



Thank you for inviting me to speak with you tonight.

I am Brian Karis, Engineering Fellow at Epic Games.

Most of you I'm sure have seen some demo of Nanite before.



Maybe this one



Or this one



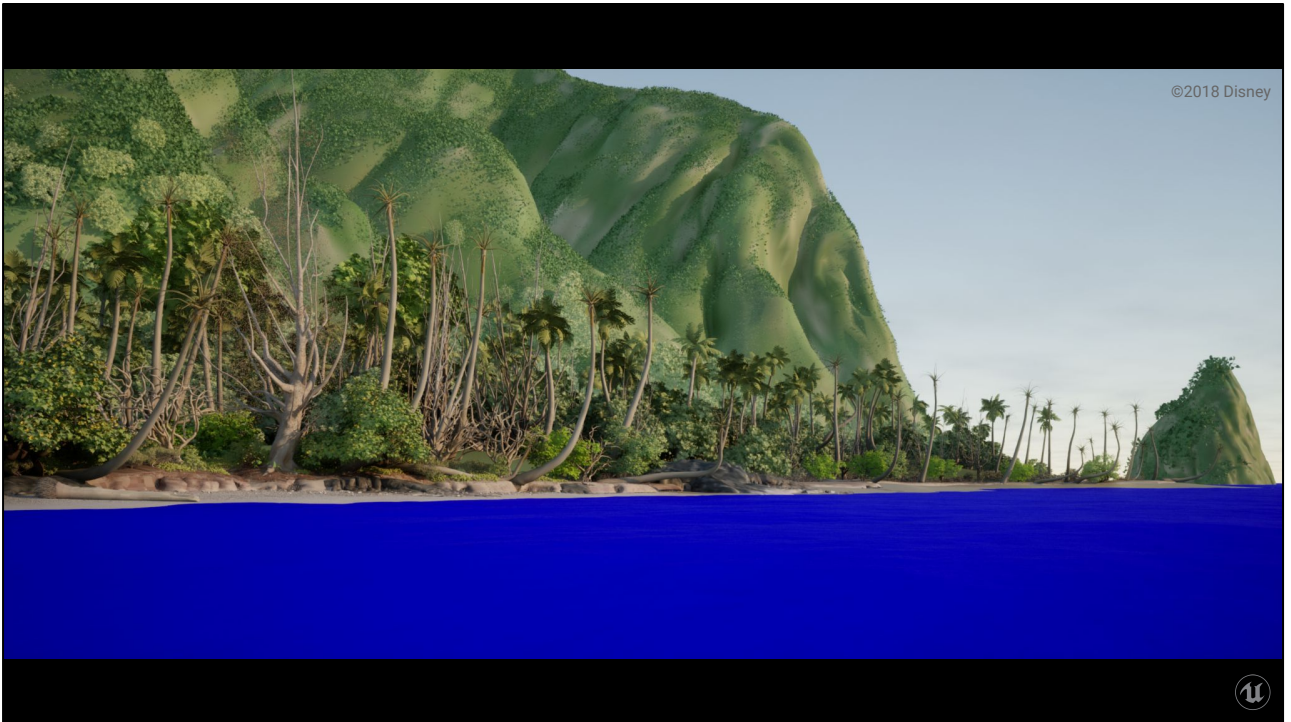
Or this



Or some of the countless things people have done with it since UE 5.0 came out.

I thought it'd be fun to show you something new.

Something you might appreciate more than the common audience.



We got the Moana island data set loaded in UE5 running with Nanite and Lumen.

This data set has become notorious for being massive and unwieldy and it sure is.

There are
146M unique triangles
27M instances

Resulting in
164B instanced triangles

Now I have to caveat this by saying Unreal doesn't support ptex so we haven't imported any of the proper materials.

This is all with vertex color only. The water looks especially silly being flat blue.

We also don't support curve primitives of which I'm told there are 10M of them. That means some of the trees look bare and there isn't any grass.

But all triangle geometry is present.

I've also relit it with a single directional light because our local lights couldn't handle this instance instance count at the moment.

We'll fix that. We've never seen instance counts anything like this before so we aren't really optimized for it.

=====

I unfortunately forgot to thank them live but I need to thank the following people for helping me get this scene to load in UE5

Anousack Kitisa

Danny Couture

Daniel Coelho



With that out of the way here is a video of me flying around the scene at consistently Over 40fps at 2742 x 1262 on an RTX 3080.
If we added hierarchical instance culling I think we could hit 60.

There is so much geometric detail here.

Grains of sand are modelled.

Twigs and needles.

All the leaves are geometry, no masked cards here.

The ocean water looks flat but that is a single 11M triangle mesh.

Under the water is detailed coral.

Look at this! 1000s of tiny protruding flutes covering the surface.

And each one of those is a separate instance!

Those grains of sand are instances!

Every leaf is an individual instance!

And finally here are the triangles.

Triangle dust indeed.

The **process** of invention



I thought it would be fun to show that but I'm not going to talk tonight about what Nanite does or how it works.

As a community we talk a lot about what we invented and how that invention works
We rarely talk about the **process** of inventing, the **process** of innovation.
Which is odd because that is the skill we practice.

The course of technology and human knowledge is built step by step on what came before it.

Progress requires understanding the state of the art and engaging with that community is why we publish our work.

But what bridges the gap between each invention is the inventor using their skill, making that progress.

Not all progress is equal though.
Some invention makes large leaps and is incredibly disruptive.
Those are what I'm **most** interested in.

Perhaps I can stand on the shoulders of giants in terms of their process as well.
What lead to large leaps in the past?
What differentiates good work from groundbreaking, evolution from revolution?

It's usually easy to identify impact after the fact. Look at the number of citations.

But I want to know **before** that and I want to do more than just identify it.
I want to know **how** groundbreaking work is done so that I can do it myself.
Are there common patterns or skills? Can it be learned or at least fostered, or is it pure luck?

I have found frustratingly little on this subject worth reading..
The best I've come across is Richard Hamming's lecture "You and Your Research" which I highly encourage you to watch or read a transcript of.
<https://www.cs.virginia.edu/~robins/YouAndYourResearch.html>

I could regurgitate bits of his talk to you but you'd be better off hearing it from him.
There is bit that I feel I can speak to though.

A question Hamming became famous for asking was "What are the most important problems in your field?"
Followed by "Why aren't you working on them?"

I'm not sure how impactful and disruptive Nanite will ultimately be. Time will tell.
What I **am** confident of though is the importance of **the problem it is trying to solve**.

So tonight I thought it would be worthwhile to tell the story of my journey in trying to solve it
and along the way tell you what I learned about both the problem and the process of invention of what ultimately became Nanite.

Understand your users

- What consumes their time?
 - Technical tasks that aren't artistic
 - UV layout, LODs, collision, profiling, optimization
- What do budgets mean to game development?
- Unknown context for assets
 - Many authors simultaneously
- Unknown final budget
 - Only predictions
- Conservative behavior to avoid future pain



Let's start at the source of it all.

Before creating a tool you must understand the people that will use it.

Most of us don't think of ourselves as tools programmers but that's ultimately what the majority of computer graphics is about.

Creating tools for artists to make art.

So we need to understand artists.

If your organization includes artists, talk to them. A lot. Become friends with artists.

Try to understand their process, what they spend their time doing, what was different between what they intended and what was actually created.

If you don't have artists in your organization, watch videos of artists working.

Watch instructional videos of how art software works.

Look at breakdowns for how they made what they did.

Many of you haven't worked in game production so it is worth explaining some of the problems artists deal with.

You might be surprised at how much of their time is sucked up in technical tasks that aren't artistic.

Laying out UVs, generating LODs, collision geometry, multiple versions of the same asset for different use cases, and then profiling and optimizing all of it to fit in budget. Optimization is familiar to everyone but in games they rule with an iron fist.

I'm not sure how I can relay how dominant of a thought budgets are to artists any better than to tell you one of the most popular forums for game artists is called polycount.com.

They named their website after the budget.

And that isn't the only budget they need to be concerned about. There are countless ones that dominate their every action.

Polycount, draw calls, texture memory, mesh memory, light count, shadow casting light count, shader instruction count.

These are just things that engineers can quantify and try and turn into a budget.

There are a 1000 more things an artist could do to make things run slower and they need to know those too.

To make it orders of magnitude harder there are many authors working simultaneously, often without the context for where their work is going to be used.

~~No one knows exactly where that chair is going to be used when it was created.~~

~~There might be two in a box room with nothing else or 1000s in a mountain of garbage at the city dump.~~

~~Likely both.~~

And that budget they have to fit in, say polycount? Where did that come from?

Some engineer like me had to pull a number out of their ass at the start of a project for brand new hardware that is faster than last time.

Ultimately the real budget is the framerate and the amount of RAM the machine has.

All this leads to conservative behavior to avoid future pain.

Going back and redoing things costs money and time.

In a production environment, money and time are just as much of a limiting factor on quality than rendering the pixels with the latest greatest rendering technique.

Anything that makes art more efficient, allows artistic vision to more directly be expressed, or enables more of the art team to contribute more widely because the process is less arcanelly technical, will reap massive dividends.

I argue there is no problem more important to work on in graphics right now than how to make high fidelity content cheaper to create.

Early years

- [Messiah characters](#)
- ROAM+Bezier patches
- Terrain system on a mesh



Let's rewind.

Back in 99 I was a senior in high school.

I was excited about graphics and knew that's what I wanted to do.

I read an article about Messiah's character rendering tech.

<https://www.gamedeveloper.com/programming/behind-the-scenes-of-messiah-s-character-animation-system>

The game hadn't even come out yet but this tessellation tech had been heavily hyped.

While I may not have understood all the technical details at the time, what definitely made an impression were the goals behind it.

It automatically scaled detail based on the capability of your machine.

The game would look even better on hardware that hadn't even come out yet.

There were no limits on the artist's models.

Instead of creating 3 versions of every character for different purposes the artists could create just the highest quality version.

When I started my very first 3d graphics hobby project I was inspired by Messiah to make my own adaptive tessellation tech.

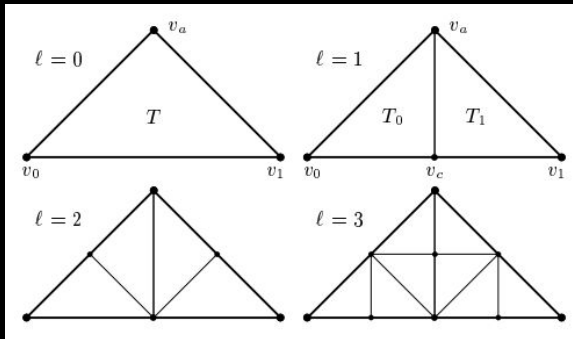
Yeah, from first triangle directly to view adaptive tessellation :)

I was going to use ROAM to adaptively tessellate a mesh of Bezier patches.

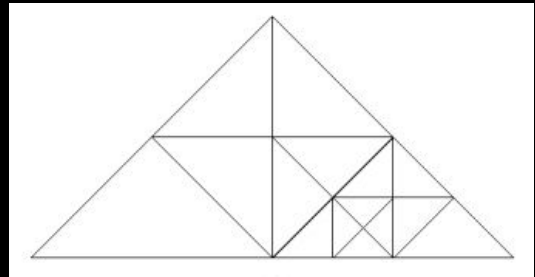
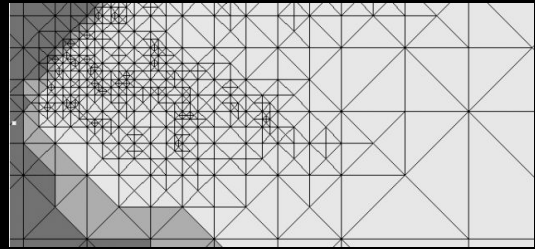
Terrain system on a mesh basically.

ROAM

- Right triangle hierarchy
- Longest edge bisection



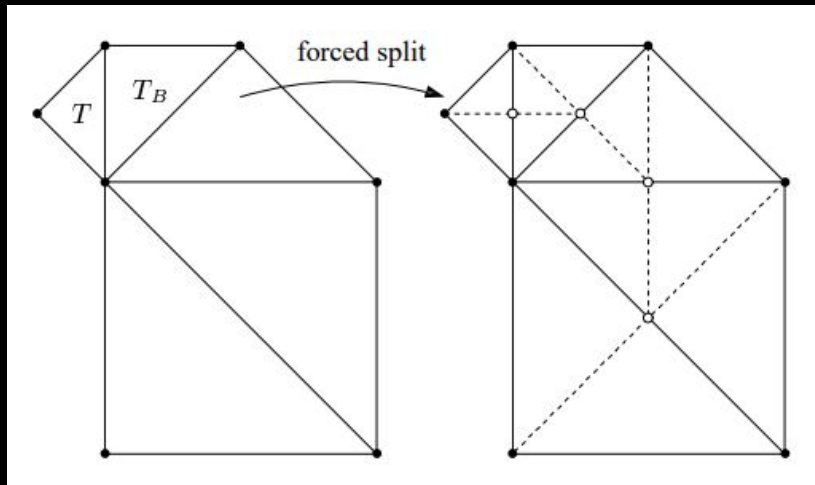
© 1997, IEEE



ROAM is what is known as a right triangle hierarchy.
Each triangle can split into 2 child triangles by bisecting its longest edge.

By refining the tree in some areas and not others it forms an adaptive tessellation.
Where to refine can be based on both geometric error and size on screen, meaning it can be view dependent.

ROAM



© 1997, IEEE



Any time an edge is split triangles on both sides of the edge must split in unison to keep the surface connected.

It is only valid to split the longest edge.

If the neighboring edge isn't the longest then it is forced to split until it is.

This can cause a chain of forced splits.

It isn't important to understand how my dumb code from high school worked.

I'm only explaining a bit of ROAM because it will come up again.

Using ROAM for Bezier patches was silly.

But what did I know? I was just starting.

The reason I'm bringing this up at all is to illustrate that this problem inspired me starting with the very first 3d graphics code I ever wrote.

I won't claim that persisted in my head ever since then. It didn't.

But I believe that perspective on abstracting art authoring from current capabilities stuck with me.

Texture budget

- Mip based texture streaming
- Texture size doesn't matter anymore!
 - $\text{NumMips} = 0.5 * \log_2 (\text{screen_area} / \text{area_in_uv_space})$
- Not quite
 - Ignores visibility
 - Ignores angle
 - Granularity too large



Fast forward many years and I'm a professional game developer with a bit of experience under my belt.

After shipping a game with no streaming and painful texture memory budgets.

I read a post by Tom Forsyth explaining the math behind deciding which mipmaps are needed for a particular view.

The texture's size cancels out in the equations which leads to an exciting albeit common sense conclusion.

The original size of the texture doesn't matter.

Quoting Tom:

"it was a pretty cool thing to go and tell the artists that there were no practical limits to texture sizes. They didn't really believe me at first ... Nevertheless, if you have a streaming system that only loads the mipmap levels you need, it is absolutely true."

<https://tomforsyth1000.github.io/blog.wiki.html#%5B%5BKnowing%20which%20mipmap%20levels%20are%20needed%5D%5D>

So we built a mip based texture streamer and I told our artists exactly that. They switched from 256 and smaller to 2k textures.

Umm... yeah.

Not quite.

Mip streaming certainly was a huge improvement but it didn't mean that texture size

didn't matter anymore.

That equation is correct but it is making some assumptions which aren't true in practice.

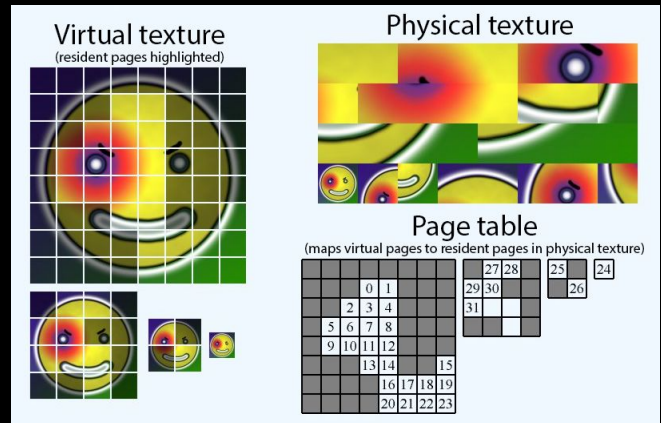
=====

It assumes the entire texture is visible on screen and that it isn't viewed at an angle. If those conditions are true then indeed the memory needed for textures scales exactly with screen resolution which is a beautiful desirable property.

But nearly everyone's mip based streamer ignores occlusion and angle and even if they didn't, they by definition work on a variable granularity, a mip. When those mips get huge it must decide all or nothing, meaning potentially bringing in a ton of data that isn't visible.

Virtual texturing

- RAGE and megatexture
- Amazing!
- Addresses those issues
 - With significant tech investment



Virtual texturing addresses those issues by working on fixed sized pages that are requested when they are sampled from.

After seeing the results that id had with megatexture I decided to write a virtual texturing system.

While it was far more effort to make shippable than I expected, it was worth it. Removing texture budgets for artists was transformative.

I could throw a bunch of numbers around about all the 4k textures we used, back in PS3 era.

But none of that can explain how freeing it was for the art team.

How much easier it made onboarding junior artists.

How much less wrangling it required from leads.

Megatexture seemed an insane gamble

- Foundational decisions dependent on unknown details
 - World surface area
 - Compression rate
 - Transcoding speed
 - Working set size
 - Working set volatility
 - Platform details finalized yet?
- Betting the whole studio on a long shot?
 - Unbelievable bravery



I didn't do megatexture though, as in the virtual texturing system I wrote was only meant to be a really nice texture streamer.

We weren't going to uniquely texture everything.

Even still there were bets with virtual texturing that I didn't yet understand how to make independently.

I only considered committing to ship a VT system an acceptable risk because id was committed to shipping a more aggressive version than I was.

That means it's possible.

But full megatexture seemed like an insane gamble from my point of view.

It required an art pipeline that was very different from before and couldn't be realized any other way.

The viability was based on numerous unknowns, each one an independent bet, all of which needed to succeed.

The risk seemed massive and going forward with it must have required tremendous courage.

I've gotten questions about Nanite that are similar in nature to these and I hope this talk can answer some of them.

Virtual geometry dream

- Virtualize geometry like we did textures
- No more budgets
 - Polycount
 - Draw calls
 - Memory
- Directly use film quality source art
- Just works
 - No manual optimization required
- No loss in quality



This is the same slide I've used multiple times for talks on Nanite.
It well summarizes the goal and the implications that I always had in mind.

The spark that started me seriously down this path was a comment from John Carmack in 2007.

He said that the hope for the next iteration of their engine was to virtualize geometry like they had textures.

He didn't explain how. Just that he had some pet ideas.

And with that I was off to the races.

I understood the impact this could have.

It was obviously one of the great problems of our domain.

And if we, as an industry, were really on the cusp of solving it I was determined to be on the forefront.

This is a classic story of inspiration coming simply from hearing something is possible
Even if it isn't yet!

Why is this such a common phenomenon?

I think it is because existence proofs give one a great deal of courage

And competition gives one drive.

Both courage and drive are extremely important in solving hard problems.

I'd argue they are absolutely vital when attempting to solve the most important problems.

To avoid giving the wrong impression, when I say I was determined to solve this, I don't mean that this is what I was working on at my job.

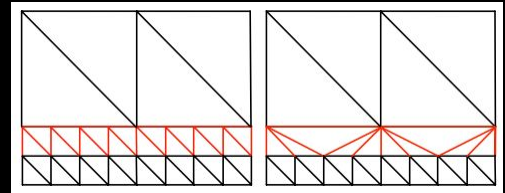
I did not have the luxury to do that. Hell I was still working on the virtual texturing system at this time.

This was a background thread, what I thought and read about in my free time, basically as a hobby.

This is a strange talk to give in that the majority of what I'm telling you about only existed in my head.

Virtualized geometry images

- Most obvious jump from virtual textures
- Instance fixed topology
- Restricted quadtree
- Border picks lower of the two resolutions



© 2007, ACM



The first major solution I zeroed in on was not very imaginative. I made the most obvious mental leap from virtual textures to virtual geometry. That being virtual geometry images.

Virtual texture tiles are replaced by instanced triangle grids.

Basically a terrain system on a mesh again, this time a quadtree.

That implies its a quad mesh which was already often the case with zbrush models of the time.

But the base cages aren't that low.

I'd like to reduce anything down really far and not have to rely on the artist modelling things in a particular way.

The point was for them **not** to worry about technical details like that.

=====

When neighboring quadtree nodes differ in resolution the border verts of the higher resolution node snap to the lower resolution grid.

Like ROAM this requires tracking neighbors which VT doesn't need to do.

Also like ROAM there is a requirement that the level difference between neighbors isn't too great or otherwise forced splits are required to continue to refine.

The reason is the amount of resolution the border can drop to can't be lower than the

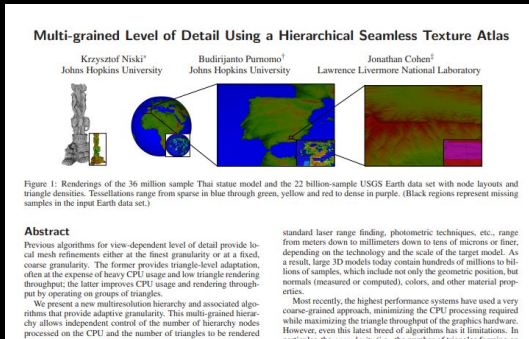
width of the node itself.

A quadtree that is restricted like this to have no more than a set difference in levels between neighboring nodes is called a restricted quadtree.

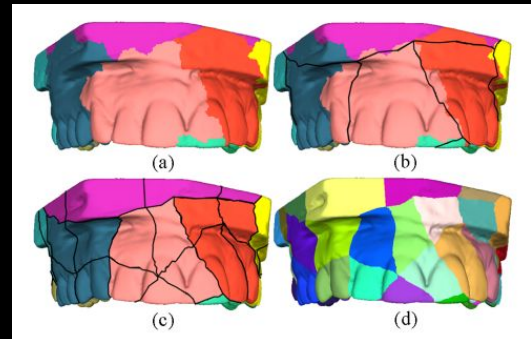
Interesting to note. The triangulation of a restricted quadtree with a 1 level difference constraint will generate the same mesh as ROAM.

Virtualized geometry images

- Based on seamless texturing



© 2007, ACM



© 2004, ACM



With a little bit of searching I found this paper which was a close match to my idea.

They formed a quad mesh through the act of parameterization with the goal of seamless texturing.

The idea is every texture chart generated by the parameterization can be topologically considered a polygon with each polygon edge being the boundary to a neighboring chart.

Any polymesh can be turned into a quad mesh through the same process Catmull-Clark subdivision uses.

This was definitely more flexible in that any mesh could be turned into a quad mesh with potentially less quads because the quad count was proportional to the number of texture charts.

This means the artists wouldn't have to know about this technical detail.

Virtualized geometry images

- Estimates in terms of VT tiles
 - # on disk
 - # per second read/transcode
 - # visible
 - # resident in memory

Saturday, January 10, 2009

Virtual Geometry Images

<https://graphicrants.blogspot.com/2009/01/virtual-geometry-images.html>



A year or so later I wrote the idea up in a blog post.

A great thing about virtual geometry images being such a direct extension to VT is that I could easily extrapolate all sorts of things.

Every stat about a VT system I could translate to geometry.

Sparse voxel octrees



In the meantime John had talked about one of those pet ideas. A year after his initial tease he explained his idea was ray casting sparse voxel octrees.

He had made a different leap than I had from 2D to 3D.

A few months later Jon Olick, who was at id at the time, did a talk at SIGGRAPH about this approach and showed a working prototype.

Voxels would store the color and normal data that used to be in 2d textures. It's a binary voxelization meaning voxels look like little cubes if you get close enough.

It's worth noting this was intended to be used for the geometric equivalent of megatextures.

Megageometry so to speak, where the geometric and color data was unique across the whole world.

While a few ideas were proposed for what could be done for animated geometry.

The recommended one was to ignore it.

Characters can stay simple triangle meshes.

Based on the prototype there were some assumptions that weren't acceptable for my goals

The memory was a bit concerning but the biggest issue was the performance.

It only appeared to be fast enough if you assumed that the majority of your frame

could be spent tracing primary rays.

No time was left for lighting and shadows.

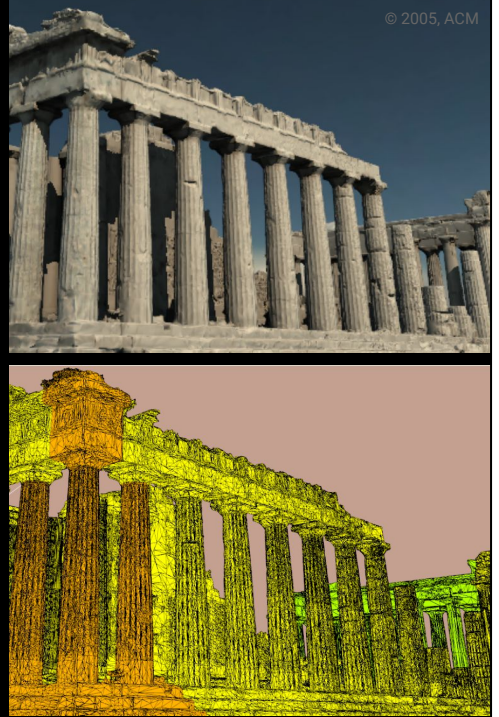
This makes sense relative to megatexture which baked lighting into the textures.

The same was true in this demo.

It was very impressive though and I for sure kept my eye on voxels going forward.

Progressive Buffers

- Next gen transition is time to try
 - Cross gen?
- Hierarchical level of detail (HLOD)



As next gen consoles got closer and we started discussing developing a next gen title I felt like my opportunity was arriving.

I might actually be able to work on this **for real**.

But there was also talk it might need to be cross gen.

I had no faith that triangles proportional to texels were going to be shippable on a ps3.

But there was something I had come across that maybe could scale down and that's progressive buffers.

This is the first example of what I'll refer to as hierarchical level of detail or HLOD. It isn't the first published but I was only aware of this one at the time.

HLOD is a hierarchy of triangle mesh chunks. Each level of the hierarchy is a simplified version of the last using typical mesh simplification algorithms. A view dependent cut of this hierarchy is rendered each frame.

Progressive Buffers

- Solves cracks through geomorphing
 - Everything morph band must be resident
 - Node must be smaller than band
- What happens when required data not resident?



To make sure level transitions don't cause cracks progressive buffers geomorph based on distance.

If every vertex knows where it will move to in the next simplified level it can make sure it does so before the next level is chosen.

That way there won't be cracks.

Because the geomorphing is based purely on distance there is an assumption that everything in a LOD band is loaded.

If a node at the desired level isn't present there are problems.

Progressive buffers are more like clipmaps than sparse virtual textures.

I want a sparse solution that adapts to visibility and where there's detail.

=====

This puts some constraints on both how the hierarchy is built and what must be resident at any one time.

A node can only geomorph to the next lower level. It thus can't span 2 transition zones at once.

That means all nodes must be spatially smaller than a LOD band.

This is guaranteed because the hierarchy is a sparse octree. A node's dimensions are dictated by its level.

Either that means the number of triangles per node has to be highly variable or it will draw and store more triangles than are necessary for flat regions.

Why don't voxels have cracks?

- Solid volumetric data
 - Not boundary representation

Thickness ≥ 1 voxel
Surface error < 1 voxel
⋮
Error $<$ thickness



I like the idea of HLOD but I don't like that crack solution.
Why don't voxels have cracks?

Sparse voxel subtrees can be built completely independently, without any knowledge of one another and still not have any risk of creating cracks if one subtree isn't loaded. Maybe if I understood why I could mimic it.

The reason is that voxels are volumetric and can be no thinner than the width of a voxel.

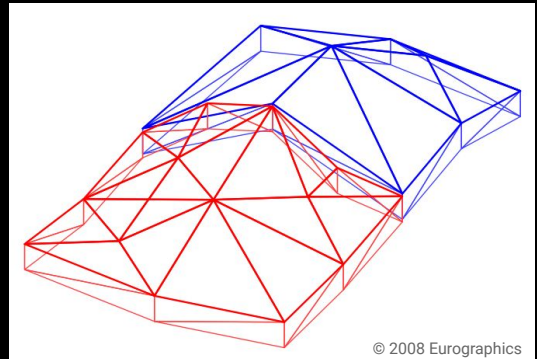
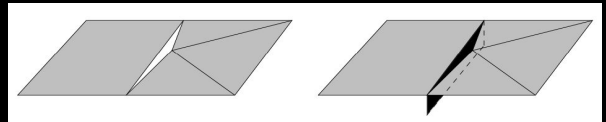
As resolution decreases voxels inherently get thicker.

Error is based on resolution and thus is less than the width of a voxel

Therefore there can't be a gap between a surface stored at 2 different resolutions because the error must be less than the thickness.

Skirts

- Treat mesh like solid volume too
- Thick enough to cover error
- Assumes volume which may not be true



© 2008 Eurographics



So to mimic that we need to give the mesh thickness at least as large as the error.

So we're borrowing another idea from terrain rendering.
You'll notice a theme.

The idea here is called skirts.

These are strips of triangles that go into the surface following the plane the volume was sliced.

They only drop far enough to cover a gap with their neighbor.

But there is a massive assumption here.

We have no guarantee that our meshes are manifold and can be operated on volumetrically.

In fact it is super common for game meshes to be open.

Also two sided.

Joined Epic!

- First week:
 - Should I build VT?
- More complex in UE
 - Shader graphs
 - No more fixed function
- “Nah, texture streaming isn’t really an issue”
 - Really!?



But I ultimately didn’t have a chance to pursue these ideas.
The project never happened and I joined Epic!

First big question when joining a company: what should I work on?
In the very first team meeting in my first week I suggested I build what had been the most impactful thing I’d done previously.
I could replace their mip based texture streamer with VT.

The team’s response shocked me.
Nah, texture streaming isn’t really an issue

Really!?

Works well.

Wat?!

Was their mip based streamer that much better than mine?
Did I come to a faulty conclusion because my code sucked?
It was probably the second largest investment in a single system I had done until that point. And it was a mistake?
I don’t think I’m particularly prone to imposter syndrome but it is really easy to fall in this trap joining a new team.

Ok, that’s fine. There are a ton of other things to work on where I think I can provide value.

Virtual geometry at Epic!

- Behind closed doors R&D for UE4
 - Targeted Larrabee
- Early UE4 rendering requirements:

- Movie quality AA
- "Unlimited" geometric detail
 - Implicitly also unlimited "texture" detail/ resolution
 - Geometric compression to deal with memory constraints
- Huge object counts
 - 100K+ objects in scene

(copied verbatim from wiki)

last modified on Feb 25, 2010



The next surprise.

Little did I know Epic had been researching virtual geometry too in the early days of UE4!

Specifically Andrew Scheidecker had been working on multi-resolution meshes. The research had been shelved a year or two before but some serious investment had happened.

Notice the same high level goals were listed

Multi-resolution meshes

- GIM wavelet encoded
- REYES style shading and micropoly
- Software rasterized
- MAPS based reduction



A sequence of progressive LODs, labeled by the number of triangles drawn.



Andrew's approach was fairly similar to my virtualized geometry image idea.

The geometry images were wavelet encoded, basically deltas from the previous mip. But he was more ambitious than I was, or at least expecting much more hardware capability than I was.

The plan was for this to be full REYES style rendering with micropoly shading and software rasterization.

Most interesting to me though was the approach taken to build the data. He was using MAPS.

MAPS

- MAPS based reduction

© 2013, Elsevier



The foundational concept behind MAPS comes from the field of topology.

You may have heard before that a coffee mug and a donut are topologically equivalent.

But we can say something stricter than that. A mug and a donut are homeomorphic.

Two surfaces are considered homeomorphic if there is a continuous invertible mapping between them.

That is exactly what we want to be able to displace one to the other.

MAPS is an algorithm for simplifying a mesh while maintaining the homeomorphic mapping of surface points on the simplified mesh back to the original.

With that information the simple mesh can be tessellated and displaced back to the original surface.

MAPS

- Genus problem
 - Worst mesh is a chain
 - MAPS can't reduce less than original genus



© 2015, Yasutoshi Mori



But that assumes that a very simple mesh exists that is homeomorphic to the complex one.

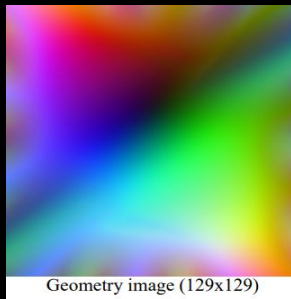
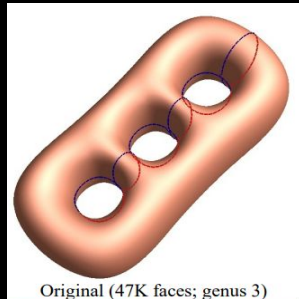
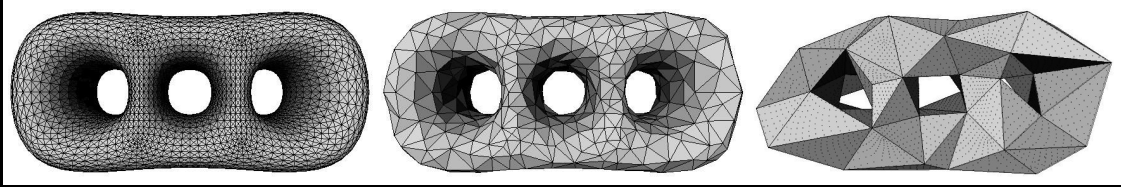
This is not true in all cases because the simple mesh must have the same genus as the complex one.

This chain is high genus.

There is a limit for how far we can simplify it without reducing the genus and breaking the ability to displace back to the original.

MAPS vs GIM

© 1998, ACM



© 2002, ACM



The mapping of geometry images doesn't have this problem because it removes a constraint.

The mapping they seek is a topological equivalence to a disk.

The boundaries of the image are free to match up fairly arbitrarily so the key thing to find is where to cut the 3d surface such that it can be unwrapped to a single 2d image..

This mapping is still invertible and is continuous in the direction of low to high. It is no longer continuous from high to low, because of those cuts.

Cuts are required for any mapping between a closed 3d surface and a 2d domain. MAPS can be continuous in both directions because it is mapping a 3d surface to another 3d surface.

Displacement fail

- Assumes surface is connected!



Genus is only the beginning of the problems

Both of these approaches choke if the surface isn't connected.

A continuous domain can't be mapped to a discontinuous domain without introducing discontinuities in the mapping.

If this chain is modelled how it actually is in the real world, where each link is an independent piece, we are screwed.

The overall lesson here is that:

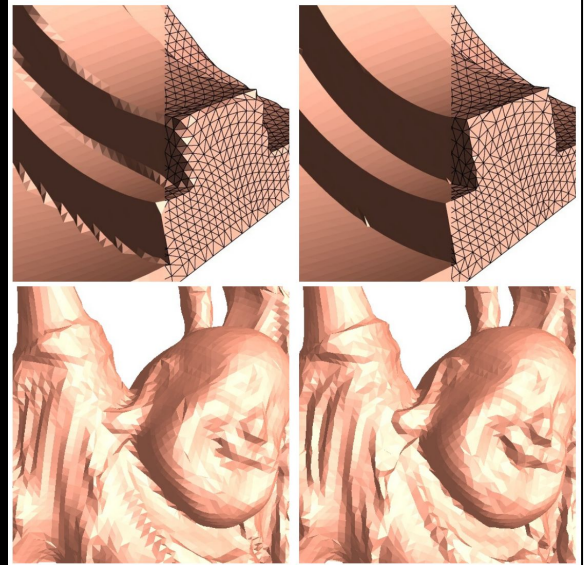
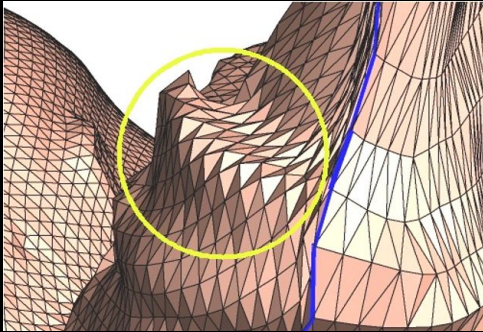
We cannot assume topology is simple, connected, and low genus.

Geometric detail can't be represented only as displacement in all situations, at all scales.

Regular remeshing

© 2002, ACM

- Asset in game isn't what I made!
 - Reinterpreted / resampled
 - Artists fuss over details



The second issue I heard more from the artists who had played with these prototypes. They really didn't like the fact that their art was being resampled and the details they fussed over were being lost.

No matter how close they got it never looked exactly like what they modelled.

This problem is basically aliasing and in the case of vector displacement it can be improved some like on the right but that only goes so far.

=====

Now criticize all you want about artists hyper focusing on little details and missing the forest for the trees.

Yeah, it happens

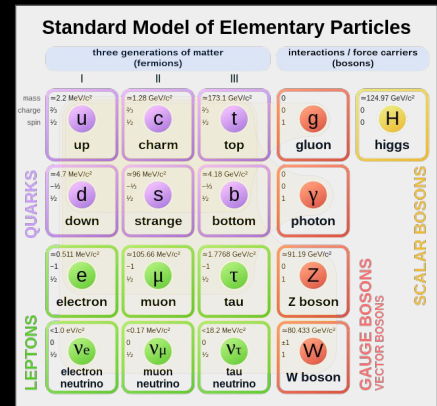
But you **want** artists that are detail oriented.

The difference between a quick sketch and a masterpiece are thousands of little details.

They need to be able to make them where they are important and understand clearly the exact effect of their changes.

Research irregular meshes

- Key problem is cracks
- Map the design space



With these insights in hand I decided to seriously explore irregular meshes and HLOD.

From my previous research the key problem is cracks.
So I would focus specifically on evaluating all potential solutions to that problem.

A friend of mine taught me an exercise to use when exploring a new design space.

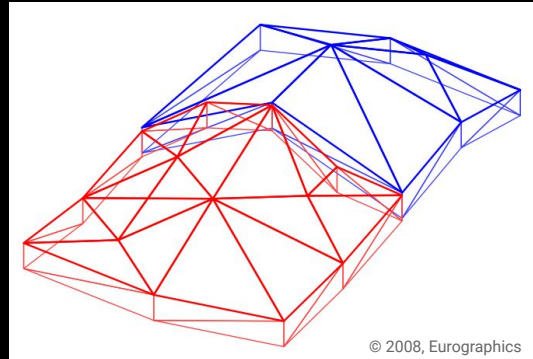
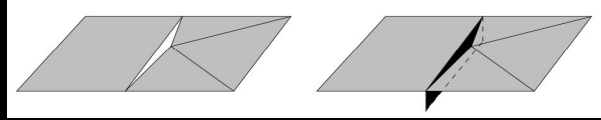
Visualize the multi-dimensional space of possibilities you are trying to optimize.
The dimensions may be discrete or continuous but the point is to identify them and explore their limits.

Even if the manifestation of some option isn't apparent to you yet, the relationship between those that are may point to a hole where something must be.

Think of how the standard model predicted many not yet discovered particles because there were holes in the map for a particular combination of properties.

Skirts

- Least knowledge of neighbor
- Spatial constraint
- Constrained to a face of clipping volume
- Not useful on arbitrary boundaries



Let's pick an abstract dimension to start with and explore its limits.

What's the least knowledge neighboring clusters of triangles must have of one another and still guarantee no cracks?

Just a spatial constraint.

We trade information for constraints.

If the geometry expresses a volume, the constraint can be a plane cutting into the volume.

This is what skirts are that we covered earlier.

Can we use skirts at any boundary?

Unfortunately no.

The constraint is a plane, or more specifically a planar face of a clipping volume that divides this cluster from the others.

The volumes don't need to be boxes so we could use a general tetrahedralization to divide space.

But we can't just use any boundary edges that zigzag.

The virtual cutting faces would only be as wide as the edges themselves

Which is effectively the same as locking them, making the skirts pointless.

=====

While writing these slides it occurred to me that a 1 level difference with the constraint that every other vertex is removed could potentially work with arbitrary zigzag boundaries.

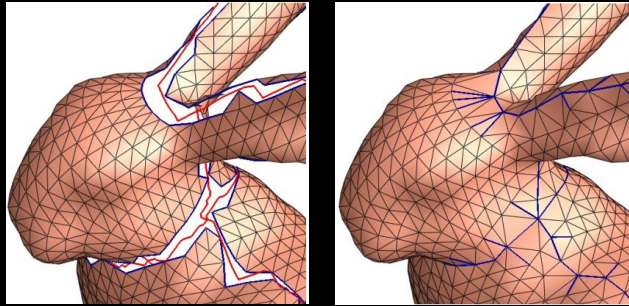
Every 3 points form a plane. The middle point collapses to on the end points does obey the constraint of keeping to the plane.

That goes in both directions.

This places a 1 level difference restriction on neighbors and the every other collapse constraint is pretty harsh but this might work.

Stitching

- Stitch mismatched boundaries at run-time
 - Whenever LOD changes



© 2003, Eurographics



What do we mean by knowledge of neighbors?

The absolute least amount of knowledge at **build** time is none.

Ignore completely what your neighbor is doing and move the boundary vertices around however you like.

But then we need to do really heavy work at run-time to stitch boundaries which don't match

That's basically the most knowledge required of neighbors at **run-time**.

While on opposite ends of this dimension both skirts and stitching could also be categorized as approaches where the boundary edges don't match.

Stitching seals the gap at run time.

Skirts seal the gap at build time.

I believe those are the only two geometric options for allowing the boundary edges of the precomputed triangle clusters to not match.

That means all other approaches must create matching boundaries.

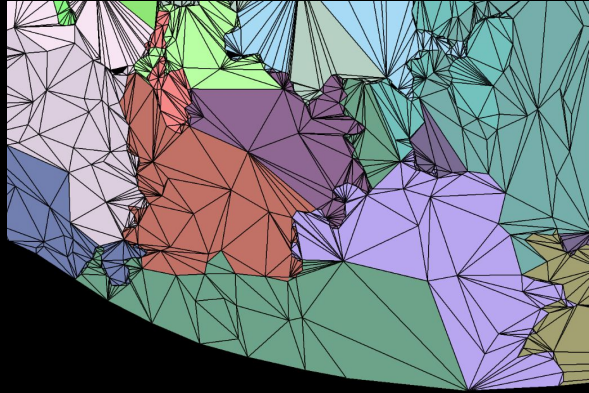
=====

Screen space hole filling is another approach which I don't cover.

It has similar issues to hole filling for points but boundary edges that should match could be potentially tagged to say so.

Locked boundaries

- Match higher detail nodes

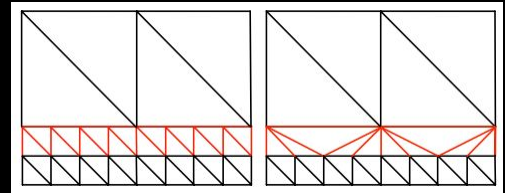


The most obvious way to make the boundaries match is to never change them. This is least information, highest constraint solution to matching boundaries. It is garbage though and fails at the heavily reduced levels once you run out of non-boundary edges to collapse.

Worth noticing that the matching is going up in detail.

Reduce boundaries to match

- Match lower detail nodes
- Need to know neighbors level
- Boundary vertex data
 - Lower detail position
 - Neighbor node we border
- Node data
 - List of neighbors
- Restricted tree



© 2007, ACM



How about the opposite direction?

If the resolution of the neighbor is known we could **reduce** our boundary to match it.

It is the approach taken by
Geometry image quadtrees and progressive buffers.

Like progressive buffers, boundary verts store the position where they move to in the next coarser level.

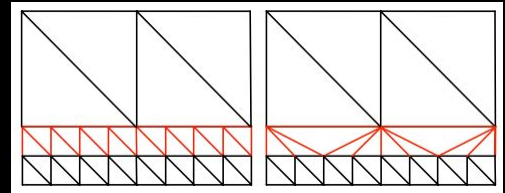
They also need to store which neighbor they border.

At run-time a node needs to know each of its neighbors level.

And the tree must be restricted such that there isn't more than 1 level difference between neighbors.

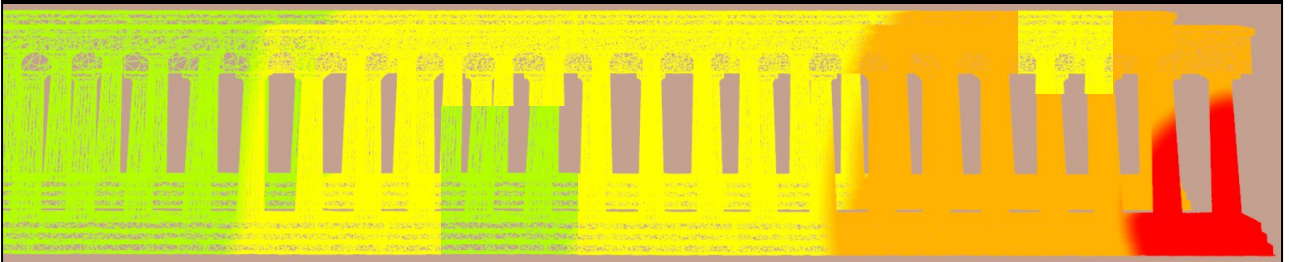
Reduce boundaries to match

- Match lower detail nodes
- Need to know neighbors level
- Boundary vertex data
 - Lower detail position
 - Neighbor node we border
- Node data
 - List of neighbors
- Restricted tree



© 2007, ACM

© 2005, ACM



We can make a trade here for less information and more constraints.

If the tree was a spatial subdivision, say an octree, then some of this data could be implied.

The neighbors of an octree node are implicit and don't need to be stored.

The verts on the boundary of an octree node can be implied from their position.

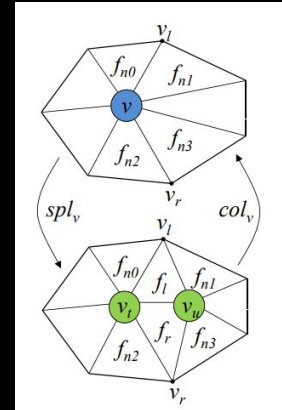
Which neighbor those verts need to match can also be implied from which face they touch.

I said progressive buffers were like a mesh version of clipmaps.

They don't need to be. They can be made sparse by adding a clamp to the geomorph to be no more detailed than their neighbor.

Directly index neighboring vertices

- View dependent progressive meshes
- Granularity too fine



© 2010, IEEE



So far we've been assuming that the data of a node in the hierarchy is independent from all others.

For boundaries to match means they have duplicate vertices as their neighbor. Seems like the approach that requires the most information is one that directly indexes the neighbor's vertices.

It's a bit of a stretch but I think the closest are probably the view dependent progressive mesh approaches.

They record and play back the finest granularity of edge collapses and splits, just like progressive meshes do.

But they handle being applied out of order or incomplete to account for view dependence.

By applying and removing these operations we change the mesh just like the simplifier would have done offline.

At no point **can** there be cracks.

They operate at a very fine granularity though, making decisions per vertex is excessive.

Comparing to VT this would be like streaming texture data at the granularity of texels. That's not to say these collapse and split operations couldn't be batched together.

Resident != rendered



But there is a trap we've fallen into that is easy to do.

That is equating the data that is resident in memory in a renderable form with the data that will be rendered..

Work to address cracks only needs to be done when level of detail changes.

That is technically true.

That is not the same thing as when data is streamed in.

There may be many instances of a model in the scene.

Not all of those should render the highest detail resident in memory.

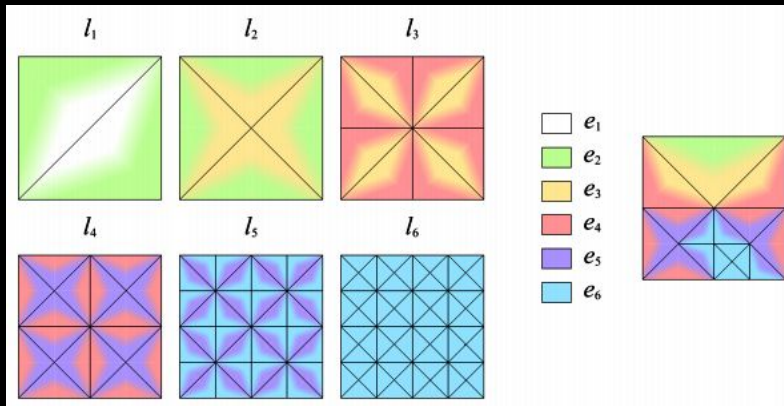
While these progressive mesh approaches don't require the update operations to be at streaming time they do require a persistent version of the mesh stored at the last seen detail level.

Each frame it will be updated based on how the view has changed.

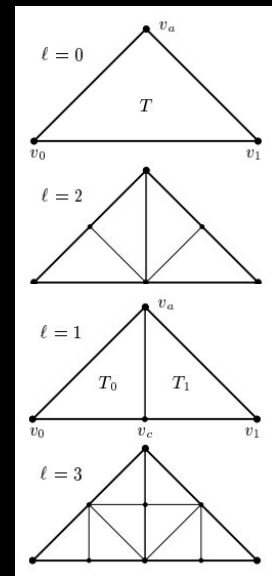
That kills instancing and is simply too much memory.

ROAM like

- BDAM is a right triangle HLOD



© 2003, Eurographics



© 1997, IEEE



We haven't encountered a method for handling cracks like ROAM yet.
Why does that work?

The trick is that not all sides of a triangle split at the same time.
Only the longest edge does and which edge is longest at each level must cycle.

The 2 right triangles that share their longest edge are dependent on one another.
Each can only refine if the other does as well, in unison.

For this level transition the shared boundary between them is essentially internal
because they act as one.

Anything internal can be modified without danger of causing a crack.

We can apply this same concept at a larger granularity and make triangular shaped
clusters.

That is exactly what Batched Dynamic Adaptive Meshes does.

=====

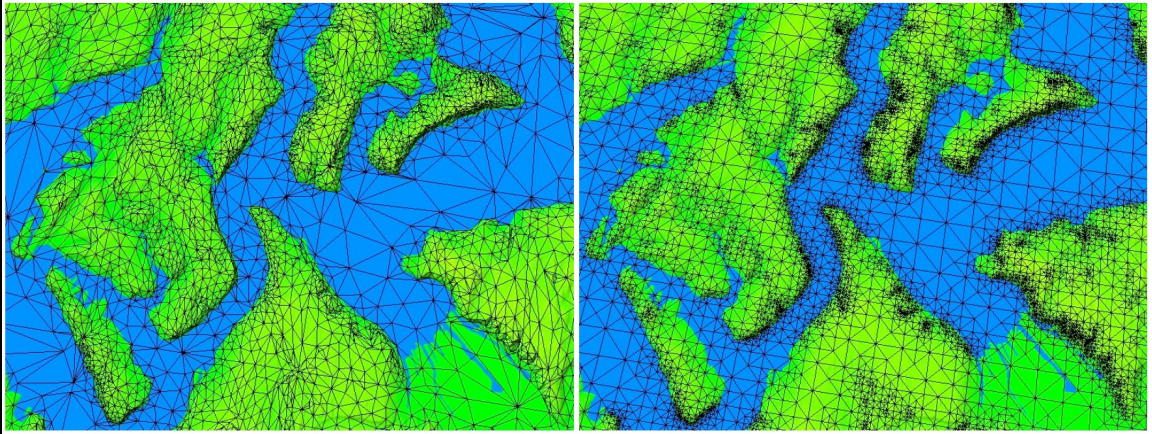
In this diagram think of the color as representing triangle density.

More density can be introduced interior to the triangle as well as along the dependent
longest edge.

A valid tree is guaranteed to have all the colors match.

This is simple to prove. The colors match the length of the edge so they have to match.

BDAM



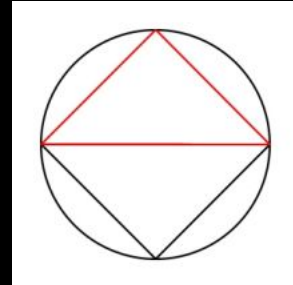
© 2003, Eurographics



Comparing to ROAM, we can also see the power of irregular meshes to achieve lower error with the same number of triangles.

No communication

- Dependent nodes must make same LOD decision
 - How? Communicate? No!
 - Same input => same output



© 2003, Eurographics



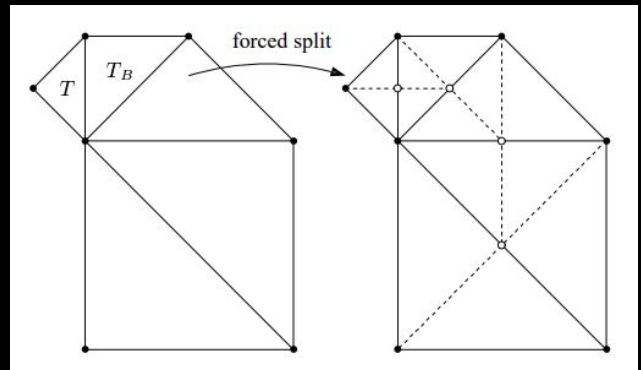
So far we've seen dependent nodes need communication such that they act in unison.

That's unnecessary.

So long as they use the same data to decide they will decide the same thing.

No communication

- Forced splits required communication
 - Not parallel



© 1997, IEEE



BDAM uses another neat trick to avoid communication.

Remember that the forced splits might be necessary in ROAM and that they could recursively chain.

Why did one branch of the tree refine further than the other to get into this situation?

The reason is the ancestors on that branch of the tree might not have reported as large of error.

So they stopped refining once the error was good enough.

Only when it gets to this level is the error known and it would want to split.

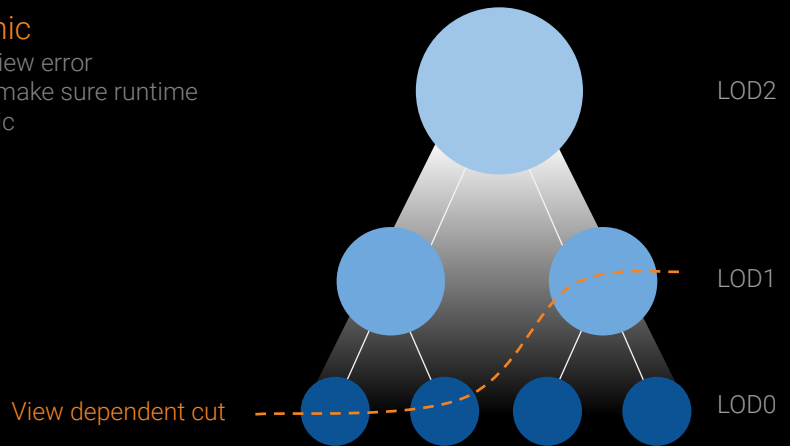
We can just tell this branch of the tree that there is higher error at a deeper level during the build.

=====

I believe this can be applied to any restricted tree.

No communication

- Force error to be **monotonic**
 - Parent view error \geq child view error
 - Careful implementation to make sure runtime correction is also monotonic



What this is doing is enforcing that the error function is monotonic.

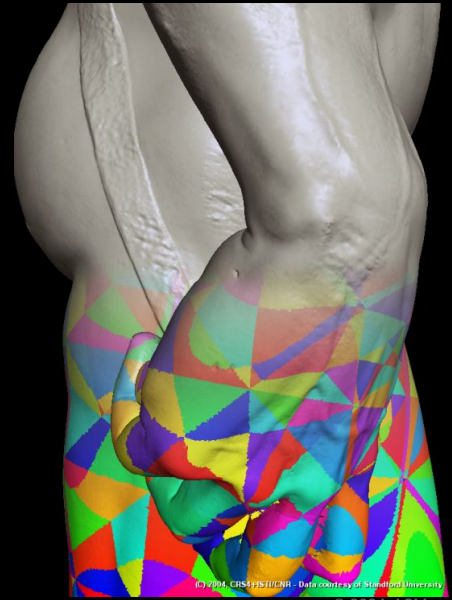
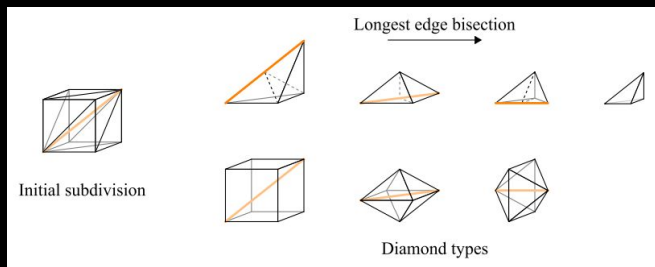
These two things together mean that no communication is required between nodes. That makes it easy to evaluate in parallel so should be easy to port to GPU.

=====

For more information see my SIGGRAPH 2021 talk
<http://advances.realtimerendering.com/s2021/index.html>

Tetrapuzzles

- 3D version of right triangle hierarchy
- All tetrahedron that share longest edge are dependent
 - “Diamonds”



So far that is only good for terrain.

Is there a 3d analog of right triangle hierarchies?

Yes, tetrahedron hierarchies called tetrapuzzles with longest edge bisection

All tetrahedron that share the common longest edge form a diamond and are dependent on one another.

Just like before these dependent nodes must refine in unison.

That means any edges on the boundary between these nodes are free to move.

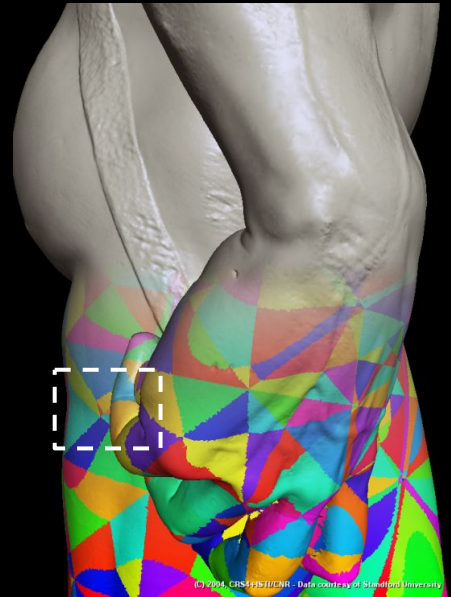
Unlike before the shape of the “diamond” is not always the same. There are 3 different types that are cycled between.

Regardless they are always formed by the tetrahedrons that share the same longest edge.

Tetrapuzzles

- Fixed spatial partitioning results in:
 - Surfaces slightly stick into partition
 - Uneven NumTris per cluster
 - Surfaces wave in and out of partition
 - Non-contiguous triangles in clusters
 - Excessive boundary edges

Tiny cluster



© 2004, CISE/FAST/UCB - Data courtesy of Stanford University

© 2004, ACM



There is an issue here that also existed for progressive buffers.

It is a problem with any regular spatial partitioning, whether its octree, right triangle hierarchy, or tetrapuzzles.

The number of triangles per cluster is dictated by the spatial size of the node.

Is it entirely possible that only a single triangle falls in a tetrahedron.

That cluster would not be efficient to render.

A surface could wave in and out of a partition which would mean lots of non-contiguous triangles and an excessive number of boundary edges which must be locked.

Triangles in typical meshes are not uniformly sized or distributed so there is no reason to believe that a particular level of the tree will have a specific size of triangles.

In fact this is an important point with most massive model visualization research.

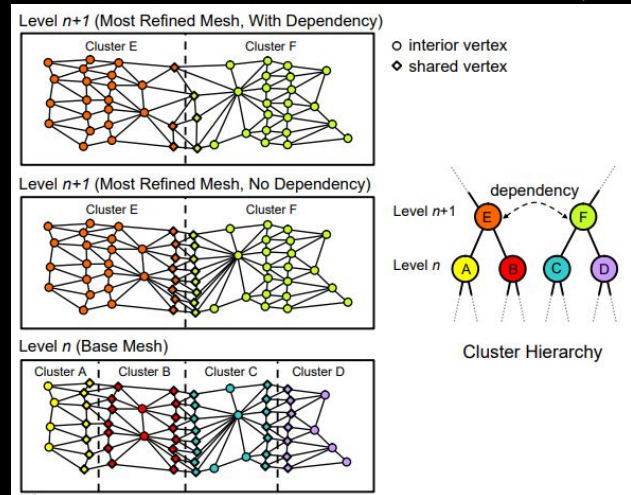
Most test with laser scanned sculptures which have a very uniform triangle density.

Uniform density is not a fair assumption.

Explicit dependency

© 2004, IEEE

- [Quick-VDR](#)
- [Batched Multi-Triangulation](#)



Just like before we can reduce constraints by adding information.

Instead of some nodes being implicitly dependent on one another we can store the dependent nodes explicitly.

That gives complete flexibility over tree building.

Both Quick-VDR and Batched Multi-Triangulation take this more flexible approach. The spatial schemes could store their data in the same structures, their implicit rules stored explicitly.

In fact, because of how dependent nodes are enforced and how the tree restriction is addressed by making the tree monotonic, there isn't actually a difference in how the data would be stored or how the run-time algorithm would work.

They are essentially all the same data structures and run-time algorithms. The difference comes in how the data structure was built.

Looking at it in this generalized way allows any variables to be optimized for. For example, number of triangles per cluster, or minimal number of locked boundary edges.

So out of all the irregular mesh choices this appears to be the best. No assumptions about the mesh's uniformity, or anything about it being closed or having thickness.

Coarse granularity LOD and occlusion decisions that are well suited to run on the GPU.

Anything complex to address cracks is handled during build, not at run-time.

Large batches of triangles are rendered directly. Triangle data is touched once and doesn't pass through memory many times.

Triangle clusters are stored completely independently from one another as just raw triangle data.

Any form of general mesh compression can be applied to the clusters so long as the boundaries match.

There's no problem with sparse residency.

Irregular mesh compression

- Seems solvable



Speaking of mesh compression, data size was a big concern before but it wasn't chief on my mind anymore.

Irregular meshes are very adaptive and are a sort of compression themselves.

The key thing that differed compared to geometry images or SVOs or whatever was storing the topology data, ie the index buffer.

But there was a fair amount of existing research that pointed to reasonable numbers.

Draw calls are an issue too!



With all this focus on high poly meshes, nothing I've explained so far really addresses draw calls unless you assume it's all unique geometry and textures, which I am not. Draw calls are just as much of a problem for artists as polycount and are a core aspect of the geometry problem.

That problem isn't just a tug of war between detail and cost. Optimizing for draw calls controls how a scene is built because it dictates the granularity that it is broken down.

Should a bookshelf full of books be merged into a single mesh?

But what if they need many bookshelves?

Do they all have the same arrangement of books on them?

Then there needs to be many different bookshelf variations just so they have different arrangement of books.

Ugh but now that's more memory.

So coarser granularity is best for performance right?

Not necessarily. Too coarse and culling is inefficient since draw calls are the culling granularity.

And none of this is necessarily the granularity that is most efficient for the artist to author.

So another part of the virtualized geometry goal should be that there is no

performance or quality impact of scene construction decisions.

There should be no ideal size or imposed granularity for breaking up the scene.

Decouple **visibility** from **material**

- Eliminate:
 - Switching shaders during rasterization
 - Overdraw for material eval
 - Depth prepass to avoid overdraw
 - Pixel quad inefficiencies from dense meshes
- Surface caching
 - Cache material eval in VT
 - VT stores compressed GBuffer



For this reason and many more I want to decouple visibility from materials.

I also only want to rasterize the triangles once per frame.

And pixel quad inefficiencies can get really bad with small triangles.

Drawing on my experience with VT I figured the best solution was to cache material evaluation into a virtual texture.

This is similar to texture space shading but does not include lighting.

What would be stored in the VT would be a compressed GBuffer.

My reasoning is that true texture space shading needs to be updated every frame and doesn't benefit much from caching.

Object space solutions of any form overshadow, usually by a substantial factor, 4x or more.

Texture space shading approaches also don't avoid pixel quad inefficiencies unless you keep around a heavyweight texture space visibility buffer.

GBuffer data on the other hand should be able to cache and persist

Maybe that could even mean another artist budget could be crushed.

That being material shader cost.

When rasterizing the triangles, all that needs to be output per pixel is the physical texture UV.

The idea was incredibly similar to what Sebastian Aaltonen described in his

SIGGRAPH 2015 talk.

We arrived at it independently though.

(circa 2014)

My money's on

- HLOD
- GPU driven
 - Multidraw indirect
- Surface caching



So to summarize at this point in time I was converging on something I thought would work well.

HLOD following a scheme like Quick-VDR

Except all the LOD logic is moved to the GPU.

That allows occlusion culling against previous frame depth buffer which I'd already had great success using HZB tests for years before.

The triangle clusters I intended to draw all with a single draw call using this new multidraw indirect functionality that I'd read about.

And the GBuffer would be cached in texture space and only sparsely progressively updated.

I talked up wanting to work on this but there were other priorities.

Geometry 2.0

- First time to really work on this at Epic
- Geometry 2.0:
 - Replace ancient BSP tools
- Also Geometry 2.0:
 - Future proof model format
 - Catmull-Clark subdivision



But folks knew I was interested in geometry stuff.

So when a new project popped up that had some interesting geometry problems I was asked to join.

The Geometry 2.0 term originally referred to geometry editing tools intended to finally replace the super ancient BSP editing that had been around since UE1.

There was a new idea of a future proof model data format also being referred to as Geometry 2.0.

See, typically every new game project builds all its art from scratch even if its a sequel because graphics technology improves enough that the old assets don't have enough fidelity anymore.

The idea here was that assets should be authored once to an archival level of detail and then never would need to be remade.

That same ideas coming back around.

The perspective of those that had been thinking about this was that with higher order surfaces, specifically Catmull-Clark subd, there would never again be a point where old assets could look low poly.

And with hierarchical edits, basically a form of sparse displacement, all types of things could be expressed.

Geometry 2.0

- One geometry unified solution
- Over constrained!
- On hold



There were also many other things from both the tools side and rendering that could use improvement as well.

If we are going to start with a clean sheet of paper maybe we can address all those things too with this new approach.

I won't go into all the details of everything that it was trying to accomplish
But suffice it to say the problem was overconstrained.

Just an example of one issue we ran into try to solve all the things with one approach:
Those old BSP modelling tools we were looking to replace were at the heart CSG.
Replacing them couldn't really be done without regressions unless the new modelling tools supported a form of CSG as well.

But the intersection of 2 Catmull-Clark surfaces doesn't result in a Catmull-Clark surface.

CSGing their control cages doesn't give a result that is even close.

But we couldn't just CSG the surface subdivided to a certain level because the original goal was that there wouldn't ever be a resolution that could one day not be enough.

There were many issues issues like that.
Eventually the project was put on hold.

The unification trap

- Elegance and unification can be a nice guide
- But sometimes there is a right tool for the job
- “Perfect is the enemy of good”
- Take it as far as reasonable and no further



Unification can be a honey pot.

It is often a nice guide but taken too far it can make everything worse.

The advantage to unified solutions is a single piece of code that can be invested in.
The point is to reduce work.

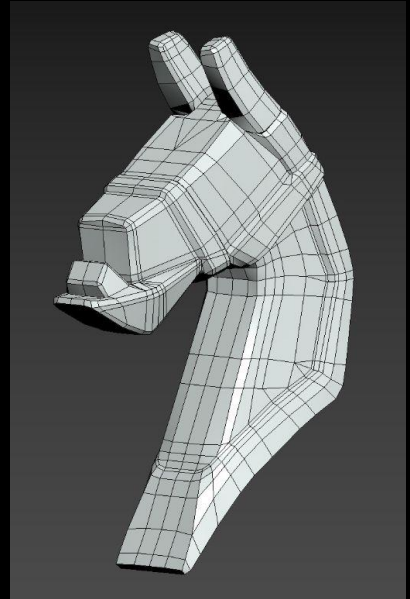
This is all good but when pushed too far the unified perfect solution that solves every problem simultaneously can become so **insanely** difficult to find or execute that you spend more time seeking it than it would take to write multiple specialized solutions. It's a form of the saying “Perfect is the enemy of good”.

Unified approach to tools can also mean a consistent user experience.
There aren't silos. Skills learned in one area transfer.

But sometimes there is a right tool for the job.

Talk to our artists

- How high are the subd cages?
- Subd is amplification
- Needs another method for simplification
 - Simplification is elegant and unified!
 - Trivial to amplify offline



All this was predicated on Catmull-Clark subd being a good solution for future proof modelling.

But does adaptive subd even solve this?

I talked to our artists and most of their control cages were higher than the game meshes we were currently using.

In film they are orders of magnitude worse.

That can't be the lowest detail we render.

Subd is amplification.

A simplification solution would be needed in addition.

Two different, very complex solutions, one for up, the other for down.

Pretty obviously out of the zone of elegance and unification.

Funny enough simplification all by itself **is** elegant and unified!

A static mesh can be subdivided offline and one method can handle both up and down.

Confidence



The sad thing is I already knew that.
I said as much in that blog post in 2009.

There was an important lesson here that I have tried my best to adopt.

No one I was working with had walked the path I had already
and we were repeating numerous mistakes I had already made.
And I didn't say so. I didn't argue my perspective.

Like before when I had just joined Epic, I was joining an established group of what
were clearly very smart people and I wasn't comfortable with challenging the high
level direction.

I needed to get past discomfort and teach others what I knew already because I
definitely knew a thing or two about geometry by this point.

I have continued to be surprised throughout my career about my expertise vs others.
Don't assume you either know more or less than others.

I guarantee at some point you will run into someone that knows more than you on
almost any one topic.

And you'll be surprised at what luminaries in the field don't know.

The only way to get at truth is to say what you think you know without fear of being
wrong and without expectation of being right.

Either it's an opportunity to learn or an opportunity to teach. There is no shame in
either.

Being wrong is not nearly as bad as you imagine. It's the state you are before being right.

Virtual texturing again

- I was right all along!
- SSDs reinforce this even more
- UE needs VT before VG



One of those things I knew all along was that mip based texture streaming doesn't scale.

It turns out it was a problem in Unreal too, the engineers just weren't hearing the pain.

Regardless of whether VT was a good idea before it was clear it was going to be important in the future because SSDs were now commonplace.

I was convinced at this point that virtualized geometry was going to be the next big thing.

In general large data, fine grained streaming, data compression, and general data virtualization was the key for a big jump in fidelity.

But we still didn't even have the last generation of virtualized data.
We need virtual textures first!

So I worked on that for a bit.

=====

Art managed the budget and never imagined the possibility that they didn't have to. Engineering didn't hear their pain and never saw what could of been since art constrained themselves.

There's another lesson here. Although it wasn't new to me at the time, I'll pass it along anyways.

The most radical innovation will not come from user requests at least not in how it was originally requested.

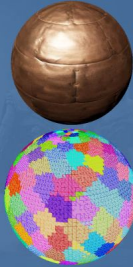
Users don't know what is actually possible so they won't ask for things that seem impossible to them.

This makes it doubly important to understand how they work and the problems they deal with so that you can identify which of those could be solved.

New work published

Mesh Cluster Rendering

- Fixed topology (64 vertex strip)
- Split & rearrange all meshes to fit fixed topology (insert degenerate triangles)
- Fetch vertices manually in VS from shared buffer [Riccio13]
- DrawInstancedIndirect
- GPU culling outputs cluster list & drawcall args



SIGGRAPH 2015: Advances in Real-Time Rendering course

© 2015, Ubisoft

Per-Triangle Culling

- ▶ Each thread in a wavefront processes 1 triangle
- ▶ Cull masks are balloted and counted to determine compaction index
- ▶ Maintain vertex reuse across a wavefront
- ▶ Maintain vertex reuse across all wavefronts - `ds_ordered_count` [5][15]
- ▶ +0.1ms for ~3906 work items - use wavefront limits

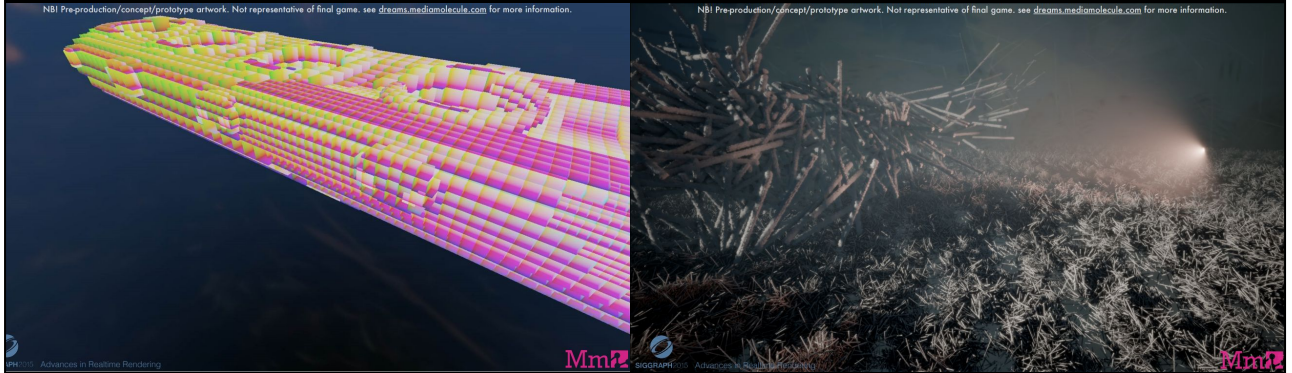


© 2016, EA



In the meantime new exciting work was published that changed the landscape. Most directly related was triangle cluster culling which was finer culling granularity than I had been considering and should be a drop in replacement for multidraw indirect batches.

New work published



But also there was exciting stuff with voxels and point based rendering from Dreams.

Define the requirements

- Not interested in completely changing all CG workflow
 - Support importing meshes authored anywhere
 - Still have UVs and tiling detail maps
 - Only replacing meshes, not textures, not materials, not tools



I had informally arrived at some requirements for the solution through this process but it is worth formally stating them.

I can't completely change all CG workflow

The vast majority of modelling and texturing will happen outside of Unreal. I don't have control over that data format.

Users may import CAD models or film assets that were authored primarily for another renderer.

They will have UVs and tiling detail maps

From the users point of view this new virtualized geometry tech should slot in exactly where static meshes do right now.

UVs

- Single 512x256 texture!

© 2012, Tor Frick
<https://www.unrealengine.com/en-US/blog/amazing-one-texture-environment>



An aside on UVs and data reuse.

Academia too often conflates texturing with painting a mesh. 2d textures are very often used more like wallpaper than paint and controlling their placement requires editing the UVs.

The best demonstration of this that I can think of is this SciFi lab created by Tor Frick. The entire level uses a single 512x256 texture.

While this example is a little old, and this amount of memory optimization is extreme, the general technique used here is everywhere in games.

UVs



Looking at something more modern.
The carbon fiber here is a tiling texture.
The brushed metal is a tiling texture.
And controlling its orientation matters.

Data reuse



Model instancing is how we reuse geometry data.

In John Carmack's argument for unique texturing he said texture tiling is just a limited form of compression.

He's correct. But it is also a damn good form of compression.

It is very common to reuse the same data 1000s if not millions of times across a game.

There is no generalized image compression that gets ratios like that.

Reuse is what enables miles of ground

Data reuse



With fidelity of pebbles.

So long as the entirety of a game needs to be delivered to the player up front, data reuse like this will be essential.

=====

It's possible in the metaverse of the future that it will be more efficient to stream the data of just what you are looking at

And a unique non-procedural version will optimize the worse case to be the same as the best case.

Maybe.

But currently we are constrained by the mechanisms we have to deliver data to the user.

For the most part, the entirety of a game needs to fit on a bluray disc.

So data reuse is essential for the foreseeable future.

Define the requirements

- Not interested in completely changing all CG workflow
 - Support importing meshes authored anywhere
 - Still have UVs and tiling detail maps
 - Only replacing meshes, not textures, not materials, not tools
- Supports instancing
 - Cheap object transforms
 - Cannot be unique world space data



So the next requirement is that it supports instancing with arbitrary instance transforms and is not a unique world space data structure.

Define the requirements

- Not interested in completely changing all CG workflow
 - Support importing meshes authored anywhere
 - Still have UVs and tiling detail maps
 - Only replacing meshes, not textures, not materials, not tools
- Supports instancing
 - Cheap object transforms
 - Cannot be unique world space data
- Shippable on next gen consoles
 - 60hz
 - >1080p



Lastly, it obviously needs to be fast.

I expect many next gen games will be 60hz.

That does not mean we have 16ms to render geometry.

Many methods claim to be fast but take the entire frame just to draw colored surfaces.

I expect a similar frame breakdown as we've had before.

< 6ms to draw geometry with materials on it.

Accounting for material shader evaluation that means probably 4ms or less for the geometry portion.

Chart a path

- Do not try and solve “Everything Everywhere All at Once”
- Plan stepping stones
 - Give yourself the ability to succeed in the steps along the way
 - Can't be a direction that clearly won't lead where you want to eventually go
 - Less chance of getting shut down
 - You will learn much when users get their hands on it
- My path:
 - Static environment most important
 - Animation should follow
 - Maybe terrain later if it makes sense



Those are the hard requirements.

It is worth prioritizing the soft goals as well.

Taking a lesson from Geometry 2.0,

Do not try and solve “Everything Everywhere All at Once”.

Plan stepping stones such that there can be success along the way to the ultimate goal.

Those successes will boost your self confidence and the confidence of your superiors meaning less chance that your project gets shut down.

Ideally if you can get intermediate results into the artists hands you will learn things you didn't previously know.

In this specific problem, the area with the biggest geometry budget problems is environment art.

So, priority #1 is virtualized geometry for what are now static meshes.

Deformable geometry and animated characters would be second and should ideally follow from extensions to the same system.

Doesn't need to be solved immediately but it needs to be conceivable that the path we are on will lead there.

Third is terrain.

While improvements to rendering quality and tooling for terrain would be nice that isn't

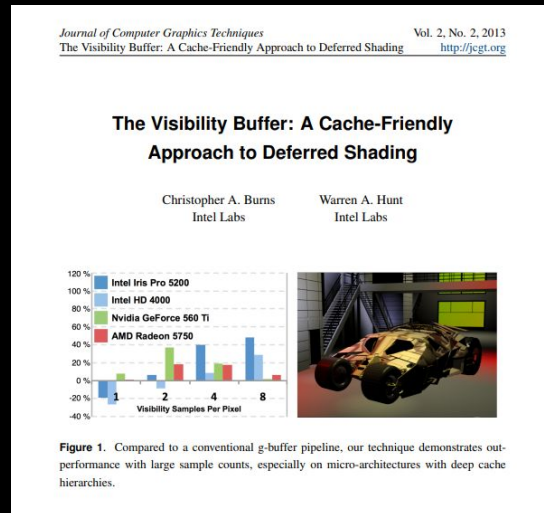
the goal here.

Terrain authoring doesn't really have budget problems.

Maybe a shared solution can be used for all these problems but this is far too difficult of a problem to introduce additional constraints or distractions.

Surface cache doesn't apply to enough

- Assumed mostly cache hits
- Our materials are commonly:
 - View dependent
 - Animating
 - Non UV based
- Visibility buffer



Through shipping Paragon I was given a lesson in how prevalent view dependent, animating, and non UV based materials are.

The surface caching idea assumed a limited number of pages would need to be generated each frame because there are mostly cache hits.

But all these things invalidate the cache which would result in tons of new pages every frame.

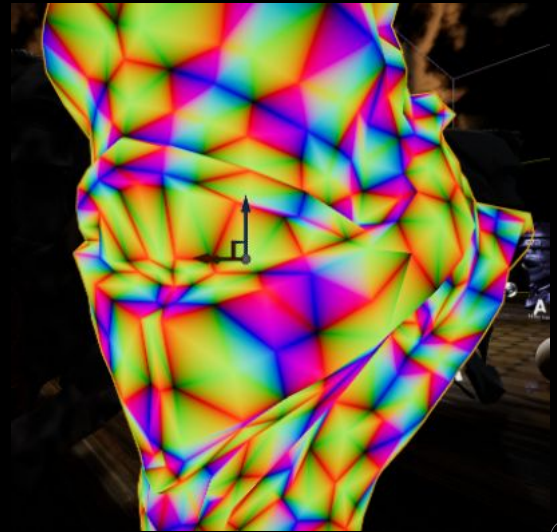
I expect it would be common to shade more texels than pixels.

The alternative is to output a visibility buffer. We lose any chance of addressing material shader budgets but it appears that wasn't reliable.

Visibility buffer checks all the other boxes.

Barycentrics by graph coloring

- Any planar graph can be 4 colored
 - Slow but 5 coloring is fast
- Graph is vertices connected by edges
- Edge can't connect same color
 - Each vertex of triangle is different color
 - Interpolating colors won't collide



[skip]

I was a little concerned that deriving barycentrics by transforming all 3 verts again was going to be too expensive.

Writing out barycentrics should be a lot faster.
The simple way of doing so breaks the post transform cache.

Does that mean visibility buffer requires hardware that supports SV_Barycentrics?
Wasn't clear at this point that it could make sense to mix GBuffer and VisBuffer at the same time in the renderer so limited hardware support could be a deal breaker.

I had a bit of time to explore this as switching to a visibility buffer might be a good call across the whole renderer, independent of any new geometry tech.

My idea was that

Any planar graph can be 4 colored. Meshes are obviously planar if they are UVed in the typical fashion.

In this context the graph is the vertices connected by edges so a valid coloring means an edge can't connect vertices that share the same color.

So there exists a vertex coloring such that every triangle has different colors for all 3 corners.

That means interpolated colors across the triangle won't collide.

So those color components are the barycentrics rearranged.

All we need to know then is which components those are for each triangle.

This turned out to be stupid because accessing `SV_PrimitiveID` in a pixel shader seriously degrades the post transform cache anyways.

And the whole thing was ultimately pointless since not writing barycentrics was actually a critical component to Nanite working in the end.

The Unreal Engine 5 logo features a large, stylized 'V' shape composed of two overlapping triangles. The left triangle is dark blue with a lighter blue gradient, and the right triangle is a solid light blue. The 'V' is centered on a black background. The text 'UNREAL ENGINE 5' is written in a bold, white, sans-serif font across the middle of the 'V'.

UNREAL ENGINE 5

Then suddenly opportunity stuck.

With the coming next generation of consoles it was decided that there should be a full version bump of Unreal Engine.

There was going to be a UE5!

I joined Epic after UE4 had been unveiled. I contributed to its evolution but not really to its initial design.

The mission of “UE5” and defining the next generation of real-time graphics put lofty expectations on this unlike anything I had ever worked on before.

While I talked about hugely ambitious goals earlier, this was the point where the weight of the opportunity really sunk in for me.

This was the next generation of Unreal

UE5 must be obviously a generational leap better than UE4.

A laundry list of miscellaneous odds and ends is not good enough.

This was our definitive moment to make a statement to the world what the next generation of games will look like.

This was **my** moment to present my vision of the future of real-time computer graphics.

I need to live up to that.

Collaborating with others we wrote up a tech direction that is surprisingly close to where we’ve ended up.

My pièce de résistance was virtualized geometry.

I argued that there was a good chance that now was finally the time that this could work.

And it was worth taking that chance.

With that I got freedom and a timeline that was unlike any I had before.

My full time job from that point on, with few exceptions, was to R&D virtualized geometry.

UE5

©Allen Wei
<https://www.artstation.com/allenwei>



In the process of trying to sell this vision I searched for images that exemplified what could be done if this problem were solved. With it one could perhaps imagine our next tech demo.

I found some amazing concept art from the artist Allen Wei. Everything covered in thin pipes, trusses and buttresses.

It's both perfect for extreme geometric complexity and would likely kill the thing I planned to make.

As I mentioned earlier, most of the existing research tested with laser scanned sculptures. These meshes are fairly smooth.

They can hit 1 pixel error with average triangles that are considerably larger than a pixel. I feared that is not the case here. These surfaces are not smooth or continuous.

I have no reason to believe I'm going to be able to rasterize pixel sized triangles. There is a major performance cliff with hardware rasterization at that scale.

I do not want to invest heavily in tech that can only make lumpy rocks.

That was a big reason why I steered away from geometry images and displacement towards irregular meshes.

I want to solve hard surface and mechanical as well.

But if mechanical fails when pushed hard it isn't really solved.

So,

I'm certain triangle cluster rasterization based will work but not to pixel level
Will something else?

I could have plowed forward with the idea I had that I was confident was good.
But the weight of UE5 caused me to hold off and explore alternatives.

=====

In retrospect this was an unusual and perhaps courageous decision.
I had the proverbial winnings in hand. I could have cashed out but instead I let it ride.

What it felt like at the time was the pressure to deliver fast dropped off and the
pressure to deliver big rose considerably.
So I kept my fallback option in my back pocket while I searched for what might be
even better.

Triangle ray tracing?

- Before we had hardware ray tracing
- LogN is nice but still increases
 - Do some napkin math. How many levels?
- Packeted tracing?
 - Levels shared amongst packet
 - Hierarchical raster can get the same
- Cache miss city
- Can't beat LOD for perf
- Need it for streaming anyways
- Also BVHs are a lot of data



<https://youtu.be/J3ue35ago3Y>



The first I considered was ray tracing instead of rasterization.

LogN means ray tracing scales well to complexity.

But it still scales. Just to get to 1M triangles in the scene with a 4way BVH you'll have to do 10 dependent loads.

15 to get to 1B triangles.

It's important to note this was before I had any RTX hardware to test.

I don't remember whether I even knew it was coming yet.

But what had just been published was Oculus research's hierarchical ray caster.

This was basically hierarchical packeted tracing.

Any levels of the tree that will be redundantly traversed for some group of rays can be done so once for a frustum bounding the rays.

Essentially it's the same BVH concept but for rays.

Not a new idea but the 10B rays per second they were touting caught my attention.

Experimenting with it made me conclude these numbers were highly dependent on triangles covering many sample points.

For VR applications that may be true but not helpful for pixel sized triangles.

I realized that if the key task is to pick out which of the very tiny triangles lands on the pixel center then a hierarchical rasterizer could be written as well.

I'm not sure if such a thing has been published before but the idea is instead of

traversing the scene BVH once per ray you traverse it once for the entire frame. Continue on to a node's children if they project on screen to cover any pixels. A regular raster grid makes intersecting pixels simpler than arbitrary rays, frustums, or cones.

Occlusion is more difficult but possible.

But the idea of picking out 1 triangle for this pixel, then skipping 1000s before finding the next triangle is cache miss city.

Clearly inferior from the performance point of view to LOD.

Besides LOD is needed for streaming anyways.

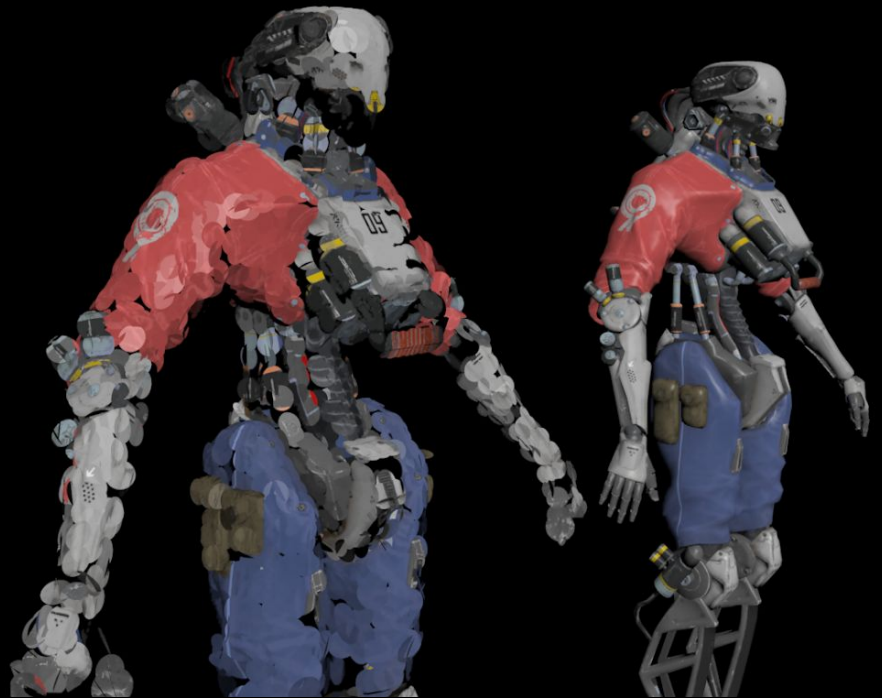
LogN cost for tracing but linear scaling in memory.

And BVH data is not cheap either.

R-LODs are an interesting hybrid but then the LogN part isn't relevant. [8][12]

Points?

- Animation?
- Vray proxies
 - Gave hope



With Dreams showing great success with points and Atom View showing some virtualized geometry looking stuff animating I thought it was worth taking a look at point based rendering.

Easily supports animation is incredibly appealing and I had no doubt that atomic scattering points on screen was fast.

I had heard an encouraging thing from Jerome, our art director, that when he worked in the digimatte team at ILM, vray proxies changed their world in terms of dealing with geometric complexity.

Memory was more an issue for them than render time. Once they started using VrayProxies they stopped having to worry about running out of memory.

Turns out this magical feature was a far more simple surfel representation than I expected given the results he claimed.

A really nice thing to see was that the surfels were parameterized. They were decoupled from the material which could be swapped. That's exactly what I need as well

Infinitesimal points

- Fast scatter through binning
 - 64bit atomics
- Holes and AA
- Thickening problem
- Prefiltered AA doesn't work
- Binning points == Nearest neighbor filtering



There are many different types of point based rendering. What I was interested in was scattering infinitesimal points to the screen using 64bit atomics.

The work per point is the least possible: a matrix multiply and pixel write.

Points can be guaranteed to be no further than 1 pixel apart so that their aren't holes. I'm not sure that is practical for performance but that doesn't really making sense with animation unless the points are dynamically generated from something else. So let's assume some amount of hole filling.

First problem that comes up is what does antialiasing mean when points are binned?

A point doesn't have a shape that a pixel center intersects.

Instead the points position is quantized.

Thin features become as thick as 1 pixel no matter how far away they are.

So, thin wires appear to get thicker the further away they get.

This is the same problem voxels have btw and it looks bad.

Could prefiltering solve this?

Points could have a coverage associated with this mip level.

While true that doesn't provide interpolation between points.

Binning points is actually equivalent to nearest neighbor texture filtering, just in reverse, if you want to picture the artifacts.

So we can have mipmaps of points but no bilinear or trilinear filtering so that wire isn't going to appear to get thinner until it jumps to the next mip.

That is unless it is done stochastically. I didn't think of stochastic at the time so some of this might be worth looking at again.

Hole fill

- Points need to represent a real surface for AA
- Point scatter seeds screen
- Post process completes surface
 - Expand points to area or volume of primitives
 - Connect points
 - Primitive is surface in between points
- “We’ll fix it in post!”



Hole filling makes this stranger yet.

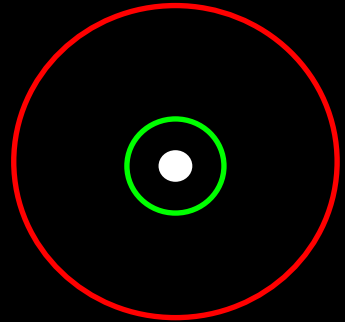
I concluded that the points needed to represent a real continuous surface such that AA could do real intersections.

Scattering the points to the screen seeds data for a screen space pass to connect and complete the surface.

The idea was that filling in post using gather would be faster than scatter by rasterization.

Hole fill

- Implicit connection rules?
- Is hole from:
 - Sampling resolution or
 - Original surface?
- Explicit fill rules
 - Store metadata to know which points to connect
- Dart board problem
 - Binning is aliasing
 - Only one point wins



But what defines a hole?

Was it supposed to be there or not?

If all gaps of a certain size are filled things are going to look bloby, like there's surface tension.

I can store something about where holes should and shouldn't be?.

How about storing the distance to the furthest point that should be connected to?

Turns out this idea of seeding the screen and completing the surface in post is faulty because of aliasing.

A simple illustration of this is this dart board configuration.

2 points can land in the same pixel, the smaller in front of the larger.

The larger needs the behind surface to be filled in but that information is lost because the closer point won out.

Hole fill

- Animation separates points when stretched
 - Violates implicit connection
 - Amount of separation is unbounded
- Storing connectivity metadata is just a mesh
 - Round about reinvented triangle meshes
 - Index buffer
- Faster to rasterize the triangles connecting those points?



While that was probably reason enough to kill this there were more problems.

The original interest driven by animation being easy doesn't work either.

There are no bounds on how much animation can stretch a surface

Which means there is no bound on how far apart points could get on screen.

If something else was generating the points on the fly then maybe stuff like this could be addressed but point clouds are not a robust data structure to express surfaces by themselves.

Carrying the connectivity metadata idea further devolved into storing the point indexes of neighboring points that the hole filling should connect to.

Basically just reinvented triangle meshes and index buffers.

All this was a ton of screwing around just to try and do that connection in post.

But honestly, would it really be that expensive to do it during the scatter?

Could fill in the pixels between 3 points,
you know,
a triangle rasterizer.

Voxels?

- Implicit surface for rendering and simulation?
- Many advantages
 - Fast intersection and collision
 - LOD is simple
 - CSG is simple
 - Inherently ray traceable
- Don't believe in it so let's rule this out fast



At this point I was approached by Michael Lentine who was starting on the beginnings of what would become UE's new physics system.

He said that in film it was a big pain that there always needed to be a dual representation.

Rendering worked with explicit surfaces and simulation with implicit.

A world of possibilities might open if there could be a single geometric representation, an implicit surface.

By that he meant signed distance fields stored as voxels.

There are numerous other advantages that appealed to me from the rendering point of view and I had not seriously explored voxels yet.

Naturally supporting incoherent secondary rays is huge.

But also cheap collisions with all sorts of shapes, CSG

And in theory level of detail is simple.

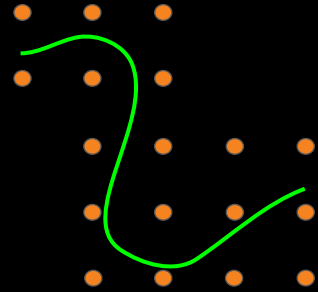
Overall I didn't believe it was going to work out though.

If it did the rewards would be huge so it was worth considering.

The goal in my mind was to rule it out quickly.

Voxel storage size

- Data size is biggest concern
- Needs to express continuous surface
 - Magnification and AA
- Signed distance field
- Narrow band



My biggest concern from the onset was data size.
Surface data scales quadratically with resolution.
Volume data scales cubically.

SDF adds more data to voxels not only from distance values but also that at least one value on either side of the surface is necessary to interpolate to zero.

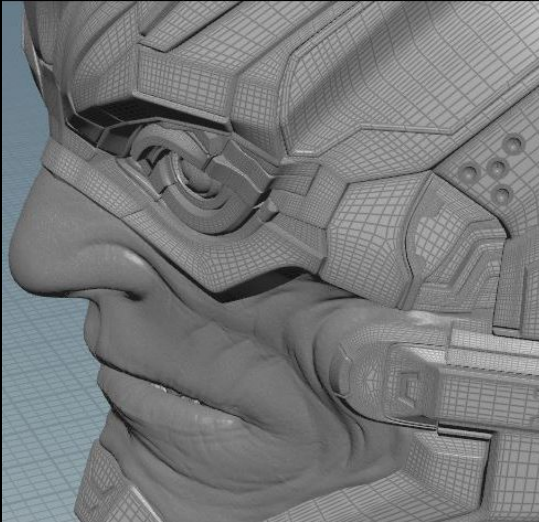
A SDF does address a few issues with voxels though.
What does it look like magnified?
Binary voxels are cubes and that is no good.
Same goes for attribute data. We want texture filtering.

The second is the antialiasing issue I mentioned earlier.
Signed distance means a sloped edge can be a straight line, not a stair stepped one.
That affects edge quality.

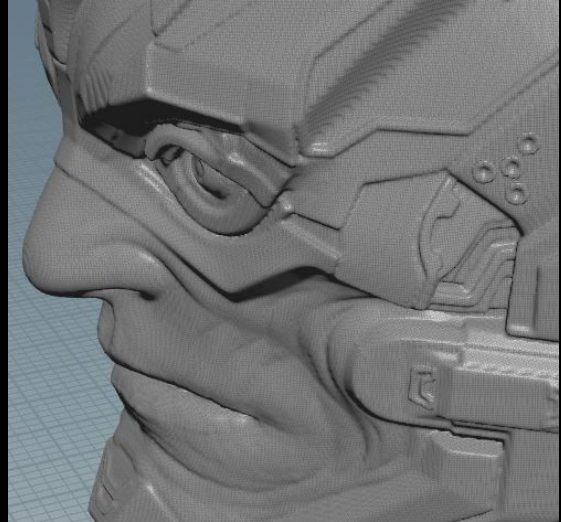
Because of the data issue I think the only way this is going to work is if I store only the narrowest band data per mip level.
By that I mean only the values immediately on either side of the surface.

Voxel storage size

Source mesh



Resampled to voxels



What resolution is needed?

This is a 2M poly bust

Resampled to 13M narrow band SDF voxels

Even though there is far more value data than the original it looks blobby

Artists didn't like resampling and this is why.

Signal theory tells us that there will be loss of high frequency details for any resolution less than the Nyquist rate.

Or in other words voxels must be half the size of the shortest triangle edge or less to not lose detail.

That would be an **insane** amount of data!

Adaptive resolution can help. Higher resolution is only needed where high frequency data is.

The irregular mesh is obviously doing that.

Adaptive distance fields are already a well known solution to this problem.

But even with perfect adaptive resolution the Nyquist rate means 4x the voxels as vertices.

That ignores the data required to express that adaptiveness, ie where voxels are and aren't, which is not free.

The difference to mesh compression at this point starts to shrink.

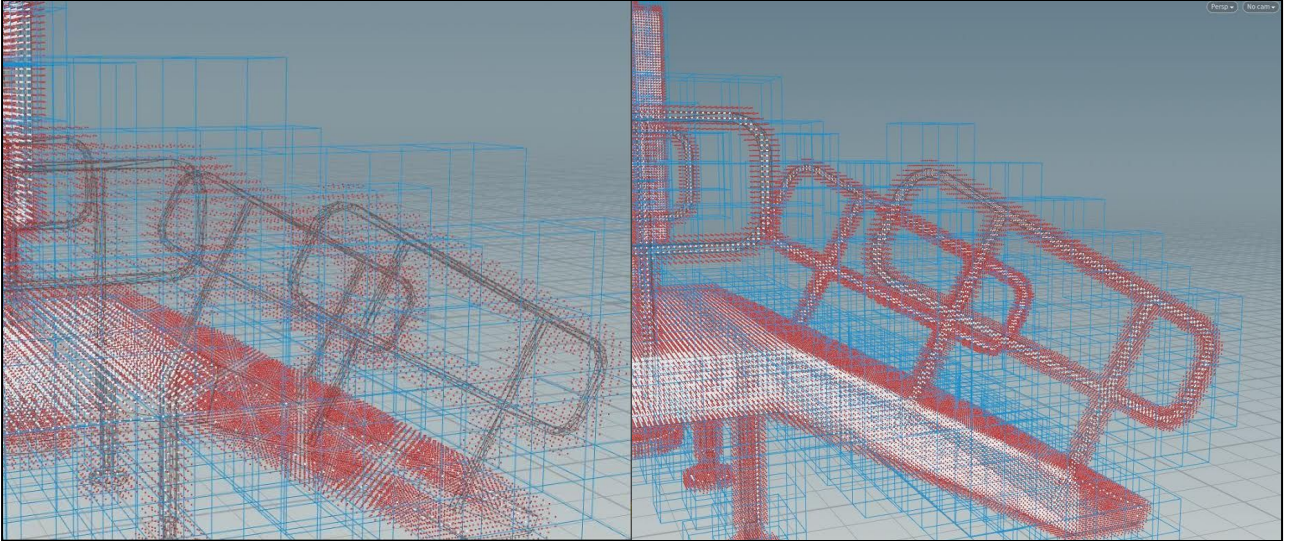
I won't argue that they are equivalent, but at the very least a great deal of effort and sophistication is required to beat meshes when attempting to store geometry which was originally authored as a mesh at some definition of lossless.

So my conclusion is the mip0 data, so to speak, is best to retain in its original format.

Either that means switching to rasterizing triangles when close enough, which kills all of those advantages voxels had and means multiple rendering paths
Or the triangles need to be voxelized on demand and cached for the finest detail levels.

That effectively provides infinite zoom without ever seeing voxels get large on screen.

Voxel thin features



A similar thin feature problem I mentioned with points happens here too.

With SDFs those features will vanish unless a voxel center happens to land inside the surface.

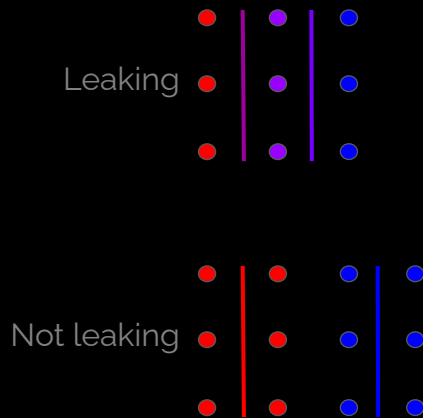
Analysis can be done and the SDF can be modified to grow such that it doesn't vanish.

But features thinner than the width of a voxel at that level of detail are not expressible as an SDF.

Voxels would have to express some sort of coverage information to handle this.

Is that in addition to the signed distances or replacing them?

Voxel leaking



This seems like just an issue with little spindly details but at some scale every wall falls into this case.

The thickness of the wall will need to be artificially increased or the wall vanishes.

Attribute data has issues before then.

As mentioned a signed distance value is required on either side of the surface with opposite signs such that there is point where they interpolate to zero.

Only 1 negative value is needed in between 2 positive for the thinnest surface.

But 2 internal values are needed to reliably interpolate attribute data.

A wall needs to be able to support 2 different colors on each side.

Having only one color between them means leaking.

So that means surfaces need to be at least 1 voxel thick to exist and 2 voxels thick to not leak attributes.

Alternatively attribute data could be stored in a directional basis but that significantly increases it's footprint and would often be worse than just increasing the resolution.

Voxel attribute data

- UVs don't filter
 - Discontinuity data?
- UV seam stair stepping



Speaking of attribute data, not all of it is filterable.

Specifically UVs.

UVs have seams and filtering across a seam is invalid.

Filtering within a chart is required to be continuous.

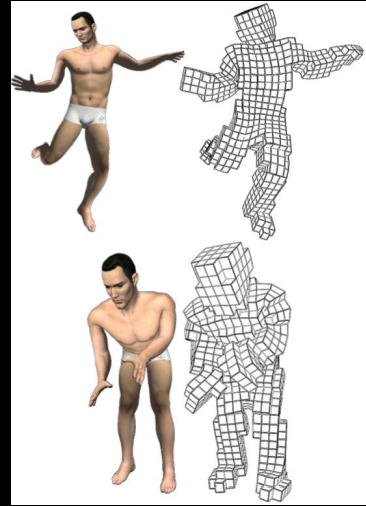
That means some sort of discontinuity information needs to be stored to say which neighbors can interpolate and which can't.

That might solve the leaking issue as well but how simple can that data really be?

If we want UV seams to not stair step either there now needs to be some sort of SDF of the UV chart boundary.

Voxel animation?

- Not required yet but is there a path?
 - Animation cages
 - Morph targets?
 - Displacement maps?



© 2009, IEEE



Should also consider some future requirements.

Don't need to solve these yet but does it seem like there is even a path there?

Can we deform and animate voxels?

Yes, some research has been done in regards to deforming the incoming ray instead of the geometry.

How controllable it is for a rigger is unknown.

How about morph targets?

I mean maybe you could do some sort of advection but no. Morph targets only really make sense with explicit surfaces.

How about displacement maps? Same thing.

A different form of procedural geometry could be done in the SDF space but would not be portable.

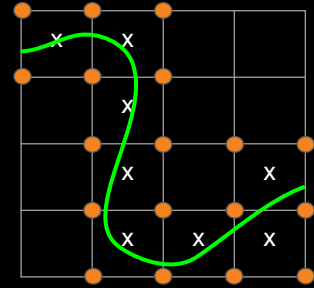
=====

There was actually an interesting paper published since then.

<https://cs.dartmouth.edu/~wjarosz/publications/seyb19nonlinear.pdf>

Voxel prototype

- Brick based
- Hybrid scatter gather
- Bit occupancy
 - 4x4x4
 - uint64
- OpenVDB like
- DDA ray march



It's time to build a prototype to test performance.

The approach I experimented with was inspired by both Dreams brick renderer and OpenVDB

Bricks are scattered to the screen. The simplest form of this is rasterizing a cube per brick.

Within a brick I do a short ray cast through the brick data.

This is a very effective empty space skipping technique and it has the advantage of supporting arbitrary transforms at the scatter phase for little cost.

A hybrid scatter gather doesn't require the data to be stored as bricks.

It could be used with any tree.

Storing the data as bricks avoids pointers near the leaves to save space.

But full bricks of value data are likely even worse. I only want to store the narrowest band within the brick.

Instead I'll store a bit mask for voxel occupancy.

A uint64 expresses a 4x4x4 brick's occupancy.

The value data can be packed and indexed using a prefix sum of the bit mask.

Not having a dense SDF sort of rules out sphere tracing.

Instead a ray can be marched through the brick using DDA. Checking a voxel bit at a time.

Only a single uint64 read is needed for the entire ray march of the brick which should

mean this is really light on bandwidth.

Once an occupied voxel is hit the SDF values could be read.

These occupancy bits aren't actually 1 for 1 with the value data.

They instead are the voxels that the surface intersects.

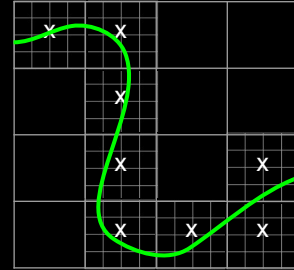
The SDF values are on either side of the surface.

They are at the 8 corners of an occupied cell.

The SDF is interpolated within the cell where at some point they will cross zero.

Voxel prototype

- Add a level
- 4x4x4 bricks of 4x4x4 voxels
 - 16x16x16 voxel brick
- Hierarchical DDA



To make the bricks bigger and get a bit more gather and less scatter I added one level of hierarchy to the bricks

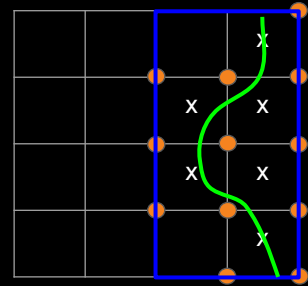
Instead of 4x4x4 they would be 4x4x4 bricks of 4x4x4 voxels.

So 16x16x16 in all.

This means a hierarchical DDA, again like OpenVDB

Voxel prototype

- Shrink bounding box



A few optimizations

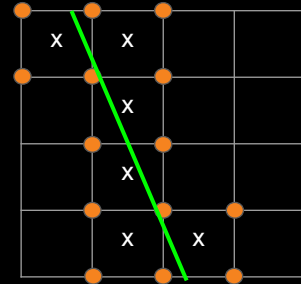
Most bricks are sparse. The box used when rasterizing them can be shrunk to fit the occupancy bits.

Finding the bounds can be done with some bit fiddling.

Or with some more memory the bounds can be shrunk to the exact surface within the brick.

Voxel prototype

- Atlas packed slabs

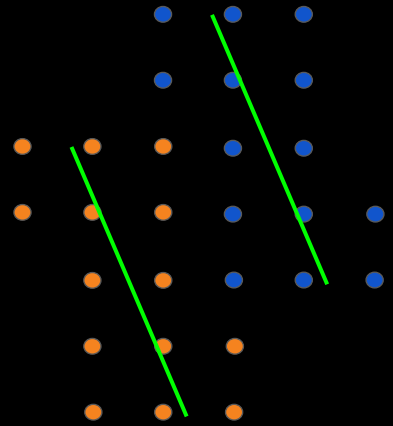


Once an occupied cell is hit fetching the 8 corners of packed data is both tricky since the occupancy bits are not 1 to 1 with values
And its expensive. 8 reads and a trilinear lerp.
Sure would be nice to use the texture unit's trilinear.

Conveniently most bricks should be fairly repetitive.
No matter how bumpy a surface is the general orientation of many bricks will be the same.

Voxel prototype

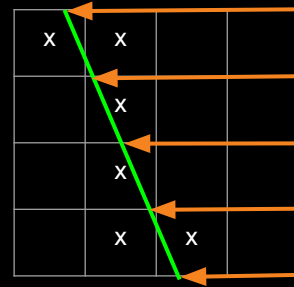
- Atlas packed slabs



So my idea was to pack the value data into volume texture atlases.
Like any atlas packing there will be waste but the hope was that it was minimal and this is only needed for the in memory representation.
The data on disk could be maximally sparse.

Voxel prototype

- Height field is cheaper



But if we look at those bricks again the majority of them could be represented as a height field along an axis.

In this example there is nearly a third less data required compared to the SDF.

Voxel realizations

- Narrow band killed CSG and physics
- Dual representation required?



At this point there are a couple of realizations that call into question the original goals
By being forced by data size to only store narrow band values
there isn't really a field anymore is there?

My conclusion at the time was that the original goal of a unified representation failed.
Thinking about it now that isn't technically true.

For a **single** level there aren't values further away.

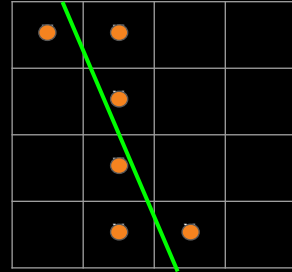
But if all mips are overlaid every point within the bounding box of the object has value
data of some resolution, so long as the root brick is dense.

I don't know whether the rapid loss in resolution of the distance field is still
serviceable.

There is even attribute data covering the whole space too which is neat. I'm not sure if
that is useful.

Voxel realizations

- Retaining source data solves magnification
 - SDFs not needed for this
 - Binary voxels good enough?



The other realization, which I still stand by, is that by deciding to retain the source triangles at mip0 so to speak, the idea that SDFs are needed for magnification doesn't hold up anymore.

The source triangles handle magnification.

So there shouldn't be a point where voxels get big and clearly look like cubes.

Given that,

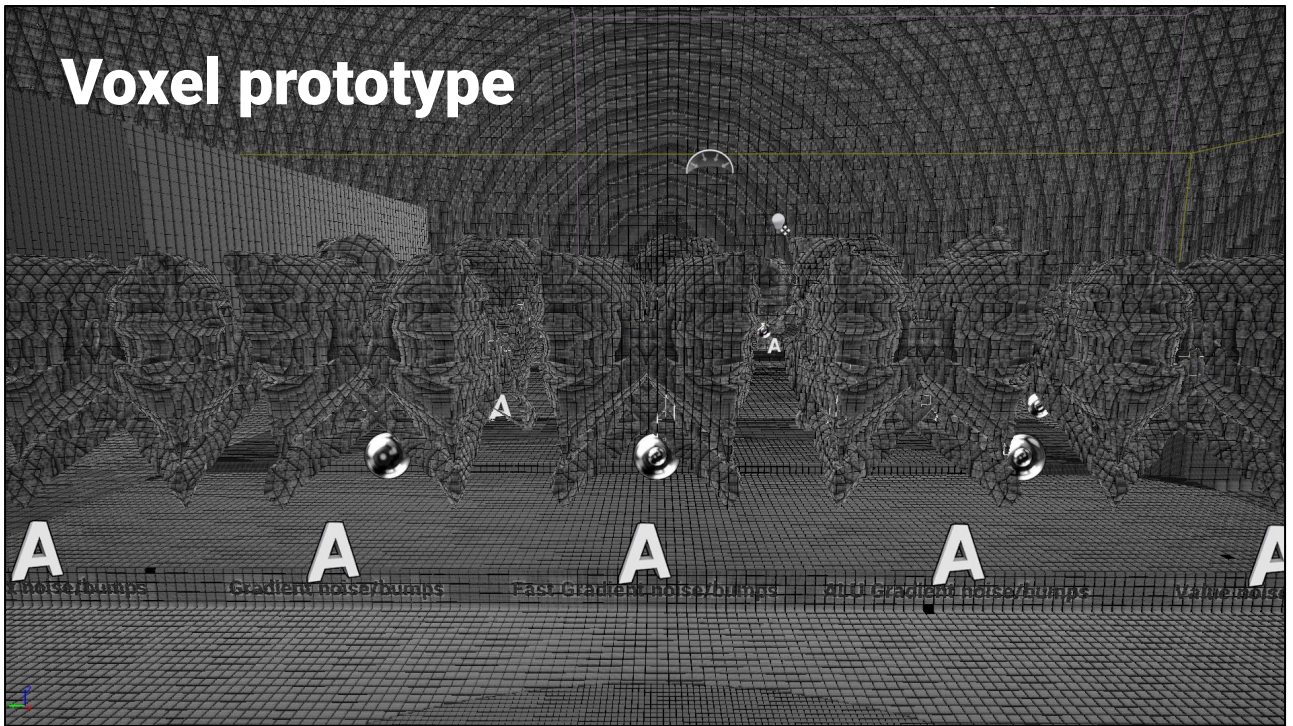
is there still a good reason for the SDF data and the SDF intersections instead the cells?

Same goes for attribute data.

How important is texture filtering in the presence of good AA if there is no magnification?

Both of these realizations combined and the value of being an SDF is seriously called into question.

Binary voxels would make things simpler and the data size smaller. No more values on both sides of the surface.



And here's the results of the prototype.

Never got to the SDF part.

I started with intersecting the occupancy bits and would add the SDF intersection later.

I did the bounds shrinking but no height fields.

The coloration in this image is the number of DDA steps.

Sorry for the lame test scene.

This image scatters the bricks using the rasterizer and the ray cast is in the pixel shader.

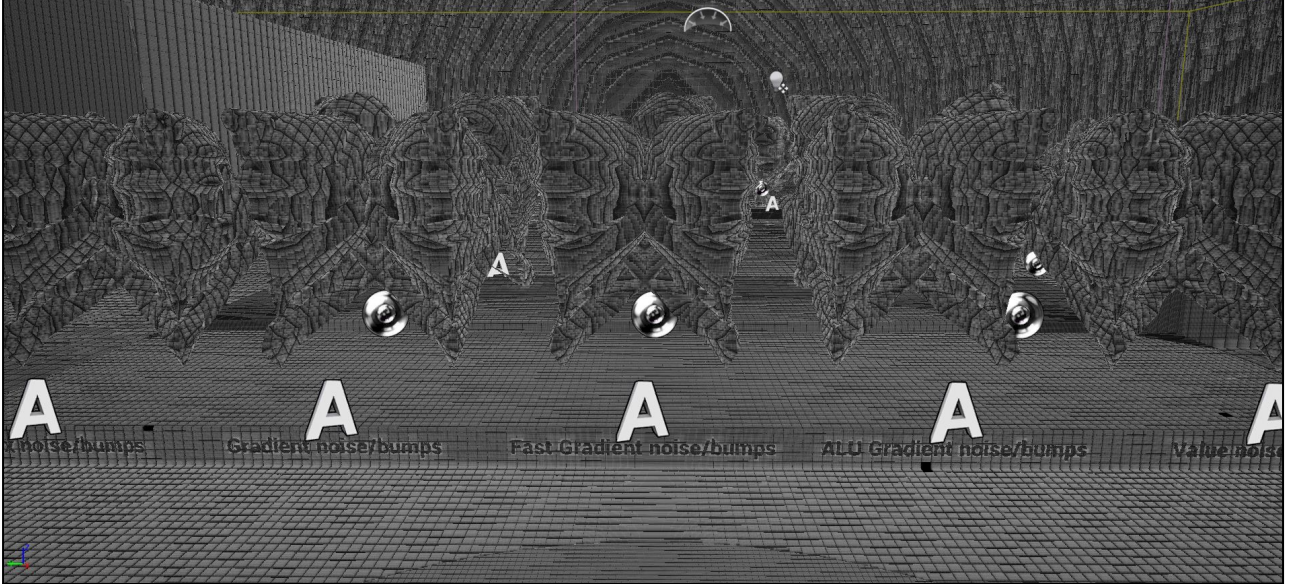
Rect list proved to be faster than cubes.

Grid gaps where due to mismatch between rasterizer position and ray position that I never tracked down.

It's important to note that these tests did not factor in everything they should.

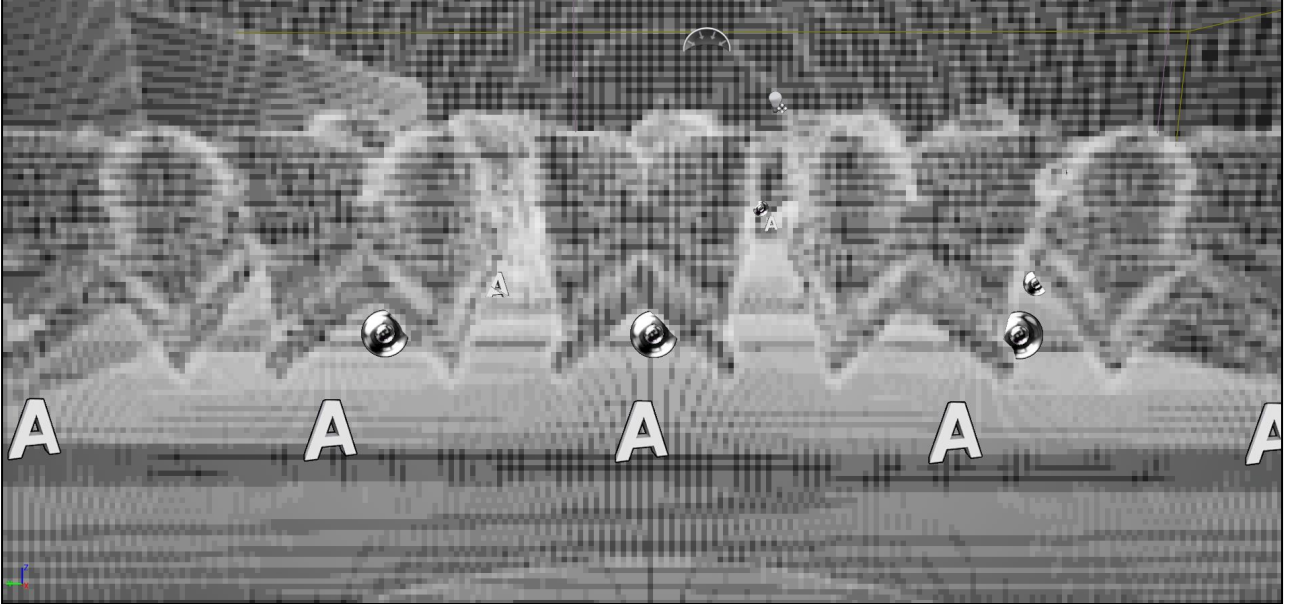
I only was timing the drawing of the visible bricks.

Voxel prototype



This image scatters bricks to screen space tile lists
And then the pixels run through the list of bricks in that tile all in compute.
Notice no more gaps.

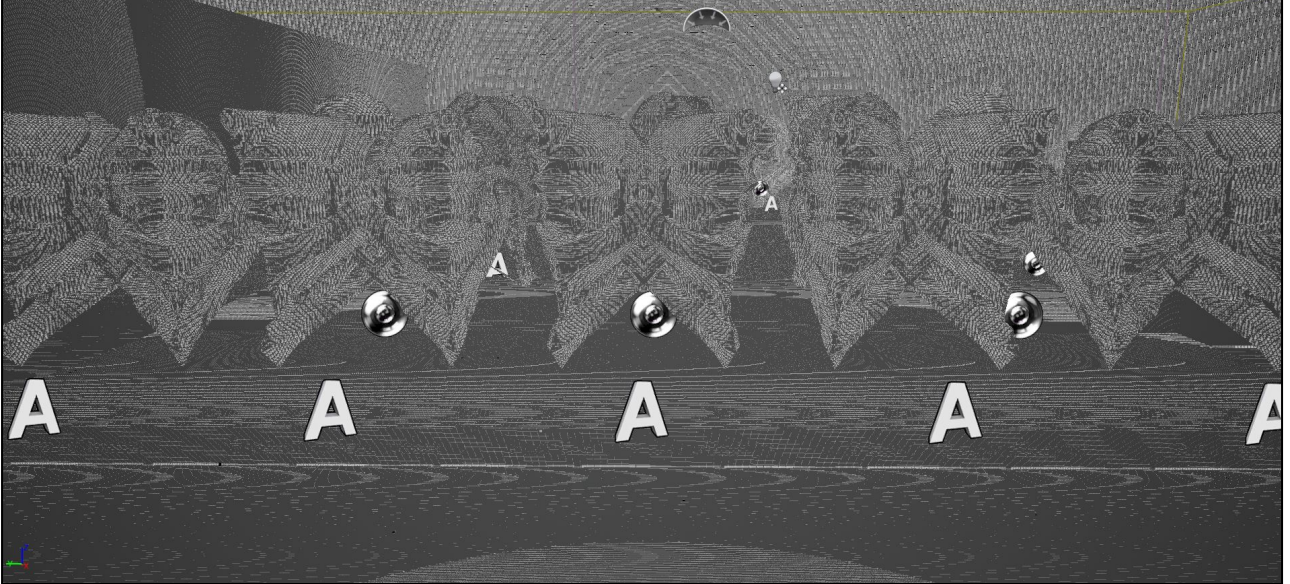
Voxel prototype



Here's the number of bricks per tile.

This was slightly faster than rasterizing bricks but not good enough in my opinion.

Voxel prototype

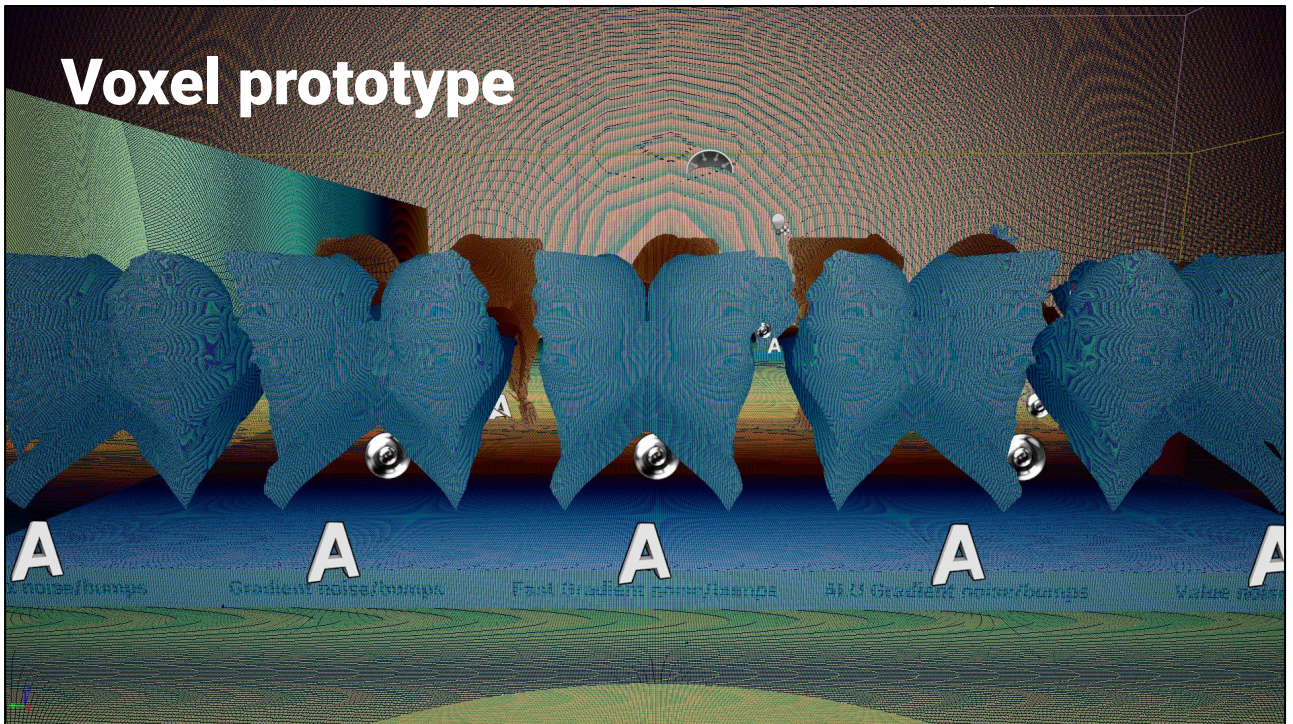


The hierarchical DDA was difficult to optimize.

Divergence was a problem with nearly every step having some pixel change levels.

So I tried ditching the 2 levels and scattering the 4x4x4 bricks.

It's a bit faster but given I wasn't accounting for the time to cull the bricks, significantly decreasing the granularity isn't exactly fair.



Well if more scatter is faster lets go full scatter.

Here is point based rendering with 1 point per voxel. This is not as efficient as it could be since I didn't pack the points.

It is 64 threads per brick and a thread early outs if the voxel isn't occupied.

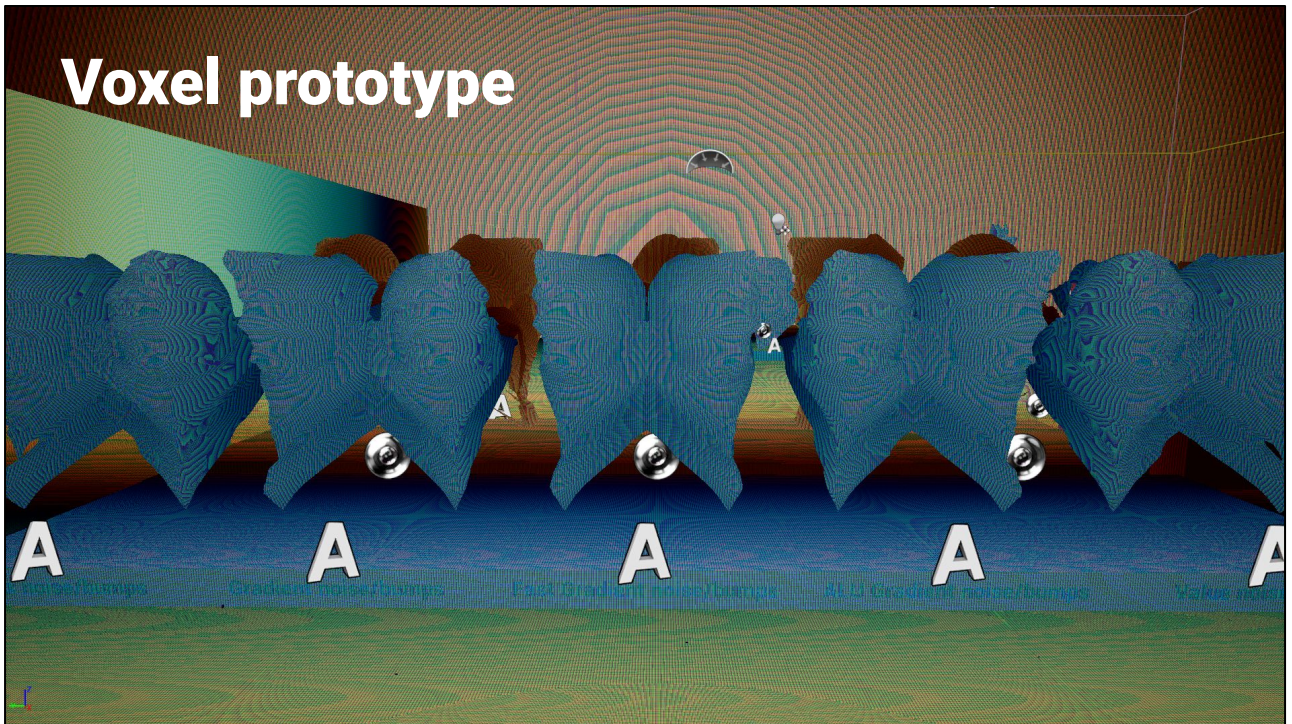
But holy crap is it fast!

This is 2ms on a PS4 Pro vs 9-10ms for 2 level brick tracing.

5x faster!

I had theorized a lot about point based rendering but this was actually the first point I had a prototype of it with even semi-representative performance.

Quality isn't comparable to the other voxel tests though. Notice the holes.



Ok well then lets fill in all the pixels that voxel should cover instead of just the closest to the center.

Project the cube to a screen space rectangle.

Iterate through the potentially covered pixels.

Do ray vs box intersection test.

Atomic write the pixel if it hits.

Basically a software box rasterizer in compute.

This fixes the quality problems but is considerably more expensive.

2 and half times slower than points.

I blame it on the overdraw.

On sloped surfaces voxels stack up. Even the flat ground had considerable overdraw.

It might be faster to draw the polys of the cube faces that are on the surface.

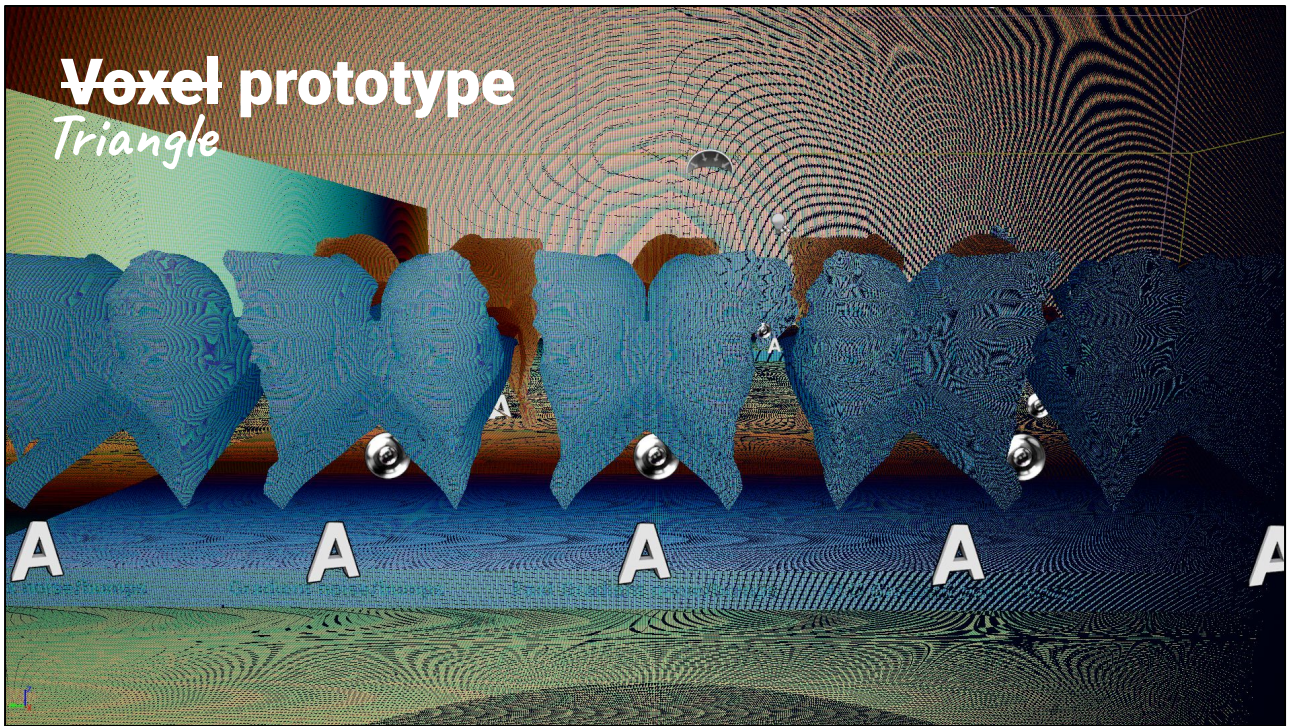
Besides how much more ALU is there to test triangle coverage compared to box coverage. Probably less.

Now I didn't think of it at the time but determining which faces of the voxel are exposed and which aren't is trivial.

Just check whether the neighbor in that direction is occupied.

But at the time my conclusion was if the exposed surface of these voxel cubes was turned into triangles I bet I could rasterize those faster.

So its finally time to try a micropoly software rasterizer.



So now 2 triangles rasterized per voxel.
It's only 70% slower than the blazing fast point scatter version.
1 point vs 2 triangles!

Lastly, let's try comparing to hardware rasterization.
Is my software rasterizer really fast or is the voxel stuff just really slow?

Holy shit!
My software rasterizer is 16X faster than hardware!

The comparison in the final form of Nanite didn't prove to be this large, at least not with real content.
For this stress test it still might. I don't know.

Time to pivot



It is clearly time to pivot.

Voxels have too many hard problems remaining.

My velocity in solving them is far too low.

I believe it would need multiple years of compounding research and industry experience to be able to replace explicit surfaces completely.

We aren't there yet and even if we were it's unclear if it would be better.

I learned a big lesson with my foray into voxels.

Listen to your gut.

I never expected this to work and I only explored it to prove it didn't.

It is nearly impossible to prove something can't be done. Don't do this.

Every failure can be followed with, well maybe if I changed this thing. There's no end to the what ifs.

While I wasted far more time on them than I should have I at least walked away with the thing I was seeking.

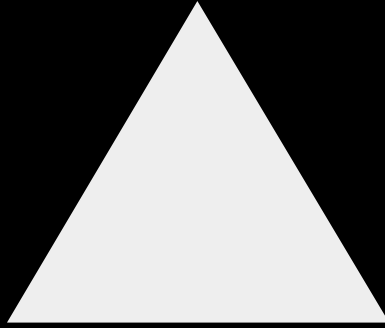
I had my solution to pixel scale detail. Software rasterization.

I probably could have arrived there far faster had I focused on the paths I predicted were most likely to succeed.

But regardless it was progress.

Triangles.

- Foundation of computer graphics for good reason



So, my conclusion.

The cubic scaling of volume data is too expensive.

Boundary representations are required even with voxels, just to manage the data size.

That is essentially what only storing the narrowest band data is.

I liked what Ryan Schmidt said in the panel about UVs and meshes.

I'm summarizing a bit but he said essentially "Voxels aren't the 3d analog of pixels.

Triangles are."

I'd add an asterisk to that and say, "for surfaces".

The most elemental, atomic unit of surface area in 3d space isn't a surfel, it's a triangle.

The minimum number of points to bound a 2d area is 3, forming a triangle.

The minimum number of points to represent a 3d plane is 3, forming a triangle.

Points themselves have no area or volume. They need connectivity to represent a continuous surface.

Once you connect them you have a triangle mesh.

Every surface can be turned into triangles.

And all I ever cared about was surfaces.

How to predict behavior

Without a full implementation



It's worth taking a moment here to note that up until this point
Only a tiny fraction of the algorithms I explored did I **implement** any portion of
Of those I did I only built simple prototypes testing a small fraction of the total
algorithm.

When the possibility space is vast efficiently navigating it is essential.

Simulate on paper

- What area is the biggest concern?
 - “Fail fast”
- Find closest existing analogs to draw data from
 - These might be others work
- Fermi estimation
- Sketch the algorithm in terms of what operations it needs to do
 - Compare to closest existing analogs for relative predicted costs
- Use to predict speed of light



Do as much simulation of options mentally or on paper.

Consider the edge cases.

Consider it at scale.

Start with the aspect that is most likely to fail.

This is classic “fail fast”.

Estimating is an important skill.

It surprises me to no end how often very technical people make feasibility claims or comparisons based on nothing more than gut feelings when there are often simple calculations that could be done.

There are a lot of ways to get data to feed those estimates.

Very often the fastest way to get initial numbers is to find an existing analog that can be plugged into an equation.

This might not even come from your work.

I've made estimates for Nanite behavior using RAGE videos that had debug stats in them and Dream's reported points per second.

I predicted the behavior of potential approaches without writing a line of code.

Write code like a scientist

- Write only what is necessary to answer questions
- Might be as simple as dumping data you didn't previously have to feed the estimates
- May look like nothing
- May do nothing but help you visualize



When you need to write code, only write what you need to get data to answer the important questions.

Code like a scientist.

Have a variable in your equation and you don't have any idea what to plug in?

Gather that data.

Or write something to help you visualize where a problem case might be.

When is it time to commit to a path?

- Intuition built from experience
- Confidence in your predictions
 - Have solutions to the biggest problems that feel like they will work
- Anything that has certainty
 - If you know you'll need to write some component regardless
 - The whole point is to avoid investing in branches that might be a waste
 - Completing that branch may provide data for remaining unknowns



When is it time to commit to a path and heavily invest in building it?
Intuition built from experience unfortunately.

But what that feels like is that your confidence in your predictions has grown to the point where you are fairly optimistic about the outcome.
You've identified what you think are solutions to the biggest problems.

There's one case that is easy to commit to.
That's any time there is certainty.

The whole point of navigating efficiently is avoiding investing in branches that are useless.

If you know that branch needs to be travelled regardless, do so because it can teach you something.

What can you delay solving?

- “Seems solvable”
- Intuition built from experience
- These should feel like leaf problems
- Based on how much room there is to maneuver
 - Can you easily think of a possible solution?
 - Are there alternatives which also seem plausible?



Do you need to be confident about every aspect before proceeding?

Have every detail solved in your head first?

Of course not!

So what can you delay solving?

When describing an idea for an approach to others I'll often say, I don't know how this bit works but that “seems solvable”.

What I mean is I am reasonably confident that there is a solution to that detail and whatever it is is unlikely to impact the viability of the high level idea.

This again comes from

Intuition built from experience

but usually is based on how flexible it feels.

How many potential options are there?

The more there are more likely one of them works.

Many unknowns still

- How about big triangles?
 - Tessellate?
 - HW raster?
 - There's no way I could get SW and HW to perfectly match right?
- Popping?
 - TAA enough to hide?
- Memory to hit 1 pixel?
 - Estimates swung from 512MB to 2GB depending on how I calculated it
- Is visbuffer only efficient for large triangles?
 - Due to cache hits



Sometimes the details don't comfortably "seem solvable".

Sometimes they are scary unknowns that could hugely impact the overall viability but there is no way to know without a working implementation.

Here's a few I remember being concerned about for Nanite that luckily worked out.

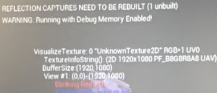


So I built a prototype
Its fast
Really fast!

Initially everything was drawing down both paths so the only thing that is Nanite here is that grey picture in picture.

Nothing but a debug view of the visibility buffer but you can tell that cluster occlusion culling was working.

Just after I get the cluster LOD selection, culling, and triangle rasterization working end to end
Rune joins the project and I have a partner to work with for the first time.
And it was awesome.



With Rune's help we delivered something really exciting.
This is a shot of that 2019 behind closed doors GDC demo running on a PS4 Pro.

But none of that is important.
I knew what this proved.
I finally cracked it.
I wanted to immediately tell the world but I couldn't.

Managing fear

- Fear of failure
- Fear of success too late



Speaking of...

It is worth touching on the emotional side to this.

There were many moments throughout this that were rough.

I have so much of my self wrapped up in this work that it's success or failure is tied to my sense of self.

It's not good but it's real and it might happen to you.

Once I had gotten my opportunity to work full time on this I thought this is my chance.

If I fuck this up I'm not going to get it again.

What if I was wrong? I've talked this up for so long, I've thought about and invested myself in this problem for so long.

Maybe I'm wrong and I can't solve it.

What do I do after this?

Once I had it working it was the opposite fear.

I became terrified that someone else was going to beat me to the punch.

Funny enough, key people I was scared of ended up joining the team.

Engineering

- 1 year to our first public demo
- 3 years until UE5.0 release

This is clearly not the whole story.

Graham joined the team
As did Andrew and Ola to work on shadows.

It took another year of hard engineering work by this team of amazing people to go from that early prototype to our first public demo.
It took another 2 years after that to get to a solid shippable version for UE 5.0's release.

Got lucky?



Looking back now, was it luck that it worked?

Was Nanite, like I perceived megatexture many years before, an insane gamble? The viability of which was based on numerous unknowns, each one an independent bet, all of which needed to succeed.

Was it more likely to have failed than succeeded?
Hopefully by seeing the process I went through it is not as mysterious.

I won't deny luck played some role.
For example I was lucky to be given the ample opportunity to pursue this research. But that opportunity would have amounted to nothing had I not been prepared with years of ideas and knowledge that I was ready to jump on when given the chance.

I don't know whether I'm unusual in this regard but this isn't the only background thread I've had going over the years.
I have many problems I have thought heavily on but have yet to get time to work on. If and when opportunity knocks I'll be prepared.

How accurate must predictions be?



The opposite of chance is foresight.
How accurate must predictions be?

When I was a child I wondered how my parents could stop the car so precisely.
When a traffic light turned red they would press the brake pedal and the car would slow from whatever speed they happened to be going, to a stop at a line on the road or within feet of the car in front of them.

How did they know exactly the right pressure to hit the target every time?
How could they be so precise with their foot, within what must have been fractions of a millimeter positioning of the brake pedal?
And the initial conditions are different every time!

Of course I know now that that isn't how it works.
It isn't constant pressure on the pedal. Every millisecond we make new measurements, factor those into new predictions, and adjust the pressure.

Hindsight

- Artist highpolys for baking normal maps weren't UVed
 - Artists textured lowpoly
 - UV unwrap tools don't scale
 - DCCs can't handle multi-million poly meshes
- Significant temporal upsampling would be prevalent
 - Nanite scales with **output** resolution pixels
 - Tracing scales with **render** resolution pixels
- Need incoherent visibility
 - Other data structures are acceptable but what is the artist story then?
- Noisy vertex attributes weigh heavily on perceptual error
 - Coupling the frequency of positions and attributes bloats cost
- Unique geometry is required in the distance
 - Object merging needed closer than expected
 - Instance transforms don't LOD
 - Occlusion with kitbashing degrades severely in the distance



We adjusted countless details of the Nanite design as they came up.

But there were also big decisions that I made based on faulty conclusions,
or with a lack of understanding of workflow,
or with blindspots in future needs.

I'm not perfect.

It's hard to say how things would have turned out.

Journey **Through** Nanite



[pause]

That's my story. It has been a **long** journey
I hope you've learned something about the problem I've been trying to solve all these years.
And maybe you've learned something from my process of trying to solve it.

I titled this talk the "Journey **to** Nanite" as that is the story I have to tell.
But the reality is this problem isn't solved yet.
Nanite approaches the dream but does not completely fulfill it.
I'm not done working on it and I don't think you should be either.
I encourage you to attack what is remaining along with me, because as I said at the beginning,

In my opinion there is no bigger problem in my field than making 3d content cheaper to create,
Than making the art of creating it more accessible to more creators.
And knocking down every last impediment left to artists manifesting their vision.

But I also encourage you to answer Hamming's question for yourself: "What are the most important problems in your field?"
and "Why aren't you working on them?"
if you aren't already.

Thanks

- Nanite co-authors
 - Rune Stubbe
 - Graham Wihlidal
- Many great geometry discussions
 - Timothy Lottes
 - Ashley Welch
 - Andrew Scheidecker
 - Jordan Walker
 - Martin Mittring
 - Tim Sweeney
 - Alex Evans
 - David Hill
 - Michael Lentine
 - Matthäus Chajdas
 - Jerome Platteaux



I of course need to thank everyone that has worked on Nanite.

But I'd also like to call out many folks I've discussed this problem with over the years.

References

Virtual texturing:

1. van Waveren 2012, "Software Virtual Textures" <https://mrelusive.com/publications/papers/Software-Virtual-Textures.pdf>
2. Barrett 2008, "Sparse Virtual Textures" <https://silverspaceship.com/src/svt/>

Voxels:

3. Carmack 2007, "Quakecon 2007 keynote"
4. Carmack 2008, "John Carmack on idTech6, Ray Tracing, Consoles, Physics and More" <https://pcper.com/2008/03/john-carmack-on-id-tech-6-ray-tracing-consoles-physics-and-more/>
5. Olick 2008, "Next Generation Parallelism In Games" <https://www.jonolick.com/uploads/7/9/2/1/7921194/olick-current-and-next-generation-parallelism-in-games.pdf>
6. Laine and Karras 2010, "Efficient Sparse Voxel Octrees" https://research.nvidia.com/sites/default/files/pubs/2010-02_Efficient-Sparse-Voxel/laine2010i3d_paper.pdf
7. Crassin 2011, "GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes" http://maverick.inria.fr/Publications/2011/Cra11/CrassinThesis_EN_Web.pdf
8. Yoon et al. 2006, "R-LODs: Fast LOD-based Ray Tracing of Massive Models" <https://gamma.cs.unc.edu/RAY/RL0D.pdf>
9. Chajdas et al. 2014, "Scalable rendering for very large meshes" <http://wscg.zcu.cz/wscg2014/Full%5C17-full.pdf>
10. Novák and Dachsbacher 2012, "Rasterized Bounding Volume Hierarchies" <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.306.4787&rep=rep1&type=pdf>
11. Reichl et al. 2012, "Hybrid Sample-based Surface Rendering" <https://www.in.tum.de/cg/research/publications/2012/hybrid-sample-based-surface-rendering/>
12. Áfra 2013, "Interactive Ray Tracing of Large Models Using Voxel Hierarchies" <https://voxelium.wordpress.com/2012/01/31/interactive-ray-tracing-of-large-models-using-voxel-hierarchies/>

SDFs:

13. Frisken et al. 2000, "Adaptively Sampled Distance Fields: A General Representation of Shape for Computer Graphics" <https://www.merl.com/publications/dpcs/TR2000-15.pdf>
14. Bastos and Celes 2008, "GPU-accelerated Adaptively Sampled Distance Fields" https://www.researchgate.net/publication/4344580_GPU-accelerated_Adaptively_Sampled_Distance_Fields
15. Evans 2015, "Learning from Failure: a Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game" https://advances.realtimerendering.com/s2015/mmalex_siograph2015_hires_final.pdf
16. Aaltonen 2018, "GPU-based clay simulation and ray-tracing tech in Claybook" https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2018/presentations/Aaltonen_Sebastian_GPU_Based_Clay.pdf



References

Subd

17. Catmull and Clark 1978, "Recursively generated B-spline surfaces on arbitrary topological meshes" https://people.eecs.berkeley.edu/~sequin/CS284/PAPERS/CatmullClark_SDSurf.pdf
18. Nießner et al. 2012, "Feature Adaptive GPU Rendering of Catmull-Clark Subdivision Surfaces" <https://niessnerlab.org/papers/2012/3feature/niessner2012feature.pdf>
19. Nießner and Loop 2012, "Patch-based Occlusion Culling for Hardware Tesselation" <http://www.niessnerlab.org/projects/niessner2012patch.html>
20. Brainerd et al. 2016, "Efficient GPU Rendering of Subdivision Surfaces using Adaptive Quadrees" <http://www.niessnerlab.org/projects/brainerd2016efficient.html>

Displacement

21. Gu et al. 2002, "Geometry images" <http://hhoppe.com/proj/gim/>
22. Sander et al. 2003, "Multi-chart geometry images" <https://hhoppe.com/proj/mcgim/>
23. Purnomo et al. 2004, "Seamless texture atlases" <https://www.cs.jhu.edu/~cohen/Publications/sta.pdf>
24. Niski et al. 2007, "Multi-grained Level of Detail Using a Hierarchical Seamless Texture Atlas" <https://www.cs.jhu.edu/~cohen/Publications/HSTA.pdf>
25. Gobbetti et al. 2012, "Adaptive Quad Patches: an Adaptive Regular Structure for Web Distribution and Adaptive Rendering of 3D Models" <http://vcg.isti.cnr.it/Publications/2012/GMBGD12/>
26. Lee et al. 2000, "Displaced subdivision surfaces" <http://hhoppe.com/proj/dss/>
27. Lee et al. 1998, "MAPS: Multiresolution Adaptive Parameterization of Surfaces" <https://cm-hell-labs.github.io/who/wim/papers/sig98/>
28. Khodakovsky et al. 2003, "Globally Smooth Parameterizations with Low Distortion" <http://multires.caltech.edu/pubs/global.pdf>
29. Maximo et al. 2013, "Adaptive multi-chart and multiresolution mesh representation" https://www.researchgate.net/publication/259096285_Adaptive_multi-chart_and_multiresolution_mesh_representation
30. Liu et al. 2017, "Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution" <https://cracl.cs.qmu.edu/seamless/>
31. Liu et al. 2020, "Neural Subdivision" <https://arxiv.org/pdf/2005.01819.pdf>

References

Points

30. Rusinkiewicz and Levoy 2000, "QSplat: A Multiresolution Point Rendering System for Large Meshes" <http://graphics.stanford.edu/software/qsplat/>
31. Gobbetti and Marton 2005, "Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms" <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.2062&rep=rep1&type=pdf>
32. Goswami et al. 2010, "High Quality Interactive Rendering of Massive Point Models using Multi-way kd-Trees" <http://www.crs4.it/vic/data/papers/pg2010-multi-way-kdtrees.pdf>
33. Schütz et al. 2020, "Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures" https://publik.tuwien.ac.at/files/publik_282669.pdf
34. Schütz et al. 2021, "Rendering Point Clouds with Compute Shaders and Vertex Order Optimization" <https://arxiv.org/abs/2104.07526>
35. Marroquim et al. 2007, "Efficient Point-Based Rendering Using Image Reconstruction" <http://graphics.tudelft.nl/~marroquim/publications/pdfs/marroquim-pbq2007.pdf>
36. Zhang and Pajarola 2007, "Deferred blending: Image composition for single-pass point rendering" <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.4705&rep=rep1&type=pdf>

GPU driven culling

37. Haar and Aaltonen 2015, "GPU-Driven Rendering Pipelines" http://advances.realtimerendering.com/s2015/aaltonenhaar_siqgraph2015_combined_final_footer_220dpi.pdf
38. Wihlidal 2016, "Optimizing the Graphics Pipeline with Compute" https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/29204330/GDC_2016_Compute.pdf

References

Decoupled materials

44. Burns et al. 2010, "A Lazy Object-Space Shading Architecture With Decoupled Sampling" http://graphics.stanford.edu/~kayvonf/papers/burns_shading_hpg10.pdf
45. Fatahalian et al. 2010, "Reducing Shading on GPUs using Quad-Fragment Merging" http://graphics.stanford.edu/papers/fragmerging/shade_sig10.pdf
46. Hillesland and Yang 2016, "Texel Shading" http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/TexelShading_EG2016_AuthorVersion.pdf
47. Burns and Hunt 2013, "The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading" <http://jcgt.org/published/0002/02/04/>

View dependent irregular meshes

48. Ulrich 2002, "Chunked LOD" <http://tulrich.com/geekstuff/chunklod.html>
49. Cignoni et al. 2003, "BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization" <http://vcg.isti.cnr.it/publications/papers/bdam.pdf>
50. Yoon et al. 2004, "Quick-VDR: Interactive View-Dependent Rendering of Massive Models" <http://gamma.cs.unc.edu/QVDR/>
51. Cignoni et al. 2004, "Adaptive TetraPuzzles: Efficient Out-of-Core Construction and Visualization of Gigantic Multiresolution Polygonal Models" <http://www.crs4.it/vic/data/papers/sig2004-tetrapuzzles.pdf>
52. Cignoni et al. 2005, "Batched Multi Triangulation" <http://publications.crs4.it/pubdocs/2005/C6GMPS05a/ieeeviz2005-gpuml.pdf>
53. Ponchio 2008, "Multiresolution structures for interactive visualization of very large 3D datasets" http://vcg.isti.cnr.it/~ponchio/download/ponchio_phd.pdf
54. Sander and Mitchell 2005, "Progressive Buffers: View-dependent Geometry and Texture LOD Rendering" <http://www.cse.ust.hk/~psander/docs/progbuffer.pdf>
55. Sugden and Iwanicki 2011, "Mega Meshes: Modelling, rendering and lighting a world made of 100 billion polygons" http://miciwan.com/GDC2011/GDC2011_Mega_Meshes.pdf
56. Hu et al. 2010, "Parallel View-Dependent Level-of-Detail Control" <http://www.cse.ust.hk/~psander/docs/pmstreami.pdf>
57. Derzapf et al. 2010, "Parallel View-Dependent Out-of-Core Progressive Meshes" <http://www.mathematik.uni-marburg.de/~derzapf/public/2010/Parallel%20View-Dependent%20Out-of-Core%20Progressive%20Meshes.pdf>
58. Derzapf and Guthe 2012, "Dependency Free Parallel Progressive Meshes" <http://www.mathematik.uni-marburg.de/~derzapf/public/2012/Dependency%20Free%20Parallel%20Progressive%20Meshes.pdf>

References

Rasterization

- 61. Fatahalian et al. 2009, "Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur" <http://graphics.stanford.edu/papers/mprast/>
- 62. Brunhaver et al. 2010, "Hardware Implementation of Micropolygon Rasterization with Motion and Defocus Blur" <http://graphics.stanford.edu/papers/hwrast/>

Miscellaneous

- 63. Walt Disney Animation Studios 2018, "Moana Island Scene (v1.1) [Data set]" <http://technology.disneyanimation.com/islandscene/>
- 64. Persson 1999, "Behind the Scenes of Messiah's Character Animation System" <https://www.gamedeveloper.com/programming/behind-the-scenes-of-messiah-s-character-animation-system>
- 65. Forsyth 2008, "Knowing which mipmap levels are needed" <https://tomforsyth1000.github.io/blog/wiki.html#%5B%5Bknowing%20which%20mipmap%20levels%20are%20needed%5D%5D>
- 66. Duchaineau et al. 1997, "ROAMing Terrain: Real-Time Optimally Adapting Meshes" <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.456.7772&rep=rep1&type=pdf>
- 67. Dick et al. 2009, "Efficient Geometry Compression for GPU-Based Decoding in Realtime Terrain Rendering" <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.641.5362&rep=rep1&type=pdf>
- 68. Hunt 2017, "Real-Time ray casting for virtual reality" https://www.highperformancegraphics.org/wp-content/uploads/2017/Hot3D/HPG2017_RealTimeRayCasting.pptx
- 69. Chen et al. 2009, "Lattice-Based Skinning and Deformation for Real-Time Skeleton-Driven Animation" https://www.researchgate.net/publication/224264785_Lattice-Based_Skinning_and_Deformation_for_Real-Time_Skeleton-Driven_Animation
- 70. Hamming 1986, "You and Your Research" <https://www.cs.virginia.edu/~robins/YouAndYourResearch.html>
- 71. Museth 2013, "VDB: High-Resolution Sparse Volumes with Dynamic Topology" <https://www.openvdb.org/>
- 72. Karis et al. 2021, "A Deep Dive into Nanite Virtualized Geometry" <http://advances.realtimerendering.com/s2021/index.html>



