# *Chapter 1.*
# Boolean Component

Component: *Booleans

The Boolean Component (BOOL), in the bool directory, performs *Boolean operations* on the model topology of bodies, first finding the intersections between bodies, and then deciding which pieces to group together and which to discard. Types of Boolean operations include:

- *Unite*, which joins the topology of two bodies together, keeping all portions of both bodies.
- *Intersect*, which keeps the portions of the two bodies that overlap in space, and discards non-overlapping portions of the bodies.
- *Subtract*, which discards the portions of the two bodies that overlap in space from one of the bodies.

A body involved in a Boolean operation may be composed of solid, sheet or wire components. Initial and resulting bodies may be manifold or nonmanifold. The results of the Boolean operation may be regularized or nonregularized point sets.

*Stitching* joins two faces along edges or vertices that are identical. Stitching is *not* a Boolean operation. It is simpler than a Boolean because it avoids face/face intersections and the evaluation of lump and shell containments. Stitching only operates on faces, not wires.

*Intersectors* find the points or intervals at which curves and surfaces meet. Intersectors are implemented in low-level C++ classes that are not related to Booleans. Intersectors operate on model geometry.

The Boolean Component implements both Booleans and stitching, and requires both intersectors and Euler operations, which are implemented in other ACIS components. Refer to the *Intersectors Component Manual* for more information about intersectors and the *Euler Operations Component Manual* for information about Euler operations.

## Boolean Operations

Topic: *Booleans

The bodies involved in Boolean operations are often referred to as a *blank body* and a *tool body*. The blank body is the body on which to perform the operation and the tool body is the body with which to perform the operation. This is analogous to a “tool” machining portions of a “blank.” There are three basic Boolean operations (illustrated in Figure 1-1):

*Unite* . . . . . Creates a body that contains all points that are in either of the two input bodies. This joins the tool body and the blank body.

*Intersect* . . . . . Creates a body that contains all points that are common to both the two input bodies. The result is all points that are in both the tool body and the blank body.

*Subtract* . . . . . Creates a body that contains all points that are in the blank body but not in the tool body. This is the difference of the points in the tool body from the points in the blank body.

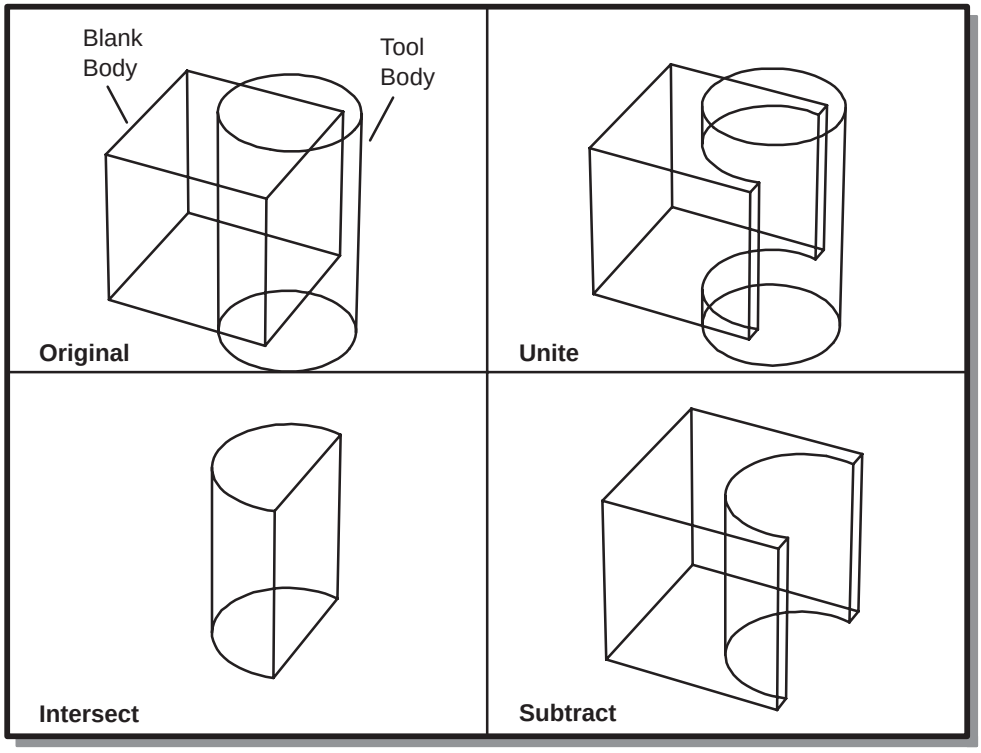


**Figure 1-1. Boolean Operations**

Each Boolean operation (unite, intersect, subtract) is separated into four stages:

1. Intersect all of the faces and edges of both bodies and build an intersection graph, which is a data structure defining the intersection of the bodies. This is described in the section *Intersection Graph*.

2. Imprint the intersection graph onto both bodies, often splitting faces in the process.

3. Determine what data (faces, shells, edges, etc.) from both bodies to keep and discard, then discard the unnecessary data.

4. Join (or un-join) the bodies along the intersection edges, and correctly organize any new shells and lumps.

The Boolean Component also provides a *chop* operation, which chops the blank body with the tool body. It returns the body formed by simultaneously subtracting the tool from the blank, and the body formed by intersecting the tool with the blank. Although the same results can be obtained by performing the intersection and subtraction separately, the chop is more efficient. The "leftovers" from the chop can be saved or discarded. Figure 1-2 illustrates the chop operation using the Scheme command bool:chop.

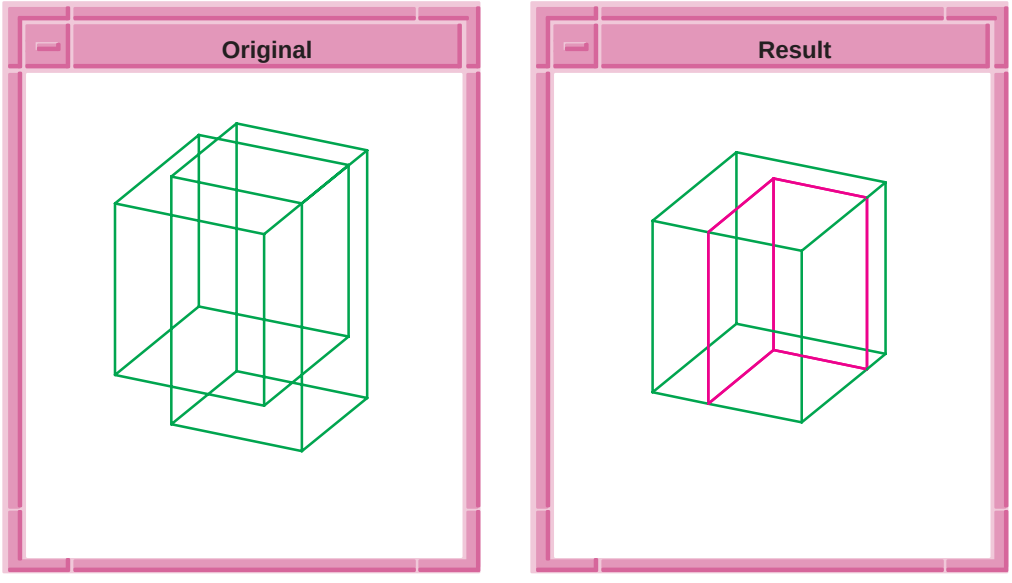


**Figure 1-2. Chop**

# Intersection Graph

Topic: *Booleans

The *intersection graph* is a set of edges and vertices that represent the intersection of two bodies. It contains no shells, but a list of one or more wires. Each wire describes, independently, a connected portion of the intersection between the two bodies. Any two wires are geometrically and structurally disjoint.

For discussion, consider a single wire. The basic constituent of any wire is the edge, which represents a connected bounded portion of a geometric curve; in this case a curve of intersection between two faces, one from each body. The shape of the edge is described in an attached curve record, and the bounds are given by a start and end vertex. The connectivity between edges of the wire is recorded by the use of coedge records attached to the edges.

Each intersection graph edge has one coedge attached to it for each face of each body that passes through the edge. A face is counted twice if it passes right through, appearing on both sides of the edge, and counted once if the intersection lies on the boundary of the face, so that the face is only on one side of the edge. The coedges corresponding to the two intersecting bodies are treated separately.

The edge record points directly to one of its coedges. The coedges are linked together, in a circular list, as *partners*. When an intersection edge lies within a face of the blank body, it has two coedges, of opposite sense, and may point to either of them. When it lies on the boundary edge of one or more body faces, its coedges correspond in number, sense, and order with those of the body edge. When this would result in all the coedges having the same direction (on an incomplete or sheet body), an additional dummy coedge is added at the end of the loop (i.e., immediately before the coedge to which the edge points), with the converse sense. A dummy coedge may be present even if not strictly necessary. An intersection edge coincident with a wire edge of the body has one coedge corresponding to the body coedge, plus a dummy coedge of the converse sense.

***Note*** *To* correspond in sense *means: If the edges go in the same direction, the intersection graph edge's sense is the same as the body edge's sense. If the edges go in opposite directions, the intersection graph edge's sense is the converse of the body edge's sense (e.g., if the body edge's sense is forward, the intersection graph edge's sense is reversed).*

An intersection graph edge with no geometry and identical start and end vertices is used to represent an intersection point. If the intersection point lies on a body face, there is one coedge corresponding to that face. If it lies on the interior of an edge, there are two coedges of opposite sense: the forward coedge corresponding to the portion of the edge starting at the intersection, and the reversed coedge corresponding to the portion ending at the intersection. If the intersection point lies on a vertex of a wire of the body, there is one forward coedge for each edge starting there, and one reversed coedge for each edge ending there. Again, if this would result in all the coedges having the same sense, a dummy coedge of the opposite sense is added.

The coedges are linked together at the vertices using the 'previous' and 'next' pointers, corresponding to the start and end vertices, respectively. For each intersection edge, exactly two coedges (one of each sense) have non-NULL pointers, except when a wire is a single open-ended edge, in which case there are no non-NULL pointers at all. When a wire branches at a vertex, the order of connection of the coedges is not important, provided that every edge is visited by the following procedure:

1. Select a coedge which ends at the vertex and has a non-NULL ‘next’ pointer.
2. Follow the ‘next’ pointer.
3. Search the partner ring for the coedge of opposite sense with a non-NULL ‘next’ pointer.
4. Repeat from step 2 unless this coedge is the original one.

The ‘next’ and ‘previous’ pointers are complementary: a coedge is always the previous coedge of the next coedge, and vice versa.

Coedges corresponding to faces of the tool body are linked together in exactly the same way, except that in each case the edge does not point directly to one of its tool coedges. Instead, the connection is through an attribute of the edge.

Each topological entity in an intersection wire carries an attribute that relates it to corresponding entities in the original bodies. These attributes contain data essential to the operation of the later stages of Boolean operations, and so must be constructed correctly.

# Slice

Topic: *Booleans

A *slice* operation computes the intersection graph between two bodies and creates a wire body corresponding to the graph. This wire body is different from the general intersection graph in that it has two pairs of coedges, one for each of the bodies, that indicate where the edge came from on each body.

Slice can be performed between any model bodies. Any faces and wires within a body, internal or external, can take part in a slice. Slice may also be performed on cellular bodies.

Uses for slice include the generation of waterlines for NC machining or the generation of a cross sectional view on an engineering drawing. These can be done by slicing a solid with a plane. Slice can also be used to provide a profile for:

- Generating new models
- Generating contours to study a shape
- Examining intersections between solids

# Imprint

Topic: *Booleans

An *imprint* operation computes the intersection graph of the tool body and the blank body and embeds it into both bodies, splitting faces and edges of the original two bodies with the edges of the intersection graph. The edges and vertices of the intersection graph are added to both bodies wherever edges and vertices do not already exits. If a closed loop of edges is created, a new face is made. An open loop of edges can be added as a spur to an existing loop on a face or as a slit in the face.

Uses for imprinting include:

- Creating edges on a body for subsequent operations, such as sweeping or skinning.
- Visualizing the intersection of two bodies without actually performing a Boolean operation.
- Debugging a Boolean operation.

# Regularized and Nonregularized

Topic: *Booleans

ACIS provides regularized and nonregularized Boolean operations.

*Regularized* Boolean operations are performed on the interior point sets, exclusive of the model boundaries. This type of Boolean operation resolves many modeling tangency issues.

*Nonregularized* Boolean operations include interactions of model boundaries, and leave all boundary coincidences unresolved. This also keeps boundary faces that are internal to a union in the model.

## Sheet–Solid and Sheet–Sheet Results

Topic: *Booleans

Table 1-1 defines sheet/solid and sheet/sheet results for regularized Boolean operations. These operations are the same for wires.

**Table 1-1. Regularized Boolean Operations**

| Operation | Intersecting | Result |
|---|---|---|
| SUBTRACT | Solid from sheet | Portion of sheet in solid region removed |
| SUBTRACT | Sheet from solid | Portion of sheet within solid interior embedded in solid |
| SUBTRACT | Sheet from sheet (coincident) | Portion of subtracted sheet that overlaps other sheet is deleted (two-dimensional subtract) |
| SUBTRACT | Sheet from sheet (not coincident) | Subtracted sheet deleted |
| UNITE | Solid with sheet | Portion of sheet exterior to solid region added |
| UNITE | Sheet with sheet (coincident) | Two-dimensional unite |
| UNITE | Sheet with sheet (not coincident) | Intersect surfaces and join |

| Operation | Intersecting | Result |
|---|---|---|
| INTERSECT | Solid with sheet | Portion of sheet interior to solid remains |
| INTERSECT | Sheet with sheet (coincident) | Two-dimensional intersection |
| INTERSECT | Sheet with sheet (not coincident) | NULL body |

# Embedded Faces and Wires

Topic: *Booleans

Subtracting a sheet from a solid leaves an embedded face, or set of faces, within the solid region. Subtracting a wire from a solid leaves an embedded edge, or set of edges, within the solid region. An embedded face represents a 2D void, sometimes called a “slit,” in the solid. Similarly, an embedded edge represents a 1D void, sometimes called a “worm hole,” in the solid.

Embedded faces and wires participate in Booleans. During a unite, any embedded face or wire portions in a body that end up inside the other body are removed. A subtraction of a body with embedded faces or wires from a solid body leaves the former embedded faces or wires as external faces or wires.

# Nonregularized Operations

Topic: *Booleans

ACIS provides the capability for performing nonregularized Booleans. This is generally implemented by specifying a special argument to a Boolean function. (Scheme extensions may implement separate nonregularized versions, or provide a command option for nonregularized.)

Specifying nonregularized essentially adds three new conditions to the Boolean operations:

- When SINGLE_SIDED faces become DOUBLE_SIDED, BOTH_INSIDE faces, they remain in the resulting body.
- Any face-face coincident region remains in the resulting body.
- No edge or vertex merging is performed at the end of the Boolean.

Due to the first condition, the unite operation always keeps all face regions from the two bodies (although it may split them into separate faces). Due to the second condition, the intersection of two blocks that share a coincident face always leaves the face instead of deleting it. Due to the third condition, subtracting a sheet from a noncoincident sheet leaves the imprint of the subtracted sheet on the other sheet.

Figures 1-3 through 1-5 show examples of regularized and nonregularized Booleans.

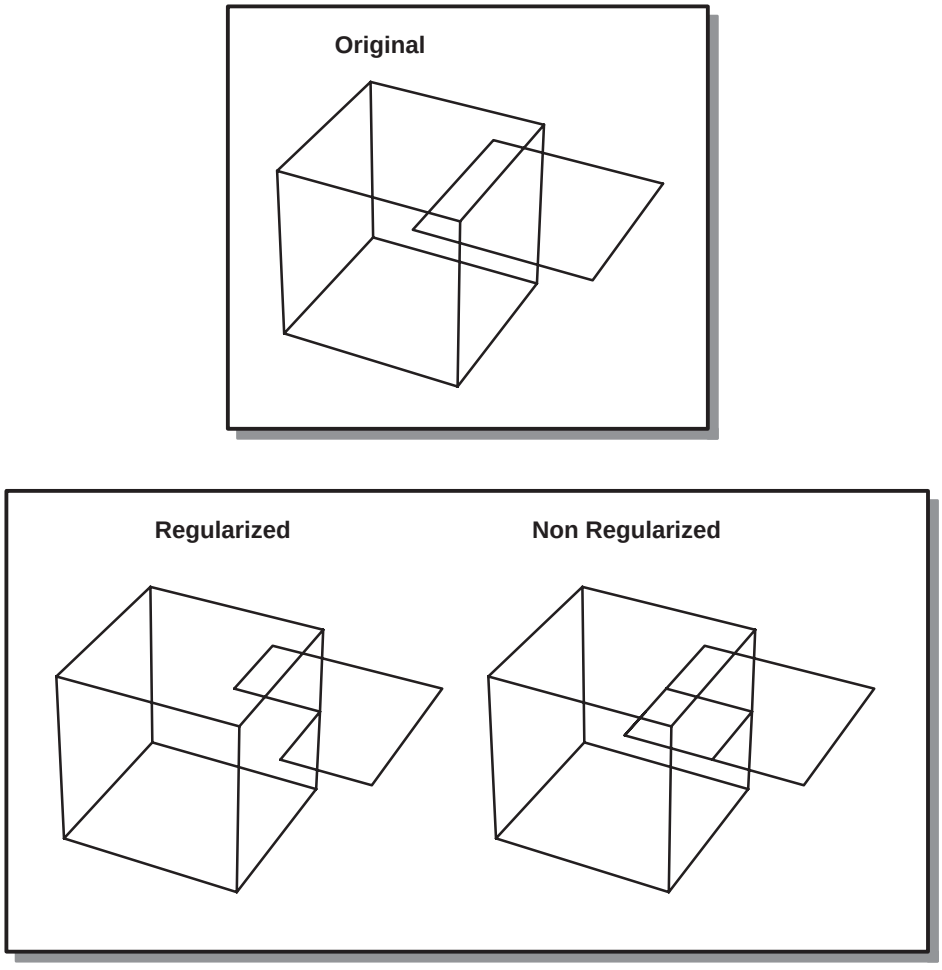


**Figure 1-3. Regularized vs. Nonregularized Unite**

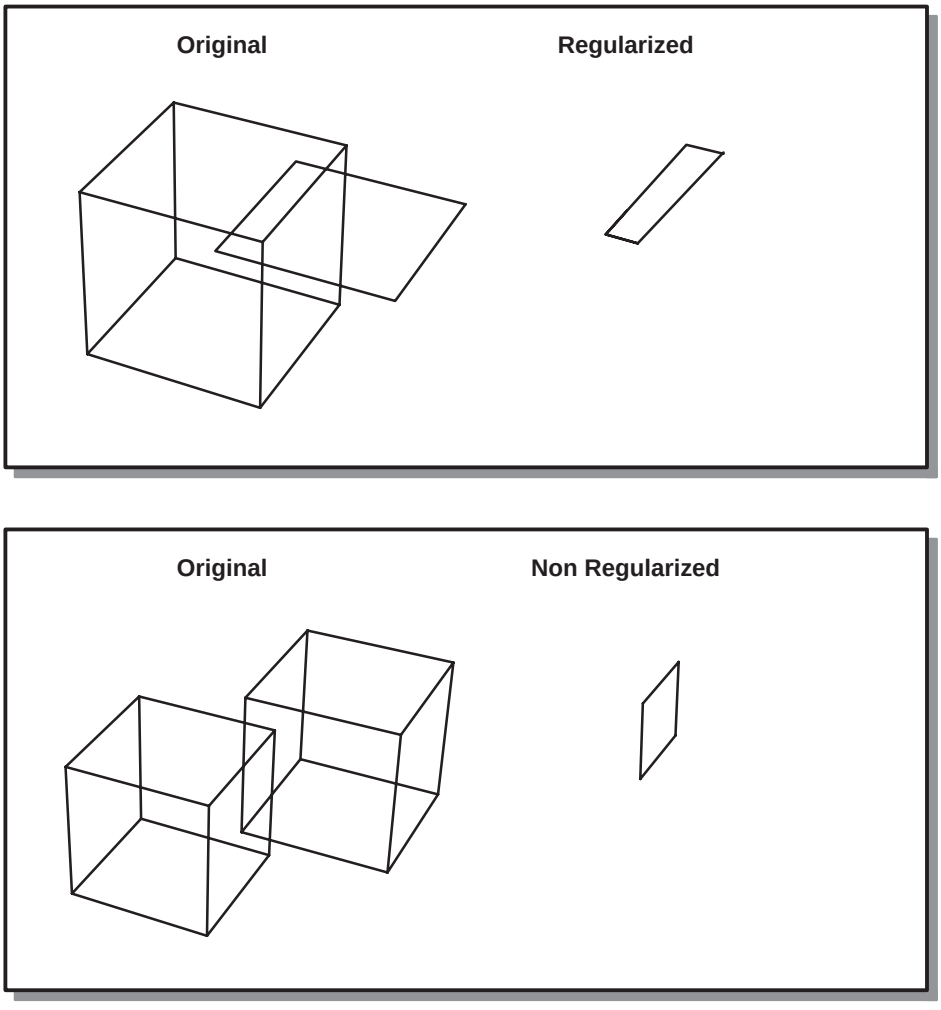


**Figure 1-4. Regularized vs. Nonregularized Intersect**

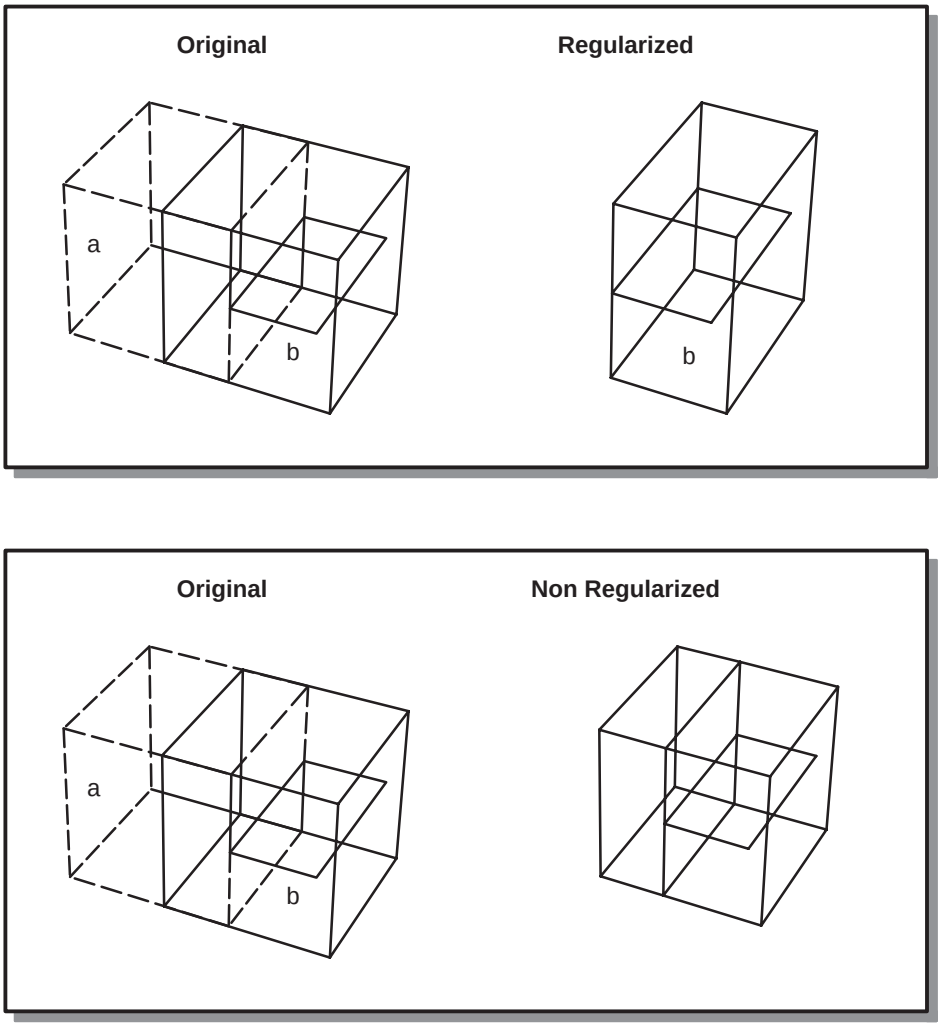


**Figure 1-5. Regularized vs. Nonregularized Subtract**

# Open Shell Booleans

Topic: *Booleans

Limited single-sided open shell Booleans are also possible. A single-sided open shell operates like a half-space in a Boolean if it extends beyond the other body on all sides; i.e., if the intersection curve lies entirely within the single-sided open shell.

Figures 1-6 and 1-7 illustrate examples of open shell Booleans. The original bodies, which are shown in Figure 1-6, are a planar face (the open shell), which is oriented such that the “inside” of the shell is downward, and a solid sphere, which is positioned such that the planar face intersects it about its equator. The face normals are shown with arrows.

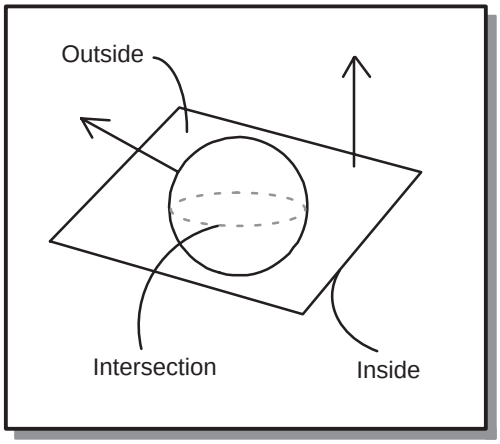


**Figure 1-6. Open Shell (Planar Face) and Solid Sphere**

Figure 1-7 illustrates four cases of open shell Booleans with the planar face and solid sphere:

1. Subtract the open shell planar face from the solid sphere.
2. Subtract the solid sphere from the open shell planar face.
3. Intersect the open shell planar face with the solid sphere.
4. Unite the open shell with the solid sphere.

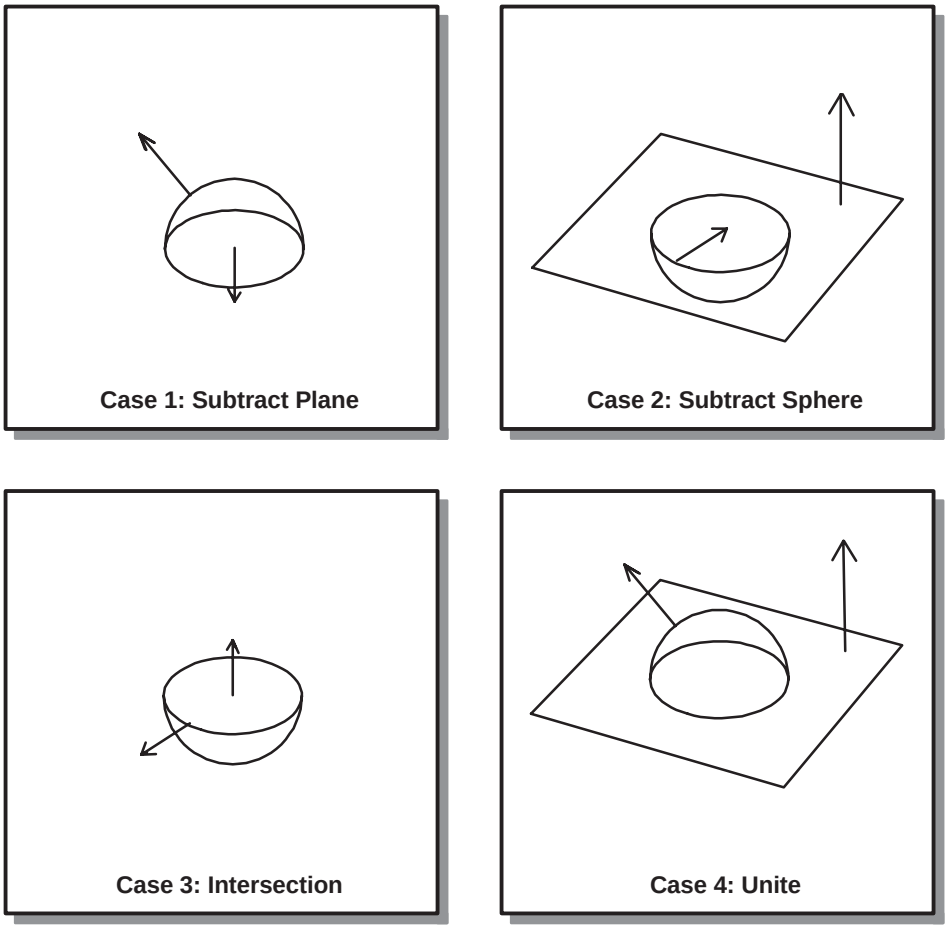


**Figure 1-7. Open Shell Booleans**

# Partial Booleans

Topic: *Booleans

The ACIS implementation of partial Boolean operations allows one to perform “smart” Boolean operations. Partial Boolean operations are more efficient than the standard Boolean operations, because they have prior knowledge about the result. In particular, speed increases are achieved by knowing which faces need to be intersected and, optionally, what the intersection curves are. Two examples in which partial Booleans would provide a speed increase are inserting a pattern of features on a planar face, and modifying the definition of a feature with no topological model changes.

***Note*** *This "selective" partial Boolean functionality is not the same as the functionality implemented in the* Selective Booleans Component *(SBOOL). SBOOL performs a Boolean operation between two bodies and allows the user to select the regions to keep in the model, using graph theory and cellular topology.*

The APIs used for performing partial Boolean operations include:

api_boolean_start . . . . . Initializes the data structures for the Boolean operation.

api_update_intersection . . . Creates a surface-surface intersection record when the intersection curve is known. The intersection curve should be transformed into the coordinate system associated with the blank body.

api_selectively_intersect . . . Intersects an array of faces from the tool body with an array of faces in the blank body. This eliminates the checking of all faces of the tool body against all faces of the blank body. If an intersection record from api_update_intersection exists, this record will be used instead of performing an intersection to modify the intersection graph.

Depending on whether one wants to do a slice, imprint, imprint stitch, or complete Boolean operation, the operation can be completed with APIs such as api_slice_complete, api_imprint_complete, api_imprint_stitch_complete, or api_boolean_complete.

Several Scheme extensions demonstrate the use of partial Booleans. The source code for the Scheme extensions is provided to all customers and can be used for examples of implementing partial Boolean operations. For example, the Scheme extension solid:imprint imprints an edge onto a face. The curve underlying the edge is known to be the intersection curve, so the imprint operation does not need to do the curve-surface intersection.

# Stitching

Topic: *Booleans, *Stitching

*Stitching* joins two bodies (faces) along edges or vertices that are identical. A stitch is simpler than a Boolean operation because stitching avoids face-face intersections and the evaluation of lump and shell containments. Most of the overhead in a stitching operation is associated with comparing edges to determine if they are entirely identical (coincident) or share some coincident subregion. Stitching only operates on faces, not on wires. It joins faces to faces. If wires exist in one of the bodies being stitched, but do not participate in the stitch (i.e., they do not coincide with edges in the other body), they will transfer to the resulting body.

Figure 1-8 is an example of stitching two sheets, *w1* and *w2*, along their common edge to form the single body, *w1*.

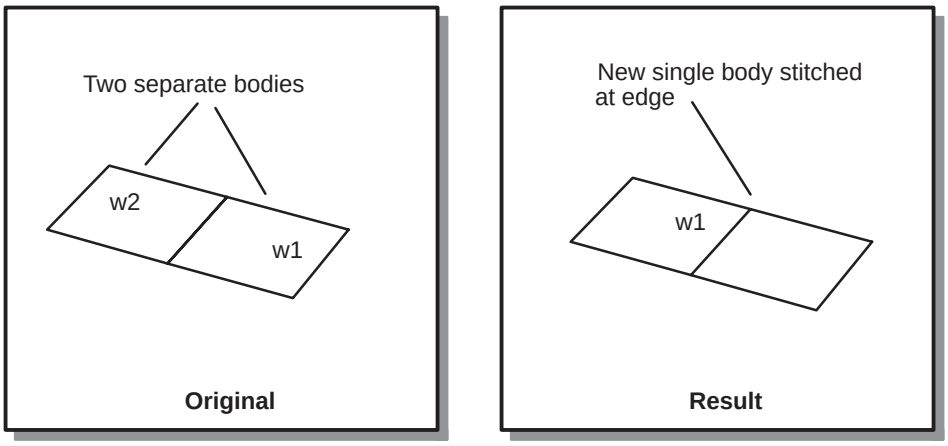


**Figure 1-8. Stitching**

The standard stitch operation, invoked by calling the api_stitch function without the optional split argument, compares the edges and vertices of the two bodies. If the function finds any edges or vertices that are coincident (within system tolerance) over their entire length, the function joins the bodies along those edges and vertices.

The only exception to this process is when two edges contain single-sided faces of incompatible orientation (i.e., having the same coedge direction, and thus incompatible face normals). Stitching will fail when incompatible coedges are encountered. However, the API api_stitch is versioned. If a pre–R10 version of this API is called, it will continue with the stitching but the incompatible edges will not be stitched. Vertices that lie on incompatible coedges are also not stitched, despite the fact that they are coincident. Consequently, the resulting body may be invalid. Any other combination of faces that share coincident edges are stitched, even if the final edge has more than two faces around it after the stitch; i.e., is nonmanifold.

All ACIS stitching operations that create nonmanifold edges attempt to preserve the validity of the ACIS model by performing what amounts to a nonregularized union operation on those edges. As a consequence, any faces that end up as internal to the final model are changed to be double-sided-inside faces. This does not include shells that do not interact with the other body, because stitching does not deal with shell or lump containment issues.

The variations on stitching are *edge splitting, self-stitching,* and *imprint stitching.*

*Edge splitting* is available by calling api_stitch with the flag argument, split, set to TRUE. In this version of stitch, two edges need only be coincident over some shared region. Edges are broken into two pieces with a new intervening vertex at boundaries of coincident regions. This produces a set of new edges, some of which have no relation and some of which are coincident in their entirety. The coincident edges are then stitched following the same rules as in standard stitching.

*Self-stitching* is available when the api_stitch function is called with the same body specified for both of the input body arguments. The function searches for any redundant vertices and edges in the body and merges them, with the limitation that only a single redundancy is handled properly. A model containing three identical edges that need to be merged is not guaranteed to be stitched correctly.

*Imprint stitching* is available as a separate API. The api_imprint_stitch function essentially performs the same operation as an imprint followed by a stitch, but the function uses the intersection information from the imprint when stitching. Therefore, it does not have to perform the expensive search for identical edges, but instead merely merges them. Imprint stitching groups the two input bodies into one, even if they do not intersect.