An Interpreted Language and System for the Visualization of Unstructured Meshes

Patrick J. Moran, MRJ Technology Solutions Inc.*
NAS Technical Report NAS-98-004

Abstract

We present an interpreted language and system supporting the visualization of unstructured meshes and the manipulation of shapes defined in terms of mesh subsets. The language features primitives inspired by geometric modeling, mathematical morphology and algebraic topology. The adaptation of the topology ideas to an interpreted environment, along with support for programming constructs such as user function definition, provide a flexible system for analyzing a mesh and for calculating with shapes defined in terms of the mesh.

We present results demonstrating some of the capabilities of the language, based on an implementation called the *Shape Calculator*, for tetrahedral meshes in $\mathbb{R}^3$.

*NASA Ames Research Center, M/S T27A-2, Moffett Field, CA, 94035, USA. ⟨pmoran@nas.nasa.gov⟩.

1 Introduction

Computing a high quality numerical solution within a domain is intimately tied to constructing a high quality meshing of the domain. A mesh in $\mathbb{R}^3$ decomposes the domain into polyhedral cells. Most meshes are classified as either structured, unstructured, or a hybrid combination of the two. In a structured mesh, the vertices can be indexed in a regular manner, and the cells are a uniform type. For example, in $\mathbb{R}^3$ the polyhedral cells may be all hexahedra, and the vertices of the mesh may be indexed as if they were in a 3-dimensional array. In an unstructured mesh, there is usually no implicit neighbor information in the vertex indices, and there may be a mix of cell types in the mesh. In one of the most common types of unstructured meshes in $\mathbb{R}^3$, all the polyhedral cells are tetrahedra.

In order to verify the quality of a mesh, scientists typically employ both quantitative and visual techniques. In some visualization and analysis techniques, one can take advantage of the regularity present in a structured mesh. For example, given a structured mesh wrapped around an aircraft wing, it is usually easy to identify the mesh vertices on the wing surface, or to obtain vertices close to but not exactly on the surface. One can also obtain an overall visualization of a structured mesh in $\mathbb{R}^3$ by displaying surfaces in the mesh defined by holding one of the three mesh vertex indices constant. In the case of unstructured meshes, the more general organization of the mesh leaves fewer options for the scientist attempting to grasp the overall mesh structure. Typically the application that generates the unstructured mesh will identify some subset of polygonal faces of the mesh as special, such as the triangles on an aircraft body, and a visualization application can easily display those faces. There are a few other standard visualization techniques, such as constructing cutting planes through the mesh [SK90], but in general the choices are very limited.

In this article we introduce a language and visualization system intended to provide flexible support for unstructured mesh analysis and visualization. A key concept in the system is that of shapes. Shapes are defined by subsets of the mesh cells, including lower-dimensional cells such as triangles, edges and vertices. We have implemented an interpreter and a visualization application based on the interpreter, which we call the *Shape Calculator*. Our current implementation supports working with shapes in 3-dimensional simplicial meshes. Using the calculator, one can interactively specify and manipulate shapes, display the shapes, and get quantitative information about the shapes, including plots. The calculator language includes programming constructs such as function definition and conditional statements, so one can also develop algorithms for visualization and analysis.

The following section contains a short overview of some previous work related to the *Shape Calculator*. In Section 3, we review definitions that will be needed in later sections. In Section 4 we describe the shape calculator language, followed in Section 5 by a description of the programming constructs in the language used in the later examples. Section 6 presents a short overview of the calculator implementation. In Section 7 we present several examples illustrating how the system can be used, and in Section 8 we conclude with some thoughts on how the language would generalize to other types of meshes, such as those with non-simplicial cells.

2 Previous Work

The *Shape Calculator* has properties that qualify it as both a modeling system and as a mesh visualization system. As a modeling system, the *Shape Calculator* has many predecessors, including some which include a modeling language. Paoluzzi *et al.* [PBCF93] describe a dimension-independent system for modeling with simplicial complexes, including a manipulation language. Their system is very general, and it includes numerous modeling capabilities. Pascucci *et al.* [PFP95] also describe a dimension-independent modeling system where their primitives are convex cells. In contrast to the systems above, the *Shape Calculator* requires that all shapes be defined in terms of simplices from a single decomposition of the whole space, the mesh $\mathcal{M}$. This requirement makes the *Shape Calculator* less applicable as a general purpose modeling tool. On the other hand, for mesh analysis, where we are given the mesh initially, the requirement that all simplices be from a common simplicial complex $\mathcal{M}$ is not a significant drawback. Furthermore, restricting the "building blocks" that the user has available for defining shapes allows many optimizations, improving the interactive performance of the system.

As an unstructured mesh visualization tool, the *Shape Calculator* has fewer predecessors. FAST [B+90] is a large, modular system for the visualization of computational fluid dynamics data. The module `surferu` in FAST supports the display of a whole unstructured mesh, or surfaces that have been predefined by the application generating the mesh. The module `shotet` in FAST allows the user to select various subsets of the tetrahedra in a mesh, based on geometric criteria such as volume or circumscribing sphere radius. The *Shape Calculator* is much more general in terms of the shapes that it can represent, compared to `surferu`. The language also contains primitives for evaluating tetrahedra as in `shotet`. Currently the calculator has fewer built-in geometric primitives than `shotet`, though we plan on adding more to the calculator in the near future.

MeshView by Gitlin and Johnson [GJ96] is a more recent tool that offers both visual and quantitative means of analyzing tetrahedral meshes. *MeshView* includes a "growth algorithm" visualization technique where one can specify a tetrahedron and the system grows the shape outwards by adding tetrahedra, based on connectivity. *MeshView* also supports some editing of a mesh. The *Shape Calculator*, through its interpreted language, supports a more general and flexible approach to the manipulation of shapes for mesh visualization and analysis. For example, the growth algorithm in *MeshView* could easily be expressed as a variation of the `Dilate` operation described later. The calculator does not currently support mesh editing, as found in *MeshView*, though the addition of such support to the *Shape Calculator* is a long term goal.

3 Definitions

3.1 Open and Closed Sets

Let $D(\mathbf{a}, \mathbf{b})$ be the Euclidean distance between points $\mathbf{a}, \mathbf{b} \subset \mathbb{R}^3$. An *open ball* centered at $\mathbf{c}$ with radius ρ is the set of points $B_\rho(\mathbf{c}) = \{\mathbf{x} \mid \mathrm{D}(\mathbf{x}, \mathbf{c}) < \rho\}$. A set $\mathcal{X} \subseteq \mathbb{R}^3$ is an *open set* if for all $\mathbf{x} \in \mathcal{X}$ there exists a $B_\rho(\mathbf{x}) \subset \mathcal{X}$ for some $\rho > 0$. The *complement* of $\mathcal{X}$ is $\mathcal{Y} = \mathbb{R}^3 - \mathcal{X}$. A *closed set* $\mathcal{X} \subseteq \mathbb{R}^3$ is a set whose complement is open. The *interior* of a set $\mathcal{X}$, Int $\mathcal{X}$, is the union of the open sets contained in $\mathcal{X}$. The *closure* of $\mathcal{X}$, Cl $\mathcal{X}$, is the intersection of all closed sets containing $\mathcal{X}$. The *boundary* of $\mathcal{X}$ is Bd $\mathcal{X}$ = Cl $\mathcal{X}$ − Int $\mathcal{X}$.

3.2 Simplices, Complexes and Shapes

A *k-simplex* is the convex hull defined by the set of $k+1$ points $T \in \mathbb{R}^3$. The points are assumed to be in general position, which in this case means that any point in T cannot be computed as an affine combination of the remaining points in T. In $\mathbb{R}^3$ we have 0, 1, 2 and 3-simplices, corresponding to vertices, edges, triangles and tetrahedra, respectively. The *dimension* of a k-simplex, Dim σ, is k. A subset of the set of points T defining a simplex τ defines another simplex σ, a *face* of τ. Simplices σ and τ are *incident* if one is the face of the other. A simplex σ defined by a proper subset of T is a *proper face* of τ. Tetrahedra, for example, have triangles, edges and vertices as proper faces. The *interior of a simplex* σ, Int σ, is the space occupied by σ minus the proper faces of σ.

A *simplicial complex* $\mathcal{K}$ is a collection of simplices satisfying two conditions:

1. If σ is in $\mathcal{K}$, then every face of σ is in $\mathcal{K}$.
2. The intersection of $\sigma_1, \sigma_2 \in \mathcal{K}$ is either empty or a face of both σ_1 and σ_2.

If $\mathcal{L}$ is a simplicial complex, then $\mathcal{K} \subseteq \mathcal{L}$ is a *subcomplex* if it also satisfies the simplicial complex conditions. The *i-skeleton* of a simplicial collection $\mathcal{C}$ is $\mathcal{C}^{(i)} = \{\sigma \in \mathcal{C} \mid \mathrm{Dim}\ \sigma \leq i\}$.

The *shape* of a simplicial collection $\mathcal{C}$, written $|\mathcal{C}|$, is $|\mathcal{C}| = \bigcup_{\sigma \in \mathcal{C}} \mathrm{Int}\ \sigma$. The shape of a simplicial complex is a closed set. Note that in the previous definitions, as well as in the following sections, we use $\mathcal{C}$ to signify an arbitrary subset of simplices, and $\mathcal{K}$ to indicate a simplicial complex.

4 The Shape Calculator Language

The shape calculator language and interpreter are designed to be easy to use and relatively unsurprising syntactically. The style is procedural, similar to that typically used in algorithm examples in the literature.

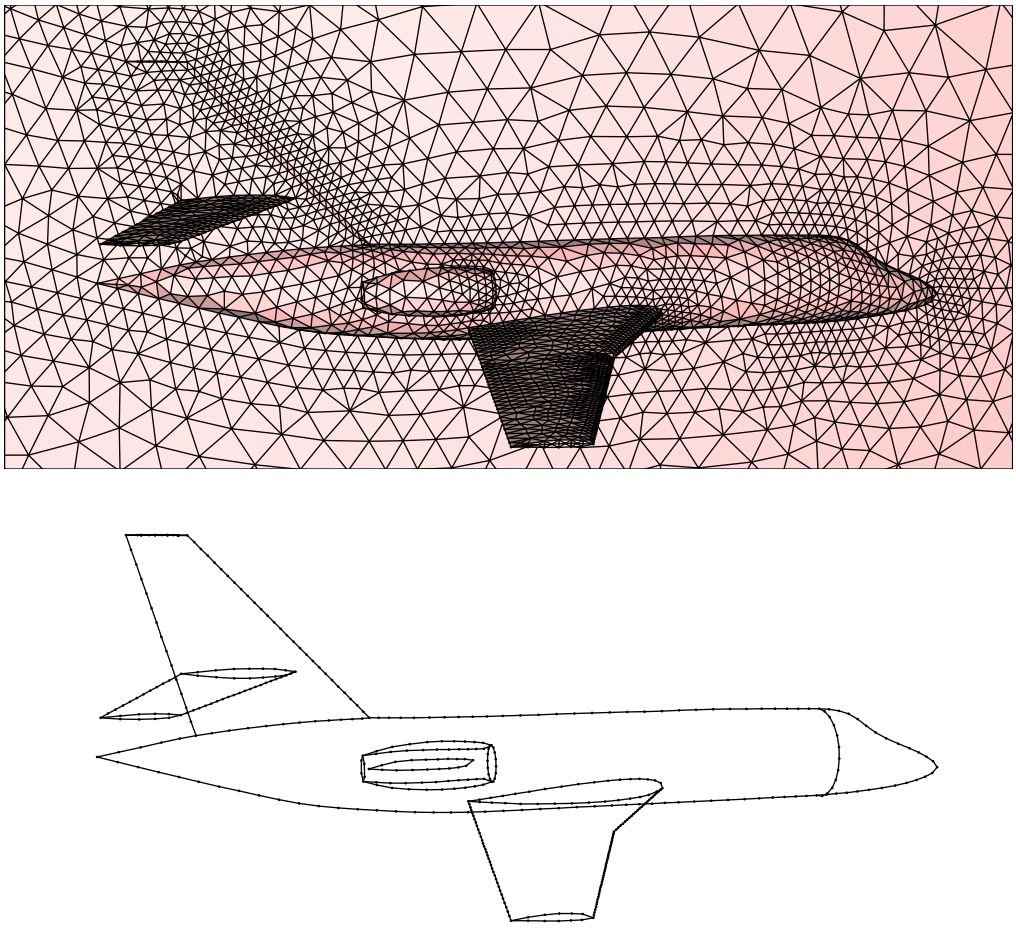


Figure 1: Shown above is a typical surface and symmetry plane mesh. In some cases sets of triangles specifying surfaces, such as the top and bottom of the wing, are included as part of a data set. Given a description of the surfaces, the calculator can compute an image such as that diplayed below, illustrating the seams between patches.

Within the calculator environment it is easy to give names to objects and to compute with them. With a bit more effort one can define small routines to customize the capabilities of the system and experiment with small algorithms. Below we consider some of the essential features of the language, in preparation for the examples to come in later sections.

4.1 Types and Collections

The shape calculator language includes built-in basic types common to many programming languages: booleans, integer scalars, floating-point scalars and character strings. The language also supports the representation of individual simplices. Identifiers beginning with the capital letters `T`, `F`, `E` or `V`, followed immediately by one or more digits, are treated as special literals specifying simplices. For example, the user can write `T1000` to specify the tetrahedron with the identity number of 1000. Similarly, one can specify triangles (faces), edges or vertices using special identifiers with `F`, `E` or `V` prefixes.

To represent collections, the language provides two types. The first, `SimplexSetExpr`, represents sets of simplices. `SimplexSetExpr` instances are used extensively in the calculator system, and most of the examples in the sections below work with `SimplexSetExpr` arguments. The second type of collection primitive is the array. Arrays are integer indexed collections of objects, and the range of valid indices are specified when the

Var.	Description
`$`	the currently displayed simplex collection
`$$`	the previously displayed simplex collection
`$0`	the empty collection of simplices
`$M`	the whole mesh

Table 1: Special calculator environment variables.

Operator	Boolean	Scalar	Simplex Set
(*unary*) `-`	not	negation	complement
`+`	or	addition	set union
`-`		subtraction	set difference
`*`	and	multiply	intersection
`/`		division	
`==`	equality	equality	equality
`!=`	inequality	inequality	inequality
(`<`), `<=`		(strictly) less than	(proper) subset
(`>`), `>=`		(strictly) greater than	(proper) superset

Table 2: The semantics for unary minus and the infix operators with various types of arguments.

array is constructed. For instance "a = `Array(1, 10)`" would construct a 10 element array named "`a`". Any type of object can be stored in each array element. Array elements can be accessed using square bracket syntax, for example one can write "`a[10]`". One can assign to an array element using the "`:=`" assignment syntax, for example "`a[5] := 0`".

4.2 The Calculator Environment

The calculator environment is a collection of name-value pairs, where each pair is also known as a *binding*. A *variable* is a name that references a binding; legal variable names for the *Shape Calculator* are similar to those in C or Pascal. The user has the choice of defining new variables or assigning to existing variables. The definition syntax uses the "`=`" token, for example "`my_variable = 3`". Assignment statements use the the "`:=`" token, for example "`v := 4`". In the *Shape Calculator* language a binding is global (*i.e.* can be referenced from anywhere after it is created) except if it is defined within the body of a user function. Bindings defined within a user function are limited to the body of the function.

The interpreter places a few special variables in the environment using the convention of starting the variable names with a "`$`" (see Table 1). The user can choose variable names starting with "`$`" and even assign to the special variables, but it is not recommended.

4.3 Infix operators

An infix operator such as `+` can be used in expressions where the operator is written in between two arguments. The *Shape Calculator* language supports the infix operators `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `<=`, `>` and `>=`. The infix operators are polymorphic, in other words they take on different (though conceptually similar) meanings depending on the type of the arguments. Many of the infix operators can accept boolean, scalar or simplex set arguments, though both arguments have to be of the same type. Table 2 lists the semantics for unary minus and the infix operators, given boolean, scalar or simplex set arguments.

Figure 1 illustrates a basic use of the boolean operators with simplex set arguments. The mesh input typically includes a specification of the triangles on the surface of interesting objects, for instance on the body of an aircraft. In some cases the surface patches are further classified, for example to distinguish whether a triangle is on the wing top or bottom. By taking the union of the patches, one can easily see whether there

are any surface triangles that are missing from the surface, and thus have not been classified. Furthermore, using intersection one can see if there are any triangles which have been assigned to more than one patch.

4.4 Topological operators

Modeling systems which feature boolean operators, such as those presented in the previous section, are not that unusual. In this section we present operators which are more likely to be found in texts on algebraic topology than in modeling systems (see for example Munkres [Mun84]). The topological operators make it easy to guarantee that a collection of simplices is a complex, to grow or shrink shapes, and to select the interior or boundary of a shape.

The topological operators are defined in terms of the simplices in $\mathcal{M}$ and their relationship as faces to one another. One way of visualizing this relationship is in terms of an *incidence graph*. An incidence graph represents each simplex as a node and includes an edge between the node for k-simplex σ and the node for $(k+1)$-simplex τ if σ is a face of τ. We draw the graph so that the k-simplices are all in one row, with the rows ordered by dimension, and the 0-simplices at the bottom. The left side of Figure 2 shows a small example complex with labeled vertices, and to the right is the corresponding incidence graph. For ease of illustration we use a two-dimensional example, though it should be clear that the same concepts can be applied in three or more dimensions.

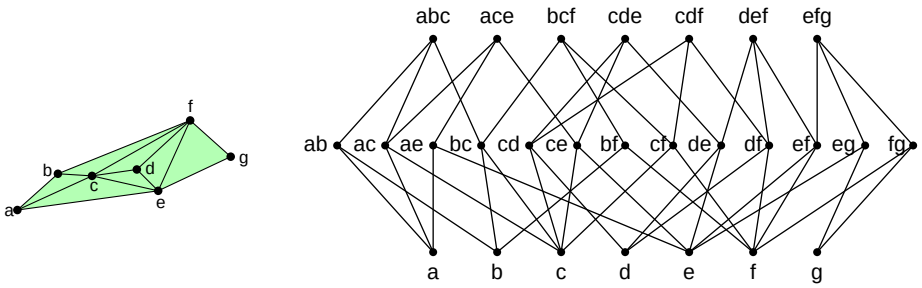


Figure 2: A small example mesh `$M` and the incidence graph for `$M`.

In the descriptions below, note that all of the topological operators can accept as an argument a collection of simplices that is not necessarily a complex. In cases where one prefers to work only with complexes it is easy to get a simplicial complex from any collection using the closure operator.

4.4.1 Closure

The *closure* of a simplex collection $\mathcal{C}$ is defined as

$$\texttt{Cl}(\mathcal{C}) = \{\sigma \in \mathcal{M} \mid \sigma \leq \tau \in \mathcal{C}\},$$



Figure 3: An example application of the `Cl` operator, given the argument `simplices` containing the vertex `f`, the edge `ab` and the triangle `cde`. Note that edges and vertices are only drawn if they are part of a collection. For example edge `cd` is not part of the original collection of simplices, but `cd` is in the `Cl` result.

where $\sigma \leq \tau$ means that σ is a face of τ. The closure operation adds any simplices necessary to make $\mathcal{C}$ a complex: $|\texttt{Cl}(\mathcal{C})|$ is the smallest closed superset of $|\mathcal{C}|$. Figure 3 illustrates the use of `Cl` with an argument consisting of an edge `ab`, a triangle `cde` and a vertex `f`. Note how the faces $\sigma \in \mathcal{M}$ added by closure (drawn from the whole mesh `$M` illustrated in Figure 2) can be found by following all possible paths downwards from the vertices `ab`, `cde` and `f` in the incidence graph of Figure 2.

4.4.2 Star

The *star* of a simplex collection $\mathcal{C}$ is defined as

$$\texttt{St}(\mathcal{C}) = \{\tau \in \mathcal{M} \mid \tau \geq \sigma \in \mathcal{C}\}.$$

The star operation adds any simplices necessary to get a shape that is the smallest open superset of $|\mathcal{C}|$. See Figure 4. In the example incidence graph of Figure 2, all the simplices in the star of 0-simplex `c` can be found by following the paths upward from node `c`. As with `Cl`, any added simplices are drawn from the whole mesh `$M` (illustrated in Figure 2).



Figure 4: An example application of the `St` operator.

4.4.3 Interior

The *interior* of a simplex collection $\mathcal{C}$ is defined as

$$\texttt{Int}(\mathcal{C}) = \{\sigma \in \mathcal{C} \mid \sigma \leq \tau \Rightarrow \tau \in \mathcal{C}\}.$$

The shape of the interior of $\mathcal{C}$ is the largest open subset of $|\mathcal{C}|$. One can see whether $\sigma \in \mathcal{C}$ is part of the interior using the incidence graph by starting at the node corresponding to σ and testing whether every node τ reachable by traveling upwards from the σ node is also in $\mathcal{C}$. Figure 5 illustrates an example use of `Int`.



Figure 5: An example application of the `Int` operator.

4.4.4 Boundary

The *boundary* of a collection $\mathcal{C}$ is defined as

$$\texttt{Bd}(\mathcal{C}) = \texttt{Cl}(\mathcal{C}) - \texttt{Int}(\mathcal{C}).$$

Figure 6 illustrates an example using `Bd`. In $\mathbb{R}^3$, the boundary of a solid object would include the triangles, edges and vertices defining the surfaces of the object. Using `Bd` one can convert from a solid representation of an object to a boundary representation.



Figure 6: An example application of the `Bd` operator.

4.5 Miscellaneous Operators

4.5.1 Skeleton

The i-skeleton of a collection $\mathcal{C}$ can be accessed using the `Sk` operator. `Sk` takes two arguments, the first should evaluate to a simplex set expression, the second to an integer specifying the value for i. For example, `Sk($, 0)` returns the vertices of the currently displayed simplicial collection.

4.5.2 Cardinality

The *cardinality* of a set S is a count of the number of elements in S. In the *Shape Calculator*, the cardinality of a simplex set expression can be accessed using the `Card` operator. For instance, `Card($M)` returns the number of simplices in the whole mesh. Given an optional second integer argument i, `Card` returns the cardinality of i-simplices in the first argument.

4.5.3 Show and Plot

The `Show` operator takes a single argument, and if that argument is a simplex set, `Show` displays it graphically. Given an argument which is not a simplex set, `Show` writes a textual representation of the object out to the interpreter window. `Show` is useful for displaying intermediate results of incremental algorithms and for producing simple animations.

Plots of arrays of values can be generated using the `Plot` command. The plotting functionality of the calculator is implemented using the application `Gnuplot` [WK95], which is run as a subprocess of the calculator. `Plot` takes one or more array arguments, all of which must have the same range of indices. `Plot` generates a temporary file where each line contains an array index value and the value at that index for each of the array arguments. `Plot` also accepts an optional final argument: a character string specifying additional commands to send to `Gnuplot`, such as commands specifying axis labels or which columns of numbers to use from the input file. An example where the `Plot` command is used to display a histogram can be found in Section 7.2.

4.5.4 Volume, SurfaceArea and CircumsphereRadius

The `Volume`, `SurfaceArea` and `CircumsphereRadius` operators take a single simplex argument and return the corresponding measure. The `Volume` operator can also take a `SimplexSetExpr` argument and return the volume of the object, computed by summing the volumes of the individual tetrahedra. `SurfaceArea` is also defined for a simplex set argument $\mathcal{C}$ as the area sum of one side of the regular triangles plus the area sum of both sides of the singular triangles, where regular triangles are the face of one tetrahedron in $\mathcal{C}$ and singular triangles are not the face of any tetrahedron in $\mathcal{C}$. Triangles which are the face of two triangles in $\mathcal{C}$ (*i.e.* interior triangles) do not contribute towards the area total.

5 Programming the Shape Calculator

Programming constructs enhance the power of the calculator primitives by allowing the user to define new operators in terms of existing operators. The calculator supports a large set of constructs based on standard

programming language features, such as "for" and "while" loops. Below we review a small subset of the constructs, for use in the following section.

In the *Shape Calculator*, every statement is required to return a value when evaluated. In this respect the language is similar to languages such as Scheme [AS85]. At first glance it may seem unnatural to define return values for some types of statements, but doing so ultimately makes the evaluation rules simpler and easier to remember.

5.1 Blocks

A *block* is a sequence of 0, 1, or more than 1 statements; multiple statements are separated using semicolons. Blocks are delimited by reserved words such as `then` and `endif`, as in the examples below. When a block is evaluated, the interpreter returns the value of the last statement in the block. If the block is empty, then a special `EmptyExpr` value is returned.

5.2 Conditional Expressions

Conditional expressions in the *Shape Calculator* language take one of two forms:

`if` *bool_expr* `then` $block_t$ `endif`
`if` *bool_expr* `then` $block_t$ `else` $block_f$ `endif`

Every `if` statement must end with an `endif`. An `if` statement is evaluated by first evaluating the *bool_expr* and then evaluating the true block or false block, depending on the boolean result. The value returned by an `if` statement is the value of the evaluated block. If the *bool_expr* evaluates to false and there is no `else` clause, then the special `EmptyExpr` value is returned.

5.3 "For" Loops

The calculator language supports two variations of "`for`" loop construction. The first form is similar to that used in the language C, with an initialization statement, a completion test statement and an increment statement, followed by the loop body. For example: "`for i = 1; i <= 10; i := i + 1 do sum := sum + a[i]; endfor`". The second form is applicable to collections represented by either an array or by a simplex set. The form requires a variable name, a collection and a block of statements forming the body of the loop. For each value in the collection, the given variable is assigned that value, and then the loop body is evaluated. For instance, to compute the volume "`v`" of a simplex collection "`c`", one could write: "`v = 0.0; for t in (c - Sk(c, 2)) do v := v + Volume(t); endfor`". Instances of both forms of "`for`" statement may be found in the histogram example in Section 7.2 below.

5.4 User-Defined Functions

User functions are defined with the following form:

`define` *funname* `(`*args*`)` *block* `enddefine`

The *args* list is a comma separated set of formal argument names. From the perspective of the user, arguments are passed by value, except for arrays. Within a user function, one can define new local variables using "`=`" statements, or assign to existing variables using "`:=`" statements. User functions can call themselves recursively. The value returned by a user function is obtained by evaluating the block representing the function body. Since the evaluation of every statement returns a value, an explicit `return` statement is not necessary.

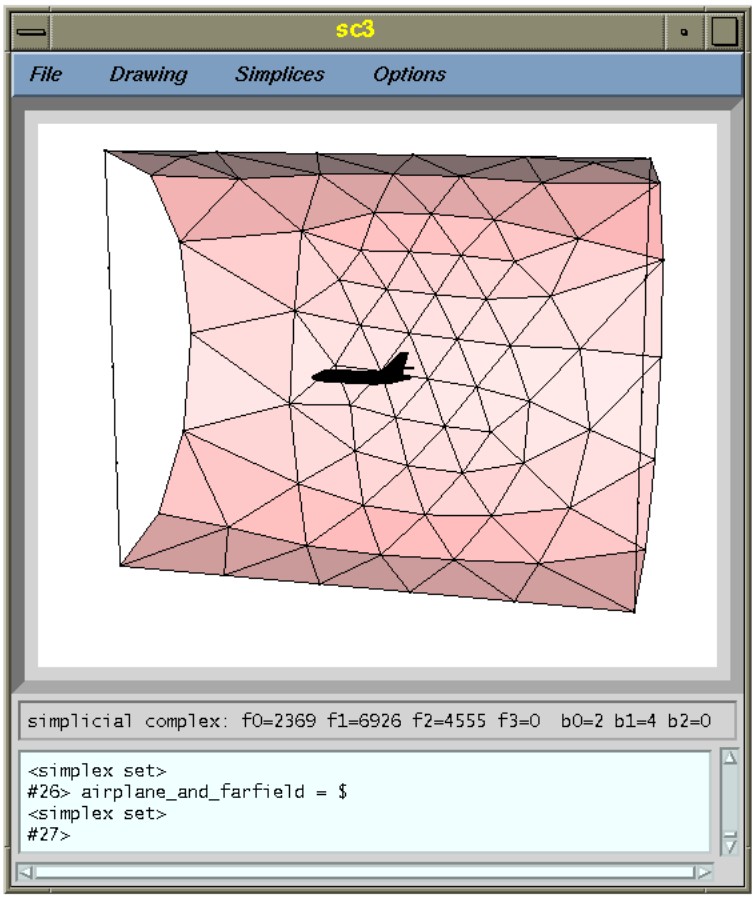


Figure 7: The *Shape Calculator* user interface.

6 Implementation

The *Shape Calculator* features an object-oriented design, implemented in C++. The interpreter portion of the system was written using the compiler development tools `lex` [Les75] and `yacc` [Joh75]. The interpreter parses user input into trees composed of `Expr` subclass instances. Interior nodes of the parse tree are made from `FuncallExpr` instances, for example to represent a function such as `Plus` and its two subtree children. There are several types of leaf nodes in a typical parse tree, such as integer scalar nodes and identifier nodes. The result of evaluating a parse tree is another `Expr` subclass instance. One of the most commonly occurring result types is `SimplexSetExpr`, which represents sets of simplices. Since every simplex σ that could be in a `SimplexSetExpr` is from a static, common universe mesh $\mathcal{M}$, it is not necessary to store detailed information about each σ in a `SimplexSetExpr` instance. Only a single bit for each $\sigma \in \mathcal{M}$, indicating whether σ is in a given instance, is necessary. The `SimplexSetExpr` class is implemented using bit vectors, one each for each dimension of simplex. Bit vectors support the efficient addition and removal of simplices from a collection, and boolean operations taking bit vectors as arguments can be computed quickly.

The user interface for the *Shape Calculator* is illustrated in Figure 7. The upper window displays the current collection of simplices, and the user can interactively rotate and zoom the collection using the mouse. In the lower window the user can type in and evaluate expressions. If the result of an evaluation is a `SimplexSetExpr` instance, then the graphics window is updated. The text line between the graphics window and the expression evaluation window is also updated with information about the current collection

of simplices. The values `f0` through `f3` list the number of 0-simplices through 3-simplices in the current collection. If the current collection is a simplicial complex, then `b0` through `b2` list the Betti numbers β_0 through β_2. The Betti numbers describe basic topological properties of the complex, for example β_0 specifies the number of connected components. The Betti numbers are computed interactively using an incremental algorithm developed by Delfinado and Edelsbrunner [DE95].

6.1 Performance

Tables 3 and 4 list example data sets and the typical performance of some basic operations applied to those data sets. The timings were made on an SGI Onyx2 (195 MHz R10000 microprocessor) with 512 MBytes of main memory. Workstations configured with less main memory could also run the *Shape Calculator* comfortably. The "Tori" data set appears later in Figure 11. The "Aircraft_2" data set appears in Figures 1 and 9.

The first two example operations involve the built-in primitives `+` (union) and `St`. Boolean operations such as union are computed by evaluating the operators with the corresponding bit vectors of the `SimplexSetExpr` arguments. Thus the boolean performance is primarily a function of the total number of simplices in $\mathcal{M}$ (*i.e.* the size of the bit vectors). The performance of the topological operators such as `St` is also dominated by the total number of simplices in $\mathcal{M}$, though the execution time is slightly slower given an argument with a large number of simplices. `Dilate`, described in Section 7.1, is an example of a simple user-defined operator combining `St` and `Cl`. Like `St` and `Cl`, the performance of `Dilate` is primarily influenced by the the number of simplices in $\mathcal{M}$.

The `Comp` operator, described in Section 7.3, supports the selection of connected components. `Comp` is defined recursively and its execution time varies a great deal, depending on how many recursive iterations are necessary. The two examples listed in Table 4 show best case and worst case conditions. The call `Comp(`*components*`,` *components*`)` represents the case where no recursion is necessary. On the other hand, `Comp(`*vertex*`,` *components*`)` represents the case where typically many iterations are necessary to grow the initial vertex to fill the selected regions. For instance, the second `Comp` example for the Aircraft_2 data set required 67 iterations in order to select the component.

The user function `volume_histogram_plot`, described in Section 7.2, is an example where a user routine processes individual simplices. In this case the routine obtains the volume of each tetrahedron as a prerequisite to computing a histogram. An example of the resulting plot, generated for the "Aircraft_2" data set, can be seen in Figure 10. Currently operators such as `Volume` and `CircumsphereRadius` compute their results on demand. The performance of routines using these operators could be increased if quantities such as volume were precomputed during the calculator initialization.

Data set	Vertices	Edges	Triangles	Tetrahedra	Total Simplices
Tori	800	6993	12197	6003	25993
Aircraft_1	13832	87587	143881	70125	315425
Aircraft_2	19048	122867	204034	100215	446164

Table 3: The data sets used for the performance tests.

7 Example User-Defined Functions

In this section we present examples demonstrating how new functions can be written using the calculator language. One of the greatest strengths of an interpreted environment is the ease with which one can interactively experiment within the system. The following examples are intended to give a small sample of how the system can be used.

Operation	Tori	Aircraft_1	Aircraft_2
+ (union)	1.1e-4	1.2e-3	1.6e-3
`St`	4.9e-2	4.7e-1	5.9e-1
`Dilate`	1.1e-1	8.6e-1	9.7e-1
`Comp(`*components*`,` *components*`)`	5.0e-2	4.6e-1	6.0e-1
`Comp(`*vertex*`,` *components*`)`	8.8e-1	3.4e1	4.9e1
`volume_histogram_plot`	1.4e0	1.6e1	2.2e1

Table 4: Timings for some typical operations in the *Shape Calculator* (in seconds).

7.1 Dilation and Erosion

Dilation and erosion operators allow the user to grow and shrink shapes—the `Dilate` and `Erode` operators below are inspired by similar operators in mathematical morphology [Ser82]. In mathematical morphology, one can think of shapes as represented by subsets of vertices from a graph. In $\mathbb{R}^2$, the vertices are typically placed in a regular rectangular or hexagonal pattern, for example corresponding to pixels in an image. Vincent [Vin89] considers the more general case where the vertices are not necessarily regularly positioned. The *Shape Calculator* generalizes on the representation of shape used in mathematical morphology by offering not only vertices but also higher-dimensional simplices to build shapes. Dilation and erosion in the calculator are defined as:

```
define Dilate(k) Cl(St(k)) enddefine;
define Erode(k) Cl(k - St(Bd(k))) enddefine;
```

As a first example of the use of `Dilate`, we consider a mesh visualization technique based on successive dilations. For ease of illustration, we begin with a mesh in $\mathbb{R}^2$, using an earlier implementation of the calculator. The upper portion of Figure 8 illustrates the whole mesh, $\mathcal{M}$. We define a new function that generates the boundaries of successive dilations as:

```
define dilate_boundaries(dilation, i, n)
  if i == n then
    Bd(dilation)
  else
    Bd(dilation) + dilate_boundaries(Dilate(dilation), i + 1, n)
  endif
enddefine;
```

Assuming that the simplices on the boundary of the airfoils have previously been assigned to the variable `airfoils`, we can visualize the mesh using:

```
dilate_boundaries(airfoils, 0, 5);
```

The lower half of Figure 8 illustrates the result. Note that we could also place a `Show` statement within the `dilate_boundaries` routine to view intermediate results as a simple animation. In $\mathbb{R}^3$ the same definition for `dilate_boundaries` would be applicable, and the dilation boundaries would in general be a surfaces. Each successive surface would tend to occlude the previous one, though the occlusion may not be a problem if the surfaces were viewed as an animation. One could also use `Erode` in a similar manner, except that one would start with the whole mesh and then strip away successive layers.

`Dilate` could have other uses as well. For example, consider the case where the the user may wish to obtain the vertices of the mesh that are close to, but not on the body of an aircraft. Using `Dilate`, the nearby vertices can be specified by

```
Sk(Dilate(airplane), 0) - airplane;
```

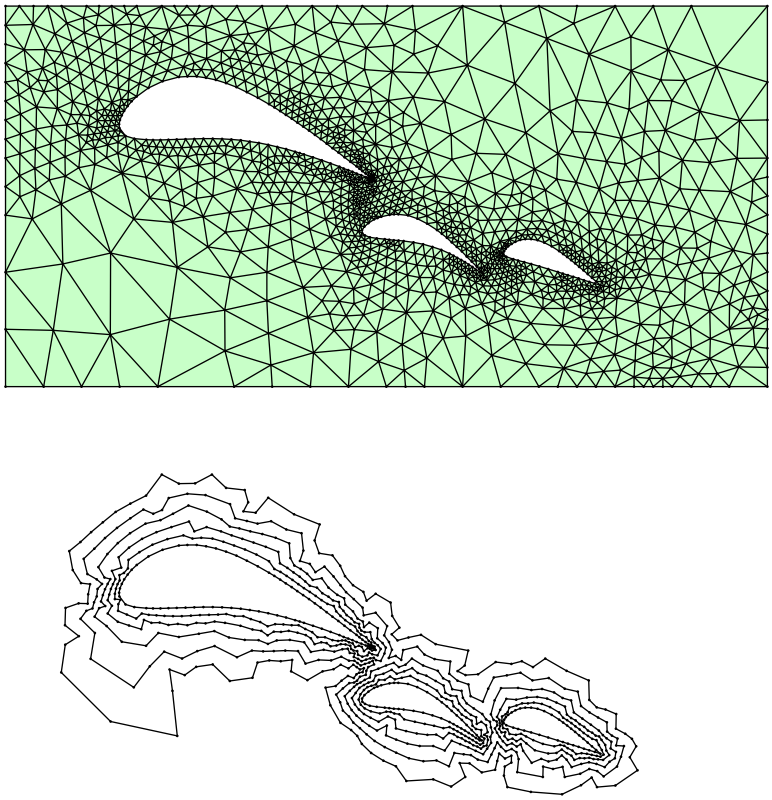


Figure 8: A mesh in $\mathbb{R}^2$ and successive dilation boundaries computed using `dilate_boundaries`.

where the simplices on the surface of the aircraft have been previously assigned to the variable `airplane`. `Sk(Dilate(airplane), 0)` returns the vertices both on and nearby the airplane body; after subtracting `airplane` only the off-body vertices remain. Figure 9 illustrates an airplane and the nearby vertices obtained with `Dilate`.

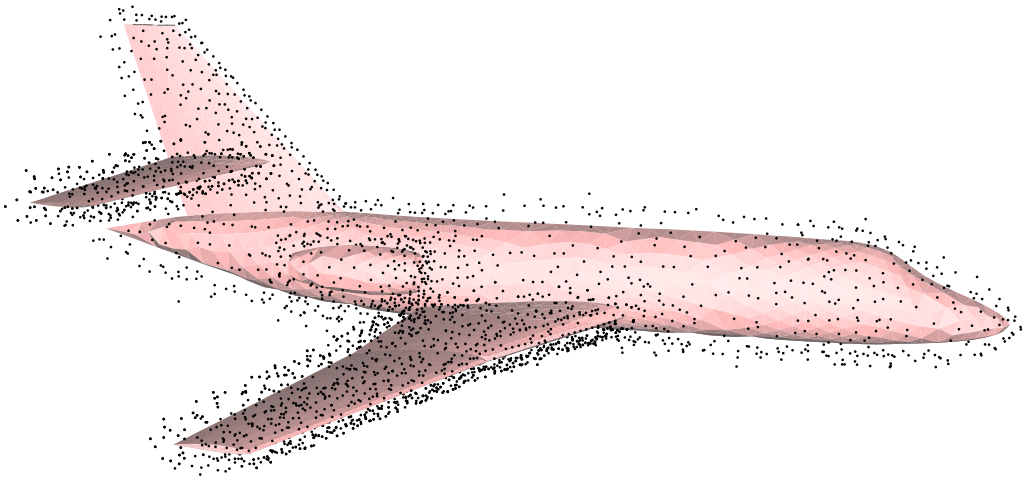
As in mathematical morphology, we can also combine dilation and erosion to get opening and closing operators:

```
define Open(k) Dilate(Erode(k)) enddefine;
define Close(k) Erode(Dilate(k)) enddefine;
```

The `Open` operation can be used to open gaps between objects which are tenuously connected, `Close` can be used to close small holes in objects. `Close` also has potential for use in grouping neighboring objects, such as individual vertices, into larger connected components.

7.2 Histogram Plotting

The built-in operators of the calculator provide basic primitives, such as `Volume` and `CircumsphereRadius`, which can be used to analyze individual simplices. Using the function definition of the language, one can build more interesting analysis routines, including plotting routines. As an example, the following two functions demonstrate how `Volume` and `Plot` could be used to compute and display a histogram showing a distribution of tetrahedral volumes:



surface_triangles + Sk(Dilate(airplane), 0) - airplane

Figure 9: An illustration of the Dilate operator (surface_triangles = airplane - Sk(airplane, 1)).

```
define histogram_plot(data, n_bins, min_range, max_range)
  bin = Array(0, n_bins - 1);
  mid_bin = Array(0, n_bins - 1);
  range = max_range - min_range;
  for i = 0; i < n_bins; i := i + 1 do
    bin[i] := 0;
    mid_bin[i] := min_range + (i + 0.5) / n_bins * range;
  endfor;
  k = n_bins / range;
  for i = MinIndex(data); i <= MaxIndex(data); i := i + 1 do
    index = Int((data[i] - min_range) * k);
    if 0 <= index and index < n_bins then
      bin[index] := bin[index] + 1;
    endif;
  endfor;
  Plot(mid_bin, bin, "set title 'Histogram'; plot '%s' using 2:3 with boxes");
enddefine;

define volume_histogram_plot(n_bins, min_volume, max_volume)
  tetrahedra = $M - Sk($M, 2);
  volumes = Array(1, Card(tetrahedra));
  i = 1;
  for t in tetrahedra do
    volumes[i] := Volume(t);
    i := i + 1;
  endfor;
  histogram_plot(volumes, n_bins, min_volume, max_volume);
enddefine;
```

Figure 10 illustrates an example plot computed with volume_histogram_plot. The histogram shows the distribution of tetrahedral volumes for the airplane data set used earlier in Figures 1 and 9. Note that

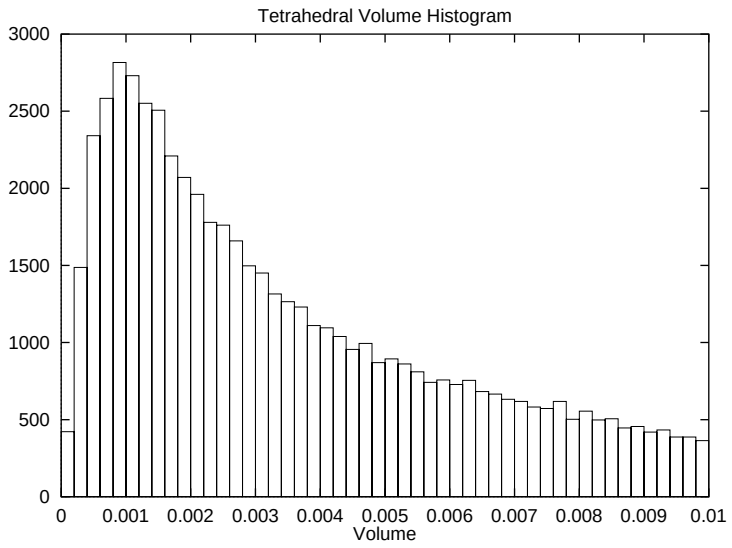


Figure 10: A histogram plot generated using `volume_histogram_plot` (volume in mesh coordinate units^3).

`volume_histogram_plot` takes arguments specifying the range of volumes to use for the plot, so tetrahedra with volumes outside the range are not counted in the histogram bins. For the plot shown in Figure 10, only tetrahedra with volumes between 0.0 and 0.01 are included in order to highlight the volume distribution for small tetrahedra.

7.3 Connected Components

A typical need when working with a shape consisting of several connected components is to choose some component subset. For example, one may want to measure the volume or surface area of individual components. In the *Shape Calculator*, connected components can be selected using the operators `Comp` and `OComp`. `Comp` expects both of its arguments to be closed sets, `OComp` expects both arguments to be open sets. Both operators start with a set of selected seed simplices and a set of regions to choose from. Both repeatedly dilate the selected shapes, restricting the dilation growth to the given regions. Termination occurs when there is no more growth in the restricted dilation. The two operators are implemented as:

```
define Comp(selected, regions)
  selected1 = St(selected) * regions;
  if selected1 > selected then Comp(Cl(selected1), regions) else selected endif;
enddefine;

define OComp(selected, regions)
  selected1 = Cl(selected) * regions;
  if selected1 > selected then OComp(St(selected1), regions) else selected endif;
enddefine;
```

The definitions for `Comp` and `OComp` differ only in the exchanged roles of `Cl` and `St`. Note that more than one connected component can be selected at a time by either operator.

For example, consider a domain containing two interlinked tori. The whole mesh, `$M`, contains the two tori surfaces as well as simplices exterior to the tori. `$M` is pictured in the upper left of Figure 11. The tori surfaces, assigned to the variable `surfaces`, are pictured in the upper right. Assume we are given a complex consisting of a single vertex on the closer torus surface, assigned to the variable `seed`. The result of `Comp(seed, surfaces)` is shown in the lower right of Figure 11, along with the 0-skeleton of the tori surfaces for reference. Given one torus surface, `surface1`, we could obtain the second torus surface by `surfaces - surface1`.

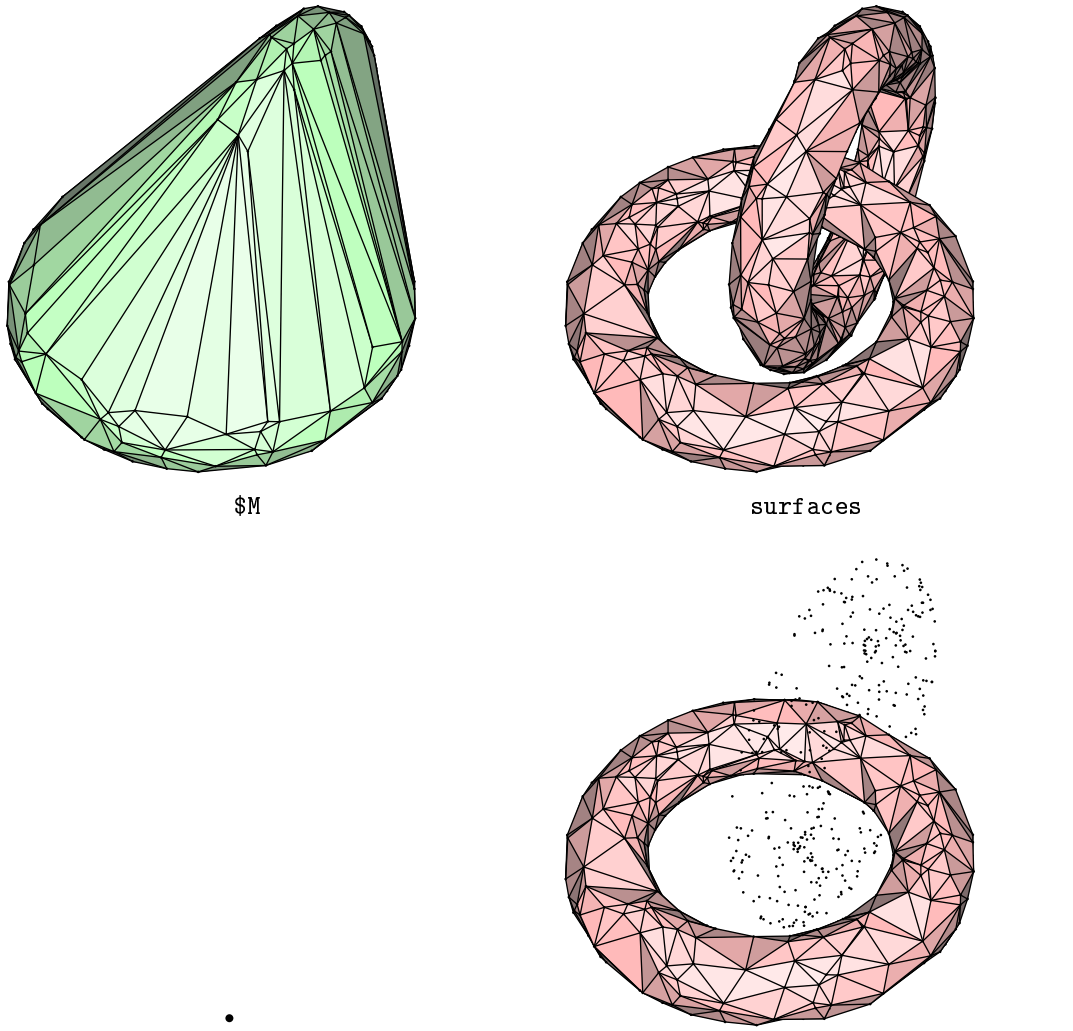


Figure 11: An illustration of the `Comp` operator.

8 Conclusion

We have presented a language supporting the visualization and analysis of unstructured meshes. The language contains a relatively small set of primitives, yet is powerful enough to perform operations which help reveal the structure of a mesh and assist the user in analyzing a mesh. The primitives presented in this article apply to simplicial complexes in spaces other than $\mathbb{R}^3$. Furthermore, the primitives of the language could easily be generalized to work with complexes which are not composed exclusively of simplicial cells. The incidence graph, which represents the face relationships between cells, readily generalizes to the non-simplicial case, and primitives such as `St` and `Cl` could easily work with other types of cell complexes. Thus the shape calculator language could provide a common means for manipulating shapes based on either structured or unstructured meshes.

In the short term future, we anticipate adding more geometric capabilities for analyzing cells, such as primitives for measuring edge lengths and various angles. In the longer term, we anticipate adding the capability to access the numerical solution data associated with the mesh, when available. Such data could be used as an argument to cell selection criteria, or perhaps as data for shading the shapes.

Acknowledgements

This work was completed in part under the support of NASA contract NAS2-14303. The initial *Shape Calculator* research was conducted at the National Center for Supercomputing Applications in Urbana, Illinois.

Our thanks to Chris Henze and Sam Uselton for reviewing early drafts of this work. Our thanks as well to Ernst Mücke for providing the tori point set used in Figure 11 and the Delaunay triangulation software `detri` used to generate the mesh for the tori example. We would also like to thank Professor Herbert Edelsbrunner and the α-shapes research group at the University of Illinois at Urbana-Champaign for some of the original inspiration for this work.

References

[AS85] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[B+90] G. Bancroft et al. FAST: A multi-processed environment for visualization of computational fluid dynamics. In *Proceedings of Visualization '90*, pages 14–24. IEEE Computer Society Press, October 1990.

[DE95] C. Delfinado and H. Edelsbrunner. An incremental algorithm for Betti numbers of simplicial complexes on the 3-sphere. *Computer Aided Geometric Design*, 12:771–784, 1995.

[GJ96] C. Gitlin and C. Johnson. Meshview: A tool for exploring 3d unstructured tetrahedral meshes. In *5th International Meshing Roundtable*, October 1996.

[Joh75] S. Johnson. Yacc: yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Les75] M. Lesk. Lex — a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Mun84] J. Munkres. *Elements of Algebraic Topology.* Addison-Wesley, Redwood City, California, 1984.

[PBCF93] A. Paoluzzi, F. Bernardini, C. Cattani, and V. Ferrucci. Dimension-independent modeling with simplicial complexes. *ACM Transactions on Graphics*, 12(1):56–102, January 1993.

[PFP95] V. Pascucci, V. Ferrucci, and A. Paoluzzi. Dimension-independent convex-cell hpc: Representation scheme and implementation issues. In *Proceedings of the Third Symposium on Solid Modeling and Applications*, pages 163–174, May 1995.

[Ser82] J. Serra. *Image Analysis and Mathematical Morphology.* Academic Press, 1982.

[SK90] D. Speray and S. Kennon. Volume probes: Interactive data exploration on arbitrary grids. *Computer Graphics*, 24(5):5–12, 1990.

[Vin89] L. Vincent. Graphs and mathematical morphology. *Signal Processing*, 16:365–388, 1989.

[WK95] T. Williams and C. Kelly. Gnuplot: An interactive plotting program.
URL: `http://www.cs.dartmouth.edu/gnuplot_info.html`, 1995. Version 3.2.