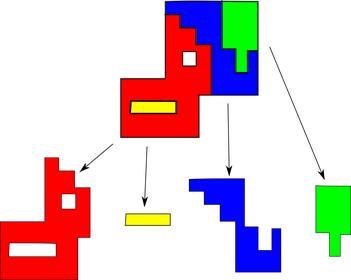
[**Meshing in a Minecraft Game (Part 2)**](https://0fps.net/2012/07/07/meshing-minecraft-part-2/)

Posted on [July 7, 2012](https://0fps.net/2012/07/07/meshing-minecraft-part-2/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

[Last time](https://0fps.wordpress.com/2012/06/30/meshing-in-a-minecraft-game/), I wrote about how to generate meshes in a Minecraft-style voxel engine.  I got a lot of interesting feedback, and so today I’m going to do a follow up highlighting some of the points that came up in the various discussions.  I’ll also talk about yet another interesting approach to generating meshes, which occurred to me in the mean time.  But first, I would like to take a moment to address some specific comments relating to greedy meshing:

Multiple Voxel Types

By far, the most frequent question about greedy meshing was how to extend it to handle multiple voxel types and different normal directions.  This surprised me somewhat, though it is perhaps understandable since I did not spend much time explaining it in the previous post.  The general idea is pretty simple.  What you do is group blocks together according to their type, and then do the meshing on each part separately:



That’s it!  This same idea works for normals too.  All you need to do is add an extra bit to keep track of the orientation.  To show that this isn’t so difficult, I made an [updated demo](http://mikolalysenko.github.com/MinecraftMeshes2/) that shows how some of these methods work for different voxel types.

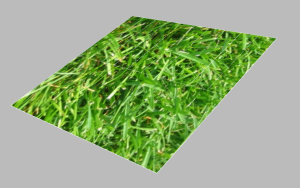
T-Junctions

Now this one is pretty bizarre.  By far the most common criticism of the greedy method I saw on reddit was that the meshes contain many T-vertices.  I’m not really sure why so many people latched on to this concept, but it is certainly true that greedy meshing will produce them.  What is less clear though is what the harm of having a T-junction is.

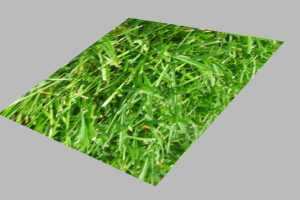
Intuitively, one would assume that as long as you have a pair of colinear edges, subdividing one of them shouldn’t make much difference (at least that’s what one would hope anyway).  After thinking fairly hard about rendering, I couldn’t come up with a good reason why this would be a problem, and so I decided to code up a demo to test it:

[Click here to see the T-Junction Demo!](http://mikolalysenko.github.com/TJunctions)

Surprisingly, no matter how I messed with the parameters I couldn’t get any visible discontinuity to show up!  Here are some pictures:

[](https://0fps.files.wordpress.com/2012/07/tjunctionedges.png)

The mesh with edges drawn on to show the T-junctions.

[](https://0fps.files.wordpress.com/2012/07/tjunctionnoedges.png)

The mesh without edges drawn on.

In summary, I’m not sure that there is actually a problem with T-junctions.  You can try out the demo yourself if you aren’t convinced.  If anyone can come up with a plausible scenario where one of the greedy meshes has visible rendering artefacts, I’d be quite interested to see it.  I’ve heard old ghost stories that on some ancient ATi cards, when the moon is full, these type of cracks actually do appear and make a big difference; but I unfortunately do not have access to such hardware.  If you can manage to get a crack to show up in this demo (using WebGL), please take a screen shot and post a comment.

From Quads to Triangles

That much concludes what I have to say about greedy meshing.  The next thing I want to discuss is an efficient alternative to greedy meshing.  The inspiration for this idea comes from a comment made by fr0stbyte on the last post.  Implicitly, he asked the question of whether triangles could be used to create better meshes than quads.  Thinking about this made me realize that there is a much more direct way to optimize Minecraft-type meshes using standard computational geometry techniques.  Ultimately, we can just think of this problem as just plain-old ordinary [polygon triangulation](http://en.wikipedia.org/wiki/Polygon_triangulation), and solve it using purely classical methods.  Starting from scratch, it is by no means a trivial thing to figure out how to triangulate a polygon, however it is by now very well understood and something of a standard technique (which makes me feel very foolish for having overlooked such a basic connection earlier 😐 ).

One of the most frequently used methods for polygon triangulation is the monotone decomposition algorithm due to Fournier et al.  The basic algorithm proceeds in two steps:

1. Decompose your polygon into [monotone polygons](http://en.wikipedia.org/wiki/Monotone_polygon).
2. Triangulate each monotone polygon.

The second step can be done in optimal linear time on the number of vertices of the monotone polygon, and is quite well documented elsewhere.  I won’t spend any time discussing it here, and instead I’ll just point you to a standard [tutorial](http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/PolyPart/polyPartition.htm) or [text](http://maven.smith.edu/~orourke/books/compgeom.html) [book](http://www.cs.uu.nl/geobook/).  The more interesting problem in this context is how to do the first step efficiently.  It can be shown that for non-simple polygons (of which our regions generally are), it requires at least \Omega(n \log(n)) operations to perform a monotone subdivision, where n is the number of vertices.

However, in the case of Minecraft-style meshing, we can actually do much better.  The key thing is to realize that the number of voxels is strictly much greater than the number of vertices.  This suggests that we can cover the cost of generating the monotone subdivision by the fixed cost of walking along the voxels, and still get an O(n) algorithm out at the end of the day (where this n is the number of voxels, not the number of vertices).  One way that we could do both at the same time is to adapt the standard sweep line algorithm for monotone decomposition to process the polygon as we march over the volume.  Here is how this works in pseudocode for a single slice:

1. Initialize polygons, frontier to empty list
2. For each scan line:
   1. Run length encode scan line
   2. Set pf = pointer to start of frontier,  pr = pointer to start of runs
   3. While pf and pr are not at the end of their respective lists:
      1. Let r = \*pr and p = \*pf be the run and polygon at current positions
      2. If r overlaps with the bottom of pf and is the same type:
         1. Merge r with p
         2. Increment pr and pf and continue
      3. If the end of r is past the bottom right of p:
         1. Close off p and remove from frontier
         2. Increment pf
      4. If the bottom end of p is past the end of r:
         1. Turn r into a new monotone polygon
         2. Increment pr
3. Close off all remaining polygons on the frontier

That looks like a terrible mess, but hopefully it makes some sense.  If it helps, here is some actual [javascript](https://github.com/mikolalysenko/mikolalysenko.github.com/blob/master/MinecraftMeshes2/js/monotone.js) which implements this method.  The time complexity of this approach is dominated by the cost of voxel traversal, and so it is O(n) again.

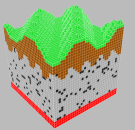
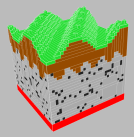
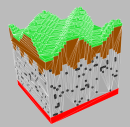
WebGL Demo!

Without further ado, here is the link:

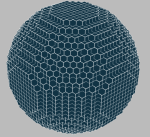
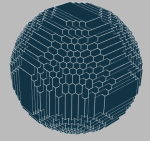
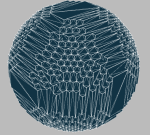
[Click here to see the WebGL demo!](http://mikolalysenko.github.com/MinecraftMeshes2/)

The main changes from last time are the addition of different colors and types for voxels, the extension to handle orientations and the addition of a new algorithm.  Here are some pictures for comparison:

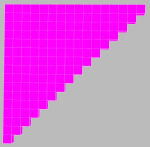
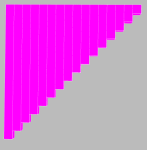
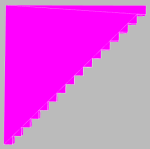
Left: Naive culling, Middle: Greedy,  Right: Monotone

[](https://0fps.files.wordpress.com/2012/07/naiveterrain.png)     [](https://0fps.files.wordpress.com/2012/07/greedyterrain.png)    [](https://0fps.files.wordpress.com/2012/07/monotoneterrain.png)

Naive: 26536 verts, 6634 quads. Greedy: 7932 verts, 1983 quads.  Monotone: 7554 verts, 4306 tris.

[](https://0fps.files.wordpress.com/2012/07/naivesphere.png)    [](https://0fps.files.wordpress.com/2012/07/greedysphere.png)     [](https://0fps.files.wordpress.com/2012/07/monotonesphere.png)

Naive: 19080 verts, 4770 quads.  Greedy: 8400 verts, 2100 quads.  Monotone: 8172 verts, 4572 tris.

[](https://0fps.files.wordpress.com/2012/07/screenshot-at-2012-07-07-161827.png)  [](https://0fps.files.wordpress.com/2012/07/greedytriangle.png)  [](https://0fps.files.wordpress.com/2012/07/monotonetriangle.png)

Naive: 1344 verts, 336 quads.  Greedy: 264 verts, 66 quads.  Monotone: 204 verts, 104 tris.

While monotone meshing is at least conceptually straightforward, the mesh quality isn’t noticeably different.  One thing to keep in mind is that the primitive counts for the monotone mesh are in triangles, while the other two measurements are given in quads.  Thus, there is a factor of two difference between the quantities.  In all of the examples, the monotone mesh had the fewest vertices.  The reason for this is that the monotone triangle decomposition code pushes all the vertices up front, while the greedy mesher emits a new set of vertices per each quad.  It would be interesting to see if the greedy mesh vertex count could be reduced using some similar method.  On the other hand, the adjusted primitive count (taking each quad as 2 triangles) swings both ways.  On the terrain example, greedy meshing was a little better, while on the triangle example monotone wins by a similar margin.

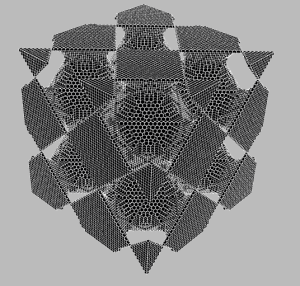
In the end it is hard to declare a clear winner from this data.  Both greedy and monotone meshing have their advantages and draw backs, and there are situations where the primitive count advantage can swing easily from one to the other.  One slight edge should be given to the greedy method in terms of code complexity, though only barely.  On the other hand, if your rendering is vertex shader limited, you may gain some small FPS boost by switching to monotone meshing since the vertex count is typically lower.  This saves a bit of GPU memory, which is always nice, but the savings are so small that I’d have a hard time believing it is that big of a deal.

Overtime Shoot Out

To break the stalemate, let’s race the different meshers against each other.  Recall that in terms of performance, the cost of each algorithm can be broken down into two factors:

1. The size of the underlying voxel grid, n.
2. The number of surface primitives, k.

One way to think of these two parameters is that O(n) measures the overhead of running any algorithm on its own, (that is the cost of running the algorithm on an empty volume), while O(k) measures the algorithm dependent overhead which is determined by the complexity of the surface and the type of mesh generation.  Since it is mesh generation that we want to study, ideally we’d like to keep n fixed and let k vary.  Another way to say this is that we want to construct some family of volumes with ever increasing surface areas.  A simple way to do this is to use trig functions of increasingly higher frequency.  For example, consider the following volume which is described b a functional equation:

[](https://0fps.files.wordpress.com/2012/07/benchmark.png)

\sin( \frac{\omega}{2 \pi} x ) + \sin( \frac{\omega}{2 \pi} y ) +\sin( \frac{\omega}{2 \pi} z )< 0.

As \omega increases, the number of chambers increases as well, along with the surface area.  This gives us a pretty good way to control for the complexity of surfaces in an experiment.  I implemented this idea in a [node.js script](https://github.com/mikolalysenko/mikolalysenko.github.com/blob/master/MinecraftMeshes2/js/benchmark.js) that takes as input a mesher and runs it on a volume of some specified size.  To control for cache warm up and JIT issues, the script runs each program  some number of iterations on a non-empty volume (in my experiments, I found 10 iterations to be sufficient on a volume where \omega=4).  Garbage collection costs are amortized by repeated runs (10 in this case).  All experiments were performed on a 65x65x65 volume with 0 \leq \omega \leq 10.

Here is the data from running this benchmark on the naive culling method:

Naive Meshing Results:

0 81.80 0 0

1 129.05 82488 20622

2 147.85 114696 28674

3 166.50 146016 36504

4 180.80 178792 44698

5 206.10 209256 52314

6 208.45 243672 60918

7 258.85 272304 68076

8 267.60 306640 76660

9 278.45 334968 83742

10 297.15 371496 92874

The first column is \omega.  The second column is the average time required to mesh the volume in milliseconds.  The third and fourth columns are the number of vertices and faces respectively.  These results shouldn’t be too surprising.  As the frequency increases, the number of surface elements goes up, and so it ends up taking longer to generate the mesh.  In the 0 frequency case, you get an empty volume, and so the time required is reduced to just the overhead of iterating over the volume.  Just for fun, here are the results for stupid meshing:

Stupid Meshing Results:

0 6.95 0 0

1 2733.65 3293184 823296

2 2848.05 3292128 823032

3 2727.35 3293184 823296

4 2673.40 3289032 822258

5 2729.50 3293184 823296

6 2741.10 3293088 823272

7 2687.75 3293184 823296

8 2729.20 3286512 821628

9 2682.40 3293184 823296

10 2772.95 3293136 823284

This may at first seem a little bizarre, but it makes sense.  For stupid meshing, iterating over the volume is basically `free’.  The only limit at the end is the memory bandwidth.  Another weird thing is that the number of primitives in the stupid mesh does not scale with surface area, but rather with volume.  In this case, the volume of each region is relatively constant and so the run-time quickly spikes to 2.7 seconds or so, then stays flat as the frequency changes.

Anyway, let’s now get to the main point, which is how well greedy and monotone meshing stack up:

Greedy Meshing: Monotone Meshing:

0 92.40 0 0 0 79.00 0 0

1 99.10 20712 5178 1 92.20 20202 11034

2 103.10 44068 11017 2 110.10 43326 23631

3 110.35 61644 15411 3 122.30 60420 32778

4 126.00 87984 21996 4 144.60 86328 47319

5 134.25 102024 25506 5 155.80 100056 54033

6 151.40 129344 32336 6 192.10 127074 68871

7 153.60 142416 35604 7 197.40 139476 75273

8 167.85 172140 43035 8 227.60 167410 92843

9 164.90 182256 45564 9 239.60 178302 96081

10 198.30 213452 53363 10 297.60 209838 113559

A victory for greedy meshing!  Not only did it beat monotone meshing, but for sufficiently complicated geometry it is also the fastest method overall.  Though this is not exactly a fair apples-to-apples comparison, since greedy meshing spits out quads while monotone meshing generates triangles.  This fact alone is enough to account for nearly a 30% difference in performance, and explains some, but not all of the discrepancy between these two charts.  The remainder of the overhead is most likely due to the fact that Monotone meshing is a more complex algorithm and that it has to do a bit more work per triangle, while greedy meshing is a bit simpler, but does more work per voxel.

While I believe that the general conclusions of these benchmarks are valid, I wouldn’t say that this definitively proves greedy meshing is faster than monotone meshing.  Javascript is far from the best language to use for doing benchmarks, and these implementations are not particularly tuned for performance.  There may also be subtle bugs lurking in the code somewhere, like silent out of bounds accesses, that could end up having disastrous consequences for performance (and account for nearly the entire gap between the two algorithms).  I’ve also tried to control for as many variables as I could, but it is still possible that I overlooked something and that there is some systemic bias in these results.

Conclusion

We’ve now seen at least two different ways to do better than naive meshing in Minecraft, and there are probably many others out there.  Portponky from r/gamedev sent me some C++ code to generate meshes, though I’ve not yet looked at it carefully enough to figure out how to port it to Javascript (sorry!).  I am still quite curious about it though, since it seems to be doing something quite different than either the greedy method or monotone triangulation.  Anyway, that’s all for now.  If you have any questions or other interesting thoughts you’d like to share, please leave a comment!