Good terrain geometry, cheap!

Mark Duchaineau[1,2], Murray Wolinsky[1], David E. Sigeti[1], Mark C. Miller[2],
Charles Aldrich[1] and Mark Mineev[1]

[1]Los Alamos National Laboratory
Los Alamos, New Mexico

[2]Lawrence Livermore National Laboratory
Livermore, California

Abstract

Real-time terrain rendering for interactive visualization remains a demanding task. We present a novel algorithm with several advantages over previous methods: our method is unusually stingy with polygons yet achieves real-time performance and is scalable to arbitrary regions and resolutions. The method provides a continuous terrain mesh of specified triangle count[1] having provably minimum error in restricted but reasonably general classes of permissible meshes and error metrics. Our method provides an elegant solution to guaranteeing certain elusive types of consistency in scenes produced by multiple scene generators which share a common finest-resolution database but which otherwise operate entirely independently[2]. This consistency is achieved by exploiting the freedom of choice of error metric allowed by the algorithm to provide, for example, multiple exact lines-of-sight in real-time. Our methods rely on an off-line pre-processing phase to construct a multi-scale data structure consisting of triangular terrain approximations enhanced ("thickened") with world-space error information. In real time, this error data is efficiently transformed into screen-space where it is used to guide a greedy top-down triangle subdivision algorithm which produces the desired minimal error continuous terrain mesh. Our algorithm has been implemented and it operates at real-time rates.

CR Categories and Subject Descriptors: [Computer Graphics]: I.3.3 Picture/Image Generation - *viewing algorithms*; I.3.5 Computational Geometry and Object Modeling - *geometric algorithms, object hierarchies*; I.3.7 Three-Dimensional Graphics and Realism - *virtual reality*.

1 Introduction

The theme of 1990's may well be *doing more with less*. Nowhere is this motif more in evidence than in interactive visualization. Advances in hardware maintain their reliable momentum yet expectations and demands grow still more rapidly. Levels of realism now expected in real-time surpass those previously attained only when lengthy periods of dedicated off-line rendering time were available. More complex scenes must now be displayed in real-time at higher levels of detail for regions of greater extent than ever before. Appearances must conform to realistic first-principles or phenomenological models and geometry must be quantitatively accurate. Moreover, the final set of generated frames may now be subject to greater scrutiny: inter-frame consistency must be assured (e.g., pop-ups

1. Give or take a few triangles.

2. In particular, the scene generators need not communicate.

must be absent or minimized) and scenes generated may be required to exhibit consistency with scenes generated by other more or less capable renderers[1]. Terrain rendering for interactive visualization systems (e.g., flight simulators) comprises an active area of research which evidences these concerns in particularly acute form.

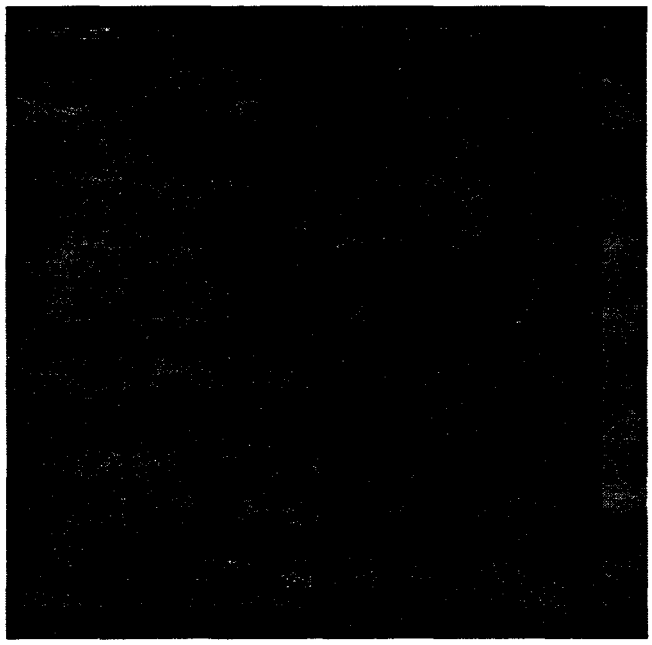

FIGURE 1. A snapshot of terrain produced using the algorithms presented here

Although graphics hardware now supports greater and ever-increasing polygon throughput, the demand for these polygons is not limited to terrain: atmospheric models, such as clouds; models of background objects and clutter; and models of objects of interest all insist on their fair share of the polygon supply. So, although more polygons are available, each individual polygon must deliver more value. Thus it is now timely to examine methods which can address polygon utilization and which can precisely place detail where needed.

We have developed and implemented an interesting new algorithm which shows promise in addressing optimal polygon utilization in real-time. The work presented here deals with rendering terrain from a moving viewpoint in real time. We focus our present treatment on issues relating to assuring accurate terrain geometry: questions related to the appearance of these terrain polygons will be addressed separately.

The precise problem definition presented here and the algorithms developed have been influenced by a variety of considerations. First, our terrain generation algorithms support an interactive real-time rendering system by supplying the renderer with terrain triangles. The renderer devotes only a fraction of its polygon budget to terrain. Therefore our algorithms must run in real-time and work with a fixed and possibly small triangle budget

1. Assuring the requisite consistency of independent simulations in real-time adds an additional challenge to an already demanding task.

whose size can be arbitrarily specified at run-time. Second, the generated frames are used to test hardware sensors (in addition to being shown to human subjects) and therefore our algorithms must not only produce accurate terrain geometry but also must be able to quantify the degree of accuracy achieved. Third, different renderers using our algorithms may run simultaneously from possibly different view points with differing fields of view while using the same static terrain database. In this situation it is important to have control over certain aspects of the geometry in order to provide requisite consistency between generated scenes[1]. Attaining this control requires a flexible algorithm capable of handling different error metrics. Fourth, the renderers are expected to have reasonably long expected useful lifetimes so our algorithms must be portable across a variety of platforms and they must be able to confront rendering problems of potentially arbitrary scale.

Our algorithms, and the terrain generator based on them, meet, to a large degree, all of these objectives. In this contribution we describe the fundamental algorithms themselves.

Ideally the basic algorithm should do the following: given an arbitrary terrain, and a specified maximum number of triangles, the algorithm should output produce a continuous mesh in real-time containing no more than the allowed number of polygons, which minimizes an arbitrary user-supplied error metric. Unfortunately, this task remains beyond current capabilities. It is not clear how to efficiently perform optimization over arbitrary polygonal meshes for general error metrics *period*, let alone in real-time. Progress requires suitable restrictions be placed on the class of meshes over which the optimization is performed and also on the nature of the allowable error metrics. Our current methods demand little of the mesh other than the existence of efficient continuity-preserving real-time methods of subdivision. The restrictions on error metrics are greater, but not unnatural: first, the error metric should be easily computed in real-time (assuming adequate pre-processing) and second, we restrict our interest to L_∞ metrics[2], because our methods are guaranteed to perform well on these metrics.

2 Related work

We are not the first to confront these problems: nor do we expect to be the last. All current methods for rendering large-scale terrains are multi-scale in nature. Two general multi-scale approaches exist: those which employ triangulated irregular (fully general) networks (TINs)[3] and those which employ quasi-regular meshes[4].

A survey of the state of the art will not be attempted here: rather we choose to compare our present work with two recent works which take a generally similar approach.

In general the overall contributions we add are

1. In particular, lines of sight may be required to be consistent.
2. By which we mean that the error for a mesh is the maximum error for any triangle in the mesh.
3. Such as the commercial product of MultiGen Corporation termed CAT, or continuously adaptive terrain.
4. Such as the method presented here.

1. more general priorities which can be exploited for real benefits (e.g. exact lines-of-sight that ensure terrain correlation)
2. guaranteed bound on errors
3. direct attack on fixed-triangle-count problem
4. fully incremental approach (fine-grained incrementalism)
5. more scalable
6. simplicity of algorithm
7. provable optimality for interesting classes of metrics and meshes

The approach most similar to ours is the work done at Georgia Tech[1]. There are many important similarities and even some advantages of this earlier work. The Georgia Tech team uses the same mesh we use. They claim optimality as do we. Both techniques exploit the fact that small changes in mesh for small changes in view frustum. Both techniques decide level-of-detail on a triangle-by-triangle basis and therefore evade many problems with earlier methods. Their methods can also be used with dynamic terrain (e.g. blowing up a hill), whereas ours cannot (yet)[2].

However, there are also some differences and advantages of our method. We use the mesh differently. In particular they use complex rules to maintain continuity which restrict the error metrics they can employ. We maintain continuity simply without sacrificing generality of error metric. Their claim of optimality is unsupported: we present here a proof for our approach. We explicitly realize (nearly) fixed triangle counts. Their error bound is not guaranteed. Ours is[3]. We seem to be faster[4] and our approach is likely to be more scalable[5].

Another recent approach, the *progressive mesh representation*, appears to offer much promise. It performs complex and generically "good" optimization (by energy minimization). It produces avery nice complexity/accuracy trade-off curve. Progressive meshes are basically sequences of edge-collapse operations, ordered according to a pre-processed priority and reversible fairly arbitrarily based on a run-time error priority. However, it is not clear how to build nested bounds for efficient view-frustum culling or line-of-sight correction. It is also not clear how to guarantee error bounds at run-time. In general, the method,

---

1. See Lindstrom *et al.*
2. Our present pre-processing is too slow for this, but it is highly localized so that, in principle, we should be able to update our database quickly.
3. However they can (fairly) reliably tune the complexity/accuracy trade-off.
4. From their charts we appear to be about three time faster for similar quality meshes with the same triangle count.
5. They wisely compute for a block of world-space terrain deltas, the minimum and maximum delta sizes where actual computation of the screen delta (expensive) is required. This allows them to make quick, incremental updates to a quadtree of blocks. However, once they get the blocks roughed out incrementally, they toss out 99% of the vertices in a second fine-grained step. The delta bounds allow them to avoid some screen-projection computations, but they still must do roughly 100 pieces of work for each output triangle.

while appealing, appears to be too slow at present to offer a practical alternative to the method presented here.

An important distinguishing aspect of our approach it is that it allows reversing the order in which queries and approximations are made. Other real-time approaches generally simplify first and then query later. Thus, in general, a line-of-sight query will be performed on an approximated terrain. Our approach allows queries to be made first and simplifications second. Thus our terrain approximations can preserve important (and fairly arbitrary) characteristics of the exact terrain. This capability is particularly important in the increasingly important area of assuring consistency of different simulations: simulations using our approach will be "consistent because correct".

3 Overview

The problem we address here is that of devising a scalable[1] algorithm for constructing terrain meshes of specified triangle count with good and known accuracy in real-time. More specifically, we identify a class of meshes and a class of error metrics such that the best mesh of the specified class with a given number of triangles can be identified[2] in real-time. Further we specify the algorithm which accomplishes this minimization and then the optimizations which are needed to attain real-time performance.

Our algorithms work as follows. The desired terrain is known at the finest (uniform) level of resolution of interest. A *pre-processing* stage is used to construct sets of triangles which approximate the desired terrain at discrete progressively coarser levels of resolution. At each (coarse) level these triangles form continuous regular triangular meshes. Generation of these regular meshes terminates (conceptually) when a mesh consisting of a single triangle which covers the entire terrain has been produced[3]. During pre-processing the data structures containing the individual terrain triangles are fortified with real-space error information. This information used to construct screen-space error metrics that guide the process of real-time subdivision. In *real-time*, a top-down greedy algorithm starts at the coarsest level of resolution and subdivides meshes locally to provide a sequence of non-uniform but continuous (quasi-regular) terrain meshes which minimize the specified screen-space error criterion. Mesh subdivision terminates when a target triangle count has been achieved. The real-time algorithm is scalable in terrain extent and resolution because of its top-down nature and is able to meet real-time needs when it is recast to perform incrementally.

The meshes we use are assembled from pre-constructed triangles which can be easily subdivided while preserving continuity. Any meshes having this property could be employed

1. We require scalability in terrain extent and resolution.
2. In practice we are content to settle for less. In general we do not care if the produced mesh is the best, we merely would like it to be within some tolerance. However, it is interesting that our algorithm can, for large classes of meshes and metrics produce the best possible mesh.
3. In practice, generation stops at the level immediately below, when two triangles which jointly cover the entire domain have been produced.

with our algorithm: however, since we ultimately output triangles it is convenient to work with triangular approximations from the outset.

As stated earlier we restrict our interest to easily-computed L_∞ metrics, because our methods are guaranteed to perform well on these metrics, but also because this class of error metric is well-suited to the rendering problems of interest. In particular, the approximations necessary to render scenes in real-time inevitably lead to possible distortions in the rendered images. The (geometric, screen-based) *distortion* of an image of a given point in a scene is naturally defined as the difference in screen pixels between where the test point was rendered and where, were the highest-resolution data used, it should have been rendered. An L_∞ metric for a scene sets the *total distortion* of the scene as the maximum distortion obtained as the test point ranges over all visible points. Thus minimizing a distortion-based L_∞ metric for a scene guarantees that no pixel is displaced more than the value of the total distortion of the scene. This metric has considerable intuitive appeal and is precisely the desired metric for a wide variety of applications. Thus our restriction to L_∞ norms can be defended[1].

We are concerned here with algorithms that minimize the total distortion of a known terrain[2]. For simplicity of exposition[3] it is convenient to use a flat-Earth reference model[4]. Choose the reference plane as $z = 0$: the desired terrain is given by $z = h(x, y)$. We assume that the terrain is specified at the finest resolution of interest by providing height samples z_{ij} for a uniformly-spaced grid of post positions (x_i, y_i)[5]. For any arbitrary number of triangles, n, we find the continuous mesh containing n triangles which minimizes the total surface distortion. Our algorithms select the cheapest (coarsest) triangles possible: if it is unnecessary to subdivide a triangle, we don't.

Our description proceeds as follows. First we describe the allowable meshes. Because our interests are primarily practical and not theoretical we do not try to present here a comprehensive and tight characterization of all of the possible classes of meshes to which our algorithm could profitably be applied. Rather, we present the class of meshes we have actually employed and describe the features of those meshes which are important to the operation of the algorithm. No argument is given to establish that this particular class of meshes is in any sense the optimal choice in the space of all possible classes of mesh[6].

---

1. In fact we can apply our algorithm to error metrics which are not L_∞ but we will not have, in general, optimality.
2. However, the freedom to modify the error metric to account for features of interest other than geometric distortions is very valuable. In Section 9 we exploit that freedom.
3. As well as to correspond to the structure of the code as it exists presently.
4. The algorithm would not differ in essence from what is presented here if any other choice of "base surface" was made.
5. For convenience the continuous height field h(x, y) at the finest resolution is assumed to be derived from the discrete samples by the triangular interpolation presented below.
6. And, in fact, there is no reason to believe that it is. But there are reasons to believe that our choice is not too bad.

These tasks are performed in Section 4. In the brief Section 5 we describe the class of error metrics we employ. Here we give a more precise account. We are concerned primarily with the particular error metric described above, namely the bound on the total distortion. Before proceeding to describe the real-time algorithm, it is useful to describe the operations performed during pre-processing. This is done in Section 6. In Section 7 we present the actual algorithm we employ in real-time and prove that it yields an optimal mesh. However, the algorithm in its simplest form is not sufficiently fast for real-time use. Therefore in Section 8 we describe the enhancements required for incremental operation, which are necessary in practice for real-time performance. In Section 9 we present additional details concerning the implementation and present an example of the flexibility of our approach by showing how the method can be used to efficiently obtain exact lines-of-sight. In Section 10 we describe necessary and desirable extensions of the present work. Our final conclusions are given in Section 11.

4 Mesh structure

Because the ultimate product of our algorithm is a set of triangles it is convenient at the outset to work with mesh structures composed only of triangles. We want triangles which can be easily assembled into a large variety of (generally non-uniform) continuous meshes that nicely approximate given terrain. The algorithm itself demands little of the meshes; only that when a triangle in a continuous mesh is subdivided, triangles must be available to form a new continuous mesh. Our current proof of the optimality of the algorithm requires a bit more: the subdivision algorithm must be *determinate*; i.e., after subdivision of a triangle in a continuous mesh the new mesh must contain a *definite* set of triangles, independent of the initial state of the mesh, such that all the triangles produced during the split are contained within this set.

Understanding the meshes requires understanding the triangles. We describe them in the next subsection. Then we explain how these triangles can be assembled into continuous meshes: more precisely, we show how a new continuous mesh is formed when a triangle in a coarser continuous mesh is split.

4.1 Binary triangle tree

The triangles we use form a binary tree and the clearest understanding of these triangles starts at the top of this tree. All of our triangles have simple projections onto the base plane ($z = 0$): *these projections are right isosceles triangles.* Consider the projections first. Take a right isosceles triangle in the base plane containing the domain of the height field (assumed to be of finite extent). This single all-encompassing triangle is called the *root* triangle[1]. Bisecting the right angle of the root triangle yields two half-scale right isosceles triangles. Recursively enumerating this procedure yields a collection of right isosceles tri-

1. In fact we start at the level immediately below, with two triangles, and these two triangles are not exactly the same as those obtained by subdividing the root. But this fictitious root triangle facilitates our exposition. In the notation below, we actually only use levels $l \geq 1$.

angles, forming a binary tree with 2^l triangles at level l, each at a scale factor of 2^{-l} relative to the root (Figure 2). These triangles form the projections of all of the triangles we use. Assigning elevations to the vertices[1] of these triangles completes the construction of the triangles.

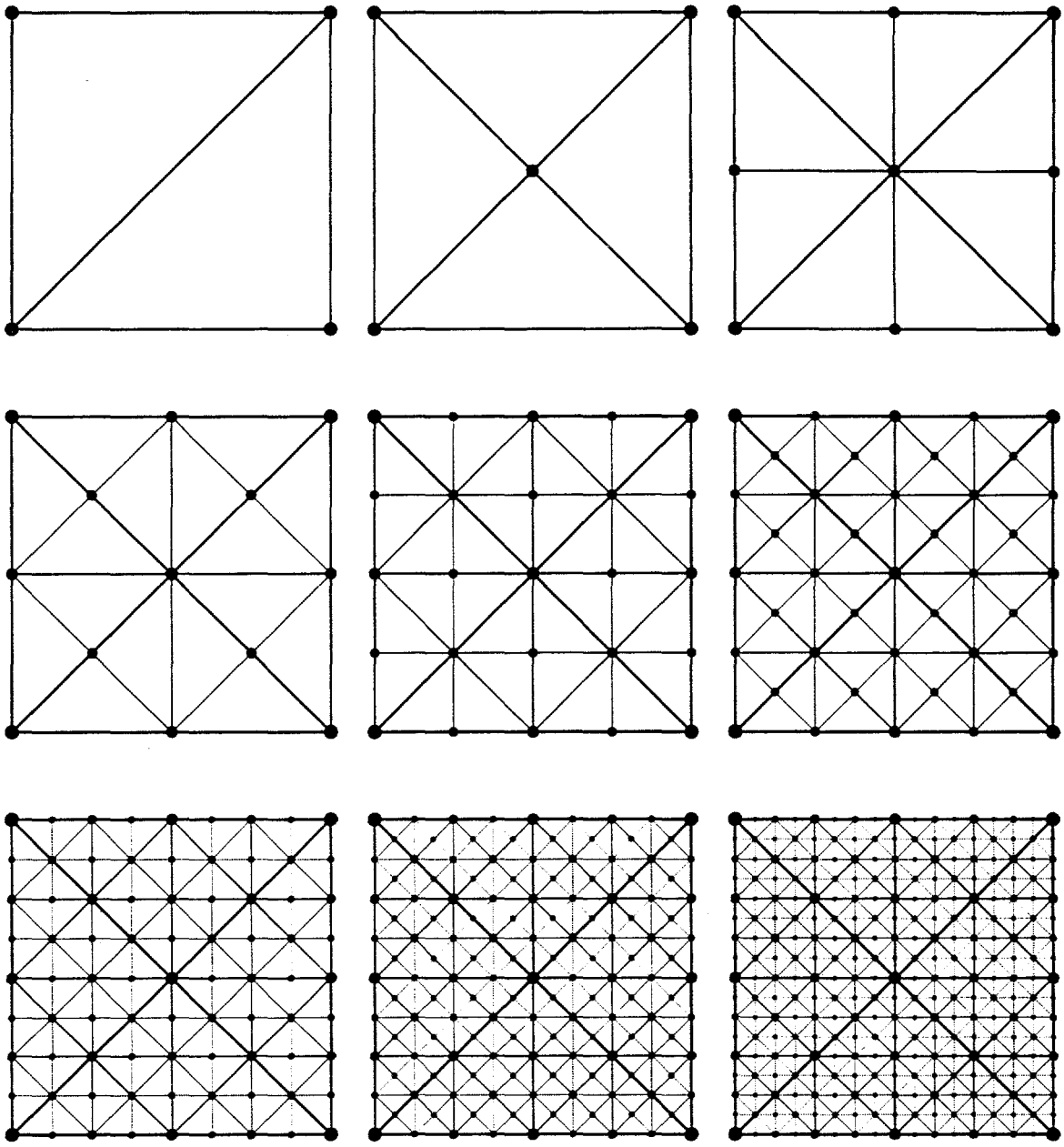

FIGURE 2. Levels 1 through 9 of the binary triangle tree. Top to bottom: triangles generated by recursive subdivision of a pair of right isosceles triangles. (Same figure, bottom to top: sub-sampling a height field to form a hierarchy of coarser approximations.)

1. As described below.

4.2 Triangle meshes

Now assemble these triangles into continuous meshes. Every point in the domain of a continuous triangle mesh must have a unique elevation. Thus a mesh is a set of triangles which collectively meet two conditions: each point in the domain must be covered by a triangle (the mesh must be *hole-free*) and no point can receive more than one elevation (the mesh must be *single-valued*). Single-valuedness requires that any two triangles in a continuous mesh are either disjoint or they share an edge. If the shared edge of two adjoining triangles is

1. a hypotenuse for both or for neither of the triangles, the triangles are at the same level,
2. a hypotenuse for the first triangle but not the second, then the first triangle is one level finer than the second ($l_1 = l_2 + 1$).

Therefore, in a continuous mesh, adjoining triangles can differ by at most one level of resolution.

The root triangle is the coarsest continuous triangle mesh. Every other continuous mesh is derived by repetitively subdividing triangles in coarser continuous meshes. This technique automatically maintains the hole-free property but some additional care is required to maintain single-valuedness. Suppose triangle a in the continuous mesh M is subdivided[1]. That is, M is replaced with a new continuous mesh M_a which does not contain a but which contains a's children. In order to preserve continuity when subdividing a it is necessary to consider the triangle b which shares a's hypotenuse. If the shared edge is a hypotenuse for b, then the mesh M_a is simply M with both a and b removed and their (total of) four children added. We will call this operation a *simple split*. Two triangles which share a hypotenuse are *opposing* triangles. The figure formed by two opposing triangles is a *diamond*. Diamonds can be *simply split* without disturbing the continuity of the surrounding mesh M.

Things can be more complicated. The edge shared by a and b may not be the hypotenuse of b. Maintaining continuity in this case requires that b must be split to form the diamond

1. If a is in M then neither the parent of a nor any of a's descendants can be in M: otherwise M would not be single-valued.

containing a before that diamond can be *simply split*. Splitting b may result in a cascade of further splits.

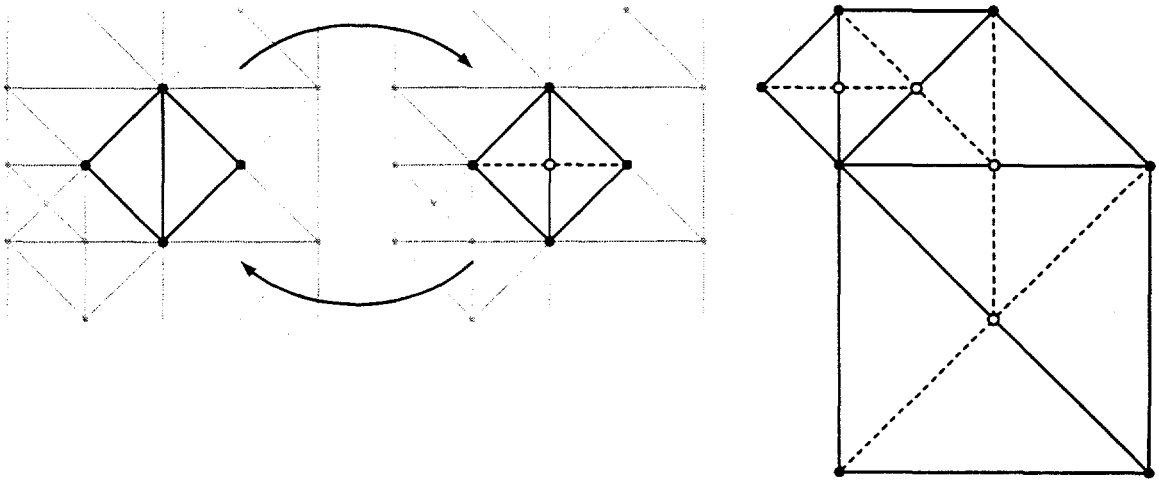

FIGURE 3. Splits and merges: a cascade of induced splits is shown at right.

Let us introduce some possibly baroque terminology. Triangle b which shares the hypotenuse of a in a mesh M will be called the *antithetical*[1] triangle of a in M. Triangle b is either the opposing triangle of a, or it is one level coarser than a. In the latter case the triangle antithetical to b is a new triangle c[2]. In order to split triangle a in mesh M, it is necessary to split a's opposing triangle, which may be missing from M. If so, the opposing triangle must be produced by first splitting a's antithetical triangle. The procedure to *split a* in *M* is presented below.

```
1. determine o = opposed (a)
2. determine b = antithetical (a, M)
3. if o ≠ b split (b, M)
4. perform simple split (a, o) on M.
```

FIGURE 4. Pseudo-code to replace a continuous mesh containing triangle *a* with a continuous mesh with triangle *a* split: *split (a, M)*

The set of triangles produced by *split* depends on the structure of the mesh M when *split* is called. However this lack of uniqueness is less than meets the eye. The only possibility is that *split* may return early: some of the triangles that *split* would otherwise produce may already be present in M. The important fact is that there is a unique set of triangles which must be split in order to maintain continuity and these triangles will either already be split

1. The use of the awkward term *antithetical* is motivated by an irresistible analogy of the splitting process described here to the process underlying Hegelian dialectics. Note that if the antithetical of a is not the opposing triangle of a, then by the recursive subdivision procedure there is only one other possible triangle.
2. Therefore antithetical is not a symmetric relation. See Engels, *What is Dialectical Materialism?*. (Citing Engels here somehow seems to help balance this paper which everywhere else extols the capitalistic virtues of greed and maximal exploitation of resources.)

in M at the outset or *split* will generate them. This property of our meshes, *subdivision determinacy*, is needed for our proof of the optimality of the run-time algorithm. Subdivision determinacy does not imply that a general split can be undone: in fact it is impossible in general to recover M from a and M_a. Hence there is no inverse for a general split.

However *simple splits* are invertible: their inverses are *merges*. (Figure 3.) Merges form a diamond from the four children involved; they automatically maintain continuity and therefore generate no cascades.

All continuous meshes built using the triangles at hand can be formed by starting with the root triangle (a continuous mesh) and repetitively selecting at will any triangle in the current mesh as argument to *split*. In pseudo-code, we have the following

```
1. Initialize M to be the set containing the root triangle.
2. While not done
       Choose a triangle t in M.
       Use split to replace M with the set M_t
```

FIGURE 5. Pseudo-code to generate an arbitrary continuous mesh

Our greedy terrain generation algorithm is a simple modification of the above. Except for the discussion of how elevations are assigned to vertices (below) our discussion of the meshes of interest is now complete. We next now turn our attention to the computation of error bounds.

5 Error metrics

Determination of whether a coarse terrain triangle is acceptable as-is or whether it must be subdivided is guided by the error associated with that triangle. The error we are most concerned with is the *total distortion* as defined earlier. The total distortion of a coarse triangle is the maximum error in screen pixels which results from replacing the exact terrain in the domain of the triangle with the coarse terrain. This error is far too expensive to calculate in real-time and we content ourselves with a bound on that error. Any bound is reasonable provided it is (a) really a bound, (b) easy to compute and (c) adequately "tight."

The exact distortion at a given point in the domain of the terrain is the projection of the distance between the exact terrain height and the approximate terrain height at that point. onto the viewing screen. A reasonably general method for calculating such a bound begins with constructing a volume in world space containing both the coarse triangle and the exact terrain for the domain of the triangle. Then it is clear that the real space maximum error of the approximation is bounded by the maximum thickness of the containing volume. However we are not interested in the real space error but in the screen space error[1] of its projection. The maximum error of the projection is not the projection of the maximum thickness. Therefore we need to be able to find the maximum error in screen space for all "thicknesses" of the containing object. It is convenient then to choose a containing volume

1. The standard proof of this fact is to take the viewpoint directly above at high altitude. Then the height of a point below is irrelevant to where it is located on the screen.

with constant thickness, equal to the maximum thickness realized for any point of the coarse terrain triangle. Such a volume is called a "wedgie" and is shown in Figure 6[1].

To be fully explicit a wedgie is a volume bounded by a surface comprised of two perpendicularly displaced triangles and the three rectangles which connect the corresponding edges. The center triangle, representing the coarse terrain approximation is the triangle "half-way between" the two bounding triangles. Thus at each point of the center triangle the true terrain may be on either side of the center but at most a half-thickness away.

It is easy to show that the maximum screen-space error of a wedgie necessarily occurs along one of the triangle's sides[2]. It is not obvious, but nor is it difficult to show that in general the maximum screen-space error does not occur at a vertex[3]. Nonetheless it is possible to show that the errors incurred by assuming that the maximum displacements occur at vertices can be neglected for all reasonable triangles[4]. Therefore one can compute the screen-space distortion by projecting the two wedgie half-spans at each vertex and selecting the maximum value. This bound will be called the wedgie bound

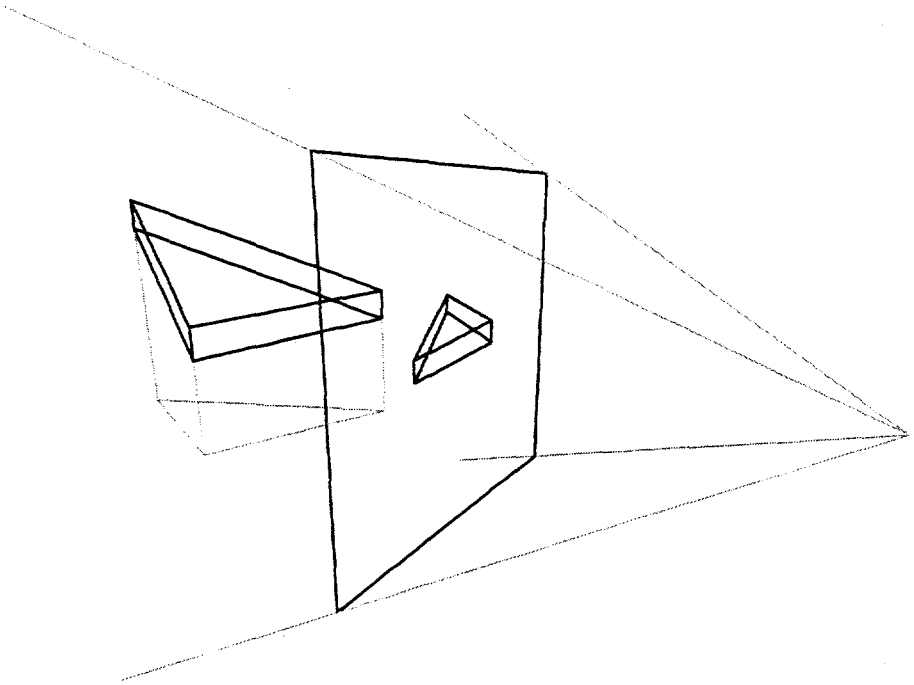

FIGURE 6. Error bound computation using "wedgies".

1. For simplicity we take the thicknesses to be perpendicular to the "base triangle" of the wedgie.
2. This follows because the screen-space distortion is convex in the directions transverse to the viewing direction (standardly called the z-axis, but not to be confused with the world-space z axis).
3. This follows because the distortion is not convex in general in the viewing direction.
4. And, if one insists on being unreasonable, the triangles can be thickened so that the error computed at a vertex is still a good bound. Therefore it is always possible to assume that the maximum error always occurs at a vertex.

All of the distortion-based error metrics (or error bounds) of interest here are L_∞-based norms. That is, each triangle in a mesh is assigned an error and the total error for the mesh is the maximum error obtained for any triangle in the mesh. If we impose a further, natural, monotonicity condition on allowable error metrics, namely that subdividing a triangle can not produce triangles with higher errors, we will be able to prove the optimality of our algorithms. Let us call monotonic L_∞ error metrics, *refinable* error metrics. Unfortunately, the wedgie bound is not refinable: it is possible that the children of triangles have larger errors than their parents. However, the wedgie bound is generally monotonic and it can be artificially modified to be exactly refinable if desired.

6 Pre-processing tasks

Pre-processing is performed for one primary purpose: to make the run-time operations efficient. There is essentially one major run-time operation: the subdivision and assembly of pre-existing triangles into continuous meshes which minimize error metrics. Therefore pre-processing support is required to (a) construct the triangles which will be assembled into meshes and (b) expedite the computation of the error metric which determines which triangles are subdivided.

6.1 Triangle pre-construction by sub-sampling

Pre-processing is best understood from the bottom up. It is convenient to start with a square grid of terrain points with $n = 2^m + 1$ points on a side. This implies that there are $2^m 2^m = 2^{2m}$ square cells in the grid, each of which contains 2 triangles for a total triangle count of $2^{2m+1} = 2^l$ triangles. Therefore, from our earlier discussion this grid will reside at level l and we can construct l levels of coarser resolution "above" it. Figure 2, when read from the bottom up, presents an example with $n = 9$ so $m = 3$ and $l = 7$.

Each point in the initial grid has an elevation in addition to its x-y coordinates. Thus the finest resolution (level l) triangles are fully specified. The triangles at level l-1 are obtained by merging adjoining triangles in level l as shown. The vertices of these merged triangles project directly onto grid points so that their elevations are obtained simply by *subsampling* the original mesh.

6.2 "Wedgie" thickness

The triangle real-space normal vector and wedgie thicknesses are also assigned during pre-computation. There is no difficulty with either computation.

6.3 Greedy is optimal

Any method which provides a continuous mesh containing a prescribed number of triangles which minimizes a specified error metric in a fixed class of continuous meshes is an

optimal method. For subdivision determinate meshes and the refinable error metrics introduced earlier the greedy algorithm below is optimal.

Begin by initializing the current continuous mesh M to the root triangle. While M contains fewer triangles than desired, choose the triangle t in M with the highest error value and refine M by splitting t. This is clearly a greedy algorithm: it makes the locally optimal choice at each stage. This algorithm is elaborated in the pseudo-code below.

```
1. Initialize M to be the set containing the root triangle.
2. While M contains fewer than n triangles
       a. Choose triangle t in M with greatest error value
       b. Use split to replace M with the set M_t
```

FIGURE 7. Pseudo-code for greedy algorithm

It may be obvious that this greedy method is optimal. If not, the following argument may help. Consider the set T of all triangles chosen by the greedy algorithm in step 2a. Two facts are important: first, the remaining splits are uniquely determined by T[1] and second, the optimal method must also split all of the triangles in T[2]. Together these two facts imply that the optimal method must split every triangle that the greedy method splits. Imposing the constraint that the total number of triangles in the final meshes is n for both methods implies that the greedy method and the optimal method produce exactly the same mesh since it rules out the possibility that the optimal method performs some additional splits.

There are two minor quibbles with the foregoing argument. First, at any given step the possibility of more than one triangle with maximum error value is harmless: at the worst it means that there may be differences in the meshes produced by the greedy method and some other optimal method, but even so any mesh produced by the greedy method will still have the same error value as the optimal meshes. The remaining quibble is more serious theoretically, but not in practice: the number of triangles in the mesh produced by the greedy algorithm may exceed the specified number n due to the effects of cascades induced by the last desired split. In practice this excess is small and amounts to a few triangles at most. We have chosen to tolerate such slight overruns. If the constraint on triangle count n is regarded as inviolate, then one need only undo the last split. This will give a mesh with fewer than n triangles. One could choose to split other triangles with the hope of finding a set of splits which would produce a final mesh of exactly n triangles. These gratuitous splits will not reduce the error of the final mesh: the final error value will remain that of the triangle whose split was undone.

The fact that greedy is optimal for refinable error metrics is nice theoretically. However, the greedy method is (a) optimal for more than just refinable error metrics and (b) has its

1. This follows from the subdivision determinacy of our meshes.
2. Suppose to the contrary that there exists a triangle t in the set T which is not split by the optimal method. Then the mesh produced by the optimal method under any refinable metric will have an error no less than *error (t)*, whereas the greedy method will have a lower error. This contradicts the definition of the optimal method.

own attractiveness and may be suitable for use even for error metrics where it is sub-optimal.

6.4 Implementation: the basic queue algorithm

The exposition thus far has been perhaps oriented more to the theoretical than the practical. Our implementation is logically equivalent to the greedy algorithm as presented above but it differs in detail for reasons of efficiency. It is perhaps obvious that the mesh M should be maintained as a priority queue of triangles (which is called the *split queue*, Q) with priority given by the error metric of interest. This makes the choice of triangle to be split in step 2a and its removal from the queue very inexpensive. It also means that adding elements to M during the split process become ordinary priority queue insertions into Q. Removal of the remaining triangles from Q (those subdivided to maintain continuity) require an additional operation beyond the mandatory priority queue operations: namely the removal of queue elements which are not at the top of the queue. The basic queue algorithm, performed on each frame has the following structure.

```
1. clear queue
2. insert root triangle into queue
3. while termination criteria not met
4.     t = remove highest priority triangle
5.     split t
6.     insert visible split triangles into queue
```

FIGURE 8. Pseudo-code for queue implementation of greedy algorithm

The termination criteria typically involve a maximum number of desired triangles and possibly a specified level of tolerable error. Here the split procedure adds triangles to the queue by priority queue insertion (which requires a computation of their priority) but also removes triangles from the queue to maintain continuity.

In this form, however, the queue algorithm is too slow for real-time use. It requires that the queue be computed from scratch on every frame. This involves $O(n)$ expensive priority and visibility calculations. On the other hand, experience with the applications of interest show that most of the mesh is constant from one frame to the next. Therefore, if we could exploit frame-to-frame coherence, the cost per frame would drop to a much smaller number of order $O(\Delta n)$where $\Delta n \ll n$ is the number of triangles which actually change from one frame to the next. The changes needed to make the queue algorithm incremental are described next and they make the method practical for real-time use.

7 Incremental split and merge

Major improvements in performance are realized when the basic triangle processing algorithm is made incremental. Rather than

Two queues are required: the first is termed the *split* queue and is identical in operation to the queue in the basic algorithm; the second queue is the *merge* queue and its operation is slightly less intuitive.

```
1. clear queue
2. insert root triangle into queue
3. while termination criteria not met
4.     t = remove highest priority triangle
5.     split t
6.     insert split triangles into queue
```

FIGURE 9. Pseudo-code for real-time incremental identification of set of triangles to be rendered

8 Implementation details and extensions

8.1 Formatting the output for hardware.

For optimum performance the mesh triangles are rendered as GL_TRIANGLE_STRIPS under Open GL. Figure??? shows a uniform level of the mesh that has been split into these GL_TRIANGLE_STRIPS strips. Each strip is outlined by a solid bold line. To output the strip to the graphics hardware three vertices of the first triangle are given to the hardware are given to the hardware with glVertex calls. Each triangle after that till the end of the strip requires only one additional vertex. We believe the this figure portrays the optimum organization of strips. It results in 8 triangles being included in each strip.

The mesh for any particular frame in a simulation will differ considerably from this picture with triangles from many different levels coexisting next to each other in the mesh. Optimal generation of the triangle strips becomes problematic. The simplest algorithm, which produces pretty good results is to pick a triangle randomly from all the triangles in the mesh. Starting at this triangle we look for one or two triangles that are visible, are not already represented in a strip and share two vertices with our starting triangle. If two triangles are found the starting triangle will be in the middle of a strip. For each of these new triangles found we extend the strip further by finding triangles that can be rendered by adding one vertex to the last triangle added. This stripping results in about 5 triangles per strip. Recreating the strips for each frame this way requires considerable CPU time.

To speed up this process the group of GL_TRIANGLE_STRIPS strips is kept at the end of each frame. As the algorithm splits and merges triangles on the mesh the algorithm attempts to add and subtract triangles from appropriate strips in such a way as to keep the average number of triangles per strip as high as possible.

To do this whenever a triangle is split or merged its representation in the strips is removed. If it is the first or last triangle on its strip the vertex is simply removed from the strip. If it is in the middle of the strip the strip is split appropriately into two strips.

The split or merge results in new triangles added to the mesh which must be added to appropriate strips. To add a triangle the algorithm looks at the tstrips that contain the three neighboring triangles on the mesh. If the triangle can extend the strips for two of these

neighboring triangles, (at the beginning or end of the strip), it adds the appropriate vertex to one of the two strips and then merges the resulting strip with the strip for the second neighbor.

This operation may require reversing the drawing order of one of the strips. If this is not possible the algorithm tries to add extend the strip for one the neighboring triangles. If no neighboring strip is suitable a new strip of three vertices is created. After the triangles have been added for each split or merge the algorithm looks at triangles that where neighbors of the original split or merged triangle to see if the strip for that triangle can be merged with strips for any of its neighbors. This results in a average triangle count of about 4 triangles per strip with almost negligible addition in CPU time beyond th

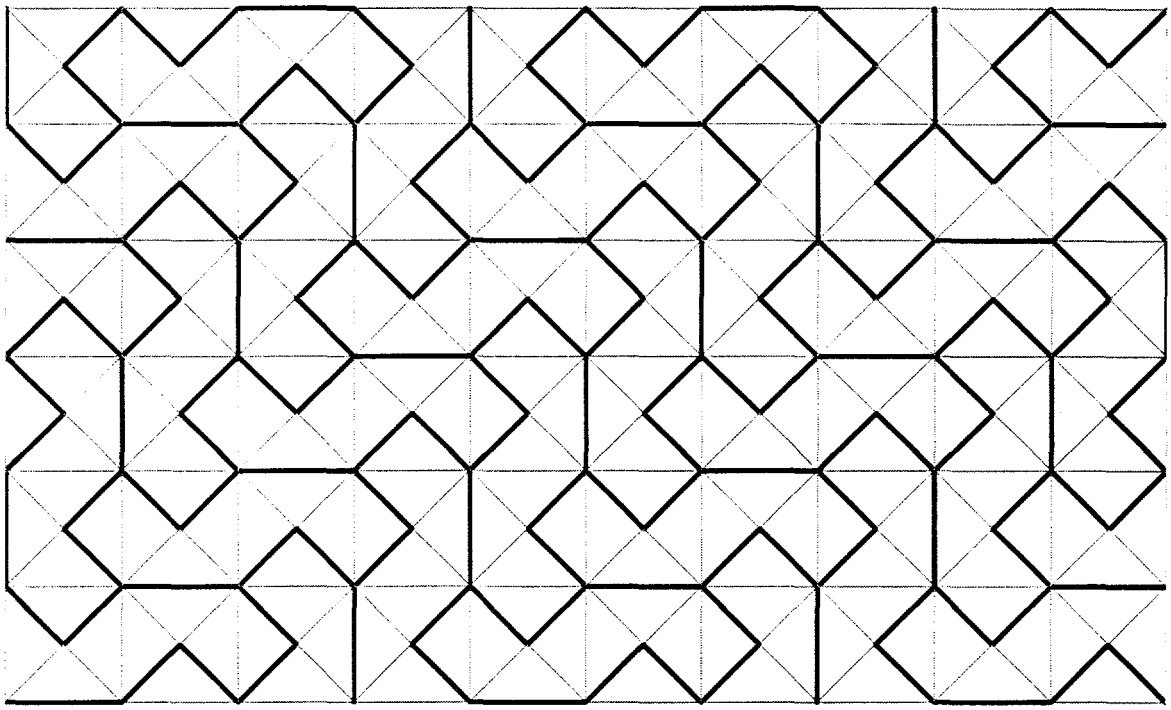

FIGURE 10. A tiling of the plane by triangle-strips of 8 uniformly-sized triangles

8.2 Priority Queueing

When the viewing position changes from frame to frame in the simulation the projected error of each triangle will change as its distance and orientation changes with respect to the viewing position. As triangles approach the viewing position their projected errors and therefore queueing priorities will tend to increase, resulting in the eventual splitting of the triangle as it approaches. The simplest approach is to calculate the priority for each triangle on the split and merge queues for every time step.

To speed up this calculation all of the triangles on the split and merge queue are put on to either of two lists, an immediate list and a deferred list. Triangles on the two immediate lists have their priorities calculated every frame. Triangles on the deferred list are calcu-

lated in batches, a few per frame, in such a way that all triangles in the deferred list are guaranteed to be calculated every ten time steps.

To determine which list each triangle belongs to the projected error for the triangle is estimated ten frames in advance using the current velocity of the viewing position and assuming the triangle is advancing directly towards the viewing position. This results in a conservative estimate of the priority (overestimate). If the estimated error indicates, (it is greater than an estimated threshold), that the triangle may be split within ten time steps it is put on the immediate list for the split queue. If the error is less than a threshold for triangles on the merge queue it is put on the immediate list for the split queue. Otherwise it is put on its respective deferred list.

The thresholds for the split and merge queues are calculated for each frame and vary slowly from frame to frame. They are set by requiring that approximately 5 percent of the triangles on either the split or the merge queue have priorities that are between the calculated threshold and the maximum priority on the split queue (minimum priority on the merge queue).

This incremental algorithm results in an almost order magnitude decrease in the CPU time needed to recalculate queue priorities for each frame.

8.3 Volume Culling

For each frame all the triangles in the terrain tree are queried to determine if the triangle should be drawn by the graphics hardware. This is determined by comparing the extents of each triangle to the six clipping planes that determine the viewing frustum. Triangles are classified as OUT if they are wholly outside one of the clipping planes or IN if they are inside all of the clipping planes or partially visible. If a triangle is outside one of the clipping planes all its children will also be outside the same clipping plane. If a triangle is marked IN all its children will be marked IN. This allows the classification of child triangles to proceed more rapidly.

Triangles are labeled by recursively descending the terrain tree from the original root triangle. If during this labeling process the triangle's classification does not change from the last frame and it is either marked IN or OUT its children's classification will not change either and the algorithm does not further classify any of the triangles children.

Whenever the classification of a triangle changes as it comes into or goes out of the viewing frustum its queueing priority is recalculating and its triangle is added or removed from the list of GL_TRIANGLE_STRIPS if the output mesh is generated incrementally.

8.4 Performance.

Performance figures where measured on a Indigo2 Silicon Graphics Workstation with Maximum Impact graphics hardware and a single R10000 Processor. These figures where obtained simulating a fighter aircraft in terrain avoidance mode flying at high speed over very hilly terrain. With all incremental features of the algorithm turned on and the number

of triangles rendered set to 3000 the total time to render one frame is approximately 37 milliseconds. Of this approximately 6 milliseconds is spent doing the volume culling, 5 milliseconds calculating the queue priorities, 6 milliseconds splitting / merging the triangles in the mesh and 20 milliseconds outputting the generated GL_TRIANGLE_STRIPS to the hardware.

Turning off the incremental requeueing results in recalculating the queue priorities for every triangle in the terrain tree (approximately 6000) for every frame. This increases the time calculating the queue priorities to about 43 milliseconds resulting in a total time to render one frame of 80 milliseconds.

Turning off generating the GL_TRIANGLE_STRIPS incrementally further increases the rendering time to 150 milliseconds. Using a single R10000 processor on a Silicon Graphics Reality Engine with a Infinite Reality graphics board decreases the total rendering time to less than the 33 milliseconds required to render 3000 triangles at a 30 Hertz rate. Triangle counts of between 4000 and 5000 can be rendered on the Reality Engine with all the incremental features turned on at a rate of 30 frames per second.

8.5 Acquiring low cost exact lines-of-sight by modifying the error metric

There are two costs associated with providing exact lines-of-sight in real-time. The first cost is that of determining whether a clear line-of-sight is present between two points or not using the exact terrain. The second cost is that of expending the triangles necessary to assure that the rendered scene is consistent with the exact answer. Here we are more concerned with the second cost. Our present method is simple and works well but it is far from optimal. We simply set the priority of any triangle which intersects a line-of-sight of interest to the maximum priority. This assures that the terrain will be resolved to the finest level of resolution in the neighborhood of all such intersections. Much better approximations are possible[1], but this simple method is already better than many other methods in current use: in particular, it does not require that the terrain is rendered at the finest resolution at all points between the two points whose ability to view each other is in question.

Since we produce scenes with constant triangle counts, it is necessary to steal triangles from lower-priority areas in the scene to place them where needed. In fact very few polygons must be stolen and the changes in scenes required to construct scenes with the correct lines of sight are generally subtle. (Figures 12 through 15.)

The ability to get these lines of sight exact allows us to guarantee that two independent simulations which share the same finest level database will get consistent answers to any line-of-sight questions. Their answers will be consistent because their answers will be correct. Although this is merely a special case of assuring consistency (because different simulations will not generally possess identical finest resolution databases) it is still significant: for one thing the problem it represents actually occurs (and is, in fact, one of

1. And compatible with our approach.

the key problems which confront us), and for another, the general problem of assuring consistency is so difficult that any result is of interest.

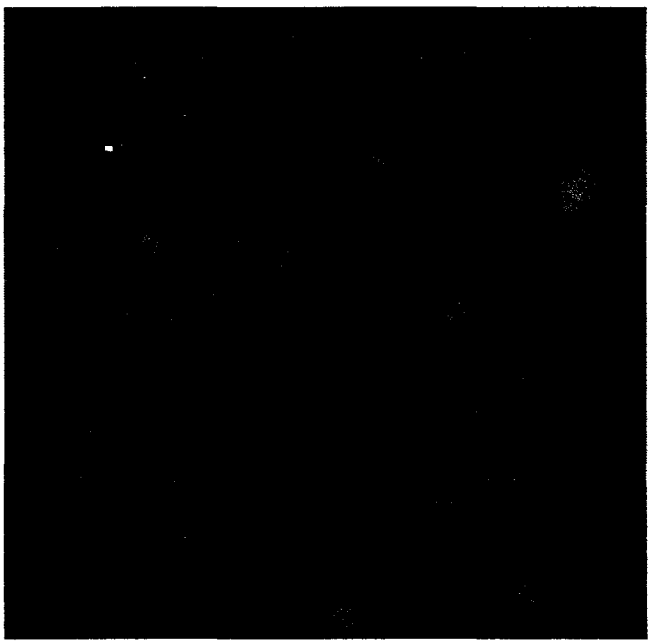

FIGURE 11. Wide view without target line-of-sight correction

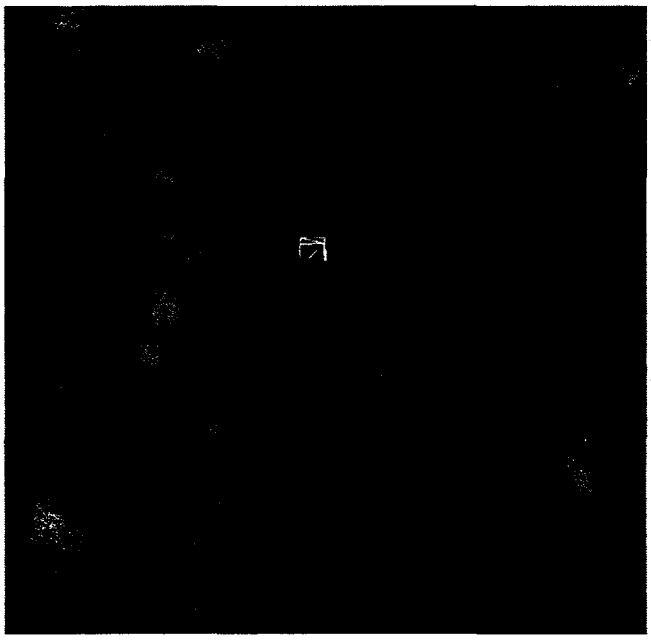

FIGURE 12. Magnified detail without target line-of-sight correction

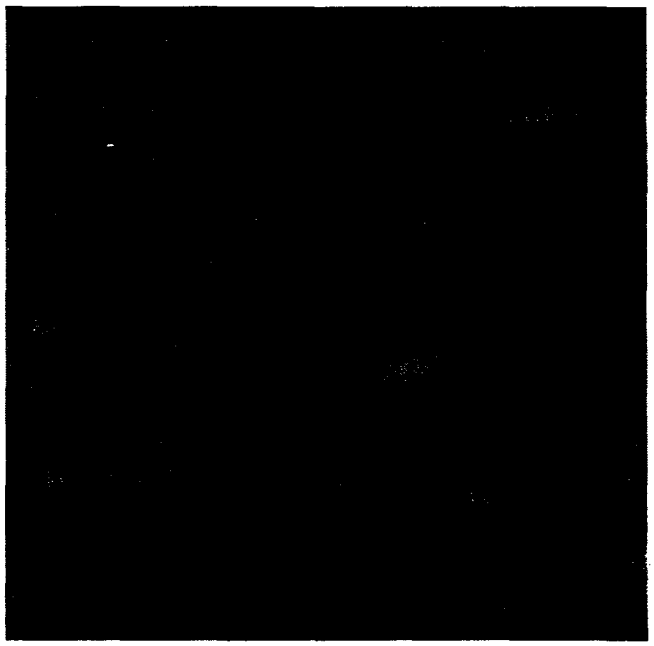

FIGURE 13. Wide view showing target line-of-sight correction

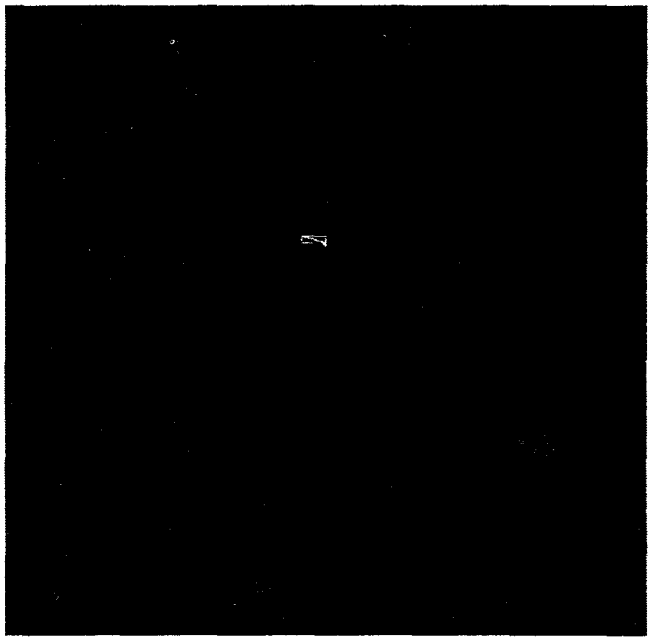

FIGURE 14. Magnified detail showing target line-of-sight correction

9 Future Work

Many areas remain for further work. Dynamic terrain and terrain appearances have not been treated here. However of most immediate interest is adding paging to our algorithms. Until this is done the good scalability properties of our algorithms are "virtual" in the pejorative sense.

Consider texture paging first. The problem lies in the finite texture memory and finite bandwidth to that memory, and a huge amount of texture (e.g. 800km x 800km at 1 meter post spacing). However, one can show that the texture detail need not be better than about one texel per pixel after projection. So a possible approach is the following. "Quadtree up" the texture (say 128x128 texel block per quadtree node) doing fitting to coarsen resolution at preprocess time. Now use an incremental priority-queue algorithm similar to the geometry algorithm to build a dynamic quadtree of active texture blocks. The priority should be the maximum projected texel size for the block over all possible view directions at the current eyepoint. Only a couple of splits/merges may be possible per frame, but this should be fine for achieving the one texel per pixel limit. InfiniteReality fast texture loading is needed for this to work, since some texture loading must be performed sometime and this must not affect the frame rate. There are two layers here, and possibly a third. The two layers are compressed blocks on disk and active blocks in texture memory. Compressed blocks in memory constitute a desirable third layer since decompression is much faster than disk I/O and compressed blocks are easily 15 times smaller than active blocks. Error metrics might be added to the priority. For example:

1. give higher priority to blocks where the bilinearly-interpolated block has large differences with high resolution data.
2. give high priority to large blocks that have small neighbors (since this causes visual discontinuities at the boundary)
3. give higher priority based on where the eye is likely to move

Geometry paging is similar to texture paging, except the priorities are different and blocks can be better stored in a non-uniform way.

10 Conclusions

The algorithm presented here are simple and work well in practice. The terrain generation algorithm operates in (interactive) real-time and works with a fixed-size triangle budget arbitrarily specified at run-time. It produces high quality approximations of terrain using surprisingly few triangles. The degree of accuracy attained by the algorithm is known in real-time: it equals the priority of the first unsplit triangle. The algorithm is capable of handling a variety of error metrics. This generality is important in practice and permits, for example, the production of low cost exact lines-of-sight. Our algorithm is portable across a variety of platforms and, once paging has been implemented, it will be capable of confronting rendering problems of arbitrary scale.

Acknowledgments

This work was performed on DOE Contract W7405-ENG-36 which is funded jointly by the Navy and Air Force. We would like to thank James Reus for his assistance in the early stages of this work. We would also like to thank our sponsors, who have provided substantial input and feedback during the course of interesting project: in particular, we would like to thank Frank Griffo, Ted Wilson, Lenart Clark, Don Jackson, Dr. Shah Mahmood, David Purdue, Tom Joyner, Pamela Stewart and David Blake, all of the Air Combat Environment Test and Evaluation Facility at the Naval Air Warfare Center, Patuxent, River, MD; and Keem Thiem and Dr. John Fong of the Electronic Combat Integrated Test Program at the Air Force Flight Test Center, Edwards Air Force Base, CA. We would also like to thank Ken Lee, Robert Webster, Ron Pistone, Lynn Cline and Fatima Woody, all of Los Alamos National Laboratory for valuable assistance during the course of this work. Finally we thank the staff of the Advanced Computing Laboratory at Los Alamos National Laboratory for sharing their expertise with us and for the generous use of their equipment.

References

Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms*, Cambridge, MA: The MIT Press, 1990.

Lindstrom, Peter, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust and Gregory A. Turner. "Real-Time, Continuous Level of Detail Rendering of Height Fields"

Miller, Mark C. Multiscale Compression of Digital Terrain Data to meet Real Time Rendering Rate Constraints. Davis, CA: University of California at Davis Ph.D. Dissertation, 1995.