50a

An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon

David Dobkin, Leonidas Guibas, John Hershberger,
Jack Snoeyink

September 10, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984—their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon*

David Dobkin[1], Leonidas Guibas[2,3], John Hershberger[3], and Jack Snoeyink[2]

[1]Princeton University, [2]Stanford University, [3]DEC Systems Research Center

September 10, 1989

*The first author would like to acknowledge the support of the National Science Foundation under Grant CCR87-00917. The fourth author was supported in part by a National Science Foundation Graduate Fellowship. This work was begun while the first author was visiting the DEC Systems Research Center.

Abstract

Modeling two-dimensional and three-dimensional objects is an important theme in computer graphics. Two main types of models are used in both cases: boundary representations, which represent the surface of an object explicitly but represent its interior only implicitly, and constructive solid geometry representations, which model a complex object, surface and interior together, as a boolean combination of simpler objects. Because neither representation is good for all applications, conversion between the two is often necessary.

We consider the problem of converting boundary representations of polyhedral objects into constructive solid geometry (CSG) representations. The CSG representations for a polyhedron P are based on the half-spaces supporting the faces of P. For certain kinds of polyhedra this problem is equivalent to the corresponding problem for simple polygons in the plane. We give a new proof that the interior of each simple polygon can be represented by a monotone boolean formula based on the half-planes supporting the sides of the polygon and using each such half-plane only once. Our main contribution is an efficient and practical $O(n \log n)$ algorithm for doing this boundary-to-CSG conversion for a simple polygon of n sides. We also prove that such nice formulæ do not always exist for general polyhedra in three dimensions.

1 Preliminaries

A key problem in solid modeling is the computer representation of solid objects. It is desirable that such representations be compact and support efficient simulation of real-world operations on the objects. Over the years two different styles of representation have emerged; these are used by nearly all geometric modeling systems currently in existence. The first style of representation describes an object by the collection of surface elements forming its boundary: this is a *boundary representation*. In effect, boundary representations reduce the problem of representing a solid object to one of representing surface elements. This is a somewhat simpler problem, since it is set in one dimension less. The second style of representation describes a solid object as being constructed by regularized boolean operations on some simple primitive solids, such as boxes, spheres, cylinders, etc. Such a description is referred to as a constructive solid geometry representation, or *CSG representation*, for short. Each style of representation has its advantages and disadvantages, depending on the operations we wish to perform on the objects. The reader is referred to one of the standard texts in solid modeling [13, 16], or the review article [23] for further details on these representations and their relative merits.

If one looks at modelers in either camp, for example the ROMULUS [16], GEOMOD [25], and MEDUSA [17] modelers of the boundary persuasion, or the PADL-1 [27], PADL-2 [2], and GMSOLID [1] modelers of the CSG persuasion, one nearly always finds provisions for converting to the other representation. This is an important and indispensable step that poses some challenging computational problems[1]. In this paper we will deal with certain cases of the boundary-to-CSG conversion problem and present some efficient computational techniques for doing the conversion. Our approach is based on that of Peterson [21].

Peterson considered the problem of obtaining a CSG representation for simple polyhedral solids, such as prisms or pyramids (not necessarily convex), based on the half-spaces supporting the faces of the solid. Such solids are, in effect, two-dimensional objects (think of the base of the prism or pyramid) in which the third dimension has been added in a very simple manner. Thus Peterson considered the problem of finding CSG representations for simple polygons in the plane. (A related problem is that of finding convex decompositions of simple polygons [4, 18, 20, 26, 28].) By a complicated argument, Peterson proved that every simple polygon in the plane admits of a representation by a boolean formula based on the half-planes supporting its sides. This formula is especially nice in that each of the supporting half-planes appears in the formula exactly once, and hence the formula is monotone (no complementation is needed). For this reason we focus our attention on such formulæ, which we call *Peterson-style formulæ*.

A monotone formula for a polygon or polyhedron is desirable because it makes it possible to predict how small perturbations of the defining half-spaces affect the overall volume. This is an important property in various applications of solid modeling, such as machine tooling. When machining a peg that has to fit in a hole, for example, it is important to know that any errors in the position of the defining half-spaces will result in making the peg only smaller. If a half-space and its negation both enter a formula, then a perturbation of the defining plane can make the volume smaller in one part of the object and simultaneously

[1] To quote from Requicha [23]: "..the relative paucity of known conversion algorithms poses significant constraints on the geometric modeling systems that we can build today."

larger in another part.

In this paper we first give a short new proof that every polygon has a Peterson-style formula (Section 3). Peterson did not explicitly consider algorithms for deriving this CSG representation from the polygon. A naïve implementation based on his proof would require $\Theta(n^2)$ time for the conversion, where n is the size (number of sides or vertices) of the polygon. We provide in this paper an efficient $\Theta(n \log n)$ algorithm for doing this boundary-to-CSG conversion (Section 4). We regard this algorithm as the major contribution of our paper; the algorithm uses many interesting techniques from the growing field of computational geometry [5, 22]. Nevertheless, it is very simple to code—its subtlety lies in the analysis of the performance and not in the implementation. Finally (Section 5), we show that Peterson-style formulæ are not always possible for general polyhedra in three dimensions and discuss a number of related issues.

We believe that the work presented in this paper illustrates how several of the concepts and techniques of computational geometry can be used to solve problems that are of clear importance in solid modeling and computer graphics. The solution that we obtain is both mathematically interesting and practical to implement. We expect to see more such applications of computational geometry to other areas in the future and hope that this paper will motivate some researchers in the graphics area to study computational geometry techniques more closely.

2 Formulation and history of the problem

Let P be a simple polygon in the plane; in this context, simple means non-self-intersecting. By the Jordan curve theorem, such a polygon subdivides the plane into two regions, its interior and its exterior. In general, we identify the polygon with its interior. Let us orient all the edges of P so that the interior of P lies locally to the right of each edge, and give each such oriented edge a name. We will call these names *literals*. To each literal we also give a second meaning. A literal m also represents the half-plane bounded by the infinite line supporting the edge m and extending to the right of that line. We will speak of such a half-plane as *supporting* the polygon (even though P might not all lie in the half-plane). See Figure 1 for an illustration of these concepts.

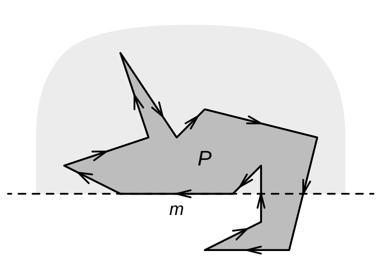

Figure 1: A simple polygon P and the half-plane supporting side m

Notice that, for each point x of the plane, if we know whether x lies inside or outside each of the half-planes supporting P, then we know in fact if x is inside P. This follows, because each of the regions into which the plane is subdivided by the infinite extensions of the sides of P lies either wholly inside P, or wholly outside it. As a result, there must exist a boolean formula whose atoms are the literals of P and which expresses the interior of P. For example, if P is convex, then this formula is simply the "and" of all the literals.

Since "and"s and "or"s are somewhat cumbersome to write, we will switch at this point to algebraic notation and use multiplication conventions for "and" and addition conventions for "or". Consider the two simple polygons shown in Figure 2. Formulæ for the two polygons are $uv(w(x+y)+z)$ for polygon (a) and $uvw(x+y+z)$ for polygon (b). The associated boolean expression trees are also shown in Figure 2. Notice that these are Peterson-style formulæ: they are monotone and use each literal exactly once. The reader is invited at this point to make sure that these formulæ are indeed correct.

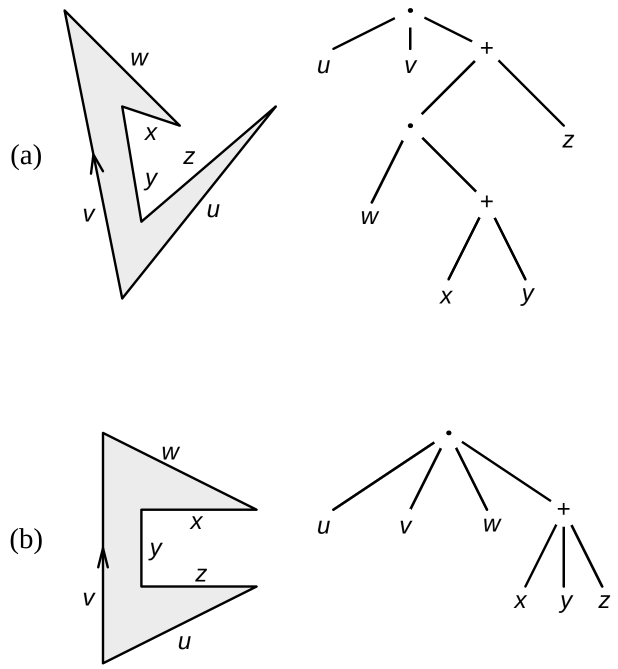

Figure 2: Formulæ for two polygons

A more complex formula for a simple polygon was given by Guibas, Ramshaw, and Stolfi [9] in their kinetic-framework paper. That style of formula for the two polygons of Figure 2 is $\overline{u\overline{v} \oplus v\overline{w} \oplus w\overline{x} \oplus \overline{x}y \oplus \overline{y}z \oplus z\overline{u}}$. Here $\oplus$ denotes logical "xor" and the overbar denotes complementation. As explained in [9], that type of formula is purely local, in that it depends only on the convex/concave property of successive angles of the polygon. The rule should be obvious from the example: as we go around, we complement the second literal corresponding to a vertex if we are at a convex angle, and the first literal if we are at a concave angle. Thus the formula is the same for both of the example polygons.

Although a formula of this style is trivial to write down, it is not as desirable in solid modeling as a Peterson-style formula, because of the use of complementation and the "xor" operator. The Peterson formula is more difficult to derive, because it captures in a sense how the polygon nests within itself and thus is more global in character. It can be viewed naïvely as an inclusion-exclusion style formula that reflects this global structure of the polygon. We caution the reader, however, that this view of the Peterson formula is too naïve and led us to a couple of flawed approaches to this problem.

In general there are many boolean formulæ that express a simple polygon in terms of its literals. Proving the equivalence of two boolean formulæ for the same polygon is a non-trivial exercise. The reason is that of the 2^n primitive "and" terms one can form on n literals (with complementation allowed), only $\Theta(n^2)$ are non-zero, in the sense that they denote non-empty regions of the plane. Thus numerous identities hold and must be used in proving formula equivalence.

The decomposition of a simple polygon into convex pieces [4, 18, 20, 26, 28] gives another kind of boolean formula for the polygon, one in which the literals are not half-planes, but convex polygons. Depending on the type of decomposition desired, the convex polygons may or may not overlap; in the overlapping case, the formula may or may not contain negations. If we expand the literals in a convex decomposition into "and"s of half-planes, the result need not be a Peterson-style formula: negations, repeated literals, and half-planes that do not support the polygon are all possible.

If we leave the boolean domain and allow algebraic formulæ for describing the characteristic function of a simple polygon, then such formulæ that are purely local (in the same sense as the "xor" formula above) are given in a paper of Franklin [6]. Franklin describes algebraic local formulæ for polyhedra as well. We do not discuss this further here as it goes beyond the CSG representations we are concerned with.

3 The existence of monotone formulæ

In this section we will prove that the interior of every simple polygon P in the plane can be expressed by a Peterson-style formula, that is, a monotone boolean formula in which each literal corresponding to a side of P appears exactly once.

As it turns out, it is more natural to work with simple bi-infinite polygonal chains (or *chains*, for short) than with simple polygons. An example of a simple bi-infinite chain c is shown in Figure 3. Such a chain c is terminated by two semi-infinite rays and in between contains an arbitrary number of finite sides. Because it is simple and bi-infinite, it subdivides the plane into two regions. We will in general orient c in a consistent manner, so we can speak of the region of the plane lying to the left of c, or to the right of c, respectively. By abuse of language, we will refer to these regions as *half-spaces*.

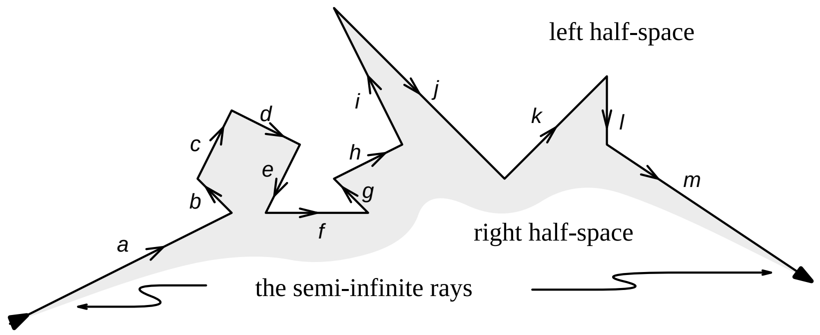

Figure 3: A simple bi-infinite chain

The interior of a simple polygon P can always be viewed as the intersection of two such chain half-spaces. Let ℓ and r denote respectively the leftmost and rightmost vertex of P. As in Figure 4, extend the sides of P incident to ℓ infinitely far to the left, and the sides incident to r infinitely far to the right. It is clear that we thus obtain two simple bi-infinite chains and that the interior of P is the intersection of the half-space below the upper chain with the half-space above the lower chain. Notice also that the literals used by the upper and lower chains for these two half-spaces form a partition of the literals of P. Thus it suffices to prove that a chain half-space admits of a monotone formula using each of its literals exactly once.

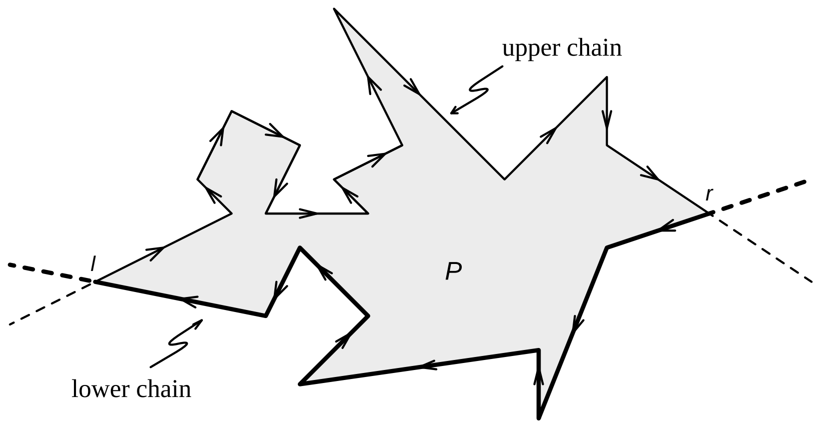

Figure 4: The interior of a simple polygon P

If a chain consists of a single line, then the claim is trivial: the formula is the single literal that defines the line.

If a chain c has more than one edge, we prove the claim by showing that there always exists a vertex v of c such that if we extend the edges incident to v infinitely far to the other side of v, these extensions do not cross c anywhere. In particular, the extensions create two new simple bi-infinite chains c_1 and c_2 that, as before, partition the literals used by c. See Figure 5 for an example. It is easy to see that the half-space to the right (say) of c is then either the intersection or the union of the half-spaces to the right of c_1 and c_2. It will be the intersection if the angle of c at v in the selected half-space is convex (as is the situation in Figure 5), and the union if this angle is concave.

The existence of the desired vertex v is relatively easy to establish. Of the two half-spaces defined by c there is one that is bounded by the two semi-infinite rays in a "convex" fashion. What we mean by this is that when we look at this half-space from a great distance above the xy-plane (so we can discern only the semi-infinite rays bounding it) it appears as a convex angle ($\leq \pi$). For example, in Figure 5, the right half-space R of c is the convex one. Now consider the convex hull $h(R)$, the intersection of all half-planes containing R. This hull is an unbounded polygonal region whose vertices are vertices of c. Clearly at least one vertex of c lies on $h(R)$, and any such vertex is a good vertex at which to break c; that is, it can serve as the vertex v of the previous argument. The reason is clear from Figure 6: at any such vertex the extensions of the sides incident upon it cannot cross c again.

It is worth remarking here that the determination of the splitting vertex v in the manner above is not at all influenced by whether we are trying to obtain a boolean formula for the right half-space of c or the left half-space of c. The choice of which half-space to take the convex hull of is determined solely by the behavior of the semi-infinite rays of c. Indeed, if we were to choose the wrong ("concave") half-space, its convex hull would be the whole plane and would contain no vertices. We can summarize the situation by saying that we always

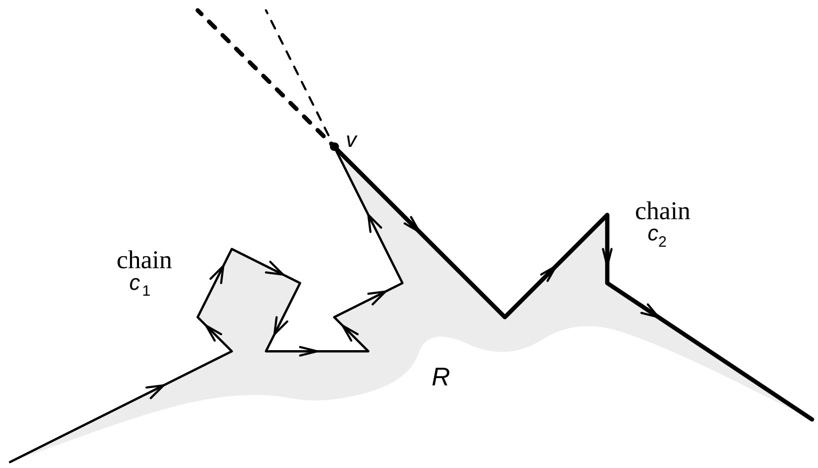

Figure 5: The splitting vertex v for a chain c

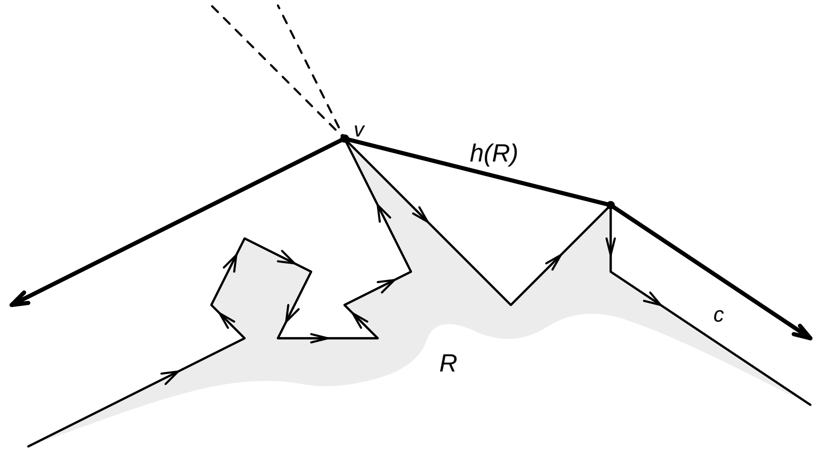

Figure 6: The convex hull $h(R)$

split at a vertex of the convex hull of the polygonal chain c; this definition automatically selects the correct half-space.

By recursively applying this decomposition procedure until each subchain becomes a single bi-infinite straight line we can derive the following theorem.

Theorem 3.1 *Every half-space bounded by a simple bi-infinite polygonal chain has a monotone boolean formula using each of the literals of the chain exactly once. The same holds for the interior of any finite simple polygon.*

If we are given a polygonal chain c, such as the one in Figure 3, then certain aspects of the boolean formula of (say) the right half-space R of c can be immediately deduced by inspection. For example, it follows from the arguments above that there exists a boolean formula for R that not only uses each literal exactly once, but in fact contains these literals in the order in which they appear along c; if we were to omit the boolean operators and parentheses in the formula, we would just get a string of all the literals in c in order. Furthermore, the boolean operators between these literals are easy to deduce. As the previous discussion makes clear, between two literals that define a convex angle in R the corresponding operator has to be an "and", and between two literals that define a concave angle the corresponding operator has to be an "or". Thus, with parentheses omitted, the boolean formula for the chain c in Figure 3 has to look like $a + bcde + f + gh + ij + kl + m$.

The preceding paragraph shows that the crux of the difficulty in computing a boolean formula is to obtain the parenthesization, or equivalently, the sequence of the appropriate splitting vertices. We call this the *recursive chain-splitting problem* for a simple bi-infinite chain. The solution of this problem is the topic of the next section. For the chain of Figure 3 a valid solution is $((a + bc)(de + f) + g(h + i))(j + k(l + m))$.

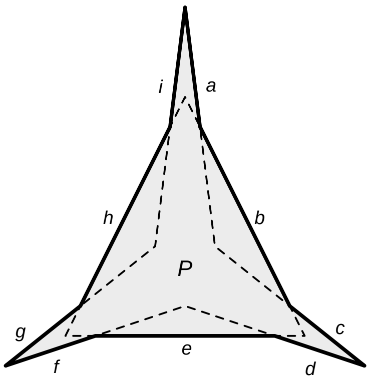

Figure 7: Our methods cannot obtain all valid formulæ for this polygon

We conclude by noticing that a procedure for solving this problem may be non-deterministic, since in general we will have a choice of several splitting vertices. We can in fact simultaneously split at any subset of them. Still, not all valid Peterson-style formulæ for a simple polygon are obtained in this fashion. Our formulæ all have the property that the literals appear in the formula in the same order as in the polygon. Figure 7 shows an example of a Peterson-style formula where that is not true: a valid formula for the polygon shown is $(a + c)(d + f)(g + i) + beh$.

4 The conversion algorithm

We have seen in Section 3 that we can find a monotone boolean formula for a simple polygon if we can solve the following *recursive chain-splitting problem*:

> Given a simple bi-infinite polygonal chain with at least two edges, find a vertex z of its convex hull. Split the chain in two at z and extend to infinity the two edges incident to z, forming two new chains. Because z is on the convex hull, both chains are simple. Recursively solve the same problem for each chain that has at least two edges.

We present an $O(n \log n)$ algorithm to solve this chain-splitting problem, where n is the number of vertices of the polygon P. The algorithm uses only simple data structures and is straightforward to implement.

Before we describe our algorithm, let us consider a naïve alternative to it. Many algorithms have been published that find the convex hull of a simple polygon in linear time [7, 15, 11, 14, 24]. With slight modifications, any of these algorithms can be used to find a vertex on the hull of a simple bi-infinite polygonal chain. If we use such an algorithm to solve the chain-splitting problem, the running time is $O(n)$ plus the time needed to solve the two subproblems recursively. The worst-case running time $t(n)$ is given by the recurrence

$$t(n) = \max_{0<k<n} (t(k) + t(n-k)) + O(n),$$

which has solution $t(n) = O(n^2)$.

This quadratic behavior occurs in the worst case, shown in Figure 8a, because each recursive step spends linear time splitting a single edge off the end of the path. In the best case, on the other hand, each split divides the current path roughly in half, and the algorithm runs in $O(n \log n)$ time. This asymptotic behavior can be obtained for the path shown in Figure 8b, if the splitting vertices are chosen wisely.

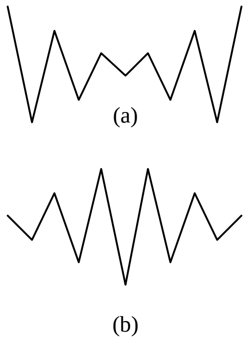

Figure 8: Two paths with worst-case and best-case splitting behavior

The best case of this naïve algorithm is like a standard divide-and-conquer approach: at each step the algorithm splits the current path roughly in half. In general, however, it is difficult to guarantee an even division, since all vertices on the convex hull might be extremely close to the two ends of the path. Thus, to avoid quadratic behavior, we must instead split each path using less than linear time. Other researchers have solved similar problems by making the splitting cost depend only on the size of the smaller fragment [8, 10]. If the running time $t(n)$ obeys the recurrence

$$t(n) = \max_{0<k<n} (t(k) + t(n-k)) + O(\min(k, n-k)),$$

then $t(n) = O(n \log n)$. Our method uses a similar idea: the splitting cost is $O(\log n)$ plus a term that is linear in the size of one of the two fragments. The fragment is not necessarily the smaller of the two, but we can bound its size to ensure an $O(n \log n)$ running time overall. The details of this argument appear in Section 4.5.

We present our algorithm in several steps. We first give a few definitions, then give an overview of our approach. We follow the informal overview with a pseudo-code description of the algorithm. Section 4.3 gives more detail about one of the pseudo-code operations, and Section 4.4 describes the data structure used by the algorithm. Section 4.5 concludes the presentation of the algorithm by analyzing its running time.

4.1 Definitions

As shown in Section 3, we can find a boolean formula for P by splitting the polygon at its leftmost and rightmost vertices to get two paths, then working on the two paths separately. We denote by π the current path, either upper or lower. If u and v are vertices of π, we use the notation $\pi(u,v)$ to refer to the subpath of π between u and v, inclusive. The convex hull of a set of points A is denoted by $h(A)$; we use $h(u,v)$ as shorthand for $h(\pi(u,v))$. A path $\pi(u,v)$ has $|\pi(u,v)|$ edges; similarly, $|h(u,v)|$ is the number of edges on $h(u,v)$.

We can use the path $\pi(u,v)$ to specify a bi-infinite chain by extending its first and last edges. Let e_u be the edge of $\pi(u,v)$ incident to u, and let $\overrightarrow{e_u}$ be the ray obtained by extending e_u beyond u. Let e_v and $\overrightarrow{e_v}$ be defined similarly. Then $\pi(u,v)$ specifies the bi-infinite polygonal chain obtained by replacing e_u by $\overrightarrow{e_u}$ and e_v by $\overrightarrow{e_v}$. In general, for arbitrary u and v, this bi-infinite chain need not be simple. Our algorithm, however, will guarantee the simplicity of each bi-infinite chain it considers. We assume in what follows that $\overrightarrow{e_u}$ and $\overrightarrow{e_v}$ are not parallel, which guarantees that the convex hull of the bi-infinite chain has at least one vertex. A slight modification to the algorithm is needed if the rays are parallel.

4.2 The algorithm

This section presents the algorithm that recursively splits a polygonal chain. We first outline the algorithm and then present it in a pseudo-code format. Subsequent sections give the details of the operations sketched in this section.

We now outline the algorithm. Given a polygonal path $\pi(u, v)$ with at least two edges, we partition it at a vertex x to get two pieces $\pi(u, x)$ and $\pi(x, v)$ with roughly the same number of edges. The vertex x is not necessarily a vertex of $h(u, v)$; this partitioning is merely preparatory to splitting $\pi(u, v)$ at a hull vertex. In $O(|\pi(u, v)|)$ time we compute the convex hulls of $\pi(u, x)$ and $\pi(x, v)$ in such a way that for any vertex z of $\pi(u, v)$, we can easily find $h(x, z)$. Our data structure lets us account for the cost of finding $h(x, z)$ as part of the cost of building $h(u, x)$ and $h(x, v)$. The details of this accounting appear in Section 4.5.

The next step of the algorithm locates a vertex z of the convex hull of the bi-infinite chain $\pi(u, v) \cup \overrightarrow{e_u} \cup \overrightarrow{e_v}$. We will split $\pi(u, v)$ at z. The vertex z can be on the path $\pi(u, x)$ or on the path $\pi(x, v)$. Without loss of generality let us assume that z is a vertex of $\pi(u, x)$; because u is not a vertex of the convex hull, $z \neq u$. We recursively split $\pi(u, z)$, partitioning it at its midpoint, building convex hulls, and so on. However, and this is the key observation, we do not have to do as much work for $\pi(z, v)$ if $z \neq x$. We already have the hull $h(x, v)$, and we can easily find $h(z, x)$ from our data structure for $h(u, x)$. Thus we can recursively split $\pi(z, v)$ without recomputing convex hulls. Intuitively speaking, we do a full recursion (including convex hull computation) only on pieces whose length is less than half the length of the piece for which we last computed convex hulls.

The key to our algorithm's efficiency is avoiding the recomputation of convex hulls. The naïve algorithm builds $O(n)$ hulls whose average size can be as much as $n/2$; our algorithm also builds $O(n)$ hulls, but their average size is only $O(\log n)$. Our algorithm locates n splitting vertices in $O(\log n)$ time apiece, which contributes another $O(n \log n)$ term to the running time. These two terms dominate the time cost of the algorithm, as Section 4.5 shows.

We present the algorithm more formally in the pseudo-code below. The pseudo-code represents a convex hull $h(x, v)$ using a data structure called a *path hull*, denoted by $PH(x, v)$; the path hull stores the vertices of $h(x, v)$ in a linear array. The algorithm uses the path hull $PH(x, v)$ to produce $PH(x, z)$ efficiently, for any splitting vertex z in $\pi(x, v)$. The algorithm consists of two mutually recursive subroutines, $f()$ and $p()$, whose names stand for *full* and *partial*. The routine $f(u, v)$ partitions $\pi(u, v)$ at x to get two equal parts, builds a path hull structure for each, and calls $p(u, x, v)$. The subroutine $p(u, x, v)$ uses $PH(x, u)$ and $PH(x, v)$ to find the splitting vertex z; Section 4.3 gives the details of this operation. The routine then splits $\pi(u, v)$ at z and recurses on each fragment; it ensures that the required path hulls have been built whenever $p()$ is called. We start the algorithm by invoking $f()$ on the entire path π.

$f(u,v)$ /* Precondition: $u \neq v$ */
begin
1. **if** $\pi(u,v)$ is a single edge **then return**;
else
begin
2. Let x be the middle vertex of $\pi(u,v)$;
3. Build $PH(x,u)$ and $PH(x,v)$;
4. $p(u,x,v)$;
end
end

$p(u,x,v)$ /* x is a vertex of $\pi(u,v)$, not equal to u or v. Path hulls $PH(x,v)$ and $PH(x,u)$ have been computed. */
begin
5. Find a vertex z of $h(\pi(u,v) \cup \overrightarrow{e_u} \cup \overrightarrow{e_v})$. Output z as part of the sequence of splitting vertices.
6. **if** $x = z$ **then begin** $f(u,x)$; $f(x,v)$; **end**
else
begin
7. **if** $z \in \pi(u,x)$ **then** build $PH(x,z)$ from $PH(x,u)$;
else build $PH(x,z)$ from $PH(x,v)$;
if z is a vertex of $\pi(u,x)$ **then**
8. **begin** $f(u,z)$; $p(z,x,v)$; **end**
else
9. **begin** $p(u,x,z)$; $f(z,v)$; **end**
end
end

The chain-splitting algorithm

4.3 Finding a splitting vertex

This section shows how to use the path hull data structure to find the splitting vertex z. Our method exploits the fact that $PH(x,v)$ represents $h(x,v)$ as a linear array of convex hull vertices: we perform binary search on the array to find the splitting vertex.

Given a path $\pi(u,v)$, we want to find a vertex of the convex hull of the bi-infinite chain that $\pi(u,v)$ specifies. Each such vertex belongs to the finite convex hull $h(u,v)$; we solve our problem by finding a vertex of $h(u,v)$ that is guaranteed to belong to the infinite hull.

The edges of the infinite hull $h(\pi(u,v) \cup \overrightarrow{e_u} \cup \overrightarrow{e_v})$ have slopes in a range bounded by the slopes of $\overrightarrow{e_u}$ and $\overrightarrow{e_v}$. Vertices of the hull have tangent slopes in the same range. We simply find a vertex of $h(u,v)$ with a tangent slope in the range. Let d_u and d_v be the direction vectors of the rays $\overrightarrow{e_u}$ and $\overrightarrow{e_v}$. Because $\overrightarrow{e_u}$ and $\overrightarrow{e_v}$ are not parallel, d_u and d_v define an angular range of less than 180 degrees; define d to be the negative of the bisector of this angular range. An extreme vertex of $h(u,v)$ in direction d is guaranteed to be a vertex of the infinite hull.[2] See Figure 9 for an example.

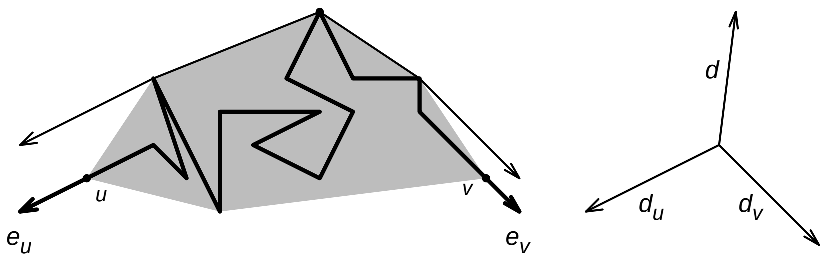

Figure 9: We find an extremal vertex in the direction d

We use binary search on each of the two path hulls $PH(u,x)$ and $PH(x,v)$ to find an extreme vertex in direction d. We compare the two vertices and pick the more extreme of the two. If we break ties consistently in the binary searches and in the comparison of the two extreme vertices (say, by preferring the left vertex of tied pairs), the vertex we find is guaranteed to be a vertex of the infinite hull.

4.4 Implementing path hulls

In this section we describe the path hull data structure used in the previous two sections. The path hull $PH(x,v)$ represents the convex hull of $\pi(x,v)$. It is not symmetric in its arguments: it implicitly represents $h(x,v')$ for all vertices v' in $\pi(x,v)$, but does not represent $h(v',v)$ for any v' not equal to x. The structure $PH(x,v)$ has three essential properties:

1. $PH(x,v)$ represents $h(x,v)$ by a linear array of vertices. Let $\hat{v}$ be the vertex of $h(x,v)$ closest to v on $\pi(x,v)$. Then the array lists the vertices of $h(x,v)$ in clockwise order, starting and ending with $\hat{v}$.

2. Given $PH(x,v)$, we can transform it into $PH(x,v')$ for any vertex v' in $\pi(x,v)$, destroying $PH(x,v)$ in the process. Let the vertices of $\pi(v,x)$ be numbered $v = v_1, v_2, \ldots, v_k = x$; we can successively transform $PH(x,v)$ into $PH(x,v_i)$ for each v_i in sequence from $v_1 = v$ to $v_k = x$ in total time proportional to $|\pi(x,v)|$.

3. $PH(x,v)$ can be built from $\pi(x,v)$ in $O(|\pi(x,v)|)$ time.

We get these properties by adapting Melkman's algorithm for finding the convex hull of a polygonal path [15]. We satisfy requirement 2 by "recording" the actions of Melkman's algorithm as it constructs $h(x,v)$, then "playing the tape backwards."

[2] To avoid computing square roots, in practice we do not compute the bisector of the angle defined by d_u and d_v. Instead, we find the normals to d_u and d_v that point away from the infinite hull, then add the two to get a direction d strictly between these normals.

Many linear-time algorithms have been proposed to find the convex hull of a simple polygon [7, 15, 11, 14, 24]. Some of these algorithms need to find a vertex on the hull to get started; we use Melkman's algorithm because it does not have this requirement. It constructs the hull of a polygonal path incrementally: it processes path vertices in order, and at each step it builds the hull of the vertices seen so far.

The algorithm keeps the vertices of the current convex hull in a double-ended queue, or *deque*. The deque lists the hull vertices in clockwise order, with the most recently added hull vertex at both ends of the deque. Let the vertices in the deque be $v_b, v_{b+1}, \ldots, v_{t-1}, v_t$, where $v_b = v_t$. The algorithm operates on the deque with *push* and *pop* operations that specify the end of the queue, bottom or top, on which they operate. The algorithm appears below; it assumes that no three of the points it tests are collinear, though this restriction is easy to lift.

Get the first three vertices of the path with the function *NextVertex*() and put them into the deque in the correct order.
while $v \leftarrow$ *NextVertex*() returns a new vertex **do**
  if v is outside the angle $\angle v_{t-1} v_t v_{b+1}$ **then**
    begin
      while v is left of $\overrightarrow{v_b v_{b+1}}$ **do** $pop(v_b, \mathrm{bottom})$;
      while v is left of $\overrightarrow{v_{t-1} v_t}$ **do** $pop(v_t, \mathrm{top})$;
      $push(v, \mathrm{bottom})$; $push(v, \mathrm{top})$;
    end

Melkman's convex hull algorithm

We now sketch a proof of correctness; Melkman gives a full proof [15]. We first consider the case in which the vertex v is discarded. This happens when v is inside the angle $\angle v_{t-1} v_t v_{b+1}$. (See Figure 10.) We know that v_{b+1} is connected to v_{t-1} by a polygonal path, and that v is connected to v_b by a polygonal path. The two paths do not intersect, so v must lie inside the current hull. When v is not discarded, it lies outside the current hull, and the algorithm pops hull vertices until it gets to the endpoints of the tangents from v to the current hull. The algorithm is linear: if it operates on a path with n vertices, it does at most $2n$ pushes and $2n - 3$ pops.

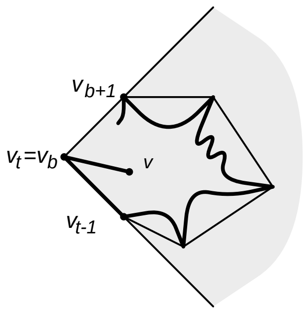

Figure 10: Discard v if it lies in the shaded sector

We can use the algorithm to build an array representation of the hull. The algorithm does at most n pushes at either end of the deque, so we can implement the deque as the middle part of an array of size $2n$. Pushes and pops increment and decrement the array indices of the ends of the queue; pushes write in a new element, pops read one out. The resulting deque contains the vertices of the convex hull in a contiguous chunk of an array.

As we have described it so far, the array data structure has path hull properties 1 and 3; how can we obtain property 2? When the algorithm builds $h(x, v)$ starting from x and working toward v, at intermediate steps it produces $h(x, v')$ for every vertex v' in $\pi(x, v)$. We need to be able to reconstruct these intermediate results. To do this, we add code to the algorithm to create a transcript of all the operations performed, recording what vertices are pushed and popped at each step. The structure $PH(x, v)$ stores not only the deque that represents $h(x, v)$, but also the transcript of the operations needed to create the deque from scratch. To reconstruct $PH(x, v')$ from $PH(x, v)$, we read the transcript in reverse order, performing the inverse of each recorded operation (pushing what was popped, and vice versa), until the deque represents $h(x, v')$. We throw away the part of the transcript we have just read, so that $PH(x, v')$ stores only the transcript of the operations needed to create $h(x, v')$. Because we discard every step we have read over, we look at each step of the transcript at most once during the playback. Therefore, reconstructing the intermediate results takes time proportional to the original cost of finding $PH(x, v)$. This completes the proof that the path hull data structure has all three of its required properties.

4.5 Analyzing the running time

In this section we analyze the running time of the chain-splitting algorithm given on page 10. The analysis uses a "credit" scheme, in which each call to $f()$ or $p()$ is given some number of credits to pay for the time used in its body and its recursive calls. We give $O(n \log n)$ credits to the first call to $f()$, then show that all calls have enough credits to pay for their own work and that of their recursive calls.

We begin the analysis by proving that $f()$ and $p()$ are called $O(n)$ times: Every call to $p(u, x, v)$ splits $\pi(u, v)$ into two non-trivial subpaths, and every call to $f(u, v)$ for which $\pi(u, v)$ has more than one edge passes $\pi(u, v)$ on to $p()$. The initial path π can be split only $O(n)$ times, so the recursion must have $O(n)$ calls altogether.

How much work is done by a call to $f(u, v)$, exclusive of recursive calls? We assume that the vertices of π are stored in an array. Therefore line 2 of $f()$ takes only constant time. Line 3 is the only step of $f()$ that takes non-constant time; as shown in Section 4.4, line 3 takes $O(|\pi(u, v)|)$ time. We define the value of a credit by saying that a call $f(u, v)$ needs $|\pi(u, v)|$ credits—one credit per edge of $\pi(u, v)$—to pay for the work it does, exclusive of its call to $p()$. The constant-time steps in $f()$ take $O(n)$ time altogether and hence are dominated by the rest of the running time.

A call to $p(u, x, v)$ does accountable work in lines 5 and 7. The cost of line 5 is dominated by two binary searches, which take $O(\log n)$ time. Line 5 therefore takes $O(n \log n)$ time over the whole course of the algorithm. Section 4.4 shows that the cost of building $PH(x, z)$ at line 7 can be accounted as part of the construction cost of the path hull from which $PH(x, z)$ is derived. Thus we can ignore the work done at line 7 of $p()$; its cost is dominated by that of line 3 of $f()$.

To complete our analysis of the running time, we must bound the cost of all executions of line 3 of $f()$. In a single call to $f(u, v)$, line 3 uses $|\pi(u, v)|$ credits. The sum of all credits used by line 3 is proportional to the time spent executing that line. We give $n\lceil \log_2 n \rceil$ credits to the first call to $f()$, then show that this is enough to pay for all executions of line 3. We use the following two invariants in the proof:

1. A call to $f(u,v)$ is given at least $m\lceil\log_2 m\rceil$ credits, where $m = |\pi(u,v)|$, to pay for itself and its recursive calls.

2. A call to $p(u,x,v)$ is given at least $(l+r)\lceil\log_2 \max(l,r)\rceil$ credits, where $l = |\pi(u,x)|$ and $r = |\pi(x,v)|$, to pay for its recursive calls.

Lemma 4.1 *If a call to $f()$ or $p()$ is given credits in accordance with invariants 1 and 2, it can pay for all executions of line 3 it does explicitly or in its recursive calls.*

Proof: Let m, l, and r be as defined above. The proof is by induction on m. A call to $f(u,v)$ with $m = 1$ gets no credits and needs none, since it does not reach line 3. There are no calls to $p()$ with $m = 1$.

A call to $f(u,v)$ with $m > 1$ gets at least $m\lceil\log_2 m\rceil$ credits and spends m of them executing line 3. It has $m\lceil\log_2(m/2)\rceil$ to pass on to its call to $p(u,x,v)$. The larger of l and r is $\lceil m/2\rceil$, and $\lceil\log_2(m/2)\rceil = \lceil\log_2\lceil m/2\rceil\rceil$, so the call to $p(u,x,v)$ gets at least $m\lceil\log_2(m/2)\rceil = (l+r)\lceil\log_2\max(l,r)\rceil$ credits, as required by invariant 2.

A call to $p(u,x,v)$ splits $\pi(u,v)$ into two paths $\pi(u,z)$ and $\pi(z,v)$ with a and b edges, respectively. The call to $p(u,x,v)$ divides its credits between its recursive calls evenly according to subpath size. If $z = x$, then the two calls to $f()$ get at least $l\lceil\log_2\max(l,r)\rceil \geq l\lceil\log_2 l\rceil$ and $r\lceil\log_2\max(l,r)\rceil \geq r\lceil\log_2 r\rceil$ credits, satisfying invariant 1. If $z \neq x$, then without loss of generality assume that z belongs to $\pi(u,x)$ and line 8 is executed; the other case is symmetric. The call to $f(u,z)$ gets at least $a\lceil\log_2\max(l,r)\rceil \geq a\lceil\log_2 a\rceil$, as required. The call to $p(z,x,v)$ gets at least $b\lceil\log_2\max(l,r)\rceil \geq b\lceil\log_2\max(b-r,r)\rceil$, as required by invariant 2. This completes the proof. ▮

Altogether the calls to $f()$ and $p()$ take $O(n\log n)$ time, plus the time spent building path hulls at line 3. The preceding lemma shows that all the executions of line 3 take only $O(n\log n)$ time, and hence the entire algorithm runs in $O(n\log n)$ time.

4.6 Implementation

The algorithm described in this section has been implemented. The implementation is more general than the algorithm we have so far described: it correctly handles the cases of collinear vertices on convex hulls and parallel rays on bi-infinite chains. These improvements are not difficult. Handling collinear vertices requires two changes: the program detects and merges consecutive collinear polygon edges, reporting them to the user, and the **while** loop tests in Melkman's algorithm are changed from "v is left of" to "v is on or to the left of the line supporting." When a chain has parallel infinite rays, its convex hull may be bounded by two infinite lines. Even though the hull has no vertices, the chain nevertheless has valid splitting vertices on the hull. The program finds a splitting vertex by searching for an extreme vertex in a direction perpendicular to the infinite rays, using a special case to avoid selecting u or v.

As input the program takes a list of polygon vertices in order (either clockwise or counterclockwise), specified as x-y coordinate pairs. As output the program produces a

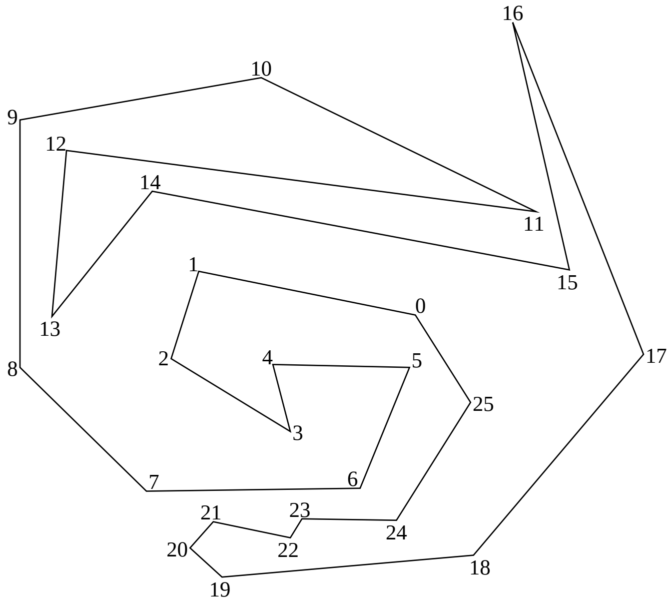

Figure 11: Sample program input, displayed as a polygon

list of the splitting vertices in the order they are computed; it also produces a correctly parenthesized boolean formula for the input polygon. When the program is applied to the polygon shown in Figure 11, it produces the following (slightly abbreviated) output:

```
main: Calling f() on 8..17
p: splitting at vertex 16, 15, 9, 10, 13, 11,
                 12, 14
main: Calling f() on 17..25, 0..8
p: splitting at vertex 18, 19, 20, 0, 25, 24,
                 21, 22, 23, 7, 1, 6, 5, 2, 3, 4

Boolean formula is:

(8 * 9 * (10 * (11 + 12) + 13 * 14) + 15) * 16 *
17 * 18 * 19 * (20 * (21 + 22 * 23) + 24 + 25 +
                (0 + (1 + 2 + 3 * 4) * 5 * 6) * 7)
```

In this formula, the number i refers to the edge joining vertex i to vertex $(i + 1) \bmod n$; here n is 26.

The program produces graphical as well as textual output. Marc H. Brown created the graphical displays using an algorithm animation system he is developing at the DEC Systems Research Center. (Brown's thesis [3] provides more information on algorithm animation.) The animation shows multiple color views of the state of the computation, which are updated as the program runs. One view shows the input polygon and highlights chains, convex hulls, and splitting vertices as the algorithm operates on them. Another view pictorially displays the formula for each subchain as a boolean combination of half-planes.

Other views show the incremental construction of the boolean formula and its parse tree. Each view emphasizes a different aspect of the algorithm; together they illuminate many of its important features.

The title page of this report shows a black-and-white snapshot of the animated algorithm running on a small example. The snapshot features three views of the algorithm. The "Geometry" view shows the polygon; in the snapshot, this view highlights the initial splitting vertices. The "Formula" view shows the development of the boolean formula over time. The "CSG Parse Tree" view shows the parse tree corresponding to the formula; each node of the tree displays the region that corresponds to the subtree rooted at the node.

5 Formulæ for polyhedra

We have shown that the interior of a simple polygon can be represented by a Peterson-style formula: a monotone boolean formula that uses each literal once. We would like to find such a formula for a polyhedron P in 3-space. Here, the literals are half-spaces bounded by the planes supporting the faces.

In this section we prove that not all polyhedra have a Peterson-style formula. Figure 12 illustrates a simplicial polyhedron (each face is a triangle) with eight vertices and twelve faces. Six of the faces are labeled; the six unlabeled faces lie on the convex hull of P. The edge between C and C' is a convex angle. The half-spaces defined by faces A and B intersect the faces A' and B'. Similarly, the half-spaces defined by A' and B' intersect faces A and B. After we establish a couple of lemmas, we will prove that P has no Peterson-style formula by assuming that it has one and deriving a contradiction.

We begin by observing that any collection of planes divides space into several convex regions. (In the mathematical literature, this division is usually called an *arrangement* [5].) If a polyhedron P has a CSG representation in terms of half-spaces, then we can specify a subset of the planes bounding these half-spaces and derive a representation for the portion of P inside any convex region determined by the subset.

More precisely, let f be a boolean formula whose literals are the half-spaces of P; we can think of f as an expression tree. If the tree for f has nodes a and b, then we denote the lowest common ancestor of a and b in f by $\mathrm{lca}_f(a,b)$. Let $H_1, H_2, \ldots, H_m$ be a subset of the half-spaces defined by faces of P. Each point in space can be assigned a string $\alpha \in \{0,1\}^m$ such that the i-th character of α is 1 if and only if the point is in half-space H_i. All the points assigned the string α are said to be in the region R_α. We use $f|_\alpha$ to denote the formula obtained by setting each $H_i = \alpha_i$ in f and simplifying the result by using algebraic rules: $a1 = 1a = a$, $a0 = 0a = 0$, $a + 1 = 1 + a = 1$, and $a + 0 = 0 + a = a$. The expression tree for $f|_\alpha$ inherits several important properties from the expression tree for f:

Lemma 5.1 *Let f be a formula that uses the half-spaces $H_1, H_2, \ldots, H_m$ (and perhaps others) and let α be a string in $\{0,1\}^m$. Then the derived formula $f|_\alpha$ has the following three properties:*

1. *If f is monotone, then so is $f|_\alpha$. Similarly, if f is Peterson-style, so is $f|_\alpha$.*
2. *If the expression tree for $f|_\alpha$ has nodes a, b, and c, with $c = \mathrm{lca}_{f|_\alpha}(a,b)$, then $c = \mathrm{lca}_f(a,b)$ in the tree for f.*

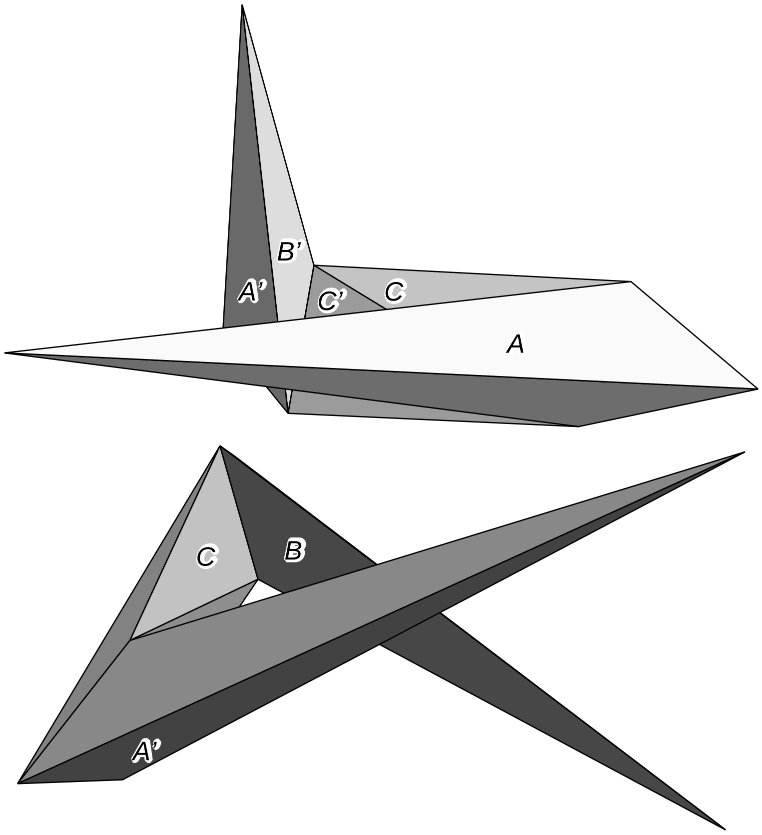

Figure 12: Two views of a simplicial polyhedron with no Peterson-style formula

3. *If the expression tree for $f|_\alpha$ contains a node c at depth k, then the tree for f contains the node c at depth $\geq k$.*

   Proof: All three properties are maintained by the rules that form the expression tree for $f|_\alpha$ by simplifying the expression tree for f. ∎

The next lemma shows the interaction between the region R_α and boolean formulæ $f|_\alpha$.

Lemma 5.2 *If a polyhedron P has a formula f that uses half-spaces $H_1, H_2, \ldots, H_m$ (and others) then, for any string $\alpha \in \{0,1\}^m$, the portion of P inside the region R_α is described by the formula $f|_\alpha$.*

> **Proof:** The statement above simply says that formulæ f and $f|_\alpha$ agree inside the region R_α. This follows from the definition of $f|_\alpha$ and the fact that the simplification rules do not change the value of the formula. ∎

Two corollaries of Lemma 5.2 give us constraints on the formula of a polyhedron based on its edges and faces. In these corollaries and the discussion that follows, we will add an argument to a formula $f|_\alpha$ to emphasize which half-spaces are not fixed by the string α.

Corollary 5.3 *Let P be a polyhedron with Peterson-style formula f. If faces A and B of P meet at an edge of P, the operator in f that is the lowest common ancestor of A and B, $\mathrm{lca}_f(A, B)$, is an "and" if and only if A and B meet in a convex angle.*

> **Proof:** Let $H_1, H_2, \ldots, H_m$ be the half-spaces of P except for the two defined by A and B. Choose an interior point of the edge formed by faces A and B, and let α be its string. The two-variable formula $f|_\alpha(AB)$ must describe the polyhedron in the vicinity of the edge, so by Lemma 5.1(2), $\mathrm{lca}_f(A, B)$ is an "and" if and only if A and B meet in a convex angle. ∎

Corollary 5.4 *Let P be a polyhedron with a Peterson-style formula f using half-spaces $H_1, H_2, \ldots, H_m$ and A and B. If the half-space defined by face B intersects face A at some point with string α then $f|_\alpha(AB) = A$.*

> **Proof:** The two-variable formula $f|_\alpha(AB)$ must describe the face A both inside and outside the half-space of B, so B cannot appear in the formula. ∎

Now we are ready to look at the polyhedron P in Figure 12. Suppose P has a Peterson-style formula f. Then it has a formula $f|_{111111}(ABCA'B'C')$ that describes the region inside the unlabeled faces. We look at the constraints on this formula and derive a contradiction.

Consider the three faces A, B, and C. By Corollary 5.3 we know that $\mathrm{lca}_f(B, C) =$ "or" and $\mathrm{lca}_f(A, B) =$ "and". Corollary 5.4 applied to faces A and C implies that the formula describing these three faces is

$$f|_{\alpha_1}(ABC) = A(B + C), \tag{1}$$

where the string α_1 appropriately fixes all the half-spaces except A, B, and C. Similarly, the formula describing A', B', and C' is

$$f|_{\alpha_2}(A'B'C') = A'(B' + C'). \tag{2}$$

Now consider the region inside all unlabeled half-spaces and outside C and C'. The portion of P within this region can be described by a Karnaugh map [12]:

		AB			
		00	01	11	10
$A'B'$	00	0	0	1	0
	01	0	0	1	0
	11	1	1	?	1
	10	0	0	1	0

The '?' appears because four planes cut space into only fifteen regions; since we want a monotone formula, Lemma 5.1(1) forces us to make it a '1'. Examining all Peterson-style formulæ on A, B, A', and B' reveals that the only formula with this map is

$$f|_{\alpha_3}(ABA'B') = (AB) + (A'B'). \tag{3}$$

In order to combine the formulæ 1, 2, and 3 into a single formula on six variables, we must determine which operators are repeated in the three formulæ. We knew from formulæ

1 and 2 that the operators $\text{lca}_f(A,B)$ and $\text{lca}_f(A',B')$ were both "and"s—now we know that they are distinct "and"s because $\text{lca}_f(\text{lca}_f(A,B),\text{lca}_f(A',B'))$ = "or" in formula 3. The "or"s of the first two formulæ are distinct because they are descendents of distinct "and"s. Finally, by Lemma 5.1(3), the "or" of formula 3 is different from the other "or"s because it is not nested as deeply as the "and"s.

Thus, all five operators of the formula on the six labeled half-spaces appear in the formulæ 1, 2, and 3. Using the nesting depth of the operators, we know that the formula looks like $(\square(\square+\square))+(\square(\square+\square))$. Filling in the half-space names gives the formula for the portion of P inside the unlabeled faces:

$$f|_{111111}(ABCA'B'C') = (A(B+C)) + (A'(B'+C')).$$

Notice, however, that in this formula the lca of C and C' is an "or". Thus $\text{lca}_f(C,C')$ = "or". But this contradicts Corollary 5.3, so the formula above cannot represent the portion of P inside the convex hull of P. This contradiction proves that P has no Peterson-style formula.

6 Closing observations

Though the previous section proves that not all polyhedra have Peterson-style formulæ, it may still be the case that the interior of a polyhedron with n faces can be represented by a formula using $O(n)$ literals. The trivial upper bound on the size of a formula is $O(n^3)$. In fact, the interiors of any set of cells formed by a collection of n planes can be described by a formula that represents each convex cell as the "and" of its bounding planes and "or"s the cell representations together. The size of the formula is at worst the total number of sides of the cells formed by n planes, which is known to be $O(n^3)$ [5].

Recent work by Paterson and Yao improves the upper bound on the size of a formula to $O(n^2)$ [19]. Their method, like the trivial one, slices the polyhedron into convex polyhedra by cutting along the planes supporting the faces. By cutting along the planes in a particular order, and by cutting a subpolyhedron with a plane only when the defining face lies on the subpolyhedron boundary, the algorithm produces convex pieces whose total number of sides is $O(n^2)$. The formula corresponding to this decomposition has size $O(n^2)$.

Neither of the formulæ just described is monotone. Every polyhedron does have a monotone formula, however: For the formula of each cell in the trivial decomposition, we can "and" all of the half-spaces that contain the cell and then "or" these cell representations. This formula uses no negations—to see that it defines the polyhedron, notice first that every point in the polyhedron is in some cell. Second, to move from a cell in the polyhedron to any point outside the polyhedron you must cross out of a half-space when you first leave the polyhedron. Thus the representation of an interior cell contains no points outside the polygon. This implies that the formula evaluates to *false* at exterior points and shows that there is a monotone formula with size $O(n^4)$.

It would be interesting to characterize the polyhedra that can be represented by Peterson-style formulæ. Peterson [21] showed that the representations of polygons give such formulæ for extrusions and pyramids. We would like to extend this class.

We can also consider looking for formulæ for polygons with curved edges. If each edge of a polygon is a piece of a bi-infinite curve that does not intersect itself, then an edge defines a half-space with a curved boundary—perhaps we can represent the polygon as a formula on these half-spaces.

Suppose we restrict the curves so that any two intersect in at most one point (the pseudoline condition). Then it can be proved that there is always a polygon vertex v such that the extensions to infinity (through v) of the two edges adjacent to v never intersect the boundary of the polygon. Since this vertex v can be used as a splitting vertex, such polygons have Peterson-style formulæ.

If the curves are allowed to intersect in more than one point, then it may be impossible for a formula to represent a polygon without representing other areas of the plane, too. For example, if two ovals intersect in four points, then the region inside the first oval and outside the second is disconnected. No boolean formula can represent one connected component without including the other (unless we are allowed to introduce auxiliary curves).

Acknowledgments

We thank D. P. Peterson and Jorge Stolfi for their helpful discussions and polyhedra. We also thank Cynthia Hibbard and Martín Abadi for their careful reading of the manuscript.

References

[1] J. Boyse and J. Gilchrist. GMSolid: Interactive modeling for design and analysis of solids. *IEEE Computer Graphics and Applications*, 2:86–97, 1982.

[2] C. Brown. PADL-2: A technical summary. *IEEE Computer Graphics and Applications*, 2:69–84, 1982.

[3] M. H. Brown. *Algorithm Animation.* ACM Distinguished Dissertations. MIT Press, 1988.

[4] B. M. Chazelle. Computational geometry and convexity. Technical Report CMU-CS-80-150, Carnegie-Mellon University, Department of Computer Science, Pittsburgh, PA, 1980.

[5] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1987.

[6] W. Franklin. Polygon properties calculated from the vertex neighborhoods. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 110–118, 1987.

[7] R. L. Graham and F. F. Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4:324–331, 1983.

[8] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[9] L. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 100–111, 1983.

[10] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan. Sorting Jordan sequences in linear time. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 196–203, 1985.

[11] D. T. Lee. On finding the convex hull of a simple polygon. *Internat. J. Comput. Inform. Sci.*, 12:87–98, 1983.

[12] M. M. Mano. *Digital Logic and Computer Design.* Prentice-Hall, 1979.

[13] M. Mäntylä. *An Introduction to Solid Modeling.* Computer Science Press, 1987.

[14] D. McCallum and D. Avis. A linear algorithm for finding the convex hull of a simple polygon. *Information Processing Letters*, 9:201–206, 1979.

[15] A. Melkman. On-line construction of the convex hull of a simple polyline. *Information Processing Letters*, 25:11–12, 1987.

[16] M. Mortenson. *Geometric Modeling.* John Wiley & Sons, 1985.

[17] R. Newell. Solid modelling and parametric design in the Medusa system. In *Computer Graphics '82, Proceedings of the Online Conference*, pages 223–235, 1982.

[18] J. O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, 1987.

[19] M. Paterson and F. Yao. Binary partitions with applications to hidden-surface removal and solid modelling. In *Proceedings of the 5th ACM Symposium on Computational Geometry*, pages 23–32, 1989.

[20] T. Pavlidis. Analysis of set patterns. *Pattern Recognition*, 1:165–178, 1968.

[21] D. Peterson. Halfspace representation of extrusions, solids of revolution, and pyramids. SANDIA Report SAND84-0572, Sandia National Laboratories, 1984.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.

[23] A. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12:437–464, 1980.

[24] A. A. Schäffer and C. J. Van Wyk. Convex hulls of piecewise-smooth Jordan curves. *Journal of Algorithms*, 8:66–94, 1987.

[25] W. Tiller. Rational B-splines for curve and surface representation. *IEEE Computer Graphics and Applications*, 3, 1983.

[26] S. B. Tor and A. E. Middleditch. Convex decomposition of simple polygons. *ACM Transactions on Graphics*, 3(4):244–265, 1984.

[27] H. Voelcker, A. Requicha, E. Hartquist, W. Fisher, J. Metzger, R. Tilove, N. Birrell, W. Hunt, G. Armstrong, T. Check, R. Moote, and J. McSweeney. The PADL-1.0/2 system for defining and displaying solid objects. *ACM Comput. Gr.*, 12(3):257–263, 1978.

[28] J. R. Woodwark and A. F. Wallis. Graphical input to a Boolean solid modeller. In *CAD 82*, pages 681–688, Brighton, U.K., 1982.