SMART:
An Efficient Technique for Massive Terrain Visualization from Out-of-core

Xiaohong Bao, Renato Pajarola, Michael Shafae

444 Computer Science building, University of California, Irvine, CA92627
Email: {xbao, pajarola, mshafae}@ics.uci.edu

Abstract

Real-time visualization of massive terrain elevation models is limited by expensive disk to memory I/O. Furthermore, level-of-detail rendering of such massive terrain models is hindered by complex per-vertex level-of-detail and culling tests. The concept of a per-vertex *live range* is introduced in this paper. Using this *live range*, the algorithm reuses per-vertex visibility computation from previously displayed frames efficiently during continuous view-dependent level-of-detail fly-over visualizations. This concept is simple but extremely effective. Experimental results show that more than 90% of vertex visibility computations and out-of-core access for those vertices can be avoided.

1 Introduction

Although there is a remarkable pace in the advance of computational resources and storage for real-time visualization the immensity of the input data continues to outstrip any advances. The task for interactively visualizing such a massive terrain is to render a triangulated mesh using a view-dependent error tolerance, thus intelligently and perceptually managing the scene's geometric complexity. At any particular instance in time (i.e. displayed frame), this *level-of-detail* (LOD) terrain surface consists of a mesh composed of hundreds of thousands of dynamically selected triangles. The triangles are selected using the current time-step's view parameters and the view-dependent error tolerance. Massive terrain data easily exceeds main memory storage capacity such that out-of-core rendering must be performed. This further complicates the triangle selection and terrain rendering owing to tertiary storage's relatively poor performance.

Consider a continuous fly-over of a massive terrain data. Arbitrarily choosing between any two consecutive frames, there exists a significantly large spatial coincidence. The proposed algorithm, *SMART*, dramatically improves the previous state-of-the-art by introducing the paradigm of a vertex *live range*. Each vertex has an associated live range. An a priori judgment can be made of what vertices need to be recalled from tertiary storage using a live range check. With this strategy, SMART exploits frame-to-frame coherence and generates an updated view-dependent LOD terrain representation by touching only those vertices that have changed between frames. Hence per-vertex LOD selection and culling costs are dramatically reduced. Moreover, this reduction directly reduces the amount of out-of-core I/O.

2 Previous Work

Many researchers have investigated efficient means to view-dependently update and render an LOD mesh. The current methods can be divided into two classes according to how the mesh updating transpires: *triangle-level updating* or *triangle-cluster-level updating*. Triangle-level updating is a per-triangle per-vertex mesh updating approach. For any change in the mesh's resolution, the LOD system will test and operate on vertices. Whereas triangle-cluster-level updating operates on aggregate triangle groups of two or more triangles. In other words, the LOD system will operate on triangle patches.

Each approach has benefits and drawbacks. Since triangle-level updating is per-vertex, this allows for fine-grain LOD vertex selection with implicitly given regular subdivision triangle mesh connectivity. For a given error tolerance, the number of vertices of an LOD mesh is minimized. Examples such as [10, 9, 19, 12, 4, 6, 7], are simple to im-

plement using a general strategy of a recursive top-down traversal of the multiresolution triangulation hierarchy.

Out-of-core terrain rendering methods based on triangle-level updating have been proposed in [13, 14, 5]. [14, 5] sort grids of the terrain data with interleaved quadtree, a vertex storage position on disk is determined by its position on a quadtree representing the terrain. This data layout is efficient for bin-tree and quadtree refinement. In this paper we realized our algorithms based on SOAR, an implementation of [5]. In [13], a vertex storage position is calculated from its position in the quadtree and its object space error, efficient out-of-core LOD vertex selection is then supported.

A recursive top-down traversal of the multiresolution triangulation hierarchy means that for every visible triangle, each vertex is re-evaluated for every rendered frame. In lieu of testing a large number of vertices, triangle-cluster based approaches have been proposed as a more efficient mesh updating alternative. In these methods such as [1, 16, 15, 3, 11], a set of triangles is clustered as one aggregate triangle to reduce LOD update computations; that is processing one patch of triangles rather than a large number of individual vertices. The speed-up costs increased algorithm complexity and loss of fine-grain LOD selection. Along cluster boundaries, the triangles are much more dense causing an uneven triangle distribution. Moreover, because of size limitations due to the triangle cluster granularity, many more triangles are rendered than necessary which offsets some of the gained performance, especially in the case that the graphics hardware is not very powerful. Due to coarse-grain refinement, jitters and popups of view are more serious.

A second avenue of improvement would be to re-use previously computed LOD and visibility information. A notable approach has been presented in [9] which re-uses previously computed LOD information by storing this information in two priority queues; one each for potential mesh simplification and refinement candidates. Only a limited number of high-priority mesh update candidates are accessed and re-evaluated for each frame.

Our method, SMART, does not maintain any sorting data structure, neither accesses the vertex itself, usually it is costly from out-of-core, if the vertex status is constant. SMART minimizes the per-vertex LOD and view culling tests to such a large degree that it is competitive to cluster-based LOD selection, however, offering at the same time fine-grain LOD vertex selection to generate smoother view with minimal number of necessary triangles.

3 View-dependent Terrain Triangulation

In this section we review and summarize the preliminary techniques for multiresolution terrain triangulation that form the basis LOD data structure used in our algorithm.

3.1 Triangle Bin-tree

Among the multiresolution terrain models reviewed in [8], we use the triangle bin-tree definition [9, 4, 14] here. In this context, mesh refinement is performed by recursively splitting a triangle at the midpoint of its longest edge [17] as illustrated in Figure 1. Hence this refinement rule defines a binary hierarchy on the triangles, or on the base vertices (midpoints of the longest edges) as shown in Figure 2. See [9, 4, 14] for more implementation details.

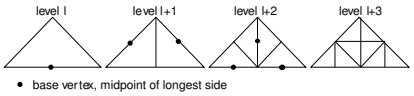

Figure 1: Recursive longest-side midpoint triangle subdivision.

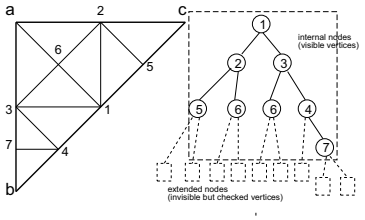

Figure 2: Vertex selection in a triangle bin-tree structure of a coarse triangle (a,b,c). All vertices of the inner (visible) nodes as well as the (next-level) extended nodes of the triangle bin-tree are tested.

3.2 Error Metric

The decision if a vertex v is selected in a particular frame is based on a view-dependent error-metric ρ_v that measures the geometric distortion of the LOD-mesh in screen-space. For each frame, all vertices

with ρ larger than some user given threshold τ, and the corresponding triangles, are selected to form the rendered LOD-mesh. As usual, we define the object-space error δ_v of any (base) vertex v as its vertical distance to the (longest-side) edge of the triangle it refines (see also Figure 1). The basic screen-space error-metric ρ_v is then defined as

$$\rho_v = \alpha \frac{\delta_v}{|P - v|}, \tag{1}$$

where P is the current viewpoint, and the factor $\alpha = \frac{m}{\phi}$ is based on the number of pixels on screen along the measured field-of-view (FOV) angle ϕ.

However, to allow efficient vertex selection by top-down traversal of the bin-tree, the error-metric has to be increasing monotonically in the multiresolution hierarchy. This can be achieved by an *error saturation* as introduced in [6] which can also be extended to view-dependent error-metrics as shown in [18, 14]. Hence we introduce a maximal ρ^{max} such that all vertices with $\rho^{max} < \tau$ can safely be ignored as:

$$\rho_v^{max} = \alpha \frac{\delta_v^{max}}{|P - v| - r_v^{max}}. \tag{2}$$

The object-space error δ_v^{max} is the saturated geometric error as in [6]. Additionally, we must also account for the saturated nested bounding sphere radius r_v^{max} of each vertex v [18, 14] to avoid cracks in the adaptive LOD triangulation. Both δ^{max} and r^{max} are easily computed in a pre-process by propagating the appropriate values bottom-up in the triangle bin-tree multiresolution hierarchy (Section 3.1), leaving each vertex' object-space error and radius equal to or greater than that of all its descendants[1] (see also [18, 14, 6]).

From here on we assume that the values δ_v, r_v and ρ_v of a vertex v are the maximal saturated values as described above. Therefore, given a user specified screen-space error threshold τ, the view-dependent LOD-mesh for a particular viewpoint P consists of all vertices with $\rho > \tau$, or from Equation 2 with

$$|P - v| < \frac{\alpha}{\tau}\delta_v + r_v. \tag{3}$$

[1] Of which there are four, as a refinement edge is shared by two triangles and there are two descendants in each triangle, see Figure 2

3.3 Refinement Cost

As illustrated in Figure 2, assuming there are I internal nodes corresponding to the set of selected vertices, then a complete refinement adds $E = I+1$ nodes or vertices (one-level binary tree extension). On the other hand, refining every of the T triangles of the current LOD means adding $E = T$ nodes. Therefore, the total number of nodes visited in a recursive top-down traversal is then:

$$N = E + I = 2T - 1. \tag{4}$$

Equation 4 and the fact that T is about twice the number of vertices in a triangle mesh clearly show that per-vertex tests are performed in an abundant way. In particular, even if no triangle is subdivided, nevertheless $E = T$ tests are performed. This testing performed for each rendered frame amounts to a significant time and possibly I/O cost.

4 LOD Mesh Updates

In this section we describe our new algorithm for optimized view-dependent LOD-mesh updates. Our goal is to avoid the repetitive cost and only perform LOD and view culling computation for the few vertices that are indeed candidates for updating the LOD-mesh for a given viewpoint.

Ideally, when we check a vertex's status, we want to know not only its status at the current moment. We would also like to know when its current status will possibly change, no matter how the view-dependent parameters will vary. This time period is called the *live range* of a vertex. To compute a vertex's live range, we define for each vertex a *safe-distance*. No matter how the view point moves, a vertex status keeps constant if the view point is within its safe distance. Only if the viewpoint invalidates the safe-distance to a vertex does its visibility status possibly change.

In the following we first introduce a vertex τ-safe-distance and culling-safe-distance. Then explain a vertex's live range and how to use it to replace costly floating point LOD and view culling computation.

4.1 τ-Sphere

Let's consider vertex LOD selection only in this section. With respect to Equation 3 let us first define

the τ-*sphere* of radius r_v^τ around a vertex v.

$$r_v^\tau = \frac{\alpha}{\tau}\delta_v + r_v. \quad (5)$$

A vertex v is only selected for any viewpoint P within its τ-sphere; where Equation 3 is satisfied. Therefore, regardless of how the viewpoint P moves, the vertex v remains rendered as long as P stays within the τ-sphere of v, see also Figure 3. Similarly, a vertex is always not selected as long as the view point stays outside of its τ-sphere. Therefore, when a vertex status is known, it will not change until the viewpoint crosses its τ-sphere.

The τ-sphere radius r_v^τ of a vertex v(Equation 5) is determined by the user given screen-space error tolerance τ, and the saturated geometric error δ_v and nested bounding sphere radius r_v. As τ changes infrequently during a visualization, it leaves r_v^τ of vertex v constant most of the time.

Let us examine a vertex v's LOD status at time t for viewpoint P_t and how its LOD status changes for subsequent viewpoints $P_{t+\delta t}$. For this we introduce the vertex v's τ-*safe-distance* $d_v^\tau(t)$ that defines if a vertex v is selected ($d_v^\tau(t) \geq 0$) or not ($d_v^\tau(t) < 0$) as:

$$d_v^\tau(t) = r_v^\tau - |P_t - v| \quad (6)$$

As illustrated in Figure 3, for a selected vertex v and viewpoint P_t with safe-distance $d_v^\tau(t) > 0$, a new viewpoint P_{t+1} with $d_1 = |P_{t+1} - P_t|$ does not alter the status of v as long as $d_v^\tau(t) - d_1$ is positive because

$$\begin{aligned} d_v^\tau(t+1) &= r_v^\tau - |P_{t+1} - v| \\ &\geq r_v^\tau - |P_t - v| - |P_{t+1} - P_t| \\ &= d_v^\tau(t) - |P_{t+1} - P_t|. \end{aligned} \quad (7)$$

The τ-safe-distance $d_v^\tau(t)$ measures the degree of freedom of unconstrained movements the viewpoint P can enjoy before the LOD status of a vertex v is potentially affected. The τ-safe-distance $d_v^\tau(t)$ is initialized when a vertex visibility is computed at the first time and is then estimated at $\bar{d_v^\tau}(t+\delta t) = d_v^\tau(t) - \sum_{i=1}^{\delta t} d_i$ with $d_i = |P_{t+i} - P_{t+i-1}|$. Only when the sign of $\bar{d_v^\tau}(t)$ changes, the actual $d_v^\tau(t)$ is re-evaluated. This conservative check guarantees that all high risk vertices are being found.

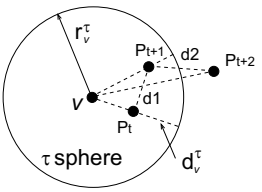

Figure 3: The concept of τ-sphere of a vertex v and its τ-safe-distance $d_v^\tau(t)$, given the view point moves from P_t, through P_{t+1}, to p_{t+2}. At P_{t+1}, $d_v^\tau(t) - d_1 \geq 0$, vertex v remains visible. But at P_{t+2}, $d_v^\tau(t) - d_1 - d_2 < 0$ causes re-evaluation of the τ-safe-distance d_v^τ at time $t+2$.

4.2 View Culling

The number of processed vertices is greatly reduced by view-frustum culling. This culling-visibility test is done simultaneously with the vertex LOD-visibility evaluation, and can be performed efficiently due to the nested bounding sphere hierarchy included in the triangle bin-tree structure (see also Section 3.2).

Essentially, view-frustum culling consists of verifying the distance of v and its bounding sphere to the bounding planes of the view frustum pyramid. Given the signed distance d_v^i, which is positive when the vertex is outside of the view volume, of vertex v to the current view frustum planes i at time t (see also Figure 4), we define the vertex v's *culling-safe-distance* $d_v^c(t)$ that defines if a vertex's bounding sphere is overlapping or within the view frustum ($d_v^c(t) \geq 0$) or not ($d_v^c(t) < 0$) as:

$$d_v^c(t) = \min_i (r_v - d_v^i) \quad (8)$$

Similar to the definition of the τ-safe-distance d_v^τ, d_v^c defines a conservative safe distance the view point can move before the culling status of vertex v is re-evaluated. Vertex culling-safe-distance is similarly computed along the nested bounding sphere hierarchy. For instance, if the bounding sphere of a vertex is in the view frustum, we set the culling-safe-distance of each of its descendants equal to its r_v without performing any further accurate computation.

4.3 User Movement

We classify the patterns of user navigation into two categories: translation and rotation. Any arbitrary motion can be deemed as a combination of those two. As illustrated in Figure 5, translation maintains

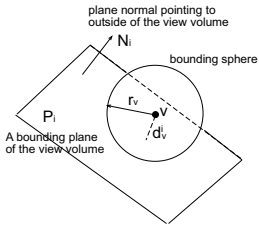

Figure 4: The concept of a vertex' bounding sphere and its culling-safe-distance $d_v^c(t) = \min_i(r_v - d_v^i)$. The view frustum plane P^i can move by $d_v^c(t)$ without changing vertex v's culling status.

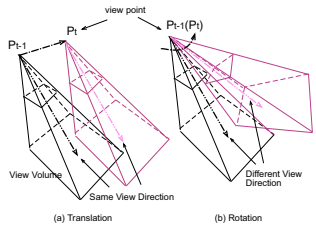

Figure 5: The two different categories of user navigation. Any arbitrary movement is a combination of translation and rotation.

a constant viewing direction while rotation keeps the viewpoint fixed.

For translation, a vertex' τ-safe-distance d_v^τ is bounded by Equation 7. Similarly, we bound the culling-safe-distance d_v^c using the following property:

$$|d_v^c(t+1)| \geq |d_v^c(t)| - d^{i'} \geq |d_v^c(t)| - |P_{t+1} - P_t| \quad (9)$$

where $d^{i'}$ is the maximal distance between the same bounding plane i of the view frustum for two different viewpoints P_t and P_{t+1}.

For rotation the coordinates of the viewpoint remain constant and hence the τ-safe-distance d_v^τ will not change for any vertex v. However, the culling-safe-distance d_v^c changes whenever the bounding planes of the view frustum rotate. Unfortunately this rotational change cannot be measured as uniformly as in Equation 9 for translation. Hence we have to re-compute the culling-safe-distance d_v^c in that case. However, we notice that because d_v^τ stays constant we only need to update d_v^c for vertices v with current $d_v^\tau \geq 0$.

We can define any user navigation with a linear combination of the above translation and rotation. In the case involving rotation, d_v^τ is checked and updated as that in translation, while d_v^c is forced to re-evaluate for a vertex if it is LOD-visible and its bounding sphere is not completely within the view volume. In practice, the number of these vertices is relatively small.

4.4 Dynamic Vertex Selection

For a translation, the safe-distances d_v^τ and d_v^c of a vertex v are both updated by the difference between consecutive viewpoints, according to the formula 7 and 9. If a vertex v is selected, both its d_v^τ and d_v^c are not negative. To be simple and uniform, combining the τ-safe-distance and culling-safe- distance we define the overall safe-distance of vertex v at moment t as

$$d_v(t) = \min(|d_v^\tau(t)|, |d_v^c(t)|). \quad (10)$$

To avoid updating safe-distance for each vertex at every moment, we then define the *live-range* of a vertex v as

$$L_v = f_t + \frac{d_v(t)}{S_{min}}, \quad (11)$$

where f_t is the frame number at time t, and S_{min} is the minimal speed the view point moves (per frame). Assume the positions of the viewpoint at time $t-1$ and t are P_{t-1} and P_t respectively , then f_t is updated as follows:

$$f_t = f_{t-1} + \lfloor \frac{|P_t - P_{t-1}|}{S_{min}} \rfloor. \quad (12)$$

The live range L_v of v defines the period that v's visibility status is constant. L_v is updated only when v's status is updated. f_t is calculated once for each frame. v's visibility status will not change at the moment t if $f_t < L_v$; otherwise L_v needs to be re-evaluated. Here L_v and f_t are both integers.

Even though we define a S_{min}, the low bound of speed here, the user's navigation speed can be less than S_{min} without violating the algorithm correctness. In that case the less the actual speed is than S_{min}, the more conservative the vertex L_v test will be.

Similarly we apply the concepts of f_t and vertex live range to rotation. The slight difference is that we use one bit to keep the sign of d_v^τ for a vertex v. If v's bounding sphere overlaps with the current view volume and d_v^τ is not negative, d_v^c is re-evaluated and L_v is updated if necessary. If the user motion is a combination of translation and rotation, we first check whether $f_t < L_v$ for a vertex

v. If it holds and the sign of d_v^τ is not negative, d_v^c is re-calculated and L_v is then possibly updated when v is not completely within the view volume. Otherwise if $f_t \geq L_v$, the vertex visibility status is re-computed and L_v is updated as in the case of translation.

The basic vertex LOD and culling status checking algorithms are summarized in Figure 6. We check each vertex in a depth-first traversal of the triangle bin-tree hierarchy. Given the return vertex status with respect to both LOD and culling, the traversal recursively subdivides triangles if necessary and generates the triangle-strip rendering primitive along the way as described in [14].

```
VerexScan-translation(v)
1   if (f_t ≤ L_v)
2       //do nothing.
3   else
4       compute d_v^τ, d_v^c;
5       update v.status;
6       update L_v;
7   return v.status;
```

```
VerexScan-rotation(v)
1   if (f_t ≤ L_v)
2       if (d_v^τ ≥ 0 &&
          v's bounding
          sphere intersects
          the view volume)
3           compute d_v^c ;
4           update
              v.status;
5           update L_v;
6   else
7       compute d_v^τ, d_v^c;
8       update v.status;
9       update L_v;
10      return v.status;
```

Figure 6: Pseudo-code for vertex selection.

4.5 Memory Management

With each vertex we store only its live range value and two bits for its visibility status and the sign of its τ-safe-distance respectively, 4 bytes in total. Similar to the technique in SOAR, the information of each vertex coordinates and other pre-computed values is loaded from out-of-core every time when needed. The vertices are organized in the binary refinement tree. This binary tree is then mapped to a fixed-length 1D array space in the main memory, shown in Figure 7, this array is called the bin-tree array. The vertices (or tree nodes) are ordered with the sequence they are exploited so that good data locality is reserved. The process of LOD mesh computation is a depth-traversal of this dynamically changed binary tree. At each moment newly split vertices are inserted into the end of the array sequentially.

When the 1D array is full at moment t, we create a new array of the same size. All vertices necessary for the moment t are copied from the old array and inserted to the new one sequentially. The old array is then destroyed. This strategy keeps the memory management simple and efficient because we need not to maintain a complex and expensive scheme to keep track of the usage of each slot in the array and to do garbage collection.

As we know, when the user navigation is continuous and smooth, relatively very few vertices are changed between two consecutive frames. With the array storing the binary refinement tree, we allocate a similar array called vertex array, to store the coordinate and the normal information for the checked vertices. We cache this vertex array to the graphics card memory. For each frame, therefore, only indices of the vertices in the triangle strip are passed to the graphics card. The vertex array is updated and handled exactly the same as the bin-tree array.

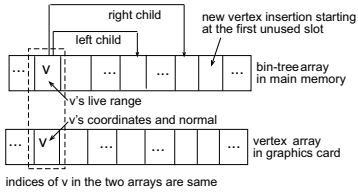

Figure 7: Illustration of vertex information storage in main memory. The vertices in the array are inserted and ordered naturally as their exploiting sequence.

5 Experimental Results

In this section we present the results of our algorithm and compare them with SOAR[2] with expensive morphing turned off. The computer we used is a DELL Dimension 8200 with P4 2.8GHz CPU, 512MB RAM , Windows XP system and nVIDIA GeForce FX5900 graphics card. In the experiments we have tested three techniques: SOAR, our algorithm (SMART) without vertex cache and our algorithm with vertex cache (SMART + Cache). With vertex cache, we store a copy of tested vertices in the memory of graphics card. The data set we used is over the Puget Sound area in Washington, made up of 8193×8193 elevation grid with 20 meters

[2]The source code of SOAR and the data set used in this paper are offered by Peter Lindstrom and Valerio Pascucci at http://www.cc.gatech.edu/gvu/people/peter.lindstrom.

horizontal and 0.1 meter vertical resolution. The data file on disk is organized with the technique of interleaved quadtree indexing used in SOAR, occupying 2GB disk space. The window size in all test cases are 800×600 pixels. The fly-over routes of all cases are identical.

Figure 8 demonstrates the effectiveness of our algorithm. When the screen error tolerance is one pixel, compared to SOAR, SMART performs less than 10% vertex visibility computation for most frames. This ratio is slightly beyond 20% in the worst case. This number shows that during the mesh computation, not only the 90% expensive visibility computations are avoided, but possible out-of-core vertex accessing for these vertices are also avoided. This lead to great improvement of the final rendering performance. The speedup over SOAR

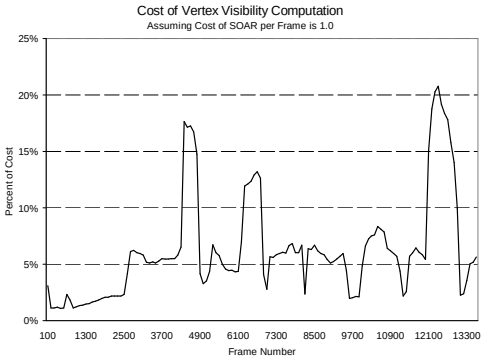

Figure 8: Compared to the technique in SOAR, the ratio of necessary vertex accesses and visibility computations with SMART along the flying route when the screen space error tolerance is 1 pixel and frame-to-frame incoherence is 0.61%.

for overall performance and pure mesh computation are displayed in Figure 9 for different screen space error tolerances, assuming the speed of SOAR is 1 respectively. Our algorithm can achieve more than 12 overall speedup when the screen space error tolerance is 1 pixel. The reason that higher speedup gained for more complex view is that costly visibility computation and out-of-core access for more vertices are avoided. SMART+Cache technique has higher speedup than SMART due to vertex caching and hardware-supported fast vertex array rendering. The process of pure mesh computation is the whole visualization process except OpenGL calls, which includes vertex visibility status computation

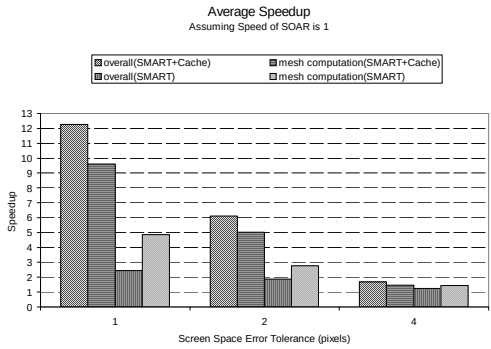

Figure 9: Overall and pure mesh computation speedups for different screen space error tolerance, comparing to SOAR, with frame-to-frame incoherence 0.61%..

and final triangle strip construction. It is noticeable that the speedup with SMART technique is also very high. Similar to the technique in SOAR, there isn't any vertex pre-computed information cached in main memory with SMART. This big speedup is purely because of the reducing of vertex visibility status computations and out-of-core vertex accesses with the technique SMART. Figure 10

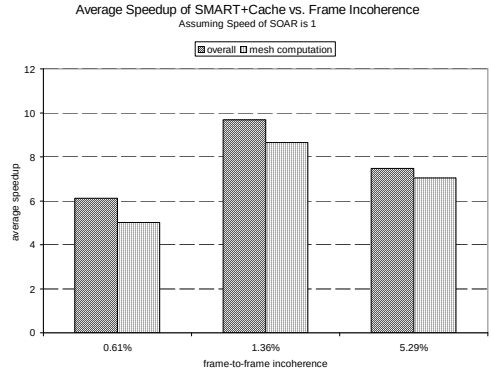

Figure 10: Overall and pure mesh computation speedup with SMART+Cache for different frame-to-frame incoherence when the screen space error tolerance is 2 pixels.

presents SMART+cache speedup over SOAR with different frame-to-frame incoherence. The average frame-to-frame incoherence is measured by the average percentage of changed triangles between consecutive frames. When the incoherence is 5.29%, in which the view point moves at around 20 km per

second, the overall speedup and pure mesh computation speedup are more than 7 over SOAR. They reach 8 and more when the frame-to-frame incoherence declines to 1.36%, due to much less floating number computations for updating vertex visibility status. The speedup is reduced to 6 when the incoherence is 0.61%. In this case good data locality is preserved with the data layout technique in SOAR, reducing the number of page faults, then improving the out-of-core data access efficiency for SOAR.

6 Conclusion

In this paper we presented a simple, easy-to-implement but very effective algorithm for interactive out-of-core terrain rendering. With this technique, the live range of each vertex is computed and kept when its visibility status is calculated for the first time. Later the costly visibility computation and real out-of-core access for this vertex are avoided for mesh computation when the live range of the vertex is not out-of-date. Due to reducing large amount of out-of-core I/O and converting more than 90% vertex visibility status computation, each of them involving several expensive floating point calculations, to a single integer comparison, great speedup is achieved. This algorithm is also suitable for those machines which have limited floating point computation capability. In the distributed or mobile environment, if we push vertex visibility computation to the server, the hardware configuration for client may be very simple. Further optimization techniques such as data prefetching, separate threads for rendering and data fetching, better methods for dynamic triangle strip updating, can also be applied to improve the performance more.

Acknowledgements

We would like to thank Peter Lindstrom and Valerio Pascucci for offering the source code of SOAR and the data set for Puget Sound area.

References

[1] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno, “BDAM - Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization”, *Proceedings EUROGRAPHICS 2003*, pp. 505-514, 2003.

[2] Hugues Hoppe, “Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering”, *Proceedings IEEE Visualization 98*, pp. 35-42, 1998.

[3] Joshua Levenberg, “Fast View-Dependent Level-of-Detail Rendering using Cached Geometry”, *Proceedings IEEE Visualization 2002*, pp. 259-266, 2002.

[4] 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”, *Proceedings SIGGRAPH 96*, pp. 109-118, 1996.

[5] Peter Lindstrom, and Valerio Pascucci, “Terrain Simplification Simplified: A General Framework for View-Dependent Out-of-Core Visualization”, *IEEE Transactions on Visualization and Computer Graphics*, pp. 239-254, vol 8, No. 3, 2002.

[6] Renato Pajarola, “Large scale Terrain Visualization using the Restricted Quadtree Triangulation”, *Proceedings IEEE Visualization 98*, pp. 19-26, 1998.

[7] Renato Pajarola, Marc Antonijuan, and Roberto Lario, “QuadTIN: Quadtree based Triangulated Irregular Networks”, *Proceedings IEEE Visualization 2002*, pp. 395-402, 2002.

[8] Renato Pajarola, “Overview of Quadtree-based Terrain Triangulation and Visualization”, Information & Computer Science, University of California Irvine, No. UCI-ICS-02-01, 2002.

[9] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Marc C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein, “ROAMing Terrain: Real-time Optimally Adapting Meshes”, *Proceedings IEEE Visualization 97*, pp. 81-88, 1997.

[10] Laurent Balmelli, Serge Ayer, and Martin Vetterli, “Efficient Algorithms for Embedded Rendering of Terrain Models”, *Proceedings IEEE International Conference on Image Processing ICIP 98*, pp. 914-918, 1998.

[11] Alex A. Pomeranz, “Roam using surface triangle clusters (Rustic)”, Master thesis, University of California at Davis, 2000.

[12] Thomas Gerstner, “Multiresolution Compression and Visualization of Global Topographic Data”, technical report, Institut für Angewandte Mathematik, Universität Bonn, No. 29, 1999.

[13] Xiaohong Bao, and Renato Pajarola, “LOD-based Clustering Techniques for Optimizing Large-scale Terrain Storage and Visualization”, *Proceedings SPIE Conference on Visualization and Data Analysis*,pp. 225-235, 2003.

[14] Peter Lindstrom, and Valerio Pascucci, “Visualization of Large Terrains Made Easy”, *Proceedings IEEE Visualization 2001*, pp. 363-370, 2001.

[15] Roberto Lario, Renato Pajarola, and Francisco Tirado, “HyperBlock-QuadTIN: Hyper-Block Quadtree based Triangulated Irregular Networks”, *Proceedings IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2003)*,pp. 733-738, 2003.

[16] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno, “Planet-Sized Batched Dynamic Adaptive Meshes (P-BDAM)”, *Proceedings IEEE Visualization 2003*, pp. 147-155, 2003.

[17] M. C. Rivara, “A Discussion on Mixed (Longest-Side Midpoint Insertion) Delaunay Techniques for the Triangulation Refinement Problem”, *Proceedings of the 4th International Meshing Roundtable*, pp. 335-346, 1995.

[18] Thomas Gerstner, “Top-Down View-Dependent Terrain Triangulation using the Octagon Metric”, technical report, Institute of Applied Mathematics, University of Bonn, 2003.

[19] Stefan Rottger, Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel, “Real-Time Generation of Continuous Levels of Detail for Height Fields”, Proc. WSCG ’98, pages 315-322, 1998.