Fast Multiresolution Terrain Rendering with Symmetric Cluster Sets

John Judnich[*] and Nam Ling[†]
Department of Computer Engineering, Santa Clara University
Santa Clara, CA 95053, U.S.A.

1 Batch Quadtree Terrain

The technique we present here improves upon a popular terrain level-of-detail algorithm which renders using a quadtree of fixed-resolution mesh blocks [Strugar 2009]. Our approach retains the flexibility of this method while improving its geometric batch efficiency, enabling performance more competitive to specialized techniques such as [Cignoni et al. 2003] and [Losasso and Hoppe 2004].

In the traditional version, a block is a single batch; each accounts for exactly one draw call. During rendering, a top-down traversal is used where each block node is recursively subdivided (Figure 1) to satisfy a screen-space error metric [Urlich 2002]. As one can see, the granularity of the resulting triangular refinement is determined by the depth of subdivision at a given point.

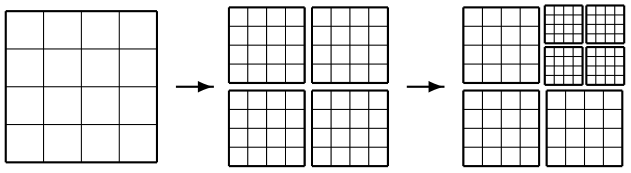

Figure 1: *Quadtree terrain batch subdivision. Note: Space shown between blocks is for illustration only and does not actually exist.*

2 Block Cluster Sets

When rendering realistically detailed terrains, block quadtrees become quite deep. Figure 2 shows a quadtree subtree, where each square represents an entire $n \times n$ grid block:

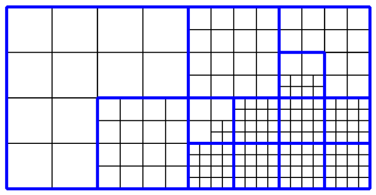

Figure 2: *A region of a terrain block quadtree.*

The quadtree in Figure 2 consists of 185 individual leaf nodes. As a result, **185** draw calls are required to render the corresponding terrain. We now present our improved algorithm, which renders this tree in exactly **13** draw calls. The regions surrounded by the bold lines in Figure 2 indicate the optimized batch structure.

2.1 Single-Level Node Clusters

When subdividing a single block node, exactly four sub-nodes are produced. Then, those four are recursively subject to subdivision until the desired level-of-detail is achieved. The final nodes rendered are always the leaves. Notice that such trees (of reasonable depth) are almost always unbalanced, due to the nature of the view-centric resolution gradient being approximated by the blocks.

Characterizing the subdivision of a node based on the final leaf-status of its children results in 16 distinct states, which we call "clusters" of blocks. All of these states are shown in Figure 3, with shaded squares representing leaf nodes.

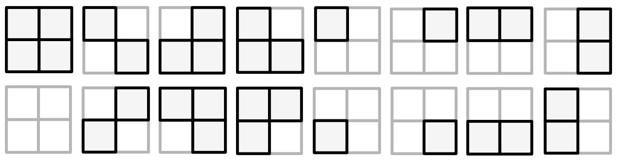

Figure 3: *16 possible "clusters" of terrain blocks resulting from one subdivision of one node. Shaded regions represent leaf nodes.*

Since the shaded regions of a cluster indicate known leaf nodes, they represent blocks (of fixed local resolution) to be drawn at that level. If appropriately gridded clusters are stored in video memory with one batch per cluster, the terrain can be rendered entirely by fitting instances of these 16 clusters together like pieces of a puzzle. But unlike a puzzle, finding the optimal construction of a block quadtree using these clusters is trivial; the tree is scanned top-down, collapsing matched node-leaf patterns along the way (Figure 4).

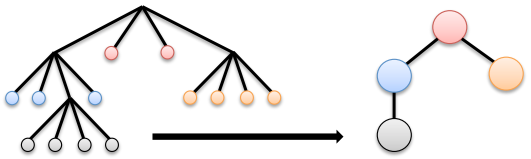

Figure 4: *Collapsing block nodes into clusters during rendering.*

It is important to note that since this approach frequently switches between using different cluster batches to cover the same area, the terrain height data cannot be embedded within the batch meshes themselves. Instead, terrain height data is stored in textures which are appropriately mapped to the batch meshes as they are rendered.

2.2 Cluster Set Symmetry

In Figure 3, there are $2^{(2 \times 2)} = 16$ different clusters – combinations of all possible leaf-states for a node's children. Generating one batch per cluster is unnecessary if we observe symmetry. For example, take any cluster, rotate it 90 degrees, and note that the resulting shape is also one of the sixteen.

We can build cluster sets which are completely reproducible from a single cluster by rotation. The rightmost four clusters in Figure 3 satisfy this property, for example. The complete set of sixteen clusters can be partitioned into six symmetric cluster sets, each of which can be reproduced with a single "seed" cluster (Figure 5).

Since the blank cluster (#6) results in no geometry being rendered at that level, the total number of cluster geometries required to represent all nontrivial trees is 5. Rendering is performed the same as before, but with the appropriate rotation added for each block.

[*]e-mail: jjudnich@scu.edu
[†]e-mail: nling@scu.edu

Copyright is held by the author / owner(s).
SIGGRAPH Asia 2011, Hong Kong, China, December 12 – 15, 2011.
ISBN 978-1-4503-0807-6/11/0012

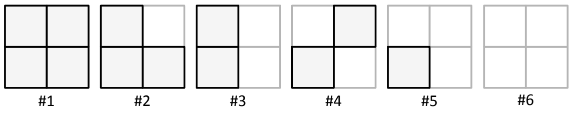

Figure 5: *The cluster set "seeds" from which all of the 16 blocks (see Figure 4) can be produced.*

2.3 Two-Level Node Clusters

The next logical step to further reduce draw calls would be to enumerate all clusters for up to two subdivision levels. However, it turns out this is impossibly huge with $2^{(4\times4)}$ = 65,536 batches.

It can be proved that leaves of the reduced cluster tree are always of type #1 from Figure 5. Informally, every original non-leaf node has 4 children, and becomes collapsed along with its children into a node of 0, 1, 2, 3, or 4 children in the reduced tree. Since the only way to produce a collapsed node of 0 children is through cluster type 1, all resulting leaf nodes are therefore of cluster type 1.

Draw calls can therefore be further reduced by merging these leaves into the set of cluster batches used.

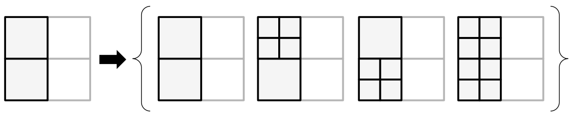

Figure 6: *Combinations from cluster type #3 (see Figure 6)*

Figure 6 shows the two-level subdivision combinations that result from cluster type 3. Figure 7 shows all 23 two-level cluster batches.

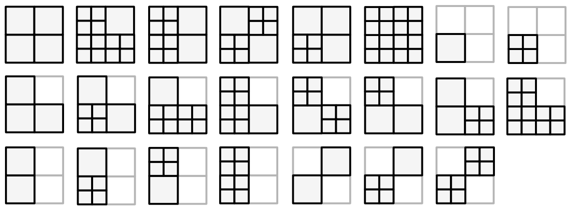

Figure 7: *All 23 block clusters / batches required in video memory.*

3 Results

In our benchmarks, we compare our optimized algorithm (which we refer to as Symmetric Cluster Set LOD, or SCSLOD) with the standard method using one draw call per leaf (CLOD), rendering a 500,000 × 500,000 terrain. Note that each technique achieves peak performance with different block grid sizes, as seen in Table 1. All benchmarks were rendered at 1080p with an NVIDIA GeForce GTX 285 and Intel Core 2 Duo E8400.

3.1 Overall Performance

Measuring average frame rates using series of screen-space error tolerance values produces the plot seen in Figure 8.

Comparing CLOD-64 with SCSLOD-32 (the fastest grid size choices for each), we see SCSLOD reliably performing between 30% to 50% faster than CLOD. At higher error settings such as 10 pixels (not shown), SCSLOD becomes 80% faster than CLOD.

Block Size	8x8	16x16	32x32	64x64	128x128
CLOD	8	34	140	**282**	167
SCSLOD	54	193	**375**	n/a	n/a

Table 1: *Performance (FPS) for various block grid sizes.*

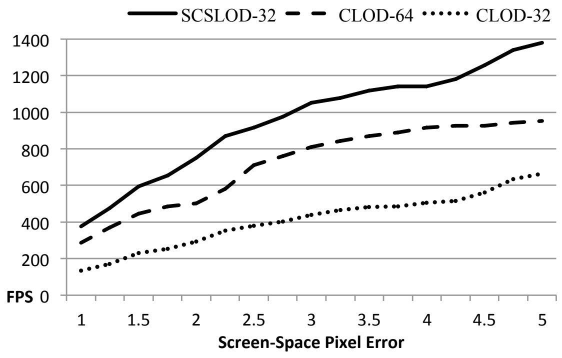

Figure 8: *Average performance in frames rendered per second (FPS) for three LOD techniques measured across a range of screen-space pixel error tolerances.*

3.2 Batch Count & Triangle Throughput

The measurements in Table 2 compare batch count, triangular complexity, and triangle throughput of relevant techniques.

Technique	Batches	Triangles	Triangles/sec
CLOD-32	420	0.9 million	167 million
CLOD-64	144	1.3 million	465 million
SCSLOD-32	68	0.9 million	440 million

Table 2: *Rendering statistics gathered from three techniques.*

We see that SCSLOD-32 is able to render triangles 2.6 times faster than CLOD-32. Compared to CLOD-64, throughput is reduced slightly (5%) with SCSLOD-32; however this is an acceptable tradeoff, as seen in Section 3.1.

SCSLOD-32 and CLOD-32 draw identical scenes in these tests, including exact triangular output as well as the underlying quadtree. In this case, SCSLOD reduces the batch count by a factor of 6 on average. CLOD-64 also reduces the number of batches from CLOD-32. However, not only does SCSLOD-32 use less than half the batches of CLOD-64, but the resulting scene uses less triangles to satisfy the same screen-space pixel error constraints.

References

CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., PONCHIO, F., AND SCOPIGNO, R. 2003. Bdam batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum 22*, 3, 505–514.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graph. 23* (August), 769–776.

STRUGAR, F. 2009. Continuous distance-dependent level of detail for rendering heightmaps. *Journal of Graphics, GPU, & Game Tools 14*, 54–74.

URLICH, T. 2002. Rendering massive terrains using chunked level of detail control. In *SIGGRAPH 2002 Course Notes*, ACM.