**[Simplifying Isosurfaces (Part 1)](https://0fps.net/2012/07/21/simplifying-isosurfaces-part-1/)**

Posted on [July 21, 2012](https://0fps.net/2012/07/21/simplifying-isosurfaces-part-1/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/" \o "View all posts by mikolalysenko)

I just got back to the US after spending 3 months in Berlin, and I’ve now got plenty of ideas kicking around that I need to get on to paper.   Recall that lately I’ve been blogging about [solid modeling](https://0fps.wordpress.com/2012/07/10/smooth-voxel-terrain-part-1/) and in particular, we talked about [meshing isosurfaces](https://0fps.wordpress.com/2012/07/12/smooth-voxel-terrain-part-2/) last time.  However, in the examples from our previous discussion we only looked at some fairly small volumes.  If our ultimate goal is to create something like Minecraft, where we use large volumetric objects to represent terrain, then we have to contend with the following difficult question:

**How do we efficiently render large, dynamic, volumetric terrains?**

One source of inspiration is to try generalizing from the 2D case.  From the mid-90’s to the early 2000’s this was a hot topic in computer graphics, and many techniques were discovered, like the [ROAM algorithm](http://www.cognigraph.com/ROAM_homepage/) or [Geometry Clipmaps](http://research.microsoft.com/en-us/um/people/hoppe/proj/geomclipmap/).  One thing that seems fundamental to all of these approaches is that they make use of the concept of [level-of-detail](http://en.wikipedia.org/wiki/Level_of_detail), which is the idea that the complexity of geometry should scale inversely with the viewing distance.  There at least a couple of good reasons why you might want to do this:

* Simplifying the geometry of distant objects can potentially speed up rendering by reducing the total number of vertices and faces which need to be processed.
* Also, rendering distant objects at a high resolution is likely to result in [aliasing](http://en.wikipedia.org/wiki/Aliasing), which can be removed by filtering the objects before hand.  This is the same principle behind [mip-mapping](http://en.wikipedia.org/wiki/Mipmap), which is used to reduce artefacts in texture mapping distant objects.

Level of detail is a great concept in both theory and practice, and today it is applied almost everywhere.  However, level of detail is more like a general design principle than an actual concrete algorithm.  Before you can use it in a real program, you have to say what it means to ***simplify*** geometry in the first place.  This motivates the following more specific question:

How do we simplify isosurfaces?

Unfortunately, there probably isn’t a single universal answer to the question of what it means to `simplify` a shape, but clearly some definitions are more useful than others.  For example, you would hardly take me seriously if I just defined some geometry to be `simpler’ if it happened to be the output from some arbitrarily defined `simplification algorithm’ I cooked up.  A more principled way to go about it is to define simplification as a type of optimization problem:

Given some input shape X, a class of simpler shapes S and a metric d on the space of all shapes, find some approximate shape Y \in S such that d(X, Y) is minimized:

\mathop{\mathrm{argmin}} \limits_{Y \in S} d(X, Y)

The intuition behind this is that we want to find a shape Y which is the **best** approximation of X that we can hope to find subject to some information constraint.  This constraint is embodied in that class S, which we could take to be the set of all shapes having a strictly shorter description than X (for example, we could require that Y has fewer vertices, edges, faces, voxels, etc.).  This is of course only a meta-definition, and to make it concrete we need to plug in a metric, d. The choice of this metric determines what shapes you consider to be good approximations.

While there are as many different approaches to this problem as there are metrics, for the purposes of discussion we shall classify them into two broad categories:

1. **Surface**:  Metrics that operate on meshed isosurfaces.
2. **Volumetric**: Metrics which operate directly on the sampled potential function.

I’ll eventually write a bit about both of these formulations, but today I’ll be focusing on the first one:

Surface Simplification

Of these two general approaches, by far the most effort has been spent on mesh simplification.  There are a many high quality resources out there already, so I won’t waste much space talking about the basics here, and instead refer you to the following reference:

D. Luebke. (2001)”[A Developer’s Survey of Mesh Simplification Algorithms](http://www.cs.virginia.edu/~luebke/publications/pdf/cg+a.2001.pdf)” IEEE Computer Graphics and Applications.

Probably the most important class of metrics for surface simplification are the so-called quadratic error metrics.  These approaches were described by Michael Garland and Paul Heckbert back in 1997:

M. Garland, P. Heckbert.  (1997) “[Surface Simplification Using Quadratic Error Metrics](http://www1.cs.columbia.edu/~cs4162/html05s/garland97.pdf)” SIGGRAPH 1997

Intuitively, quadratic error metrics measure the difference in the discrete curvature of two meshes locally.  For piecewise linear meshes, this makes a lot of sense, because in flat regions you don’t need as many facets to represent the same shape.  Quadratic error metrics work exceptionally well — both in theory and in practice — and are by far the dominant approach to mesh simplification.  In fact, they have been so successful that they’ve basically eliminated all other competition, and now pretty much every approach to mesh simplification is formulated within this framework.  Much of the work on mesh simplification today, instead focuses on the problem of implementing and optimizing.  These technical difficulties are by no means trivial, and so what I will discuss here are some neat tricks that can help with simplifying isosurface meshes.

One particularly interesting paper that I’d like to point out is the following:

D. Attali, D. Cohen-Steiner, H. Edelsbrunner. (2005) “[Extraction and Simplification of Iso-surfaces in Tandem](ftp://ftp-sop.inria.fr/geometrica/dcohen/Papers/sgp_he_da.pdf)” Eurographics Symposium on Geometry Processing.

This article was brought to my attention by [Carlos Scheid](http://cscheid.net/) in the comments on my last post.  The general idea is that if you run your favorite mesh simplification algorithm in parallel while you do isosurface extraction, then you can avoid having to do simplification in one shot.  This is a pretty neat general algorithm design principle: often it is much faster to execute an algorithm incrementally than it is to do it in one shot.  In this case, the speed up ends up being on the order of O(n^{2/3}).  This is probably the fastest method I’ve seen for simplifying an isosurface, but the speed comes at the cost of greater implementation complexity.  But, if you are planning on doing mesh simplification, and you feel confident in your coding skills, then there is no practical reason not to use this technique.

I also learned of another neat reference from [Miguel Cepero’s blog](http://procworld.blogspot.com/2011/10/playing-dice-with-mesh-simplification.html):

J. Wu, L. Kobbelt. (2002) “[Fast Mesh Decimation by Multiple-Choice Techniques](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.7091&rep=rep1&type=pdf)” VMV 2002

It describes an incredibly simple Monte-Carlo algorithm for simplifying meshes.  The basic idea is that instead of using a priority queue to select the edge with least error, you just pick some constant number of edges (say 10) at random, then collapse the edge in this set with least error.  It turns out that for sufficiently large meshes, this process converges to the optimal simplified mesh quite well.  According to Miguel, it is also quite a bit faster than the standard heap/binary search tree implementation of progressive meshes.  Of course Attali et al.’s incremental method should be even faster, but I still think this method deserves some special mention due to its extreme simplicity.

General purpose surface simplification is usually applied to smaller, discrete objects, like individual characters or static set pieces.  Typically, you precalculate several simplified versions of each mesh and cache the results; then at run time, you just select an appropriate level of detail depending on the distance to the camera and draw that.  This has the advantage that only a very small amount of work needs to be done each frame, and that it is quite efficient for smaller objects.  Unfortunately, this technique does not work so well for larger objects like terrain.  If an object spans many levels of detail, ideally you’d want to refine parts of it adaptively depending on the configuration viewer.

This more ambitious approach requires dynamically simplifying the mesh.  The pioneering work in this problem is the following classic paper by Hugues Hoppe:

H. Hoppe.  (1997) “[View-dependent refinement of progressive meshes](http://research.microsoft.com/en-us/um/people/hoppe/proj/vdrpm/)” ACM SIGGRAPH 1997

Theoretically, this type of approach sounds really great, but in practice it faces a large number of technical challenges.  In particular, the nature of progressive meshing as a series of sequential edge collapses/refinements makes it difficult to compute in parallel, especially on a GPU.   As far as I know, the current state of the art is still the following paper:

L. Hu, P. Sander, H. Hoppe. (2010) “[Parallel View-Dependent Level-of-Detail Control](http://research.microsoft.com/en-us/um/people/hoppe/proj/pvdlod/)” IEEE Trans. Vis. Comput. Graphics. 16(5)

The technology described in that document is quite impressive, but it has a several drawbacks that would prevent us from using it in volumetric terrain.

1. The first point is that it requires some fairly advanced GPU features, like geometry shaders and hardware tesselation.  This rules out a WebGL implementation, which is quite unfortunate.
2. The second big problem is that even though it is by far the fastest known method for continuous view-dependent level of detail, it still is not quite good enough for real time game.  If you have a per-frame budget of 16 ms **total**, you can’t afford to spend 22 ms just generating the geometry — and note that those are the times reported in the paper, taking into account amortization over multiple frames :(.  Even if you throw in every optimization in the book, you are still looking at a huge amount of work per frame just to compute the level of detail.
3. Another blocking issue is that this approach is does not work with dynamic geometry, since it requires a lot of precalculated edge collapses.  While it might be theoretically possible to update these things incrementally, in practice it would probably require reworking this algorithm from scratch,.
4. Finally, it also is not clear (at least to me) how one would adapt this type of method to deal with large, streaming, procedural worlds, like those in Minecraft.  Large scale mesh simplification is by nature a global operation, and it is difficult to convert this into something that works well with paging or chunked data.

Next Time

Since this post has now grown to about 1700 words, I think I’ll break it off here.  Next time we’ll look at volumetric isosurface simplification.  This is a nice alternative to surface simplification, since it is a bit easier to implement view dependent level of detail.  However, it has its own set of problems that we’ll soon discuss.  I still haven’t found a solution to this problem which I am completely happy with.  If anyone has some good references or tips on rendering large volumetric data sets, I’d be very interested to hear about it.