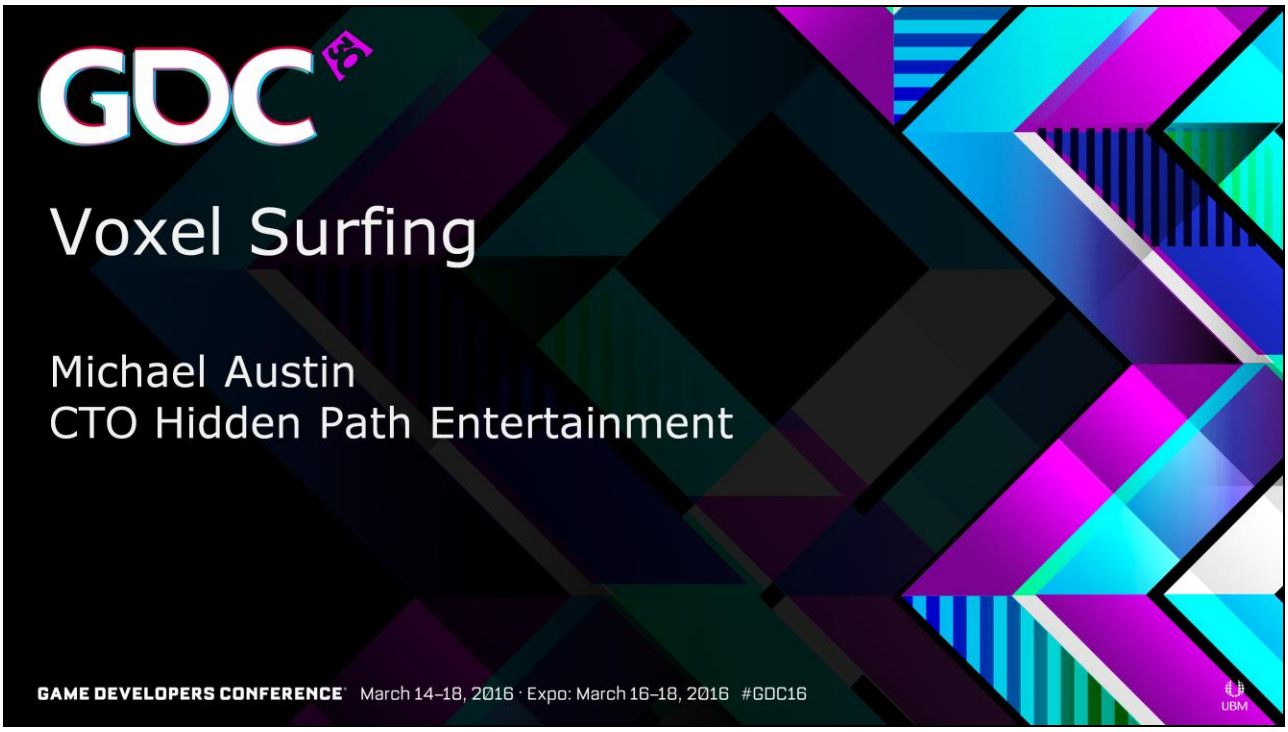

GDC
Voxel Surfing
Michael Austin
CTO Hidden Path Entertainment
GAME DEVELOPERS CONFERENCE March 14–18, 2016 · Expo: March 16–18, 2016 #GDC16
UBM

Overview

- Volume representations
- Voxelization survey
- Lessons from Windborne

Who has worked on a voxel engine before? Who wants to?

My goal is to give the talk I wish I would have had before I started on our procedural engine.

Three parts to this talk. A lot of content, so I'll move fast, but my goal is to give a background on what to look up (that was the biggest problem I ran into when first approaching this subject). You have to KNOW that you should look up contour indicator functions for instance.

Applications

Image credit:
http://www.pcgamer.com/everquest-next-landmark-hands-on/

Image credit:
http://www.minecraftforum.net/news/60286-original-minecraft-surpasses-22-million-sales

Image credit:
http://pixologic.com/zbrush/gallery/

Image credit:
http://www.autodesk.com/products/3ds-max/

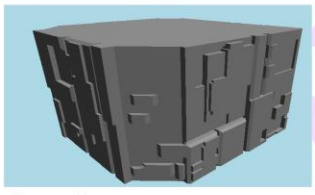


Image credit:
Defense Grid level editor (Hidden Path)

Procedural geometry is useful for far more than just dynamic worlds as people first think- really all 3d art creation requires it.

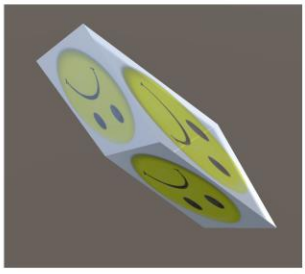
While we normally don't deal with this as engineers, knowing the methods and having a rich toolset can help us make better games.

Part 1
Volume Representations

Quaternions, a love story

Smiley Cube

Smiley Cube, random Quaternion transform



Smiley Cube, random Matrix transform

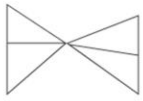
Everyone knows Quaternions are good for working with rotations, and Matrices have problems.

The issue I want to call attention to is that matrices are overspecified. There are lots of things that matrices can represent that aren't normalized orthogonal basis sets, whereas that's all quats can represent.

If I fill a quaternion randomly with data, it's good. If I fill a matrix randomly, it's not good.

Surface Representations

Manifold



Non-Manifold

Non-Manifold

Image credit:
http://www.autodesk.com/techpubs/aliasstudio/2010/index.html?url=WS73099cc142f48755723 0b50811d7d192c64-79b1.htm,topicNumber=d0e88419

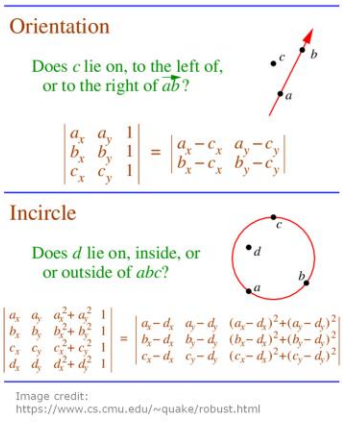


Image credit:
https://www.cs.cmu.edu/~quake/robust.html

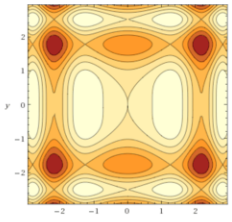
Surface representations have the same problem matrices do. They are subject to euler's identity and manifold issues, exacerbated by floating point error.

It's very hard to make a left right plane test that's 100% robust to floating point error. You WILL hit problems if you roll your own surface rep CSG.

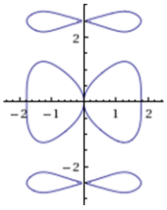
Validation + Repair can be very complicated

Volume Representations

- Any continuous f(x,y,z)
- Implicitly define volume as f(x,y,z) > 0
- Surface is level set where f(x,y,z) = 0



F(x,y) = Sin(x^2)+cos(y^2)-1



Level set (c=0)

Image credit: wolfram alpha

Function can be ANYTHING, no robustness problems- it's strictly a sampling problem. No verify and fix step. Programming without worrying about edge cases, just provide a function.

Each point in space has a very concrete 'in' or 'out' and it's based on a continuous function, so it's manifold.

Volume Representations

- Just need a continuous function
  - Any algebraic function
  - Signed distance field
    - CSG trees (http://www.merl.com/publications/docs/TR2006-054.pdf)
    - Trilinear interpolation over a grid
  - Density function
    - Trilinear interpolation over a grid

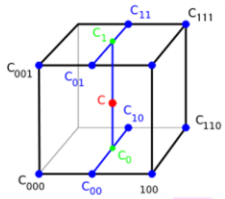


Image credit:
https://en.wikipedia.org/wiki/Trilinear_interpolation

Or more complicated interpolation works

Density vs Signed Distance

Density

Image credit: http://forum.unity3d.com/threads/unity-signed-distance-field-textures.61243/

Distance Field

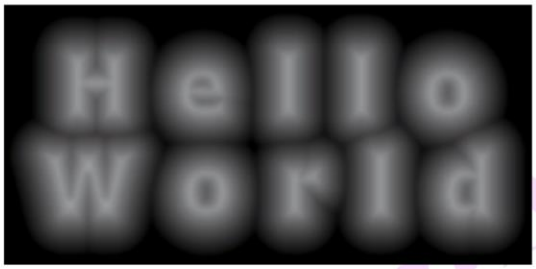


Image credit: http://forum.unity3d.com/threads/unity-signed-distance-field-textures.61243/

Density is much easier to work with (changes are local), but you lose out on some useful operations on the distance field (ability to expand/shrink volume and higher quality gradient calculations)

You can treat density as a distance field with a clamped distance of approximately one sampling grid cell.

Note: the level set / surface for both these methods is almost identical (mid grey in the bitmaps)

Rendering

Volume Rendering

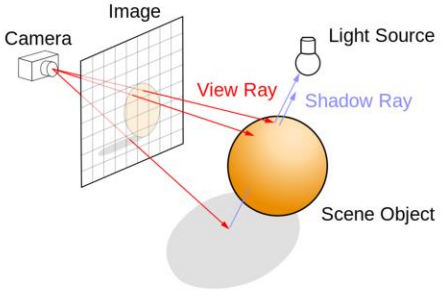


Image credit:
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Convert to Surface

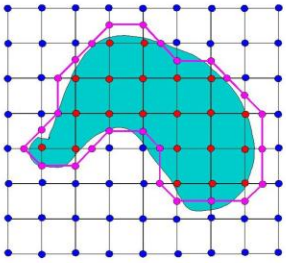


Image credit:
http://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html

Ray tracing – evaluate function along ray, looking for 0 crossings

Convert to surface is what I'm going to talk more about because it fits better in the full volume of video game rendering knowledge we have now

Voxelization

1. Sample f on a grid
2. Approximate the surface in each cell
3. Make sure surfaces align at cell boundaries
4. Profit!

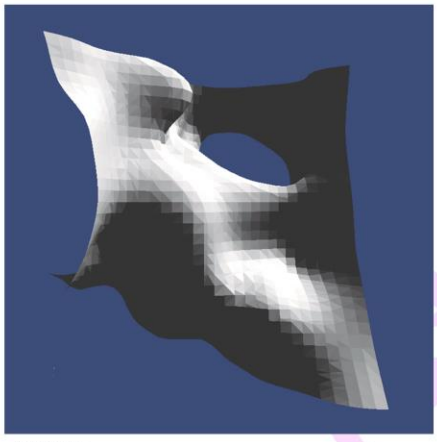


Image credit: https://scrawkblog.com/2013/04/01/marching-cubes-plugin-for-unity/

We used density fields because they are more dynamically updatable

Part 2
Voxelization Survey

Voxelization Ideals

- Easy to Implement
- Local Independence
- Smooth
- Adaptive / suited to LOD
- Minimize triangle slivers
- Preserve sharp features
- Preserve thin features

Image credit:
http://aptww.org/IntlCatalog.nsf/vPlaylistItemByID/70345-3/$FILE/tree_large.jpg

Low intercell dependencies if we want to be dynamic (remember, density fields over distance fields)

Reduced triangle count – ideally large flat areas aren't an array of polygons

Preserve sharp features (trilinear filter is a blur). We can't represent anything over the nyquist frequency of two grid cells.

Simple Cubes

- Sample f(x,y,z) in the middle of each grid cell
- Draw a face between grid cells with different signs

-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-0.8	-0.6	-0.2	0.5
-1	-0.6	-0.4	-0.2	0.2	0.5	1	1
0.2	0.4	0.6	0.8	1	1	1	1
1	1	1	1	1	1	1	1

The nice thing is we only have polies on the surface, so it's pretty efficient

Simple Cubes

Easy to Implement: ++

Local Independence: +

Smooth: -

Adaptive: -

Minimize slivers: +

Sharp features: -

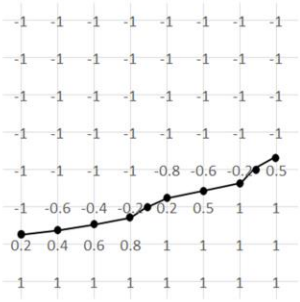
Thin features : -

Image credit:
http://atg.lychnobi.com/screenshots/ATG%20screen%2003.jpg

Marching Cubes

http://www.eecs.berkeley.edu/~jrs/meshpapers/LorensenCline.pdf

- Sample f(x,y,z) in the corner of each grid cell
- Use signs of corners for topology of triangles
- Locate verts at interpolated 0 point along edges



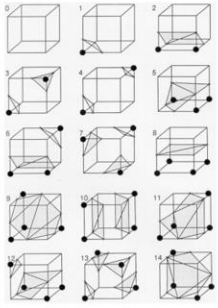


Image credit:
http://www.eecs.berkeley.edu/~jrs/meshpapers/LorensenCline.pdf

If you only look at the sign of the corners of each grid cell, there are only 15 topologies to choose from. You can slide the points along the edges based on relative weights if you have more than -1/1

Marching Cubes

Easy to Implement: +

Local Independence: +

Smooth: +

Adaptive: -

Minimize slivers: -

Sharp features: -

Thin features : -

Image credit: https://c1.iggcdn.com/indiegogo-media-prod-cld/image/upload/c_fill,f_auto,h_413,w_620/v1406808220/vv4nvjafctgzemwbevua.jpg

Transvoxel Algorithm

http://transvoxel.org/

- Sample f(x,y,z) in the corner of each grid cell
- Allow subdividing grid cell sides once (to seam adjacent LOD levels)
- Use signs of sample points for topology of triangles
- Locate verts at interpolated 0 point along edges

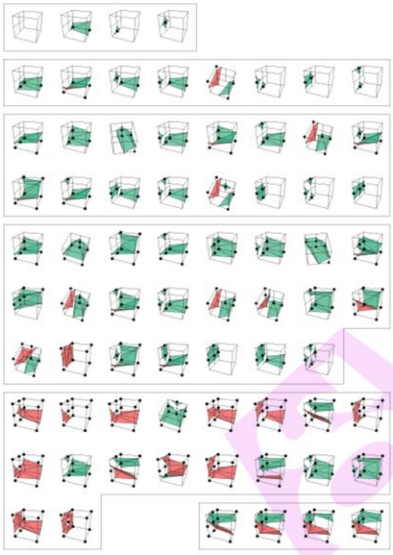


Image credit: http://transvoxel.org/

Transvoxel is a method to allow marching cubes to span different LOD levels. 71 total topologies to handle a tessellation of any combination of sides

Transvoxel Algorithm

Easy to Implement: +

Local Independence: +

Smooth: +

Adaptive: +

Minimize slivers: -

Sharp features: -

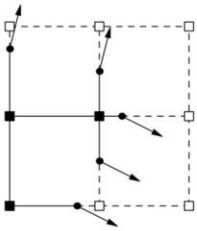
Thin features : -

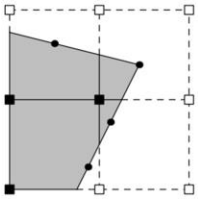
Image credit: http://the31stgame.com/

Dual Contouring

http://www.frankpetterson.com/publications/dualcontour/dualcontour.pdf

- Sample f(x,y,z) in the corner of each grid cell
- Sample f'(x,y,z) at the location of each edge intersection
- Find an 'ideal' point within each cell, on the contour
- Connect dual points for neighboring grid cells (supports multiple LOD resolutions)





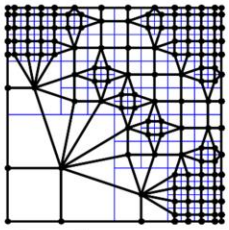


Image credit:
http://www.cs.rice.edu/~jwarren/papers/dmc.pdf

Dual contouring is a method to both spawn LOD levels and preserve sharp features.

It's a dual method, so instead of create vertices on the edges of the grid, we create vertices inside the cells and connect them

Dual Contouring

Easy to Implement: -

Local Independence: -

Smooth: +

Adaptive: +

Minimize slivers: +

Sharp features: +

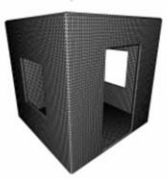
Thin features : ~

Image credit: http://www.voxelfarm.com/

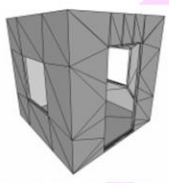
Dual Marching Cubes

https://www.cs.rice.edu/~jwarren/papers/dmc.pdf

- Sample f(x,y,z) over a fine grid
- Find the point that minimizes error (QEF)
- Subdivide octree at that point if error > ε
- Repeat 1-3 until error < ε everywhere
- Construct topological dual of this octree
- Tessellate (as Dual Contouring)



Marching Cubes



Dual Marching Cubes

This extension realizes that marching cubes can be ran on the dual of an octree itself. Fit the grid to the data.

Minimizing slivers is a proposed extension.

Dual Marching Cubes

Easy to Implement: -

Local Independence: -

Smooth: +

Adaptive: +

Minimize slivers: +

Sharp features: +

Thin features : +

Image credit: http://www.duangle.com/nowhere
http://www.ogre3d.org/tikiwiki/tiki-index.php?page=VolumeComponent

Cubical Marching Squares

https://www.csie.ntu.edu.tw/~cyy/publications/papers/Ho2005CMS.pdf

- Build an octree with error subdivision (similar to DMC)
- For any voxel (at any octree level):
  - Unfold the voxel, look at each side individually
  - Create a curve using marching cubes, tessellating if error > ε
  - Fold sides back together and triangulate

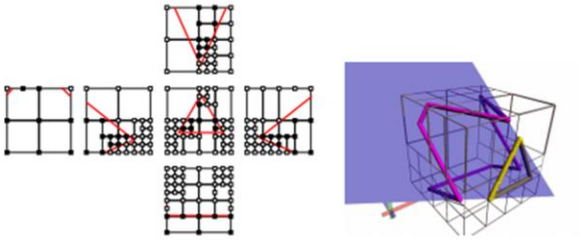


Image credit:
https://www.csie.ntu.edu.tw/~cyy/publications/papers/Ho2005CMS.pdf

Cubical marching squares treats each side as an independent problem, subdivides to match adjacent octree nodes or fit geometry better.

Triangulation can use any method once you reassemble the box and extract the curves

Cubical Marching Squares

Easy to Implement: -

Local Independence: +

Smooth: +

Adaptive: +

Minimize slivers: +

Sharp features: +

Thin features : +

Image credit: https://upvoid.com/
https://community.upvoid.com/uploads/default/76/f2205d419db9bbc8.jpg

Windborne - Overview

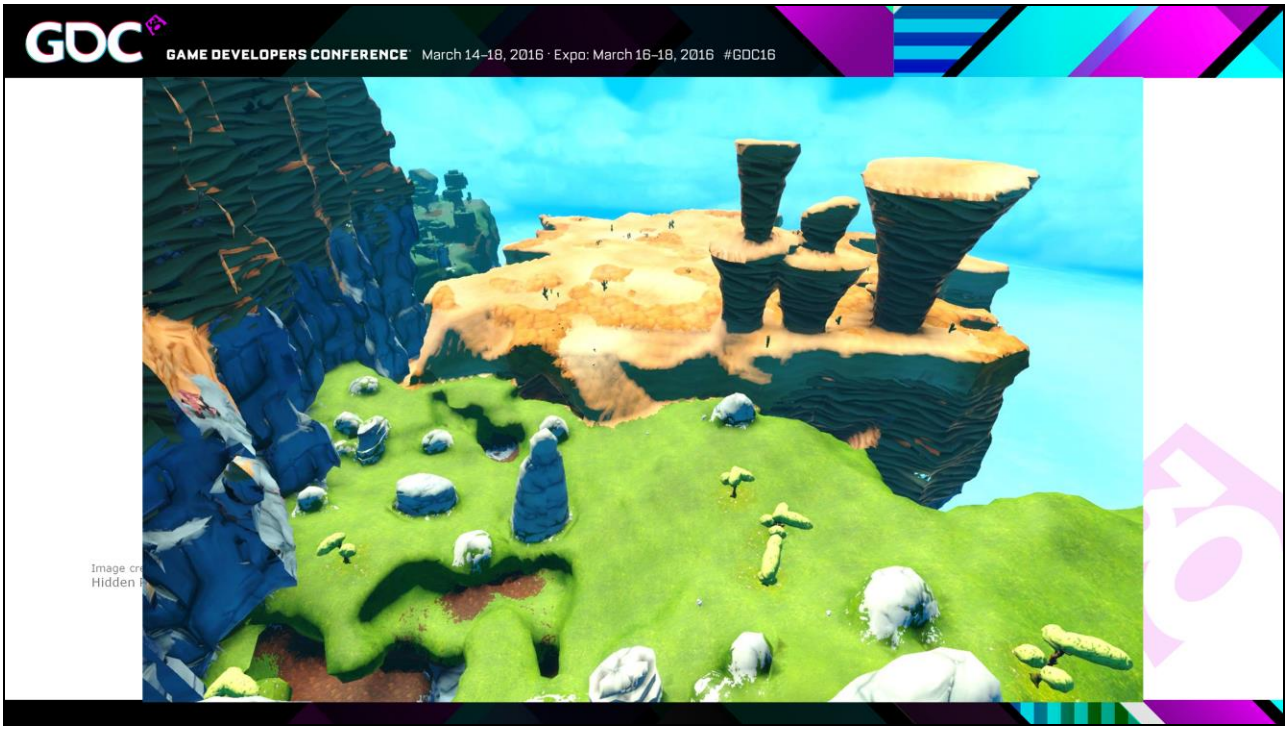
F(x,y,z) = composited, axis-aligned density grid

- Each chunk has the list of features overlapping it
- On calculation, accumulate to chunks with boolean ops

Windborne was a voxel game we worked on, mostly in 2013-2014.

Chunks are just regions of space (16x16x16 for instance) that help manage vertex buffers, visibility, and streaming.

Basic operation is “Here’s a chunk, give me back the density grid (and materials)”



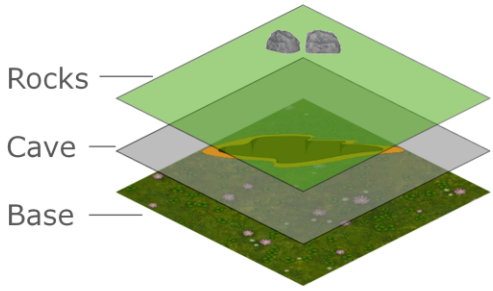
Voxel game, regular grid because we found it was much easier for users to understand and interact with. It's interesting as well that you need a lot more detail and variation when you go smooth terrain; we initially started with something like MC source and it was very boring once the high frequency of individual voxels was removed.

Lessons - Compositing

- Each feature is a layer
- Each layer either subtracts or overlays
- Alpha blend densities

$$\alpha_a + \alpha_b (1 - \alpha_a)$$

Image credit:
https://en.wikipedia.org/wiki/Alpha_compositing



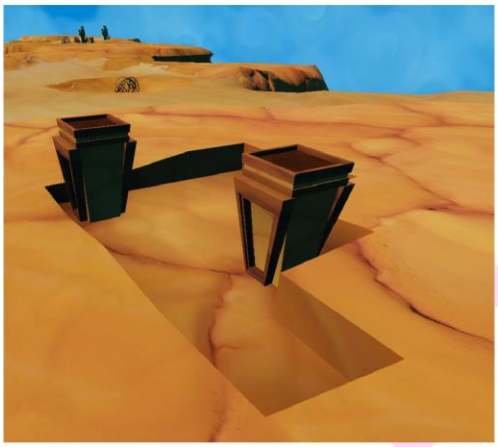
This was what worked best. Min/max/etc didn't for various reasons (dealing with surfaces that were almost aligned)

Lessons - Compositing

- Caves are a 'subtract' layer

Lessons – Uniform representation

- Player wants objects + terrain to feel the same
- Collocate density grid and hand crafted voxel geometry
  - Needs independence
  - Draw sides of the voxel



Storing solid sides with our MC solution was pretty simple, and it meant we could draw them when there was a gap (to avoid seams), while ignoring them if the densities of neighboring voxels lined up

Lessons – Uniform representation

- Represent density as 24 bit mask (3 bits for each corner), fits in with non-terrain voxels
- When well structured, VERY fast query operations

```
Block::Terrain_IsSideFullySolid( BlockFace::Face face ) const
{
    return (( Terrain.mVertexDensity & sBHsolidMaskForFace[ face ] ) == sBHsolidMaskForFace[ face ] );
}

Block::Terrain_IsBlockFullySolid() const
{
    return (( Terrain.mVertexDensity & 044444444 ) == 044444444 );
}
```

Note that for each voxel, we store the densities of all the corners. This is redundant, but important from a user perspective – independence was VERY important – don't edit one thing and then have something neighboring change.

Lessons – Uniform representation

Raise terrain under features to line up density field / non-density-field features

Lessons – Contour Indicator

http://josiahmanson.com/research/contour_indicator/

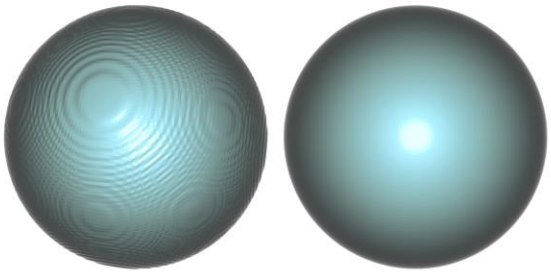
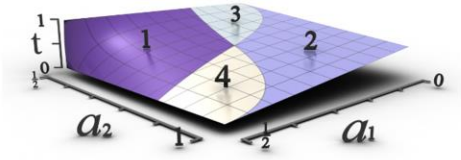
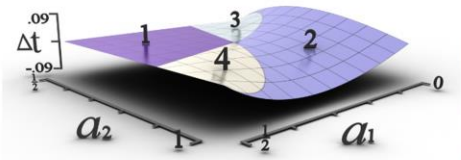


Image credit:
http://josiahmanson.com/research/contour_indicator/





MC underrepresents volumes – for instance if a single corner has a density of '1' and the other 7 have a density of 0, you'd expect about an eighth of the voxel to be filled in. Instead of that though, the traditional algorithm basically slices off a corner half way down three of the edges, whose volume is $1/6^{th}$ of what it should be.

With some simple math we can correct for this, and it mitigates the ridges that you will otherwise see.

Lessons – Density Grid

- Dynamic Lighting
  - Light flow
  - AO

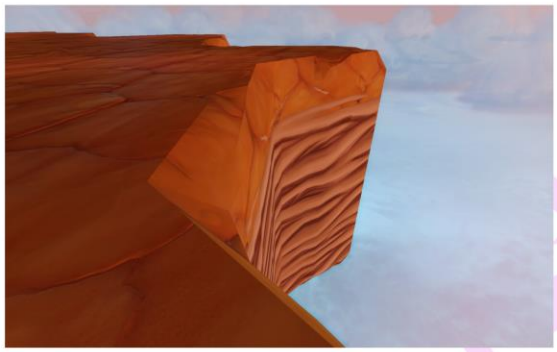
Grids are very good for dynamic lighting- many modern lighting techniques simulate light flow over a grid, and we additionally have densities along each edge and materials. For our placed objects, we actually made a uniform model of density and material to unify lighting across everything.

Lessons – Density Grid

- Occlusion mapping
  (Heirarchial)

Because we had a well structured grid, we built an octree for occlusion queries- we could keep track of whether everything in the octree node was solid or transparent, and make some simplifying assumptions for rendering as a result.

Lessons – Expose Parameters



Exposing the parameters of the voxelization algorithm allowed for some great effects to minimize some of the poor features of the algorithm (Marching cubes lack of high frequency features for instance). We let the normal smoothing parameter vary across materials and it really helps with varying the character of what you see.

Lessons – Leverage Idiosyncrasies

Remember the ‘2 voxel’ nyquist frequency- well we had redundancy which gave us the ability to represent higher frequencies, if we used it. We were able to build terracing into our generation algorithm.

Questions?

Email: michael@hiddenpath.com