[**Meshing in a Minecraft Game**](https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/)

Posted on [June 30, 2012](https://0fps.net/2012/06/30/meshing-in-a-minecraft-game/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

The [last post I wrote on Minecraft-like engines](https://0fps.wordpress.com/2012/01/14/an-analysis-of-minecraft-like-engines/) got a lot of attention, and so I figured it might be interesting to write a follow up.  This time I’ll try to tackle a different problem:

Meshing

One of the main innovations in Infiniminer (and Minecraft) is that they use polygons instead of raycasting to render their volumes.  The main challenge in using polygons is figuring out how to convert the voxels into polygons efficiently.  This process is called *meshing*, and in this post I’m going to discuss three different algorithms for solving this problem.   But before doing this, we first need to say a bit more precisely what exactly meshing is (in the context of Minecraft maps) and what it means to do meshing well.  For the sake of exposition, we’ll consider meshing in a very simplified setting:

* Instead of having multiple block types with different textures and lighting conditions, we’ll suppose that our world is just a binary 0-1 voxel map.
* Similarly we won’t make any assumptions about the position of the camera, nor will we consider any level of detail type optimizations.
* Finally, we shall suppose that our chunks are stored naively in a flat bitmap (array).

I hope you are convinced that making these assumptions does not appreciably change the core of the meshing problem (and if not, well I’ll try to sketch out in the conclusion how the naive version of these algorithms could be extended to handle these special cases.)

In a typical Minecraft game chunks do not get modified that often compared to how frequently they are drawn.  As a result, it is quite sensible to cache the results from a mesher and only ever call it when the geometry changes.  Thus, over the lifetime of a mesh, the total amount of computational work it consumes is asymptotically dominated by the cost of rendering.

Recall that at a high level polygon rendering breaks down into two parts: 1.) primitive assembly, and 2.) scan conversion (aka polygon filling).  In general, there really isn’t any way to change the number of pixels/fragments which need to be drawn without changing either the resolution, framerate or scene geometry; and so for the purposes of rendering we can regard this as a fixed cost (this is our second assumption). That leaves primitive assembly, and the cost of this operation is strictly a function of the number of faces and vertices in the mesh.  The only way this cost can be reduced is by reducing the total number of polygons in the mesh, and so we make the following blanket statement about Minecraft meshes:

Criteria 1: Smaller meshes are better meshes.

Of course, the above analysis is at least a little naive.  While it is true that chunk updates are rare, when they do happen we would like to respond to them quickly.  It would be unacceptable to wait for more than a frame or two to update the displayed geometry in response to some user change.  Therefore, we come to our second (and final) criteria for assessing mesh algorithms:

Criteria 2: The latency of meshing cannot be too high.

Speed vs. Quality

Intuitively, it seems like there should be some tradeoff here.  One could imagine a super-mesher that does a big brute force search for the best mesh compared to a faster, but dumber method that generates a suboptimal mesh.  Supposing that we were given two such algorithms, it is not a-priori clear which method we should prefer.  Over the long term, maybe we would end up getting a better FPS with the sophisticated method, while in the short term the dumb algorithm might be more responsive (but we’d pay for it down the road).

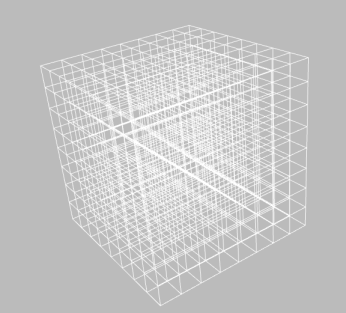
Fortunately, there is a simple solution to this impasse.  If we have a fast, but dumb, mesher; we can just run that on any meshes that have changed recently, and mark the geometry that changed for remeshing later on.  Then, when we have some idle cycles (or alternatively in another thread) we can run the better mesher on them at some later point.  Once the high quality mesh is built, the low quality mesh can then be replaced.  In this way, one could conceivably achieve the best of both worlds.

Some Meshing Algorithms

The conclusion of this digression is that of the two of these criteria, it is item 1 which is ultimately the most important (and unfortunately also the most difficult) to address.  Thus our main focus in the analysis of meshing algorithms will be on how many quads they spit out.  So with the preliminaries settles, let’s investigate some possible approaches:

The Stupid Method:

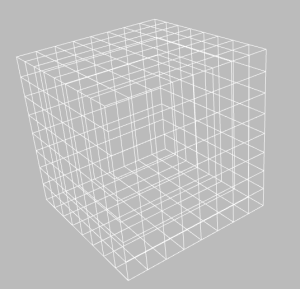
The absolute dumbest way to generate a Minecraft mesh is to just iterate over each voxel and generate one 6-sided cube per solid voxel.  For example, here is the mesh you would get using this approach on a 8x8x8 solid region:

[](https://0fps.files.wordpress.com/2012/06/stupid8x8x8.png)The time complexity of this method is linear in the number of voxels.  Similarly, the number of quads used is 6 \* number of filled voxels.  For example, in this case the number of quads in the mesh is 8\*8\*8\*6 = 3072.  Clearly this is pretty terrible, and not something you would ever want to do.  In a moment, we’ll see several easy ways to improve on this, though it is perhaps worth noting that for suitably small geometries it can actually be workable.

The only thing good about this approach is that it is almost impossible to screw up.  For example, I was able to code it up correctly in minutes and it worked the first time (compared to the culling method which took me a few iterations to get the edge cases right.)  Even if you don’t plan on using this approach it can still be handy to have it around as a reference implementation for debugging other algorithms.

Culling:

Clearly the above method is pretty bad.  One obvious improvement is to just cull out the interior faces, leaving only quads on the surface.  For example, here is the mesh you get from culling the same 8x8x8 solid cube:

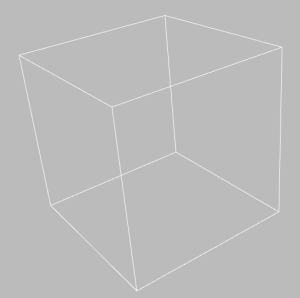
[](https://0fps.files.wordpress.com/2012/06/culled8x8x8.png)

Practically speaking, this method is pretty much the same as the previous, except we have to do a bit of extra book keeping when we traverse the volume.  The reason for this is that now we not only have to read each voxel, but we we also have to scan through their neighbors.  This requires a little more thought when coding, but time complexity-wise the cost of generating this mesh is asymptotically the same as before.  The real improvement comes in the reduction of quads.  Unlike the stupid method, which generates a number of quads proportional to the boundary, culling produces quads proportional to the surface area.  In this case, the total number of faces generated is 8\*8\*6 = 384 — a factor of 8x better!

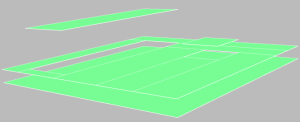
It is not hard to see that the culled mesh is strictly smaller than the stupid mesh (and often by quite a lot!).  We can try to estimate how much smaller using a dimensional analysis argument:  assuming that our geometry is more-or-less spherical, let n be its radius.  Then, in the stupid method we would expect to generate about O(n^3) quads, while in the culled  version we’d get only O(n^2).  This gives an improvement of a factor of about O(n), which is pretty good!  So, if our chunks were say 16x16x16, then we might expect the culled volume to have about 16x fewer faces than the naive method (heuristically).  Of course, spherical geometry is in some sense the best possible case.  If your world is really jagged and has lots of holes, you wouldn’t expect much improvement at all.  In the worst case (namely a checkerboard world) the size of the meshes produced would be identical!  In practice, one would usually expect something somewhere in between.

Greedy Meshing:

The previous two approaches are probably the most frequently cited methods for generating Minecraft-like meshes, but they are quite far from optimal.  The last method that I will talk about today is a greedy algorithm which merges adjacent quads together into larger regions to reduce the total size of the geometry.  For example, we could try to mesh the previous cube by fusing all the faces along each side together:

[](https://0fps.files.wordpress.com/2012/06/lexicographic8x8x8.png)

This gives a mesh with only 6 quads (which is actually optimal in this case!)  Clearly this improvement by a factor of 64 is quite significant.  To explain how this works, we need to use two ideas.  First, observe that that it suffices to consider only the problem of generating a quad mesh for some 2D cross section.  That is we can just sweep the volume across once in each direction and mesh each cross section separately:

[](https://0fps.files.wordpress.com/2012/06/slices.png)

This reduces the 3D problem to 2D.  The next step (and the hard one!) is to figure out how to mesh each of these 2D slices.  Doing this optimally (that is with the fewest quads) is quite hard.  So instead, let’s reformulate the problem as a different type of optimization.  The idea is that we are going to define some type of total order on the set of all possible quadrangulations, and then pick the minimal element of this set as our mesh.  To do this, we will first define an order on each quad, which we will then extend to an order on the set of all meshes.  One simple way to order two quads is to sort them top-to-bottom, left-to-right and then compare by their length.

More formally, let’s denote a quad by a tuple (x,y,w,h), where (x,y) is the upper left corner and (w,h) is the width / height of the quad.  Given such a representation of a quad, we denote its underlying set by:

By a 2D quad mesh, here we mean a partition of some set S into a collection of quads Q_i.  Now let’s define a [total order](http://en.wikipedia.org/wiki/Total_order) on these quads, given (x_0, y_0, w_0, h_0), (x_1, y_1, w_1, h_1) define the relation:

Or if you prefer to read psuedocode, we could write it like this:

compareQuads( (x0, y0, w0, h0), (x1, y1, w1, h1) ):

if ( y0 != y1 ) return y0 < y1

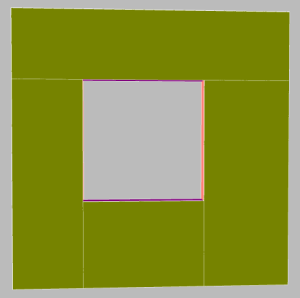
if ( x0 != x1 ) return x0 < x1

if ( w0 != w1 ) return w0 > w1

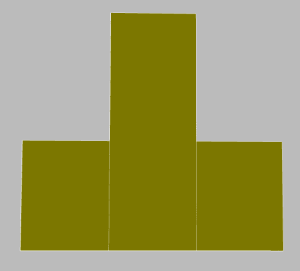
return h0 >= h1

**Exercise:**Check that this is in fact a total order.

The next thing we want to do is extend this ordering on quads to an ordering on meshes, and we do this in a very simple way.  Given two sorted sequences of quads ( q_0, q_1, ... ), ( p_0, p_1, ... ) such that q_i \leq q_{i+1} and and p_i \leq p_{i+1}, we can simply compare them lexicographically.  Again, this new ordering is in fact a total order, and this means that it has a **unique** least element.  In greedy meshing, we are going to take this least element as our mesh, which we’ll call the *greedy mesh*.  Here is an example of what one of these lexicographically minimal meshes looks like:

[](https://0fps.files.wordpress.com/2012/06/hole.png)

You may be skeptical that finding the least element in this new order always produces a better quad mesh.  For example, there is no guarantee that the least element in this new order is also the mesh with the fewest quads.  Consider what you might get when quad meshing a T-shape using this method:

[](https://0fps.files.wordpress.com/2012/06/tshape.png)

**Exercise 2:**  Find a mesh of this shape which uses fewer quads.

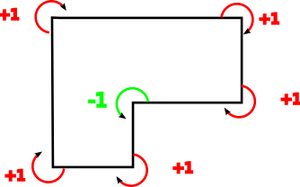
However, we can say a few things.  For example, it always true that each quad in the greedy mesh completely covers at least one edge on the perimeter of the region.  Therefore, we can prove the following:

**Proposition:**The number of quads in the greedy mesh is strictly less than the number of edges in the perimter of the region.

This means that in the worst case, we should expect the size of the greedy mesh to be roughly proportional to the number of edges.  By a heuristic/dimensional analysis, it would be reasonable to estimate that the greedy mesh is typically about O( \sqrt(n) ) times smaller than the culled mesh for spherical geometries (where n is the radius of the sphere as before).  But we can be more precise about how good the greedy mesh actually is:

**Theorem:**The greedy mesh has no more than 8x as many quads than the optimal mesh.

That’s within a constant factor of optimality!  To prove this, we’ll need to introduce a bit of nomenclature first.  We’ll call a vertex on the perimeter of a region convex if it turns to the right when walking around the mesh clockwise, and otherwise we’ll call it concave.  Here is a picture, with the convex vertices colored in red and the concave vertices colored in green:

[](https://0fps.files.wordpress.com/2012/06/convexconcave1.png)

There’s a neat fact about these numbers.  It turns out that if you sum them up, you always get the same thing:

**Proposition:**For any simply connected region with V_+ convex and V_- concave vertices on the perimeter, V_+ - V_- = 4.

This can be proven using the [winding number theorem](http://en.wikipedia.org/wiki/Winding_number) from calculus.  But we’re going to apply this theorem to get a lower bound on the number of quads in an arbitrary mesh:

**Lemma:** Any connected genus G region with a perimeter of E edges requires at least \frac{E + 2G + 4}{8} quads to mesh.

**Proof:**Let V_+, V_- denote the number of convex/concave vertices on the perimeter respectively.  Because each quad can cover at most 4 convex vertices, the number of quads, N, in the mesh is always at least N \geq \frac{V_+}{4}.  By the winding number theorem, it is true that for simply connected regions V_+ = V_- + 4, and so V_+ = \frac{E}{2} + 2.  Therefore any mesh of a simply connected region requires at least \frac{E + 4}{8} quads.  To extend this to non-simply connected regions, we observe that any quad mesh can be cut into a simply connected one by slicing along some edges in the quad mesh connecting out to the boundary.  Making this cut requires introducing at most two edges per each hole in the object, and so we could just as well treat this as a simply connected region having E + 2 G edges, and applying the previous result gives us the bound in the lemma. \Box

To prove our main theorem, we just combine these two lemmas together, using the fact that G \geq 0.  In summary, we have the following progression:

Optimal \leq Greedy \leq Culled \leq Stupid

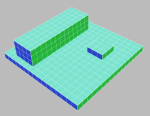
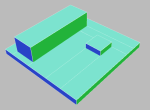
So now that I’ve hopefully convinced you that there are some good theoretical reasons why greedy meshing outperforms more naive methods, I’ll try to say a few words about how you would go about implementing it.  The idea (as in all greedy algorithms) is to grow the mesh by making locally optimal decisions.  To do this, we start from an empty mesh and the find the quad which comes lexicographically first.  Once we’ve located this, we remove it from the region and continue.  When the region is empty, we’re done!  The complexity of this method is again linear in the size of the volume, though in practice it is a bit more expensive since we end up doing multiple passes.  If you are the type who understands code better than prose, you can take a look at this [javascript implementation](https://github.com/mikolalysenko/mikolalysenko.github.com/blob/gh-pages/MinecraftMeshes/js/greedy.js) which goes through all the details.

Demo and Experiments

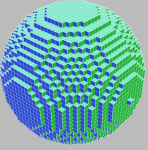
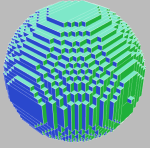
To try out some of these different methods, I put together a quick little three.js demo that you can run in your browser:

[**Click here to try out the Javascript demo!**](http://mikolalysenko.github.com/MinecraftMeshes/index.html)

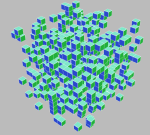
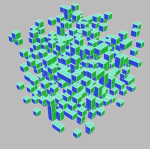
Here are some pictures comparing naive meshing with the greedy alternative:

[](https://0fps.files.wordpress.com/2012/06/bad.png)         [](https://0fps.files.wordpress.com/2012/06/good.png)

Naive (left), 690 quads vs. Greedy (right), 22 quads

[](https://0fps.files.wordpress.com/2012/06/spherebad.png)         [](https://0fps.files.wordpress.com/2012/06/spheregood.png)

4770 quads vs. 2100 quads

.[](https://0fps.files.wordpress.com/2012/06/noisebad.png)       [](https://0fps.files.wordpress.com/2012/06/noisegood.png)

2198 quads vs. 1670 quads

As you can see, greedy meshing strictly outperforms naive meshing despite taking asymptotically the same amount of time.  However, the performance gets worse as the curvature and number of components of the data increase.  This is because the number of edges in each slice increases.  In the extreme case, for a collection of disconnected singleton cubes, they are identical (and optimal).

Conclusion and some conjectures:

Well, that’s it for now.  Of course this is only the simplest version of meshing in a Minecraft game, and in practice you’d want to deal with textures, ambient occlusion and so on (which adds a lot of coding headaches, but not much which is too surprising).  Also I would ask the reader to excuse the fairly rough complexity analysis here.  I’d actually conjecture that the true approximation factor for the greedy algorithm much less than 8, but I’ve spent enough time writing this post so I guess I’ll just leave it as is for now.  I think the real weakness in my analysis lies in the computation of the upper bound on the size of greedy mesh.  I’d actually guess the following is true:

**Conjecture:** The size of the greedy mesh is at most \frac{E}{2}.

If this conjecture holds true, then it would reduce the current bounds on the approximation factor to 4x instead of 8x.  It’s also not clear that this is the best known greedy approximation.  I’d be interested to learn how Minecraft and other voxel engines out there solve this problem.  From my time playing the game, I got the impression that Minecraft was doing some form of mesh optimization beyond naive culling, though it was not clear to me exactly what was going on (maybe they’re already doing some form greedy meshing?)

It is also worth pointing out that the time complexity of each of these algorithms is optimal (ie linear) for a voxel world which is encoded as a bitmap.  However, it would be interesting to investigate using some other type of data structure.  In the previous post, I talked about using run-length encoding as an alternative in-memory storage of voxels.  But greedy meshing seems to suggest that we could do much better:  why not store the voxels themselves as a 3D cuboid mesh?  Doing this could drastically reduce the amount of memory, and it is plausible that such a representation could also be used to speed up meshing substantially.  Of course the additional complexity associated with implementing something like this is likely to be quite high.  I’ll now end this blog post with the following open problem:

**Open Problem:** Do there exist efficient data structures for dynamically maintaining greedy meshes?

If you have other questions or comments, please post a response or send me an email!

Errata

6/30/12: Corrected an error in the definition of the order on each quad.  Was written (x,y,w,h) when it should have been (y,x,w,h) (Thanks Matt!)