorrectly, and we reject the input.

We need to choose pairs of faces whose edge-uses along the edge in question are

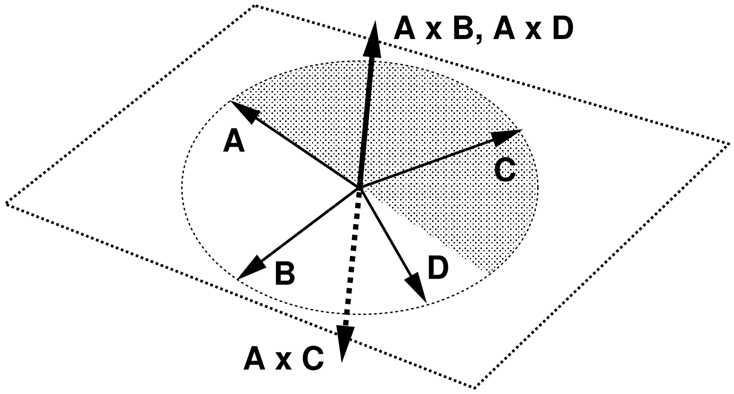


Figure 6.14: *Here we see the projected face vectors, with A as the reference vector.* $\overline{A} \times \overline{C}$ *points down into the plane; therefore,* $\overline{C}$ *is in the right hemi-circle (shaded) relative to* $\overline{A}$*.* $\overline{A} \times \overline{B}$ *and* $\overline{A} \times \overline{D}$ *have the opposite orientation; therefore,* $\overline{B}$ *and* $\overline{D}$ *are in the left hemi-circle.*

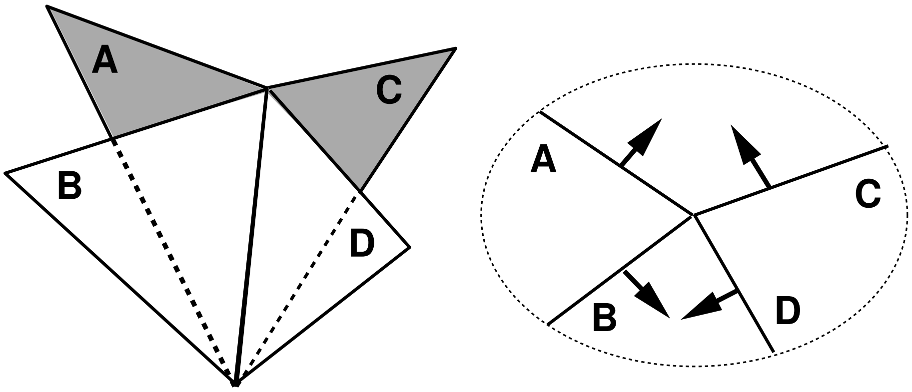


Figure 6.15: *On the left, the faces are shown (the gray ones are back-facing). On the right, the projected faces with the projected face normals are shown. A and D have clockwise normals; B and C have counterclockwise normals.*

in opposite directions (corresponding to clockwise or counter-clockwise face normals) in order to make separate 2-manifold edges. If we arbitrarily pair them up we risk introducing intersections in the boundary if there are more than four faces adjacent to the edge, as shown in Figure 6.17. There are two obvious choices that will not form intersections in the boundary: either we could pair faces adjacent in the radial ordering whose normals point away from each other (e.g. A & B, C & D in Figure 6.15), or adjacent faces whose normals point toward each other (e.g. B & D, A & C in Figure 6.15). The first alternative means that in a slice through the separated edges, the surrounding contour(s) will bound separate regions of material that happen to touch at this edge (see Figure 6.18), and the second

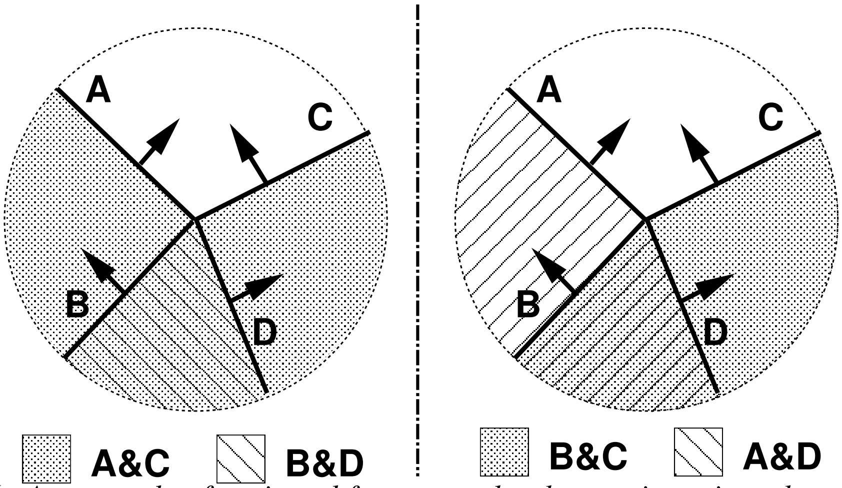


Figure 6.16: *An example of projected face normals whose orientations do not alternate. If A was matched with C, and B with D, the faces would bound two solid regions, one inside the other, as shown on the left. If A was matched with D, and B with C, the faces would bound two solid regions that overlapped as shown on the right. Neither choice is a legal configuration.*

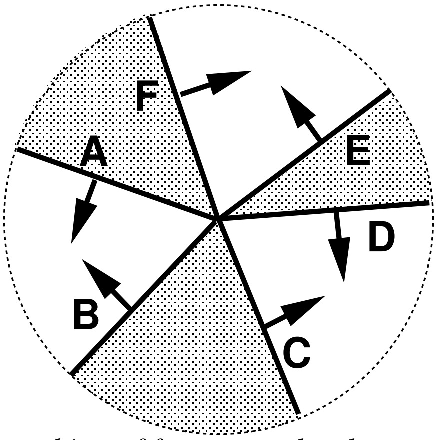


Figure 6.17: *An arbitrary matching of face normals whose orientations do alternate can also introduce intersections in the boundary, e.g. matching B & E, D & A, F & C. (Even though pairing F & E, A & D, B & C will not introduce intersections, it is simpler to pair up faces that are adjacent in the radial ordering than to check for intersections explicitly.)*

alternative means that in a slice through the separated edges, the surrounding contour will bound the same region of material that just happens to pinch down to zero thickness at this edge (see Figure 6.19). If more than four edge-uses are coincident, combinations of these two choices are also possible.

We have chosen to match the faces into pairs of adjacent faces with normals pointing

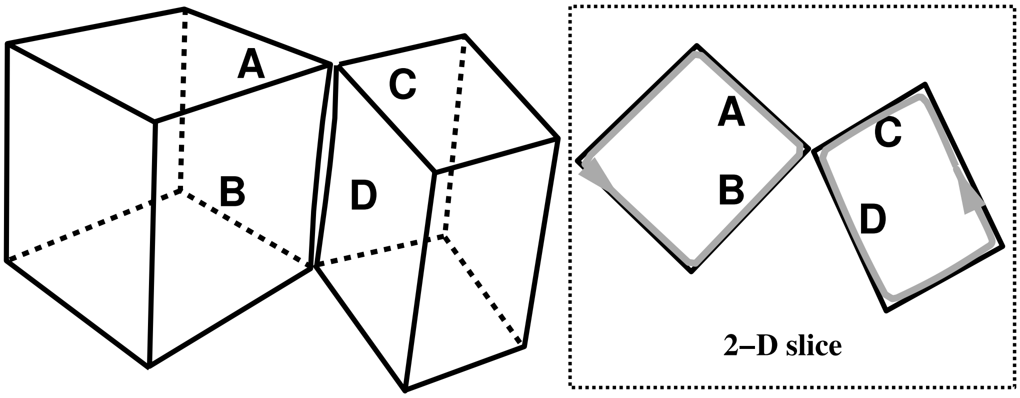


Figure 6.18: *Matching faces A & B, C & D, whose normals point away from each other, is topologically equivalent to the geometry pictured here.*

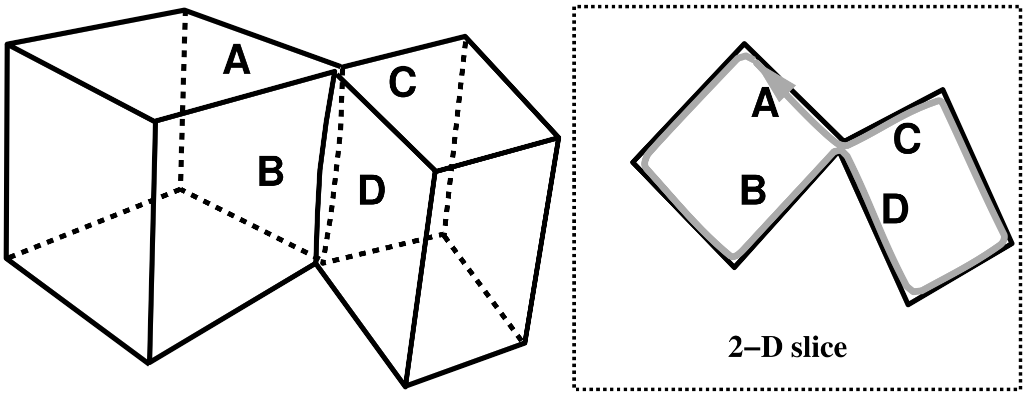


Figure 6.19: *Matching faces B & D, A & C, whose normals point toward each other, is topologically equivalent to the geometry pictured here.*

away from each other, but this choice is somewhat arbitrary. We have chosen it to make the slicer output more consistent in the event that the input contains non-manifold vertices that abut the middles of edges or faces in addition to the pseudo-2-manifold edges. Our algorithm is not affected when a non-manifold vertex touches the middle of another face or edge; our data structure does not capture this adjacency and so a slice through such a vertex will yield a vertex coincident with the boundary of a separate region (see Figure 6.20). Similarly, we arrange it so that a slice through a pseudo-2-manifold edge will also result in coincident points on the boundary of separate regions (as in Figure 6.18).

The end result from the point of view of the edge-uses is separate but coincident 2-manifold edges that have the same vertices as endpoints, each with exactly two edge-uses in opposite directions with mutually referencing sibling pointers. While this matching

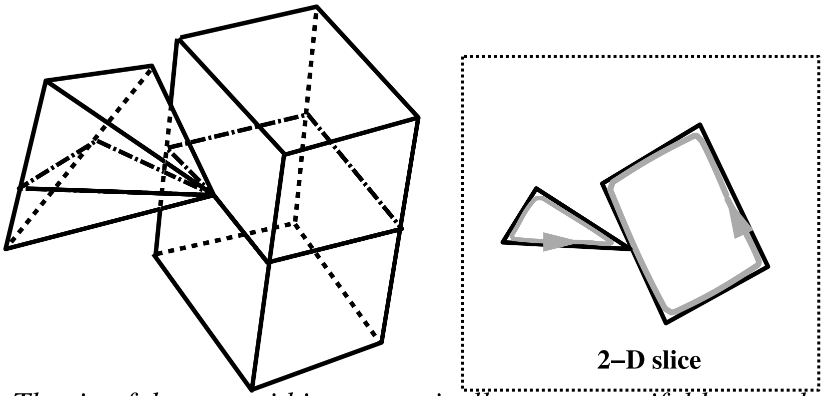


Figure 6.20: *The tip of the pyramid is geometrically a non-manifold vertex because it just touches the cube, but we never capture this adjacency information in the LEDS. A slice through this vertex will yield the two separate contours shown on the right.*

scheme does not give us full symmetry between full and empty regions, it does give us a globally consistent boundary. For example, if the two touching cubes were attached by a solid pedestal underneath, then after the pseudo-2-manifold edge separation the vertex at the top of the edge would have two separate disk cycles around the two cubes, while the vertex at the bottom would have a single disk cycle connected by the faces of the pedestal (see Figure 6.21). This configuration is still consistent even though it is not symmetric. (Conceptually, we assume that all such infinitely thin regions will crack and separate.)

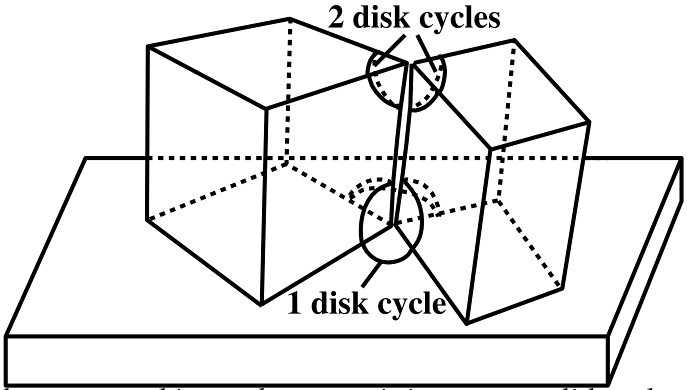


Figure 6.21: *If the two touching cubes are sitting on a solid pedestal, then after we separate the non-manifold edge there will be two disk cycles at its top vertex but only one disk cycle at its bottom vertex.*

Separating Non-2-Manifold Vertices

The second and final step in making our representation pseudo-2-manifold is to locate and separate any remaining non-2-manifold vertices whose edge-uses form multiple disk cycles, such as the middle vertex in Figure 6.22. Unlike edges, vertices are represented explicitly; therefore, we must actually make a duplicate vertex for each additional disk cycle. We wait until after we have separated non-manifold edges because their separation can make duplicating some of their endpoints unnecessary, as is the case with the lower vertex on the pseudo-manifold edge in Figure 6.21, which has only a single disk cycle after the edge has been split.

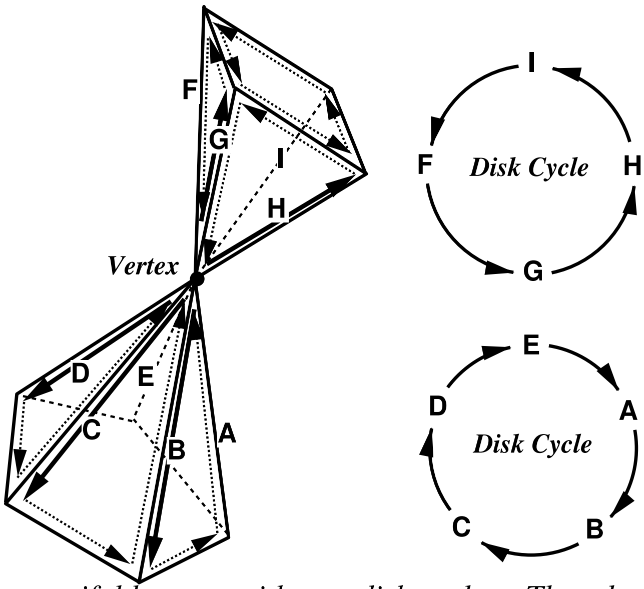


Figure 6.22: *A non-manifold vertex with two disk cycles. The edge-uses are shown as arrows on the visible faces; those that appear in the disk cycles for the center vertex are solid and those that do not are dotted. The disk cycles are pictured on the right.*

In addition to duplicating the non-2-manifold vertices, we need to divide up their edge-uses into disk cycles, updating the edge-uses' vertex pointers to point to the appropriate copy of the vertex and relinking their next vertex edge-use pointers so that they connect edge-uses in the same disk cycle. The algorithm we have developed to identify the non-2-manifold vertices and their disk cycles simultaneously relinks the next vertex edge-use pointers to put them in the order in which the edge-uses appear around the disk cycles, even for manifold vertices that do not need to be duplicated. Unlike the Noodles disk cycle shown in Figure 2.10, we orient our disk cycles clockwise as seen from the exterior of the part for compatibility with our slice data structure.

We begin by marking the edges in the first disk cycle for the vertex. We start with the vertex's first edge-use (**B** in the example shown in Figure 6.23), mark it and follow its sibling's next-in-loop pointer to find the next edge-use in the disk cycle (**C** in this example), and so on around the disk cycle. When we are back to the first edge-use we marked, we have traversed a complete disk cycle.

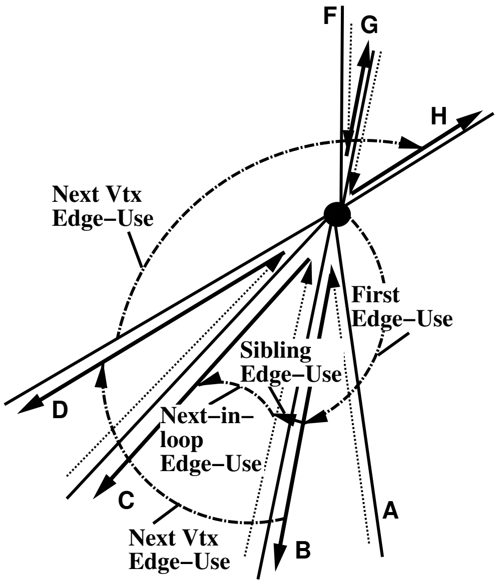


Figure 6.23: *Following LEDS pointers to locate multiple disk cycles.*

Next, we check if there are more disk cycles around the vertex, again starting with its first edge-use (**B**), but this time iterating through the next vertex edge-use pointers. If the edge-use is already marked (as **B** is), we follow its next vertex edge-use pointer to another edge-use for the vertex (**D** in this example), then overwrite the pointer we just followed to point to the next edge-use in the disk cycle instead (e.g. we overwrite **B**'s next in vertex edge-use pointer to point to its sibling's next in loop pointer, **C**). This overwriting will eventually divide the original unordered next vertex edge-use circular list of all edge-uses for the vertex into separated ordered circular lists for each disk cycle. In this example, since **D** is also already marked, we repeat the same step, following its next vertex edge-use pointer to **H** and then overwriting the pointer to point to **E** instead. If we come to an unmarked edge-use that has not yet been included in a disk cycle (such as **H** in Figure 6.23), then we know that this is a non-manifold vertex that must be duplicated. We make a copy of the vertex for this next disk cycle that points to this edge-use (**H**) as its first edge-use. Then we traverse and mark this disk cycle in the same manner as the first one, as described in the previous paragraph, starting from its first unmarked edge-use (**H**). During the traversal, we also update the edge-uses in this new disk cycle to point to the new copy of the vertex. When we have marked all of this new disk cycle's edge-uses, we go back to traversing and updating the original next vertex edge-use ordering where we left off (at **H**, in this example), until we return to the first edge-use from the original vertex (**B**). At this point all of the edge-uses rooted at the vertex will have been marked as included in a disk cycle and their next vertex edge-use pointers will all have been updated to point to the next vertex use in their disk cycle.

Chapter 7

Slicing

In this chapter we describe the design and implementation of a coherent sweep plane slicer which "slices" a tessellated 3-D CAD model into horizontal, 2.5-D layers of uniform thickness for input to layered manufacturing processes.

Previous algorithms for slicing a 3-D b-rep into the layers that form the process plan for these machines have treated each slice operation as an individual intersection with a plane. But for a typical stereolithography build with .005" layers, a 5" high part will be made from one thousand parallel slices with significant coherence between slices, which can be used to calculate neighboring slices more efficiently. An additional shortcoming of many existing slicers that we address is a lack of robustness when dealing with non-manifold geometry.

Our algorithm exploits both geometric and topological inter-slice coherence to output clean slices with explicit nesting of contours. The algorithm relies on the LEDS for the initial topological information. If the solid contains non-manifold edges, we preprocess the LEDS to logically separate them into coincident manifold edges as described in the previous chapter. The main body of the algorithm uses a sweep plane approach, using the connectivity information for the 3-D solid to derive and update the connectivity of the 2-D slices. The resulting slice descriptions are topologically consistent, connected, nested contours, rather than simply unordered collections of edges.

7.1 Sweep Plane Algorithms: Background

The general approach in a sweep plane algorithm involves moving a virtual plane through the input. (For 2-D problems, a sweep line is used in place of a sweep plane.) While the plane moves along, the algorithm maintains a dynamic *status structure*, generally a parameterized representation of the intersection of the input with the current position of the sweep plane. This status information only changes at certain points called *event points*, which are maintained in an event queue. Whenever the sweep plane reaches an event point, the point is processed to update the status structure, and additional calculations, depending on the particular problem to be solved, are performed. Most sweep algorithms are *output sensitive*; that is, the running time increases (ideally linearly) with the amount of output produced, even though the worst case performance may be quadratic in the input size.

For example, a simple sweep line algorithm can be used to find all intersections of a group of line segments in the plane in time $O((n + k)logn)$, where k is the number of intersections discovered, without checking all n^2 possible intersections [17]. In this example, the status structure contains the left to right ordering of the segments intersecting the current position of a horizontal sweep line that moves from bottom to top over the input, and the event points are the segment endpoints and the intersections, which we compute as we go. We insert segments into the status structure (in the appropriate position) when we encounter their bottom vertices and delete them when we reach their top vertices. The additional calculation at all event points is to check for intersections between each pair of segments that have newly become neighbors in the current status structure. These intersection points get inserted in the event queue to be processed when the sweep line reaches them, since the two segments that intersect will switch positions in the status structure as they intersect, giving us new pairs of neighbors to check.

In our slicer, the *event points* are the vertices of the input polyhedron and the z-heights of the slices, and the *status structure* contains all of the edges currently intersected by the sweep plane, stored in circular linked lists according to their connection order in the slice intersection contours.

7.2 Slicing Algorithm: Overview

As initialization for our slicer, we build the LEDS for the input, verify that it describes a closed solid, and make it pseudo-2-manifold, as described in the previous chapters. If necessary, we also rotate the part (by applying a rotation transformation to each of its vertices) in order to align the desired build direction with the positive z axis. In our algorithm, we will move a virtual horizontal sweep plane from the bottom to the top of the input, the direction in which SFF machines typically build parts; therefore, we sort the vertices from least to greatest z-coordinate at the start. The algorithm's "z-events" occur each time the sweep plane hits a vertex (an event that changes the topology of the current slice) and each time the sweep plane hits a height at which we wish to output a slice (an event where we wish to derive the geometry of the current slice).

For a given position of the sweep plane, after we have processed any new vertices that intersect that plane, our status structure will reflect the topology of a slice – the connectivity of its vertices – at the current height, recorded in a list of contours. For each slice contour in the list, the data structure will contain a circular, ordered, doubly-linked list of the edges of the solid intersected by the current sweep plane that determines this contour (see Figure 7.1). An outer contour also points to a list of any inner hole contours nested within it, and an inner hole contour points to the outer contour that contains it.

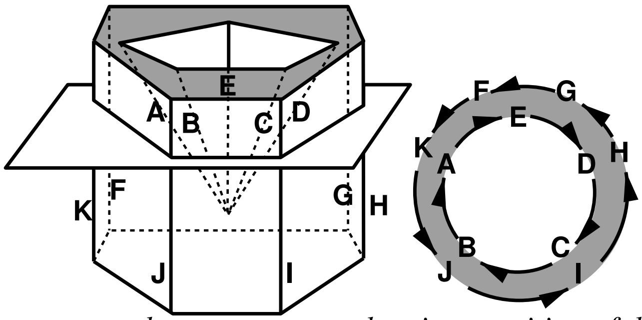


Figure 7.1: *The slice contour data structure at the given position of the sweep plane will consist of the two circular linked lists pictured at right. Their orientation and nesting indicate that the slice contains a single polygon with a hole.*

Each of the currently intersected edges of the solid determines a vertex in the 2-D slice, with the same connectivity as in the contour's circular list of edges. The geometric position of each intersection vertex is derived by finding the point at the current z-height on each edge in the circular list. Each contour's clockwise or counter-clockwise orientation (from the point of view of looking down on the slices) determines whether it is an inner hole contour or an outer contour, respectively. Because we do not need this orientation classification until we output a slice, we wait until the first slice that occurs after the contour is formed to classify it, since we will be calculating intersection point coordinates then anyhow, and those coordinates are needed for deriving the orientation. In addition to contour orientation, we also derive the nesting of contours inside one another at the first slice after their creation. The contour orientations as well as their nesting will remain unchanged until we hit a saddle point (assuming that the input is not self-intersecting), at which point we mark the orientation and nesting invalid and re-derive them at the next slice (see section 7.4).

The algorithm proceeds by moving the plane up to the next z-event. If there are any new vertices at this height, we process them in arbitrary order, using information about the incident edges to modify the appropriate contours, as explained in detail below. If we want to output a slice at this height, then (after first processing any new vertices) we calculate the (x,y) coordinates of the intersections with the edges, derive orientation and nesting for any new or changed contours, and then output the geometry as well as the topology of the slice.

7.3 Vertex Processing

Our first step in processing a vertex is to build an explicit disk cycle for it consisting of a circular, doubly-linked list of pointers to its edge-uses. Pieces of this disk cycle will be incorporated directly into the status structure's slice contour lists, also doubly linked (though all figures show only the "next" arrows of these doubly linked lists).

Next we process the disk cycle we have constructed for the vertex. Each edge-use in the disk cycle is classified as one of two types. Either we have already processed its sibling (and its sibling is in one of the existing slice contours in our status structure), in which case this vertex is the top of an edge-use which will not appear in future slices, and we refer to it as an *ending edge-use*; or it is an edge-use whose sibling we have not seen before, in which case this vertex is the bottom of a new edge-use, and we refer to it as a *beginning edge-use*. For a horizontal edge, the disk cycle for the first endpoint we process will treat the horizontal edge-use as a beginning edge-use, and the disk cycle at the other endpoint will treat its sibling as an ending edge-use. No special case is required for horizontal edges, nor does the order in which we process their endpoints affect the output. Similarly, the order in which we process the disk cycles around vertices that were originally non-2-manifold does not affect the output of the algorithm.

For the first vertex we process, and again at local minima on the part, the disk cycle will consist entirely of beginning edge-uses. For the final vertex and at local maxima, the disk cycle will consist entirely of ending edge-uses. The majority of disk cycles will consist of a mix of beginning and ending edge-uses. We consider each of these three cases separately.

7.3.1 Beginning Vertices

A disk cycle linking all beginning edge-uses is converted directly to a new contour that will appear in slices above the vertex. For the first vertex processed, i.e., the vertex at the bottom-most point on the part, its disk cycle, with its clockwise ordering of edge-uses around the vertex as seen from the outside of the part, gives a counter-clockwise ordering of the edge-uses in the slice contour viewed from the top of the part. This counter-clockwise ordering indicates an outer contour in the slice (see Figure 7.2). This same situation is encountered at any local minimum for geometry with only convex faces. For the case shown in Figure 7.3, the clockwise ordering of edge-uses in the disk cycle corresponds to a clockwise ordering in the corresponding slice contour viewed from the top of the part, indicating a hole in the slice. In either case, the disk cycle becomes a slice contour; we add it to the list of contours and update all of its edge-uses to point to the slice contour that they now belong to (making use of the LEDS edge-use's Extra pointer). The slice contour pointer also serves to flag whether an edge has been seen before: to tell if an edge-use is an ending edge-use we check to see if its sibling's slice contour pointer is set. We do not try to determine if our new slice contour is an outer contour or a hole contour until we reach a height where we need to output an actual slice.

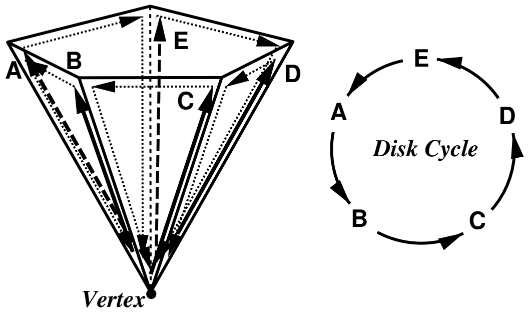


Figure 7.2: *The disk cycle around the indicated vertex contains all beginning edge-uses; it starts a new counter clockwise outer contour in the slice data structure (seen from above).*

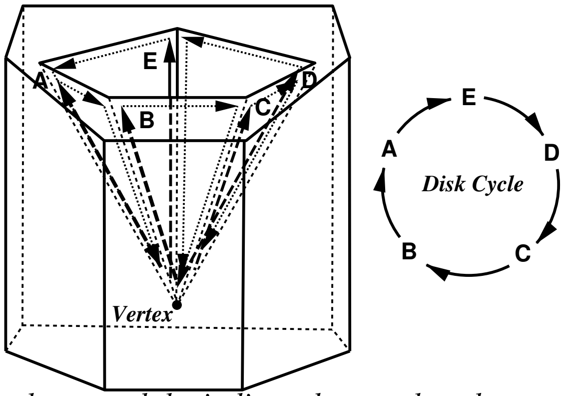


Figure 7.3: *The disk cycle around the indicated vertex has the opposite orientation as the previous example; it starts a new clockwise inner hole contour in the slice data structure (seen from above).*

7.3.2 Ending Vertices

A vertex with all ending edge-uses (we will use a prime (') on edge-use labels in subsequent figures to indicate ending edge-uses) is the analogue of the beginning vertex case. It arises when an existing slice contour in the status structure will disappear from all slices above this vertex. We find the existing slice contour that it matches, which could be either an outer contour or a hole (see Figures 7.4 and 7.5), and delete it from the list of contours. To find the matching existing contour, we look at the slice contour pointer of the sibling of any of the ending edge-uses in the disk cycle; this existing contour will contain the siblings of these ending edge-uses ordered in the opposite sense (clockwise or counterclockwise).

When we delete a contour, it has shrunk to zero area; thus there should be no other contours contained in it, except for the case of a pseudo-2-manifold vertex where any contained contours should disappear at the same vertex when we process their disk cycles. Therefore, we do not need to look at contained contours when we delete a contour. A deleted contour at any vertex may, however, have been contained in another contour; if so, we delete it from that contour's list of contained contours.

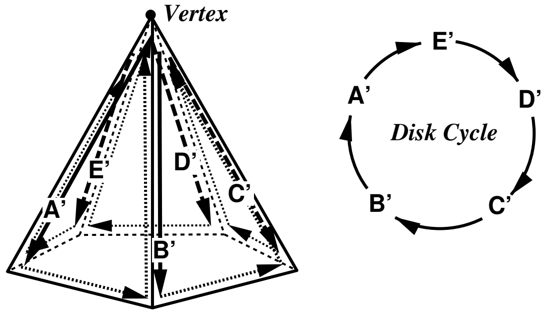


Figure 7.4: *An outer contour is deleted at this vertex where all the edge-uses are ending edge-uses.*

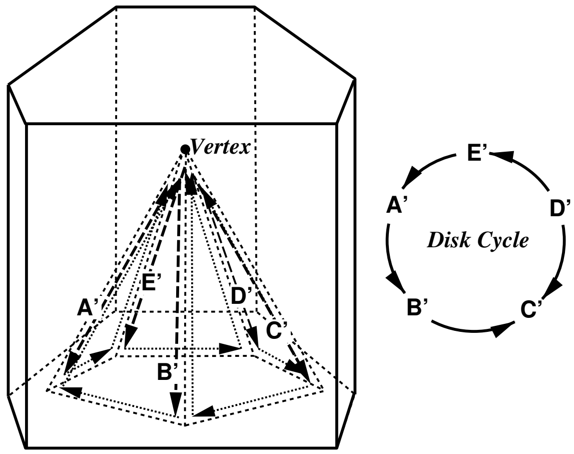


Figure 7.5: *An inner hole contour is deleted at this vertex where all the edge-uses are ending edge-uses.*

7.3.3 Mixed Vertices

When we have both beginning and ending edge-uses in a disk cycle, we use the ending edge-uses to determine where to add the beginning edge-uses to the existing slice contours. First let's consider the simple case illustrated in Figure 7.6, where we start with the slice contour shown at the bottom right, and end up with the contour shown at the top right.

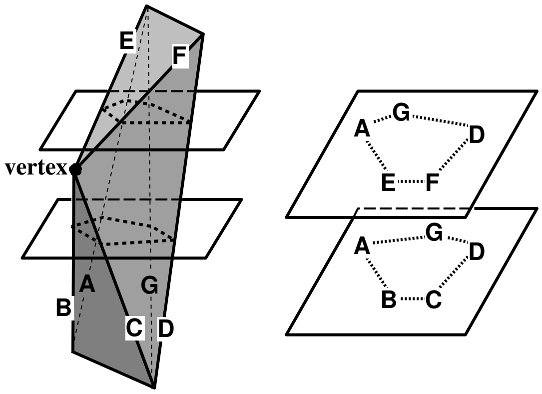


Figure 7.6: *We look at how processing the disk cycle at the center vertex changes the slice contours pictured at right.*

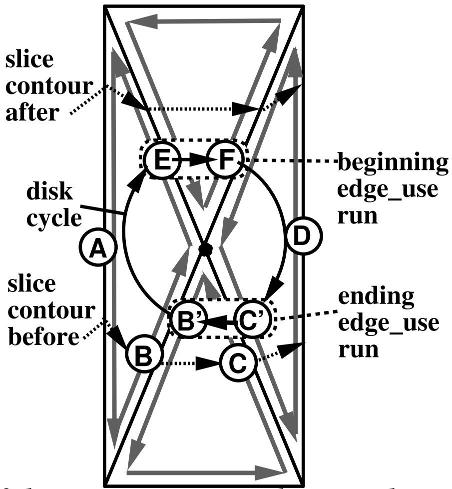


Figure 7.7: *A front view of the same geometry showing the connectivity of the disk cycle in the center (solid black arrows between E, F, C' and D') and the relevant connectivity of the slice contour (dotted arrows) before (below) and after (above) processing the disk cycle. Ending edge-uses are denoted with a prime ('); their beginning edge-use siblings have the same label without the prime. The ending edge-use run (C', B') and beginning edge-use run (E, F) within the disk cycle are demarcated with dashed ovals.*

Below the center vertex, we have the existing slice contour (**A⇒B⇒C⇒D⇒G⇒**). At the center vertex, we process the disk cycle (**C'⇒B'⇒E⇒F⇒**) (see Figure 7.7). This disk cycle contains one *ending run*, **C'⇒B'**, which is a run of consecutive ending edge-uses whose siblings match up with a run of consecutive edge-uses in the existing slice contour (the matching siblings will be in the opposite order, **B⇒C**). The remaining edge-uses in the disk cycle, **E⇒F**, will be replacing the matching siblings to form the new slice contour.

Pointers from the first edge-use in the ending run, **C'**, provide us with the information needed to splice between **D** in the existing slice contour and **F** in the disk cycle. We take the predecessor of **C'** in the disk cycle, **F**, and have it point to **D**, the edge-use following **C'**'s sibling in the existing slice contour. The pseudo-code for this operation follows, where FirstEnd refers to the first ending edge-use in the ending run in the disk cycle:

```
FirstEnd->Previous->Next = FirstEnd->Sibling->Next;
```

Similarly, we follow pointers from the last edge-use in the ending run, **B'**, to splice between **A** in the existing slice contour and **E** in the disk cycle. We take **B'**'s sibling's predecessor in the existing slice contour, **A**, and have it point to **E** instead, the edge-use following **B'** in the disk cycle. The pseudo-code for this operation follows, where LastEnd refers to the last ending edge-use in the ending run in the disk cycle:

```
LastEnd->Sibling->Previous->Next = LastEnd->Next;
```

The piece of the linked list between the first and last edge-uses in the beginning edge-use run, **E**⇒**F** in this case, is retained from the disk cycle and incorporated directly into the updated slice contour. In addition, we make the complementary changes to the corresponding "previous" pointers to maintain the doubly linked lists:

```
FirstEnd->Sibling->Next->Previous = FirstEnd->Previous;
LastEnd->Next->Previous = LastEnd->Sibling->Previous;
```

The final step is to traverse the beginning edge-use run from the disk cycle, **E**⇒**F**, and store with each beginning edge-use a pointer to the slice contour to which it has been added.

Saddle Vertices

For the simple case of a convex part, there is only a single ending run (a run that matches up with a single connected run, in the opposite order, in an existing slice contour); we merely need to replace the corresponding run in the slice contour with a new run of beginning edge-uses from the disk cycle. The beginning edge-uses all get added to the same contour that the ending run was in, and no other edge-uses or contours are affected. For more complicated geometry such as the three-pronged part shown in Figure 7.8, we may encounter multiple ending runs in a single disk cycle (see Figures 7.9 and 7.10). When this occurs, we process the ending runs separately. The splicing code for each is identical to the code for a single end run, but instead of just substituting new edge-uses into a single spot in an existing slice contour, the splices may split apart or merge together existing slice contours; thus the updating of the contour pointers of all the edges in the affected contours will be necessary.

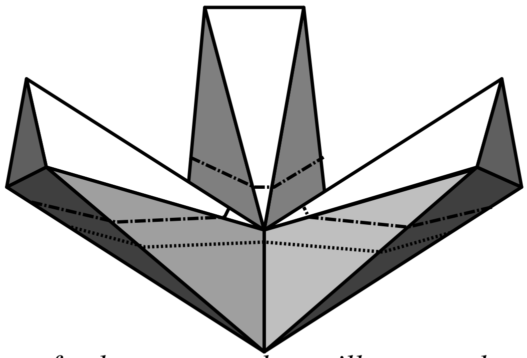


Figure 7.8: *This side view of a three-pronged part illustrates that there is a single contour per slice below the center vertex, as for the lower slice indicated by the lower dotted line, and there are three contours per slice above the center vertex, as for the upper slice indicated by the upper dashed lines.*

In the example shown, a single contour splits into three contours after the three ending runs are processed for the vertex at the center saddle point. If the same part was upside down, we would have three slice contours merging into one at this vertex.

When such splitting or merging occurs, we need to identify which contours have changed, update their edge-uses to point to the contour they now belong to, and update the list of slice contours. We determine if a splicing operation has caused a split or a merge by comparing the old contour pointers of either pair of edge-uses that we've spliced

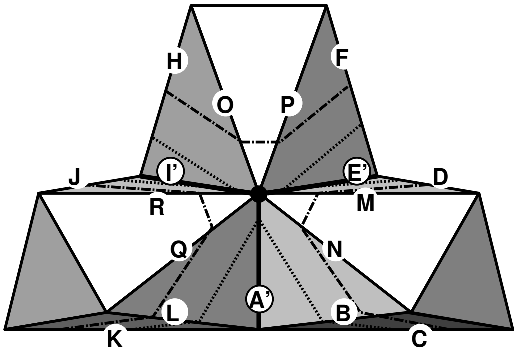


Figure 7.9: *Looking down on this same part, we see the center vertex whose disk cycle has three ending edge-uses (I', E', and A', with black circles around their primed labels). Each of the ending "runs" is one of these edge-uses.*

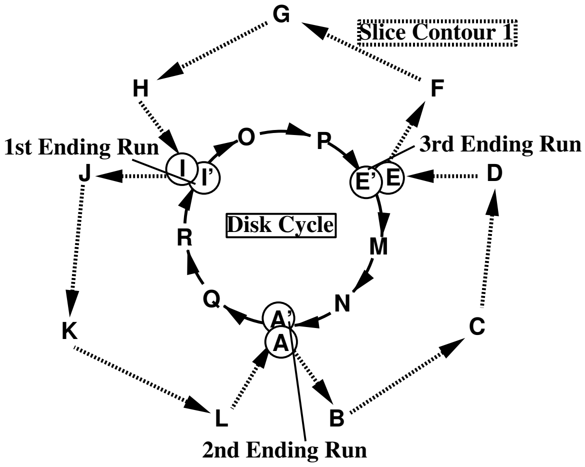


Figure 7.10: *The disk cycle and existing slice contour before we process the center vertex. Ending edge-uses are denoted with a prime (').*

together. If they had different contour pointers before we connected them, then the splicing operation merges these two contours together (e.g. **J** and **R**, the pair spliced around the first edge-use in the first ending run in this example have different contour pointers, as shown in Figure 7.11). If they had the same contour pointer before we connected them, then the splicing operation splits that contour (see Figure 7.12). In either case, in one of the two contours that merged or that resulted from the split, we need to reset the pointer from each edge-use to its new slice contour. We process each ending run, performing its corresponding splicing operation and updating all the edge-uses' contour pointers, based on the connectivity that resulted from the previous splicing operation, before processing the next ending run.

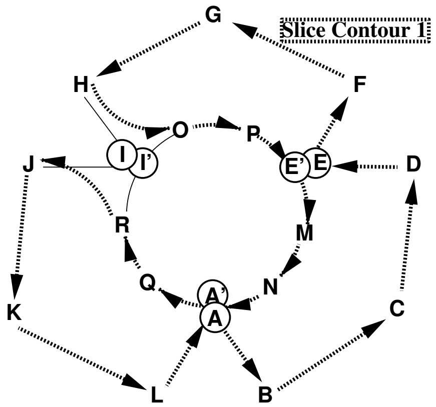


Figure 7.11: *After splicing around the first ending run, I', the disk cycle is merged with the existing slice contour.*

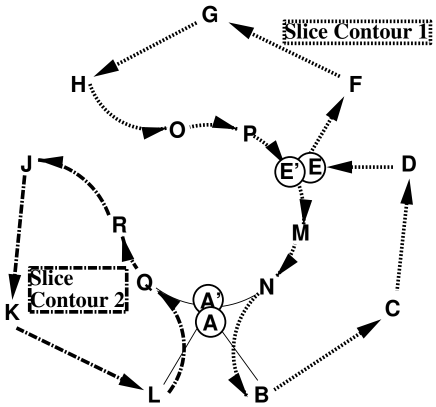


Figure 7.12: *After splicing around the second ending run, A', the slice contour splits into two pieces.*

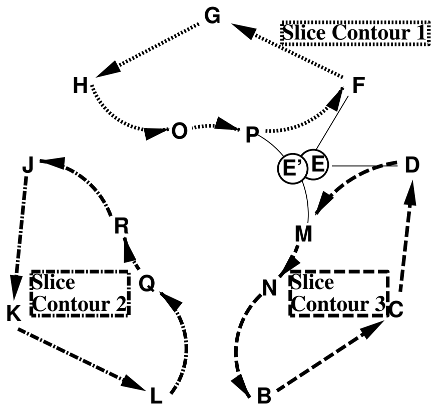


Figure 7.13: *After splicing around the third ending run, E', slice contour 1 splits again.*

The splicing operation for the first ending run in a disk cycle always merges the new disk cycle with an existing slice contour; all of the remaining unprocessed edge-uses of the disk cycle have their contour pointers set to this slice contour (as in Figure 7.11). For the simple case of a single ending run described in the previous section, this is the end of processing. After the first splicing operation, subsequent splicing operations generated by the same disk cycle may cause splits, if other ending runs in the disk cycle match up with edge-uses in this same slice contour (as in Figures 7.12 and 7.13), or they may cause further merges, such as would happen at this vertex if the same part were upside down.

With the upward facing three prong part, an outer contour splits into contours that are themselves outer contours, but a split can also result in a change of contour orientation, as in the part shown in Figure 7.14, where an outer contour splits into one inner and one outer contour. All of these topological changes must be reflected in the slice contour status structure.

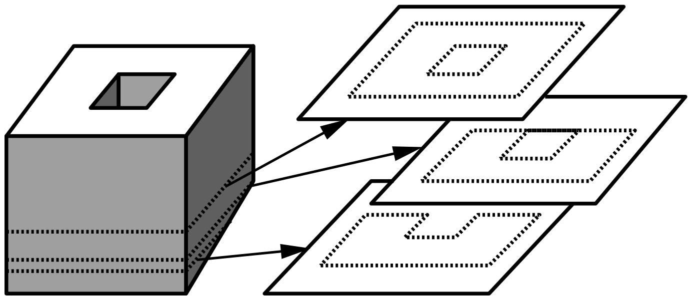


Figure 7.14: *When this part is sliced, an outer contour (bottom right) splits into one inner and one outer contour (middle and top right).*

7.4 Contour Classification and Nesting

7.4.1 Orientation Classification

Before we determine how contours are nested, we determine whether they are outer contours or inner hole contours. This classification is based on whether they are oriented counterclockwise or clockwise as viewed from above. As mentioned previously, we wait until we get to the first slice after the contour is formed to classify it, since at that time we'll be making intersection point calculations anyhow. This classification will remain valid for the contour until it splits or merges, through any number of inserted or deleted edge-uses that do not change its topology.

We cannot determine a contour's orientation by looking merely at the local geometry around one point on the boundary, as the local geometry could be identical for an outer and an inner contour, as shown in Figure 7.15. We can, however, find the contour's orientation

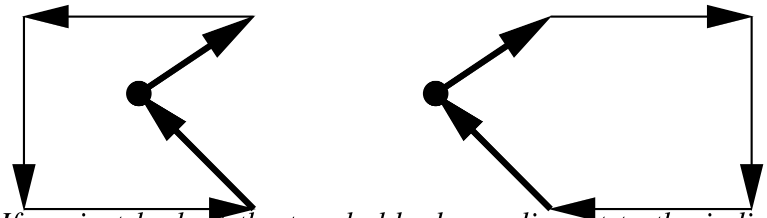


Figure 7.15: *If we just look at the two bold edges adjacent to the indicated points on these two contours, the local geometry is identical. Yet the contour on the left is oriented counter-clockwise while the contour on the right is clockwise.*

using a single cross product if we first find an extreme point on the contour. For example, if we take the right-most point, then we know that the region to its right is outside of the contour; therefore, if we cross the contour to the adjacent region between the two edges that meet at this point, we know that this region is inside of the contour. Then we can take the cross product of these two directed edges to get a normal. By the right-hand orientation rule, if this normal points up out of the plane, then the region inside of the contour contains material and thus the contour is an outer contour; conversely, if the normal points down into the plane, then it is an inner contour.

Occasionally, when there are multiple points in the contour with the same x-coordinate at the right-most extreme of the part, we will have to be judicious about which right-most point we use. We cannot find a plane normal if the adjacent edges are colinear; therefore, if our initial right-most point candidate was between two vertical edges we pick another right-most point. The other problematic case is when the right-most point is non-manifold (see Figure 7.16). In this case, depending on which instance of the right-most point we choose, the zone between its edges may not be adjacent to the region to the right of this right-most point, the region that we knew was outside of the contour. If these regions are not adjacent, then we cannot tell whether the zone between the edges is inside or outside the contour. In this case, we must be sure to pick an instance of the point with at least one of its adjacent edges adjacent to the region to the right of the right-most point (for example,

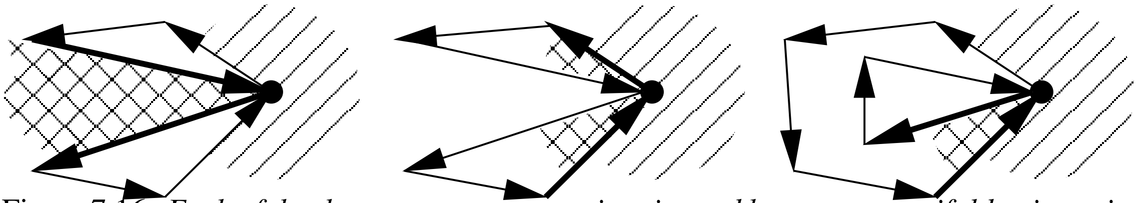


Figure 7.16: *Each of the three contour geometries pictured has a non-manifold point as its right-most point, with identical adjacent edges. (Each example shows a single contour.) If we chose the instance of this point with the adjacent edges indicated by bold lines in the figure on the left, their cross product would tell us whether there was material in the cross hatched region. The cross hatched region is not adjacent to the striped region outside the contour that is to the right of the point; therefore, we do not know if this region is inside the contour (and in fact it is not). Therefore, this cross product does not tell us the orientation of the contour. We must choose the instance of the non-manifold point with the adjacent edges indicated by bold lines in the figure in the middle, since here the cross hatched region between the edges is adjacent to the striped region to the right of the point. Alternately, the edges might be connected so that instead of one region pinching down to zero width at the non-manifold point, two separate pointy regions come together and just touch, as shown in the figure on the right (again showing a single contour). In this case, we could choose either instance of the point, since both of the pointy regions are adjacent to the striped region to the right of the point. The figure illustrates the interior region between the two lower bold adjacent edges.*

by picking the instance with an adjacent edge that is closest to vertical).

If we determine that we have an inner hole contour, we must find the outer contour that contains it (the contour within which it subtracts a hole). If we have any inner hole contours, then we must also determine if any of our outer contours are islands within the holes, and nest them appropriately as well. We derive full nesting by finding the container of each contour whose container is unknown (the query contour), considering each contour with opposite orientation a candidate container. (Other algorithms derive the mutual nesting of an entire collection of contours at once using a sweep line approach [17], but this is inefficient for our application since the majority of our nesting information remains valid across slices, as explained below in Section 7.4.4.)

7.4.2 Bounding Box Containment Test

We use bounding boxes to reduce the problem size for the case of many complex candidate contours. To find the container of an inner hole contour, for example, the first pass of our algorithm compares its bounding box with the bounding boxes of all outer contours. If it is contained in the bounding box of only a single outer contour, then that is its containing contour, shown in Figure 7.17. If it is contained in the bounding boxes

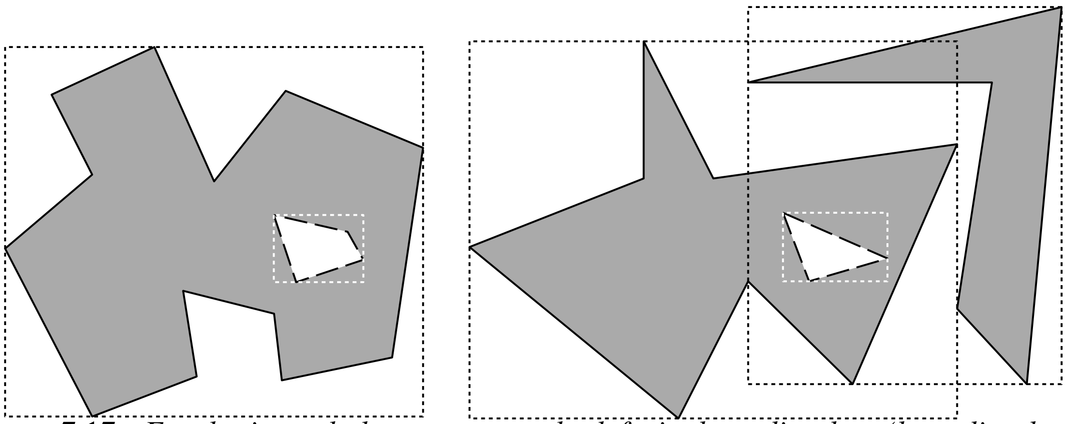


Figure 7.17: *For the inner hole contour on the left, its bounding box (bounding boxes are shown with dotted lines) is contained in the bounding box of a single outer contour, its container. For the inner hole contour on the right, its bounding box is contained in the bounding boxes of two outer contours; thus further tests are needed to find its actual container.*

of multiple outer contours, we use a ray test to determine which of these is the containing contour, as explained below in Section 7.4.3. If the query contour whose container we sought was an outer contour, on the other hand, and only a single inner contour's bounding box contained it, this does not mean that it is contained in this inner contour, because an outer contour may have *no* container. We still use the bounding box test to narrow down the possible containers for an outer contour that we will then have to test with the ray test, even if there is only one candidate.

If the bounding box of a candidate containing contour is coincident with the bounding box of our query contour, it is a possible container, but first we check if the two contours are completely coincident. A pair of coincident contours in alternate orientations cancel each other out; therefore, we do not need to output them (see Figure 7.18). But a valid solid could contain one more coincident contour in one orientation than the other, and in this case we do want to output a contour in the dominant orientation (see Figure 7.19). Multiple coincident contours is the one case where we cannot reliably derive nesting information that will be consistent with the nesting in the next slice, even if no splits or merges occur

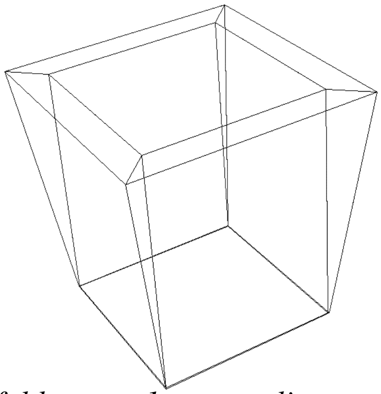


Figure 7.18: *For this 2-manifold, genus 1 part, a slice near the top parallel with the top face will consist of two separate nested contours. But if we slice through exactly at the bottom, the inner and outer contours will be coincident; the hole extends to fill its outer contour. We do not output this zero-area slice. (Alternately, if this shape were subtracted from a larger block, at the bottom we would have an island extending to fill a hole, and again we would not output the hole or island.)*

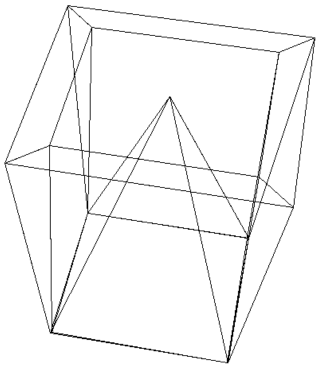


Figure 7.19: *For this pseudo-2-manifold part (a pyramid inside a toroidal frame), a slice through the very bottom will consist of three coincident contours: two outer contours and one inner hole contour. Here we do have material; we output an outer contour.*

during splice operations in between the two slices. Therefore we must mark the nesting information invalid for coincident contours and derive the correct nesting at the next slice.

In the examples pictured here, the contours that are coincident are coincident when they are first created. We can check them for coincidence as part of determining their nesting when we find that their bounding boxes are coincident; the only overhead will be checking for coincidence of other contours when they have coincident bounding boxes. This overhead is necessary because we must rederive the nesting of coincident contours in the next slice.

Coincident contours can also arise, due to floating point roundoff, just below where two contours are about to collapse together and disappear (as might happen in these same parts if they were upside down). We do not check for coincident contours in such cases where the topology has not changed. To do so would require checking every single contour in every single slice, even when there was no need to rederive nesting. If coincident contours arise due to roundoff, the existing nesting is still valid – in fact, it is more correct than the geometry, since if the contours had become truly identical they would have disappeared from the topological slice. Therefore, our output will sometimes contain coincident contours.

7.4.3 Ray Test for Containment

If the bounding box test finds any candidate containing contours for an outer query contour or finds multiple candidate containing contours for an inner query contour, we perform a ray test to find which of these is the query contour's immediate container. We take a vertex of the query contour and use it as the base of a horizontal ray shooting right. We perform an intersection test between this ray and each of the edges in the candidate container contours (the contours with the opposite orientation whose bounding boxes completely enclosed the bounding box of the query contour).

No Vertex on the Ray

Consider first the simple case where the ray does not intersect any edge endpoints. In this case, the first contour that the ray intersects, and which it also intersects an odd number of times, is its container, as shown in Figure 7.20.

Each time the ray intersects a contour, it is transitioning between the inside and the outside of that contour. At infinity, the ray is outside the contour, so if it transitioned an even number of times overall, then it started outside the contour, and if it transitioned an odd number of times overall, then it started inside the contour (see Figure 7.21). If there are multiple contours that it started inside (i.e. that it crossed an odd number of times), then these contours are mutually nested; the first one that the ray intersected is its immediate container (see Figure 7.22)

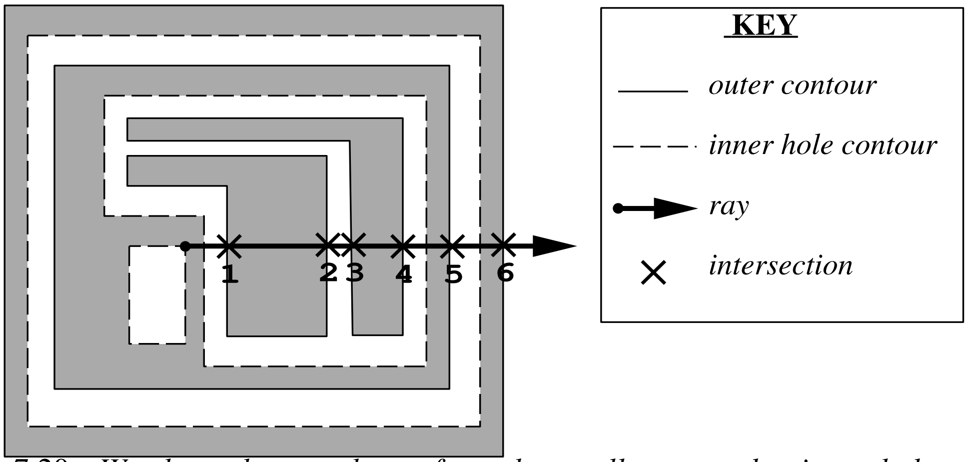


Figure 7.20: *We shoot the ray shown from the small rectangular inner hole contour. It has the six intersections shown with edges of outer contours (note that we do not calculate intersections with other inner hole contours, because they cannot be containers). Intersections 1-4 are with contours that the ray intersects an even number of times; thus they are not containers. Intersections 5 and 6 are with contours that the ray intersects an odd number of times; thus they contain our query contour. Intersection 5 comes first; thus its contour is the immediate container.*

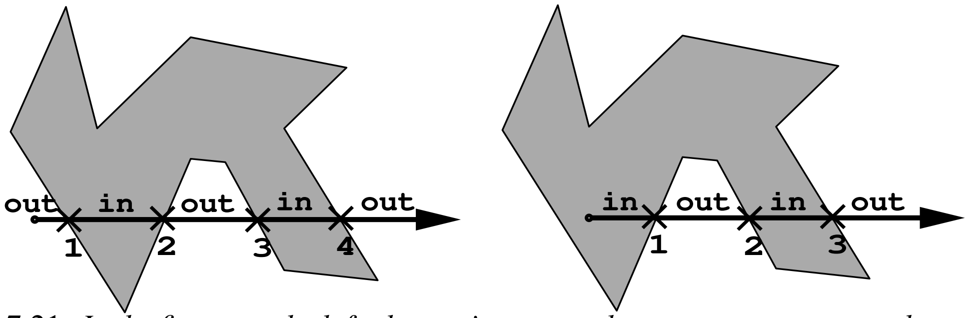


Figure 7.21: *In the figure on the left, the ray intersects the contour an even number of times because its base point was outside the contour. In the figure on the right, the ray intersects the contour an odd number of times because its base point was inside the contour.*

We could shoot our ray in any direction, but for faster tests, we always shoot our ray in the positive x direction. This makes testing whether a vertex is on, above, or below the ray a simple matter of comparing coordinate values. We can use these sidedness tests to avoid the relatively expensive edge intersection calculation in many cases: a) if one of the edge's endpoints is on the ray (in which case it defines the intersection), or b) if both endpoints are above or both below the ray, or c) if both endpoints are to the left of the ray's base point. In cases b and c, there is no intersection, as shown in Figure 7.23. These sidedness tests are

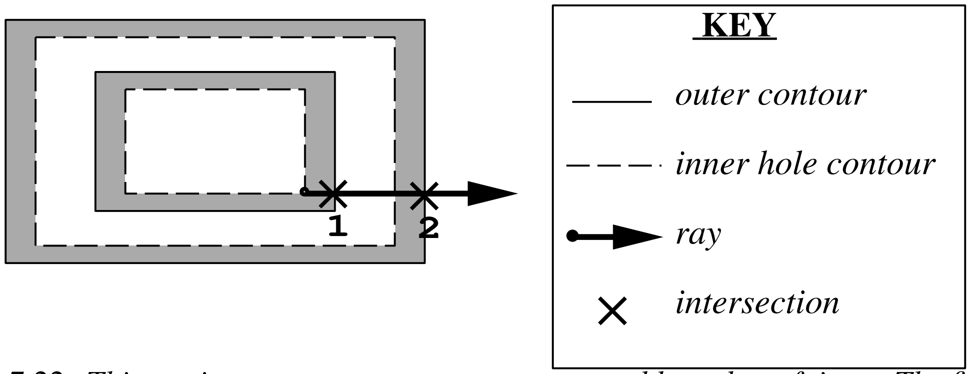


Figure 7.22: *This ray intersects two outer contours an odd number of times. The first one it intersects is the immediate container of the innermost hole contour from which the ray originates.*

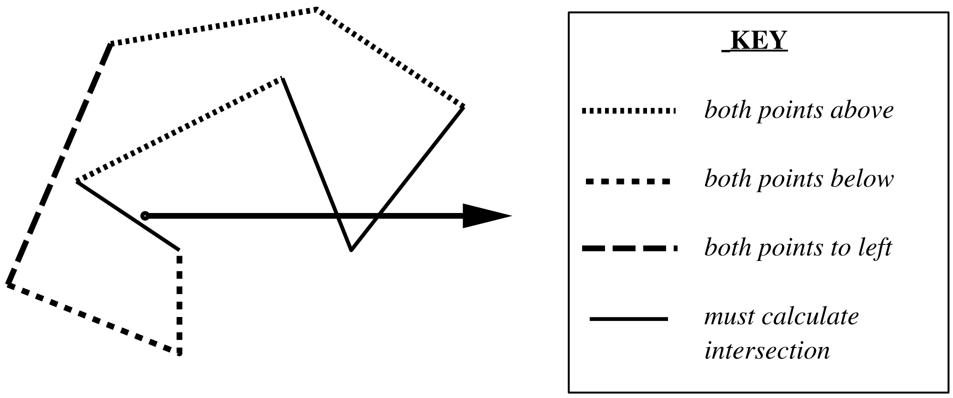


Figure 7.23: *Edges for which we can avoid performing an intersection calculation between the edge and the ray are indicated by dashed and dotted lines.*

also used for determining whether to count an intersection (for determining whether there are an odd or even number) when an edge's endpoint is on the ray, as described below.

In the examples above, none of the vertices of the candidate contour were on the ray. How do we count intersections when they are? If we just looked at individual edges in isolation, we would count an intersection with both the edges that shared the endpoint on the ray. But we only want to count one intersection at this point between the two edges if one is above and the other below the ray. We cannot merely consider the vertex to be part of one of the edges adjacent to it and not the other edge, since we do not want to count *any* intersections in the case where both edges were below or both above and just touching the ray at this point. Similarly, horizontal edges that lie on the ray should not directly affect the intersection count.

Vertex on the Ray But Not on Ray Base Point

In this section, we describe how to determine intersections in the case where a candidate contour vertex is on the ray, but not on the starting point of the ray (call its coordinates R_x, R_y). The technique described here also assumes that if there are any number of horizontal edges adjacent to this vertex in the candidate contour, that they do not intersect the ray base point either. (Both of these exceptions will be addressed by a different test described in the next section.)

We take into account the direction of each edge of the oriented candidate contour. If an edge's first endpoint has $y = R_y$ (this includes horizontal edges), we do not count it as intersecting the ray, in order to avoid double counting intersections. Otherwise, if an edge's second endpoint has $y = R_y$, we take the edge as the first in a pair of edges that we will test to determine whether to count an intersection with the contour at this spot. We take the next non-horizontal edge in the contour as the second edge in the pair. Both edges in this pair will have exactly one endpoint with $y = R_y$ (in fact, it will be the same exact point if there were no horizontal edges between them).

If the second edge's R_y-height endpoint has $x < R_x$, or if one R_y-height endpoint has $x > R_x$ and the other R_y-height endpoint has $x < R_x$, then this is a case where a neighboring edge does intersect the ray base and we must use the test described in the next subsection instead. If both are strictly to the right of the base of the ray, and the edge pair's other two endpoints are on opposite sides of the ray, then we count one intersection for the pair. However, if both the edges' y_R-height endpoints are strictly to the right of the base of the ray, and the other two endpoints are both above or both below the ray, then we do not count this as an intersection (see Figure 7.24). If both of the edges' y_R-height endpoints are strictly to the left of the base of the ray, then neither is on the ray and we count no intersection, regardless of the position of their other endpoints.

The idea is that if the contour is crossing over the ray, we want to count an intersection, but if it merely glances the ray and then turns back around, we do not want to count an intersection (or alternately, we could count this second case as two intersections, since ultimately we just look at whether the total number of intersections is odd or even).

There is a potential ambiguity with our algorithm as described above: there may be

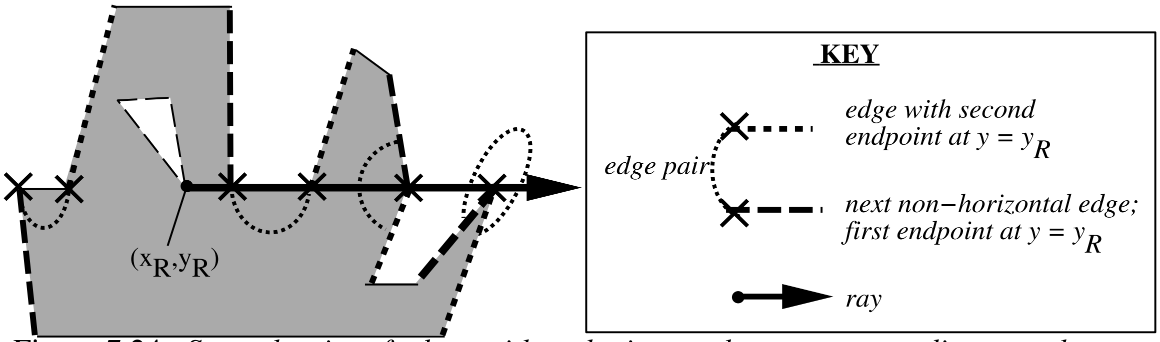


Figure 7.24: *Several pairs of edges with endpoints at the same y-coordinate as the ray, y_R, are indicated by dotted and dashed bold lines with crosses on the endpoints at that y-coordinate. The leftmost pair's y_R-height endpoints are both to the left of the base of the ray; thus they do not intersect it. The other pairs have both y_R-height endpoints to the right of the base of the ray; thus we look at the positions of their other endpoints. For the second and fourth pair (counting from the left), the other endpoints are on the same side of the ray; thus we do not record an intersection for these pairs. For the third pair, the other endpoints are on opposite sides of the ray; thus we record an intersection. There is one intersection for the contour, an odd number; thus the contour is a container.*

multiple "closest" intersections if multiple contours with an odd number of intersections intersect the ray at the same spot, as illustrated in Figure 7.25.

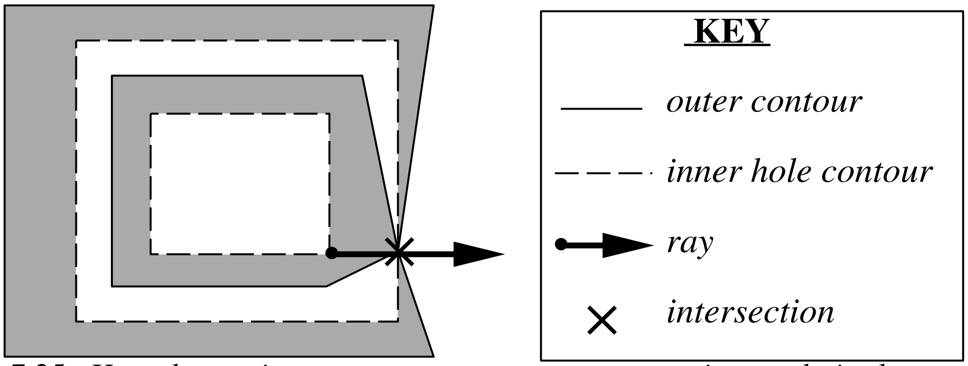


Figure 7.25: *Here the ray intersect two outer contours, one time each, in the same spot. There is no way to tell which is the innermost container from their intersections with this ray; therefore, we compare their areas.*

In the case where these multiple coincident intersections are not on the ray base, all of these multiple contours with odd intersections are containers, and we just need to determine which is the immediate container. For valid input, slice contours cannot actually intersect (only coincide at some or all points). Therefore, if our query contour is inside all of these candidate containers, they must be mutually nested, and it is the innermost that is the immediate container. The innermost can be determined simply by finding the one with the smallest area. (An analysis of the relative ordering of the edges emerging from this joint intersection point would lead to many special cases.)

Intersection Calculation for Edges on Ray Base

The intersection counting technique described in the previous section (checking whether the other endpoints of a (non-horizontal) edge pair with endpoints on the ray are on opposite sides of it) will not work if either of the edge's y_R-height endpoints are on the ray base, or if one is to the left of and the other to the right of the ray base, or for any edge that intersects the ray base exactly, as illustrated in Figure 7.26.

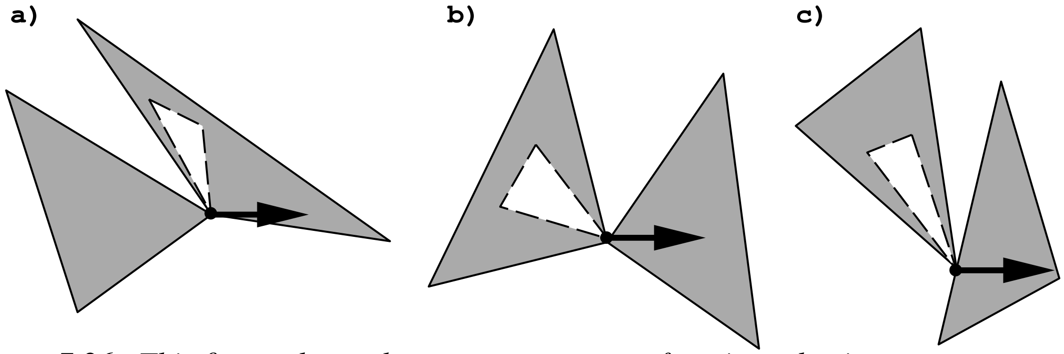


Figure 7.26: *This figure shows three separate cases of a triangular inner query contour inside of one triangular outer contour and adjacent to another triangular outer contour. In a), we should not count the ray as intersecting either of the outer contours at the ray base (even though for each candidate, their pair of adjacent edges are on opposite sides of the ray). In b), however, we should count the ray as intersecting both outer contours at the ray base (a different result even though again each candidate's pair of adjacent edges are on opposite sides of the ray). In c), we should count the ray as intersecting the left outer contour but not the right outer contour at the ray base.*

To determine if we should count an intersection with a candidate contour at the ray base, we need to look not only at the geometry of the candidate contour but also the geometry of the query contour. We need to examine the radial sequence of the pair of candidate contour edges and the pair of query contour edges with the ray base as an endpoint. If we have an edge of a candidate contour (including horizontal edges) that intersects the ray base somewhere other than at its endpoint, we treat it as if there were another vertex at the intersection point, dividing the edge into two edges with endpoints at the ray base. Note that if the ray base is on a horizontal edge in the candidate contour, we will not be looking at the same pair of candidate contour edges that we tested above.

For this test, we choose the ray base to be the rightmost point on the query contour to ensure that this ray points to the outside of the contour. We imagine sweeping the ray radially around its base, starting from the $+x$ direction and proceeding counterclockwise. If, and only if, the radially sweeping ray crosses first a candidate container edge, followed by the two query contour edges, and finally the other candidate contour edge, then we count the ray as intersecting the candidate contour. This order of edges demonstrates that the candidate contour is outside of the query contour and contains it at the ray base; therefore, the ray from the query contour intersects the candidate contour. If the radially sweeping ray crosses both the candidate contour edges before either query contour edge, or both candidate contour edges after both query contour edges, then this is the same case we saw above of the candidate contour just glancing the ray, and there is no intersection. If the radially sweeping ray crosses first a query contour edge, followed by both candidate contour edges, followed finally by the other query contour edge, then the candidate contour is inside of the query contour; thus the ray from the query contour does not intersect it. (And if the sweeping ray alternates between crossing the candidate and query contour edges, that means that the two contours intersect and the input was invalid.)

To implement the radial ordering test, we calculate the angles that both candidate contour edges and both query contour edges make with the ray. We take the two edges from the candidate contour that meet at the ray base, and treating them as vectors with their bases at the ray base, find the angle α, $0 \leq \alpha < 360$ degrees that each makes with the ray. Similarly, treating the two edges of the query contour that meet at the ray base as vectors with their bases at the ray base, we find the angle β, $0 \leq \beta < 360$ degrees that each of them makes with the ray. Taking the smaller of the two β angles from the query contour, $\beta_<$, we look to see if there were any vectors from the candidate contour between it and the ray. If both $\alpha_< \leq \beta_<$ and $\alpha_> \leq \beta_<$, then we count no intersection (as shown in Figure 7.27 a). If there is no $\alpha \leq \beta_<$, then we count no intersection (as shown in Figure 7.27 b). Otherwise, if $\alpha_< \leq \beta_<$ and $\alpha_> \geq \beta_>$, then we count an intersection (as shown in Figure 7.28 a and b).

Note that if $\alpha_< = \beta_<$ and $\alpha_> = \beta_>$, we count an intersection; if this candidate contour

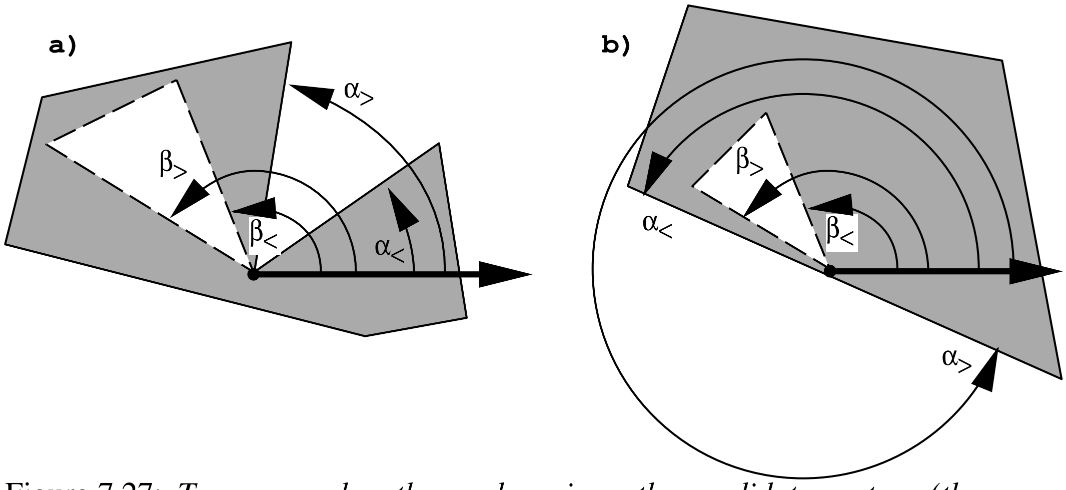


Figure 7.27: *Two cases when the ray base is on the candidate contour (the query contour is the hole contour denoted with dashed lines). In a) both $\alpha_< \leq \beta_<$ and $\alpha_> \leq \beta_<$; therefore, there is no intersection at the ray base. In b) there is no $\alpha \leq \beta_<$; therefore, there is no intersection at the ray base.*

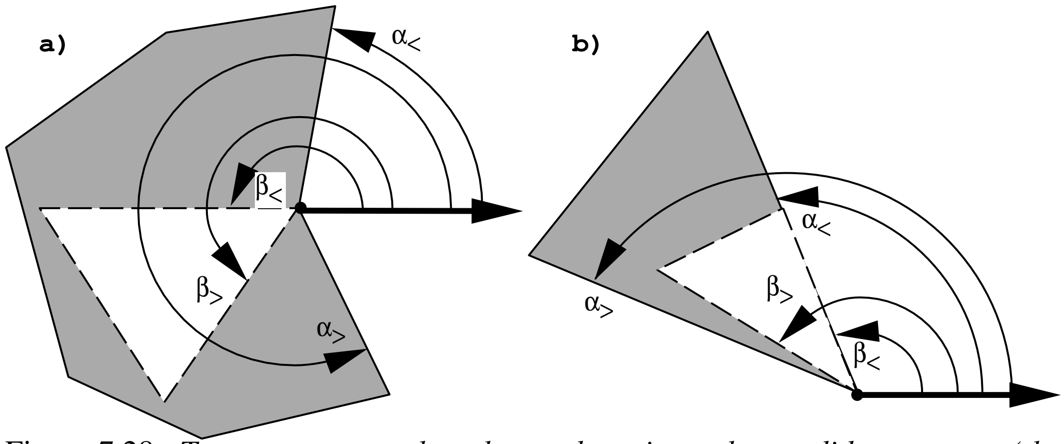


Figure 7.28: *Two more cases when the ray base is on the candidate contour (the query contour is the hole contour denoted with dashed lines). In a) there is exactly one $\alpha \leq \beta_<$; therefore, the ray intersects the candidate contour at the ray base. In b) there is one $\alpha = \beta_<$; again the ray intersects the candidate contour at the ray base.*

has an odd total number of intersections, then the candidate could be either a container of the query contour, or it could be contained inside of the query contour, as shown in Figure 7.29. Using the technique described above to differentiate coincident closest intersections with contours, we calculate the areas of the contours to differentiate these two cases. If the candidate contour's area is smaller than the query contour's area, then we remove the candidate from consideration as a container.

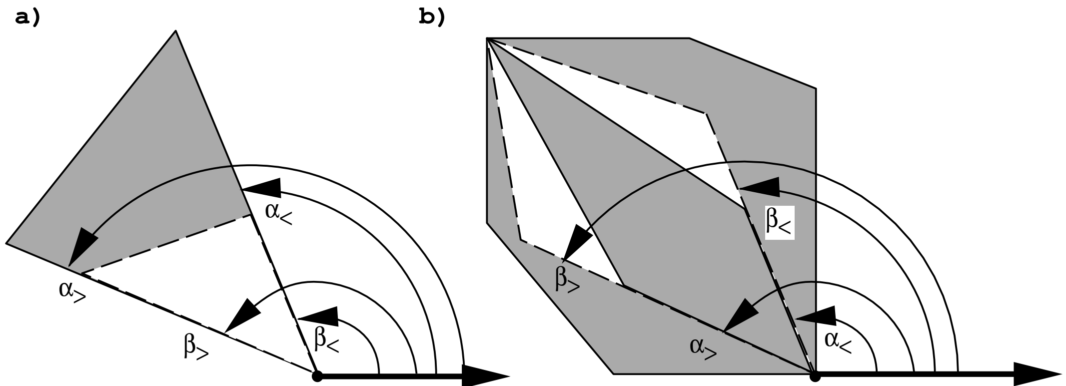


Figure 7.29: *In a)* $\alpha_< = \beta_<$ *and* $\alpha_> = \beta_>$ *and the candidate contour contains the query contour. In b)* $\alpha_< = \beta_<$ *and* $\alpha_> = \beta_>$*, but the innermost candidate contour being tested is inside of the query contour.*

In addition to this ambiguity as to which of two coincident contour and query edges the radially sweeping ray encounters first (resolved with the area test), it is ambiguous whether an edge exactly on the ray is the first or the last edge that the sweeping ray encounters. We resolve this ambiguity by looking at the next (in the direction away from the ray base, regardless of the contour's orientation) non-horizontal edge in the contour. If this next edge is above the ray, we treat the horizontal edge as the first edge the ray encounters on its counter-clockwise sweep, at an angle of 0. But if the next edge is below the ray, then we treat the horizontal edge as the last edge that the ray encounters (assigning it the angle 360 instead of 0). Two examples are illustrated in Figure 7.30.

Non-Manifold Query Contour Containment

The remaining case to consider is that of slice contours that are themselves non-manifold. In a non-manifold potential container, the different pairs of edges adjacent to the non-manifold point will be considered separately; the non-manifoldness will not affect the calculations. For a non-manifold query contour, our nesting algorithm will still find the correct container, but some of the intermediate results could change if the ray base is on a non-manifold point in the query contour. If none of the candidate contours touch at the non-manifold point, the intersection calculations will not be affected. If candidate contours touch at this point, the angle test may return a different number of intersections

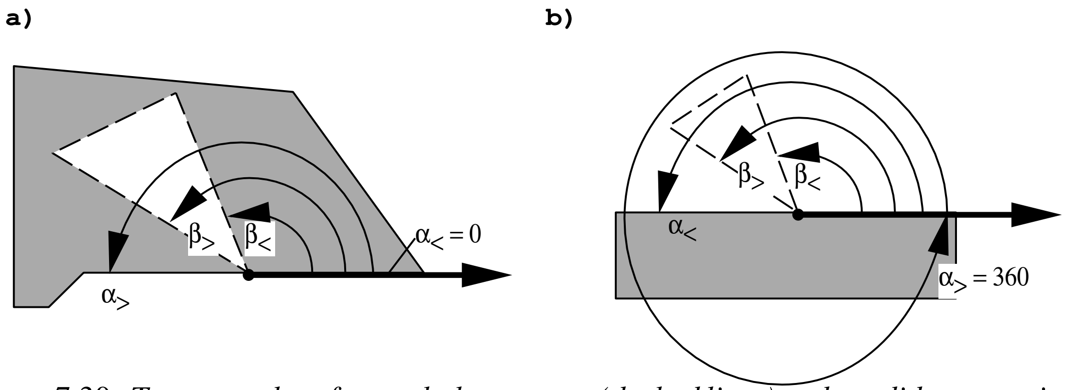


Figure 7.30: *Two examples of query hole contours (dashed lines) and candidate containers with horizontal edges at the ray base. In a), the next edge after the edge of the candidate contour on the ray is above the ray; therefore, the angle is 0 and we count an intersection. In b), the next edge after the edge of the candidate contour on the ray is below the ray; thus the angle is 360 and there is no intersection.*

depending on which instance of the non-manifold point was chosen for the ray base, as shown in Figure 7.31. (If we are not using the angle test, the intersection test will be unaffected since the neighbors of the ray base in the query contour are never examined.) In

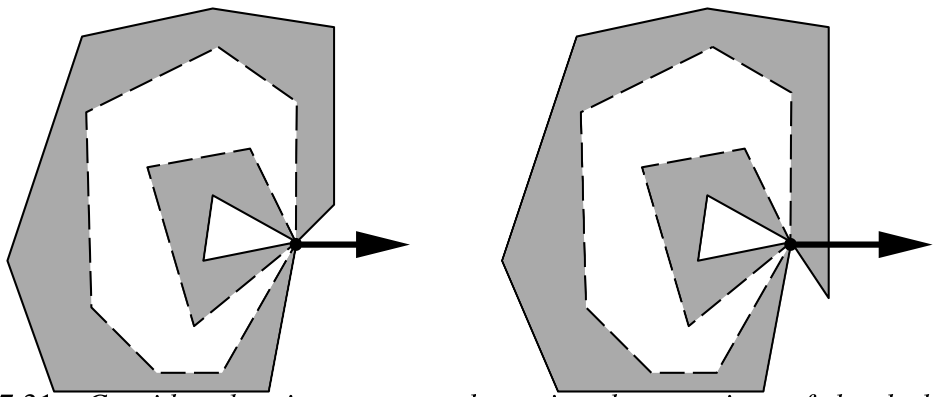


Figure 7.31: *Consider shooting a ray to determine the container of the dashed hole contour in these two instances of nested, non-manifold contours. In the instance on the left, we calculate a single intersection, whether our ray base is the vertex instance on the hole contour's upper part or lower part. In instance on the right, we calculate either one intersection or three intersections, depending on whether our ray base is the vertex instance on the hole contour's upper part or lower part.*

this example, the non-manifold point appears twice in the query contour; we may be using either of these instances (and their adjacent edge vectors) to calculate the βs. For certain geometries we will get a different answer for the number of intersections with a candidate contour depending on which instance of the non-manifold point in the query contour is chosen, but whether the number of intersections is odd or even does not change (as shown in Figure 7.32).

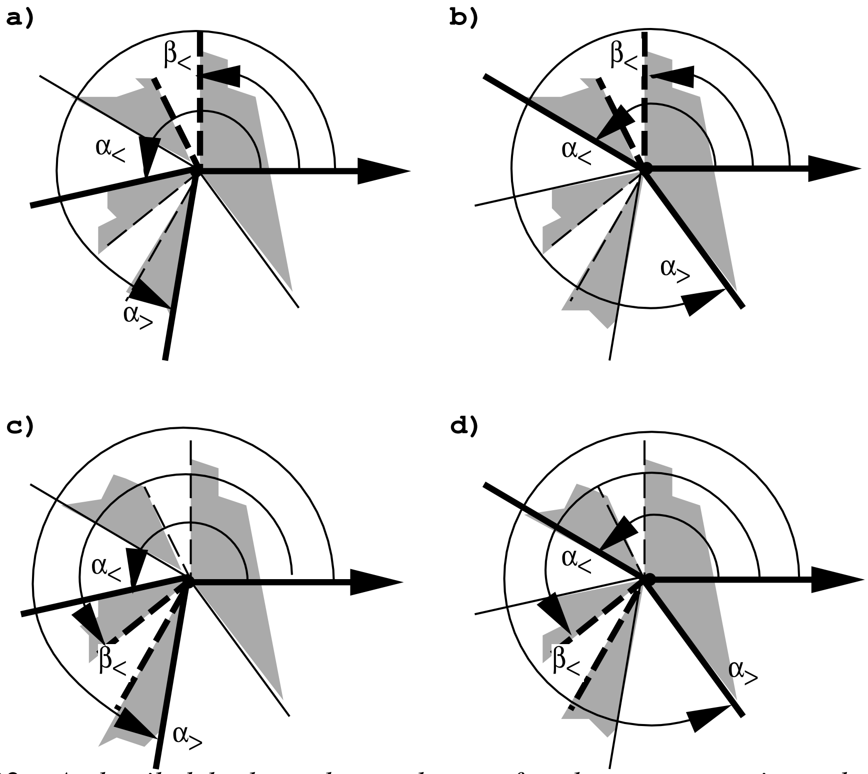


Figure 7.32: *A detailed look at the angle test for the contours pictured at right in Figure 7.31. If the ray base is the instance on the hole contour's upper part as pictured in a) and b), then $\beta_<$ is less than both $\alpha_<$ and $\alpha_>$ for both the pieces of the candidate contour that intersect it at the ray base. The angle test gives us no intersections (and there is one additional intersection not at the ray base). If the ray base is the instance on the hole contour's lower part as pictured in c) and d), then $\beta_<$ is between $\alpha_<$ and $\alpha_>$ for both the pieces of the candidate contour that intersect it at the ray base. The angle test gives us two intersections (and again there is one additional intersection not at the ray base). In both cases, the number of intersections at the ray base is even and the total number of intersections is odd, because the outer contour contains the inner hole contour.*

Note that depending on which instance of the non-manifold point in the query contour we choose as the ray base, the position of the first intersection with an odd-intersections candidate contour may also change (as it does between Figure 7.32 c and d). But the immediate container will always be intersected before or at the same time as any other odd-intersecting candidate contours (it could only be otherwise if the odd-intersecting candidate contours, and hence the original input polygons, intersected).

Alternate Ray Shooting Strategies

We could have avoided using the angle test entirely if we could guarantee that our ray base did not lie on any other contours. Two alternate approaches that try to guarantee this are described below.

The first alternate approach is to try to find a point on the boundary of the query contour that does not touch any other contour. Note that there may not be any vertex that satisfies this property (see Figure 7.33a). Furthermore, there may not be any point on the boundary that is not touching some other contour (see Figure 7.33b). But for each

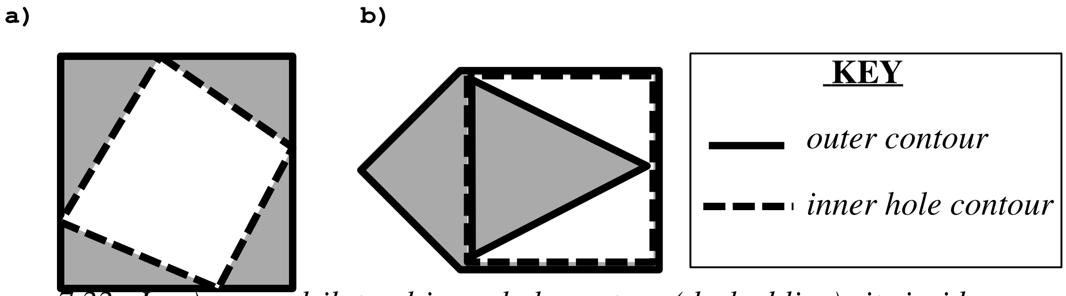


Figure 7.33: *In a), a quadrilateral inner hole contour (dashed line) sits inside a square outer contour (solid line). All of the vertices of the inner hole contour are coincident with the outer contour. In b), three sides of a square inner hole contour are coincident with three sides of a containing pentagonal outer contour, and the square's fourth side is coincident with the side of a triangular outer contour which is an island within it. Every point on the edges of the inner hole contour is coincident with one of the two outer contours.*

individual candidate container, there will almost always be some point on the boundary of the query contour that does not touch that candidate contour, though it may be a different point for each candidate. We could test whether a given candidate is a container of the query contour by counting intersections with a ray shot from the point that is non-coincident with that candidate, and then would not need to use the angle test. Repeating for all of the candidate contours using a different ray for each, from a boundary point non-coincident with respect to the contour being tested, we would find which were containers. We cannot find the immediate container based on the closest intersection point since the intersections are with different rays, but we can use the area test comparison to find which of these containers has the smallest area and thus will be the immediate container.

One disadvantage of this approach is the complexity of finding an appropriate ray base for each of the candidate contours. In the worst case this will require comparing each of the query contour edges against all edges of candidate contours looking for intersections. It will also require the relatively expensive area test for multiple containing contours in all cases, not just when the containing contours touch each other at their first intersection with the ray. And finally, for certain degenerate cases, as shown in Figure 7.34, there will be no point on the boundary of the query contour that is not also on the candidate contour.

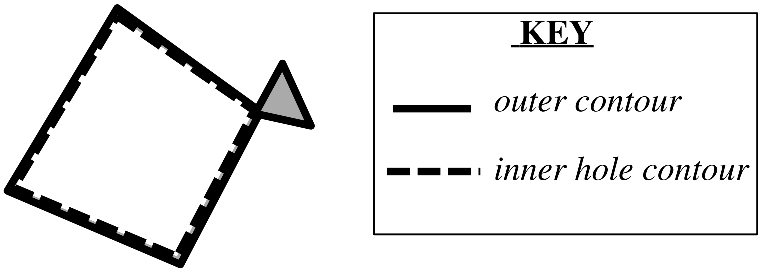


Figure 7.34: *This quadrilateral inner hole contour (dashed line) contains no points that are not coincident with its non-manifold containing contour (solid line).*

This suggests a second alternate approach of avoiding angle tests by shooting the ray from a point inside of the query contour. Then we could use the same ray for all candidate contours, and the complexity of finding this ray would be independent of the number or size of the candidate contours. We would still need the area test to resolve situations where the ray's closest intersection point with a container was on more than one container. A simple approach to finding the necessary interior point is to triangulate the contour and take the centroid of one of the triangles. We do not need to fully triangulate the contour, of course; we could modify the standard y-monotoning polygon triangulation algorithm [55] to just find the first triangle.

Of course, if the query contour had oppositely oriented contours inside of it (which we may not yet know about), then this ray might intersect them; therefore, we must use the area comparison test to determine which contours are containers and which are inside. A more serious shortcoming is that the interior point might be on the boundary of an oppositely oriented contour. For a true interior point, this would indicate that the candidate contour was inside and thus we could ignore it for the purpose of finding the query contour's container. But for an input contour that had barely any area, there might not be any truly interior point that is representable with the precision available (see Figure 7.35). Then the "interior" point we chose, rounded off, could lie on an oppositely oriented exterior contour. In this case, we would be obliged to shoot our ray from a vertex point on the query contour after all.

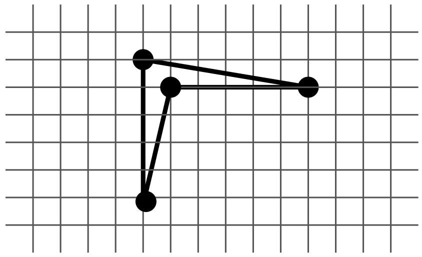


Figure 7.35: *If the grid shows the underlying precision of the representation, then this contour has no interior point that can be represented at this precision.*

The advantages of both of these approaches is that they are less sensitive to roundoff errors in floating point calculations than the angle test. The second alternate algorithm will be faster than the first alternate algorithm, but in some cases neither will work, so we would still need to implement a third algorithm such as the angle test to handle those cases, adding unwelcome and unnecessary complexity to our code. For these reasons, we use the ray shooting algorithm originally presented, combined with the angle test and the area test when necessary, to determine the container when bounding boxes alone do not suffice.

7.4.4 The Persistence of Nesting

The nesting of contours, like their orientations, remains unchanged during the processing of the disk cycles of most of the vertices of the original solid. Merely inserting or deleting a single run of edge-uses from a slice contour will not affect its nesting. It is only when existing slice contours split or merge that their nesting can change (given that our input is non-self-intersecting). For this reason, when we have multiple ending runs in a vertex disk cycle (indicating splitting or merging will occur), we delete all of the nesting information associated with the existing contours that match up with the disk cycle. Not only must we delete the nesting data stored with the contours themselves, but we must also eliminate references to them in the nesting data of any other contours that contain them or are contained in them.

The nesting is then re-derived at the next physical slice. An inner contour's nesting will only have changed if its containing contour has been split or merged. We can identify such inner contours because their container pointer will be empty (as described in the last paragraph, the container's split or merge would cause the contained inner contour's container pointer to be zeroed). In the example shown in Figure 7.36, when the rectangular outer contour in the left slice splits into the two rectangular outer contours in the right slice, only its (rectangular) inner contours need to re-derive their nesting. None of the circular inner contours' nestings are affected.

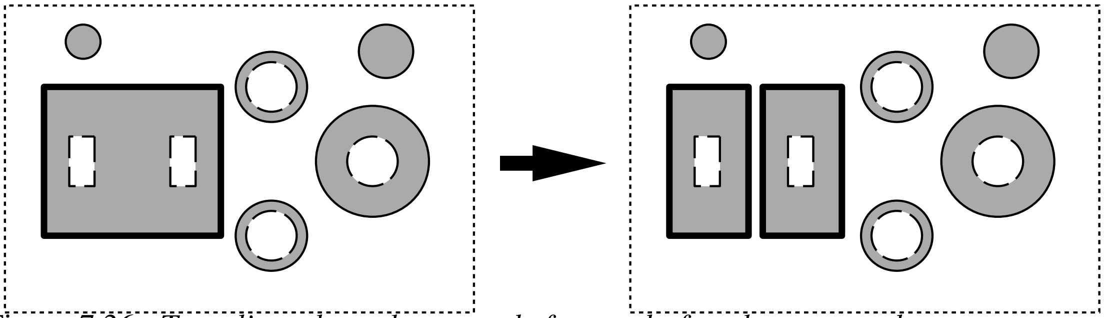


Figure 7.36: *Two slices through a part, before and after the rectangular outer contour in the left slice splits into the two rectangular outer contours in the right slice. Only the rectangular inner contours, which were inside the contour that split, need to re-derive their nesting. The circular inner contours, whose outer contours topology is unchanged, will have their nesting unchanged.*

On the other hand, if an outer contours splits into a new inner contour nested within an outer contour, then the nesting of *all* outer contours must be re-derived. This is because any outer contour could now be an island within the new inner hole contour, as shown in Figure 7.37.

As mentioned above, the persistence of much of the nesting information across slices affects our approach to deriving the nesting. If the nesting changed completely between each slice so that we had no prior relevant information about it, then we might be better off using a global algorithm that found the mutual nesting information for all contours at the same time. But since the nesting can change incrementally, we use an algorithm that finds the container of each individual contour using a separate calculation.

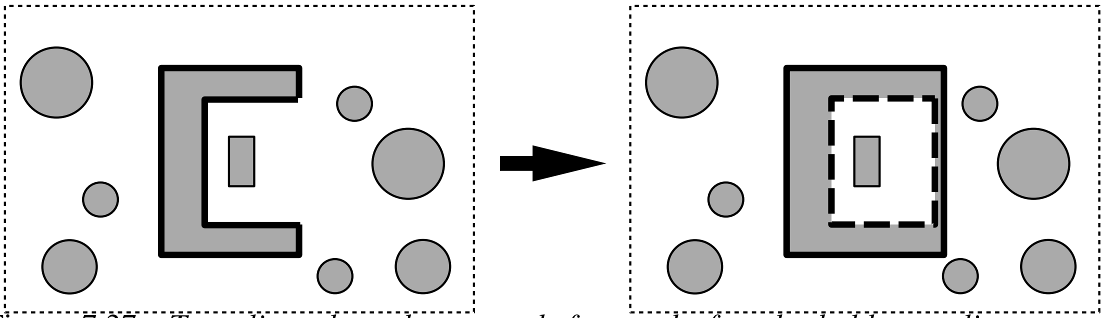


Figure 7.37: *Two slices through a part, before and after the bold recta-linear outer contour in the left slice splits into an outer and inner rectangular contour in the right slice. Although only the small rectangle's nesting is affected, we must re-derive the nesting of all the circular outer contours as well to see if they are also islands in the new hole. (For the example pictured here, this rederivation can be accomplished using the simple bounding box test.)*

7.5 Geometric Slicing

When our sweep plane reaches a height at which we want a physical slice, we need to calculate intersection points for each of the edge-uses in the current slice contour(s). If this is the first intersection with the edge-use, we calculate the deltas in x, y, and z between the top and bottom vertices of the edge and store them with the edge. For subsequent slicing planes that intersect the same edge-use, we can re-use these values. In this way, we exploit the geometric as well as the topological coherence between adjacent slices.

One calculation that we do perform on a point by point basis on every slice is to eliminate consecutive matching points and hence zero-length edges in the geometric slice contours as we output them. These occur when a slice plane exactly intersects a vertex and we get coincident intersection vertices for each of the beginning edge-uses at this vertex. We also eliminate zero-area intersection contours from the output if there are only one or two distinct intersection vertices in the contour.

An additional output optimization that is particularly useful for triangulated input is to omit redundant output points that are colinear with their two neighboring points. For example, a slice through the middle of a triangulated cube will have eight vertices by default, but only four of these are needed to define the geometry. We do not want to output points for the intersections with the edges that are not at the cube corners. For this reason, as we iterate through our contour list of edges, finding the intersection for each, we do not output an intersection point until we have calculated the subsequent intersection point and found that the middle intersection point is not colinear with the ones before and after it.

We also exploit inter-slice coherence to speed the elimination of the colinear middle points. Such points are colinear because the two triangles that join along the edge that produced them are coplanar. Therefore, once we have determined that such a point does not need to be output for one slice, all subsequent slices that intersect the same edge can also omit calculating and outputting the intersection with that edge. In practice, since the first intersection with an edge between two triangles can be very near to the tips of those triangles, the intersection edges on those triangles may be extremely short and appear to be colinear due to resolution limitations when the triangles are close to being, but not exactly, coplanar. If either of the intersection edges is too short to meaningfully test for colinearity given precision limitations, we do not mark the edge to have its future intersections ignored just yet.

7.6 Floating Point Issues

We use floating point arithmetic for all of the calculations described, and hence all are subject to roundoff error. This affects some of the details of our algorithms.

We cannot eliminate expensive division operations from any of our intersection calculations during the geometric slicing the way a standard scan-line visible surface determination algorithm [21] is able to. Using the scan line algorithm approach, with the first intersection with an edge we would calculate the delta in x and y between intersections a slice height apart. For subsequent intersections with the same edge, we would just add these quantities to the coordinates of the intersection with the slice below. The problem with this approach is that the deltas are only represented to a certain precision, and with each addition we can drift further and further from the true intersection point. In scan conversion, with 32-bit floating point numbers, the drift will typically at worst result in a one pixel wide shading error, so it will not be noticeable. While the drift for slicing will likewise be small in absolute terms, it can grow relatively large for edges that span many slices, leading to self-intersections in the output, as illustrated in Figure 7.38. This type of slice self-intersection occurred fairly often with an earlier version of our slicer that optimized intersection calculations in this manner (by using consecutive additions). These topological changes are

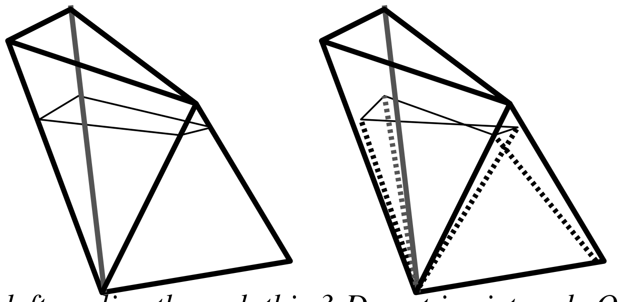


Figure 7.38: *On the left, a slice through this 3-D part is pictured. On the right, the drift for the active edges below the slice is represented by the dotted lines, and the resulting self-intersecting slice is pictured.*

unacceptable, since they can cause the tool path generation in QuickSlice, the fused deposition modeling (FDM) machine's software, to fail catastrophically (though software that cleaned up contours using the positive winding number rule, implemented in the readily available OpenGL tessellator [81], before performing offsets would not suffer from this limitation). Topological changes can also cause problems with our nesting calculations, which all rely on the assumption that separate contours do not intersect.

One approach to eliminating local self-intersections would be to merge successive vertices in a slice contour that were less than some epsilon distance apart (basing the epsilon on the worst possible drift). We would want to wait until after eliminating colinear vertices to do so, because we would not want the presence or absence of such a vertex to affect the shape of the output. However, this approach could only eliminate local intersections, but not "global" ones, which could occur far from any other vertices (see Figure 7.39). Another disadvantage is that ray shooting and area testing would be less accurate, since we would have much greater variability in the accuracy of the input to these operations. Therefore, we use division to calculate each new intersection with the same edge during geometric slicing for more accurate calculations than successive addition will produce.

The ray shooting nesting calculations themselves are also subject to roundoff errors. For most slice edges, we can determine if they intersect the horizontal ray without using floating point arithmetic; only for edges that have one end point above and to one side

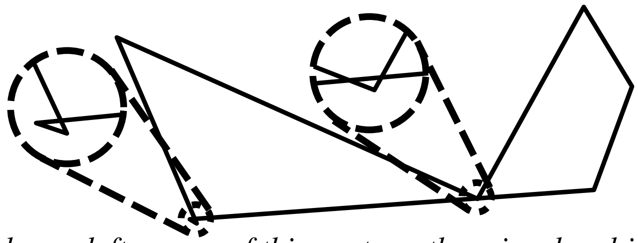


Figure 7.39: *In the lower left corner of this contour, there is a local intersection that can be eliminating by merging two adjacent vertices. The other "global" intersection cannot be fixed via vertex merging.*

of the ray base and the other end point below and to the other side of the ray base will we need to use floating point arithmetic to determine if they intersect the ray. If such a slice edge comes very close to or touches the ray base, then ideally we would repeat the intersection calculation using exact arithmetic to guarantee the correct answer. But that would only be as correct as the vertices in the slice, which were themselves calculated by floating point intersections with edges in the original part; therefore, these intersections would need to be recalculated using exact arithmetic first. To determine the correct epsilon to use to decide whether to redo the calculations with exact arithmetic, we would need to calculate the maximum possible error during each calculation and pass it on with the results to subsequent calculations.

In order to gauge whether this overhead is worthwhile, we implemented the test with the relatively large epsilon of .0001 and printed out a warning message when it was violated. Since we virtually never saw the warning during several months of testing, we did not feel it was worthwhile to implement the error calculation and exact arithmetic. (We were able to trigger the warning by taking the dragon part containing almost a million triangles, scaled to a height of .1", and making 1000 slices through it, giving us a slice thickness equal to our epsilon of .0001". This test was at a far finer scale than current SFF machines operate.)

Note that roundoff errors may also change whether a given edge of a contour intersects the ray for an edge not near the ray base. This would occur if the roundoff caused a vertex to move to the other side of the ray, but this will not affect whether the total number of intersections between the ray and the contour is odd or even, because the intersection with the other edge that shares the vertex (or another nearby edge in the contour in the case of multiple vertices switching sides) will compensate, as illustrated in Figure 7.40.

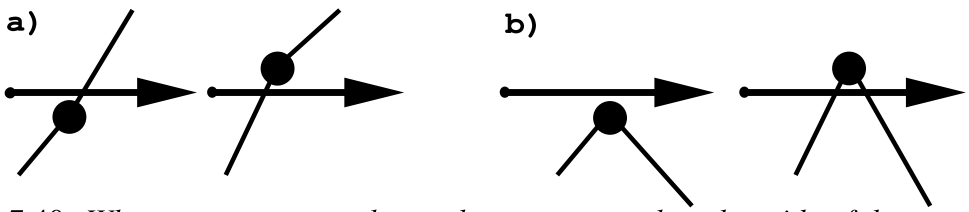


Figure 7.40: *When a vertex not at the ray base moves to the other side of the ray, it does not affect the intersection odd/even parity count for the contour. If only one of the vertex's edges crossed the ray before, generally the other will cross afterwards, as illustrated in a), leaving the intersection count unchanged. If neither edge crossed before, generally both will cross afterwards, and vice versa, as illustrated in b), changing the intersection count by two.*

In addition to affecting the test of whether a contour edge intersects the ray, roundoff errors will be present in the intersection point calculated. If we find multiple containers whose first intersection points are very close together, we use the area test as an additional check.

The standard formula for finding the area of a polygon does so by adding up the signed areas of the triangles formed between the origin and each directed edge of the polygon [20]. However, the floating point error in the triangle area calculations will be magnified proportional to the distance between the origin and the polygon edges. To minimize the effect that this will have on the area test, we find the average of all of the polygon's vertices and apply a translation that moves this average point to the origin before calculating the polygon's area. This reduces the error, though it does not eliminate it entirely. Ideally, we should still calculate the size of the possible error and take it into account when ranking the relative sizes of the areas. If there is too big a region of uncertainty to do a reliable total ordering of container areas during the area test, the original intersection and area calculations should be redone using exact arithmetic.

7.7 Complexity Analysis of the Slicing Algorithm

Call the number of faces in the triangulated, pseudo-manifold input n. Each face has three edges, and each edge is used twice, so the total number of edges is $3/2n$ (or $3n$

edge-uses). Each vertex is used by at least three edges and each edge is defined by two vertices, so the total number of vertices will be less than or equal to 2/3 of the number of edges, in other words $\leq n$ vertices. Therefore, the number of vertices will be $O(n)$.

If there are n faces in the input and the total number of vertices in the output is k, then the total time for calculating intersections and outputting the geometric slices from slice contours will be $O(k+n)$. We may remove up to $O(n)$ redundant vertices from the output, but in the worst case we will still need to calculate all these intersections to discover which ones to remove. In the worst case, we'll also need to recalculate orientation and nesting for each slice, again $O(k + n)$. (Of course, in the common case, we will be exploiting the coherence between slices to avoid many of these calculations. If we expected the worst case to be common, there would be no advantage to our entire approach of using a sweep-plane.)

The majority of the steps of the algorithm for building the slice contours are $O(n)$ or $O(nlogn)$. Building the topological data structure is dominated by the time to build hash tables for the edge-uses (to set sibling pointers) and their vertex-uses (to find if a vertex has been seen before and set the appropriate next vertex edge-use pointer); this takes time $O(nlogn)$. Dividing up pseudo-2-manifold edges is linear except for the sort; in the worst case, where virtually all the edge-uses are coincident, this will be $O(nlogn)$. Sorting the vertices to get the correct processing order is again $O(nlogn)$.

Building disk cycles takes time linear in the number of edge-uses to find their connectivity and to construct the linked lists, as well as time linear in the number of edge-uses to follow the next vertex edge-use pointers to find the starting edge-uses for additional disk cycles at non-manifold vertices. Identifying all runs of ending edge-uses just requires iterating through each disk cycle; therefore this step is also linear in the number of edge-uses. The total number of splices is limited to one pair of splices per ending edge-use; therefore, this step is also linear in the number of edge-uses. Checking for splitting or merging also occurs at most once per ending edge-use. All of these steps are $O(n)$.

The only part of the algorithm that has the potential to be quadratic in n is resetting the edge-uses' contour membership pointers after splitting or merging. An example worst case is a part of very high genus, with its through-holes lined up vertically over each other, with as many as possible of the other edge-uses that do not define the holes extending from below the lowest hole to above the highest hole, half on each side of the line of holes. With such a geometry, half of the long edge-uses will need to change their contour membership at the bottom of each hole as a single contour splits, and then again at the top of each hole when the two contours merge. Since such a part can be defined with n/c holes and n/c long edge-uses for a small constant c, updating the contour pointers could take total time $O(n^2)$. To accurately prototype such a part, however, we would want to output at least one slice through each hole and one slice between each pair of holes, in which case the update time for the contour pointers would be no more than the $O(k + n)$ time for outputting slices. Parts of such high genus relative to the number of triangles in their boundaries are vanishingly rare; for example, the genus 37 sculpture part pictured in Table 6.1 had 107,520 input triangles. Thus the total running time for reasonable input will be $O(k + nlogn)$, though the worst case running time will be $O(k^2 + n^2)$.

7.8 Slicing Booleans

If our input is a SIF file (see Chapter 3) that contains Booleans, we will slice without resolving the Booleans and output an LSIF file that contains the same Booleans, as pictured in Figures 7.41 and 7.42.

As with a non-hierarchical STL file, the first step in slicing a SIF file containing Booleans is to verify that the input is valid. The parser checks that the input is syntactically valid and builds a hierarchical representation of the Boolean tree with a separate LEDS for each shell; the shells do not share geometric information. Then we iterate through all of the shells, and for each, check that it is a valid closed solid, and separate any coincident 2-manifold edges, exactly as for the STL input.

When we slice, we will maintain a separate slice contour status structure for each LEDS shell, but share one event queue containing all of the vertices from all of the LEDS shells. Thus, we start by sorting an array of all of the vertices in all of the LEDS geometries by z-coordinate. Each vertex will have a disk cycle composed only of edge-uses connected to it in its own LEDS; therefore, processing a vertex will only affect the status structure for that LEDS. We can process the vertices exactly as described above, accessing the status

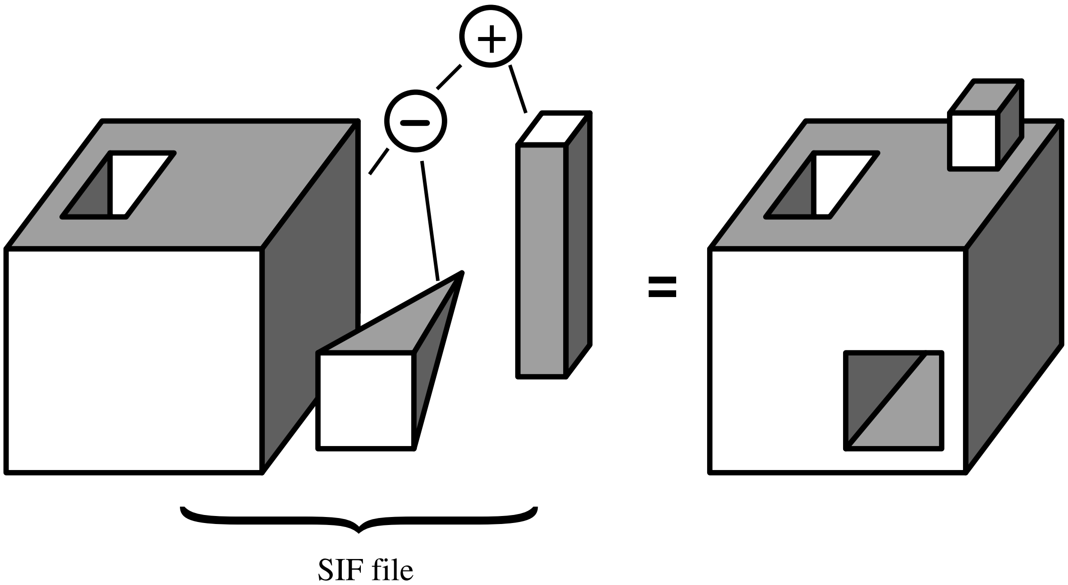


Figure 7.41: *The object pictured on the right could be represented in SIF as a Boolean tree with three leaf nodes, as pictured on the left. Each leaf node is a solid with a single shell.*

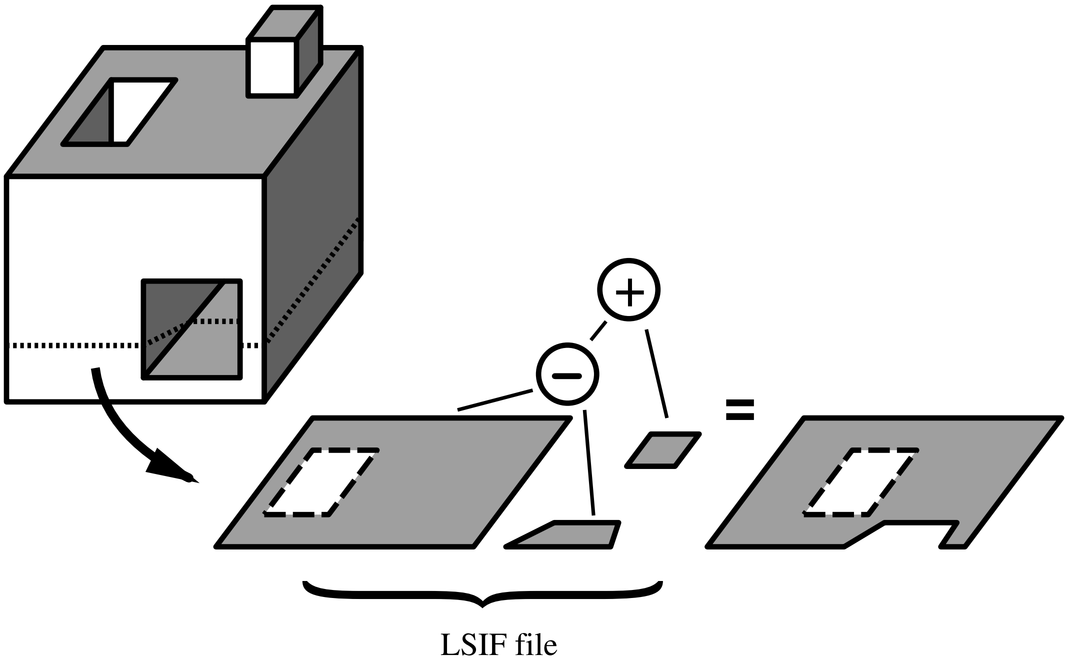


Figure 7.42: *For the slice indicated by the dotted line on the object on the left, our LSIF file will contain a Boolean tree identical in structure to the Boolean tree in the SIF input file. Each leaf node contains a contour or contours nested relative to other contours in that node.*

structure (the slice contour list) through a pointer stored with the LEDS.

Contour classification and nesting is done on a per-LEDS basis. (Since we will not be resolving the Booleans during slicing, we only nest contours at the same leaf of the Boolean tree). Although different LEDS shells in the Boolean tree will often intersect, within a single LEDS shell the slice contours will not intersect for valid input. Therefore, we can use the same algorithm described above in Section 7.4 to calculate nesting. Invalidating and rederiving nesting also takes into account only contours within the local LEDS.

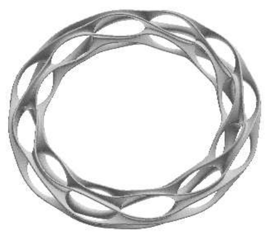
When we perform geometric slicing, we proceed one LEDS at a time. For each of the contours, we calculate the edge intersections as before. Unlike non-hierarchical input, with Booleans a typical leaf node will not be as tall as the full z-extent of all the leaf nodes; thus for some slices there may be no contours for a given LEDS leaf. In this case, we do not want an empty leaf in the LSIF output for this slice. Similarly, we do not want the LSIF output to include a union or intersection operator with only empty operands, or a difference operator whose first operand is empty, since all these operations produce the empty set. (While empty leaves or empty operands can be legal LSIF, we prefer to omit them from the output in order to make it more compact and simpler to process.) For a union or intersection operator with only one non-empty operand, or a difference operator whose only non-empty operand is its first operand, the Boolean does not modify the operand, so we just want to output the operand.

Therefore, when we do the intersection calculations that determine the slice geometry, we keep track of whether each LEDS leaf has any non-zero area contours to be output for the current level. After performing the intersection calculations and any contour classification and nesting updates, we are ready to output the LSIF Boolean tree (or trees) for the slice. Starting from the top of each Boolean tree, we query its operators to see which ones are empty, and only output the Boolean operator if it is meaningful, as described in the previous paragraph. Then we sequentially output each of its non-empty operands, invoking the same output function recursively on Boolean operands.

7.9 Results

We have implemented our algorithm in C++ and tested it on a variety of topologies and geometries. The running times reported are on an SGI Onyx Reality Engine with two 150 MHz MIPS R4400 processors and 128 MB of RAM.

The first series of tests we ran was for slicing the 107,520 triangle ring sculpture STL file (see Figure 7.43). We sliced it using several different slice thicknesses, both with our slicer and the commercial QuickSlice 6.2 software that ships with the FDM machine [70]. The preprocessing time (for reading the STL file and building the internal data structure) is



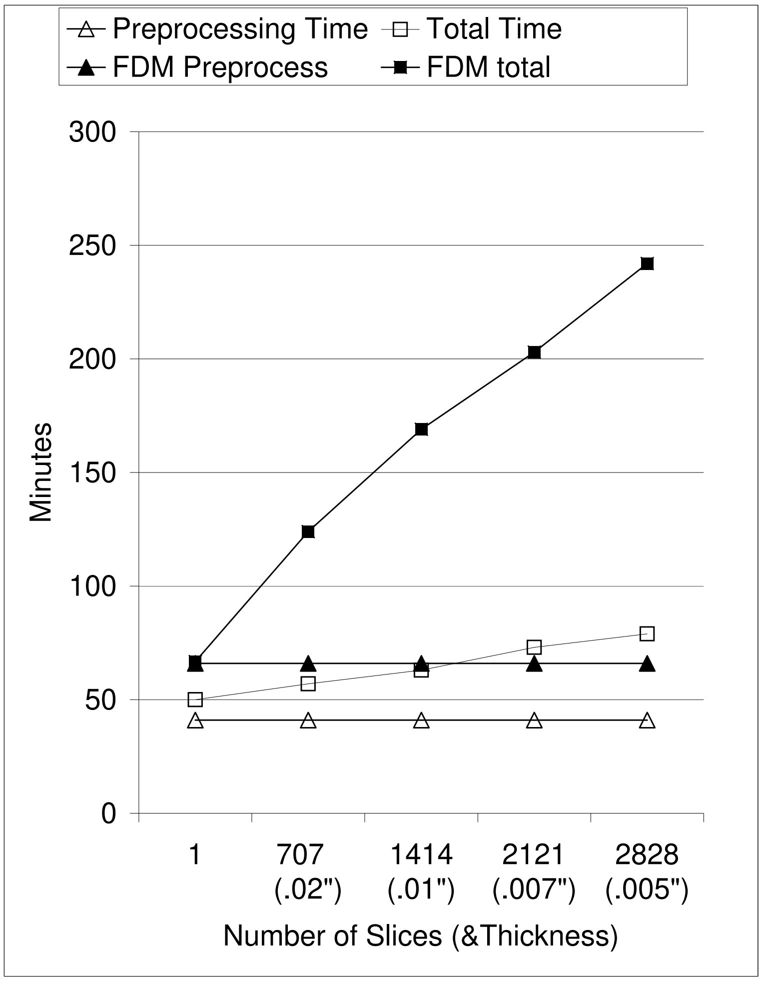


Figure 7.43: *Comparison of slicing performance for the ring sculpture pictured at left (107,520 input triangles) using our algorithm and using the commercial FDM QuickSlice 6.2 software.*

broken out for both slicers. With our slicer, the preprocessing takes longer than the actual slicing for typical slice thicknesses for this part. Of this preprocessing time, 75% is spent just parsing the STL file; only 25% of the preprocessing time is actually spent building the topological data structure. The preprocessing time for QuickSlice is longer than our preprocessing time, and also longer than our total slicing time for slices .01” thick and larger. (For FDM, .01” is the default slice thickness.) Furthermore, the slicing portion of the QuickSlice implementation is slower than ours and its time grows more rapidly than our slicing time as the number of slices increases. QuickSlice takes over twice as long for .01” thick slices through this part, and three times as long for .005” thick slices.

Our algorithm outperforms the commercial algorithm most dramatically when more slices are calculated. The time our algorithm spends determining connectivity is independent of the number of slices; if we can re-use the same connectivity information in multiple adjacent slices, our performance will be that much more efficient. We are also able to calculate more intersections incrementally with the more closely spaced slices, as shown in Figure 7.44.

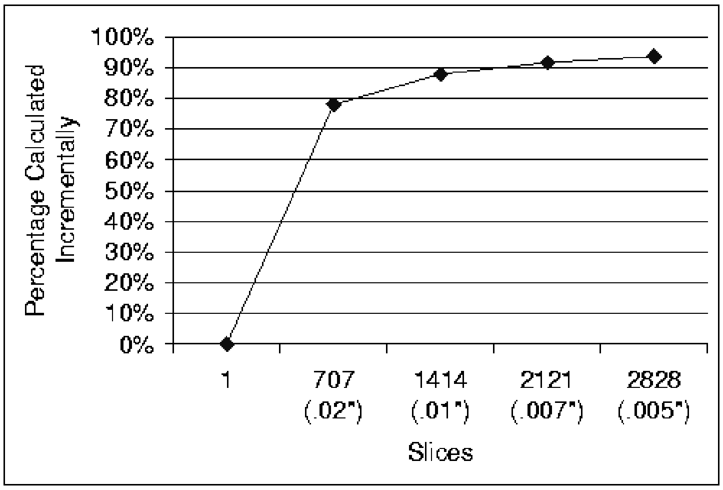
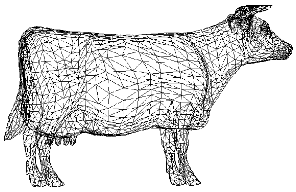


Figure 7.44: *The percentage of intersections calculated incrementally for the ring sculpture.*

We also compared the performance of our slicer and QuickSlice on a much less complicated file with large triangles, the cow (see Figures 7.45 and 7.48), and a much more complicated file with small triangles, the dragon (see Figures 7.46 and 7.47). With the 5,804 triangle cow file, there was much more coherence between slices; therefore, our performance was even better compared to QuickSlice (as much as six times faster). With the 869,898 triangle dragon file, on the other hand, we were not able re-use the connectivity information or calculate intersection incrementally nearly as often, and our preprocessing time grew because of the need to use an out-of-core algorithm to build our data structure; therefore, our performance was not much better than for QuickSlice (only 25% better). Currently, typical STL files are on the order of 100,000 triangles or less (the largest sample file that ships with QuickSlice contains 96,040 triangles), close to the size of the sculpture part; therefore, the results for the sculpture are probably the most representative.



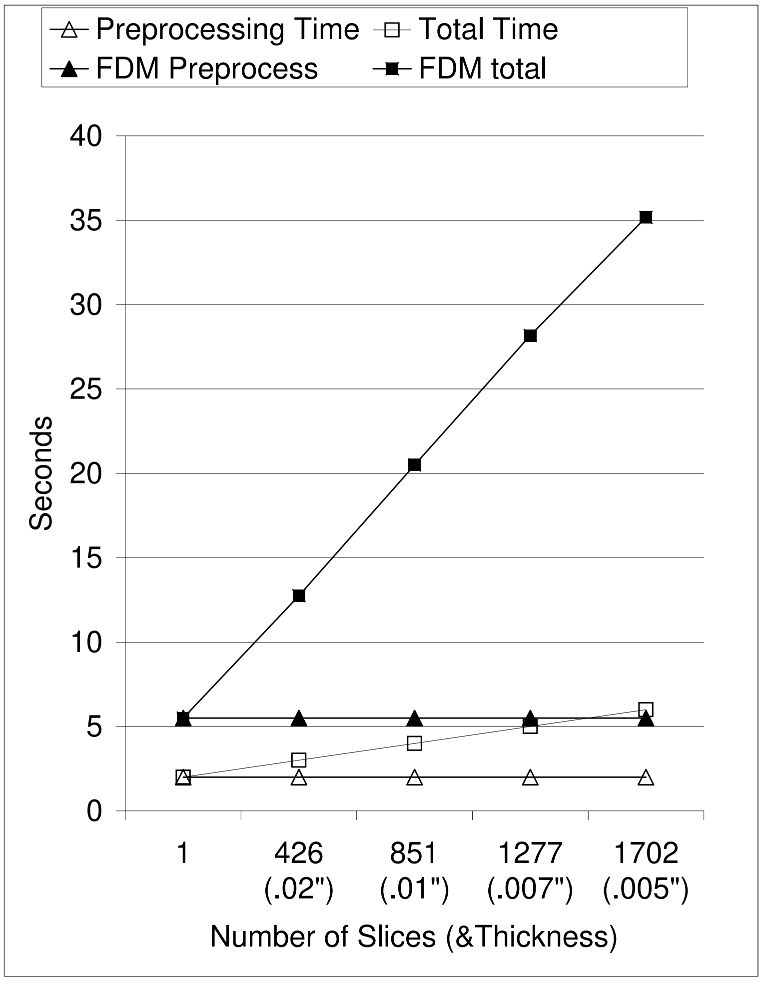


Figure 7.45: *Comparison of slicing performance for the cow file at left (5,804 input triangles) using our algorithm and using the commercial FDM QuickSlice 6.2 software.*

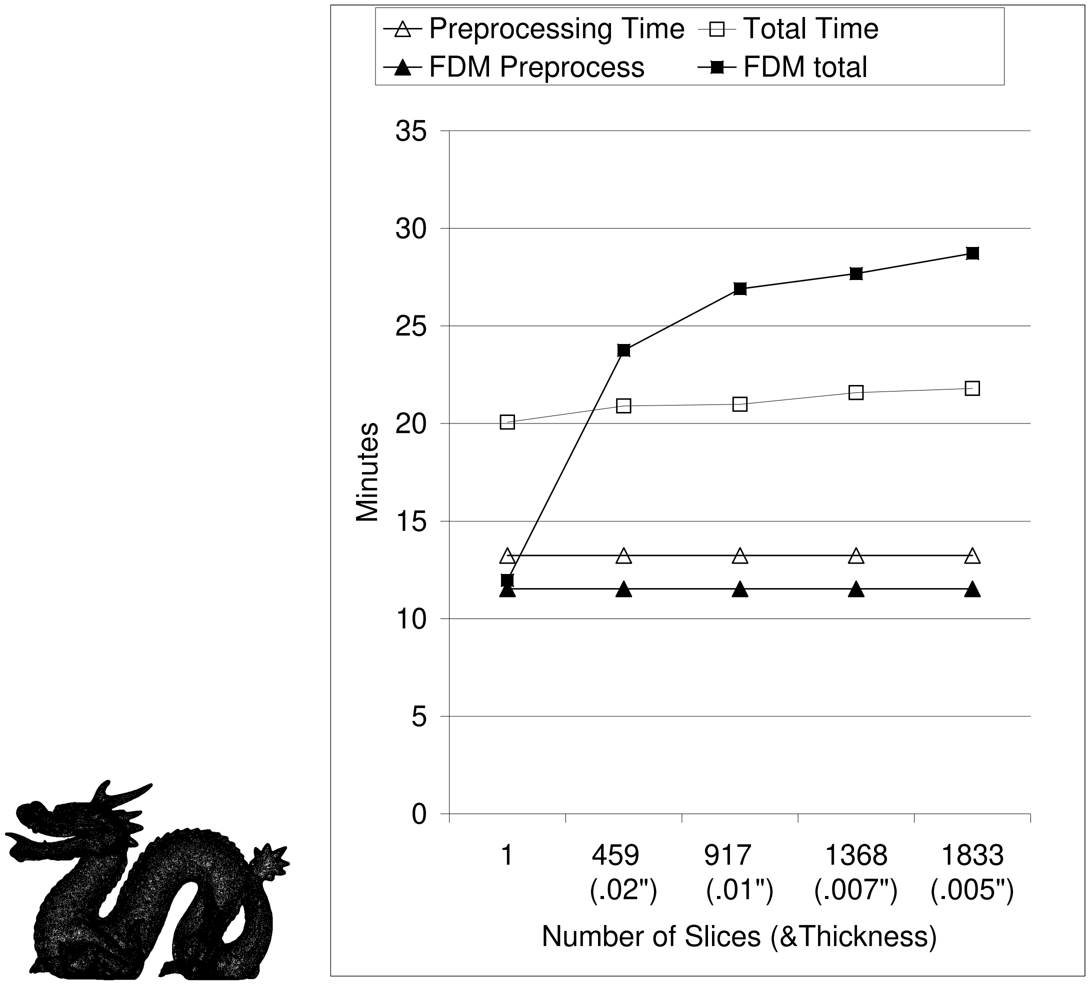


Figure 7.46: *Comparison of slicing performance for the dragon file pictured at left (869,898 input triangles) using our algorithm and using the commercial FDM QuickSlice 6.2 software.*

7.10 Discussion

Our algorithm is more efficient because it exploits coherence between consecutive slices. Such an approach only makes sense, of course, in the case where such coherence actually exists. With most SFF processes, slices must be quite thin, and the number of vertices to be processed between slices is only a small percentage of the total number of edges that intersect a slice. For an application where only a few slices need to be made relative to the number of edges, it would make more sense to use an algorithm that only considered those faces actually cut by the slicing plane, and to do the connectivity analysis

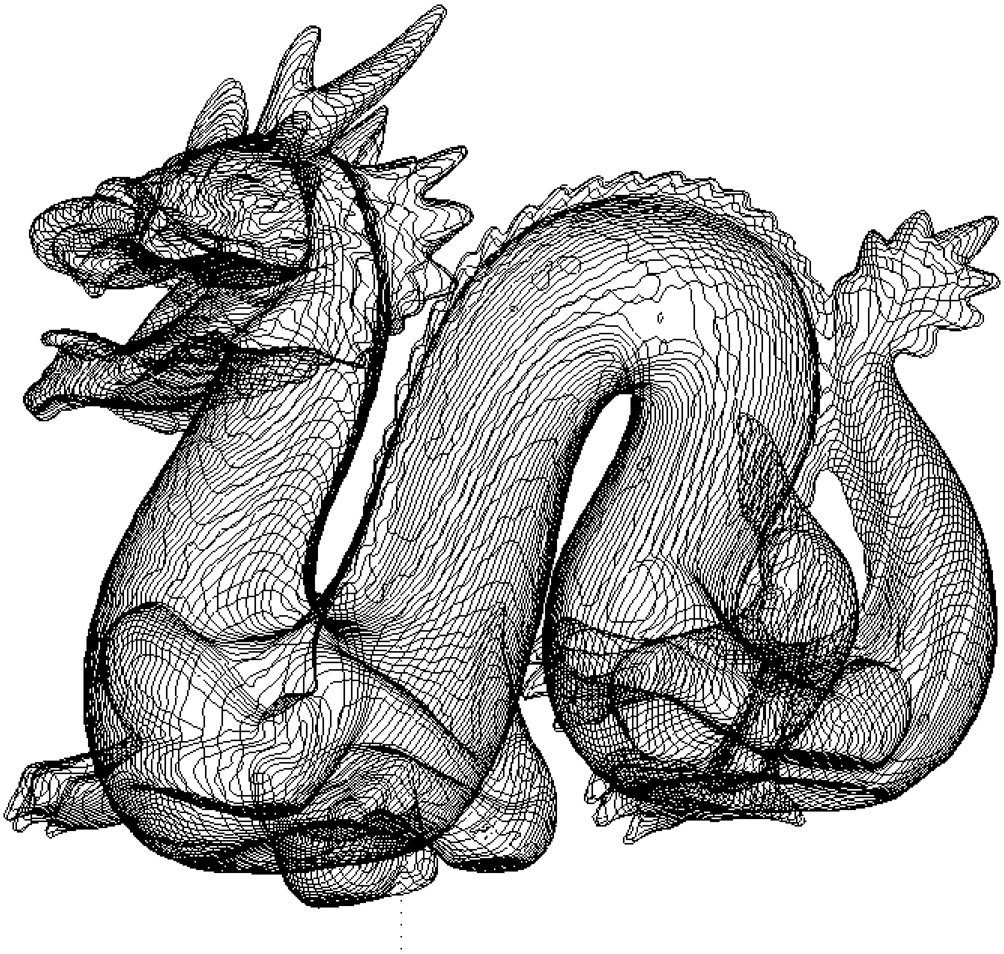


Figure 7.47: *Visualization of the slices through the dragon part. Only a subset of the slices are pictured so that the individual slice contours can be seen clearly.*

and matching of edges into contours in two dimensions. In general, our sweep plane approach will be most efficient when every polyhedron edge is sliced multiple times.

The proper nesting of islands in holes is probably not important for most applications. While QuickSlice's SSL format for specifying 2-D slices includes this information, an incorrectly nested island does not seem to cause any problems. An incorrectly nested hole, on the other hand, can cause a fatal internal error that aborts the entire QuickSlice application. With our incremental nesting algorithm, we could further reduce nesting calculations by only finding containers for hole contours.

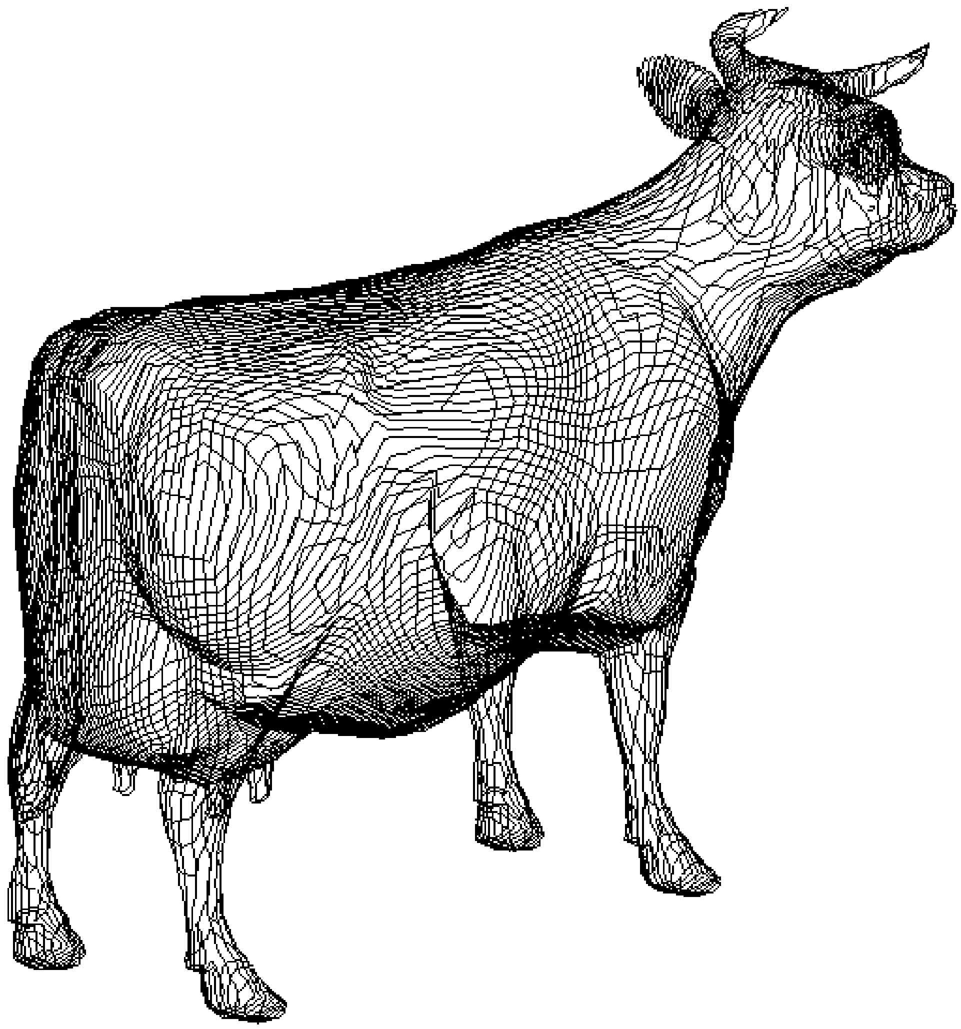


Figure 7.48: *Visualization of the slices through the cow part. Only a subset of the slices are pictured so that the individual slice contours can be seen clearly.*

The sweep plane topology updates in our slicing algorithm are not complicated to implement. There are no special cases for horizontal edges and we use identical splicing code for all vertex configurations, including saddle points. The incremental nesting calculations, on the other hand, can grow extremely complex for certain configurations. From an implementation point of view, it would be easier to use a sweep line algorithm to determine the mutual nesting of all the contours for each slice where nesting might have changed, since such an algorithm does not have nearly as many special cases to contend with. Our nesting algorithm is more efficient because it better exploits inter-slice coherence, but in practice nesting calculations take up so little of the slicing time that this increased efficiency is probably not worth the increased coding complexity.

Chapter 8

Summary

We have presented a new Solid Interchange Format, SIF, a neutral interchange format for exchanging SFF part geometries unambiguously. Unlike STL, SIF files share vertices between triangles, have no redundant surface normals, specify their units and version, and can include information about surface and volume properties. We have also introduced the concept of constellations. Despite these additional features, the specification of SIF is short and simple in order to limit the complexity of generating and interpreting SIF files. SIF puts the burden of determining the connectivity of shells on the design side, where it belongs, so that problems in the description of the geometry are caught and corrected before a design is transmitted to a manufacturer. While outputting SIF requires more work than STL for a part whose connectivity is not already known, outputting SIF for a part described with CSG operations requires less work, since the unevaluated CSG tree can be output directly. This puts the burden of resolving the Booleans on the manufacturing side, but it can then be postponed until after slicing, so that only 2-D Booleans need to be resolved, which can be implemented using off-the-shelf OpenGL software and hardware. SIF represents a significant improvement over STL in reducing ambiguity in the exchange of SFF parts, without favoring a particular CAD or SFF vendor, and without requiring a substantially increased investment in software infrastructure from either the designer or the manufacturer.

We have developed a new topological data structure, the loop edge data structure (LEDS), optimized for SFF applications. The LEDS can answer most adjacency queries in time linear in the number of responses, independent of the input size. The LEDS captures connectivity for all valid solids, including non-2-manifold solids. In addition, we have demonstrated how to capture mixed-dimension geometry in the unified framework of the same LEDS representation.

We have implemented a simple in-memory algorithm for efficiently building a LEDS from unorganized triangulated input for small and medium sized files. We have experimentally verified that we have chosen hash functions for our implementation that are suitably random and yet do not require lengthy computations. We have also developed the first out-of-core algorithm for building a LEDS or any similar topological data structure for large files that do not fit in memory. With this algorithm we achieved running times that were two orders of magnitude faster than the optimized in-memory algorithm on typical STL input. We achieved these speed-ups while actually using slightly *less* memory for the larger files.

Our analysis module identifies non-closed boundaries and cleans up round-off errors in the input using a straightforward epsilon vertex merging approach. We also describe how to transform non-2-manifold LEDS geometry into a pseudo-2-manifold representation in order to simplify downstream processing.

We have designed and implemented a new coherent sweep plane slicing algorithm for faster generation of parallel slices for SFF process planning. Our slicer accepts input in STL or SIF containing unevaluated Booleans, correctly processing arbitrary topology including non-2-manifold solids. Our slicer also uses a new algorithm for determining 2-D contour nesting which is designed for incremental updates and robust for non-manifold slices. With our slicer implementation, on typical parts we achieved speed-ups of two or three times compared to a comparison commercial slicer.

Bibliography

[1] 3D Systems, Inc. Stereolithography Interface Specification. Company literature, 1988.

[2] Steven Ashley. Rapid Prototyping Systems. *Mechanical Engineering*, pages 34–43, April 1991.

[3] Stacey Au and Paul K. Wright. A Comparative Study of Rapid Prototyping Technology. In *Intelligent Concurrent Design : Fundamentals, Methodology, Modeling, and Practice*, pages 73–82, New Orleans, Louisiana, November 28–December 3, 1993. ASME Winter Annual Meeting.

[4] C. L. Bajaj, V. Pascucci, D. Thompson, and X. Y. Zhang. Parallel accelerated isocontouring for out-of-core visualization. In *Proceedings 1999 IEEE Parallel Visualization and Graphics Symposium*, pages 97–104,122, San Francisco, CA, October 1999. IEEE.

[5] Gill Barequet and Subodh Kumar. Repairing CAD Models. In *Proceedings Visualization '97*, pages 363–370, Phoenix, AZ, October 1997. IEEE.

[6] Gill Barequet and Micha Sharir. Filling Gaps in the Boundary of a Polyhedron. *Computer-Aided Geometric Design*, 12(2):207–29, March 1995.

[7] Richard H. Bartels, John C. Beatty, and Brian Barsky. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. M. Kaufmann Publishers, 1987.

[8] B. G. Baumgart. A Polyhedron Representation for Computer Vision. In *Proceedings of the National Computer Conference*, pages 589–596, 1975.

[9] Joseph J. Beaman et al. *Solid Freeform Fabrication : A New Direction in Manufacturing*. Kluwer Academic Publishers, Dordrecht, 1997.

[10] Jan Helge Bohn and Michael J. Wozny. A Topology-Based Approach for Shell-Closure. In *Geometric Modeling for Product Realization*, pages 297–319. North-Holland, Amsterdam, 1992.

[11] Jan Helge Bohn and Michael J. Wozny. Automatic CAD-model Repair: Shell-Closure. In *Proceedings Solid Freeform Fabrication Symposium*, pages 86–93. University of Texas at Austin, August 1992.

[12] Y.-J. Chiang, C. T. Silva, and W. J. Schroeder. Interactive out-of-core isosurface extraction. In *Proceedings Visualization '98*, pages 167–74, Research Triangle Park, NC, October 1998. IEEE.

[13] Yi-Jen Chiang and Claudio Silva. I/O optimal isosurface extraction. In *Proceedings Visualization '97*, pages 293–300, Phoenix, AZ, October 1997. IEEE.

[14] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[15] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Proceedings Visualization '97*, pages 235–44,547, Phoenix, AZ, October 1997. IEEE.

[16] D. Davis, T. Y. Jiang, W. Ribarsky, and N. Faust. Intent, perception, and out-of-core visualization applied to terrain. In *Proceedings Visualization '98*, pages 455–458, Research Triangle Park, NC, October 1998. IEEE.

[17] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, Berlin, 1997.

[18] A. Dolenc. Software Tools for Rapid Prototyping Technologies in Manufacturing. *Acta Polytechnica Scandinavica, Mathematics and Computer Science Series*, (no.Ma62):1–111, 1993.

[19] A. Dolenc and I. Mäkelä. Slicing Procedures for Layered Manufacturing Techniques. *Computer Aided Design*, 26(2):119–26, February 1994.

[20] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

[21] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*, pages 680–685. Addison-Wesley, Reading, MA, second edition, 1990.

[22] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Computer Graphics Proceedings, Annual Conference Series*, pages 247–54, Anaheim, CA, August 1993. ACM SIGGRAPH.

[23] Thomas Allen Funkhouser. *Database and Display Algorithms for Interactive Display of Architechtural Models*. PhD thesis, University of California, Berkeley, Computer Science Division, 1993. Also available as UC Berkeley Technical Report UCB/CSD-93-771.

[24] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[25] Sashidhar Guduri, Richard H. Crawford, and Joseph J. Beaman. A Method to Generate Exact Contour Files for Solid Freeform Fabrication. In *Proceedings Solid Freeform Fabrication Symposium*, pages 95–101. University of Texas at Austin, August 1992.

[26] Sashidhar Guduri, Richard H. Crawford, and Joseph J. Beaman. Boundary Evaluation for Solid Freeform Fabrication. In *Conference on Towards World Class Manufacturing 1993*, pages 301–12, Litchfield Park, AZ, September 1993. IFIP TC5/WG5.3.

[27] E. Gursoz, Y. Choi, and F. Prinz. Vertex-based Representation of Non-Manifold Boundaries. In *Geometric Modeling for Product Engineering*, pages 107–130. North-Holland, Amsterdam, 1990.

[28] E. Gursoz, Y. Choi, and F. Prinz. Boolean set operations on non-manifold boundary representation objects. *Computer Aided Design*, 23(1):33–39, January/February 1991.

[29] Christoph M. Hoffmann. *Geometric and Solid Modeling : An Introduction*. Morgan Kaufmann, San Mateo, CA, 1989.

[30] Hughes Hoppe et al. Piecewise smooth surface reconstruction. In *Computer Graphics Proceedings, Annual Conference Series*, pages 295–302, Orlando, FL, July 1994. ACM SIGGRAPH.

[31] Josef Hoschek and Kieter Lasser. *Fundamentals of Computer Aided Geometric Design*. A K Peters, Wellesley, MA, 1993.

[32] Gan G. K. Jacob, Chee Kai, and Tong Mei. Development of a new rapid prototyping interface. *Computers in Industry*, 39(1):61–70, June 1999.

[33] Paul F. Jacobs. *Rapid Prototyping and Manufacturing : Fundamentals of Stereolithography*. Society of Manufacturing Engineers, Dearborn, MI, 1992.

[34] H. V. Jagadish. Linear Clustering of Objects with Multiple Attributes. In *1990 ACM SIGMOD International Conference on Management of Data*, pages 332–42, Atlantic City, NJ, May 1990. SIGMOD.

[35] H. J. Jee and E. Sachs. A visual simulation technique for 3D printing. *Advances in Engineering Software*, 31(2):97–106, February 2000.

[36] Y. E. Kalay. The Hybrid Edge: A Topological Data Structure for Vertically Integrated Geometric Modelling. *Computer Aided Design*, pages 130–40, April 1989.

[37] Y. E. Kalay and C. Séquin. Designer-client relationships in architectural and software design. *ACADIA'95: Conference of the Association for Computer Aided Design in Architecture*, pages 383–403, October 1995.

[38] Ibrahim Kamel and Christos Faloutsos. On Packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Managementn*, pages 490–9, Washington, DC, November 1993. ACM.

[39] C. F. Kirschman and C. C. Jara-Almonte. A Parallel Slicing Algorithm for Solid Freeform Fabrication Processes. In *Proceedings Solid Freeform Fabrication Symposium*. University of Texas at Austin, August 1992.

[40] Masaru Kitsuregawa, Masaya Nakayama, and Mikio Takagi. The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. In *Proceedings of the Fifteenth International Conference of Very Large Databases*, pages 257–266, Amsterdam, August 1989. Morgan Kaufmann.

[41] Donald E. Knuth. *The Art of Computer Programming*, volume III. Addison-Wesley, Reading, MA, 1973.

[42] Detlef Kochan. *Solid Freeform Manufacturing : Advanced Rapid Prototyping*. Elsevier, Amsterdam and New York, 1993.

[43] Prashant Kulkarni and Debasish Dutta. An Accurate Slicing Procedure for Layered Manufacturing. *Computer Aided Design*, 28(9):683–97, September 1996.

[44] Ren C. Luo and Yawei Ma. A Slicing Algorithm for Rapid Prototyping and Manufacturing. In *Proceedings of 1995 IEEE International Conference on Robotics and Automation*, pages 2841–6 vol. 3, Nagoya, Japan, May 1995. IEEE.

[45] K. Mani, P. Kulkarni, and D. Dutta. Region-based adaptive slicing. *Computer-Aided Design*, 31(5):317–33, April 1999.

[46] Martti Mäntylä. Boolean Operations of 2-Manifolds through Vertex Neighborhood Classification. *ACM Transactions on Graphics*, 5(1):1–29, 1986.

[47] Martti Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, MD, 1988.

[48] Sara Anne McMains. Rapid prototyping of solid three-dimensional parts. Master's thesis, University of California, Berkeley, 1995. Also available as UC Berkeley Technical Report UCB/CSD-96-892.

[49] C. Mead and L. Conway. The Caltech Intermediate Form for LSI Layout Description. In *Introduction to VLSI Systems*, pages 115–127. Addison-Wesley, 1980.

[50] Michael E. Mortenson. *Geometric Modeling*. Wiley, New York, 1985.

[51] T. M. Murali and Thomas A. Funkhouser. Consistent Solid and Boundary Representations from Arbitrary Polygonal Data. In *1997 Symposium on Interactive 3D Graphics*, pages 155–162, Providence, R.I., April 1997. ACM SIGGRAPH.

[52] Society of Manufacturing Engineers. Manufacturing Insights: Rapid Tooling, Rapid Parts. Videocassette, 1994.

[53] M. Pellegrini. Repetitive hidden surface removal for polyhedra. *Journal of Algorithms*, 21(1):80–101, July 1996.

[54] Matt Pharr, Craig Kolb, Reid Geshbein, and Pat Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *Computer Graphics Proceedings, Annual Conference Series*, pages 101–108, Los Angeles, CA, August 1997. ACM SIGGRAPH.

[55] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry : An Introduction*. Springer-Verlag, New York, 1985.

[56] Ari Rappoport and Steven Spitz. Interactive Boolean Operations for Conceptual Design of 3-D Solids. In *Computer Graphics Proceedings*, pages 269–78, Los Angeles, CA, August 1997. ACM SIGGRAPH.

[57] A. A. G. Requicha. Representations for Rigid Solids: Theory, Methods, and Systems. *ACM Computing Surveys*, pages 437–464, December 1980.

[58] Tony ReRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *Computer Graphics Proceedings, Annual Conference Series*, pages 85–94, Orlando, FL, July 1998. ACM SIGGRAPH.

[59] Stephen J. Rock and Michael J. Wozny. A Flexible File Format for Solid Freeform Fabrication. *Proceedings Solid Freeform Fabrication Symposium*, pages 1–12, 1991.

[60] Stephen J. Rock and Michael J. Wozny. Utilizing Topological Information to Increase Scan Vector Generation Efficiency. *Proceedings Solid Freeform Fabrication Symposium*, pages 28–36, 1991.

[61] Stephen J. Rock and Michael J. Wozny. Generating Topological Information from a "Bucket of Facets". *Proceedings Solid Freeform Fabrication Symposium*, pages 251–259, 1992.

[62] Jarek Rossignac and David Cardoze. Matchmaker: Manifold BReps for non-manifold r-sets. In *Fifth Symposium on Solid Modeling and Applications*, pages 31–41, Ann Arbor, MI, June 1999. ACM.

[63] E. Sachs, P. Williams, D. Brancazio, M. Cima, and K. Kremmin. Three Dimensional Printing: Rapid Tooling and Prototypes Directly from a CAD Model. In *Proceedings of Manufacturing International '90*, pages 131–136, Atlanta, GA, March 25–28, 1990.

[64] S. Sarma, S. Schofield, J. A. Stori, J. MacFarlane, and P. K. Wright. Rapid Product Realization from Detail Design. *Computer Aided Design*, 28(5):383–92, 1996.

[65] Thomas W. Sederberg, Jianmin Zheng, David Sewell, and Malcolm Sabin. Non-uniform recursive subdivision surfaces. In *Computer Graphics Proceedings, Annual Conference Series*, pages 387–94, Orlando, FL, July 1998. ACM SIGGRAPH.

[66] C. H. Séquin and J. Smith. Parameterized procedural synthesis of artistic geometry. *International Journal of Shape Modeling*, 5(1):81–99, June 1999.

[67] Xuejun Sheng and Ingo R. Meier. Generating Topological Structures for Surface Models. *IEEE Computer Graphics and Applications*, 15(6):35–41, November 1995.

[68] Charles S. Smith. *The Manufacturing Advisory Service: Web Based Process and Material Selection*. PhD thesis, 1999.

[69] Spatial Technology, Inc, Boulder, CO. *ACIS Save File Format Manual*, 1996.

[70] Stratasys, Inc., Eden Prairie, MN. *QuickSlice 6.2*, 1999.

[71] I. Stroud and P. C. Xirouchakis. STL and extensions. *Advances in Engineering Software*, 31(2):83–95, February 2000.

[72] P.D. Sulatycke and K. Ghose. A fast multithreaded out-of-core visualization technique. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 569–75, San Juan, Puerto Rico, April 1999. IEEE.

[73] Seth Teller, Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan. Partitioning and ordering large radiosity computations. In *Computer Graphics Proceedings, Annual Conference Series*, pages 443–50, Orlando, FL, July 1994. ACM SIGGRAPH.

[74] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walk-throughs. In *Computer Graphics Proceedings, Annual Conference Series*, pages 61–9, Las Vegas, NV, July 1991. ACM SIGGRAPH.

[75] Seth Jared Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, University of California, Berkeley, Computer Science Division, 1992. Also available as UC Berkeley Technical Report UCB/CSD-92-708.

[76] Shyh-Kuang Ueng, C. C. Sikorski, and Kwan-Liu Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):370–380, Oct.-Dec. 1997.

[77] Herbert B. Voelcker and A. G. Requicha. Research in Solid Modeling at the University of Rochester: 1972-87. In *Fundamental Developments of Computer-Aided Geometric Modeling*, pages 203–254. Academic Press, San Diego, CA, 1993.

[78] Kevin Weiler. The Radial Edge Structure: A Topological Representation for Non-Manifold Geometric Boundary Modeling. In *Geometric Modeling for CAD Applications*, pages 3–36. North-Holland, Amsterdam, 1988.

[79] Kevin Weiler. Vertex Neighborhood Topological Information and Data Structures in a Non-Manifold Environment. Technical report, Autodesk, April 5, 1996.

[80] Janet L. Wiener and Jeffrey F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 30–41, Zurich, Switzerland, September 1995. Morgan Kaufmann.

[81] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison-Wesley, Reading, Massachusetts, third edition, 1999.

[82] Lamont Wood. *Rapid Automated Prototyping : An Introduction*. Industrial Press, 1993.

[83] J. R. Woodwark. Splitting set-theoretic solid models into connected compounds. Technical report, IBM United Kingdom Ltd., Winchester, UK., 1989.

[84] Paul Wright and Carlo Séquin. CyberCut: A Networked Manufacturing Service. In *Managing Enterprises–Stakeholders, Engineering, Logistics and Achievement*, UK, July 22-24 1997. Loughborough University.

Appendix A

Appendix: User Guides

NAME

sif2ssl - slices a SIF file into .01 thick slices, output in SSL

SYNOPSIS

sif2ssl *file.sif file.ssl*

DESCRIPTION

Script to read in a SIF file, slice it into .01 thick slices, and then resolve the booleans, translate the geometry to the positive quadrant, and convert it into an SSL file compatible with QuickSlice 6.2.

EXAMPLES

sif2ssl cheese.sif cheese.ssl

FILES

/project/cs/sequin/caffe/sif/bin/sif2ssl
/project/cs/sequin/caffe/sif/bin/SYSTEM/IRIX/slicer
/project/cs/sequin/caffe/sif/bin/SYSTEM/IRIX/glsif

AUTHOR

Sara McMains

NAME

slicer - analysis, cleanup, and slicing for Solid Freeform Fabrication (SFF)

SYNOPSIS

slicer [-a [-e *n*][-stl *file2*]] *file*
slicer [-m [*n*][-b *n*]][-a [-e *n*][-stl *file2*]] *file*
slicer [-t *n* [-u *n*] [-v *x y z*] [-of *file2*] *file*
slicer [options] *file*

DESCRIPTION

In the first two forms, the **slicer** program analyzes and cleans up the input STL or SIF file, optionally writing out the cleaned up input file, but does not actually slice it. With the first form, an in-core algorithm is used to build the internal Loop Edge Data Structure (LEDS). In the second form, an out-of-core algorithm is used to build the LEDS. In the third form, the **slicer** builds the LEDS from the input file, confirms that it describes a valid part, and slices it with the specified thickness. An LSIF file describing the slices is output. The analysis and cleanup and out-of-core algorithm options in the first and second form can also be combined with the slicing options in the third form.

The **slicer** recognizes the following options and command-line arguments:

-a Run the analyzer/clean up portion of the **slicer** program. Checks whether the input part is 2-manifold or potentially pseudo-2-manifold, calculates its bounding box, and prints number of faces, edges and their uses, vertices and their order. If the part is not 2-manifold or potentially pseudo-2-manifold, it tries to make it so by merging non-manifold vertices within epsilon and then re-analyzes the part to determine if this fixed the problem. The default epsilon is 10% of the shortest edge length.

-b *n* Invoke the vertex bin sorting option in the out-of-core algorithm for building the internal data structure, using *n* bins.

-e *n* Use *n* as epsilon for vertex merging in the analyzer. We only try to merge vertices that are endpoints of unmatched edges. If epsilon is not specified with this option, 10% of the shortest edge length is used as epsilon.

-m [*n*] Use the multi-stage, out-of-core algorithm for building the internal topological data structure. If *n* is specified, it controls the number of partitions allocated: *n* vertex hash table partitions, and 2 * *n* edge hash table partitions. This option is only implemented for STL input.

-of *file* Specifies the name for the output LSIF file generated by the slicer.

-u *n* Scale the entire part before slicing by a uniform scale factor n.

-t *n* If the input is valid, make consecutive horizontal slices through the part, starting at the lowest point and using a slice thickness of *n*. Outputs an LSIF file (to stdout, unless **-of** option is used).

-v *x y z* Changes the up vector for slicing to be (*x, y, z*) from a default up vector of the positive z-axis, (0, 0, 1).

EXAMPLES

Analyze and clean up the STL file complex.stl, using the out-of-core algorithm with 10 partitions for the vertex hash table. Use an epsilon of .001 for the vertex merging, and write the cleaned up version of the stl file to clean.stl:

slicer -m 10 -a -e .001 -stl clean.stl complex.stl

Slice the SIF file cube.sif, scaled down 50%, with slice axis perpendicular to the y-axis and slice thickness .01, and write the output file to cube.lsif:

slicer -t .01 -u .5 -v 0 1 0 -of cube.lsif cube.sif

FILES

/project/cs/sequin/caffe/sif/bin/SYSTEM/IRIX/slicer

AUTHOR

Sara McMains