

A Mesh-Database-Independent Edge- and Face-Swapping Tool

Carl Ollivier-Gooch*

*Advanced Numerical Simulation Laboratory
The University of British Columbia*

SUMMARY

Software re-use is an increasingly common practice in development of scientific computing software. Recently, there has been an emphasis on the definition of software *components*: software objects that use a clearly defined interface to encapsulate a specific functionality. In particular, at least two such component definitions have been developed for low-level mesh functionality and at least one of these already has several working implementations. These components open the door for the development of mesh manipulation services — such as swapping, smoothing, edge collapse, face splitting, etc — that are independent of the internals of the underlying mesh database; in turn, higher-level operations can be built on top of these.

This paper describes a software component for simplicial mesh swapping in both two and three dimensions. Applications can control the cell quality measure used as a swapping criterion, including defining new quality measures. The swapping component is coded using the Terascale Simulation Tools and Technologies (TSTT) mesh component, and is therefore usable with any mesh database that implements the TSTT mesh interface. Results are given comparing the new software component with a database-specific implementation of the same swapping algorithm. The two approaches produce statistically indistinguishable meshes. The overhead for the TSTT-based swapping component can be as low as about 50% compared with the native implementation; analysis of remaining sources of overhead is also given. Copyright © 2007 John Wiley & Sons, Ltd.

Abbreviations

EH	entity handle	SO	storage order enum	Type	entity type enum
iter	entity iterator	Topo	entity topology enum	VH	vertex handle
SH	entity set handle				

1. Introduction

Modern numerical solvers for physical problems described by partial differential equations require the use of sophisticated techniques in a variety of areas, from discretization schemes to numerical linear algebra to managing mesh and geometry data. Because most applications programmers prefer to focus on the physics of their problem rather than on numerical issues, they wisely choose to use existing software tools to the

*Correspondence to: Advanced Numerical Simulation Laboratory, Department of Mechanical Engineering, 6250 Applied Science Lane, The University of British Columbia, Vancouver, BC V6T 1Z4.

extent that this is possible. For example, numerical linear algebra is one specialized area in which successful software libraries[5, 2, 3] have long been available.

From the point of view of applications programmers, the drawback to library use is challenges with interchangeability. Because each library has a different applications programming interface (API), replacing one library with another requires a significant amount of programming and as such is not undertaken lightly, if at all. This is a direct barrier to use of newly developed algorithms and tools. A better solution for interchangeability is software *components*, defined as software objects that use a clearly defined interface to encapsulate a specific functionality. Because algorithms and/or data representations may differ significantly between component implementations of the common API, changing from one implementation to another may provide a significant benefit, *and* there is no re-programming required in the applications code.

In recent years, researchers working in meshing have started to develop software components. The two most notable efforts to date are the Unstructured Grid Consortium (UGC) [20] and the Terascale Simulation Tools and Technologies (TSTT) [18] project. The first version [19] of the UGC standard focused on very high-level mesh functionality, including mesh generation and quality assessment. Recognizing a need for lower-level functionality, the UGC working group has recently published a second version [21] of their interface, including primitive mesh query and modification operators as well as a mechanism for accessing arbitrary third-party mesh manipulation tools. The TSTT project is broader in scope, aimed not only at developing a mesh component, but also components for geometry query and manipulation and for representing discretized solution and other field data [7]. We will focus here on the mesh component,[15] which contains a full suite of functions for low-level mesh query and modification.

Both the UGC and TSTT mesh modification interfaces contain only primitive operations, describable verbally as “create a vertex here” or “destroy this triangle”. Such operations are the assembly language of mesh modification: any complex operation can be built up out of these, but considerable effort and knowledge is required to do it properly. The next higher level of sophistication (analogous to compiled languages) is the creation of components for mesh modification operations that begin and end with a valid mesh. Operations that are candidates for componentization at this level include creating a vertex and connecting it topologically with the mesh; edge / face collapse; vertex smoothing; and local mesh topology improvement. These components can be implemented using a low-level mesh component API like UGC or TSTT, so that they will be portable across all implementations of that component API. In this scenario, for instance, two different swapping schemes could be interchanged simply by relinking an application program. Likewise, high-level components, like mesh generation or adaptation, could be built on top of mid-level components, again leveraging investment in the base level component technology.

Note also that the mid-level operations discussed above are often specific to a particular type of mesh or particular meshing algorithm; for instance, correct vertex insertion is a very different process for Delaunay refinement schemes than for advancing front schemes because of how insertion affects mesh topology. Because individual mesh databases are inevitably written with a bias toward a particular mesh type and approach, not all of the available mesh databases will necessarily have a native implementation of a given mid-level operation. Hence, portability between mesh databases of higher-level algorithms (including, for instance, mesh generation and adaptation) built on these mid-level tools will be awkward to achieve if the mid-level functions are incorporated directly into a mesh component definition. At the very least, reference implementations of mid-level tools based on a common low-level interface are required; a component designed to provide a given service independent of a particular mesh database could easily act as such a reference implementation.

This paper describes a new mid-level meshing software component for local topology improvement of simplicial meshes (i.e., swapping) written to use in turn a standard low-level mesh component API for all mesh query and modification operations. The swapping algorithms used are described in Section 2. The

mesh interface used in this work has been developed by the Terascale Simulation Tools and Technologies (TSTT) consortium [18]; brief descriptions of the TSTT data model and the relevant parts of the TSTT mesh interface are given in Section 3. Section 4 provides some insights into the process of converting a database-specific tool to use the TSTT API, as well as giving performance results for use with one TSTT implementation. A detailed specification of the swapping component is given in Appendix II.

2. Swapping Algorithms

The goal of swapping algorithms is to improve geometric mesh quality by improving mesh topology. In broad terms, each $d - 1$ dimensional mesh entity is examined in turn to determine whether it is advantageous to modify mesh topology locally. This decision typically compares the current configuration with one or more possible alternative configurations to determine which maximizes some measure of geometric mesh quality.

The most common techniques for simplicial mesh topology change are edge swapping in two dimensions and its three-dimensional analog, face swapping.

In two dimensions, edge swapping chooses the best diagonal for the quadrilateral formed by two neighboring triangles; the quadrilateral must of course be convex for edge swapping to be performed. The diagonal can in principle be chosen on the basis of any triangle-based quality measure. Among the most common quality measures is the Delaunay criterion, which requires that the circle defined by the vertices of a triangle have no vertices in its interior[†]; minimizing the maximum angle is also a common choice.

[Figure 1 about here.]

In three dimensions, reconfiguration is more complex. The canonical cases are exchanging two tetrahedra that share a face with three tetrahedra sharing a common edge, as shown in the top left part of Figure 1; or its inverse swap, from three to two tetrahedra. In addition, two tetrahedra may be exchanged for two (T22 case in the figure); in this case, the two shaded faces must be co-planar, and swapping decisions reduce to choosing the best diagonal for the coplanar quadrilateral. If two pairs of tetrahedra in the interior of the mesh share a pair of coplanar faces, this swap is also permitted when two T22 configurations are back-to-back in the mesh. In addition to these swappable configurations, there are a number of unswappable cases, some of which are illustrated in the bottom of Figure 1.

[Figure 2 about here.]

For some unswappable configurations, the two tetrahedra sharing a face take the general form of two of the three tetrahedra in the T32 case, but the third tetrahedron is missing from the mesh, replaced instead by two or more tetrahedra. Figure 2 shows such a case with five tetrahedra incident on the central edge, TB , which is perpendicular to the page with T toward the reader. In such cases, the N tetrahedra incident on edge TB can be replaced by $2N - 4$ tetrahedra, each incident on exactly one of T and B , a process called *edge swapping*. In this example, the five original tetrahedra ($01BT$, $12BT$, $23BT$, $34BT$, and $40BT$) are replaced by six new tetrahedra, two for each of the triangles of the (non-planar) triangulation of polygon 01234 : $012T$, $024T$, $234T$, $021B$, $042B$, and $324B$.

[†]In two dimensions, maximizing the minimum sine of angles is equivalent to applying the Delaunay criterion.

The challenge with edge swapping is that the number of possible configurations grows rapidly with the number of tetrahedra incident on the edge to be removed, as seen in Table I. Clearly, checking the quality of each tetrahedron in each possible configuration is a costly undertaking; instead, an efficient implementation will compute the quality for each unique tetrahedron only once, then determine the quality of a given configuration by finding the minimum quality among its tetrahedra. In practice, the number of successful 7-for-10 swaps is very small, so exploring possible swaps for more complex initial configurations is not worthwhile.

[Table 1 about here.]

[Figure 3 about here.]

Bookkeeping is simplified by taking advantage of the symmetries of the post-edge-swapping configurations; this allows an implementation to store only a small set of canonical configurations, as shown in Figure 3, including post-swap connectivity information. For full details of this approach to edge swapping, see Freitag and Ollivier-Gooch [11].

In three dimensions, the Delaunay criterion is again often used for face swapping. Other choices have been shown to be better for improving extremal angles, including minimizing the maximum dihedral angle and especially maximizing the minimum sine of dihedral angles, and are therefore preferable in practice for computation [11]. For edge swapping, tetrahedron-based quality measures are required, eliminating the Delaunay criterion from consideration. Several excellent discussions of geometric mesh quality measures are available in the literature [4, 13].

Regardless of the quality measure used, a global mesh swapping scheme must iterate over edges (2D) or faces (3D), performing reconfiguration as required. Changing mesh topology can change whether nearby faces test as needing to be swapped. In this case, immediately checking nearby edges/faces for swapping, as shown in in Figure 4, can significantly reduce the need for making multiple passes over all edges/faces in the mesh [6], and is recommended practice.

[Figure 4 about here.]

3. Overview of the TSTT Mesh Interface

The TSTT mesh interface [8, 9, 15] is built on a data model that defines three basic data types required for general meshing operations: entities, entity sets, and tags. For data structure neutrality, objects of all these types are identified by unique, opaque handles. These handles must be invariant, in the sense that a given entity must always have the same handle; note that after an entity is deleted, a new entity may re-use its handle.

All mesh primitives — vertices (0D), edges (1D), faces (2D), and regions (3D) — are referred to as *entities*. Entities are defined with topologies to match all the usual non-degenerate finite elements, including canonical orderings for the lower-dimensional entities bounding them. Entity adjacency relationships describe topological relationships between entities. In particular, a first-order adjacency request for an entity of dimension d returns all entities of dimension p that lie on the closure of the requesting entity ($d > p$; example: all triangles bounding a tetrahedron) or all entities of dimension q for which the requesting entity is part of the closure ($d < q$; example: all tetrahedra with a common vertex). Note that this definition implies that there are no adjacent entities of dimension d ; such a request would require finding, for instance, faces incident on the vertices adjacent to a face.

Entities can be collected arbitrarily into *entity sets*. The overall mesh database is referred to as the *root set*; all entities and all other entity sets are necessarily members of this set. To be computationally useful, the root set or one or more of the entity sets it contains must be a valid computational mesh covering the physical domain. This coverage can be in the form of a simple, conformal mesh of the domain or a more complex patched or chimera mesh system. Entity sets can be nested, allowing (for example) an application to represent all entities on a geometric face as a set, with entities on geometric edges of that face as subsets. Also, hierarchical relationships between entity sets are supported, enabling parent-child relationships between levels in a multigrid scheme or successive meshes in a refinement sequence. Finally, set Boolean operations — intersection, union, and subtraction — are defined on entity sets.

Tags can be used to associate arbitrary application-defined data (either scalar or vector) with any entity or entity set. A single tag can have different data associated with different entities; for instance, a boundary condition tag might have a value indicating a no-slip wall for some faces on the boundary, a value indicating a far-field condition for other faces on the boundary, and no value at all for interior faces. Specific functions exist for three common types of tag data: integers, doubles, and entity handles; in addition, arbitrary data can be stored as an array of bytes.

The TSTT mesh interface encapsulates a variety of commonly needed and supported functionality for mesh and entity query and mesh modification, as well as entity sets and tags. A detailed description is given elsewhere [7, 9, 15], but in summary form the defined functionality includes:

1. A basic mesh interface containing global queries. These include functions requesting properties of the mesh database as a whole and query functions for an entire entity set. Examples of the latter include requests for all entities of a given type and topology and requests for all vertex coordinates. The interfaces defined below are explicitly extensions of the basic mesh interface.
2. An entity-based traversal and query interface, including both traversal of all entities of a given type and topology and analogs of the global query functions for single entities.
3. A block traversal and query interface. This is analogous to the entity-based interface, but with data accessed in blocks of user-specified size instead of entity by entity.
4. A mesh modification interface, providing functionality to create, destroy, and move vertices and to create and destroy higher-dimensional entities.
5. A block mesh modification interface, analogous to the single-entity modification interface.

The TSTT interface is designed to be not only data-structure neutral, but also programming language neutral. That is, a mesh server can be written in one language and client code in another. The TSTT interface is specified using an interface description language (SIDL), and translated into language-specific interfaces through a tool called Babel [14, 10]. Babel also generates glue code that mediates all inter-language issues, including function name mangling and passage of string and array arguments. As an example of how this works in practice, consider the case of a request for mesh adjacency information. An application code using the TSTT interface makes an adjacency request by calling a *stub* function (auto-generated by Babel) in the language of the application. This function re-packages function arguments and calls an *internal object representation* function (auto-generated by Babel, in C), which again repackages arguments and calls a *skeleton* function (auto-generated by Babel) in the language of the server. Finally, this function calls the server implementation of the original SIDL function. This approach eliminates all language-specific issues, including name mangling schemes and the treatment of strings and arrays, including dynamic array handling. In exchange, four versions of each SIDL function exist (three of which are autogenerated), and a call from client code must pass through all these layers. Not surprisingly, this complexity in call sequences can have a significant impact on application efficiency.

4. A TSTT-Based Implementation of Mesh Swapping

4.1. Implementation

Interactions between a swapping algorithm and a mesh database can be categorized roughly into iteration over $d - 1$ -dimensional mesh entities; mesh topology and geometry queries to support decisions about whether swapping will improve mesh quality; and local mesh modifications.

[Table 2 about here.]

For swapping, we choose to use TSTT's single entity iterator capability, which is supported by a suite of four functions which create an iterator; increment an iterator and retrieve data; reset the iterator; and destroy the iterator. These functions are summarized in Table II. In two dimensions, the swapping routine iterates over edges, and in three dimensions, over triangular faces. This application is actually a severe test for a TSTT iterator implementation, because the edge or face that was to be retrieved next by the iterator may be removed from the mesh by swapping; this is especially true with recursive swapping, but in three dimensions such deletions can occur even without recursion and independent of mesh database implementation. The TSTT specification requires that `getNextEntIter` return false and output an invalid handle if there are no entities remaining at or after the input value of the iterator.

[Table 3 about here.]

Making a correct swapping decision requires identifying all d -dimensional candidate entities that would be involved in the possible swap and assessing mesh quality both before and after the proposed reconfiguration. In two dimensions, the candidate triangles are the two that share the edge to be swapped. In three dimensions, the situation is more complicated. The candidate face must be categorized based on the topology and geometry of its neighborhood, as discussed in Section 2. For swappable configurations (i.e., those shown in the upper part of Figure 1), quality assessment can proceed at once. For configurations that are candidates for edge swapping, further local mesh interrogation is required to identify all tetrahedra incident on the candidate edge. Collectively, this information gathering stage requires extensive use of entity adjacency and vertex coordinate queries, as well as entity type and topology queries to confirm, for example, that all candidate regions are tetrahedra. Some of these adjacency calls are most efficiently implemented using the block adjacency call; unfortunately, from an efficiency point of view, the block sizes are never more than four, so the efficiency gains are small. The requisite mesh query functions are summarized in Table III. Not all possible adjacency requests are exercised by the swapping driver. Two-way adjacency information between $d - 1$ -dimensional and d -dimensional entities must be supported by the implementation, as well as adjacency between these entities and vertices; no upward adjacency information from vertices is required. The offset array return by `getEntArrAdj` gives information about where the list of adjacent entities for an input entity begins. The storage order argument to `getVtxArrCoord` can be used to specify whether coordinate data is returned in blocked (`xxx...yyy...zzz...`) or interleaved (`xyzxyzxyz...`) order; if this argument is unspecified on input, the implementation returns the data in its default storage order, with the value of this default returned in `SO`; the swapping code requests interleaved coordinate data.

Having identified the local submesh affected by a possible swap, the next step is to assess quality of the initial and final configurations to determine whether the swap is beneficial. This quality assessment is done using two functions defined as part of the swapping component. One of these functions tells the swapping driver whether to choose the configuration that maximizes the quality measure or the one that minimizes it, while the other assesses quality given the vertex locations for a candidate triangle (2D) or tetrahedron (3D). See Appendix II for syntactic details.

Once the decision to swap has been made, mesh topology must be modified locally. Using the TSTT interface, this is done by deleting old entities and creating new ones; changing adjacency for existing entities is not allowed. Note that deletion of old entities must proceed from high-dimensional entities to low-dimensional entities (i.e., tetrahedra must be deleted before their faces), while creation proceeds from low dimensions to high dimensions. Entity creation calls require the topology of the entity to be created and an array of equal-dimension entities from which to create it; that is, a tetrahedron can be constructed from vertices, from edges, or from faces, but not from a face and a vertex. Entity creation returns a handle for the new entity and a status variable indicating whether creation succeeded. These functions are summarized in Table IV.

[Table 4 about here.]

In addition to the TSTT calls discussed above, the swapping code assumes the presence of functions to get and set entity *classification*, defined as the relationship between a mesh entity and an entity in the geometric model of the domain. Classification information is essential both for identifying internal boundaries in the mesh and for correct boundary swapping. In the former case, tetrahedra (triangles, in 2D) from different subdomains would classify onto different geometric regions (faces); swapping such mixed neighborhoods would jumble the internal boundary. On domain boundaries in 3D, triangles can be classified onto different geometric surface faces; again, swapping should be prevented when surface triangles classify on different geometric faces to avoid producing invalid meshes. In the TSTT framework, classification is properly described as a relationship between mesh and geometry data, placing it outside the mesh interface proper. Instead, these relationships are managed by using the TSTT relations interface (TSTTR). Table V summarizes the three TSTTR functions used internally by the swapping service to update the mesh-to-geometry classification information;[‡] removal of classification information for entities that are being deleted is necessary in case the implementation re-uses handles of deleted entities. Note also that an application wishing to use the swapping service must initialize classification data prior to swapping.

[Table 5 about here.]

4.2. Verification

Once the TSTT-based swapping code was implemented, it was tested to confirm that it was correct, in the sense that it produced the same meshes as a database-specific implementation of the same swapping routines. In this case, the GRUMMP [17, 16] libraries were used as the reference implementation for swapping and as the implementation of the TSTT mesh interface.

[Figure 5 about here.]

To test the performance of the TSTT swapping code, several poor quality meshes were swapped using the TSTT swapping code; the GRUMMP swapping code was used as a direct, fair performance comparison. In two dimensions, poor-quality test meshes were created by taking good meshes and deliberately applying swapping to *reduce* the mesh quality. Figure 5 shows close-ups of typical mesh connectivity for these sample meshes before and after using the swapping code to improve quality. Also shown in the figure are angle statistics for two meshes, showing poor initial quality and excellent final quality after swapping the meshes

[‡]Existing implementations of TSTTR can be used to provide this functionality; an application wishing to provide mesh support for the swapping service can link to one of these instead of providing TSTTR capability internally.

to meet the Delaunay criterion. In each case, the database-specific and TSTT-based implementations gave identical output meshes, aside from meaningless differences in the ordering of edges and cells.

[Figure 6 about here.]

In three dimensions, the test meshes used were two random meshes in a cube that were previously used for testing mesh quality improvement by Freitag and Ollivier-Gooch [11]. Swapping was first performed using the Delaunay criterion, then using the maxmin sine criterion, as this combination was found to be the most effective in improving mesh quality. In three dimensions, due to differences in the order of operations performed during recursive swapping, mesh results were not precisely identical. However, in all cases studied, the output meshes were statistically identical in quality, as shown in Figure 6, and the number of swaps required varied by less than 2.5%. Because of variations in the exact number of swaps performed, performance data will be reported in terms of relative swapping rate.

4.3. Performance

The next major question for the swapping module is how its performance compares with the database-specific version. The initial TSTT-based implementation was a translation of the database-specific version: essentially, database-specific function calls were replaced one-for-one with TSTT calls, and temporarily variables were created as necessary to hold argument values. Performance testing and profiling indicated that this naive approach performed poorly, primarily for three reasons. First, many more calls through the TSTT interface were made than was strictly necessary, especially in retrieving vertex coordinates and adjacency information; these calls were combined by using block calls instead. Second, many of the TSTT calls in innermost loops require small temporary arrays which are comparatively expensive to create and destroy; these arrays were replaced by static work arrays, eliminating the creation and destruction overhead. Finally, significant time was spent in using function calls to set and retrieve data in SIDL arrays, both in the swapping service and in the underlying implementation; these calls were replaced by direct access to array data by exposing the underlying pointer to this data. Taken together, these three classes of efficiency improvements gave about a factor of three reduction in overhead for the TSTT-based implementation. These results suggest several best-practice principles to apply in programming using the TSTT interface, and give an estimate of the cost of violating those principles.

Table VI shows results for several test cases. The differences in CPU time between the TSTT and native implementations are largely due to calling overhead: the glue code produced by Babel to ensure that inter-language calls work properly requires several levels of function calls, which is a significant overhead for function calls with a very short body. Much of the remaining overhead is related to packing and unpacking adjacency and coordinate information into contiguous arrays. Data in the column labeled “-Babel” used the TSTT interface, but in a non-standard way that bypasses the Babel glue code. While the details of this approach to reducing overhead are not applicable when the mesh database is implemented in a different language than the swapping tool, these results do suggest that there is room for significantly reducing the overhead associated with the TSTT interface, perhaps by replacing calls to TSTT functions with macros that automatically produce language-specific calls to server functions. This is currently an area of active work for the TSTT interface development group.

[Table 6 about here.]

4.4. Overhead Analysis

The overhead that remains after bypassing the Babel glue code represents work done by the TSTT implementation in servicing requests from the swapping driver that is not required by the native

implementation of the same swapping algorithms using direct access to the GRUMMP data structures. Other TSTT implementations would doubtless have different amounts and distributions of overhead, depending on details in implementation both of the underlying mesh database and of the TSTT access functions. Nevertheless, some operations are likely to be common sources of overhead, so examination of the breakdown of overhead for the current implementation may be instructive.

Overhead was identified and quantified through careful examination of data gathered by callgrind (the valgrind profiling tool [1]) for all TSTT calls used by the 3D swapping code. Table VII shows the breakdown of the overhead that remains after bypassing the Babel glue code.

Nearly 57% of this overhead comes from the three functions that retrieve mesh adjacency and coordinate data (`getEntAdj`, `getEntArrAdj`, and `getVtxArrCoords`), including subsidiary calls to `getEntType`. Whereas the native implementation accesses this same data directly through inline functions, the TSTT functions transcribe the data retrieved using these same inline functions into arrays that are returned to the caller. Also, the native implementation is always aware of what entity type it is retrieving adjacencies for, while the TSTT calls must first identify the type of entity to be able to correctly retrieve adjacency information; these calls are necessary in the TSTT implementation, but are pure overhead compared with the native implementation.

The second-largest category of overhead is entity creation and deletion, at about 18%. Much of this overhead, both for creation and deletion, is required to check for possible duplicate entities in the mesh, a check which is not performed in the native implementation, because the GRUMMP data model does not allow duplicate entities with the same lower-dimensional entities and its mesh modification functions never create them. Most of the rest of the creation/deletion overhead is for dynamic casts from generic pointers to GRUMMP native type pointers.

Classification of mesh entities onto the geometry — reading, writing, and erasing — takes 14% of time in the TSTT implementation. The GRUMMP native implementation handles this information by tagging entities directly and inline, at essentially zero cost. The final category of overhead is SIDL array overhead. SIDL arrays are reference counted, so whenever one is passed by value, its reference count must be incremented, then decremented again on exit.

[Table 7 about here.]

5. Conclusions

This paper has described a mesh swapping tool built on a standard mesh component (the TSTT mesh interface). The portable swapping tool has been shown to give statistically identical results to a database-specific implementation of the same algorithms. Performance results indicate that there is significant overhead (50-100%) incurred by using the TSTT mesh component architecture. However, it appears that much of this overhead can be eliminated by bypassing the extra layers of glue code produced by Babel to address client/server language differences; doing this in a portable way is an active area of research. Also, it should be noted that the nature of data access patterns for swapping make this a worst-case scenario for overhead in using a mesh component. The same approach used in developing this database-independent mesh swapping tool can also be applied to produce similar tools for other primitive mesh operations, such as incremental vertex insertion and edge collapse, which in turn could provide a basis for building database-independent high-level mesh modification algorithms, including mesh generation and adaptation.

Acknowledgments

This work has been funded by the Canadian Natural Sciences and Engineering Research Council under Discovery Grant OPG-0194467 and Special Research Opportunities Grant SRO-299160.

REFERENCES

1. Valgrind. <http://valgrind.org>, 2000–2007.
2. Lapack webpage. <http://www.netlib.org/lapack/>, 2004.
3. Linpack webpage. <http://www.netlib.org/linpack/>, 2004.
4. T. J. Baker. Deformation and quality measures for tetrahedral meshes. European Congress on Computational Methods in Applied Sciences and Engineering, Sept. 2000. ECCOMAS 2000.
5. S. Balay, K. Buschelman, D. Gropp, W.D. Kaushik, M. Knepley, B. McInnes, L.C. Smith, and H. Zhang. PETSc home page. <http://www.mcs.anl.gov/petsc>, 2004.
6. T. J. Barth. Aspects of unstructured grids and finite-volume solvers for the Euler and Navier-Stokes equations. In *Unstructured Grid Methods for Advection-Dominated Flows*, pages 6–1 – 6–61. AGARD, Neuilly sur Seine, France, 1992. AGARD-R-787.
7. K. Chand, L. F. Diachin, C. Ollivier-Gooch, M. Shephard, and T. Tautges. Toward interoperable mesh, geometry and field components for PDE simulation development. *Submitted to Engineering with Computers*, 2005.
8. K. Chand, B. Fix, T. Dahlgren, L. F. Diachin, X. Li, C. Ollivier-Gooch, E. Seol, M. Shephard, T. Tautges, and H. Trease. The TSTT Base Interface. http://www.tstt-scidac.org/software/TSTTB_userguide.pdf, Oct. 2005.
9. K. Chand, B. Fix, T. Dahlgren, L. F. Diachin, X. Li, C. Ollivier-Gooch, E. Seol, M. Shephard, T. Tautges, and H. Trease. The TSTT Mesh Interface. http://www.tstt-scidac.org/software/TSTTM_userguide.pdf, Oct. 2005.
10. T. Dahlgren, T. Epperly, G. Kumpf, and J. Leek. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, Livermore, California, version 0.9.4 edition, 2004.
11. L. A. Freitag and C. F. Ollivier-Gooch. Tetrahedral mesh improvement using swapping and smoothing. *International Journal for Numerical Methods in Engineering*, 40(21):3979–4002, 1997.
12. B. Joe. Construction of three-dimensional improved quality triangulations using local transformations. *SIAM Journal of Scientific Computing*, 16:1292–1307, 1995.
13. P. Knupp. Algebraic mesh quality metrics. *SIAM Journal of Scientific Computing*, 23(1):133–149, 2001.
14. Lawrence Livermore National Laboratory. Babel. <http://www.llnl.gov/CASC/components/babel.html>, 2004.
15. C. Ollivier-Gooch, K. Chand, B. Fix, T. Dahlgren, L. F. Diachin, J. Kraftcheck, X. Li, E. Seol, M. Shephard, T. Tautges, and H. Trease. The TSTT Mesh Interface. AIAA Paper 2006-0529. Presented at the 44th AIAA Aerospace Sciences Meeting, Jan. 2006.
16. C. F. Ollivier-Gooch. An unstructured mesh improvement toolkit with application to mesh improvement, generation and (de-)refinement. AIAA 98-0218, Jan. 1998.
17. C. F. Ollivier-Gooch. GRUMMP — Generation and Refinement of Unstructured, Mixed-element Meshes in Parallel. <http://tetra.mech.ubc.ca/GRUMMP>, 1998–2005.
18. The Terascale Simulation Tools and Technology (TSTT) Center. <http://www.tstt-scidac.org>, 2003.
19. Unstructured Grid Consortium. Unstructured Grid Consortium standards document, version 1. <http://www.aiaa.org/tc/mvce/ugc/ugcstandv1.pdf>, June 2002.
20. Unstructured Grid Consortium. Unstructured Grid Consortium webpage. <http://www.aiaa.org/tc/mvce/ugc/>, March 2002.
21. Unstructured Grid Consortium. Unstructured Grid Consortium standard, version 2. http://www.aiaa.org/tc/mvce/ugc/API-doc/ugc_2.0.3.tar.gz, July 2005.

APPENDIX

II. SIDL Description of the Mesh Swapping Component

The mesh swapping service described in this paper has been implemented as a particular case of the following SIDL interface. Applications using this swapping tool would require client-side stubs for these functions, generated as described in the Babel User's Guide [10]. Also, an implementation of the TSTTM interface is required; several are available through the TSTT web site [18].

The TSTT_Swap package provides functions for setting and querying the internal state of a swapping

implementation, including the swap criterion to be used; whether recursive swapping is allowed; the mesh database to operate on; and in three dimensions whether edge swapping and boundary reconfiguration are permitted. The function `swap` decides whether to swap away a given mesh entity; `swapAll` iterates over the entire mesh until no more swaps that improve mesh quality are possible.

Also, the package provides a uniform framework for implementing new, user-defined quality measures. Using the `QualMeasure` interface, any quality measure computable based on vertex coordinates for a triangle (2D) or tetrahedron (3D) can be implemented. Also, the swapping driver can be told whether this measure should be maximized or minimized.

```

package TSTT_Swap {
  enum SwapType {
    DELAUNAY,
    MAX_MIN_SINE,
    MIN_MAX_ANGLE,
    USER_DEFINED
  }

  interface QualMeasure {
    bool shouldMinimize() throws TSTTB.Error;
    double calcQuality(in array<double> coords, in int coords_size) throws TSTTB.Error;
  }
  interface Swap {
    void setSwapType(in SwapType ST) throws TSTTB.Error;
    /* The following implies setSwapType(USER_DEFINED) */
    void setUserQualMeasure(in QualMeasure QM) throws TSTTB.Error;
    SwapType getSwapType() throws TSTTB.Error;

    void setSwapRecursion(in bool allow_recursion) throws TSTTB.Error;
    bool isRecursionAllowed() throws TSTTB.Error;

    void setMeshData(in TSTTM.Modify MMod) throws TSTTB.Error;
    void setModelData(in TSTTG.Model GModel) throws TSTTB.Error;
    void setAssocData(in TSTTR.Associate Assoc_Data) throws TSTTB.Error;

    int swap(inout opaque entity_set, inout opaque entity_handle)
      throws TSTTB.Error;
    int swapAll(inout opaque entity_set)
      throws TSTTB.Error;
  }
  class Swap2D implements-all Swap {};
  class Swap3D implements-all Swap {
    void setEdgeSwapping(in bool allow_edge_swapping)
      throws TSTTB.Error;
    bool isEdgeSwappingAllowed() throws TSTTB.Error;
    void setBdryMutable(in bool allow_bdry_changes) throws TSTTB.Error;
    bool isBdryMutable() throws TSTTB.Error;
  }
}

```

```
class Delaunay3D implements-all QualMeasure {};  
class MaxMinSine3D implements-all QualMeasure {};  
class MinMaxAngle3D implements-all QualMeasure {};  
}
```

List of Figures

1	Face swapping in three dimensions. Labels for configurations are from Joe's taxonomy of local mesh configurations.[12]	14
2	Edge swapping example	15
3	Canonical configurations for edge swapping, including the number of unique rotations of each.	16
4	Swap recursion example. If edge AC is swapped for BD , then edges AB , BC , CD , and DA should be checked recursively.	17
5	Two-dimensional swapping test cases. Mesh quality after swapping was independent of whether the TSTT or GRUMMP swapping code was used.	18
6	Mesh quality for three-dimensional swapping test cases.	19

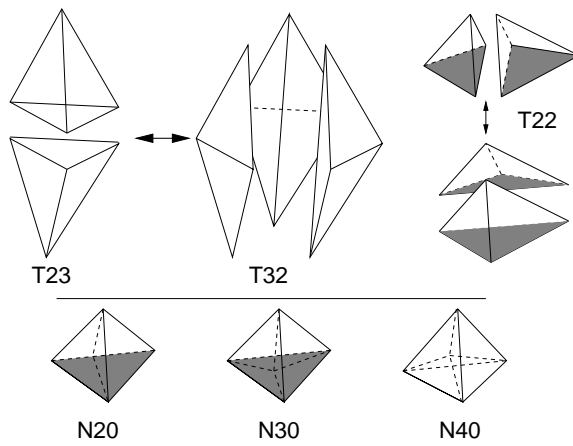


Figure 1. Face swapping in three dimensions. Labels for configurations are from Joe's taxonomy of local mesh configurations.[12]

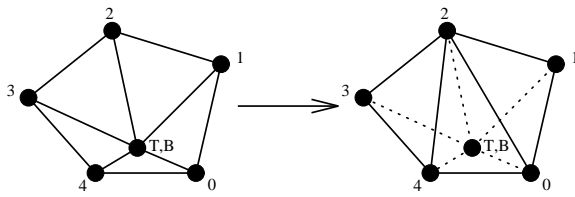


Figure 2. Edge swapping example

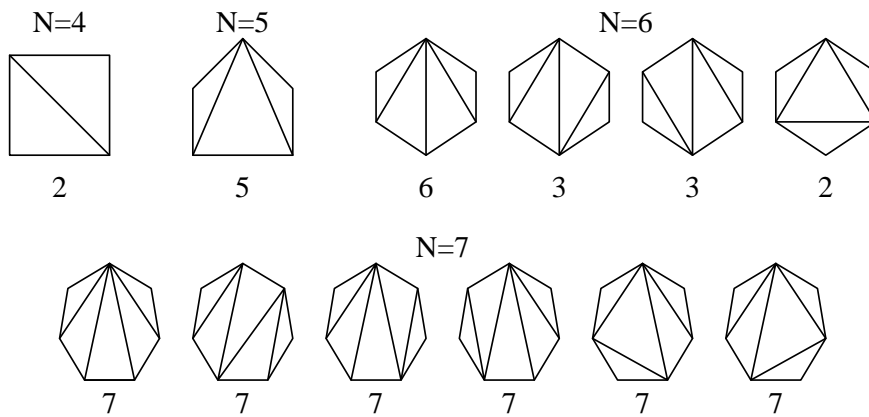


Figure 3. Canonical configurations for edge swapping, including the number of unique rotations of each.

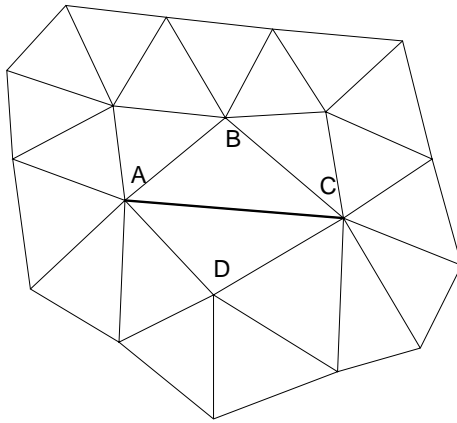
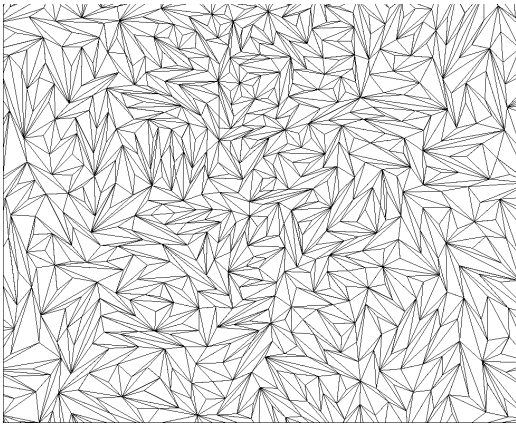
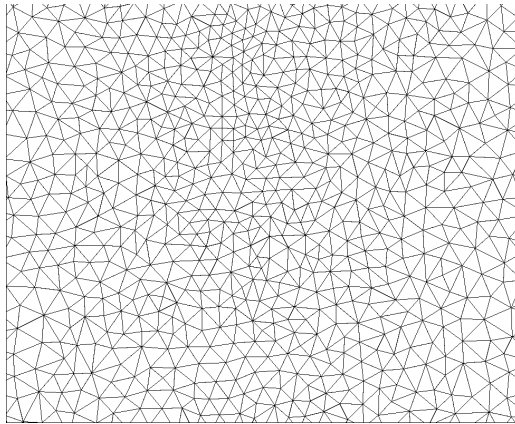


Figure 4. Swap recursion example. If edge AC is swapped for BD , then edges AB , BC , CD , and DA should be checked recursively.



(a) Close-up of poor-quality mesh before swapping.



(b) Close-up of good-quality mesh after swapping.

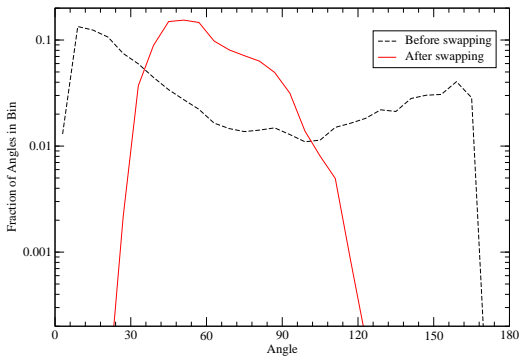
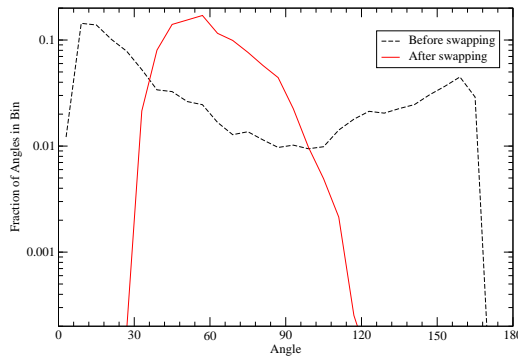
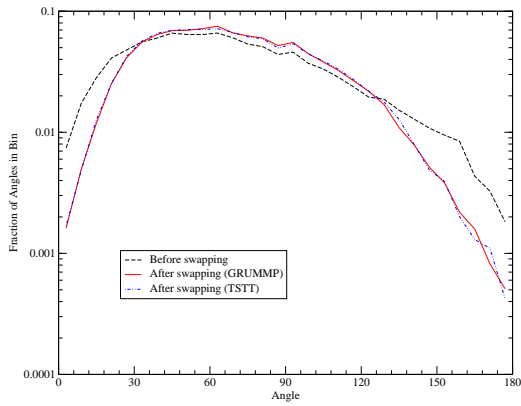
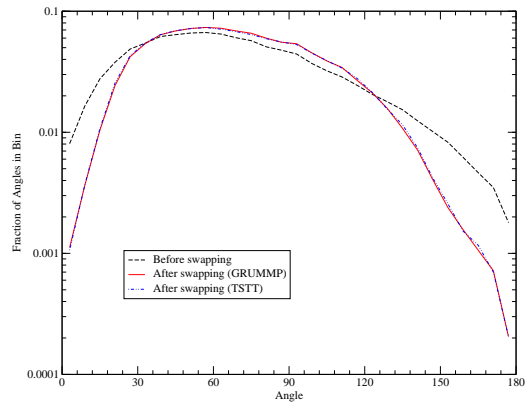
(c) Mesh quality for *Hole* test case(d) Mesh quality for *Circle* test case

Figure 5. Two-dimensional swapping test cases. Mesh quality after swapping was independent of whether the TSTT or GRUMMP swapping code was used.



(a) Mesh quality for *Rand1* test case



(b) Mesh quality for *Rand2* test case

Figure 6. Mesh quality for three-dimensional swapping test cases.

List of Tables

I	Size of edge swapping configurations, including the number of possible candidate tetrahedra.	21
II	Semantic summary of TSTT iterator functions used by current swapping code.	22
III	Semantic summary of TSTT mesh query functions used by current swapping code.	23
IV	Semantic summary of TSTT mesh modification functions used by current swapping code. . .	24
V	Semantic summary of TSTT relationship functions used by current swapping code.	25
VI	Performance results for TSTT-based swapping. Swap type D = Delaunay, S= max min sine of dihedral angle.	26
VII	Breakdown of overhead for TSTT-based swapping, after eliminating Babel overhead, for a three-dimensional case, expressed as percentages of total overhead. Totals do not equal 100 because of rounding.	27

Table I. Size of edge swapping configurations, including the number of possible candidate tetrahedra.

Tets before	3	4	5	6	7
Tets after	2	4	6	8	10
Configurations	1	2	5	14	42
Tets \times configs	2	8	30	112	420
Unique tets	2	8	20	40	70

Table II. Semantic summary of TSTT iterator functions used by current swapping code.

Name	Input args	Output args	Return value
initEntIter	SH, Type, Topo	iter	true iff data exists
getNextEntIter	iter	iter, EH	true iff data exists
resetEntIter	iter	—	—
endEntIter	iter	—	—

Table III. Semantic summary of TSTT mesh query functions used by current swapping code.

Name	Input args	Output args	Return value
getEntType	EH	—	Type of entity
getEntTopo	EH	—	Topo of entity
getEntAdj	EH, Type	EH array	—
getEntArrAdj	EH array, Type	EH array, offset array	—
getVtxArrCoord	VH array, SO	SO, coord array	—

Table IV. Semantic summary of TSTT mesh modification functions used by current swapping code.

Name	Input args	Output args	Return value
deleteEnt	EH	—	—
createEnt	Topo, EH array	EH, CS	—

Table V. Semantic summary of TSTT relationship functions used by current swapping code.

Name	Input args	Output args	Return value
getEntEntAssociation	Mesh EH	Geom EH	—
setEntEntAssociation	Mesh EH, Geom EH	—	—
rmvEntEntAssociation	Mesh EH, Geom EH	—	—

Table VI. Performance results for TSTT-based swapping. Swap type D = Delaunay, S= max min sine of dihedral angle.

Test case			Swapping			Relative CPU time		
Name	2D/3D	# elem	Type	Passes	Swaps	Native	TSTT	-Babel
Hole	2D	4,309	D	2	3,411	1	2.18	1.98
Circle	2D	14,496	D	1	12,095	1.08	2.15	2.09
Rand1	3D	5,104	D+S	2+3	10,926	1.01	1.86	1.59
Rand2	3D	25,704	D+S	3+4	67,796	1	1.50	1.29

Table VII. Breakdown of overhead for TSTT-based swapping, after eliminating Babel overhead, for a three-dimensional case, expressed as percentages of total overhead. Totals do not equal 100 because of rounding.

Category		Function	
Data retrieval	56.9%	getEntArrAdj	26.2%
		getVtxArrCoords	16.3%
		getEntAdj	10.1%
		getEntType	4.3%
Creation and deletion	17.9%	createEnt	6.2%
		deleteEnt	6.3%
		Dynamic casts	5.4%
Classification	14.2%	getEntEntAssociation	5.7%
		setEntEntAssociation	4.4%
		rmvEntEntAssociation	4.0%
SIDL arrays	10.9%	Delete reference	4.5%
		Opaque array*	3.6%
		Add reference	2.9%