Persistent Triangulations*

Guy Blelloch Hal Burch Karl Crary Robert Harper Gary Miller Noel Walkington

Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Triangulations of a surface are of fundamental importance in computational geometry, engineering simulation, and computer graphics. For example, the convex hull of a set of points may be constructed as a triangulation, and there is a close relationship between Delaunay triangulations and Voronoi diagrams in geometry. Triangulations are ordinarily represented as mutable graph structures for which both edge traversal and adding 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 propose a new representation of triangulations that supports persistence. To demonstrate its use we give a new algorithm for the three-dimensional convex hull that is asymptotically optimal in the expected case, and we give an implementation of a terrain-modelling algorithm based on this representation. To assess its practicality we measure the performance of both applications.

1 Introduction

A *persistent* data structure is one for whose operations—even those that "update" or "modify" it—preserve the data structure across calls. To achieve persistence, update operations must create a "fresh" copy of the data structure so that the original is not disturbed by the operation. Persistent data structures arise naturally in "value-oriented" programming languages such as ML or Haskell. In these languages all data structures are values that are passed as arguments and returned as results, much as numbers are handled in nearly every language.

In contrast, most familiar data structures are *ephemeral*—the operations on the data structure "mutate" it by modifying its representation in memory in such a way that all references to it change simultaneously. Ephemeral data structures arise naturally in "object-oriented" programming languages, including Java or C++. In these languages data structures are thought of as regions of mutable storage that is modified by the operations on the structure.

Persistent data structures offer a number of advantages not shared by their ephemeral counterparts. The term persistence, however, is sometimes use loosely in the literature and there are several types of persistence that can be categorized by the features they support. The key features we are concerned with are:

1. *time travel:* the ability to go back and view any previous version of a data structure,

*This research was supported in part by NSF Grant CCR-9706572. Burch was supported by an NSF Graduate Fellowship.

2. *multiple futures*, the ability to go back and make a sequence of modifications to a previous version of the data structure without affecting the current version (this allows for a version tree),

3. *combining*, the ability to combine persistent data structures, such as taking the union of two persistent sets, and

4. *implicit parallelism*, the ability to view and modify different versions of a data structure in parallel.

Driscoll, *et. al.*. [10] distinguish between *partially persistent* and *fully persistent* data structures. The former support only time travel, whereas the latter also support multiple futures. They describe techniques for building both partially and fully persistent data structures, but their methods do not support combining or implicit parallelism. Driscoll, Sleator and Tarjan [11] introduced the term *confluently persistent* for fully persistent structures that also support combining, and showed a technique for supporting *confluently persistent* lists with catenation. Their technique does not support parallelism. We will use the term *strongly persistent* for a data structure that supports all four features.

The above mentioned techniques are heavily based on the use of side-effects. It is well known, however, that purely functional programs (no side-effects) are inherently persistent, and support at least time-travel, multiple-futures and combining. Furthermore, if the program is *strictly functional* (*i.e.*, does not use lazy evaluation or any other form of memoization) then it will also support implicit parallelism. Tarjan and Kaplan [18] make use of this in a design of a strictly functional catenable list that is strongly persistent. Okasaki [20] developed a simpler algorithm with similar time bounds based on functional programs. Since his method uses lazy evaluation, however, it does not support parallelism.

The subject of this paper is the persistent representation of *closed surfaces* in multi-dimensional space. Closed surfaces are of fundamental importance in computational geometry, and have a number of applications in a wide variety of areas, including geographic information systems, mesh generation for engineering simulations, and surface representations in computer graphics. One application of closed surfaces is the construction of the convex hull of a set of points in three dimensions. The convex hull of a set of points is the surface, or boundary, of the smallest convex polytope containing those points. Another application is to terrain modelling, in which the topography of a geographical region is approximated to within a specified resolution by a closed surface.

A number of representations of closed surfaces have been considered in the literature [16, 4, 8, 7], but all are based on ephemeral data structures such as graphs or mutable dictionaries. Using these ephemeral data structures, several asymptotically optimal algorithms for the construction of the convex hull of a set of points in three dimensions are known [4, 7]. Garland and Heckbert's terrain modelling algorithm [13] is also based on an ephemeral representation of surfaces.

Persistence offers a number of advantages over the more familiar ephemeral representations. In the case of the convex hull algorithm, we may exploit implicit parallelism to provide a simultaneous display of the construction of the hull during its construction, without imposing any synchronization or checkpointing overhead. By exploiting time travel, we may move backwards and forwards among stages of the construction of the hull, allowing the user to explore the dynamics of its creation or to modify the unconsidered points to see the effect on the final hull. In the case of terrain modelling, persistence supports viewing the model at many different resolutions, on demand. Although we have not explored the direct use of persistent surfaces in algorithm design, persistent data-structures are used as components of several algorithms in other areas of computational geometry [21, 15, 17].

Conventional implementations of the 3-dimensional convex hull algorithm rely on mutable data structures such as the doubly-connected edge list described by de Berg, et al. [4]. The hull is represented by a planar graph whose nodes are triangles and whose edges represent the adjacency relation among them. The analysis of these algorithms relies on the assumption that graph edges may be traversed, and new edges added, in

constant time.[1] A naïve translation of these algorithms to the persistent setting would proceed by simply replacing the ephemeral graph structure by a persistent dictionary recording the adjacency relation. Using a simple dictionary representation the cost of the fundamental graph operations increases from $O(1)$ to $O(\lg n)$, which would lead to a sub-optimal $O(n \lg^2 n)$ time bound in the persistent case.

One approach to reducing this cost might be to use a persistent representation of graphs (such as the one given by Erwig [12]) that performs well in the case that each graph has only one "logical future". The obvious disadvantage of this approach is that in the multiple-future case the performance once again degrades to sub-optimal. Moreover, these methods usually rely on so-called "benign" effects that inhibit parallelism. Another approach might be to use Dietz's persistent representation of arrays [9] for which fully persistent array updates and searches require only $O(\lg \lg n)$ time. Used naïvely in a convex hull algorithm Dietz's method would lead to a suboptimal time bound of $O(n \lg n \lg \lg n)$. Moreover, Dietz's representation does not support combining or implicit parallelism.

We therefore consider whether it is possible to achieve the $\Omega(n \lg n)$ lower bound while retaining the advantages of strong persistence. We present a new randomized algorithm, called the *bulldozer algorithm*, for the construction of the convex hull of a set of points in three dimensions that achieves the lower bound in the expected case. The crucial feature of the algorithm is that the expected number of adjacency relationship checks and updates among faces of the hull is at most $O(n)$, while the expected number of floating-point operations to determine the spatial relationships among points is $O(n \lg n)$. Since the algorithm only requires $O(n)$ operations on the surface, we can afford to spend $O(\lg n)$ time per operation without effecting the overall time bound. We can therefore use a simple dictionary based on balanced trees to represent the adjacency relationship of the surface.

Asymptotic complexity is important, but so are empirical performance measurements. To assess the practicality of our representation, we measure the performance of the hull algorithm and the terrain modelling algorithm on representative data sets. There are several types of experiment we could run. Rather than just measuring running times, which are strongly influenced by the machine used and the optimization of the algorithm, we counted the number of basic operations. For the numerical component of the algorithm we counted the number of floating point operations. For the manipulation of the surface we counted the number of key comparisons that are made in our dictionary implementation of the surface, which is based on Red-Black trees. For the convex-hull algorithms we expect both these counts to grow as $O(n \log n)$, but we were interested in comparing the constant factors. Our results are given in Section 4 and show that there are more floating-point operations than key comparisons over a wide variety of point distributions and sizes. Similar results for the terrain-modeling code are given in Section 5.

Since one could argue that a "key comparison" and associated overhead could be much greater than the cost of a floating-point operation, we also ran a timed experiment. To avoid biasing the results, our comparisons are made to the fastest 3D convex-hull algorithm we knew of [2], which was developed at the Minnesota geometry center. We first measured the time for Minnesota Quickhull using its own emphemeral surface representation. We then measured the additional time required to implement the surface operations using a dictionary instead of the ephemeral structure. Our initial experiments show that the dictionary adds a cost of about 50% to the overall cost of the algorithm.

2 Surfaces

Both the convex hull algorithm and the terrain modelling algorithm presented in later sections construct a two dimensional surface enclosing a set of points. 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 modelling, the surface represents

[1] This may not be a reasonable assumption when the number of nodes is extremely large, due to memory hierarchy effects. Nevertheless, it is a standard assumption to ignore such issues.

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 [14], 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. The set of edges "opposite" any vertex, 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 $\triangle ACB$ (and its even permutations) 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.

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* [14, 1], which applies in an arbitrary dimension. Our implementation of the three-dimensional convex hull and of the terrain modelling algorithm are based on an abstract type of simplicial complexes. Not only does this support generalization to higher-dimensional spaces, but it also allows us to experiment with various implementations of them 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 that ensure that the simplices "fit together" to form a coherent "solid" in n-dimensional space.

We assume 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 ordered simplex is *oriented* iff we do not distinguish between two orderings that differ by an even permutation (one that can be expressed as an even number of swaps). A simplex s is a *sub-simplex*, or a *face*, of a simplex t, written $s \leq t$, iff s is a subsequence of t.

[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.

[3] This 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 point
    val new : point -> vertex
    val loc : vertex -> point
  end
```

Figure 1: Signature of Vertices

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 K is also a simplex of K.
3. Every simplex $s \in S$ is a sub-simplex of some n-simplex in S. That is, there are no m-simplices, with $m < n$, other than those that are faces of an 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. To fix ideas we summarize the operations provided by these abstractions.

The signature `VERTEX` specifies that vertices admit a total ordering, which is required for efficiently associating data with vertices. In particular 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 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 dictionary. The operation `dim` yields the dimension of a simplex. Since the order of vertices in a simplex is significant, we distinguish one vertex as the *apex* of the simplex, with the others following in order; this is the first vertex in the enumeration of vertices of the simplex. The `vertices` operation yields the sequence of vertices of a simplex in order, apex first.[4] An n-simplex is created by applying the `simplex` operation to a sequence of $n + 1$ vertices; the first vertex in the sequence is the apex. The `orders` operation yields a sequence of $n + 1$ orderings of the simplex with the same orientation, one ordering for each choice of apex. The `faces` operation yields a sequence of $(n - 1)$-dimensional subsimplices of a given n-simplex. The `flip` operation inverts the orientation of a simplex (flips to its reverse side). The `down` operation passes from an n-simplex to its apex and the "opposing" $(n - 1)$-simplex of that

[4] We make use of an abstract type of sequences, a form of immutable array whose primitive operations are designed to support implicit parallelism [5].

```
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 orders : simplex -> simplex seq
    val faces : simplex -> simplex seq
    val flip : simplex -> simplex
  end
```

Figure 2: Signature of Simplices

apex. (In the case of a triangle, this is the base opposite to a specified vertex of the triangle.) 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 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 dimension of the complex is a fixed property of the abstract type; different instances of the abstraction may have different dimension. The empty complex is the value `empty`; the operation `isempty` tests for it. The sequence of vertices of a complex are returned by the `vertices` operation, in an arbitrary order. The simplices of a given dimension (at most `dim`) 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 $d \leq$ `dim` and a d-simplex s, `grep` returns the sequence (in unspecified order) of simplices of dimension `dim` having s as a face. The `find` operation is a specialization of `grep` for dimension `dim` -1. The operation `add` adds a simplex to a complex, with specified data value; to ensure that purity 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 values of every simplex in the complex, yielding a new complex.

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. Simplices are compared by comparison of sequences so that different orderings determine different simplices. We implement complexes using the `Map` signature taken from the SML/NJ library. An $n > 1$ complex is represented by a mapping from vertices to the set of $(n-1)$-complexes incident on it. (In the case $n = 2$, each vertex has associated with it the edges, together with their vertices, incident on that vertex.) 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 [3]. Each node of the red-black tree then stores as its value an association list that maps the second vertex to the third

```
signature SIMPCOMP =
  sig
    structure Simplex : SIMPLEX
    type 'a complex
    val dim : int
    val empty : '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 option
    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
```

Figure 3: Signature of Simplicial Complexes

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. We note that when a simplex is added, it needs to be added to the tree in all three orders that have the same orientation. 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 adding, the deletion needs to be executed in all three orderings.

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 [4]. Asymptotically optimal algorithms for the problem are also known [8, 7] for the ephemeral case. In this section we will give a randomized optimal algorithm for the persistent case.

We will be concerned with *incremental* methods that extend the convex hull of a set of points to include a new point. Many algorithms, including our own, are based on *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. We view the exterior point as a *light source* illuminating a subset of the faces of the hull. The boundary of the lit faces is a set of edges, which we call the *horizon*. We then construct a pyramidal *tent* whose apex is the exterior point and whose base is the horizon, removing the lit faces. This construction extends the convex hull to include the given point as a new vertex.

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. Our algorithm maintains, for each exterior point, one face which is visible to that point. In particular, the algorithm begins by selecting a point that will always be interior to the hull—we call this the *center point*. Consider the ray

from the center point to each exterior point. Each such ray penetrates a face, to which we associate the point. A face is visible to each of its associated points. The set of faces that are visible to any one point is connected, so knowing one visible face allows one to walk through the visible faces to find them all. Clarkson and Shor's algorithm [8], the Minnesota Quickhull algorithm [2], the algorithm presented by Motwani and Raghavan [19], our basic algorithm, and the Bulldozer algorithm [6] all maintain such information. All the other algorithms, however, either do not care which visible face the point is associated with or keep a complete list of visible faces for each point.

Our algorithm requires a representation of the hull, for which we use a simplicial complex, and some method for associating points with faces, for which we use the data associated with each simplex in the complex.

Each incremental step of our algorithm selects a random face that has points associated with it. A random point associated with this face is designated as the light source. The algorithm then finds all the other faces visible to the light source by searching adjacent triangles on the surface starting with the selected face. The algorithm 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. The selected face has in-degree zero (*i.e.*, it is the root), and the horizon edges have out-degree zero. The faces are visited in a topological ordering of the graph. When visiting a face, every point assigned to the face is either discarded, because it is interior to the hull, or pushed out along an out-going arc (hence the name "bulldozer" algorithm). This requires at most two plane-side tests per point. 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 face formed by this edge and the light source.

This algorithm visits each visible face once. It is possible to show, by backwards analysis and Euler's formula, that the expected number of faces visited is linear in the number of input points when summed across all steps. The cost of the algorithm can be separated into plane-side tests and the cost of the graph traversal and surface manipulation (*i.e.*, finding adjacent faces). Each visit requires at most a constant number of graph and surface operations, each taking $O(\log n)$ time, hence the total expected cost of graph traversal and surface manipulation is $O(n \lg n)$. It is also possible to show that the expected number of plane-side tests is $O(n \lg n)$ [6]. Thus the total expected cost is $O(n \lg n)$.

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 3d 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 5 distributions of points in 3d:

1. **OnSphere**: Random uniformly distributed points on the unit 2-sphere (*i.e.*, the surface of the unit ball in 3d).

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.

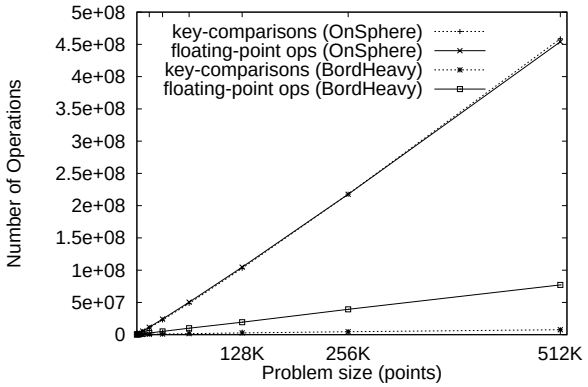

Figure 4: Operation counts as a function of input size for two of the distributions using the Bulldozer algorithm. The two sets of counts for OnSphere are almost identical.

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 $(x, y, x^2 + y^2 + z^2)$. It can be shown that using this distribution for n points, the size of the convex hull is expected to be $\Theta(n^{2/3})$.

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 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

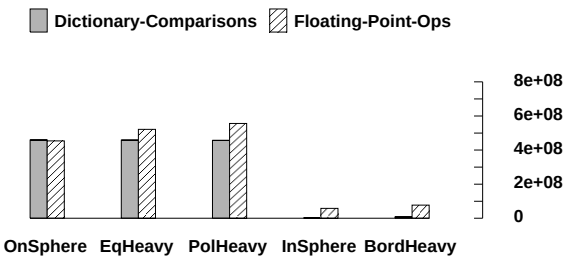

Figure 5: Operation counts for all five data distributions using the Bulldozer algorithm. The input size is 512K points.

is given in Figure 4 for two of the distributions. A bar graph showing the operation counts for the five distributions on 512K points is given in Figure 5. 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 very much 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 than the size of the input, and the simplicial-complex operations are only used on the simplexes that are actually created, while plane-side tests are required on all the input points.

We were also interested in actual running time 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 3D Convex Hull. We therefore selected the Minnesota Quickhull code [2]. 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 6. 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 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 actually to separate this out) it is reasonably safe to conclude that using a persistent dictionary in their code for 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 to terrain modeling [13]. 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 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

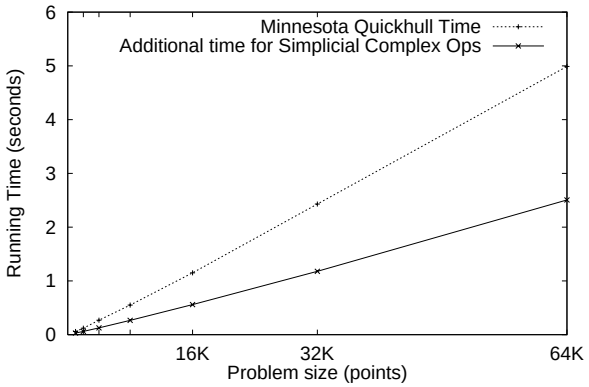

Figure 6: Running time as a function of the input size for both Minnesota Quickhull and for the C implementation of the dictionary operations. The distribution uses is OnSphere.

and the values given by the triangulation. Moreover, the triangulation so determined, when projected onto the plane, is required to have the Delaunay property[5] [4], 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 plane 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.[6] 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. Delaunay 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 [4], 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. A 1000-point triangulation of each of these data sets is given in Figures 7 and 8. As in the previous section, we counted key-comparisons and floating-point operations for each run. The results appear in Figure 9 and show that the number of key comparisons is significantly smaller than the number of floating-point operations, especially for the smaller sizes.

[5]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.

[6]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.

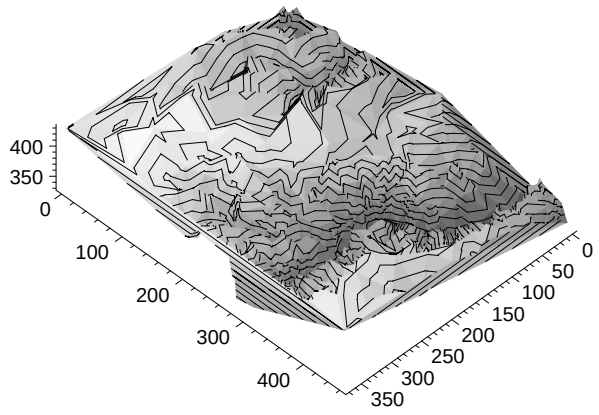

Figure 7: 1000-point triangulation of Ozark

6 Conclusion

Purely functional, persistent data structures offer a number of programming advantages over their more familiar, ephemeral counterparts. Many algorithms in computational geometry are based on low-level graph structures. The analysis of these algorithms is based on a unit-cost assumption for the fundamental operations on a graph. A naïve translation of these algorithms using a persistent tree structure to represent the graph would introduce an $O(\lg n)$ factor into the asymptotic complexity, resulting in asymptotically sub-optimal performance in the persistent case.

This paper addresses the question of whether this logarithmic penalty is avoidable in specific cases. We consider the fundamental problem of constructing the convex hull of a set of points in three dimensions. We give a high-level description of the hull as a simplicial complex, and provide a persistent implementation of it. We also present a new algorithm, called the bulldozer algorithm, that achieves the asymptotically optimal $O(n \lg n)$ time bound (in a randomized sense) that works with this abstract representation of the hull. To assess the practicality of the algorithm, we implemented this algorithm in Standard ML and measured its performance on a variety of artificial data sets. We also used this algorithm to build a terrain modelling application derived from work of Garland and Heckbert [13]. Our results confirm that the persistent representation of the hull, together with our new algorithm for constructing it, are both theoretically and practically efficient.

Acknowledgements

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

References

[1] Paul Alexandroff. *Elementary Concepts of Topology*. Dover Publications, Inc, New York, 1961.

[2] C. B. Barber, D. P. Dobkin, and H. T. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, December 1996.

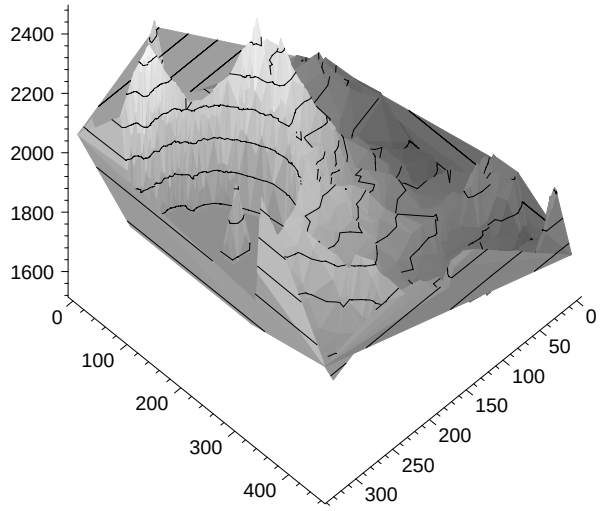

Figure 8: 1000-point triangulation of Crater Lake

[3] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[5] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, March 1996.

[6] Hal Burch and Gary Miller. Computing the convex hull in a functional language. In Preparation, September 1999.

[7] Bernard Chazelle. An optimal convex hull algorithm and new results on cuttings. In *FOCS*, pages 29–38, 1991.

[8] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry. *Discrete and Computational Geometry*, 4:387–421, 1989.

[9] Paul F. Dietz. Fully persistent arrays. In *Workshop on Algorithms and Data Structures*, volume 382 of *Lecture Notes in Computer Science*, pages 67–74. Springer-Verlag, August 1989.

[10] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[11] James R. Driscoll, Daniel D. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *Journal of the ACM*, 41(5):943–959, 1994.

[12] Martin Erwig. Functional programming with graphs. In *Proc. ACM Sigplan International Conference on Functional Programming*, pages 52–55, June 1997.

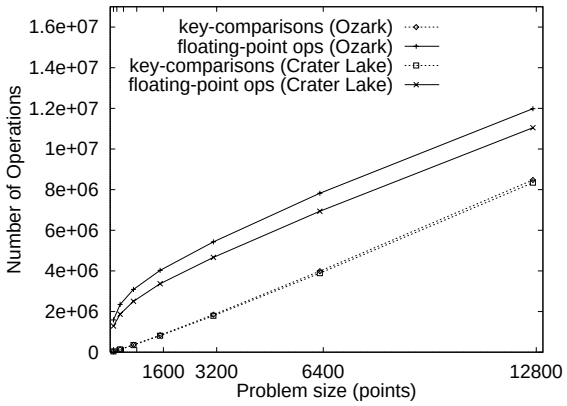

Figure 9: Operation counts as a function of input size.

[13] Michael Garland and Paul Heckbert. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, CS Dept, Carnegie Mellon U., September 1995.

[14] P. J. Giblin. *Graphs, Surfaces, and Homology*. Chapman and Hall, London, 1977.

[15] C. M. Gold and P. R. Remmele. Voronoi methods in GIS. In *Algorithmic foundations of geographic information systems*, pages 21–35. Springer-Verlag, 1997.

[16] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[17] N. Gupta and S. Sen. An improved output-size sensitive parallel algorithm for hidden-surface removal for terrains. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 215–219, March 1998.

[18] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *ACM Symposium on Theory of Computing*, pages 93–102, May 1995.

[19] Rajeev Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] Chris Okasaki. Amortization, lazy evaluation, and persistence: Lists with catenation via lazy linking. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 646–654, October 1995.

[21] Neil Sarnak and Robert Endre Tarjan. Planar point location using persistent search trees. *CACM*, 29(7):669–679, 1986.