Progressive TINs: Algorithms and Applications*

Anil Maheshwari Pat Morin Jörg-Rüdiger Sack

School of Computer Science
Carleton University
Ottawa, ON
Canada, K1S 5B6
{maheshwa,morin,sack}@scs.carleton.ca

Abstract

Transmission of geographic data over the Internet, rendering at different resolutions/levels of detail, or processing at unnecessarily fine detail pose interesting challenges and opportunities. In this paper we explore the applicability to GIS of the notion of progressive meshes, introduced by Hoppe [13] to the field of computer graphics. In particular, we describe progressive TINs as an alternative to hierarchical TINs, design algorithms for solving GIS tasks such as selective refinement, point location, visibility or line of sight queries, isoline/contour line extraction and provide empirical results which show that our algorithms are of considerable practical relevance. Moreover, the selective refinement data structure and refinement algorithm solves a question posed by Hoppe.

1 Introduction

Triangle meshes are used in a number of fields for modelling the surfaces of real world objects. In the field of geographic information systems (GIS), triangle meshes are used as to model the surface of the earth. In computer graphics, triangle meshes are used to model the surfaces of objects in virtual worlds. In CAD systems triangle meshes are used to model the surfaces of machine parts.

Often meshes are generated from data obtained using electronic imaging equipment such as laser range finders, remote sensing equipment, and satelite imaging units. As this equipment increases in fidelity, the size of the resulting meshes increases as well. One method of

*This work was supported by the Natural Sciences and Engineering Research Council of Canada.

dealing with this dramatic increase in data size has been *mesh simplification*. Mesh simplification involves reducing the number of vertices in a mesh in an intelligent manner, so that accuracy of the model is maintained as much as possible.

Mesh simplification is a time consuming process, and in many cases, manual intervention is necessary in order to preserve features of the mesh which may be important but can not be recognized by a computer. Additionally, mesh simplification involves simplifying a mesh to a given level of detail (number of vertices or error tolerance). For some applications, this level of detail may be inappropriate. If the level of detail is insufficient, then the application may produce poor or even incorrect results. If the level of detail is too high, then the application may become computationally inefficient or even infeasible.

It is this latter problem of mesh simplification that is addressed by the field of multiresolution surface modelling. Essentially, multiresolution surface modelling involves performing mesh simplification using a series of localized simplification operations to produce a sequence of progressively less detailed meshes $M_n, M_{n-1}, \ldots, M_0$. While the simplification takes place, the simplification operations are recorded. In this way, the operations can be inverted, and the original mesh can be recreated. In fact, given the mesh M_i, any of the of the meshes $M_n, M_{n-1}, \ldots, M_0$ can be recreated. Thus, the level of detail can be varied according to the requirements of the application. For illustration, Figure 1 shows an example of a multiresolution terrain model at 3 different levels of resolution.

Hoppe [13] defines some desirable features of a multiresolution surface model.

- *Level-of-Detail (LOD) Approximation* allows for viewing and manipulating the mesh at varying levels of detail (number of vertices).

- *Progressive transmission* allows for the transmission of a mesh across a network, so that the receiver can immediately display an approximation of the mesh, and refine this display as more data is received.

- *Selective refinement* is the ability to view and manipulate part of the mesh at a very high level of detail while the remainder of the mesh remains at a low level of detail.

In this paper we are interested in multiresolution surface models and their applications, with special emphasis on geographic information systems. The main technical contribution of this paper is a solution to the problem of efficient selective refinement of progressive meshes

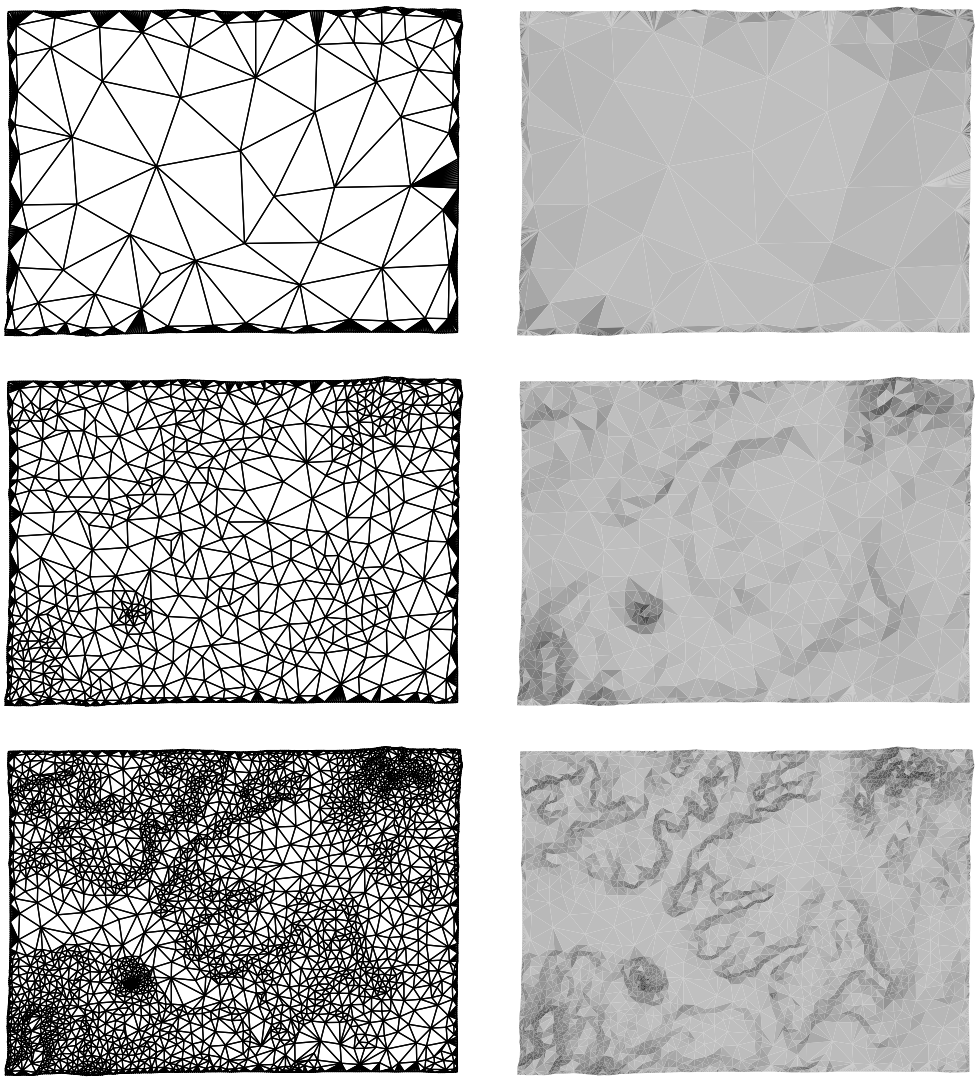

Figure 1: A triangle terrain at 3 different levels of resolution (left) and it's corresponding shaded image (right).

posed by Hoppe in [13].[1] Using the ideas behind this solution, a number of applications are presented, including point location, elevation queries, visibility queries, isoline/contour extraction (conversion to contour lines). Empirical results are presented for some of these algorithms which show that they are of considerable practical relevance. Furthermore, we sketch representations a multiresolution surface model in external memory. An extended abstract containing the main results of this work was presented at the ACM-GIS'97 workshop [17].

The remainder of the paper is organized as follows: Section 2 discusses other approaches to multiresolution terrain modelling and reviews the progressive mesh representation upon which our work is based. Section 3 describes our solution to the problem of efficient selective refinement in progressive meshes. Section 4 discusses some applications of this solution. Finally, Section 5 summarizes and outlines directions for future work.

2 Survey of Existing Work

The problem of level of detail approximation in triangulated meshes has received a significant amount of attention in recent years. In this section, we give a brief critical survey of this work. We broadly classify these schemes into three categories: (1) the *tree-based* schemes of De Floriani *et. al.* [3, 5, 4], (2) the *DAG-based* scheme of Dobrindt and de Berg [2] and Puppo [22], and (3) the *Progressive Mesh* scheme of Hoppe [13].

2.1 Tree-Based Schemes

The approach to multiresolution modelling taken by De Floriani *et al.* [4, 5] is to generate a mesh consisting of nested triangles, and to arrange these triangles into a tree shaped hierarchy (see Figure 2). At the top level of the hierarchy, the mesh consists of a fixed number of very large triangles. To refine (all or part of) the mesh, the hierarchy is searched in a top-down manner, and large (parent) triangles are replaced by groups of small (child) triangles. This process is repeated until a sufficient level of detail has been achieved.

Since the triangles of the hierarchy are arranged as a tree with small fixed degree, this

[1] A selective refinement scheme similar to the one described in Section 3 has been independently and concurrently proposed by Hoppe [14].

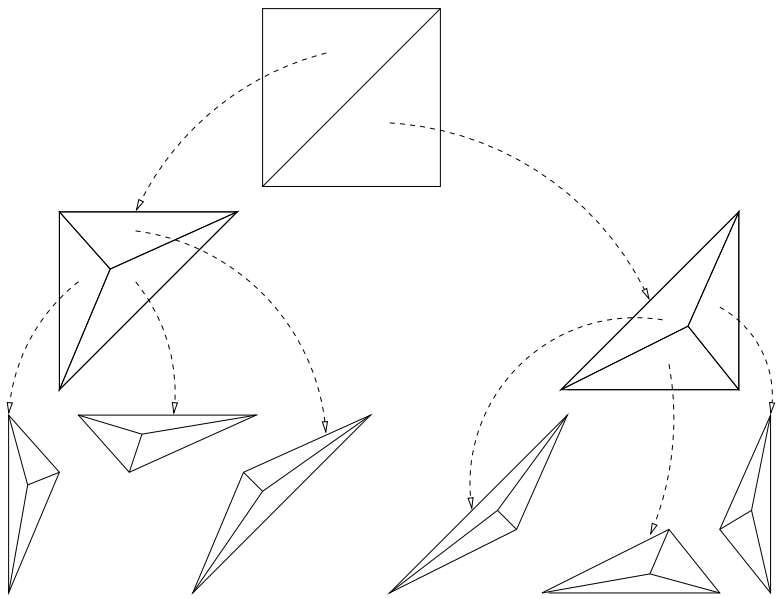

Figure 2: An example of a tree-based hierarchy.

scheme tends to be very efficient in practice. However, it does have some disadvantages. As is already evident in the three level hierarchy of Figure 2, triangles at lower levels in the hierarchy tend to become elongated, due to the fact that the long edges of the low-resolution mesh are also present in the high-resolution mesh. Such long triangles can lead to numerical instabilities in computations, and aliasing effects in graphics applications. Methods which avoid these elongated triangles by splitting long edges are described in [4], but these can lead to vertical faces (discontinuities) in the TIN.

2.2 DAG-Based Schemes

Dobrindt and De Berg [2], and De Floriani [3] address the problem of the thin triangles which occur in tree-based schemes by using a very different approach. By repeatedly deleting an independent set of TIN vertices, and retriangulating the resulting holes, a hierarchy of triangulations which has a depth of $O(\log n)$ and a total size of $O(n)$ is achieved (see Figure 3). In this case, refinement is achieved by replacing a *group* of triangles from a higher level in the hierarchy, with a group of triangles from a lower level in the hierarchy. Using this method, Dobrindt and de Berg show that the Delaunay triangulation of the data can be maintained at all levels of the hierarchy.

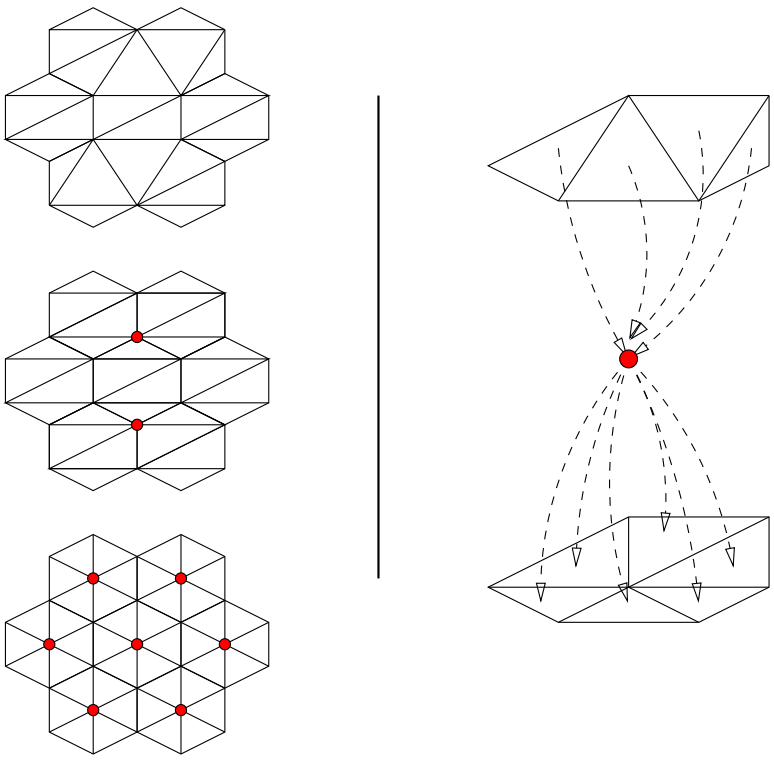

Figure 3: An example of a DAG-based hierarchy.

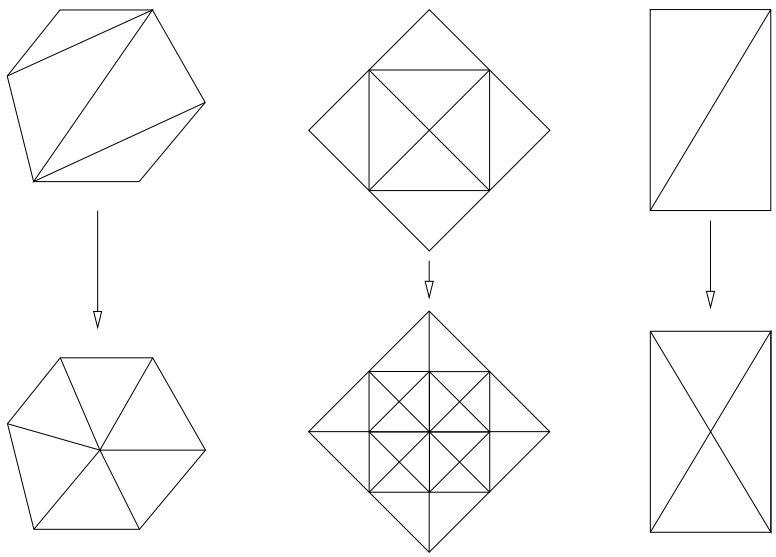

Figure 4: Some possible transformations in a MultiTriangulation.

While this approach solves the problem of thin triangles, it tends not to be as efficient as tree based schemes. The storage requirements of this scheme are much larger since, on average, every vertex which is deleted generates an additional 10 pointers in the hierarchy (see Figure 3). Furthermore, although the hierarchy has depth $O(\log n)$, there are constant factors hidden in the big-O notation (Dobrindt and De Berg report hierarchies of with depths varying between $1.7 \log_2 n$ and $2.4 \log_2 n$ depending on the method used to select vertices for deletion).

Puppo [22] takes the DAG-based approach one step further by describing a general DAG-based framework called the *MultiTriangulation*. In this scheme, a group of triangles can be replaced by another group of triangles as long as the boundaries of the two groups match (see Figure 4). In a companion paper [6], it is shown that most existing schemes, including those considered herein, can be viewed as special cases of the MultiTriangulation (although sometimes the MultiTriangulation is less efficient than the original scheme). Although the MultiTriangulation is quite powerful and expressive, current implementations still seem to exhibit the same high overheads associated with the DAG-base scheme of Dobrindt and de Berg [7].

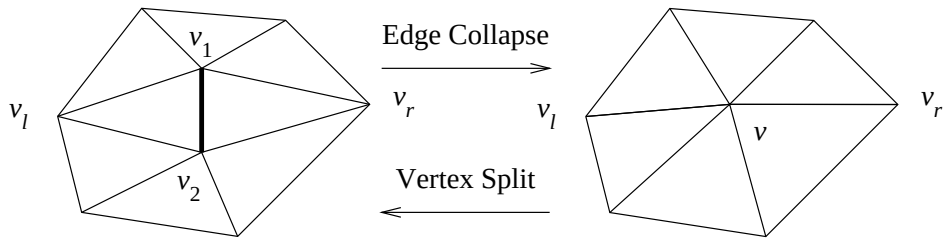

Figure 5: The edge collapse transformation and its inverse, the vertex split

2.3 The Progressive Mesh Representation

Hoppe's progressive mesh (PM) representation [13] is based on the *edge collapse* transformation and its inverse the *vertex split*. An edge collapse involves collapsing an edge by identifying its two incident vertices, v_1 and v_2, into a single aggregate vertex v. The two adjacent triangles (v_1, v_2, v_l), and (v_1, v_2, v_r) vanish in the process. An edge collapse and its inverse vertex split are illustrated in Figure 5.

In the progressive mesh representation, a mesh M, is represented as a pair, $(M_0, vsplits)$, where M_0 is a coarse mesh and *vsplits* is a list of vertex splits which will reproduce the original mesh when applied in order. A vertex split transformation $vsplit(v, v_1, v_2, v_l, v_r, A)$ splits the aggregate vertex v into two vertices v_1 and v_2 and adds the edges (v_1, v_l), (v_1, v_r), (v_2, v_l), and (v_2, v_r). A stores attribute information for the neighbourhood of the transformation, including, but not limited to, the positions of the two new vertices.

We call v the *parent* of v_1 and v_2, and we call v_1 and v_2 the *children* of v. We say that a vertex u is an *ancestor* of a vertex v if $u = v$ or if u is an ancestor of the parent of v. In this case we call v a *descendent* of u. We denote by M_i the mesh obtained by performing the first i elements of *vsplits* on the coarse grained mesh M_0. The original mesh, M, can be obtained by applying all the elements of *vsplits* in order, i.e., $M_{|vsplits|} = M$.

The progressive mesh construction algorithm takes a mesh, M, performs a series of edge collapse operations to get the mesh M_0, and sets *vsplits* to the list of vertex split operations that invert the edge collapses performed. The order in which the edge collapses are performed is determined by an application specific fitness function (e.g., minimizing the geometric error). The edges of the mesh are placed in a priority queue based on their fitness, and are removed one at a time as they are collapsed. Edges in the neighbourhood of a collapse may have their priorities updated. A fitness function which considers the geometry,

discrete attributes, and scalar attributes of the mesh is described by Hoppe [13]. Sufficient requirements for an edge collapse to be valid are described in by Hoppe *et al.* [15].

3 Selective Refinement of Progressive Meshes

In Section 1, we saw that it may be desirable to work with a mesh at a lower level of resolution than the original data. More generally, we may wish to work with parts of the mesh at a high level of resolution, and the rest of the mesh at a lower level of resolution. For example, if the mesh is a terrain representing a city, the user may only be interested in a particular part of the city. Therefore, it is not necessary (and inefficient) to display the uninteresting parts of the city at a high level of resolution. The *selective refinement* problem is that of extracting a mesh which is at a high level of resolution in a query region q while leaving the mesh outside of q at a low level of resolution.

In this section we are concerned with the selective refinement problem on progressive meshes. In particular, we devise an algorithm to extract all the triangles of of M which intersect a query region q while leaving the mesh outside the query region relatively untouched.

Hoppe [13] suggests a brute force method of selective refinement is described which examines every vertex split record to determine whether or not it should be split. There are two disadvantages of this algorithm. The first is that the running time is $O(|vsplits|)$ regardless of the number of splits actually performed. The second is that it is difficult to perform an exact selective refinement, that is, to extract all the triangles of M which intersect a specified query region, q. The reason for this is that there may be vertex splits which lie outside of q, but which affect the triangles within q.

The following sections describe a new efficient method of performing selective refinement in progressive meshes. The technique uses information about dependencies in the *vsplits* list to achieve better efficiency. The method is simple, robust, and easy to implement.

The selective refinement scheme can be summarized as follows: In a preprocessing phase, we associate with each vertex split record a *region of influence* outside of which splitting the vertex has no effect. By using the parent/child relationships between vertex split records, these regions of influence can be organized as a forest of rooted binary trees. When the records are organized in this manner it is possible to efficiently identify all the vertex split records whose regions of influence intersect a given query region q. Once the relevant vertex

split records have been identified, they can be sorted and applied in order.

3.1 Computing the Region of Influence

We treat the *vsplits* list as a dependency graph in which a vertex v_1 depends on a vertex v if v is split and one of the resulting vertices is v_1. Observe that the dependency graph is a forest of rooted binary trees whose roots are the vertices of the coarse mesh, M_0, and whose leaves are the vertices of the original mesh, M (see Figure 6 for an illustration).

With each node v in the dependency graph we associate an axis aligned 3-Dimensional bounding box, denoted $roi(v)$ which represents the region outside of which this vertex split has no influence. We define $roi(v)$ recursively as follows: if v is a leaf then $roi(v)$ is the smallest box which encloses all the neighbours of v in M, otherwise $roi(v)$ is the smallest box which encloses $roi(v_1)$ and $roi(v_2)$ where v_1 and v_2 are the two children of v. These boxes, i.e., the *roi*'s can be computed in a bottom-up fashion during the progressive TIN construction procedure. Alternatively, these boxes can be constructed using a post order traversal of the forests.

The following lemma shows us that if we apply to M_0 every vertex split v such that $roi(v)$ intersects q, then all triangles of M which intersect q will be reconstructed.

Lemma 1. *Let u be a vertex incident to a triangle in M which intersects the query region q. Then, for all ancestors v of u, $roi(v)$ intersects q.*

Proof. The proof is by induction on the level of v in the dependency graph. The base case is when $v = u$, and is true since $roi(u)$ contains all triangles incident to u. Next, w.l.o.g., suppose v is an internal vertex which was formed by collapsing the edge (v_1, v_2). Note that one of v_1 or v_2 is an ancestor of u, and by the inductive hypothesis one of $roi(v_1)$ or $roi(v_2)$ intersects q. This implies that $roi(v)$ intersects q as well, since $roi(v)$ contains $roi(v_1)$ and $roi(v_2)$. □

For completeness, the following pseudocode shows how to construct the *roi* information. The notation $neighb(v)$ is used to denote the neighbourhood of the vertex v in M, i.e., $neighb(v) = \{v\} \cup \{u : (u, v) \in E(M)\}$. The notation $bb(X)$ is used to denote the bounding box of the set X, i.e., the smallest axis-aligned box which contains X.

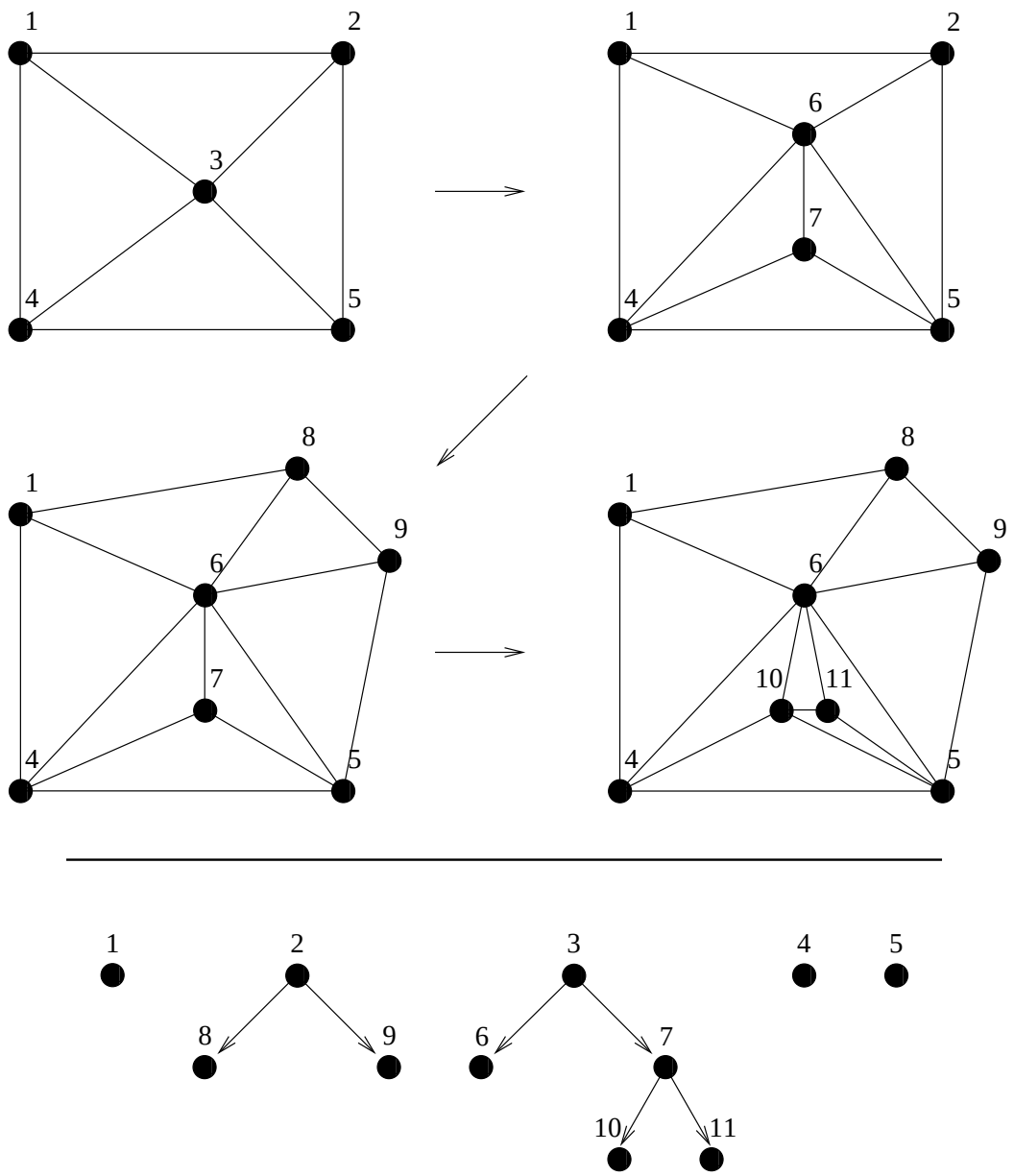

Figure 6: A sequence of vertex splits and its associated dependency graph.

```
BUILD-ROI()
1  forall v ∈ V(M) do
2      roi(v) ← bb(neighb(v))
3  forall v ∈ V(M_0) do
4      BUILD-ROI-2(v)
5  return
```

```
BUILD-ROI-2(v)
1  if v is not a leaf then
2      BUILD-ROI-2(vsplits[v].v_1)
3      BUILD-ROI-2(vsplits[v].v_2)
4      roi(v) ← bb(roi(vsplits[v].v_1), roi(vsplits[v].v_2))
5  return
```

Lemma 2. *The running time of procedure* BUILD-ROI() *is* $O(n)$.

Proof. Lines 1 and 2 of BUILD-ROI() take $O(n)$ time, since each vertex of M is examined once, and each edge is examined twice (once for each end point). Lines 3 and 4, including the call to BUILD-ROI-2, take $O(n)$ time, since they perform a post-order traversal of the dependency graph, which is a forest of size $O(n)$. □

3.2 Retrieving the Vertex Splits

Once the *vsplits* list is augmented with the *roi* information, selective refinement can be done in the following manner: For each vertex of M_0 we search the tree rooted at M_0 and retrieve all vertex splits, v, for which $roi(v)$ overlaps the query region, q. It is not necessary to search the children of v if $roi(v)$ does not intersect q, since the *roi* of the children of v are contained in $roi(v)$.

```
GET-SPLITS(q)
splits ← λ
forall v ∈ V(M_0) do
    splits ← splits ∪ GET-SPLITS-2(v, q)
return splits
```

```
GET-SPLITS-2(v, q)
1  if v = nil then
2      return λ
3  if q intersects roi(v) then
4      return {v} ∪ GET-SPLITS-2(vsplits[v].v_1, q)
           ∪ GET-SPLITS-2(vsplits[v].v_2, q)
5  return λ
```

Lemma 3. *The procedure* GET-SPLITS(q) *returns exactly the vertex splits v such that $roi(v)$ intersects q.*

Proof. Clearly, a vertex split, v, such that $roi(v)$ does not intersect q is never reported. Furthermore, any vertex split which is examined in Line 3 and has a region of influence which intersects v is reported in Line 4 of GET-SPLITS-2. Thus we need only show that all such vertices are examined in Line 3. Note that the only reason a vertex split, v, is not examined in Line 3 is if one of its ancestors, v' had a region of influence that did not intersect q. But in this case, $roi(v)$ can not intersect q, since $roi(v)$ is contained in $roi(v')$, which does not intersect q. Therefore, every vertex split which is not examined in Line 3 of GET-SPLITS-2 does not have a region of influence which intersects q. □

Lemma 4. *The procedure* GET-SPLITS(q) *has running time $O(|M_0| + k)$ where k is the number of vertices, v, such that $roi(v)$ intersects q.*

Proof. We count the number of calls to the GET-SPLITS-2 procedure, since this clearly bounds the total running time. The procedure is called at most $|M_0|$ times in the GET-SPLITS procedure. The number of recursive calls is at most $3k$, since each vertex which is reported results in calling the procedure for at most two unreported vertices. □

3.3 Sorting and Applying the Vertex Splits

Once the *vsplits* records are retrieved they need to be sorted and applied in order. The sorting could be done by a standard sorting algorithm, but this would take $O(k \log k)$ time. A more efficient method can be obtained by observing that the order in which the vertex splits can be applied is not necessarily unique. In fact, the only dependencies between vertex

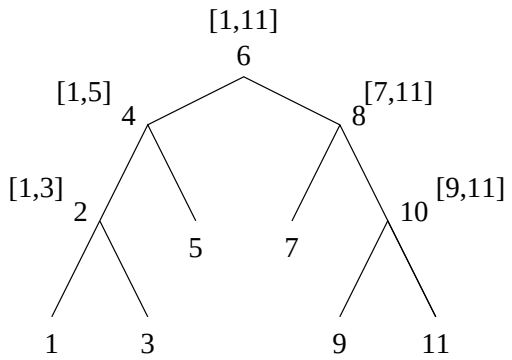

Figure 7: An example of the *rank* and *range* numbering.

splits come from the parent-child relationship on which our data structure is based, and the *left-neighbour*, and *right-neighbour* relationships. I.e., using the notation in Section 2.3 the vertices v_1 and v_2 are dependent on the vertex v, and v is dependent on the vertices v_l and v_r. Both these relationships can be represented as a directed acyclic graph of size $O(k)$ and thus a feasible order for the edge collapses can be obtained in $O(k)$ time by performing a topological sort (c.f. [1]).

When splitting a vertex v, it is possible that v_l (or v_r) may not exist, since it was not split. Then the dependency is with the nearest ancestor of v_l which appears in the retrieved *vsplits* records. This requires that we locate the nearest living ancestor of a vertex. In order to do this we use an in-order numbering of the vertices in the dependency graph, and associate with each node v, a number, $rank(v)$ which is the in-order traversal number of v, and an interval, $range(v)$ which contains the ranks of the descendents of v as well as v itself. This information is computed in a precomputation phase. With this information, it is possible to find the nearest living ancestor of v_l by searching the neighbourhood of v for a vertex, v_l', such that $rank(v_l) \in range(v_l')$. See Figure 7 for an example of such a numbering.

The following functions produce the *rank* and *range* information used for finding ancestors. After these functions are run, one can determine if a vertex u is an ancestor of v by checking if $rank(v) \in range(u)$.

```
BUILD-RANGES()
next ← 0
forall v ∈ V(M_0) do
    next ←BUILD-RANGES-2(v, next)
return
```

```
BUILD-RANGES-2(v, next)
marker ← next
if vsplits[v].v_1 ≠ nil then
    next ← BUILD-RANGES-2(vsplits[v].v_1, next)
    rank(v) ← next
    next ← BUILD-RANGES-2(vsplits[v].v_1, next + 1)
else
    rank(v) ← next
    next ← next + 1
range(v) ← [marker, next − 1]
return next
```

Lemma 5. *The running time of procedure* BUILD-RANGES() *is* $O(n)$.

Proof. The functions implement a post-order traversal of the dependency graph, which has size $O(n)$. □

The SELECTIVE-REFINE function is the main entry point for the selective refinement algorithm. It first finds all the vertex splits whose regions of influence intersect the query region, and then applies these splits in an order which respects all dependencies.

```
SELECTIVE-REFINE(q)
splits ← GET-SPLITS(q)
forall v ∈ splits do
    if v ∈ M_0 then
        APPLY-SPLITS(v, splits)
return
```

```
APPLY-SPLITS(v, splits)
1  if v ≠ nil and v ∈ splits then
2      SPLIT(v, vsplits)
3      APPLY-SPLITS(vsplits[v].v_1, vsplits)
4      APPLY-SPLITS(vsplits[v].v_2, vsplits)
5  return
```

The SPLIT function is what performs a vertex split operation. Lines 1–2 ensure that we don't split a vertex that was already split. Lines 3–6 ensure that the left and right neighbours or their nearest ancestor are present before we perform the split. Line 7 actually performs the split as discussed above. The predicate $active(v)$ is true when vertex v appears in the mesh at its current state of refinement and false otherwise. The $activeancestor(u, v)$ function returns the active ancestor of vertex v which is adjacent to vertex u in the current mesh. This is determined by searching the neighbourhood of u for the vertex v' which satisfies $rank(v) \in range(v')$.

```
SPLIT(v, splits)
1  if not active(v) then
2      return
3  while [not active(vsplits[v].v_l)] and [activeancestor(v, vsplits[v].v_l) ∈ splits]
4      SPLIT(activeancestor(v, vsplits[v].v_l), splits)
5  while [not active(vsplits[v].v_r)] and [activeancestor(v, vsplits[v].v_r) ∈ splits]
6      SPLIT(activeancestor(v.vsplits[v].v_r), splits)
7  PERFORM-SPLIT(vsplits[v])
8  return
```

Lemma 6. *The running time of the* SELECTIVE-REFINE *procedure is* $O(|M_0| + kd)$*, where* k *is the number of vertices whose regions of influence intersect* q*, and* d *is the average degree of these vertices.*

Proof. The running time of the GET-SPLITS procedure has been shown to be $O(|M_0| + k)$ in Lemma 3. Therefore, we need only prove the running time of the APPLY-SPLITS procedure. Note that the number of calls to the APPLY-SPLITS procedure is at most most 3 times the number of calls to the SPLIT procedure. In turn, the number of calls to the SPLIT procedure

is at $2k$, since for each $v \in splits$, SPLIT is called at most once from APPLY-SPLITS and once recursively. Disregarding recursive calls, the running time of APPLY-SPLITS is constant, and the running time of *Split* is $O(d')$ where d' is the degree of the vertex being split. Amortizing this value over all calls to SPLIT we get a running time of $O(kd)$. □

3.4 Analysis and Comments

Finally, we summarize with a theorem describing our selective refinement scheme and comment on other practical aspects of the scheme.

Theorem 1. *Given a mesh in the progressive mesh representation, the selective refinement scheme uses $O(n)$ preprocessing time and $O(|M_0| + kd)$ query time, where $|M_0|$ is the size of M_0, k is the number of vertex splits whose regions of influence intersect the query region, and d is the average degree of the vertices retrieved.*

Proof. The preprocessing resources follow from Lemma 2 and Lemma 5. The query time follows from Lemma 6. □

At this point we note that it is also possible to generalize this scheme to perform selective refinement on any mesh $M_t \in \{M_0, \ldots, M_{|vsplits|}\}$. In this case, the query region is returned at exactly the level of detail at which it appears in M_t. To achieve this generalization, we simply "prune" the search when a vertex split record is reached who's index is greater than t. In analyzing the running time of the generalized algorithm, the value of k is defined in the same manner as above, but with respect to the vertex split sequence $v_1, \ldots, v_t$.

From the pseudocode, it is easy to see that the preprocessing and selective refinement procedures are quite simple, and hide only small constants in the big-Oh notation (see Section 3.6 for empirical results). Another merit of this scheme is that since it uses only axis aligned bounding boxes, the search procedure can be implemented using only comparison operations and is therefore not subject to the rounding errors inherent in floating point arithmetic computations.

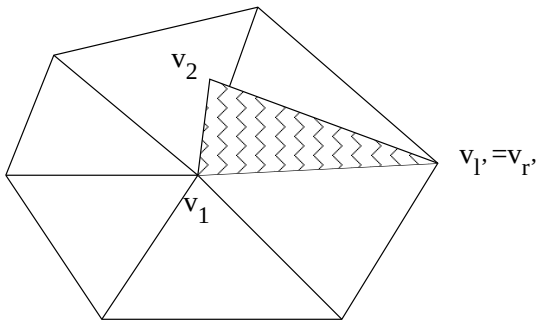

Figure 8: An example of the case where $v_l' = v_r'$.

3.5 Extensions and Implementation Notes

3.5.1 Dealing with Missing Neighbours

In Section 3.3, it was noted that when splitting a vertex v, it may be the case that one (or both) of the vertices v_l or v_r may not be present in the mesh. In this case we suggest using the nearest living ancestor of v_l or v_r, call these v_l' and v_r', respectively. An important point to note when implementing this is that it may be the case that $v_l' = v_r'$. In this case, the two resulting triangles (v_1, v_2, v_l) and (v_1, v_2, v_r) are identical, and are only connected to the mesh by a single edge (see Figure 8 for an illustration). Depending on the application, these types of triangles may or may not cause problems. For display purposes, a reasonable solution is to simply flag these special triangles and not render them.

Another approach to solving this problem, which is proposed by Hoppe [14] is to force splits of v_l' and v_r' and their descendents until v_l and v_r are recreated, at which point the vertex v can be split. This approach seems to work well, and prevents extremely abrupt changes in resolution which may be visually unpleasant. Unfortunately, it may also significantly increase the number of vertices which are split thereby increasing the running time of the selective refinement algorithm.

3.5.2 Invalid Triangles in Query Regions

Although the selective refinement scheme described in this section guarantees that all the triangles of M which intersect the query region q are extracted, it does not guarantee that there are not other triangles not in M which intersect the query region. Figure 9 shows an example in which this occurs. The figure shows two edge collapses, and the resulting mesh

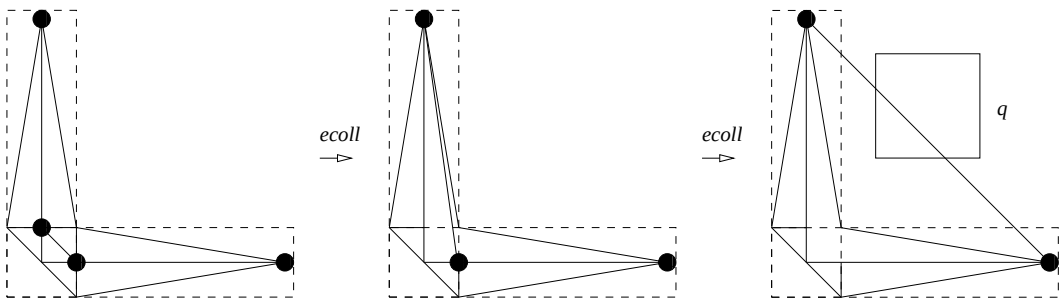

Figure 9: An example in which a triangle not in M intersects q.

with the *roi* boxes shown as dashed lines. The query region q intersects a triangle which is not in M, but does not intersect any of the *roi* boxes.

For the applications described in Section 4 this is not a problem. However, for some applications it may be necessary to avoid this situation. In order to do so, we must ensure that every triangle $\triangle abc$ that is in an intermediate mesh is completely covered by the $roi(a) \cup roi(b) \cup roi(c)$. In this way, if the triangle intersects q, then so does one of $roi(a)$, $roi(b)$, $roi(c)$, and the triangle will not remain in the selectively refined mesh since one of the vertices of the triangle will have been split.

A simple way to achieve this is to augment the definition of *roi* so that for a vertex v, $roi(v)$ contains all triangles incident to v. If this approach is combined with Hoppe's method of forcing vertex splits (see Section 3.5.1), then the query region q will not intersect any triangle not in M, since every triangle not in M will be covered by the *roi* of each of its vertices. Unfortunately, this increases the running time of the selective refinement procedure on two counts. Firstly, since the size of the *roi* increases, the likelihood that a query region will intersect them also increases, which increases the number of vertex splits which are reported. Secondly, using Hoppe's method of forcing vertex splits results in increased running times as well.

3.6 Empirical Results

In order to verify the viability of the bounding box approximation to the regions of influence used in the selective refinement algorithm, some empirical tests were performed on triangulated terrains (TINs). Tests were performed on random TINs as well as two real TINs, containing 10000 and 20000 points, each. Random TINs were generated by choosing uni-

formly distributed points in the unit square and then computing a Delaunay triangulation of these points. We measured and compared our results on random TINs with those on the two TINs, with the same number of vertices, and found that the results obtained were almost identical.

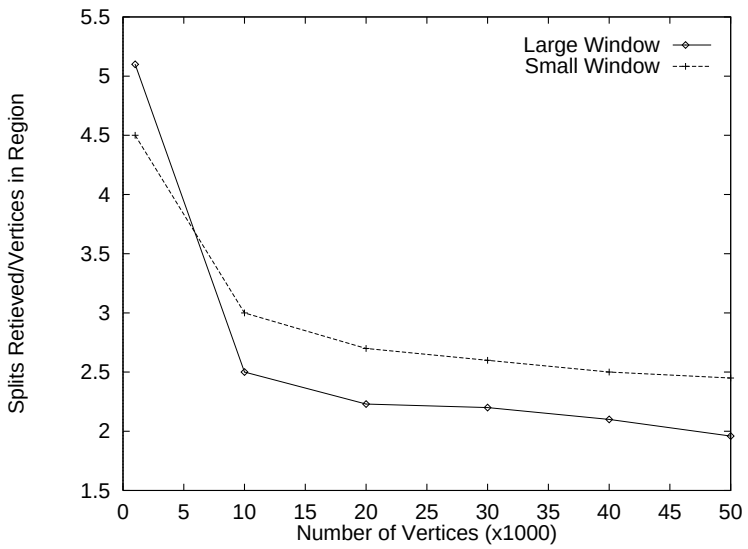

Figure 10: Performance of selective refinement algorithm for medium and large query regions.

Figure 10 shows the ratio between the number of vertices of the TIN, T, in a query region and the number of vertex splits which are retrieved when selectively refining the region. The tests take a query window of a fixed size and places it at 2500 regularly spaced locations on the TIN and perform selective refinement at each location. Both small (1/25 of the TIN's surface area) and large (1/4 of the TINs surface area) query windows were tested. The main results of these tests is that the ratio between the number of splits performed and the number of vertices in the query window converges to a small constant (< 3) as n increases.

Figure 11 shows results for a query region consisting of a single point on the surface of the TIN. Again, the query point is placed at 2500 regularly spaced locations on the TIN and selective refinement is performed. Figure 11 shows the results for TINs with up to 50000 vertices, and shows that even the worst-case running times tend to be logarithmic in the size of T. Also of interest are the absolute values in Figure 11 since these show that the constants are quite small. In no case does the number of vertex splits performed exceed 70.

When taken together, these two sets of experiments would suggest that the running time of the selective refinement algorithm is of the form $O(\log n + k)$, where k is the complexity

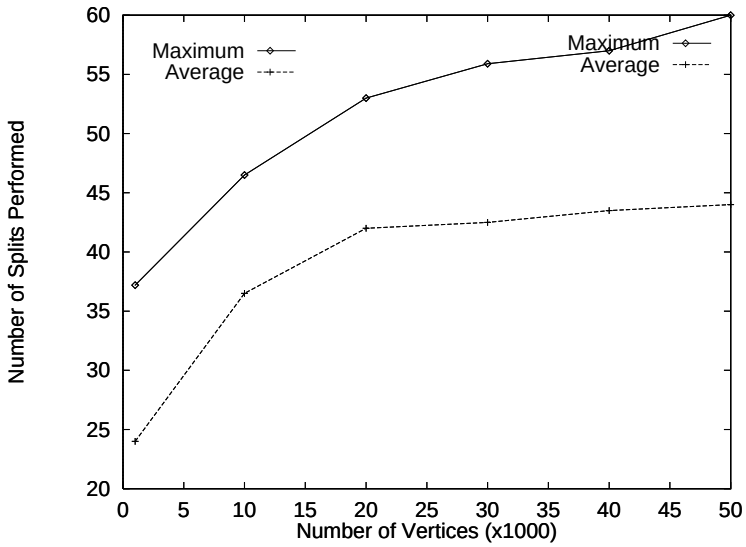

Figure 11: Performance selective refinement algorithm for small query regions.

of the TIN in the query region, and n is the number of vertices in the TIN. Thus, at least empirically, the selective refinement procedure exhibits optimal behaviour up to small constant factors.

4 Applications

The triangulated irregular network (TIN) is one of the basic models for representing geographical data where a triangulated set of points is stored together with its elevation. The TIN is essentially a mesh with the added constraint that no two points on it's surface have the same projection on the (x, y) plane. TINs were introduced in 1978 (see [8, 20, 21]) and they are a fundamental data structure in GIS and related areas.

In this section, we describe some applications of our selective refinement scheme, with particular attention paid to applications to TINs. We show that meshes stored in the progressive mesh representation naturally supports a number of operations common in both computational geometry and geographic information systems. Because of the small constants involved in the PM representation, the algorithms presented in this section are competitive with existing algorithms which operate on meshes in a standard representation. The advantage of these algorithms over these existing algorithms are:

1. these algorithms work on meshes in the PM representation, maintaining all the advantages of the PM representation,

2. these algorithms require little or no additional preprocessing or storage beyond the progressive mesh construction process, making them, in some cases, more efficient than existing solutions, and

3. by using the "pruning" technique described in Section 3.4, all the algorithms presented here can be used to efficiently solve approximate versions of the problem in question, i.e., the problem can be solved on any of the meshes $M_0, \ldots, M_{|vsplits|}$.

Throughout the remainder of this section we use T when discussing a TIN or a planar triangulation in the same manner as we have previously used M to denote an arbitrary mesh.

4.1 Point Location

Point location in a triangulation is a well studied problem in computational geometry, and a number of point location algorithms exist. The point location problem consists of determining, given a planar triangulation, T, the triangle, t, in which a query point lies. Although theoretically optimal algorithms ($O(n)$ preprocessing time and $O(\log n)$ query time) exist for the point location problem, these algorithms have large constants hidden in the big-Oh notation which makes them less useful in practice. This is witnessed by the fact that most real world implementations use schemes that are less than optimal but which work well in practice [9, 18].

The selective refinement procedure described in Section 3 can be used as an efficient method for locating a point in a triangulation. The motivation for this application is twofold: (1) the selective refinement algorithm is fast in practice, and thus should yield a fast point location algorithm, and (2) if a triangulation is already in the progressive mesh representation then point location can be performed without any additional preprocessing.

To perform point location on a progressive mesh, we perform selective refinement in which the query region is a vertical line, namely the vertical line which passes through the query point q (see Figure 12). The selective refinement algorithm is run and the triangle in which q lies is found. The running time of this algorithm is clearly $O(|M_0| + kd)$ where k and d are defined as in Section 3.

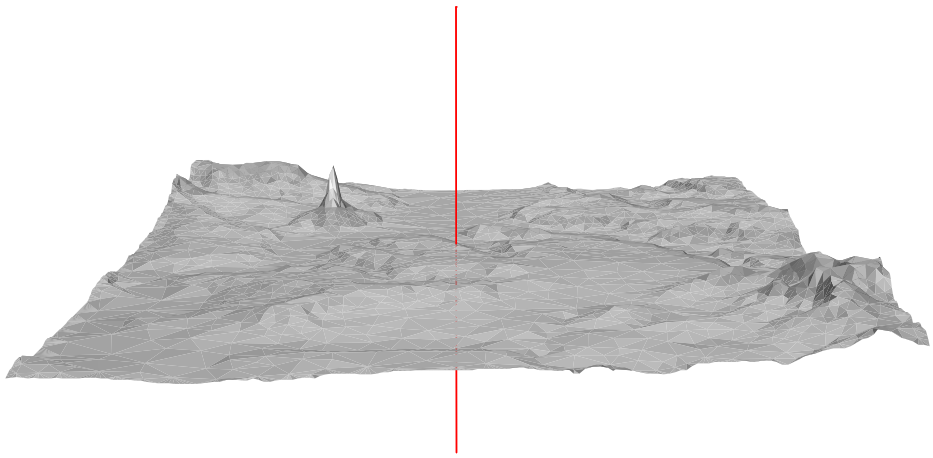

Figure 12: Expressing a point location query as a selective refinement query.

If our objective is to do largely point location queries in a triangulation, then the fitness function that can be used in the construction algorithm to obtain a progressive mesh is to collapse the edge (v_1, v_2) such that the resulting region of influence is the smallest among all possible edge collapses. A similar heuristic is used in the construction of R-Trees [12].

4.2 Isoline Extraction

The problem of isoline or contour line extraction is stated as follows: given a query elevation h, return all triangles of T which occur at elevation h. Given the triangles which occur at elevation h, the polygons and polygonal chains which form the isolines can be found in $O(k)$ time (where k is the number of triangles) by starting at an arbitrary triangle and walking along triangles until a triangle already visited is reached, or the border of T is reached. When this occurs, another unvisited triangle is chosen, and the same procedure is applied.

For answering isoline queries on a progressive TIN, we simply perform selective refinement in which the query region is the plane $z = h$, and report all triangles of T which intersect this plane. An example of this approach is shown in Figure 13. The running time of this algorithm is $O(|M_0| + kd)$ where k and d are defined as in Section 3.

From this, it follows that the edge collapse sequence can be optimized for isoline extraction by always collapsing the edge which produces the smallest z-interval in the resulting *roi*. This heuristic is similar to that suggested in Section 4.1 for optimizing the edge collapse sequence for point location.

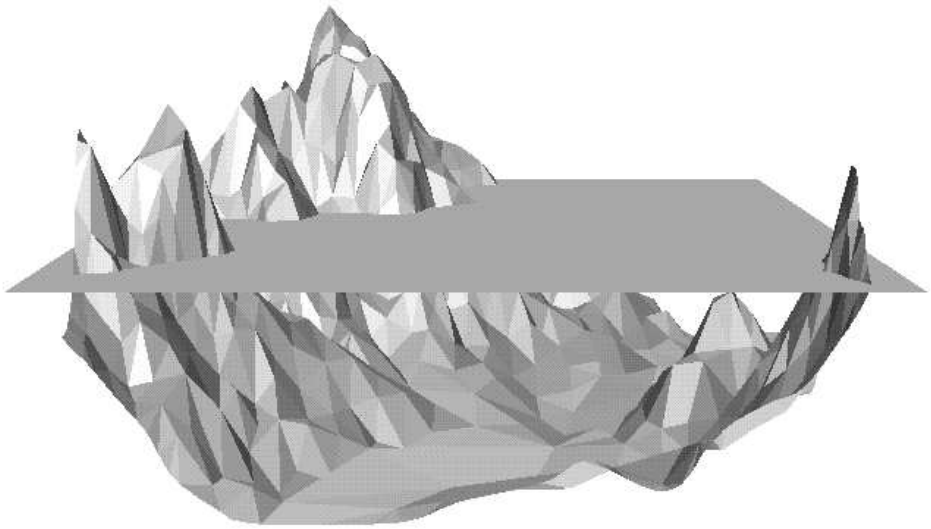

Figure 13: Expressing an isoline query as a selective refinement query.

Progressive TINs also support progressive transmission of isolines. To perform progressive transmission of isolines, we note that a vertex split operation can only affect isolines in the neighbourhood of the split vertex, and so maintaining isolines under the vertex split operation is a straightforward localized matter. Therefore, progressive transmission of isolines involves first transmitting the coarse grained TIN T_0 and then transmitting the vertex split records which affect the elevation(s) in question. At the receiving end, extracting the isolines from T_0 can be done using a linear time brute force method, and these isolines can be maintained using only local operations as new vertex splits records are received.

4.3 Visibility Queries

Two points on a surface are said to be visible if the line segment joining them does not properly intersect with the surface. The visibility query problem is the following: given two query points p and q, is p visible from q?

A straightforward solution to the visibility query problem is as follows. Locate the triangle in which p lies and then walk along triangles in the direction of q until either (1) an edge is crossed which occludes q in which case the answer is negative, or (2) the triangle containing q is reached in which case the answer is positive. The preprocessing required by

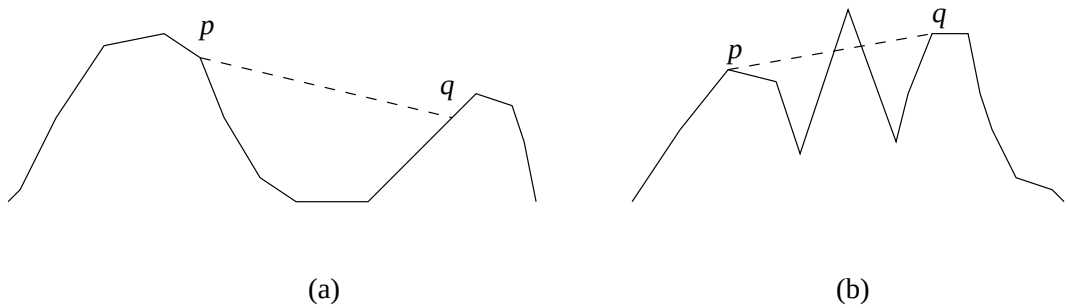

Figure 14: An example of two visbility queries on a TIN. In case (a), the selective refinement procedure will not do any refinement in the valley between p and q. In case (b), the selective refinement procedure will determine very quickly that visbility is blocked by the peak between p and q.

this algorithm is the same as that for point location. The query time is the time to locate q plus the number of edges, k, which intersect the segment $\overline{pq}$.

On a progressive TIN, visibility between a pair of points can be answered by selectively refining the TIN along the (3 dimensional) line segment $\overline{pq}$. In many cases, this will lead to a faster query time. If p and q are visible, then the line segment $\overline{pq}$ does not properly intersect T. In this case $\overline{pq}$ is not likely to intersect many of the regions of influence of the vertex splits, and the selective refinement procedure will finish very quickly indeed. In p and q are not visible, then $\overline{pq}$ may intersect many of the regions of influence of vertex splits, but the selective refinement procedure can be stopped the first time a triangle of T is found which intersects $\overline{pq}$. See Figure 14 for an example. This method also has the advantage that it doesn't require the points p and q to be on the surface.

4.4 External Memory Progressive Meshes

External memory methods for meshes are of significant practical relevance to the field of GIS, as the amount of geographic data currently available exceeds the capacity of internal memories. This motivates the development of an external memory storage scheme for progressive meshes. With such a scheme, the coarse grained mesh, M_0, could be stored in internal memory, and user could perform selective refinement to refine a small portion of this mesh and work with it. A number of external memory spatial data structures exist which can be used to store progressive TINs in the external memory. We believe that the R-Tree

[12], or one of it's variants, is a data structure well suited for this application. R-trees are designed specifically for storing axis aligned boxes. (A variety of other possible spatial index structures exist see e.g. [10, 11, 16, 23] and also consult the new survey article by Nievergelt and Widmayer [19].) In particular, one variant of the R-tree, the packed R-tree constructs an R-tree in a bottom up fashion from a static set of axis aligned boxes.

Thus in order to construct an external memory representation of a progressive mesh, build a packed R-tree on the list of vertex splits, where the box associated with a vertex split v is $roi(v)$. When performing selective refinement we use the packed R-tree to extract the relevant vertex splits and then proceed in the manner described in Section 3 to sort and apply the splits.

Using this representation, the algorithms for point location, elevation queries, and visibilities queries can all be applied to TINs stored in the PM representation in external memory.

5 Conclusions

This work has extended Hoppe's progressive mesh representation so that it efficiently supports selective refinement. Using this selective refinement scheme a number of algorithms for fundamental problems in computational geometry and geographic information systems have been devised.

Perhaps as important as the refinement scheme itself are some of the ideas behind it. Augmenting the DAG-based hierarchies of Dobrindt and de Berg [2] and Puppo [22] with the *roi* information described in Section 3 could be beneficial to these schemes as well. For one, it would allow the type of exact selective refinement described in Section 3 (as opposed to the heuristic computer graphics oriented methods currently used), and could improve the performance of these schemes by replacing complicated point-in-triangle tests with the much simpler point-in-rectangle test. Of course, an empirical study would be needed to determine whether the increase in performance due to the simpler tests is enough to offset the increase in work caused by the bounding box approximation.

It is also worth noting that algorithms in Section 4 which were described for the progressive mesh could also be used with other multiresolution surface models. For this to work, these models would need a method of exact selective refinement like the one described in Section 3. Since such methods can be designed by using analagous techniques, namely the

roi information, the algorithms in Section 4 actually solve a number of problems on most multiresolution surface models.

Any problem related to surface models can be attempted in multiresolution models as well. It may be possible to use the multiresolution aspect of the model to advantage. One obvious approach is to provide faster approximation algorithms by simply running the standard algorithm on a mesh with a lower level of detail. In this case, it is important to bound the quality of the resulting approximation, either theoretically or empirically. One could also imagine using these models to develop interruptible algorithms, i.e., algorithms that can be interrupted at any time to produce an approximate result but which will eventually converge to an exact solution given enough time. This could be done by finding an exact solution on the coarse mesh M_0 and improving it by increasing the resolution of the mesh in certain areas.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990.

[2] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *ACM Symposium on Computational Geometry*, 1995.

[3] L. de Floriani. A pyramidal data structure for triangle based surface description. *IEEE Computer Graphics and Applications*, 9:67–78, 1989.

[4] L. de Floriani, P. Magillo, E. Puppo, and M. Bertolotto. Variable resolution operators on a multiresolution terrain model. In *4th ACM Workshop on Advances in Geographic Information Systems*, pages 123–130, 1996.

[5] L. de Floriani, D. Mirra, and E. Puppo. Extracting contour lines from a hierarchical surface model. In *Eurographics '93*, pages 249–260, 1993.

[6] L. De Floriani, P. Magillo, and E. Puppo. Building and traversing a terrain at variable resolution. Technical Report PDISI-97-02, University of Genova, 1997.

[7] L. De Floriani, P. Magillo, and E. Puppo. Variant - processing and visulazing terrains at variable resolution. In *Proceedings of the 5th International Workshop on Geographic Information Systems (ACM-GIS'97)*, pages 15–19, 1997.

[8] C. M. Gold. The practical generation and use of geographic triangular element data. *Harvard Papers on Geographic Information Systems*, 5, 1978.

[9] M. T. Goodrich, M. Orletsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *ACM Symposium on Discrete Algorithms*, 1997.

[10] O. Guenther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proceedings of the 5th IEEE Conference on Data Engineering*, pages 598–615, 1989.

[11] R. H. Güting. An introduction to spatial database systems. *The VLDB Journal*, 3:357–399, 1994.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference 1984*, pages 47–57, 1985.

[13] H. Hoppe. Progressive meshes. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 99–108, 1996.

[14] H. Hoppe. View-dependent refinement of progressive meshes. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, page to appear, 1997.

[15] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuezle. Mesh optimization. *siggraph95*, pages 247–254, 1995.

[16] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, London, 1992.

[17] A. Maheshwari, P. Morin, and J.-R. Sack. Progressive TINs: Algorithms and applications. In *Proc. 5th ACM Symposium on Geographic Information Systems*, pages 24–29, 1997.

[18] S. Näher. The LEDA manual, version R-3.4.1. Technical report, Max-Planck Institut, 1996.

[19] J. Nievergelt and P. Widmayer. Spatial data structures: concepts, design, and choices. In J. Urrutia and J.-R. Sack, editors, *Handbook of Computational Geometry*. Elsevier Sciences, to appear.

[20] T. K. Peucker. Data structures for digital terrain models. *Harvard Papers on Geographic Information Systems*, 5, 1978.

[21] T. K. Peucker, R. J. Fowler, J. J. Little, and D. M. Mark. The triangulated irregular network. In *Proceedings DTM Symposium American Society of Photogrammatry - Amercian Congress on Survery and Mapping*, pages 24–31, 1978.

[22] E. Puppo. Variable resolution terrain surfaces. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, 1996.

[23] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.