Persistent Triangulations∗

Guy Blelloch, Hal Burch†, Karl Crary, Robert Harper, Gary Miller, and Noel Walkington
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Triangulations of a surface are of fundamental importance in computational geometry, computer graphics, and engineering and scientific simulations. Triangulations are ordinarily represented as mutable graph structures for which both adding and traversing edges take constant time per operation. These representations of triangulations make it difficult to support *persistence*, including "multiple futures", the ability to use a data structure in several unrelated ways in a given computation; "time travel", the ability to move freely among versions of a data structure; or parallel computation, the ability to operate concurrently on a data structure without interference.

We present a purely functional interface and representation of triangulated surfaces, and more generally of simplicial complexes in higher dimensions. In addition to being persistent in the strongest sense, the interface more closely matches the mathematical definition of triangulations (simplicial complexes) than do interfaces based on mutable representations. The representation, however, comes at the cost of requiring $O(\lg n)$ time for traversing or adding triangles (simplices), where n is the number of triangles in the surface. We show both analytically and experimentally that for certain important cases, this extra cost does not seriously affect end-to-end running time. Analytically, we present a new randomized algorithm for 3-dimensional Convex Hull based on our representations for which the running time matches the $\Omega(n \lg n)$ lower-bound for the problem. This is achieved by using only $O(n)$ traversals of the surface. Experimentally, we present results for both an implementation of the 3-dimensional Convex Hull and for a terrain modeling algorithm, which demonstrate that, although there is some cost to persistence, it seems to be a small constant factor.

Capsule Review

Capsule summary goes here.

1 Introduction

The PSciCo[1] Project at Carnegie Mellon is investigating the use of functional programming languages for scientific computing. Our working hypothesis is that, as the scale and complexity of scientific computing problems increases, so the need for more sophisticated languages also increases. A rich type structure that provides

∗ This research was supported in part by NSF Grant CCR-9706572.
† Research supported by an NSF graduate fellowship.
[1] Parallel Scientific Computing

not only conventional numeric and vector types but also function types, recursive types, and user-defined abstract types seems especially important, as does a flexible module system that supports libraries of interchangeable components. To achieve acceptable performance, and to expand feasible problem sizes, it is also important to exploit parallelism, which is increasingly available on stock platforms. To address these requirements, we are investigating the use of functional programming as a basis for advanced scientific computing, combining NESL (Blelloch, 1996), an applicative language that provides implicit data parallelism, and Standard ML (Milner *et al.*, 1997), a higher-order language with a rich module system. We are working on a variety of problems, including n-body problems in physics, computational geometry problems, such as Delaunay Triangulation and multi-dimensional convex hull problems, and the solution of partial differential equations by the finite element method.

This paper focuses on a particular problem, the representation and construction of triangulated surfaces. Triangulated surfaces (more generally, simplicial complexes) are widely used in a number of fields such as computer graphics and scientific computing. For example, the starting point for solving a boundary value problem by the finite element method is the triangulation of the region over which the solution is constructed. Standard representations of surfaces (Guibas & Stolfi, 1985; Berg *et al.*, 1997) are *ephemeral* in the sense that they are defined in terms of mutable graph structures. Once a modification to the surface is made, the previous version of the surface is destroyed. In some situations it is convenient to have a *persistent* representation of the surface, one for which operations on the surface do not destructively update it, but rather create an independent "version" while retaining the original. For example, a persistent surface representation admits a simple formulation of Ruppert's algorithm (Ruppert, 1995) for triangulating regions with boundaries in which we "undo" a partial triangulation on the fly in the rare case that boundary constraints lead to undesirable acute triangles. Persistent representations of surfaces also permit interactive exploration and incremental modification of a triangulation through the convenience of "time travel", the ability to maintain a complete history of the evolution of a data structure.

This paper is concerned with the use of persistent representations of triangulated surfaces. A straightforward approach to achieving persistence is to replace uses of arrays by a purely functional (immutable) dictionary using balanced search trees (Myers, 1984). The disadvantage of this approach is that it imposes an additional $O(\lg n)$ cost per operation stemming from the logarithmic cost of dictionary lookup compared to the constant cost of array access. Various attempts have been made to reduce this overhead (Driscoll *et al.*, 1989; Dietz, 1989; O'Neill & Burton, 1997; Okasaki, 1998). These methods tend to be quite complex, apply only to particular data structures, or employ imperative (mutable) data structures internally. Reliance on underlying mutable data structures complicates parallelization, since interlocks are required to avoid interference between processors. In contrast purely functional implementations parallelize without special provision, precisely because they avoid mutation entirely.

Rather than attempt to improve the *local* (per-operation) cost of operations on the data structure, we instead emphasize the *global*, or *end-to-end*, efficiency of algorithms that make use of the data structure. By being clever at the higher level we can get away with a naïve purely functional representation, without sacrificing the global efficiency of the algorithms of interest. In this paper we explore this approach in the context of a persistent representation of triangulated surfaces. We employ a simple surface representation with an elegant interface that follows very closely the mathematical definition of a surface. Although we incur a logarithmic cost overhead per operation, we nevertheless are able to use our data structure to build efficient implementations of two higher-level algorithms, in both an asymptotic and empirical sense.

The first example is the construction of the convex hull (Berg *et al.*, 1997) (a triangulated surface) of a set of n points in three-dimensional space. An $\Omega(n \lg n)$ lower bound for the problem is well-known (Berg *et al.*, 1997), and matching upper bounds are achieved by several algorithms in the ephemeral case. A naïve transcription of these algorithms to the persistent case would result in a sub-optimal $O(n \lg^2 n)$. At first glance this may seem to be an inherent cost of a locally sub-optimal representation of surfaces. We prove that this supposition is false by giving a new, randomized convex hull algorithm, which we call the *bulldozer algorithm*, that achieves the optimal $O(n \lg n)$ time bound in the expected case, despite the reliance on the simple persistent representation of surfaces. The key observation is that, whereas our algorithm requires $O(n \lg n)$ floating point operations (to determine whether a point lies inside or outside of the partially-constructed hull), it requires only $O(n)$ surface operations. Since each surface operation takes $O(\lg n)$ time, the running time of the algorithm is still bounded by $O(n \lg n)$. To assess the practicality of this algorithm, we measure its performance on a variety of data sets. Our second example is the implementation of a terrain modeling algorithm given by Garland and Heckbert (Garland & Heckbert, 1995) using our persistent surface representation. Here no analytic bounds are known, but we are able to demonstrate practical efficiency on realistic data sets.

In Section 2 we describe our representation of triangulated surfaces as simplicial complexes. In Section 3 we present a brief summary of the bulldozer algorithm, which is described more fully in a companion report (Burch *et al.*, 2000). In Section 4 we evaluate the performance of the bulldozer algorithm on a variety of data sets, and compare it to the performance of the Minnesota Quickhull (Barber *et al.*, 1996) implementation. In Section 5 we evaluate an implementation of Garland and Heckbert's terrain modeling algorithm (Garland & Heckbert, 1995) based on our representation of surfaces.

2 Surfaces

Both the convex hull algorithm and the terrain modeling algorithm presented in later sections construct a two dimensional surface embedded in three dimensional space. In the case of the convex hull, this is the surface of the smallest enclosing convex polytope of a set of points, and in the case of the terrain modeling, the surface represents an approximation to the topography of a geographical region. In both cases it is convenient to think of the surface as a connected set of triangles covering the surface; if the surface is specified by polygonal faces they are subdivided into triangles. Consequently, the surfaces of interest are sometimes called *triangulated surfaces*, or simply *triangulations*.

Following Giblin (Giblin, 1977), we define a *closed surface* to consist of a set of triangles satisfying the following three conditions:

1. Any two triangles have at most one vertex or one edge (and its two vertices) in common; no other forms of overlap are permitted. This is called the *intersection condition*.
2. The surface is *connected* in the sense that there is a path from any vertex to any other vertex consisting of edges of the triangles of the surface.
3. For each vertex, the set of edges opposite that vertex in any triangle, called the *link* of that vertex, forms a simple, closed polygon.

This definition relies on the familiar concept of a triangle. A triangle consists of a set of three distinct vertices, specified in some order. This raises the question of when two triangles are equivalent. Under an *ordered* interpretation, $\triangle ABC$ is distinct from both $\triangle BCA$ and $\triangle CAB$, even though they enumerate the vertices in the same sequence, and is also distinct from $\triangle ACB$, which reverses the order of presentation. Two orderings that differ by an even permutation (i.e., that can be obtained from one another by an even number of swaps) are said to determine the same *orientation*. Thus $\triangle ABC$, $\triangle BCA$ and $\triangle CAB$ all have the same orientation, whereas $\triangle ACB$, $\triangle BCA$ and $\triangle CAB$ have the opposite orientation. The orientation may be thought of as determining two "sides" of a triangle; $\triangle ABC$ is the "front" of $\triangle ABC$, and, correspondingly, $\triangle ACB$ is the "back" of $\triangle ABC$. Under an *oriented* interpretation we identify triangles that have the same orientation, and distinguish those that do not. This idea generalizes to higher dimensions in that two orderings of vertices have the same orientation iff they differ by an even permutation.

Following Giblin, we maintain a careful distinction between the configuration of the triangles on the surface (i.e., their adjacency relationships) and the embedding of the triangles in three-dimensional space (i.e., the assignment of coordinates to their vertices). When embedding a triangle $\triangle ABC$ in three-dimensional space, we require that the points assigned to the vertices be *affinely independent*, which is to say that the vectors $B - A$ and $C - A$ are linearly independent, or, equivalently, that the three points are not collinear. The convex hull algorithm will determine not only the configuration of triangles, but also their embedding in three-dimensional space.[2]

A closed surface is a special case of the more general concept of a *simplicial complex* (Giblin, 1977; Alexandroff, 1961), which applies in an arbitrary dimension. Our implementations of the three-dimensional convex hull and of the terrain modeling algorithm are based on an abstract type of simplicial complexes. Not only does this

[2] To avoid degeneracies and to simplify the presentation, we assume that the input set of points to the hull algorithm has the property that no four points are coplanar.

support generalization to higher-dimensional spaces, but it also allows us to experiment with various implementations of simplicial complexes without disturbing the application code. Indeed, we experimented with several different implementations before settling on the one we describe here.

Just as a closed surface is a set of triangles satisfying some conditions, a simplicial complex is a set of *simplices* over a set of *vertices* satisfying some related conditions. A zero-dimensional simplex is a "bare" vertex, a one-dimensional simplex is a line segment, a two-dimensional simplex is a triangle, a three-dimensional simplex is a tetrahedron, and so on. A complex is a configuration of simplices subject to some simple conditions.

We assume we are given a totally ordered set V of *vertices*.[3] An *n-dimensional ordered simplex*, or *n-simplex*, is an $(n+1)$-tuple of distinct vertices. An *oriented* simplex is an equivalence class of ordered simplices with the same orientation. A simplex s is a *sub-simplex*, or a *face*, of a simplex t, written $s \leq t$, iff s is a subsequence (not necessarily proper) of t. An *n-dimensional, oriented, pure simplicial complex*, or just *n-complex* for short, consists of a set V of vertices and a set S of oriented simplices satisfying the following conditions:

1. Every vertex determines a 0-simplex. We usually do not distinguish between a vertex v and its associated 0-simplex (v).
2. Every sub-simplex of a simplex in S is also a simplex of S.
3. Every simplex $s \in S$ is a sub-simplex of some n-simplex in S.

A *closed surface* is a 2-complex in which the link of every 0-simplex is a simple, closed polygon having that 0-simplex as an interior point.

The *signature* (interface) of the simplicial complex abstract type is given in Figure 3. This abstraction relies on an abstract type of vertices, whose signature is given in Figure 1, and an abstract type of simplices, whose signature is given in Figure 2. Taken together, these signatures summarize the entire suite of operations available to applications that build and manipulate complexes. We have omitted some operations that are not required in this paper, such as a list of the exceptions.

The signature `VERTEX` specifies that vertices admit a total ordering, which is required for the efficient implementation of simplicial complexes. We associate a point with each vertex; this is used to embed a simplex in space, as described earlier. The embedding is established by the `new` operation, which creates a "new" vertex at the specified point. The `state` is used to generate new vertex "labels" purely functionally. The location of a vertex in space is obtained using the `loc` operation, which yields the point in space associated with vertex. The type of points is left completely unspecified since the simplicial complex package need not be concerned with its exact representation.

The signature `SIMPLEX` defines the abstract type of (ordered) simplices over a given type of vertices. As with vertices, we require that simplices be totally ordered by some unspecified order relation so that simplices may be used as keys in a

[3] This total ordering is not ordinarily required in the mathematical setting, but is necessary for implementation reasons.

```
signature VERTEX =
  sig
    type vertex
    val compare : vertex * vertex -> order
    type state
    type point
    val new : state * point -> state*vertex
    val loc : vertex -> point
  end
```

Fig. 1. Signature of Vertices

```
signature SIMPLEX =
  sig
    structure Vertex : VERTEX
    type simplex
    val compare : simplex * simplex -> order
    val dim : simplex -> int
    val vertices : simplex -> Vertex.vertex seq
    val simplex : Vertex.vertex seq -> simplex
    val down : simplex -> Vertex.vertex * simplex
    val join : Vertex.vertex * simplex -> simplex
    val faces : simplex -> simplex seq
    val flip : simplex -> simplex
  end
```

Fig. 2. Signature of Simplices

dictionary. The operation `dim` yields the dimension of a simplex. The *apex* of the simplex is the first vertex of the sequence of vertices defining the simplex. The `vertices` operation yields the sequence of vertices of a simplex, apex first.[4] The `simplex` operation creates an n-simplex from a sequence of $n+1$ vertices. The `faces` operation yields a sequence of $(n-1)$-dimensional sub-simplices of a given n-simplex in arbitrary order. The `flip` operation inverts the orientation of a simplex (flips to its reverse side). The `down` operation takes an n-simplex and returns its apex and the $(n-1)$-simplex opposite the apex. The `join` operation builds an n-simplex from a given vertex and $(n-1)$-simplex, taking the vertex as apex and the $(n-1)$ simplex as its opposite face.

The signature `SIMPCOMP` specifies the abstract type of oriented simplicial complexes. There are no mutation operations on complexes. Instead we supply operations to create new complexes from old, as discussed in the introduction. The type `'a complex` of n-dimensional simplicial complexes is parameterized by a type `'a` of data values associated with the n-simplices of the complex. The `new` operation creates an empty complex of specified dimension n. The sequence of vertices of a

[4] We make use of an abstract type of sequences (`'a seq`), a form of immutable array whose primitive operations are designed to support implicit parallelism (Blelloch, 1996).

```
signature SIMPCOMP =
  sig
    structure Simplex : SIMPLEX
    type 'a complex
    val dim : 'a complex -> int
    val new : int -> 'a complex
    val isEmpty : 'a complex -> bool
    val vertices : 'a complex -> Simplex.Vertex.vertex seq
    val simplices : 'a complex -> int -> Simplex.simplex seq
    val data : 'a complex * Simplex.simplex -> 'a option
    val grep : 'a complex -> int * Simplex.simplex -> Simplex.simplex seq
    val find : 'a complex * Simplex.simplex -> Simplex.simplex seq
    val add : 'a complex * Simplex.simplex * 'a -> 'a complex
    val rem : 'a complex * Simplex.simplex -> 'a complex
    val update : 'a complex * Simplex.simplex * ('a -> 'a) -> 'a complex
  end
```

Fig. 3. Signature of Simplicial Complexes

complex are returned by the `vertices` operation, in an arbitrary order. The simplices of a given dimension are returned by the `simplices` operation. The `grep` operation finds all the simplices of maximal dimension having a given simplex as a face. More precisely, given a dimension $d \leq n$ and a d-simplex s, `grep` returns a sequence containing (in unspecified order) the simplices of dimension `d` having s as a face. The `find` operation is a specialization of `grep` for dimension $n-1$. The operation `add` adds a simplex to a complex, with specified data value; to ensure that the condition 3 in the definition of simplices is preserved, we may only add an n-simplex to an n-complex. The operation `rem` removes a simplex from a complex, yielding the reduced complex. The `update` operation applies a specified function to the data associated with the given simplex. We note that for all operations on a simplicial complex the simplices are viewed as oriented but not ordered (all orders of the same orientation are considered equivalent). For example if a simplex with a particular order is added to the complex, later searches on other orders with the same orientation will find that simplex. Similarly the `simplices` and `grep` functions will only return a single order for each oriented simplex.

In our implementation, an n-simplex is represented by a sequence of vertices of length $n+1$, with the apex being the lead vertex of the sequence. The `down` operation strips off the apex and returns the remaining $(n-1)$-simplex, as described above. We implement complexes using the `Map` signature taken from the SML/NJ library. An $n > 1$ complex is represented by a mapping from each vertex to the $(n-1)$-complex consisting of the faces opposite it. A 1-complex is implemented specially to avoid the overhead of maintaining the map.

We may build an n-complex by a sequence of $n-1$ applications of a "bootstrapping functor" that builds an n-complex from an $(n-1)$-complex, starting with the direct implementation of the 1-complex. However, for reasons of efficiency, we choose to implement the 2-complexes directly, rather than by bootstrapping. In this optimized implementation we use the first vertex of a simplex as a key into a red-black tree (Bayer, 1972; Okasaki, 1998). Each node of the red-black tree then stores as its value an association list that maps the second vertex to the third vertex and the data. Using an association list is adequate in practice since the number of entries is small (the average number is 6). To make the implementation optimal in theory one could convert to a balanced tree if the size of the list becomes too long.

In our direct implementation searching for a simplex involves searching the red-black tree and then the association list. Adding a simplex involves searching the red-black tree to see if the vertex is already there. If it is, the simplex is added to the existing association list, otherwise a new association list is created. When a simplex is added, it is added to the tree in all three orders with the same orientation. This is important so that searching (i.e. `grep` and `find`) can be implemented efficiently. Deleting a simplex involves searching the tree and deleting the simplex from the corresponding association list. If the association list becomes empty, then the tree node is also deleted. As with addition, the deletion needs to be executed in all three orders.

3 Convex Hull: The Bulldozer Algorithm

It is well known that the problem of constructing the convex hull of a set of points in three dimensions requires $\Omega(n \lg n)$ time (Berg *et al.*, 1997). Asymptotically optimal algorithms for the problem are also known (Clarkson & Shor, 1989; Chazelle, 1991) for the ephemeral case. In this section we will give an optimal randomized algorithm for the persistent case. The algorithm represents the surface of the convex hull as a two-dimensional simplicial complex.

We will be concerned with *incremental* methods that expand the convex hull of a set of points to include a new point. Many algorithms, including our own, are based on a *tent construction.* Given a point p exterior to the hull of a set of points, we may extend the hull to include this point as follows. View the exterior point p as a *light source* illuminating a subset of the faces of the hull (note that a face is either fully light or fully dark). These lit faces are removed by the construction. The boundary of the lit faces is a set of edges called the *horizon.* The construction then creates a pyramidal *tent* whose apex is the exterior point and whose base is the horizon. This construction extends the convex hull to include the point p as a new vertex (see Figure 4). Any points that become interior to the hull when a tent is added are discarded. A complete hull is constructed by going through the points and adding them to the hull one at a time.

Several incremental algorithms based on the tent construction are known; they differ in how the exterior point is chosen, and how the set of exterior points is maintained during the construction. During construction most of these algorithms maintain, for each exterior point, one face that is visible to that point. Clarkson and Shor's algorithm (Clarkson & Shor, 1989), the Minnesota Quickhull algorithm (Barber *et al.*, 1996), the Motwani and Raghavan algorithm (Motwani & Raghavan, 1995), and the Bulldozer algorithm described here, all maintain such information.

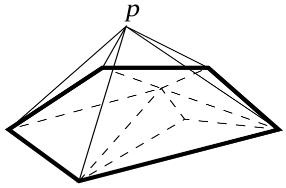

Fig. 4. Example of a tent with exterior point p. The lit faces belonging to the old hull are marked with dashed borders and the horizon is shown in bold.

The motivation for keeping this association is that, since the set of faces that are visible to any one point is connected, knowing one visible face allows the algorithm to walk through the visible faces to find them all. When a tent is added to the hull the lit faces are removed from the hull. To maintain the association of points to faces, each point that is associated with one of these lit faces needs to be associated with a new visible face, or determine that it is now interior and can be dropped.

One reason that previous algorithms are inefficient in the persistent case is that the process of finding a new visible face can require the traversal of "too many" faces. Since in the persistent case each traversal step requires $O(\lg n)$ steps, we must bound the overall number of traversals to $O(n)$ to ensure that the construction may be completed in $O(n \lg n)$ time. To do this the algorithm is careful about how it associates a face with each point and how it reassigns the points. The algorithm begins by selecting a point that will always be interior to the hull—the *center point* of the hull. Consider the rays from the center point to each exterior point. If such a ray penetrates the surface at a face, we associate the point to that face.[5] We associate exterior points with their appropriate faces by storing on each face a list of the associated points.

Each step of the algorithm selects an exterior point p uniformly at random for addition to the hull. Since the points are associated with faces, rather than faces with points, to do so the algorithm first selects at random a face that has points associated with it, where the probability of selecting a face is proportional to the number of associated points. It then selects a random point associated with this face to serve as a "light source". Once the light source p is selected, the algorithm finds and then removes all faces visible to p by searching the surface in a particular order starting with the associated face of p. The search defines a directed acyclic graph whose nodes are the visible faces and the horizon edges, and whose arcs connect adjacent faces or a face with one of its horizon edges; we call this graph the *walking graph* of the hull with respect to the light source. Figure 5 gives one example of a walking graph.

Walking graphs have several important properties:

1. The associated face of the light source has in-degree zero.
2. The graph is acyclic.
3. All nodes of the graph are reachable from the associated face of the light source.

[5] We make the assumption that no four points are co-planar.

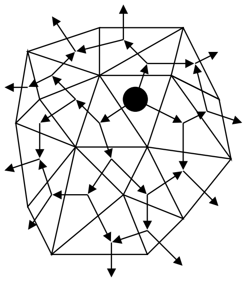

Fig. 5. An example of a walking graph. The black dot represents the light source.

The algorithm visits each node in the graph in topological order. When visiting a face, every point assigned to the face is either discarded, because it is interior to the hull, or pushed out along one of the face's out-edges to another face or horizon edge in the walking graph (which we call "bulldozing"). This process for a single face requires at most two plane-side tests per point, and is based on the location of the center point as well as the light source. When the search is complete, each point associated with any of the visible faces has either been discarded or associated with a horizon edge. One additional test can determine whether a point is interior to the hull or visible to the tent panel formed by this edge and the light source. The bulldozing is done so that the tent panel associated with the horizon edge to which a point is pushed is the associated face of that point. Thus, if the point cannot see that face, it cannot see any face and can therefore be discarded. The precise details of the construction of the walking graph are given in the Appendix, where Theorem 10 states that this construction is correct in the sense that it does extend the convex hull to include the new point p.

The runtime of the algorithm can be separated into the cost of plane-side tests, required to walk points out to the horizon, and the cost of manipulating the simplicial complex. Theorem 11 of the Appendix uses a backwards analysis and Euler's formula to show that that the average number of faces inserted at each stage is at most six, so that the expected number of faces created during the construction of the convex hull is of $O(n)$. This immediately leads to a linear bound on the number of traversals of the simplicial complex. Clearly the simplicial complex is traversed whenever a face is added (during the tent construction) or deleted (during a search for light triangles). The only other time the simplicial complex is traversed occurs when a triangle is detected on the dark side of the horizon. Since each horizon edge gives rise to exactly one new tent face, the number of such traversals is bounded by the total number of faces. Since the cost of each traversal is $O(\lg(n))$ it follows that the total expected cost of maintaining the simplicial complex to describe the surface is $O(n\lg(n))$.

We next consider the cost of "bulldozing" points not yet in the hull. If H_k denotes the convex hull of the first k points, consider the addition of the point $p = p_{k+1}$ to the hull. Since points get "bulldozed" along faces of H_k that they can see, the number of edges a point $q \in \mathcal{P}_{n-k-1} = \{p_{k+2}, \dots, p_n\}$ traverses in the walking graph is bounded by the number of faces $F \in H_k$ visible to both p and q. Theorem 12 of the Appendix shows that the expected total number of such point-face pairs,

$$\{(q, F) \in \mathcal{P}_{n-k-1} \times H_k \,|\, F \text{ visible to both } p \text{ and } q\}$$

(and hence the number of plane side tests) is bounded by $O(n/k)$. It follows that the expected cost of bulldozing the points is $\sum_{k=1}^{n} O(n/k) = O(n \lg(n))$.

4 Convex Hull: Experimental Evaluation

Although our theory shows using a purely persistent dictionary for storing a simplicial complex is asymptotically optimal, we are interested in the actual overhead. In particular we were worried that the constant factors could make the ideas impractical. For this reason we ran several experiments to study the overhead. These experiments involved measurements on the bulldozer three-dimensional hull algorithm, and on a terrain triangulation algorithm, described in the next section. The goal in the experiments is to compare the work needed to maintain the simplicial complex to the other work in the algorithm. This other work mostly consists of the numerical aspects and is dominated by floating-point operations.

In our experiments we used the following five distributions of points in three dimensions:

1. **OnSphere**: Random uniformly distributed points on the unit 2-sphere (i.e., the surface of the unit ball in three dimensions).
2. **EqHeavy**: Random points on the sphere that are weighted to be mostly on the equator. These are generated by producing random points on the sphere, stretching the equator (x and y coordinates) by a factor of 100 so that the distribution is on a disk-like surface, and then projecting the points back down onto a sphere by scaling their length to one.
3. **PolHeavy**: Random points on the sphere that are weighted to be mostly at the poles. These are generated by producing random points on the sphere, stretching the poles (z coordinate) by a factor of 100 so that the distribution is on a stretched ellipsoid surface, and then scaling the points back down onto a sphere as in the EqHeavy distribution.
4. **InBall**: Random uniformly distributed points in the unit ball.
5. **BordHeavy**: Generated by producing points randomly in a unit ball and then mapping each point (x, y, z) to the point $(x, y, x^2 + y^2 + z^2)$. This creates a distribution for which most of the points are near or on the surface of the hull.

We selected these since we wanted data sets both where all the points are in the final result (the expensive case) and where some are inside. We also wanted nonuniform distributions, which are what EqHeavy, PolHeavy, and BordHeavy give us.

To get a machine- and language-independent measurement of the costs we first measured various operation counts. For the manipulation of the simplicial complex (the topological part of the algorithm) we count both the number of dictionary operations and the total number of key-comparisons made by the dictionary code. For

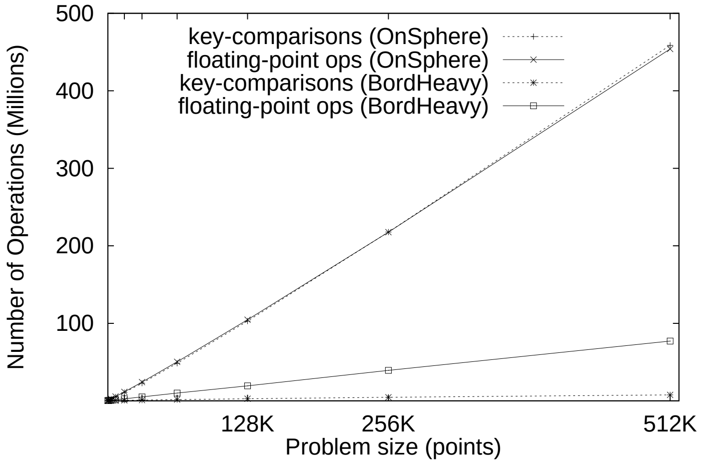

Fig. 6. Operation counts as a function of input size for two of the distributions using the Bulldozer algorithm. The two sets of counts for OnSphere are almost identical.

the numerical (geometric) part of the algorithm we count the number of plane-side tests, from which we can easily determine the number of floating-point operations.

As mentioned in Section 2, the simplicial complex is implemented using red-black trees with vertex identifiers used as keys. For a tree of size n, each insertion, deletion or search will traverse $O(\lg n)$ nodes. At each node, the key being searched (an integer identifier for the vertex) is compared to the key at the node. In addition to the key-comparisons made in the red-black tree, which are based on the first vertex of the simplex being searched, key-comparisons are also required when searching for the second vertex of the simplex in the association-list of the node that is found (see Section 2). Our key-comparison counts include these association-list comparisons. The *key-comparisons* is therefore a measure of the total number of red-black-tree nodes visited, plus the total number of association-list elements visited. Our theory states that the expected total number of dictionary operations is $O(n)$ and since the red-black tree operations visit $O(\lg n)$ nodes, the total number of expected key-comparisons is $O(n \lg n)$.

We measured the number of key-comparisons and floating-point operations for all the distributions and for a range of input sizes up to 512K points. A graph showing the operation counts as a function of size is given in Figure 6 for two of the distributions. A bar graph showing the operation counts for the five distributions on 512K points is given in Figure 7. The graphs show that the number of key-comparisons is approximately the same as the number of floating-point operations for the first three distributions in which all the points are on the sphere. For the other two distributions in which some points are inside the ball, the number of key-comparisons is significantly less than the number of floating-point operations (by a factor of 30 for the InBall distribution and a factor of 10 for the BordHeavy distribution). This is to be expected since the resulting hull is significantly smaller

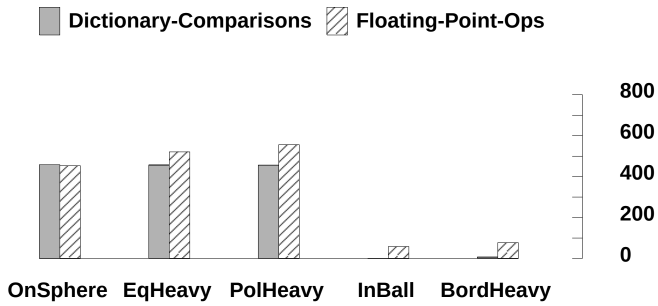

Fig. 7. Operation counts (in millions) for all five data distributions using the Bulldozer algorithm. The input size is 512K points.

than the size of the input, and the simplicial-complex operations are only used on the simplices that are created, while plane-side tests are required on all the input points.

We were also interested in actual running times of the simplicial complex code since one might imagine that traversing a node of a tree is more expensive than a floating-point operation. To be fair on this measure we wanted to compare times to a well tuned existing implementation of three-dimensional Convex Hull. We therefore selected the Minnesota Quickhull code (Barber *et al.*, 1996). Since our code is written in ML and the Minnesota code is written in C, we could not compare the times directly. We also did not want to completely rewrite our code in C, or the Minnesota code in ML. Instead we instrumented our code to dump out traces of all the operations on the simplicial complex. We then wrote C code that simulates the complex operations using balanced trees and linked lists. The idea is to get an sense of how much time relative to the Quickhull code the persistent implementation of the simplicial complex requires. The results are shown in Figure 8. As can be seen, the cost of the simplicial-complex operations is at most half the total cost of the Minnesota code, and this is for a distribution, OnSphere, where the number of operations on the complex is high. Since some of the cost of the Minnesota code is dedicated to manipulating its representation of the simplicial complex (it would be hard to separate this out) it is reasonably safe to conclude that using a persistent dictionary in their code to manipulate the surface would incur less than a 50% overhead, and for many distributions very much less.

5 Terrain Modeling: Experimental Evaluation

One interesting real-world application of the convex hull algorithm is terrain modeling (Garland & Heckbert, 1995). Terrain data is important to many real-world applications, such as flight simulators. However, rendering a terrain at full resolution is impractical for terrains of any significant size. Therefore, applications that

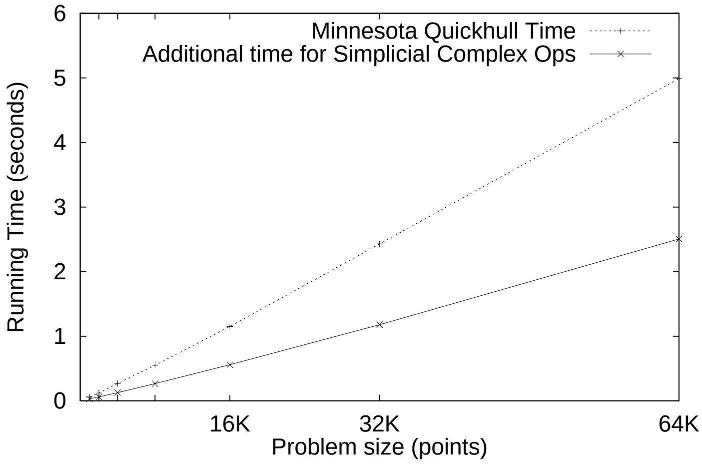

Fig. 8. Running time as a function of the input size for both Minnesota Quickhull and for the C implementation of the dictionary operations. The distribution uses is OnSphere.

rely on terrain data require terrain models that approximate full terrains using substantially fewer polygons.

Given a two-dimensional array of evenly spaced height samples from the full terrain, a terrain modeling procedure computes a triangulation of the terrain that minimizes the error between the actual sample values and the values given by the triangulation. Moreover, the triangulation so determined, when projected onto the plane, is required to have the Delaunay property[6] (Berg *et al.*, 1997), as such triangulations have several desirable properties. However, since it is prohibitively expensive to compute a triangulation that is actually optimal, heuristics are typically employed that perform well in practice.

One such heuristic is the *greedy insertion heuristic.* The greedy insertion heuristic starts by dividing the rectangle into two triangles, and initializes a priority queue with one point from each triangle, the point having the greatest error between the sample value and the value given by the triangle. The heuristic then builds the triangulation incrementally, at each step obtaining the sample point with maximum error from the priority queue and updating the Delaunay triangulation to include that point.[7] The priority queue is then updated to include the points of maximum error for each new triangle. Typically only a few triangles are created in each step, resulting in only moderate rescanning of the terrain samples. This process is then repeated until an acceptable maximum error is achieved.

We implemented this heuristic using our persistent triangulation package. Delau-

[6] The Delaunay property specifies that no point lies within the circumcircle of any triangle of which it is not a vertex, except in certain degenerate circumstances.
[7] An alternative greedy heuristic, designed to avoid narrow triangles, is to add the circumcenter of the triangle containing the point of maximum error, rather than the point of maximum error itself.

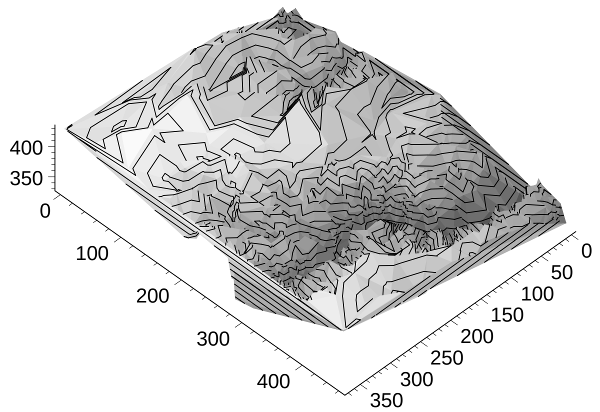

Fig. 9. 1000-point triangulation of Ozark

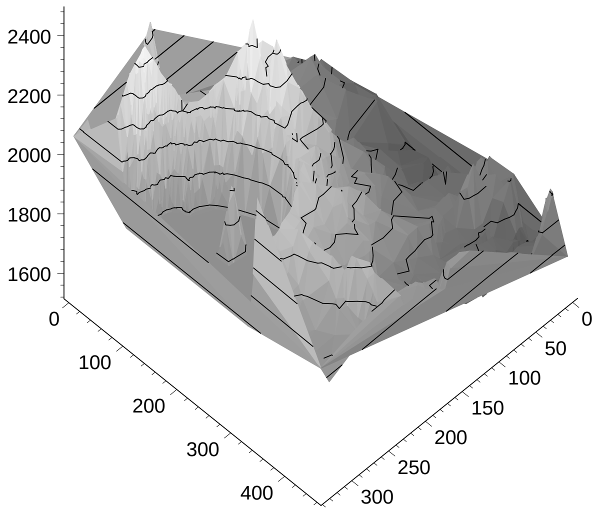

Fig. 10. 1000-point triangulation of Crater Lake

nay triangulations can be computed using a three-dimensional convex hull procedure by projecting the points from the plane onto a paraboloid (the surface specified by the equation $z = x^2 + y^2$) and computing the convex hull of the projected points (Berg *et al.*, 1997), so the implementation was straightforward. To measure its performance, we ran it on two sets of terrain sample data, one from the vicinity of Ozark, Missouri, and the other from the west end of Crater Lake, Oregon. 1000-point triangulations of these two data sets are given in Figures 9 and 10. As in the previous section, we counted key-comparisons and floating-point operations for each run. The results appear in Figure 11 and show that the number of key comparisons is significantly smaller than the number of floating-point operations, especially for the smaller sizes.

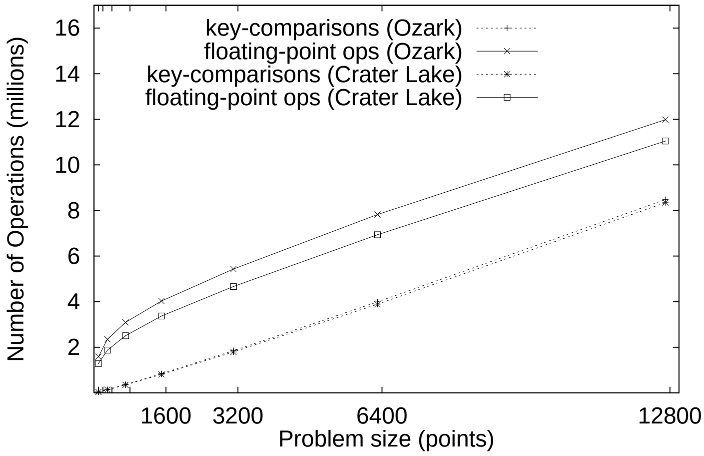

Fig. 11. Operation counts as a function of input size.

Acknowledgements

We thank Chris Okasaki for his red-black tree library, which we used in our implementation of simplices.

References

Alexandroff, Paul. (1961). *Elementary concepts of topology*. New York: Dover Publications, Inc.

Barber, C. B., Dobkin, D. P., & Huhdanpaa, H. T. (1996). The quickhull algorithm for convex hulls. *Acm trans. on mathematical software*, Dec.

Bayer, R. (1972). Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, **1**, 290–306.

Berg, M. De, Kreveld, M. Van, Overmars, M., & Schwartskopf, O. (1997). *Computational geometry : Algorithms and applications*. Springer Verlag.

Blelloch, Guy E. (1996). Programming parallel algorithms. *Communications of the acm*, **39**(3), 85–97.

Burch, Hal, Miller, Gary, & Walkington, Noel. 2000 (March). *Computing the convex hull in a functional language*. Tech. rept. CMU–CS–00–115. Carnegie Mellon University Computer Science Department.

Chazelle, Bernard. (1991). An optimal convex hull algorithm and new results on cuttings. *Pages 29–38 of: Focs*.

Clarkson, K. L., & Shor, P. W. (1989). Applications of random sampling in computational geometry. *Discrete and computational geometry*, **4**, 387–421.

Dietz, Paul F. (1989). Fully persistent arrays. *Pages 67–74 of: Workshop on algorithms and data structures*. Lecture Notes in Computer Science, vol. 382. Springer-Verlag.

Driscoll, James R., Sarnak, Neil, Sleator, Daniel D., & Tarjan, Robert E. (1989). Making data structures persistent. *Journal of computer and system sciences*, **38**(1), 86–124.

Garland, Michael, & Heckbert, Paul. 1995 (Sept.). *Fast polygonal approximation of terrains and height fields*. Tech. rept. CMU-CS-95-181. CS Dept, Carnegie Mellon U.

Giblin, P. J. (1977). *Graphs, surfaces, and homology.* London: Chapman and Hall.

Guibas, L., & Stolfi, J. (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *Acm transactions on graphics*, **4**(2), 74–123.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The definition of Standard ML (revised).* MIT Press.

Motwani, Rajeev, & Raghavan, P. (1995). *Randomized algorithms.* Cambridge University Press.

Myers, Eugene W. 1984 (January). Efficient applicative data structures. *Pages 66–75 of: Proceedings of the 11th acm symposium on principles of programming languages.*

Okasaki, Chris. (1998). *Purely functional data structures.* Cambridge: Cambridge University Press.

O'Neill, Melissa E., & Burton, F. Warren. (1997). A new method for functional arrays. *Journal of functional programming*, **1**(1), 1–14.

Ruppert, Jim. (1995). A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of algorithms*, **18**(3), 548–585.

A Analysis of the Bulldozer Algorithm

We give a detailed description and an analysis of the algorithms outlined in Section 3.

Define the *Convex Closure* of a set S, denoted CC(S), to be the smallest convex set containing S, and the *Convex Hull* of S, denoted CH(S), to be the surface of CC(S). Thus the convex closure of a finite set of points in three dimensions is a three-dimensional polytope and its hull is a surface of two-dimensional polytopes. If no four points are coplanar then the hull will consist of two-dimensional simplices (triangles). If S is a set of points or points and an edge, let $\mathcal{S}(X)$ be the simplex formed by the elements of the set X. For example, if p, q and r are points and e is an edge, then $\mathcal{S}(pqr)$ is a triangle, while $\mathcal{S}(pqe)$ is a tetrahedron.

A.1 Algorithm

The bulldozer algorithm takes a set of points $\mathcal{P} = \{p_1, \ldots, p_n\}$, with the points assigned in a random order. We assume that no four points are coplanar.

In three dimensions a point p is said to *see* a face $F \in H$, where H is a convex figure, if and only if there exists a ray that begins at p and that enters H through the interior of the face F. Note that if F is coplanar with p, p is *not* said to be able to see F. If p can see $F \in H$, then F is said to be *visible* to p with respect to H. An edge e adjacent to faces A and B is said to be a *horizon edge* of p iff exactly one of A and B is visible to p. It is interesting to note that if e is an horizon edge of p in H, then $\mathcal{S}(pe)$ is a face in $CH(H \cup \{p\})$. (This will be proved later.) The definitions for two dimensions are analogous.

For a given convex hull $H = CH(P')$, where $P' \subset P$ and a fixed interior point c, define the *associated face* of a point $p \in \mathcal{P}$ as the face of H penetrated by the ray $\vec{cp}$. If c is not coplanar with any three points of $\mathcal{P}$, each point has a unique associated face. It is clear that if a point p cannot see its associated face, then it is interior to the hull.

The first step in the algorithm is to let $H = CH(\{p_1, p_2, p_3, p_4\})$, where for each $1 \leq i \leq 4$, $p_i \in \mathcal{P}$. Let c be a fixed point interior to H. For every other point p_i $(4 < i \leq n)$, let F_i be the associated face of p_i with respect to H and c. For each face, maintain a list of the points associated with that face. In addition, keep track of the face associated with each point p_i $(4 < i \leq n)$ with respect to H.

The inductive hypotheses are:

1. H is the convex hull of $\{p_1, p_2, \ldots, p_{k-1}\}$,
2. the associated face, if any, for each point p_j $(k \leq j \leq n)$ exterior to H is known, and
3. for each point p_j $(k \leq j \leq n)$, either p_j is interior to H or p_j is in the list of its associated face.

The incremental update associated with adding p_k is composed of three logical operations:

1. Remove the faces that p_k can see.
2. For each horizon edge e of p_k in H, add $\mathcal{S}(ep_k)$ to H.
3. For all points previously assigned to removed faces, determine in which list they belong, if any. Update the points appropriately.

The algorithm performs all three operations simultaneously. The idea is to remove a face once it is determined to be visible. The points that were associated with that face are then "bulldozed" across one of the edges of the face, creating new faces if some of the edges of the current face are horizon edges.

A.1.1 The Walking Graph

In order to facilitate explaining the algorithm, we will first define the notion of a walking graph.

Definition 1
The *walking graph* of a hull H, with a fixed interior point c, and an exterior point p is a directed graph on the faces of H that are visible to p and the horizon edges of p in H. Consider two visible faces A and B that share an edge e. There is an arc from A to B in the walking graph if and only if the face $\mathcal{S}(ec)$ is visible to p with respect to $\mathcal{S}(Bc)$. There is an arc from a visible face A to a horizon edge e if and only if e is an edge of A.

This graph represents all the possible "bulldozing" which may occur. Each point currently associated with a visible face is pushed along the arcs of the walking graph until either the point is determined to be interior to the new hull or the point arrives at a horizon edge. The methodology for this bulldozing is described more precisely below.

Theorem 2
If there is an arc between two adjacent faces, from face A to face B of a hull H, and they are both visible to p with respect to H, then the intersection of the planar extension of A and $\vec{cp}$ is closer to c than the intersection of the planar extension of B and $\vec{cp}$.

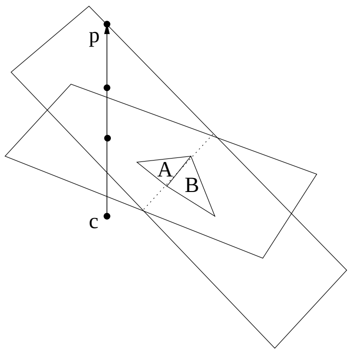

Fig. A 1. Example of Theorem 2

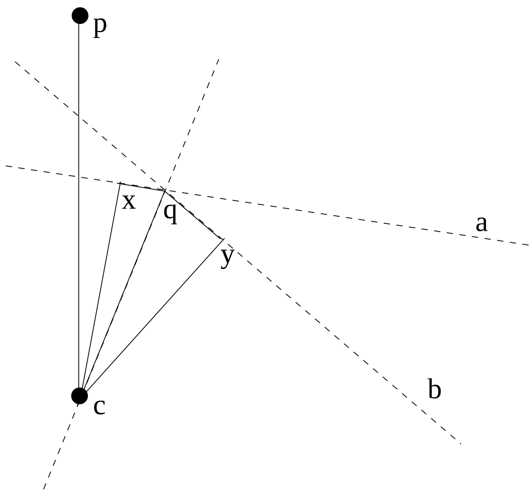

Fig. A 2. Illustration of proof of Theorem 2

Proof
Assume that there is an arc from A to B in the walking graph. Let e be the edge in both A and B, and let q be the midpoint of e. Consider the plane P formed by c, p, and q, and look at the intersections of the planar extensions of A and B with this plane. Since q is on both planar extensions and on P, we know that these intersections must be non-empty, so they must be lines; let a and b denote these intersections. The intersections of A and B with P are segments along a and b respectively, with q as an end point of both segments. Let x and y denote the other end point of the intersections of A and B with P respectively. This arrangement is illustrated in Figure A 2.

Let H' denote the intersection of H and P. Since the intersection of two convex figures is convex, H' must be convex. Moreover, a and b are bounding lines for H', since A and B are bounding planes for H.

Since the edge $\bar{cq}$ is visible to p with respect to $\mathcal{S}(cqy)$, p is on the opposite side of $\vec{cq}$ as y. This means that p and x must be on the same side of $\vec{cq}$. However, since

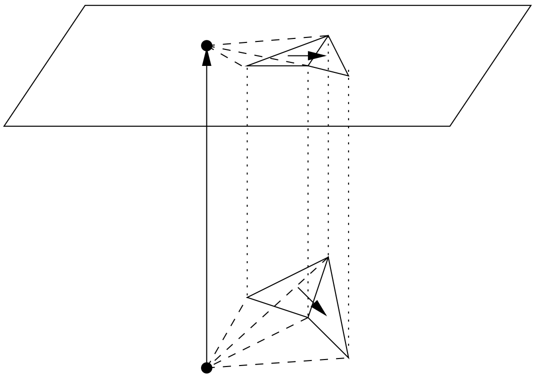

Fig. A 3. Example of face projection definition

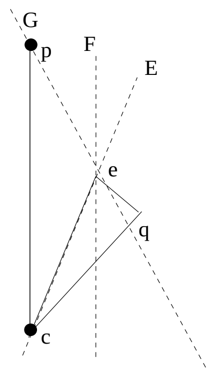

Fig. A 4. Figure for Theorem 3 projected to the plane containing p and c and perpendicular to e.

b is a bounding line for H' and p is not within H', p and x must lie on opposite sides of b. Therefore, the intersection of a and $\vec{cp}$ must also be on the same side of b as c. Thus, the intersection of b with $\vec{cp}$ is closer to c than the intersection of a with $\vec{cp}$. $\square$

It follows from Theorem 2 that the walking graph is acyclic. Moreover, for any two adjacent visible faces A and B, there exists exactly one arc between them.

Theorem 3

Given a convex hull H, an interior point c, and an exterior point p, let P be a plane perpendicular to $\vec{cp}$. There is an arc from a visible face A to an adjacent visible face B if and only if the shared edge e projected onto P is visible to the projection of p with respect to the projection of B.

Proof

Let $B \in H$ be a face visible to p, and let e be an edge of B. Let q be the vertex of B such that $q \notin e$. Define P as above. Let E be the plane that contains e and c. Let F the plane that contains e and is perpendicular to P. Let G be the plane that contains e and p. Note that p and c are in the same half space bounded by F.

Assume there is an arc into B across the edge e. Then, by definition, p is in the opposite half space as q with respect to E. Since p can see B, we know that q and c are on the opposite sides of G. Examine Figure A 4, a projection of the set of points to the plane contain $\bar{pc}$ and perpendicular to e. These constraints imply that p and q are on opposite sides of F, since c and q are on the same side of F. Thus, p can see e of B when projected to P.

Assume that p is in the opposite half space of F as q. This is equivalent to saying e is visible to p with respect to B, when all are projected to P. Since p can see B, we know that q is on the opposite side of G as c. This gives us, as demonstrated in Figure A 4, that q is on the opposite side of E as G, so p can see the face $\mathcal{S}(e, c)$ with respect to $\mathcal{S}(B, c)$. Clearly, the face A that shares e is visible, so there is an arc from A to B across e. $\square$

Note that if a face A which is visible to p with respect to H has an edge e such that e is not visible to p when all are projected as described in Theorem 3, and if there is no arc across e, then Theorem 3 shows that the face B sharing e is not visible, so e is a horizon edge.

Let q denote a point that is exterior to H, and whose associated face in H is visible to p.

Lemma 4
Given a hull H, a fixed interior point c, an exterior point p, and an exterior point q, the associated face of q in H intersects the ray $\vec{pq}$ when both are projected to a plane perpendicular to $\vec{cp}$.

Proof
The ray $\vec{cq}$ projects to the same ray as $\vec{pq}$ in the plane. Since the intersection of $\vec{cq}$ and the associated face must also lie on the ray's projection, this means that this point lies on the projection of $\vec{cq}$. $\square$

Definition 5
Given an exterior point q, draw a ray from p to q and project it to the P described in Theorem 3. The *walking path* of q is defined as follows: start at the projection of the associated face (which is on the projection of $\vec{cp}$ by Lemma 4) and proceed to the next face out along the ray (which is along an arc by Theorem 3). Continue until there is no next face, at which point a horizon edge has been reached. The path generated is the *walking path* of q in the walking graph of H.

The fact that the planar extension of $\mathcal{S}(pcq)$ intersects all faces traversed by the walking path of q in H follows directly from the definition.

Theorem 6
The face created by p and the horizon edge of the walking path of q is the associated face of q in $H' = CH(H \cup \{p\})$.

Proof
If the associated face of q in H' is a face from H, then its associated face in H must be that same face, which contradicts the given. Thus, the associated face of q in H'

must contain the point p. Project all of the faces of H' that contain p to a plane P that is perpendicular to $\vec{cp}$. The rays $\vec{cq}$ and $\vec{pq}$ coincide in P. Any ray that starts at p can intersect only one face that contains p (at any point other than p), so this ray, by definition, intersects the horizon edge of that face. Also, the projection of the intersection of $\vec{cq}$ with H' must lie along the projection of $\vec{pq}$, that only intersects that same face. Thus, $\vec{cq}$ intersects H' in the face created by p and the horizon edge of its walking path. □

Theorem 7
If a point q cannot see a face F in its walking path, then it cannot see its associated face in H'.

Proof
Let G be the walking graph for p in H, and let W be the walking path of q in G. Let A be a face in W that q cannot see, and B be the next face. If no such face exists, then the theorem trivially holds. Let P be the plane defined by p, q and c, and examine the intersection of H and P. Since A and B are in the walking graph, they must have a non-zero intersection with this plane. Since A and B are in the walking graph, q must be on the same side of the intersection of the plane $\mathcal{S}(c, e)$ and P as p is. By Theorem 2, however, the intersection of A is closer to c than B is. Thus, a point may be above A but not above B, but not the reverse (see Figure A 2). Thus, q must be able to see the intersection of B and P, and thus B itself. Thus, if a point cannot see a face F in its walking path, then it cannot see any face past that face in the walking path. A similar argument shows that if a point cannot see the last face in its walking graph then it cannot see its associated face, since the associated face's intersection with $\vec{cp}$ is at p, which must be above the ray's intersection with the last face in the walking path. □

A.1.2 Iterative Algorithm

Adding an interior point is simple, as it does not affect the hull, so the inductive hypotheses are maintained. To add an exterior point p_k and maintain the inductive hypotheses, start with the associated face F in H of p_k. For each point $q \neq p_k$ associated with F, project it to a plane P perpendicular to c, and determine which of the edges is penetrated by $\vec{pq}$. Delete F from the complex. For each adjacent face G, recur into G, passing the set of points whose rays penetrated the shared edge along with the shared edge e.

The recursive call includes a set of points S, a face F and an edge e. For each recursive step, determine if F is visible to p. If it is not, then add the face created from e and p to the complex. Test each of the points from S, and associate them with the new face if they can see it. If they cannot see it, discard them as interior to the hull.

If F is visible to p with respect to H, first throw away any points from the recursive call which cannot see F. For each edge $e \in F$, determine if p can see $\mathcal{S}(c, e)$ with respect to $\mathcal{S}(F, e)$. If any other edges are, and the face across that edge is still a member of the complex, add the set S to the set of points associated with F

and return. Otherwise, for each point q, determine which of the edges is penetrated by the ray $\vec{pq}$, projecting the entire system to the plane P. For each non-empty set, make recursive calls with the appropriate set of points, the edge, and the other face from the simplex which contains that edge.

A.2 Analysis

We will first show that the algorithm is correct, and then look at its asymptotic behavior.

A.2.1 Correctness

The proof of correctness hinges on maintaining of the induction hypotheses:

1. H_k is the convex hull of $\{p_1, p_2, \ldots, p_k\}$.
2. Every point p_i $(k < i \leq n)$ is associated with the appropriate face if it is exterior to H_k, and marked as interior otherwise.

Lemma 8
$CH(S \cup \{p\}) = CH(S) \cup T \setminus V$, where V is the set of faces that p can see in $CH(S)$, and T is the set of faces constructed from horizon edges of $CH(S)$ and p.

Proof
Let $F \in V \subset CH(S)$. Let H_F be the half space bounded by F that contains S. Since $p \notin H_F$, by the definition of visible, $F \notin CH(S \cup \{p\})$.

Suppose $F \notin V$ and $F \in CH(S)$. Since $p \in H_F$ and $S \cup \{p\} \in H_F$, so $F \in CH(S \cup \{p\})$.

Otherwise, $F \in T$ and $F \notin CH(S)$. Let A and B be the faces whose intersection is the horizon edge F contains. As $S \in H_A$ and $S \in H_B$, $S \in H_A \cap H_B \subset H_F$. Thus, $F \in CH(S \cup \{p\})$.

Let $F \in CH(S \cup \{p\})$ but $F \notin T$ and $F \notin CH(S)$. Clearly, $p \in F$, so let e be the edge from F that does not contain p. Let a and b be the vertex of A and B respectively that is not an endpoint of e. $a \in CH(S)$, since otherwise $a \notin CH(S \cup \{p\})$, because $a \neq p$. Similarly, $b \in CH(S)$. If $\bar{ab} \notin CH(S)$, then $\bar{ab}$ intersects the interior of $CH(S)$, and thus the interior of $CH(S \cup \{p\})$, so $\bar{ab} \in CH(S)$. If $\bar{ab}$ is not visible to p in $CH(S)$, then $\bar{pa}$ penetrates the interior of $CH(S)$, and thus the interior of $CH(S \cup \{a, b\})$ Thus, $e = \bar{ab}$ is visible, so one of the faces that share e is visible to p. If only one face is visible, then e is a horizon edge, and $F \in T$, thus both must be visible. Let A and B be the faces that share e, and a and b the points of A and B respectively that are not endpoints of e. Since both A and B are visible, a and b must be on opposite sides of F, which means that $F \notin CH(S)$. □

Lemma 9
For any hull H, interior point c, and exterior point p, all faces visible to p are reachable from the associated face of p in the walking graph of H.

Proof

As all faces for the associated face of p have positive in-degree and the graph is acyclic, it is sufficient to prove that the graph is weakly connected. As the convex hull is a simple closed polygon, the set of adjacent nodes to any given node is a simple cycle. Thus, the set of boundary edges forms a simple cycle, so the set of faces must be connected. $\square$

Theorem 10

If H_k is the convex hull of $\{p_1, p_2, \ldots, p_k\}$ and the associated face for p_k is known, then the bulldozing algorithm produces H_{k+1}, the convex hull of $\{p_1, p_2, \ldots, p_{k+1}\}$.

Proof

Lemma 8 implies that the insert/delete process produces the correct convex hull. Lemma 9 implies that the walking methodology visits the entire walking graph, so the insert/delete process is finished. $\square$

This shows that the convex hull is maintained, and Theorems 6 and 7 demonstrate that the associations are kept correctly. It follows that the induction hypotheses are maintained by the iterative steps, which establishes correctness of the algorithm.

A.2.2 Asymptotics

The initialization can be done in $O(n)$ operations by enumerating all the faces and determining which face $\vec{cp_i}$ intersects for $4 < i \leq n$. When adding p_k to the hull H_k, each light face is visited exactly once. Determining the in-degree and out-degree of that face in the walking graph takes $O(1)$ time. Each point q associated with a deleted face of H_k requires the associated face in H_{k+1} to be computed. q may be associated with every face of H_k that both it and p_k can see in the course of the update step, but it will never be associated with the same one twice. The cost of ensuring that the point q can see its associated face is subsumed by the cost of walking it out. Determining if a point can see a face on its walking path take $O(1)$ time, and determining which edge is penetrated by the projection of $\vec{pq}$ also takes $O(1)$ time. Thus, for each face $F \in H_k$ and point q such that F is visible to both p_k and q, and q's associated face is visible to p, the algorithm takes $O(1)$ time.

For a given convex hull H, a point p *dominates* a point $q \in H$ if $\bar{pq} \cap H = \{q\}$. A point p *strongly dominates* q if $q \notin CH(H \cup \{p\})$. A point p *weakly dominates* q if it dominates q but does not strongly dominate it.

Theorem 11

The expected number of faces inserted by the inclusion of p_{k+1} is at most six.

Proof

Instead of counting the number of visible faces, consider the number of visible points. Let y be the number of points p_{k+1} weakly dominates in H_k. This is equal to the number of faces inserted by p_{k+1}'s inclusion.

In order to determine the expected values of x and y, define G to be a digraph such that:

1. The vertices of the digraph are points of $\mathcal{P}$.

2. All arcs are labeled with a set $S \subset \mathcal{P}$ such that $\|S\| = k$.
3. There is one arc from p to q with label S if and only if $p \in S$ and $\bar{pq} \in CH(S \cup \{p\})$.

The expected number of faces inserted is equal to the the expected number of out-arcs that a point $p \notin S$ has with label S.

Relabel the graph as follows: If there is an arc from p to q with label S, relabel that arc with $S \cup \{p\}$. Thus, there is one arc from p to q if and only if $p, q \in S$, and $\bar{pq} \in CH(S)$. For any q, if $q \notin S$, then its degree is 0. If $q \in S$, then either $q \in CH(S)$ or $q \notin CH(S)$. The average degree of $q \in S$ is 6 by Euclid's formula (since all faces have exactly three edges). If $q \notin CH(S)$, then q has degree 0. Therefore, the maximum in-degree of such a q is 6. Thus, the average in-degree of any $q \in S$ is less than 6.

Since the number of labels is $\binom{n}{k+1}$, and the number of arcs per label is less than $6(k+1)$, the number of arcs is less than $6(k+1)\binom{n}{k+1} = 6(n-k)\binom{n}{k}$. Then, for a random labeling S ($\|S\| = k$) and point p, the expected out-degree of p with label S is less than $\frac{6(n-k)}{n} \leq 6$. It follows that the expected number of faces inserted at each step is less than 6. □

A pair of points p, q is said to *weakly dominate* a point $x \in H$ if p and q both dominate x and at least one of them weakly dominates x. The pair p, q *strongly dominates* x if both p and q strongly dominate x.

Theorem 12
The expected number of pairs of faces F and points q such that (1) q is associated with a face of H_k that p can see, and (2) both q and p can see F, is $O(\frac{n}{k})$.

Proof
Instead of counting the number of faces that an arbitrary point p and q can see, examine the number of points that p and q can see. By Euler's formula, the number of faces is linear in the number of points, so proving that this expectation is $O(\frac{n}{k})$ is sufficient. Instead of computing the expectation, count the total number of sets S and points p, q and x such that $S \subset \mathcal{P}$, $\|S\| = k$, $x \in S$, $p, q \notin S$, both p and q can see x in $CH(S)$, and q's associated face in $CH(S)$ is visible to p. By a simple relabeling, this is the same as the number of sets S and points p, q, x such that $S \subset \mathcal{P}$, $\|S\| = k + 2$, $x, p, q \in S$, both p and q can see x in $CH(S \setminus \{p, q\})$, and q's associated face in $CH(S \setminus \{p, q\})$ is visible to p.

If p and q can both see any face of $CH(S \setminus \{p, q\})$, and $p, q \in CH(S)$, then $\bar{pq} \in CH(S)$, then both p and q lie entirely in the half space bounded by the face both can see, which is convex, so $\bar{pq}$ lies entirely outside $CC(S \setminus \{p, q\})$.

Assume $p \in CH(S)$. This may result in an undercount that is off by at most a factor of 2. There are thus the following cases:

1. $\bar{qx} \notin CH(S)$
   Let $A = CH(S \setminus \{p\}) \setminus CH(S)$. By Euler's formula, $\|A\| = O(d_p + e_p)$, where d_p is the degree of p in $CH(S)$ and e_p is the number of points $CH(S \setminus \{p\})$, but not in $CH(S)$. Since $\bar{qx} \notin CH(S \setminus \{p\})$, $\bar{qx} \in A$. The sum of the degrees

in $CH(S)$ is $O(k)$, again by Euler. For any point $y \notin CH(S)$, there are at most 3 points whose deletion may expose y. Thus, the sum of e_p is $O(k)$. Therefore, the total number of triplets p, q, x of this form is $O(k)$.

2. $\bar{qx} \in CH(S)$
   (a) $\bar{px} \in CH(S)$
   This means that $\mathcal{S}(pqx) \in CH(S)$. There are $O(k)$ such triangles in $CH(S)$, so there are $O(k)$ such triplets.
   (b) $\bar{px} \notin CH(S)$
   Swapping p and q around yields a triple p', q', x such that $p' \in CH(S)$ and $\bar{q'x} \notin CH(S)$. It has already been shown that there are $O(k)$ such triples for any S.

Thus, the total for each case is $O(k)$. This means the total number of triplet/set pairs is $O(\binom{n}{k}\frac{(n-k)^2}{k})$, as $\binom{n}{k+2} = \frac{(n-k-1)(n-k-2)}{(k+1)(k+2)}\binom{n}{k}$. Thus, the expected number of pairs q, x for a random S and a random $p \notin S$ is $O\left(\binom{n}{k}\frac{(n-k)^2}{k}\frac{1}{(n-k)\binom{n}{k}}\right) = O(\frac{n}{k})$.
$\square$