Constructive Solid Geometry for Triangulated Polyhedra

Philip M. Hubbard

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-90-07
September 1, 1990

Constructive Solid Geometry for Triangulated Polyhedra

Philip M. Hubbard

Department of Computer Science
Brown University
Providence, RI 02912 *

November 16, 1991

Abstract

Triangulated polyhedra are simpler to process than arbitrary polyhedra for many graphics operations. Algorithms that compute the boundary representation of a constructive solid geometry (CSG) model, however, may perform poorly if the model involves triangulated polyhedral primitives. A new CSG algorithm specifically tailored to triangulated primitives is presented. The key features of this algorithm are its global processing of intersections between polyhedra and its avoidance of ray-casting when classifying polyhedra against one another. The new algorithm is shown to perform substantially better than one published algorithm, and arguments are presented suggesting its benefits over several others in the context of an interactive modeling environment.

Keywords: constructive solid geometry, Boolean set operations, solid modeling, polyhedra, triangulation.

1 Introduction

An interactive modeling system should provide its users with intuitive yet powerful operations for building solid objects. One group of operations that has proven successful is the Boolean set operations: union, intersection and difference. When used to combine primitive solids such as cubes, spheres or tori, these operations allow a variety of complicated models to be specified easily.

The constructive solid geometry (CSG) modeling paradigm formalizes this idea. A CSG model is a tree whose leaves are primitive solids and whose interior nodes are *regularized*

*This work was sponsored in part by grants from IBM, Sun Microsystems, NCR, Hewlett-Packard and Digital Equipment Corporation

Boolean set operations [3] on their children. A regularized Boolean set operation (denoted $\cup^*$, $\cap^*$ or $-^*$) on two solids is defined to be the closure of the corresponding ordinary Boolean set operation ($\cup$, $\cap$ or $-$) applied to the interiors of the two solids. Intuitively, regularizing a Boolean operation on solids prevents that operation from producing a degenerate solid, such as a point, line or plane. As an example, note that the intersection of two cubes that touch only along one edge is a line; the regularized intersection is empty space, a more desirable result.

Any system that creates a CSG model must be able to produce an image of the object it represents. There are two general approaches to the production of this image. One approach involves two steps: evaluate the polyhedral boundary representation (b-rep) of the CSG model, then render its polygonal faces with an ordinary renderer [7, 9, 13, 14]. The other approach does not create an intermediate representation of the CSG tree, but instead produces the image directly from the tree with a special CSG rendering algorithm [6, 8]. To understand the relative merits of the two methods, note that there are two distinct classes of changes that can be made to a CSG model. *Moving* a model consists of applying a single linear transformation to every leaf solid of the CSG tree; the name was chosen because translation is an obvious example of this class of changes. The other class of changes consists of any operation that is not an example of moving, such as the application of a transformation to only some of the leaves or the removal of any of the nodes of the tree; making such a change will be called *restructuring* the model. A direct CSG rendering algorithm must be invoked if either type of change occurs, because the fact that the model has changed at all requires that its image be recomputed. A b-rep evaluation algorithm, however, needs to be called only if the model has been restructured; moving a model can be accomplished by applying the linear transformation to the current b-rep and invoking the ordinary renderer.

If either imaging method could handle both classes of changes in real time, the distinction between the classes would be irrelevant. But neither method is sufficiently fast in all situations. The best performance is obtained by picking the method most appropriate for the changes to be made. Evaluating a b-rep can take a long time; even the efficient algorithms, such as the one published by Thibault and Naylor [13], can handle only certain kinds of restructuring changes quickly. CSG renderers generally take less time to produce one image than is required to construct an elaborate b-rep. The time required still prohibits true interactive performance in most cases, though; the rendering algorithm described by Goldfeather *et al.* [6], produces images at interactive rates only when rendering models of modest complexity on a high-end multiprocessor graphics system like the Pixel-Planes system [4]. One advantage of the b-rep method is that it can have good amortized performance in some common situations. These situations occur whenever the number of times a model is moved exceeds the number of times it is restructured, such as when a user "walks around" her model to admire from several angles the results of a restructuring operation. Most graphics workstations can transform and render a b-rep very quickly, so the time lost to restructuring can be made up while moving. Until high-end graphics systems become commonplace and affordable, a modeling system should offer its users the choice of using direct CSG rendering or b-rep evaluation. A user will then be able to choose between the image-generation methods based on the modeling actions she anticipates making.

The remainder of this paper focuses on the evaluation of CSG b-reps. In particular, it discusses the subtleties involved when b-reps are triangulated polyhedra, as in the latest version of the Brown Animation Generation System (BAGS). A new algorithm is presented which extends the algorithm of Laidlaw *et al.* [9] (designed for general polyhedra) to process the special case of triangulated polyhedra efficiently. The new algorithm is shown to provide a significant performance improvement over the Laidlaw *et al.* algorithm. The new algorithm is then compared to several other b-rep evaluation algorithms, and some of its advantages are discussed.

2 Triangulated Polyhedra

A *triangulated polyhedron* differs from a *general polyhedron* only in that its faces must be triangles. Notice that several faces may be adjacent and coplanar, so when their edges are not visible they may appear to be one face. For example, the simplest representation of a cube as a triangulated polyhedron consists of twelve triangular faces, with one pair of faces forming each of the sides of the cube. We reserve the term "face" to describe the triangles that compose a triangulated polyhedron, and do not use the term in the informal sense to mean a "side" or a "facet" of a polyhedron.

Triangulated polyhedra offer a number of advantages over general polyhedra. The data structures that represent a general polyhedron are complicated by the fact that each face can involve an arbitrary number of vertices and edges. This problem does not arise in the data structures for a triangulated polyhedron, since each of its faces is defined by exactly three vertices and edges. The triangulated polyhedra used in BAGS are represented by the TRIP ("TRIangular Polyhedra") package. TRIP represents each polyhedron as three arrays of structures, one each for vertices, edges and faces. The only structure which must have variable length is the structure for a vertex, as it contains a reference to each of the arbitrary number of edges incident on that vertex. The face structure has references to the three edges that define the face. The edge structure contains references to the two vertices at its endpoints and to the two faces adjoining it. When more than two faces must adjoin an edge (i.e. when the polyhedral boundary is not a 2-*manifold*), multiple coincident edges are used. Arbitrary application-specific data can be associated with vertices, edges, faces or face-vertex pairs. An example of the last type of data is vertex normal vectors for Gouraud shading that preserves edges. Management of this type of data is made easier by the restriction to triangular faces, because the data can be stored as a three-element array associated with each face.

Polygon normal vectors, used extensively in a variety of graphics operations, can easily be computed for triangulated polygons. The direction of the normal comes from the cross-product of the vectors defined by any two of the triangle's edges. This procedure cannot be used for a general polygon because some of its edges may be colinear. A normal vector computed this way is guaranteed to be correct across the whole surface of the triangle, a fact that is not true for general polygons since they can be warped. Finally, many algorithms, both software and hardware, that operate on polygons can be made simpler if the polygons are triangles [15]; examples are rendering [1, 2, 5] and collision detection [10].

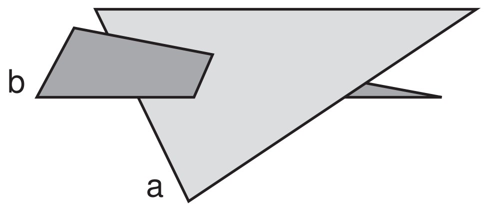

Figure 1: The intersection of two three-dimensional faces.

3 Algorithm Overview

One might expect that computing triangulated polyhedral b-reps from CSG models would also be easy and efficient, but care must be taken to choose an efficient algorithm. An algorithm designed for general polyhedra will not necessarily perform well on triangulated polyhedra. We came to this unpleasant conclusion when we began using only triangulated polyhedra in BAGS and noticed that the the b-rep evaluation algorithm of Laidlaw *et al.* performed poorly. Investigation suggested that the degraded performance is a result of the way in which intersections between polyhedra are handled by that algorithm.

The Laidlaw *et al.* algorithm processes the interior nodes of a CSG tree one at a time, starting immediately above the leaves and working towards the root. By the time the algorithm reaches a given interior node ν, the algorithm will have already computed the b-reps that represent the subtrees rooted at ν's children. Call these b-reps A and B. The algorithm must then combine A and B according to the regularized Boolean set operation associated with ν. The result will be the b-rep that represents the tree rooted at ν. If, for instance, the Boolean operation is regularized intersection, the new b-rep should contain the faces from A which are inside B and vice versa (the treatment of coincident faces is ignored for the sake of clarity). Unfortunately, there may be faces from one polyhedron that are partially inside *and* partially outside the other polyhedron. To deal with this situation, all the faces from one polyhedron that intersect faces from the other must be found. These faces must then be split along their lines of intersection to make new faces that are completely inside or completely outside the other polyhedron (or completely on its boundary). So the algorithm that processes one interior node has two phases: the *splitting phase* compares each face from A with each face from B and splits them if they intersect, and then the *classification phase* decides which faces belong in the new b-rep and removes those that do not belong.

The main weakness of the Laidlaw *et al.* algorithm is the purely local approach used by its splitting phase. Even though a face from A might intersect several faces from B, each of these intersections is processed independently. By handling each intersection independently, extra intersections are created. Consider a face a from A that intersects a face b from B, as illustrated in Figure 1. When the splitting phase processes this intersection, it must split a and b along their *intersection segment*, the line segment along which they intersect. Making only this one split will leave a "slit" in a and break b into a triangle and a trapezoid, neither of which are allowable results in polyhedra whose faces can be only triangles. So a must be replaced by a collection of new triangular faces, two of which have the intersection segment

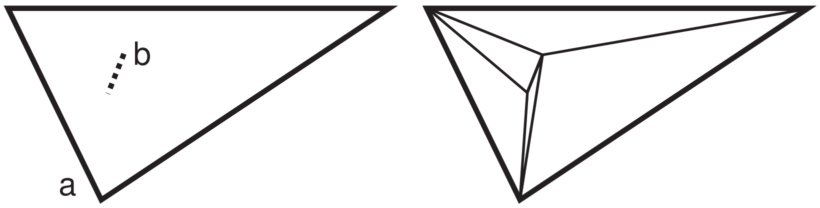

Figure 2: The intersection segment for faces a and b, and one set of new faces that can replace a to eliminate the intersection.

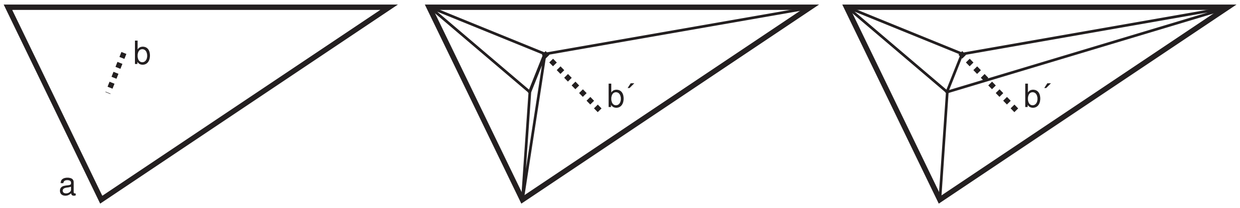

Figure 3: Two different constrained planar triangulations of face a that could result from intersection between a and face b, and the intersection of face b' with each of them.

as an edge; b must be replaced in an analogous manner. Figure 2 shows a and one set of new faces that can replace it. The new faces that replace an old face form a *constrained planar triangulation* of the old face, meaning that they include four *constraint edges*: the three edges that bound the old face and the edge corresponding to the intersection segment. A given set of constraint edges may allow more than one constrained planar triangulation of a face. For example, Figure 3 shows two possible triangulations of a that could be produced when the intersection with b is found. This figure also depicts what will happen when another face b' from B is discovered to intersect A. Notice that b' intersects one face produced by the first triangulation, but it intersects two faces produced by the second triangulation. Since each such intersected face will have to be triangulated, choosing the second triangulation for the intersection with b will result in more faces after the intersection with b'. A local splitting algorithm will not know about the intersection with b' when it is processing the intersection with b. The local algorithm therefore cannot choose a triangulation that will reduce the number of faces produced after future intersections. This approach tends to create b-reps that contain more faces than are necessary.

The degraded performance of the Laidlaw *et al.* algorithm is a result of these extraneous faces. If ν, the node just processed, is not the root of the CSG tree, the new b-rep just created will be combined with the b-rep for ν's sibling node. The extraneous faces in ν's b-rep will have to be compared against the faces of this other b-rep. The Laidlaw *et al.* algorithm finds intersections by comparing each face from one b-rep with every face from the other b-rep, so extra faces cause a significant number of unnecessary comparisons to be performed. As will be explained later, a comparison is a non-trivial operation, so the extra comparisons will waste a great deal of time. Extraneous faces would also be created if the algorithm were run on polyhedra whose faces are not restricted to be triangles, but our experience indicates that the problem is most severe for triangulated polyhedra. Triangular faces cause

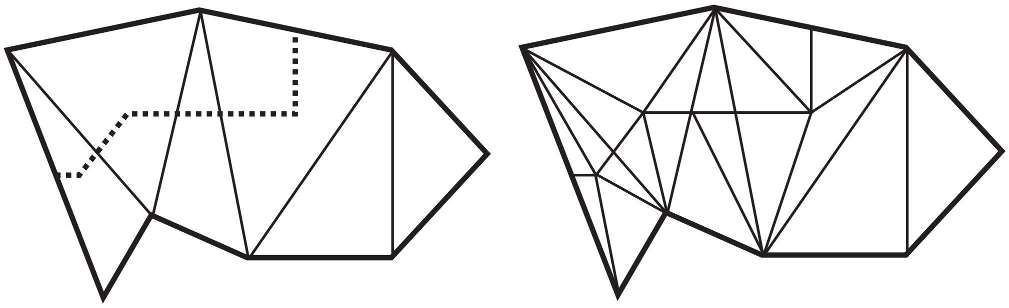

Figure 4: A cluster with several intersection segments, and the results of splitting the faces of the cluster individually.

particular problems because splitting one into a new set of triangular faces creates more faces than would be created if the original face could be split into arbitrary polygons.

4 An Improved Splitting Phase

To improve the performance of the algorithm, we have devised a new, more global splitting-phase algorithm that does not create extraneous faces. The key to this algorithm is that it does not split faces a and b as soon as the intersection between them is discovered. Instead, all splitting is postponed until after all intersections have been found. Each face is then split into the minimum number of faces needed to keep it from penetrating the other polyhedron.

This approach can still create extraneous faces, however, in an area of a polyhedron where there is a *cluster* of adjacent coplanar faces. An example of such a cluster is any side of a cube, which is represented as two right triangles sharing the same hypotenuse. Clusters partition the edges of a polyhedron into two classes: each *interior edge* separates two faces within a cluster, and each *exterior edge* separates one cluster from another. In the cube example, the shared hypotenuse edge is an interior edge, while the four edges that constitute the perimeter of the side are all exterior edges. The exterior edges are the only ones that actually matter because they define the overall shape or *boundary* of the cluster; the interior edges were added (probably arbitrarily) to break that boundary down into triangles. If the constrained triangulation mentioned earlier is performed on individual triangular faces, the interior edges act as extra constraints that cause more new faces to be generated. A better alternative is to use only exterior edges and intersection segments as constraints. Figure 4 depicts a cluster that has been intersected by several faces from the other polyhedron, and shows how splitting the cluster's faces individually results in extraneous faces. Figure 5 shows that fewer faces are produced if interior edges are ignored. In essence, the splitting that is performed when all intersections have been found should be done as if the faces were the polygons defined by the boundaries of the clusters, and not the triangles that belong to the clusters.

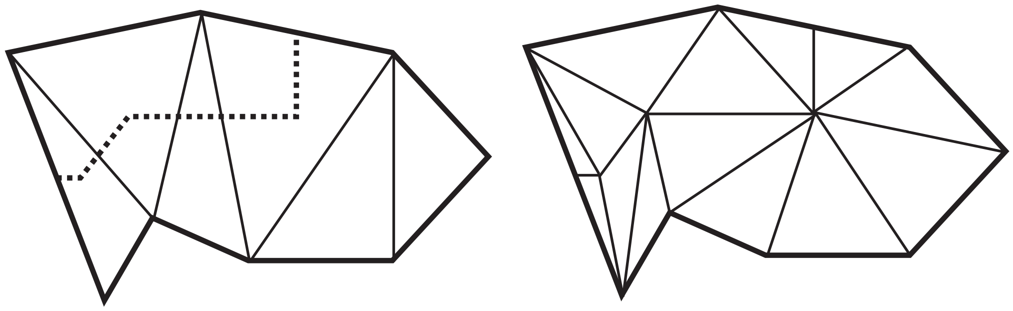

Figure 5: A cluster with several intersection segments, and the results of splitting the cluster as a whole (ignoring interior edges).

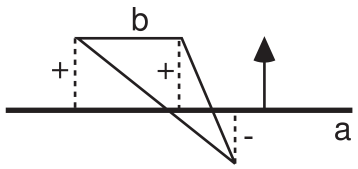

Figure 6: Signed distances from the vertices of face b to the plane of face a.

4.1 Finding Intersections

The fact that the faces are indeed triangular can be exploited when finding intersections. Unlike the polygons defined by the cluster boundaries, triangles are guaranteed to be convex, so two of them can have only one intersection segment. The new splitting phase therefore performs its intersection calculations on pairs of triangular faces, using a simpler algorithm than could be used if the faces were not known to be triangles and multiple disjoint intersection segments had to be handled. Since the intersection algorithm used by Laidlaw *et al.* was also designed to process convex polygons, the new algorithm is a variation on that one. Simplifications were made, however, to exploit the restriction of polygons to triangles.

The first step of the intersection algorithm for faces a and b involves the *extent* of each face, the minimum-volume axis-aligned cuboid that encloses that face. If the extents of a and b do not intersect, then a and b themselves cannot intersect. This test is valuable because it prevents time from being wasted on further processing of faces that cannot possibly intersect. This test is also very quick since the extent of each face can be precomputed once and associated with the face as TRIP data. If the extents of a and b do intersect, the next step is to find the *signed distance* from each vertex of a to the plane of b, and vice versa. The sign of a signed distance is positive only if the vertex is in front of the plane of the other face, i.e., on the side pointed to by the plane normal vector, as shown in Figure 6. A signed distance is computed by projecting the vector from the origin to the vertex (i.e., its coordinates) onto the outward-pointing normal vector of the plane. Intersection processing halts if all of the vertices from a have distances of the same sign, and likewise for b. Hence, no further time is spent on faces that are not pierced by the plane of the other face, or on faces that are coplanar.

To see why processing can stop if a and b are coplanar, consider the faces that are adjacent to a and b. Let a' be the face that adjoins a along an edge e that intersects b. Obviously, a' intersects b and the intersection occurs along edge e. If a' is not coplanar with a, then a' is also not coplanar with b. So even if the processing of a and b is stopped, the processing of a' and b will not be stopped and the intersection along e will eventually be found. Now consider the case in which a' is coplanar with a. The faces a and a' will be in the same cluster c_A, and e will be an interior edge by definition. Cluster c_A overlaps some or all of the cluster c_B that contains b. Intersection segments will be introduced into c_A from intersections with the faces of B that are adjacent to c_B. These intersection segments will partition c_A into the region that overlaps c_B and the regions that do not overlap it. Intersection segments created by the faces adjacent to c_A will partition c_B in an analogous fashion. Since e intersects b, e is in the region of c_A that overlaps c_B. When this region is triangulated, using e as a constraint might create extraneous triangles because e is an interior edge; the triangulation of the overlap region is analogous to the triangulation of a cluster, discussed earlier, on page 6. So the intersection processing of a and b should be stopped to prevent a constraining intersection segment corresponding to e from being created. Note that this policy will allow the region of c_A that overlaps c_B to be triangulated independently of the region of c_B that overlaps c_A. These two regions coincide, but the triangles produced by the two triangulations will not necessarily coincide. Non-coinciding triangles do not cause a problem, though, because regularized Boolean set operations will be used to combine A and B into a new polyhedron. As discussed in more detail in Section 5, each regularized operation is defined so that the part of the new polyhedron corresponding to the coinciding region is made from faces of A only.

If the signed distances indicate that a and b may intersect, the exact line segment σ_a along which a intersects b's plane must be found, and also the segment σ_b where b intersects a's plane. Both σ_a and σ_b will lie along the same line L, whose direction is computed by taking the cross-product of the plane normals of a and b. An endpoint for a segment—without loss of generality, assume the segment is σ_a—can be either a vertex of a or a position on an edge of a. A vertex of a is an endpoint of σ_a if the vertex lies in the plane of b. An edge of a contains an endpoint of σ_a if the vertices at the two ends of that edge have signed distances of different signs. The exact point q at which σ_a intersects the edge can be found easily because the distance from one of the edge's endpoint vertices to the plane of b is proportional to the distance along the edge from that vertex to q. The endpoints of σ_a and σ_b are stored as distances along L with respect to some reference point p on L; Segal and Sequin [12] sketch a clever way of setting up a system of equations that allows an appropriate p to be found without inverting a matrix.

Now the intersection segment σ_{ab} between the faces a and b can be found: it is the intersection of σ_a and σ_b. This intersection may be a line segment, a single point or empty. If σ_{ab} is non-empty, it must be associated with both a and b so it can be used later as a constraint for the triangulation (splitting) of their clusters. Our implementation uses ordinary TRIP vertex structures to represent the endpoint(s) of σ_{ab}. Each end of σ_{ab} is represented by two vertex structures with identical coordinates, one in A and one in B. If an endpoint of σ_{ab} touches an existing vertex of either face—without loss of generality, assume it is a—the vertex structure will already be present and associated with a. If not, a "floating" vertex is created; this vertex is not connected to any edges, but is referenced by a pointer in the TRIP data associated with a.

To represent the span between the endpoints of a σ_{ab} that is a line segment, a data structure called a *link* is used. Each polyhedron gets its own copy of the link, which represents the edge that will be created at the site of σ_{ab} when the triangulation is performed some time in the future. A link is referenced by the TRIP data associated with the vertices at each end of σ_{ab}. As a result, a link corresponding to σ_{ab} with an endpoint v will be implicitly connected to any link representing another intersection segment ending at v. An example of two such adjacent intersection segments are those created in face a by faces b and b', as depicted in Figure 3 above. The routines that perform the triangulation rely on this connectivity being maintained. The triangulation routines also need to know which intersection segments touch exterior edges. This connectivity can be represented with links if a link is created for each exterior edge. These edge-links are created in a preprocessing step. Whenever an intersection segment is found to touch an exterior edge, a new vertex will be introduced at that point. The one link corresponding to that edge then needs to be split into two links, and references to the two new links must be given to the new vertex. The intersection segment's link will already be referenced by that vertex, so the connectivity will be complete. So by the time the triangulation routines are called, all the constraints for the triangulation are expressed as links, simplifying the data structure manipulations needed in those routines. Once the triangulation is complete, the links are no longer necessary, and the memory they occupy can be freed.

Not all the vertices created to represent the endpoints of intersection segments are actually necessary. Consider two coplanar faces b and b' that intersect a face a. Even though the union of the intersection segments σ_{ab} and $\sigma_{ab'}$ is one continuous line segment that could be represented by just two vertices and one link, the separate processing of the two intersections will introduce an unnecessary third vertex between the segments. This vertex is called an *interior vertex* because it is superfluous in a manner analogous to an interior edge, that is, it is not an essential constraint on the triangulation that will be generated. We call interior vertices and edges *interior features* and call exterior edges and their endpoint vertices *exterior features*. The most obvious way an interior vertex can be created is by the intersection of an interior feature from one face with a part of another face that is not an exterior feature. For example, Figure 7 shows an interior vertex p created by the intersection of an interior edge of B (an interior feature) and the interior area of a face from A (not an exterior feature). The figure also shows an exterior vertex q. Vertex q exemplifies the way an intersection involving an exterior feature (in this case, an exterior edge) almost always creates an exterior vertex. There is, however, one type of intersection between an interior feature and an exterior feature that will produce an interior vertex. If *both* endpoints of the intersection segment σ_{ab} lie on the same exterior edge of A, then either endpoint will produce an interior vertex if that endpoint lies on an interior feature of B. Two examples of this situation are illustrated in Figure 8. Interior vertices may also be present in the definition of a primitive polyhedral solid. A polygonalized cylinder, for example, may be defined to have an interior vertex at the center of each of its flat ends if the triangles that define those ends were created in a "pieces-of-pie" pattern.

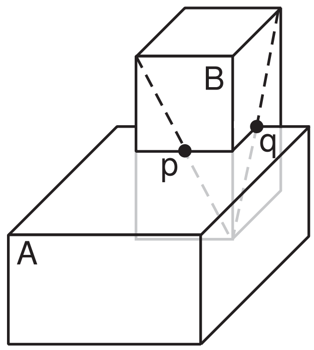

Figure 7: Vertex p is an interior vertex and vertex q is an exterior vertex. (For clarity, all interior edges except those creating p or q have been hidden.)

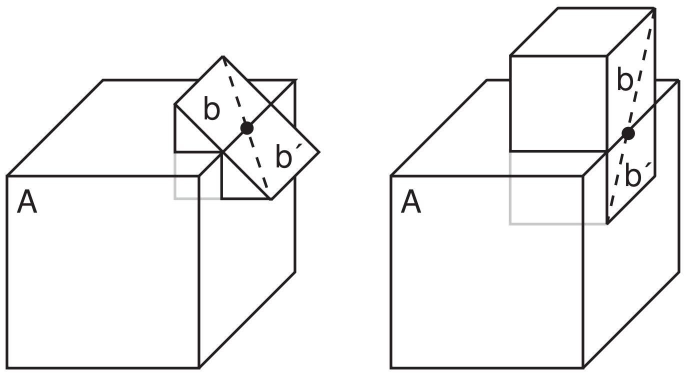

Figure 8: Two intersections that produce an interior vertex, denoted by the large black dot. (For clarity, all interior edges have been hidden except for the edge between b and b' that produced the interior vertex.)

Since interior vertices put unnecessary constraints on the triangulation, they should be removed before it is performed. The simplest way to deal with interior vertices is to flag them any time they are detected, but wait to remove them until after all intersections have been found. Interior vertices that are inherent in a primitive polyhedral solid can be detected in a preprocessing step; the key observation is that any vertex adjoining only interior edges can be considered an interior vertex. Interior vertices created by the ends of intersection segments can be detected when the segments are created, by means of the criteria mentioned in the previous paragraph. It is important to realize that a vertex flagged by the preprocessing step may need to be un-flagged as the result of an intersection; this situation occurs if the vertex is touched by an exterior feature of another polyhedron. After all intersections have been found, the interior vertices can be removed. The removal of an interior vertex requires that the two adjacent links for which it is an endpoint be coalesced into one link. The links processed by the triangulation routine, therefore, will be free of all internal vertices and the unnecessary constraints that correspond to them.

4.2 Splitting Clusters By Triangulation

Once all the links have been introduced and connected and the interior vertices removed, the actual splitting can be performed. As mentioned earlier, the splitting is an instance of a constrained triangulation problem. The constraints are the links corresponding the exterior edges of a cluster and to the intersection segments. The goal is to tile the interior of the polygon defined by the exterior edges with the minimum number of triangles such that each intersection segment is an edge between some pair of triangles. Because the exterior edges and intersection segments all lie in the plane of the cluster, we have a constrained *planar* triangulation problem.

Rather than devise our own algorithm to solve this problem, we implemented one described by Preparata and Shamos [11]. It involves two phases, which will be sketched only briefly here. The first phase builds a set of *monotone polygons* from the original polygon and the constraint edges. A monotone polygon can be put into an orientation such that any line perpendicular to a *monotone axis* intersects the polygon in no more than two places. Intuitively, this property means that the polygon cannot "double back on itself" in the direction of the monotone axis. The advantage of monotone polygons is that they are easier to triangulate than arbitrary polygons, as is exploited by the second phase of the algorithm. This phase triangulates each monotone polygon individually by stepping along its vertices in the direction of its monotone axis. Edges are added from the current vertex to a previously visited vertex as long as that edge crosses no existing edges. The fact that the polygon is monotone ensures that this procedure will result in a valid triangulation of the polygon. The total time required to perform this algorithm is $O(n \log n)$, where n is the number of unique vertices at the ends of the constraining links.

5 The Classification Phase

As mentioned earlier, the local splitting phase of the original Laidlaw *et al.* algorithm was primarily responsible for its poor performance on triangulated polyhedra. Once we implemented our new splitting-phase algorithm, the performance improved, and execution profiles indicated that the splitting phase was no longer the dominant factor in the overall run time. The classification phase then became the bottleneck, but fortunately the restriction to triangular faces provided an opportunity to optimize this phase of the algorithm as well.

The classification phase of the Laidlaw *et al.* algorithm is based on ray casting. To classify a face a as being inside, outside or on the surface of a polyhedron B, a ray is cast from the barycenter of a in the direction of its outward-pointing normal vector. Of all the faces from B pierced by the ray, the face b closest to a is chosen, and the orientation of b's normal vector is used to classify a. For example, if b's normal points towards a, then a is in front of b and hence outside B; if b's normal points away from a, a is inside B. This classification scheme works only if b is the closest face in front of a. If B is a doughnut, for instance, and a is inside B with its normal pointing towards the hole, the ray will pierce a face b on the near side of the hole and a face b' on the far side; if the normal of b' were used, it would be found to be pointing towards a, and a would be incorrectly classified as being outside of B.

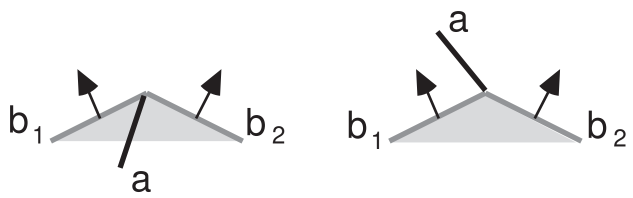

Figure 9: Classification of face a when faces b_1 and b_2 define a convex surface. (The plane of each face extends into the page. The arrows represent face normals, and the light shading represents the interior of polyhedron B.)

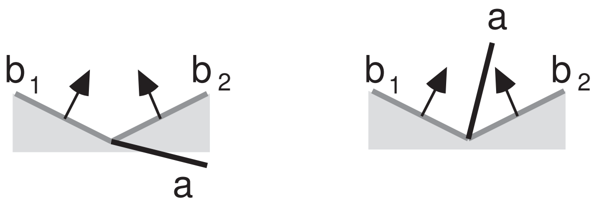

Figure 10: Classification of face a when faces b_1 and b_2 define a concave surface. (The conventions of Figure 9 still hold.)

To make sure the face closest to a is found, the ray casting must check *every* face of B to see if it is pierced by the ray. The ray-casting step is therefore computationally expensive. The expense can be amortized somewhat, however, by propagating the classification of a face to its neighbors in its polyhedron. This propagation can continue as long it does not cross the surface of the other polyhedron (because classification changes at this point). Even with this amortization, the ray-casting step is expensive enough that the classification phase is quite slow.

For the case in which an edge e_a of a touches the surface of B, however, the ray-casting step is unnecessary. This edge of a will correspond to an edge e_b between two faces b_1 and b_2 from B because the splitting phase has already been performed. Since a is a triangle, only one of its vertices, v, will not be on e_a. The position of v with respect to b_1 and b_2 completely determines the classification of a, assuming that B is a 2-manifold (i.e. b_1 and b_2 are the only faces that adjoin edge e_b and there are no other edges completely coincident with e_b in B). The classification depends on whether b_1 and b_2 locally define a convex surface or a concave surface. If they are convex, then a is inside B if the normal vectors of *both* b_1 and b_2 point away from v, as illustrated in Figure 9. If they are convex, then a is inside B if the normal vector of *either* b_1 or b_2 point away from v, as shown in Figure 10. A special case occurs if v lies in the plane of either b_1 or b_2. In this case, a is on the surface of B if a overlaps either b_1 or b_2; if not, one of the two cases mentioned above will hold. The special case is depicted in Figure 11. If a is coplanar with either face from B, the normal vectors of the two faces are compared to determine if they point the same or opposite directions. This comparison is necessary to determine which of the coplanar faces belong in the result of the Boolean set operation, as will be explained later in this section. The two faces b_1 and

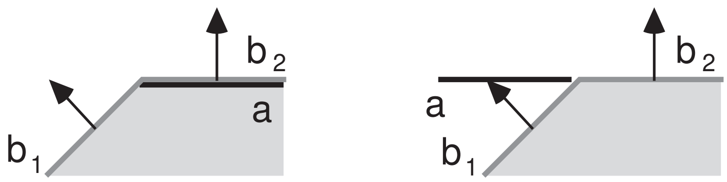

Figure 11: Classification of face a when its vertex v lies in the plane of face b_2. (The conventions of Figure 9 still hold.)

	Faces from A				*Faces from B*			
operation	*inside B*	*outside B*	*same*	*opposite*	*inside A*	*outside A*	*same*	*opposite*
$A \cup^* B$	no	yes	yes	no	no	yes	no	no
$A \cap^* B$	yes	no	yes	no	yes	no	no	no
$A -^* B$	no	yes	no	yes	yes	no	no	no

Figure 12: The classifications of faces that belong in the results of the regularized Boolean set operations.

b_2 used in these tests can be found in constant time if appropriate steps are taken during the splitting phase to remember what faces from A intersect faces from B, and vice versa. Hence, the classification of a with this algorithm is substantially faster than if ray casting is used.

Fortunately, the case in which a face touches the surface of the other polyhedron is quite common. A number of faces can be classified quickly using the algorithm just described. Propagation can then be used to classify most of the remaining faces that do not touch the surface. Only two situations are not directly handled by this approach. The first occurs if a surface that is not a 2-manifold is being simulated by the use of coincident edges. In this situation, the edge e_a corresponds to more than just one edge from B, so the classification of a is dependent on all the faces from B that adjoin those edges. The easiest way to deal with this situation is to skip a and allow it to be classified by propagation from one of its neighbors; in the highly unlikely event that a cannot receive its classification by propagation, it must be classified by ray casting. The other situation occurs if the combination of two polyhedra produces no intersections whatsoever, for instance when a cube is unioned with a sphere that completely surrounds it. Ray casting must be used to classify one face of each polyhedron in this situation, and then propagation will classify all the rest. The situations that require ray casting occur so infrequently, though, that they do not adversely affect the performance of the new classification phase.

Once the faces of A and B have been classified, A and B can be combined. The regularized Boolean set operation being performed defines which faces from A and B belong in the combination. Figure 12 shows which faces are retained by each of the operations. Faces that are classified as "same" or "opposite" correspond to areas of A and B that are coincident, as mentioned earlier. We choose to retain only the faces from A in these situations. It would also be possible to retain only the faces from B, but not a mixture of faces from the two

Object	CSG *nodes*	*Laidlaw et al. algorithm* *Faces*	*Time (in seconds)*	*New algorithm* *Faces*	*Time (in seconds)*	*Speedup factor*
handle	2	262	10.64	84	0.65	16.4
glasses	14	4333	385.46	692	9.24	41.7
fan	17	2036	88.69	1363	13.43	6.6
couch	44	8825	2138.03	1357	27.31	78.3
android	72	18602	3953.52	4734	79.17	49.9

Figure 13: A comparison of the performance of the Laidlaw *et al.* algorithm and the new algorithm. The tests were performed on a Sun SPARCstation 330GX with 40 megabytes of main memory and 327 megabytes of swap space.

polyhedra. This restriction is the result of the choice to ignore coplanar intersections during the splitting phase. One final detail concerns the difference operation. The faces from B that are retained after taking a difference must have their orientations reversed, because the interior of B corresponds to the exterior of $A \cap^* B$.

6 Performance

The precise asymptotic behavior of a CSG algorithm is difficult to characterize accurately because it is highly dependent on the pattern of intersections between the polyhedra being combined. In order to compare our algorithm with the Laidlaw *et al.* algorithm, we resorted to empirical data. Figure 13 shows the execution time of the two algorithms on five CSG models of varying complexity. All the models except the first were produced by artists for use in animated films, so they represent the applications for which a CSG system would actually be used in a production environment. Images of the models can be found in Figures 14 through 18. As the execution times indicate, the new algorithm yields a dramatic improvement over the old algorithm.

Roughly 30.5 ± 6.6 percent of the time attributed to our algorithm in Figure 13 is spent by the TRIP package in maintaining the polyhedral representation of the model. This figure seems exorbitant, especially because TRIP wastes much of its time through inefficient memory management. Work is underway to streamline TRIP, and the performance of our algorithm should improve as a result.

The speedup value for the "fan" model appears anomalous because it is smaller than that obtained for the "glasses" model, despite having more nodes in its CSG tree. There are more intersections among the polyhedra in the "glasses" model, however, so there are more opportunities for the improved splitting algorithm to save time. Another example of this phenomenon can be seen by comparing the performance on the "couch" and "android" models.

7 Comparison to Other Approaches

We have found only one published account of a b-rep evaluation algorithm specifically designed to process triangles. Yamaguchi and Tokieda [14] describe an algorithm which performs Boolean operations on general polyhedra after first triangulating them. The triangulation is done because it is easier to find intersections between triangles than between arbitrary concave polygons, as we indicated earlier. Although no performance data is given, the authors claim that the simplified intersection processing makes their algorithm faster than the other algorithms then extant. When the splitting and classifying is finished, a general polyhedron is built by essentially throwing away all the interior edges of the triangles. Because the triangulated representation is purely temporary, a local splitting algorithm is used and the extraneous triangles produced are not considered to be a problem. It would not be appropriate, however, to return the triangulated representation for situations (such as ours) in which a triangulated polyhedron must be produced, because the extraneous triangles then would become a problem.

The other b-rep evaluation algorithms found in the literature could be used if a post-processing step were added to triangulate the polyhedra they produce. One recent algorithm, devised by Goodrich [7], uses ideas from the theory of parallel processing to compress the CSG tree being processed. The classification phase then uses this "dwarf" CSG tree to determine efficiently which faces belong in the final b-rep. Goodrich's analysis of his algorithm indicates that it has good asymptotic performance, and the algorithm should work well when the b-reps of static models are needed. Adapting the algorithm to produce high throughput for dynamically-changing models appears difficult, however, so the algorithm does not lend itself to interactive modeling environments. The problem is that the compression of the CSG tree makes it difficult to explicitly produce b-reps for all the tree's subtrees as a byproduct of evaluating the b-rep for the entire tree. An algorithm (like ours) that does produce these subtree b-reps can arrange to have them cached (as memory permits). If a user then decides to restructure the CSG tree by changing a node ν, the only nodes whose b-reps must be re-evaluated are the ancestors of ν; all other b-reps can be retrieved from the cache and reused immediately. Experience with our BAGS modeling system has indicated that the savings in processing time provided by exploiting *coherency* in this manner can be very significant.

An algorithm more appropriate for an interactive modeling system was published by Thibault and Naylor [13]. This algorithm uses binary space partitioning (BSP) trees to represent polyhedra as intersections of sets of half-spaces. A BSP tree representing a CSG model is constructed by combining the b-rep for a leaf (primitive object) with the BSP tree representing the portion of the model already computed, repeating this process until all the leaves have been added. The combining of a b-rep with a BSP tree can be done quickly because intersection calculations are performed between polygons and planes instead of between pairs of polygons, allowing a more efficient intersection algorithm to be used. The inherent space-partitioning properties of BSP trees also prevent time from being wasted in attempting to find intersections between spatially distant polygons and planes. Execution profiles of our algorithm suggest, however, that the improvement in intersection-processing performance offered by BSP trees might not significantly reduce the overall execution time. Our algorithm spends about 17.0 ± 8.4 percent of its time performing intersection calculations, indicating that they are no longer the dominant factor in its overall performance.

BSP trees have the disadvantage that they frequently cannot represent polyhedra as compactly as b-reps, because the tree that represents a polyhedron with n faces may require considerably more than n nodes to handle concavity. Another disadvantage is the extra time required to convert a polyhedron between its BSP tree representation and its b-rep, a conversion which is necessary because many other graphics operations require the b-rep. This extra time might not be significant if the only other graphics operation being performed is rendering.[1] To evaluate a model and prepare it for rendering, the Thibault and Naylor algorithm would require only a post-processing step that converts the BSP tree for the model into one b-rep. In comparison, our algorithm must build many b-reps because it uses them to express intermediate results; it spends about 38.5 ± 9.9 percent of its time on the triangulation and the other activities needed to construct a b-rep. When, however, the leaves of a CSG tree are produced or modified by another graphics operation, the BSP-tree-based algorithm may require a preliminary representation conversion that our algorithm will not need. For example, the interactive modeling system we are implementing supports several kinds of deformation operations that are most easily performed on b-reps. Users need to be able to deform CSG models and then incorporate the results into other CSG models. To use the Thibault and Naylor algorithm in this context would require either a time-consuming representation conversion between each deformation and CSG operation, or a reformulation of our deformation algorithms to work on BSP trees (not a trivial task).

8 Conclusions and Extensions

We have shown that our b-rep evaluation algorithm has advantages over several other such algorithms. Our algorithm cannot generate b-reps from complex models at interactive rates, however, so we are looking for ways to improve its efficiency. Inefficiency has been identified in two aspects of the algorithm. As already mentioned, the TRIP polyhedral representation package adds unnecessary overhead to our algorithm, and the package needs to be improved. The other portion of our algorithm that could be made more efficient is the triangulation process. We suspect that a significant fraction of the total time spent performing triangulations produces faces that are subsequently removed by the classification phase. A remedy for this problem would be to determine the classification of a cluster before it is triangulated, so that areas that are excluded by the Boolean operation can be ignored.

Another issue related to triangulation is the size of the triangular faces produced. If a face is smaller than a certain size, the magnitude of its normal vector (obtained by taking the cross-product of its edges) will be too small to be represented accurately given the precision of the machine. A face this small cannot be allowed, because all faces must have reliable normal vectors. The Preparata and Shamos triangulation algorithm that we use makes no effort to maximize the size of the triangles it produces, and does occasionally

[1] At the 1990 SIGGRAPH Symposium on Interactive 3D Graphics in March, 1990, Naylor demonstrated an implementation of the algorithm that allowed sculpting of a model with a "tool" in real time. We have not yet had the opportunity to carefully compare the performance of our implementation to his, but will do so in the near future.

produce triangles that are too small to represent. Our current solution is to leave holes where those triangles should be, and make provisions in our algorithm to consider small holes to be covered with surfaces aligned with the adjacent faces. A better solution would be to adjust the adjacent faces so the hole is closed. Changing the geometry of the faces without affecting the visual appearance of the polyhedron is a tricky maneuver, but we hope to add this capability to our algorithm. The ability to make small, imperceptible changes to the structure of a polyhedron could also be beneficial in other ways. Specifically, it might be possible to identify facets of a polyhedron that are small enough to be removed without affecting the visual appearance of the polyhedron. The same algorithm used to close a hole could be used to remove the small facets. Eliminating these facets would reduce the number of faces involved in future processing of this polyhedron, a goal whose value has already been shown in this paper.

9 Acknowledgements

The author would like to thank John Hughes, Michael Natkin and Roberto Tamassia for their advice on the development of the new splitting-phase algorithm. Thanks also are due to Mark Nodine for designing and implementing TRIP, and to Brook Conner, Dan Robbins, Scott Snibbe and David Yang for creating the beautiful models on which the algorithm was tested. Finally, thanks go to Andries van Dam and the rest of the Brown Computer Graphics Group for creating the environment in which this research was necessary but feasible.

References

[1] Apgra, B., Bersack, B. and Mammen, A., "A Display System for the Stellar Graphics Supercomputer Model GS1000," Proceedings of SIGGRAPH '88, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 255-262.

[2] Deering, M. *et al.*, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics," Proceedings of SIGGRAPH '88, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp.229-238.

[3] Foley, J. D., van Dam, A., Feiner, S. and Hughes, J. F., *Computer Graphics: Principles and Practice*, Addison Wesley, Reading, Massachusetts, 1990, chapter 12.

[4] Fuchs, H. *et al.*, "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," Proceedings of SIGGRAPH '89, published as *Computer Graphics*, Vol. 23, No. 3, July 1989, pp. 79-88.

[5] Gharachorloo, N., Gupta, S., Sproul, R. F. and Sutherland, I. E., "A Characterization of Ten Rasterization Techniques," *ACM SIGGRAPH '89 Course Notes 16*, pp. 187-200.

[6] Goldfeather, J., Molnar, S., Turk, G. and Fuchs, H., "Near Real-Time CSG Rendering Using Tree Normalization and Geometric Pruning," *IEEE Computer Graphics and Applications*, May 1989, pp. 20-28.

[7] Goodrich, M. T., "Applying Parallel Processing Techniques to Classification Problems in Constructive Solid Geometry," Johns Hopkins University, Department of Computer Sciences, Technical Report JHU-89/6, June 1989.

[8] Jansen, F. W., "A Pixel-Parallel Hidden Surface Algorithm for Constructive Solid Geometry," *Proceedings of Eurographics '86*, Elsevier Science Publishers, New York, 1986, pp. 29-40.

[9] Laidlaw, D. H., Trumbore, W. B. and Hughes, J. F., "Constructive Solid Geometry for Polyhedral Objects," Proceedings of SIGGRAPH '86, published as *Computer Graphics*, Vol. 20, No. 4, August 1986, pp. 161-170.

[10] Moore, M. and Wilhelms, J., "Collision Detection and Response for Computer Animation," Proceedings of SIGGRAPH '88, published as *Computer Graphics*, Vol. 22, No. 4, August 1988, pp. 289-298.

[11] Preparata, F. P. and Shamos, M. I., *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985, pp. 237-241.

[12] Segal, M. and Sequin, C. H., "Partioning Polyhedral Objects into Nonintersecting Parts," *IEEE Computer Graphics and Applications*, January 1988, pp. 53-67.

[13] Thibault, W. C. and Naylor, B. F., "Set Operations on Polyhedra Using Binary Space Partitioning Trees," Proceedings of SIGGRAPH '87, published as *Computer Graphics*, Vol. 21, No. 4, July 1987, pp. 153-162.

[14] Yamaguchi, F. and Tokieda, T., "A Unified Algorithm for Boolean Shape Operations," *IEEE Computer Graphics and Applications*, Vol. 4, No. 6, June 1984, pp. 24-37.

[15] Yamaguchi, F., "A Unified Approach to Interference Problems Using a Triangle Processor," Proceedings of SIGGRAPH '85, published as *Computer Graphics*, Vol 19, No. 3, July 1985, pp. 141-149.

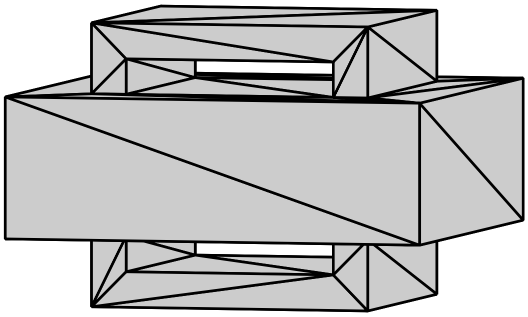

Figure 14: The "handle" model.

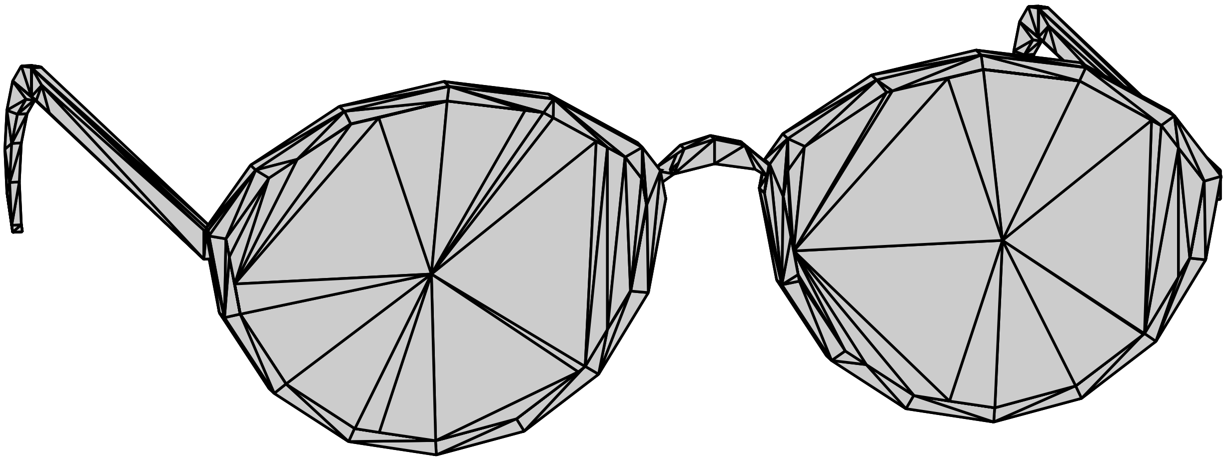

Figure 15: The "glasses" model. (The lenses were made opaque to emphasize the individual polygonal faces.)

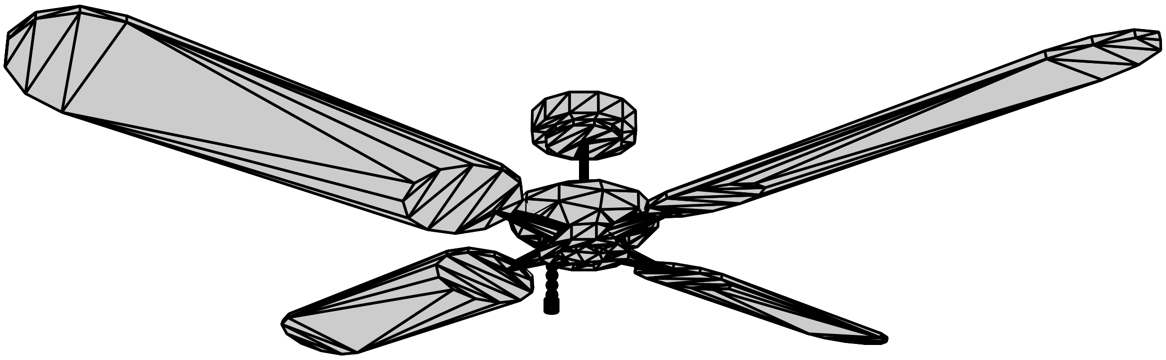

Figure 16: The "fan" model.

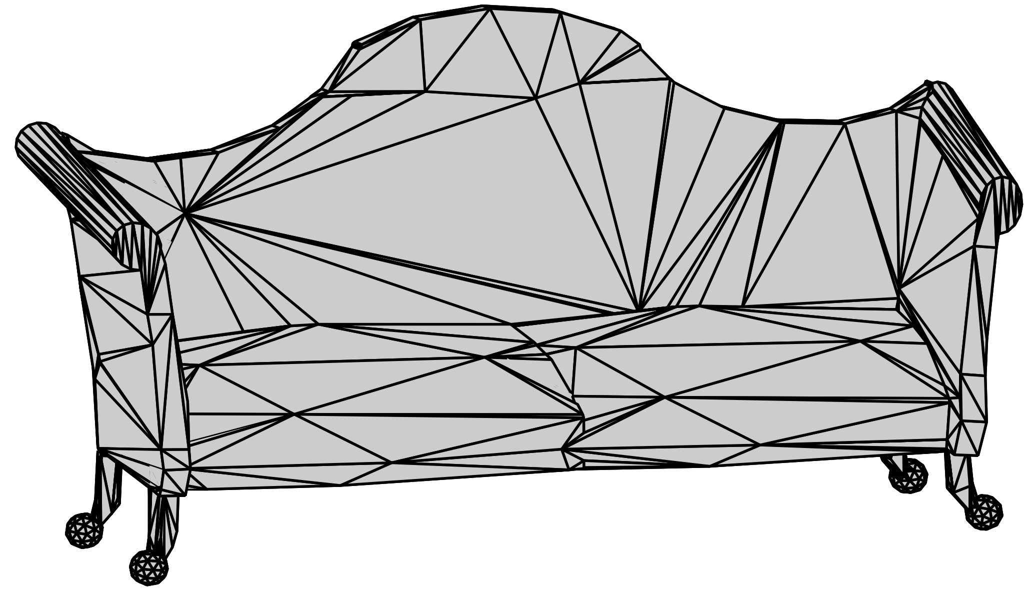

Figure 17: The “couch” model.

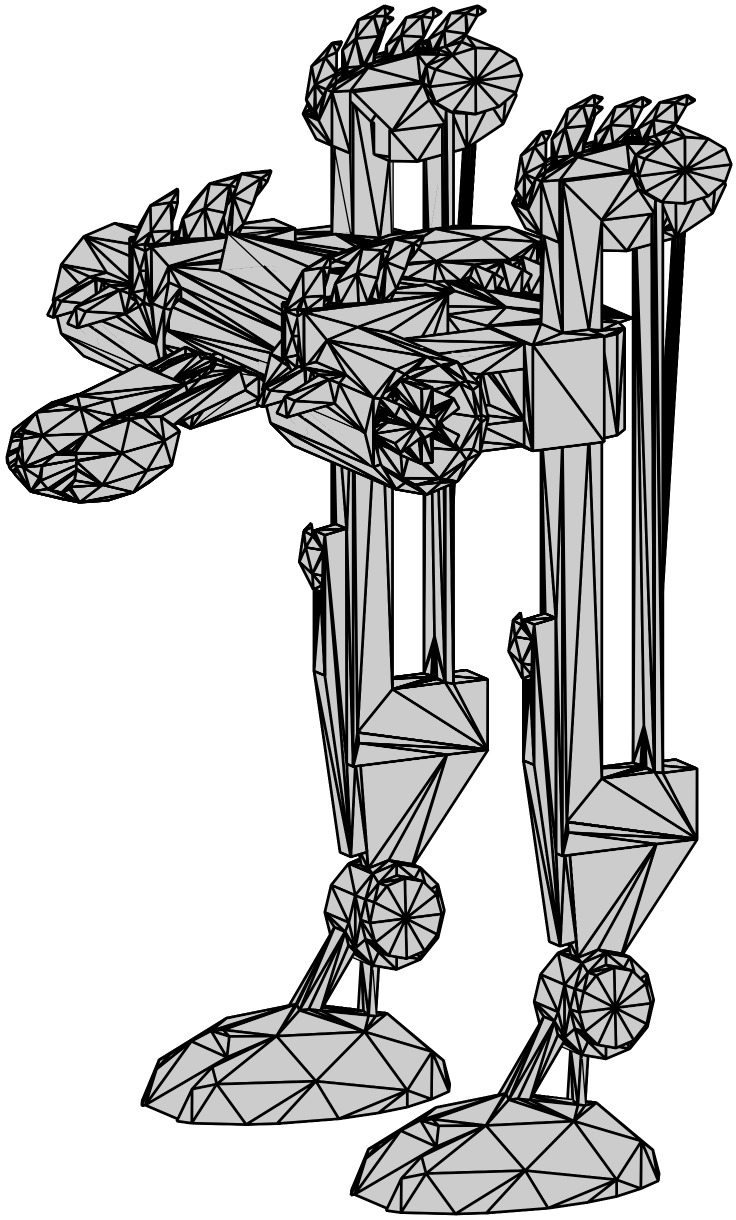

Figure 18: The “android” model.