Combinatorial Solid Geometry, Boundary Representations, and Non-Manifold Geometry

Michael John Muuss and Lee A. Butler

Abstract

The history of solid modeling is reviewed. Both traditional combinatorial solid geometry (CSG) systems and traditional boundary representation (B-rep) solid modeling systems are considered. CSG models are formed by the boolean combination of "primitive" solids. Only the unevaluated CSG tree is stored, rather than an explicit representation of the final surfaces. Many applications would like an approximation of these complex CSG shapes expressed as a collection of planar N-gons. This paper focuses on a technique for computing approximate three dimensional surface tessellations. So far, B-rep solid modelers have used variations on the winged-edge data structure. The winged-edge data structure can only describe manifold shapes, limiting these systems to manifold geometry. While not restricting the set of manufacturable parts, this has made implementing boolean operations on these B-rep models more difficult.

Boolean operations and other useful modeling operations can be much more easily implemented using a representation that permits non-manifold topology to be expressed directly. A detailed look is taken at the radial-edge data structure and how it represents non-manifold conditions. While several previous publications describe the general nature of algorithms required to perform boolean operations on non-manifold geometry, details are sparse. This report describes the necessary algorithms in detail.

Some of the systems analysis issues that can result from integrating non-manifold B-rep geometry into a hybrid CSG solid modeling system are discussed. Of particular note are the specification of tessellation tolerancing, the implementation of robust tessellation, the difficulties of carrying dual representations of objects, ray-tracing non-manifold B-rep solids, using the B-rep for driving high-performance polygon rendering hardware, and interfacing to facet-based analysis codes.

To appear in **Advanced Computer Graphics Techniques**, ed. Rogers and Earnshaw, Springer-Verlag.

A History of Solid Modeling

The digital computer provides the opportunity for nearly infinite variety in the representation of information within it. The design of complex, expensive objects is always an area ripe for technological improvements, and with the emergence of graphics displays, computer aided design (CAD) packages began to appear. Designers of early CAD packages focused their efforts on the most tedious, time-consuming, and unrewarding aspect of conventional design: the process of converting a designer's sketches and notes into finished engineering drawings – the drafting process. Thus, initial CAD systems replicated traditional two-dimensional drafting techniques. Only modest gains in efficiency were possible in creating the first version of a design, but tremendous gains were realized when design modifications were needed, because the computer can retain and redraw the unchanged portions of the drawing.

Once a computer aided drafting system has been used to create a computer representation of a design, the designer (and his management) is often tempted to expect more out of the computer system than simple drawings. One would expect that a representation that depicted the boundaries of an object from several views would provide sufficient information for computing a whole variety of useful facts about the model, such as center of mass, volume, cross-sectional area, etc. Two-dimensional drafting systems were not designed for this sort of interrogation.

Four major types of difficulties have plagued wireframe systems. First, the user is required to supply a large amount of information, often at a very low level. Because of the drafting heritage, some of this information may be "construction lines" that do not actually contribute to the ultimate shape of any object. Second, because the user is providing such low-level information, it is easy to define objects which cannot be physically realized due to non-closed faces and dangling lines. Third, it is possible to construct a wireframe which has ambiguous interpretations from different views. Finally, wireframe models may include view-specific lines representing false edges such as profile lines and silhouettes [REQU82]. Engineering drawings are suitable only for interpretation by human beings, not for automatic computerized analysis.

As a new product is being designed, there are generally two or more different kinds of engineering analyses that need to be performed, such as structural analysis, thermal analysis, computational fluid dynamics (CFD), and vulnerability analysis, as well as the calculation of predictive optical, radar, infra-red (IR), and X-ray images or "signatures". Therefore, computerized drawings cannot be used to provide the geometric input required for these analysis codes. These requirements have focused attention on the evolution of an entirely different approach to representing objects: solid models.

What is a solid model?

A solid model [MUUS87a] is a computer description of closed, solid, three dimensional shapes represented by an analytical framework within which the three dimensional material can be completely and unambiguously defined [DEIT83]. Solid models differ from drafting-type systems in several important ways: objects are composed of combinations of primitive objects (some quite complex), each of which is complete, unambiguous, physically realizable, and modifiable. Because these properties hold for the primitive objects, they hold for any boolean combinations of them as well.

Completeness is assured because the representation contains a full description of a piece of solid matter; there is no view-specific information. Because the solid matter is completely described, there is no possibility for ambiguity. For primitive solids defined by specifying parameters to an analytic function, there is no possibility of having missing faces, loose edges, or other similar defects. Systems which offer boundary representations as primitive solids must carefully validate each such solid when it is created. A solid model is always amenable to further modification by boolean combination with other shapes.

These properties guarantee that all the spatial information necessary for any subsequent analysis is directly available from the model representation. Object structure and material properties can be computed at any arbitrary point in the model at any time. Therefore, solid modeling technology is particularly suited to the automation of many manufacturing and analysis tasks.

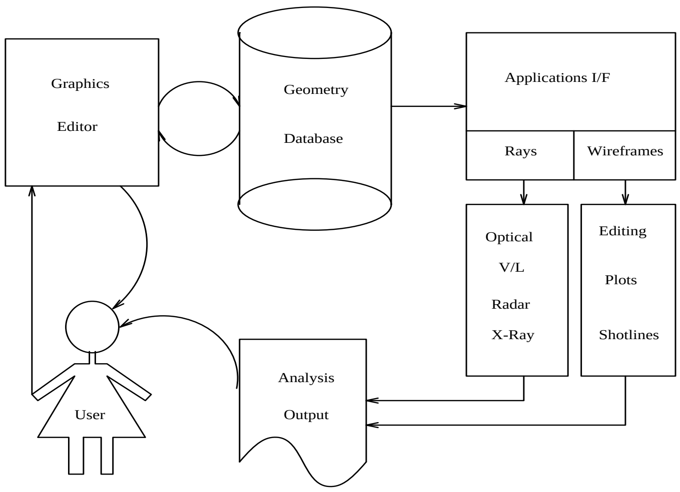

Figure 1 – The Design Loop

The Design Loop

Solid models are very useful for generating drawings or pictures of the modeled object from any viewpoint. This capability alone usually pays for the cost of developing the model. However, the solid model has a much larger role in the design process than simply automating the production of pictures and engineering drawings. Properly utilized, the solid model becomes the central element in the iterative process of taking a design from idea to prototype design to working design to optimized design. The model can be subjected to numerous engineering analyses, allowing the effects of varying many parameters to be studied in a controlled and automatic way. This iterative process is termed the "design loop", and is illustrated in Figure 1.

In a full scale solid modeling system, there is no need for initial drawings: the designer expresses the initial structures directly into the modeling system's editor, just as a modern author creates his "rough draft" directly into a word processor. At the completion of each version of the design, the model is subjected to a battery of analyses appropriate to the function of the object being designed. Strength, volume, weight, level of protection, and other similar evaluations can be reported, along with the production of a variety of images and/or drawings. These automated analyses help identify weaknesses or deficiencies in a design *early in the design process.* By detecting flaws early, the designer has the opportunity to correct his plans before having invested too much time in a bad design, or the designer can switch to an entirely different approach which may seem more promising than the original one.

In this way, the solid modeling system allows the designer to concentrate on the important, creative aspects of the design process. Freeing the designer of routine analysis permits designs to be finished in less time than previously required, or allows much more rigorously optimized designs to be delivered in comparable timeframes and at the same cost as unoptimized designs created using older techniques [DEIT85]. Furthermore, the modeling system allows sweeping design changes to be made quickly and cheaply, allowing great flexibility in the face of ever changing requirements and markets. The time needed to create a new product can be further decreased by re-utilizing elements of earlier models and then modifying them as appropriate. If an existing component already in inventory is entirely suitable for use in a new design, significant manufacturing and inventory savings will be realized. A highly interactive modeling system can allow full designs to be completed in a matter of days, where weeks or months may have previously been required [DEIT82].

Thus, the real payoff from building a solid geometric model comes when it is time to analyze it. This capability is so powerful that it ordinarily justifies any extra time or equipment investments needed to support the construction of the three dimensional solid model. Allowing the designer the opportunity to explore and analyze more design options will allow the development of the highest quality product, while also improving the work environment of the designer by eliminating boring, repetitive tasks.

Model representations

Two major families of solid model representations exist, each with several unique advantages. The first representation, developed by MAGI under contract to the Ballistic Research Lab (BRL) in the early 1960s [MAGI67], is the combinatorial solid geometry representation (CSG-rep). Solid models of this type are expressed as boolean combinations of primitive solids. Each primitive solid is a geometric entity described by some set of parameters that occupies a fixed volume in space.

The simplest solid that can be used is the halfspace [REQU82], defined by the infinite plane $ax + by + cz + d = 0$ plus all points on one side of that plane. Systems which defined all objects in terms of Boolean combinations of halfspaces include SHAPES [LANI79] and TIPS-1 [OKIN78]. While this choice of representation limits these systems to modeling convex objects with planar faces, and excludes smooth objects (or forces them to be approximated), the simplicity of this representation lends itself to very natural processing by VLSI hardware [KEDE85a, KEDE85b]. Most CSG-rep systems in use today offer quite a variety of primitive solids, ranging from various types of spheres and ellipsoids, boxes and cones, and solids defined by swept or extruded curves.

The alternative to describing solids with primitives is to adopt a boundary representation (B-rep), of which there are two sub-types: the *explicit* and *implicit* boundary representations. In an explicit boundary representation, each solid is described by an explicit specification of all the points on the surface of the solid, typically by exhaustively listing the vertices of many planar facets. Alternatively, there are implicit boundary representations, where the surface of the solid is described by an analytic function such as Coons patches [COON67], Bezier patches [BEZI74], splines [deBO78], etc.

Boundary representations offer the advantage of being able to naturally model solid objects with arbitrarily shaped surfaces, but can require a large amount of information to achieve acceptable results. Both CSG-reps and B-reps have certain advantages. With only the traditional CSG primitives, it can be exceedingly difficult and non-intuitive to attempt to describe sculptured, free-form surfaces as a boolean combination of primitives. But similarly, implementing powerful modeling operations like boolean intersection and boolean differences *on the fundamental representation itself* can be difficult with pure boundary representations. Many current B-rep modelers implement boolean operations as an external post-processing operation, because current schemes to evaluate Boolean operations are not closed. As an example of this, B-spline $\cap$ B-spline might result in polygons rather than another B-spline. In a boundary representation closed under the set of boolean operations, B-rep $\cap$ B-rep $\rightarrow$ B-rep. Thus, pure B-rep systems may be difficult to use for some types of objects, especially those with sculptured surfaces pierced by sharp rectangular gouges [THOM84].

Even though existing systems are often considered as being purely CSG-rep or B-rep, the reality is that many production CAD systems are actually hybrids of the two approaches, offering the designer the choice of primitive solids or boundary representations, as appropriate for each task. In practice, the implementation of the CSG-rep and B-rep portions of the software may be quite different, but at the highest level of abstraction each representation is just a different way of viewing the other. Faceted primitives such as boxes and wedges can be thought of as explicit B-reps, and smooth primitives such as spheres and cones can be thought of as implicit B-reps defined by analytic functions.

Interrogating a Solid Model

For more than thirty years, solid geometric modeling methods have been used to support engineering design and analyses [MAGI67, DEIT82, DEIT84a]. In such item-level studies geometry and material information are passed to various application codes to derive certain measures-of-performance. This is commonly done in structural analysis, thermal analysis, computational fluid dynamics (CFD), and vulnerability analysis. Building on the general paradigm, the techniqsues have been extended to support many predictive signature models [DEIT84b] including optical, millimeter wave (MMW), infra-red (IR), magnetic and X-ray signature generation tools. It is important to note that this type of predictive analysis must generally be supported by a solid geometric model.

The objective of a given application will to a large degree determine the most "natural" form in which the model might be presented. For example, a representation of just the *edges* of the objects in a model would be suitable for a program attempting to construct a wire-frame display of the model. There exists another family of applications which must be able to find the intersection of small object paths (e.g., photons) with the model. Generally, these alternatives are motivated by the representation of a physical process being simulated, and each alternative is useful for a whole family of applications.

Unfortunately, it is not often the case that building only *one* three dimensional model of the product is enough. Each of the different engineering analysis software packages needed to perform the analyses usually requires a different form of input. As a result, more than one kind of geometric model may have to be constructed. However, rarely do these application codes read the three-dimensional geometry and material data base directly. Rather, each application has a specific interrogation method that is invoked to obtain geometric and material attributes from a source or reference file. The physical simulation techniques used in the application software are

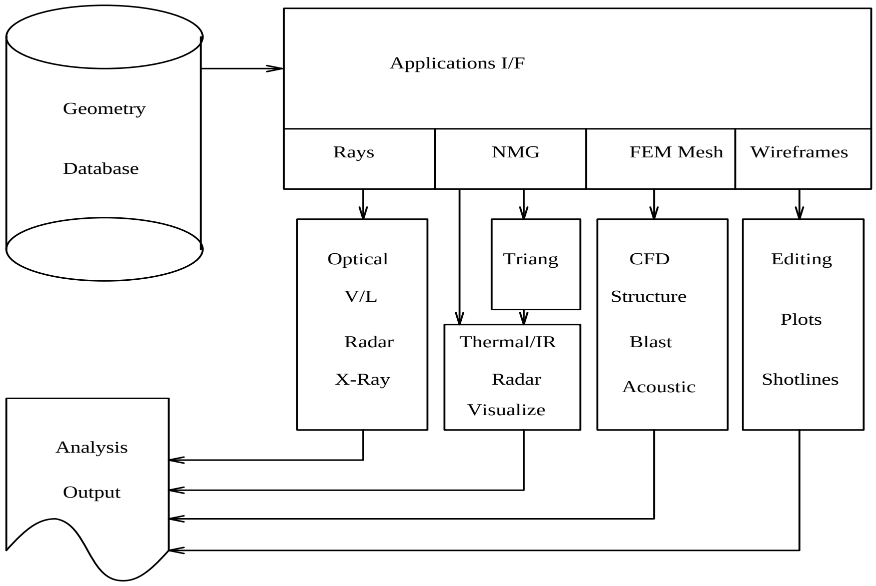

Figure 2 – The Applications Interface

therefore constrained by the available techniques for extracting geometric information from the model. Each analysis package often requires a unique form of input. Without a central geometry database that can drive all the analysis packages, the designer can be forced into having to create many different representations of each design, one for each distinctly different type of analysis code. This can be very costly and time consuming. The time needed to create a single model ranges between one man-week and several man-years, depending on the complexity of the design. Having to spend the effort to manually create the same design in different formats to drive several analysis codes is an unfortunate and expensive necessity.

The philosophy adopted in the BRL-CAD Package [MUUS87c, MUUS88a] has been to develop a broad set of analysis codes which access the same geometry database [DEIT84b]. These analyses cover the spectrum from engineering decision aids, to design validators, to signature prediction codes, to the generation of wireframe drawings, to high-resolution image generation for management comprehension and sales advantage. Key analysis capabilities have been developed to assess the strength, weight, protection, and performance levels offered by the structures represented by a solid model. Using this analysis information and additional domain-specific applications tools makes it possible to produce highly detailed designs constructed with a philosophy of *system optimization* right from the start [DEIT88]. This facilitates the rapid development of products with the desired levels of performance at the best attainable price.

To accomplish all these goals, the BRL-CAD Package provides a variety of procedural interfaces so that the diverse collection of analysis codes can be driven from a single, central geometric model [MUUS90b]. These procedural interfaces follow the natural object-oriented programming interface. An application program retrieves one or more objects from the model database, and then requests those objects to either interrogate themselves in the desired way, or to convert themselves into the desired representation. This applications interface is depicted in Figure 2. The software is designed to be highly portable to different hardware architectures, platforms and environments, and is implemented using the C programming language [RITC78b] running on the UNIX^{tm} operating system [RITC78a].

Wireframe Representation

The interactive model editor **mged** program employs three dimensional wireframe outlines of the various solid objects in order to maintain the highest possible speed of user interaction. In addition to the primary use in supporting interactive geometry viewing and modification, wireframes are also useful for visualizing complex analysis results. A particularly powerful form of this is to create a color display of the output of an analysis code (for example, temperature distribution across the surface of an object), and then overlay the analysis data with a wireframe drawing of the geometry. Wireframes are also very useful for the previewing of animation sequences. The conversion of database objects into wireframe drawings is the simplest of the application interfaces, and is very easy for the application program to utilize.

After the user specifies which objects from the model database should be displayed, **mged** retrieves the necessary database records and invokes the **ft_plot()** interface provided by **librt**. **ft_plot()** passes the database object to the appropriate object-specific wireframe converter, which generates a wireframe outline of that object. The wireframe is composed of a collection of three dimensional virtual pen-plotter *move* and *draw* operations, returned to the application as a linked list of **vlist** structures attached to the application provided **vlhead** structure. Each **vlist** structure has three elements, vl_pnt, the XYZ coordinates of a point in space, vl_draw, a flag which indicates whether the virtual pen should be moved invisibly from the current position to vl_pnt (vl_draw= VL_CMD_LINE_MOVE) or moved visibly, drawing a line from the current position to vl_pnt (vl_draw= VL_CMD_LINE_DRAW).

Ray Tracing

Many phenomena that are ordinarily difficult to model can be handled simply and elegantly with ray-tracing. For example, an illumination model based on ray-tracing first fires a ray from the eye plane into the model geometry. To approximate the total amount of light at that point in the model, the algorithm simply fires a ray at each light source and sums the contributions from visible light sources. Ray-tracing also makes it easy to deal with objects that are partly or entirely reflective, and with transparent objects that have varying refractive indices. Furthermore, by applying the proper sorts of dither [COOK84], motion-blur, shadow penumbra, depth-of-field, translucency, and other effects are easily achieved.

The power of the lighting model can be further extended by making a provision to record the paths of all the rays followed when computing the light intensity for each pixel in an auxiliary file. This capability allows one to follow the path of the light rays passing through lenses reflecting from mirrors while performing image rendering, with no additional computation. Studying the paths of light rays as they are repeatedly bent by passing from air to glass and back again has traditionally been a painstaking manual procedure for lens designers. By modeling, it becomes possible to predict lens behavior, including making a determination of the exact focal length, finding the precise influence of spherical distortions and edge effects, determining the amount of image distortion due to internal reflection and scattering, and finding the level of reflections from the lens mounting hardware. Furthermore, experiments can be conducted to determine the effects of adding or removing baffles, irises, special lens coatings, etc.

Rays begin at a point $\vec{P}$, and proceed infinitely in a given direction $\vec{D}$. The *direction vector* or *direction cosines* for the ray $(\vec{D}_x, \vec{D}_y, \vec{D}_z)$ are the cosines of the angle between the ray and each of the Cartesian axes. This vector $\vec{D}$ is of unit length, i.e. $|\vec{D}| = 1$. Any point $\vec{A}$ on a ray may be expressed as a linear combination of $\vec{P}$ and $\vec{D}$ by the formula $\vec{A} = \vec{P} + t * \vec{D}$ where valid values for t are in the range $[\,0,\ \infty)$.

The traditional approach to ray-tracing has been batch-oriented, with the user defining a set of "viewing angles", initiating a large batch job to compute all the ray intersections, and then post-processing all the ray data into some meaningful form. However, the major drawback of this approach is that the application has no immediate control over ray paths, making another batch run necessary for each level of reflection, etc.

In order to be successful, applications need: (1) interactive control of ray paths, to naturally implement reflection, refraction, and fragmenting into multiple subsidiary rays, and (2) the ability to fire rays in arbitrary directions from arbitrary points. Nearly all non-batch implementations have closely coupled a specific application (typically a model of illumination) with the ray-tracing code, allowing efficient and effective control of the ray paths. The most flexible approach of all is to provide the ray-tracing capability through a general-purpose library, and make the functionality available to any application as needed. For example, the decision of when a ray should be reflected, transmitted, or absorbed should be entirely under the control of the application program.

LIBRT Library Interface

The third generation ray-tracing capability in the BRL-CAD Package is a set of library routines in **librt** to allow application programs to intersect rays with model geometry. There are two parts to the interface: "preparation" routines and the actual ray-tracing routine. **rt_dirbuild()** opens the database file, and builds the in-core database table of contents. **rt_gettree()** adds a database sub-tree to the active model space, and can be called multiple times to join different parts of the database together.

To compute the intersection of a ray with the geometry in the active model space, the application must call **rt_shootray()** once for each ray. Ray behaviors such as perspective, reflection, refraction, etc, are entirely determined by the applications program logic, and not by the ray-tracing library. The ray-path specification determined by the applications program is passed as a parameter to **rt_shootray()** in the **application** structure, which contains five major elements: the vector a_ray.r_pt ($\vec{P}$) which is the starting point of the ray, the vector a_ray.r_dir ($\vec{D}$) which is the unit-length direction vector, the pointer *a_hit() to an application-provided routine to be called when some geometry is hit by the ray, the pointer *a_miss() to an application-provided routine to be called when the ray does not hit any geometry, and the variable a_onehit. In addition, there are various locations for applications to store state information such as recursion level, intermediate color values, and cumulative ray distance.

When the a_onehit variable is set to zero, the ray is traced through the entire model. Applications such as lighting models may often only be interested in the first object hit; in this case, a_onehit may be set to the value one to stop ray-tracing as soon as the ray has intersected at least one piece of geometry. Similarly, if only the first three hits are required (such as in the routine that refracts light through glass), then a_onehit may be given the value of three. Then, at most three hit points will be returned, an in-hit, an out-hit, and a subsequent in-hit. When only a limited number of intersections are required, the use of this flag can provide a significant savings in run-time.

The **rt_shootray()** function is designed for full recursion so that the application provided a_hit()/a_miss() routines can themselves fire additional rays by recursively calling **rt_shootray()** before deciding their own return value. In addition, the function **rt_shootray()** is fully capable of operating in parallel with other instances of itself in the same address space, allowing the application to take advantage of parallel hardware capabilities where such exist.

A simple application program that fires one ray at a model and prints the result is included below, to demonstrate the simplicity of the interface to **librt**.

```
struct application ap;
struct rt_i *rtip;
main() {
      rtip = rt_dirbuild(``model.g'');
      rt_gettree(rtip, ``car'');
      rt_prep(rtip);
      VSET( ap.a_point, 100, 0, 0 );
      VSET( ap.a_dir, -1, 0, 0 );
      ap.a_hit = &hit_geom;
      ap.a_miss = &miss_geom;
      ap.a_rt_i = rtip;
      rt_shootray( &ap );
}


hit_geom(app, pp)
struct application *app;
struct partition *pp;
{
      printf(``Hit %s'', pp->pt_forw->pt_regionp->reg_name);
}
miss_geom(){
      printf(``Missed'');
}
```

Ray Intersection Data

If a given ray hits something, the a_hit() routine is called, and is provided a pointer to the head of a doubly-linked list of **partition** structures. Each **partition** structure contains information about a line segment along the ray; the partition has both an "in" (**pt_inhit**) and an "out" (**pt_outhit**) hit point. Each hit point is characterized by the hit distance **hit_dist**, which is the distance t from the starting point **r_pt** along the ray to the hit point. The linked list of **partition** structures is sorted by ascending values of **hit_dist**. As a result of this definition, the "line-of-sight" distance between any two hit points can be determined simply by subtracting the two **hit_dist** values. This will give the distance between the hit points, in millimeters.

If the variable **a_onehit** was set non-zero, then only the first **a_onehit** hit points along the partition list are guaranteed to be correct; any additional hit points provided should be ignored. This is usually important only when **a_onehit** was set to an odd number; the value of **pt_outhit** in the last **partition** structure may not be accurate, and should be ignored.

If the actual three-space coordinates of the hit point are required, they can be computed into the **hit_point** element with the C-language version of $\vec{A} = \vec{P} + t * \vec{D}$:

```
VJOIN1(hp->hit_point, rayp->r_pt, hp->hit_dist, rayp->r_dir);
```

Surface Normals

As an efficiency measure, only the hit distances are computed when a ray is intersected with the model geometry. The surface normal at any hit point can be easily acquired by executing a C macro. In addition to providing the unit-length outward-pointing surface normal in struct **hit** element **hit_normal**, this macro also computes the three-space coordinates of the hit point in struct **hit** element **hit_point**:

```
RT_HIT_NORM( hitp, stp, rayp );
```

Gaussian Curvature

For any hit point, after the surface normal has been computed, the Gaussian surface curvature at that hit point can be acquired by executing the C macro:

```
RT_CURVE( curvp, hitp, stp );
```

A **curvature** structure has three elements, the unit vector **crv_pdir** pointing in the direction of principle curvature, the scalor **crv_c1** (or c_1) giving the curvature in the principle direction, and the scalor **crv_c2** (or c_2) giving the curvature in the other direction. c_1 and c_2 are the inverse radii of curvature, and $|c_1| \leq |c_2|$, i.e. c_1 is the most nearly flat principle curvature. A positive curvature indicates that the surface bends toward the (outward pointing) normal vector at that point. The other principle direction is implied and can be found by taking the cross product of the normal with **crv_pdir**, i.e., $\vec{p}dir2 = \vec{N} \times \vec{p}dir1$.

U-V Mapping

Both the U and V coordinates range from 0.0 to 1.0 inclusive. A given (U,V) coordinate may appear at more than one place on the surface of the object. The (U,V) coordinate of the hit point is returned in **uvcoord** structure elements **uv_u** and **uv_v**. For any hit point, after the value of **hit_point** has been computed, the U-V coordinates of that point can be acquired by executing the C macro:

```
RT_HIT_UVCOORD( ap, stp, hitp, uvp );
```

For some simple lighting-model applications, it is sometimes desirable to create a mapping between the coordinate system on the surface of an object to the coordinate system of a square, the U-V coordinates. This is generally used to drive simple, two dimensional *texture mapping* algorithms. The most common application is to extract a "paint" color from a rectangular RGB image file at coordinates (U,V), and apply this color to the surface of an object. These parameters can also be used to simulate the effect of minor surface roughness using the *bump mapping* technique. Here, the U and V coordinates index into a rectangular file of perturbation angles; the surface normal returned by RT_HIT_NORM() is then modified by up to ± 90 degrees each in both the U and V directions, according to the stored perturbation.

In addition, the approximate "beam coverage" of the ray, in U-V space, is returned in the structure elements **uv_du** and **uv_dv**. These approximate values are based upon the ray's initial beam radius (**a_rbeam**) and beam divergence per millimeter (**a_diverge**) as specified in the application structure. These delta-U and delta-V values can be helpful for anti-aliasing or filtering areas of the original texture map to produce an "area sample" value for the hit point.

Three Dimensional Surface Mesh

Combinatorial Solid Geometric (CSG) models are formed by the boolean combination of "primitive" solids [MUUS87a]. For example, a plate with a hole is most easily modeled as a plate primitive minus a cylinder primitive. It is important to note that in CSG models, there is no explicit representation of the surfaces of the solids stored; indeed, for complex boolean combinations of complex primitives, some of the resultant shapes may have very convoluted topology and surfaces that may be at best high degree polynomials.

There are many applications that would benefit from being able to express an *approximation* of the complex shapes created using CSG modeling as a collection of planar N-gons which together enclose roughly the same volume of space as the original CSG solid. The most obvious such application is to drive polygon-based rendering routines (lighting modules) for predictive optical signatures. On many modern workstations there is direct hardware or firmware support for high-speed rendering of polygons [MOLN87]. In addition, there are whole collections of polygon-based predictive infra-red and radar signature programs. The very best predictive radar signatures can be calculated using the Method of Moments, which requires having a three dimensional surface tessellation to sub-wavelength resolution of the entire model. A technique for computing this approximate three dimensional surface tessellation is the focus of the majority of this paper.

Topological Representation

Some predictive radar signature codes, such as the TRACK code of GTRI [PELF86], do not operate directly on a geometric representation of an object. Instead, they rely on the fact that large radar returns occur primarily due to the existence of dihedral and trihedral structures in the object. Rather than describing a vehicle simply as a collection of these topological structures, it is possible to analyze a three dimensional solid model to locate all instances of the topological features of interest. For example, the software could locate planar face elements; edges where two locally planar elements join to make a dihedral, edges where three locally planar elements join to make a trihedral, etc. Then this list of topological features is used as input to the feature-based analysis code.

System-Level Issues

It should be abundantly clear that it is highly desirable to have a single, central geometric database which can be interrogated by a full suite of analysis codes. Many fundamentally different kinds of model interrogation need to be supported in order to meet this goal. If a CAD system was being designed afresh, without *any* ties to the past, then any underlying representation could be chosen and used. In this section, the design considerations for expanding the kinds of model interrogation available to the BRL-CAD Package will be explored.

While the BRL-CAD Package comprises a large amount of finished computer software, the main investment is in the existing library of geometric models. The amount of effort required to replicate these models can be conservatively estimated in the hundreds of man-years. In addition, the experience of the large number of trained model designers and design analysts is quite significant. Clearly, any design intended to expand the kinds of model interrogation available within the system must protect these investments.

The BRL-CAD Package as it stood in 1989 could be considered to be one of the very few production CAD systems which was based purely on the combinatorial solid geometry (CSG) technique. The final shapes of all objects were created by boolean combinations of primitive solids. No attempt was made to represent in explicit form either the topology or the surface geometry of any object. The exact nature of the final form of an object was discovered only on a point by point basis, by sampling the object with ray-tracing. At the same time, the assortment of primitive solids available to designers was definitely a hybrid combination of traditional CSG primitives and more recent boundary-representation (B-rep) primitives. The primitive solids described by their boundaries included a variety of faceted solids, as well as solids defined by a closed collection of non-rational B-spline surfaces. This rich collection of primitive solids could be combined by the designer using any number of boolean operations. However, complex combinations of primitives could be difficult to visualize except through careful study of the three dimensional wireframe approximation rotating in real time, coupled with the judicious use of ray-traced renderings.

Given that modern computer workstations [MOLN87] are now commonly available with integral polygon rendering hardware, and that hardware rendering speeds can exceed one million polygons per second, it seems highly desirable to be able to take advantage of this hardware. Also, as discussed earlier, there is a burgeoning supply of analysis software that simply can not make do with a geometry interrogation interface that supports only the ray-tracing paradigm. What was missing from the BRL-CAD Package was a way of obtaining an *explicit* description of the final shape of modeled objects. Our quest then is for a technique suitable for obtaining an explicit description of geometric objects.

Boolean Operations

There are several classes of modeling operations that are very conveniently expressed in terms of boolean operations. For example, holes can be bored in objects by subtracting a "drill bit" solid from the original work piece, and the primary shape of an injection mold can be created by subtracting the desired final product from one or more blocks of mold material. Designers who have experienced the conceptual power of using boolean operations to construct complex shapes are unlikely to want to switch to a system that does not permit the use of boolean operations. Given that the existing investment in geometric models depends heavily on the use of boolean operations, it was concluded that adding the capability for obtaining an explicit description of modeled objects must provide full support for the use of boolean operations.

This conclusion immediately places several rather severe requirements on the design. Most importantly, it requires that the underlying representation used to hold the explicit description of the modeled objects must be *closed* under boolean operations. That is, given the explicit description of objects A and B, then any boolean combination of A and B must be representable as an explicit description expressed in terms of the same underlying representation. Several available choices of representation will be considered.

The Underlying Representation

A strictly polygonal representation could be selected. While performing boolean operations on solids described as collections of polygons is not easy, the representation can (with certain special definitions) be considered to be closed under boolean operations, and algorithms to accomplish the boolean evaluation have been published for several years [LAID86].

A representation comprised exclusively of rectangular parametric surfaces, such as B-splines or similar tensor-product surface patches could be used. However, research to date has shown that while B-spline surfaces can be combined using boolean operations, the resulting object can not be expressed strictly in terms of B-splines [THOM84]. Instead, a mixed representation of B-splines and polygons is produced, and this mixture becomes ungainly when subjected to repeated boolean operations. This occurs because the boolean combination of rectangular parametric surfaces is not necessarily bounded by rectangular parametric surfaces; i.e., the representation is not closed. Recent work has suggested that a representation comprised of trimmed B-splines and shared-edge polylines might be closed under boolean operations [COBB84], but a full implementation is not yet known to exist.

No other good choices for an underlying representation could be found. Because the B-spline representation does not have closure under the set of boolean operations, it regrettably could not be used. Therefore, there was no choice; the explicit representation of modeled objects would have to be expressed in terms of collections of polygons. This certainly met the requirements of the polygon based application codes, but it posed a whole host of new questions.

Topology

Given that the explicit geometric representation of surfaces will be approximated using polygons, there still remains the question of how to deal with the representation of the topology. The simplest strategy would be to simply ignore topology, and store solid objects as collections of polygons. This has several drawbacks because these objects are intended to represent enclosed volumes of space. It could be quite difficult to verify that a given collection of polygons is in fact a valid solid, without any cracks, dangling faces, or missing faces. Also, implementing boolean operations without any explicit topology would be nearly impossible.

Traditionally, solid modeling systems based on boundary representations such as PADL [REQU82] have employed the *winged edge* data structure for storing topology, as discussed earlier. The disadvantage of the winged edge data structure is that it is simply unable to represent many of the non-manifold conditions that arise in the construction of complex shapes (such as finite element meshes), and the non-manifold conditions that arise from the use of the intersection operator ($\cap$) on two objects that share only a single face, edge, or vertex. Those systems employing the winged edge data structure that also support boolean operations (such as PADL) have had to resort to the use of *regularized boolean operators* which are defined in such a way as not to produce any non-manifold results. These regularized boolean operators are denoted by a superscript asterisk, i.e., regularized union is $\cup^*$, regularized intersection is $\cap^*$, and regularized subtraction is $-^*$.

It seems unfortunate to have to restrict boolean operations on the polygonal representation to not include any non-manifold results. Some applications might desire them, and others might not; unwanted non-manifold results are easy to discard, but there is no easy way to re-infer the existance of non-manifold results if the representation can not express them. This is akin to the "language limits thought" concept from cognitive psychology. Non-manifold results can be useful in a variety of ways. One example is in interference or overlap checking. The intersection of two objects can be computed. If the intersection is the null set, then the two objects are disjoint, and there is space between them. If the intersection is a solid object, then the two objects overlap in the given volume, which generally signals a modeling error. If the intersection is a lone face, edge, or vertex, then the two objects exactly touch, but do not overlap.

The inability of the "winged-edge" data structure to represent non-3-manifold conditions prompted Weiler to evaluate existing edge-based data structures [WEIL85]. This resulted in the development of a new data structure suitable for representing all the Non-3-Manifold Geometric (NMG) and topological configurations that boolean operations might produce [WEIL87]. This new data structure has been dubbed alternately the "radial-edge", or "NMG" data structure. Because of this structure's ability to handle 3-manifolds (solids), 2-manifolds (faces), 1-manifolds (edges), and 0-manifolds (points), NMG objects are closed under boolean operations.

The NMG data structures have the advantages of complete generality and closure under boolean operations, plus they encode the full topological structure of an object, as well as the geometric information. The representation contains full topology information, so that the relationships between vertices, edges, loops, faces, and shells are continuously available. Geometry is associated with each topological element. For example, the geometry information associated with a planar face is the plane equation which includes the outward-pointing surface normal; the plane equation does not have to be re-derived from the vertices. This generality does come at a significant price in increased memory use compared to the winged-edge data structure. However, because of the anticipated frequency of occurance of non-3-manifold conditions in CSG modeling, both intentionally and as part of various analysis operations, the NMG data structures were selected to contain the approximate surface representation.

Conversion versus Post-Processing

Several of the existing and planned primitive solids have very complex curved shapes. Some examples include the torus, the truncated generalized cone, the hyperboloid of two sheets, the general polynomial solid, and the B-spline solid. In general, these solids have few or no sub-sections which are flat. Thus, these solids can not be represented by a collection of polygons without loosing essential information about the very nature of the solids. Yet, the desire to have a homogeneous representation suggests that all existing solids should be converted to a single new underlying representation, so as to take advantage of the boolean closure property, and to enable applications to access the explicit surface representation.

The two options to consider are either (a) making a one-time conversion of all existing models to a polygonal form, and then performing all subsequent editing and processing on a polygonal database; or (b) retaining the existing, implicit combinatorial solid geometry database, and providing some form of post-processing capability to convert the implicit CSG shapes into an explicit, polygonal form.

Many existing application codes that use the ray-tracing paradigm depend on being able to obtain very accurate surface normal and Gaussian curvature information at any point on an object. While surface normal interpolation [GOUR71] can be used with a polygonal representation, many artifacts can be introduced, including surface normal discontinuities across polygon boundaries and inconsistencies between the apparent and actual location and shape of profile edges. While these artifacts may be entirely tolerable in simple rendering applications, they are entirely inappropriate in a CAD system targeted for high-resolution engineering analysis. Finally, it would be extremely distasteful to have to *reduce* the quality of the ray-tracing style of interrogation in order to accomodate the ability to obtain explicit object surfaces.

Therefore, the conclusion was that the existing combinatorial solid geometry database would remain unchanged, and that any application that required an explicit representation of the modeled shapes would obtain that explicit representation through some form of conversion of the underlying CSG database. While this capability needs to be available to any application at any time, it can still be thought of as being a "post-processing" operation.

This choice of strategy has a number of fortunate implications. Because a polygonal representation of a fundamentally curved shape could never be more than an approximation of that shape, it is impossible to choose a single resolution approximation that would satisfy all applications. Using a coarse approximation would quickly produce complaints about a lack of accuracy in the modeling system. For example, few engineering applications could tolerate octagonal pipes being substituted for genuine round pipe. On the other hand, using a very fine approximation would consume gargantuan amounts of memory, which would impede simple operations such as model editing and display.

With this strategy, the polygonal representation can be considered just an *approximation* of a much more accurate underlying geometric model. Furthermore, because this approximation can be created dynamically, each application has the opportunity to control the resolution of the approximation being created. This permits each application to obtain exactly the degree of resolution required, without having to worry about the approximation being too coarse or too bulky.

Implementation Tactics

The strategy that has been adopted has several key points. Whenever an application requires an explicit description of the surface of an object, it will access the *Application Interface* to obtain an approximation created to meet accuracy requirements of this particular application. The explicit representation of the object will be returned as a collection of planar faces embedded in the NMG data structures.

The existing Application Interface to the geometry database already has a nice simple programming interface. Thus, it seems appropriate to have a high-level interface similar to **rt_gettree()**. The application would indicate what objects or object heirarchies are to be retrieved from the database, and what the accuracy requirements are, and would be given in return a collection of NMG data structures that would contain the surface approximation of the indicated objects.

The actual implementation of the application interface can be further broken into several pieces. The existing routine **db_walk_tree()** is used to retrieve the indicated objects from the database. Each solid retrieved from the database needs to be converted to an approximate faceted form stored in NMG data structures. This operation is referred to as *tessellation*. As each boolean operation is encountered in the database, the appropriate tessellated solids will have that boolean operation performed by the routine **nmg_do_bool()**. This routine will take the two tessellated objects and combine them according to the boolean operation back into a consistent set of solid tessellated objects. Until very recently, it has been this step that has proven the most difficult [LAID86]. Once all the objects have been retrieved from the database and combined with their boolean formulas, the resultant collection of NMG objects is returned to the application. (More precisely, the application is returned a **struct model** which contains pointers to the requested objects, stored as one or more **struct nmgregion**s). This architecture gives rise to the schematic diagram in Figure 3.

In the existing ray-tracing implementation, **librt** makes heavy use of an object-oriented style of procedural interfaces to geometry support routines. This object-oriented programming interface already defined a standardized set of operations that could be performed on geometric objects. The operations include having a geometric object read itself into memory, describe

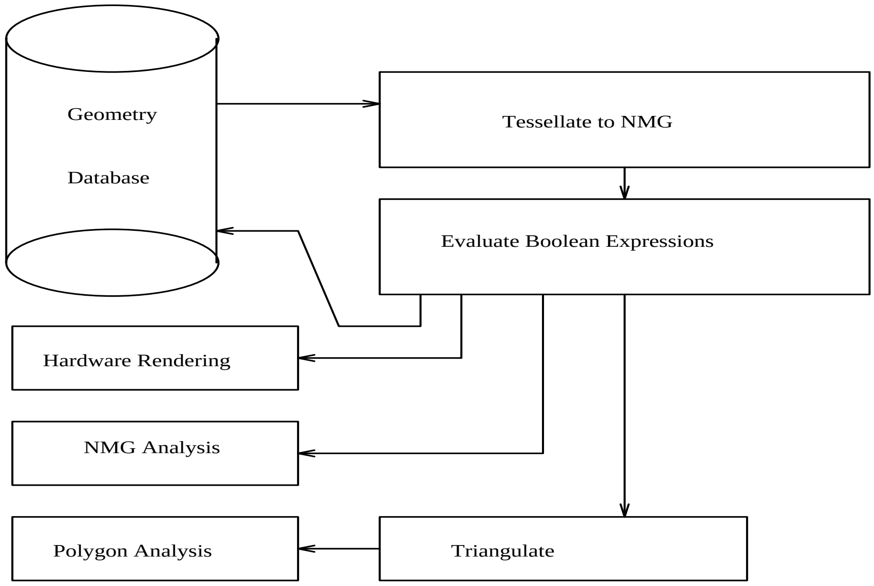

Figure 3 – Schematic NMG Wiring Diagram

itself, produce a wireframe representation of itself, and intersect a ray with itself. This interface was extended to define a new operation to require an object to tessellate itself into an NMG data structure. This has the highly desirable property that all the processing related to a given primitive solid remains centralized in a single solid-specific geometry module. Thus, adding a new primitive solid requires only the addition of a single module to the library; none of the analysis codes ever need to be modified when designers begin using a new kind of primitive.

Interface to Applications

From the schematic diagram it should be clear that the final evaluated NMG solid object can be employed in a variety of ways. The primary use will be for input to visualization and analysis software that needs an approximate three dimensional surface mesh of the solid model. However, a very powerful additional use will be to create new faceted shapes which are then stored back in the database as new geometric objects, suitable for future editing or analysis.

As will be explained in more detail in subsequent chapters, each face of an NMG object will be composed of one or more planar N-gons, each potentially non-convex and with embedded internal loops. Applications that are prepared to deal with this topological richness may operate on the NMG representation directly. However, for those applications that require a simpler face topology, a simplification routine exists that will reduce each face to a collection of planar N-gons. These simplified N-gons may be non-convex but will have no embedded internal loops. Finally, for applications that prefer faces to be collections of simple triangles, a *triangulator* routine will be provided that convertes the NMG faces into well-behaved triangles [GOOD89].

Non-Manifold Geometry

One of the keys to the NMG approach lies with the concept of the radial-edge representation as opposed to the classical winged-edge. Within the winged-edge representation, an edge represents the boundary between exactly two faces. The drawback to this is that an infinite number of planes or faces can intersect in a single line in three dimensional space. The winged edge representation only allows for a pairwise topological connection between these faces. The radial-edge representation topologically links all faces which share the edge as a line of intersection. The implementation described in this paper is heavily patterned after a description of the radial-edge data structures and operations written by Weiler [WEIL87].

Separation of Topology and Geometry

The basic topological elements are the vertex, edge, loop, face, shell, region, and model. The relationship between these elements of the hierarchy is depicted in Figure 4. Note that for any element within the hierarchy, there is a direct path from that element to the element which is one level higher, and also to the element which is one level lower.

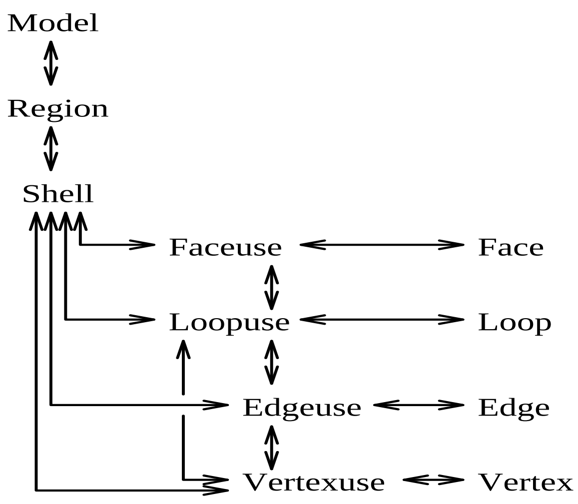

Figure 4 – NMG Structure Hierarchy

The *vertex* represents a unique topological point. The *edge* is a line or curve in space terminated by either one vertex, or two distinct vertices. The *loop* is either a single vertex, or a circuit of one or more edges. A loop defines a circuit or a boundary of a space. The *face* consists of one or more loops, and represents an actual surface area. The use of a loop within a face may define either an exterior loop or an interior loop. Exterior loops include an area in the face. Interior loops exclude an area from the face surface, thereby causing a hole in the face.

The *shell* is either a single vertex, or a collection of faces, loops, and edges. The collection of faces in a shell may enclose a volume, thereby creating closed objects, or may represent arbitrary surfaces. Loops and edges of a shell are refered to as "wire loops" and "wire edges." They may be used in creating wireframe aspects of the model.

The *region* is a collection of shells, and the *model* is a collection of regions.

For the elements vertex, edge, loop, and face, there is a distinction between the existence of the element and instances of the use of the element. This allows multiple topological elements to share the same underlying form and geometry. The *vertexuse* is an instance or a use of a vertex. The *edgeuse* is a directed instance of an edge. The *loopuse* is an instance of a loop. The *faceuse* is an instance or a use of a face. Each side of the face is uniquely represented by a faceuse, i.e., every face is referenced by exactly two faceuses.

Finally, note that each topological element makes reference to a separate geometric element. As a result of this separation of topology and geometry, the kinds of geometric support in the modeling system may evolve into richer and richer forms, while continuing to enjoy a common set of topological elements with a stable interface. For example, the system described in this paper is based upon manipulation of planar facets. However, in the future the geometry support of the system will be expanded to support curved facets or B-splines, while retaining the same interface to the topology.

The Data Structures in Detail

Magic Numbers and Memory Descriptors

The first **long** (longword of memory) in any of the NMG structures is dedicated to a *magic number*. It will be found either listed explicitly as the first entry in the structure definition, or it will be hidden, obtained implicitly from the **struct nmg_list** sub-structure (described below) which is the first entry in the structure definition. Thus, it is always located at an offset of zero bytes from the start of the structure. This magic number serves a dual purpose. First, every subroutine that is passed a pointer as a parameter can dereference that pointer to obtain the magic number. If the magic number obtained does not match the magic number assigned for use with that kind of data structure, then either memory has become corrupted, or a defective pointer has been provided as a parameter. Given that some NMG operations may have to dereference pointers through seven connected data structures, it is advantageous to detect invalid pointers as early in the process as possible. Second, some data structures employ *generic* pointers which may refer to one of several different kinds of structures. Rather than using an extra word of memory to store a type indicator, the generic pointer can be dereferenced to obtain the magic number, and thus the identity, of the referred-to structure.

Structure Indices

A count of the number of structures within a model is kept. Each structure type contains an integer "index" member which uniquely identifies the instance of a structure type within the model. This assists algorithms which must temporarily associate some flag or bit of information with structures. In these instances, an array can be allocated with the appropriate size so that there exists one array element for each structure in the model. As the model hierarchy is walked, the index elements can be used to quickly access the temporary information appropriate for this particular structure. Applications for this include copying the model between memory and disk, and flags to help assure that every structure is visited or operated upon exactly once.

Linked lists

The NMG data structures make frequent use of doubly linked lists. With one exception, they are all implemented using the same strategy and list manipulation macros. (It is the radial-edge linked list which does not follow this form.) All linked lists are made up of "nmg_list" structures:

```
struct nmg_list  {
    long            magic;  /* magic number */
    struct nmg_list *forw;  /* ''forward'', ''next'' */
    struct nmg_list *back;  /* ''back'', ''last'' */
};
```

The magic number field of this structure identifies the node as either a list head or as a structural element. The two pointers are to the successor and predecessor of the node in the list. Every list has one structure which is dedicated to functioning as the "head" node. An empty list consists of a "head" node whose "forw" and "back" members are pointers to the "head" node. Defining the doubly linked list as having an explicit "head" means that the enqueue and dequeue operations can operate on any member of the list, and they do not need to refer to the head.

Consider the task of making a linked list of **edgeuse** structures. The first element of the **edgeuse** structure is an **nmg_list** structure named **l**. Thus, the address of **l** is a *pun* (or homonym) for the address of the **edgeuse** structure. By adding **l** to a linked list, in reality the whole **edgeuse** structure has been added to the linked list. This allows the list manipulation functions to be generalized to handle lists of any kind of structures. The manipulation routines merely operate on **nmg_list** objects and need not know details about what structure types are in the list. The only requirement is that the **nmg_list** structure must appear as the first element in the containing structure. A rich set of macros exist for insulating the programmer from all the details of inserting and deleting elements from a list, walking a linked list, and various initialization and clean-up operations.

Naming Conventions

When creating variable names and structure member names, the implementation makes heavy use of abbreviations. As a result, it was important to regularize the abbreviation strategy. The suffix "_p" is appended to the end of all pointer variables in the structures. The first characters of the variable indicate the type of object the pointer references. For example, a pointer to a vertex structure would be "v_p", and "vu_p" would be a pointer to a vertexuse structure.

Some structures types may have a variety of different structure types as "parent" or "child" structures. Since each structure maintains a pointer to its parent and children, a method for maintaining a syntactically correct handle for such objects is required. Such pointers are stored as unions of pointers to each possible type of structure required, plus a pointer to a magic number. For example, Figure 4 demonstrates that a vertexuse may have either a shell, a loopuse or an edgeuse for a parent. As a result, the vertexuse structure will contain something similar to the following:

```
union {
    struct shell   *s_p;
    struct loopuse *lu_p;
    struct edgeuse *eu_p;
    long           *magic_p;
} parent;
```

This union provides a handle for each possible type of parent for the vertexuse structure. In addition, it contains a "pointer to magic number" handle. This allows the type and validity of the parent to be identified as a particular structure type, *before* the parent structure is referenced. This union owes its existence to the Pascal origins of the implementation by Weiler. In the C language, a simple "pointer to long integer" and the language's ability to *coerce* one type of pointer into another type of pointer using type-casting would have been sufficient. It is debatable whether type-casting or the union-name-handle approach produces more readable source code.

The names for linked list "head nodes" use the "_hd" suffix, and the first letters indicate the type of object in the list. For example, the head of a list of vertexuse structures would be called "vu_hd". When a structure is to be a part of a linked list, the first element of the structure will be an element "`l`", the list node which becomes a member of a linked list. As described in the section on linked list implementation, by keeping this item as the first element in the list, the list manipulation interface can be made fully general for lists of all types of elements.

Vertex and Vertexuse

The simplest element in the system is the *Vertex*. A vertex represents a single point within the topological space of the object being modeled. It also serves as a linkage point for connecting the topological model with the geometrical data. The structures "vertex" and "vertexuse" can be conceptually viewed as in Figure 5. The magic number is stored directly as the first element of the structure. The second item is an "nmg_list" substructure. This substructure forms the head of a doubly-linked list of all the *uses* of this vertex. The member "vg_p" is a pointer to the geometry. The index element keeps the structure index for the particular vertex instance. The vertex structure is referenced through a vertexuse structure.

The "l" element of the vertexuse structure is entered on the vertex structure's "vu_hd" list, which is a list of all uses of the vertex. This linked list node also contains the magic number for the vertexuse structure. A vertexuse may be needed by any of the higher level objects: shell, loopuse, or

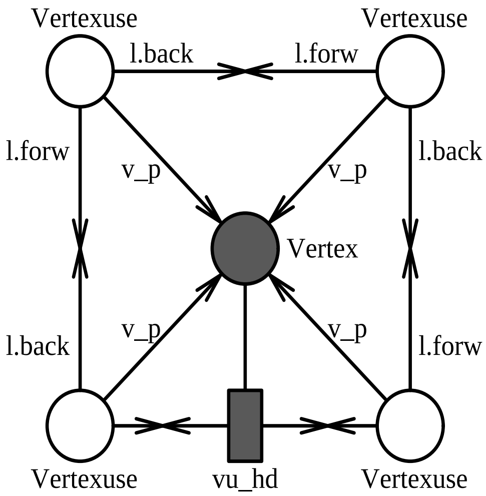

Figure 5 – Vertex and Vertex Uses

```
struct vertex {
    long            magic;
    struct nmg_list vu_hd;
    struct vertex_g *vg_p;
    long            index;
};
struct vertexuse {
    struct nmg_list     l;
    union {
        struct shell    *s_p;
        struct loopuse  *lu_p;
        struct edgeuse  *eu_p;
        long            *magic_p;
    } up;
    struct vertex       *v_p;
    struct vertexuse_a  *vua_p;
    long                index;
};
typedef double point_t[3];
struct vertex_g {
    long     magic;
    point_t  coord;
    long     index;
};
```

edgeuse. The union "up" contains a pointer to each of these three structure types. It also contains a pointer to a magic number, which can be used as a handle for getting the magic number of the parent. The element v_p is a pointer to the vertex being *used* by this vertexuse. The vertexuse may have an associated vertexuse attribute structure. This structure is referenced through the "vua_p" pointer. One example of such information might be to store a surface normal for each vertexuse, suitable for intensity interpolation shading algorithms used in Gouraud shading [GOUR71], or for normal-vector interpolation shading algorithms.

The vertex geometry structure is very simple and straightforward. It consists of a magic number, the coordinates of the vertex, and a structure index. The coordinates are stored in a variable of type point_t, which is an array of three double-precision floating point numbers.

Edge and Edgeuse

The next topological element is the *edge*. The edge represents a line or curve between a pair of vertices. The unique members of this structure

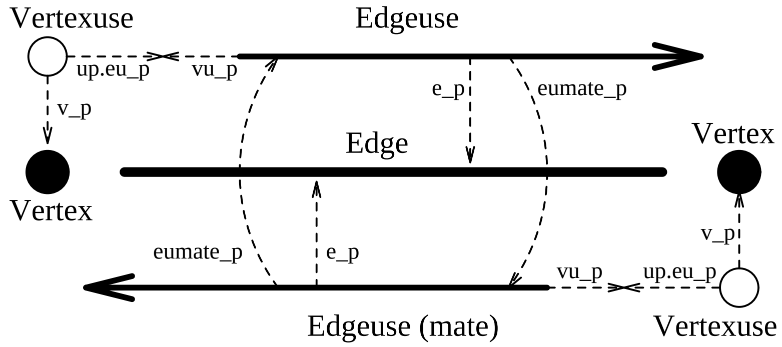

Figure 6 – Edge and Edgeuse

are "eu_p" which is a pointer to one of the uses of the edge, and "eg_p" which is a pointer to the edge geometry. The edge geometry structure is reserved for future curved-edge support. Note that the edge structure itself does not reference the endpoints of the edge. The endpoints are accessed through the edgeuse structure because almost all references to the edge endpoints occur while processing the edgeuse structures [WEIL87]. Within the edgeuse structure, the edge struct pointer "e_p" ties the edgeuse to the appropriate edge.

```
struct edge {
    long            magic;
    struct edgeuse  *eu_p;
    struct edge_g   *eg_p;
    long            index;
};
struct edgeuse {
    struct nmg_list     l;
    union {
        struct loopuse  *lu_p;
        struct shell    *s_p;
        long            *magic_p;
    } up;
    struct edgeuse      *eumate_p;
    struct edgeuse      *radial_p;
    struct edge         *e_p;
    struct vertexuse    *vu_p;
    long                index;
};
```

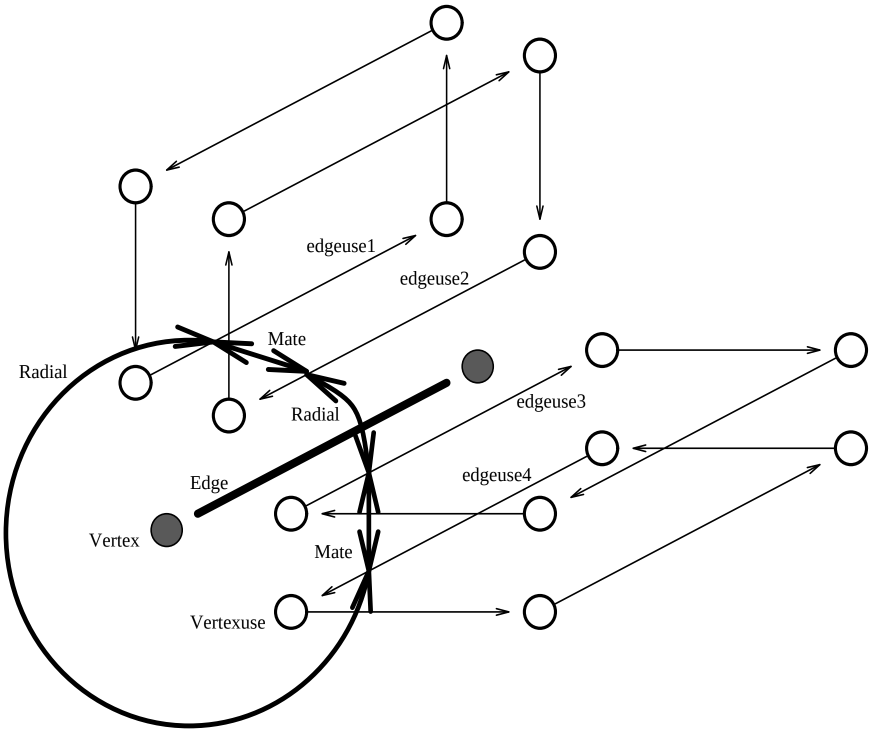

Figure 7 – Radial Edge Structure

Each edgeuse references one vertex via the vertexuse structure pointer "vu_p". The other end of the edge/edgeuse is referenced through the first edgeuse's *mate* edgeuse. The pointer "eumate_p" connects an edgeuse to it's edgeuse mate. The eumate_p edgeuse is conceptually the use of the edge on the opposite side (interior/exterior wise) of the face from the existing edgeuse. The "eumate_p" and "radial_p" pointers form a linked list of the "radial uses" of an edge. The edgeuse referenced by "eumate_p" is on the opposite loopuse/faceuse of the same loop/face. The edgeuse "l" list node is used to form loops of edges, or to keep lists of all of the "wire edges" which are a part of a shell. Like the vertexuse, the edgeuse can be referenced by one of two different types of structures higher in the hierarchy. The same technique is used for providing handles for these "parent" structures as was used for the vertexuse.

Loop and Loopuse

A loop defines a boundary or circuit. Conceptually, a loop consists of either a single vertex, or a series of one or more edges in a circuit. Like the edge, most of this information is stored in the use structure. The loop structure member "lu_g" is a pointer to the geometry support for the loop.

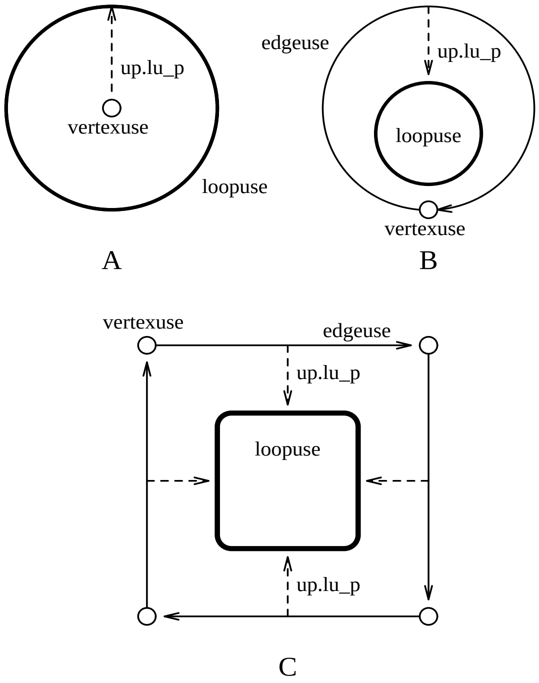

Figure 8 – Variations of the Loop

The member "lu_p" connects the loop structure to one of the loopuses of the loop. From that loopuse, the other use of the loop is reached through the "lumate_p" structure pointer. The geometry structure continas a bounding box for the loop.

The loopuse structure is where most of the loop details are handled. The first element is a linked list node, so that faces and shells may keep track of collections of loopuses. The union "up" provides a handle for all possible parents of the loopuse. The "lumate_p" member provides a pointer to the other use of the same loop as this loopuse. Often the other loopuse will be the use of this loop on the opposite surface of a face. The "orientation" member is used to define whether this loopuse is being used to enclose space, or exclude space within a face. The loopuse is associated with a particular loop through the pointer "lu_p." The structure member "down_hd" is a list head which is used to access the component elements which form the loop. In the case where the loop is made up of one or more edges, this list will consist of a series of edgeuses. When the loop is formed on a single vertex, the first and only item in the list will be the vertexuse associated with the loopuse.

Figure 8A depicts a use of a loop (a loopuse) where the boundary formed by the loop consists of a single vertex. Figure 8B depicts a loop which is formed of one edge, and Figure 8C depicts a loop which is formed of four

```
struct loop {
    long            magic;
    struct loopuse  *lu_p;  /* Ptr to one use of this loop */
    struct loop_g   *lg_p;  /* Geometry */
    long            index;
};
struct loopuse {
    struct nmg_list     l;
    union {
        struct faceuse  *fu_p;
        struct shell    *s_p;
        long            *magic_p;
    } up;
    struct loopuse      *lumate_p;
    char                orientation;
    struct loop         *l_p;
    struct nmg_list     down_hd;
    long                index;
};
```

edges. When the loopuse is made up of edgeuses, they will be arranged to form a circuit. The mates to these edges will form a circuit for the loopuse mate. If the loopuses are a part of a faceuse, the edgeuses are arranged in a special orientation. When the cross product of the vector of the edgeuse and the normal vector for the faceuse is taken, the resultant vector should point into and along the surface of the face. See Figure 9.

Face and Faceuse

The face represents a planar or curvilinear two-dimensional boundary or surface. The pointer "fu_p" is a pointer to one of the two faceuses of the face, the other being reached through that faceuse's "fumate_p" pointer. The pointer "fg_p" references the geometry structure for the face. The face geometry structure keeps a bounding box, and the plane equation of the face. The equation is stored as a 4-tuple of double precision floating point numbers. A point $\vec{A}$ which lies on the plane will satisfy the following relation:

$$N_x * A_x + N_y * A_y + N_z * A_z - N_3 = 0$$

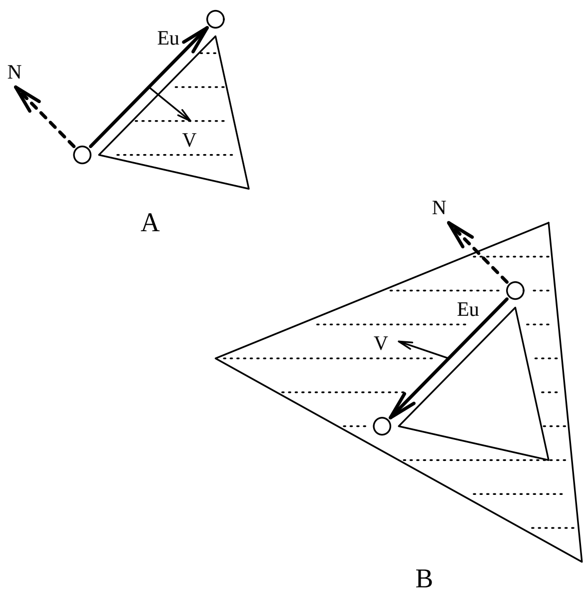

Figure 9 – Orientation of Edgeuses within a Loopuse

```
struct face {
    long                magic;
    struct faceuse      *fu_p;
    struct face_g       *fg_p;
    long                index;
};
```

```
typedef double plane_t[4];
struct face_g {
    long                magic;
    plane_t             N;
    point_t             min_pt;
    point_t             max_pt;
    long                index;
};
```

Faceuses represent a side of the surface of a face. Each face will therefore have exactly two faceuses associated with it. The first element in the faceuse structure is a linked list node. This is here so that the shell may keep a list of all the faceuses which are in the shell. The pointer "s_p" is a pointer to the parent shell of the faceuse. The other use of the same face is indicated by the faceuse pointer "fumate_p". The "orientation" member indicates which side of the face is represented by the faceuse. It actually indicates whether the surface normal of the faceuse is the same as the normal stored in the face geometry structure, or whether the normal is reversed. The faceuse is associated with a particular face through the "f_p" pointer. "lu_hd" keeps track of all loops in the face.

```
struct faceuse {
    struct nmg_list     l;
    struct shell        *s_p;
    struct faceuse      *fumate_p;
    char                orientation;
    struct face         *f_p;
    struct nmg_list     lu_hd;
    long                index;
};
```

Shell

The shell represents a set of collected, inter-related (and probably connected) items. The first element is a linked list node to allow groups of shells to be collected into a list. The member "r_p" is a pointer to the parent nmgregion for the shell. The attribute or geometry structure for the shell as a whole are refered to through the pointer "sa_p". The lists "fu_hd", "lu_hd", "eu_hd" are used to keep track of the faceuses, wire loopuses, and wire edgeuses which make up the shell. The pointer "vu_p" points to a single vertexuse when the shell consits of a single vertex. The shell attribute structure "sa_p" is used to store a bounding box for the entire shell.

```
struct shell {
    struct nmg_list     l;
    struct nmgregion    *r_p;
    struct shell_a      *sa_p;
    struct nmg_list     fu_hd;
    struct nmg_list     lu_hd;
    struct nmg_list     eu_hd;
    struct vertexuse    *vu_p;
    long                index;
};
```

Region

Regions are used to keep collections of associated shells within the model space. The nmgregion is included in the model's list of regions through the linked list node at the head of the region structure Because the structure name "region" was already in extensive use through out the BRL-CAD Package when the development of the NMG capability was begun, the structure has been called "nmgregion" instead of simply "region". At some point in the future, the implementation will probably be altered so that all NMG structures begin with the prefix "nmg". The "nmgregion" structure consists of a linked list node "l", a pointer to the parent model "m_p" and a list of shells which make up the region "s_hd". The region attributes structure is used to store a bounding box for the entire region.

```
struct nmgregion {
    struct nmg_list     l;
    struct model        *m_p;
    struct nmgregion_a  *ra_p;
    struct nmg_list     s_hd;
    long                index;
};
```

Model

The model represents the top of the hierarchy for the NMG structures. The list "r_hd" keeps track of all regions or "nmgregions" in the model space. The long integer "maxindex" contains the number of structures which have been allocated in the model. It exists so that arrays can be allocated with one element for each structure in the model.

```
struct model {
    long                magic;
    struct nmg_list     r_hd;
    long                index;
    long                maxindex;
};
```

Variations

There are several differences between this implementation of the radial edge data structures and Weilers implementation [WEIL87]. Some of the differences are a result of the different programming languages used, while others are a result of the different goals of the implementations. While Weiler was implementing an entire modeling system based exclusively upon the NMG structures, within the BRL-CAD Package NMG objects are just one representation of many. The differences are outlined as follows:

- o Wire loops are permitted as members of the shell to simplify the creation and manipulation of loops and faces within the system. This also permits a shell to be comprised of a "point cloud" made up of loops on individual vertex points.
- o Each of the structures faceuse, loopuse, and edgeuse as described by Weiler had attribute substructures which were used for storing information unique to a particular use of the object. This has not proven necessary in this implementation.
- o The shell keeps a list of all wire edges within the shell. Under the system described by Weiler, some wire edges were "discovered" by looking at vertices (vertexes) used in the shell, to find other uses of the same vertex(es) which were children of edgeuses whose parent is the same shell.
- o The linked lists in the Pascal language structures described by Weiler were maintained by directly manipulating "next" and "last" pointers within the structures. In this implementation, all structures which are kept in lists contain a generic "list node"; manipluation of the linked lists is via a standardized set of macros.
- o In this implementation, generic pointers which may reference more than one type of structure are disambiguated by dereferencing the pointer and inspecting resulting "magic number". Because Pascal is a "strongly typed" programming language, Weiler's implementation required that a structure pointer be disambiguated before it was dereferenced. Hence an additional variable in the structure containing the pointer was used to indicate the appropriate interpretation for the generic pointer.
- o All uses of a face must belong to one shell.

NMG Library Interface

The applications programmer is insulated from the details of the NMG radial-edge data structures by a library of functions which perform all of the basic tasks. Each of the routines accepts a valid NMG model or a part of one, and performs an operation, returning a valid model upon completion. The routines within the library can be classified as constructive, destructive, or manipulative. This chapter provides a basic description of the routines in these categories. Within the library, all routines which are intended for use by the applications programmer or user begin with the "nmg_" prefix.

Construction Routines

The basic constructive routines all have the letter "m" (which stands for "make") immediately after the "nmg_" prefix.

- o struct model *nmg_mm() *"Make Model"* creates a new NMG model structure and fills in the appropriate fields. The result is an empty model.
- o struct model *nmg_mmr() *"Make Model, Region"* creates a new model with a call to "nmg_mm" and creates a region within the model. The region is empty.
- o struct shell *nmg_msv(struct nmgregion *r_p) *"Make Shell, Vertex"* creates a new shell consisting of a single vertex in the parameter region. A new vertex is created for the shell.
- o struct nmgregion *nmg_mrsv(struct model *m) *"Make Region, Shell, Vertex"* creates a new region within the existing model, and creates a shell of a single vertex within the region.
- o struct vertexuse *nmg_mvvu(long *upptr) *"Make Vertex, Vertexuse"* exists so that shells, loops and edges can be created on a new vertex. "upptr" points to the parent structure for which the vertex and vertex use are being created. The new vertexuse will reference this structure as its parent.
- o struct vertexuse *nmg_mvu(struct vertex *v, long *upptr) *"Make Vertexuse"* allocates a new vertexuse for an existing vertex. The vertexuse becomes a child of the structure indicated by "upptr".
- o struct edgeuse *nmg_me(struct vertex *v1, struct vertex *v2, struct shell *s) *"Make Edge"* creates a new wire edge in the shell specified. The vertex pointer parameters may either indicate vertexes for the endpoints of the new wire edge, or may be NULL pointers. New vertex structures will be generated for each NULL vertex parameter.
- o struct edgeuse *nmg_meonvu(struct vertexuse *vu) *"Make Edge on existing Vertexuse"* is used to create an edge on a vertex in a loop or shell. The resultant edge has the same vertex at each endpoint.
- o struct edgeuse *nmg_eusplit(struct vertex *v, struct edgeuse *eu) *"Edgeuse Split"* splits an existing edgeuse pair of a wire or "dangling" face-edge by inserting a new vertex.
- o struct edge *nmg_esplit(struct vertex *v, struct edge *e) *"Edge Split"* causes a new vertex to be inserted along an existing edge. If the parameter vertex pointer is NULL, an new vertex is created. The vertex inserted need not lie along the existing edge. See Figure 10.
- o struct edgeuse *nmg_eins(struct edgeuse *eu) *"Edge Insert"* inserts a new, zero length edge between the edge associated with the parameter edgeuse and the edge associated with the edgeuse "previous" to the parameter edgeuse. See Figure 10.
- o struct loopuse *nmg_ml(struct shell *s) *"Make Loop"* takes the largest possible number of contiguous wire edges which form a circuit from the parameter shell and uses them to create a wire loop in the shell.
- o struct loopuse *nmg_mlv(long *magic, struct vertex *v, int orientation) *"Make Loop, Vertex"* creates a new vertex-loop. The loop will be a child of the structure indicated by the magic number pointer parameter, and

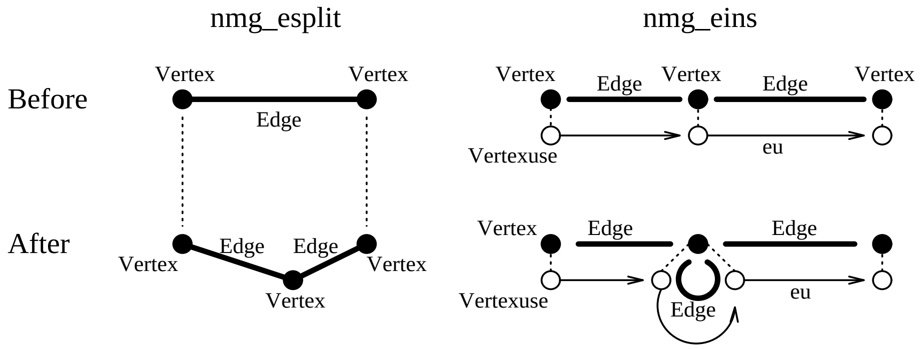

Figure 10 – Edge Operators

will have the specified orientation. If the vertex pointer is NULL, a new vertex is created for the loop.

- struct faceuse *nmg_mf(struct loopuse *lu1) *"Make Face"* generates a new face from the parameter wire loop and its mate.

Convenience Routines

In order to simplify the creation of manifold and non-manifold faces, The following two routines are provided for the application developer.

- struct faceuse *nmg_cface(struct shell *s, struct vertex **vt, int n) *"Create Face"* creates a (non-manifold) face from a list of vertex structure pointers. The face will be a child of the shell indicated in the parameter list. The parameter "vt" is an array of pointers to vertex structures. The length of the array is indicated by the parameter "n". If "vt" is a null pointer, then the face will be created as a polygon on "n" new vertex structures. If "vt" is non-null, a null entry in the list will cause a new vertex to be allocated for that position. The vertexes in the list should be ordered in a clockwise manner for a viewer looking backwards along the normal vector of the desired face.
- struct faceuse *nmg_cmface(struct shell *s, struct vertex **vt[ ], int n) *"Create Manifold Face"* generates a manifold face in the indicated shell. The parameter "vt" is an array of "n" pointers to pointers to vertex structures. If a pointer in vt is a pointer to a null vertex pointer, a new vertex is created for that position. In this way, new vertices can be created conveniently within a user's list of known vertices. The vertices should be listed in "clockwise" order for a viewer looking backwards along the normal of the desired face. In addition to creating the face, the routine will join edges of the new face with dangling edges of other faces in the same shell. This makes it easier for the applications code to generate topologically correct, closed, manifold objects.

Creating Faces

The usual technique for creating a model is to first allocate a model with a call to the "Make Model" routine "nmg_mm". The resultant model does not have any regions within it. Next, a region, and shell consisting of a single vertex are created with a call to "Make Region, Shell, Vertex" or "nmg_mrsv". This lone vertex will be consumed when the first edge or loop is created in the shell.

At this point, there are many separate paths to creating the first face. If a manifold object is the desired goal, then one of the routines "nmg_cface" or "nmg_cmface" should probably be employed. Alternatively, a loop consisting of a single vertex can be created with a call to "nmg_mlv" (Make Loop, Vertex). Then a face can be made with the loop via a call to "nmg_mf". The loop may then be expanded to an edge-loop with calls to "nmg_meonvu" and new vertex points inserted with "nmg_esplit".

Creating a Closed Object

Ordinarily, the applications programmer will be interested in creating simple closed objects. The final topology of the object is already well understood at the time of creation. That is to say, the number of vertices is known, as well as their relationships to various edges and faces. In order to simplify the creation of such objects, the library provides the routine *nmg_cmface* or *"Create Manifold Face."* In addition to providing a simple interface for creating complex faces, the routine also takes care of meshing the edgeuses of the new face with the radial edge structure of pre-existing edges. This is important for the creation of topologically closed objects.

To build a face using *nmg_cmface*, the application programmer must provide a pointer to the parent shell for the face, an array of pointers to pointers to struct vertex, and the length of the array. The reason for the complexity of the array is actually to simplify the job of the applications programmer when making faces for which vertices do not yet exist.

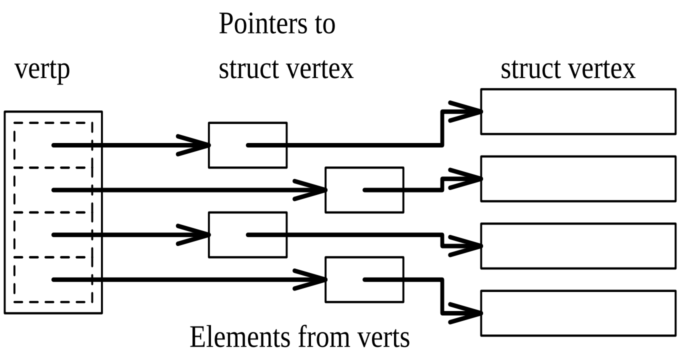

Figure 11 – Conceptual View of Arguments to nmg_cmface

The application keeps an array of pointers to struct vertex (the array *verts* in the example below). This array contains pointers to the vertex structures used in the object being built. A null pointer in this list indicates a vertex structure which does not exist prior to the creation of the face or object. When these pointers are encountered by the nmg_cmface routine, a new vertex will automatically be allocated and a pointer to the new vertex inserted in the list. The mechanism for this will become clear later. Another array (*vertp* in the example below) is used as the argument to *nmg_cmface*. This second array will contain pointers to elements in the first array. For example, take as given an array of pointers to struct vertex called *verts[ ]*, and an array of pointers to pointers to struct vertex called *vertp[ ]*. Now suppose a triangular face is desired, and verts[0], verts[2], and verts[3] are pointers to the vertices at the corners of the face. The result can be conceptually viewed as in Figure 11. The elements of *vertp* should be ordered so that when the new face is viewed from the outside or normal-ward direction, the vertices will be listed in clockwise order. The face can then be created in the shell indicated by the pointer *shell_p* with the following call to nmg_cmface:

```
struct vertex  *verts[4], **vertp[3];
vertp[0] = verts[0];
vertp[1] = verts[2];
vertp[2] = verts[3];
nmg_cmface(shell_p, vertp, 3);
```

Thus to create an NMG object (as is done in the tesselation process), the application first builds the list of vertex structures needed. This consists of pointers to pre-existing vertex structures, and null pointers for new vertex structures. Then each face is constructed in turn.

Destruction Routines

When the model or some part of the model is no longer needed, the structures involved must be deleted. Each of the functions described below "kills" or deletes a structure type, and all its children. Normally, the only destruction routine called directly by the application is "nmg_km", which is called when the model is no longer needed.

- void nmg_km(struct model *m) *"Kill Model"* deletes an entire NMG model.
- void nmg_kr(struct nmgregion *r) *"Kill Region"* deletes an nmgregion and all of its children.
- void nmg_ks(struct shell *s) *"Kill Shell"* removes a shell and all of its children.

- void nmg_kfu(struct faceuse *fu1) *"Kill Faceuse"* deletes a faceuse, its mate and the face which they share, as well as all components which made up the faceuses and face.
- void nmg_klu(struct loopuse *lu1) *"Kill Loopuse"* removes a loopuse, it's mate, and all children.
- void nmg_keu(struct edgeuse *eu) *"Kill Edgeuse"* removes an edgeuse, and its mate from a radial edge, and deletes them. If these were the last radial uses of the edge, then the edge structure and children are deleted as well.
- void nmg_kvu(struct vertexuse *vu) *"Kill Vertexuse"* removes a vertexuse from the list of uses of the vertex and deletes the vertexuse structure. If this was the last use of that vertex, the vertex structure and children are deleted as well.

Manipulation Routines

These routines mold and manipulate the NMG structures.

- void nmg_face_g(struct faceuse *fu, plane_t p) *"Face Geometry construction"* assigns a plane equation to the face of the given faceuse.
- void nmg_face_bb(struct face *f) *"Construct Bounding Box for Face"* by looking at (and computing) the bounding boxes for all the constituent loops.
- void nmg_vertex_gv(struct vertex *v, pointp_t pt) *"Assign vertex Geometry"* to a vertex.
- void nmg_loop_g(struct loop *l) *"Compute Geometry* (bounding box) *for loop"*.
- void nmg_shell_a(struct shell *s) *"Compute Shell Attributes"* (bounding box) for a shell.
- void nmg_region_a(struct nmgregion *r) *"Compute Region Attributes"* computes the bounding box for the nmgregion from all constituent shells.
- void nmg_movevu(struct vertexuse *vu, struct vertex *v) *"Move Vertexuse"* causes a vertexuse to be moved to a new vertex.
- void nmg_unglueedge(struct edgeuse *eu) *"Un-Glue Edge"* removes an edgeuse and its mate from a shared radial edge, and creates a new edge for them to share.
- void nmg_moveeu(struct edgeuse *eudst, struct edgeuse *eusrc) *"Move Edgeuse"* causes an edgeuse "eusrc" and its mate to become uses of a new edge. In the new edge, "eusrc" will be immediately radial to "eudst" in the radial edge list of that edge. If "eusrc" and its mate were the last uses of the old edge, then the old edge structure and it's children are deleted.
- void nmg_moveltof(struct faceuse *fu, struct shell *s) *"Move first wire Loop in shell to an existing Face"*. Not normally called by applications programmer. The first wire loop (loopuse pair) of the specified shell is moved to the given faceuse (and faceuse mate).

- void nmg_jv(struct vertex *v1, struct vertex *v2) *"Join Vertexes"* causes all uses of vertex "v2" to become uses of "v1". Vertex "v2" is deleted.
- void nmg_isect_faces(struct faceuse *fu1, struct faceuse *fu2) *"Intersect Faces"* causes two faces to be intersected and the line of intersection to be identified and inserted.
- void nmg_mesh_faces(struct faceuse *fu1, struct faceuse *fu2) *"Mesh Faces"* performs the topological and geometrical meshing of two faces which share one or more edges of intersection. When complete, opportunities for edge sharing between the two faces will be taken advantage of, and radial edge orientations of the two faces will be appropriate for all shared edges.
- struct model *nmg_find_model(long *magic_p) *"Find Model"* returns the parent model for a given structure. The structure is identified by a pointer to its "magic number".
- struct vertexuse *nmg_find_vu_in_face(point_t pt, struct faceuse *fu, fastf_t tol) searches for a vertex with geometry coordinates within tolerance in a particular face.
- struct nmgregion *nmg_do_bool(struct nmgregion *s1, struct nmgregion *s2, int oper, fastf_t tol) performs a specified boolean operation using the indicated regions as inputs. Produces a region as output.
- int nmg_ck_closed_surf(struct shell *s) *"Check for Closed Surface"* tries to determine if the parameter shell represents a closed object.
- int nmg_manifold_face(struct faceuse *fu) returns a non-zero value if the parameter face represents a part of a closed surface.
- int nmg_demote_eu(struct edgeuse *eu) turns an edgeuse into a pair of loopuses. Each loopuse consists of a single vertexuse. The edgeuse and its mate are deleted, and if they were the last use of their edge, the edge and its children are delete.
- int nmg_demote_lu(struct loopuse *lu) turns a loop into a collection of wire edges. The loopuses of this loop are deleted, as is the loop and its children.
- void nmg_simplify_loop(struct loopuse *lu)
  void nmg_simplify_face(struct faceuse *fu)
  void nmg_simplify_shell(struct shell *s) all serve to simplify the topology of the faces in a shell. Unnecessary edges within a face are deleted, and the adjacent loops are joined together. This service is primarily required to simplify the topology generated as a result of boolean operations.
- void nmg_jl(struct loopuse *lu, struct edgeuse *eu) *"Join Loops"* combines loops which share a common edge. Useful primarily as a support routine for the "Simplify" routines.
- char *nmg_identify_magic(long magic) returns a string that describes the name of the structure type associated with the given magic number. This is most useful in program diagnostics associated with error detection and recovery.

The Tessellators

The job of a tessellator is to convert a given solid primitive into a faceted approximation stored in NMG data structures. There are two aspects to this conversion: establishing the topology of the approximation, and then generating the geometry to associate with the approximate topology.

Conversion of a faceted primitive solid will be exact, with each face, edge, and vertex of the NMG representation one-to-one with a topological element in the original representation. In general, effecting the conversion of a faceted primitive to the NMG representation is direct. Difficulties usually arise only when the original representation did not contain enough topology information to permit direct generation of the NMG topology. For example, the **PolySolid** "bag of polygons" solid contains a collection of faces which together are known to enclose one or more volumes, yet there is absolutely no explicit topology. For this solid, the tessellator routines must rediscover the topology of the original object. This is done by a geometric comparison of each vertex against all the vertices previously encountered in this solid. If the vertex has been previously seen, then a new *use* of the existing vertex is made. If the vertex is unique, then a null pointer is added to the parameter list for **nmg_cmface()** so a new topology vertex can be created. When all the vertices of a face have been processed in this manner, the list of vertices are passed to the routine **nmg_cmface()**, which connects all the vertices together with edges. If an edge existed between two vertices in the new face, then a new use of the existing edge is created, otherwise an entirely new edge is created. Finally, these edges are formed into a loop, and the loop is embedded in a new face.

Error Tolerances

Conversion of curved, implicitly defined primitive solids to a faceted representation will necessarily be inexact. To provide control over the nature and magnitude of the errors that may be introduced by the faceted approximation used in the tessellation, three types of tolerances are passed to the tessellator. The *absolute* tolerance, which limits the maximum permissible difference between any point on the tessellation and the corresponding point on the original solid, is expressed as an absolute distance. In the subroutine interface, this absolute distance is passed as a double precision floating point value, given in millimeters. In the **mged** user interface, this absolute distance is specified in terms of the current working units (millimeters, inches, etc). The absolute tolerance permits users to make absolute statements about the maximum distance error contained in any tessellation. For example, using this mechanism it is possible to ensure that no face deviates from the true surface by more than 2 mm. The *relative* tolerance also limits the maximum error of any point, but is expressed as a fraction between 0.0 and 1.0 of the diameter of the bounding sphere which encloses the original solid. This relative tolerance permits users to make statements about the relative error contained in any tessellation. For example, this could ensure that no face deviates from the true surface by more than five percent of the size of the solid. The *normal* tolerance limits the maximum angular error of the surface normal. This normal tolerance permits users to make statements about the accuracy of the surface normals. For example, this could ensure that the surface normal of all faces do not deviate from the exact surface normals of the corresponding points on the original solid by more than 5 degrees.

The tolerances can be set singly or in any combination. If no tolerances are set, each tessellator module establishes a default minimum tessellation; for example, most tessellators will not approximate a circle with fewer than 6 line segments. If more than one tolerance is specified, then on a solid by solid basis, the most restrictive tolerance is applied. If both an absolute and a relative tolerance are given, then large solids would most likely be tessellated to satisfy the absolute tolerance, while small solids would most likely be tessellated to satisfy the relative tolerance. For example, if an abs=10mm and rel=1%, a large sphere of diameter 1000mm would be subjected to the absolute tolerance, while a small sphere of diameter 10mm would be subjected to the relative tolerance.

Tessellating the Torus

The topology of the torus is a rectangular mesh where the “edges” of

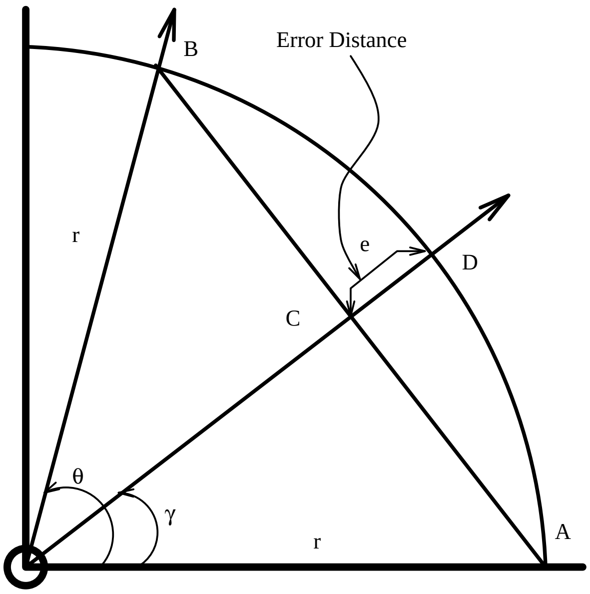

Figure 12 – Calculating Arc/Chord Error

the mesh are connected together, side to side, and top to bottom. This can be visualized as follows: if the torus is cut once and straightened, it becomes a right circular cylinder. If the cylinder is cut longitudinaly and uncurled, it becomes a rectangle. The "length" of this rectangle is the distance around the rim of the torus and is traversed as the angle α varies from 0 to 2π. The "width" is the distance around one cross-section of the torus and is traversed as the angle β varies from 0 to 2π. The torus can also be constructed by constructing a circle in α with radius r_2, and then sweeping that circle out around β with radius r_1.

The torus tessellator must determine the minimum number of facets that can be used to represent the torus while still satisfying the given error tolerances. Fortunately, the two parts of the torus are separable, and both parts are circles, so this problem reduces to determining the fewest number of line segments that can be used to approximate a circle while still meeting the error tolerances.

Consider a triangle inscribed inside a circle of radius r. One vertex of the triangle is at the center of the circle O; the other two touch the circle at points A and B, as shown in Figure 12. The line segment AB subtends an angle

$$\angle AOB = \theta$$

The point of maximum error is C, the midpoint of AB:

$$C = \frac{A+B}{2}$$

$$\angle AOC = \frac{\theta}{2} = \gamma$$

Extending the line OC until it hits the circle results in the point D.

Satisfying the surface normal tolerance is the easiest. If the circle is divided into line segments, the surface normal is exact at the line segment midpoint C, and has the largest error at the endpoints of the line segment: A and B. The surface normal error at the endpoints is $\frac{\theta}{2}$. The relationship between the number of line segments n and the angle θ that each line segment subtends is

$$n = \frac{2\pi}{\theta}$$

Thus, to satisfy a surface normal error tolerance of $ntol$, the error at the endpoints

$$\frac{\theta}{2} \leq ntol$$

Therefore the minimum number of line segments that can be used is

$$nseg = \frac{2\pi}{2ntol} = \frac{\pi}{ntol}$$

The face distance error inherent in the linear approximation of the circle is

$$e = |D - C| = r(1 - cos\frac{\theta}{2})$$

Thus, to meet the face distance error tolerance, a choice of θ must be made so that e satisfies the relation

$$e \leq dtol$$

The maximum value of θ which can be used is

$$\theta = 2cos^{-1}(1 - \frac{dtol}{r})$$

and the minimum number of line segments that must be used is:

$$nseg = \frac{2\pi}{\theta} = \frac{\pi}{cos^{-1}(1 - \frac{dtol}{r})}$$

To efficiently satisfy the maximum surface error tolerance, it is necessary to find the minimum number of line segments that must be used in each of the "length" and "width" directions. The appropriate radius is substituted into the formula for $nseg$ to determine $nlen$ (the number of segments needed in the length direction), and to determine nw (the number of segments needed in the width direction). The largest of either of $nlen$ or nw or the number of segments required to satisfy the surface normal tolerance is used.

All the vertices on the surface of the torus can be generated by varying len from 0 to $nlen$ while also varying w from 0 to nw, and computing:

$$\alpha = w\frac{2\pi}{nw}$$

$$\beta = len\frac{2\pi}{nlen}$$

$$R = Acos(\beta) + Bsin(\beta)$$

$$P = V + R + r_2\frac{R}{|R|}cos(\alpha) + Hsin(\alpha)$$

where

V	vertex point at the center of the torus
A, B	perp. vectors, lie in the plane of the torus, define "length"
G, H	perp. vectors, define "width" direction
H	normal to plane of the torus
P	surface point
r_2	radius of torus around the rim

```
        E
       /\
      /  \
    G/____\H
    /\    /\
   /  \  /  \
  /____\/____\
 D      I     F
```

Figure 13 – Initial Octahedron Face

Tessellating the Ellipsoid

An ellipsoid is defined by a vertex point V at the center and three mutually perpendicular vectors A, B, and C which define the eccentricities. Through an affine transformation, the ellipsoid can be mapped to a unit sphere located at the origin. The algorithm begins by approximating the sphere by an octahedron, with the six vertices at $V \pm A$, $V \pm B$, and $V \pm C$. Each face of the octahedron is a triangle $\triangle DEF$, as in Figure 13.

Points DEF line on the surface of the unit sphere. Pick the points GHI as the midpoints of the three edges of ABC. If each of these points GHI are viewed as vectors from the origin in the direction of the surface of the unit sphere, then re-normalizing the vectors to have unit length will cause the points to lie on the surface of the unit sphere. Consider each of the four new triangles $\triangle DGI$, $\triangle GHI$, $\triangle IHF$, and $\triangle GEH$ recursively until the tolerance is satisfied.

It is tempting to consider an adaptive subdivision algorithm, so that when the magnitudes of A, B, and C are not equal, the areas of higher curvature could be tessellated more finely. Unfortunately, it is not possible to use different levels of subdivision without introducing "cracks" into the tessellation. Consider the case where triangle $\triangle GEH$ needs further subdivision, but triangles $\triangle DGI$, $\triangle GHI$, and $\triangle IHF$ do not, as in Figure 14. The problem here arises because the edge GH in $\triangle GHI$ will no longer match up with edge GL in $\triangle GJL$ and edge LH in $\triangle LKH$, because point L will have been normalized out to meet the unit sphere. While cracks could be prevented with by splitting $\triangle GHI$ into $\triangle GLI$ and $\triangle LHI$, this would produce an irregularity in the topology of the tessellation that would make a non-recursive formulation significantly more difficult.

Let

$$r = max(|A|, |B|, |C|)$$

Then, the distance tolerance $dtol$ can be expressed as a maximum angular (normal) tolerance as before:

$$\theta = 2cos^{-1}(1 - \frac{dtol}{r})$$

```
             E
            /\
           /  \
         J/____\K
         /\    /\
        /  \  /  \
      G/____\/____\H
      /\     L    /\
     /  \        /  \
    /    \      /    \
   /      \    /      \
  /        \  /        \
 /__________\/__________\
D            I           F
```

Figure 14 – Partially Subdivided Octahedron Face

The final tolerance to use is the more strict of the surface normal tolerance $ntol$ and the distance tolerance:

$$tol = min(ntol, \theta)$$

Thus, the number of triangles to be used around any one circumference of the ellipsoid will be

$$nseg = \frac{\pi}{ntol}$$

However, because the initial approximation to the ellipsoid is an octahedron, the number of segments must be a multiple of four.

The coordinates of all the vertices of the tessellation of a unit sphere with $nseg$ triangles are computed. Each of these coordinates is then transformed back into the coordinate system of the ellipsoid.

Evaluating Booleans

To evaluate a boolean combination of two tessellated solids, three distinct sub-operations must be performed. First, all elements of the two shells must be intersected. Second, every element in the two shells must be classified as **in**, **on**, or **out**. Finally, all undesired elements are eliminated.

Intersection Operations

The first step in the boolean operation process is the intersection and cutting of the two shells with respect to each other. This process is summarized in Figure 15.

The process of comparing the face bounding boxes involves a check to see if the bounding boxes overlap along the line of intersection, not just whether they overlap at all. This is important for reducing the number of face intersections performed.

When the bounding boxes of two faces overlap, they must be intersected with each other. The plane equations of the two faces are compared for equality. If the two faces are coplanar, they must have their loops intersected using a two dimensional polygon clipping approach. If the two faces are not coplanar, they must be intersected so that the faces will share an edge at the intersection. A list of all vertexuses or points which are on the line of intersection between the two planes is first generated. To do this, each edge of face A is intersected with the plane of face B and each edge of face B is intersected with the plane of face A.

This intersection process takes the following form: If an endpoint of the edge is either topologically or geometrically in the plane of the other face, that vertex is registered in the list of points on the line of intersection. If the span of the edge crosses the plane of the other face, then the edge is divided into two edges. The new vertex which divides the edge lies on

```
if bounding boxes of shell A and shell B overlap
  for each face in shell A,
    if bounding box of face A overlaps bound box of shell B
      for each face in shell B
        if bounding boxes of face A and face B overlap
          intersect edges of face A with plane of face B
          intersect edges of face B with plane of face A
          insert new topology & perform meshing
  for each wire edge in shell A,
    if wire edge A overlaps bounding box of shell B
      for each face in shell B
        if edge A intersects face B
          if necessary split edge at plane,
          insert vertex at plane intersection into face
      for each wire edge in shell B
          if wire edges intersect
            create any needed verticies in both wire edges
```

Figure 15 – Shell Intersection Procedure.

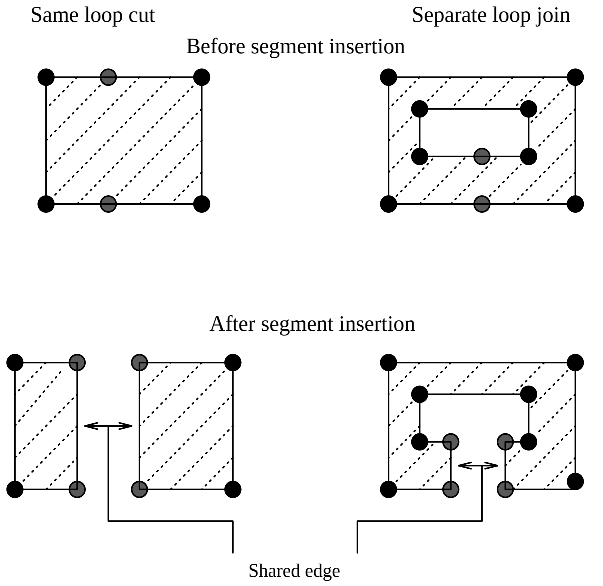

Figure 16 – Segment Insertion: Both Points in Face

the line of intersection and is added to the list of points along the line of intersection.

After each face has been intersected with the plane of the other face, the resulting list of points of intersection is sorted geometrically along the line of intersection. The list is then used to determine which segments of the line of intersection are shared between both faces. Such segments must be added to each face. For each face, a segment to be added will fall into one of three categories: (1) both endpoints are in the face, (2) one endpoint is in the face, or (3) no endpoints are in the face. In the first case, a check is made to make certain that the edge does not already exist. If it does not, then each of the two endpoints are examined. If they are part of the same loop of the face, then that loop is divided into two separate loops, which share a common (new) edge. If the vertices belong to different loops, then an edge which connects the two loops is created, and the two loops are joined into a single loop. Examples of each of these cases can be seen in Figure 16.

When only one of the endpoints exists in the face, the loop which contains the existing vertex is extended to include the new point. This can be seen in Figure 17. Finally, when neither of the segment endpoints exists in the face, a new interior loop must be created in the face as in Figure 18.

When all of the appropriate topology has been inserted into the faces, the topology of the two faces is connected so that edges and vertices which can be shared between the faces are indeed shared. This process consists

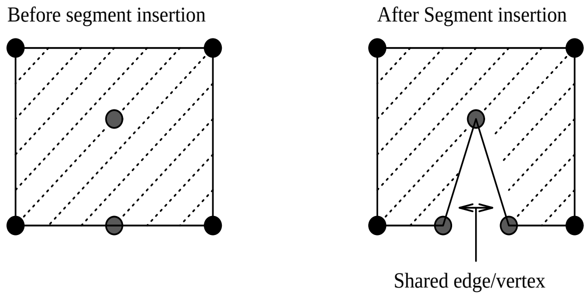

Figure 17 – Segment Insertion: One Point in Face

mainly of combining vertexuses onto a common vertex, and arranging the radial-edge orientation of edgeuses about a shared edge.

Wire edges of the shell must also be intersected with the other shell. Wire edges exist either as part of a wire loop, or as an individual wire edge. Both types are processed in the same manner. Each edge is compared to the bounding box of the other shell. Where there is overlap, the edge is intersected with each face and wire of the other shell. If the edge intersects any of these, the edge is split if necessary, and the point of intersection is topologically linked to the other face or wire.

Object Classification

When all of the intersections have been performed, every object in each shell must be classified with respect to the other shell. Each face, loop, edge, and vertex must be classified as being inside, on the surface of (referred to here as "on"), or outside of the other shell. Here the topology becomes helpful. This process is easier if each shell is classified in turn against the other shell. For notational clarity, we refer to the classification of an object in shell A with respect to shell B.

The classification of all structures is stored in a single table. The "index" fields in the NMG structures serve as offsets into the table. Once the classification is complete, an individual structure's classification can be

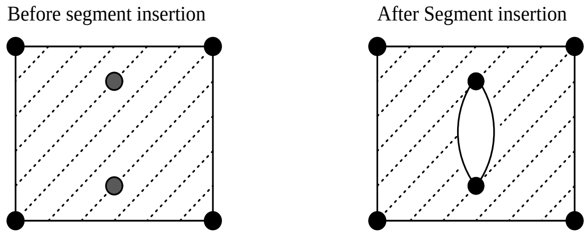

Figure 18 – Segment Insertion: No Points in Face

quickly gotten by using the "index" field from the structure as the offset into the table of classifications.

Vertex Classification

If by, referring to the topology, it can be determined that the vertex has previously been classified against shell B, then obviously any vertexuse of that vertex which is in shell A shares the same classification. If it has not previously been classified, it is classified now.

By looking at the other vertexuses of a vertex, it is possible to determine if there is a use of the vertex in the topology of shell B. If such a use exists, then the vertex, and the current vertexuse are classed as being "on" shell B.

If the topology does not indicate that the vertex is "on" shell B, the geometry must be used to determine if the vertex is "inside" or "outside" of shell B. First of all it should be obvious that all vertices which lie outside of the bounding box of shell B are easily classified as being outside of shell B. Should this fail to classify the vertex, a raytracing approach is employed.

A single ray is "fired" from the vertex along an arbitrary line and is intersected with all of the faces in shell B. Typically, this line will be along one of the major axis directions, although it may be useful to send the ray in some other, non-aligned direction to reduce the probability of hitting the edge of a face or hitting the plane of a face edge-on. The number of *Manifold* faces which the ray encounters before leaving the bounding box of shell B is counted. If the number of crossings is odd, then the vertex is classified as "inside" of shell B. While this appears simple at first glance, there are several problems involved in the raytracing approach which are worth noting: (1) identifying a face as either manifold or non-manifold, (2) recognizing an actual hit on a face, and (3) coping with a ray hit on an edge or a vertex.

Manifold Faces

If a face has any dangling edges, it can be quickly classed as being non-manifold. This is a relatively quick check. If for each edge of the face, there exists an odd number of other face-loops of manifold faces of the same shell adjacent to the same edge, then the face is manifold. This is a tedious check which requires an exhaustive analysis of most, if not all of the faces of the shell to determine if one face is manifold. While this is important for some cases, most of the time it is not necessary to perform a full manifold check of the radial faces. A check for dangling edges of radial faces is sufficient.

Intersecting a Ray and Face

Since faces consist of both interior loops (holes in the face) and exterior loops, it is important to be sure that a ray which hits the plane of a face actually hits inside the surface area of the face as well. The in/out status

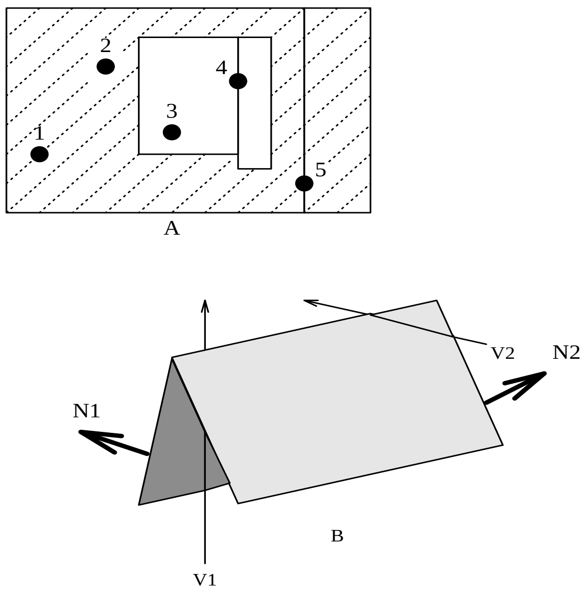

Figure 19 – Raytracing Hit-Points

for the hit-point is determined by the loop which has the sub-element (edge or vertex) closest to the hit-point.

If the hit-point is closest to an exterior loop of the face, then (a) the ray hits the face if the point is inside the loop, as in hit-point 1 in Figure 19A, and (b) the ray misses the face if the point is outside the loop. Likewise, if the hit-point is closest to an interior loop, then (c) the ray misses the face if the hit-point is inside the loop, as in hit-point 3 in Figure 19A, and (d) the ray hits the face if the hit-point is outside the loop, as in hit-point 2 in Figure 19A.

Intersecting a Ray and Edge Or Vertex

When the hit-point of the ray lies on an edge or vertex of a face, further logic must be used to determine whether or not a hit actually occurs. If the ray hits an edge of a loop in a face, and there is another loop of the face of the same type (interior/exterior) adjacent on the edge, the hit is registered as if the ray had intersected the interior of the loop. For example, hit-point 4 in Figure 19A is not a hit on the face because the edge is shared between two interior loops of the same face. On the other hand hit-point 5 in Figure 19A is a hit on the face because the edge being hit is shared between two exterior loops of the face.

If the edge is actually a boundary of the face, then the ray must be compared to the surface normals of the faces involved. For example, the ray V1 in Figure 19B scores a single hit as it encounters the edge between the two faces. Ray V2 does not score a hit. It is important to note that only one hit may occur for the pair of faces, since it was the edge between them which was encountered. The situation becomes slightly more complex when there are more than two faces of the shell sharing the edge. In this case, the hit/miss status for each pair of faces is determined at the time the edge is first encountered, and all faces are marked as having been processed.

Edge Classification

Once all the vertex structures in shell A have been classified against shell B, it is possible to classify the edges. First cut classification of edges is done by referring to the classification of the endpoints. If one or more of the endpoints is not "on" shell B, the edge takes it's classification from that vertex. If both endpoints are classified as "on" shell B, the mid-point of the line segment is computed, and classified using the raytracing technique described in the previous section. An edge with endpoints inside and outside shell B indicates an error in the intersection process. Table 1 summarizes the classification of edges.

Endpoint classifications	Edge classification
both out	out
out/on	out
both on	use mid-point ray
in/on	in
both in	in
in/out	intersection error

Table 1 – Edge Classification by Endpoint Classification

Loop Classification

A loop of a single vertex inherits the vertex's classification. A loop of edges which contains an edge which is not "on" shell B inherits that classification. For example, if a loop contains an edge classified "inside" shell B, then the loop is classified "inside" shell B. A loop with edges both inside and outside shell B indicates an error in the intersection process. If all edges of a wire loop are classified as "on" shell B, the loop is classified as "on" shell B. There are other conditions required for a face loop to be classified as being "on" shell B. First, there must exist a loop in the topology of shell B which has exactly the same set of edges. Secondly, the loop must be classified as being either "shared" or "anti-shared". A "shared" face loop not only has a counterpart in the other shell, but the normals of the faces of which the loops are a part of are pointing in the same direction. A loop is "anti-shared" when the surface normals of the parent faces point in opposite directions.

Face Classification

Faces are not classified except to the extent to which the loops which make up the face are classified.

Boolean Evaluation

After all the topological elements in objects A and B have been classified, the boolean evaluation simply becomes a task of deciding which elements to retain, and which to destroy. Every element has been classified with respect to both objects A and B, and was assigned one of eight combined classifications. The object from which the element originally came from is always given an **on** classification. Elements from A are classified as one of **onAinB, onAonBshared, onAonBanti-shared, onAoutB,** while elements from B are classified as one of **inAonB, onAonBshared, onAonBanti-shared,** or **outAonB**. All of these possibilities occur in the two example object pairs (four blocks viewed end-on) in Figure 20; the important face-loop classifications are appropriately labeled.

For the two blocks on the left of Figure 20, the bottom center face-loop exists in both objects A and B, and the orientation (outward surface normal) of the face-loop in both objects is the same (pointing towards the bottom of the page), so the face-loop is classified **onAonBshared**. For the two blocks on the right of Figure 20, the middle face-loop also exists in both objects A and B, but the orientation of the two face-loops are opposite, so this face-loop is classified **onAonBanti-shared**.

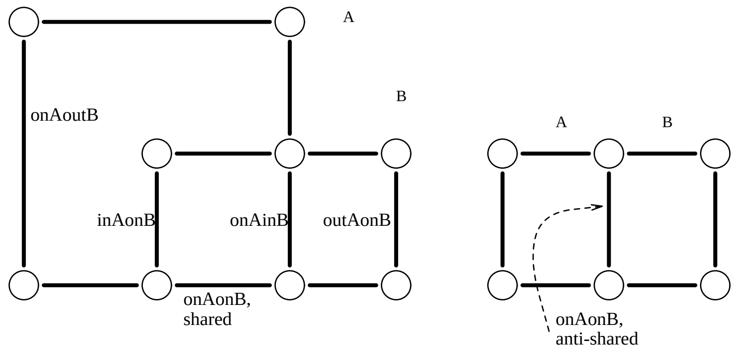

Figure 20 – Classification of Example Objects

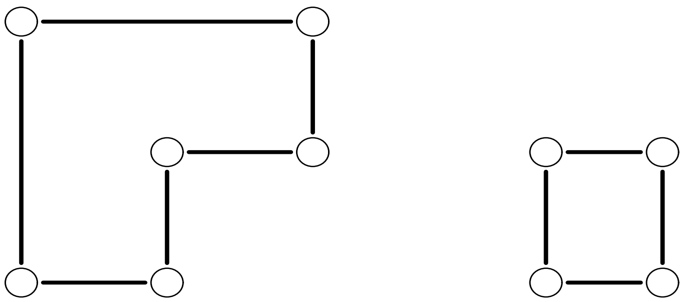

Figure 21 – Subtraction Performed on Example Objects

A	B	A − B	A ∪ B	A ∩ B
on	in	kill	kill	retain
on	on shared	kill	retain	retain
on	on anti-shared	retain	kill	retain
on	out	retain	retain	kill
in	on	retain+flip	kill	retain
on shared	on	kill	kill	kill
on anti-shared	on	kill	kill	kill
out	on	kill	retain	kill

Table 2 – Boolean Evaluation Decision Table

Because there are only eight possible classifications, the appropriate action for the boolean evaluation algorithm can be easily tabulated. These actions are found in Table 2. Indeed, this exact same table exists in the source code for the boolean evaluator in file **nmg_eval.c**. Placing these actions into a table permits a complete separation of *policy* and *mechanism* in the implementation. The policy is encoded in the entries of the table, and the mechanism is embodied in the code of the subroutine. This has two main benefits. First, code to perform any one action exists in only one place, ensuring consistent treatment of all cases. Second, this offers the potential for adding more boolean operations at a later date, simply by supplementing the table with another column.

Consider first the case of the subtraction operation, A minus B. The intent is to retain every part of A that is **in** only A, to kill every part of A that is also **in** B, and to kill every part of B. The policy just stated is implemented by the tabulated rules for processing all the topological elements. The entry for an element which has classification **on** A and **out** B has a table entry of "retain" (denoted more suscinctly as **onAoutB**=retain). This rule preserves the main body of A. **onAinB**=kill and **onAonBshared**=kill because these elements are **in** B, and therefore represent a portion to be subtracted out. **outAonB**=kill to eliminate unused parts of

B. **onAonBanti-shared** elements are retained, because they are on the surface of A and on the surface of B, where the two surfaces touch. To implement subtraction, this case has to be defined as either (a) shaving an infinitesimally thin layer off of the surface of A, or (b) a no-op, where the A surface is retained unmodified, and the B surface is killed. Because choice (a) would modify the volume of the resulting solid, it can not be used; instead, choice (b) has been selected. This policy is implemented in these three table entries: **onAonBanti-shared**=retain, **onBonAshared**=kill, and **onBonAanti-shared**=kill. Finally, elements which have been classified as **inAonB** are listed in the table as "retain+flip". The surface has to be retained because it will become part of the new boundary between A and the outside world. However, the existing surface normal points *into* solid A, reflecting the fact that this surface was originally part of the exterior of solid B, so the surface normal must be flipped. The results of performing a subtraction on the two sample objects is illustrated in Figure 21.

While the discussion of the table entries has been made in terms of surfaces defined by face-loops, the same reasoning and actions apply to the inferior topological elements: wire-loops, loop-edges, wire-edges, edge-vertices, and lone-vertices. The overall strategy is to process all the topological elements of object A, and then to process all the elements of object B. The meat of the algorithm exists in internal subroutine **nmg_eval_shell()**, which starts by processing the loops in each face. Any loopuse which has a classification that maps to an action of "kill" is demoted into a collection of wire edges using **nmg_demote_lu()**. The loop is demoted rather than killed so that edges and vertices that are shared in common with both objects can be properly considered later in the subroutine. If none of the loops in the face are retained, then the faceuse is killed using **nmg_kfu()**. If some loops in the face are retained, then the faceuse is retained; if the faceuse came from object B then it is moved to object A, the faceuse's mate is also moved to object A, and the face normal is flipped. The algorithm then proceeds down from faces to processing wire-loops. If the loopuse has a classification that maps to an action of "kill" then it is demoted into a collection of wire edges, otherwise it is retained, and moved into object A if necessary. Next, wire-edges are processed. If the edge is marked 'kill" it is demoted into vertices using **nmg_demote_eu()**, otherwise it is retained, and moved into object A if necessary. Finally, lone-vertices are processed. Vertices marked "kill" are disposed of using **nmg_kvu()**. Because vertices are 0-manifolds there is no lower topological element to demote them to.

Consider next the case of the union operation, A ∪ B. The intent here is to retain all elements that are on the exterior of either A or B, while eliminating any interior structure or redundant elements. More precisely, for solid modeling, the formula for union is interpreted as

$$A \cup B \;:=\; (A - B) + (B - A)$$

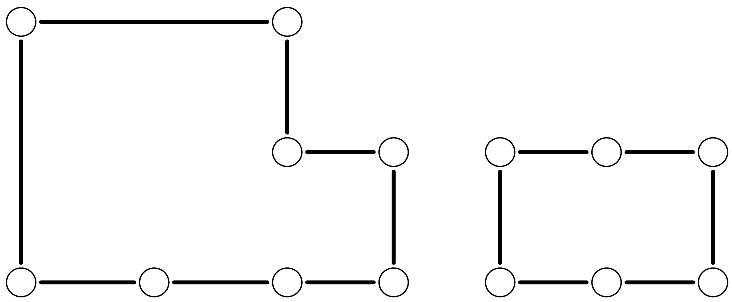

Figure 22 – Union Performed on Example Objects

where the + operation is a simple combination, or sum, operation. The elements that are on the exterior of exactly one of either A or B are classified as **onAoutB** and **outAonB** and are retained. Elements that are on the exterior of both A and B appear twice, first as **onAonBshared** which is retained, and again as **onBonAshared**, which is killed (to avoid having the element become duplicated in the result). Interior structure is found in all elements classified as **onAinB**, **inAonB**, plus any anti-shared faces **onAonBanti-shared** and **onBonAanti-shared**. All elements with these classifications are killed. The result of performing the union operation on the two example objects is shown in Figure 22.

Finally, consider the case of the intersection operation, A ∩ B, as shown in Figure 23. The intersection operation retains all elements that are simultaneously part of both A and B, while discarding the excess. Clearly, elements classified **onAonBshared** and **onAonBanti-shared** are elements common to the two objects and should be retained; elements classified **onBonAshared** and **onBonAanti-shared** need to be killed to prevent duplication. The elements exterior to one of the objects are classified **onAoutB** and **outAonB** and should also be killed. Previously interior structure **onAinB** and **inAonB** should also be retained, as these elements will form the new boundary when the non-common elements have been removed.

The technique just described for evaluating a boolean formula is simple to program and debug, works reliably, and is easy to understand. Storing all the policy decisions in a table makes the algorithm very straightforward.

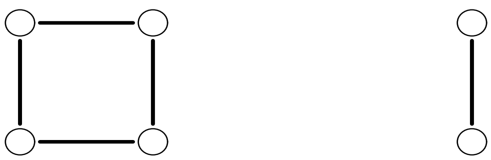

Figure 23 – Intersection Performed on Example Objects

User Interface

The majority of a BRL-CAD user's contact with the new non-manifold geometry capability will be through the Multi-Device Graphics Editor (**mged**) [APPL88]. This is the program that is used to rapidly view existing geometry in wireframe form, to create new geometry and update existing shapes, to select viewpoints for ray-tracing runs, to select keyframes for animation sequences, and to preview animation sequences in wireframe form.

At any time in an **mged** session, the user may add one or more objects to the active model space, using the command

`mged>` *e object*

If the viewing cube is suitably positioned, the newly added subtrees become visible on the display. The normal mode of operation is for users to work with wireframe displays of the unevaluated primitive solids. These wireframes can be created from the database very rapidly.

On demand, the user can request the calculation of approximate boundary wireframes that account for all of the boolean operations specified along the arcs of the directed acyclic graph in the database. The evaluation of the approximation is performed by tessellating each of the primitive solids into an NMG object meeting the current tolerance, and combining them according to the indicated boolean operations. Each edge and vertex in the resulting NMG object are placed in a **struct vlist** chain (by subroutine **nmg_r_to_vlist()**) and drawn on the **mged** display. This operation is invoked with the command

`mged>` *E object*

or

`mged>` *ev -w object*

Evaluating the surface takes somewhat longer than creating wireframes of the unevaluated primitive solids, so it is not used by default for viewing large assemblies. However, it is quite reasonable to evaluate the surface whenever the design has reached a new plateau, or when a detailed examination of a complex part is required. When viewing objects of modest complexity at loose tolerances, the evaluation process is quite rapid.

Note that the evaluated boundary wireframes are not stored in the database, and are primarily intended as a visualization aid for the designer.

Polygons

On those hardware platforms where polygon drawing capabilities exist, it is possible to have a flat-shaded polygonal rendering of database objects drawn. This operation is invoked with the *evaluate* command

`mged>` *ev object*

Once the polygonal rendering of the object is on the screen, it can be rotated in real time. This capability gives the designer the opportunity to more fully appreciate the complex shape which has been created, and to judge whether the evaluated shape matches the intended design.

On hardware platforms which have hardware support for rendering lit and shaded polygons with multiple light sources (such as the Silicon Graphics 4D workstations), it is possible to activate the hardware lighting model. This provides a much more realistic rendition of the evaluated objects, and also gives clues about face normals.

When further used in conjunction with hardware clipping planes, the evaluated solids can provide significant new insight for the designer.

Surface Normals

As an aid to the user, an option to the *ev* command exists to add surface normal vectors to the evaluated surface polygon display. The outward pointing surface normals are drawn as single vectors from the centroid of each polygon. They resemble "whiskers" on the polygons. This option is invoked with the command

`mged>` *ev -n object*

If the polygon rendering of the object shows all the faces, but no surface normal vectors can be seen, then this suggests that either an error exists in the database entry for this object, or an internal software error in **librt** has been encountered. In either of these cases, the vectors should be pointing inside the solid object. This can be easily verified by using the hardware clipping planes to permit the inside of the solid to be seen. If the vectors are on the inside, then the surface normals have become reversed.

Tolerances

As discussed the in section on tessellation, there are three types of tolerances that are user selectable. The absolute tolerance ensures that no point on the polygonal approximation will be further away from the true surface than the tolerance value. The relative tolerance ensures that no point on the polygonal approximation will have a distance from the true surface which is greater than the given fraction of the primitive object's size. The surface normal tolerance ensures that no face normal at any point on the polygonal approximation will differ from the true surface normal at the corresponding point on the true surface by more than the indicated angular error. If multiple tolerances are given, the tesselator is required to satisfy all of the tolerances. This enables the user of the approximation to make firm statements about the upper bound on the tessellation error produced.

When **mged** is first started, a relative tolerance of one percent is the default tolerance. At any time the tolerance settings can be examined, using the *tol* command. Thus, immediately after starting **mged**, the *tol* command would show:

```
mged> tol
Current tolerance settings are:
        abs None
        rel 0.01 (1%)
        norm None
```

The absolute tolerance specification is given in the current working units, and is converted internally into millimeters. For example, adding an absolute tolerance of three millimeters might be done like this:

```
mged> tol abs 3
mged> tol
Current tolerance settings are:
        abs 3 MILLIMETERS
        rel 0.01 (1%)
        norm None
mged> units inches
mged> tol
Current tolerance settings are:
        abs 0.11811 INCHES
        rel 0.01 (1%)
        norm None
```

The absolute tolerance can be disabled by setting the tolerance to zero (or a negative number). Providing a tessellation with zero error is impossible; in general, to create a tessellation of a curved object with zero error would require an infinite number of planar faces. Therefore, zero tolerance can be used as a flag to turn off the absolute tolerance requirement.

The relative tolerance specification is given as a fraction between zero and one.

```
mged> tol rel .001
mged> tol
Current tolerance settings are:
        abs 3 MILLIMETERS
        rel 0.001 (0.1%)
        norm None
mged> tol rel 0
mged> tol
Current tolerance settings are:
        abs 3 MILLIMETERS
        rel None
        norm None
```

The surface normal tolerance is given as a positive angle in degrees between zero and 90. Specifying a normal tolerance of zero is used to disable the normal tolerance requirement. The angular tolerance is echoed back both in fractional degrees, and as degrees minutes and seconds of arc, as can be seen in the second example below.

```
mged> tol norm 2
mged> tol
Current tolerance settings are:
        abs 3 MILLIMETERS
        rel 0.01 (1%)
        norm 2 degrees (2 deg 0 min 0 sec)
mged> tol norm 0.5
mged> tol
Current tolerance settings are:
        abs 3 MILLIMETERS
        rel 0.01 (1%)
        norm 0.5 degrees (0 deg 30 min 0 sec)
```

If a model included parts that had been created from the boolean combination of both very large and very small primitives, where the small primitives have a size within a few orders of magnitude of the desired absolute tolerance, it may be desirable to specify both an absolute and a relative tolerance. For the large objects, the absolute tolerance would ensure that the tessellation produced enough facets to represent the surface to the desired degree of accuracy, whereas the relative tolerance would probably have generated significantly fewer facets. For the small objects, the absolute tolerance would probably not generate very many facets, but the presence of a relative tolerance would ensure that the tessellation produced a reasonable rendition of the desired shape.

The surface normal tolerance will also limit the error in the tessellation. However, the number of faces in the tessellation can grow rapidly as the tolerance is tightened. For example, consider tessellating a single torus. If no normal tolerance is given, the torus is approximated by 36 (6×6) facets, and a normal tolerance of one degree yields 32400 facets. Other values are listed in Table 3. While specifying a tight tolerance on the surface normal will produce tessellations with a pleasingly smooth surface, the large number of facets that result can consume substantial amount of memory. If all tolerances are disabled (including the default one percent relative tolerance), then each tessellator will choose an object-specific minimum number of facets needed to produce a reconizable rendition of the primitive solid.

Tolerance	Facets
none	36
10	326
5	1296
2	8100
1	32400
0.5	129600

Table 3 – Relationship Between Normal Tolerance and Facets for Torus

Visualizing Boolean Evaluation

During the development of the boolean evaluation software, several graphical displays were created to assist in the debugging process. By default, this feature is not active, but setting a run-time debugging flag allows any user to enable it.

The first display shows the process of intersecting two faces together. The outline of all the loops in the first face are drawn. Performing the intersection operation can result in the addition of new edges. Adding new edges may also result in the cutting or joining of loops in the face. After the intersection is complete, the outline of all the loops in the face are drawn again so that the changes are visible. The display is updated each time.

The second display shows the process of evaluating the boolean formula on the fully intersected and cut objects. First all the faces are evaluated. Faces to be retained do not change, but faces that are not part of the result are demoted to wire loops. This is indicated by the color of the edges changing to yellow. Wire loops that are not retained are demoted to wire edges. Wire edges that are not retained are demoted to lone vertices and disappear from the display. Lone vertices that are not retained are killed, and vanish from the display.

Even for the boolean combination of two relatively simple overlapping solids, quite a few face intersections can be computed. However, on typical workstations it is not uncommon to see ten intersections computed and the diagnostic wireframes drawn to the screen each second. This high speed animated review of the intersection and boolean evaluation operations is very valuable for teaching users how the boolean operations work. It can also be beneficial when a user wants to understand the exact process that created a final NMG shape. This capability also served as a very valuable debugging tool, often permitting an intuitive grasp of programming errors to be rapidly acquired – pouring through formatted dumps of thousands of NMG data structures was so difficult that it was reserved as the technique of last resort.

Creating an NMG Database Object

So far, the primary motivation for computing an approximate surface representation for an object has been to drive some "post processing" application. For example, the *ev* command computes an approximate surface representation for the purpose of creating wireframes and shaded polygon renderings for the user to view inside the **mged** editor. Approximate surface representations are also used to provide appropriate kinds of input files for existing analysis applications. But, the facetization and boolean combination mechanism that has been constructed is completely general, and can be used for other purposes also.

There may be circumstances when a designer may wish to take a collection of primitive solids, tessellate and combine them into some faceted shape, and then store that faceted shape as the finished design. If the object of the design task is to create a faceted object, then this can be a powerful way of accomplishing that task. It can be accomplished with the *facetize* command:

```
mged> facetize newsol oldsol
mged> facetize newreg oldreg
```

The *facetize* command takes either a single pre-existing solid, or a single pre-existing combination of solids (such as a group or region), tessellates them according to the current tolerance settings, and creates a new solid in the database that is the facetized result, represented using the NMG data structures. This newly created solid has exactly the same standing as any other primitive solid: it can be ray-traced, instanced, or combined with other solids to create new shapes. However, it is important to note that no record is made of how this new solid was formed. Thus, when the parameters of one of the original solids inside the original region *oldreg* is modified, this does *not* result in any changes being propagated to the facetized version *newreg*. If this effect is desired, it is necessary to delete *newreg* and re-execute the *facetize* command, e.g.:

```
mged> kill newreg
mged> facetize newreg oldreg
```

The same procedure must be followed if a different tolerance for the facetization is desired.

Applications

The capability to produce an explicit approximate representation of the surface of any object stored in the geometry database exists as a general capability. Any application program that links to the library **librt** can make use of this capability, at any time in the analysis process. Furthermore, use of this capability can be simultaneously intermixed with other forms of interrogation supported by the library, so that an application might perform some operations using the approximate surface description, and other operations using ray-tracing. The application will have no knowledge of the underlying primitives used to describe the objects in the database, nor will the application be aware that the library creates the surface description by extracting the objects from the geometry database, tessellating them into NMG solids, and then combining them via boolean formulas.

Image Rendering

Prior to the existence of this polygon rendering capability, all renderings of the geometry databases had to be performed using an optical simulation program based on ray-tracing. While ray-tracing can produce some very beautiful images, it can require a non-trivial amount of processing time to create the images. For the purpose of visualizing the shape of an object, an image with much lower quality than those produced by ray-tracing would be entirely acceptable, if they could be produced in significantly less time.

The most immediate application of the explicit surface representation is for visualization purposes: making an optical image or *rendering* of objects in the geometric database. These images can be useful for a variety of applications. Within the geometry editor **mged**, a high-speed rendering of a polygonal approximation of a shape is very useful for inspecting the model. This permits the user to verify that the design is as it should be. It is also a very powerful technique for conveying information about the design to others.

If the display hardware is fast enough, this capability could provide an improved interface for many geometry editing operations. Presently, modifications to solids are performed using the wireframe display. The modifications are entered either numerically (by entering specific parameters), or interactively (by modifying key parameters through cursor manipulations on the screen). In both cases, it can occasionally be difficult to judge by eye whether the shape created is the desired one. Being able to edit the object in a (seemingly) solid, rendered form would greatly assist in this task.

Along the spectrum of rendering quality, there is a point midway between the low quality of simple hardware polygon rendering, and the high quality of ray-traced images. This intermediate level of quality can be obtained using software-based polygon rendering algorithms. These algorithms tend to run faster than rendering algorithms based on ray-tracing. Especially when creating animation sequences that require the rendering of hundreds or thousands of images, having the ability to make this quality *vs.* speed tradeoff can be a real boon. This opportunity now exists. An approximate surface description, tessellated with appropriate tolerances, can be converted into polygonal form and passed to existing polygonal rendering software.

Thermal Predictions

In the design of vehicles, it is very useful to be able to make predictions about the thermal behavior of the vehicle before the prototype is constructed. This is important for ensuring passenger comfort and proper cooling of temperature sensitive components. In military applications, the patterns of heat radiation are also quite important. If simulation of the thermal behavior of the vehicle reveals heat distribution that is not consistent with the design criteria, it is a simple matter to modify the vehicle geometry or substitute different construction materials in an attempt to improve the situation. Re-running the thermal simulation will assess the effect of these changes.

To simulate the thermal behavior of the vehicle, it is necessary to calculate a complete heat budget for the entire vehicle, in addition to obtaining the geometry and material property information. The heat budget must account for all the internal sources of heat such as engines, bearings, and road wheels, and radiators of heat such as cooling fins, air vents, and surface area "skin" in contact with the open air. The simulation must also take into account external thermal loading due to such factors as solar radiation and contact with the surface of the earth.

Accurate predictive thermal modeling is possible based on solid modeling, using three dimensional finite element mesh (FEM) techniques. The geometry is subdivided into small isomorphic solid volumes or *nodes*, and the thermal properties of the material of each node are recorded. Heat flow is calculated between all node pairs, where the links between mesh elements pairs are characterized by thermal coupling coefficients.

However, a full three dimensional thermal model requires that a great deal of material property information be known to a reasonable degree of accuracy. In many cases it is possible to compute a reasonable approximation to the thermal behavior of a vehicle using a description of the surfaces of the the vehicle, and some "lump parameters" for the thermal mass of the various components, and for the primary heat sources. This is the approach taken by PRISM, the Physically Reasonable Infra-red Simulation Model, developed by the Tank Automotive Command [REYN89]. Having the capability for generating an approximate surface description of a BRL-CAD model will permit vehicle designs stored in a BRL-CAD geometry database to be converted to a form suitable for analysis by the PRISM application program.

Once the thermal behavior of the vehicle is predicted, passenger comfort considerations can be directly assessed, and no further processing is required. For predicting a thermal signature, such as might be seen on an imaging infra-red sensor, it is necessary to take the simulated patterns of thermal energy radiation and convolve them with the transfer function of the atmosphere that lies between the vehicle and the sensor to determine the patterns of energy presented to the sensor. That pattern in turn must be convolved with the transfer function of the sensor itself, in order to predict the signal measured by the sensor. In the design of sensor systems, it is important to know the nature of vehicle signatures over a range of detection bands and for a variety of signal-processing schemes [RAPP76, RAPP83]. Linking BRL-CAD with PRISM provides a method for computing this information as part of the design loop. This linkage provides opportunities for vehicle designers to retain control over the thermal signatures of their vehicles, as well as giving sensor designers an environment for testing and refining improved sensors.

Radar Predictions

When a metallic vehicle is illuminated with radar energy, that energy is partly absorbed, and partly dispersed back into the surroundings. Some of the illumination energy returns to the transmission position, and it does so carrying an electronic "signature" of the vehicle [TOOM82]. Depending on the applications envisioned, a vehicle designer is usually interested in either maximizing the strength of the radar signature (for example, to make boats and commercial aircraft easier to locate in foul weather), or minimizing the strength and recognizability of the radar signature, such as in the design of low-observable ("stealth") aircraft.

A variety of different techniques exist to calculate the predicted radar signature of a given vehicle. The algorithms based on ray-tracing tend to handle multi-bounce effects very well but are unable to simulate edge diffraction and creeping wave phenomena. However, algorithms based on feature-based descriptions of the the vehicle or coarse polygonalizations tend to handle diffraction and creeping waves acceptably, but are unable to handle multiple bounce effects. The best known technique for the simulation of radar signatures is the Method-of-Moments technique [HARR82, MOOR84], which requires a polygonalization of the surface of the vehicle as input.

The nature of the calculation is much like that employed for predicting heat flow, as outlined earlier. However, in order to achieve high accuracy, the Method-of-Moment technique requires that each surface polygon be no wider than one fifth of one wavelength of the radar signal. The relationship between frequency f and wavelength λ is given by

$$\lambda = \frac{c}{f} = \frac{3 \times 10^8 \;\; m/s}{f \;\; Hz}$$

Radar frequencies begin in the UHF range with P-band radars transmitting from 225 MHz to 390 MHz, at a wavelength of about one meter [IEEE76]. A millimeter wave (W-band) radar transmitting at 94 GHz emits a signal with a wavelength of 3.2 millimeters. A radar transmitting at a higher frequency would have a very short wavelength indeed. The relationship between frequency f, wavelength λ, and the maximum facet size is given in Table 4. Thus, the method of moments technique requires exceptionally fine surface tessellations to be used. Tessellating full size vehicles this finely produces a gargantuan number of facets. Computing a solution to problems of such size is barely within the reach of present-day supercomputers.

Band	Frequency	Wavelength	Facet Size
P	300 MHz	1000 mm	600mm
L	1 GHz	300 mm	60mm
X	10 GHz	30 mm	6mm
W	94 GHz	3.2 mm	0.64mm

Table 4 – Relationship Between Frequency and Facet Size

Interfacing to Other CAD Systems

There are many reasons why the geometric database for a given design might need to be transferred from the CAD system which originated it to a completely different kind of CAD system. This might be done because two different organizations use different software packages and they need to exchange design files. Similarly, as part of a comparison of the advantages of different CAD systems, or of different analysis codes, it would be necessary to import a reference design. Or, some aspects of a design might be best handled using software uniquely suited to a particular specialized task, such as numerically-controlled (NC) machinery toolpath generation, or ISO-compliant blueprint generation.

Having NMG objects as "first class citizens" in the geometry database makes the task of importing faceted geometry from other CAD systems very straightforward. Existing faceted objects either with no explicit topology, or including explicit topology (most commonly using a winged edge representation), can be easily converted into an NMG object with full generality and no loss of topological information. There are a great many systems that employ faceted representations, so this is a significant capability.

Being able to convert models described using boolean combinations of the rich set of primitive solids into explicit surface descriptions also enables transfer of geometry in the outward direction as well. Shapes that are originally modeled using the existing CSG database can be exported to other CAD systems for additional processing.

Future Directions

Editing an NMG object

At present, the only way to create an NMG object is via the procedural interface of **librt**, either from the **mged** *facetize* command, or from an outboard database converter or procedural database generator that uses **libwdb**, the library for writing databases. Just as it is possible to edit all the other primitive solids, it seems reasonable to permit users to edit NMG solids. In principle, a basic NMG editing capability should not be much different than the existing ARB editing capability, which includes the ability to move vertices, move edges, and move faces. In addition to these familiar editing features, some kind of interface to the Euller operators will need to be created, so that new topological elements can be created, existing elements can be killed, and sets of existing elements can be joined together or split apart.

Editing Under Constraints

If the NMG editing capability is implemented in a very general way, it should be possible to replace most of the existing faceted object editing capabilities in **mged** with interfaces to a subroutine that implements editing of NMG objects under a set of general constraints. For example, when editing an ARB, if an edge is split, all faces sharing that edge need to be split as well. When moving a vertex, it must satisfy the constraint of keeping planar all the faces that contain it. A simple primitive shape like an ARB4 could be modified and expanded into a much more complicated topological arrangement, without forcing the user to destroy the original solid, and re-create the shape using a more general primitive.

Editing B-spline control meshes could also be implemented using this general NMG editing capability. In this case, if an existing edge was to be split, in order to create an extra control point, a whole row or column of extra control points would have to be inserted, to maintain the logically rectangular topology of the B-spline control mesh.

When an NMG editing operation would complete, a series of simple checks would see if the topology of the new shape qualified as an existing primitive shape, such as an ARB8 or an ARBN. If so, the new shape would be stored as the simpler and less general primitive, so as to take advantage of the more efficient data storage and faster analysis processing speeds inherrent in the less general primitive shapes.

This capability should replace a large portion of the existing geometry editing interface, and should provide geometry builders with unparalleled flexibility and power in creating new shapes.

NMG with Trimmed NURBS Faces

One of the most exciting current research projects at BRL is the extension of the NMG framework to permit faces either to be planar N-gons, or trimmed non-uniform rational B-splines ("trimmed NURBS"). This will permit many of the tessellation operations to be implemented exactly, rather than as approximations. This will also permit solids to enjoy the economy of having most faces be represented as planar N-gons, which are very compact and efficient to process, while those few faces that require sculptured surface shape control can be represented as trimmed NURBS. This combination provides both efficiency and full shape control in the rich non-Manifold topological framework: a combination that does not exist in any current commercial CAD system.

Interrogation Extensions

To date, most BRL-CAD applications programs have been implemented using the ray-tracing paradigm, because of ray-tracing having a lengthy head start. By choosing the ray sampling density within the Nyquist limit for a given spatial resolution, many applications based on ray-tracing are well satisfied by extracting ray/geometry intersection information. However, a mathematical ray has as its cross section a point, while physical objects have significant cross-sectional area. This lack of cross-sectional area will always lead to some sampling inaccuracies. Applications which simulate particles or small rocks approaching the model might benefit from having a direct cylinder/geometry intersection capability, and applications which shine beams of light on the model such as spotlights or even highly collimated light such as laser light might benefit from cone/geometry intersection capabilities [AMAN84, KIRK86]. Applications which are attempting to simulate wave effects might be well expressed in terms of plane/geometry intersection curves, and structural analysis routines would probably prefer to obtain the geometry as a collection of connected hyperpatches.

While recent research has begun to explore techniques for intersecting cylinders, cones, and planes with geometry [KAJI83], ray-tracing and polygon-based techniques are by far the most well developed approaches. However, there are several additional type of interface to the model geometry that are likely to be of general applicability.

3-D Finite-Element Volume Mesh

Many forms of energy flow analysis, such as heat flow, vibrational analysis (acoustic energy flow), and stress analysis require the use of 3-D Finite-Element Mesh (FEM) techniques. While there has been some work on using the ray-tracing paradigm to construct finite element and finite difference meshes [LAGU89], it has been difficult to deal with high spatial frequency (fine detail) portions of the model. In particular, meshing small diameter pipes is problematic, since undersampling can cause the pipe to incorrectly be separated into multiple pieces. In order to improve on the current state of affairs, it seems necessary to provide support for the generation of volume meshes directly as part of the application interface. This would provide the meshing algorithm to have unrestricted access to the underlying geometry, the space partitioning tree, and other internal data in order to perform a better job.

Even more promising still would be a strategy that takes advantage of the NMG support. A first pass might tessellate the model and evaluate the booleans to produce a surface mesh. The second pass would then take the surface mesh and fill the interior (or exterior) volumes with appropriately chosen volume elements. A very good fit could probably be achieved using only parallelepiped ("brick") elements and 20-node "superelements". The brick elements would be used to fill interior volume that does not border on a face, and the superelements would be used for volume that contacts a face. Recourse could be made back to the underlying geometry (perhaps via firing a few well chosen rays) to get the curvature of the superelement faces to match the curvature of the underlying primitive, rather than having to rely strictly on the NMG planar-face approximation.

Homogeneous Trimmed B-Splines

When support for trimmed NURBS faces has been added to the NMG capability, it will be possible to represent all existing primitives either with exact rational B-spline versions, or with very good rational B-spline approximations. This could be done even for faces that were completely planar.

This offers the hope that it might be possible (albeit memory intensive) to convert an entire CSG solid model into a homogeneous collection of non-uniform rational B-spline faces organized in a non-manifold topological data structure. In addition to the conceptual simplicity afforded by having a uniform representation for shape, this affords the opportunity to create new analysis codes that can process curved surfaces, yet at least initially only have to deal with one kind of shape. This would also provide a very direct and natural interface to spline based [ROGE90] and Bezier-patch [BEZI74] based modeling systems.

Analytic Analysis

Given a homogeneous geometric representation such as the Trimmed B-Splines just discussed which also has an analytic representation, a further processing capability arises. Rather than interrogating the data base by means sampling or subdivision techniques, the direct mathematical manipulation of the source geometry through its parametric representation becomes possible. Calculations of physical properties requiring integration over a surface can often be evaluated with greater accuracy using an explicit analytic calculation than would be provided by numerical evaluation. While this may be difficult in general due to the complexity of the parametric expression, some classes of surface representations good candidates. Splines, for example, are piecewise-polynomial functions which have relatively simple Fourier transform representations. Since 2-D spatial Fourier transforms arise frequently in far-field electromagnetic scattering calculations, exploitation of the parametric spline representation is of interest in predictive scattering calculations.

With the rapidly developing potential of symbolic calculation, treatment of seemingly impossible formulas resulting from the geometry/physics interaction may become tenable. This could help to reduce the trend towards employing numerical methods at the onset of a problem and avoid some of the accompanying instabilities and errors.

Summary

In this paper, a brief history of solid modeling has been presented, with special emphasis placed on the central role that the solid model plays in the design and analysis cycle. A detailed look was taken at how different analysis applications can interrogate the solid model to extract relevant information. For CSG-Rep solid modeling system, the lack of an explicit representation for the final, developed shapes was identified as a critical lack.

The systems engineering issues associated with creating an explicit representation for CSG solid models without loosing the fidelity of existing geometric databases were considered, and a strategy using Non-Manifold Geometry data structures was adopted. The implementation of the NMG data structures and algorithms were presented in significant detail. To convert existing primitive shapes into NMG objects, the details of several tessellation algorithms were examined, with particular attention being paid to the topic of rigorous user-controlled error bounds on the algorithms.

Evaluating boolean combinations of tessellated objects has only recently become tractable, since the use of the NMG data structures makes these operations much more straightforward. The details of a three-stage algorithm for boolean evaluation were presented in sufficient detail to permit the average reader to be able to implement this technique.

Finally, the existing user interface was presented, as well as an overview of a whole gamut of new applications that will now be able to process existing CSG geometry databases. This represents a major and important capability, the true significance of which will only become apparent over the next few years.

Acknowledgements

The authors would like to thank Dr. Paul Deitz for providing unflagging support and encouragement for this effort. He has established a research atmosphere that breeds good work, and is fun to work in. The authors would also like to thank Prof. Dave Rogers for once again persuading us to take the time to write all this down. Finally, the clarity of the paper was greatly improved thanks to numerous suggestions by Susanne Muuss and Christopher Johnson.

References

[AMAN84] J. Amanatides, "Ray Tracing with Cones," *Computer Graphics (Proceedings of Siggraph '84)*, vol. 18, no. 3, July 1984.

[APPL88] K. A. Applin, M. J. Muuss, R. J. Reschly, M. Gigante, and I. Overend, *Users Manual for BRL-CAD Graphics Editor MGED*, BRL Internal Publication, October 1988.

[BEZI74] P. E. Bezier, *Mathematical and Practical Possibilities of UNISURF*, Academic Press, New York, 1974.

[COBB84] E. S. Cobb, *Design of Sculptured Surfaces using the B-spline Representation*, PhD dissertation, University of Utah, June 1984.

[COOK84] R. Cook and Porter, L. Carpenter, "Distributed Ray Tracing," *Computer Graphics (Proceedings of Siggraph '84)*, vol. 18, no. 3, pp. 137-145, July 1984.

[COON67] S. A. Coons, *Surfaces for Computer-Aided Design of Space Forms*, Tech. report MAC-TR-41, Project MAC, MIT, NTIS AD No. 663-504, Cambridge MA, June 1967.

[deBO78] C. deBoor, *A Practical Guide to Splines*, Applied Mathematical Sciences 27, Springer-Verlag, New York, 1978.

[DEIT82] P. H. Deitz, *Solid Modeling at the US Army Ballistic Research Laboratory*, 2, pp. 949-960, Proceedings of the 3rd NCGA Conference, 13-16 June 1982.

[DEIT83] P. H. Deitz, *Solid Geometric Modeling – The Key to Improved Materiel Acquisition from Concept to Deployment*, Defense Computer Graphics 83, Washington DC, 10-14 October 1983.

[DEIT84a] P. H. Deitz, *Modern Computer-Aided Tools for High-Resolution Weapons System Engineering*, DoD Manufacturing Technology Advisory Group MTAG-84 Conference, Seattle WA, 25-29 November 1984.

[DEIT84b] P. H. Deitz, *Predictive Signature Modeling via Solid Geometry at the BRL*, Sixth KRC Symposium on Ground Vehicle Signatures, Houghton MI, 21-22 August 1984.

[DEIT85] P. H. Deitz, *The Future of Army Item-Level Modeling*, Army Operations Research Symposium XXIV, Ft. Lee VA, 8-10 October 1985.

[DEIT88] P. Deitz, W. Mermagen Jr, and P. Stay, "An Integrated Environment for Army, Navy, and Air Force Target Description Support," *Proceedings of the Tenth Annual Symposium on Survivability and Vulnerability*, April 1988.

[GOOD89] Michael T. Goodrich, "Triangulating a Polygon in Parallel," *Journal of Algorithms*, vol. 10, 1989.

[GOUR71] H. Gouraud, "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers*, vol. C-20, no. 6, pp. 623-628, June 1971.

[HARR82] R. F. Harrington, *Field Computation by Moment Methods*, Krieger, Malabar, Florida, 1982.

[IEEE76] IEEE, *IEEE Standard 521*, Institute of Electrical and Electronic Egnineers, Piscataway NJ, November 30, 1976.

[KAJI83] J. T. Kajiya, "New Techniques for Ray Tracing Procedurally Defined Objects," *Transactions on Graphics*, vol. 2, no. 3, pp. 161-181, July 1983.

[KEDE85a] G. Kedem, *Computer Structures and VLSI Design for Curve-Solid Classification*, Siggraph '85 Tutorial "VLSI for Computer Graphics", San Francisco CA, July 23, 1985.

[KEDE85b] G. Kedem and J. L. Ellis, *Computer Structures for Curve-Solid Classification in Geometric Modeling*, Siggraph '85 Tutorial "VLSI for Computer Graphics", San Francisco CA, July 23, 1985.

[KIRK86] D. B. Kirk, "The Simulation of Natural Features Using Cone Tracing," in *Advanced Computer Graphics*, ed. T. L. Kunii, pp. 129-144, Springer-Verlag, 1986.

[LAGU89] G. Laguna, *Recent Advances in 3D Finite Difference Mesh Generation Using the BRL-CAD Package*, pp. 21-35, BRL-CAD Symposium '89, Aberdeen Proving Ground, MD, 24-25 October, 1989.

[LAID86] David H. Laidlaw, W. Benjamin Trumbore, and John F. Hughes, "Constructive Solid Geometry for Polyhedral Objects," *Computer Graphics*, vol. 120, no. 4, Proceedings of SIGGRAPH 86, Dallas, Texas, August 1986.

[LANI79] J. H. Laning and S. J. Madden, "Capabilities of the SHAPES System for Computer Aided Mechanical Design," *Proc. First Annual Conference on Computer Graphics in CAD/CAM Systems*, pp. 223-231, Cambridge MA, April 9-11, 1979.

[MAGI67] MAGI, *A Geometric Description Technique Suitable for Computer Analysis of Both Nuclear and Conventional Vulnerability of Armored Military Vehicles*, MAGI Report 6701, AD847576, August 1967.

[MOLN87] Zsuzsanna Molnar, "Advanced Engineering/Scientific Graphic Workstations," in *Techniques for Computer Graphics*, ed. D. F. Rogers, R. A. Earnshaw, Springer-Verlag, 1987.

[MOOR84] J. Moore and R. Pizer (eds), *Moment Methods in Electromagnetics*, Wiley, New York, 1984.

[MUUS87a] M. J. Muuss, "Understanding the Preparation and Analysis of Solid Models," in *Techniques for Computer Graphics*, ed. D. F. Rogers, R. A. Earnshaw, Springer-Verlag, 1987.

[MUUS87c] M. J. Muuss and P. Dykstra, K. Applin, G. Moss, E. Davisson, P. Stay, C. Kennedy, *Ballistic Research Laboratory CAD Package, Release 1.21*, BRL Internal Publication, June 1987.

[MUUS88a] M. J. Muuss and P. Dykstra, K. Applin, G. Moss, P. Stay, C. Kennedy, *Ballistic Research Laboratory CAD Package, Release 3.0 – A Solid Modeling System and Ray-Tracing Benchmark*, BRL Internal Publication, October 1988.

[MUUS90b] M. J. Muuss, *Multiple Families of Engineering Analyses Interrogating a Single Geometric Model*, Proceedings of the 8th Army Math

Conference, Ithaca NY, 19-22 June 1990.

[OKIN78] N. Okino and et al., *TIPS-1, '77 Version*, Institute of Precision Engineering, Hokkaido University, Sapporo Japan, March 1978.

[PELF86] John Pelfer, *Georgia Tech Research Institute Radar Cross Section Modeling Software*, Modeling and Analysis Division, Georgia Tech Research Institute, October 1986.

[RAPP76] J. R. Rapp, *A Computer Model for Predicting Infrared Emission Signatures of An M60A1 Tank*, BRL Report No. 1916, NTIS AD No. B013411L, August 1976.

[RAPP83] J. R. Rapp, *A Computer Model for Estimating Infrared Sensor Response to Target and Background Thermal Emission Signatures*, BRL Memorandum Report ARBRL-MR-03292, August 1983.

[REQU82] A. A. G. Requicha and H. B. Voelcker, "Solid Modeling: A Historical Summary and Contemporary Assessment," *IEEE Computer Graphics and Applications*, vol. 2, no. 2, pp. 9-24, March 1982.

[REYN89] William R. Reynolds, *PRISM User's Manual Version 2.0*, Keweenaw Research Center, Michigan Technological University, Houghton, MI 49931, October 1989.

[RITC78a] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1905-1929, 1978.

[RITC78b] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1991-2019, 1978.

[ROGE90] D. F. Rogers and J. A. Adams, *Mathematical Elements for Computer Graphics, 2nd ed.*, McGraw-Hill, New York, 1990.

[THOM84] S. W. Thomas, *Modelling Volumes Bounded by B-spline Surfaces*, PhD dissertation, University of Utah, June 1984.

[TOOM82] J. C. Toomay, *Radar Principles for the Non-Specialist*, Lifetime Learning Publications, London, 1982.

[WEIL85] Kevin J. Weiler, "Edge-based Data Structures for Solid Modeling in Curved-Surface Environments," *IEEE Computer Graphics and Applications*, vol. 5, no. 1, pp. 21-40, January 1985.

[WEIL87] Kevin J. Weiler, "The Radial Edge Structure: a Topological Representation for Non-Manifold Geometric Modeling," in *Geometric Modeling for CAD Applications*, ed. M. Wozny, H. McLaughlin, and J. Encarnacao, Springer Verlag, December 1987.