Solid Models

Solid Representations: An Introduction

Most geometric objects we see every day are solids. That is, they are geometric objects with interior. Solids can be very simple like a cube or very complex like a piston engine. To be processed by computers, solids must have some representations that can describe the geometry and characteristics completely. In fact, a good representation should address the following issues:

1. ***Domain*** :
   While no representation can describe all possible solids, a representation should be able to represent a useful set of geometric objects.
2. ***Unambiguity*** :
   When you see a representation of a solid, you will know what is being represented without any doubt. An unambiguous representation is usually referred to as a ***complete*** one.
3. ***Uniqueness*** :
   That is, there is only one way to represent a particular solid. If a representation is unique, then it is easy to determine if two solids are identical since one can just compare their representations.
4. ***Accuracy*** :
   A representation is said ***accurate*** if no approximation is required.
5. ***Validness*** :
   This means a representation should not create any invalid or impossible solids. More precisely, a representation will not represent an object that does not correspond to a solid.
6. ***Closure*** :
   Solids will be transformed and used with other operations such as union and intersection. "**Closure**" means that transforming a valid solid always yields a valid solid.
7. ***Compactness and Efficiency*** :
   A good representation should be compact enough for saving space and allow for efficient algorithms to determine desired physical characteristics.

These issues may be contradictory with each other. For efficiency purpose, a curvilinear solid may be approximated by a polyhedron. There are many efficient and robust algorithms for handling polyhedra; however, accuracy may not be maintained in the process of approximation. For example, given two curvilinear solids that are tangent to each other, this tangency may disappear after converting to a polyhedron.

Problems occur even for the polyhedra world. Many graphics APIs such as PHIGS PLUS and OpenGL have built-in data structures for representing polyhedra; but, these representations could generate invalid solids. There are representations that can always represent valid solids; but, these representations are in general more complex than those available in graphics APIs.

In summary, designing representations for solids is a difficult job and compromises are often necessary. This course will only discuss the following representations: *wireframes*, *boundary representations* and *constructive solid geometry*.

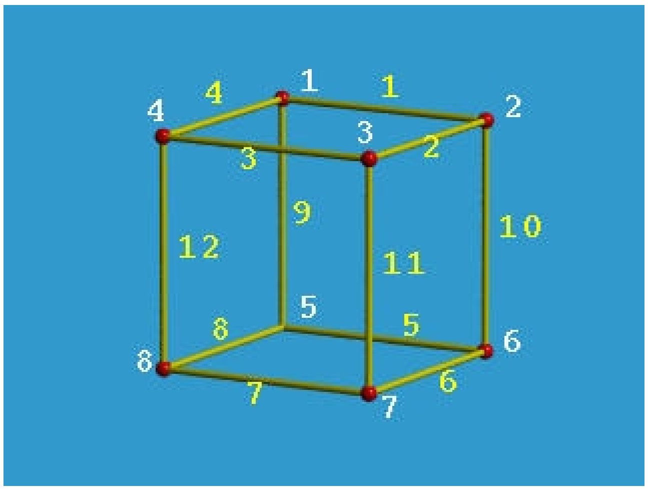
Wireframe Models

The wireframe model is perhaps the oldest way of representing solids. A wireframe model consists of two tables, the *vertex* table and the *edge* table. Each entry of the vertex table records a vertex and its coordinate values, while each entry of the edge table has two components giving the two incident vertices of that edge. A wireframe model *does not* have face information. For example, to represent a cube defined by eight vertices and 12 edges, one needs the following tables

Vertex Table			
Vertex #	***x***	***y***	***z***
1	1	1	1
2	1	-1	1
3	-1	-1	1
4	-1	1	1
5	1	1	-1
6	1	-1	-1
7	-1	-1	-1
8	-1	1	-1

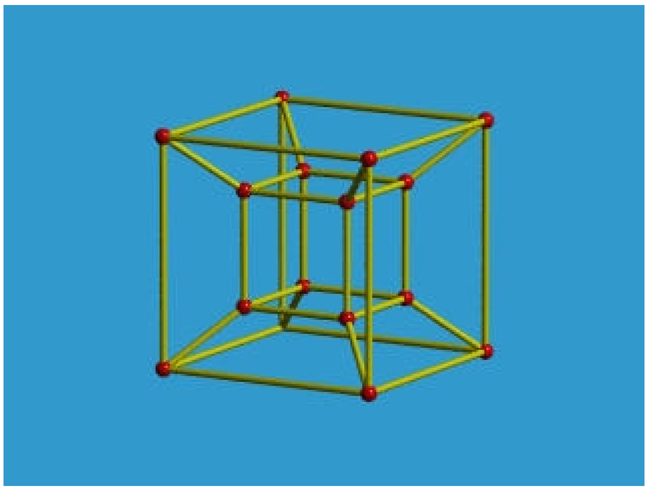
Edge Table		
Edge #	***Start Vertex***	***End Vertex***
1	1	2
2	2	3
3	3	4
4	4	1
5	5	6
6	6	7
7	7	8
8	8	5
9	1	5
10	2	6
11	3	7
12	4	8

The following figure shows all eight vertices and twelve edges and their numbers in white and yellow, respectively.

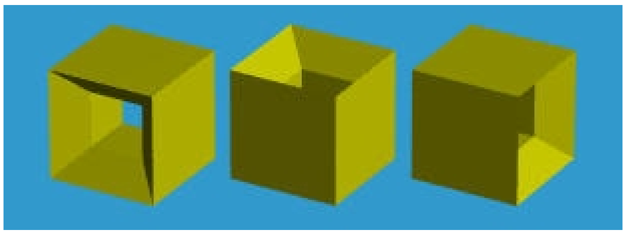


Wireframe Models Are Ambiguous

While wireframe uses the simplest data structures, it is ambiguous. The following is a well-known example that consists of 16 vertices and 32 edges. We know it represents a solid and each of the quadrilaterals (some of them are squares) defines a face of the solid.



The inner cube represents a hole; but, we cannot tell the direction of the opening of the cube. As shown in the following figure, there are three possibilities for this opening. While the other two can be obtained by rotating the remaining one, in general we will not be so lucky because the outer boundary may be a box of different side lengths and because this model is part of a big one which does not allow free interpretation.



Because of wireframe models are ambiguous, their uses are limited. However, wireframe models are popular, because they are efficient (*i.e.*, only vertices and edges are displayed and processed) when they work. For example, wireframe models can be used for preview purpose. Rendering a complex model or an animation sequence could be very time consuming if all objects are to be rendered. If wireframe models (usually including its face information) are available, one can easily obtain a general feeling of the final result without waiting for minutes or even hours before spotting a design flaw.

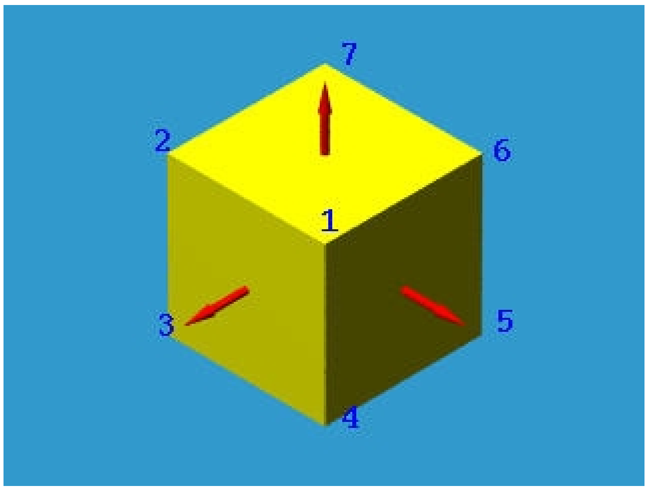
Note that the edges in a wireframe model do not have to be line segments. They can be curve segments and in this case the edge table will be more complicated since in addition to the two endpoints a description of the joining curve segment (*e.g.*, equation) is required.

Boundary Representations

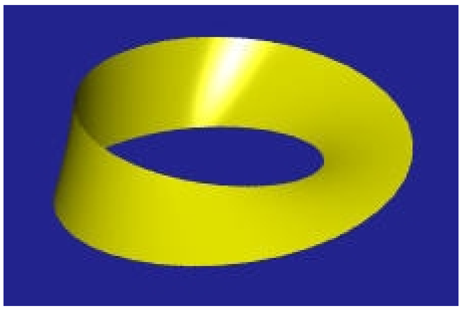
Boundary Representation, or *B-rep* for short, can be considered as an extension to the wireframe model. The merit of a B-rep is that a solid is bounded by its surface and has its *interior* and *exterior*. The surface of a solid consists of a set of well-organized faces, each of which is a piece of some surface (*e.g.*, a surface patch). Faces may share vertices and edges that are curve segments. Therefore, a B-rep is an extension to the wireframe model by adding face information to the latter.

There are two types of information in a B-rep: *topological* and *geometric*. *Topological* information provide the relationships among vertices, edges and faces similar to that used in a wireframe model. In addition to connectivity, topological information also include *orientation* of edges and faces. *Geometric* information are usually equations of the edges and faces.

The orientation of each face is important. Normally, a face is surrounded by a set of vertices. Using the right-handed rule, the ordering of these vertices for describing a particular face must guarantee that the normal vector of that face is pointing to the exterior of the solid. Normally, the order is *counter clockwise*. If that face is given by an equation, the equation must be rewritten so that the normal vector at every point on the part that is being used as a face points to the exterior of the solid. Therefore, by inspecting normal vectors one can immediately tell the inside and outside of a solid under B-rep. This orientation must be done for *all* faces. The following shows three faces and their outward pointing normal vectors. To describe the top surface, the vertices should be 6, 7, 2, 1 or 7, 2, 1, 6 or 2, 1, 6, 7 or 1, 6, 7, 2. To describe the left face, the order should be 1, 2, 3, 4 or 2, 3, 4, 1 or 3, 4, 1, 2 or 4, 1, 2, 3.



Unfortunately, not all surfaces can be oriented this way. If the surface of a solid can be oriented this way, it is called *orientable*; otherwise, it is *non-orientable*. The following shows the well-known Mobius band which is one-sided and non-orientable.

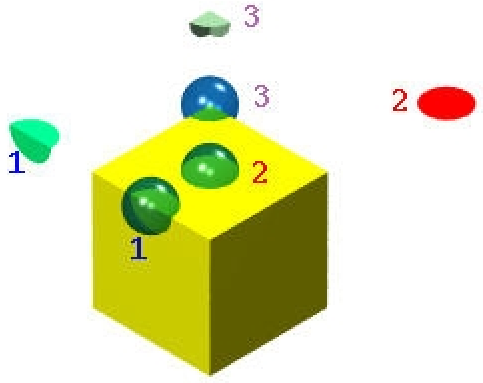


Manifolds

The surface of a solid must satisfy some conditions so that the resulting solid is well-behaved. This is the so-called manifold condition. While recent research has shown that the surface of a solid does not have to be a manifold, we shall restrict to manifold surfaces to simply our discussion.

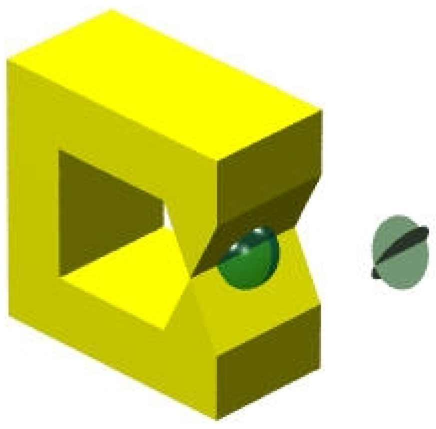
A surface is a *2-manifold* if and only if for each point x on the surface there exists an open ball with center x and sufficiently small radius so that the intersection of this ball and the surface can be continuously deformed to an open disk. An *open ball* with center at the coordinate origin and radius r is defined by equation $x^2 + y^2 + z^2 < r^2$. It contains all points inside of the sphere $x^2 + y^2 + z^2 = r^2$. An *open disk* is defined similarly $x^2 + y^2 < r^2$. By *continuously deformed*, it means one can twist or bend the shape without cutting (*i.e.* adjacency relations must be maintained) and gluing (*i.e.* a one-to-one relation is required).

Let us take a look at the following figure:



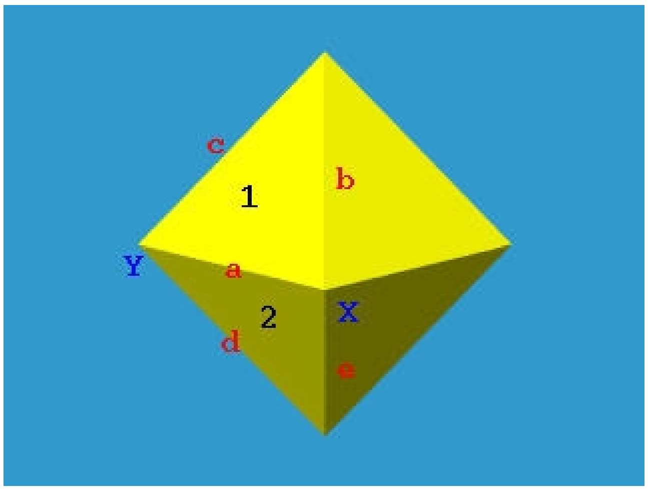
There is a cube and three open balls. Ball 2 has its center on the top face. The intersection of this ball and the surface of the cube is an open disk (shown in red). Ball 1 has its center on an edge. Its intersection with the surface of the cube is a "bent" open disk, which of course can be "unbent" to make it an open disk. Ball 3 has its center at a corner. Its intersection with the cube's surface is a three-way bent open disk.

The following shows a solid whose bounding surface is *not* a manifold. The intersection of the open ball and the surface of the solid is the union of two intersecting open disks. This intersection cannot be deformed to an open disk without "gluing." Consequently, the surface is *not* a manifold.



The Winged-Edge Data Structure

Perhaps the oldest data structure for a B-rep is Baumgart's *winged-edge* data structure. It is quite different from that of a wireframe model, because the winged-edge data structure uses *edges* to keep track almost everything. ***In what follows, we shall assume there is no holes in each face*** and later extend it to cope with holes. Moreover, we shall assume edges and faces are line segments and polygons. Topologically, one can always stretch curvilinear edges and faces so that they become flat without changing the relationships among them.

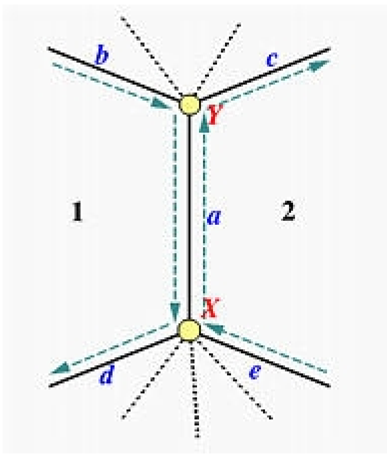


The above figure shows a polyhedron with vertices, edges and faces indicated with upper cases, lower cases and digits, respectively. Let us take a look at edge *a* = **XY**. This edge has two incident vertices **X** and **Y**, and two incident faces **1** and **2**. A face is a polygon surrounded by edges. For example, face **1** has its edges **a**, **c** and **b**, and face **2** has its edges **a**, **e** and **d**. Note that the ordering is clockwise viewed from outside of the solid. If the direction of the edge is from **X** to **Y**, faces **1** and **2** are on the right and left side of edge *a*, respectively. To capture the ordering of edges correctly, we need four more pieces of information. Since edge *a* is traversed once when traversing face **1** and traversed a second time when traversing face **2**, it is used twice in *different* directions. For example, when traversing the edges of face **1**, the predecessor and successor of edge *a* are edge **b** and edge **c**, and when traversing the edges of face **2**, the predecessor and successor of edge *a* are edge **d** and edge **e**. Note that although there are four edges incident to vertex **X**, only three of them are used when finding faces incident to edge *a*. Therefore, for each edge, the following information are important:

1. vertices of this edge,
2. its *left* and *right* faces,
3. the predecessor and successor of this edge when traversing its left face, and
4. the predecessor and successor of this edge when traversing its right face.

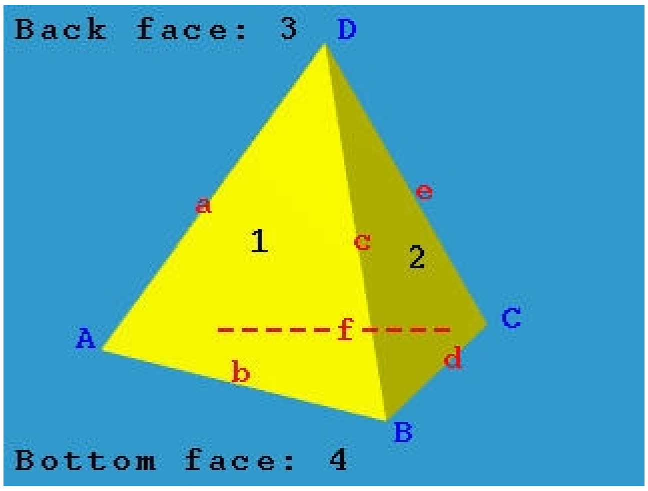
The Edge Table

Each entry in the edge table contains those information mentioned earlier: edge name, start vertex and end vertex, left face and right face, the predecessor and successor edges when traversing its left face, and the predecessor and successor edges when traversing its right face. Note that ***clockwise*** ordering (viewing from outside of the polyhedron) is used for traverse. Note also that the ***direction*** of edge *a* is from **X** to **Y**. If the direction is changed to from **Y** to **X**, all entries but the first one in the following table must be changed accordingly.



Edge	*Vertices*		*Faces*		*Left Traverse*		*Right Traverse*	
Name	*Start*	*End*	*Left*	*Right*	*Pred*	*Succ*	*Pred*	*Succ*
a	**X**	**Y**	1	2	*b*	*d*	*e*	*c*

The above shows the information for the entry of edge **a**. The four edges **b**, **c**, **d** and **e** are the *wings* of edge **a** and hence edge **a** is "winged."



The above is a tetrahedron with four vertices A, B, C and D, six edges a, b, c, d, e and f, and four faces 1, 2, 3 (back) and 4 (bottom). Its edge table is the following. Please use the above diagram to verify this table.

Edge	*Vertices*		*Faces*		*Left Traverse*		*Right Traverse*	
Name	*Start*	*End*	*Left*	*Right*	*Pred*	*Succ*	*Pred*	*Succ*

a	A	D	3	1	e	f	b	c
b	A	B	1	4	c	a	f	d
c	B	D	1	2	a	b	d	e
d	B	C	2	4	e	c	b	f
e	C	D	2	3	c	d	f	a
f	A	C	4	3	d	b	a	e

Other Tables

The winged-edge data structure requires two more tables, the *vertex table* and the *face table*. These two are very simple. The vertex table has one entry for each vertex which contains an edge that is incident to this vertex. The face table has one entry for each face which contains an edge that is one of this face's boundary edges. Therefore, we have the following table. Note that since there are multiple choices of edges, you may come up with different tables:

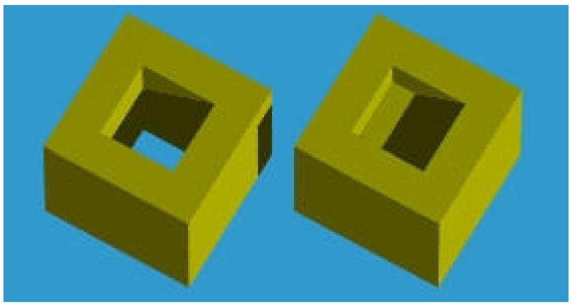
Vertex Name	*Incident Edge*
A	a
B	b
C	d
D	e

Face Name	*Incident Edge*
1	a
2	c
3	a
4	b

With this data structure, one can easily answer the question: which vertices, edges, faces are adjacent to each face, edge, or vertex. There are nine of these *adjacency relations*. For example, is vertex **X** adjacent to face 5? Are faces 3 and 5 adjacent to each other? The winged-edge data structure can answer these queries very efficiently and some of them may even be answered in constant time. However, it may take longer time to answer other adjacency queries. Note also that once the numbers of vertices, edges and faces are known, the size of all three tables are fixed and will not change.

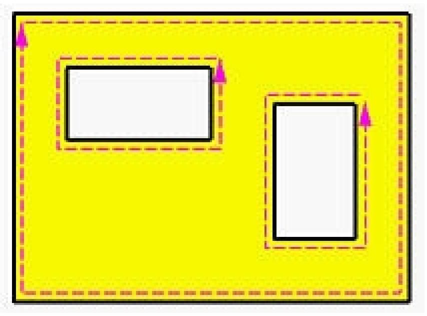
What If Faces Have Holes?

If some faces of a solid have holes, the above form of winged-edge data structure does not work. These holes may penetrate the solid (the left box below) or just like a pothole (the right box below).

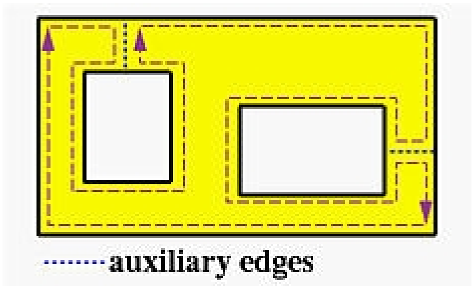


There are two ways for resolving this problem:

- For a face with inner loops, the outer boundary is ordered clockwise, while its inner loops, if any, are ordered counter clockwise.



- Another simple method is adding an *auxiliary* edge between each inner loop and the outer loop as shown below. This auxiliary edge will have the same face for its left and right faces. In this way, a face with holes becomes a single loop which can be represented with the winged-edge data structure. When traversing a loop, auxiliary edges can be identified easily since its left and right faces are the same.



The Euler-Poincaré Formula

The Euler-Poincaré formula describes the relationship of the number of vertices, the number of edges and the number of faces of a manifold. It has been generalized to include potholes and holes that penetrate the solid. To state the Euler-Poincaré formula, we need the following definitions:

- **V**: the number of vertices
- **E**: the number of edges
- **F**: the number of faces
- **G**: the number of holes that penetrate the solid, usually referred to as *genus* in topology
- **S**: the number of *shells*. A shell is an internal void of a solid. A shell is bounded by a 2-manifold surface, which can have its own genus value. Note that the solid itself is counted as a shell. Therefore, the value for **S** is at least 1.
- **L**: the number of loops, all outer and inner loops of faces are counted.

Then, the Euler-Poincaré formula is the following:

$$V - E + F - (L - F) - 2(S - G) = 0$$

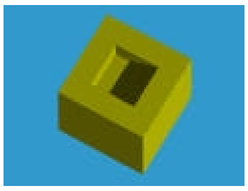
Examples

- A cube has eight vertices (**V** = 8), 12 edges (**E** = 12) and six faces (**F** = 6), no holes and one shell (**S**=1); but **L** = **F** since each face has only one outer loop. Therefore, we have

$$V-E+F-(L-F)-2(S-G) = 8-12+6-(6-6)-2(1-0) = 0$$

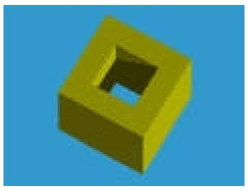
- The following solid has 16 vertices, 24 edges, 11 faces, no holes, 1 shell and 12 loops (11 faces + one inner loop on the top face). Therefore,

$$V-E+F-(L-F)-2(S-G) = 16-24+11-(12-11)-2(1-0)=0$$



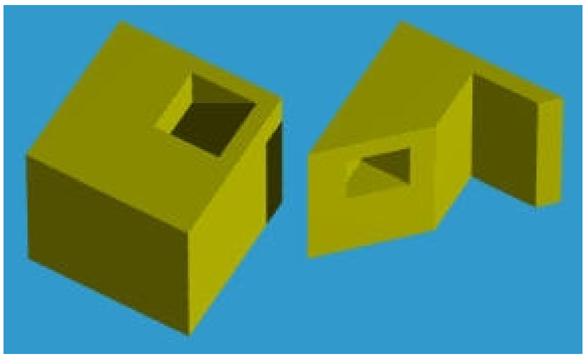
- The following solid has 16 vertices, 24 edges, 10 faces, 1 hole (*i.e.*, genus is 1), 1 shell and 12 loops (10 faces + 2 inner loops on top and bottom faces). Therefore,

$$V-E+F-(L-F)-2(S-G) = 16-24+10-(12-10)-2(1-1)=0$$



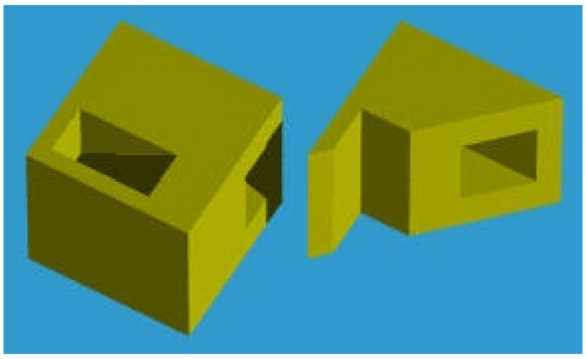
- The following solid has a penetrating hole and an internal cubic chamber as shown by the right cut-away figure. It has 24 vertices, 12*3 (cubes) = 36 edges, 6*3 (cubes) - 2 (top and bottom openings) = 16 faces, 1 hole (*i.e.*, genus is 1), 2 shells and 18 loops (16 faces + 2 inner loops on top and bottom faces). Therefore,

V-E+F-(L-F)-2(S-G) = 24-36+16-(18-16)-2(2-1)=0

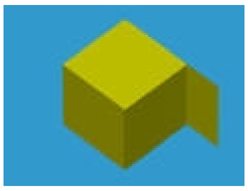


- The following solid has two penetrating holes and no internal chamber as shown by the right cut-away figure. It has 24 vertices, 36 edges, 14 faces, 2 hole (*i.e.*, genus is 2), 1 shells and 18 loops (14 faces + 4). Therefore,

V-E+F-(L-F)-2(S-G) = 24-36+14-(18-14)-2(1-2)=0



Part of the information recorded in a B-rep is topological (*i.e.*, adjacency relations). Invalid solids may be generated if the representation is not carefully constructed. One way of checking this topological invalidity is to use the Euler-Poincaré formula. If its value is not zero, we are sure something must be wrong in the representation. However, this is only a *one-side* test. More precisely, a zero value of the Euler-Poincaré formula does not mean the solid is valid.



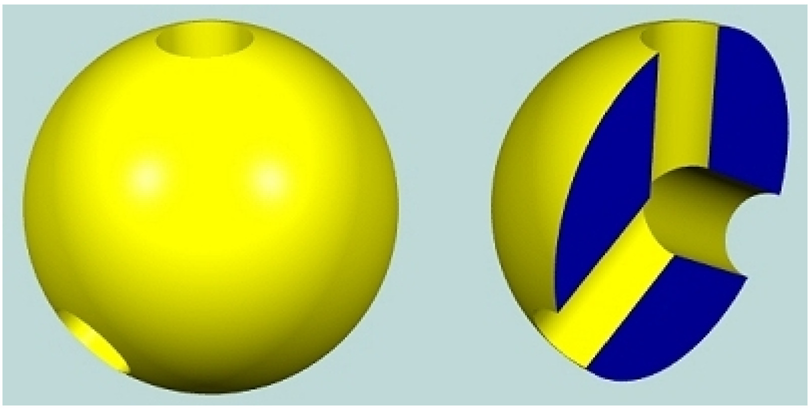
The figure above has a box and an additional sheet which is simply a rectangle. This object has 10 vertices, 15 edges, 7 faces, 1 shell and no hole. Its loop number is equal to the number of faces. The value of the Euler-Poincaré formula is zero as shown below,

V-E+F-(L-F)-2(S-G) = 10-15+7-(7-7)-2(1-0)=0

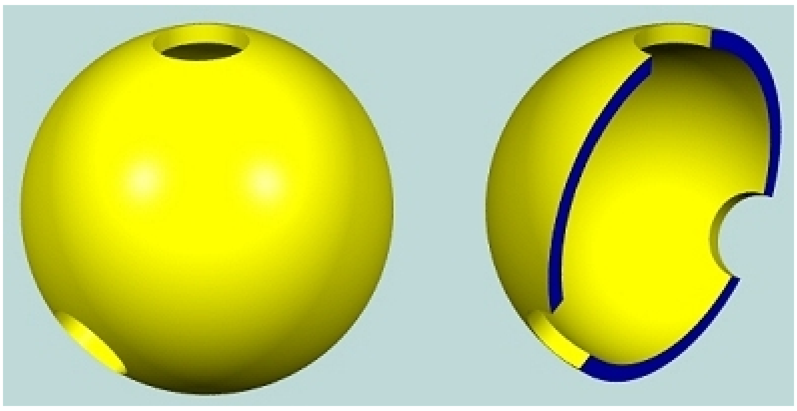
but this is *not* a valid solid! Therefore, if the value of Euler-Poincaré formula is non-zero, the representation is definitely not a valid solid. However, the value of the Euler-Poincaré formula being zero does not guarantee the representation would yield a valid solid.

How to count genus correctly?

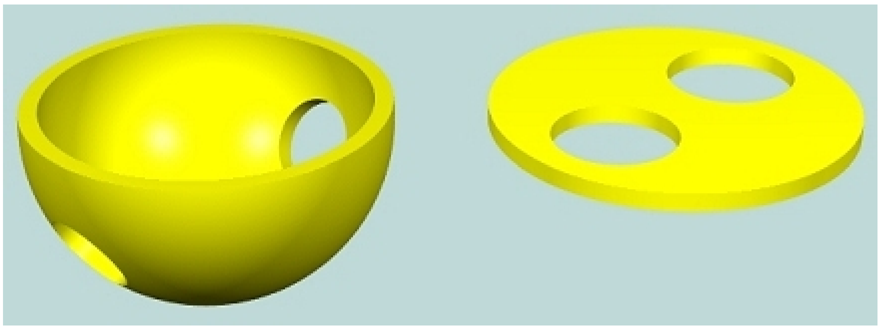
It is not always easy to count genus **G** correctly. For example, suppose we have a sphere punched by three tunnels as shown below. The left object shows the outside look. The right one shows the inside by cutting half of the sphere off. What is the genus value **G**? We know that the genus value counts the number of penetrating holes. But, in this example, it is somewhat ambiguous. In fact, by combining any two center-going tunnels we will have three penetrating holes. However, this is incorrect!



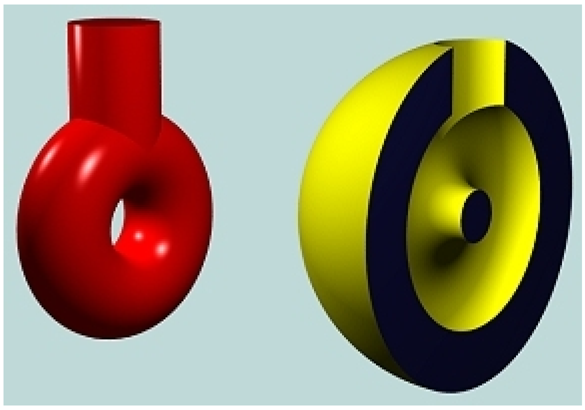
The Euler-Poincaré Formula describes the topological property amount vertices, edges, faces, loops, shells and genus. Any topological transformation applied to the model will *not* alter this relationship. Intuitively, applying topological transformations means we can twist, stretch and squash the model but we cannot cut some parts off nor glue some parts together. Let us apply some intuitive topological transformations to this model for computing the genus. We can push the "walls" surrounding the three tunnels so that the interior of the model becomes a thin shell. This is shown in the image below.



Stretch the top hole so that it is large enough (below left). Then, collapse the top portion to flatten the model. This is shown in the right model below. How many penetrating holes are there? Two! Therefore, **G** is 2.



Sometimes, penetrating holes may appear in an unlikely situation. Consider the following model which is obtained by taking out a torus and tube from the interior of a sphere. What is the genus of this model? It does not look like there is a penetrating hole. So, is genus equal to 0? In fact, **G** = 1! Figure it out yourself. Do some twisting, stretching and squashing.



Euler Operators

Once a polyhedron model is available one might want to edit it by adding or deleting vertices, edges and faces to create a new polyhedron. These operations are called ***Euler Operators***. However, it has been shown that in the process of editing a polyhedron with Euler operators, some intermediate results may not be valid solids at all.

Recall from the discussion of the Euler-Poincaré formula that the following holds for all polyhedra:

$$\mathbf{V - E + F - (L - F) - 2(S - G) = 0}$$

where **V**, **E**, **F**, **L**, **S** and **G** are the numbers of vertices, edges, faces, loops, shells and genus, respectively. Based on this relation, some Euler operators have been selected for editing a polyhedron so that the Euler-Poincaré formula is always satisfied. There are two groups of such operators: the Make group and the Kill group. Operators start with **M** and **K** are operators of the Make and Kill groups, respectively.

Euler operators are written as **Mxyz** and **Kxyz** for operations in the Make and Kill groups, respectively, where **x**, **y** and **z** are elements of the model (*e.g.*, a vertex, edge, face, loop, shell and genus). For example, **MEV** means adding an edge and a vertex while **KEV** means deleting an edge and a vertex.

It has been proved by Mantyla in 1984 that Euler operators form a complete set of modeling primitives for manifold solids. More precisely, every topologically valid polyhedron can be constructed from an initial polyhedron by a finite sequence of Euler operators. Therefore, Euler operators are powerful operations.

The Make Group of Euler Operators

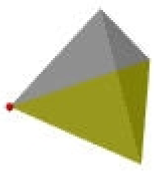
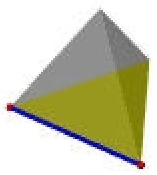
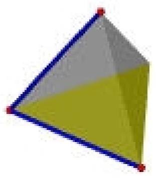
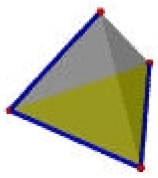
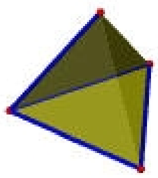
The Make group consists of four operators for adding some elements into the existing model creating a new one, and a Make-Kill operator for adding and deleting some elements at the same time. Here are the operators:

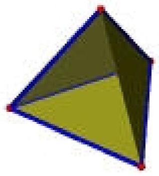
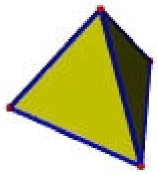
Operator Name	***Meaning***	***V***	***E***	***F***	***L***	***S***	***G***
MEV	Make an edge and a vertex	+1	+1				
MFE	Make a face and an edge		+1	+1	+1		
MSFV	Make a shell, a face and a vertex	+1		+1	+1	+1	
MSG	Make a shell and a hole					+1	+1

MEKL	Make an edge and kill a loop		+1		-1		

The above table shows the change of values of **V**, **E**, **F**, **L**, **S** and **G**. Note that adding a face produces a loop, the outer loop of that face. Therefore, when **F** is increased, **L** should also be increased. This new loop and the new face will cancel each other in the subexpression **L - F**. Please verify that none of these operators would cause the Euler-Poincaré formula to fail.

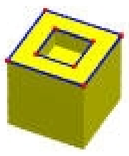
The following table illustrates the way of using Euler operators to construct a tetrahedron. Vertices, edges, and faces are in red, blue and green; the only shell is in transparent gray. The first step uses **MSFV** to obtain a shell with a face and a vertex. The next three steps use **MEV**s, each of which adds an edge and a vertex. The last three steps use **MFE**, each of which adds a face and an edge. Thus, in seven steps or seven Euler operators a tetrahedron is constructed.

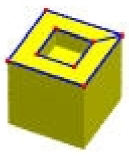
Operator Name	*Meaning*	*V*	*E*	*F*	*L*	*S*	*G*	*Result*
MSFV	Make a shell, a face and a vertex	+1		+1	+1	+1		
MEV	Make an edge and a vertex	+1	+1					
MEV	Make an edge and a vertex	+1	+1					
MEV	Make an edge and a vertex	+1	+1					
MFE	Make a face and an edge		+1	+1		+1		

MFE	Make a face and an edge		+1	+1		+1		
MFE	Make a face and an edge		+1	+1		+1		

MSG simply makes a shell with a hole. After this, one can add vertices, edges, faces, loops. There must be loops, because the new hole penetrates at least one face.

MEKL makes an edge and at the same time kills a loop. A commonly used **MEKL** is adding an edge connecting the outer loop and the inner loop of a face. In this case, the number of edges **E** is increased by 1 and the number of loops **L** is decreased by 1 since that loop is *killed*. The left figure below shows two loops of the top face while the right one shows the new edge added after performing a **MEKL**. Thus, the inner and outer are "joint" together with the new edge becoming a single face.





The Kill Group of Euler Operators

The Kill group just performs the opposite of what the Make group does. In fact, replacing the **M** and **K** in all Make operators with **K** and **M**, respectively, would get the operators of the Kill group. Therefore, the Kill group consists of the following five operators:

Operator Name	*Meaning*	*V*	*E*	*F*	*L*	*S*	*G*
KEV	Kill an edge and a vertex	-1	-1				
KFE	Kill a face and an edge		-1	-1	-1		
KSFV	Kill a shell, a face and a vertex	-1		-1	-1	-1	
KSG	Kill a shell and a hole					-1	-1
KEML	Kill an edge and make a loop		-1		+1		

With these operators, one can start with a tetrahedron and reduce it to nothing. Since these operators are the opposites of the Make operators, we shall not go further here.

Constructive Solid Geometry

Constructive Solid Geometry, or *CSG* for short, is yet another way of representing solids. A CSG solid is constructed from a few *primitives* with *Boolean* operators (*i.e.*, set union, intersection and difference). Thus, a CSG solid can be written as a set equations and can also be considered a design methodology.

CSG Primitives

The standard CSG primitives consist of the block (*i.e.*, cube), triangular prism, sphere, cylinder, cone and torus. These six primitives are in some *normal* or *generic* form and must be *instantiated* by the user to be used in his/her design. Moreover, the instantiated primitive may require transformations such as scaling, translation and rotation to be positioned at the desired place.

Suppose the block primitive is defined by its "lower left" corner < -1, -1, -1 > and "upper right" corner < 1, 1, 1 >. To produce a rectangular box with center at < 3, 2, 3 > and height and width 3 and length 5, a user may first scale the block primitive 1.5 times in the *y*- and *z*-direction and 2.5 times in the *x*-direction, and then translate the result to < 3, 2, 3 >. If the block primitive is called **Block** in a CSG system, the result may be obtained as follows:

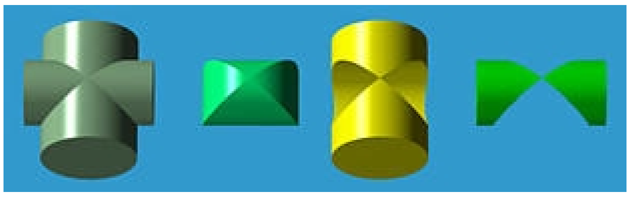
translate(scale(Block, < 2.5, 1.5, 1.5 >), < 3, 2, 3 >)

In the above, the object to be transformed and the transformation data are the first and second arguments, respectively.

Boolean Operators

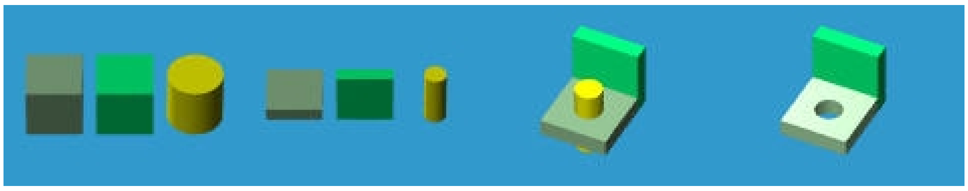
We can combined two instantiated and perhaps transformed primitives into one with set union, set intersection and set difference operators. However, simple set operators may create problems as will be discussed in regularized Boolean operators, modifications are required. Let us just use set operations on this page.

Given two sets, ***A*** and ***B***, its union consists of all points from either ***A*** or ***B***; its intersection consists of all points in *both* sets; and its difference, written as ***A* - *B*** (*resp.*, ***B* - *A***), consists of all points in ***A*** but not in ***B*** (*resp.*, in ***B*** but not in ***A***). In the following, ***A*** is the vertical cylinder and ***B*** is the horizontal cylinder. From left to right, the four solids are the union and intersection of ***A*** and ***B***, ***A* - *B*** and ***B* - *A***.



Therefore, a solid can be considered as the result of applying Boolean operators to a set of instantiated and transformed CSG primitives.

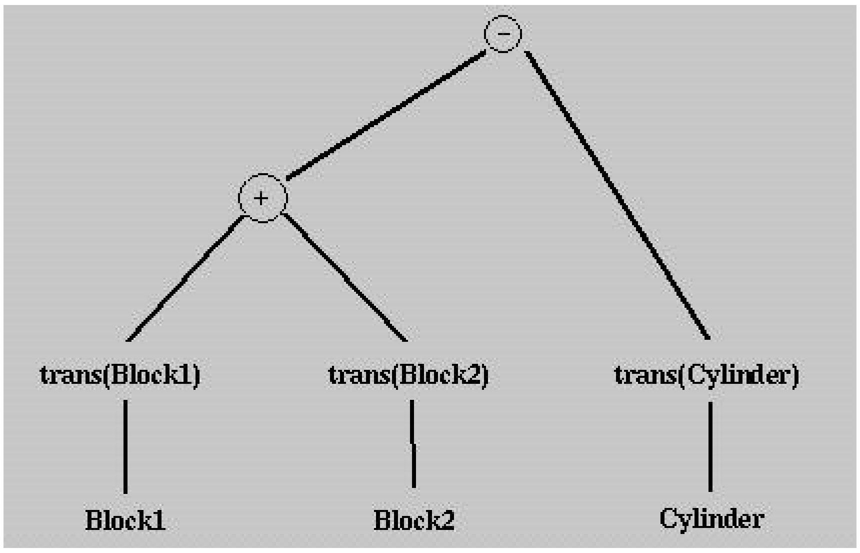
Let us take a look at a simple example. We want to design a bracket-like shape with a hole shown on the right-most figure below. We start with two instantiations of blocks and one instantiation of a cylinder (the left-most figure). Then, the two blocks are scaled and one of them is rotated to a vertical position. The cylinder is also scaled so that its radius matches that of the hole. These three instantiations are than transformed to their desired positions. The final product is obtained by computing the union of the two blocks and then subtracting from it the cylinder.



Please note that the design of the above solid is ***not*** unique. For example, the L shape can be constructed from subtracting a cube from another one.

CSG Expressions

The design procedure of the above bracket can be written as an expression:
diff(union(trans(Block1), trans(Block2)), trans(Cylinder))
where **union(*A*,*B*)** and **diff(*A*,*B*)** are the union and difference of ***A*** and ***B***, and **trans()** indicates appropriate transformations. Or, if we use +, ^ and **-** for set union, intersection and difference, the above function calls can be rewritten as a set expression as follows:
(trans(Block1) + trans(Block2)) - trans(Cylinder)
This expression can be converted to an expression tree, the *CSG Expression*, of the design:



In fact, every solid constructed using the CSG technique has a corresponding CSG expression which in turn has an associated CSG tree. The expression of the CSG tree is a representation of the final design. Recall that the same solid may have different CSG expressions/trees. For example, one might punch a hole from **Block1** first and then compute the union of this result with **Block2**. As a result, *CSG representations are not unique*.

Interior, Exterior and Closure

We need the concept of interior, exterior and closure to fully appreciate the discussion of regularized Boolean operators. Intuitively, the interior of a solid consists of all points lying inside of the solid; the closure consists of all interior points and all points on the solid's surface; and the exterior of a solid is the set of all points that do not belong to the closure.

Consider a sphere, $x^2 + y^2 + z^2 = 1$. Its interior is the set of all points that satisfy $x^2 + y^2 + z^2 < 1$, while its closure is $x^2 + y^2 + z^2 <= 1$. Therefore, the closure is the union of the interior and the boundary (its surface $x^2 + y^2 + z^2 = 1$). Obviously, its exterior is $x^2 + y^2 + z^2 > 1$.

A solid is a three-dimensional object and so does its interior and exterior. However, its boundary is a two-dimensional surface.

Formal Definitions

Recall that the open ball with center (a,b,c) and radius r consists of all points that satisfy the following relation:
$(x - a)^2 + (y - b)^2 + (z - c)^2 < r^2$
A point ***P*** is an *interior point* of a solid ***S*** if there exists a radius r such that the open ball with center ***P*** and radius r is contained in the solid ***S***. The set of all interior points of solid ***S*** is the *interior* of ***S***, written as **int(*S*)**. Based on this definition, the interior of an open ball is the open ball itself.

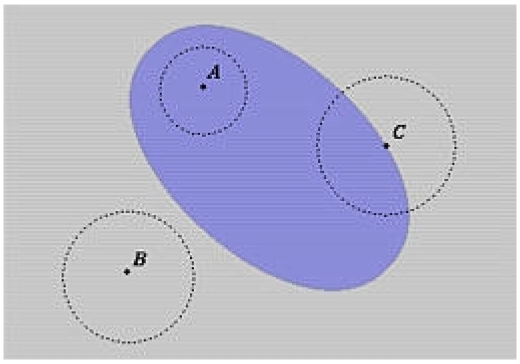
On the other hand, a point ***Q*** is an *exterior point* of a solid ***S*** if there exists a radius r such that the open ball with center ***Q*** and radius r does not intersect ***S***. The set of all exterior point of solid ***S*** is the *exterior* of solid ***S***, written as **ext(*S*)**.

Those points that are not in the interior nor in the exterior of a solid ***S*** constitutes the *boundary* of solid ***S***, written as **b(*S*)**. Therefore, the union of interior, exterior and boundary of a solid is the whole space.

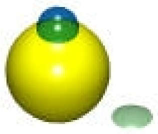
The *closure* of a solid ***S*** is defined to be the union of ***S***'s interior and boundary, written as **closure(*S*)**. Or, equivalently, the closure of solid ***S*** contains all points that are not in the exterior of ***S***.

Examples

Here is an example in the plane. You should change all open balls to open disks. Point ***A*** is an interior point of the shaded area since one can find an open disk that is contained in the shaded area. Similarly, point ***B*** is an exterior point. Point ***C*** is a boundary point because whatever the radius the corresponding open ball will contain some interior points and some exterior points.

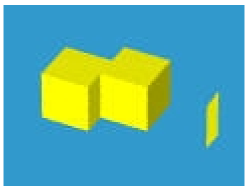


Note that a surface (a two-dimensional object) is never a solid (a three-dimensional object). In fact, a surface does not have any interior point. Take any point of the surface (see figure below), the open ball with arbitrary radius and center at that point always intersects the sphere in an open disk (in pale green in the lower right corner). Therefore, no open ball can be contained in the sphere, and, as a result, that point is not an interior point of the sphere. Thus, we conclude that a surface does not have any interior point.



Regularized Boolean Operators

We certainly expect that the union, intersection and difference of two solids is a solid. Unfortunately, in many cases this is not always true. In the following figure, two cubes touch each other and their intersection is a rectangle shown on the right. A rectangle is not a three-dimensional object and hence not a solid!



To eliminate these lower dimensional branches, the three set operations are *regularized* as follows. The idea of regularing is very simple.

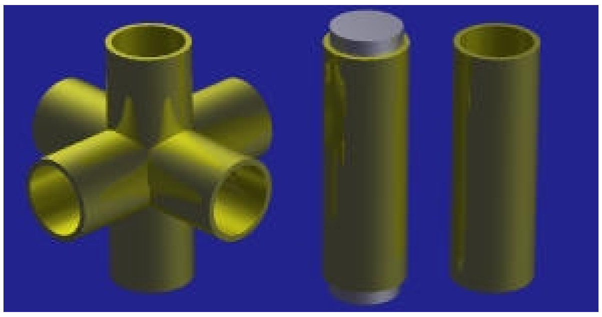
- Compute the result as usual and lower dimensional components many be generated.
- Compute the interior of the result. This step removes all lower dimensional components. An example has been shown in previous page. The result is a solid without its boundary.
- Compute the closure of the result obtained in the above step. This adds the boundary back.

Let +, ^ and **-** be the *regularized* set union, intersection and difference operators. Let ***A*** and ***B*** be two solids. Then, ***A* + *B***, ***A* ^ *B*** and ***A* - *B*** can be defined mathematically based on the above description:

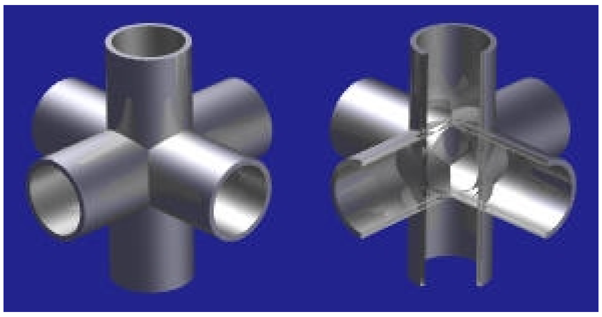
A* + *B = **closure(int(**the set union of ***A*** and ***B*)**
A* ^ *B = **closure(int(**the set intersection of ***A*** and ***B*)**
A* - *B = **closure(int(**the set difference of ***A*** and ***B*)**
where **closure()** and **int()** are the *closure* and *interior* discussed in previous page. Based on this definition, the intersection of the two cubes shown at the beginning of this page is empty. As mentioned earlier, the set intersection of these two cubes is a rectangle, which is a two dimensional object and has no interior. Hence, after taking interior (*i.e.*, **int()**), we get an empty set, whose closure is also empty. Consequently, the intersection is empty.

A CSG Design Example

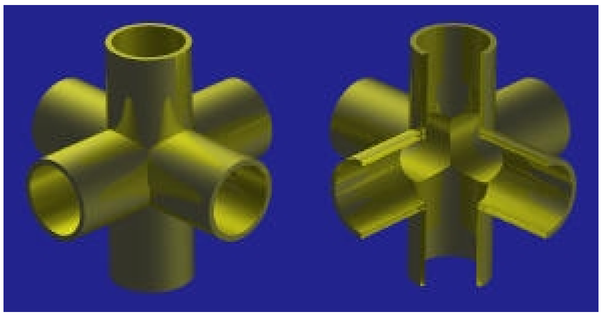
Let us deign three perpendicular tubes as shown below left.



An immediate reaction is that let us subtract a smaller cylinder from a larger one, yielding a tube which is shown in below right. Then, take two more instances of this tube, each of which is rotated an appropriate angle; compute the union of these three tubes; and the result is shown in the figure below.

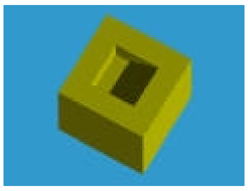


But, the result is not quite right, because the inner junction of the tubes are blocked as seen in the right figure above. A correct solution is to design two instances of three perpendicular cylinders, one larger and the other smaller. Then, subtracting the smaller from the larger yields the designed result as shown below.

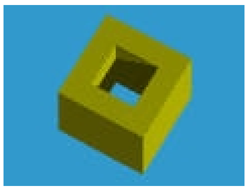


Problems

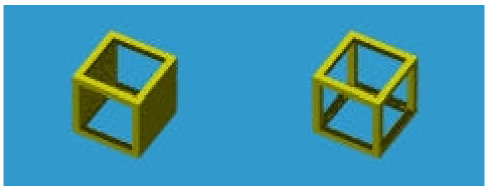
1. Given a winged-edge representation of a solid, design algorithms to answer the following queries:
   - list all edges that contain a given vertex
   - list all faces that contains a given vertex
   - list all vertices and edges of a given face
2. Find the winged-edge data structure of a cube.
3. Find the winged-edge data structure of the following solid:



4. Find the winged-edge data structure of the following solid:



5. Verify Euler-Poincaré formula with the following polyhedra. List the number of vertices, edges, faces, loops, shells and holes.



6. Use Euler operators to construct the following polyhedra: (1) a cube, (2) a cube with "pothole" and (3) a cube with a penetrating hole. Figures for the last polyhedra can be found in the winged-edge problems.
7. One can define a new operator **MEFL** in the Make group. This operator add an edge, a face and a loop. In general, this new edge subdivides a face into two, thus creating a new face and a new loop. In the following, the left one is a cube with vertices and edges of the top face shown. The right one is the result of apply an **MEFL** to the top face by adding an diagonal edge. The original square face is replaced by two triangles and **F** is increased by

one. Similar, **L** is also increased by 1. Note that the value of the Euler-Poincaré formula does not change.

One can use this operator **MEFL** to subdivide all faces into triangles.

Use **MEKL** and **MEFL** to make all faces of a cube with a penetrate hole triangles. In the final result, do you have to worry about loops?

8. Starting with nothing, use Euler operators to construct a cube.

References

Most of this week's material are from Requicha's paper and Hoffmann's book. Requicha's paper is quite readable. Braid's short paper contains a brief history of boundary representation and Mantyla's book contains a detailed discussion of B-rep and Euler operators and their applications.

- Ian C. Braid, Boundary Modeling, in *Fundamental Development of Computer-Aided Geometric Modeling*, edited by Les Piegl, Academic Press, 1993, pp. 165-183.
- Christoph M. Hoffmann, *Geometric & Solid Modeling: An Introduction*, Morgan Kaufmann, San Mateo, California, 1989.
- Martti Mantyla, *An Introduction to Solid Modeling*, Computer Science Press, Rockville, Maryland, 1988.
- Michael E. Mortenson, *Geometric Modeling*, John Wiley & Sons, 1985.
- A. A. G. Requicha, Representations for Rigid Solids: Theory, Methods, and Systems, *ACM Computing Surveys*, Vol. 12 (1980), pp. 437-464.