**[A level of detail method for blocky voxels](https://0fps.net/2018/03/03/a-level-of-detail-method-for-blocky-voxels/)**

Posted on [March 3, 2018](https://0fps.net/2018/03/03/a-level-of-detail-method-for-blocky-voxels/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

Large voxel terrains may contain millions of polygons.  Rendering such terrains at a uniform scale is both inefficient and can lead to aliasing of distant objects.  As a result, many game engines choose to implement some form of [level of detail](https://en.wikipedia.org/wiki/Level_of_detail) based rendering, so that distant terrain is rendered with less geometry.

In this post, I’ll talk about a simple technique based on vertex clustering with some optimizations to improve seam handling.  The specific ideas in this article are applied to Minecraft-like blocky terrain using the same clustering/sorting scheme as [POP buffers.](https://x3dom.org/pop/)

M. Limper, Y. Jung, J. Behr, M. Alexa: “[The POP Buffer: Rapid Progressive Clustering by Geometry Quantization](https://x3dom.org/pop/files/popbuffer2013.pdf)“, Computer Graphics Forum (Proceedings of Pacific Graphics 2013)

Review of POP buffers

Progressively Ordered Primitive (POP) buffers are a special case of vertex clustering, where for each level of detail we round the vertices down to the previous power of two.  The cool thing about them is that unlike other level of detail methods, they are [implicit,](https://en.wikipedia.org/wiki/Succinct_data_structure) which means that we don’t have to store multiple meshes for each level detail on the GPU.

When any two vertices of a cell are rounded to the same point (in other words, an edge collapse), then we delete that cell from that level of detail.  This can be illustrated in the following diagram:



Suppose that each vertex v_j \in V \subset \mathbb{Z}^3 has integer coordinates.  Define,

L_i(v) = 2^i \left \lfloor \frac{v}{2^i} \right \rfloor

This determines a [filtration](https://en.wikipedia.org/wiki/Filtration_(mathematics)) on the vertices,

V = L_0(V) \supseteq L_1(V) \supseteq ... \supseteq L_n(V) = \{ 0 \} 

Which extends to triangles (c_0, c_1, c_2) \in C \subseteq V^3 according to the rule,

P_i = \left \{ (c_0, c_1, c_2) \in C : L_i(c_0) \neq L_i(c_1), L_i(c_1) \neq L_i(c_2), L_i(c_2) \neq L_i(c_0) \right \}

And so it follows that,

C = P_0 \supseteq P_1 \supseteq ... \supseteq P_n = \emptyset

Each of the sets P_i represents the topology mesh at some level of detail, with P_0 being the finest, full detail mesh and P_n the coarsest.   To get the actual geometry at level i, we can take any j \leq i and compute,

M_i = \{ (L_i(c_0), L_i(c_1), L_i(c_2)) : (c_0, c_1, c_2) \in P_j \}

Using this property, we can encode the different levels of detail by sorting the primitives of the mesh from coarse-to-fine and storing a table of offsets:

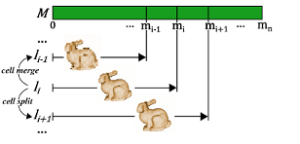


Figure taken from <https://x3dom.org/pop/>

To render the mesh at any level of detail we can adjust the vertex count, and round the vertices in the shader.

Building POP buffers

To construct the POP buffer, we need to sort the quads and count how many quads are in each LOD.  This is an ideal place to use [counting sort,](https://en.wikipedia.org/wiki/Counting_sort) which we can do in-place in O(n) time, illustrated in the following psuedo-code:

// Assume MAX\_LOD is the total number of levels of detail

// quadLOD(...) computes the level of detail for a quad

function sortByLOD (quads) {

const buckets = (new Array(MAX\_LOD)).fill(0)

// count number of quads in each LOD

for (let i = 0; i < quads.length; ++i) {

buckets[quadLOD(quads[i])] += 1

}

// compute prefix sum

let t = 0;

for (let i = 0; i < MAX\_LOD; ++i) {

const b = buckets[i]

buckets[i] = t

t += b

}

// partition quads across each LOD

for (let i = quads.length - 1; i >= 0; --i) {

while (true) {

const q = quads[i]

const lod = quadLOD(q)

const ptr = buckets[lod]

if (i < ptr) {

break;

}

quads[i] = quads[ptr]

quads[ptr] = q

buckets[lod] += 1

}

}

// buckets now contains the prefixes for each LOD

return buckets

}

The quadLOD() helper function returns the coarsest level of detail where a quad is non-degenerate.  If each quad is an integer unit square (i.e. not the output from a greedy mesh), then we can take the smallest corner and compute the quad LOD in constant time using a call to [count-trailing zeroes](https://en.wikipedia.org/wiki/Trailing_zero).  For general quads, the situation is a bit more involved.

LOD computation

For a general axis-aligned quad, we can compute the level of detail by taking the minimum level of detail along each axis.  So it then suffices to consider the case of one interval, where the level of detail can be computed by brute force using the following algorithm:

function intervalLOD (lo, hi) {

for (let i = 0; i <= 32; ++i) {

if ((lo >> i) === (hi >> i)) {

return i

}

}

}

We can simplify this if our platform supports a fast [count-leading-zeroes](https://en.wikipedia.org/wiki/Find_first_set#CLZ) operation:

function intervalLOD (lo, hi) {

return countLeadingZeroes(lo ^ hi)

}

Squashed faces

The last thing to consider is that when we are collapsing faces we can end up with over drawing due to rounding multiple faces to the same level.  We can remove these squashed faces by doing one final pass over the face array and moving these squashed faces up to the next level of detail.  This step is not required but can improve performance if the rendering is fragment processing limited.

Geomorphing, seams and stable rounding

In a voxel engine we need to handle level of detail transitions between adjacent chunks.  Transitions occur when we switch from one level of detail to another abruptly, giving a discontinuity.  These can edges be hidden using [skirts](https://procworld.blogspot.com/2011/10/playing-dice-with-mesh-simplification.html) or [transition seams](http://transvoxel.org/) at the expense of greater implementation complexity or increased drawing overhead.

In POP buffers, we can avoid the discontinuity by making the level of detail transition continuous, similar to 2D terrain techniques like ClipMaps or CLOD.  Observe that we can interpolate between two levels of detail using vertex morphing,

In the original POP buffer paper, they proposed a simple logarithmic model for selecting the LOD parameter as a function of the vertex coordinate x:

t(x) = b - \log ( \text{viewDist}(x) )

Where b is a bias parameter (based on the display resolution, FOV and hardware requirements) and viewDist is a function that computes the distance to the vertex from the camera.  Unfortunately, this LOD function is discontinuous across gaps due to rounding.

The authors of the original POP buffer paper proposed modifying their algorithm to place vertices along the boundary at the lowest level of detail.  This removes any cracks in the geometry, but increases LOD generation time and the size of the geometry.

Instead we can solve this problem using a stable version of LOD rounding.  Let M be the maximum LOD value for the chunk and all its neighbors.  Then we compute a fixed point for t:

t_0 = M

t_n = t(L_{t_{n - 1}}(x))

In practice 2-3 iterations is usually sufficient to get a stable solution for most vertices.  This iteration can be implemented in a vertex shader and unrolled, giving a fast seamless level of detail selection.

As an aside, this construction seems fairly generic.  The moral of the story is really that if we have geomorphing, then we don’t need to implement seams or skirts to get crack-free LOD.

World space texture coordinates

Finally, the last issue we need to think about are texture coordinates.  We can reuse the same texturing idea from greedy meshing.  For more info see [the previous post](https://0fps.net/2013/07/09/texture-atlases-wrapping-and-mip-mapping/).