Research Article

Diamond Terrain Algorithm: Continuous Levels of Detail for Height Fields

H Hakl[a] L van Zijl[b]

[a]*University of Stellenbosch , Stellenbosch, South Africa,* henri@cs.sun.ac.za
[b]*University of Stellenbosch , Stellenbosch, South Africa,* lynette@cs.sun.ac.za

Abstract

The generation of terrain meshes for real-time rendering of landscape data has a wide field of application ranging from military simulations and geographic information systems to flight testing and computer games. Duchaineau et al. presented a successful approach – real-time optimally adapting meshes (ROAM) – for terrain triangulation. However, the choice of geometric substructure in ROAM does not lend itself well to vertex optimizations such as triangle-strips. We present an alternative mesh representation for ROAM based on a triangle quadtree that naturally decomposes to triangle-strips. Additionally we present a new algorithm for the generation of continuous levels of detail (CLOD) for height fields that utilizes our proposed mesh representation. The algorithm is similar to ROAM in that it supports split and merge operations, but we ignore priority queueing in favor of four LIFO queues that support constant time insertion and deletion.

Keywords: *Computer graphics, terrain rendering*
Computing Review Categories: *I.3.3, I.3.5, I.3.6, I.3.7*

1 Introduction

Visualization of terrain data enjoys a wide field of applications ranging from military and industrial simulations and testing to commercial applications such as geographic information systems and computer games. Many of these applications require real-time dynamic interaction from end-users and thus are required to rapidly process terrain data to adapt to user input.

Typically a terrain data set contains more data than can be processed in a brute force manner at interactive rendering rates, in spite of recent advances in graphics processing hardware. A primary concern to meet application requirements is thus to find ways to reduce processing time of the terrain data, usually by efficiently reducing the data set to a sufficiently small subset that approximates the landscape and can be rendered at interactive speeds.

In this paper we present a new algorithm to perform continuous levels of detail (CLOD) tesselation in height fields. To this end we first present prior and related work in terrain rendering. We then consider the ROAM [4] algorithm and discuss a number of its advantages, disadvantages and features, including the choice of geometric substructure in ROAM. In section 4 we present an alternative geometric substructure and will argue for the advantages of this mesh representation over the one typically used in ROAM. Lastly, we present a new ROAM-like algorithm, dubbed Diamond, that makes use of our proposed geometric substructure.

In the rest of this paper, we assume that the reader has a working knowledge of terrain rendering.

2 Related work

Several approaches to efficiently handle terrain sets have been published or feature regularly in terrain rendering implementations: Quadtrees are perhaps the most influential data structure in terrain algorithms; approaches using potential visibility sets or block-based levels of detail have been presented; and triangular irregular networks (TINs) are popular in continuous levels of detail algorithms.

Quadtrees [15] are well suited to spatially organize two-dimensional height field data and thus feature regularly in terrain tessellation algorithms. The hierarchical nature of quadtrees additionally lends itself well to view frustum culling and quadtrees are therefore regularly used on top of terrain algorithms to serve as a view culling mechanism.

Stewart [14] describes the implementation of a hierarchical potential visibility set for terrain that is stored in an implicit quadtree and can be used to efficiently cull large regions of occluded terrain. This approach works well in conjunction with other algorithms which are based on level of detail (LOD) computations.

De Boer [1] describes a hardware-friendly algorithm that uses block-based level of detail sets called GeoMipMaps. He suggests an improvement to the basic algorithm to reduce vertex popping which takes the form of *trilinear GeoMipMapping*, which morphs one block LOD to another; essentially this transforms the block-based levels of detail algorithm to a continuous levels of detail algorithm.

Several classic approaches, as well as the Diamond algorithm presented in this paper, utilize triangulated irregular networks (TINs) to perform terrain rendering. TINs are networks of triangles where all triangles share a similar shape but triangles may vary in size relative to others. Lindstrom *et al.* [10] describe an algorithm that achieves continuous levels of detail in a regular grid using a dynamically changing quadtree and a bottom-up strategy. Röttger *et al.* [12] conversely refine the work by Lindstrom and offers an algorithm that yields similar results using an implicit quadtree that is traversed in a top-down manner. Duchaineau *et al.* [4] offer the ROAM algorithm that maintains a continuous levels of detail terrain triangulation using split and merge vertex operations. We examine the ROAM algorithm more closely in the following section.

3 The ROAM algorithm

In this section we briefly review the ROAM algorithm and consider some of its advantages and disadvantages, paying particular attention to the choice of geometric substructure in ROAM.

3.1 Mesh representation and operations

The basis of the ROAM algorithm is its choice of mesh representation: A triangle bintree (or bintritree) which is the triangular equivalent of a rectangular bintree [3]. The root triangle in ROAM is chosen to be right-isosceles, which ensures that all child triangles are right-isosceles too. The following figure demonstrates the nature of the geometric substructure.

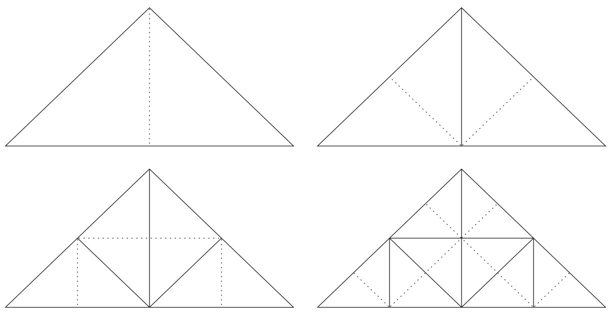

Figure 1: Various levels of refinement for a triangle bintree

Two vertex operations are defined over the mesh representation, namely split and merge. A split operation applies to a triangle that does not possess a base neighbor, or to a pair of triangles that are each other's base neighbors[1]. In each case the split operation replaces the original triangles with the children of the triangles. A merge operation is the inverse of a split operation. Figure 2 below demonstrates split and merge operations.

[1]The base of a triangle is defined as the edge opposite the apex (right-angled) vertex. Two triangles that are adjacent so that their adjacent bases form a straight line, are called base neighbors.

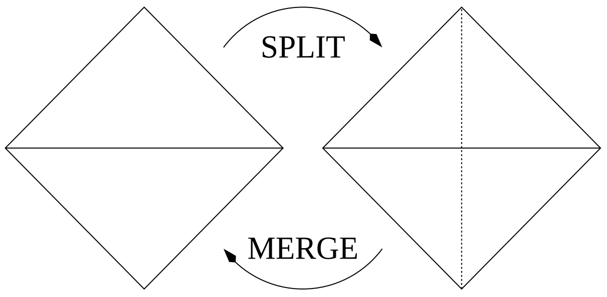

Figure 2: Representation of split and merge operations

A triangle within a triangulation may not be immediately splitable if it possesses a coarser base neighbor. In such an event the base neighbor must be forced to split first, which may in turn require recursive splits, as depicted below.

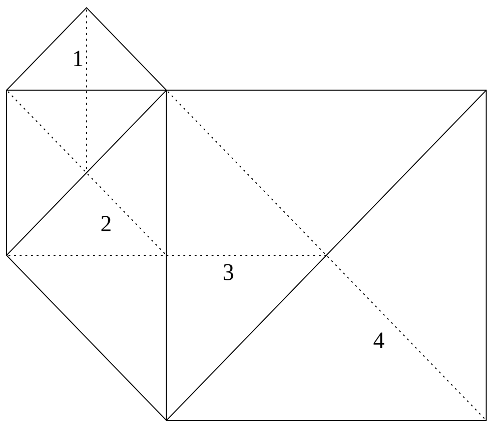

Figure 3: Example of recursively forced splits, to perform split 1, splits 4, 3 and 2 have to be performed first

3.2 Priority metrics

A metric is used to determine which triangles are split and which are merged, therefore the choice of metric directly determines the resultant mesh of terrain triangulation. Duchaineau [4] lists a number of possible metrics that can be used in a terrain algorithm.

Both Duchaineau [4] and Röttger [12], as well as our implementation, make use of a variance metric that is of the form:

$$priority = \frac{terrain\ variance}{distance}$$

The priority calculated determines whether a triangle must be split or merged. The *terrain variance* is a measure of terrain complexity – flat surfaces possess a low variance, whereas sudden changes in terrain slope possess a high variance. The *distance* is the distance from the point of view to the target triangle. The above metric therefore computes a generally decreasing priority with distance, but higher priority for high variance objects and lower priority for low variance objects in the terrain.

3.3 Queueing and the ROAM algorithm

The ROAM algorithm is driven by two priority queues: A split queue and a merge queue. The split queue contains the sorted priorities of all splitable triangles, while the merge queue contains the sorted priorities of all mergable triangles.

A simplified rendition of the ROAM algorithm reads as follows:

```
while terrain triangulation is not of
  target size/accuracy do
  if triangulation too large then
    merge lowest priority triangles
    in merge queue
  else
    split highest priority triangles in
    split queue
```

If the terrain triangulation is reset for each frame to contain only the bintritree root, then only split operations are performed, resulting in a split-only implementation. Such an implementation can take advantage of lower memory requirements and simplified queue and vertex operations.

If the base terrain triangulation for a given frame is the resultant triangulation from the prior frame then both split and merge operations are necessary to maintain optimal triangulations. Such an implementation is said to be frame-coherent and takes advantage of frame coherence: Typically the resultant meshes from one frame to the next only differ at a few points, thus by starting from the resultant mesh of the prior frame, only a few split/merge operations need to be performed to produce the optimal mesh for the current frame.

3.4 Advantages and disadvantages

3.4.1 Advantages

The ROAM algorithm is very popular in terrain rendering applications due to its high performance and support for additional benefits:

- ROAM is an incremental greedy algorithm that yields optimal resultant meshes whilst using the least number of split/merge operations possible.
- The algorithm is simple and elegant and is conceptually easy to understand and implement.
- The choice of mesh representation and the vertex operations defined on top of it do not allow T-vertices [2] to occur, thus no efforts need to be made to avoid them.
- Since the priority queues are sorted, it is possible to achieve final mesh triangle counts directly.

[2]A T-vertex is a vertex which exists only on one side of an edge.

- For applications that require direct line of sight calculations the ROAM implementation can ensure that highest priority is offered to lines of sight.
- ROAM supports frame coherency which improves performance.
- A variety of performance enhancing error metrics and operations can be implemented for ROAM, including line of sight optimizations.

3.4.2 Disadvantages

The only notable disadvantages associated with the ROAM algorithm are:

- The per-vertex memory requirement is expensive relative to some other approaches such as the algorithm by Röttger *et al.* [12].
- The mesh representation in ROAM does not lend itself well to vertex representations such as triangle-strips, which leads to reduced rendering performance.

The algorithm presented in this paper eliminated the last disadvantage above, whilst sacrificing the ability to achieve specific mesh triangle counts. In the next section we discuss an alternative mesh representation that offers many advantages over a triangle bintree.

4 A triangle quadtree

4.1 Mesh representation and operations

As the rectangular bintree possesses a triangular bintree equivalent, so does a square-shaped quadtree possess a triangle quadtree equivalent – see Figure 4 below. Any triangle can potentially serve as root to such a triangle quadtree (also quadtritree) and the children of this triangle will be shaped as the root triangle. For the purposes of this paper we assume, without loss of generality, an equilateral triangle root.

Two types of triangles result from the triangle quadtree representation, namely, triangles with an upwards orientation and triangles with a downward orientation. The CLOD algorithm that we present functions independently of the triangle orientation. However, during rendering it is important to distinguish between these two orientations, in order to accommodate back-face culling.

To assist in implementing the algorithm it is useful to define a triangle and its relations as follows (see Figure 5):

- three vertices $v1, v2, v3$ defined as the left-most, middle and right-most vertices
- three neighbors $n1, n2, n3$ defined as the left-most, middle and right-most neighbor

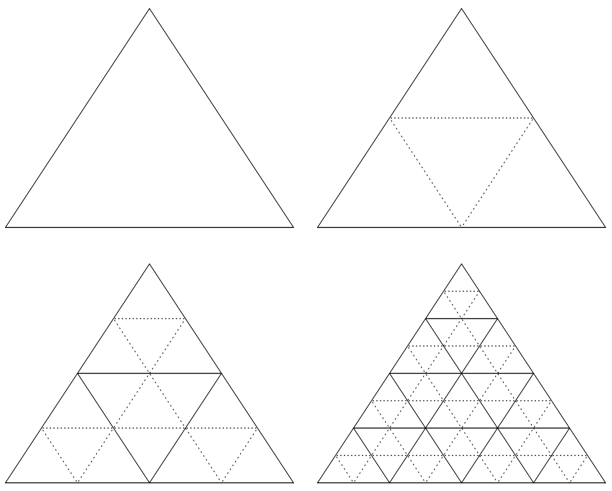

Figure 4: Various levels of refinement for a triangle quadtree

- four children a,b,c,d – a is always the middle-child, and the remaining children are defined from left to right

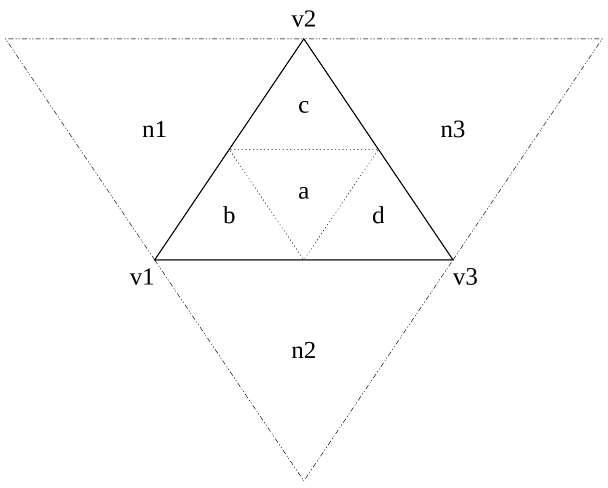

Figure 5: Definition of triangle and relations

We define split and merge operations over the mesh representation as follows: A split operation replaces a triangle with its four children, and a merge operation performs the inverse of the split operation. Figure 6 depicts these operations.

4.2 Mesh representation properties

As demonstrated above a triangle quadtree offers the same functionality as a triangle bintree. However, a triangle quadtree possesses a number of properties that distinguishes it from a triangle bintree and that may offer increased realism and rendering performance.

In a bintritree representation a given vertex may be shared by four to eight triangles, whereas in a quadtritree representation a vertex is always shared with six triangles[3]. Thus in a triangle quadtree representation a given

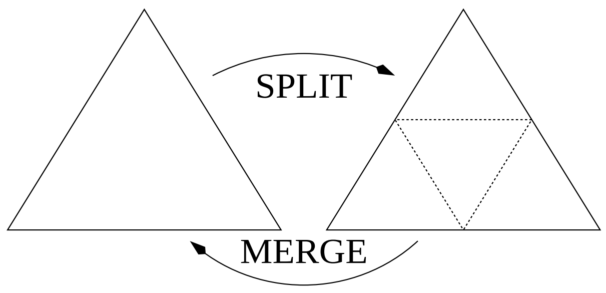

Figure 6: Representation of split and merge operations

rendered element consistently possesses six triangles per vertex point. This representational consistency results in a slight increase in perceived rendering accuracy. In addition perceived vertex popping [4] is slightly reduced, as instances where vertices are shared with four triangles in one frame and eight triangles in the next frame are eliminated.

More importantly, a mesh resulting from a quadtritree representation is significantly more suitable to rendering optimizations such as the use of triangle-strips. Triangle-strips that are implemented in a bintritree representation require constant maintenance and are typically short, averaging four to five triangles per strip (independent of rendering detail and height field size), as the geometric substructure does not lend itself well to long strips. Triangle quadtrees on the other hand naturally decompose into long, hardware-friendly triangle-strips that can be extracted after mesh creation. The average length of a given strip depends on terrain variance, height field size and rendering detail.

4.3 Mesh representation continuity

The mesh representation we have chosen is not perfect, as it fails to offer mesh continuity by introducing T-vertices between neighboring triangles of varying refinement. Figure 7 below offers an example of such mesh discontinuities.

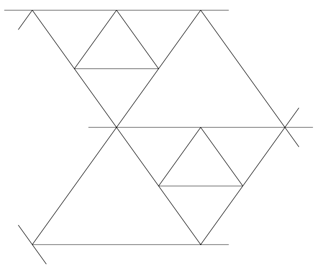

Figure 7: Mesh discontinuities

This problem is easily overcome: Triangulation is performed as if no mesh discontinuities are introduced by split and merge operations; only the level of triangle refinement needs to be considered at this stage. During the generation of the resultant mesh T-vertices are eliminated in an *ad hoc*

[3]With the exception of some triangles with coarser neighbors that share a vertex with four other triangles.

[4]A vertex pop is a temporal artifact due to frame changes (for example, a shadow jumping into existence where just now there was no shadow).

manner by considering triangle neighbors, as shown in Figure 8 below.

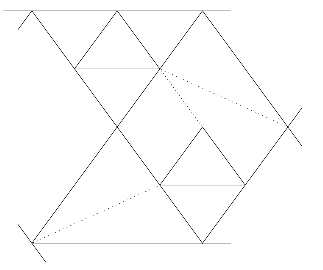

Figure 8: Correction of mesh discontinuity

5 The Diamond algorithm

In this section we describe our proposed terrain rendering algorithm. First we describe four LIFO queues that drive the terrain triangulation process; following this we offer descriptions and pseudo-code of the algorithm's rendering loop and the merge and split operations. Lastly we discuss strengths and weaknesses of our terrain rendering algorithm.

5.1 Triangle queues

The ROAM algorithm makes use of two queues, known as priority queues, that contain all splitable and all mergable triangles, sorted by priority. The priority queues are maintained with each operation the ROAM algorithm performs. This allows the algorithm to achieve specific triangle counts directly: An implementation may specify that a given frame should consist of, for example, 3000 triangles. The algorithm then splits and merges triangles as appropriate until the terrain triangulation consists of 3000 triangles. Due to the priority queues each split and merge operation is locally optimal, resulting in an optimal mesh for any triangle count (refer to [4] for proof).

A few years ago the purpose of maintaining constant triangle counts was to ensure that an application runs at a constant frame rate, since the processing requirements from frame to frame were nearly the same for constant triangle counts. However, the advent of powerful graphics hardware, and specifically hardware-based transforms and lighting, has virtually eliminated the need to achieve constant frame rates by maintaining constant triangle counts. Modern graphics hardware can easily cope with a thousand triangles more or less while leaving the rendering rate mostly unaffected.[5]

Since the rendering rate is now more independent of triangle counts, sorted priority queues are no longer a prerequisite for the terrain triangulation process. Instead the algorithm that we describe in this paper uses four LIFO queues that offer constant time insertions and deletions:

- *SplitBelow* - contains all splitable triangles that are below target priority
- *SplitAbove* - contains all splitable triangles that are above target priority
- *MergeBelow* - contains all mergable triangles that are below target priority
- *MergeAbove* - contains all mergable triangles that are above target priority

5.2 The algorithm

In this subsection we describe the Diamond terrain algorithm. We first present the algorithm's main loop, followed by an algorithmic description of the split operation (and its child generation subroutine) and the merge operation.

5.2.1 The main loop

The rendering loop of our algorithm consists of two distinct activities: Priority evaluations and triangulations. During priority evaluation the priorities of all queued triangles are re-evaluated and triangles are requeued as necessary. During triangulation all mergable triangles that are below target priority are merged and all splitable triangles that are above target priority are split.

```
if ResetDiamond then begin
   reset mesh and queues
   cull and prioritize primary base triangles
   enqueue triangles in SplitAbove
      or SplitBelow as appropriate
end else begin
   pass through SplitBelow, re-evaluate
      priority and requeue item to SplitAbove
         if necessary
   pass through MergeAbove, re-evaluate
      priority and requeue item to MergeBelow
         if necessary
end

while MergeBelow is not empty do
   merge item from MergeBelow
while SplitAbove is not empty do
   split item from SplitAbove
```

ResetDiamond is a boolean variable that is true the first time the loop is executed. If *ResetDiamond* is true for each later frame, then the algorithm only performs split operations on the mesh; if it is false, then the algorithm is frame-coherent and will perform both splits and merges.

[5] For example, a low-end GeForce2 card possesses a theoretical transform rate of 20 million tris per second; it can thus theoretically maintain a constant frame rate of 200 000 tris per frame, at 100 frames a second.

Note that after each execution of the main loop the MergeBelow and Split-Above queues will be empty – all splitable and mergable triangles are in the SplitBelow and MergeAbove queues after each iteration of the main loop.

5.2.2 The split operation

Geometrically a split operation is fairly simple and straightforward to implement, however, it is of vital importance to the execution of the algorithm that the four LIFO queues are correctly maintained.

The split operation also ensures that neighboring triangles differ at most by one level of refinement. This is done by forcing a neighbor to split if it is too coarse to allow the current triangle to split.

```
if triangle is enqueued then
   dequeue triangle (from SplitAbove)
if parent is enqueued then
   dequeue triangle (from a merge queue)
if a neighbor is coarser compared to triangle
then
   split the neighbor
if a neighbor's parent is enqueued then
   dequeue neighbor's parent (from a
      merge queue)

generate children for triangle
cull and prioritize children and enqueue
   in SplitAbove/SplitBelow as appropriate

if triangle is mergable then
   enqueue it (into MergeAbove)
```

Note that a triangle is mergable if it has children but no grandchildren.

5.2.3 Child generation

Child generation forms part of the split operation. When implementing child generation it is important to ensure that neighboring triangles correctly point to each other.

```
create children and attach them to triangle
assign appropriate vertices/coordinates
   to children

if no neighbor then
   set neighbor pointers to nil
else
   if the neighbor has children then
      point triangle and neighbor children
         to each other
   else
      point triangle children to neighbor
```

5.2.4 The merge operation

Similar to splitting, merging is a geometrically simple task. However, correct queue maintenance requires special attention.

```
if children are enqueued then
   dequeue children (from a split queue)
   dequeue triangle (from MergeBelow)
   correct neighbor child pointers
      if necessary
   delete children from triangle

if a neighbor's parent can now merge then
   enqueue neighbor's parent (into
      appropriate merge queue)
if triangle's parent can now merge then
      enqueue triangle's parent (into
         appropriate merge queue)
      enqueue triangle (into SplitBelow)
```

5.3 Results

5.3.1 Implementation

We implemented the algorithm described above, using indexed vertex (vertex-buffered) rendering; both source and executable are available [9]. We use a precalculated 1025×1025 height field for terrain data; the height field was created using the method of mid-point displacement [6]. Our implementation accurately models the terrain data whilst maintaining a high frame rate and good image quality. See Figure 11 and Figure 12 for screenshots from the Diamond implementation.

5.3.2 Analysis of ROAM and Diamond

ROAM claims an order of execution of $O(\textit{mesh changes})$.[6] However Duchaineau admits that current hardware does not exist that could achieve this order of execution, but that in principle the necessary hardware could be built [2]. Given that we accept this statement the order of execution of ROAM is more accurately described as $O(\textit{mesh changes} \times \log(\textit{mesh changes}))$ to reflect the computations necessary to maintain accurate priority queues.

Still assuming that Duchaineau's argument holds, the Diamond algorithm achieves a true order of execution of $O(\textit{mesh changes})$ due to the use of constant time queues.

5.3.3 Experimental results

To demonstrate the performance of ROAM and Diamond, both algorithms have been implemented and statistical data

[6]That is, the processing time of the current frame depends on the number of differences that exist between the mesh of the prior frame and the mesh of the current frame. Typically as little as five to fifty mesh changes need to be evaluated for a given frame.

has been collected over the course of several hours of simulation. The simulations did not include render calls, instead only the capability of each algorithm at generating resultant meshes was measured, in each test the frame-coherent version of the respective algorithms was used – the resultant data sets have been made available [9].

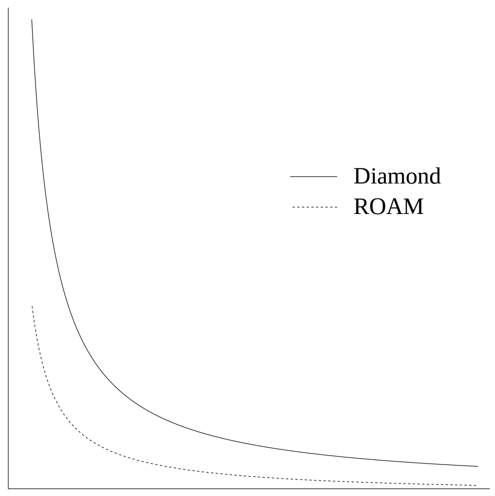

Figure 9: Performance graphs of ROAM and Diamond algorithms

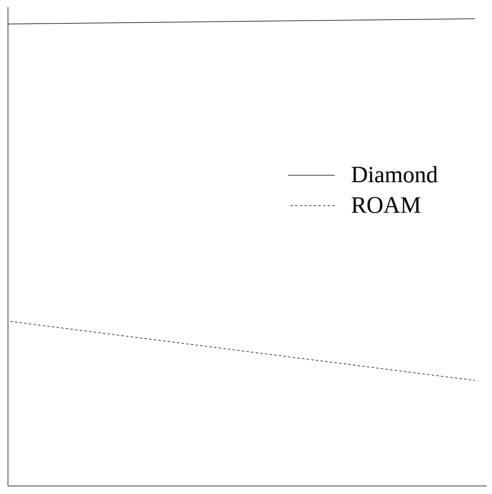

Figure 10: Graphs of total work performed by ROAM and Diamond algorithms

We assume that the algorithms have been implemented with a similar degree of efficiency, therefore any differences in performance are due to the order of execution in the algorithms.

Figure 9 displays *triangles per frame* against *frame count*, which indicates the performance of ROAM and Diamond; Figure 10 plots *triangles per frame* against *triangle count*, as the terrain data and experiment conditions remain constant, the increase in average *triangles per frame* are an indication of the accuracy with which the height field is approximated by the respective algorithms. The *triangle count* is the total number of triangles generated by the respective algorithms over a constant period of time and thereby forms a strong measure of the total work performed by the algorithms.

The graphs clearly indicate that Diamond performs stronger by approximately a factor of three. This can be attributed to the split and merge operations of the respective ROAM and Diamond algorithms, as a single Diamond split operation is equivalent to three seperate ROAM split operations. Additionally Figure 10 demonstrates that the total work performed by Diamond remains roughly constant, whereas the total work delivered by ROAM decreases with increasing tri counts. This reflects the stronger order of execution in the Diamond algorithm compared to ROAM.

5.3.4 Triangle-strips

We have not presented a method for extraction of triangle-strips from the resultant mesh. However, a number of approaches exist specifically for this purpose; for example: ROAM [4] continuously maintains short triangle-strips, STRIPE [5] is a successful triangle-stripping algorithm and Meshifier [7], by Brad Grantham, has been used in commercial applications.

To demonstrate that Diamond generates a triangle-strip friendly mesh a simple strip extraction algorithm was implemented. The approach used is described by Tom Nuydens [11]. Triangle-Strip extraction tests were performed within the same data set and conditions used for the base implementation (subsection 5.3.1).

Within our test environment the average strip size extracted was between 45 and 60 triangles per strip, with relatively few very short strips.[7] This basic result can be further improved by noting that short triangle strips should be rendered in a triangle list to optimize rendering performance. By deferring triangle-strips consisting of a single triangle into a seperate triangle list the average length of remaining triangle-strips increases to 150 to 250 triangles per strip.

To further improve performance the use of indexed triangle-strips (vertex-buffered triangle strips) is advised.

5.3.5 Improvements

Various improvements to the ROAM algorithm, both on a theoretical and implementation specific level, have been put forth [4, 8] such as geomorphing and deferring priority calculations. These improvements are independent of the geometric substructure and thus can be applied to the Diamond algorithm.

[7] As a reminder: ROAM averages 4.5 triangles per strip.

6 Conclusion

We have reviewed the ROAM algorithm and its mesh representation, noting that this choice of geometric substructure offers poor rendering optimization. We put forward the use of a triangle quadtree as an alternative mesh representation that maintains the advantages of a triangle bintree but in addition offers good rendering optimization in the form of triangle-strips. Additionally we suggested the replacement of the priority queues in ROAM with four LIFO queues. Lastly we presented an algorithm for terrain rendering that makes use of our suggestions.

The primary workload of a terrain renderer rests on the shoulders of the graphics processing hardware. The algorithm that we have presented utilizes four LIFO queues that nearly eliminate all queueing overhead, allowing more efficient use of processing resources. Additionally the mesh representation we have chosen can potentially significantly reduce rendering time with the use of long triangle-strips.

In recent years the use of the ROAM algorithm on arbitrary objects received exposure [13]; indeed the ROAM algorithm can be applied successfully to offer continuous levels of detail in arbitrary meshes, provided that some form of variance computation can be defined over the mesh. The Diamond algorithm presented in this paper can offer the same universal applicability: Provided a variance computation is defined over an arbitrary mesh, the Diamond algorithm computes a continuous levels of detail representation over that mesh.

Figure 11: Screenshot of textured terrain

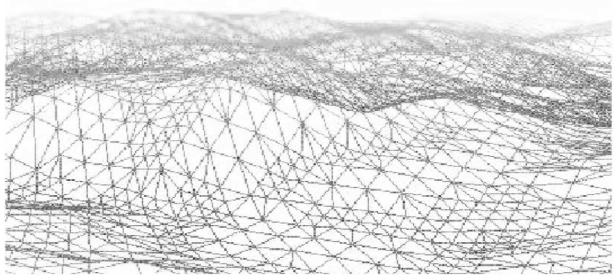
Figure 12: Screenshot of wireframed terrain

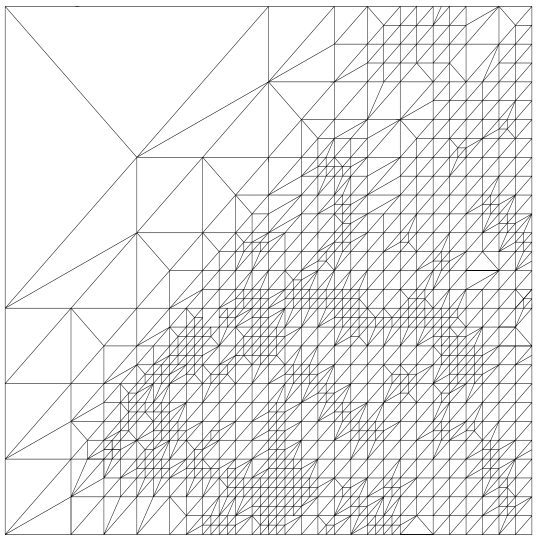
Figure 13: A top-down view of a sample terrain mesh

References

[1] W.H. de Boer. Fast terrain rendering using geometrical mipmapping. 2000. `http://www.flipcode.com/tutorials/geomipmaps.pdf`.

[2] M. Duchaineau. How can you claim O(delta output) work per frame? `http://www.cognigraph.com/ROAM_homepage/deltaoutput.html`, June 2001.

[3] M. Duchaineau. ROAM Algorithm: Real-time Optimally-Adapting Meshes. `http://www.cognigraph.com/ROAM_homepage/`, February 2002.

[4] M. Duchaineau, M. Wolinsky, D.E. Sigeti, M.C. Miller, C. Aldrich, and M.B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proceedings of the 8th IEEE Visualization '97 Conference*, pages 81–88, Phoenix, AZ, October 1997.

[5] F. Evans, S. Skiena, and A. Varshney. Optimizing triangle strips for fast rendering. In *In Proceedings of Visualization 96*, pages 319–326, 1996. `http://www.cs.sunysb.edu/evans/stripe.html`.

[6] A.R. Fernandes. Terrain tutorial. `http://www.lighthouse3d.com/opengl/terrain/`.

[7] B. Grantham. Converting sets of triangles with shared edges into triangle strips. `http://www.plunk.org/~grantham/public/meshifier/show.html`.

[8] Y. Gyurchev. Roam implementation optimizations. `http://www.flipcode.com/tutorials/tut_roamopt.shtml`, 2001.

[9] H. Hakl. Diamond implementation. `http://www.cs.sun.ac.za/~henri/terrain.html`, 2002.

[10] P. Lindstrom, D. Koller, W. Ribarsky, L.F. Hodges, N. Faust, and G.A. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. In *Proceedings of ACM SIGGRAPH 96*, pages 109–118, August 1996. `http://www.cc.gatech.edu/gvu/people/peter.lindstrom/`.

[11] T. Nuydens. Triangle strip generation. `http://www.delphi3d.net/articles/viewarticle.php?article=tristrips.htm`.

[12] S. Röttger, W. Heidrich, P. Slusallek, and H-P. Seidel. Real-Time Generation of Continuous Levels of Detail for Height Fields. In *Proceedings of WSCG '98*, pages 315–322, Plzen, Czech Republic, 1998.

[13] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[14] A.J. Stewart. Hierarchical Visibility in Terrains. *Eurographics Rendering Workship*, pages 217–228, June 1997.

[15] U. Thatcher. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. `http://www.gamasutra.com/features/20000228/ulrich_01.htm`, 2000.