THE COMMON GEOMETRY MODULE (CGM): A GENERIC, EXTENSIBLE GEOMETRY INTERFACE
Timothy J. Tautges¹
Sandia National Laboratories, Albuquerque, NM, tjtautg@sandia.gov

ABSTRACT

Geometry modeling has in the past five years emerged as a commodity capability; several geometry modeling engines are available which provide largely the same capability, and most high-end CAD systems provide access to their geometry through APIs. However, subtle differences still exist between these modelers, both at the syntax level and in the underlying topological models. A modeler-independent interface to geometry is critical to solving many of the geometry-based problems which exist in applications like mesh generation. The Common Geometry Module, or CGM, provides such an interface to geometry.

CGM consists of a set of wrapper functions which translate function calls such that they access geometry in its native format. However, CGM also provides functionality not found in most modelers, like support for non-manifold topology, and alternative geometric representations, including mesh-based and facet-based geometry. CGM is designed to be extensible, so that applications can define application-specific behavior for geometry entities. CGM is also designed to be easily portable to other geometry modeling engines. Ports to SolidWorks and Pro/Engineer are underway.

Keywords: geometry; solid modeling; componentization.

1. INTRODUCTION

Geometry is a critical part of mesh generation and other discretization-based simulation techniques. It forms the initial description of the analysis domain, is used in subsequent refinement and other modeling operations, and often is at the heart of bottlenecks in mesh generation and other processes. Geometric modeling capability has advanced rapidly in the past ten years, to the point where it can be considered a commodity [1]. Geometric modeling has been provided in library form for some time, and many third party applications have been developed on top of this capability; mesh generation is just one of these applications. For various reasons, attempts have been made to port these third party applications to multiple geometric modeling engines. In the process of developing such an application, methods for encapsulating geometric functionality and simplifying the porting process have been developed, and have been packaged in a set of higher level code libraries. These libraries, grouped in the Common Geometry Module, or CGM, are described in this paper.

Solid modeling libraries and Application Programming Interfaces (APIs) have over the last five years become a commodity, i.e. multiple sources exist for this capability, all providing functionality similar enough to meet most needs. Examples of such “solid modeling engines” are ACIS [2], the SolidWorks API [3], IDEAS Master Series Open I-DEAS [4], Parasolids [5], and Pro/Engineer’s Pro/Toolkit [6]. All such engines provide both representation of geometry (both topological and geometric) as well as API functions for constructing and modifying that geometry. Some engines expose the software design of their data structures, allowing the derivation of application-specific data structures based on them (e.g. ACIS), while others just expose functional interfaces to these data structures (e.g. Pro/Toolkit, SolidWorks). While there are subtle differences between the core data structures and functional interfaces in each solid modeling engine, in general the overall capability is quite similar, and fundamental capabilities (e.g. topological traversal, surface evaluation functions) are always provided.

Recent experience has demonstrated that geometry lies at the heart of many difficulties in the mesh generation process. First, the mesh generation tool of choice may not be capable of working directly with the solid modeling engine in which the part to be analyzed has been designed. In these cases, translation between geometry formats is necessary. This translation often introduces various geometric artifacts because of differences in modeling processes.
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
accuracy [7]. After those artifacts are removed, and all geometric features represent design intent, these features are often at a resolution different than that targeted for analysis. This results in modifications to the geometry, usually simplifying the model. Finally, various mesh generation approaches, especially in hex mesh generation, require further decomposition of the solid model.

Since geometry is so important to applications like mesh generation, and because the fundamental problems like detail suppression and decomposition are difficult enough to deal with on their own, it does not make sense to introduce further complications by requiring translation of the geometry from its native format to that required by an application. This is especially true when there are multiple applications, each requiring a different representation type, and even more unnecessary considering that most CAD systems now provide APIs for accessing geometry directly.

Rather, the applications should be written such that they can access geometry in any format, as long as the interface to those data provides the proper functionality needed by the application.

This paper describes a geometry library called the Common Geometry Module, or CGM. CGM provides a modeler-independent means of accessing geometry, while leaving the geometry in its native format. In addition, CGM provides capability not found in many geometry engines, but useful in many applications; these capabilities include support for non-manifold topology, virtual geometry, and support for simultaneous access to geometry in multiple geometry engines. CGM was constructed by isolating and modularizing the geometry capability in the CUBIT mesh generation toolkit [8].

This paper is arranged as follows. Section 2 explores further why generic interfaces to geometry are needed, and outlines the requirements of such an interface, from both the applications' and underlying representations' points of view. Section 3 describes the basic design of CGM, including the topological model used by CGM and how that model is created, modified and accessed through the CGM API. Section 4 gives more detail on the extensibility features of CGM, and describes the implementation of CUBIT using CGM. Section 5 includes a discussion of outstanding issues and future work planned for CGM. Section 6 gives conclusions of this paper.

2. BACKGROUND AND REQUIREMENTS

Before describing CGM, it is instructive to explore the general issues of why a generic interface to geometry is needed; this is done in the following subsection. Afterwards, some general requirements are given for a generic interface to geometry such that these needs are satisfied. This section concludes with a discussion of previous efforts in this area, and why these efforts do not meet the requirements laid out here.

---

**Generic Interfaces Needed**

Until recently, it was very difficult if not impossible to build applications in a way which was portable to multiple geometry engines. Either the target engine did not provide access to its geometry, or the geometry system providing this type of access was not robust enough to support mainstream design activities. These barriers have been removed with the development of the ACIS [2] and Parasolids [5] modelers, among others. However, small barriers to portability persist.

One major area of difficulty is in non-manifold topology, for two reasons. First, most modelers provide either no capability in this area or capabilities which are closely tied to how the modeler handles assembly models. Second, users are responsible for identifying "mating" surfaces which are common to adjacent parts. This is a tedious task, and furthermore is a geometric operation that should be supported directly by the modeler.

Another area of difficulty is the slight differences between topological models used in each modeler. For example, Pro/Toolkit sometimes provides topological faces having two or more disjoint regions. Another common difference is whether support for periodic geometry is provided. While relatively small, these differences can affect the way applications are written. For example, parametric space meshing algorithms must account for the jump discontinuity along a periodic boundary. Restricting an application to working only on a specific type of geometry, e.g. non-periodic geometry, inhibits portability by requiring geometry to be modified before access by these applications.

These differences can be overcome by defining a common, generic interface to geometry, along with a standard topological model. Standardizing the interface allows applications to work on multiple geometry engines; defining a standard topological model is necessary for determining what kind of assumptions applications can make about model topology. Implementing some changes to geometry at the interface layer allows the interface to "smooth over" the slight differences between the various modeling systems. We refer to this as a "tall" interface, that is an interface which provides not only translation between the common and engine-specific syntax, but also any modifications required for the engine-specific topological model to correspond with the chosen "standard". Ideally, these modifications are minimal.

Solutions to some geometry problems have resulted in the development of alternative representations of geometry. One of the first solutions of this type was the representation of non-manifold topology, used for the generation of mesh models with contiguous mesh through material interfaces. The CUBIT mesh generation toolkit [8] has long represented this in code outside the underlying ACIS representation. Another example is the removal of small features in the model (either translation artifacts or fine design details). This is done in CUBIT using "virtual geometry", where the topology of the model, as exposed to
the meshing tool, is modified while keeping the original solid model entities in their original state [9]. Other alternative representations which have been developed include mesh-defined geometry [10] and faceted geometry [11].

Using “tall” interfaces introduces the possibility of building these alternative representations, while still delivering them in a form which looks like more traditional representations. This allows applications to use these alternative representations without modification. If designed carefully, this type of interface can also simplify the porting of the interface itself to different geometry engines. This is described further in a following section.

**Interface Requirements**

Before describing other interface efforts, or the CGM design, it is useful to outline the basic requirements of the geometric module.

1. **Linkable:** Obviously, a geometry interface must be linkable into applications needing geometry functionality. This allows the geometry functionality to be separated from applications that use it.

2. **Independence from non-geometric data, code:** Because applications use widely varying data structures, even within applications of the same class (e.g. mesh generation codes), this data must be kept separate from the geometry interface. This not only reduces the size of the interface (both data size and the number of functions), but, more importantly, allows development of applications and geometry representations independently. This greatly simplifies the development and maintenance overall, while also facilitating the testing of each capability in production application environments.

3. **Encapsulation** For the geometry module as a whole, encapsulation ensures that the geometry interface is not dependent on other code, e.g. meshing algorithms. Within the implementation of the interface, encapsulation means that code specific to a particular geometric modeling engine, e.g. ACIS, is separate from the more general geometric modeling capability implemented in the interface, e.g. detection and representation of non-manifold topology. This simplifies the implementation of the interface on additional solid modeling engines.

4. **Extensibility** Geometry forms the basis of most modern mesh generation toolkits[12]. In fact, geometry data structures are typically used as the objects around which meshing capabilities are implemented (this is the approach used by CUBIT, for example). As geometry propagates further into the analysis process, this will become the case in analysis codes as well. For this to be possible at the same time as requirement #2, the geometric data structures must be extensible by applications built on the interface.

5. **Portability** One of the goals of this effort is to simplify the porting of geometry-based applications to other geometric modeling engines. We also require that this geometry interface be portable to other modeling engines below the interface. The porting of CGM to other geometric modeling engines is described later in this paper.

6. **Multiple, simultaneous geometric representations** Sometimes, the modeling capabilities of one representation can be used to overcome shortcomings of another. For example, virtual geometry can be used to remove small geometric artifacts introduced inadvertently by a geometry engine.

Simultaneous access to multiple representations is also desirable when applications need to reference parts of a model designed in different CAD systems. From the code design point of view, this requirement depends on the code within CGM being encapsulated (requirement #3), so that representations can be interchanged without extensive code modifications.

7. **Support for query, modification and creation:** Obviously, a geometry interface needs to support querying of geometry entities (e.g. closest point to surface, point in body, etc.) However, as described earlier, many of the current bottlenecks to mesh generation are really geometry problems; these problems are difficult or impossible to overcome without making modifications to the geometry. For example, most approaches to hexahedral mesh generation currently require geometry decomposition. Thus, if it is to meet needs in mesh generation, a geometry interface must allow modification to or creation of geometry.

8. **Support for non-manifold topology:** FEA capabilities have advanced to the point where it is possible (and common, in some contexts) to analyze model assemblies consisting of multiple parts. For structural problems, these assemblies have multiple materials or parts, which meet at shared surfaces. Modeling such assemblies requires the representation of non-manifold geometry.

Conspicuously missing from the list of requirements above is a requirement of minimality of the interface, or of allowed types of access to the geometric data structures. These requirements were left out not because they would not be desirable, but because they are not crucial to the basic function of such a geometry interface. However, it could be argued that these characteristics would either

---

2 This is not strictly true; a shared interface could be constructed by representing the interface twice, but requiring mesh on either side be duplicated on the other. It is clear that this process is much simpler if it is supported in geometry rather than as a mesh matching operation.
simplify the interface or make it more efficient. This is discussed more in Section 5.

Previous Work

There have been several past attempts to generalize the interface to geometry; these efforts are now reviewed.

The DJINN API attempts to define an interface common to most geometry engines, based on set theoretic definitions for accessing and modifying geometry [13]. Since there is no implementation provided with this API, it is difficult to evaluate with respect to requirements 1-3 and 5-6. Requirements 7-8 are met, according to the DJINN specification. However, since DJINN provides only a functional interface to geometry, it follows that the interface would probably not be extensible. As will be described later, extensibility is critical for building applications on top of geometric models.

Another attempt at defining a common interface to geometry is the CAPRI effort [14]. Here, interface functions are defined independently of an underlying solid modeling engine. Query-only operations are supported, but geometry cannot be modified or created through this interface. Also, it does not appear that CAPRI has support for non-manifold topology, at least in a manner that is independent from the underlying solid modeling engine. It is not stated which if any implementations exist for widely available solid modeling engines.

Another effort at specifying a geometry interface is the OLE for Design & Modeling standard, or OLE for D&M. This standard, promoted by DMAC, defines functions for accessing geometry and topology in a manner independent of any particular solid modeling engine [12]. However, the functionality is limited to query operations, and does not define functions for creating or modifying geometry. Since many of the tools used in mesh generation require the definition of additional geometry (e.g. decomposition), the OLE for D&M interface would be insufficient for supporting this application. It is unclear whether there are many full implementations of this specification.

The Virtual Geometry Interface, or VGI, was implemented in CUBIT to serve as a middle layer of geometry between applications above and specific solid modeling engine below [16]. The VGI is the predecessor to CGM, and the two share many of the same design concepts and capabilities. However, CGM has extended the design concepts of the VGI in key areas, simplifying and sometimes enabling extensibility, portability, and other requirements. Differences between CGM and the VGI are described later in this paper.

CGM is the result of isolating all geometric capabilities available in CUBIT into a standalone set of libraries. It acts as a wrapper around the functionality in ACIS, which serves as the primary geometry engine underneath CUBIT. However, CGM also includes key functionality not found in ACIS, such as non-manifold topology detection and representation, virtual geometry, and mesh-based and faceted geometry representations. Although CGM grew out of existing code, substantial efforts were made to improve its extensibility, and to simplify its porting to other geometry engines.

One obvious question is whether any existing solid modeling engines meets enough of the requirements above to serve as an adequate starting point. Of those available, ACIS comes the closest to providing this level of support. ACIS gives full access to its data structures, both for calling functions from specific data classes, and even for deriving application-specific data structures from them. There are two reasons we chose to not take this approach. First, using the ACIS data model would require us to either use ACIS code directly in CGM, even in versions not including the ACIS solid modeling engine as one of the geometry representations, or to rewrite ACIS classes for those versions of CGM. Either way, it would be cumbersome to maintain compatibility with ACIS, since CGM would depend on both functional interfaces and data structures in ACIS. Secondly, a concept fundamental to the design of CGM is the topology model used. The topological model in CGM differs from that in ACIS in key areas of non-manifold geometry. Extending ACIS data structures to represent non-manifold topology would be difficult and would exacerbate the difficulty of maintaining CGM in the face of changes to ACIS. Finally, using data structures derived from those in ACIS might make it difficult to implement fundamentally different kinds of geometry representations. For example, the implementation of virtual geometry in CGM depends on hiding geometric entities from the model visible to applications, but using these hidden entities directly for geometric evaluations. This requires specific changes to the topology data and traversal functions that would be very difficult to make in ACIS.

For these reasons, and also because this effort started long before ACIS was robust enough to support these needs, we decided to use our own data structure on top of those provided by ACIS or other solid modeling engines. This data structure, and the topological model reflected by it, are described in the next section.

3. CGM DESIGN

CGM can be thought of as a "tall" interface. That is, it provides both wrapping functions for the underlying solid modeler(s), and data and functions to provide missing capabilities and smooth over differences between various solid modelers.

In this section, CGM is described at the "user" level, that is, from the perspective of how it would be used, without modification, to access geometric models. This section concludes with a short example of how CGM is used to query and modify geometry.

TOPOLOGY AND GEOMETRY ENTITIES

The basic geometric representation used in CGM is that of a Boundary Representation, or BREP. CGM defines two
At the highest level, CGM defines its own TEs, which are the primary means of access to geometry and topology by CGM applications. The TEs are separated into three groups: "basic" topology entities, grouping entities, and sense entities. Basic topology entities (BTEs) are the only TEs which have an underlying geometric description. Grouping entities (GEs) group together basic topology entities to form the boundary of higher level BTEs; for example, a Loop groups together edges bounding a face. Sense information is represented explicitly in Sense Entities (SEs). The three groups of Topology Entities are listed in Table 1, for reference.

Table 1: Topology entities used in CGM topological model.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RefVertex</td>
<td>0d entity in topological model.</td>
</tr>
<tr>
<td>RefEdge</td>
<td>1d entity in topological model.</td>
</tr>
<tr>
<td>RefFace</td>
<td>2d entity in topological model.</td>
</tr>
<tr>
<td>RefVolume</td>
<td>3d entity in topological model.</td>
</tr>
<tr>
<td>Body</td>
<td>One or more volumes, share a transform and participate together in geometry</td>
</tr>
<tr>
<td></td>
<td>transforms and boolean.</td>
</tr>
<tr>
<td>Shell</td>
<td>Group of CoFaces describing a closed or open shell; if closed, describes a</td>
</tr>
<tr>
<td></td>
<td>boundary of a volume. Can be inner or outer boundary.</td>
</tr>
<tr>
<td>Loop</td>
<td>Group of CoEdges describing a boundary of a face. Can be inner or outer</td>
</tr>
<tr>
<td></td>
<td>boundary.</td>
</tr>
<tr>
<td>CoFace</td>
<td>Entity describing the orientation of a face as used in a shell, with respect</td>
</tr>
<tr>
<td></td>
<td>to the face's normal.</td>
</tr>
<tr>
<td>CoEdge</td>
<td>Entity describing the orientation of an edge as used in a loop, with respect</td>
</tr>
<tr>
<td></td>
<td>to the edge's tangent direction.</td>
</tr>
</tbody>
</table>

At the lowest level, Geometry Entity's are created to match one to one the topological entities in the underlying solid modeler. For example, there is a CGM Geometry Entity corresponding to each FACE, EDGE, etc. in an ACIS model. GEs also provide geometric evaluations through calls to the solid modeling engine. The types of Geometry Entity's used in CGM are defined in Table 2.

There are several specific arrangements of topological models that are allowed in CGM models that are not common to all modeling engines. First, CGM allows RefEdges to be bounded by a single RefVertex; these are called periodic curves, in reference to the periodic nature of the curve's parameter space. Likewise, periodic RefFaces are allowed; these are surfaces whose parameter space contains jump discontinuities, for example a cylindrical surface. Not all modelers are capable of modeling periodic entities, and will not generate entities of that type; however the capability of CGM to model periodic entities is a superset of these, and so can be used with either type of solid modeling engine.

In rare cases, topology exists where an entity of dimension d is not strictly bounded by an entity of dimension d-1. The best example of this is a cone, where the apex is represented topologically by a vertex. Although there is no apparent curve corresponding to this vertex, CGM models this using a zero-length RefEdge. Applications not desiring this behavior can request that the periodic entities in a Body be split into non-periodic ones.

Geometry can be envisioned whereby a single basic topological entity bounds a higher order entity twice; an example would be a torus with a split face on one end. While this geometry can be represented using judicious application of CoFaces, it is not typically found in design geometry. Cases involving surfaces included twice by a single shell are not supported in CGM; however, edges can occur twice in a given loop.

Non-Manifold Topology

Non-manifold topology is a general term describing allowable combinations and arrangements of the basic topological entities into a solid of arbitrary dimension. In the context of CGM, the term "non-manifold topology" is used to indicate the presence of entities of dimension d-1 shared by more than one entity of dimension d. In practice, we use "non-manifold" to describe groups of volumes which share surfaces between them. CGM implements non-manifold topology by associating multiple Geometry Entity's with single Topology Entity's. This is implemented using a TopologyBridge object, which links one TopologyEntity object to one or more GeometryEntity objects. Thus, the underlying solid model is typically manifold, whereas the CGM model can be manifold or non-manifold. Since the CGM topology can be different than the underlying solid model topology, CGM implements its own data and functions for representing and implementing topology traversal data and functions. When using a single underlying solid model representation containing no non-manifold topology, this data is identical to that in the underlying modeler. CGM constructs this model starting with an entity in the solid modeling engine by first constructing the corresponding geometry entities, in an engine-specific tool, then passing the highest-order geometry entity to a generic tool, which fills in the topology entity layer.
Table 2: CGM Geometry Entity's and their associated Topology Entity's.

<table>
<thead>
<tr>
<th>Geometry Entity</th>
<th>Associated Topology Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point</td>
<td>RefVertex</td>
</tr>
<tr>
<td>Curve</td>
<td>RefEdge</td>
</tr>
<tr>
<td>Surface</td>
<td>RefFace</td>
</tr>
<tr>
<td>Lump</td>
<td>RefVolume</td>
</tr>
<tr>
<td>BodySM</td>
<td>Body</td>
</tr>
<tr>
<td>ShellSM</td>
<td>Shell</td>
</tr>
<tr>
<td>LoopSM</td>
<td>Loop</td>
</tr>
<tr>
<td>CoFaceSM</td>
<td>CoFace</td>
</tr>
<tr>
<td>CoEdgeSM</td>
<td>CoEdge</td>
</tr>
</tbody>
</table>

The CGM class hierarchies for TopologyEntity and GeometryEntity are shown in Figure 1 and Figure 2.

**OTHER CGM DATA, TOOL OBJECTS**

In addition to the TEs and GEs used to define the basic entities making up the CGM geometric model, there are other data and tool objects defined by CGM that are useful for creating, modifying or otherwise interacting with geometric models. These are described below.

**RefEntitys**

RefEntity is a special class of TopologyEntitys that is useful for meshing and other applications. RefEntitys include all the BTEs, as well as Bodies and RefGroups (explained in the next section). This class complicates the implementation of CGM considerably, and may be removed in the future.

**RefGroups**

The Grouping Entitys listed in Table 1 are specific to the kinds of geometry they contain and bound; for example, Shells consist only of one or more CoFaces, and bound RefVolumes. In contrast, RefGroups are used in CGM to store one or more RefEntitys of arbitrary type. The entities stored in a given group are grouped only for convenience, either for the user or for a specific application. For example, users may want to create a group of all entities falling on one side of a coordinate plane. RefGroups are used extensively in the CUBIT application of CGM, as well as for some internal actions inside CGM itself.

**Attributes**

Attributes are defined as information which can be associated with a particular geometric entity, but which are not intrinsic to the representation of that entity. For example, the name of a geometric entity, while not required to represent an entity, is associated directly to that entity. CGM provides functions for storing application-specific attributes directly on geometry entities, and functions for managing that data when geometry is stored to disk.

**Tools**

The primary interface to CGM is through its tools, and in particular through the GeometryTool class. GeometryTool is implemented as a singleton, or static, class, which implies that the GeometryTool functions can be viewed as API functions rather than functions associated with a geometry entity. GeometryTool implements functions for reading and writing geometry files, creating geometry from primitives, and most other general geometric functionality. GeometryTool is also the point of access for global lists of geometric entities. Other tools in CGM, most of them also implemented as singleton classes, perform functions in specific areas. Two of the most important tools in CGM are MergeTool, which detects coincident geometry and changes manifold to non-manifold geometry, and VirtualGeometryEngine, which builds the representation for virtual geometry.

**Implementation Notes**

**RefEntity:*** This class is a parent of the basic topology entities as well as Body and RefGroup. As described earlier, the RefEntity class is a convenient mechanism for grouping entities of most common interest to users. Body and RefGroup are included in this hierarchy because users often perform operations on these entities which simply get applied to all the basic topology entities included in them, e.g. meshing, drawing and listing operations.

**CoVertex, Chain, CoVolume:** These classes are simply place-holders in the topology hierarchy. Though they do not perform any functions, they are retained for consistency. This guarantees that, for example, every Basic Topology Entity (e.g. RefEdge) will contain a lower level Grouping Entity (e.g. Chain), and each grouping entity (e.g. Chain) will consist of a list of sense entities (e.g. CoVertex).

**EXAMPLE: BASIC CGM USAGE**

In this section, a simple CGM driver application is described. This application, called mergechk, imports one or more geometry files, imprints the entities together, and merges geometry into non-manifold geometry. Although very simple, this application demonstrates how to import geometry, perform boolean operations on it, then further query the geometry. The source code for this application is shown in Table 3. Some declarations and comments in the driver code have been removed for brevity; the complete mergechk application is distributed with the CGM libraries.

There are three primary parts of this application: initialization, importing geometry, and modifying/querying geometry. In the initialization phase, certain objects (static CubitApp and GeometryTool objects) are created by requesting instances of the objects. Instantiating these objects also initializes internal data structures for global entity lists and other things. Reading in the geometry is accomplished by calling a function in the GeometryTool class. This function reads geometry from the specified file, storing pointers to that geometry in global lists stored in the GeometryTool class. In the final part of mergechk, bodies
are imprinted together, which ensures that neighboring geometry also shares like topology; and the entities satisfying the topological and geometric proximity tests are merged together.

Several important points about using CGM should be noted here:

- CGM keeps track of its internal state, e.g. by keeping geometric entities in global lists. However, individual entity pointers can be retrieved and accessed if desirable as well. This keeps the CGM interface both simple and powerful.
- CGM provides data structures which it uses to pass arguments back and forth, e.g. lists, coordinate triples, etc. These data structures can be used directly inside applications, or they can be copied into application-specific data structures.

Table 3: Mergechk, a simple CGM application.

```cpp
#include "GeometryTool.hpp"
#include "MergeTool.hpp"

int main (int argc, char **argv)
{

    // Initialize the application
    CubitApp *app = CubitApp::instance();

    // Initialize the GeometryTool
    GeometryTool *gti = GeometryTool::instance();

    // Read in the geometry from files
    // specified on the command line
    file_ptr = fopen(argv[1], "r");
    status = gti->import_solid_model(file_ptr, argv[1], "ACIS_SAT");

    // imprint the bodies together, discarding
    // old bodies
    DLBodyList old_bodies, new_bodies;
    gti->bodies(old_bodies);
    gti->imprint(old_bodies, new_bodies);

    // Merge bodies
    status = MergeTool::instance()->merge_all_bodies();
}
```

This example is meant to be illustrative of the basic capabilities of CGM. For a full description of CGM data classes and functions, see Ref. [17].

4. EXTENSIBILITY AND ENCAPSULATION FEATURES

One of the primary differences between CGM and other standard geometry APIs is that CGM is designed to be both extensible and encapsulated. CGM is extensible in that it allows derivation of application-specific entity classes from its topology and geometry entity classes; because of its encapsulation, this can be done without modification of the CGM code itself. CGM is also encapsulated on the lower end, from the solid modeling engine(s), to simplify the addition of alternative geometry representations. These features, along with how they facilitate specific functions in CGM or its applications, are described in this section.

CGM ENTITY CONSTRUCTION

In its simplest form, CGM constructs its TE and GE entities starting with a topological object represented in an underlying solid modeling engine. Constructing these CGM objects requires some operations that are specific to the solid modeling engine and some which are not. These operations are separated carefully, in order to simplify the porting of CGM to other solid modeling engines.

Starting with a solid modeling engine BODY (which has data and functions defining lower order entities and traversal between them), the procedure shown in Table 4 is used to build the corresponding CGM Body (in this description, the entity names corresponding to ACIS are used for illustrative purposes, but these could correspond to any modeler giving access to the BREP model). Several things should be noted from this procedure:

- **Linking TEs to GEIs to solid modeling engine objects:** There are three levels of bi-directional pointers; one is between solid modeling entities and the corresponding CGM GE objects (this is a one-to-one relationship, and is implemented using attributes); another is between a GE and its corresponding BridgeManager object (there can be multiple GEIs per BridgeManager, in cases of non-manifold topology); and finally, a link between TEs and their corresponding BridgeManager (this is a one-to-one relationship).
- **Solid modeling engine encapsulation:** The construction of TEs from the highest-level GE object (usually BodySM) is done in a solid modeling engine-independent manner. This requires traversal functions defined for the GE objects, which return other GE objects. For example, BodySM defines a function, surfaces(), which returns a list of Surface objects connected to the BodySM. These traversal functions are implemented in a solid modeling engine-specific manner, however, to avoid duplicating solid model topology at the GE level.
- **Missing entities in solid modeling engine topology:** In some cases, the solid modeling engine may not define an entity defined in CGM; for example, some solid modeling engines do not use CoFaces, since they do not support non-manifold topology. In this case, CGM defines its own representation for CoFaceSMs which provides the functions necessary to relate it to other OEs. This is a case where the "tall interface" approach can smooth over differences between solid modeling engines. Another example of this concept is the construction of multiple GE Surface objects for a single face in Pro/Engineer topology; this is necessary because Pro/Engineer can represent topological faces
having disjoint regions, a topological configuration which is not allowed in CGM.

- **Persistent objects:** Using attributes and the fact that CGM implements the construction of TEs in solid modeler-independent code, CGM provides a persistent objects capability which can easily be ported to multiple solid modeling engines. This allows persistence of some topological entities in the CGM model even after these entities are involved in geometry modification operations like booleans. For example, if a body is cut in half, the CGM objects corresponding to unmodified faces, edges and vertices in the solid modeling engine body are unchanged. This greatly facilitates the process of geometry decomposition in applications like CUBIT. This capability does depend however on the use of attributes in the underlying solid modeler and the persistence of solid modeler objects through these operations. This is described in more detail in [18].

Table 4: Procedure for creating CGM TopologyEntities from solid modeling engine entities; items with asterisks are engine-dependent.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Construct one GE object for each solid modeling engine topological entity in the new BODY (VERTEX, EDGE, COEDGE, etc.)*.</td>
</tr>
<tr>
<td>2.</td>
<td>Assign double pointers between GE and corresponding ACIS entity using an owner attribute*.</td>
</tr>
<tr>
<td>3.</td>
<td>Starting with a BodySM, construct one CGM TE object for each unique GE object that is part of the BodySM.</td>
</tr>
<tr>
<td>4.</td>
<td>Link TE objects in CGM topology graph in the same way the GE objects are linked together.</td>
</tr>
</tbody>
</table>

**APPLICATION-SPECIFIC ENTITY DERIVATION AND CREATION**

Topology entities in CGM have data and functions associated with geometry modeling, like topology functions, geometric extent data, etc. Embedding information about mesh data or meshing algorithms inside CGM would violate the encapsulation of CGM. In order to add meshing data and functionality to topology entities, application-specific entities must be derived from the CGM entities; these application-specific entities exhibit geometric modeling behavior and also have the capability to store mesh and define meshing algorithm data. For example, in CUBIT, this is accomplished by deriving CUBIT-specific classes from each of the child classes of RefEntity. These child classes are also given a common parent class in CUBIT, named MRefEntity, which centralizes the mesh data and functions common to all types of RefEntities. This class structure is shown in Appendix A, Figure 5.

The creation and deletion of geometric entities in CGM and other geometry modeling packages must be done carefully, to avoid situations where entities need to be referenced after they have already been deleted. Therefore, the construction and destruction of topology entities is initiated in a very specific manner from within CGM. This raises the issue of how application-specific entities can be created while still maintaining independence of CGM from its applications. That is, how does CGM initiate the construction of topological entities, when those entities (or at least part of them) are defined in application-specific classes. This problem is solved by routing the construction of Topology Entities through a common *entity factory*, which itself can be derived from in an application. This software construct follows closely the Abstract Factory design pattern from Ref. [16]. Functions defined in RefEntity are redefined in MRefEntity, such that they construct objects like MRefEdge and MRefFace and return them as RefEdge and RefFace, respectively. The application-specific factory is substituted for the default CGM RefEntityFactory by passing it to the initialization function for GeometryTool (see [17] for a complete description of this process). In this way, when CGM calls the factory function to construct a particular RefEntity object, it does not need to know whether that factory object is the default or an application-specific one; either way, the RefEntity gets constructed and passed back to the calling code in CGM. This is an example of the polymorphism principle in object-oriented programming languages.

**ATTRIBUTES**

CGM uses attributes like entity groups, merge information, and virtual geometry information to achieve persistence of geometric constructs across sessions. This is accomplished by storing this information directly with the solid model description, e.g. in the ACIS "*.sat" file.

Support for arbitrary application-specific attributes is included in CGM, and these attributes are passed to the solid modeling engine (and to CGM for application-specific attributes) in attribute-independent data structures. These data structures include only basic data types (ints, doubles, character strings). This minimizes the requirements placed on the attributes functionality in any particular solid modeling engine; specifically, solid modeling engines are only required to provide attributes that can store data in the form of ints, doubles and character strings. All major solid modeling engines (e.g. ACIS, SolidWorks API, Parasolid) provide attributes that meet these requirements. The design of the CGM attributes capability, and of the various types of attributes used in CUBIT and CGM, are described in a separate paper [18].

Similarly to CGM-defined attributes, applications often define data which is associated directly with geometric entities, and which would be convenient to store directly with the solid model entities. In CUBIT, information like mesh intervals, mesh schemes, and even mesh itself is associated directly with geometry objects. Similar to how application-specific RefEntity objects are managed, CGM utilizes an abstract factory pattern for the creation of attributes. A pure virtual base class, named CubitAttribFactory, is defined in CGM with two pure virtual functions, both named create_cubit_atrib; these
functions create an application-specific attribute from an attribute type and a simple attribute pointer, respectively. After writing the application-specific attributes, an application-specific attribute factory is derived from CubitAttribFactory:

```
Class AppCAFactory :
  public CubitAttribFactory
{
  ...
}
```

Before importing any geometry, the application creates an AppCAFactory and registers it with CGM:

```
ApplicationCAFactory *factory =
  new ApplicationCAFactory();
CubitAttrib::set_cubit_attrib_factory (factory);
```

When CGM imports geometry in which attributes have been embedded, it creates the attributes having definitions inside CGM; if any attributes are encountered which do not have definitions inside CGM, and an application-specific attribute factory has been registered, that application-specific factory is asked to create those attributes.

### OBSERVERS

The behavior of an application often depends on what happens to the geometry in the model. For example, CUBIT must know when a body gets deleted, so that it can also delete any mesh on that body. Since many operations which modify geometry are initiated within CGM or the underlying solid modeling engine (e.g. geometry booleans), that means CGM must have a means of notifying applications when geometry changes. Similar to attributes, this must be done in a way which maintains the encapsulation of CGM. This notification is accomplished using an Observer design pattern [16]. In this pattern, application-specific observers "register" themselves with the entities which they observe (the "observable"). When an observable is destructed or otherwise modified, any registered observers are notified of the operation.

Observers are used to implement entity groups, boundary conditions, and graphics rendering in CGM and CUBIT, among other things. Again, using a derivation methodology with associated virtual functions, these capabilities are provided in CGM without embedding any application-specific code or data types there.

### TOOLDATAS

Various tools often have a need for storing transient information associated directly with an entity. For example, a tool evaluating local and global constraints for finding geometry features may need to leave information about local constraints directly on topology entities. These data could be stored directly in the TopologyEntity class, however this would waste space when that tool was not operating, and would violate the encapsulation of TopologyEntity. To solve this problem, we use a data structure called a ToolData. Various types of ToolData child classes are defined by various tools, and are stored in a linked list off of each RefEntity object. Since a linked list is used, this increases the data size of each entity by only one word, no matter how many ToolData objects are associated with that entity (at the expense of increased access time because of the traversal of the linked list).

CUBIT uses ToolData objects to embed information on TopologyEntities pertaining to automatic scheme selection traversal [20], submapping parameter information [21], vertex and edge types [22], and many other types of information. As an aside, the ToolData construct is also quite useful for embedding tool-specific information on mesh objects, where storage space is at an even higher premium than for geometry.

### 5. IMPLEMENTATION, OUTSTANDING ISSUES AND FUTURE WORK

CGM grew out of a large body of code extant in CUBIT. Since the isolation of that code in CGM, CUBIT has been modified to use CGM without modification of CUBIT. This was done for many reasons. First, using CGM to support geometry needs in CUBIT guarantees support and further development of CGM. More importantly, CUBIT provides a means for testing both the encapsulation and the extensibility features of CGM. Thus, the extensibility features described in this paper are already working to support CUBIT in a production sense. Finally, as CUBIT geometry needs grow, these needs are met by extending CGM, which in turn enhances CGM itself.

The current version of CGM is built on the ACIS solid modeling engine; however, CGM also provides a faceted-based representation, as well as virtual geometry. Ports to SolidWorks and Pro/Engineer are underway, and other ports are being considered.

Since CGM grew out of a large body of code already extant in CUBIT, there remains a great deal of code in CGM which remains to be "cleaned up". In addition, there are several specific issues which are discussed below and which should be addressed:

- **Entity types**: A number of areas of CGM, and of CUBIT, rely on retrieving an entity type for a given object. Since Run Time Type Identification was only recently supported in most C++ compilers, CUBIT and CGM embed type information explicitly in the code. CUBIT type identifiers are still stored in CGM, in enumerated variables and type name strings. To be completely separable from CUBIT, the CUBIT types should be removed from CGM. This will probably be accomplished using the C++ language support for RTTI, rather than embedding type information in the CUBIT code.

- **Memory space**: When linked into an application, the CGM libraries add approximately 50MB to the size of a static executable; the ACIS libraries account for about 90% of this space. Obviously, this is a concern, especially when using CGM on parallel processing machines or in other memory-limited environments. There are several potential solutions to this problem. First, CGM could be linked with only a facet-based or
This work is described here. In addition to the porting work addressed above, there is also ongoing work on applications using CGM. Some of this work is described here.

- **Applications**: Several applications besides CUBIT which use CGM are being developed. The PMESH code is being re-written based on geometry capabilities provided by CGM. PMESH is a set of tools used to do user-assisted block-structured meshing in a massively parallel computing environment [23]. CGM is also being used to introduce CAD geometry capability to the MCNPX monte carlo transport code [24]. This will greatly simplify the construction of particle transport models. This code is used to model neutral and charged particle transport in nuclear and medical physics applications.

- **Parallel mesh generation**: Many attempts have been made to parallelize the mesh generation process [25][26][27]. In these efforts, the spatial domain is usually decomposed in some way, then meshed, either from the inside out or outside in. Spatial subdomains are in most cases treated as agglomerations of elements or other representations. In our work, spatial subdomains are modeled in the same way as geometric volumes, differing only in the representation method underlying the CGM interface. In the case of parallel meshing, subdomain geometry will be built using a combination of facet-based and geometric boundaries, depending on whether an interface is artificial (i.e. an inter-processor boundary only) or in fact a real geometric boundary in the domain. In essence, CGM is serving as the basis around which a parallel meshing capability is being built. This process has been demonstrated in an earlier effort, before the introduction of the VGI [28].

- **Other representations**: In large-deformation transient solid dynamics, mesh often distorts to a point where it is unsuitable for further analysis. At this point, the domain must be re-meshed with elements of sufficient quality for further analysis. However, the original domain, obtained from a CAD system, is no longer the domain being analyzed; rather the analysis must continue from a deformed geometry. The new mesh must be constructed starting from the deformed mesh, for which there is no CAD description.

The two options available for modeling these geometries is to reconstruct a CAD description from the deformed elements, or to simply use those deformed elements as the underlying geometric representation. The latter approach is being used. Because of the common interface provided by CGM, the mesh tools do not need to be modified in any way to mesh geometry based on this representation. However, care is being taken to provide geometry evaluations with the required level of continuity across the topology entities based on these representations.

One other representation already provided with CGM is that of "virtual geometry", i.e. composites or partitions made above the solid modeling engine level [9]. This capability provides a means for decomposing or combining topology entities without costly solid modeling operations. It also provides geometry through the same CGM interface, so meshing tools require no modification to operation on it.

## 6. Conclusions

Geometry plays a critical role in applications such as mesh generation. Despite the availability of APIs to geometric modeling engines, there remain barriers to porting a given application to multiple engines. Some of these barriers include lack of support for non-manifold topology, subtle differences in the topological models, and support or lack thereof for periodic geometry. This paper describes the Common Geometry Module, which is a "thick interface" to geometry, that it provides both wrapping functions which access geometry in its native format, as well as geometry modeling capability which makes slight changes to the underlying solid model (e.g. manifold to non-manifold representation), or more substantial changes (e.g. virtual geometry). CGM provides a means of accessing geometry through a common interface, even though the geometry may be stored in one or multiple representations.

CGM is designed for extensibility, both above and below the CGM implementation. Above CGM, applications can derive application-specific geometry entities; this capability is used by the CUBIT code to implement geometry entities which also define meshing algorithms and which can store mesh and meshing algorithm data. Below CGM, different geometric representations can be defined, which can be accessed through the same geometry interface. This simplifies the development of alternative representations, e.g. mesh-based or facet-based geometry.

CGM has been demonstrated using an ACIS-based implementation; virtual geometry representation is also implemented. The extensibility features of CGM are demonstrated by implementing the CUBIT mesh generation toolkit on top of CGM. Ports of CGM to SolidWorks and Pro/Engineer are underway. Several new applications based on CGM are also being developed.

## 7. References


APPENDIX A: CGM CLASS DIAGRAMS

Figure 1: Core topological classes in CGM.
Figure 2: Geometry entity classes in CGM.

APPENDIX B: CUBIT CLASS DIAGRAMS

Figure 3: ACIS-based implementation for lower level geometry entity hierarchy.
Figure 4: Geometry tool structure with respect to solid modeling engines.

Figure 5: Derivation of meshable classes from RefEntity child classes in CGM (top). Derivation of application-specific entity factory, which creates application-specific entities (bottom).