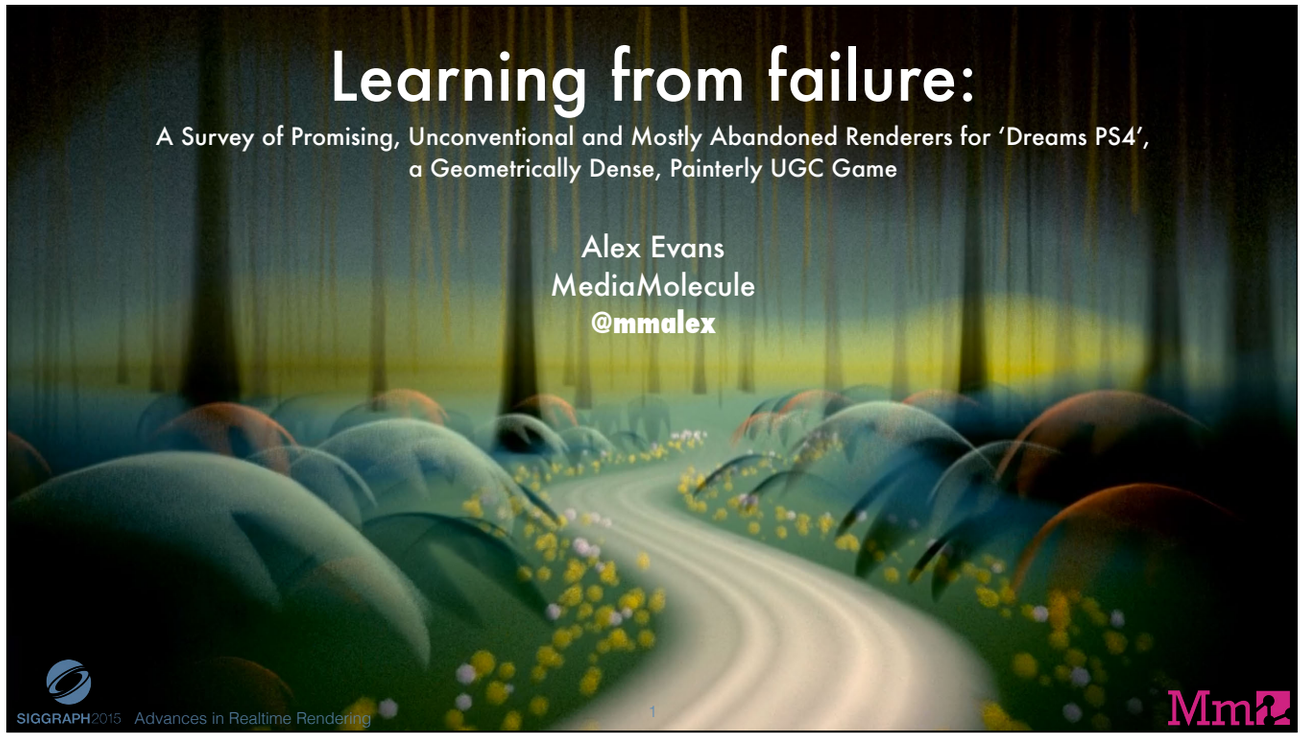


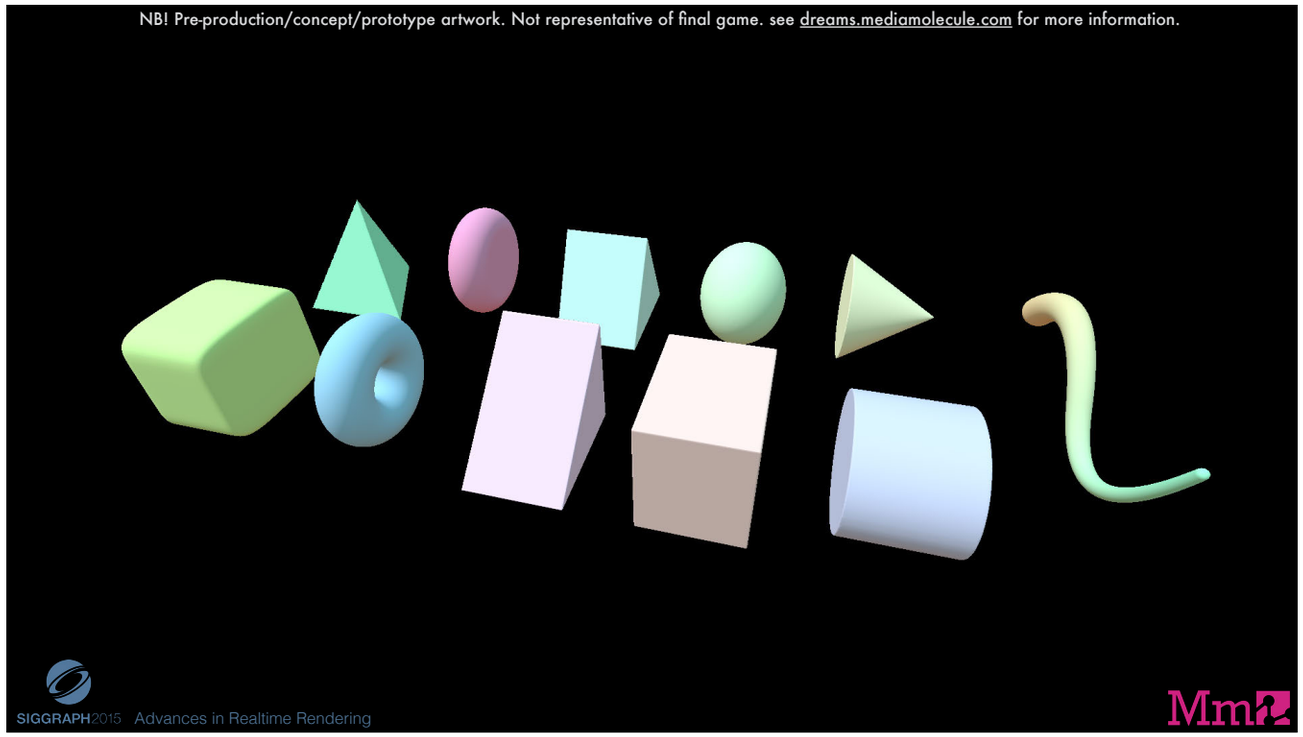
(suggested listening: I was listening to https://soundcloud.com/futureastronauts/transmission011-sora-guest-mix on a loop while I wrote this, so if you want to get into the mood…)
story time!
==========
this talk is about showing you some of the approaches that we tried and failed to make stick for our project. if you're looking for something to take away, hopefully it could be inspiration or some points are places to start, where we left off. I also just think it's interesting to hear about failures, and the lessons learnt along the way. it's a classic story of the random walk of R&D…

spoiler section!
==========
this is where we're headed if you didnt see it at e3 {e3 trailer}
https://www.youtube.com/watch?v=4j8Wp-sx5K0

Engine 1: Anton's Blob Editor

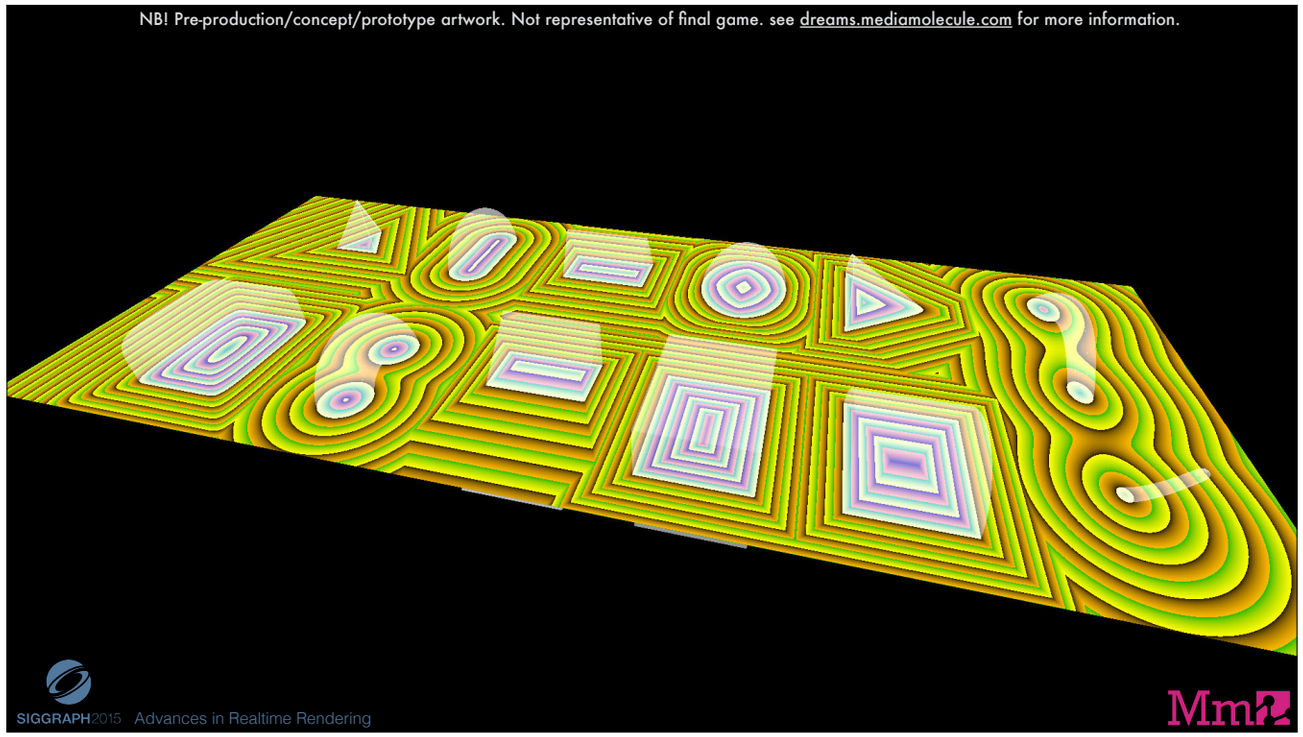
back to the beginning
==========
it all began with @antonalog doing an experiment with move controllers, and a DX11 based marching cubes implementation.

here he is! this was on PC, using playstation move controllers. the idea was to record a series of add & subtraction using platonic shapes with simple distance field functions

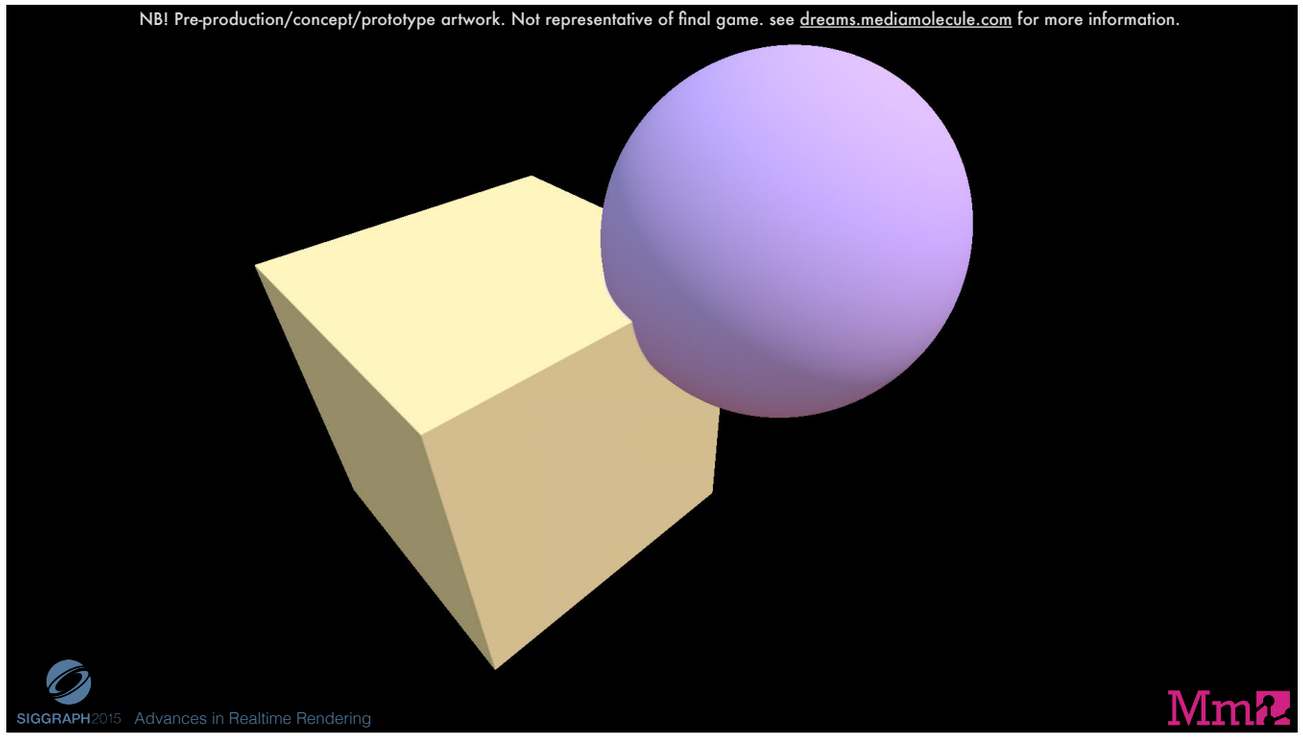


we use (R to L) cubic strokes, cylinders, cones, cuboids, ellipsoids, triangular prisms, donuts, biscuits, markoids*, pyramids.

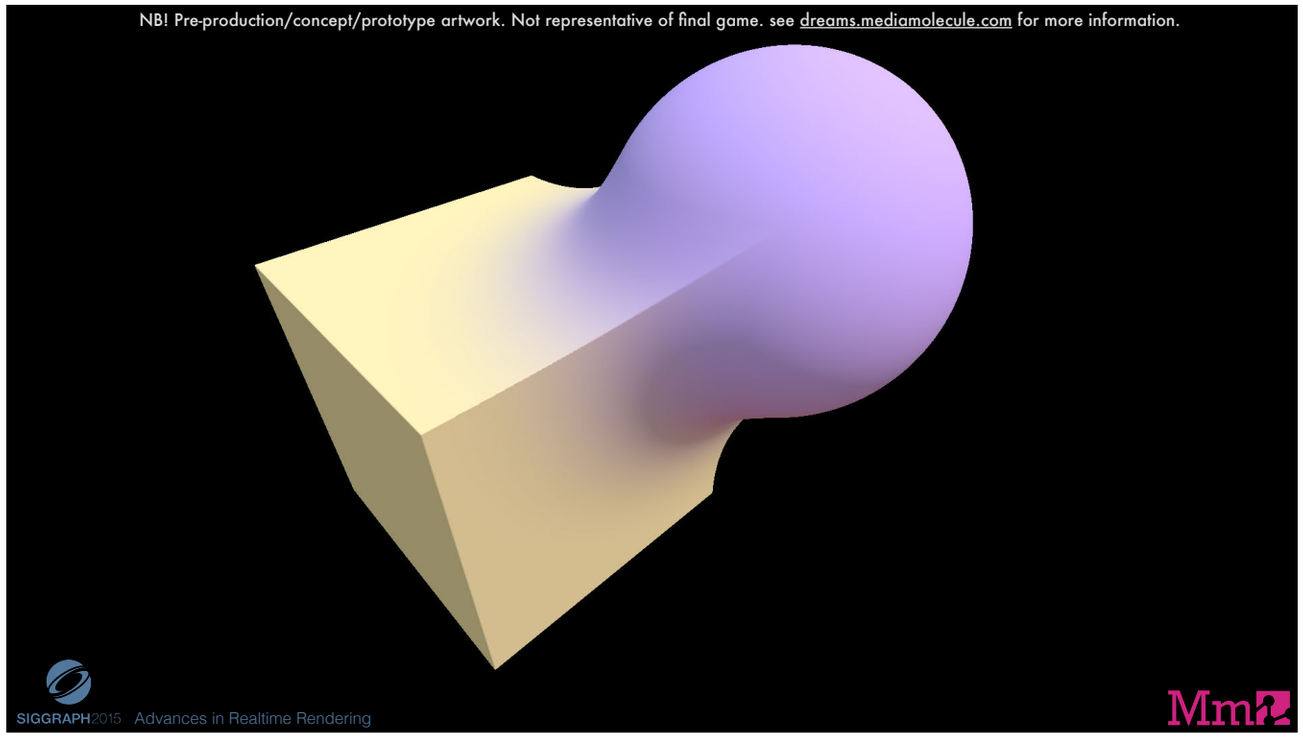
(markoids are named for our own mark z who loves them; they’re super ellipsoids with variable power for x,y,z)



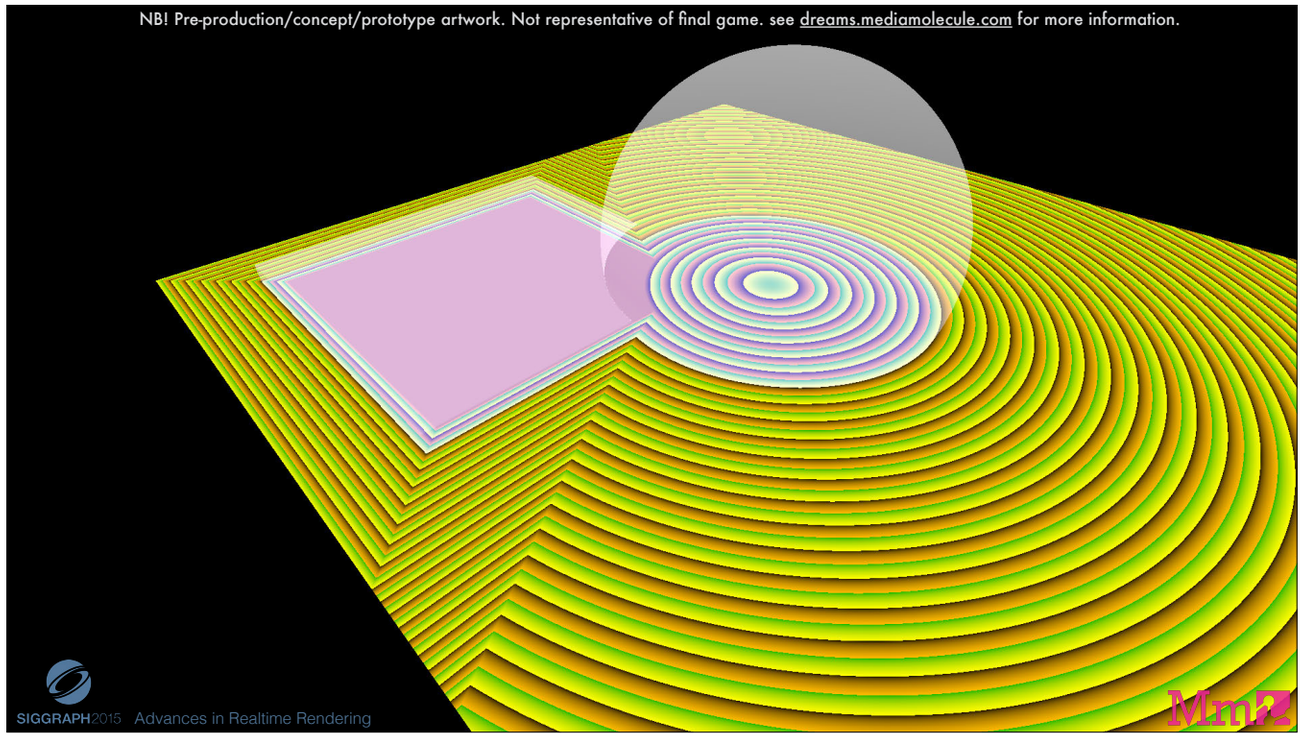
here’s the field for the primitives…



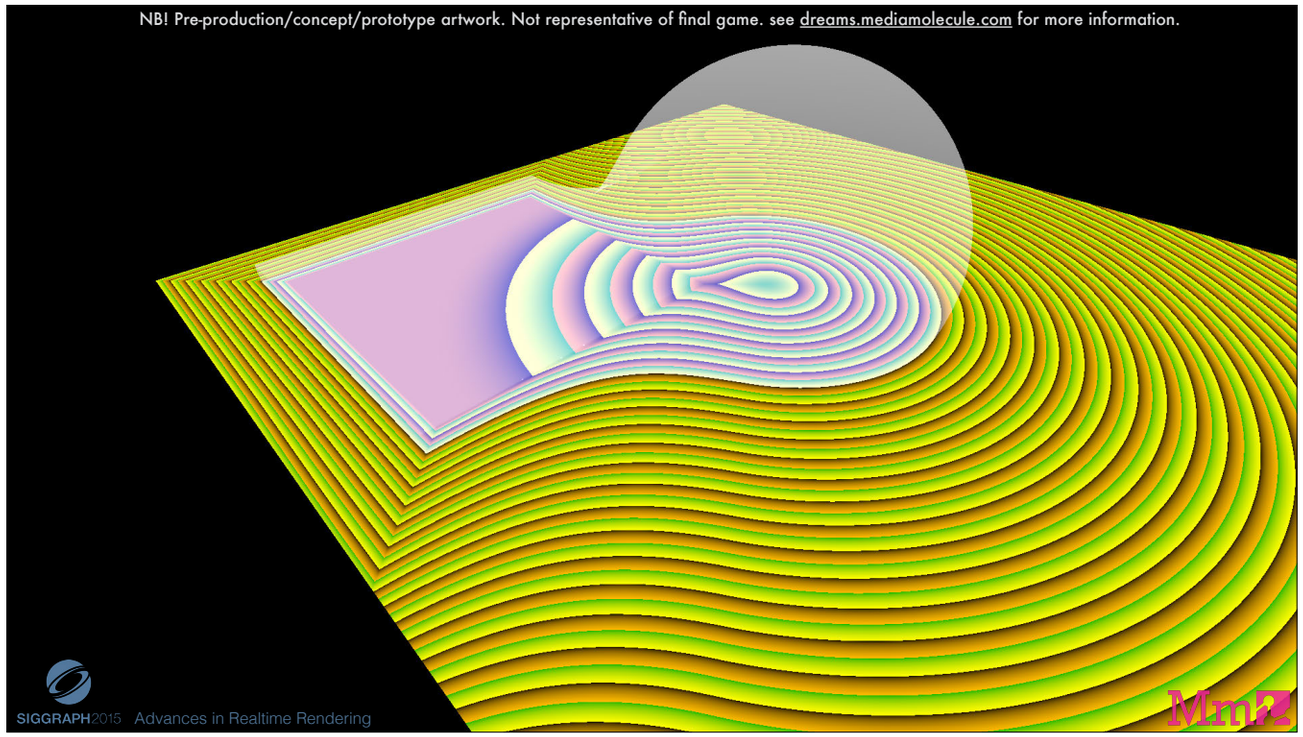
we called each primitive an 'edit',
we support a simple list, not tree of CSG edits.
and models are made up of anything from 1 to 100,000 edits
with add, subtract or ‘color’ only, along with…



soft blend, which is effectively soft-max and soft-min functions.



here’s the field for the hard blend.



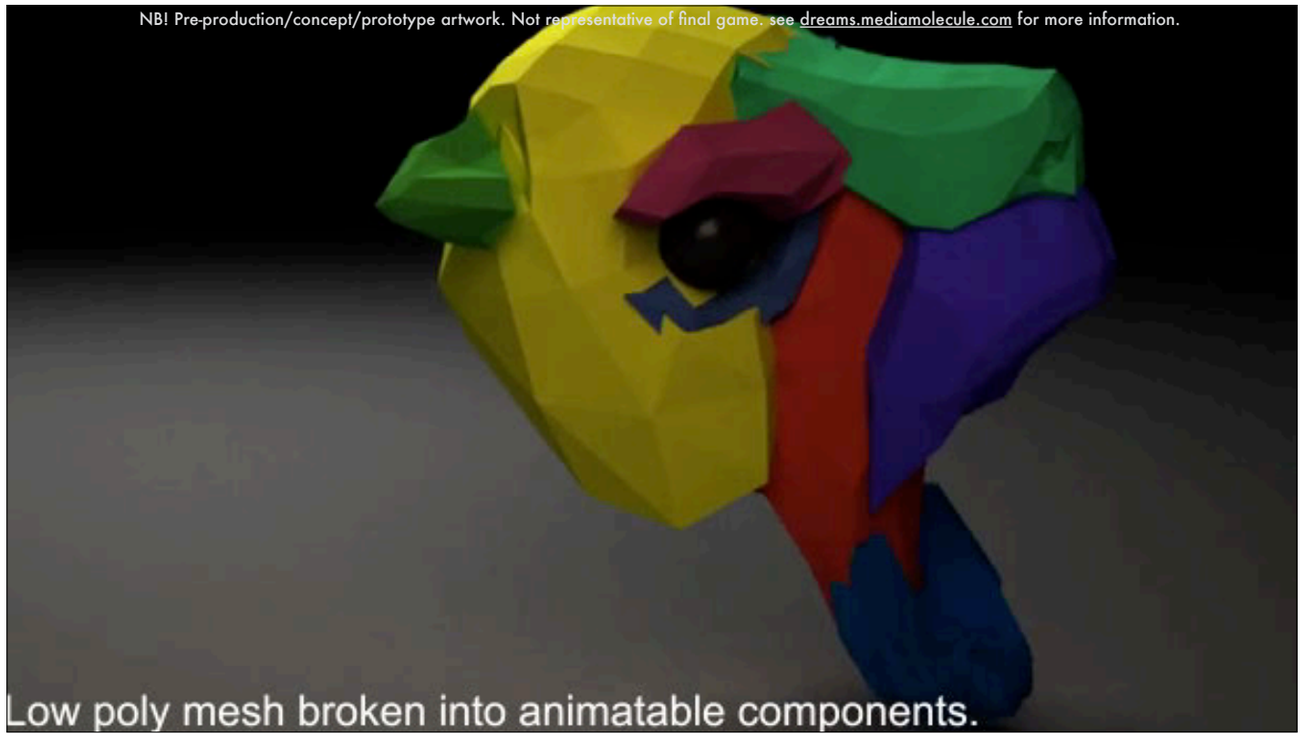
and the soft. I'll talk more about the function for this in a bit. note how nicely defined and distance-like it is, everywhere!

[timelapse of dad’s head, with randomised colours] he’s 8,274 edits.
(an side: MM artists Kareem, Jon B and Francis spent a LONG time developing artistic techniques like the ‘chiselled’ look you see above, with half made early versions of this tech. It’s their artistry which convinced us to carry on down this path. It can’t be understated how important it is when making new kinds of tools, to actually try to use them in order to improve them. Thanks guys!). Anyway:
the compound SDF function , was stored in 83^3 fp16 volume texture blocks, incrementally updated as new edits arrived. each block was independently meshed using marching cubes on the compute shader; at the time this was a pretty advanecd use of CS(as evidenced by frequent compiler bugs/driver crashes) - many of the problems stemmed from issues with generating index buffers dynamically on the GPU. the tech was based on histopyramids, which is a stream compaction technique where you count the number of verts/indices each cell needs, iteratively halve the resolution building cumulative 'summed area' tables, then push the totals back up to full resolution, which gives you a nice way to lookup for each cell where in the target VB/IB its verts should go. there's lots of material online, just google it.

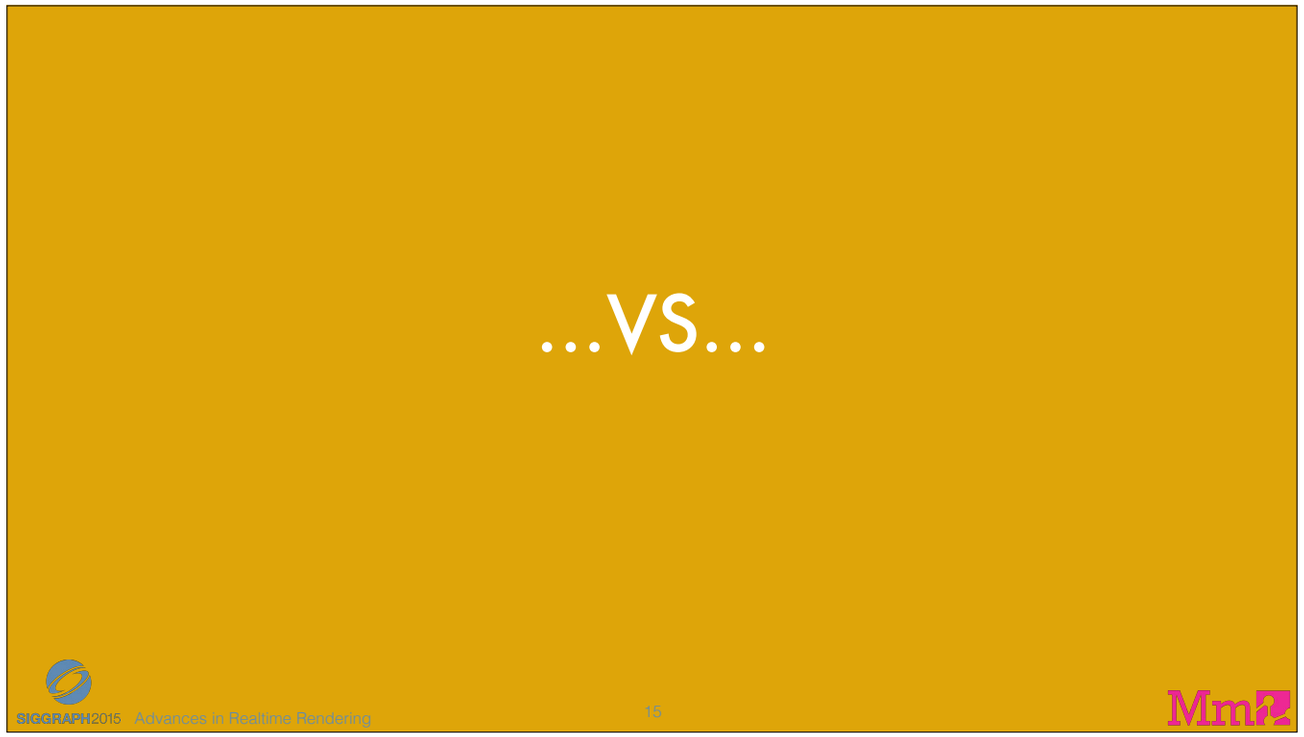
the core idea of lists of simple SDF elements, is still how all sculptures are made in dreams, and is the longest living threads. this was the opposite of a failure! it was our first pillar in the game. Anton worked with Kareem, our art director, to get some pretty cool gestural UI going too; there's minimal UI intrusion so artists can get into flow state. I think he was planning to implement classic z-brush style pull/smear/bend modifications of the field - which is probably what some of you may have thought we did first- but luckily he didn't. Why? wellllll.....

Animation test time!

some early animation tests were done around this time to see what could be achieved - whether with purely with semi- or fully rigid pieces, or some other technique. the results were varied in quality and all over the place in art style - we didn't know what we wanted to do, or what was possible; so we imagined lots of futures:

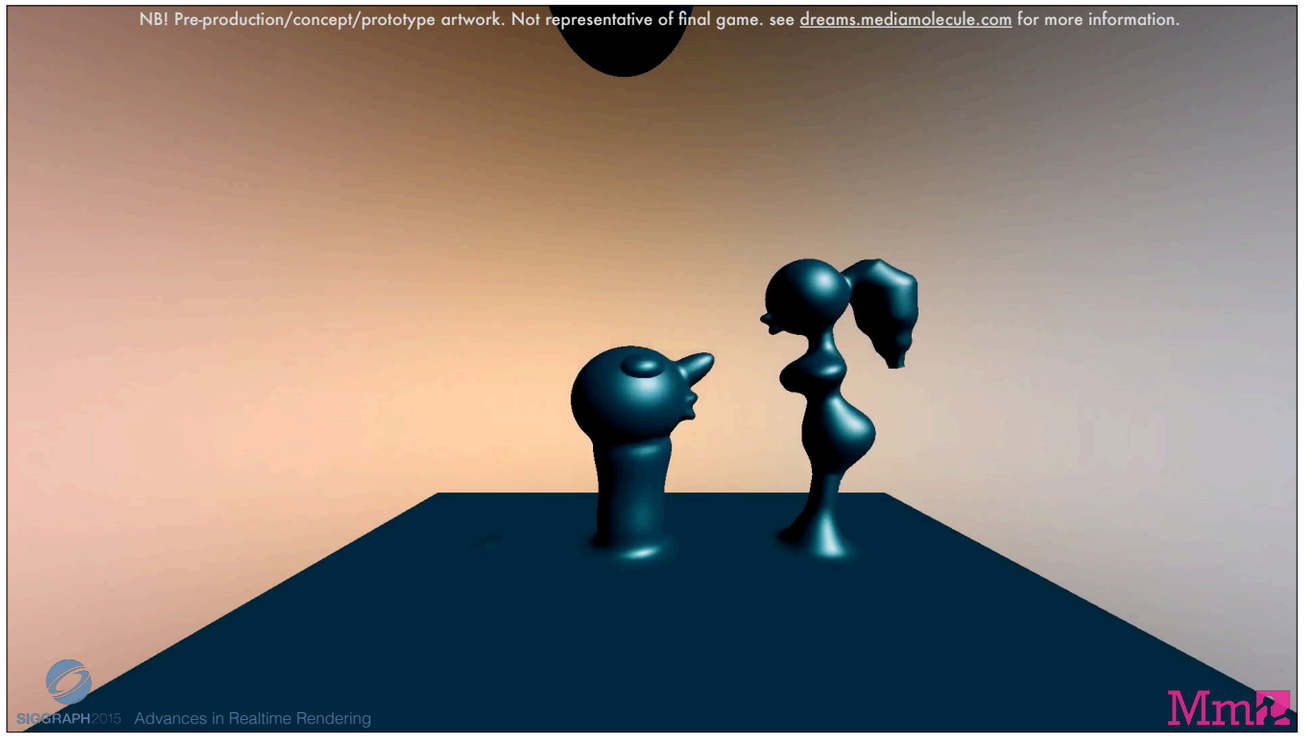


rigid-ish pieces (low resolution FFD deformer over rigid pieces):

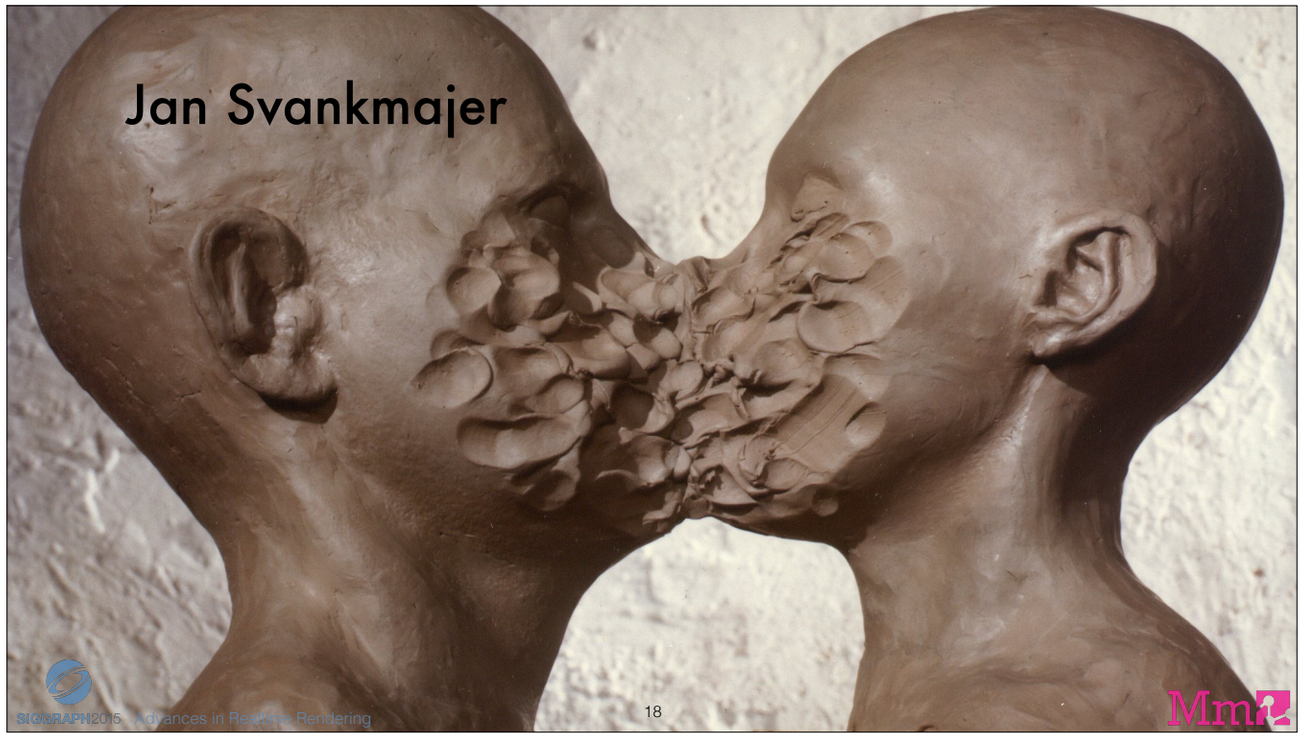


competing with that was the idea of animating the edits themselves. the results were quite compelling -

this was an offline render using 3DS Max’s blob mode to emulate soft blends. but it shows the effect.



this was in Anton’s PC prototype, re-evaluating and re-meshing every frame in realtime.

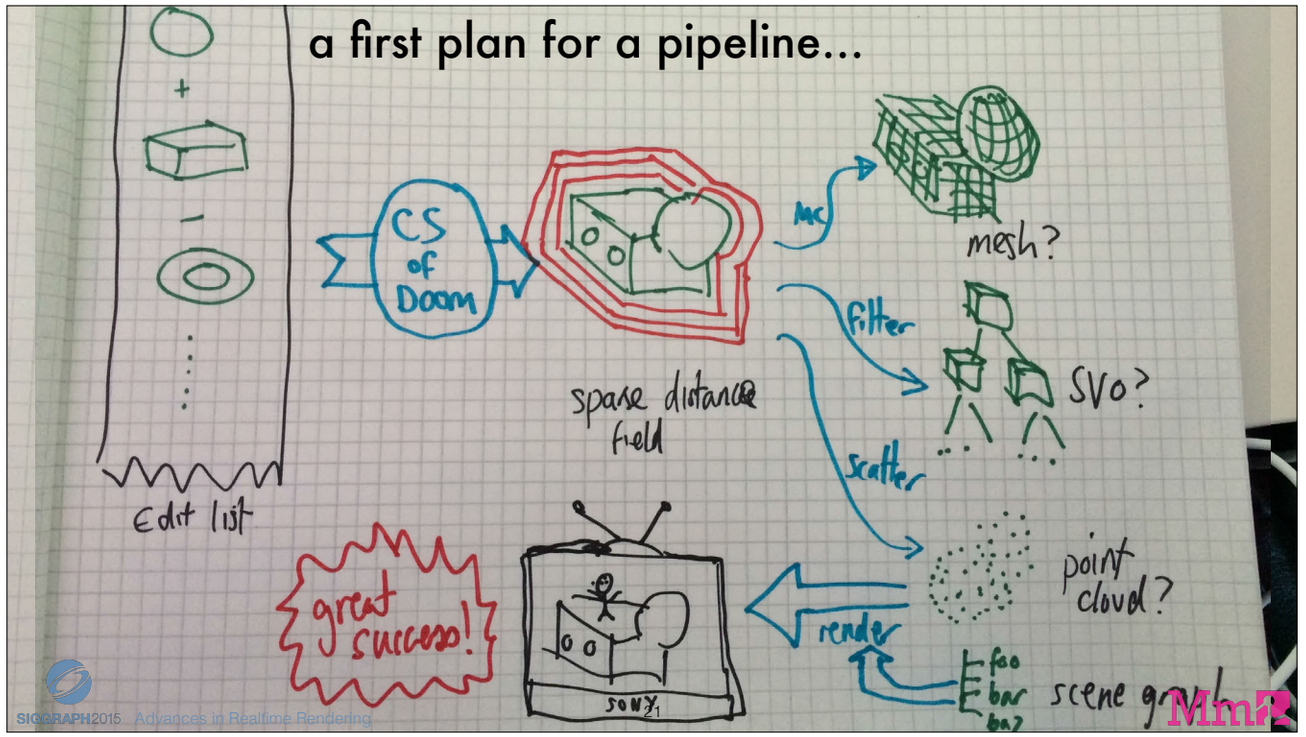


and there was a visual high bar, which everyone loved, inspired by the work of legendary claymation animator & film maker jan svankmajer

here we made stop motion by scrubbing through the edit history, time lapse style (just like the earlier dad's head). and on a more complex head model… pretty expensive to re-evaluate every frame though!

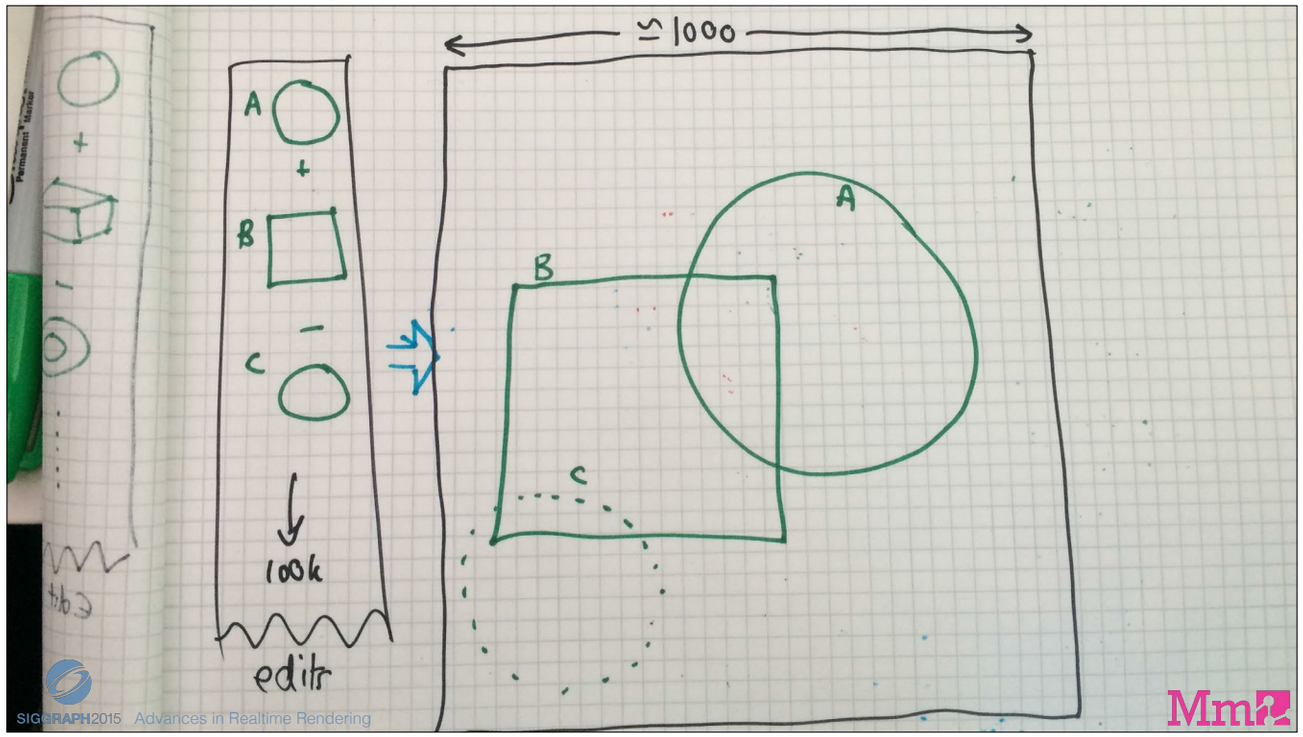
‘Oh F….’

however to achieve this, the SDF would need to be re-evaluated every frame. in the first pc prototype, we had effectively added each edit one at a time to a volume texture - it was great for incremental edits, but terrible for loading and animation. the goal of dreams is for UGC to be minimal size to download, so we can't store the SDF fields themselves anyway - we need a fast evaluator!



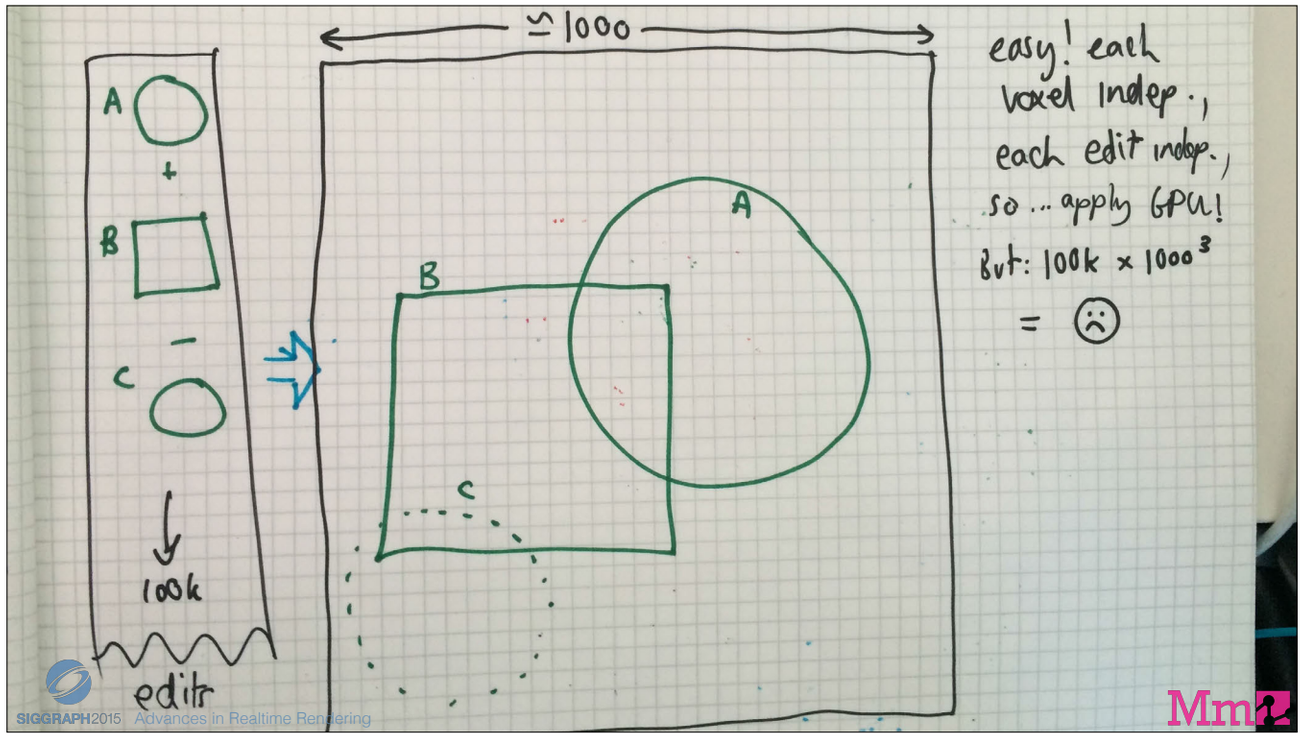
Nevertheless, a plan was forming! the idea was this
{'csg' edit list => CS of doom => per object voxels => meshing? => per object poly model =>
    scene graph
render! profit!
    Before getting to rendering, I'd like to talk about the CS of doom, or evaluator as we call it. The full pipeline from edit list to renderable data is 40+ compute shaders in a long pipeline, but the “CS of doom” are a few 3000+ instruction shaders chained together that make the sparse SDF output. fun to debug on early PS4 hardware!

here are some actual stats on dispatch counts for the a model called crystal’s dad to be converted from an edit list to a point cloud and a filtered brick tree:
eval dispatch count: 60
sweep dispatch count: 91
points dispatch count: 459
bricker dispatch count: 73

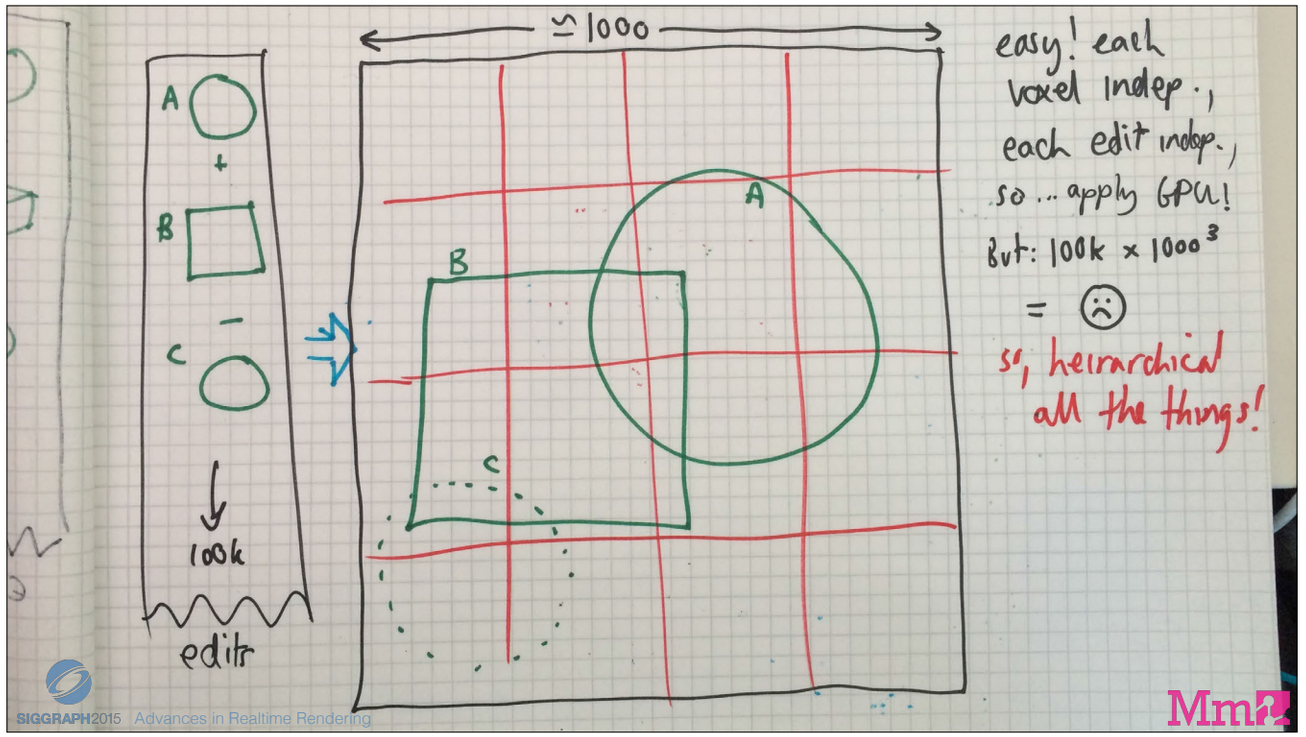


We had limited the set of edits to exclude domain deformation or any non-local effects like blur (much to the chagrin of z-brush experienced artists), and our CSG trees were entirely right leaning, meaning they were a simple list. Simple is good!

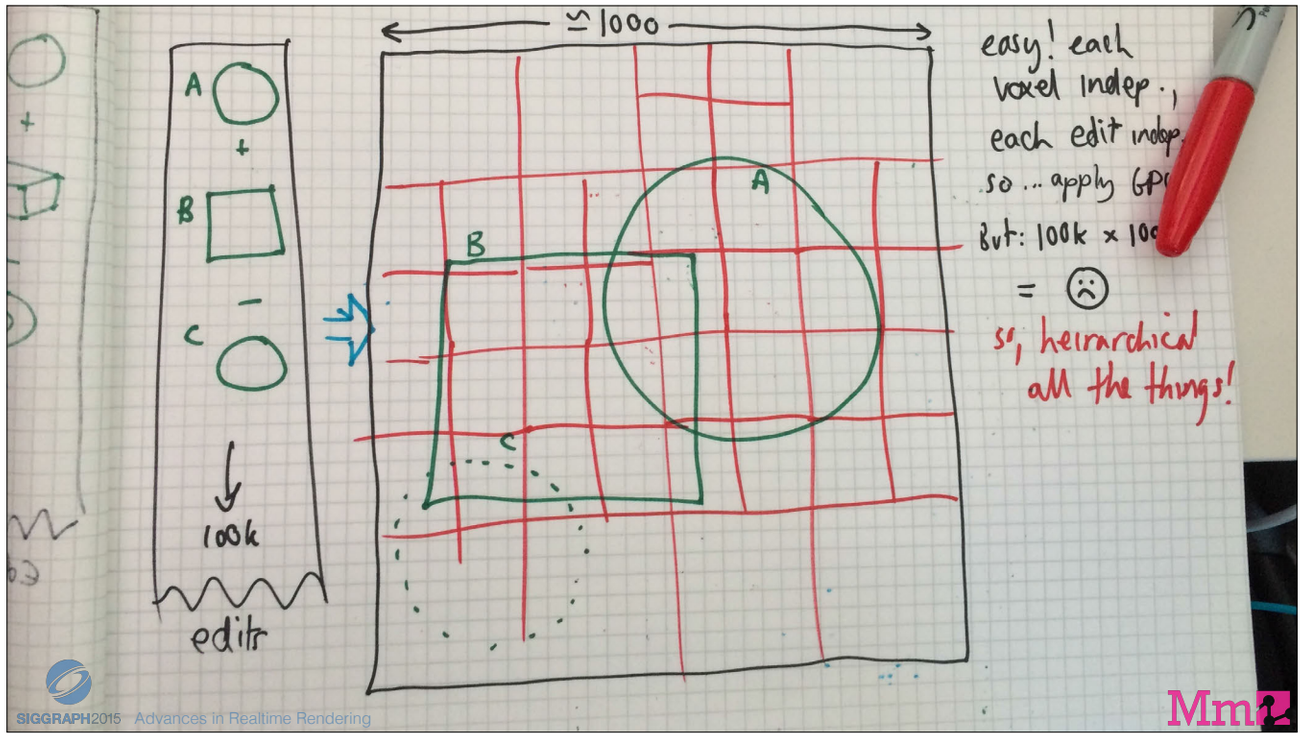
so in *theory* we had an embarrassingly parallel problem on our hands. take a large list of 100k edits, evaluate them at every point in a ~1000^3 grid, mesh the result, voila! one object!



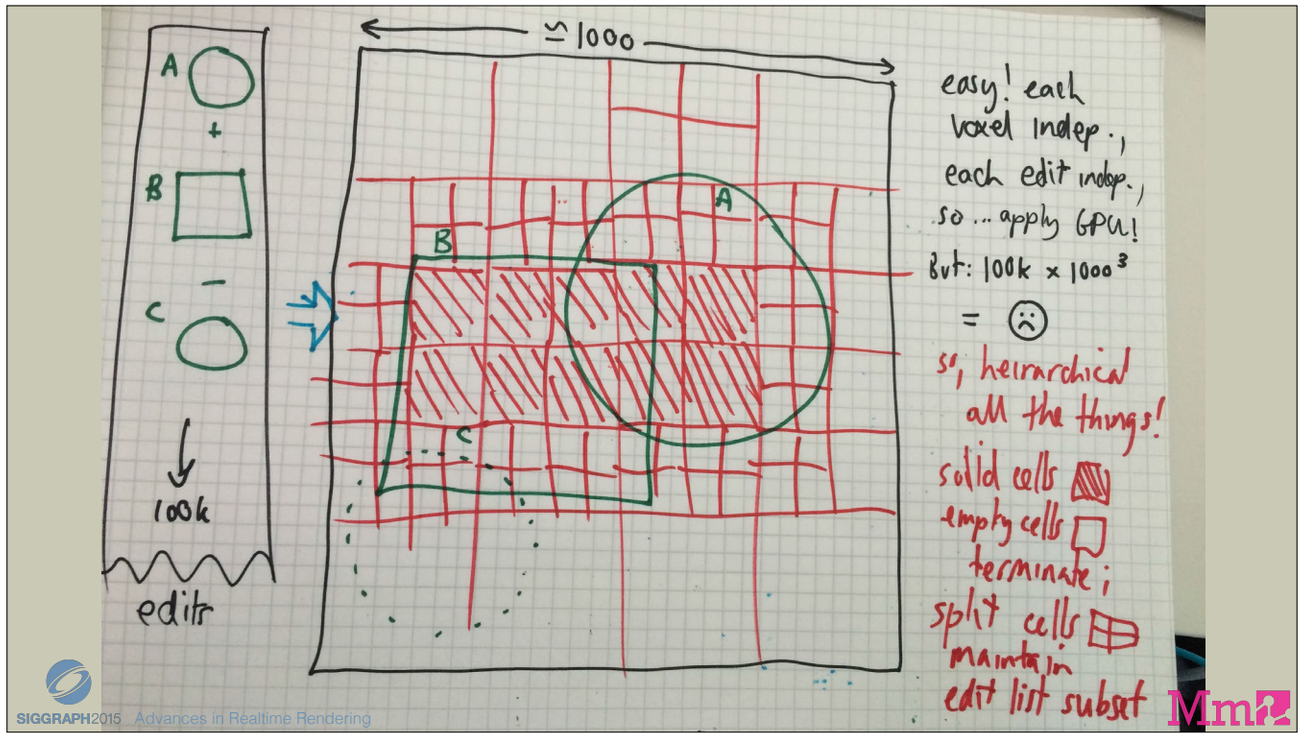
alas, that's 100 billion evaluations, which is too many.



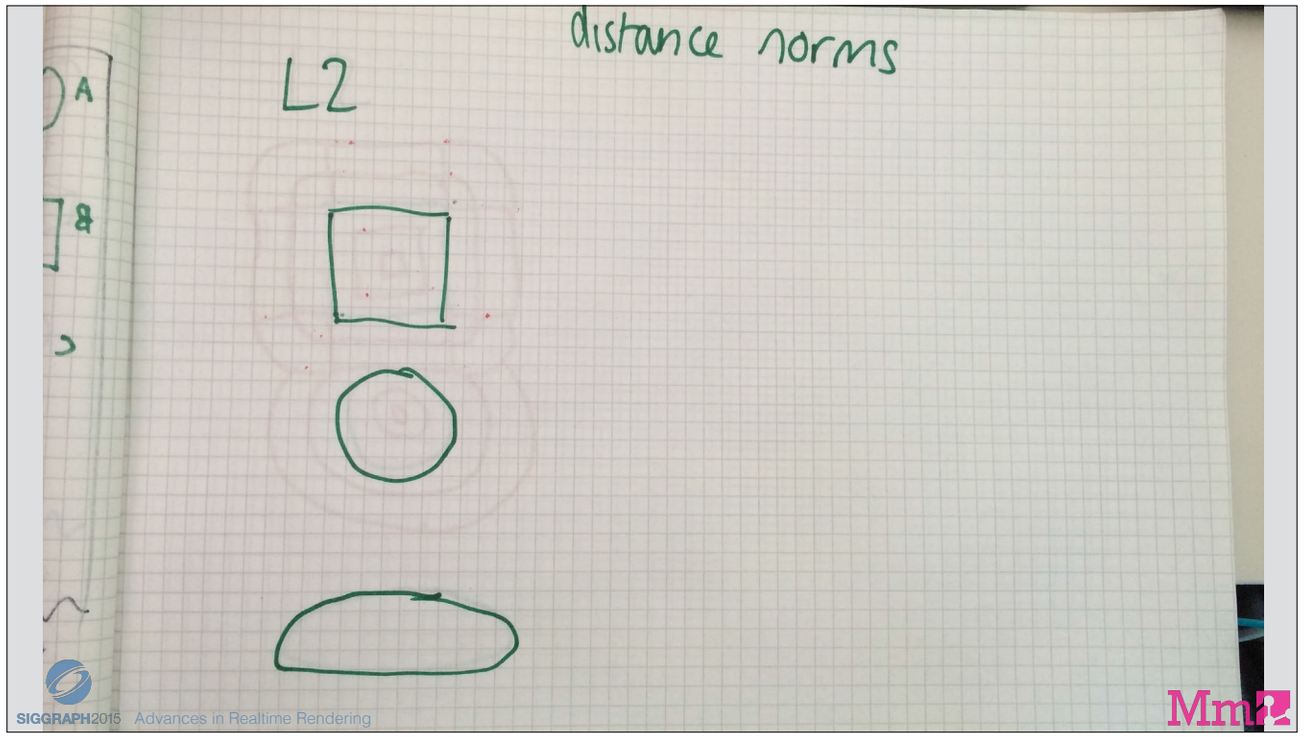
anton wrote the first hierarchical prototype, which consisted of starting with a very coarse voxel grid, say 4x4x4
{slide}



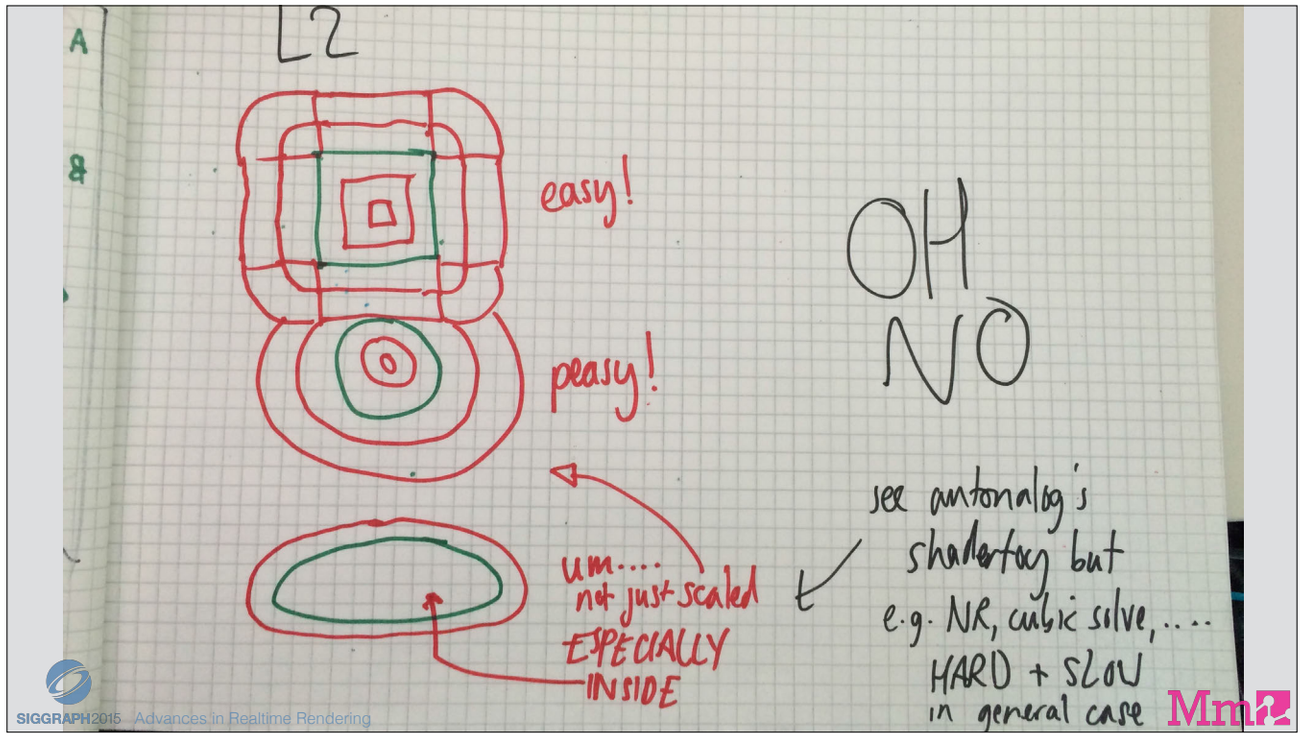
building a list of edits that could possibly overlap each voxel, and then iteratively refining the voxels by splitting them and shortening the lists for each.



empty cells and full cells are marked early in the tree; cells near the boundary are split recursively to a resolution limit. (the diagram shows a split in 2x2, but we actually split by 4x4x4 in one go, which fits GCN's 64 wide wavefronts and lets us make coherent scalar branches on primitive type etc)
the decision to split a given cell and when not to, is really tricky. if you err on the 'too little split' side, you get gaps in the model. most of the renderering backends we were trying required at least 1 to 1.5 voxels of valid data on each side of the mesh.
if you err on the 'too much split' side, you can easily get pathological cases where the evaluator ends up doing orders of magnitude too much work.
Also, the splits must be completely seamless.
The quality constraints are much, much more stringent than what you'd need for something like sphere tracing.
Both Anton and I had a crack at various heuristic evaluators, but neither was perfect. And it was made worse by the fact that even some of our base primitives, were pretty hard to compute 'good' distances for!

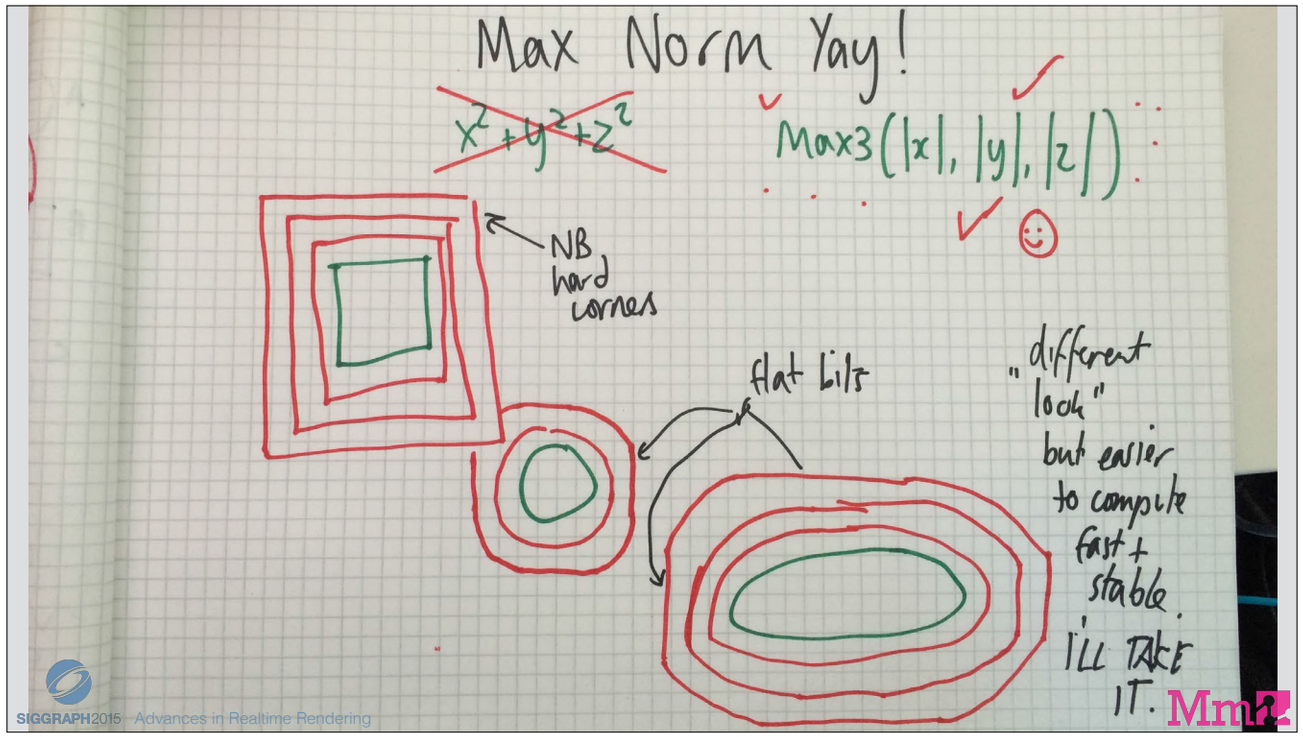


an aside on norms. everyone defaults to the L2 distance (ie x^2+y^2+z^2) because it’s the length we’re used to.



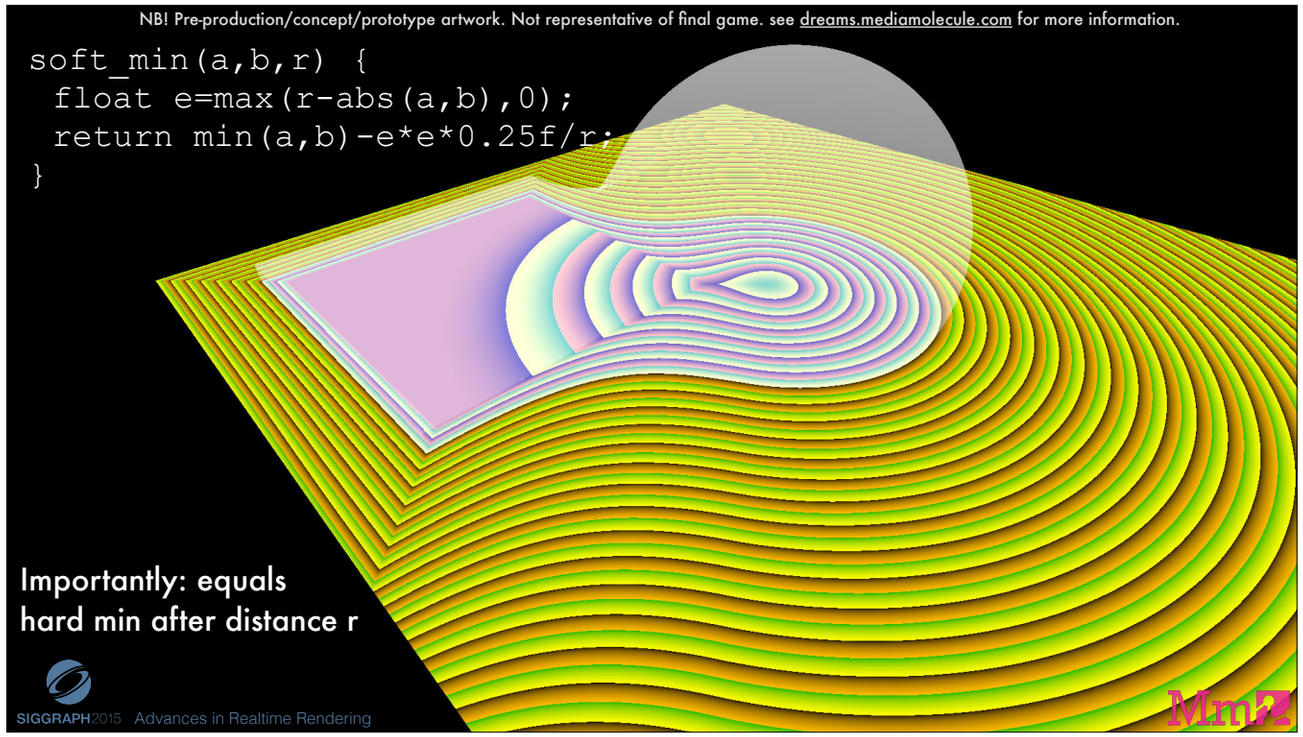
the L2 norm for boxes and spheres is easy. but the ellipsoid… not so much. Most of the public attempts at ‘closest point on an ellipsoid’ are either slow, unstable in corner cases, or both. Anton spent a LONG time advancing the state of the art, but it was a hard, hard battle.

Ellipsoid: https://www.shadertoy.com/view/ldsGWX
Spline: https://www.shadertoy.com/view/XssGWl

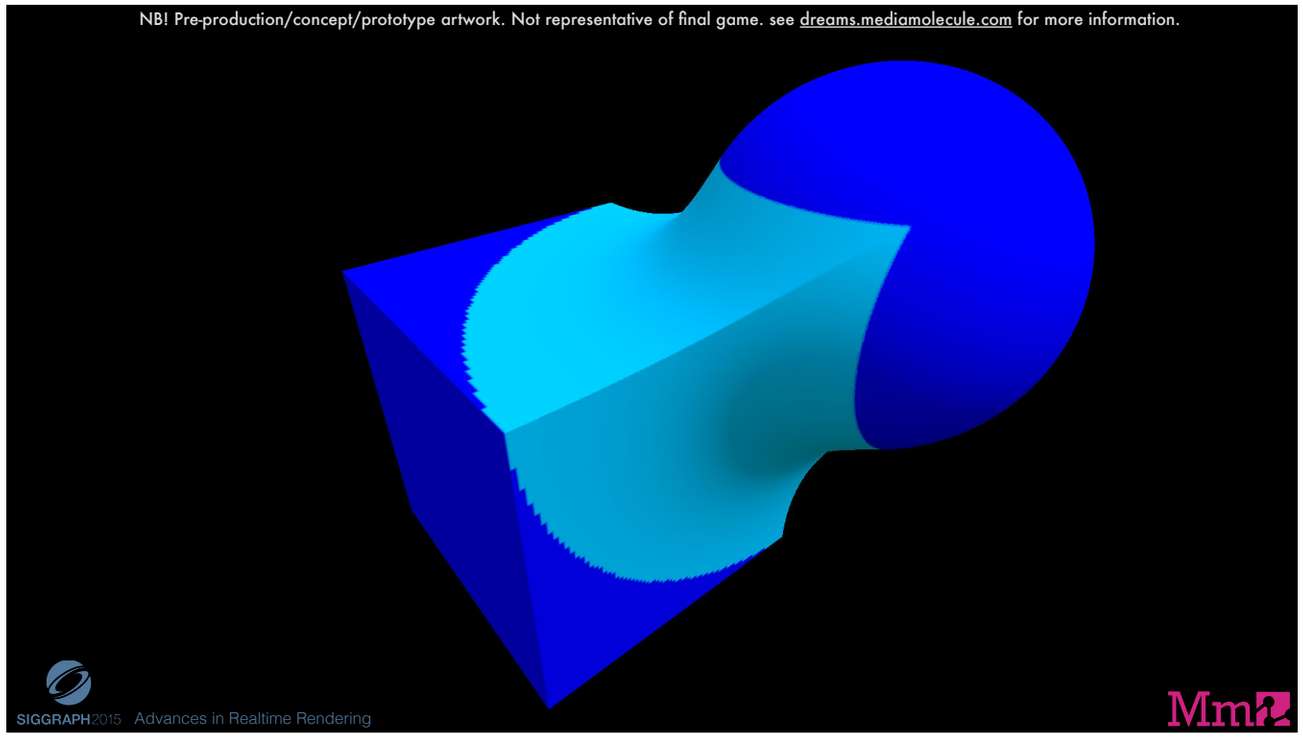


luckily, anton noticed that for many primitives, the max norm was simpler and faster to evaluate.
Insight from "Efficient Max-Norm Distance Computation and Reliable Voxelization" http://gamma.cs.unc.edu/RECONS/maxnorm.pdf
-Many non-uniform primitives have much simpler distance fields under max norm, usually just have to solve some quadratics!
- Need to be careful when changing basis as max norm is not rotation-invariant, but a valid distance field is just a scaling factor away

So evaluator works in max norm i.e. d = max(|x|,|y|,|z|). The shape of something distance ‘d’ away from a central origin in max norm is a cube, which nicely matches the shape of nodes in our hierarchy. :)



Soft blend breaks ALL THE CULLING, key points:
- Soft min/max needs to revert to hard min/max once distance fields are sufficiently far apart (otherwise you can never cull either side)
- Ours is for some radius r: soft_min(a, b, r) { float e = max(r - abs(a - b), 0); return min(a, b) - e*e*0.25/r; }, credit to Dave Smith @ media molecule
- Has no effect once abs(a - b) > r
- Need to consider the amount of ‘future soft blend’ when culling, as soft blend increases the range at which primitives can influence the final surface (skipping over lots of implementation details!)
- Because our distance fields are good quality, we can use interval arithmetic for additional culling (skipping over lots of implementation details!)



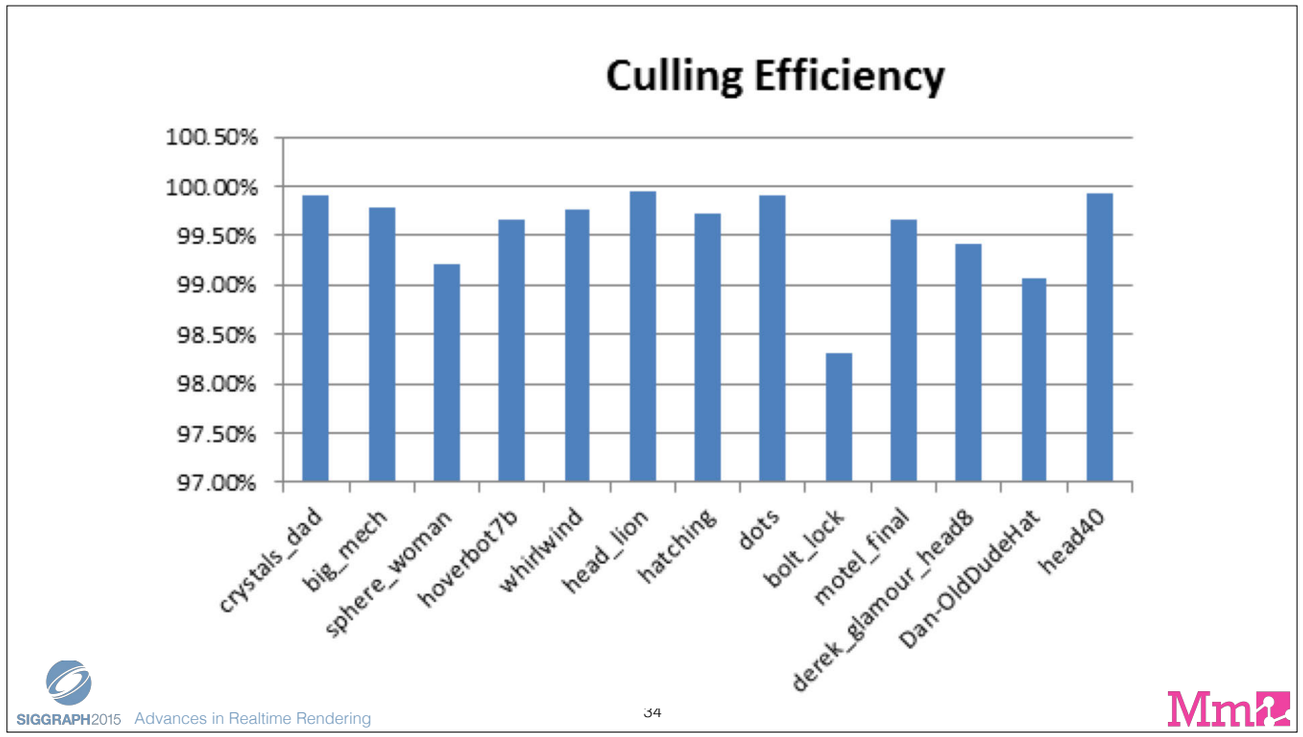
this is a visualisation of the number of edits affecting each voxel; you can see that the soft blend increases the work over a quite large area. however, compared to the earlier, less rigorous evaluators, simon’s interval-arithmetic and careful-maxnorm-bounds was a tour-de-force of maths/engineering/long dependent compute shader chains/compiler bug battling.

Simon ‘I break compilers’ FTW

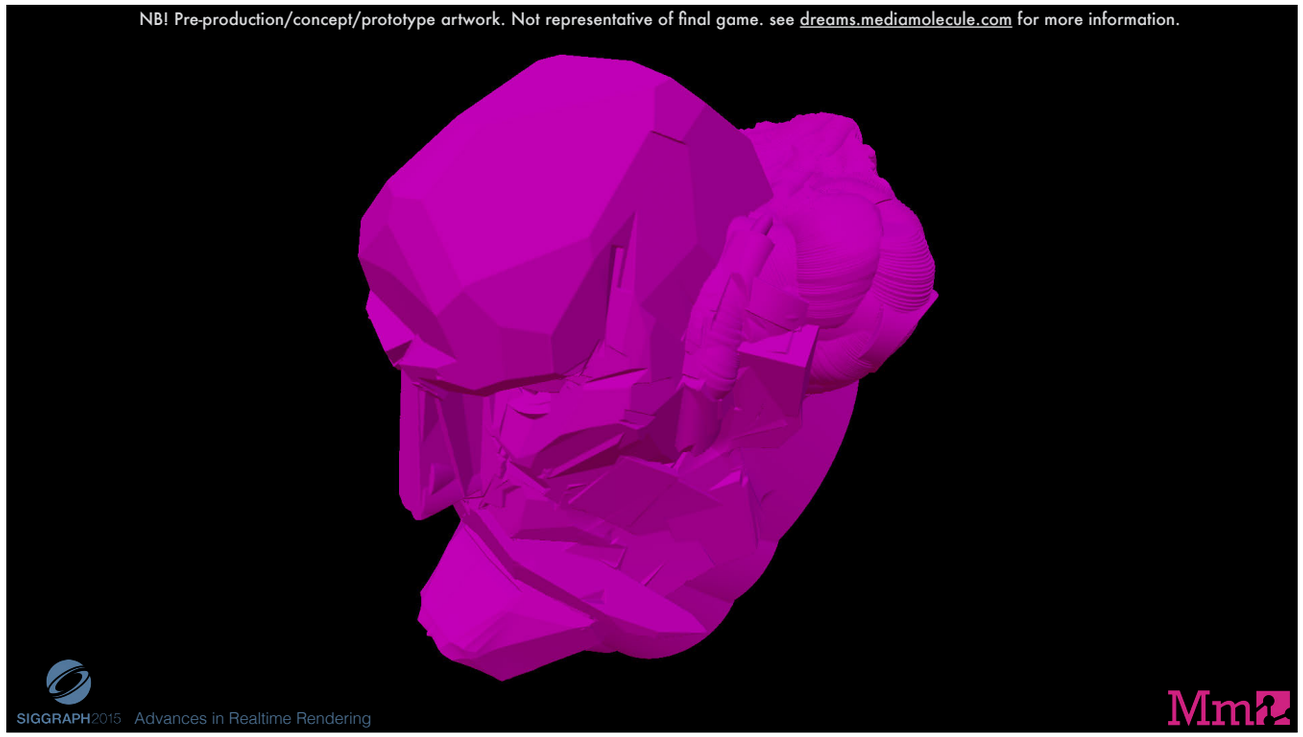
thanks for saving the evaluator sjb!

Filename	ElementCount	BlockCount	BlockEvalCount	VoxelCount	EvaluatorTime	ClusterCount	BrickCount		Time Per Prim Per Block	Culling Efficiency
crystals_dad	8274	235286	1879414	5241466	0.09109472	11190	5497		0.00000004847	99.90%
big_mech	850	156759	292582	4119346	0.01510454	8651	5741		0.00000005163	99.78%
sphere_woman	24407	36625	6995798	848194	0.3423158	1408	694		0.00000004893	99.22%
hoverbot7b	604	168234	349221	4317399	0.01616372	8411	5187		0.00000004629	99.66%
whirlwind	791	410945	763795	10411468	0.0344279	19222	10431		0.00000004507	99.77%
head_lion	53976	182451	4148168	4676180	0.2271382	10555	5866		0.00000005476	99.96%
hatching	736	68793	143484	1729905	0.009373858	3939	4446		0.00000006533	99.72%
dots	1480	149988	190265	3816626	0.01703693	6241	4436		0.00000008954	99.91%
bolt_lock	1830	41936	1302908	1018246	0.02170936	2298	1216		0.00000001666	98.30%
motel_final	610	373423	752668	9214195	0.03267893	17606	6882		0.00000004342	99.67%
derek_glamour_head8	3824	266152	5906261	6997031	0.1903296	12668	5052		0.00000003223	99.42%
Dan-OldDudeHat	907	146283	1235569	3923554	0.04069558	7159	3606		0.00000003294	99.07%
head40	22484	96477	1593633	2464223	0.1425964	5087	2571		0.00000008948	99.93%

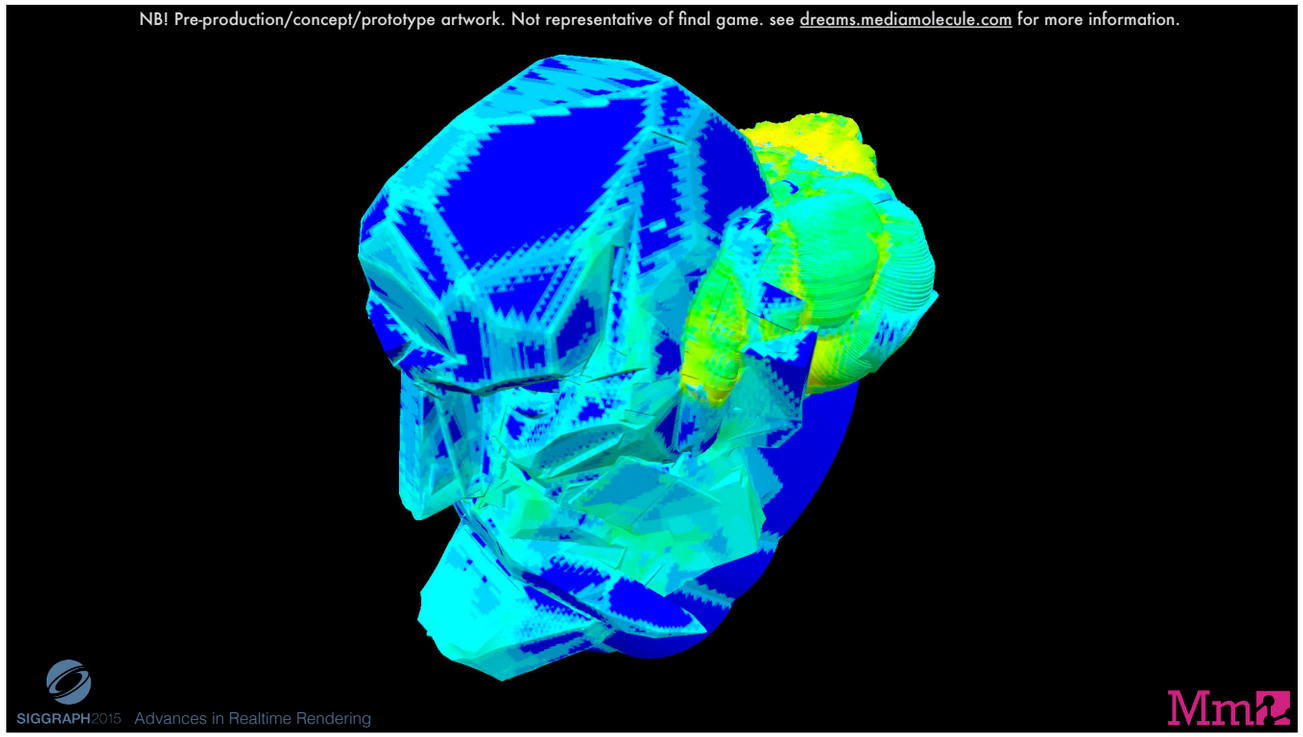
STATS! for some test models, you can see a range of edits (‘elements’) from 600 - 53000 (the worst is around 120k, but thats atypical); this evaluates to between 1m and 10m surface voxels (+-1.5 of surface),



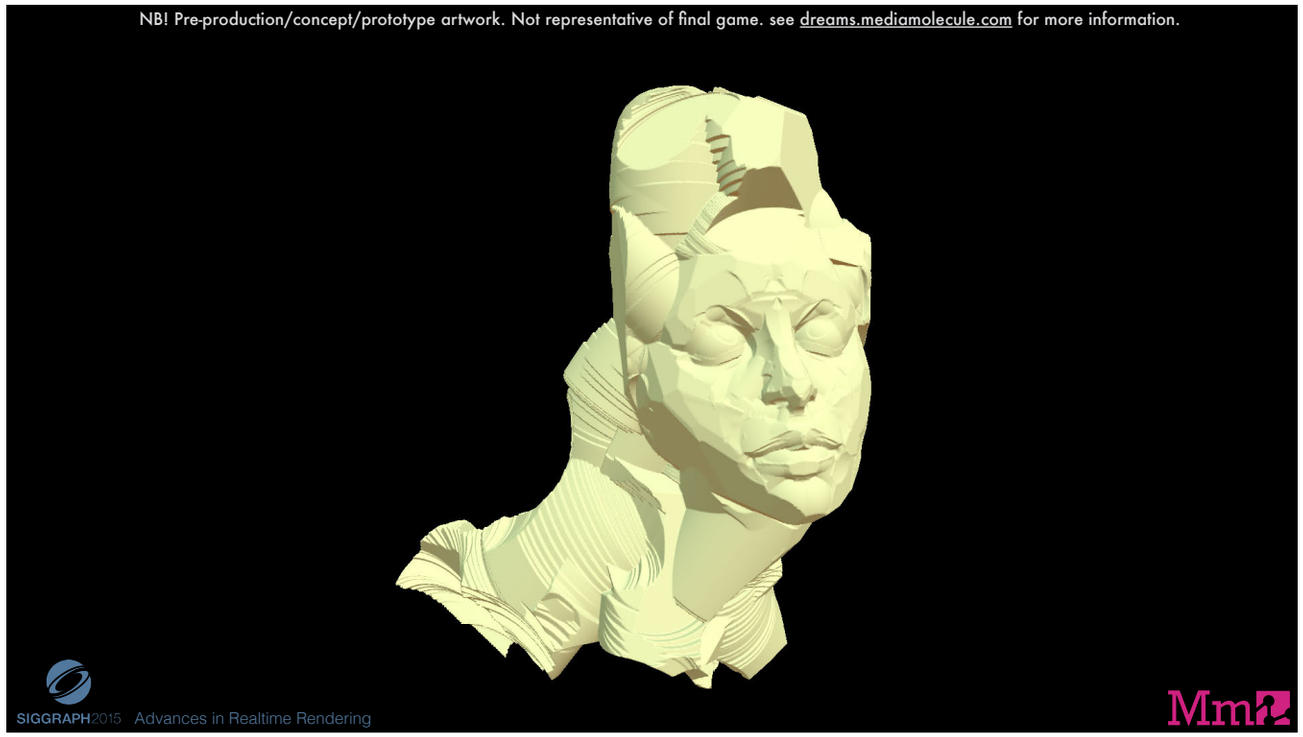
… the culling rates compared to brute force are well over 99%. we get 10m - 100m voxels evaluated per second on a ps4, from a model with tens of thousands of edits.

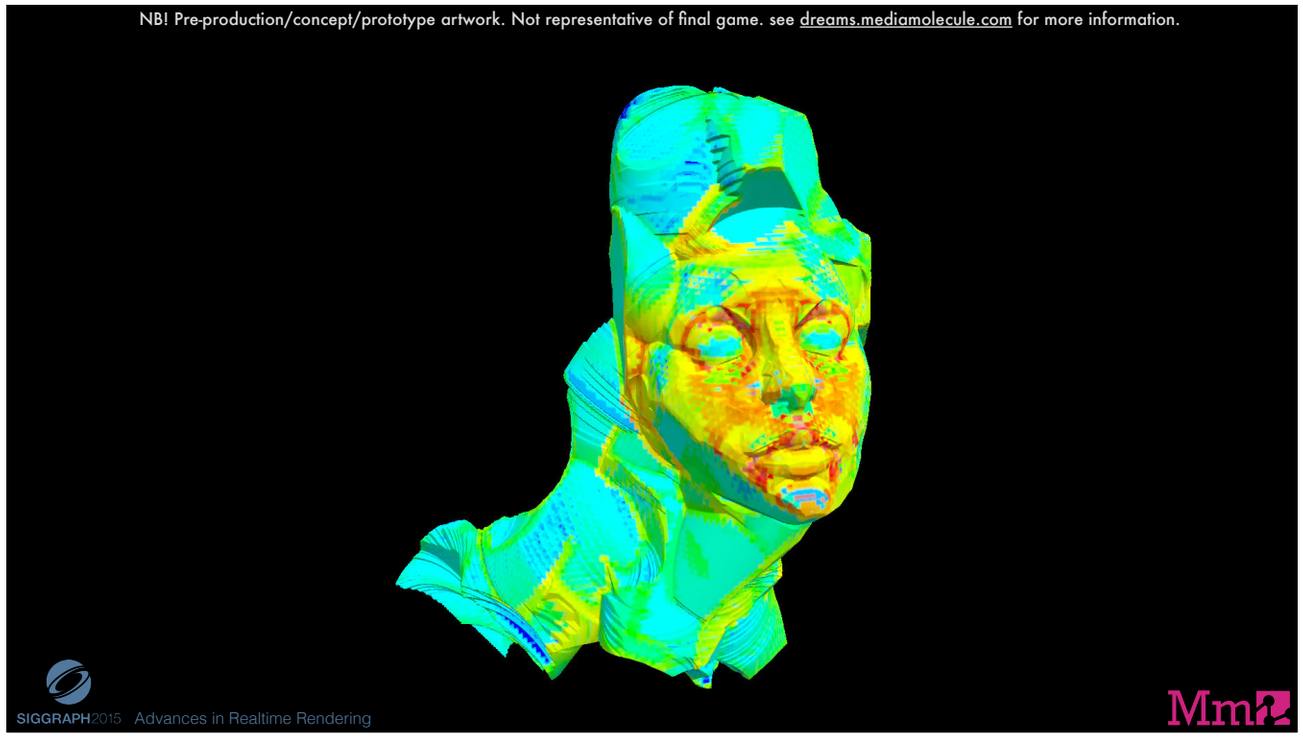


this is one of those models… (crystals dad, 8274 edits, 5.2m voxels)



…and this is a visualisation of the number of edits that touch the leaf voxels


NB! Pre-production/concept/prototype artwork. Not representative of final game. see dreams.mediamolecule.com for more information.
SIGGRAPH2015 Advances in Realtime Rendering
Mm



moar (head40) (22k edits, 2.4m voxels)
note the colouring is per block, so the voxel res is much higher than the apparent color res in this debug view

Rendering attempt 1 - polys

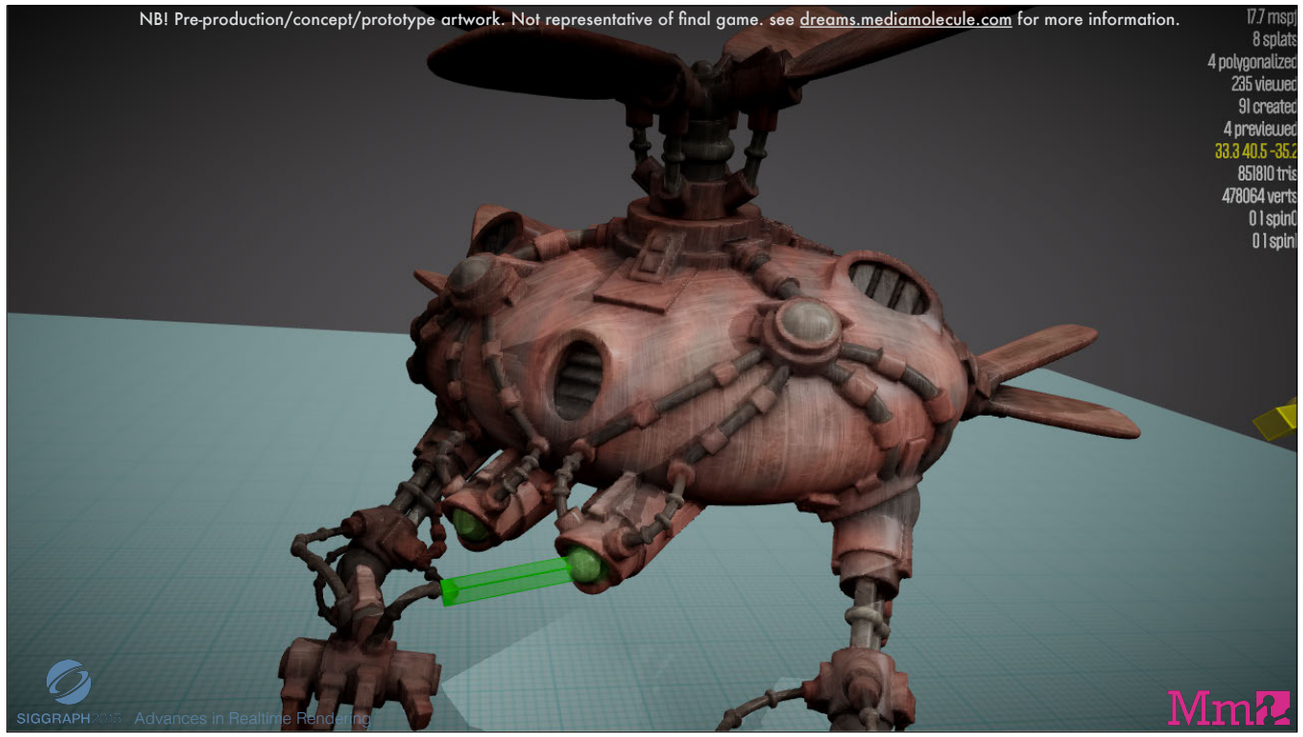
aka...

the meshes output from the blob prototype, as it was called, were generally quite dense - 2m quads at least for a large sphere, and more as the thing got more crinkly. In addition, we wanted to render scenes consisting of, at very least, a 'cloud' of rigidly oriented blob-meshes. at this point anton and I started investigating different approaches. anton looked into adaptive variants of marching cubes, such as dual marching cubes, various octree schemes, and so on. let's call this engine - including the original histopyramids marching cubes, engine 1: the polygon edition.

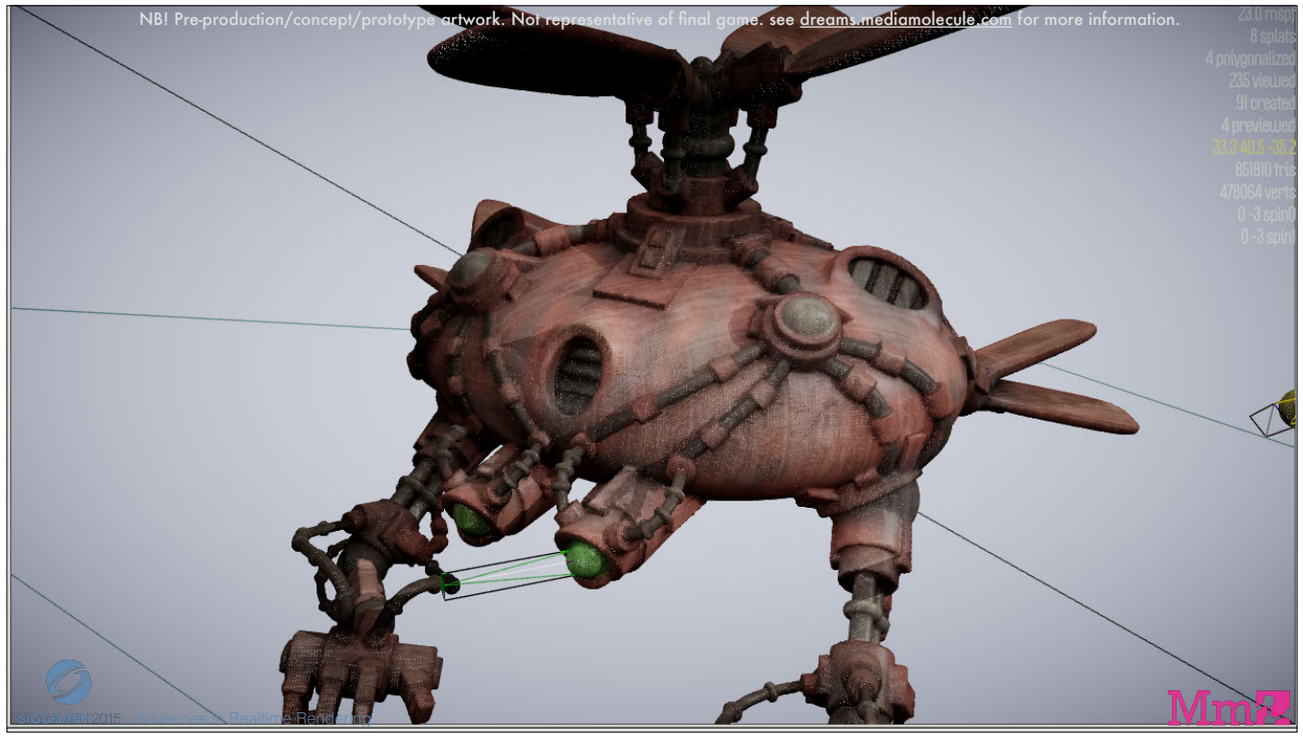
Cultural Learnings For Make Benefit Glorious Polygonalization of Signed Distance Fields

by Anton Kirczenow

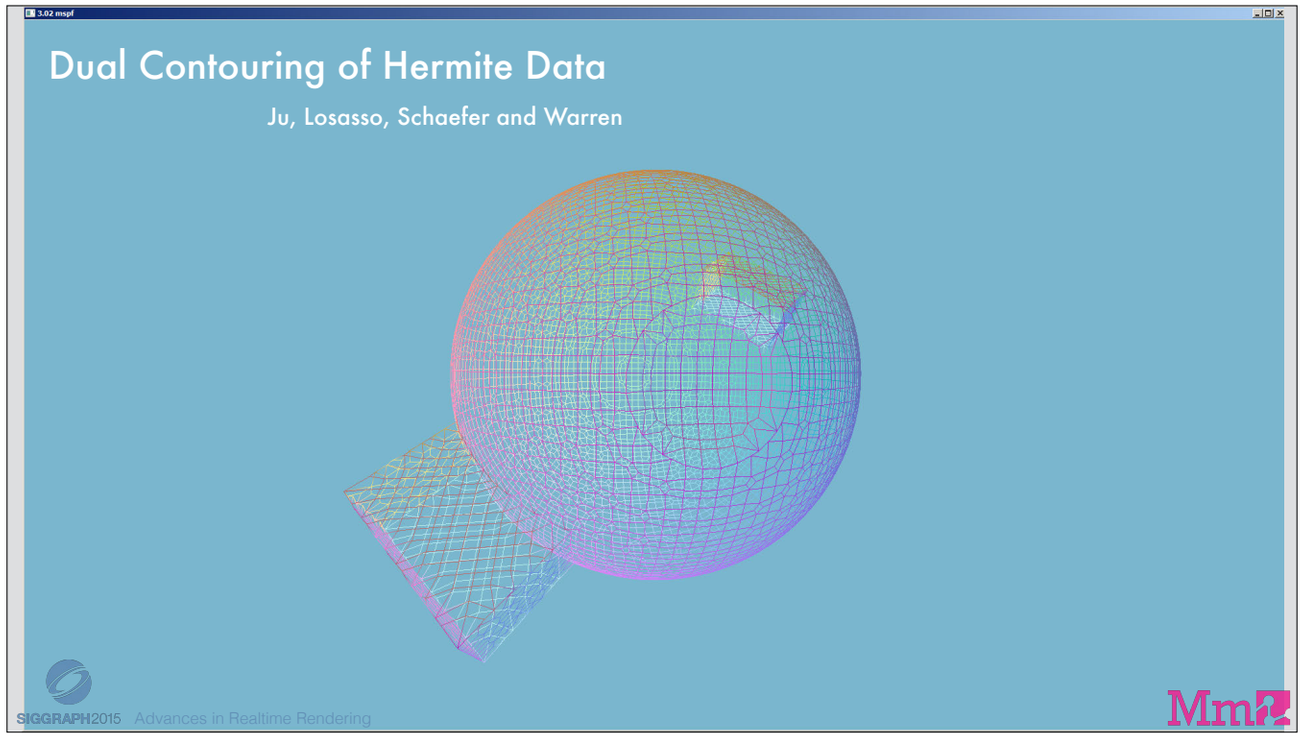
here are some notes from the man himself about the investigations SDF polygonalization



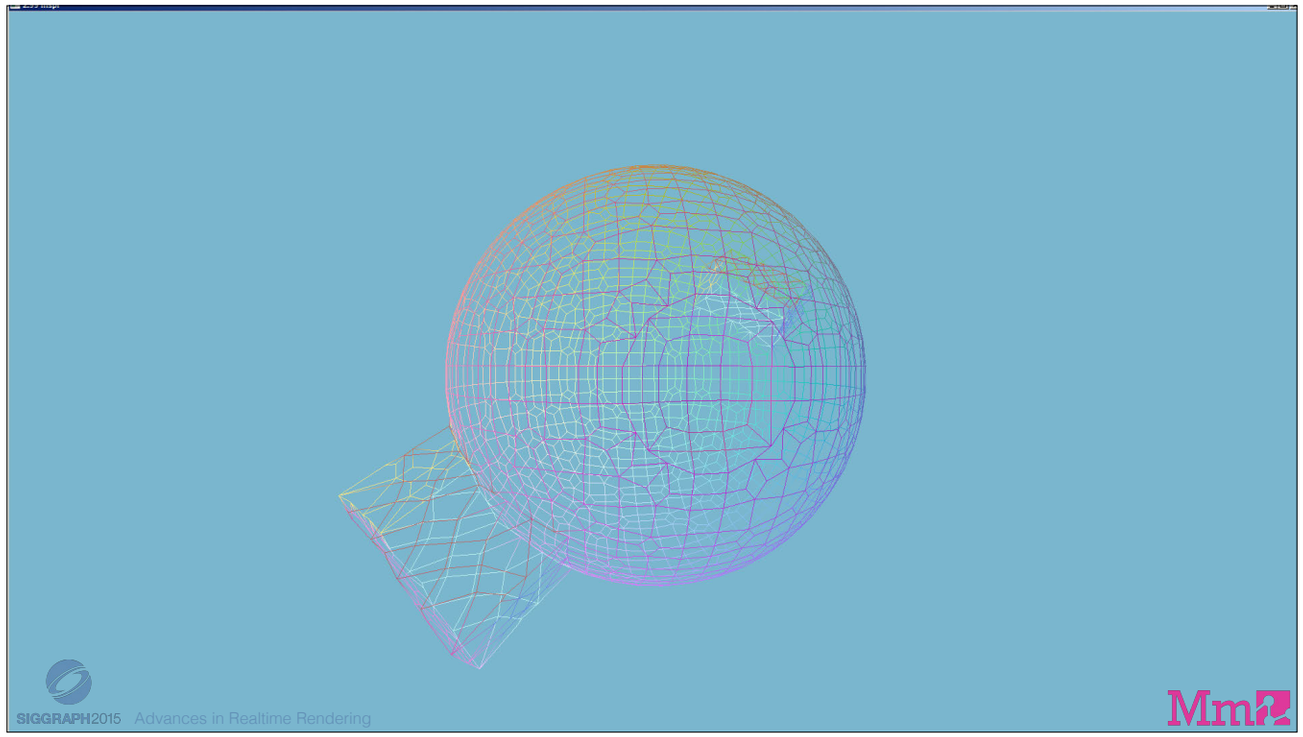
Marching cubes: Well it works but the meshes are dense and the edges are mushy and there are slivers and the output makes for asymmetrical code in a GPU implementation.



I dont know if you can tell but that’s the wireframe! oh no



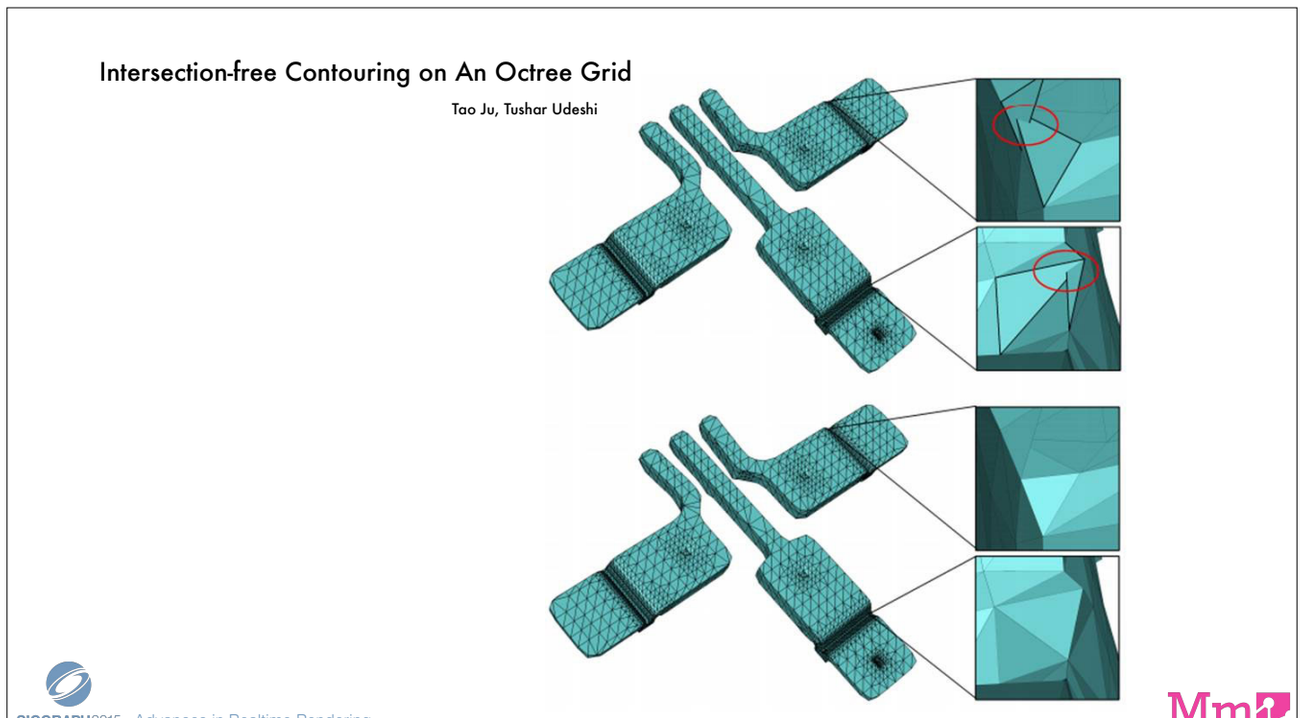
Dual Contouring: Hey this is easy on GPU. Oh but it’s kind of hard to keep sharp edges sharp and smooth things smooth and it doesn’t really align to features for edge flow either.
http://www.frankpetterson.com/publications/dualcontour/dualcontour.pdf
‘Dual Contouring of Hermite Data’
Ju, Losasso, Schaefer and Warren



note the wiggly edge on the bottom left of the cuboid - really hard to tune the hard/soft heuristics when making animated deathstars.

more complex model….

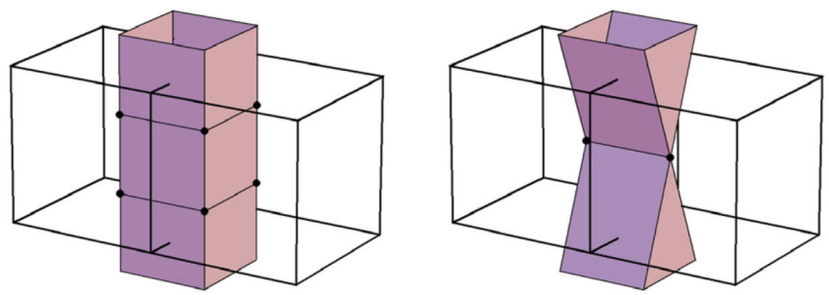
the DC mesh is still quite dense in this version, but at least it preserves edges. however it shows problems: most obviously, holes in the rotor due to errors in the evaluator we used at this stage (heuristic culling -> makes mistakes on soft blend; pre simon eval!) - also occasionally what should be a straight edge ends up wobbly because it cant decide if this should be smooth or straight. VERY tricky to tune in the general case for UGC.



ALSO! Oh no, there are self intersections! This makes the lighting look glitched - fix em:
http://www.cs.wustl.edu/~taoju/research/interfree_paper_final.pdf
‘Intersection-free Contouring on An Octree Grid’
Tao Ju, Tushar Udeshi

Manifold Dual Contouring

Scott Schaefer, Tao Ju, Joe Warren



(a) (b)

Fig. 1. Vertex clustering in two neighboring cells (a) results in a non-manifold edge on the surface (b).

Oh no, now it’s not necessarily manifold, fix that.
http://faculty.cs.tamu.edu/schaefer/research/dualsimp_tvcg.pdf
Manifold Dual Contouring
Scott Schaefer, Tao Ju, Joe Warren

ARGH!

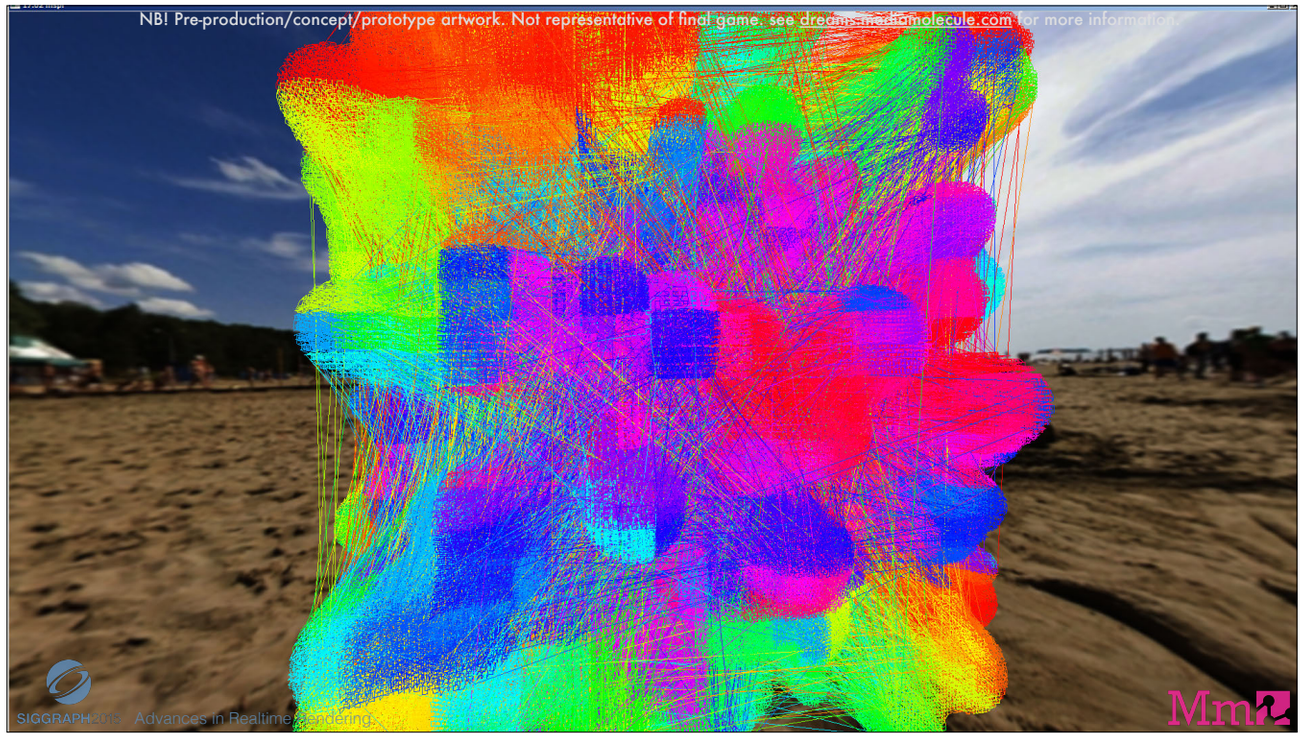
Manifold? Non-Self-Intersecting?
Pick one :(

Oh no, it’s self intersecting again. Maybe marching cubes wasn’t so bad after all... and LOD is still hard (many completely impractical papers).

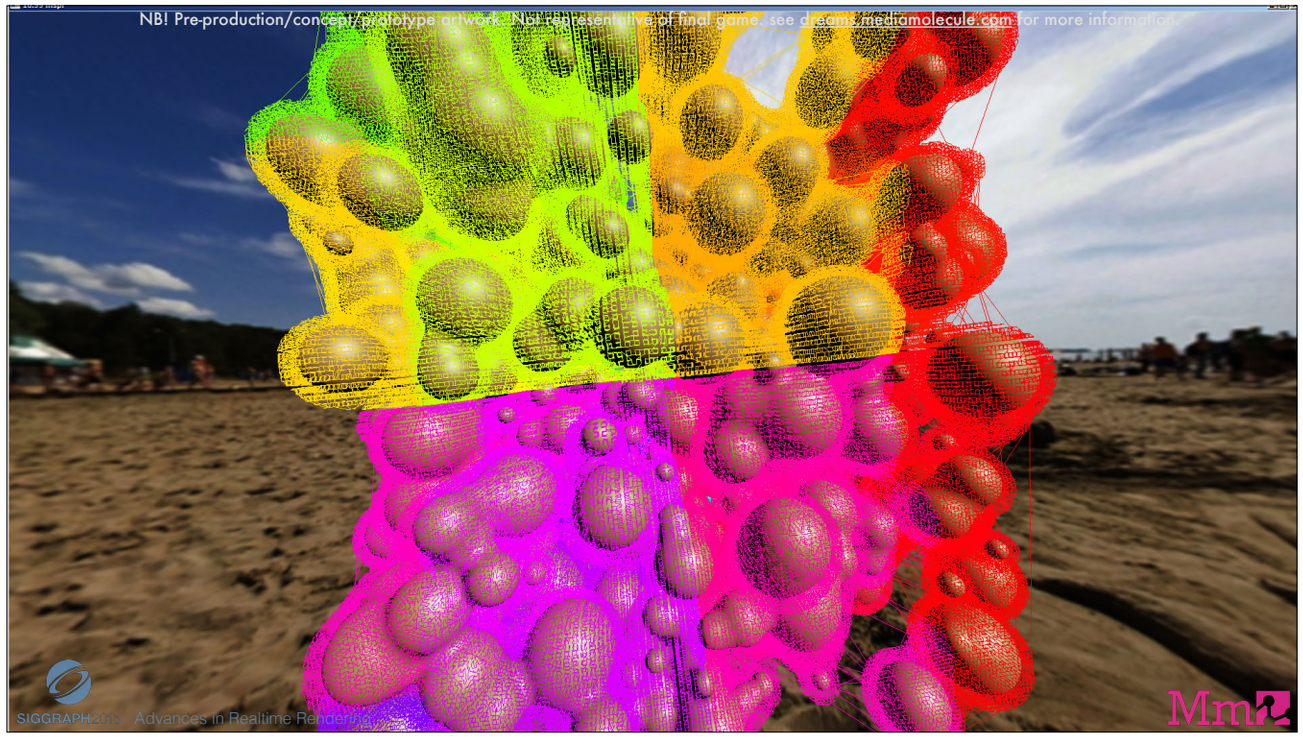
Aside: GCN ds_ordered_count FTW

deterministic append buffers we <3 u

the ability to accumulate to an ‘append buffer’ via DS_ORDERED_COUNT *where the results are magically in deterministic order based on wavefront dispatch index* is … magical and wonderful feature of GCN. it turns this…



(non deterministic vertex/index order on output from a mesher, cache thrashing hell:)



into this - hilbert ordered dual contouring! so much better on your (vertex) caches.
we use ordered append in a few places. it’s a nice tool to know exists!

back to the story! the answer to Isla’s question is,

No, Isla, I do not.

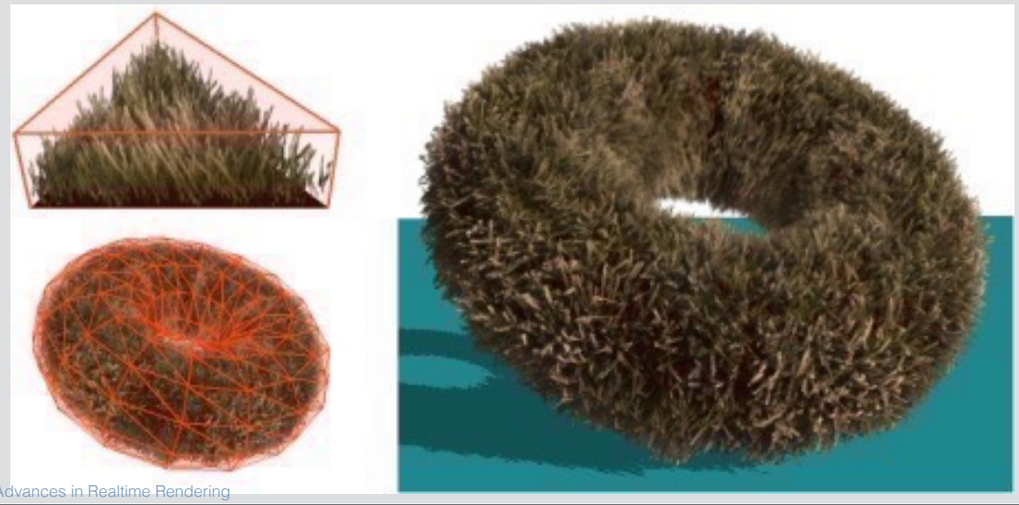
(they are amazing as a surface rep! but… they're surprisingly hard.
I don't like hard. I run away and return to ~~bitmaps~~ pictures…)

no, I do not like polygons.
I mean, they *are* actually pretty much the best representation of a hard 2D surface embedded in 3D, especially when you consider all the transistors and brain cells dedicated to them.
but… they are also very hard to get right automatically (without a human artist in the loop), and make my head hurt. My safe place is voxels and grids and filterable representations. Plus, I have a real thing for noise, grain, 'texture' (in the non texture-mapping sense), and I loved the idea of a high resolution volumetric representation being at the heart of dreams. it's what we are evaluating, after all. why not try rendering it directly? what could possibly go wrong?
so while anton was researching DC/MC/…, I was investigating alternatives.

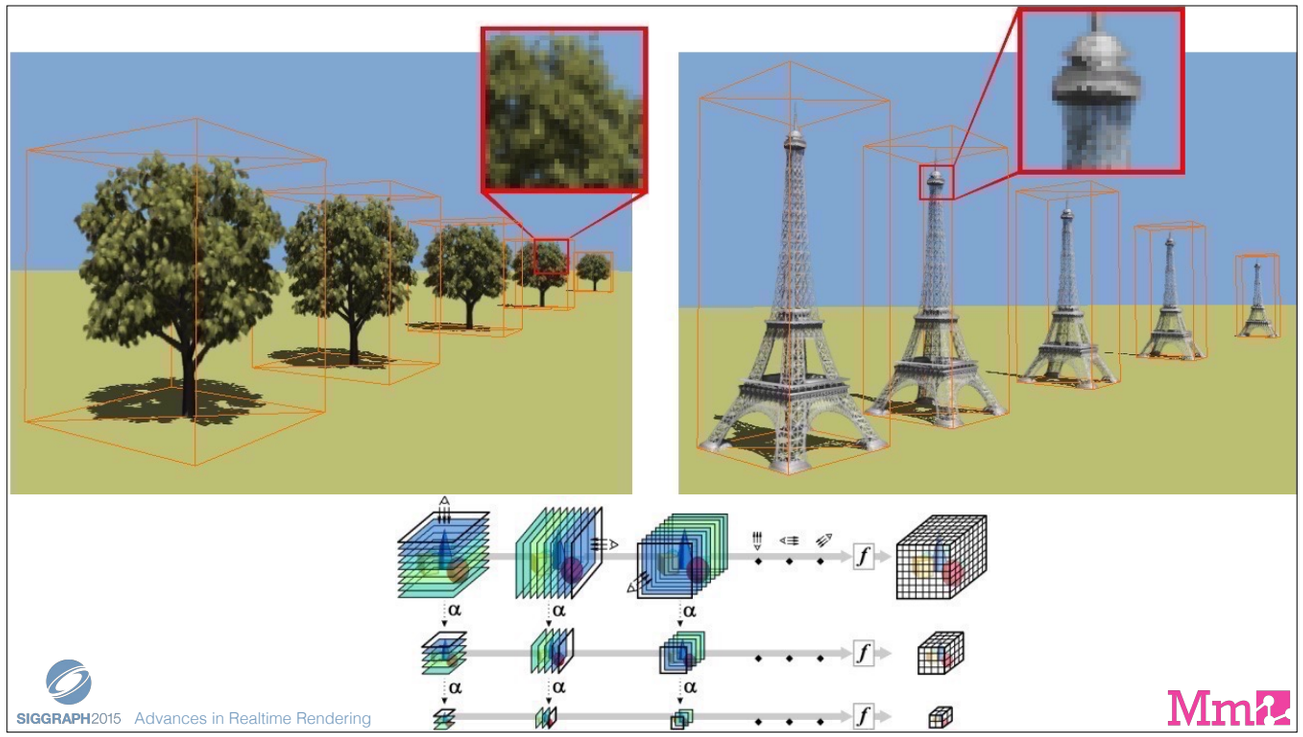
there was something about the artefacts of marching cubes meshes that bugged me. I really loved the detailed sculpts, where polys were down to a single pixel - and the lower res / adaptive res stuff struggled in some key cases.
so, I started looking into... other techniques.

Inspiration: Volumetric Billboards

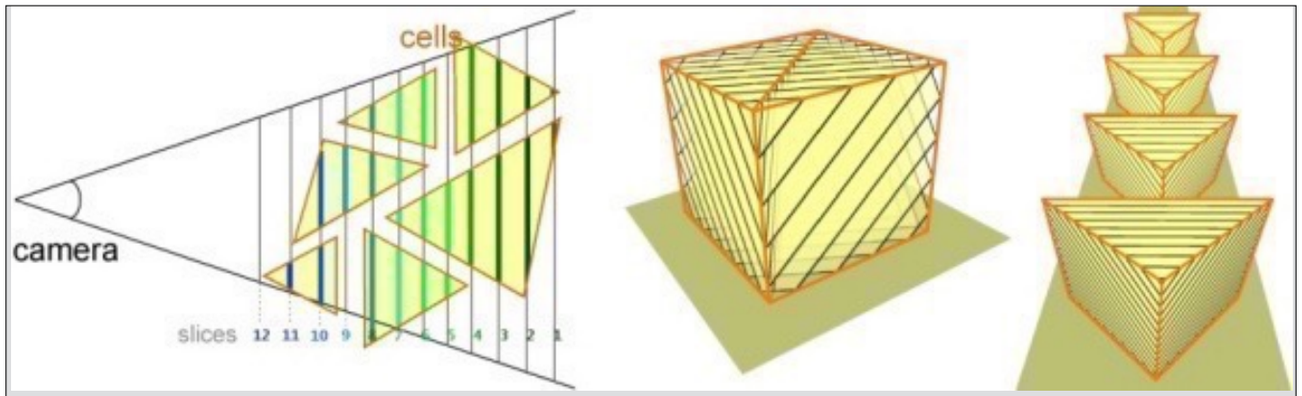
Philippe Decaudin, Fabrice Neyret



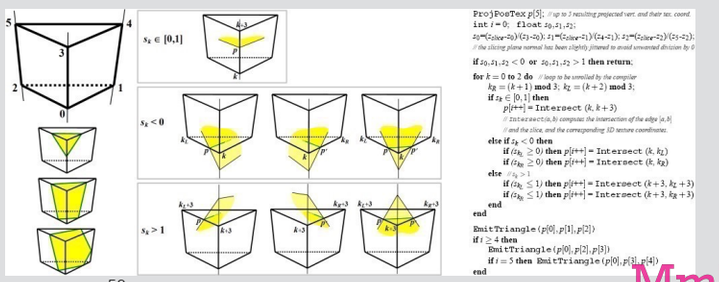
since the beginning of the project, I had been obsessed by this paper:
http://phildec.users.sourceforge.net/Research/VolumetricBillboards.php
by Philippe Decaudin, Fabrice Neyret. it's the spiritual precursor to gigavoxels, SVOs, and their even more recent work on prefiltered voxels. I became convinced around this time that there was huge visual differentiation to be had, in having a renderer based not on hard surfaces, but on clouds of prefiltered, possibly gassy looking, models. and our SDF based evaluator, interpreting the distances around 0 as opacities, seemed perfect.
this paper still makes me excited looking at it. look at the geometric density, the soft anti-aliased look, the prefiltered LODs. it all fitted!



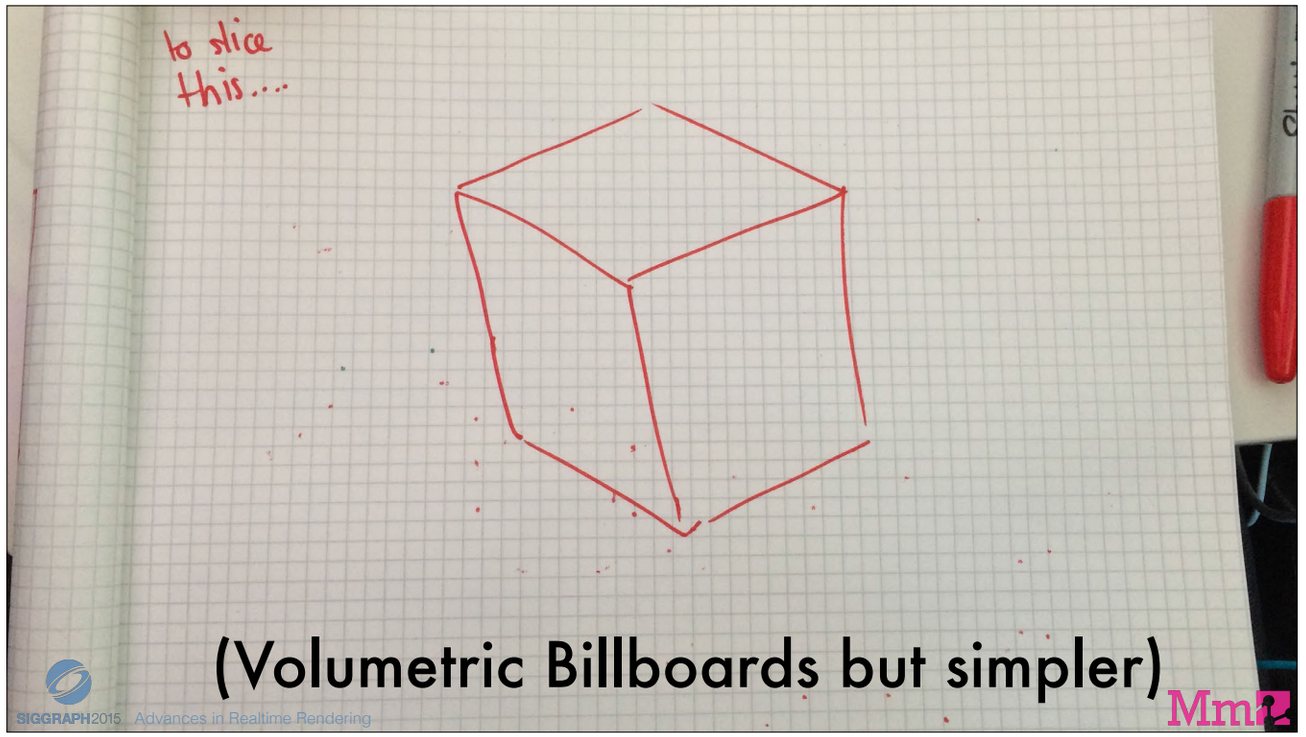
the paper contributed a simple LOD filtering scheme
{lod images}
based on compositing 'over' along each axis in turn, and taking the highest opacity of the three cardinal directions. this is the spiritual precursor to 'anisotropic' voxels used in SVO. I love seeing the lineage of ideas in published work. ANYWAY.



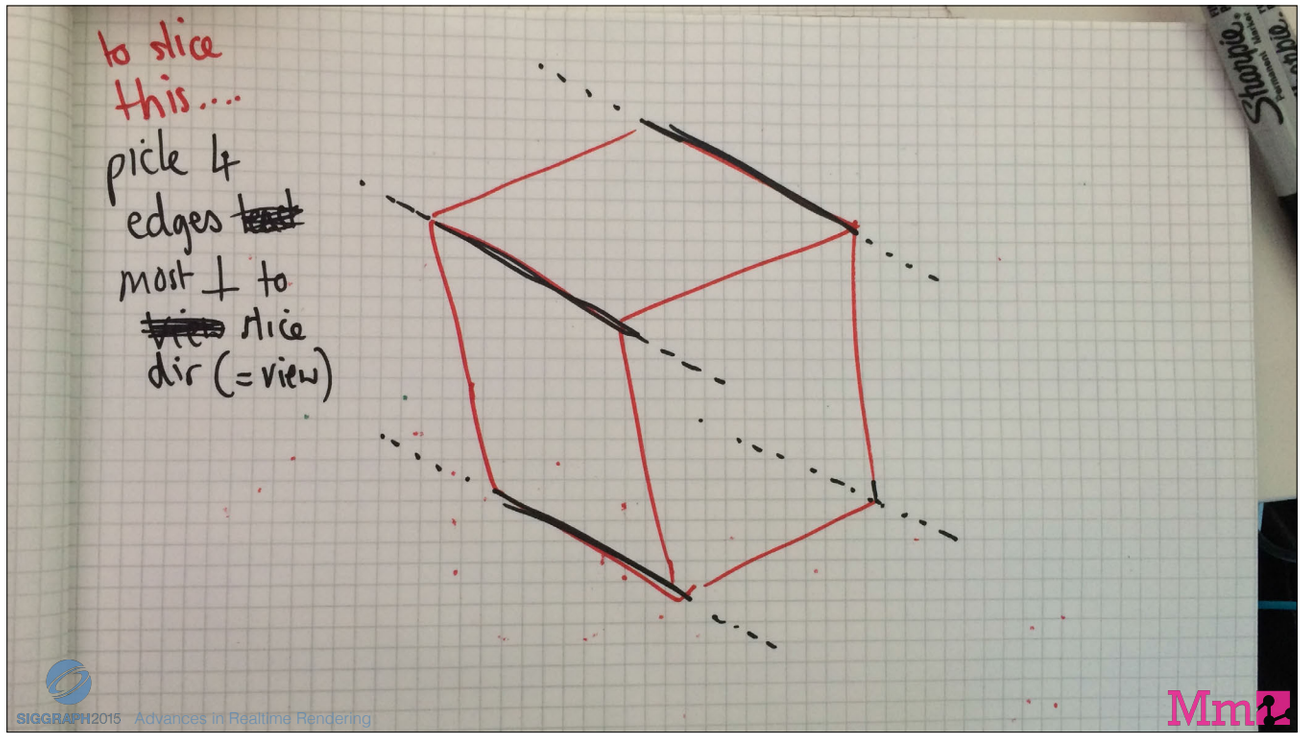
Overly complex->



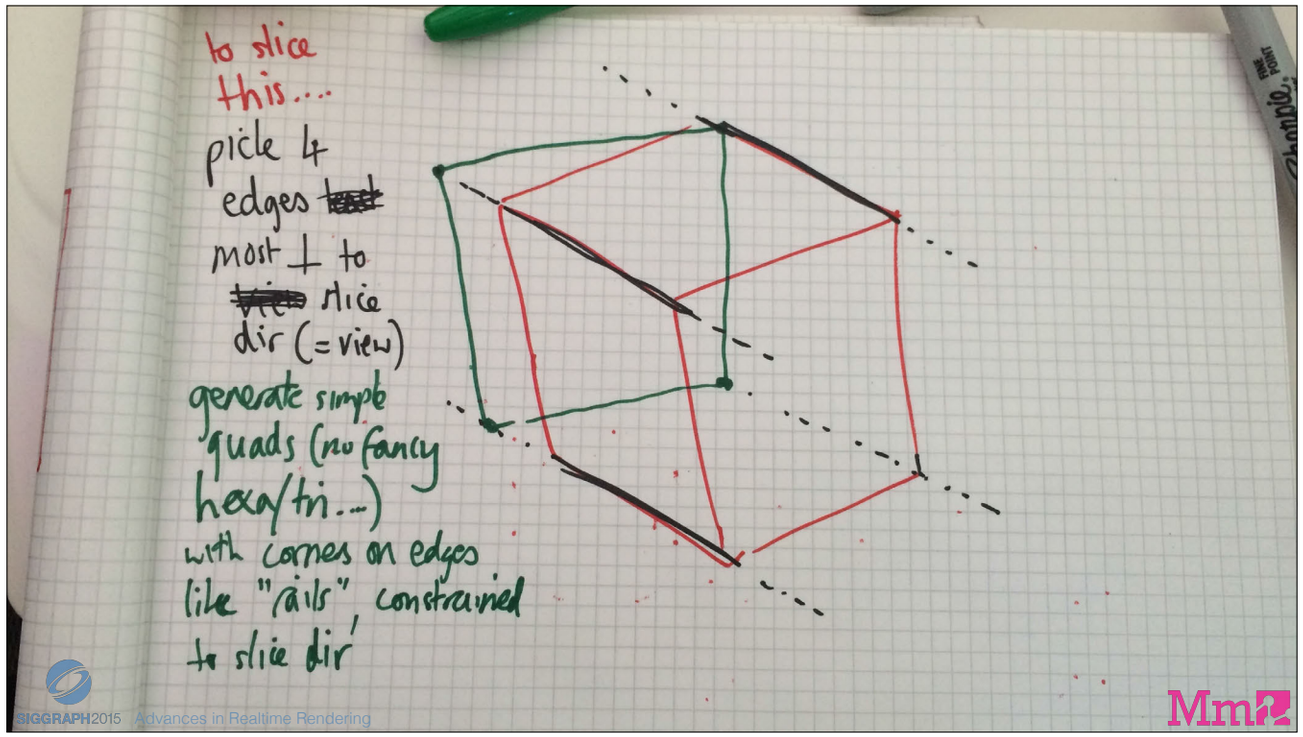
the rendering was simple too: you take each rigid object, slice it screen-aligned along exponentially spaced z slices, and composite front to back or back to front. it's a scatter-based, painters algorithm style volume renderer. they exploit the rasterizer to handle sparse scenes with overlapping objects. they also are pre-filtered and can handle transparent & volumetric effects. this is quite rare - unique? - among published techniques. it’s tantalising. I think a great looking game could be made using this technique.
{slice image}
I have a small contribution - they spend a lot of the paper talking about a complex Geometry shader to clip the slices to the relevant object bounds. I wish it was still 2008 so I could go back in time and tell them you don’t need it! ;) well, complex GS sucks. so even though I’m 7 years late I’m going to tell you anyway ;)



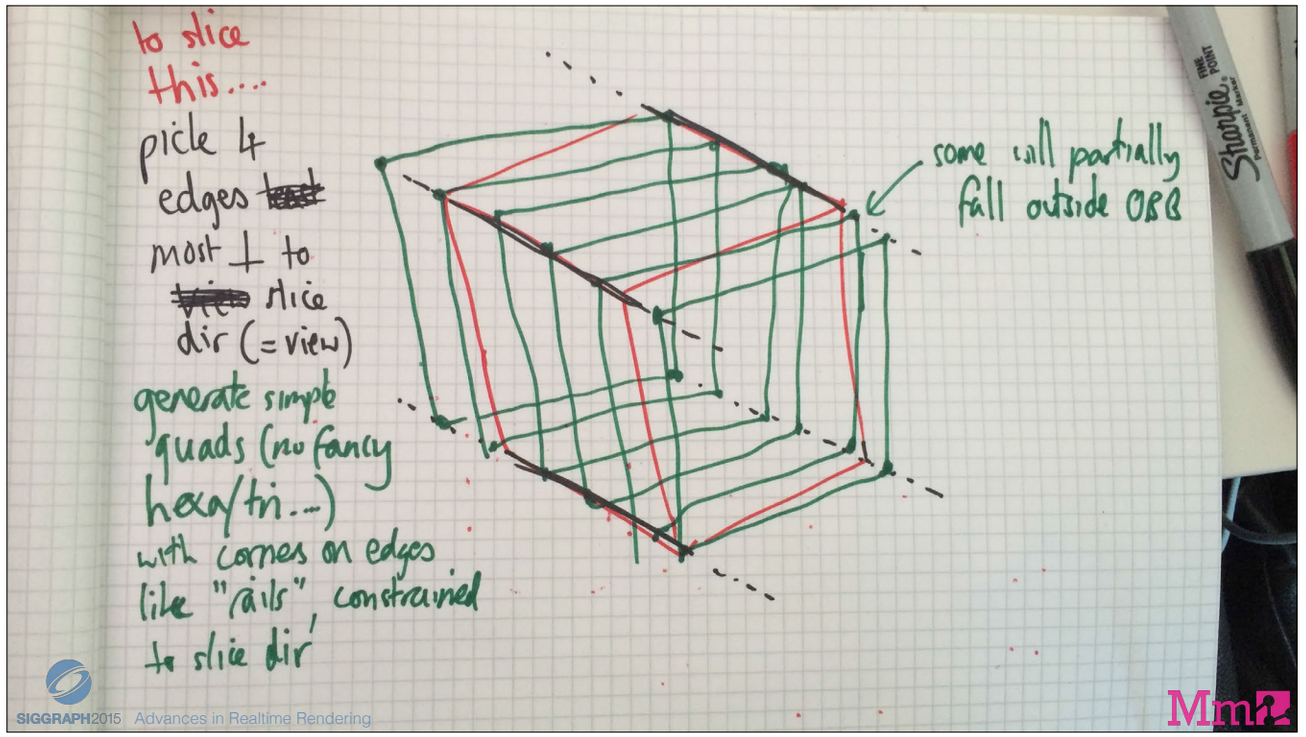
to slice an object bounded by this cube…



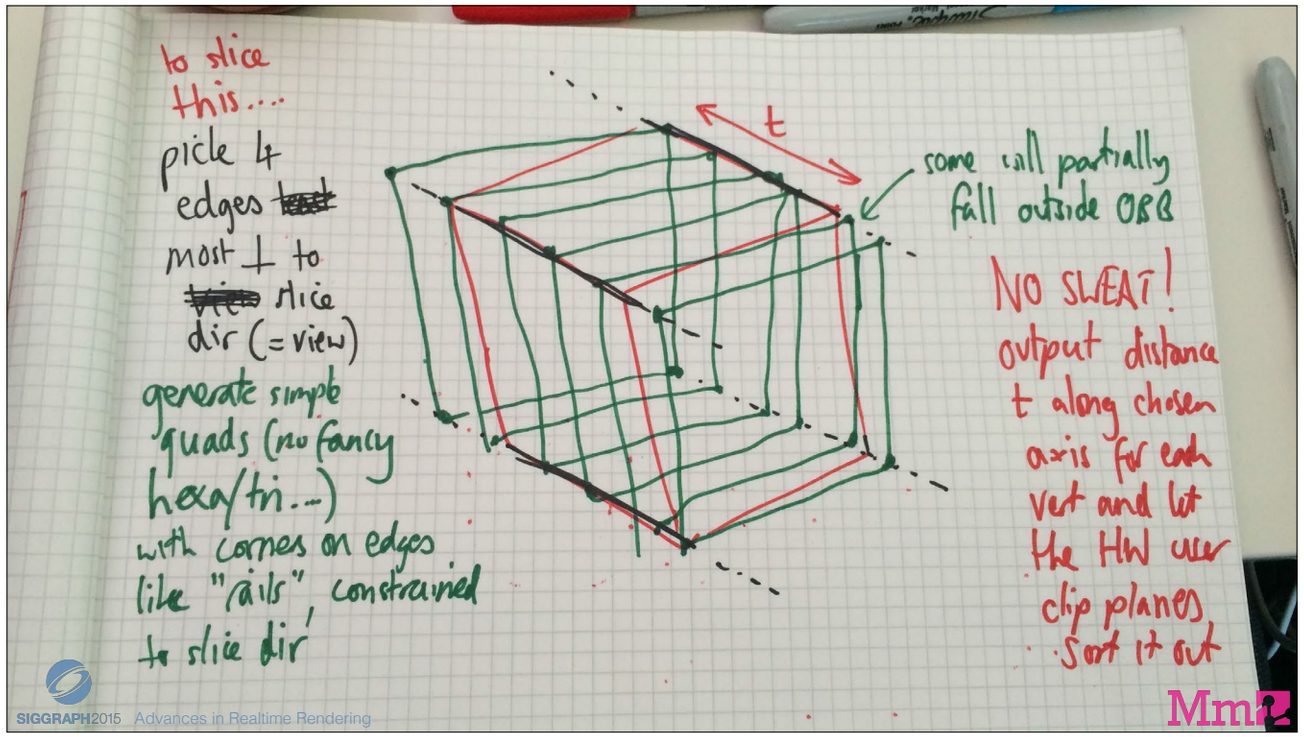
pick the object axis closest to the view direction, and consider the 4 edges of the cube along this axis.



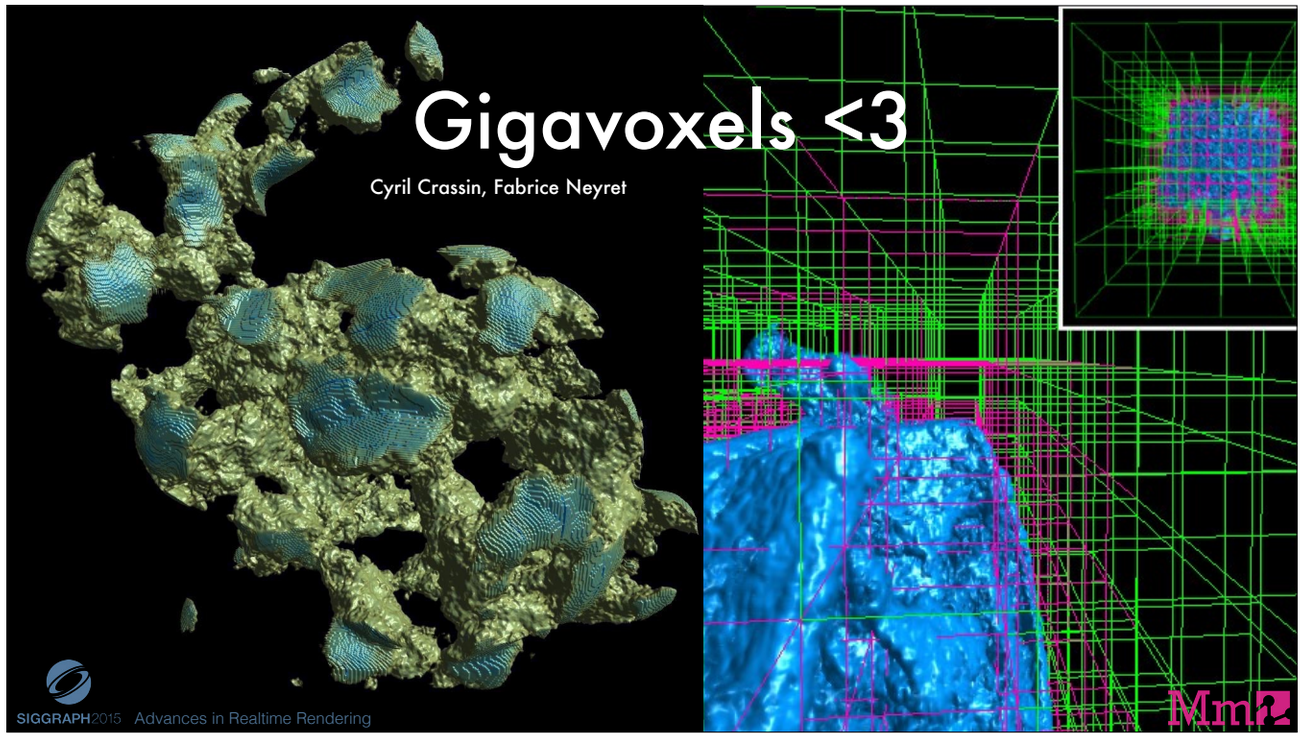
generate the slices as simple quads with the corners constrained to these 4 edges,



some parts of the slice quads will fall outside the box. that’s what the GS was there for! but with this setup, we can use existing HW:

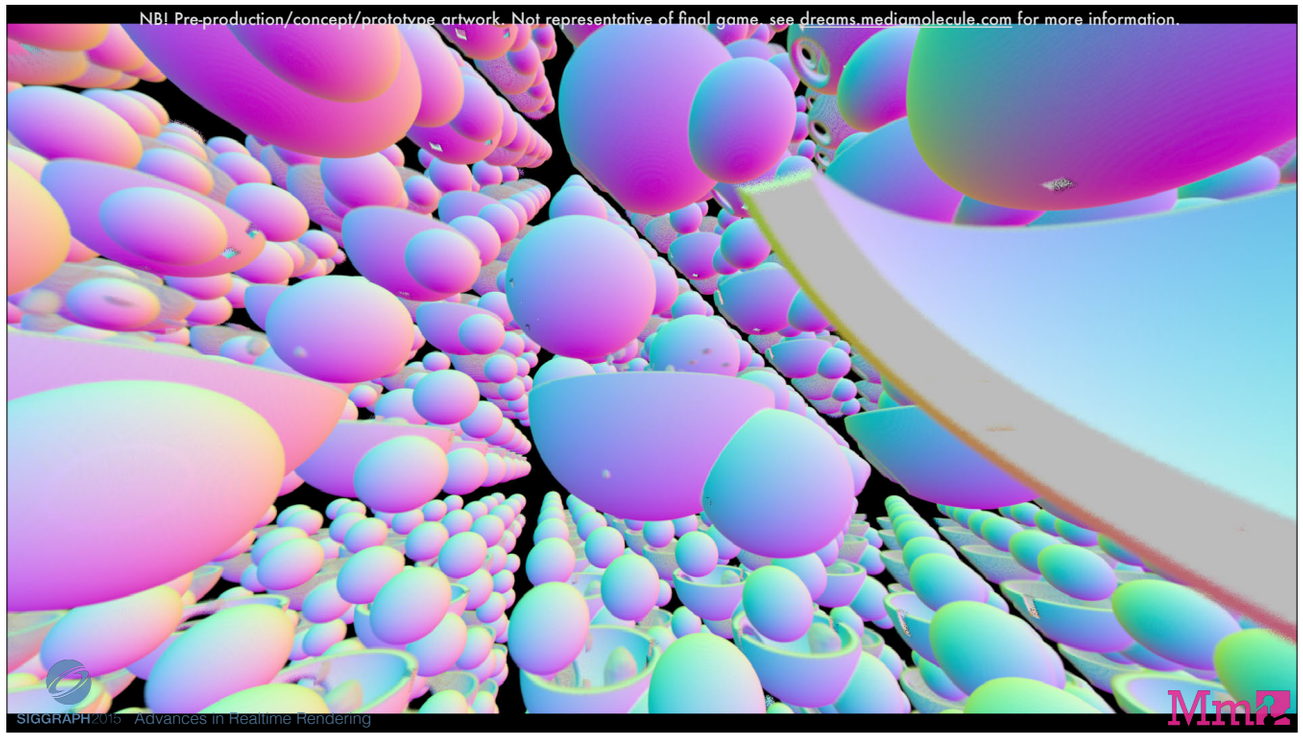


just enable two user clipping planes for the front and back of the object. the hardware clipping unit does all the hard work for you.

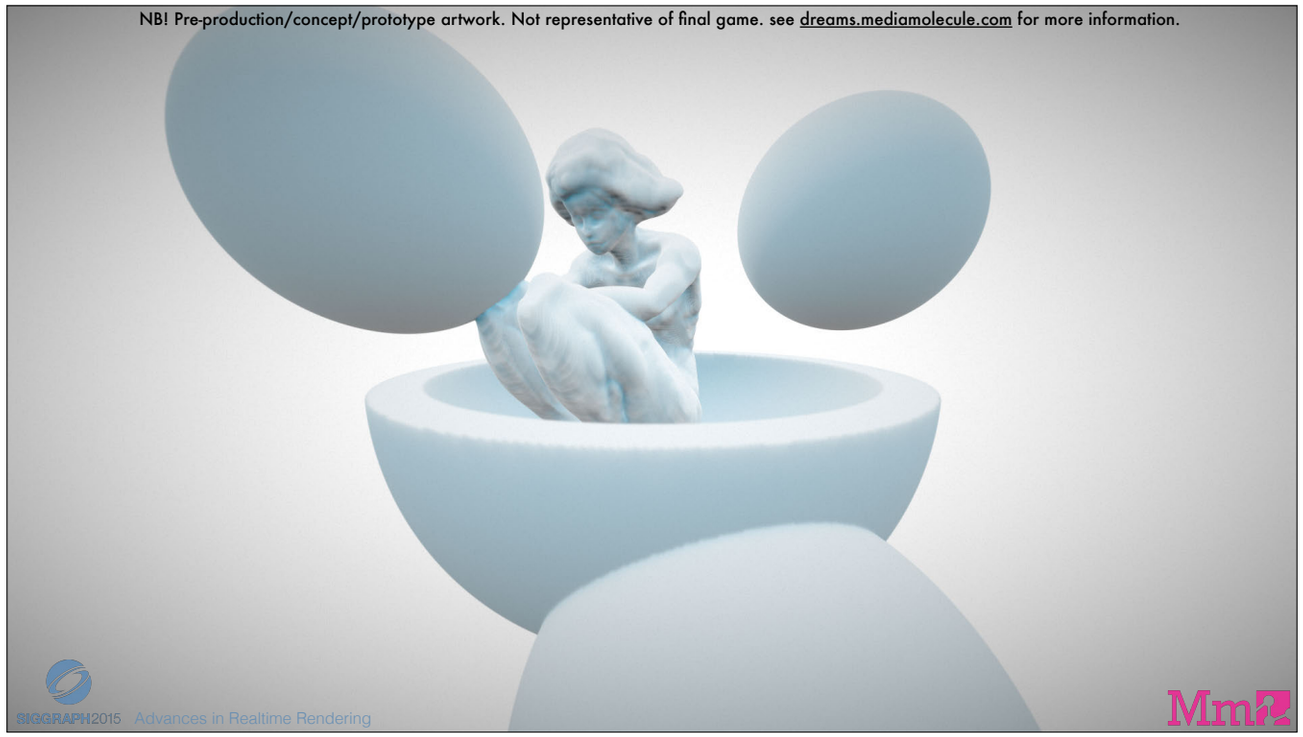


ANYWAY. this idea of volumetric billboards stuck with me. and I still love it.
fast forward a few years, and the french were once again rocking it.
http://maverick.inria.fr/Members/Cyril.Crassin/
Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre (note: neyret is the secondary author on VBs) had put out gigavoxels.

this is the next precursor to SVOs. seen through the lens of the earlier VB work, I loved that it kept that pre-filtered look, the geometric density from having a densely sampled field. it layered on top a heirachical, sparse representation - matching very well the structure of our evaluator. hooray!
however it dispensed with the large number of overlapping objects, which makes it less immediately applicable to Dreams/games. but…
I did implement a quick version of gigavoxels, here are some shots.



its impossible to resist domain repetition when you’re just raytracing a field…



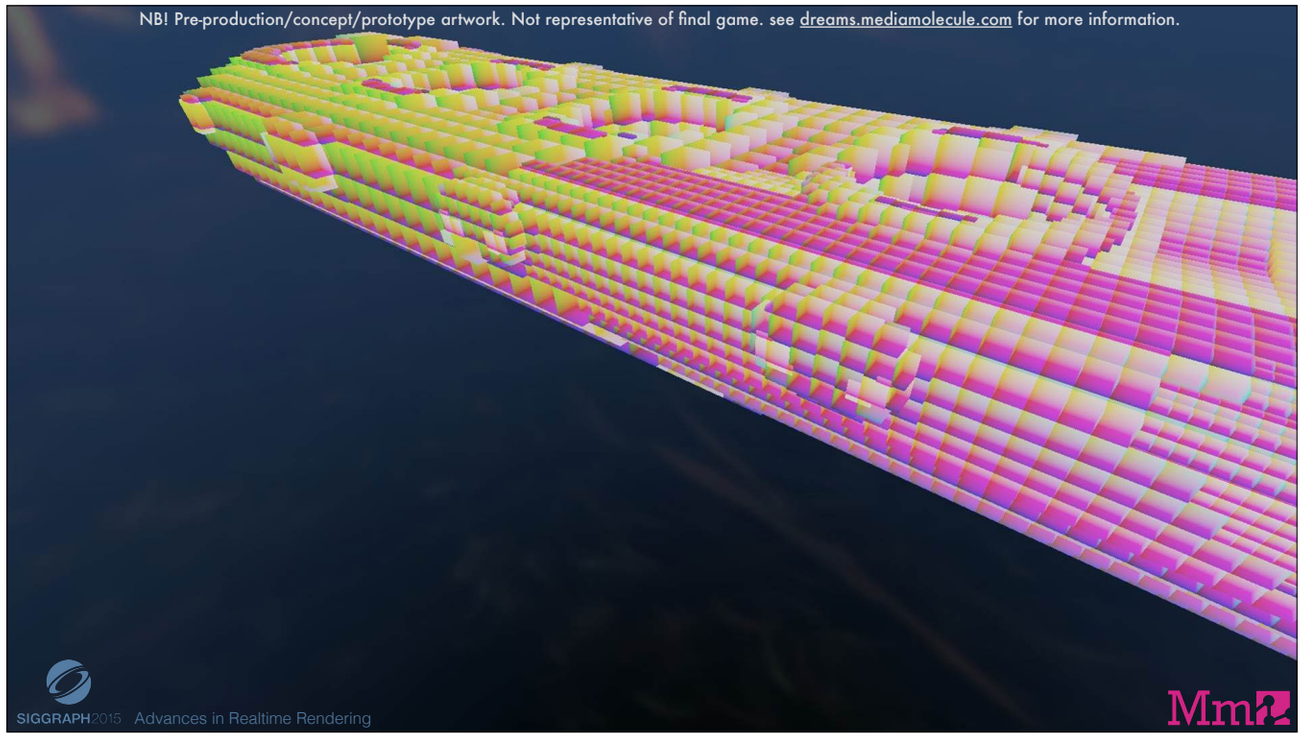
add some lighting as per my earlier siggraph advances talk (2006 was it?), the sort of thing that has since been massively refined e.g. in the shadertoy community (sampling mip mapped/blurred copies of the distance field - a natural operation in gigavoxel land, and effectively cone tracing)
I think it has a lovely alabaster look.

however it focussed on a single large field that (eye) rays were traced through, and I needed the kind of scene complexity of the earlier VB paper - a cloud of rigid voxels models.

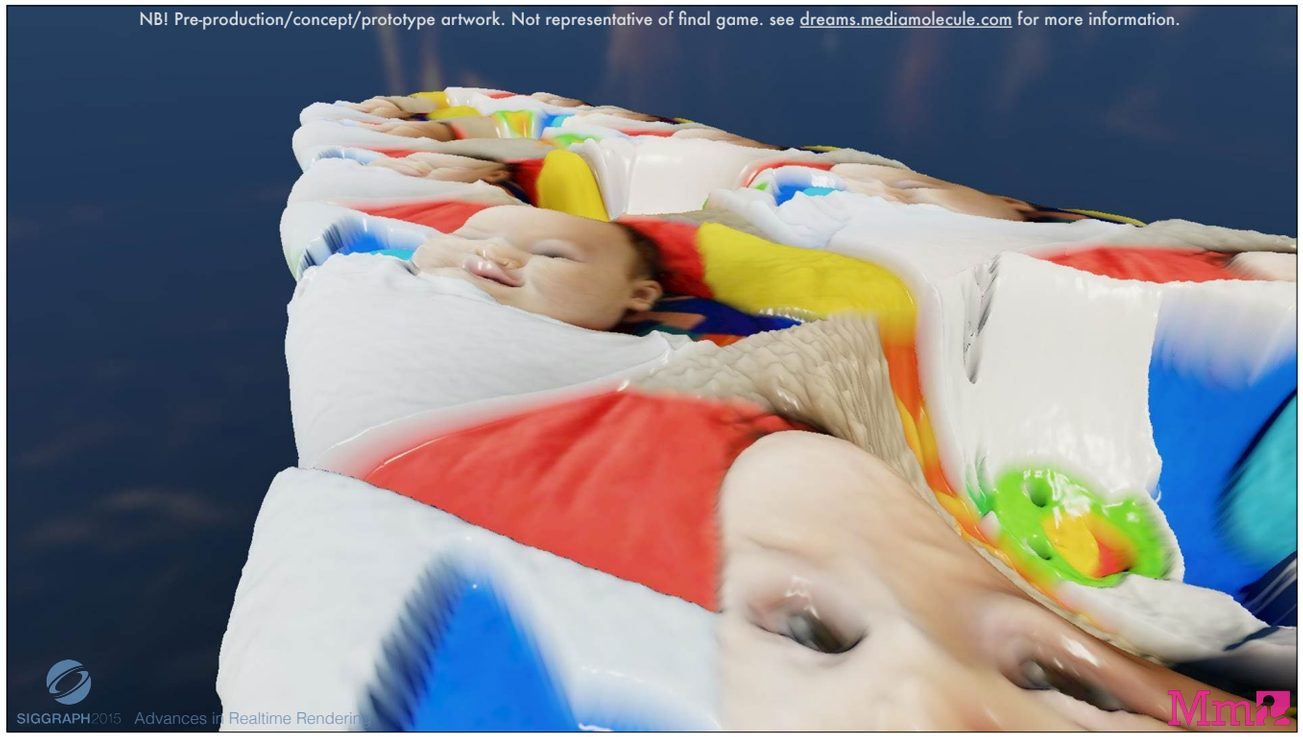
Engine 2: The brick engine

i.e. Let's try a hybrid Volumetric Billboards / Gigavoxels approach!

the idea is to take the brick tree from gigavoxels, but instead of marching rays from the eye, directly choose a 'cut' through the tree of bricks based on view distance (to get nice LOD), then rasterise each brick individually. The pixel shader then only has to trace rays from the edge of the bricks to any surface.
As an added advantage, the bricks are stored in an atlas, but there is no virtual-texturing style indirection needed in the inner loop (as it is in gigavoxels), because each rastered cube explicitly bounds each individual brick, so we know which bit of the atlas to fetch from at VS level.

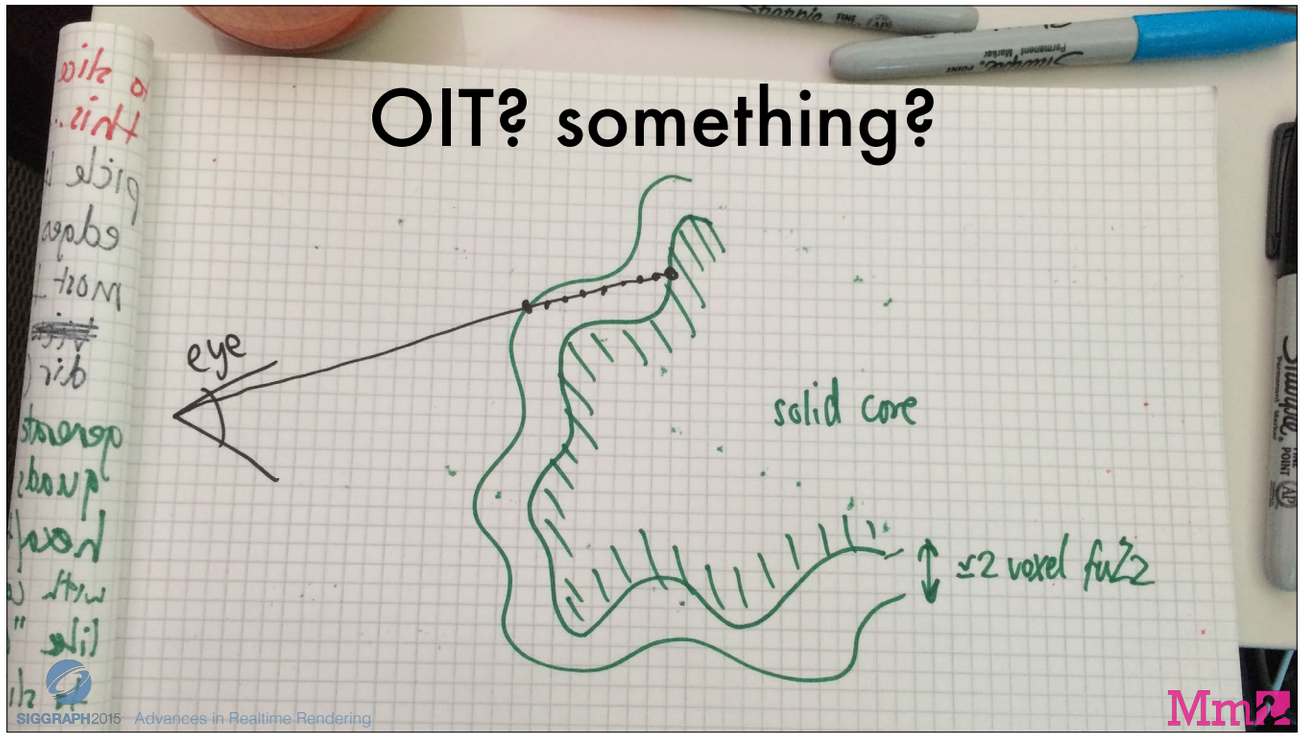


here you can see the individual cubes that the VS/PS is shading. each represents an 8x8x8 little block of volume data, giga-voxels style. again: rather than tracing eye rays for the whole screen, we do a hybrid scatter/gather: the rasteriser scatters pixels in roughly the right places (note also that the LOD has been adapted so that the cubes are of constant screen space size, ie lower LOD cut of the brick tree is chosen in the distance) then the Pixelshader walks from the surface of the cubes to the SDF surface.
also, I could move the vertices of the cubes around using traditional vertex skinning techniqes, to get animation and deformation... oh my god its going to be amazing!

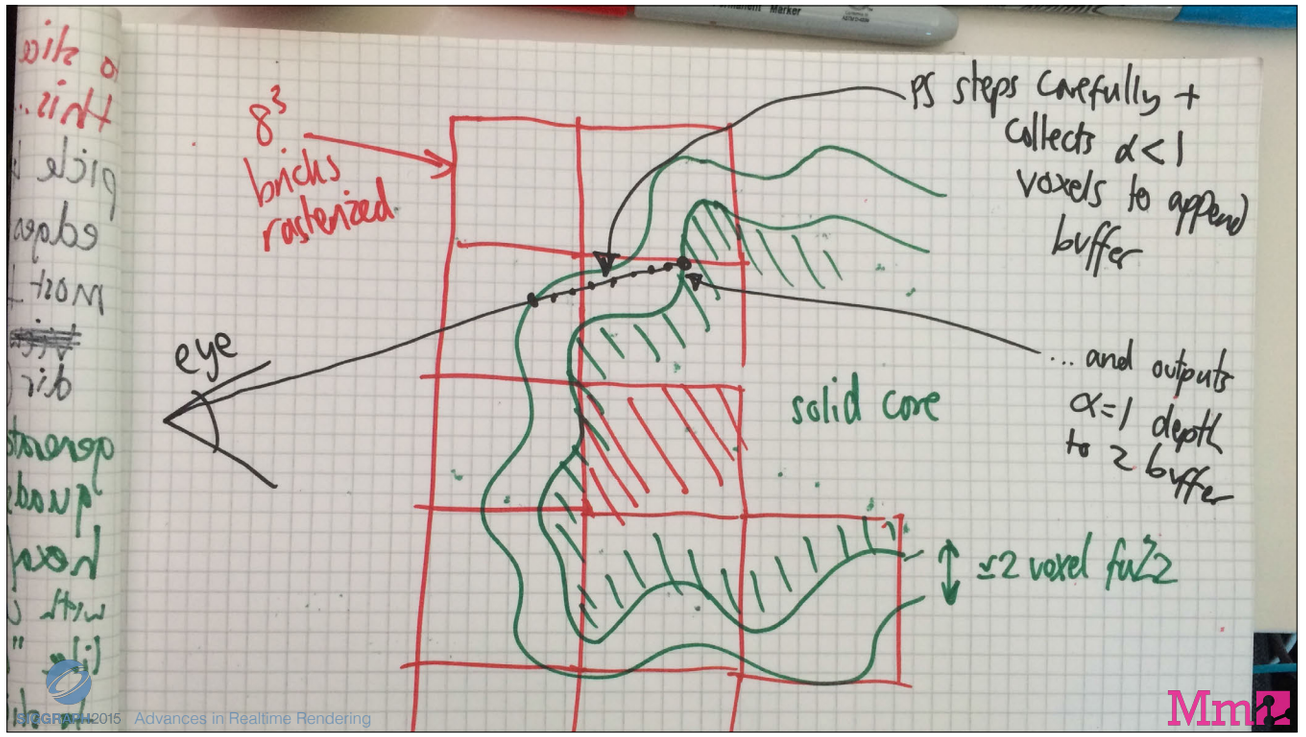


(sorry for the bad screenshot - I suck at archiving my work)
It sort of amounts to POM/tiny raymarch inside each 8x8x8 cube, to find the local surface. with odepth to set the zbuffer.
it has the virtue of being very simple to implement.

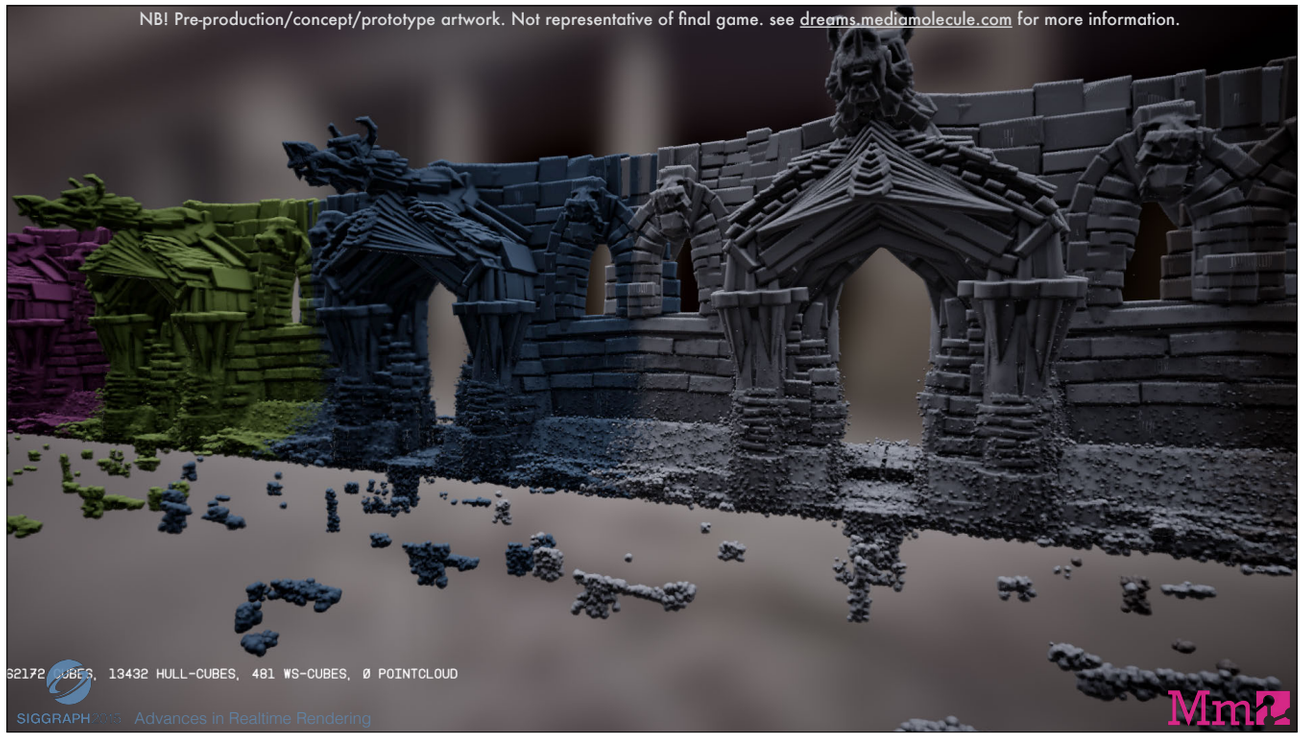
Because of that simplicity, This technique actually ended up being the main engine a lot of the artists used for a couple of years; you'll see a couple more shots later. so while the 'bricks' engine as it was known, went into heavy use, I really wanted more.



I wasn’t happy! why not? I also wanted to keep that pre-filtered look from Volumetric Billboards. I felt that if we pursued just hard z buffered surfaces, we might as well just do polys, or at least, the means didn’t lead to a visual result that was different enough. so I started a **long** journey into OIT.



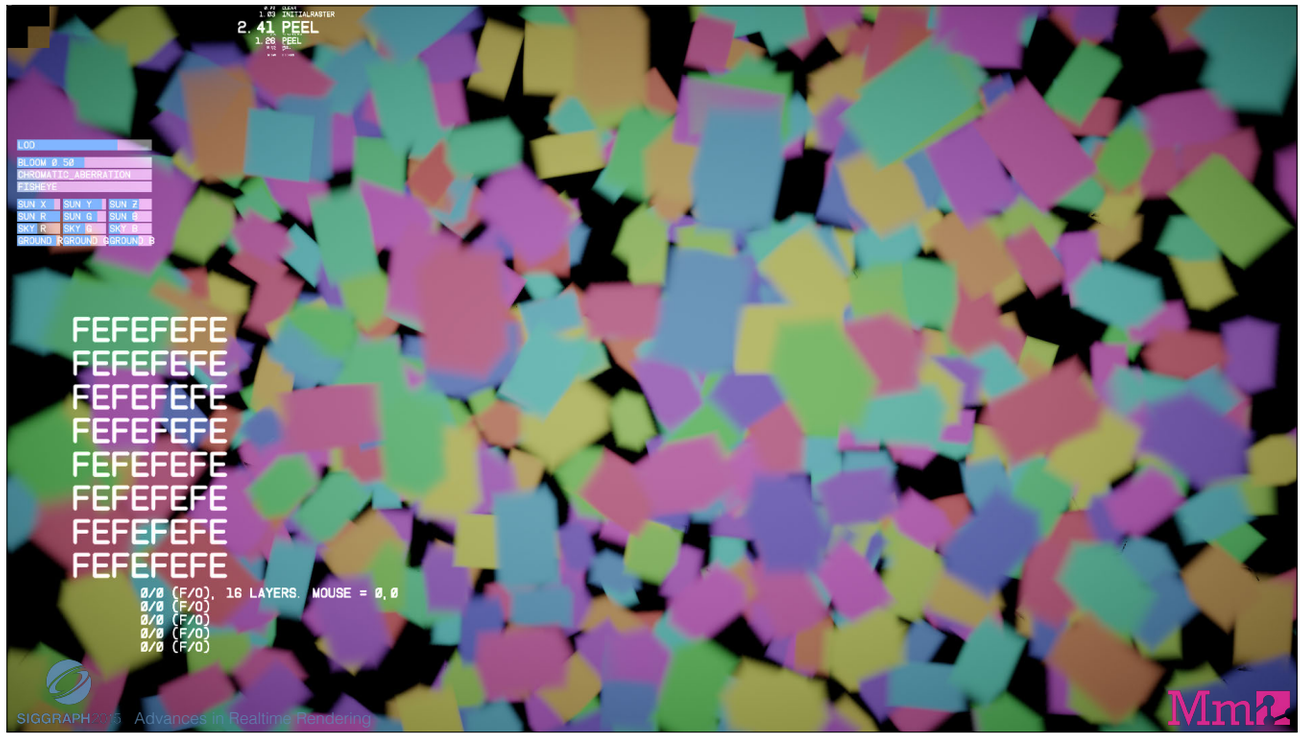
I immediately found that slicing every cube into 8-16 tiny slices, ie pure 'VB', was going to burn way too much fill rate.
so I tried a hybrid where: when the PS marched the 8x8x8 bricks, I had it output a list of fuzzy ‘partial alpha’ voxels, as well as outputting z when it hit full opacity. then all I had to do was composite the gigantic (10s of millions) of accumulated fuzzy samples onto the screen… in depth sorted order. hmm



so it was 'just' a matter of figuring out how to composite all the non-solid voxels. I had various ground truth images, and I was particularly excited about objects overlapping each other with really creamy falloff - e.g. between the blue arch and the grey arch - thats just the two overlapping and the ‘fuzz’ around them smoothly cross-intersecting.

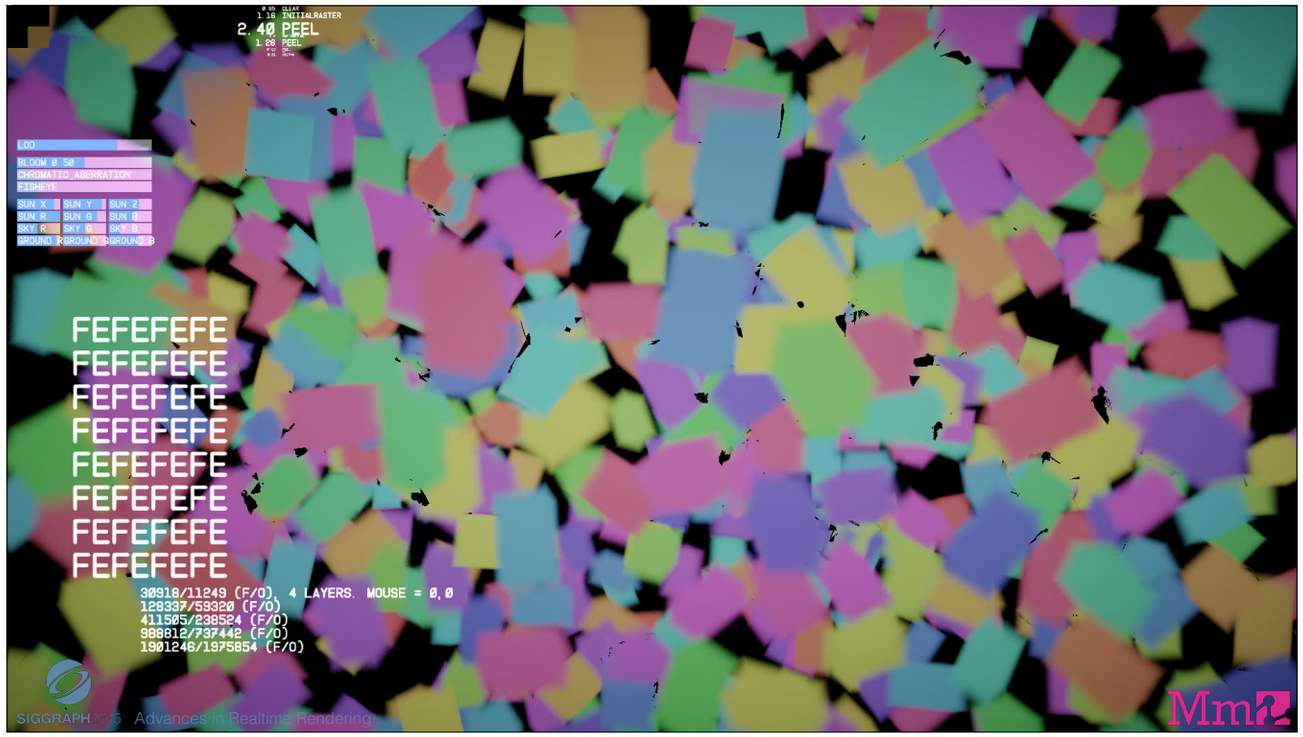
and pre filtering is great for good LOD! this visualizes the pre-filtered mips of dad’s head, where I’ve added a random beard to him as actual geometry in the SDF.

and here’s what it looks like rendered.



but getting from the too-slow ground truth to something consistently fast-enough was very, very hard. prefiltering is beautiful, but it generates a lot of fuzz, everywhere.

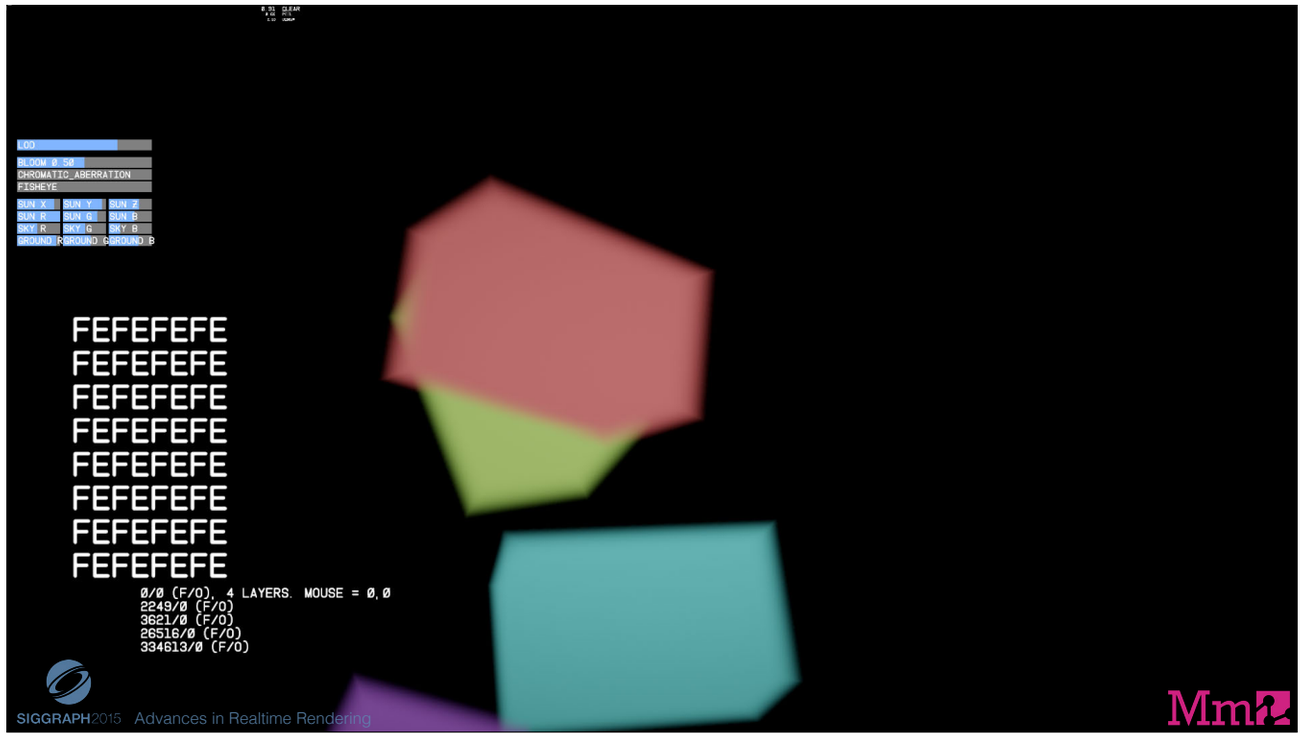
the sheer number of non-opaque pixels was getting high - easily 32x 1080p
I spent over a year trying everything - per pixel atomic bubble sort, front k approximations, depth peeling.. one thing I didn’t try because I didn’t think of it and it hadn't been published yet, was McGuire style approximate commutative OIT. however it wont work in its vanilla form - it turns out the particular case of a very 'tight' fuzz around objects is very unforgiving of artefacts - for example, if adjacent pixels in space or time made different approximations (eg discarded or merged different layers), you get really objectionable visible artefacts.



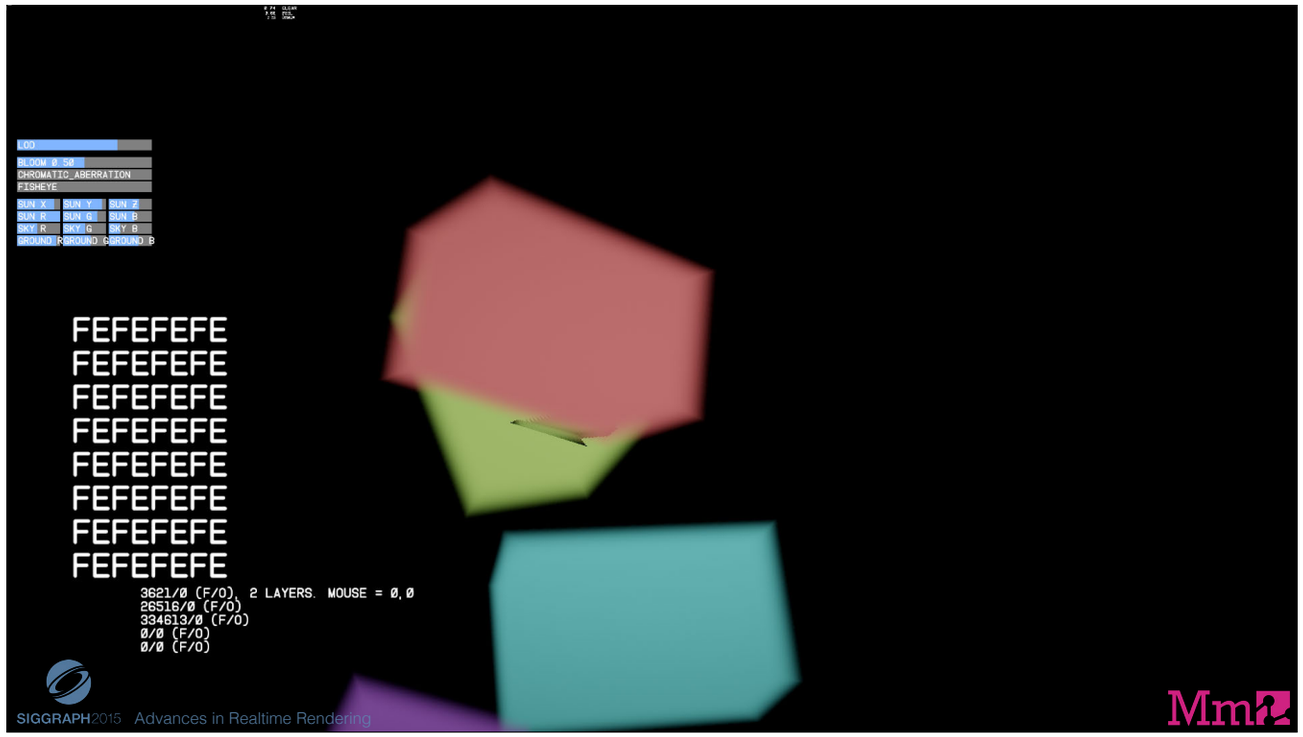
it's even worse because the depth complexity changes drastically over 2 orders of magnitude between pixels that hit a hard back and 'edge on' pixels that spend literally hundred of voxels skating through fuzz. this is morally the same problem that a lot of sphere tracing approaches have, where edge pixels are waaaay harder than surface pixels. (cf shadertoy). I did have some interesting CS load balancing experiments, based on wavefronts peeling off 8 layers at a time, and re-circulating pixels for extra passes that needed it -

{peeling images}

a kind of compute shader depth peel but with load balancing its goal.



here’s a simpler case. fine when your sort/merge algo has enough layers. but if we limit it to fewer blended voxels than necessary…



I couldn’t avoid ugly artefacts.

in the end, the 'hard' no-fuzz/no-oit shader was what went over the fence to the designers, who proceeded to work with dreams with a 'hard' look while I flailed in OIT land.

ARGH 2

Polygons? too difficult
Volumetric Billboards? too much filtrate
Gigavoxels? single object
Hybrid with rasterised gigavoxel cubes? too hard to render OIT overlaps

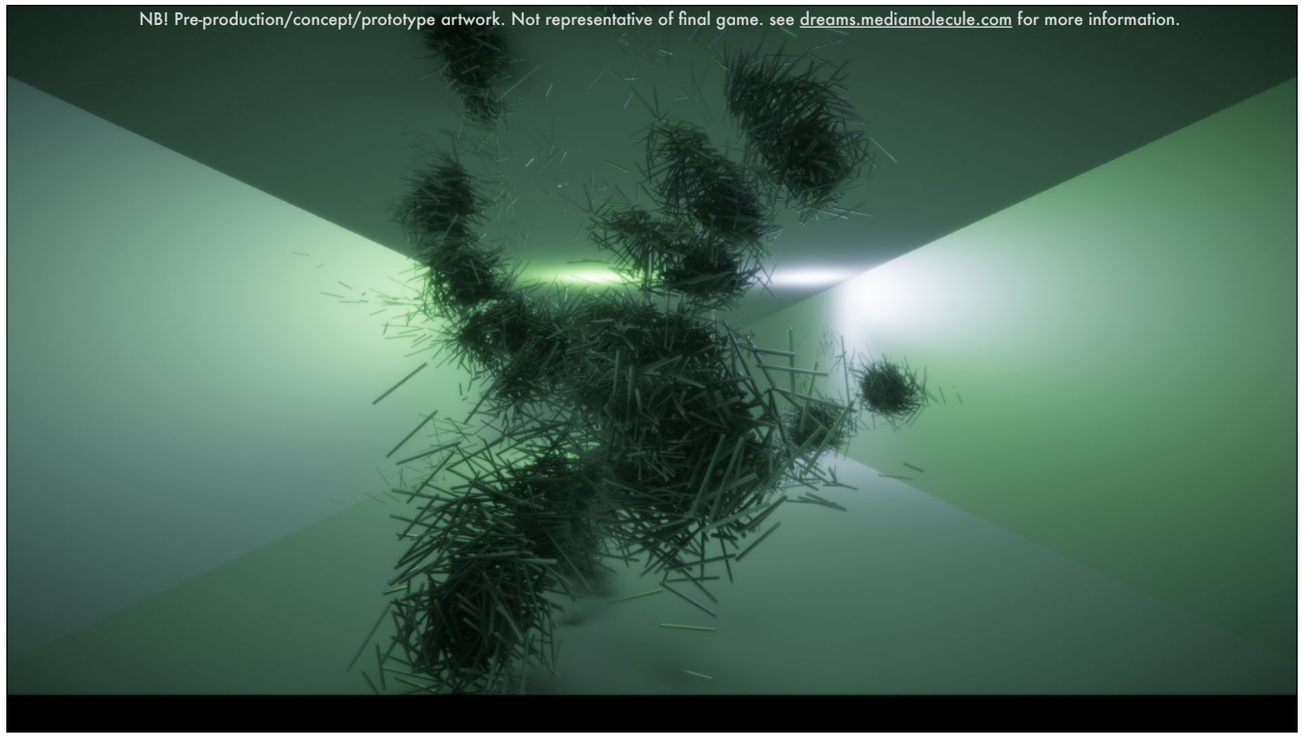
All is not well :(

see what I mean about failure?
and this is over the period of about 2 years, at this point

Engine attempt 3: Refinement renderer

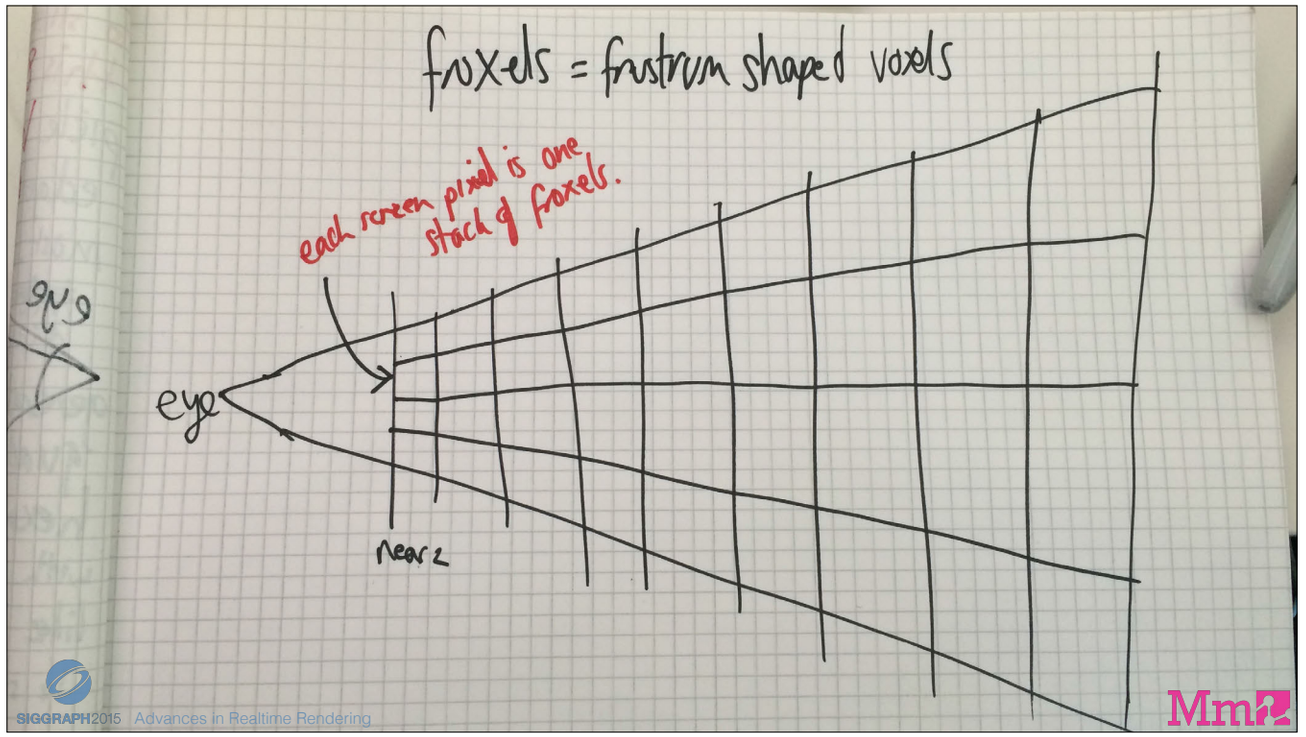
aka ‘Let’s try again’

I think this is a really cool technique, its another one we discarded but I think it has some legs for some other project.
I call it the refinement renderer.



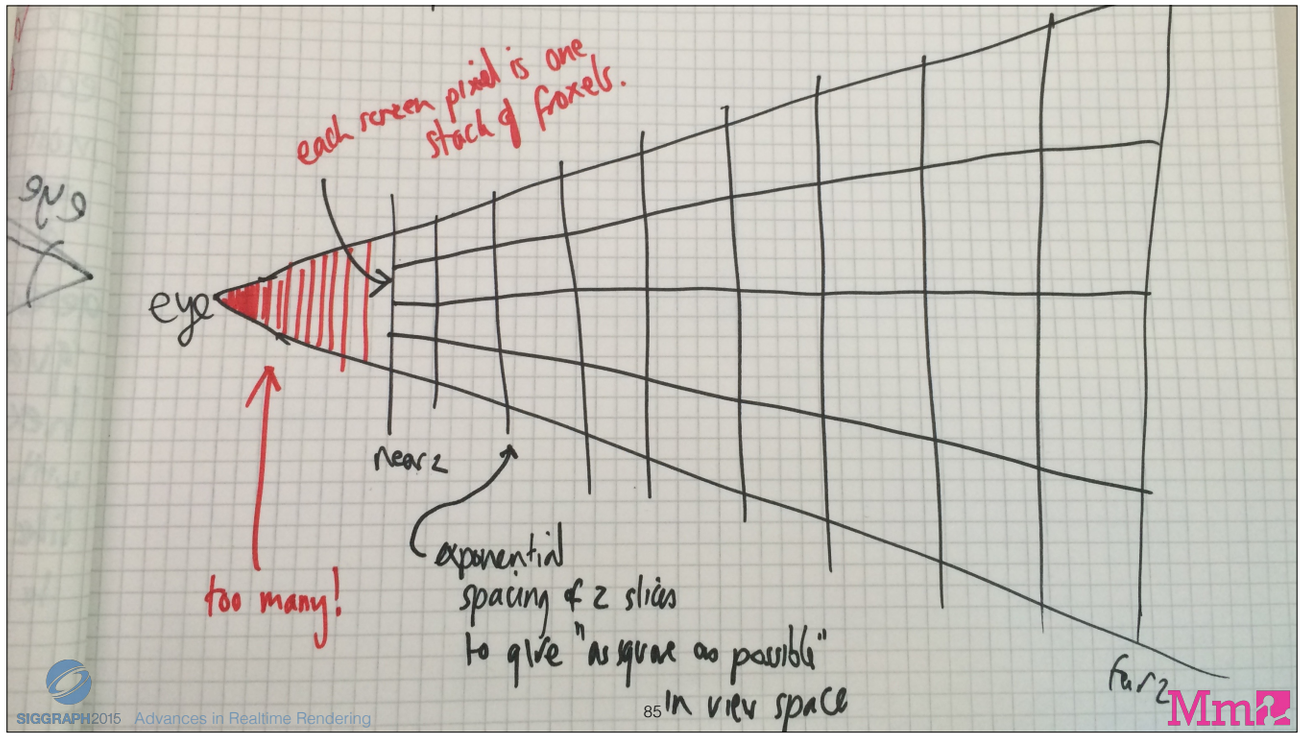
there are very few screenshots of this as it didn't live long, but its interestingly odd. have this sort of image in your mind for the next few slides. note the lovely pre-filtered AA, the soft direct lighting (shadows but no shadow maps!). but this one is pure compute, no rasterised mini cubes.

the idea is to go back to the gigavoxels approach of tracing eye rays through fuzz directly... but find a way to make it work for scenes made out of a large number of independently moving objects. I think if you squint a bit this technique shares some elements in common with what Daniel Wright is going to present in the context of shadows; however since this focuses on primary-ray rendering, I'm not going to steal any of his thunder! phew.



a bit of terminology - we call post projection voxels - that is, little pieces of view frustum- 'froxels' as opposed to square voxels. The term originated at the sony WWS ATG group, I believe.
if you look at a ray marcher like many of the ones on shadertoy , like iq's famous cloud renderer, you can think of the ray steps as stepping through 'froxels'.

https://www.shadertoy.com/view/XslGRr - clouds by iq
typically you want to step the ray exponentially so that you spend less time sampling in the distance.
intuitively you want to have 'as square as possible' voxels, that is, your step size should be proportional to the inverse of the projected side length, which is 1/1/z, or z. so you can integrate and you get slices spaced as t=exp(A*i) for some constant A (slice index i), or alternatively write it iteratively as t+=K*t at each step for some constant K.

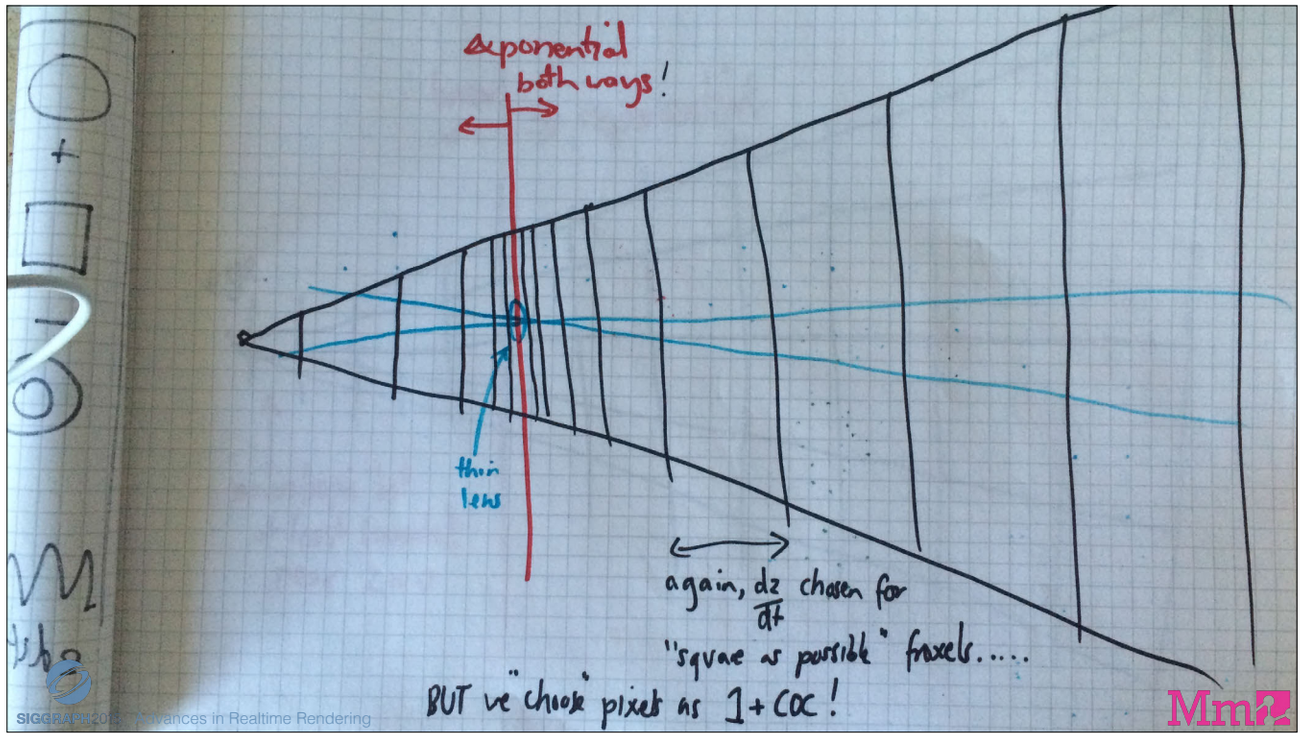


the only problem with this is that near the eye, as t goes to 0, you get infinitely small froxel slices. oh dear. if you look at iq's cloud example, you see this line:

```
t += max(0.1,0.02*t);
```

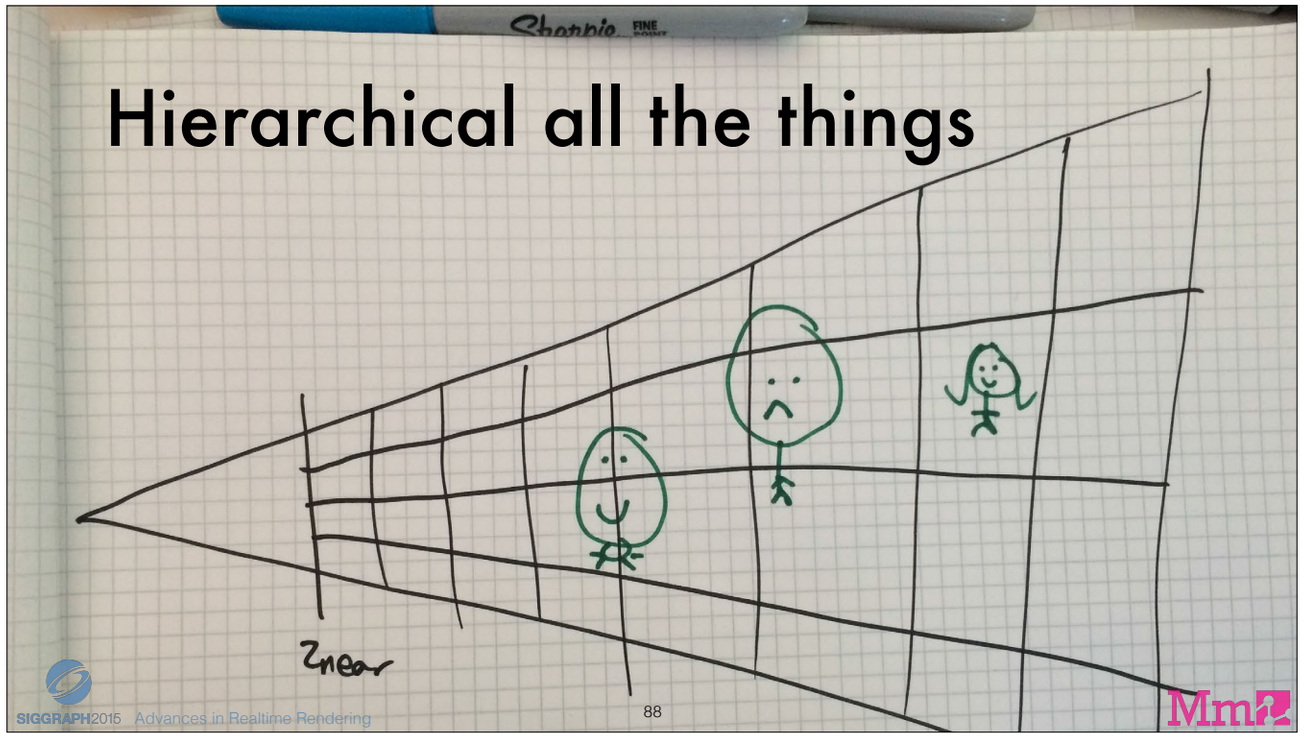
(aka, linear steps near the camera,
exponential steps further away)

t += max(0.1,0.02*t);
which is basically saying, let's have even slicing up close then switch to exponential after a while.
I've seen this empirically used a few times. here's an interesting (?) insight. what would real life do? they dont have pinhole cameras.

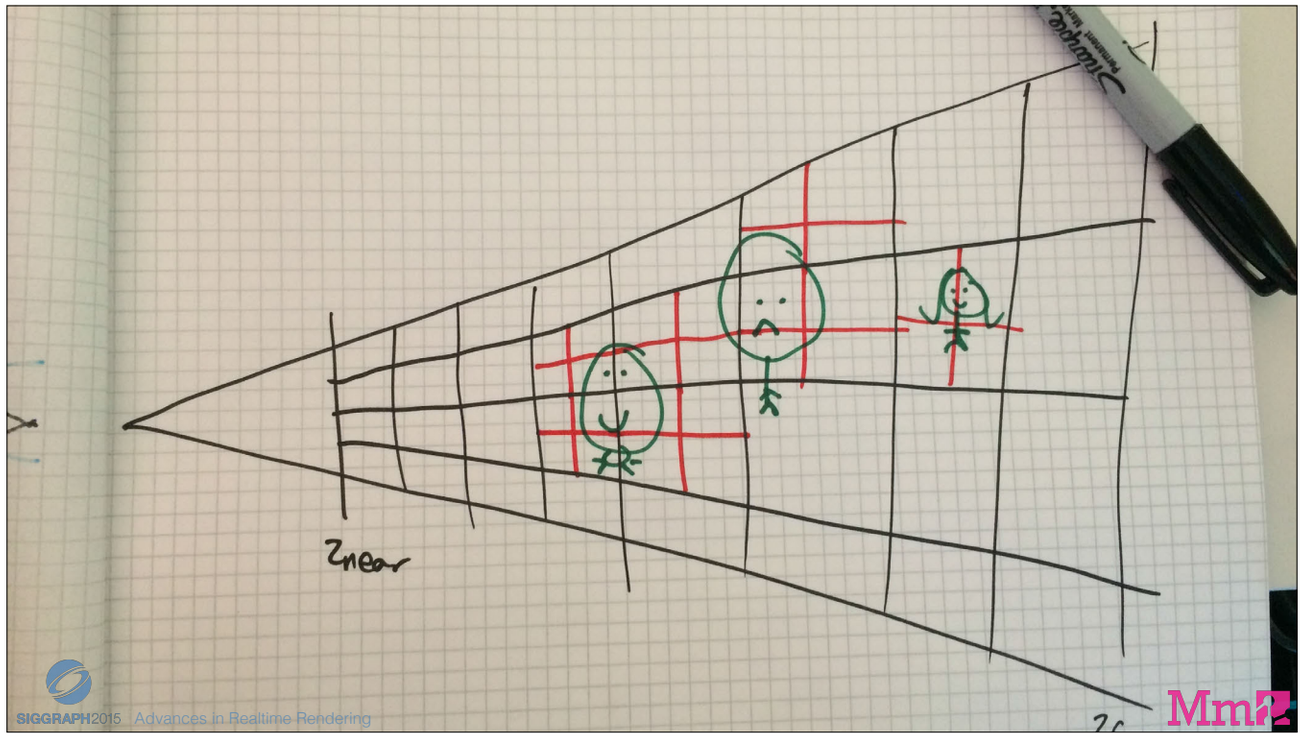


so, consider a thin lens DOF model for a second. what if you tuned your froxel sampling rate not just for projected pixel size, but for projected bokeh size. the projected bokeh radius is proportional to (z-f)/z, so we want A(z-f)/z + 1/z where A is the size in pixels of your bokeh at infinity. (the +1/z is the size of single ‘sharp’ pixel, i.e. the footprint of your AA filter)

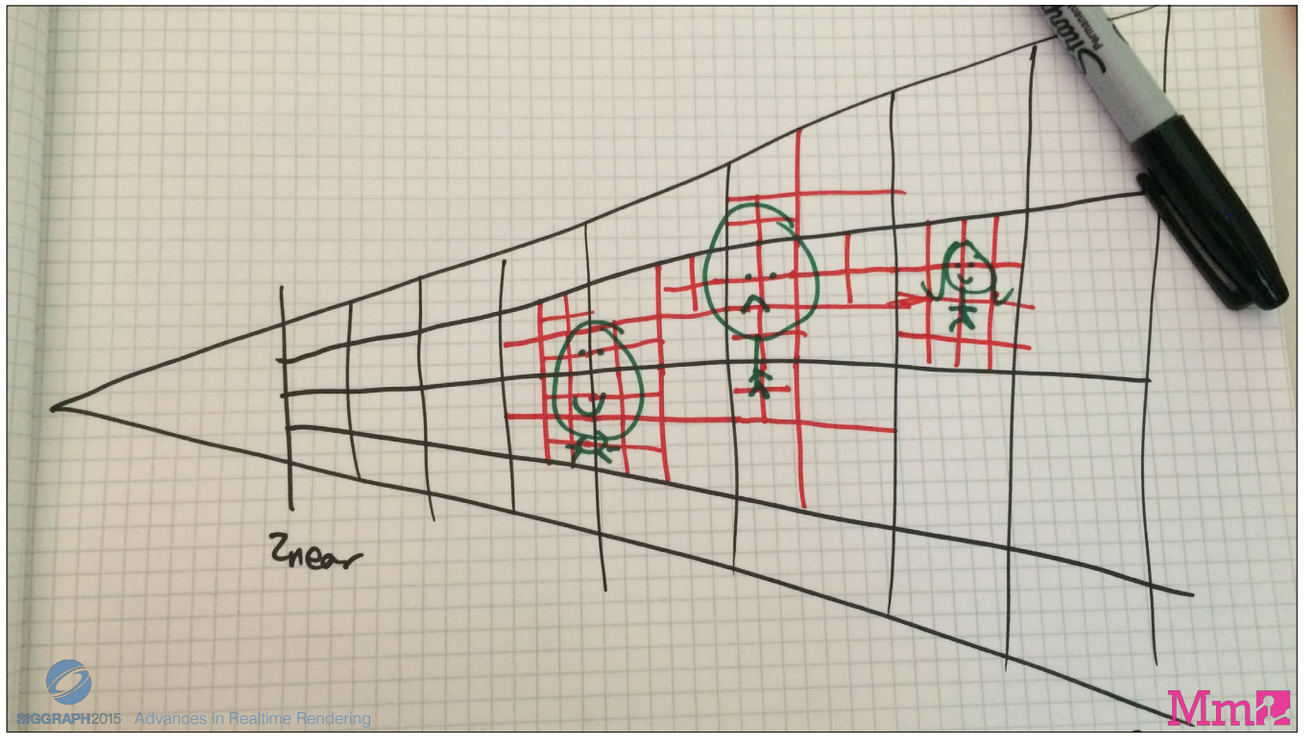
if you put this together, you can actually compute two exponential slicing rates - one for in front of the focal plane, and one for behind. at the focal plane, it's the same step rate you would have used before, but in the distance it's a little sparser, and near to the camera it's WAY faster. extra amusingly, if you work through the maths, if you set A to be 1 pixel, then the constant in the 'foreground' exponential goes to 0 and it turns out that linear slicing is exactly what you want. so the empirical 'even step size' that iq uses, is exactly justified if you had a thin lens camera model with aperture such that bokeh-at-infinity is 1pixel across on top of your AA. neat! for a wider aperture, you can step faster than linear.



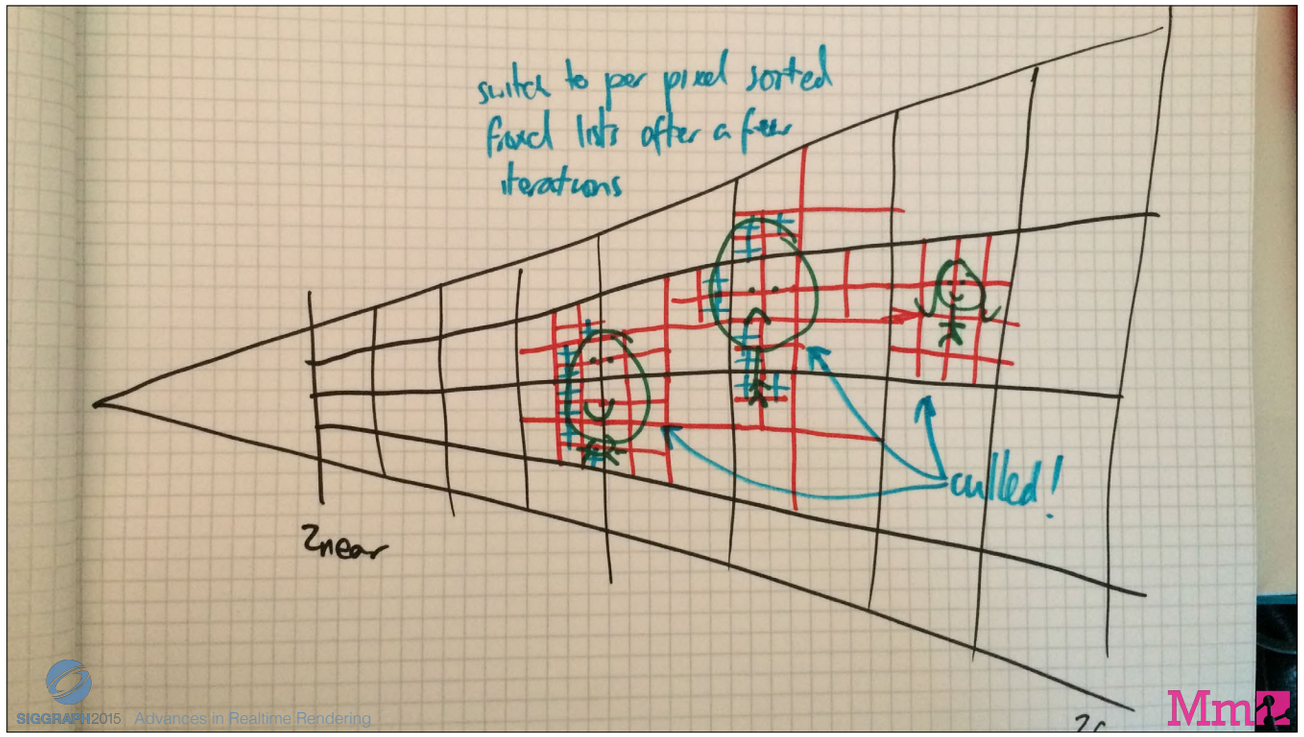
ANYWAY.
how does this relate to rendering lots of objects?
the idea I had was to borrow from the way the evaluator works. you start by dividing your frustum into coarse froxels. I chose 64th res, that is about 32x16 in x and y, with 32-64 in z depending on the far z and the DOF aperture. (blurrier dof = fewer slices needed, as in previous slides).
then you do a simple frustum vs object intersection test, and build a list per froxel of which objects touch it.
{pic}



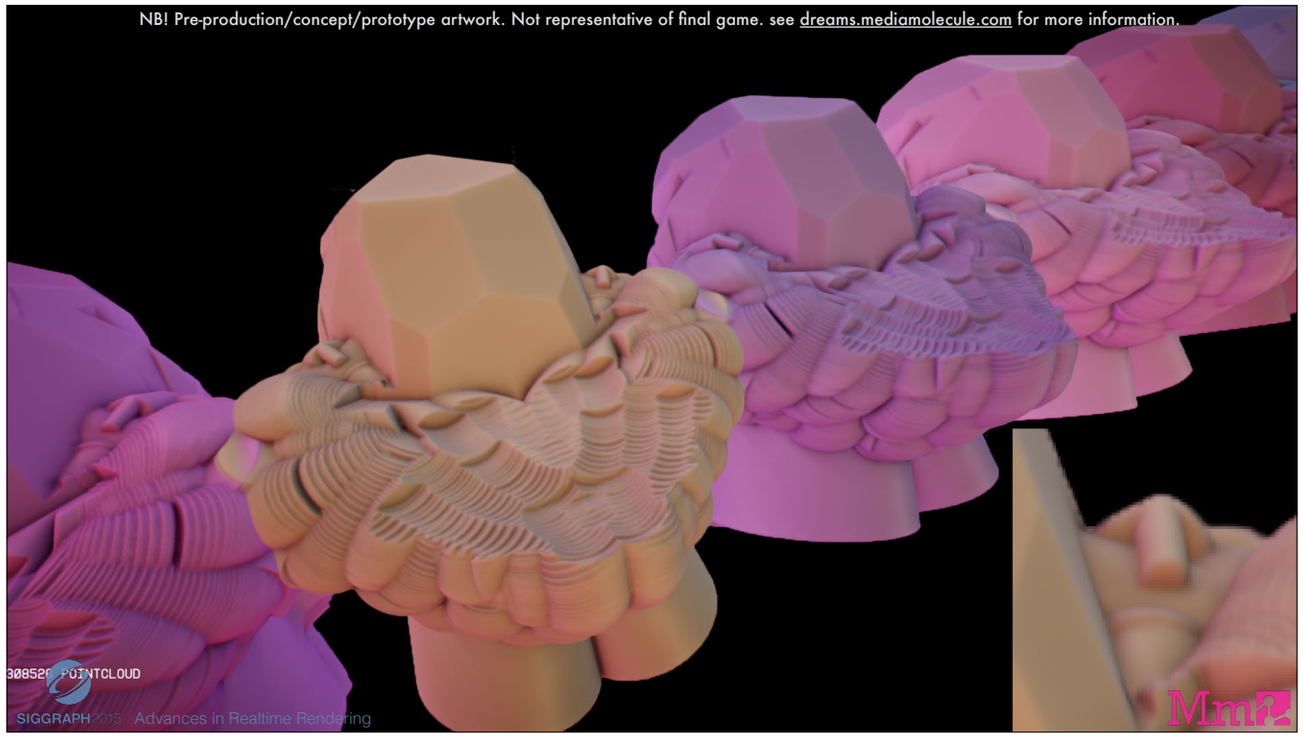
then, you recursively subdivide your froxels! for each froxel, in a compute shader you split them into 8 children. as soon as your froxel size matches the size of gigavoxel prefiltered voxels, you sample the sparse octree of the object (instead of just using OBBs) to futher cull your lists.



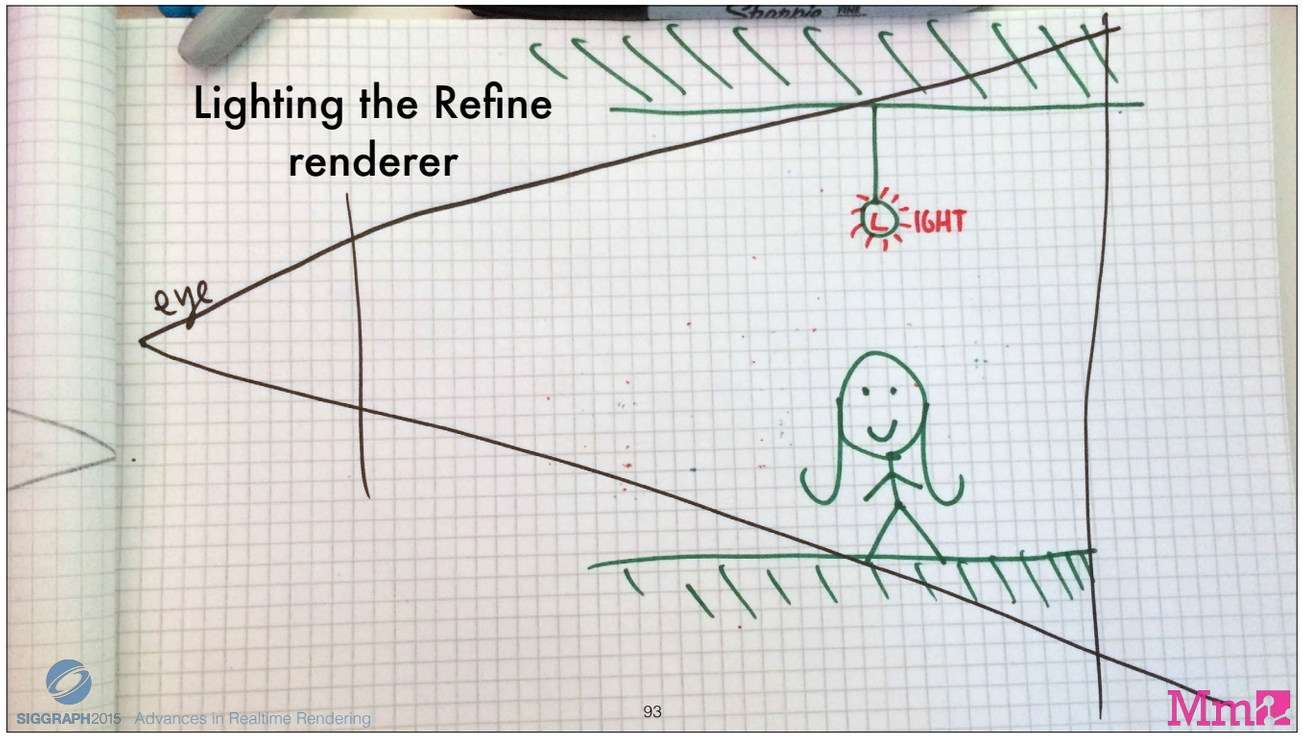
as you get finer and finer, the lists get shorter as the object's shape is more accurately represented. it’s exactly like the evaluator, except this time we have whole objects stored as gigavoxel trees of bricks (instead of platonic SDF elements in the evaluator), we don't support soft blend, and our domain is over froxels, not voxels.



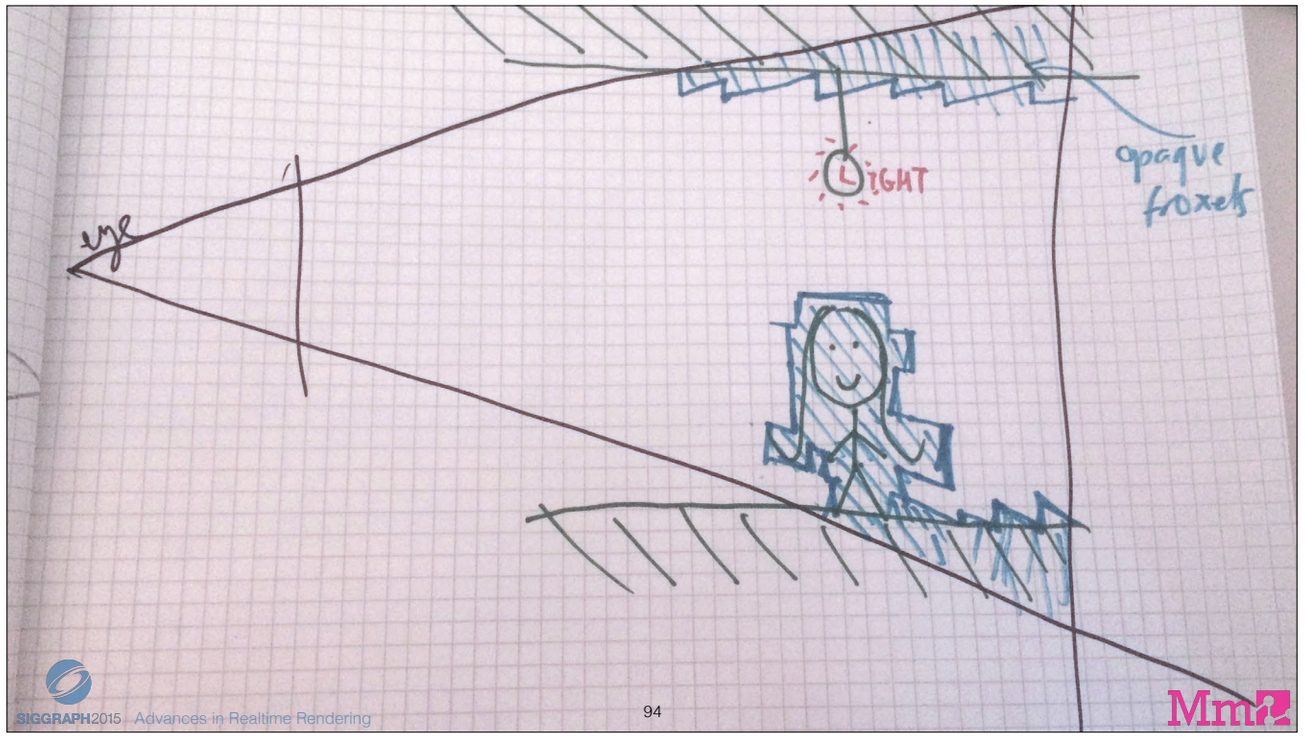
for the first few steps, I split every froxel in parallel using dense 3d volume textures to store pointers into flat tables of per froxel lists.
however at the step that refines from 1/16th res to 8th res (128x64x128 -> 256x128x256) the dense pointer roots get too expensive so I switch to a 2d representation, where every pixel has a single list of objects, sorted by z. the nice thing is that everything is already sorted coming out of the dense version, so this is really just gluing together a bunch of small lists into one long list per screen pixel.
each refine step is still conceptually splitting froxels into 8, but each pixel is processed by one thread, serially, front to back.
that also means you can truncate the list when you get to solid - perfect, hierarchical occlusion culling!.
SHOW ME THE PICTURES! OK
the results were pretty



and the pre-filtered look is really special.
Look how yummy the overlap of the meshes is! Really soft, and there's no 'post' AA there. It's all prefiltered.
so I did a bit of work on lighting; a kind of 3d extension of my siggraph 2006 advances talk.

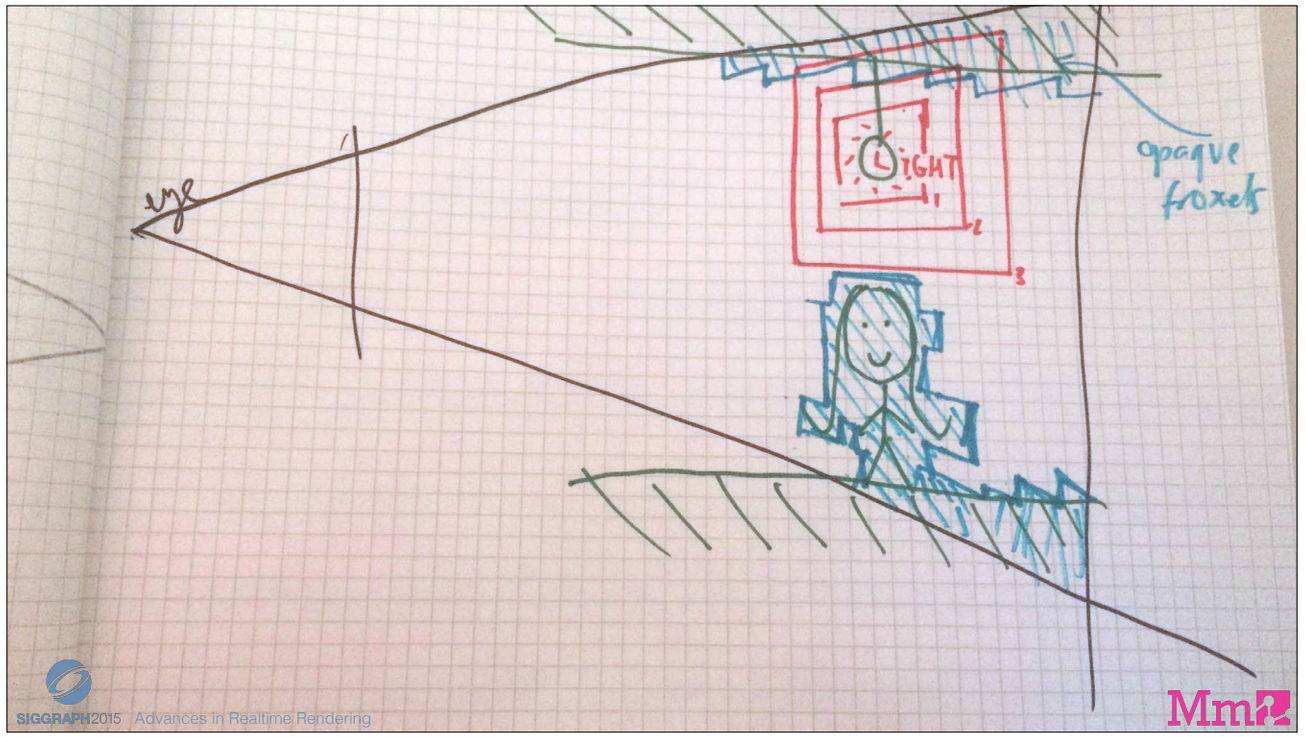


imagine this setup. this is basically going to be like LPV with a voxelized scene, except we use froxels instead of voxels, and we propagate one light at a time in such a way that we can smear light from one side of the frustum to another in a single frame, with nice quality soft shadows. ‘LPV for direct lights, with good shadows’, if you will.

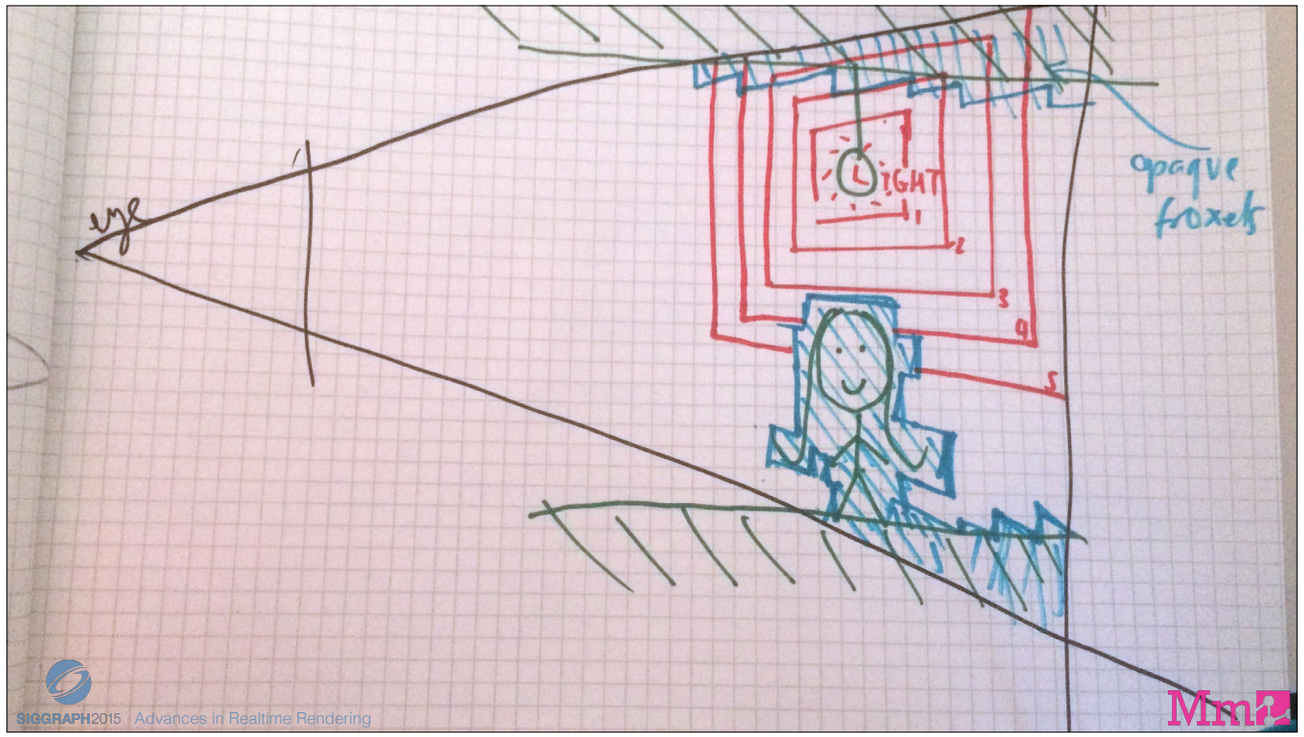


imagine a single channel dense froxel grid at low resolution, I think I used 256x128x256 with 8 bits per froxel. We will have one of those for the ‘density’ of the scene - defined everywhere inside the camera frustum. - As a side effect of the refinement process I write that ‘density’ volume out, more or less for free. Now we are also going to have one extra volume texture for each ‘hero’ light. (I did tests with 4 lights).

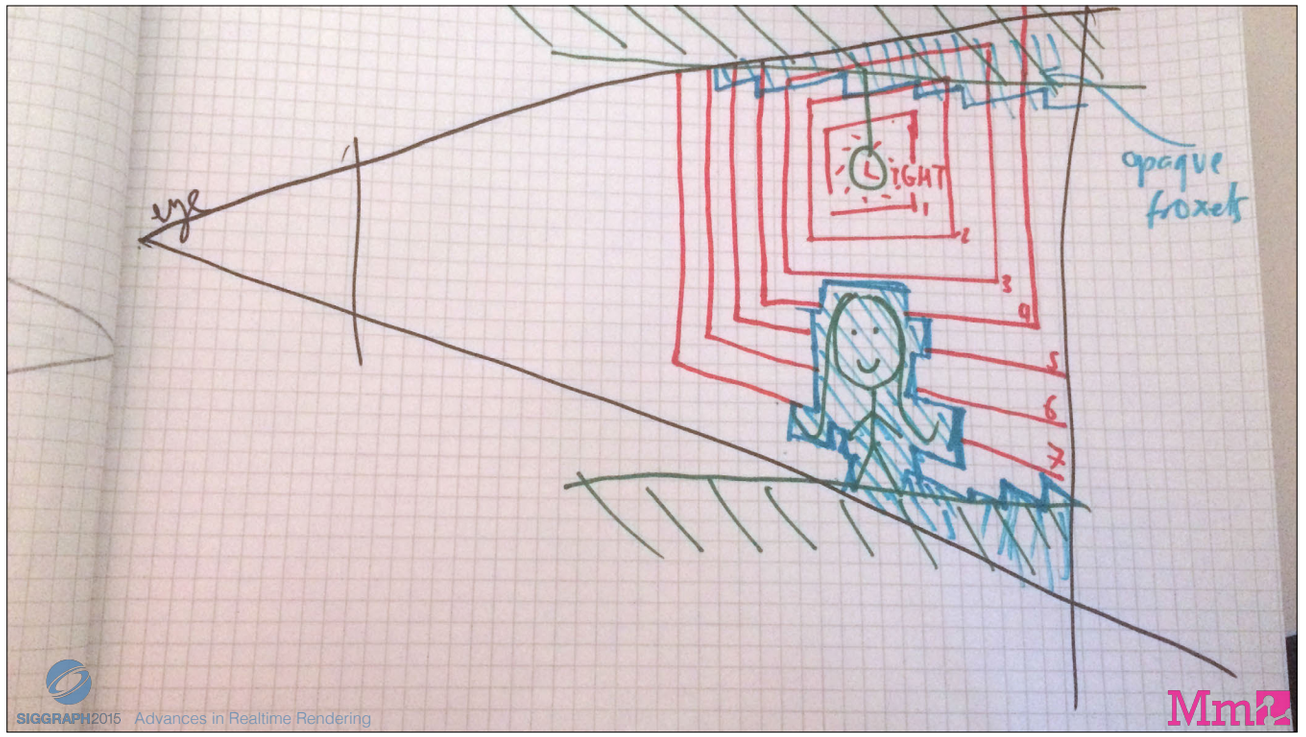
STOP PRESS - as far as I can tell from the brilliant morning session by frostbite guys, they have a better idea than the technique I present on the next few slides. They start from the same place -a dense froxel map of ‘density’, as above, but they resample it for each light into a per light 32^3 volume, in light-space. then they can smear density directly in light space. This is better than what I do over the next few slides, I think. See their talk for more!

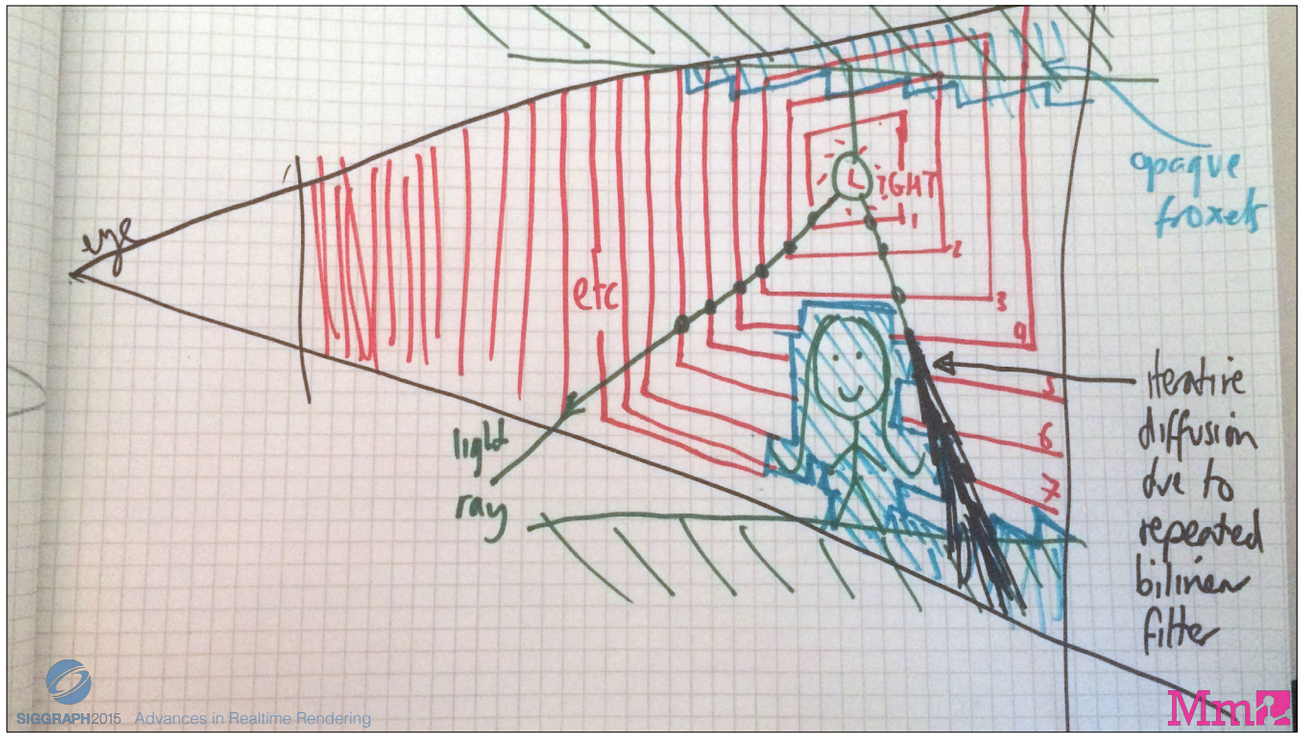


To wipe the light around, you set the single froxel where the light is to ‘1’ and kick a compute shader in 4 froxel thick ‘shells’ radiating out from that central light froxel. (with a sync between each shell). Each thread is a froxel in the shell, and reads (up to) 4 trilinear taps from the density volume, effectively a short raycast towards the light. Each shell reads from the last shell, so it’s sort of a ‘wipe’ through the whole frustum.



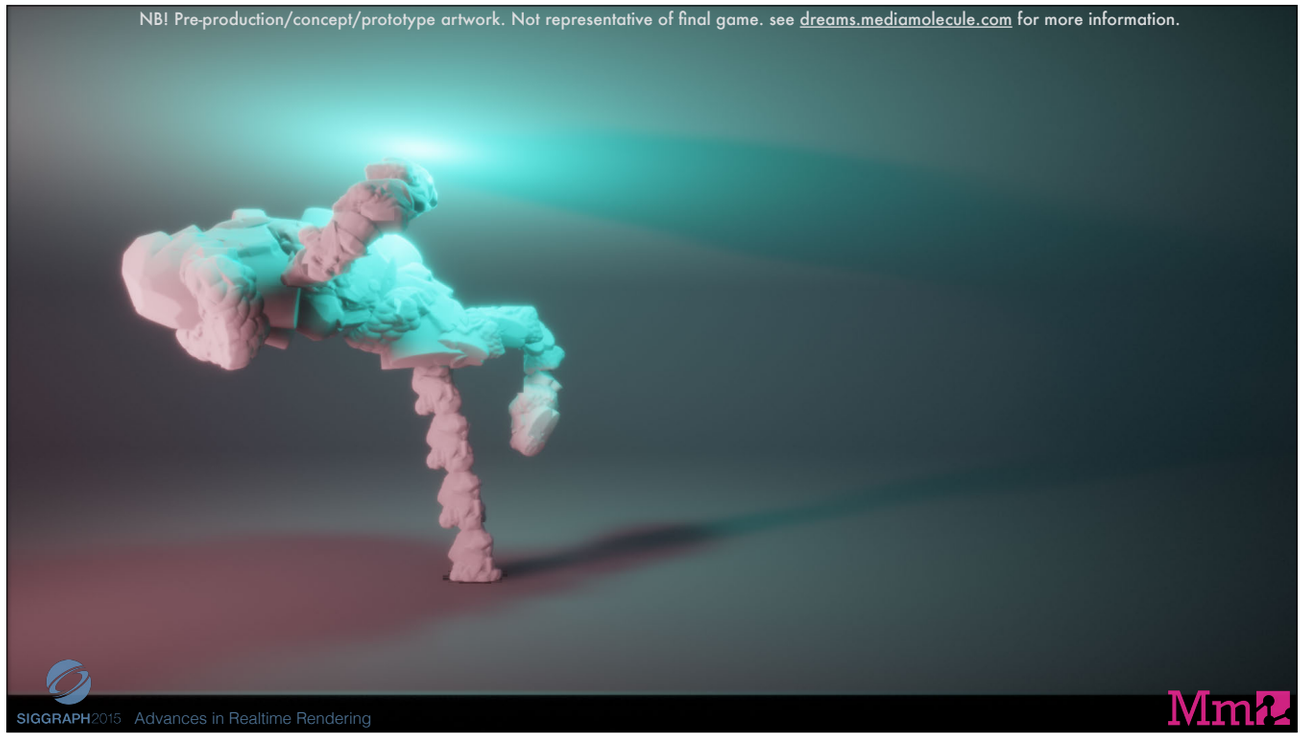
here come the shells! each one reads from the last. yes, there are stalls. no, they’re not too bad as you can do 4 lights and pipeline it all.


eye
LIGHT
opaque
froxels
1
2
3
4
5
6
7

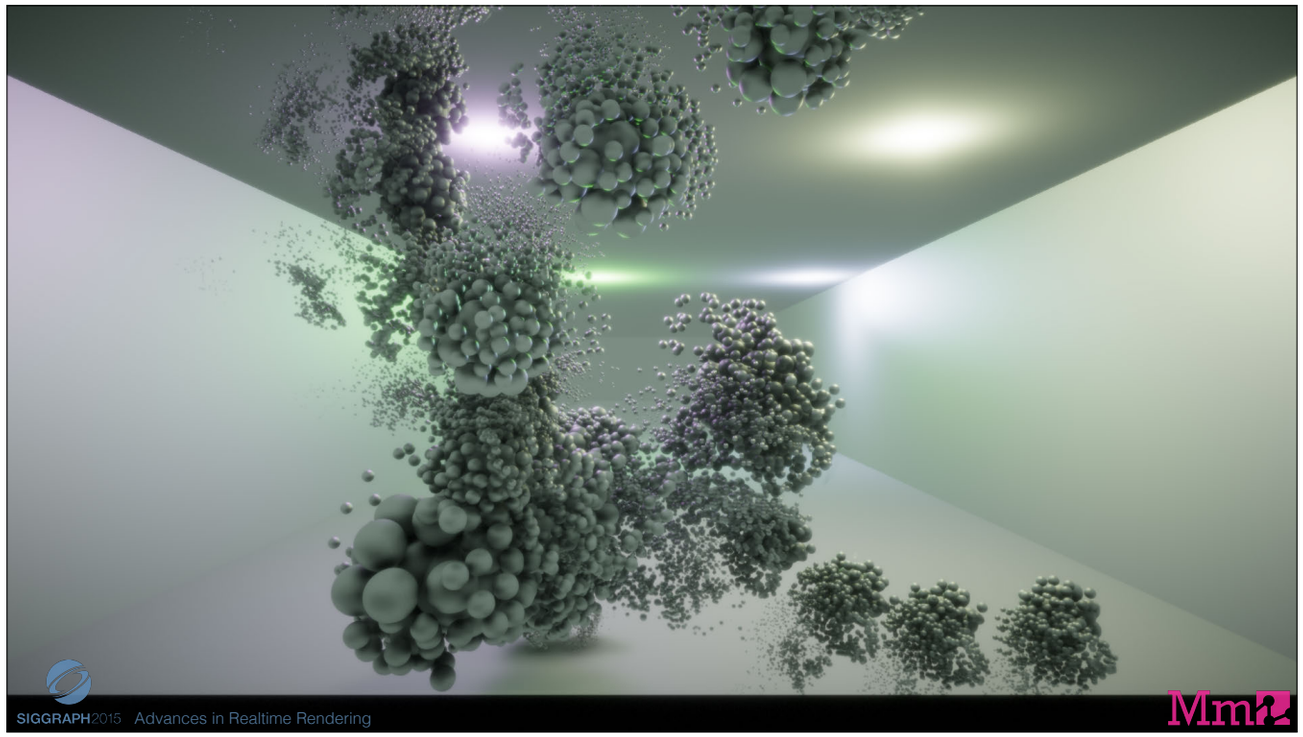


The repeated feedback causes a pleasant blur in the propagated shadows.
it’s like LPV propagation, except that it’s for a single light so you have no direction confusion, and you can wipe from one side of the screen to the other with a frame, since you process the froxels strictly in order radiating out from the light.

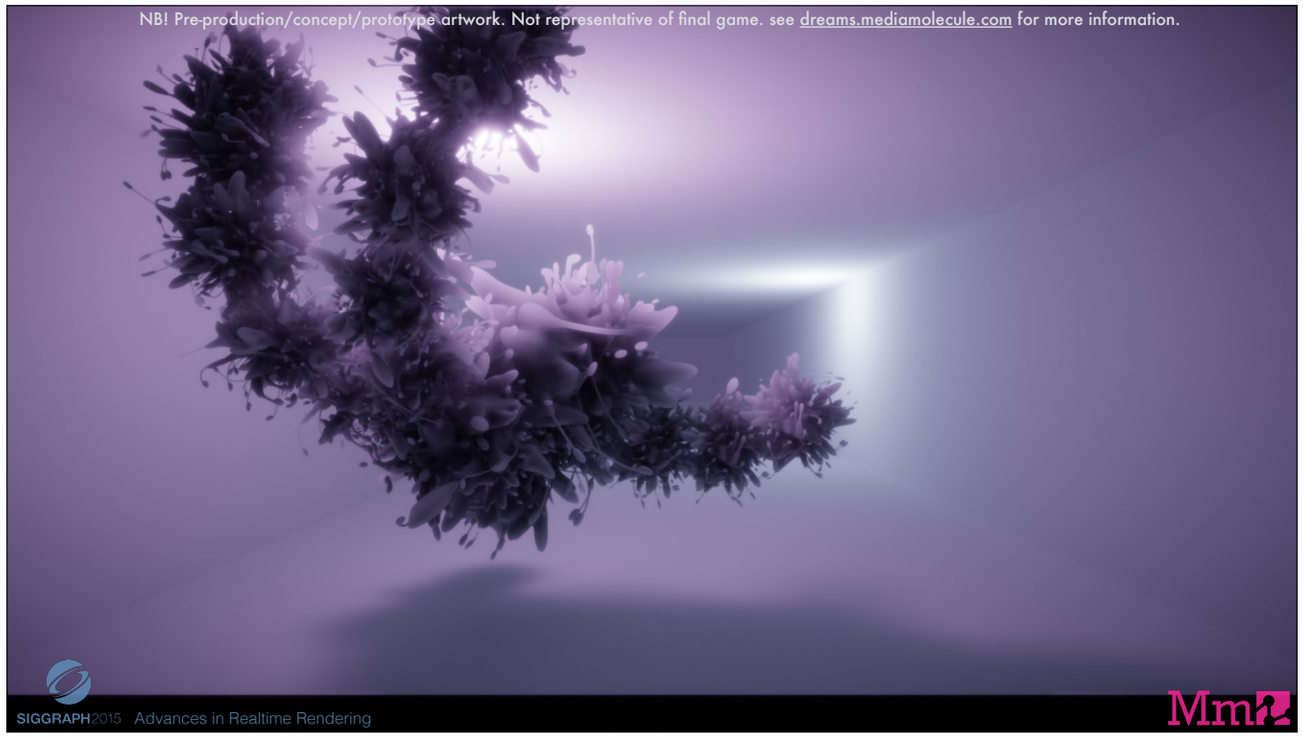
You can jitter the short rays to simulate area lights. You do 4 lights at once, to overlap the syncs, and you do it on an async pipe to mop up space on your compute units so the syncs don’t actually hurt that much. (offscreen lights are very painful to do well and the resolution is brutally low). However the results were pretty, and the ‘lighting’ became simple coherent volume texture lookups.
PICS PLZ:



Look ma! no shadowmaps!
would be super cool for participating media stuff, since we also have the brightness of every light conveniently stored at every froxel in the scene. I didn't implement it though….



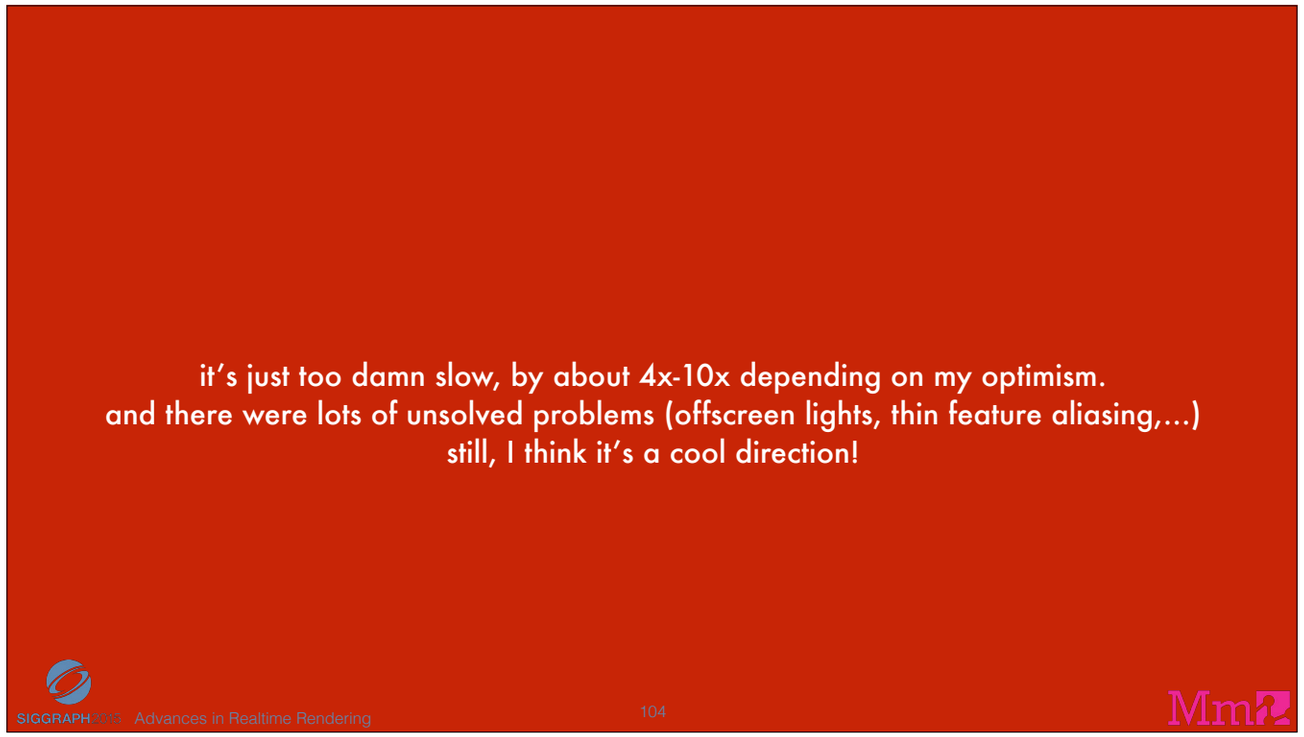
Ambient occlusion was done by simply generating mip-maps of the density volume and sampling it at positions offset from the surface by the normal, ie a dumb very wide cone trace.
The geometric detail and antialiasing was nice:



You could also get really nice subsurface effects by cone tracing the light volumes a little and turning down the N.L term:

BIG
SUCCESS!!?

However- the performance was about 4x lower than what I needed for PS4 (I forget the timings, but it was running at 30 for the scenes above ñ but only just! For more complex scenes, it just died). The lighting technique and the refinement engine are separate ideas, but they both had too many limitations and performance problems that I didn't have time to fix.



(ie I still think this technique has legs, but I can’t make it work for this particular game)
in particular, since edge pixels could still get unboundedly 'deep', the refinement lists were quite varied in length, I needed to jump through quite a few hoops to keep the GPU well load balanced. I also should have deferred lighting a bit more - I lit at every leaf voxel, which was slow. however everything I tried to reduce (merge etc) led to visible artefacts. what I didn’t try was anything stochastic. I had yet to fall in love with 'stochastic all the things'.... definitely an avenue to pursue.

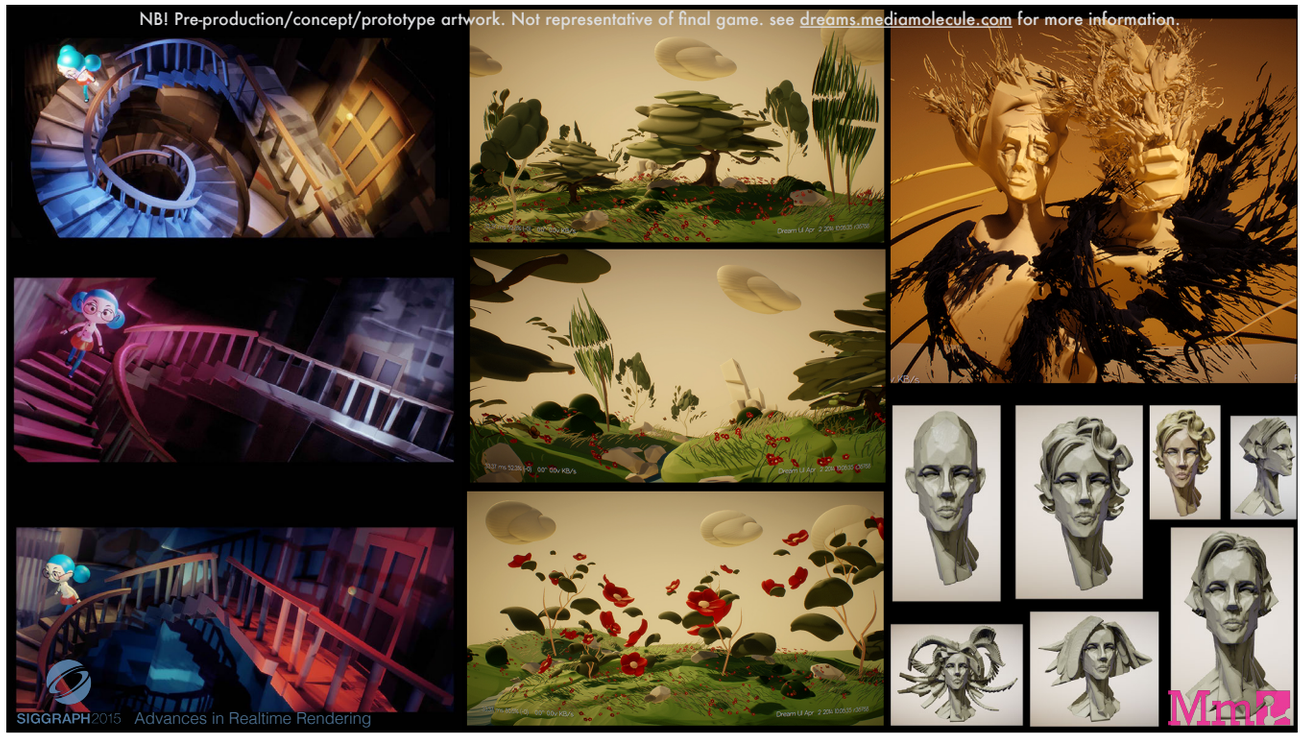
We were also struggling with the memory for all the gigavoxel bricks.

ARGH 3 (meltdown edition)

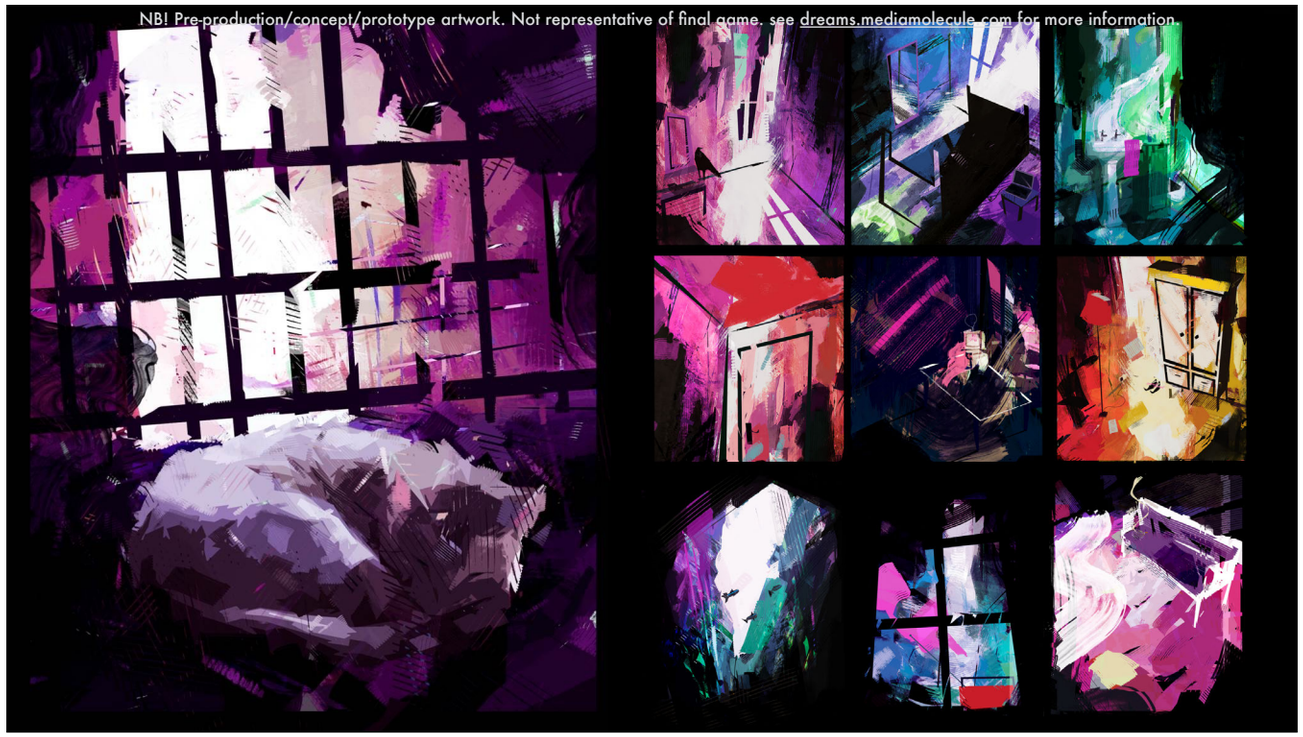
Polygons? too difficult
Volumetric Billboards? too much filtrate
Gigavoxels? single object
Hybrid with rasterised gigavoxel cubes? too hard to render OIT overlaps
Froxel Refinement volumetric render? too slow
AND THE GAME DOESNT EVEN LOOK THAT 'DIFFERENT' FROM POLYS

FML x 10000000 :(

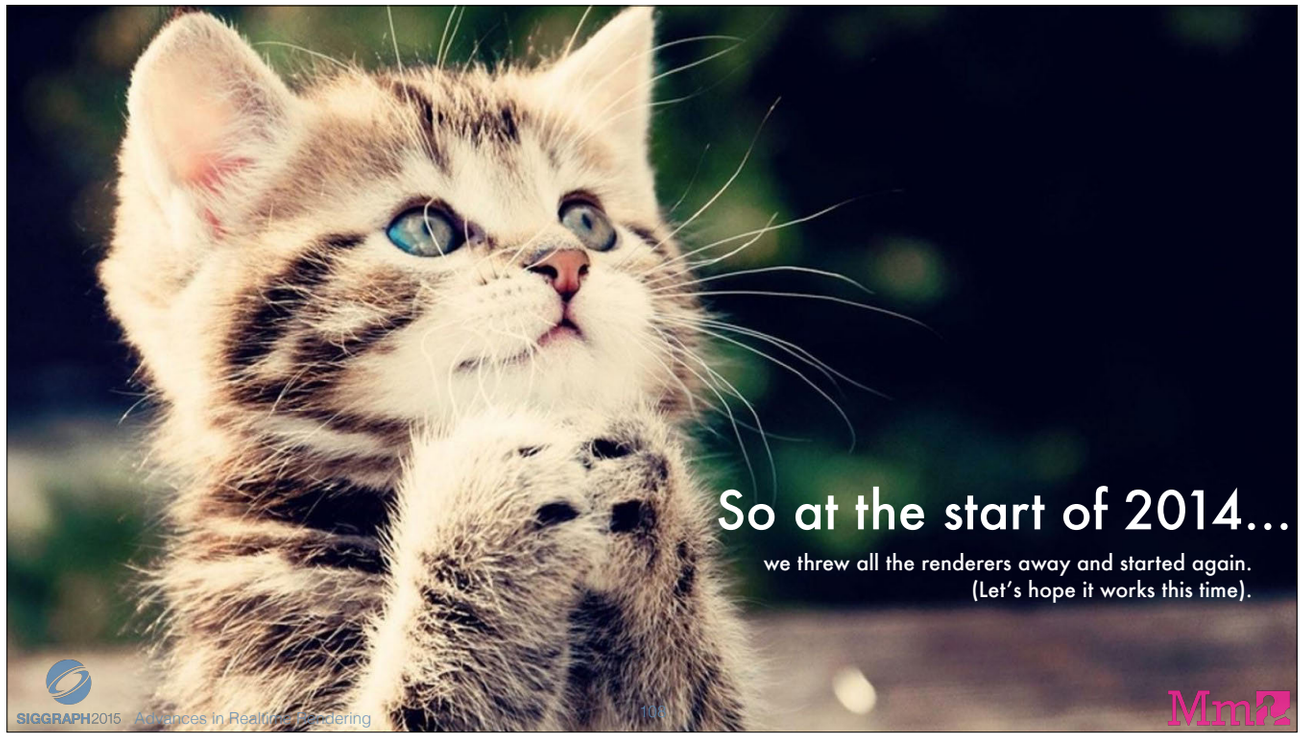
The nail in the coffin was actually to do with art direction.
directly rendering the distance field sculptures was leaving very little to the imagination . So it was very hard to create 'good looking' sculptures; lots of designers were creating content that basically looked like untextured unreal-engine, or 'crap' versions of what traditional poly engines would give you, but slower. It was quite a depressing time because as you can see it's a promising tech, but it was a tad too slow and not right for this project.
TL;DR:
this is the start of 2014. we're 3 years in, and the engine prototypes have all been rejected, and the art director (rightly) doesn't think the look of any of them suits the game.
argh.
SO……..



there was a real growing uneasiness in the studio. I had been working on OIT - refinement and sorting and etc for a LONG time; in the meantime, assets were being made using the ‘hard’ variant of the bricks engine, that simply traced each 8x8x8 rasterised brick for the 0 crossing and output raw pixels which were forward lit. at its best, it produced some lovely looking results (above) - but that was more the art than the engine! It also looked rather like ‘untextured poly engine’ - why were we paying all this runtime cost (memory & time) to render bricks if they just gave us a poly look?



also, there was a growing disparity between what the art department - especially art director kareem and artist jon - were producing as reference/concept work. it was so painterly!
there was one particular showdown with the art director, my great friend kareem, where he kept pointing at an actual oil painting and going 'I want it to look like this' and I'd say 'everyone knows concept art looks like that but the game engine is a re-interpretation of that' and kareem was like 'no literally that'. it took HOURS for the penny to drop, for me to overcome my prejudice.

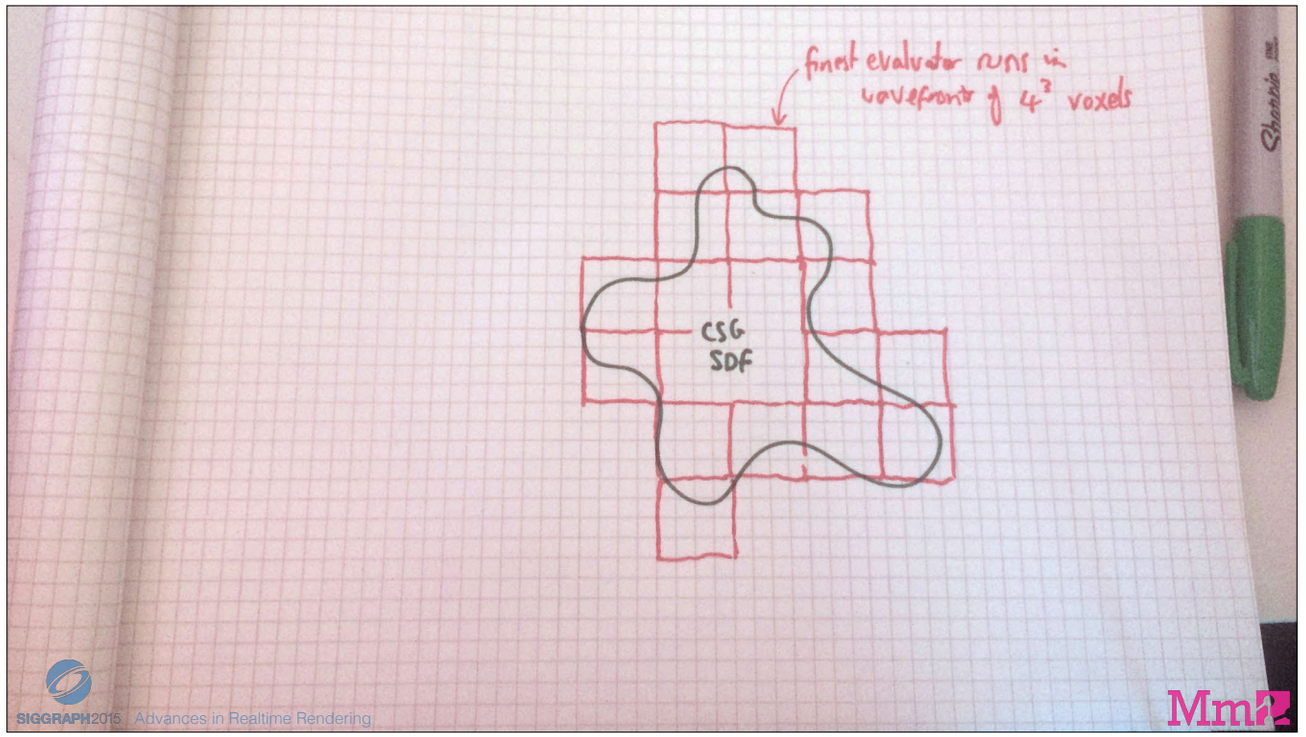


So after talking to the art director and hitting rock bottom in January 2014, he convinced me to go with a splat based engine, intentionally made to look like 3d paint strokes. I have a strong dislike of 'painterly post fx' especially 2d ones, so I had resisted this direction for a looooooooooong time.

(btw this is building on the evaluator as the only thing that has survived all this upheaval)

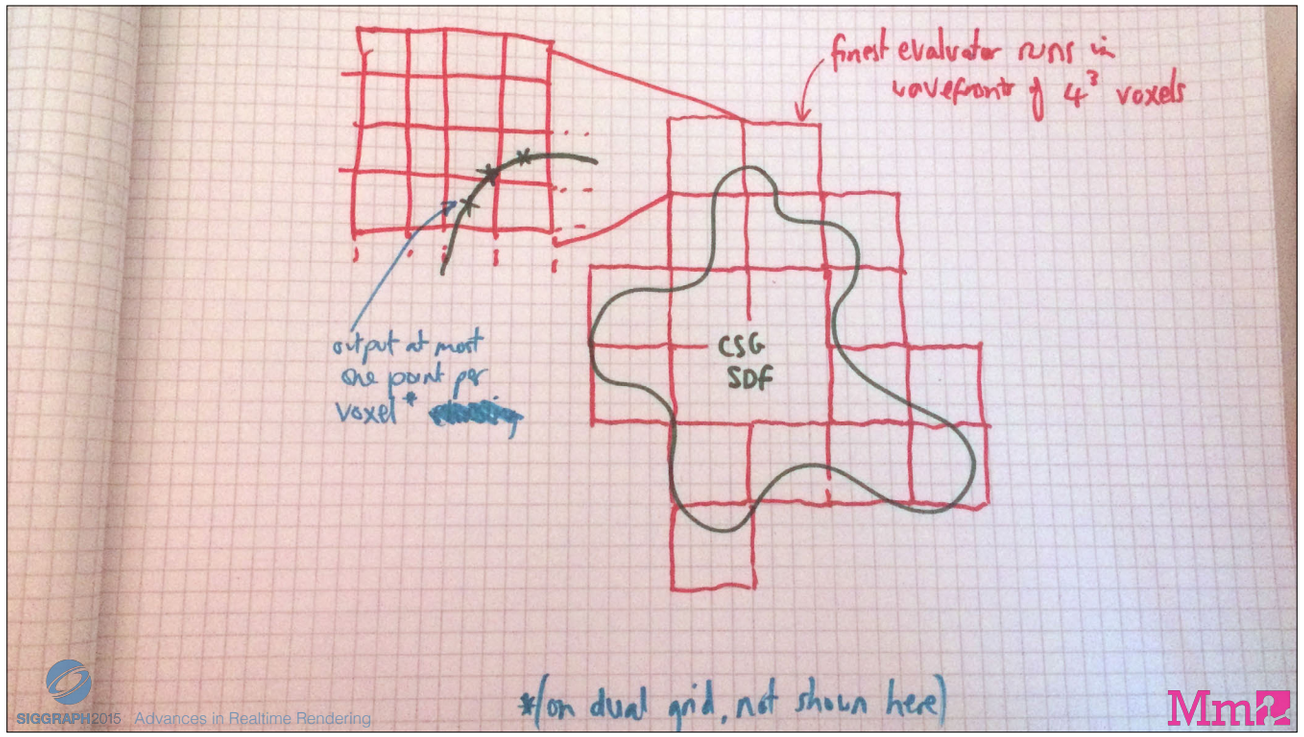
I had to admit that for our particular application of UGC, it was *brutal* that you saw your exact sculpture crisply rendered, it was really hard to texture & model it using just CSG shapes. (we could have changed the modelling primitives to include texturing or more noise type setups, but the sculpting UI was so loved that it was not movable. The renderer on the other hand was pretty but too slow, so it got the axe instead).

So I went back to the output of the evaluator, poked simon a bit, and instead of using the gigavoxel style bricks, I got point clouds, and had a look at what I could do. There's a general lesson in here too - that tech direction and art direction work best when they are both considered, both given space to explore possibilities; but also able to give different perspectives on the right (or wrong) path to take.



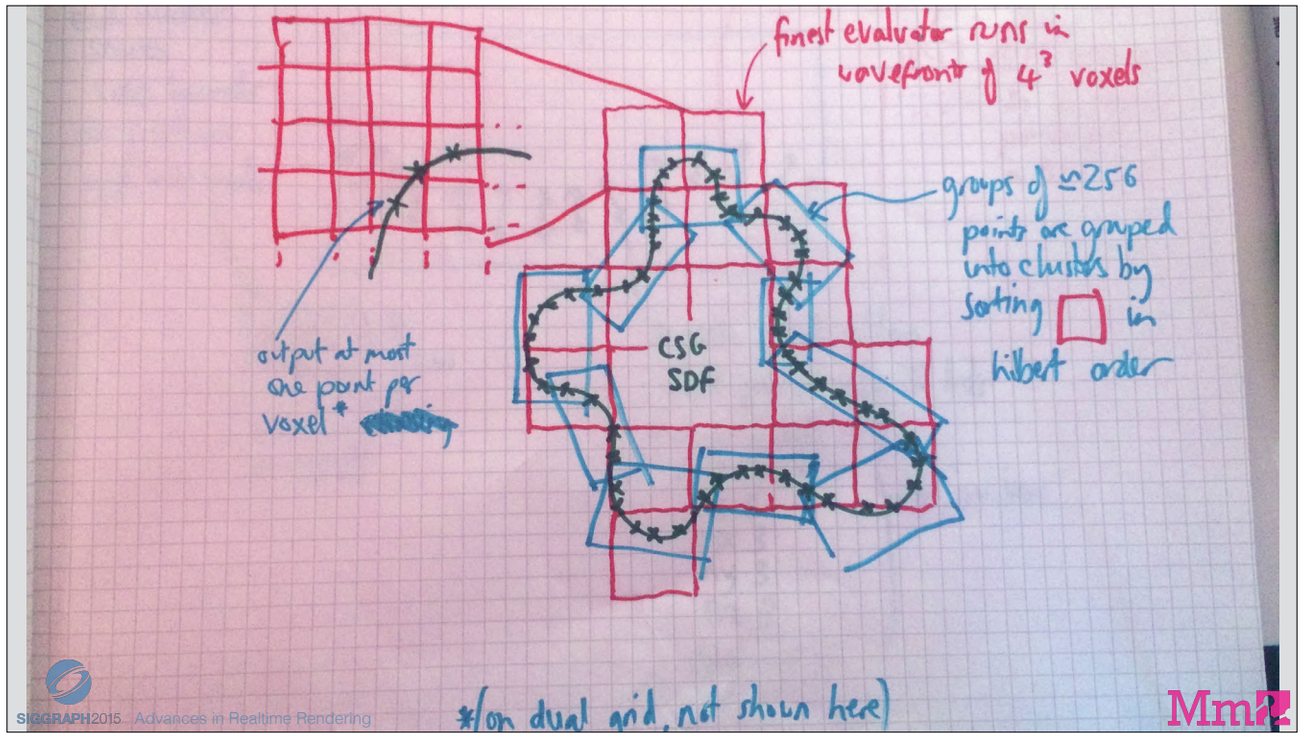
So! now the plan is: generate a nice dense point cloud on the surface of our CSG sculpts.
EVERYTHING is going to be a point cloud. the SDF becomes an intermediate representation, we use it to spawn the points at evaluation time, (and also for collision. but thats another talk)

we started from the output of the existing evaluator, which if you remember was hierarchically refining lists of primitives to get close to voxels on the surface of the SDF.
as it happens, the last refinement pass is dealing in 4x4x4 blocks of SDF to match GCN wavefronts of 64 threads.

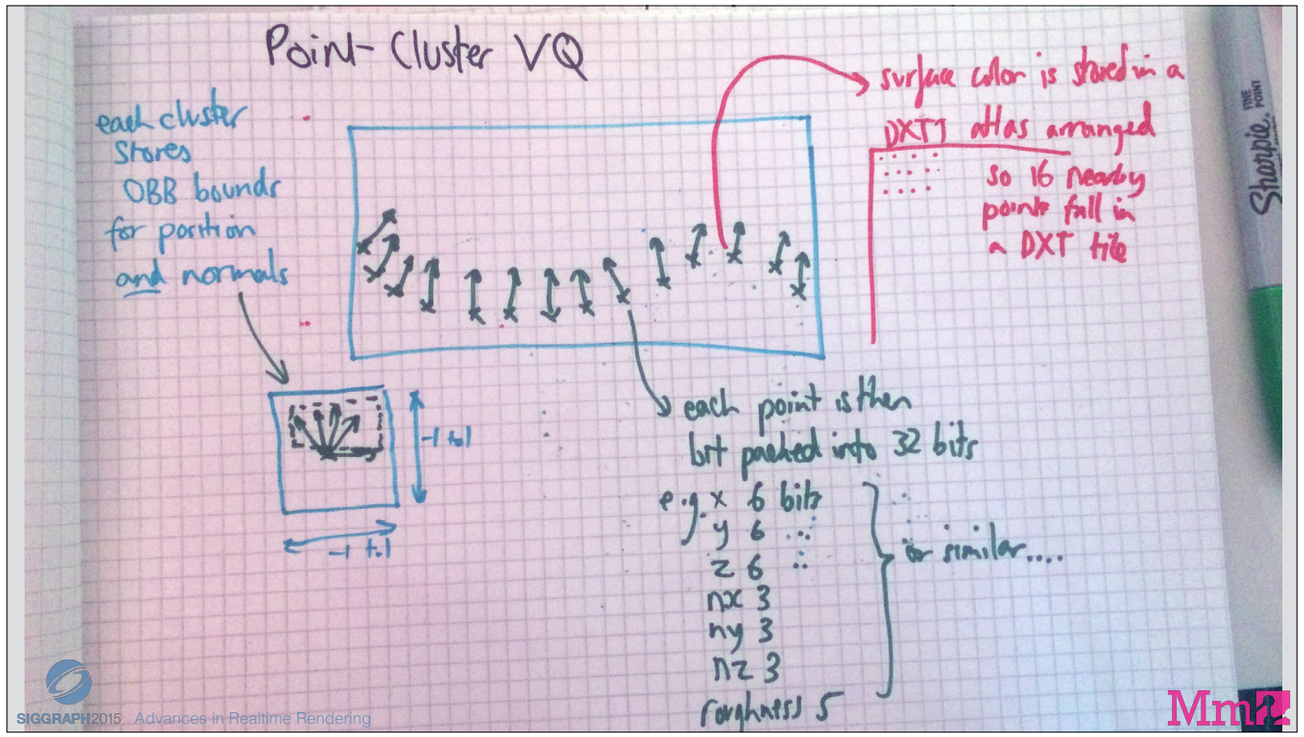


We add one point to the cloud per leaf voxel (remember, thats about 900^3 domain, so for example, a sphere model will become a point cloud with diameter 900 and one point per integer lattice cell that intersects the sphere surface)

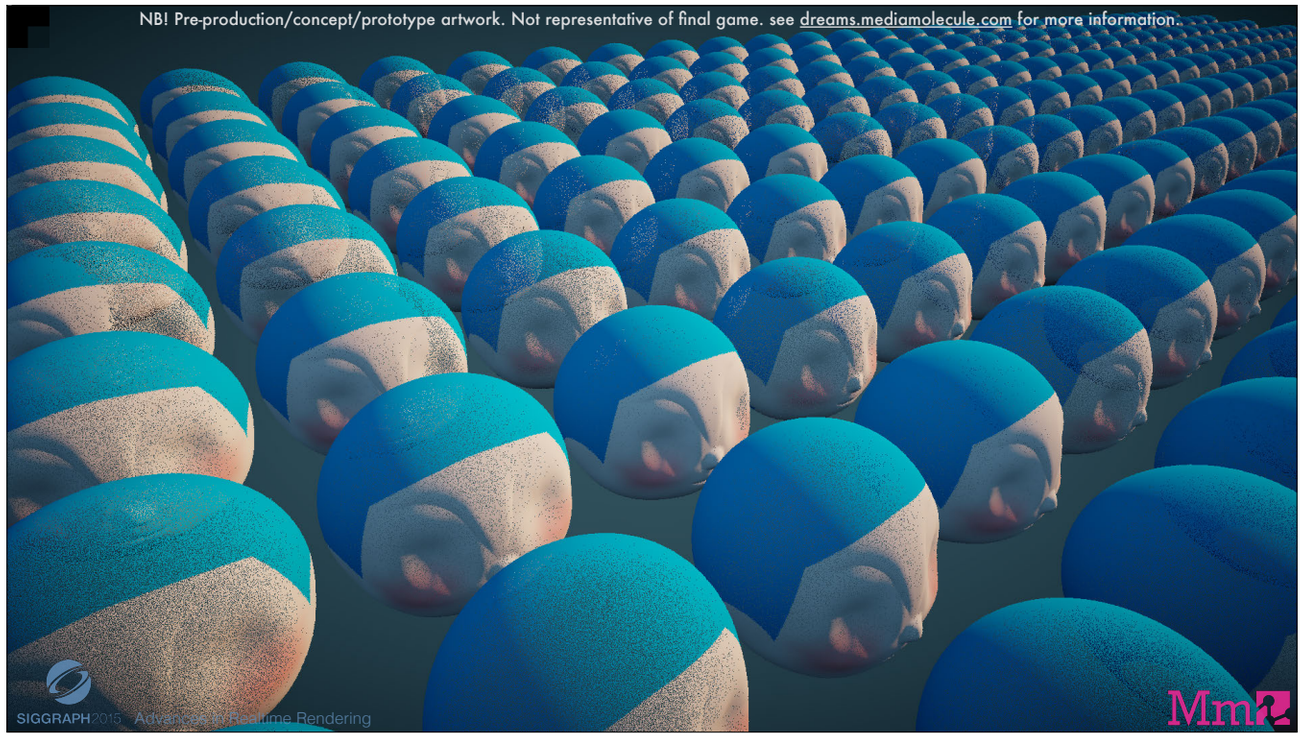
actually we are using a dual grid IIRC so that we look at a 2x2x2 neighbourhood of SDF values and only add points where there is a zero crossing.
So now we have a nice fairly even, dense point cloud. Since the bounding voxel grid is up to around 900^3 voxels -> around 2 million surface voxels -> around 2 million points.



The point cloud is sorted into Hilbert order (actually, 4^3 bricks of voxels are in Hilbert order and then the surface voxels inside those bricks are in raster order, but I digress) and cut into clusters of approximately 256 points (occasionally there is a jump in the hilbert brick order so we support partially filled clusters, to keep their bounding boxes tight).



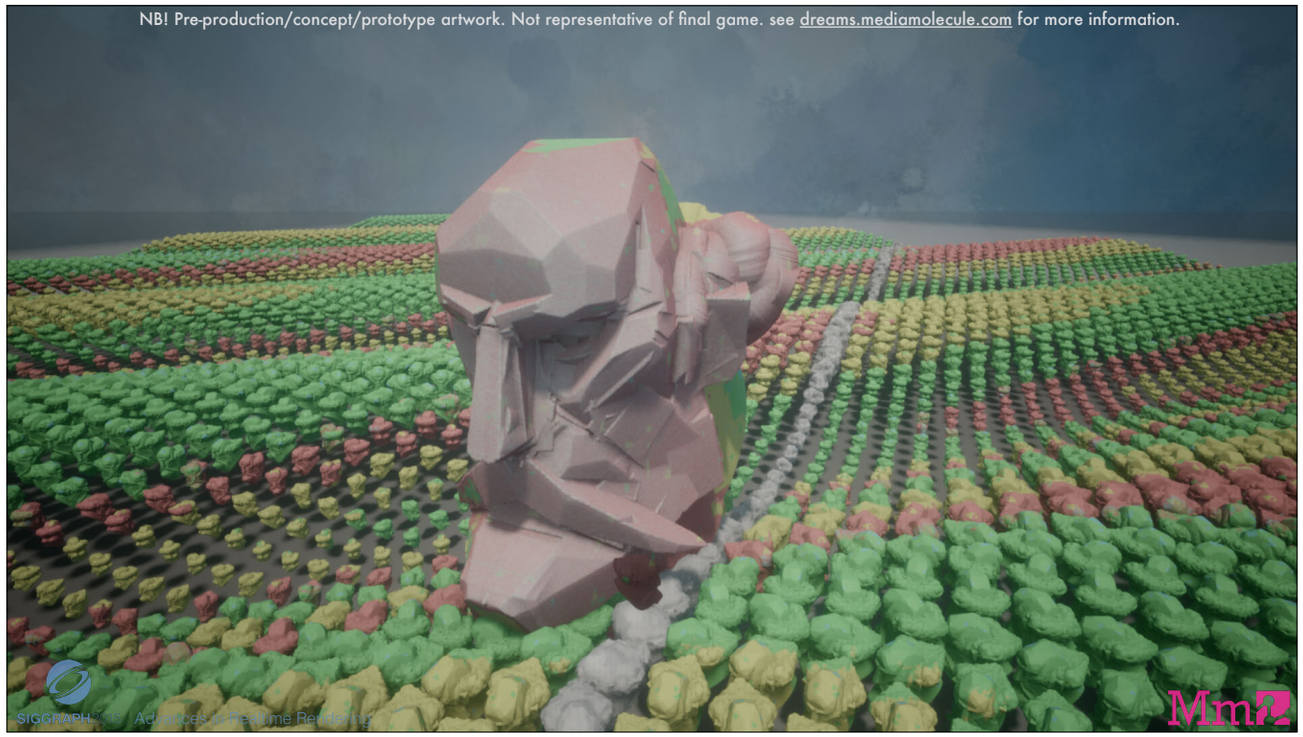
Each cluster is tightly bounded in space, and we store for each a bounding box, normal bounds. then each point within the cluster is just one dword big, storing bitpacked pos,normal,roughness, and colour in a DXT1 texture. All of which is to say, we now have a point cloud cut into lumps of 256 points with a kind of VQ compression per point. We also compute completely independent cluster sets for each LOD - that is, we generate point clouds and their clusters for a ‘mip pyramid’ going from 900 voxels across, to 450, to 225, etc.



I can’t find many good screenshots but here’s an example of the density, turned down by a factor of 2x to see what’s going on.

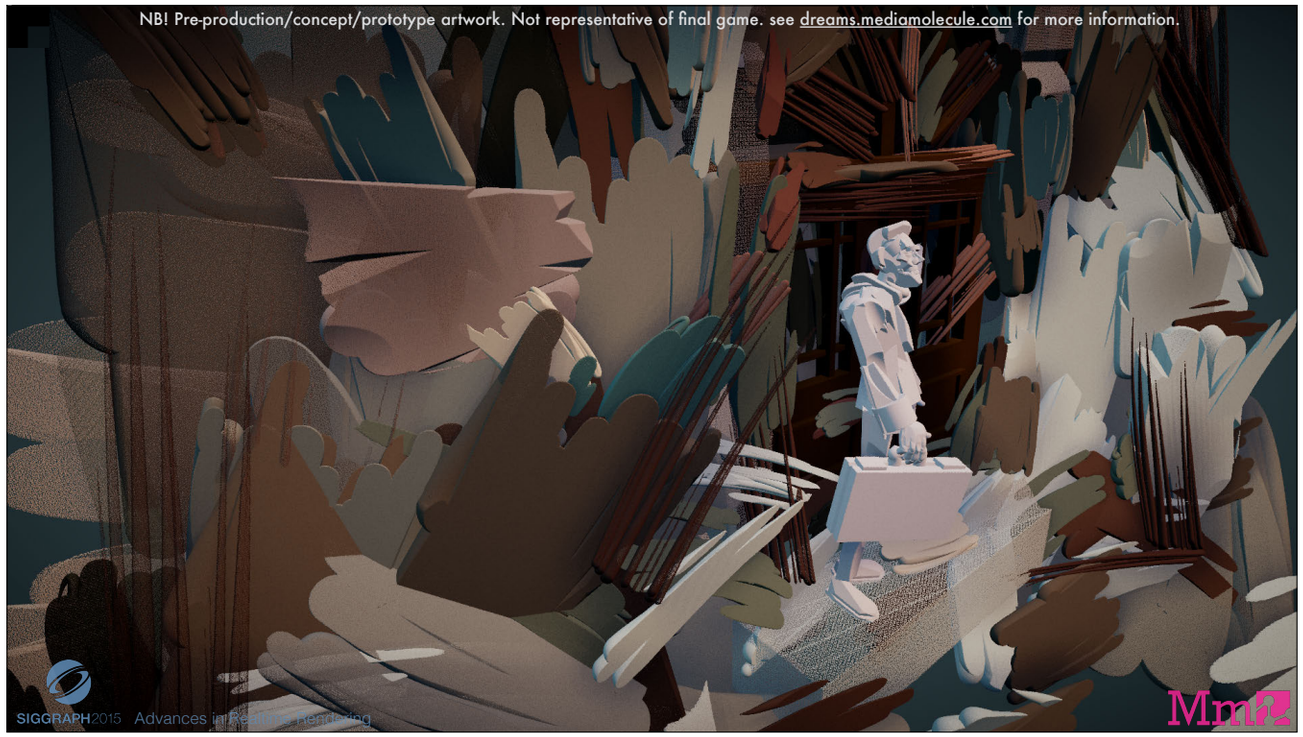
my initial tests here were all PS/VS using the PS4 equivalent of glPoint. it wasn’t fast, but it showed the potential. I was using russian roulette to do 'perfect' stochastic LOD, targeting a 1 splat to 1 screen pixel rate , or just under.

At this point we embraced TAA *bigtime* and went with ‘stochastic all the things, all the time!’. Our current frame, before TAA, is essentially verging on white noise. It’s terrifying. But I digress!

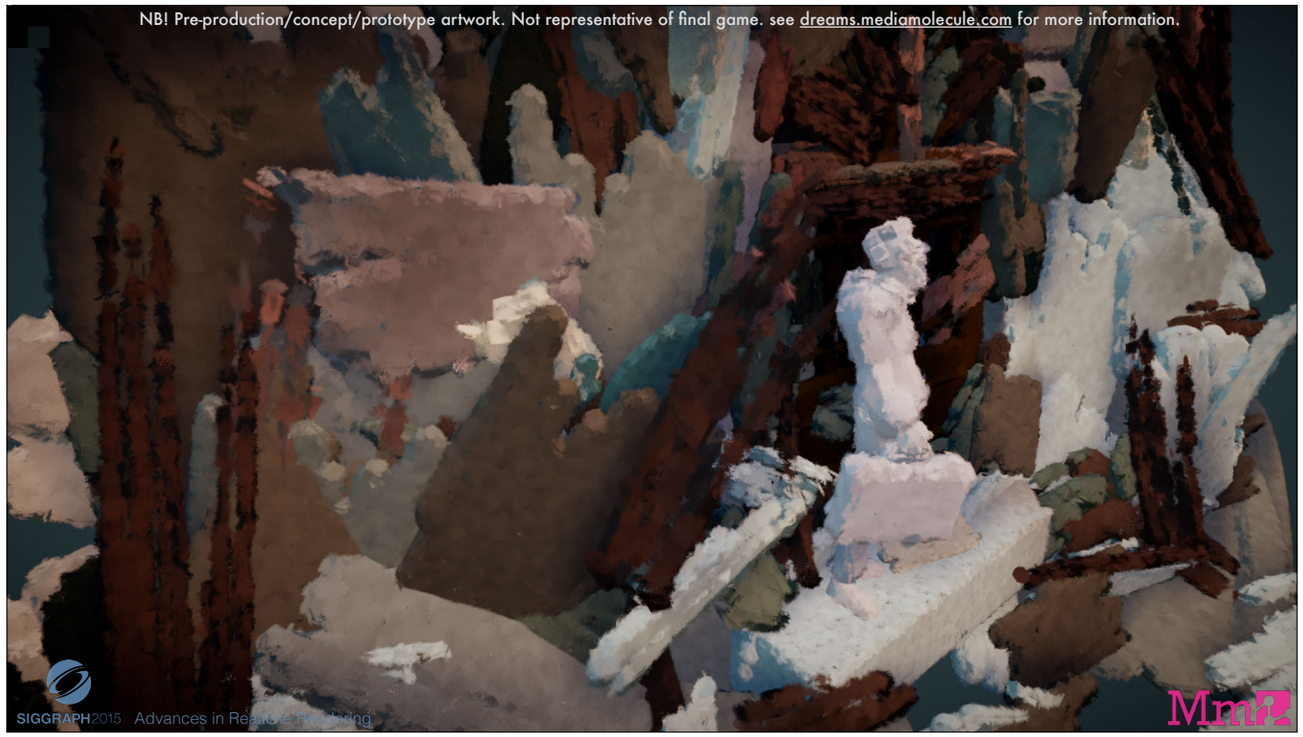


for rendering, we arranged the clusters for each model into a BVH. we also computed a separate point cloud, clustering and BVH for each mipmap (LOD) of the filtered SDF. to smooth the LOD transitions, we use russian roulette to adapt the number of points in each cluster from 256 smoothly down to 25%, i.e. 256 down to 64 points per cluster, then drop to the next LOD.
simon wrote some amazingly nicely balanced CS splatters that hierarchically culled and refined the precomputed clusters of points, computes bounds on the russian roulette rates, and then packs reduced cluster sets into groups of ~64 splats.

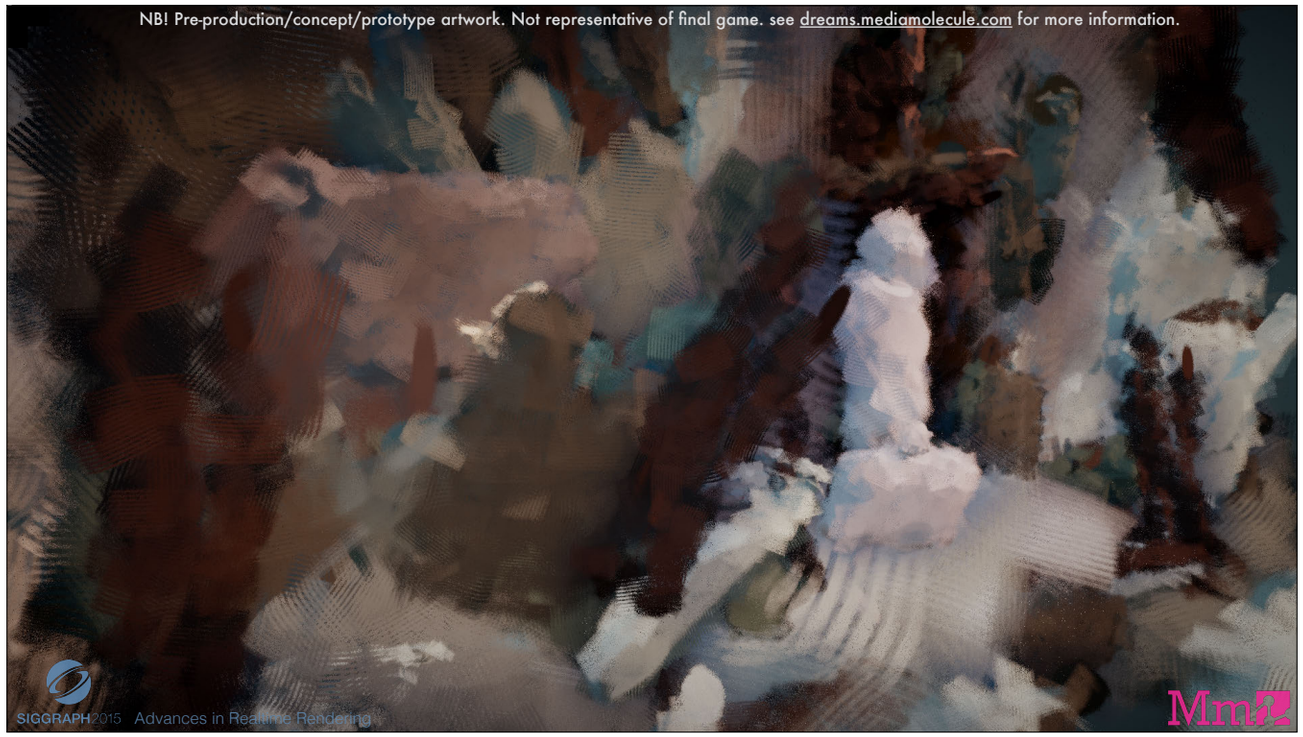
so in this screenshot the color cycling you can see is visualizing the steps through the different degrees of decimation, from <25%, <50%, <75%, then switching to a completely different power of 2 point cloud;



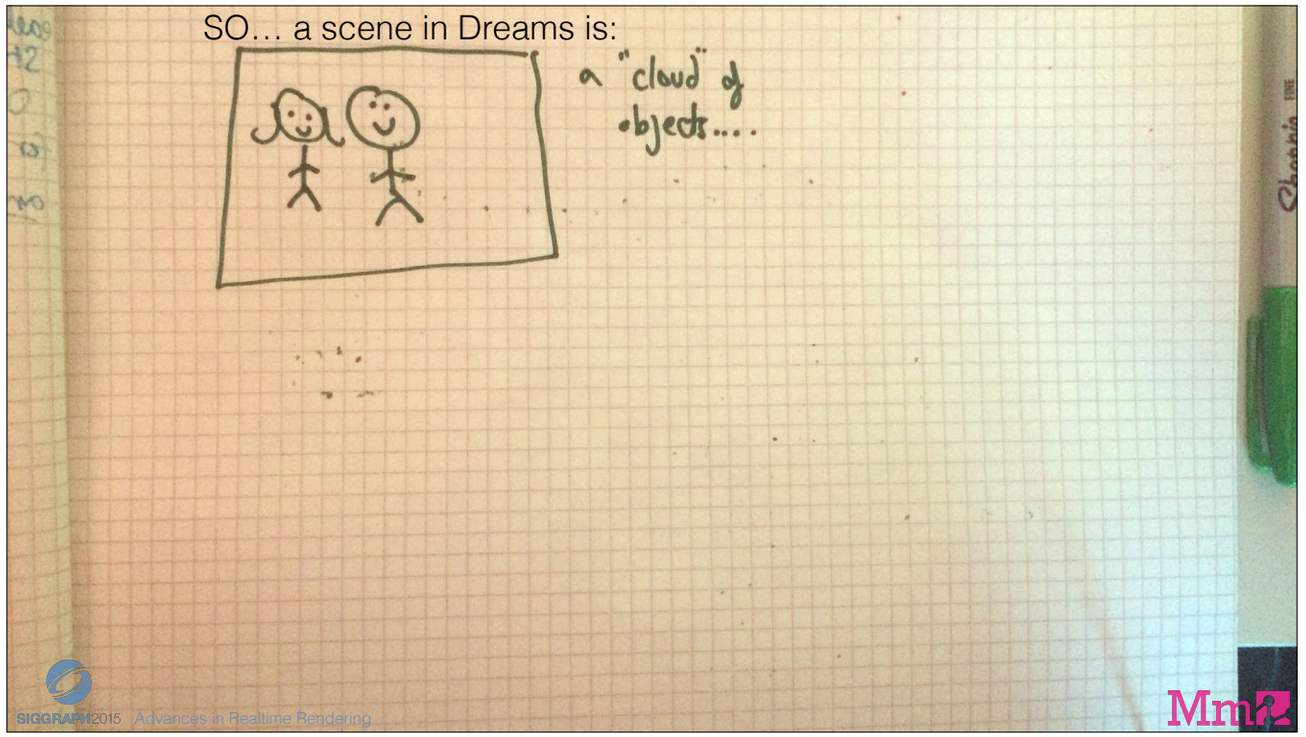
What you see is the ‘tight’ end of our spectrum. i.e. the point clouds are dense enough that you see sub pixel splats everywhere. The artist can also ‘turn down’ the density of points, at which point each point becomes a ‘seed’ for a traditional 2d textured quad splat. Giving you this sort of thing:


NB! Pre-production/concept/prototype artwork. Not representative of final game. see dreams.mediamolecule.com for more information.
SIGGRAPH2015 Advances in Real-Time Rendering

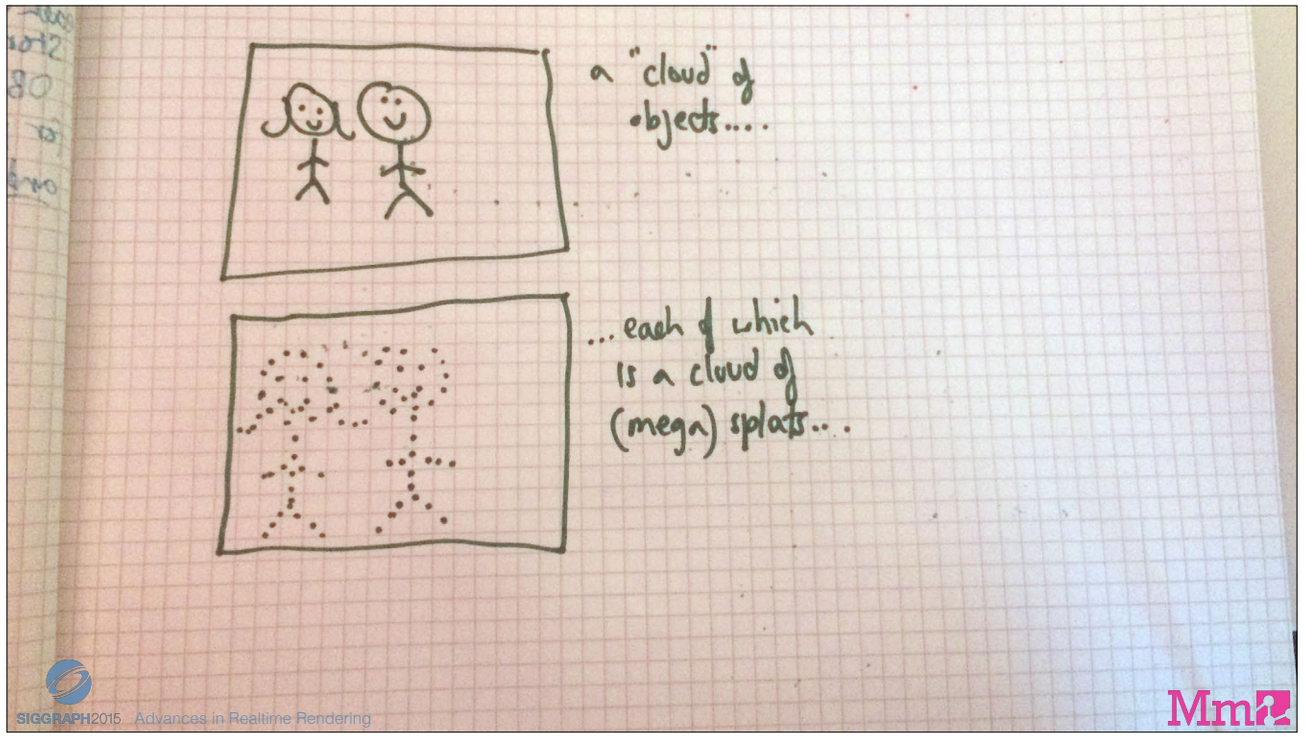
We use pure stochastic transparency, that is, we just randomly discard pixels based on the alpha of the splat, and let TAA sort it out. It works great in static scenes.
However the traditional ‘bounding box in color space’ to find valid history pixelsí starts breaking down horribly with stochastic alpha, and we have yet to fully solve that.
So we are still in fairly noisy/ghosty place. TODO!
We started by rendering the larger strokes - we call them megasplats - as flat quads with the rasterizer. thats what you see here, and in the E3 trailer.



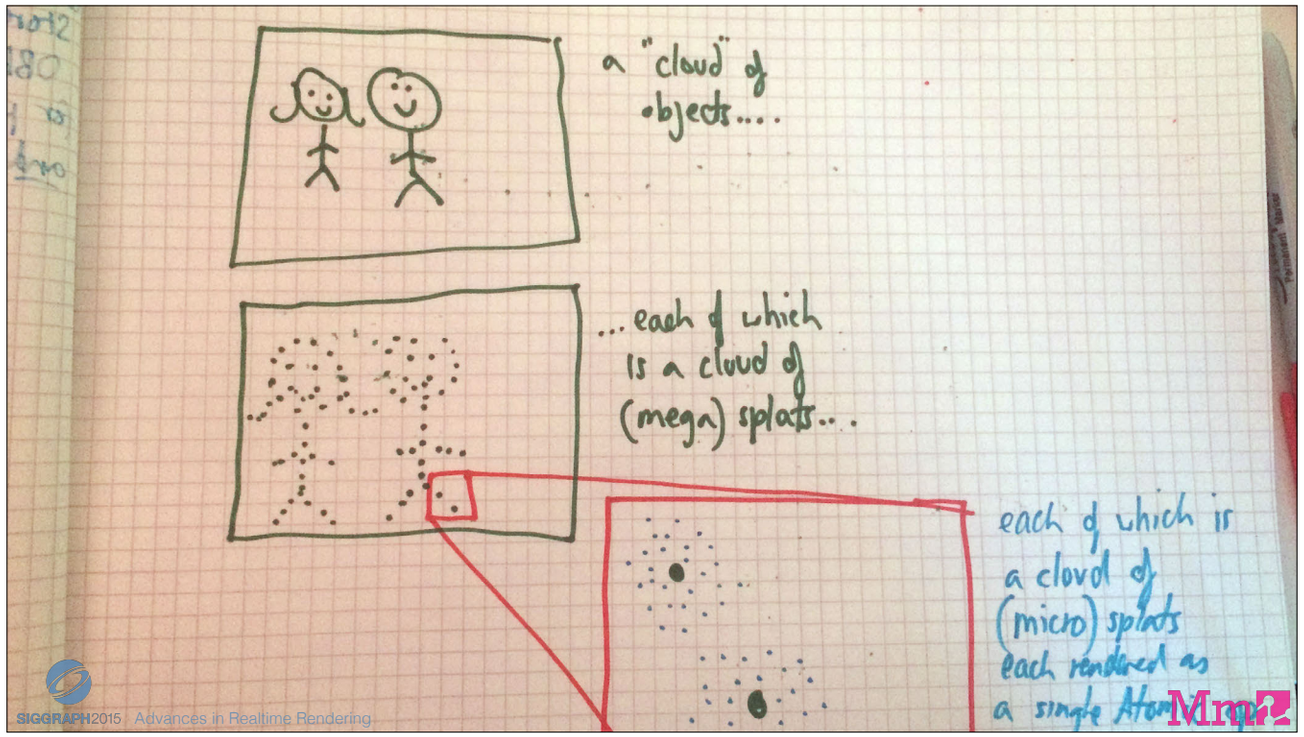
Interestingly , simon tried making a pure CS ‘splatting shader’, that takes the large splats, and instead of rasterizing a quad, we actually precompute a ‘mini point cloud’ for the splat texture, and blast it to the screen using atomics, just like the main point cloud when it’s in ‘microsplat’ (tight) mode.



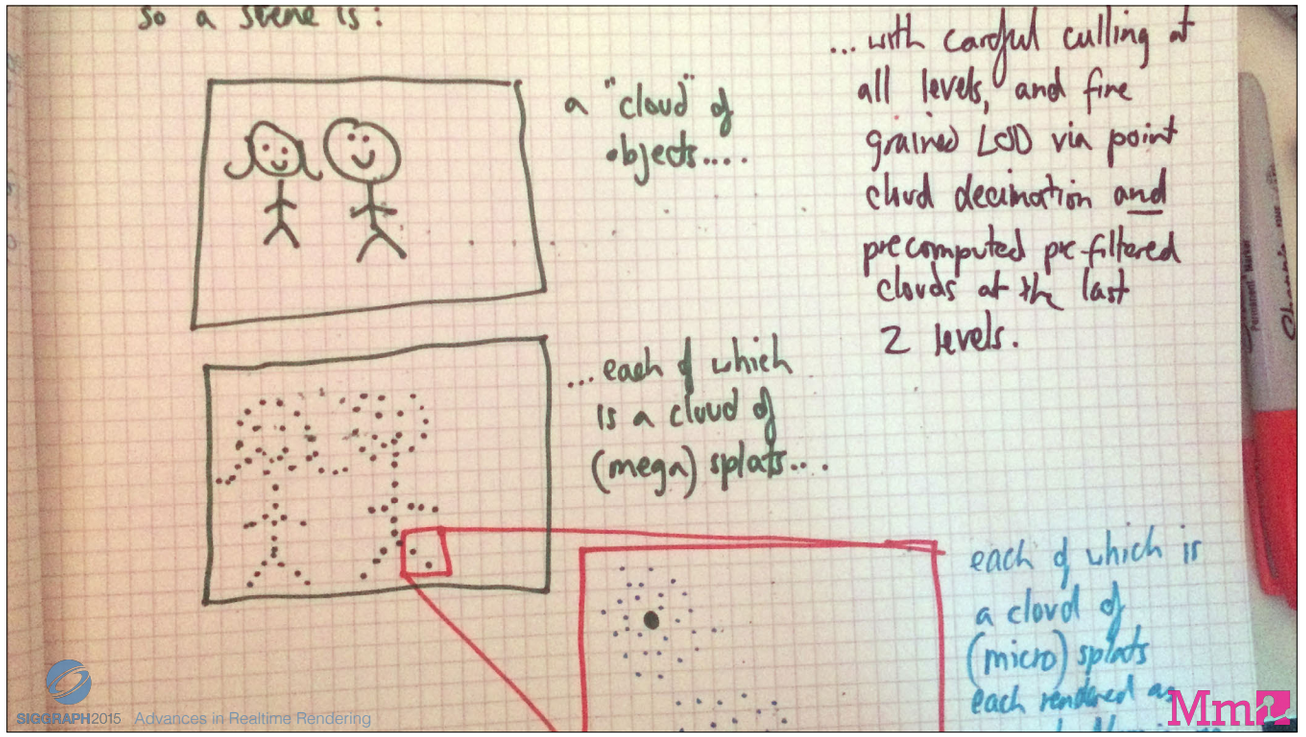
So now we have a scene made up of a whole cloud of sculpts…



which are point clouds,



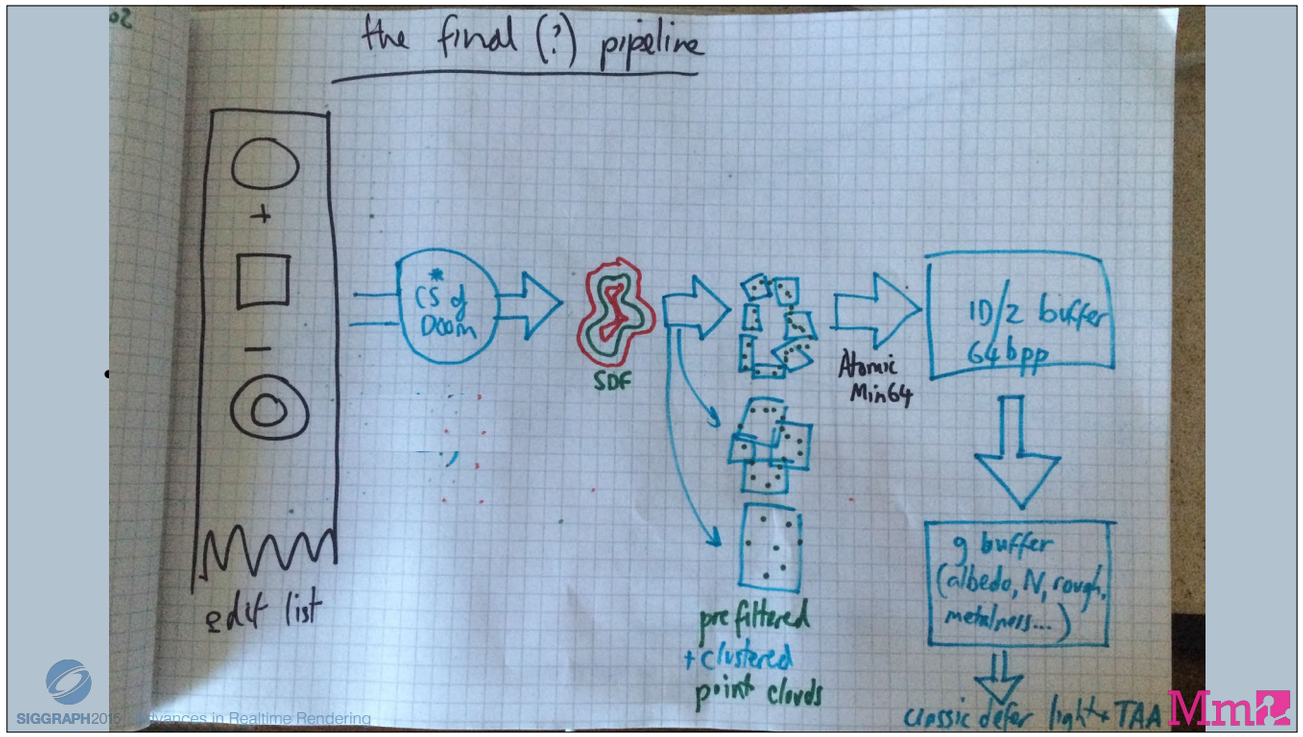
and each point is itself, when it gets close enough to the camera, an (LOD adapted) ‘mini’ point cloud - Close up, these mini point clouds representing a single splat get ‘expanded’ to a few thousand points (conversely, In the distance or for ‘tight’ objects, the mini points clouds degenerate to single pixels).
Amusingly, the new CS based splatter beats the rasterizer due to not wasting time on all the alpha=0 pixels. That also means our ‘splats’ need not be planar any more, however, we don’t yet have an art pipe for non-planar splats so for now the artists don’t know this! Wooahaha!



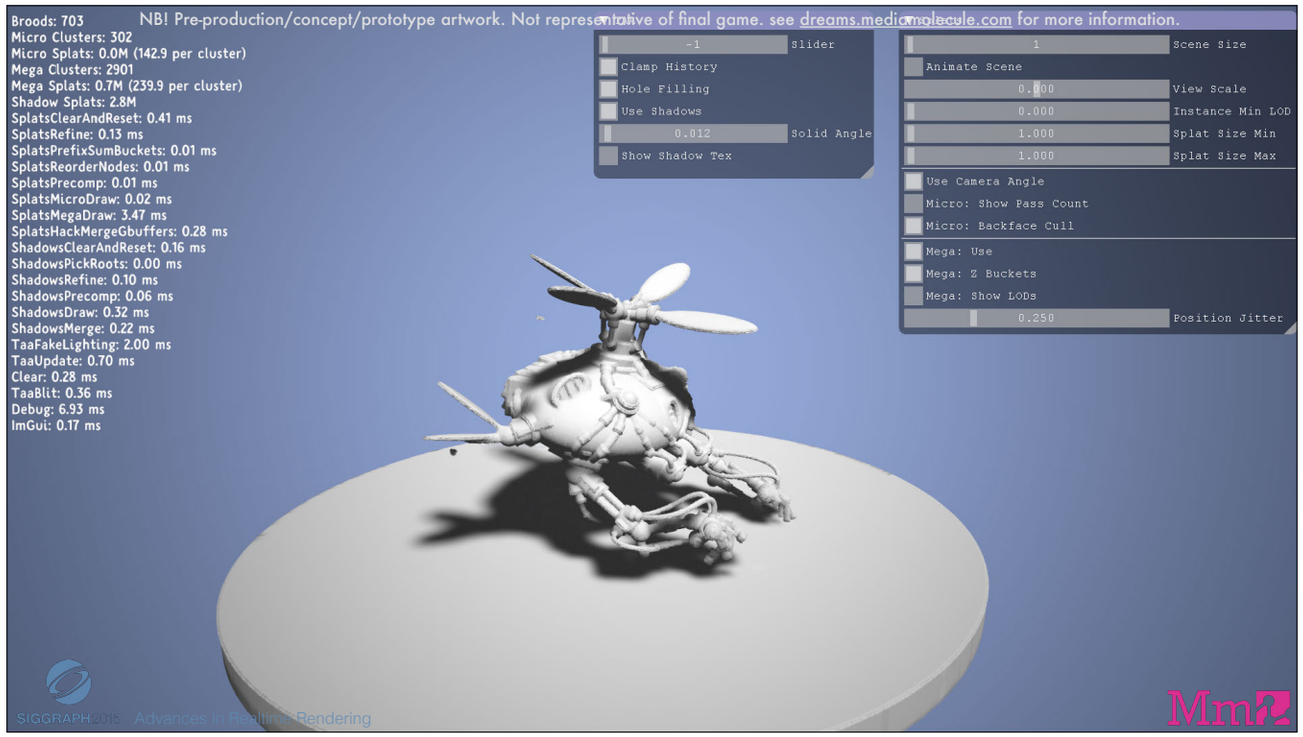
That means that if I were to describe what the current engine is, I'd say it's a cloud of clouds of point clouds. :)

Incidentally, this atomic based approach means you can do some pretty insane things to get DOF like effects: instead of post blurring, this was a quick test where we simply jittered the splats in a screenspace disc based on COC, and again let the TAA sort it all out.
It doesn’t quite look like blur, because it isn’t - its literally the objects exploding a little bit - but it’s cool and has none of the usual occlusion artefacts :)

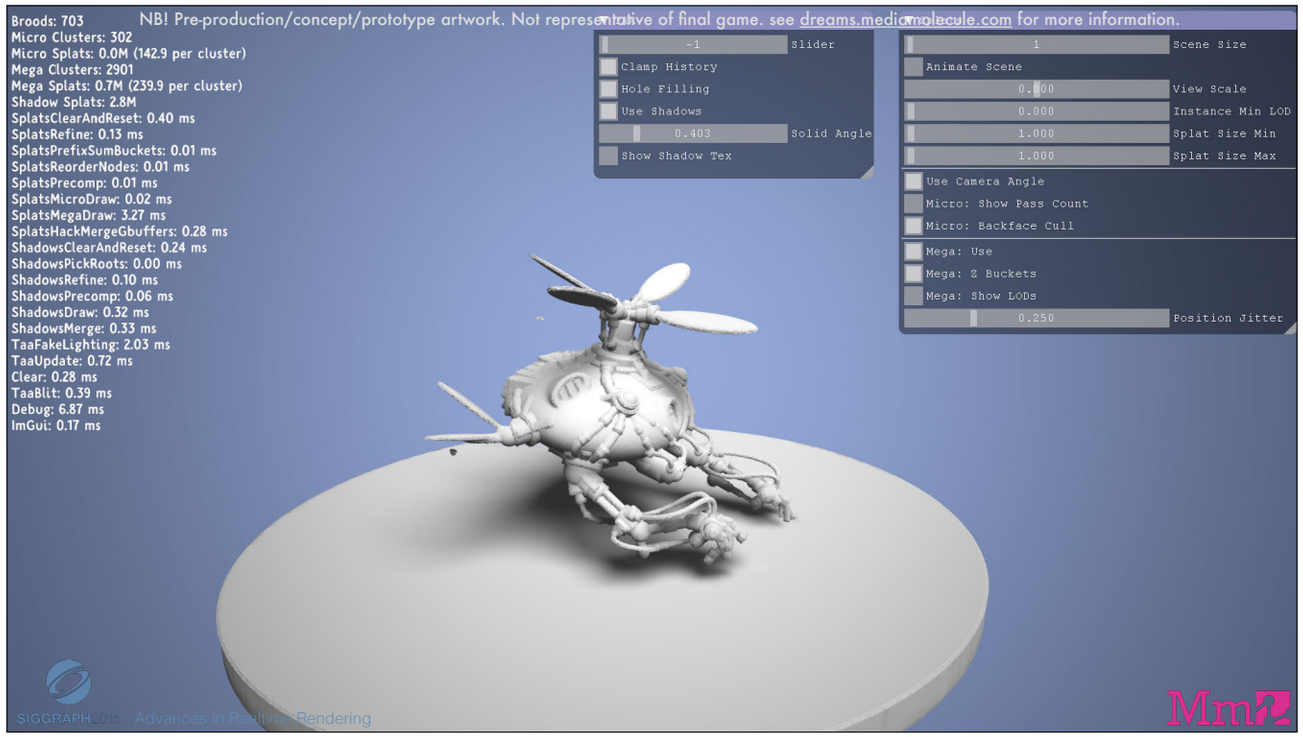
We’ve left it in for now as our only DOF.



I should at this point pause to give you a rough outline of the rendering pipe - it’s totally traditional and simple at the lighting end at least.
We start with 64 bit atomic min (== splat of single pixel point) for each point into 1080p buffer, using lots of subpixel jitter and stochastic alpha. There are a LOT of points to be atomic-min’d! (10s of millions per frame) Then convert that from z+id into traditional 1080 gbuffer, with normal, albedo, roughness, and z. then deferred light that as usual.
Then, hope that TAA can take all the noise away. ;)

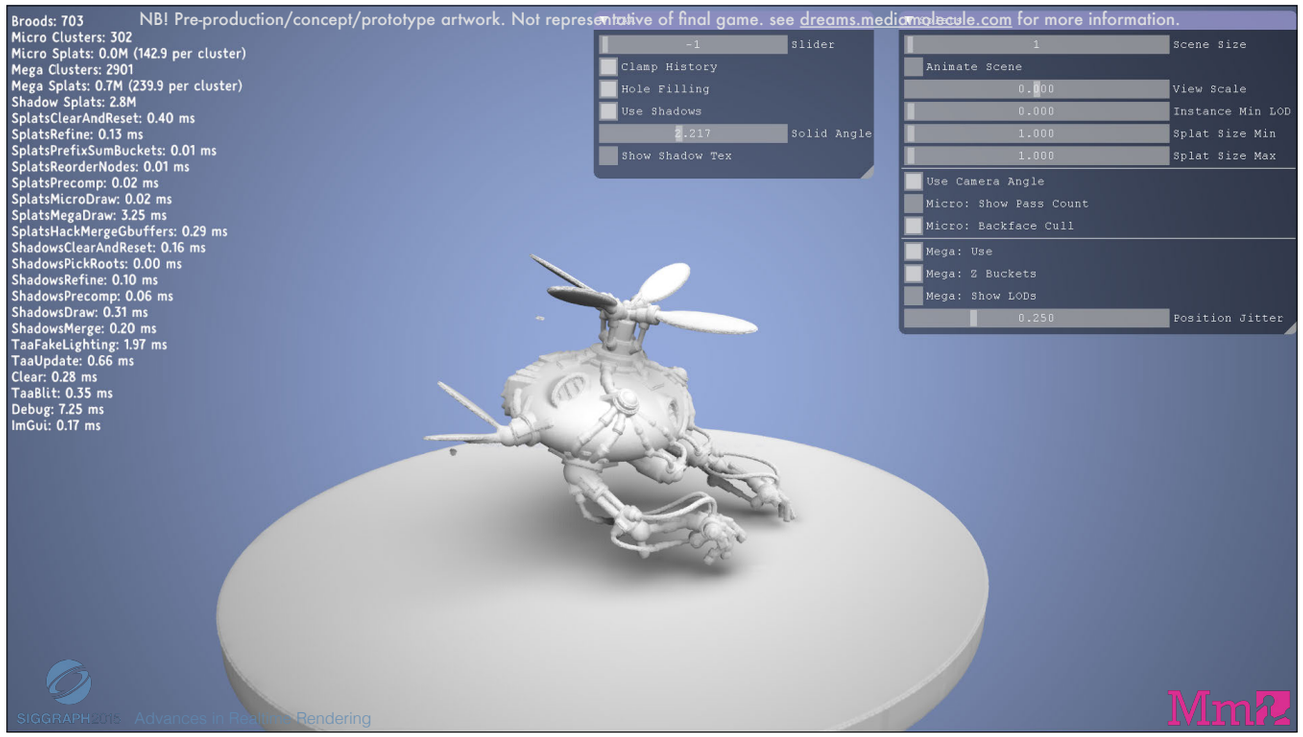


I’m not going to go into loads of detail about this, since I don’t have time, but actually for now the lighting is pretty vanilla - deferred shading, cascaded shadow map sun. there are a couple of things worth touching on though.



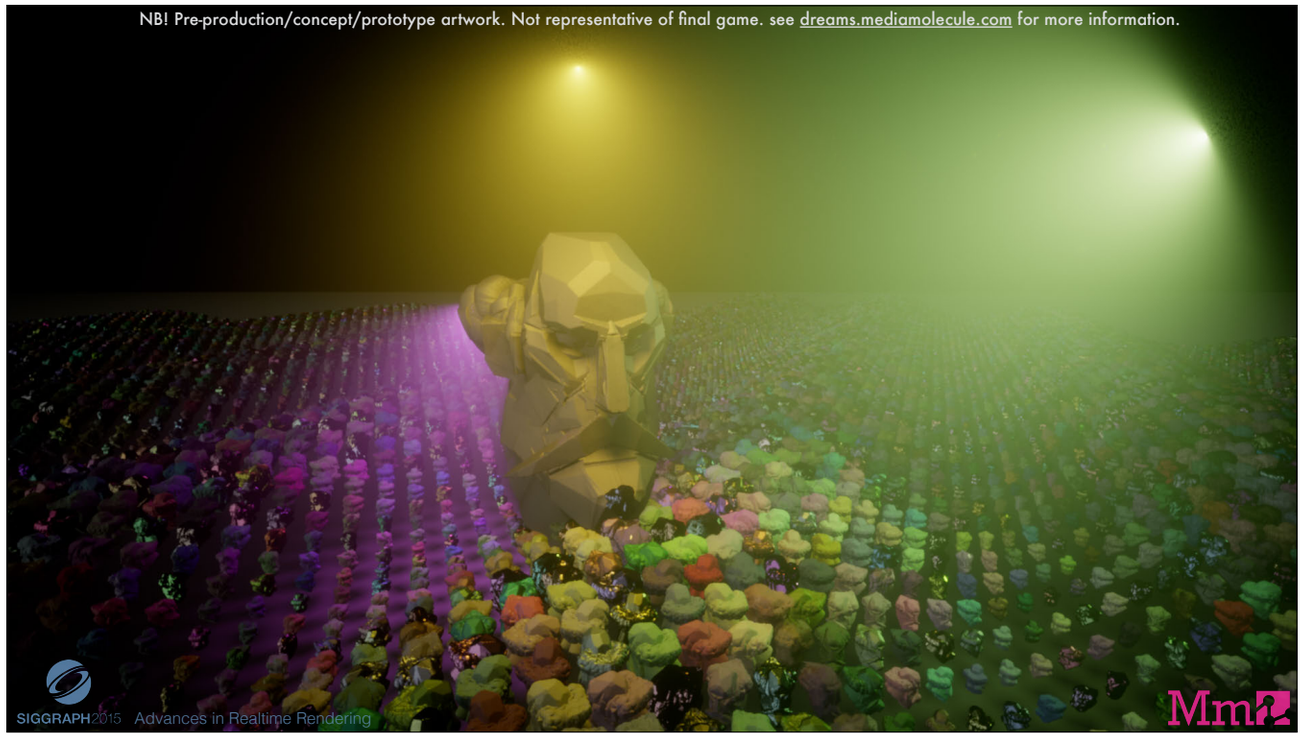
ISMs: Now we are in loads-of-points land, we did the obvious thing and moved to imperfect shadow maps. We have 4 (3?) cascades for a hero sun light, that we atomic-splat into and then sample pretty traditionally (however, we let the TAA sort out a LOT of the noise since we undersample and undersplat and generally do things quite poorly)

We have a budget of 64 small (128x128) shadowmaps, which we distribute over the local lights in the scene, most of which the artists are tuning as spotlights. They are brute force splatted and sampled, here were simonís first test, varying their distribution over an area light:

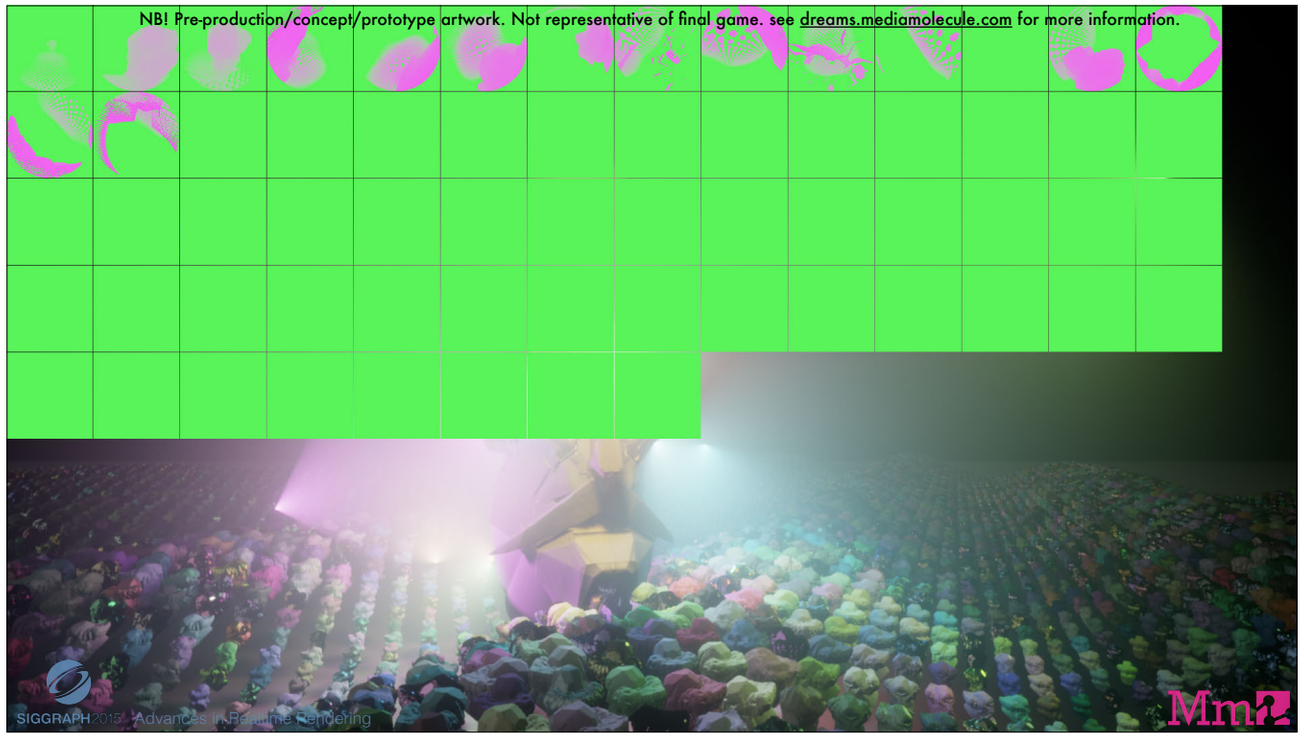


these images were from our first test of using 64 small ISM lights, inspired by the original ISM paper and the ‘ManyLODs’ paper. the 3 images show spreading a number of low quality lights out in an area above the object.
Imperfect Shadow Maps for Efficient Computation of Indirect Illumination
T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, J. Kautz
http://resources.mpi-inf.mpg.de/ImperfectShadowMaps/ISM.pdf

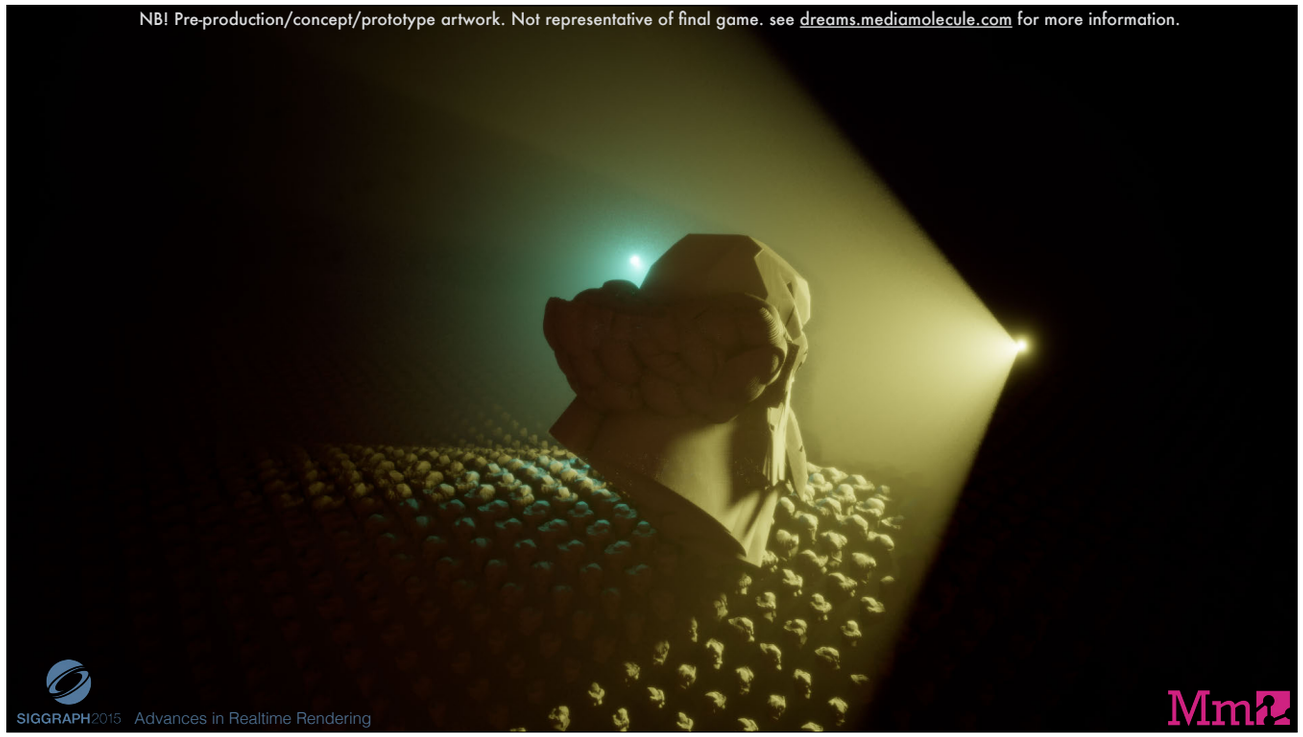
ManyLoDs http://perso.telecom-paristech.fr/~boubek/papers/ManyLoDs/
Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination
Matthias Holländer, Tobias Ritschel, Elmar Eisemann and Tamy Boubekeur



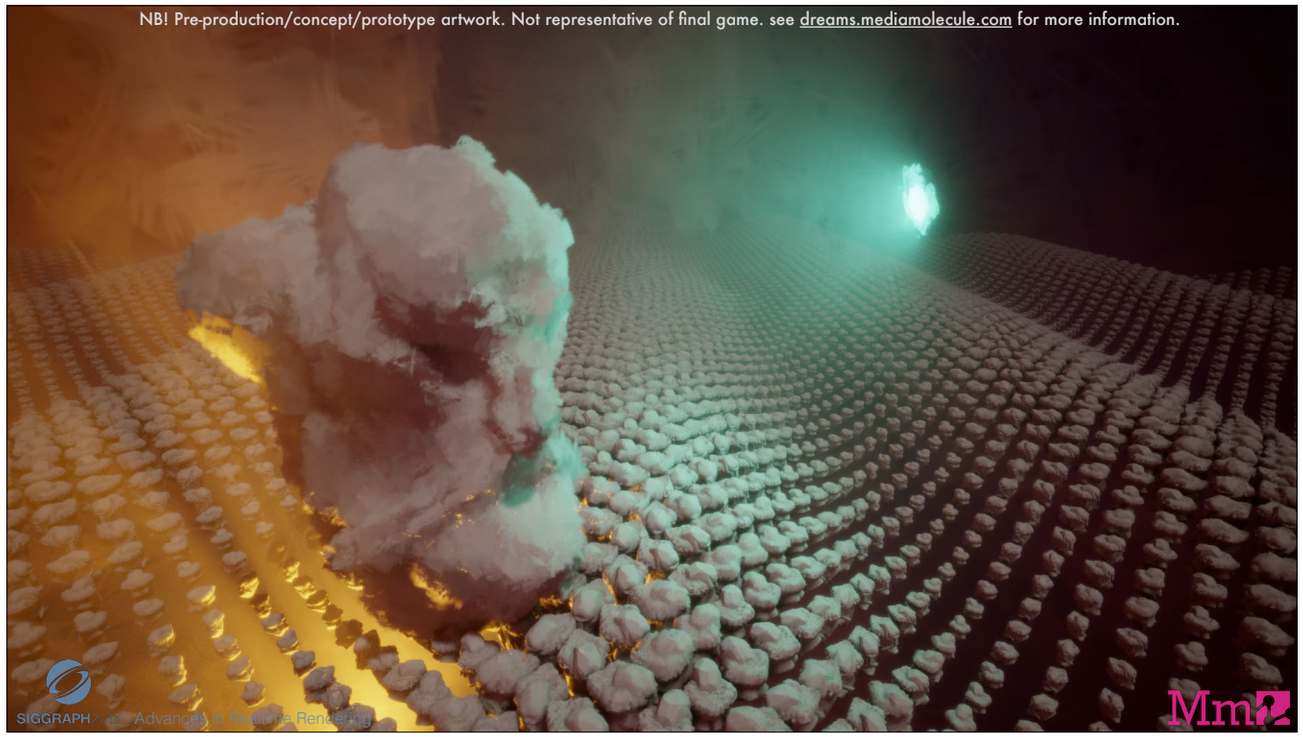
I threw in solid-angle esque equi-angular sampling of participating media for the small local lights. See https://www.shadertoy.com/view/Xdf3zB for example implementation. Just at 1080p with no culling and no speedups, just let TAA merge it. this one will DEFINITELY need some bilateral blur and be put into a separate layer, but for now itís not:



(just a visualisation classic paraboloid projection on the ISMs)
sorry for the quick programmer art, DEADLINES!



this ‘vanilla’ approach to lighting worked surprisingly well for both the ‘tight’ end… (single pixel splats, which we call microsplats)… as well as

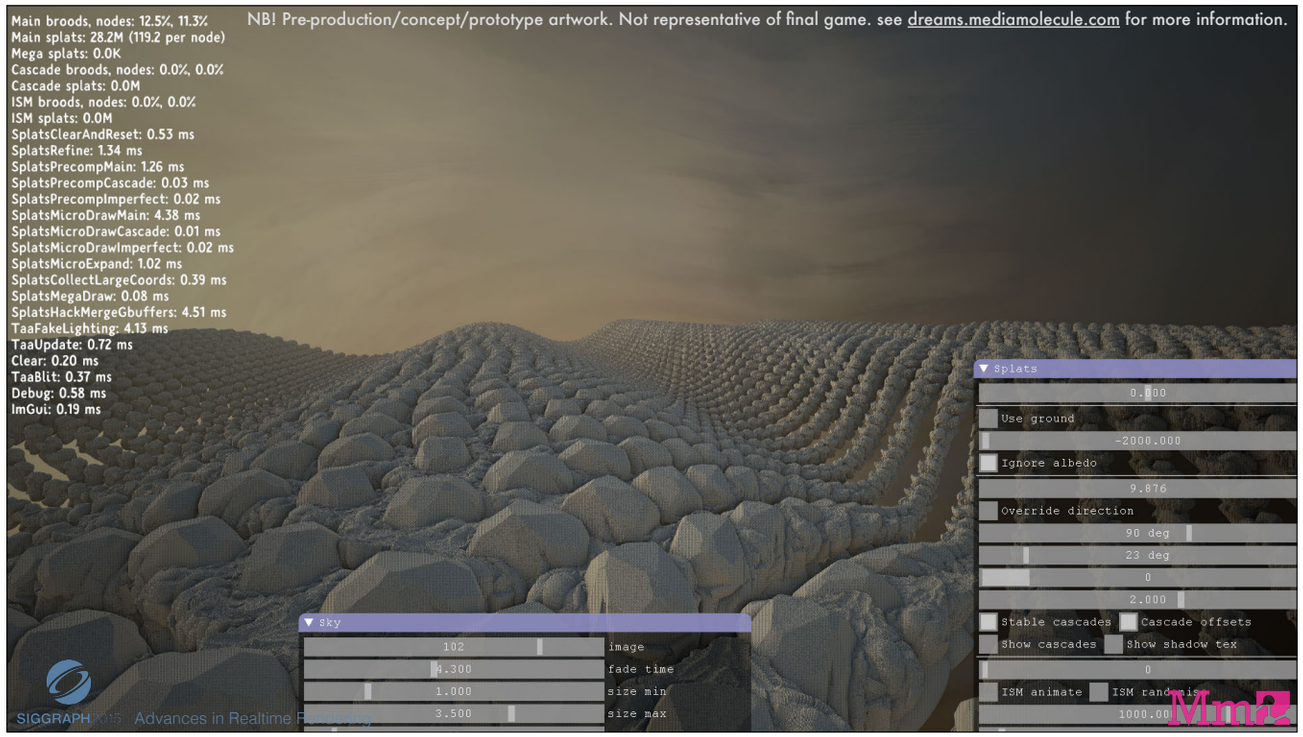


…the loose end (‘megasplats’).

this was the first time I got specular in the game! two layers of loose splats, the inner layer is tinted red to make it look like traditional oil underpainting. then the specular hi lights from the environment map give a real sense of painterly look. this was the first image I made where I was like ‘ooooh maybe this isn't going to fail!’

At this point you'll notice we have painterly sky boxes. I wanted to do all the environment lighting from this. I tried to resurrect my previous LPV tests, then I tried 'traditional' Kapalanyan style SH stuff, but it was all too muddy and didn't give me contact shadows nor did it give me 'dark under the desk' type occlusion range.

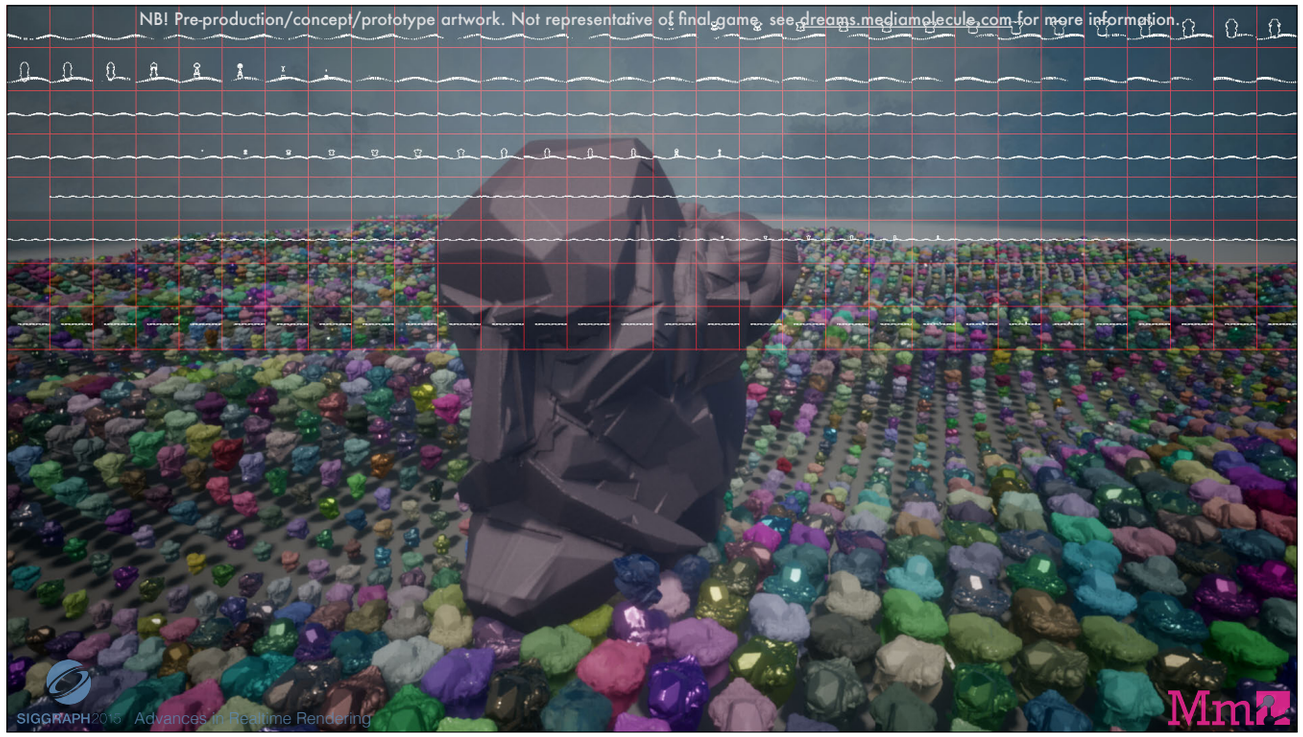
For a while we ran with SSAO only, which got us to here (point clouds give you opportunities to do ridiculous geometrical detail, lol)



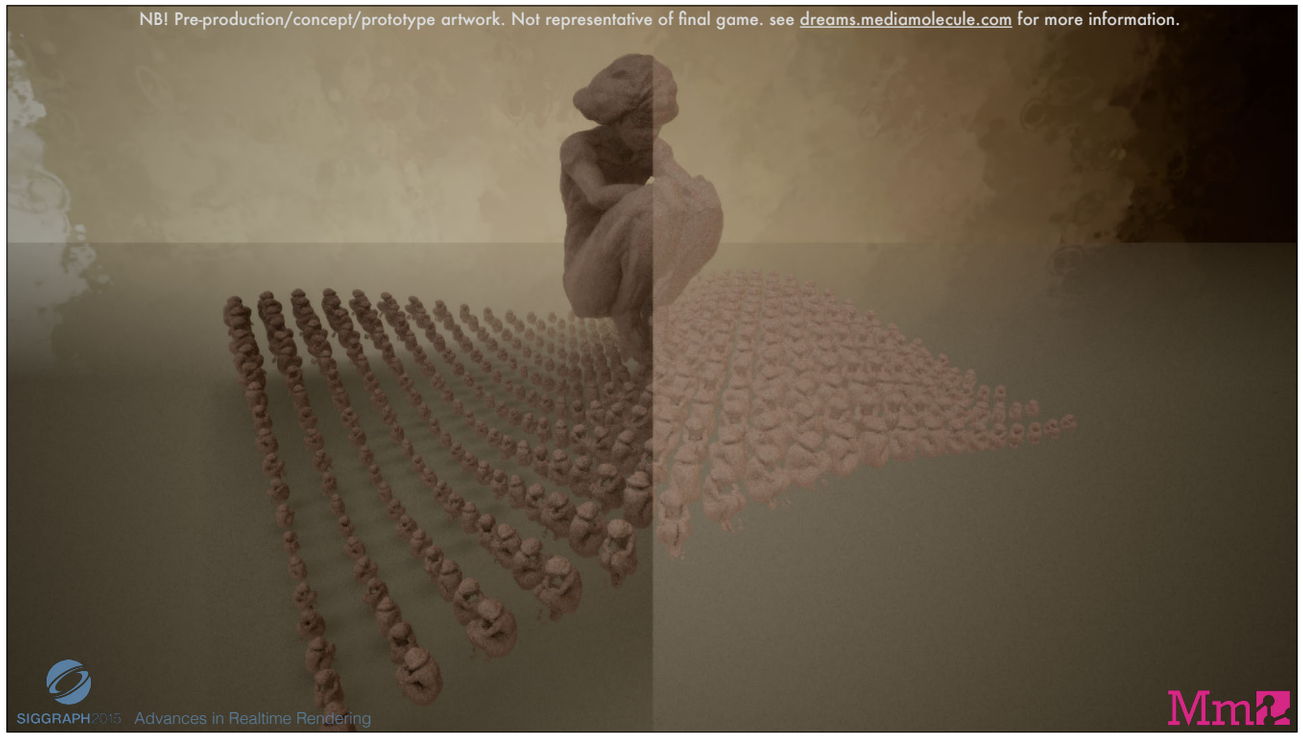
the SSAO we started with was based on Morgan McGuire's awesome alchemy spiral style SSAO, but then I tried just picking a random ray direction from the cosine weighted hemisphere above each point and tracing the z buffer, one ray per pixel (and let the TAA sort it out ;)) and that gave us more believable occlusion, less like dirt in the creases.

From there it was a trivially small step to output either black (occluded) or sky colour (from envmap) and then do a 4x4 stratified dither. here it is without TAA (above). However this is still just SSAO in the sense that the only occluder is the z buffer.

(random perf stat of the atomic_min splatter: this scene shows 28.2M point splats, which takes 4.38ms, so thats about 640 million single pixel splats per second)

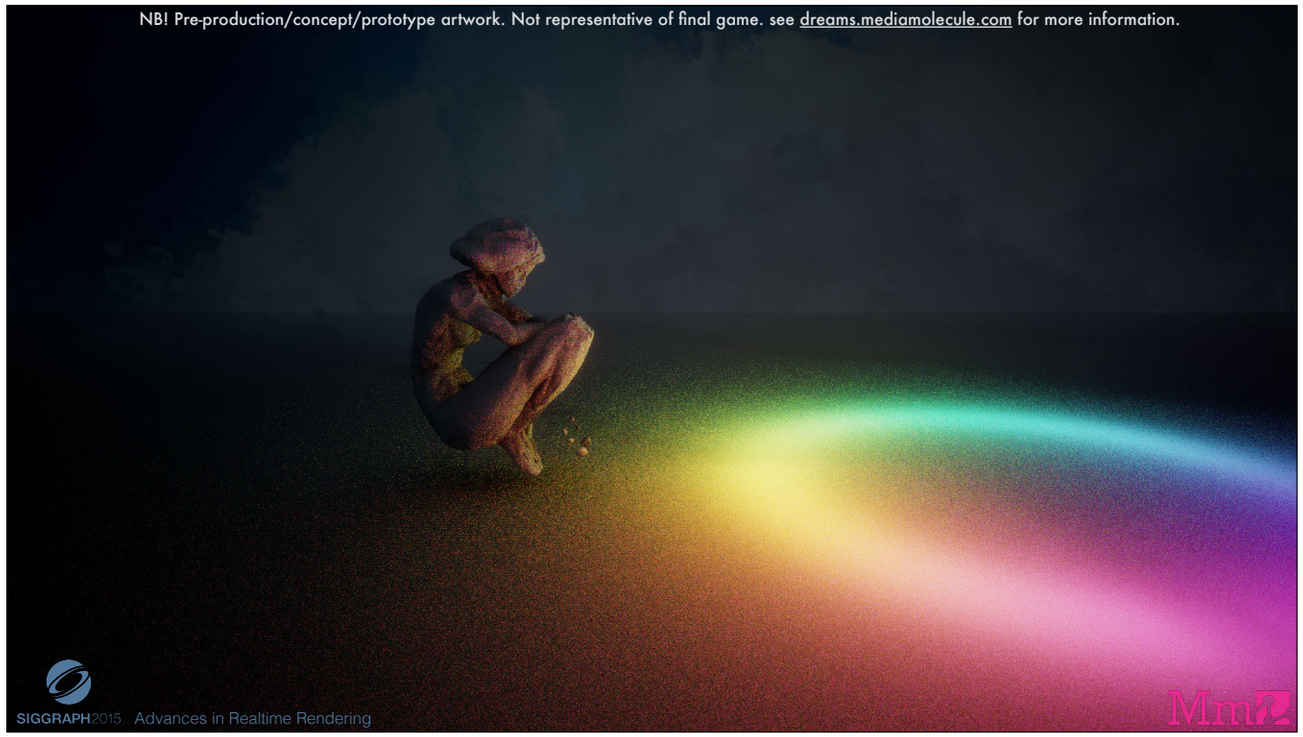


For longer range, I tried voxelizing the scene - since we have point clouds, it was fairly easy to generate a work list with LOD adapted to 4 world cascades, and atomic OR each voxel - (visualised here, you can see the world space slices in the overlay) into a 1 bit per voxel dense cascaded volume texture



then we hacked the AO shader to start with the z buffer, and then switch to the binary voxelization, moving through coarser and coarser cascades. it’s cone-tracing like, in that I force it to drop to lower cascades (and larger steps), but all the fuzziness is from stochastic sampling rather than prefiltered mip maps. The effect is great for mid range AO - on in the left half, off in the right.

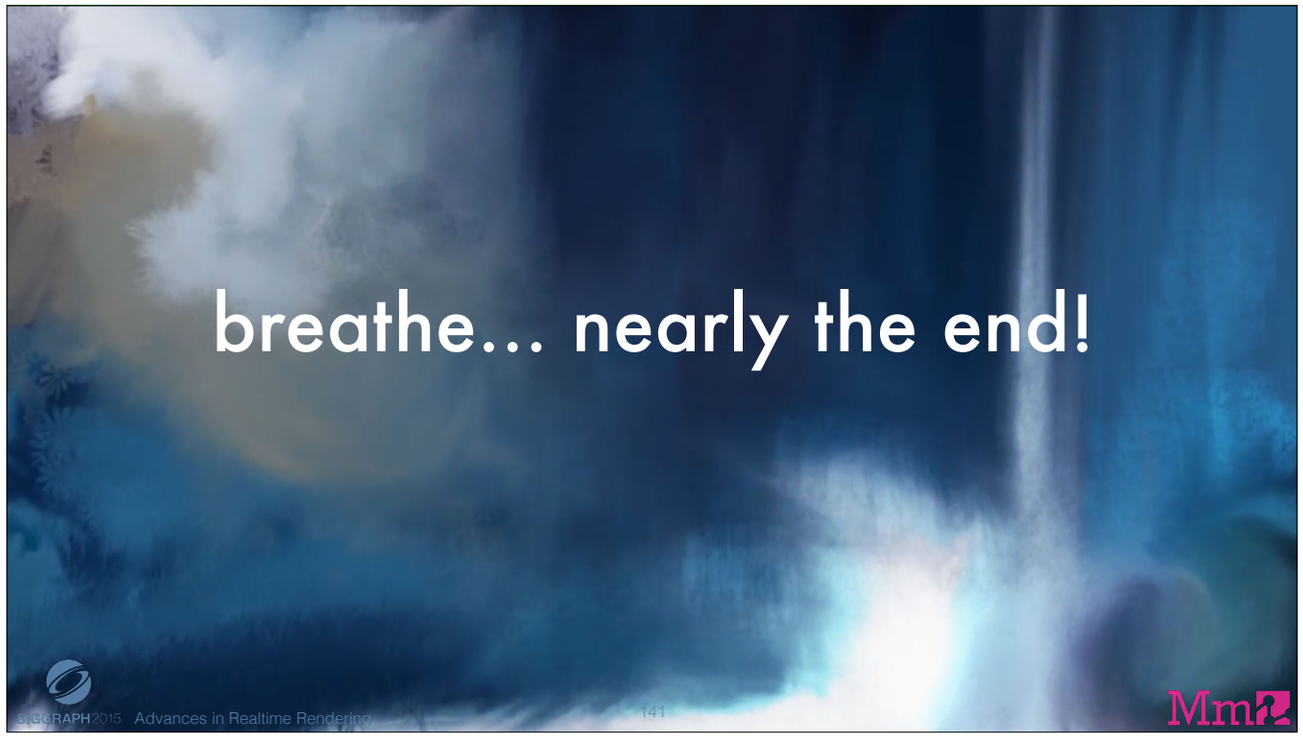
That gets us to more or less where we are today, rough and noisy as hell but extremely simple.I really like the fact you get relatively well defined directional occlusion , which LPV just can’t give you due to excessive diffusion.



(at this point we're in WIP land! like, 2015 time!)
The last test, was to try adding a low resolution world space cascade that is RGB emissive, and then gather light as the sky occlusion rays are marched. The variance is INSANELY high, so it isn't usable, and this screenshot is WITH taa doing some temporal averaging! But it looks pretty cool. It might be enough for bounce light (rather than direct light, as above), or for extremely large area sources. I don't know yet. I'm day dreaming about maybe making the emissive volume lower frequency (-> lower variance when gathered with such few samples) by smearing it around with LPV, or at least blurring it. but I haven't had a chance to investigate.

Oh wait I have! I just tried bilateral filtering and stratified sampling over 8x8 blocks, it does help a lot.
I think the general principle of z buffer for close, simple bitmask voxelization for further range gather occlusion is so simple that it’s worth a try in almost any engine. Our voxel cascades are IIRC 64^3, and the smallest cascade covers most of the scene, so they’re sort of mine-craft sized voxels or just smaller at the finest scale. (then blockier further out, for the coarser cascades). But the screenspace part captures occlusion nicely for smaller than voxel distances.

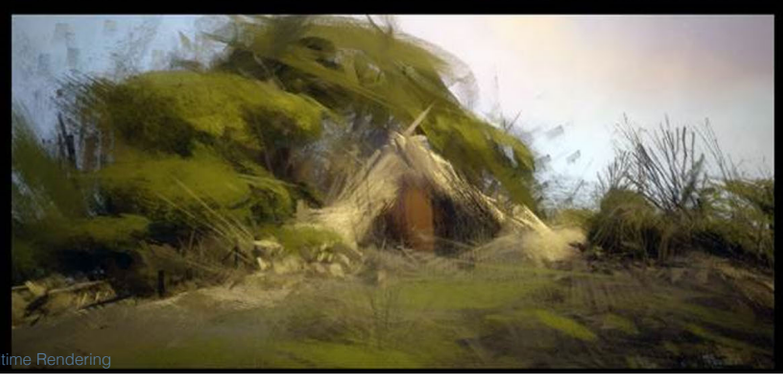
another bilateral test pic. WIP ;)



and that's pretty much where we are today!
as a palette cleanser, here’s some non-testbed, non-programmer art

NB! Pre-production/concept/prototype artwork. Not representative of final game. see dreams.mediamolecule.com for more information.

NB! Pre-production/concept/prototype artwork. Not representative of final game. see dreams.mediamolecule.com for more information.



It feels like we're still in the middle of it all; we still have active areas of R&D; and as you can see, many avenues didn’t pan out for this particular game. But I hope that you’ve found this journey to be inspiring in some small way. Go forth and render things in odd ways!

Thank you

- MM & SCE WWS for supporting this work and letting me share it
- Everyone at MM on the rendering and art side, especially Kareem (art director), Jon (a lot of whom art I used without permission ;)); Simon and Anton, the rendering team who did much of the code shown today. <3
- Natasha for inviting me (again)
- You for listening ;)

The artwork in this presentation is all the work of the brilliant art team at MediaMolecule. Kareem, Jon (E & B!), Francis, Radek to name the most prominent authors of the images in this deck. But thanks all of MM too! Dreams is the product of at least 25 fevered minds at this point.

And of course @sjb3d and @antonalog who did most of the engine implementation, especially of the bits that actually weren't thrown away :)

Any errors or omissions are entirely my own, with apologies.
if you have questions that fit in 140 chars I'll do my best to answer at @mmalex.