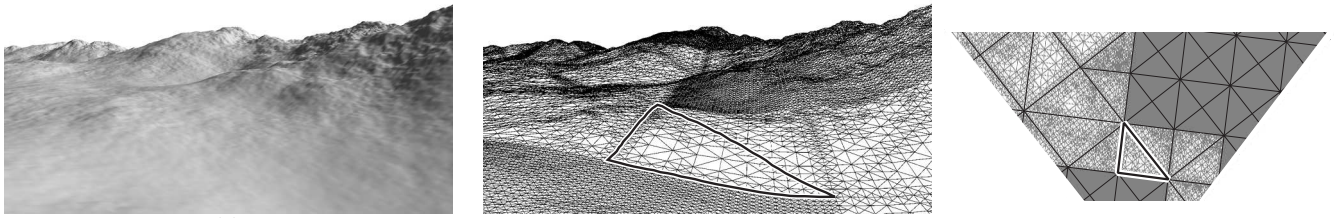
Fast View-Dependent Level-of-Detail Rendering Using Cached Geometry

Joshua Levenberg*

University of California at Berkeley



(a) (b) (c)

Figure 1: (a) shows an example of a 2049×2049 height field rendered with aggregate triangles; (b) shows the triangles used to render (a); (c) is a top view of the view frustum with aggregate triangles in black. The same aggregate triangle is highlighted in (b) and (c).

ABSTRACT

Level-of-detail rendering is essential for rendering very large, detailed worlds in real-time. Unfortunately, level-of-detail computations can be expensive, creating a bottleneck at the CPU.

This paper presents the CABTT algorithm, an extension to existing binary-triangle-tree-based level-of-detail algorithms. Instead of manipulating triangles, the CABTT algorithm instead operates on clusters of geometry called aggregate triangles. This reduces CPU overhead, eliminating a bottleneck common to level-of-detail algorithms. Since aggregate triangles stay fixed over several frames, they may be cached on the video card. This further reduces CPU load and fully utilizes the hardware accelerated rendering pipeline on modern video cards. These improvements result in a fourfold increase in frame rate over ROAM [7] at high detail levels. Our implementation renders an approximation of an 8 million triangle heightfield at 42 frames per second with an maximum error of 1 pixel on consumer hardware.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Viewing Algorithms; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric Algorithms, Object Hierarchies; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

Keywords: view-dependent mesh, level of detail, height fields, terrain, binary triangle trees, triangle bintree, multiresolution meshes, displacement maps, frame-to-frame coherence

1 INTRODUCTION

The goal of geometric level-of-detail rendering algorithms is to adjust the geometry of rendered objects for improved detail and performance. Greater realism is achieved by adding geometric detail to close and important objects. Performance is improved by simplifying objects far from the camera. Level-of-detail algorithms can be essential for realizing very large worlds. They can be used to control the total amount of geometry in a scene. They can also help meet fixed performance goals under varying conditions (such as running on different machines).

To make the best use of available resources, a level-of-detail algorithm should be *view-dependent*. That is, it should tailor the rendered geometry to the location of the view point, on a frame-by-frame basis. The algorithm should have fine-grained control over the geometry so that changes in the level of geometric detail are not objectionable. These desires tend to be at odds with keeping geometry static so that it may be cached [23].

Consumer graphics hardware has improved dramatically, doubling in speed every six months and now integrates the full rendering pipeline. Recent fine-grained level-of-detail algorithms generate a new mesh every frame. This requires a great deal of CPU power and has become a major bottleneck. Consequently, graphics board manufacturers have in recent years been discouraging the use of all but the simplest level-of-detail algorithms.

Our goal is to describe a level-of-detail algorithm appropriate for commercial applications. As such, it needs to be able to take advantage of video hardware available to consumers now and in the future. Further, it should be adaptable to give acceptable performance on older machines. A desirable level-of-detail rendering algorithm would:

- reduce CPU workload without a large increase in the number of triangles needed to achieve a given error rate,
- be adjustable to take advantage of the relative performance of the video card and CPU, and
- dynamically adjust the mesh based on the view point in order to render as quickly as possible without visible errors.

The starting point for meeting these goals is an existing view-dependent level-of-detail algorithm that outputs triangles. However, we modify the algorithm to manage collections of geometry called *aggregate triangles*, instead of individual triangles. These aggregate triangles may then be cached on the video card. This has several benefits, including:

- **improving transformation performance**: the graphics subsystem operates more efficiently in retained mode than immediate mode

*Research supported by NSF Grant No. DMS-0209617

- **takes advantage of temporal coherence**: changes between frames are localized, and there is little or no processing for triangles in the unchanged areas
- **reducing stalls in the graphics pipeline**: cached data can be processed directly by the video card without delays

In order to realize the benefits of caching, a level-of-detail algorithm must incrementally modify small fractions of the mesh each frame. Further, the CABTT algorithm needs to include a strategy that ensures adjacent aggregate triangles match up without T-junctions. This strategy ideally allows an aggregate triangle to use the same geometry regardless of the detail level of adjacent aggregate triangles.

Binary triangle trees (BTTs, also called *triangle bintrees* [7], *right triangular irregular networks* [9], or, in finite elements, newest-vertex-bisection [24]) are ideally suited to represent meshes for the CABTT (Cached Aggregated Binary Triangle Trees) algorithm. BTTs use a single shape, the right isosceles triangle, and have a refinement rule that keeps the mesh crack-free. This simple structure means a straightforward policy will ensure aggregate triangles abut without T-junctions. Contrast this with red-green triangulations [1] which have two different refinement rules and use two different shapes. The ROAM [7] algorithm uses BTTs to incrementally update a mesh from frame to frame.

Previous algorithms have difficulty maintaining interactive frame rates at high detail levels. The CABTT algorithm not only scales well, but also balances the workload between the video card and CPU to achieve good performance on a wide variety of machines.

2 PAST WORK

There is a very large field of level-of-detail algorithms, but real-time view-dependent algorithms are relatively recent [23].

There are several level-of-detail techniques applicable to general meshes [8, 14, 22, 32]. While able to use relatively few triangles to approximate a mesh, they generally are expensive in time or memory and do not scale to high detail levels.

Some height field level-of-detail algorithms use Delaunay triangulations [6]. View-dependent progressive meshes have been specialized to the terrain case [15], allowing a very general class of meshes, and generally achieve higher accuracy per given triangle count.

Regular subdivision meshes have been more popular in recent years. These allow simpler and faster processing at the expense of an increase in the number of triangles needed to achieve the same error threshold. Restricted quadtree triangulations [26, 28, 29, 30, 31] were among the first used. They use a similar but somewhat less flexible class of meshes as BTTs. In particular, the coarsest mesh representable is finer for restricted quadtrees and they are less applicable to non-height-field models.

Binary triangle trees have seen a lot of recent research interest [4, 7, 9, 20, 21, 27]. While typically used to render height fields, they may be used to render any base mesh with an offset map [7, 24]. Blow [4] gave a method of reducing error metric computation for high detail meshes. Lindstrom and Pascucci [20] described a cache-friendly vertex indexing scheme. RUSTiC [27] has triangle clusters very similar to our aggregate triangles, though with much greater memory cost than the approach presented here.

We maintain a crack-free mesh using split and merge operations that incrementally modify the mesh without introducing T-junctions, as in ROAM [7]. Another approach to maintain a crack-free mesh [10, 20, 26, 28] adjusts the error metric so that adjacent triangles in the mesh differ by at most one level of refinement. This increases the number of triangles rendered but allows rendering to be done in a single pass and does not require temporal coherence.

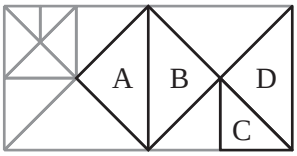


Figure 2: Adjacent triangles are always within one level of detail.

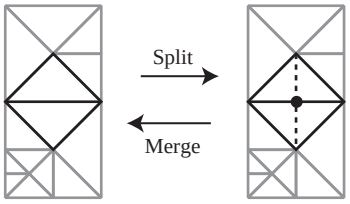


Figure 3: The split and merge operations

Some previous algorithms deal with blocks of geometry [15, 21], typically as a starting point for view-dependent simplifications. RUSTiC [27] uses triangle clusters to reduce CPU bottlenecks, though it does not perform any caching.

3 REVIEW OF BINARY TRIANGLE TREES

Binary triangle trees are typically used to represent terrain, given as a $(2^n + 1) \times (2^n + 1)$ regular grid of height field data. More generally, BTTs can represent a base mesh with a displacement map or offset map (such as from a remeshing algorithm [11, 16, 17, 18, 19, 25]). In the case of a height field, the base mesh is a square divided into two right triangles. At the other end of the spectrum, the *full mesh* is the most detailed mesh.

While this paper focuses on BTTs representing height fields, the techniques are equally applicable to more general meshes.

A binary triangle tree represents a mesh of *binary triangles* satisfying certain rules. Each binary triangle corresponds to a right isosceles triangle in the grid of offset data. For the specific case of a height field, the projection of any binary triangle onto a horizontal plane is a right isosceles triangle. The hypotenuse or *base* of the triangle can only abut either the base of a triangle at the same level of detail (like triangles A and B in Figure 2) or the leg of a triangle one level coarser (triangles C and D in Figure 2).

Two triangles that meet base-to-base are called a *diamond*. A *split* divides each triangle of a diamond in two, as shown in Figure 3. This introduces a vertex in the middle of the base of the original two triangles. The inverse of this operation is called a *merge*. Also note that one can split a triangle whose base is part of the boundary of the mesh. Figure 4 shows meshes corresponding to the first few levels of a binary triangle tree, formed by repeatedly splitting every triangle. The vertices in the base mesh are associated with grid positions 2^n apart to ensure that only vertices on the regular grid are generated in a $2n$-level tree.

In order to refine a triangle that is not part of a diamond, we must split its base neighbor. This operation is called a *force split*. Note that this may recursively force other triangles to split, as in Figure 5. This operation always terminates, since a split can only force the split of a coarser triangle.

Rendering is a two-step procedure. First, the BTT from the previous frame is updated using splits and merges. These updates are determined by the error metric (see Section 5) and the current location of the view point. Second, the BTT is drawn using a simple recursive algorithm. We can easily perform hierarchical frustum

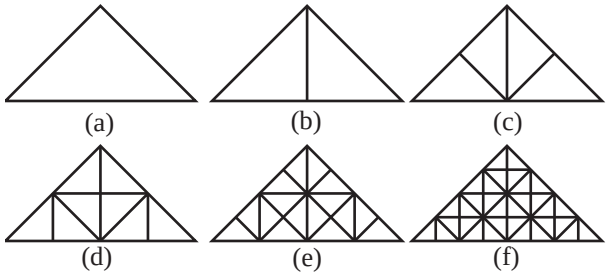


Figure 4: Uniform subdivision of a binary triangle.

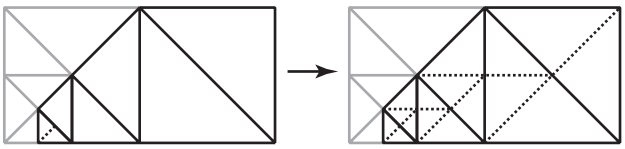


Figure 5: Splitting a triangle may force other triangles to split.

culling as part of this traversal. We can traverse the BTT with a Sierpinski space-filling curve, enabling efficient triangle strip or triangle fan rendering.

To avoid lighting-related artifacts as geometric detail is modified, vertex lighting should be avoided. Per-pixel techniques such as bump maps [3], normal maps [5], or pixel shaders can be employed instead. Our implementation uses simple, baked-in lighting (fixed lighting included in the texture map) to avoid these problems.

4 AGGREGATE TRIANGLES

An aggregate triangle is a collection of triangles that substitutes for a single binary triangle in a BTT. An aggregate triangle must therefore correspond to a right isosceles triangle in the grid of offsets. In the case of a height field, this means an aggregate triangle projects to a right isosceles triangle in the horizontal plane.

Instead of constructing a BTT to represent the triangulation of the terrain, we will construct a much shallower *aggregated BTT* dividing the terrain into aggregate triangles. We will cache the aggregate triangles, so it is important that the geometry (or *sub-triangulation*) of each aggregate triangle remains fixed for several frames. When an area of an aggregated BTT needs to be refined, we replace a diamond of aggregate triangles with four aggregate triangles. This happens just like the split operation described in Section 3, except with aggregate triangles instead of triangles. The merge operation is analogous.

The number of triangles in the average sub-triangulation determines the granularity of these operations. Coarser granularity means fewer aggregate triangles to manage and fewer split and merge operations, but more data to upload to the video card every time a split or merge occurs. Also, coarser granularity will increase the number of triangles needed to meet a particular maximum error tolerance. By profiling, we can find the level of aggregation that performs best. This will vary depending on several factors including the relative performance of the CPU and video card.

We render an aggregated BTT much like a regular BTT (see Section 3). When splitting or merging, we must release the caches of the aggregate triangles we are replacing and upload the geometry for the new aggregate triangles. Then, when traversing the BTT, we simply render the caches associated with any on-screen aggregate triangle. As an optimization, the caching is performed during the drawing stage instead of the split and merge stage (see Figure 10). This avoids wasted work if more than one split or merge happens

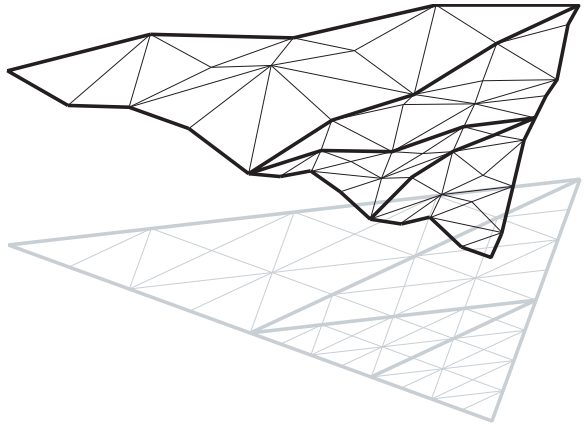


Figure 6: An aggregated BTT with uniform sub-triangulations. The boundaries of the aggregate triangles are bold.

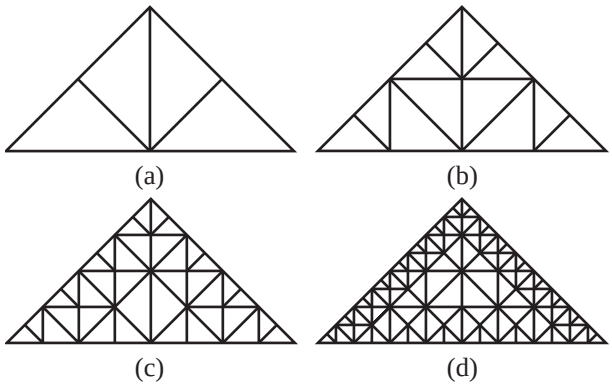


Figure 7: Splitting a binary triangle so that its edges have (a) 2 segments, (b) 4 segments, (c) 8 segments, and (d) 16 segments.

in same area of mesh. Even better, we also avoid wasting time uploading culled or off-screen geometry.

When creating the sub-triangulations for aggregate triangles, we must ensure that adjacent aggregate triangles match up along their common boundaries. We have adopted the simple policy of uniformly subdividing all of the boundaries into a fixed number of segments (defining the aggregation level). This policy ensures continuity between adjacent aggregate triangles even when they are at different levels of detail. It isolates sub-triangulations from each other, so mesh operations are local and fast. Sub-triangulations may be generated on the fly with a minimal amount of pre-computed data (contrast with RUSTiC [27] which pre-computes and stores all sub-triangulations).

The simplest sub-triangulation policy is to uniformly subdivide each aggregate triangle, as in Figure 6. That is, every aggregate triangle is subdivided the same way, except that the vertices are displaced according to the offset or displacement map. Typically, aggregate triangles will be subdivided using one of the patterns from Figure 4 (a), (c), (e).

We may achieve a closer fit to the full mesh with an adaptive method of sub-triangulation (see Figure 1). The CABTT algorithm creates a miniature BTT that represents a single aggregate triangle. It starts with a single binary triangle. It splits the triangle's boundary until it has the desired number of segments (see Figure 7). We then apply additional splits to minimize the error metric (described in Section 5). This continues until we either reach a maximum number of triangles or we want to perform a split that would cause the boundary of the aggregate triangle to be further subdivided. This

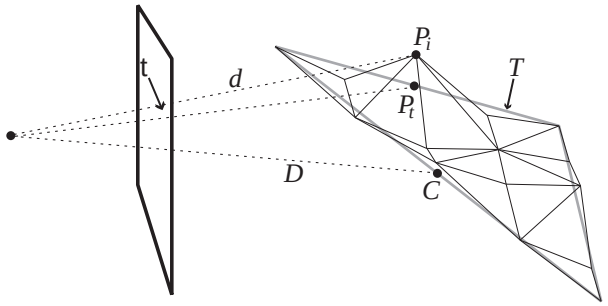


Figure 8: Isosphere error metric notation.

procedure is called SubTriangulation in Figures 9 and 10. Note that we can discard this BTT as soon as it has been uploaded to the graphics card.

In our tests, adaptive sub-triangulations reduced the number of triangles needed to achieve a particular error tolerance by as much as 50% over uniform sub-triangulations. Since the time to compute these sub-triangulations is modest, this almost doubles the frame rate.

In our test case, we achieved the best performance with aggregate triangles containing 16 segments in each edge. On average, these sub-triangulations would consist of 206 triangles. At this level of aggregation, at most 20% of these triangles are an overhead cost introduced by the boundary rule. That is, if we run the test without restrictions on the boundaries of the aggregate triangles, we achieve the same error tolerance with 20% fewer triangles (though with cracks in the mesh between adjacent aggregate triangles).

5 ERROR METRICS

We use an error tolerance τ (measured in pixels) to trade quality for speed. Lower τ corresponds to higher quality at reduced speed. We introduce splits anywhere the screen space error is larger than τ and merge when that does not introduce errors larger than τ. After describing the error metric, we will show how to adapt it to aggregated BTTs.

To determine when to split a triangle, we use Blow's *isosphere error metric* [4]. Suppose we are given a triangle T with point C at the midpoint of its base (see Figure 8). We want to compute the distance D such that whenever the view point is at least D away from C, the rendering of T will be within τ pixels of the full mesh. T is split when the view point enters the sphere (an error isosphere) with center C and radius D.

Let K be the screen resolution divided by the tangent of the field of view.[1] Let P_i be a point in the full mesh corresponding to the point P_t in T (for a height field, these points will be aligned vertically). Let d be the distance from the view point to P_i,[2] then we want to ensure that $\tau \geq K\frac{|P_t - P_i|}{d}$. This equation is conservative — it assumes the triangle is viewed edge-on. We want D to be at least d plus the distance from C to P_i, so (assuming fixed τ):

$$D = \max_{P_i} \left\{ \frac{K \cdot |P_t - P_i|}{\tau} + |P_i - C| \right\} .$$

Away from the center of the screen, this may underestimate the error by 10–40% (typically, depending upon the field of view). This is normally ignored since the center of screen is considered the most important (a conservative bound is possible). This error metric is *orientation insensitive* — it does not depend on the angle at which the triangle is viewed.

This error metric is *non-monotonic* [26], that is, splits may increase the total error. Non-monotonic error metrics produce triangulations with fewer triangles than corresponding monotonic metric where the error is bounded hierarchically. However, if the mesh is altered locally, monotonic metrics may be recomputed more quickly.

In order to meet a target frame rate, we may want to adjust τ. Recomputing D for every triangle is computationally expensive, but accurate. However, when D is large compared to the cluster width, D is nearly linear in $1/\tau$. If we want to increase the number of triangles rendered, scaling D by a constant factor $\alpha > 1$ will reduce τ by a factor of α.

Aggregate error measures the approximation error of using an aggregate triangle just as a regular error metric measures the approximation error of a single triangle. We use the point C in the middle of the base edge of the aggregate triangle and compute a distance D for the whole triangle. In fact, we could use the equation for D given above directly, with the understanding that T is an aggregate triangle.

An approximation to this is to simply choose D so that it contains every isosphere of the sub-triangulation. This is fast enough[3] that several aggregation levels may be profiled at startup to determine which is optimal.

When constructing an adaptive sub-triangulation for an aggregate triangle, we want to minimize D. We therefore prioritize the triangles in the miniature BTT by their distance to C plus the isosphere radius. We may then use a priority queue to efficiently determine which binary triangle to split next.

6 OPTIMIZATIONS

To reduce the number of times per frame that the error metric is evaluated, we set up queues that track how soon a triangle could need to split or merge. If a split sphere for a triangle is x units away from the current camera position, we only need to consider splitting the triangle after the camera has moved at least x units. The advantage of this technique is that it does not require a heuristic to estimate how many frames before an error needs to be reevaluated, as required by ROAM's [7] dual queues. ROAM, however, supports control over the number of triangles rendered, while this technique only controls the maximum error.

Blow [4] uses a hierarchical sphere tree to reduce metric evaluations at higher detail levels. With aggregation, we found our queue implementation sufficient. The CABTT algorithm required 45 error metric evaluations per frame, compared to about 2,100 without aggregation.

To keep the comparison fair, our ROAM implementation includes additional optimizations. Since queue operations were a large part of the CPU work in ROAM, we used an optimized bucket queue instead of a standard STL set. This increased frame rates by 28%. Instead of sending individual triangles using immediate-mode OpenGL, we stitched adjacent triangles together into fans, averaging 3.7 triangles per fan. Fan rendering increased frame rates by a further 12%. These optimizations gave ROAM an advantage over the lowest levels of aggregation. Other factors, such as the error metric, were kept the same between the ROAM and aggregate triangle implementations.

Pseudo-code for our CABTT implementation is given in Figures 9 and 10. Please note that triangles that may be split are in-

[1] Assuming the pixels are square, K will be the same if computed from the horizontal resolution and field of view or from the vertical resolution and field of view.

[2] To be conservative, we should use P_t when it is farther from C.

[3] Pre-computing D for every aggregate triangle takes less than a tenth of a second for a 257×257 height field on a 450 MHz Pentium II.

```
procedure Initialize(BaseMesh, InitialCameraPosition,
                     AggregationLevel):
compute and store D for each diamond
for each possible splittable Diamond
         of aggregate triangles:
   Error1 = error of SubTriangulation(Diamond.Triangle1)
   Error2 = error of SubTriangulation(Diamond.Triangle2)
   AggregateError[Diamond]=Max(Error1, Error2)
DistanceSoFar = 0
LastPosition=InitialCameraPosition
AggregateTree = BaseMesh
MergeQueue = empty
SplitQueue = [each diamond of BaseMesh]
```

Figure 9: Initialization for aggregate BTT rendering algorithm.

```
procedure RenderCABTT(NewCameraPosition):
update DistanceSoFar and LastPosition
while Diamond in MergeQueue/SplitQueue
      needs to be reevaluated:
   remove Diamond from the queue
   recompute distance until merge/split
   if negative:
      merge/split Diamond (updating queues)
   else:
      add Diamond back to the queue
for each Triangle in AggregateTree:
   if off-screen:
      discard caches for Triangle
   else:
      if Triangle has no cache:
         Triangle.Cache =
               Cache(SubTriangulation(Triangle))
      RenderCache(Triangle.Cache)
```

Figure 10: Render procedure called once per frame.

dexed by the midpoint of their base edges. Diamonds in the split and merge queues are indexed by the value of `DistanceSoFar` when they will be reevaluated.

7 RESULTS

Our test machine was a 450 MHz Pentium II with 128 MB of RAM. It was equipped with a NVIDIA GeForce 2 with 32 MB of video memory. Our implementation of the algorithm used OpenGL extensions[4] to cache geometry on the video card.

Unless otherwise noted, our test scene was a 2049×2049 texture-mapped height field. The camera followed a circular path in the horizontal plane. The error metric was set to $K/\tau = 600$, corresponding to a maximum error of $\tau \approx 1$ pixel at a resolution of 640×480 with a 45° field of view. *Popping*, or discontinuities from changes in the level of detail, was barely detectable in our tests. We found that window size had only a small effect on average speed. Results were averaged over the 2400 frames of the test.

Figure 11 summarizes the performance of our implementation. Observe that both aggregation and caching improve total triangle throughput (Figure 11(b)), though with diminishing returns at high aggregation levels. Higher aggregation levels require more triangles to achieve a given error bound, as shown in Figure 11(c). Caching aggregate triangles with an average of 206 sub-triangles gave the highest frame rate of 44 frames per second (Figure 11(a)).

[4]Specifically NV_vertex_array_range and NV_fence. The alternative for ATI video cards would be ATI_vertex_array_object. OpenGL 2.0 plans to directly support this form of caching.

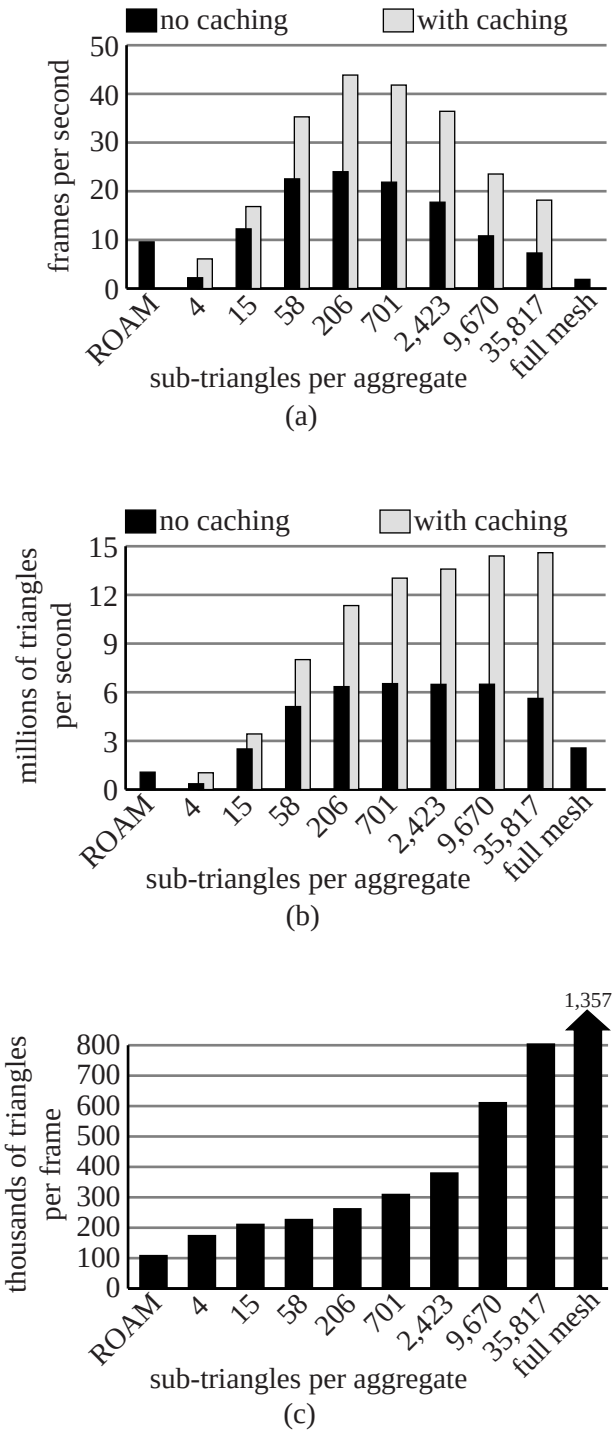


Figure 11: Performance of aggregation and caching with a maximum error τ of one pixel, a 640×480 window, and a 2049×2049 height field. *Triangles* refers to triangles requested to be drawn after using a simple field-of-view culling procedure. The columns of each graph correspond to 1, 2, 4, ..., 512 edges per side of the aggregate triangles. Graph (a) demonstrates that peak performance occurs with an average of 206 sub-triangles in each aggregate. Graph (b) shows that triangle throughput improves with greater aggregation. Graph (c) indicates how many triangles are rendered at each aggregation level.

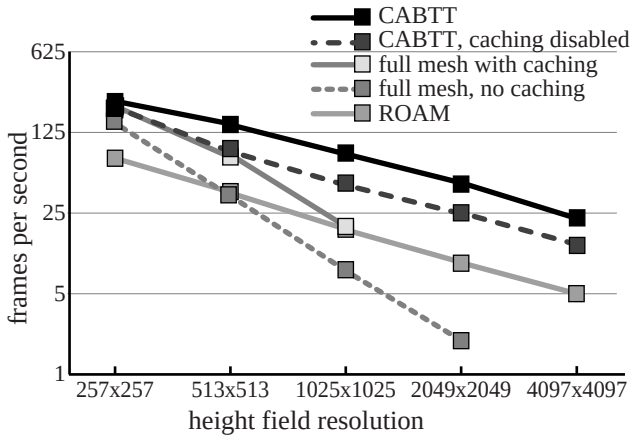


Figure 12: A log-log plot of frame rate versus resolution.

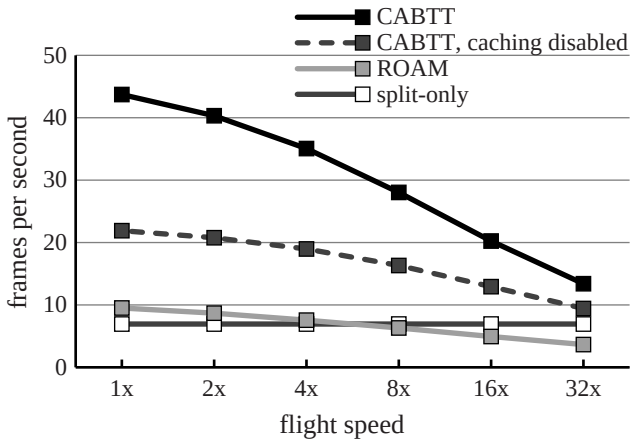


Figure 13: The frame rate at various camera-movement speeds shows the influence of temporal coherence. For comparison, *split-only* calculates each frame from scratch, and so is not influenced by flight speed.

In this case, an average of 1.8 splits/merges of aggregate triangles were required per frame.

Aggregation also kept the memory requirements modest. Height field and error data used the most memory (a total of two floats per vertex in the full mesh). Cached geometry data, taking 12 bytes per rendered triangle,[5] accounted for the remainder. The aggregated BTTs and aggregate errors used a negligible amount of storage.

Figure 12 demonstrates how well our algorithm scales to higher resolution meshes. A 4097×4097 height field renders at interactive rates (24 frames per second) using the CABTT algorithm. As expected, rendering the full mesh every frame is efficient at lower resolutions, but not at higher resolutions. On the other hand, the level-of-detail algorithms (ROAM and CABTT), handle higher resolutions more gracefully. Aggregation and caching are significant improvements to ROAM at all resolutions. The graph only shows the optimal aggregation level for each resolution. Lower resolution meshes used a lower aggregation level.

Since the algorithm exploits temporal coherence, frame rates drop as the camera moves faster. Figure 13 shows how increased flight speed degrades performance of ROAM-based algorithms. One way of controlling level of detail without using temporal coherence is to generate a mesh from scratch every frame. For comparison, we implemented a binary triangle tree algorithm that applies splits to the base mesh every frame, labeled *split-only* in the figure. The higher flight speeds tested were so fast that the animation did not appear smooth. Still, the CABTT algorithm performed better than the algorithm that did not use temporal coherence.

[5] Half of this, the vertices, are kept on the video card – ATI cards support ATI_element_array which allows everything to be stored on the video card.

8 RELATED AND FUTURE WORK

There are several optimizations, areas for future research, and related work to enhance the presented algorithm.

The isosphere error metric was constructed for efficient evaluation. It replaced the orientation sensitive metrics from Lindstrom, et al. [21] or ROAM [7] since they have been found to be too CPU intensive [4, 20]. Aggregation would reduce the CPU burden and benefit from fewer triangles to render. However, these metrics would complicate the split criteria for computing adaptive sub-triangulations.

ROAM has a dual-queue system that controls the number of triangles rendered and prioritizes the work performed for each frame. Adapting this system to the CABTT algorithm would allow rendering with hard real-time deadlines.

To handle high resolution texturing, we could precompute the resolution needed for each aggregate triangle. Chunks of texture data could then be swapped along with the geometry.

A common optimization, used by Lindstrom and Pascucci [20] for example, is to split rendering and geometry updates into separate threads. The implementation for the CABTT algorithm will be similar. Extra thread synchronization will be needed before freeing caches, however. Other threads could prefetch or precache texture or geometry data, based on possible future view points.

To render several instances of a single model, in addition to displacement map, texture map, and base mesh data, we can share caches between the instances. This requires a data structure that maps a position in the binary triangle tree to a reference count and cache identifier. Depending on the situation, it may help to have a separate cache for the lowest level of detail.

Using a fixed level of aggregation for aggregate triangles prevents very low levels of detail. For faraway objects, alternative level-of-detail approaches should be employed.

Lindstrom and Pascucci [20] give an indexing scheme appropriate for standard binary triangle trees with large data sets. The access patterns of our algorithm are somewhat different, but their approach should still work well. An approach tailored to CABTT would divide data into two sets. Low-resolution data is appropriate for splits, merges, and error metric evaluations. High-resolution data is needed for generating sub-triangulations.

The geometric error metric given in Section 5 will allow distant sand dunes to be approximated by a plane. This will reveal the valleys of the dunes when you should only be able to see the tops. Some research exists [2, 5, 12] that addresses this problem.

Several papers address automatically converting a model into a base mesh plus displacement data. MAPS [19] and Kobbelt, et al. [16] address standard remeshing. Other model representations appropriate for use with our algorithm are:

- normal meshes [11],
- displaced subdivision surfaces of Lee, et al. [18]; combined with the incremental evaluation scheme of Müller and Havemann [25], and
- spline patches with displacement maps by Krishnamurthy and Levoy [17], and Blow [4].

Once a base mesh has been found, Mitchell [24] has a method of consistently marking which edges are bases.

For models other than height fields, it makes sense to store culling or occlusion information for each aggregate triangle. Depending on the form of the displacement data, it should be straightforward to generate a frustum where the entire aggregate triangle is facing away from the view point.

Currently, the CABTT algorithm is only appropriate for rigid-body animation. Research into updating the error data for soft-body animation or other mesh updates would increase the applicability of the algorithm.

Geomorphing [13], morphing between level of detail changes, has not been included in our implementation. Geomorphing would require a significant amount of extra CPU work and bus bandwidth. On some cards, geomorphing could be implemented in a vertex shader program for a more modest speed hit.

9 ACKNOWLEDGMENTS

Special thanks to Rebecca Middleton, Ka-Ping Yee, and Georgia Saltsman for their help revising this paper. We are also grateful to Jonathan Blow and Thatcher Ulrich for sharing their algorithms and their insightful correspondence. Finally, the (anonymous) reviewers of this paper helped direct and improve this paper in numerous ways.

REFERENCES

[1] Randolph E. Bank, Andrew H. Sherman, and Alan Weiser. Refinement Algorithms and Data Structures for Regular Local Mesh Refinement. *Scientific Computing*, pages 3–17, 1983.

[2] Barry G. Becker and Nelson L. Max. Smooth Transitions between Bump Rendering Algorithms. In *ACM SIGGRAPH 93*, volume 27, pages 183–190. ACM, 1993.

[3] Jim Blinn. Simulation of Wrinkled Surfaces. In *ACM SIGGRAPH 78*, volume 12, pages 286–292. ACM, 1978.

[4] Jonathan Blow. Terrain Rendering at High Levels of Detail. In *Game Developers' Conference 2000*. CMP, 2000.

[5] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-Preserving Simplification. In *ACM SIGGRAPH 98*, pages 115–122. ACM, Jul 1998.

[6] Daniel Cohen-Or and Yishay Levanoni. Temporal Continuity of Levels of Detail in Delaunay Triangulated Terrain. In *IEEE Visualization '96*, 1996.

[7] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *IEEE Visualization '97*. IEEE, 1997.

[8] Jihad El-Sana and Amitabh Varshney. Generalized View-Dependent Simplification. *Eurographics '99*, 18(3), 1999.

[9] William Evans, David Kirkpatrick, and Gregg Townsend. Right Triangular Irregular Networks. Technical Report TR97-09, University of Arizona, 30 1997.

[10] Thomas Gerstner, Martin Rumpf, and Ulrich Weikard. Error Indicators for Multilevel Visualization and Computing on Nested Grids. *Computers & Graphics*, 24(3):363–373, 2000.

[11] Igor Guskov, Kiril Vidimce, Wim Sweldens, and Peter Schröder. Normal Meshes. In *ACM SIGGRAPH 2000*, pages 95–102. ACM, Jul 2000.

[12] Wolfgang Heidrich, Katja Daubert, Jan Kautz, and Hans-Peter Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *SIGGRAPH 2000*, pages 455–464. ACM, Jul 2000. Published as Computer Graphics Proceedings, Annual Conference Series, 2000.

[13] Hugues Hoppe. Progressive Meshes. In *ACM SIGGRAPH 1996*, pages 99–108. ACM, 1996.

[14] Hugues Hoppe. View-Dependent Refinement of Progressive Meshes. In *ACM SIGGRAPH 1997*, pages 189–198, 1997.

[15] Hugues Hoppe. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. In *IEEE Visualization 1998*, pages 35–42, Oct 1998.

[16] Leif P. Kobbelt, Jens Vorsatz, Ulf Labsik, and Hans-Peter Seidel. A Shrink Wrapping Approach to Remeshing Polygonal Surfaces. *Eurographics '99*, 18(3), Sep 1999.

[17] Venkat Krishnamurthy and Marc Levoy. Fitting Smooth Surfaces to Dense Polygon Meshes. In *ACM SIGGRAPH 96*, pages 313–324. ACM, Aug 1996.

[18] A. Lee, H. Moreton, and H. Hoppe. Displaced Subdivision Surfaces. In *ACM SIGGRAPH 2000*, pages 85–94. ACM, 2000.

[19] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. In *ACM SIGGRAPH 98*, pages 95–104. ACM, 1998.

[20] P. Lindstrom and V. Pascucci. Visualization of Large Terrains Made Easy. In *IEEE Visualization 2001*. IEEE, Oct 2001.

[21] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-Time, Continuous Level of Detail Rendering of Height Fields. In *ACM SIGGRAPH 96*, pages 109–118. ACM, Aug 1996.

[22] D. Luebke and C. Erikson. View-Dependent Simplification of Arbitrary Polygonal Environments. In *SIGGRAPH 97*, pages 199–208, 1997.

[23] David Luebke. A Survey of Polygonal Simplification Algorithms. Technical Report TR97-045, University of North Carolina at Chapel Hill, 1997.

[24] W. F. Mitchell. *Unified Multilevel Adaptive Finite Element Methods for Elliptic Problems*. PhD thesis, U.I. at Urbana CS Dept., 1988. Report No. UIUCDCS-R-88-1436.

[25] Kerstin Müller and Sven Havemann. Subdivision Surface Tessellation on the Fly using a versatile Mesh Data Structure. *Eurographics 2000*, 19(3):151–138, Aug 2000.

[26] Renato Pajarola. Large Scale Terrain Visualization Using The Restricted Quadtree Triangulation. In *IEEE Visualization '98*. IEEE, 1998.

[27] Alex Pomeranz. ROAM Using Triangle Clusters (RUSTiC). Master's thesis, U.C. Davis CS Dept., June 2000.

[28] Stefan Röttger, Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Real Time Generation of Continuous Levels of Details for Height Fields. In *Sixth International Conference in Central Europe on Computer Graphics and Visualization (Winter School on Computer Graphics)*. WSCG, Feb 1998.

[29] Ron Sivan and Hanan Samet. Algorithms for Constructing Quadtree Surface Maps. In *Proc. 5th Int. Symposium on Spatial Data Handling*, pages 361–370, August 1992.

[30] Thatcher Ulrich. Continuous LOD Terrain Meshing Using Adaptive Quadtrees. *Gamasutra*, Feb 2000. See http://www.gamasutra.com/features/20000228/ulrich_01.htm.

[31] Brian Von Herzen and Alan H. Barr. Accurate Triangulations of Deformed, Intersecting Surfaces. In *Proceedings SIGGRAPH 87*, pages 103–110. ACM SIGGRAPH, 1987.

[32] Julie C. Xia and Amitabh Varshney. Dynamic View-Dependent Simplification for Polygonal Models. In *IEEE Visualization 96*, pages 327–334, Oct 1996.