2012 IEEE International Conference on Multimedia and Expo

Symmetric Cluster Set Level of Detail for Real-Time Terrain Rendering

John Judnich and Nam Ling
Department of Computer Engineering
Santa Clara University
Santa Clara, CA, USA
Email: jjudnich@scu.edu, nling@scu.edu

***Abstract*—In this paper, we present an improvement for batch-based quadtree terrain rendering that drastically reduces the number of draw calls to the graphics processing unit. As a result, more fine-grained triangular optimization is possible without sacrificing triangle throughput. No extra preprocessing is required. In general, quadtree terrain algorithms recursively subdivide mesh geometry to meet visual error constraints. Batch-based techniques use buffered grid blocks as the subdivision primitive for better triangle throughput. We base our algorithm on structural observations of such terrain quadtrees. First, we show that the four sub-nodes of any non-leaf can be categorized into sixteen distinct states of drawing behavior. These states are symmetric in such a way that allows just five unique geometries to represent all of them. With the additional observation that leaf nodes appear in groups of four across regions of homogeneous grid resolution, we develop a technique employing 23 unique geometric batches from which any terrain can be rendered. The resulting algorithm reliably reduces draw calls by a factor of 6 on average, and achieves render performance 30 to 50 percent faster than comparable techniques.**

***Keywords*-Terrain rendering; Data visualization; Geometry; Graphics; Rendering (computer graphics)**

I. Introduction

In many applications, efficiently rendering huge planet-sized terrains of extremely high-resolution is important. To do so requires adaptive triangulation by a level-of-detail (LOD) algorithm, through some process of progressive mesh refinement or coarsening. Such techniques serve to reduce an otherwise overwhelmingly high-resolution geometry into something the graphics processing unit (GPU) can manage. With integrated screen-space error metrics, rendering optimizations can be achieved without significantly reducing visual quality.

Since modern GPUs can render geometry much faster than the central processing unit (CPU) can generate it, CPU-based solutions are no longer practical. The solution is to cache geometry in video memory (VRAM), buffered in groups commonly called batches. Then, only a few draw calls are sent to the GPU every frame to render the scene.

The technique we present in this paper improves upon a popular terrain level-of-detail algorithm commonly called "Chunked LOD" [1] which renders using a quadtree of fixed-resolution mesh blocks. Our approach retains the flexibility of this method while improving its geometric batch efficiency, enabling performance more competitive to specialized techniques such as BDAM [2] and Geometry Clipmaps [3].

Our technique is lightweight and easy to implement. It achieves high quality results with negligible CPU and memory overhead, requires no extra preprocessing, and imposes no additional restrictions on how the terrain dataset is managed (integrating paging systems for out-of-core rendering is straightforward, for example).

II. Existing Approaches

Due to the wide range of approaches used for terrain rendering, a broad overview of the field is beyond the scope of this paper. For a more in-depth overview, refer to a survey of such algorithms [4]. In this section we outline a few relevant batch-based terrain rendering algorithms.

Two important techniques are Batched Dynamic Adaptive Meshes (BDAM) [2] and HyperBlock-QuadTIN [5]. These algorithms show that semi-continuous irregular triangular mesh optimizations can still be done efficiently, avoiding bottlenecks encountered by originally CPU-based techniques such as ROAM [6]. BDAM, for example, uses a top-down triangle refinement process, based on a bintree. Most importantly, it groups clusters of triangles into a hierarchy of regions. Each region can be grouped and buffered in VRAM. As a result, BDAM performs very well on modern hardware, and scales well to virtually any viewing range due to its hierarchical nature. Such algorithms have been adapted to render entire planets from any perspective [7]. Unfortunately, these techniques require heavy preprocessing on the terrain dataset.

In practice, regular grid triangulations are often preferred. These methods tend to require little or no preprocessing and are easily integrated with data management subsystems. Chunked LOD [1] is a simple and elegant technique, based on a quadtree structure of square terrain "chunks", each of which consist of a single batch in VRAM. At the lowest detail level, the terrain consists of a single chunk - an optimized $n \times n$ regular grid. As more detail is needed, the algorithm recursively swaps out chunks with four smaller chunks replacing each quadrant of the parent. This subdivision process is repeated to achieve any local level of detail desired. Chunked LOD performs quite well in

978-0-7695-4711-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICME.2012.178

practice, and is very easy to implement. With large enough n, the number of batches required to render very detailed terrains is fairly small.

Geometry Clipmaps [3] continues the trend towards comparatively brute-force terrain rendering. This technique renders the terrain using nested regular grids surrounding the viewer. Each successive outer grid uses the same vertex resolution but twice the width and length of the previous (inner) one. As the viewer moves along the terrain, these grids are progressively updated to surround the viewer while uploading the appropriate data. Unfortunately, the technique does not tolerate a dynamic field of view, and implementing planetary (spherical) terrains without distortion is problematic.

III. Batch Quadtrees

Recall that a batch is a hardware buffer containing indivisible geometry (vertex and tessellation information) which can be displayed with a single draw call. With batch-based rendering, visualized terrain is constructed entirely from these pre-loaded hardware buffers, although the particular method of organization varies depending on the technique.

Our optimized terrain rendering algorithm is based on a quadtree of square terrain blocks. Each block is a fixed size regular grid. In the traditional unoptimized version, a block is a single batch; each accounts for exactly one draw call. The terrain is rendered using a simple top-down traversal where each node is recursively subdivided into four smaller quadrant blocks if determined necessary by a screen-space error metric. Figure 1 illustrates this subdivision process.

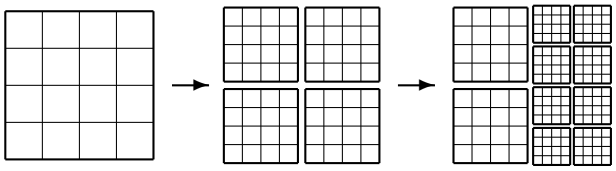

Figure 1. An example quadtree batch subdivision producing a simple multi-resolution terrain mesh. Note: The space seen between blocks is intended to illustrate the distinction between each unified batch. No spacing exists when rendering.

As one can see, the granularity of the resulting triangular refinement is determined by the size of block grids. In the case of Figure 1, each grid is 4x4 cells. In practice, grid size is usually set somewhere around 32x32 or sometimes 64x64, since much smaller sizes usually require too many draw calls to achieve the same overall resolution.

As the renderer runs, it is quite simple to calculate the hierarchical block tree immediately needed for the current frame. The root node is recursively subdivided until the desired screen-space error is satisfied for all leaf nodes. Screen-space pixel error ρ is straightforward to compute, given a block's maximum vertex error ε (world-space) [1]:

$$\rho = \frac{\varepsilon}{d}\left(\frac{w}{2\tan(\phi/2)}\right) \tag{1}$$

Where w is the width of the screen in pixels, ϕ is the horizontal field of view, and d is the distance from the viewer to the closest point on the block's bounding box or sphere.

A block's vertex error value can be computed in a number of ways. In general, this value represents the maximum vertical deviation from this particular block compared against the full resolution dataset. If this is too expensive to compute for whatever reason, an approximation can be synthesized. For example, for a non-root node n:

$$\varepsilon(n) = \varepsilon(\text{parent}[n])/2 \tag{2}$$

Equation 2 tends to maintain roughly screen-space uniform grid resolution regardless of terrain data.

IV. Block Cluster Sets

When the subdivision process is represented as a quadtree, only the tree's leaf nodes make up the actual rendered terrain. Since very large, detailed terrains require a considerable number of subdivisions, this results in block quadtrees that are quite deep, and consequently contain many leaf nodes. A small portion of a subdivided quadtree is shown in Figure 2 (each square represents an entire $n \times n$ grid block):

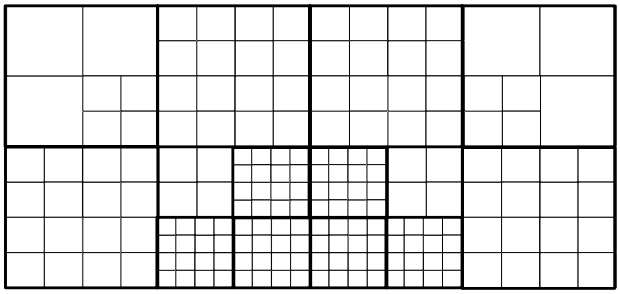

Figure 2. A region of a terrain block quadtree.

The quadtree in Figure 2 consists of 182 individual leaf nodes. Consequently, 182 draw calls are required to render the corresponding terrain. Our improved algorithm renders the above tree in exactly 14 draw calls. The optimized batch structure is indicated by regions surrounded by bold lines.

A. Single-Level Node Clusters

Since quadtree leaf nodes represent rendered $n \times n$ grid blocks, the depth of the tree at various points determines the corresponding local terrain resolution. As long as the viewer is sufficiently close to the terrain surface, the quadtree naturally becomes quite unbalanced in order to approximate a roughly screen-space uniform vertex density.

Based on the final leaf-state of a subdivided node's four children, we characterize 2^4 distinct subdivision states; each of the four quadrants ends up being either a leaf node or

not. We call these 16 states "single-level node clusters". All of these states are shown in Figure 3.

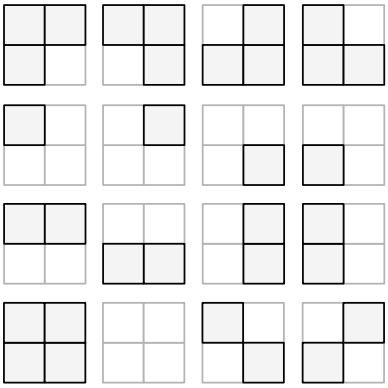

Figure 3. 16 possible "clusters" of terrain blocks resulting from one subdivision of one node. Shaded regions represent leaf nodes – rendered grid blocks.

Like pieces of a puzzle, instances of these 16 clusters can then be fit together to produce any nontrivial terrain quadtree. Finding the optimal cluster representation is computationally trivial, simply requiring a top-down traversal during which node-leaf patterns are matched and collapsed appropriately along the way (Figure 4). By storing geometric batches of each cluster type in video memory, any given cluster from the collapsed tree can be rendered with a single draw call.

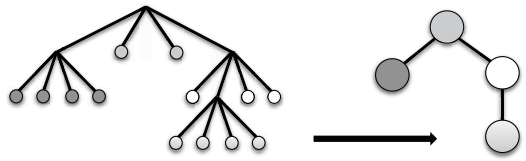

Figure 4. Collapsing block nodes into clusters during rendering.

Since a dynamically adapting quadtree results in frequent changes to which cluster type covers a given area, terrain height data cannot be directly embedded within the batched cluster geometry. Instead, height data is stored as displacement maps (textures), which are applied to the vertices of transformed clusters to achieve the desired terrain geometry.

B. Cluster Set Symmetry

For single-level node clusters, $2^{(2\times2)} = 16$ combinations of child-leaf states exist. However, not all 16 states need represented by their own cluster batch if we observe rotational symmetry. Apply a 90 degree rotation to any of the clusters shown in Figure 3, for example, and note that the resulting shape is also among the sixteen.

Thus, we can identify rotationally unique "seed" clusters from which symmetric cluster sets may be produced. The topmost four clusters in Figure 3 demonstrates one such case, forming a complete cluster set entirely reproducible from any one its members. In all, six seeds (Figure 5) are required to produce all sixteen clusters.

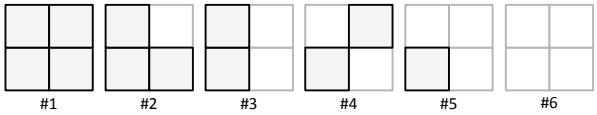

Figure 5. The cluster set "seeds" from which all of the 16 blocks (see Figure 3) can be produced.

Since the blank cluster (#6) represents no geometry to be rendered at all, the actual number of cluster geometries required is 5. Rendering is performed the same as before, but with the appropriate rotation applied to each cluster instance. Implementing this rotation is computationally negligible for the GPU, since a transformation matrix must be applied to cluster batch vertices whether or not rotation is used.

C. Two-Level Node Clusters

Enumerating all clusters for two levels of subdivision would further reduce draw calls. Unfortunately, this would result in $2^{(4\times4)}$ = 65,536 combinations. Rotational symmetry could reduce this by no more than a factor of four, and 16,384 types remains far too many to preload into video memory.

It is straightforward to prove that leaves of the reduced cluster tree are always of type #1 from Figure 5. By definition of a quadtree, every non-leaf node from the uncollapsed tree has exactly 4 children. Each non-leaf node will become collapsed into a node of 0, 1, 2, 3, or 4 children. Assuming no spare leaf nodes are left unaccounted for in the collapse, cluster type 1 is the only means of producing a collapsed node of 0 children – a leaf node.

Therefore, producing a set of two-level node clusters with the second level limited to cluster type #1 covers a very common case seen in practice (particularly across regions of homogeneous grid resolution as well as transitions between two adjacent resolutions).

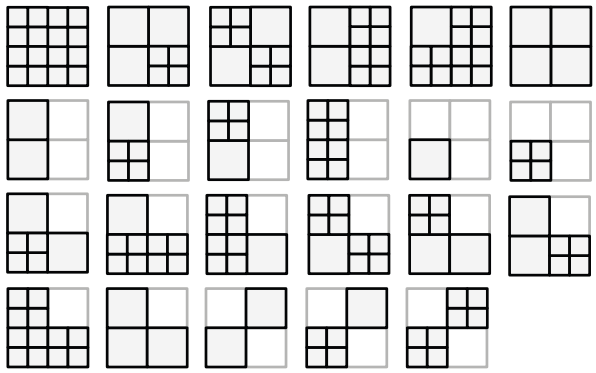

Figure 6. All 23 block clusters / batches required in VRAM.

Figure 6 shows all 23 cluster batches resulting from this reduced classification of two-level subdivisions.

With this relatively complex mapping from the leaf states to rotated batches, calculating the appropriate batch and its rotation during the render process is efficiently achieved with a lookup table.

V. LOD Transitions

A. Seams

When rendering clusters of differing resolution side-by-side, seams will result from the differing contour of each block's edges. We fill these in using a method known as edge walls, or skirts. The edge vertices of a batch are extruded downward, and filled with triangles creating a vertical "wall". Because the terrain is rendered with a constant upper bound on screen-space pixel error, it is guaranteed that the vertical fillers for these cracks will not exceed a certain on-screen height.

B. Interpolation

As the viewer's perspective moves over the terrain, blocks are constantly being subdivided and merged. Unless these transitions are smoothly interpolated, the viewer will notice "popping" artifacts due to the slight geometric deviations between the new and old grid resolutions.

With our cluster system, four individual quadrants per batch require interpolation between three resolutions (lower, normal, and double resolution for the subdivisions explained in Section 4C). The transition states for all four quadrants can be stored as a vector parameter $(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ passed to the vertex program, where $\alpha \in [0, 0.5]$ represents a transition from lower resolution to normal resolution and $\alpha \in [0.5, 1]$ represents the transition from normal resolution to double resolution. We assign each vertex a vector value $\mathbf{v_q}$ representing the quadrant to which the vertex belongs: $(1, 0, 0, 0)$ for the first quadrant, $(0, 1, 0, 0)$ for the second, and so on. The vertex program extracts the appropriate alpha value for a given vertex with eq. 3:

$$\alpha = (\alpha_1, \alpha_2, \alpha_3, \alpha_4) \cdot \mathbf{v_q} \tag{3}$$

Then, the final height value h is calculated as a combination of the three height functions h_0, h_1, and h_2 for relatively low, normal, and high resolutions, respectively (eq. 4).

$$\alpha_a = \text{saturate}(2\alpha), \text{ and } \alpha_b = \text{saturate}(2\alpha - 1) \text{ in } h = \mathrm{h}_2(u,v) \cdot \alpha_b + [\mathrm{h}_1(u,v) \cdot \alpha_a + \mathrm{h}_0(u,v) \cdot (1 - \alpha_a)] \cdot (1 - \alpha_b) \tag{4}$$

The "saturate" function clamps any input value to a range of $[0, 1]$, and u and v are coordinates to the height functions.

VI. Results

The following performance analysis measurements were performed using an NVIDIA GeForce GTX 285 and Intel Core 2 Duo 3.0 GHz processor. All tests were rendered at full 1080p (1920x1080 resolution), displaying a memory-resident terrain dataset of 8,192 $\times$ 8,192 points tiled to span over 500,000 $\times$ 500,000 vertices. It is textured with a "splat" mapped blend of four normal-mapped detail textures with dynamic lighting. Viewing range is not limited. See Figure 7 for a screenshot of this test scene. All measurements are averaged from 20 second periods, during which the viewer is moved along a predetermined path near the ground.

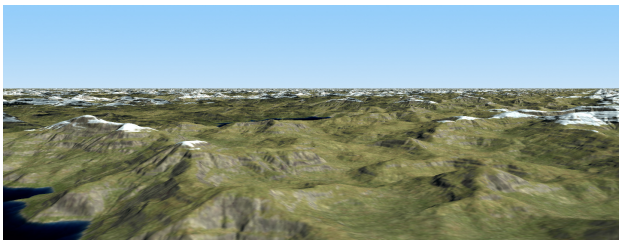

Figure 7. A screenshot of the terrain used in benchmarks.

A. Overall Performance

We compare two batch-based terrain rendering techniques. The first is a straightforward batch quadtree algorithm (CLOD), where each $n \times n$ grid block is rendered with exactly one draw call [8]. The second implements our algorithm (which we will refer to as Symmetric Cluster Set Level of Detail, or SCSLOD) with an identical screen-space error metric. Each technique achieves peak performance with different block grid sizes. Table 1 lists performance in frames per second (FPS) achieved using various block grid sizes.

Table I
Performance (FPS) for various block grid sizes, with screen-space error tolerance set at 1.0 pixels.

Block Size	8x8	16x16	32x32	64x64	128x128
CLOD	7	30	126	**253**	150
SCSLOD	49	174	**338**	n/a	n/a

Recall that for a given block size, SCSLOD clustering results in batches up to 4^2 times larger than CLOD. As Table 1 indicates, this results in SCSLOD reaching the GPU's index buffer size limit before CLOD does. Approaching this hardware limit is preferred, since it means we are utilizing the GPU near the maximum extent modern hardware allows while minimizing CPU-GPU synchronization overhead.

Measuring average frame rates using series of screen-space error tolerance values (ranging from 1.0 to 5.0 pixels) produces the plot seen in Figure 8.

Compared to CLOD-32, SCSLOD-32 renders the exact same scene (triangle for triangle) anywhere from 2.0 to 2.9

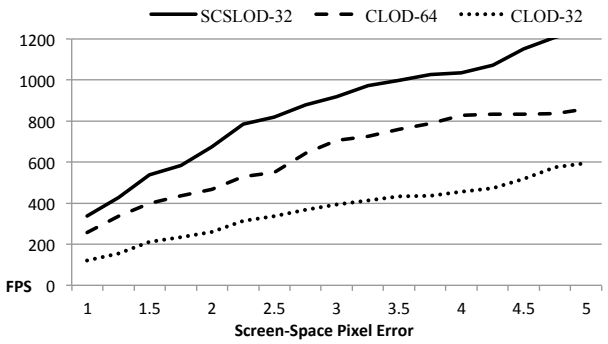

Figure 8. Average performance of three LOD techniques measured across a range of screen-space pixel error tolerances.

times as fast. However, since CLOD performance peaks with 64x64 blocks, it is more realistic to compare SCSLOD-32 with CLOD-64. In this case, SCSLOD reliably performs between 30% to 50% faster than CLOD. This is a significant improvement, especially when the reduction in overall batch and triangle count is considered (Section 6B).

Note that in many applications it is not necessary or desirable to display such high resolution terrains. As shown in Figure 8, SCSLOD scales quite well to lower detail levels as well, where even greater performance benefit is achieved. At higher error settings such as 10 pixels (not shown), SCSLOD becomes 80% faster than CLOD-64. However, we will focus particularly on high resolution comparisons in order to confirm that SCSLOD offers significant performance improvement even in the worst case.

B. Batch Count & Triangle Throughput

The measurements in Table 2 compare batch count, triangular complexity, and triangle throughput of relevant techniques. In this case we see that SCSLOD-32 is able to render triangles 2.7 times faster than CLOD-32. Compared to CLOD-64, throughput is reduced slightly (5%) with SCSLOD-32; however this is not a significant loss, as overall performance indicates.

Table II
TERRAIN SCENE GEOMETRY STATISTICS, GATHERED FROM THREE DIFFERENT TECHNIQUES.

Technique	Batches	Triangles	Triangles/sec
CLOD-32	421	1.0 million	165 million
CLOD-64	146	1.3 million	460 million
SCSLOD-32	69	1.0 million	438 million

SCSLOD-32 and CLOD-32 draw identical scenes, down to the exact triangular output and underlying block quadtree. In this case, SCSLOD reduces the batch count by a factor of 6 on average. CLOD-64 results in fewer batches than CLOD-32 as well; however, SCSLOD-32 uses less than half as many batches as CLOD-64 while simultaneously producing a more efficient geometric representation of the scene (using roughly 25% fewer triangles to satisfy the same screen-space pixel error constraints).

VII. CONCLUSION

We have presented an efficient algorithm that considerably reduces the number of draw calls required to render batch-based 3-D terrains. Compared to similar algorithms, batches are reduced by a factor of 6 on average. As a result, our algorithm enables corresponding improvement to the geometric optimality of the terrain mesh without significantly reducing overall triangle throughput. As we have shown, performance improvement is considerable, reliably achieving frame rates 30% to 50% higher than comparable techniques. The algorithm requires no additional preprocessing of terrain data.

ACKNOWLEDGMENT

The work of the first author is supported in part by the Kuehler Undergraduate Research Grant 2010 and Santa Clara University School of Engineering Dean's Fund.

REFERENCES

[1] T. Urlich, "Rendering massive terrains using chunked level of detail control," in *SIGGRAPH 2002 Course Notes*. ACM, 2002.

[2] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Bdam batched dynamic adaptive meshes for high performance terrain visualization," *Computer Graphics Forum*, vol. 22, no. 3, pp. 505–514, 2003. [Online]. Available: http://dx.doi.org/10.1111/1467-8659.00698

[3] F. Losasso and H. Hoppe, "Geometry clipmaps: terrain rendering using nested regular grids," *ACM Trans. Graph.*, vol. 23, pp. 769–776, August 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015799

[4] R. Pajarola and E. Gobbetti, "Survey of semi-regular multiresolution models for interactive terrain rendering," *Vis. Comput.*, vol. 23, pp. 583–605, July 2007. [Online]. Available: http://portal.acm.org/citation.cfm?id=1275117.1275121

[5] R. Lario and D. A. De, "Hyperblock-quadtin: Hyper-block quadtree based triangulated irregular networks," in *Proc. IASTED Visualization, Imaging and Image Processing*, 2003, pp. 733–738.

[6] 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 conference on Visualization '97*, ser. VIS '97. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997, pp. 81–88. [Online]. Available: http://portal.acm.org/citation.cfm?id=266989.267028

[7] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno, "Planet-sized batched dynamic adaptive meshes (p-bdam)," *Visualization Conference, IEEE*, p. 20, 2003.

[8] F. Strugar, "Continuous distance-dependent level of detail for rendering heightmaps," *Journal of Graphics, GPU, & Game Tools*, vol. 14, pp. 54–74, 2009.