Design and Implementation of a Corporate Mesh Object

John T. Svitek, Wa Kwok, Joseph R. Tristano

{john.svitek,wa.kwok,joe.tristano}@ansys.com

ANSYS, Inc

ABSTRACT

Today, finite element technologies allow engineers to analyze complex assemblies and subsystems. With CPU power constantly increasing, it is not unreasonable to state that the engineer will hope to analyze the whole complex system, such as an entire automobile, in a single study. This study may include parameteric design, result animation, crash analysis, and so on. The traditional mesh data structure, which mainly serves a particular type of mesh algorithm, is far from enough to meet the challenges of tomorrow. This paper mainly focuses on storing, accessing, and manipulating mesh data within the vast scope of the analysis system for multiple purposes, such as meshing generators, solvers, pre- and post-processors, and so on. It details the decision-making put into the design of the ANSYS Corporate Mesh Object, the programming methods used to implement those designs, and future enhancements planned to meet ever-changing requirements.

1. INTRODUCTION

As machines have become more powerful and finite element analysis methods have become more elaborate, the amount of finite element data that must be managed and stored has grown to previously unimaginable levels. The vast amount of data produced by today's meshing technology needs to be readily available to several different users, accessing it in several different ways, all the while fitting in the small memory space afforded by a Windows-based PC. The data needs to be accessed quickly for graphical support, accurately for solving, but also easily for third-party users. In the face of these challenges, ANSYS Incorporated noticed the need to replace its existing (and very simplistic) mesh data structure with a much more flexible and powerful data structure, an ANSYS Corporate Mesh Object or ACMO.

1.1 Purpose and Previous Work

The main purpose of the ACMO as described in this paper is to act as a repository to store nodes and elements, whether created by a meshing algorithm or manually created by the user. This type of mesh object does not contain full mesh generation data such as the edge or face representations of each element, and the data is not expected to substantially change very often. Therefore, this object is not suited for use as a data structure for the actual meshing process, when nodes and elements are being generated, deleted, and changed continuously. Such data structures have been detailed extensively in the past, often coupled to a particular algorithm [4,5,6,7,8,10], or rigorously tailored to the output of a particular algorithm [3,9]. Rather, once a mesh has been successfully and completely created, the ACMO acts as a common location to store the data, and provides easy access to the data from a range of different users. Much previous research is available on the various ways of representing mesh data [1,2], though the ACMO attempts to also address the usability and accessibility of the data.

1.2 Background

In the ANSYS Workbench software, geometry is stored in a sophisticated object called the Part Manager (Fig. 1), which encapsulates the geometry data in easy to understand concepts such as “Assembly”, “Part”, “Body”, “Face”, “Edge”, and “Vertex” as well as several other layers in between. An assembly is a collection of parts, which in turn is a collection of boundary representations. When a user asks Workbench to mesh geometry, each part in turn is meshed, and all these part meshes are then collected. In this way, a continuous mesh exists between bodies, but not between parts. Nodes are not duplicated at places where bodies share topology.

Unlike the Part Manager, the original mesh data structure used in the ANSYS Workbench software took a very simple approach to storing the nodes and elements created during meshing. After each part was meshed, the nodal coordinates were stored in an array, and the element’s nodal connectivity was stored in an array, along with topological information. There was no real distinction between different bodies in the mesh, and one part mesh had no relationship whatsoever to another part mesh. Managing the various part meshes was the duty of the overall system; for example, a two part meshed assembly had two nodes numbered zero and it became the system’s job to differentiate between those two nodes.

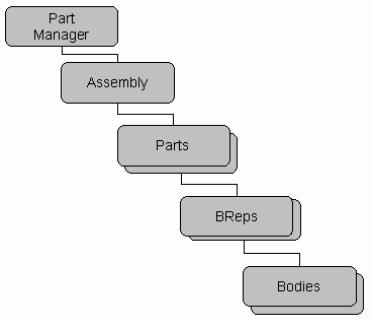

Fig. 1. Part Manager

1.3 Need for Change

Originally, the mesh data structure used in ANSYS Workbench was sufficient because there never was a part containing more than one body. The mesh was not exposed to many uses beyond graphics and solving. There was very little data actually stored in the mesh and any newly required data could be programmed in as necessary.

However, as the system around the mesh data structure matured, the mesh itself had trouble keeping up. The concept of a multiple-bodied part was introduced into the Part Manager, but the mesh was still stored part by part so that old code using the mesh would not need to be changed. The nodes and elements were stored exactly as before, so accessing them on a body-by-body basis became a time-consuming operation.

Furthermore, the mesh was not easily extensible. If a user wanted to store some data on each element, such as a value describing the volume of the element, a new array of values would need to be added to the data structure, along with all supporting implementation such as saving, resuming, copying, clearing, setting, and getting. If a user then wanted to store another value per element, such as a metric value, the entire process would have to be repeated.

As machines became more powerful, larger and larger meshes were being generated. Not long ago, one million nodes could be considered a large mesh, but a few short years later, ten million nodes were on the horizon. Such enormous amounts of data were putting strain on the mesh data structure, which did not have any kind of memory management system. Memory fragmentation became a very real and very serious concern as other uses of the same memory space (such as solvers) were unable to allocate enough large-block memory to operate.

New uses of the mesh were constantly being developed, and users often wanted to access the data in the mesh in a certain way. A user would need to query for an element based on its index on the part, or maybe its index on a particular body in that part. Further, the nodal connectivity of that element could be returned using nodal indices on either the part or the body. As a result, the user interface to the mesh became monstrous, so that every permutation of data access could be afforded to the user.

1.4 Future Challenges

With the need to integrate different mesh technologies into ANSYS Workbench, standardizing the mesh object became even more paramount. For example, ICEM CFD, Cadoe, and CFX all brought new technologies to the ANSYS product line, but they all brought different data representations that were not compatible with each other or with existing ANSYS software. In order to move data between the various technologies, a corporate mesh object that could be used by everyone was necessary.

2. DESIGN

2.1 Use Types

Before detailing the design process on the ACMO, it is important to illustrate the many different ways a single mesh could be used. (Fig. 2)

- **Mesher**: After the meshing process, the nodes and elements need to be stored into the mesh object. Meshing is done part-by-part, and one part has no bearing on or relationship to another part. Mesher only needs to add data to the mesh object.
- **Refiner**: An existing mesh can be refined one part at a time. The entire part mesh is removed from the assembly, and replaced with a new mesh after the refinement is finished. The refiner must be able to easily access all data in a part mesh, and to add data to the mesh object.
- **Graphics**: The mesh is drawn to the graphics screen body-by-body. In order to minimize its footprint in memory, graphics requires a reference to the mesh data, instead of a copy of the data. All nodes of a body are required, and nodal connectivity of elements must be returned such that it references the body-based indices of the nodes.
- **Preprocessor**: Data, such as beam orientation nodes and contact or surface effect elements are added by a separate preprocessor to the solver
- **Solver**: The solver requires all nodes and elements across the entire assembly. References to the data are necessary to minimize memory usage. All nodes and elements must be uniquely identified across all parts.
- **Postprocessor**: The post processor performs calculations on the mesh to report results on the mesh produced by the solver
- **Import/Export**: Specialty tools can build a mesh from legacy data such as NASTRAN, and convert legacy data between different representations. All nodes and elements must be uniquely identified. Data might be added to the mesh. The import/export operation exercises all possible permutations of data access. Speed, memory usage, and memory fragmentation are all very important.
- **SDK**: An openly published interface to the mesh object for third-party developers. Usability and a well-developed abstraction of the data are more important than speed or memory usage.

From this sampling of uses of the mesh, there are several clear general requirements for the ACMO. Mesh data must be accessed by a unique identifier, but also by a direct index to the data. For example, the first node in a mesh can be access by its index 0, or by a given identifier that is unique across the entire assembly. Also, some users of the mesh may access the node based on the part it exists on while another user may access it based on its lo-

cation in the entire assembly. A node at index 0 on a certain part is not necessarily at index 0 on the assembly.

Also, different users of the mesh can use the mesh simultaneously. Therefore, the mesh object must be thread-safe since these users could exist on different threads. But even more challenging, one user could actually change the mesh while a different user is still using it. For example, the mesher could remesh (and therefore change) a part that had already been solved. The original solved part mesh can still exist in graphics and be available for perusal by the end user, and the newly meshed part must also be available. A single part would now exist twice in the assembly, and this condition would need to be automatically managed by the mesh object.

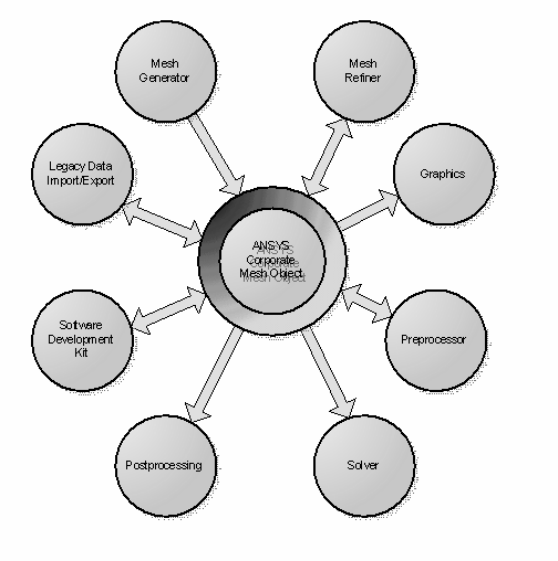

Fig. 2. Mesh Uses

2.2 Physical Design of the Mesh

With these cases in mind, the mesh object is designed to closely mirror the Part Manager in ANSYS Workbench, which manages geometry. An assembly mesh contains a collection of part meshes, each containing a collection of body meshes. Node and element identifiers are stored at the assembly, mapped to the physical location of those nodes and elements in the mesh. Nodal coordinates and topological information are stored on the part mesh because a node can be shared between multiple bodies. Elements are stored on the body mesh. To facilitate extensibility, an attribute mechanism that can attach an arbitrary data field to a node, an element, or an entire mesh is planned, so that any data that must be added to the mesh in the future could be done so easily.

A thin "configuration" layer between the assembly mesh and its collection of part meshes allows the assembly to change the collection of part meshes. By changing the as-

sembly's configuration, different views of the same assembly are possible, without having to make an entire copy of the assembly.

2.3 Interface Design

Since the mesh will need to communicate across processes, a COM interface is necessary. The COM interface is a thin layer of code between the user and the core data object. However, the interface layer is completely separate from the actual core object and is not required for the actual mesh object to function. The internal core objects have no awareness of the COM interface layer; they are free to communicate with one another directly. In this way, the core objects can be used independent of the COM interface layer. This type of functionality facilitates unit-testing the mesh object, since it is trivial to write a driver program to create and operate the core objects without the general hassle of using COM.

Code for thread-safing the mesh object is placed in the COM object layer, which makes the actual implementation of the core objects more understandable and readable. Also, the COM object layer can be used to hide the internal storage of the data from the user to better abstract the data. For instance, there is no actual node object inside the mesh object. Rather, the concept of a node is made up from several data fields in the mesh object, such as the nodal coordinates stored in one array, the nodal topologies stored in another, and the elements, which use the node. A lightweight node object can be created in the COM interface layer which encapsulates all this data, allowing a very user-friendly view of the data to a third party. However, since the various bits of data that makes up the node must be collected and stored together, the advantage of user-friendliness comes at the expense of speed and memory usage.

2.4 Design Goals

The mesh object typically contains more data than any other object in a finite element analysis. With this in mind, the ACMO was written primarily to reduce memory fragmentation. Memory fragmentation can become a serious problem, which occurs when small chunks of memory are requested and then returned to the system. Over time, the largest free blocks of memory are repeatedly decimated in order to fulfill the user's request for smaller blocks, but these smaller blocks may not necessarily be rebuilt into larger blocks as they are returned to the system. Memory fragmentation can be minimized by preventing small memory allocations whenever possible using techniques such as pooling, where large amounts of small allocations are made as one large allocation. Pooling is used extensively in the ACMO.

Along with memory fragmentation, memory usage is a main concern when dealing with huge amounts of data. For example, on a 32-bit Windows PC, a single process is limited to 2-3 GB of memory, regardless of how much memory is actually available. At 10 million nodes, over 8% of usable memory may be utilized just to store the nodal coordinates, which is just a small fraction of all the data which is stored in the mesh, not to mention the memory requirements of other objects residing in the same memory space. To minimize memory usage, "lazy evaluations" are used throughout the ACMO. Lazily evaluated data is generated at the time of access, such as a node-to-element cross-reference, so it is not loaded into memory by default. Rather, the first time such data is requested by the user, it is calculated and stored into memory in its entirety. If the user never requests the data, it is never loaded into memory.

Finally, in some situations, the speed of accessing data is extremely important, even more so than memory issues. Any access to the mesh object during meshing would take only a minute fraction of the overall time of the meshing operation, and so the speed of the data access is not too important. On the other hand, when dealing with graphics, speed becomes a very real issue, whereby the access to the mesh data can be the limiting factor to the overall speed of drawing to the screen. Therefore, the data in the mesh is always organized in such a way to maximize speed when accessing it from the graphics system, even if it adversely affects speed when accessing the data from one of the users.

3. IMPLEMENTATION

3.1 Programming Conventions

As mentioned earlier, the mesh is comprised of a core group of objects and a COM interface layer. For instance, the top-level mesh object is known as an AnsAssemblyMesh, and a COM object known as an AssemblyMesh wraps it. The "Ans-" prefix identifies the object as an internal core object. Since the AssemblyMesh object does not have an "Ans-" prefix, it is understood to be the COM interface to an object named AnsAssemblyMesh.

3.2 Object Overview

The mesh is comprised internally of three main objects. The first, the AnsAssemblyMesh, is a top-level object that contains all other objects. The middle-level object is the AnsPartMesh, and the lowest-level object is the AnsBodyMesh. An AnsAssemblyMesh contains a collection of AnsPartMeshes, which in turn contains a collection of AnsBodyMeshes. As mentioned earlier, each of these objects mirrors similar concepts in the ANSYS Part Manager, in which a geometric assembly is comprised of a collection of parts, and a part is made up of a collection of bodies. (Fig. 3)

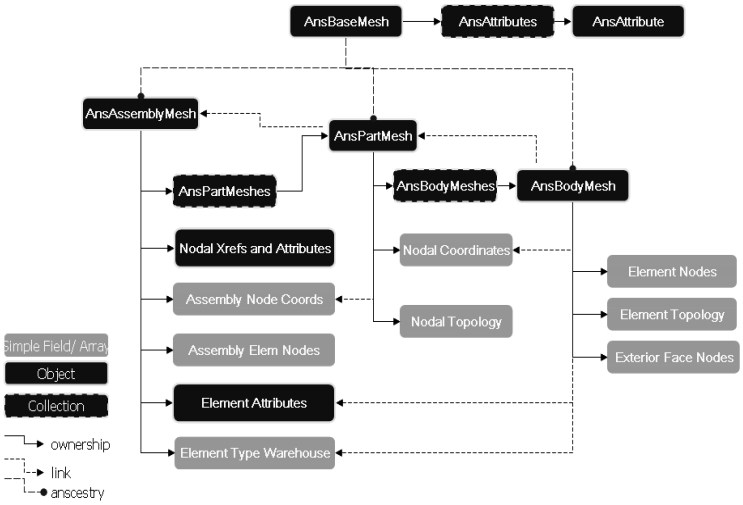

Fig. 3. Internal Object Model

Each of these core objects has a COM object that "wraps" it and acts as an interface. For each AnsAssemblyMesh, there exists a corresponding AssemblyMesh object. The COM object contains a reference to the core object. A user calls into the mesh object through these COM objects, which delegates the call into the core object. However, the core objects know nothing about the COM objects, and in fact can function without them. (Fig. 4)

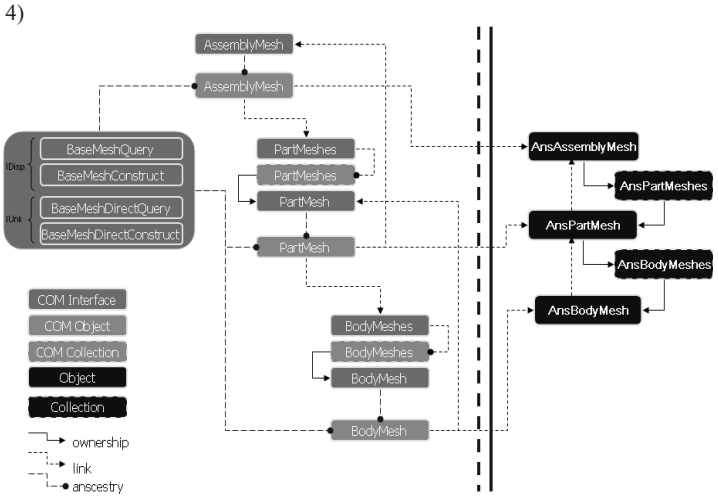

Fig. 4. COM Interface Model

The three mesh objects all derive from a single base mesh object, the AnsBaseMesh, which describes their common functionality. For example, a node can be accessed from a

particular part, or from a body, or from the entire assembly, and so the AnsBaseMesh defines the ability to query for a node. Likewise, the three mesh interfaces all derive from a common interface, the BaseMesh. Since the functionality to query a node exists in the AnsBaseMesh, the corresponding interface to that functionality exists in the BaseMesh.

3.3 Identifiers

All data in the ACMO is given an identifier, which is either provided by the user or simply begins at 1 and accumulates as more data is added to the mesh. The identifiers are unique to a specific data type such that there can be a node identified as 1, and an element identified as 1, but never more than one node in the entire mesh identified as 1.

The identifiers are stored in the AnsAssemblyMesh in a grouped hash table, which maps the identifier to the data's physical location in the assembly and in its "group," usually the part in which the data exists. Entries in the grouped hash table are made in the form group g: [i, j, k], where i is the identifier, g is the index of the group in the table, j is the index of the identifier across all groups, and k is the index of the identifier in group g. For example, a typical entry in the hash table might look like "group 2: [1, 7, 5]," meaning identifier 1 is found at index 7 in the entire assembly, and at index 5 in its particular group. If this was a node, then the node identified as 1 is the eighth node in the assembly (located at index 7), and it is also the sixth node in the third part on the assembly. Using this design, if the groups of the hash table correspond to the parts in the assembly, the part meshes can be moved in the assembly mesh, and the hash table is easily manipulated to reflect the change.

When data is actually stored in the mesh object, a reference to the actual location of the data is used rather than the identifiers. For example, if an element were made up of nodes identified as 1, 2, and 3, which are located at indices 0, 1, and 2, then the element's nodal connectivity would be stored with 0, 1, and 2. This allows quick access to the nodal data without performing a lookup in the hash table.

3.4 Data Organization

Data stored in the mesh must be well organized in order to be accessed quickly with minimal memory overhead. Nodal coordinates and nodal topological information is stored in the part mesh, but it is ordered according to the node's classification and bodies. A node's classification is based on the elements that use the node. If a node only ever lies at the vertex of an element, it is considered a "corner" node and is placed at the beginning of the node list. Nodes that lie only on the edges of an element are considered "midside" nodes, and follow the corner nodes in the node list. Next stored in the node list are "multipurpose" nodes that lie on both vertices and edges of elements. Finally, "zombie" nodes, which are not used by any elements, are stored last. In this way, a user only interested in corner nodes can easily access the desired type without having to search through the entire node list.

Within the four categories of nodes, the node list is further organized based on the bodies that use them. Referred to as "exclusive" nodes, any nodes used only by elements that exist on the first body of the part mesh are stored at the beginning of the list, and so on. Any nodes shared by elements existing on more than one body are called "interface" nodes and are stored at the end of the node list following all exclusive nodes. (Fig. 5) Using this method, it is easy to determine the body on which an exclusive node exists, though an interface node necessarily requires more work.

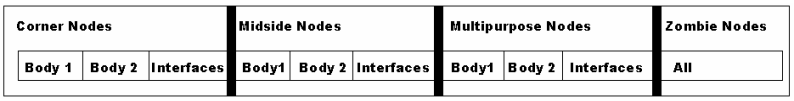

Fig. 5. Part-based Node Organization

Element connectivity and topological information is stored on the body. Elements of the same "type" are grouped together. An element type is based on the shape of an element, whether or not it has midnodes along its edges, and any arbitrary data assigned by the user. An element shape can be general or concrete. A concrete shape, such as a triangle or tetrahedron, has a known number of nodes making up the element and the number is always the same. In a general shaped element, the number of nodes is unknown; each element of that particular type is unique. By abstracting the concept of an element type into its own object, new types can easily be added to the mesh without additional programming.

Some nodes and elements transcend the boundaries of parts and bodies. For instance, a contact element may exist between two parts, and is used to identify the contact of the two parts to the solver. Rather than being generated during a normal mesh generation, a contact element is generally added to the mesh as a preprocess to the solver. Since the contact element does not exist on any one body, it is considered "independent," and is stored in assembly mesh, though it might reference non-independent nodes.

3.5 Memory Pools

Whenever possible, the mesh utilizes memory pools to combine many small allocations into one large allocation. As just stated, since the number of nodes in a concrete shaped element is known, the exact amount of memory required to store all element connectivity can be easily determined. For instance, when storing 1,000 tetrahedron, each with 20 nodes (which are referenced using one long each), allocating one long array of length 20,000 can prevent fragmentation of available memory that may occur by allocating 20 longs 1,000 times.

3.6 Decoupled Data Versus Committed Data

Organizing mesh data as described above can be an expensive operation, and in some cases, the organization is impossible until all data has been added to the mesh. Also, the exact length of memory pools cannot be determined until all data has already been put into the mesh. Therefore, data can exist in the mesh in two distinct states: decoupled or committed.

As data is added into the mesh, it is considered to be in a temporary "decoupled" state. When decoupled, the data is not organized in any way. Querying for decoupled data based on its unique identifier requires an expensive linear search through the mesh. However, the data is only stored in this state until the user is ready to "commit," or finalize all changes to the mesh. When the mesh object is committed, the data is sorted and moved to permanent memory pools. Also, the internal structure of the data is changed to afford the quickest possible access to the user. For instance, the unique identifiers of the nodes are stored to describe a decoupled element's nodal connectivity. However, in a committed element, the indices of those nodes are stored instead. Small changes to committed mesh data happen

immediately, such as changing the value of a node's coordinates, but large changes, such as adding data or removing data, is deferred until the user commits the mesh, allowing the user to determine exactly when the speed hit caused by organizing the data occurs.

Storing data quickly and efficiently is the main goal of the decoupled state, rather than allowing quick access. Since the exact amount of data being added to the mesh may not be known, granularized memory pools can be utilized to store the data. The first time memory is requested to store data, enough memory is allocated to store many instances of the data, instead of just one instance. For example, the first time a user adds a node to the mesh, instead of allocating enough memory to store one node, enough memory to store one thousand nodes may be allocated. Once all of this memory has been used, enough memory for two thousand more nodes may be allocated. The exact granularities used are left to the user, allowing the memory allocation to be fine-tuned, and if the exact amount of data being added to the mesh is known in advance, then that amount can be used as the granularity, preventing any unnecessary memory reallocations.

Since data in a decoupled state is not tied to the assembly mesh (it does not use the assembly level hash tables to map identifiers to locations in the mesh), the data can be moved between assemblies. An entire part can be decoupled, basically a reverse-commit. The part can then be removed safely and easily from one assembly and added to another. Once in the other assembly, the data can be recommitted and reorganized as necessary, and the second assembly's hash tables would be updated with the new data's unique identifiers.

3.7 Filtered Data Access

The mesh object affords many different types of access to the same data through the same interfaces. To accomplish this, all method calls to the mesh object are filtered. A typical method declaration looks like this:

```
HRESULT GetNode( long    id,
                 ULONG   ulAppliedFilter,
                 ULONG   ulFilter0,
                 ULONG   ulFilter1,
                 _sNode *piNode );
```

where ULONG is an unsigned long, and _sNode is a structure containing nodal information such as coordinates.

The three ULONG filter values can affect both the input and output of the method. The ulAppliedFilter is made up by bitwise OR-ing the desired filters together. Each individual filter is simply an integer with exactly one bit set to one and all other bits set to zero. When two individual filters are bitwise OR-ed, the resultant integer has exactly two non-zero bits. Some filters, called unvalued filters, can work solely on their own and require no other input from the user. For example, the default behavior of the GetNode function treats the id input as the user-defined unique identifier of the node. By applying the DIRECT_ACCESS filter to the method call, the id input is instead treated as the index of the node at the level of the mesh from which it is queried. That is, if the GetNode call were made from assembly mesh filtered with DIRECT_ACCESS, then the id input would be treated as the index of the node on the assembly. If the same call were made from a part mesh, the id input would instead be the index of the node on that particular part. The same method call can behave differently based on the filters applied to it or based on the level of the mesh it is called from. However, if no filter were applied to the GetNode method, it would behave

exactly the same no matter what level of the mesh it was called from, since the id input would be treated as a unique identifier, which is the same throughout the mesh.

Some filters require further input from the user, such as BY_PART. This filter, when used in conjunction with the DIRECT_ACCESS filter, instructs the method to treat the input as an index on the given part. In this case, the id of the desired part would be provided in the ulFilter0 field. An example use might be:

```
HREULT hr = piMeshObject->GetNode( 5, DIRECT_ACCESS |
BY_PART, 3, 0, &iNode );
```

In this case, the user is querying for the node at index 5 on part 3. Since the user only used one valued filter (BY_PART), the ulFilter1 is unused. Any number of unvalued filters, and up to two valued filters, can be applied together to any particular method call.

Some examples of unvalued filters:

```
DIRECT_ACCESS       (1L<<0)
DIRECT_RETURN       (1L<<1)
NO_ALLOC            (1L<<2)
ASSEMBLY_ACCESS     (1L<<3)
ASSEMBLY_RETURN     (1L<<4)
PART_ACCESS         (1L<<5)
PART_RETURN         (1L<<6)
BODY_ACCESS         (1L<<7)
BODY_RETURN         (1L<<8)
FORCE_COPY          (1L<<9)
NO_CLEANUP          (1L<<9)
NO_MIDSIDES         (1L<<10)
NO_ERRORCHECK       (1L<<11)
EMPTY_RETURN        (1L<<12)
INDEPENDENT         (1L<<13)
DECOUPLED           (1L<<14)
DESCRIPTION_ACCESS  (1L<<15)
```

Some valued filters:

```
BY_PART             (1L<<24)
BY_BODY             (1L<<25)
BY_PID              (1L<<26)
BY_ATTRIBUTETYPE    (1L<<27)
BY_ATTRIBUTEVALUE   (1L<<28)
USE_OPTION          (1L<<29)
```

In these examples, the individual filters are created by shifting the non-zero bit by a unique amount. DIRECT_ACCESS does not shift the non-zero bit, so it remains as integer 1. PART_ACCESS shifts the non-zero bit 5 times to the left, and is binary 100000 or decimal 32. These two values can be bitwise OR-ed together (DIRECT_ACCESS | PART_ACCESS) to binary 100001, or decimal 33. Using this implementation, 32 unique filters can be created, and they can be used together in any combination.

Completely unfiltered methods to the mesh object also exist, mirroring each of the normal filtered methods. For example, the unfiltered version of GetNode looks like this:

```
HRESULT GetNodeEZ( long    id,
                   _sNode *piNode );
```

In this case, the input id is always treated as a unique identifier rather than an index, because that is the default behavior of the GetNode method. In this way, code that only uses the default behavior of the mesh is made much cleaner.

The concept of filtered methods works well, but can be improved upon. Most users of the mesh always use the exact same filters for all method calls. To accommodate, in the future, filtered methods will be removed in favor of filtered mesh objects, discussed below in Section 4.3.

3.8 Attribute Mechanism

Attributes are used to tag nodes, elements, or entire meshes with arbitrary data. They are also used to define element types. By utilizing an attribute mechanism, a user can easily add new data to the mesh without performing any additional programming. In the ACMO, an attribute is defined with an identifier, an optional description, and a variant data value. The description makes access easier to the end user (by asking for attribute "volume" rather than attribute id 4 for example) at the expense of some memory to store the string. In order to keep memory fragmentation in check, attributes attached to similar data are stored together in granularized pools, and share strings (rather than making multiple copies of the same string). For instance, all attributes connected to nodes on a particular part are stored together in one pool; all attributes connected to the actual part itself are stored in a separate pool.

To reduce database sizes, attributes can also be treated as transient data that is not saved. Rather, it needs to be lazily evaluated when required.

Some attributes without fail are tagged to every bit of data in a mesh. For instance, the user may apply a "volume" attribute to every element. In this case, a meta-attribute is used to further reduce memory usage. Rather than storing only one variant data value, a meta-attribute stores an array of such values, one for each possible instance of the attribute. The identifier of the attribute and a reference to its description only need to be stored one time for a meta-attribute, instead of multiple times for normal attributes.

3.9 Memory Management

Since the mesh can be accessed in many different ways through the same methods (using filters), the same method call can sometimes cause a memory allocation and other times not. Any time possible, the mesh returns references to its data, rather than copies of the data, though this behavior can also be overridden using the FORCE_COPY filter. For example, if a user calls GetNodes on a part without filters, the mesh can return a pointer directly to its nodal coordinates. If the same call is made using the NO_MIDSIDES filter to only return corner nodes, the mesh might allocate memory for the output. The mesh uses standard HRESULT return codes such as S_OK for success and E_FAIL for general failure, but also uses a custom code S_ALLOC to signify a success but with memory allocation that must be freed by the caller.

The S_ALLOC code (defined as 512) alerts the user that memory has been allocated, but it cannot tell the user exactly what memory must be freed. Some method calls can return several different outputs simultaneously, and not all of them may need to be freed. Also, some methods may return "nested" allocations, when an allocated array contains an allocated array, such as in the case of the GetElements method without filters. By default, the mesh always uses the global identifiers of nodes and elements, even though internally those nodes and elements may be stored by reference using indices. If a user calls GetElements, the mesh will return the nodal connectivity using nodal identifiers, which will require a memory allocation. In this case, the mesh must allocate an array to store each element, and a pool to store the nodal connectivity of each element.

To ease the burden on the user, a simple and very lightweight memory manager is in place to handle memory allocations. For each and every memory allocation made, a reference to the allocated memory is added to a stack. If a method allocates three arrays, three memory references are pushed onto the memory stack. After calling a method that allocated memory, the user simply needs to call ReturnFunctionMemory, and any memory al-

located by the last function call will be freed. In no memory was allocated, the call will be ignored. If a user calls another method that allocates memory, any references to memory left on the memory stack will be removed and pushed onto a queue for later handling. Users can also request to have memory removed from the memory manager, in which case the user becomes responsible for freeing the memory when it is no longer needed.

3.10 Multiple Configurations

Many times, multiple users of the mesh require simultaneous but conflicting access to the same data. For example, in the Workbench software, the end user can solve a mesh and view the results even while remeshing the original mesh. In this case, the same mesh must exist in multiple states. In the ACMO, configurations allow different "views" of the same assembly. The assembly mesh contains a configuration manager that can construct the various views and switch between them. Each configuration contains a list of part "instances", which are references to actual part meshes. In this way, the same part mesh can be shared between multiple configurations of the mesh. When the configuration needs to change, the configuration manager retrieves the necessary part meshes for the assembly, and then the hash tables that map global identifiers for the nodes and elements are rebuilt.

For instance, if a three-part assembly is meshed and the solved, the assembly mesh will contain one configuration with three part meshes. If the user then changes the mesh on the second part, the assembly mesh does not immediately remove the original part mesh that was remeshed, since the end user can still view the results of the original solve. At this point, the assembly will contain two configurations, but four part meshes. One configuration has an instance of part one $P1^1$, an instance of part two $P2^1$, and an instance of part three $P3^1$. The second configuration has instances of the same parts one and three as in the first configuration, but it has an instance of a completely different part two $P2^2$. As the assembly needs to change its view of the data, these parts are switched in and out of the assembly as required. For example, if the user asks to view the results of the solve, then the assembly switches to configuration one. If the user asks to view the current mesh, the assembly switches to configuration two. (Fig. 6) If the user resolves the mesh at this point, the original configuration is no longer required and is deleted. Since one part mesh no longer has any instances in the configuration manager, it too is deleted. The ACMO relies on accurate reference counting in the COM interface layer to determine when deletion can safely take place.

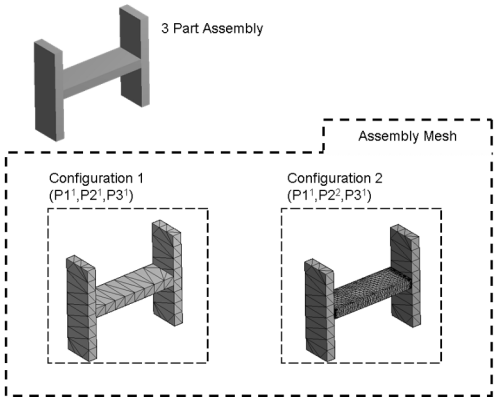

Fig. 6. Mesh Configurations

3.11 Thread-Safing

A final layer of complexity lies above the configuration manager: multiple threads accessing the mesh data simultaneously. A meshing process can be writing mesh data into one configuration of the object while a user is viewing another configuration. In the midst of the writing process, the user can change the configuration at will, and all the while the ACMO must be protected from access conflicts that can occur if the same memory is being read and written at the same time, not to mention that the physical structure of the mesh is changing while data access is taking place. To prevent such a catastrophe, the mesh object uses a simple technique utilizing a variation of semaphores called a critical section. A critical section prevents a thread from accessing an object when another thread has requested exclusive access to the same object. The mesh uses an extremely lightweight object that locks the critical section when the object is created, and releases the lock when the object is destroyed. In this way, it becomes a simple matter to create an instance of this critical section object at the beginning of every call into the mesh object in the COM layer. When the call finishes, the object is automatically released from memory and the critical section is unlocked. However, when using this method of thread-safing, only one operation can be performed on the mesh at any given time, so during a long operation such as committing mesh data, access to the mesh by other threads will be completely blocked until the operation finishes. The advantages of this method, such as ease of programming and inherent stability, offset the disadvantage of one thread needing to wait for another thread to completely finish accessing the data.

Since threads may be using different configurations in the ACMO, the mesh stores a map of thread ids to configuration ids, and every call into the mesh checks the calling thread immediately after creating the critical section. If the thread has changed since the last access to the mesh, then the current configuration is automatically changed. The calling system only needs to set the required mesh configuration once instead of setting it before every single call into the mesh object.

As an example, consider the situation where a user is refining a mesh that has already been solved. The refinement process starts on a second thread, and a new configuration is created in the ACMO that is tied to the refinement thread. While refinement runs, the primary GUI thread can also access the mesh object. If the user requests to view the results of

the solve, the GUI thread will try to call into the mesh object, which will first attempt to lock the critical section. If the refinement thread is currently accessing the mesh, the GUI thread is forced to wait until the method call on the refinement thread finishes. At that time, the GUI thread can lock the critical section, forcing the refinement thread to wait if it attempts to call into the ACMO. Next, the mesh object checks the thread id against the last stored thread id. Since the id of the GUI thread differs from the stored thread (because the last access to the mesh object was made by the refinement thread), the ACMO changes the configuration of the mesh automatically. The GUI's method call is then allowed to finish. When the method call completes, the critical section wrapper is destroyed, and the actual critical section is automatically unlocked. If the refinement thread then accesses the ACMO, the entire process is repeated. (Fig. 7)

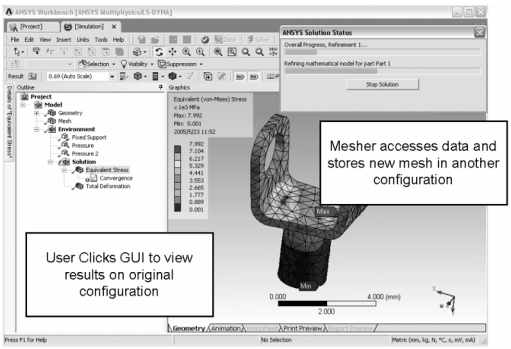

Fig. 7. Multiple Thread Access

4. FUTURE WORK AND ENHANCEMENTS

4.1 Part Spooling

As meshes grow ever larger in size, new ways of reducing memory usage must be pursued. The single largest memory gain may be found in the complete removal of parts from memory. A part that is not instanced by the current configuration of the assembly mesh can be spooled off to disk and freed from memory. The part can remain on disk until such time as it is needed. The configuration manager would then read the part from the disk back into memory. Though the time to switch between configurations would grow, the memory savings could be substantial. Along the same line, the implementation of memory map storage could also benefit the ACMO. A memory map acts like normal memory, except that it is written to disk instead of actual RAM. Access to the data is slowed considerably, but the fact that the disk is being used instead of physical RAM is transparent to the user.

4.2 Part Ghosting

Oftentimes, the meshes of several parts are exact copies of one another, but with a transform (a translation or rotation) applied. By storing the data only once and applying the

transform when the data is accessed, memory usage can be greatly reduced at the expense of speed.

4.3 Filtered Interfaces to the Mesh

As mentioned briefly, many users always require the same type of access to the mesh object. The user may always use part-based indices when accessing nodes for instance. Instead of being burdened with always remembering to use the right combination of filters and always calling methods from the right level of the mesh (be it assembly, part, or body), a user can create a new interface to the mesh data which always acts the same way. As preliminary work, a FilteredBodyMesh interface to the mesh has been created. A user can create a FilteredBodyMesh from a normal BodyMesh interface to the mesh. The user can then apply various filters to the interface, which will always be respected when accessing the mesh data. This approach leads to much more understandable and readable code than the filtered methods currently implemented in the mesh, as well as allowing much more complex types of filtering.

4.4 Iterators

Iterator access may be the best solution to the memory management problem in the mesh, by removing memory allocations all together. When the user first accesses data in the mesh, instead of retrieving the actual data, the mesh would return an iterator, which points to the first instance of the data. Once the user queries the data being pointed to by the iterator, the iterator points to the next instance of the data. For example, if the user queries the mesh for the nodes lying along an edge, instead of building an array containing that information, the mesh would return an iterator pointing at the first node on the edge. Once the user has finished querying that node, the iterator moves on and points to the next node along the edge.

As a compromise between iterator access and array access to the mesh data, a bucket iterator can be used to retrieve the data in chunks, the size of which is controlled by the user. As in the previous example, if the user accesses nodes along an edge using an iterator with a bucket sized for three nodes, then the first three nodes along the edge would be retrieved and stored in the iterator. Once the user has queried those three nodes, then the next three nodes on the edge would be stored in the iterator.

5. CONCLUSION

As meshes become larger and the systems surrounding mesh data structures become more diverse and complicated, the ability to efficiently store and easily access mesh data becomes more and more vital. The ACMO strikes a reasonable balance between several conflicting priorities, such as speed of data access, memory usage, and memory fragmentation, by identifying the many users of the data and how those users interplay with one another. Further, the ACMO allows users to query for data in many ways, but through the exact same interface. Also, the ACMO is designed to be easily expanded without additional programming.

REFERENCES

1. Beall, Mark W, Shephard, Mark S "A General Topology-Based Mesh Data Structure", International Journal for Numerical Methods in Engineering, John Wiley & Sons, Ltd., Vol 40, Num 9, pp.1573-1596, May 1997

2. Garimella, Rao V "Mesh data structure selection for mesh generation and FEA applications", International Journal for Numerical Methods in Engineering, John Wiley & Sons, Ltd., Vol 55, Num 4, pp.451 - 478, October 2002

3. George, Paul-Louis and Houman Borouchaki "Delaunay Triangulation and Meshing: Application to Finite Elements", Hermes, pp.311-315, 1998

4. Hitchsfeld, N "Algorithms and Data Structures for Handling a Fully Flexible Refinement Approach in Mesh Generation", Proceedings, 4th International Meshing Roundtable, Sandia National Laboratories, pp.265-276, October 1995

5. Jia, Xiangmin, Herbert Edelsbrunner and Michael T Heath "Mesh Association: Formulation and Algorithms", Proceedings, 8th International Meshing Roundtable, South Lake Tahoe, CA, U.S.A., pp.75-82, October 1999

6. Karamete, B K, T Tokdemir and M Ger "Unstructured Grid Generation and A Simple Triangulation Algorithm For Arbitrary 2-D Geometries Using Object Oriented Programming", International Journal for Numerical Methods in Engineering, Wiley, Vol 40, pp.251-268, 1977

7. Kwok, W, K Haghighi and E Kang "An Efficient Data Structure for the Advancing Front Triangular Mesh Generation Technique", Communications in Numerical Methods in Engineering, Vol 11, pp.465-473, 1995

8. Mobley, A, J Tristano, and C Hawkins "An Object Oriented Design for Mesh Generation and Operation Algorithms", Proceedings, 10th International Meshing Roundtable, Newport Beach, CA, U.S.A., pp.179-183, October 2001

9. Noel, F J C Leon and P Trompette "A Data Structure Dedicated to an Integrated Free-form Surface Environment", Computers and Structures, Pergammon, Vol 57, Num 2, pp.345-355, 1995

10. Remacle, J F , B Karamete, M Shepard "Algorithm Oriented Mesh Database", Proceedings, 9th International Meshing Roundtable, Sandia National Laboratories, pp.349-359, October 2000