

Nanite

A Deep Dive

Brian Karis
Rune Stubbe
Graham Wihlidal



SIGGRAPH 2021

ADVANCES IN REAL-TIME RENDERING IN GAMES course



UNREAL
ENGINE

Hello, I am Brian Karis, Engineering Fellow at Epic Games.
Today we are going to take a deep dive into Nanite, UE5's new virtual geometry system.

The Dream

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

Thinking back on the things I've worked on over the years the ones that had the largest impact were those that fundamentally changed how art was made.

Many years ago I got to see the impact that a virtual texturing system could have on an art team.

How freeing it was for the artists to use giant textures wherever they wanted and not to have to worry about memory budgets.

Ever since I've dreamt of the day where we could do the same for geometry. The impact could be even greater.

No more budgets for geometry would mean
No more concern over
Polycounts, draw calls, or memory.

Without those limitations artists could directly use film quality assets without wasting any time manually optimizing for use in real-time.

It's **ridiculous** how much time is wasted in optimizing art content to hit framerate.

All this should hold true for set dressing as well.

Artists should be able to:

Place as many meshes they want, however they want.

It should be up to them how best to build a scene

And of course, all that with no loss in quality!

What's the point of running fast if it no longer looks like what the artist originally created?

=====

Bonus:

How sad is it that once a game is released, the art team posts a ton of their work to artstation, and a huge amount of what was posted are high polys and offline rendered images of the models they proudly made but the player never sees look like that?

I heard an interesting perspective a few years ago from an artist with an offline film background who was experimenting with real-time rendering and Unreal for their last short film project.

He said that while there was a ton of time saved from render time iterations, the extra work to optimize assets for real-time made the whole thing a wash.

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 wildly because the process is less arcanelly technical, will reap massive dividends.

Getting on my soapbox for a second, I think as an industry we should be working more on how to make high fidelity games cheaper than we are.

Reality

- **MUCH** harder than virtual texturing
 - Not just memory management
 - Geometry detail directly impacts rendering cost
 - Geometry isn't trivially filterable

The reality is this is a **MUCH** harder task than virtual texturing
Because it is not just a memory management problem.

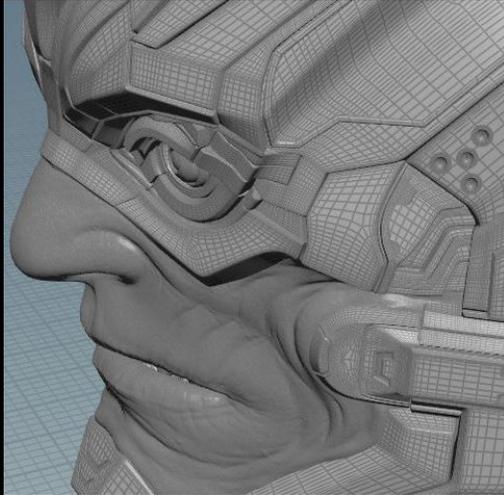
Geometry detail directly impacts rendering cost unlike textures.
Geometry isn't trivially filterable like textures either.

Options?

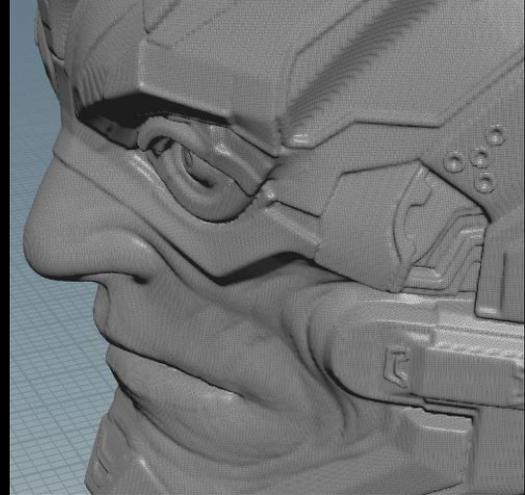
So what are our options?

Voxels?

Source mesh



Resampled to voxels



Voxels and implicit surfaces have a lot of potential advantages and are the most discussed direction to solve this problem.

This is a 2M poly bust
Resampled to 13M narrow band SDF voxels

We've already accounted for sparsity in that no empty voxel is stored and yet with 6 times the amount of data
It looks blobby

The reason for this is voxelization is a form of uniform resampling and
Uniform resampling means loss

This is the 3D equivalent of converting vector graphics into pixel graphics
For organic uniformly sampled meshes like the sculpted skin in this example,
resampling likely won't be very destructive.

For hard surface modelling on the other hand it can be **extremely** so, as you can see.

In essence, the chief issue with voxels is a data size problem
Maximum sparsity is needed to keep data size small but we can't sacrifice ray casting
performance in the process
And the data structure needs to be super adaptive to get sharp edges but not waste
samples where its smooth.

And even with all that the finest resolution stored still won't be what was authored
No matter what we are resampling.

Do we have to store the original mesh too and draw that if you get close enough?

Voxels?

- 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
- Never ending list of hard problems

It's also important to note that:

We are not interested in completely changing all CG workflow

We need to support importing of meshes authored anywhere. We don't have control over how geometry is authored.

Those meshes will have UVs and tiling detail maps

Although UV mapping is universally hated by artists it is an incredibly powerful tool for procedural texturing surfaces.

Voxel colors are not an acceptable replacement nor is triplanar projection.

We are only looking to replace meshes, not textures, not materials, not all the tools associated with authoring those things.

With that in mind:

Voxels with UVs?

Maybe but what to do about UV seams?

Not to mention countless other problems such as:

Features which vanish as signed distance fields when they are thinner than a voxel

Or leak attributes from one side to the other when they are less than a few voxels.

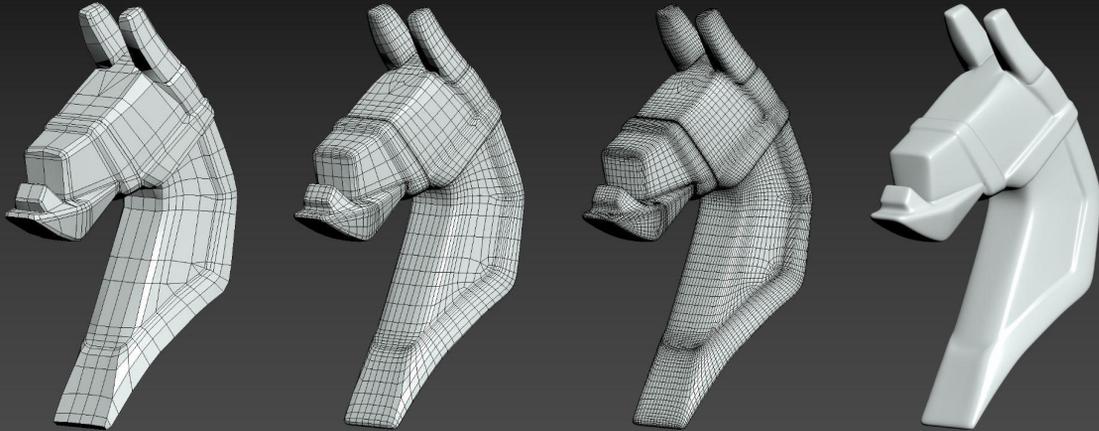
Or how the hell to animate them

All this **might** be possible but

my conclusion was that it would need **many 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.

Subdivision surfaces?



How about subdivision surfaces?
They can get infinitely smooth right?

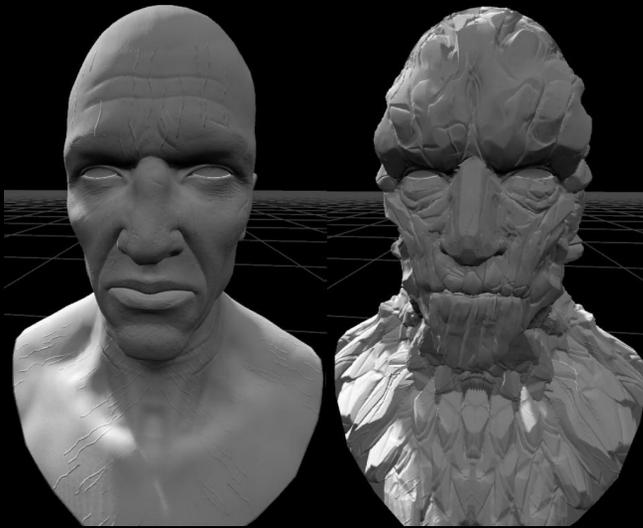
Subdivision by definition is amplification only
Great for up close but
Doesn't get simpler than base cage

I spoke with our artists and
The cage is very often higher than typical game low polys

In film it's an order of magnitude worse.

We need artist authoring choices disconnected from rendering cost.

Displacement maps?



© 2015, Yasutoshi Mori

SIGGRAPH 2021

SIGGRAPH 2021 ADVANCES IN REAL-TIME RENDERING IN GAMES course

UNREAL ENGINE

How about displacement maps?

We could capture displacement like we do normal maps now.

With vector displacement the low poly could be even lower right?

In some cases sure,

But what's the low poly for a chain like the image on the right?

How would you displace something simple into that chain?

You can't really because displacement can't increase the genus of a surface.

That's fancy way of saying you can't displace a sphere and turn it into a torus.

It's also worth noting, projecting to normal or displacement maps is a form of uniform resampling.

Works really well for organic surfaces that already are uniformly sampled

But can be really destructive to hard surface features if not very carefully controlled by the artist.

So great for up close and amplification but

Not good enough for general purpose simplification.

=====

Bonus:

Geometry images are essentially like vector displacement maps relative to the origin

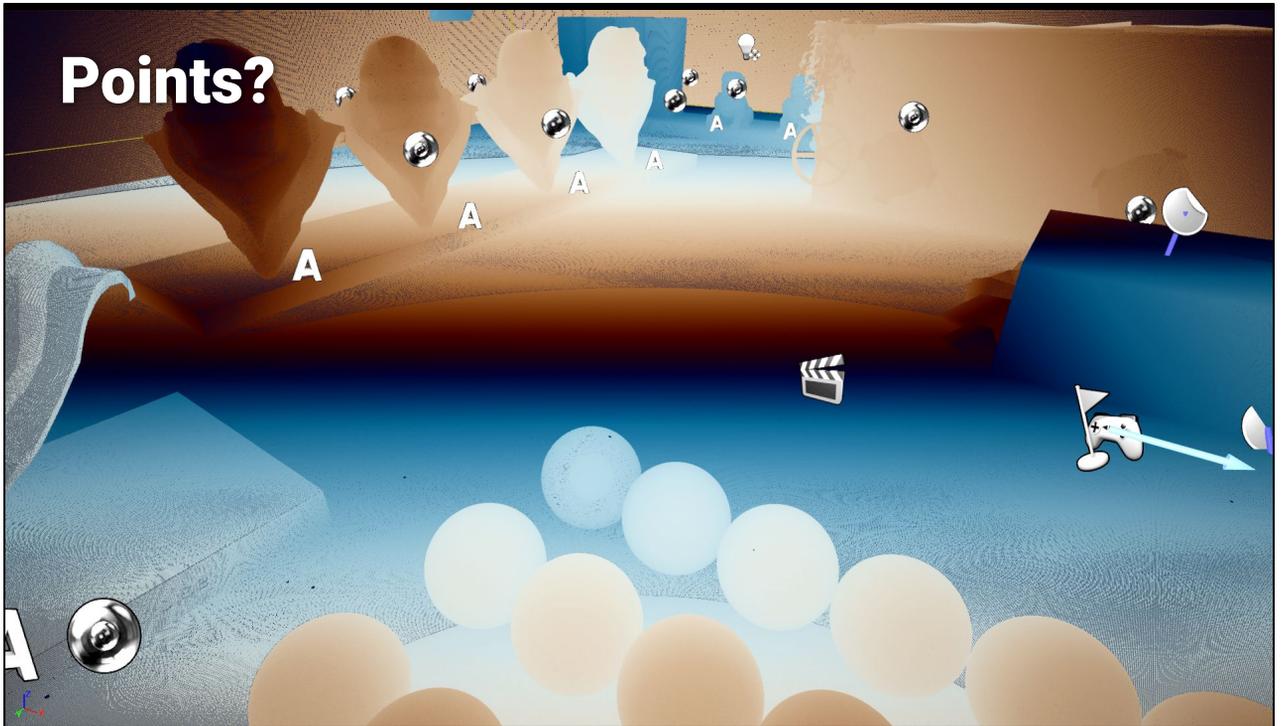
instead of another surface.

All the above issues are the same.

Technically the genus can be anything if the surface isn't closed to begin with.

The perimeter can be arbitrarily stitched creating different genus which was used in the original GIM paper but not really in a practical way due to texture stretch.

Also that doesn't really count as changing.



How about point based rendering?

Points are super fast to blast to the screen.

But unless you accept massive amounts of overdraw, points require hole filling
How do we know the difference between a small gap that should be there and a hole that should be filled?

It's impossible to know for certain without extra connectivity data. Also known as the index buffer in a triangle mesh.

=====

Bonus:

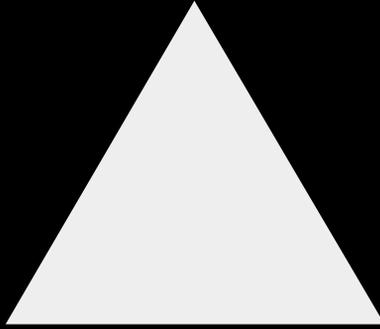
This seems an area ripe for ML which wasn't quite as popular at the time I was exploring this as it is now. Denoising is a form of sparse data interpolation and abstractly is quite similar to this problem.

Although that doesn't solve guaranteeing the connectivity, a trained NN could work well enough in practice, at least in a minimization context to be reliable.

That wouldn't solve maximization though, which requires connectivity so at least some portion of that solution would still need to be meshes.

Triangles.

- Foundation of computer graphics for good reason



I spent a very long time exploring other options and for our requirements have found no higher quality or faster solution than triangles.

There are good uses for these other representations but triangles are the core of Nanite.

If you can build an art style around the aesthetics of one of these other representations and embrace their pros and cons they could be incredibly valuable but Unreal can't **impose** an art style.

GPU Driven Pipeline

- Renderer now retained mode
 - GPU scene representation persists across frames
 - Sparsely updated where things change
 - All vertex/index data in single large resource
- Per view:
 - GPU instance cull
 - Triangle rasterization
- If only drawing depth the entire scene can draw with 1 DrawIndirect

Well if we are going to do triangles what does it take to build a state of the art triangle rendering pipeline?

Nothing novel here but let's quickly establish the state of the art and bring Unreal up to it.

As of a few years ago Unreal's renderer was refactored to be a retained mode design. We've expanded on this

A complete version of the scene is stored in video memory and persists across frames.

It is sparsely updated where things change.

In addition, all Nanite mesh data is stored in single large resources.

This means we can touch any of it at once without requiring bindless resources to do so.

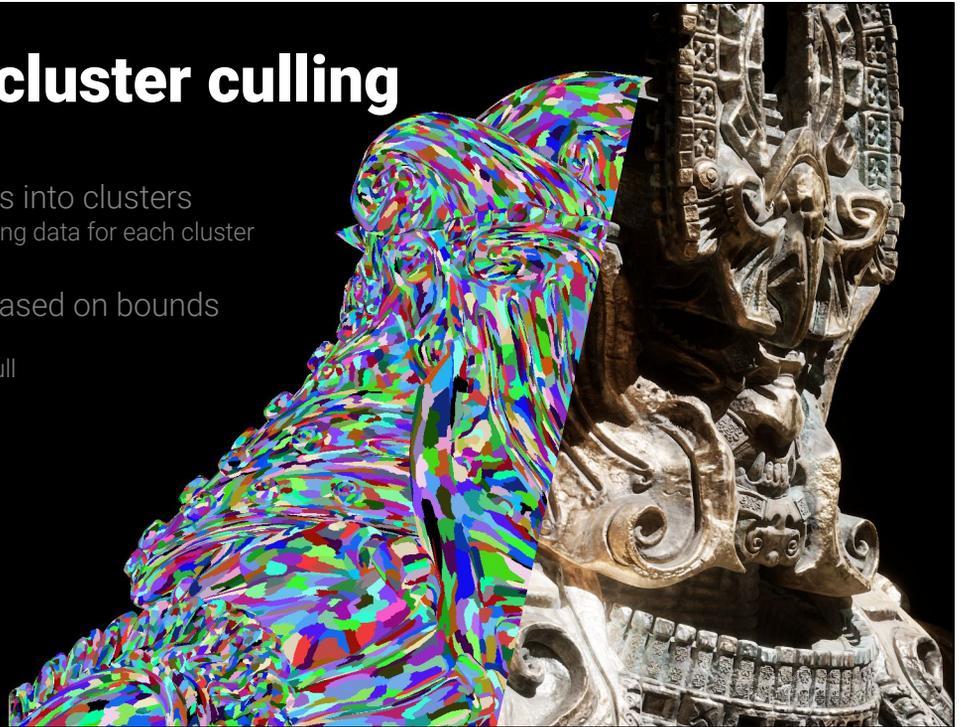
These are the necessary building blocks for a GPU driven pipeline.

For each view we can determine the visible instances in a single dispatch

And then if we are only drawing depth we can rasterize all the triangles with in a single DrawIndirect

Triangle cluster culling

- Group triangles into clusters
 - Build bounding data for each cluster
- Cull clusters based on bounds
 - Frustum cull
 - Occlusion cull



 SIGGRAPH 2021

We have all the benefits of GPU driven now but we are still doing a fair amount work for triangles that aren't visible.

So, lets add triangle cluster culling which trims out that unnecessary work.

To do that group triangles into clusters and build a bounding box for each cluster
Then we can cull the clusters based on their bounds

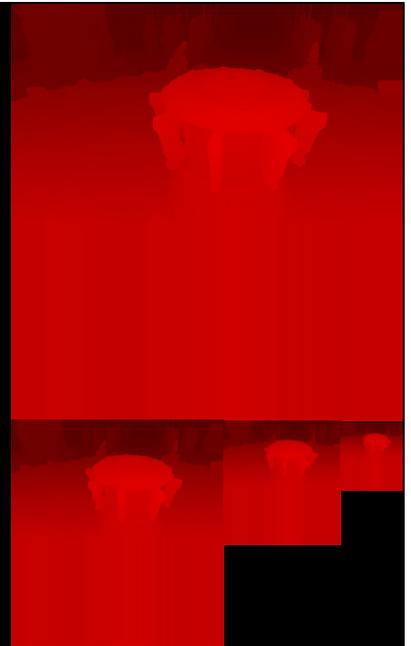
=====

Bonus:

We are all familiar with cluster culling made popular by [40] and, [41] but the idea goes way back, at least to 1996 [37]

Occlusion culling

- Occlusion cull against Hierarchical Z-Buffer (HZB)
- Calculate screen rect from bounds
- Test against lowest mip where screen rect $\leq 4 \times 4$ pixels



We cull against the frustum as well as occlusion cull.
We have found cone based backface culling is not typically worthwhile as almost all backfacing clusters are occlusion culled anyways.

Occlusion culling is done against a hierarchical Z-buffer I'll refer to as HZB.

Calculate a screen rect from the bounds, find the lowest mip where the rect is 4×4 pixels and test if it is occluded.

=====

Bonus:

The test is identical to [39].

We've seen improvements from using depth gradients of the bounding box instead of just the closest depth but are not currently using them.

Occlusion culling

- What HZB? Haven't rendered anything yet
- Reproject z-buffer from previous frame into current frame?
 - Needs hole filling to be useful
 - Not conservative

Ok, but how do we get an HZB to test against?
We haven't rendered anything yet.

Some games have tried to reproject the previous frame's z-buffer into the current frame.

This is always approximate and not conservative.

Two pass occlusion culling

- Visible objects last frame are likely still visible this frame
 - At least a good choice for occluders
- Two pass solution
 - Draw what was visible in the previous frame
 - Build HZB
 - Draw what is visible now but wasn't in the last frame
- Almost perfect occlusion culling!
 - Conservative
 - Only falls apart under extreme visibility changes

The core assumption there is a good one though:

What was visible last frame is very likely to be what is visible this frame.

So last frame's visible surfaces represented by the depth buffer are a good choice for occluders.

But why try to reproject the depth buffer when we can instead reproject the **geometry we rendered** to the depth buffer?

So, draw what we determined was visible last frame and use that to occlude anything new

This is two pass occlusion culling

- Draw what was visible in the previous frame
- Build HZB from that
- Test the HZB to determine what is visible now but wasn't in the last frame and draw anything that's new

With that we now we have a state of the art triangle rendering pipeline. What next?

Obviously we want to render more than a depth only pass.

=====

Bonus:

The first GPU driven occlusion culling I'm aware of is "March of the Froblins" [38]

The first two pass occlusion I'm aware of is from "Patch-based Occlusion Culling for

Hardware Tessellation" [19]

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

- Eliminate:
 - Switching shaders during rasterization
 - Overdraw for material eval
 - Depth prepass to avoid overdraw
 - Pixel quad inefficiencies from dense meshes
- Options:
 - REYES
 - Texture space shading
 - Deferred materials

If we want to move past depth only and support materials too there are many options.

But we would prefer an option where we decouple visibility from materials. By that I mean determining visibility per pixel (which is what depth buffered rasterization does) is disconnected from the material evaluation.

The reasons we want this are to eliminate
Switching of shaders during rasterization
Overdraw of material evaluation or a depth prepass to avoid that overdraw
And pixel quad inefficiencies from dense meshes and we want to draw very dense meshes

There are a few options that fulfill these:
Object space shading either in the form of
REYES or
Texture space

But object space solutions of any form overshade, usually by a substantial factor, 4x or more.

Solving the overshade cost through caching was very attractive but
view dependent,
animating,

and non UV based materials are too prevalent for us to build our rendering foundation on something that doesn't support them well.

The other option is deferred materials.

Visibility Buffer

- Write geometry data to screen
 - Depth : InstanceID : TriangleID
- Material shader per pixel:
 - Load VisBuffer
 - Load instance transform
 - Load 3 vert indexes
 - Load 3 positions
 - Transform positions to screen
 - Derive barycentric coordinates for pixel
 - Load and lerp attributes

This was the best choice for us specifically in the form of a visibility buffer.

The basic idea here is

To write the smallest amount of geometry data to the screen in the form of depth, InstanceID, and TriangleID

Then a material shader per pixel

Loads the vis buffer

Loads the triangle data

Transforms the 3 vertex positions to the screen

Uses those to derive the barycentric coordinates for this pixel

And with those, loads and interpolates the vertex attributes.

=====

Bonus:

Terminology nitpick.: although some use the term “Visibility Buffer” for many forms of deferred material or deferred texturing, personally I only consider the smallest data format possible, the form of Object ID + Triangle ID to be “Visibility Buffer”.

I guess I might make the one concession of adding triangle barycentrics as still being the same thing.

Storing any vertex attributes like UVs or normals in a screen buffer IMO is no longer Visibility Buffer. Those I would call deferred texturing.

Visibility Buffer

- Sounds crazy? Not as slow as it seems
 - Lots of cache hits
 - No overdraw or pixel quad inefficiencies
- Material pass writes GBuffer
 - Integrates with rest of our deferred shading renderer
- Now we can draw all opaque geometry with 1 draw
 - Completely GPU driven
 - Not just depth prepass
 - Rasterize triangles once per view

That sounds **crazy** doesn't it?
But it's not as slow as it seems

There's lots of cache hits
And no overdraw or pixel quad inefficiencies

Typically the visibility buffer approach would combine the material evaluation with shading

But in our case we write to a GBuffer.

This is to integrate with the rest of our deferred shading renderer

We still have good reasons to stay deferred but I won't get into that here.

So now we can draw all opaque geometry
with a single draw call
Completely GPU driven.
Not restricted to just the depth prepass

CPU cost is independent from number of objects in the scene or in view.
Materials are a draw per shader but those are far fewer than objects.

It is also convenient that we only need to rasterize triangles once per view,
no need for multiple rasterization passes to reduce overdraw

Sub-linear scaling



It's much faster than before but still scales linearly with both instance and triangle count.

Linear scaling in **instances** can be ok, at least within the limit of the scale of the level you typically want loaded around you. We can handle a million instances easily. Linear scaling in **triangles** is **not** ok. We can't achieve our goals of "just works no matter how much you throw at it" if we scale linearly.

Ray tracing is $\log N$ which is nice but not enough. We couldn't fit all the data of these scenes in memory even if we could render it fast enough. Virtualized geometry is partly about memory. But ray tracing isn't fast enough for our target even if it fit in memory.

We need better than $\log N$.

=====

Bonus:

The difference in theoretical efficiency between $O(\log n)$ and $O(1)$ may in practice be unimportant. Going from 10k triangles to 1M tris is 43% more tree levels.

The problem is big O ignores memory access. Even if the additional ALU cost was hypothetically free, the cache misses won't be.

Once the triangles become subpixel the rays effectively become incoherent, every additional level a cache miss.

Memory bandwidth is not a good vector to lean on.

Sub-linear scaling



To think about it another way, there are only so many pixels on screen. Why should we draw more triangles than pixels?

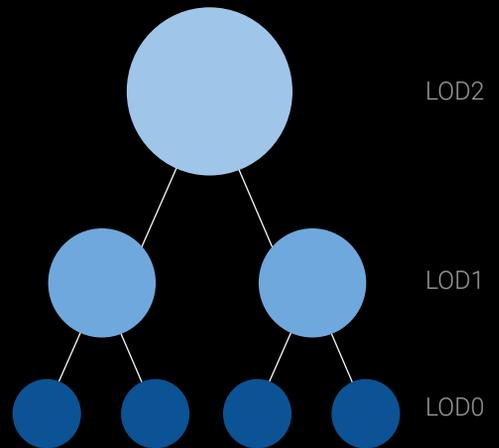
In terms of clusters, we want to draw the same number of clusters every frame regardless of how many objects or how dense they are.

It's impractical to be perfect there but in general the cost of rendering geometry should scale with screen resolution, not scene complexity.

That means constant time in terms of scene complexity and constant time means level of detail.

Cluster hierarchy

- Decide LOD on a cluster basis
- Build a hierarchy of LODs
 - Simplest is tree of clusters
 - Parents are the simplified versions of their children

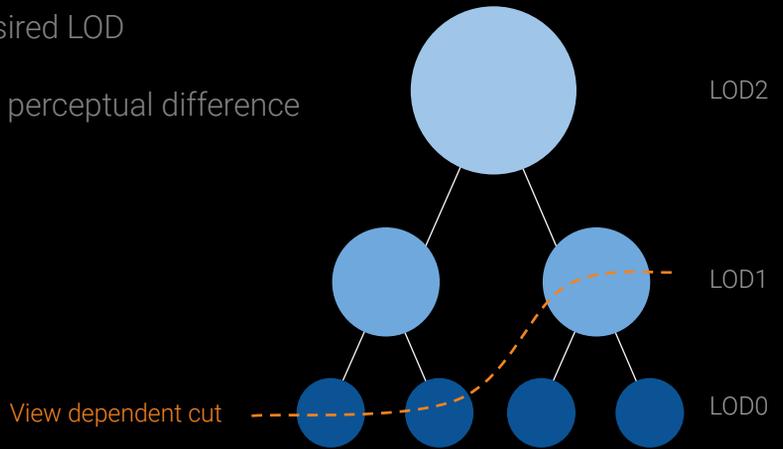


We can do level of detail with clusters too if we build a hierarchy of them.

In the most basic form imagine a tree of clusters where the parents are simplified versions of their children.

LOD run-time

- Find cut of the tree for desired LOD
- View dependent based on perceptual difference



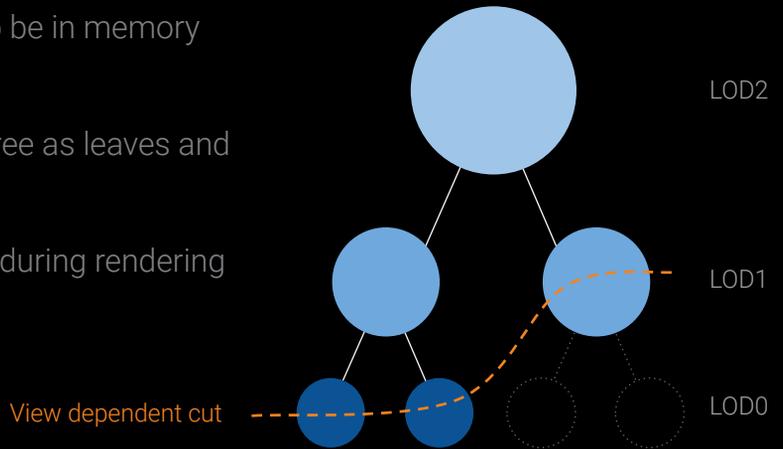
At run time we find a cut of the tree that matches the desired LOD. That means different parts of the same mesh can be at different levels of detail based on what's needed.

This is done in a view dependent way based on the screen space projected error of the cluster.

A parent will draw instead of its children if we determine you couldn't tell the difference from this point of view.

Streaming

- Entire tree doesn't need to be in memory at once
- Can mark any cut of the tree as leaves and toss the rest
- Request data on demand during rendering
 - Like virtual texturing



This gives us all we need to achieve the virtualized part of virtual geometry.

We don't need the entire tree in memory at once to render it.

We can at any point mark a cut of the tree as leaves and not store anything past it in RAM.

Just like virtual texturing we request data based on demand.

If we don't have children resident and want them they are requested from disk.

If we have children resident but haven't drawn them in a while we can evict them.

LOD cracks

- If each cluster decides LOD independent from neighbors, **cracks!**
- Naive solution:
 - Lock shared boundary edges during simplification
 - Independent clusters will always match at boundaries

Ok, that's the high level concept but this simplistic idea doesn't work in practice due to cracks.

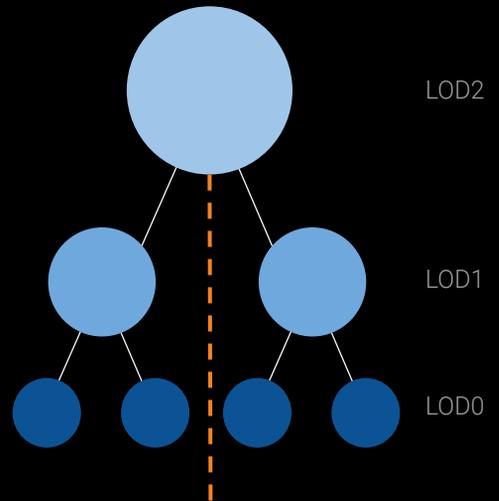
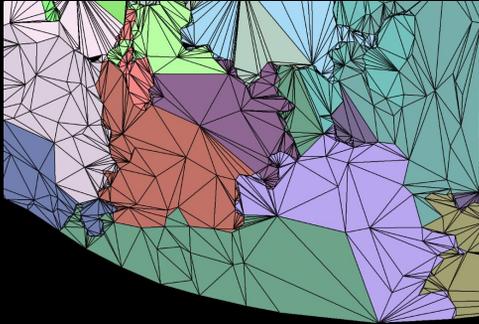
Cracks form when independent clusters make different LOD decisions and the edges at their boundaries no longer match.

The naive solution to this problem is to lock the shared boundary edges between clusters during simplification.

This way independent clusters will always match at their boundaries because we've forced the boundaries not to change.

Locked boundaries

- Collects dense cruff
 - Especially between deep subtrees



This works really poorly though because the same boundaries persist over many levels and dense triangle cruff collects there.

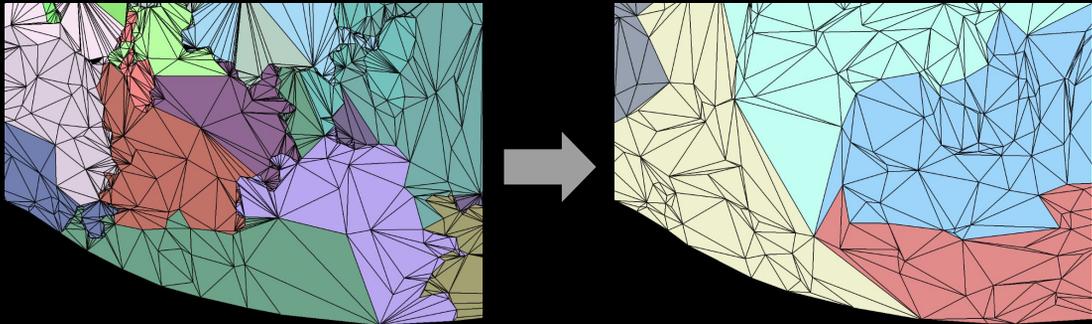
If you can draw a line through many levels of the hierarchy without crossing an edge you just found a boundary that will be locked and won't be simplified for every level it doesn't cross an edge.

With a balanced tree you can draw such a line down the middle from LOD0 all the way to the root.

There will be so much cruff on the boundary between those subtrees that you won't even be able to half the triangles per level and it completely breaks down.

LOD cracks solution

- Can detect these cases during build
- **Group** clusters
 - Force them to make the same LOD decision
 - Now free to unlock shared edges and collapse them



Thankfully we can detect these cases during the build.

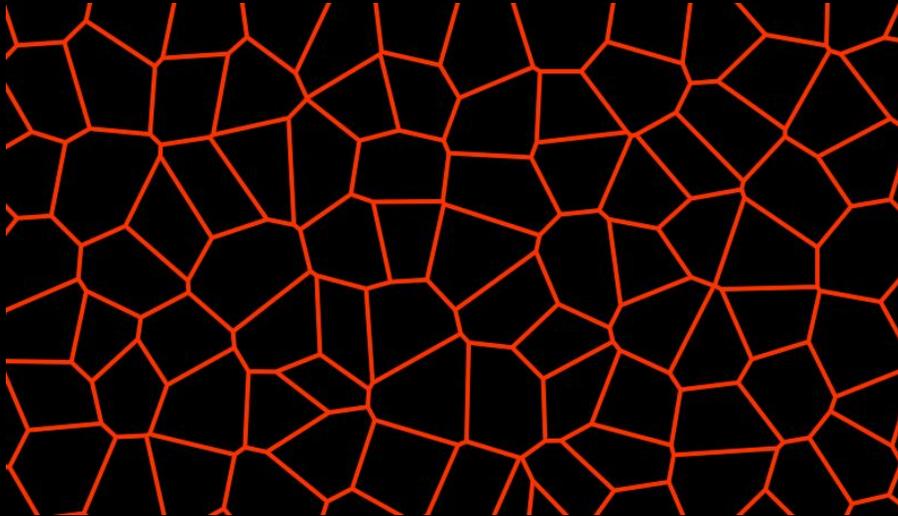
We can group clusters and force them to make the same LOD decision for a level. That means they are no longer independent.

If they always make the same LOD decision then they can't mismatch. Cracks come from differences in level between neighboring clusters so no difference in level means no possibility for cracks.

We are now free to unlock any shared edges between the clusters and collapse them during simplification just like they were interior edges.

Now obviously we can't group all clusters for a level or we get back to good old discrete LODs. So we need to group clusters only where needed.

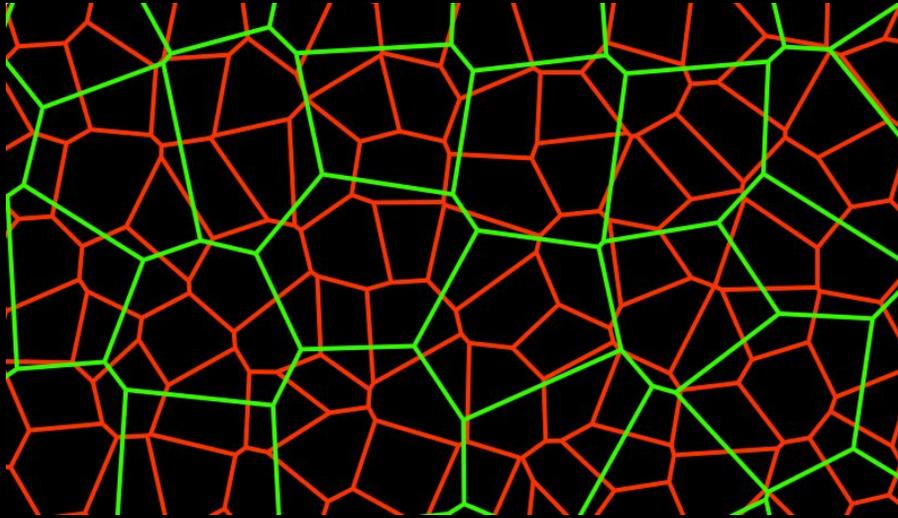
LOD cracks solution



Here's a conceptual visualization of what this looks like over multiple levels. These lines represent the group boundaries and hence the edges that will be locked during simplification of the groups.

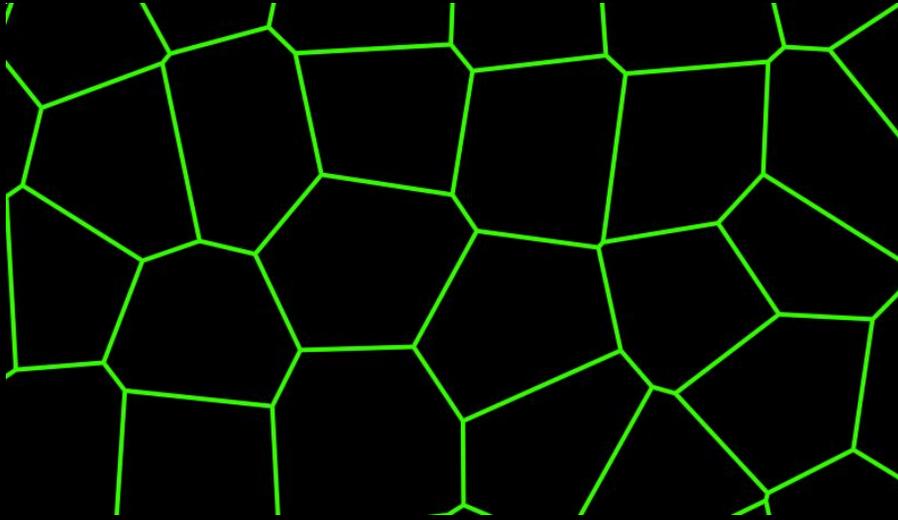
Starting with level 0

LOD cracks solution



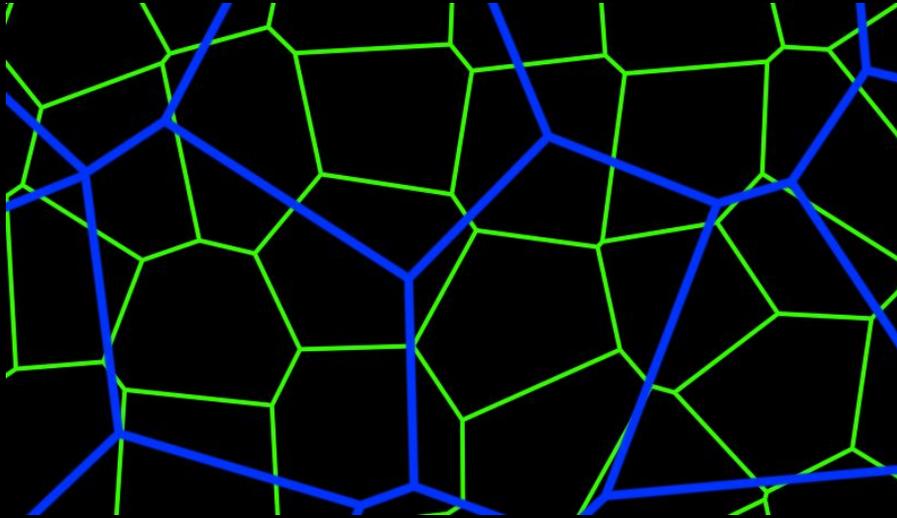
The key idea is to alternate group boundaries from level to level by grouping different clusters.

LOD cracks solution



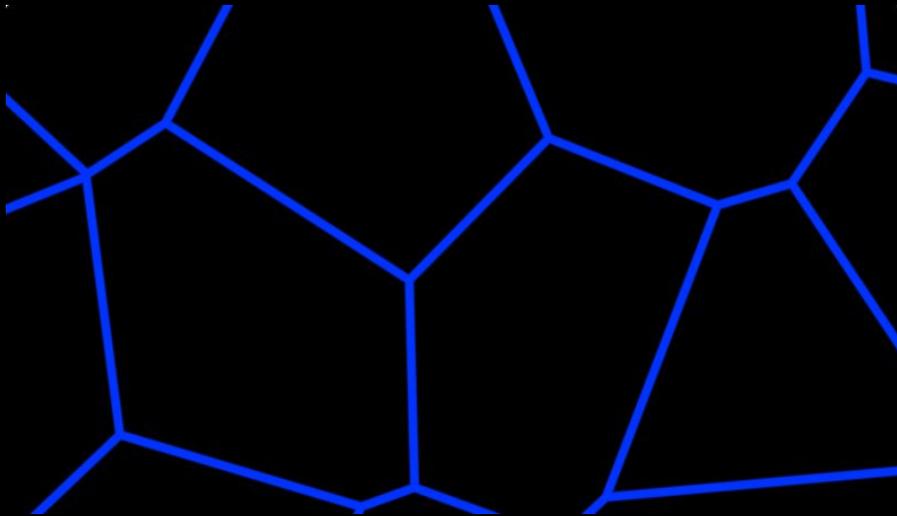
Now at level 1

LOD cracks solution



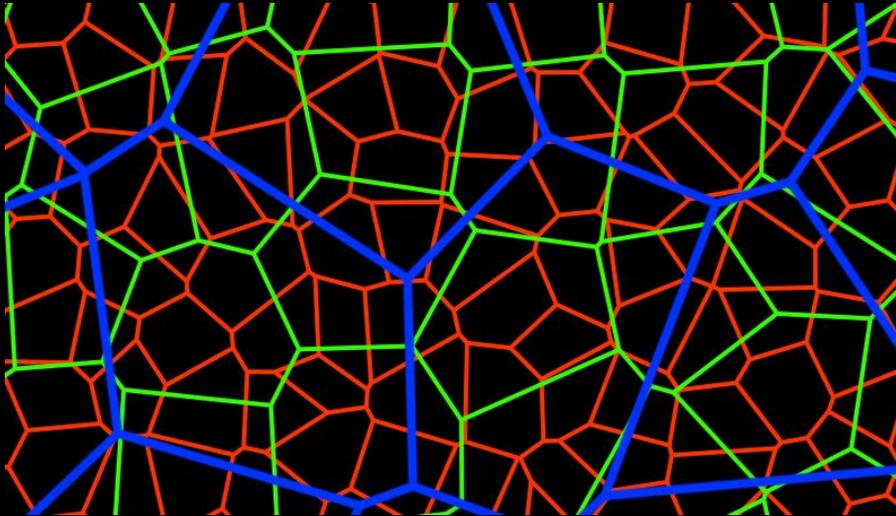
What is a boundary in one level becomes the interior in the next level

LOD cracks solution



Level 2

LOD cracks solution



This means locked edges on the boundaries don't stay locked over many levels because they aren't the same boundaries. Thus they can't collect dense cruft. Locked one level, unlocked the next.

LOD crack options

- Directly index neighboring vertices
 - [Parallel View-Dependent Level-of-Detail Control](#)
 - [Parallel View-Dependent Out-of-Core Progressive Meshes](#)
 - [Dependency Free Parallel Progressive Meshes](#)
 - Need to be able to index boundary verts in any state
 - Impossible to have cracks due to precision
 - Complex and expensive computationally and in terms of memory
 - Granularity of triangles, too fine
- Skirts
 - Soft relationship to neighbor
 - Voxels don't have cracks, why not?
 - Solid volumetric data, not boundary rep
 - Treat mesh like solid volume too
 - Clusters must be closed, at least within band of movement
 - [Chunked LOD](#)
 - Only if boundaries are straight spatial splits

Before we dive into the details there were many other potential solutions to this crack problem I considered.

A dear friend of mine, Jimmy Shin, taught me when exploring a new design space it is useful to try and map the territory.

Visualize the possibility space as continuous with all options present and where those options exist in the landscape in relation to one another.

Even if the manifestation of that option isn't apparent to you yet, the relationship between those that are known to you may point to a hole in the map where something must be, an option that should exist that you haven't thought of yet and how it must work.

Either this can spark ideas of your own or point to research directions that may uncover solutions that you wouldn't have found through direct links of referenced papers.

This is a summary of the map I formed when exploring this problem. The entire virtualized geometry map is far, FAR, bigger.

All the direct VDPM methods are working on individual triangles or edges, communicate streaming at that granularity as well. The data sizes for those granular edits are fairly large as well.

While fine granularity means the number of triangles to hit an error threshold is optimal that isn't the ultimate goal.

We can rasterize triangles fairly fast so spending too much time or data to find the

perfect view dependent mesh to render is counterproductive.

Skirts are a classic technique from terrain rendering. Really they come from the realization that volumetric data structures can't have holes and we can use a b-rep in place of the volumetric data. This allows T-junctions without the typical cracks.

They are not very straightforward though in the case of arbitrary meshes.

First, we have no guarantee that our meshes are manifold and watertight such that they can be operating on volumetrically.

Second, even if they were, T-junctions without cracks only helps if different triangulations end at a plane. In the 2.5D terrain case the corners of a tile are locked but vertices along the edge between those corners are free to move so long as they don't leave the vertical plane that goes from corner to corner and they don't go past the corners.

In 3D the constraint will still be planar but will be additionally constrained to be within the face of some volumetric solid that is cutting up the mesh, probably a box or tetrahedron.

Reducing the caps to just skirts of maximum separation is a complex 2D bounding problem within the faces.

This is all complex, requires lots of clipping and generating lots of extra hidden geometry which makes it less efficient than locking boundaries.

[51], [58], [59], [60]

LOD crack options

- Implicit dependency
 - Space implies dependency between nodes
 - [Progressive Buffers](#)
 - What happens when required data not resident?
 - [Adaptive TetraPuzzles](#)
 - Forces nodes of level X to be certain size or shape

Progressive buffer and tetrapuzzles are what I consider implicit dependency approaches. The dependencies between clusters come implicitly and don't need to be stored in any way.

This means something about the node must be known implicitly about its spatial location and this causes problems.

In the case of progressive buffers a single node can match with both one level and another at the same time because geomorphing is calculated per vertex. Ignoring the fact that geomorphing is both data and computationally costly, this means a node (cluster) can only span 1 LOD without cracking.

To guarantee this, LOD always is applied in strict distance bands and nodes must be smaller than the width of those bands such that a node can't fall in more than two bands at once.

This means LOD can't be based on visual error of that node would introduce, meaning it can't be adaptive.

It also means that if a LOD isn't resident for a certain portion of the surface it may be unable to obey the band requirement and thus will crack.

Tetrapuzzles don't have those issues but have others. Because the clustering is purely spatial (which tetrapuzzle a triangle falls in) we have no control over number of triangles per cluster.

Is it entirely possible that only a single triangle falls in a tetrahedron. The amount of wasted lanes in the rasterizer due to this lack of control makes it very inefficient to render.

Triangles in typical meshes are not uniformly sized or distributed. This also means that just because the size of tetrahedrons from one level to the next jump in expected volume increments doesn't mean a corresponding multiple of triangles will fall in them.

Again it is entirely possible. That a cluster from one level gets no new triangles added to it in the next and yet it is expecting to double.

[53], [56]

LOD crack options

- Explicit dependency
 - Dependency between nodes is determined during building and stored
 - [Quick-VDR](#)
 - [Batched Multi-Triangulation](#)

This is the direction I took and the prior work we primarily build upon. Both are two sides of the same coin in a way and what we do is hybrid of both with additional insights I'll get into later.

Batched Multi-triangulation is an excellent theoretical framework but I found this paper very hard to digest and understand as it is excessively abstract and theoretical. It wasn't until years later after partly implementing QuickVDR that I tried rereading it once again and it sunk in how insightful it really was as a super set of multiple schemes.

I highly recommend reading this dissertation by Federico Ponchio <https://d-nb.info/997062789/34> if the original paper doesn't make sense at first. The slides for the paper are also helpful http://vcg.isti.cnr.it/Publications/2005/CGGMPS05/Slide_BatchedMT_Vis05.pdf

The break down of building steps are the same basic ones I'll cover next with some tweaks.

Prior work groups the triangles themselves which results in variable number of triangles per group

But we want groups of multiples of 128 triangles such that they can be divided into clusters of exactly 128.

Grouping clusters instead of triangles allows this.

Build operations

- Cluster original triangles
- While NumClusters > 1
 - **Group** clusters to clean their shared boundary
 - **Merge** triangles from group into shared list
 - **Simplify** to 50% the # of triangles
 - **Split** simplified triangle list into clusters (128 tris)

Ok so here are what the build operations are.

First we need to build the leaf clusters

In our case a cluster is 128 triangles

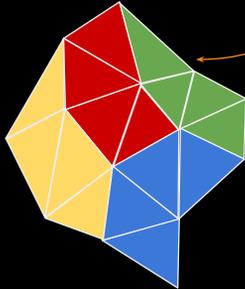
Then for each LOD level

- We group clusters to clean their shared boundary
- Merge the triangles from the group into a shared list
- Simplify to half the number of triangles
- Then split the simplified list of triangles back into clusters

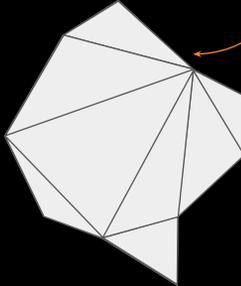
Repeat this process until there is only 1 cluster left at the root.

Build operations

Pick **group** of N clusters

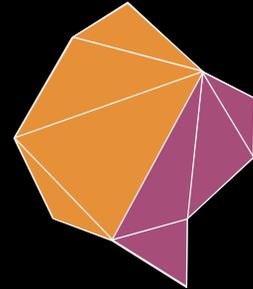


Merge and **Simplify**

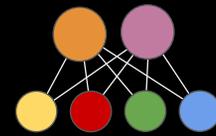
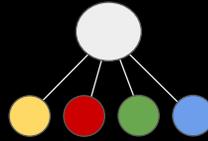


Boundary unchanged

Split into N/2 clusters



DAG



Here is an illustration of the process.

In this example, we have **Grouped** these 4 adjacent clusters.
The corresponding DAG is at the bottom.

(=>)

In the next step the clusters are **Merged** and **Simplified** to half the number of triangles.

In the DAG this corresponds to the 4 nodes getting this new simplified mesh as a their parent.

(=>)

Note that the boundary is locked so we are not causing cracks with any neighboring clusters in the original mesh.

(=>)

Finally, the simplified triangle list is **Split** back into 2 new clusters.

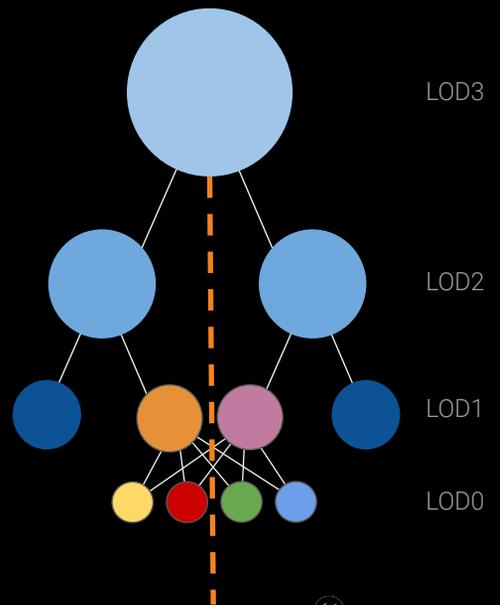
In the DAG this corresponds to splitting the parent into two, while all children are still connected to both parents.

We now have reduced 4 4-triangle clusters to 2 4-triangle clusters.

We put them back in the pool to choose from for grouping and the process continues until everything has been reduced to a single cluster at the root of the DAG.

DAG

- **Merge** and **split** makes this a DAG instead of a tree



The **merging** and **splitting** steps makes this a DAG instead of a tree.

(=>)

This is a good thing in that you can't draw a line from LOD0 all the way to the root without crossing an edge.

Meaning there can't be locked edges that stay locked and collect cruft.

DAAG: Which clusters to group?

- **Group** those with the most shared boundary edges
 - Less boundary edges == less locked edges
- This problem is called graph partitioning
- Partition a graph optimizing for minimum edge cut cost
 - Graph nodes = clusters
 - Graph edges = connect clusters with directly connected triangles
 - Graph edge weights = number of shared triangle edges
 - Additional graph edges for to spatially close clusters
 - Adds spatial info for island cases
- Minimum graph edge cut == least locked edges
- Use METIS library to solve

Lets walk through each step of the build operations.

First, how do we decide which clusters to group?

We group those with the most shared boundary edges because the less boundary edges we have, the less locked edges we have.

The less locked edges the better because locked edges restrict the simplifier from reducing triangles.

This problem is called graph partitioning and is well explored in computer science although probably not that familiar to most people in graphics.

Graph partitioning algorithms partition a graph into a specified number of partitions such that total weight of all the edges that cross from one partition to another, called the edge cut cost, is minimized.

In our case graph nodes are clusters.

Graph edges connect clusters that have directly connected adjacent triangles.

The graph edge weights are the number of shared triangle edges between those clusters

We add additional graph edges for spatially close but not connected clusters to make sure there aren't islands in the graph that would get grouped arbitrarily.

The minimum edge cut of this graph corresponds to the least number of locked edges across the mesh for this level which is basically the ideal thing to optimize for.

Graph partitioning is a very complex task and thankfully there are existing libraries that are used to do it.

We use the popular METIS library.

Initial clustering

- How to build the initial leaf triangle clusters?
- Optimizing for many variables
 - Cluster bounds extent
 - Culling efficiency
 - NumTris per cluster \leq max
 - Fully fill waves in rasterizer
 - NumVerts per cluster
 - Prim shader limit
 - Number of cluster boundary edges
 - Boundaries are locked so limits simplification



But to start building this DAG we need the leaf clusters. How do we build those?

Unfortunately this is a multi-dimensional optimization problem.

- We'd like to minimize cluster bounds extent for culling efficiency
- The number of triangles per cluster needs to be as close to but no greater than 128 to work with the rasterizer
- The number of vertices can't be above a limit to work with primitive shaders
- And we want to minimize the number of boundary edges between clusters, again to give the simplifier the most unlocked edges possible to allow it to do its work.

Graph partitioning

- Pick two and hope the rest works out
 - Number of cluster boundary edges
 - NumTris per cluster \leq max
- Exactly the same problem as cluster **grouping**
 - Graph is the dual of the mesh
- Requires strict upper bound on partition size
 - Graph partitioning algorithms don't guarantee this
 - Managed to coerce it with small slack and fallbacks

We can't really optimize for all dimensions simultaneously so pick 2 and hope the rest works out because they are correlated.

Specifically, we optimize for the number of boundary edges and number of triangles per cluster

Minimizing the number of shared edges between clusters is exactly the same problem we had with cluster grouping

Once again we are graph partitioning

This time the graph is the dual of the mesh

The one difference is that this time we have a strict upper bound on the partition size.

Unfortunately, graph partitioning algorithms don't guarantee this.

I've managed to coerce it though with a small amount of slack and fallbacks when it very rarely fails

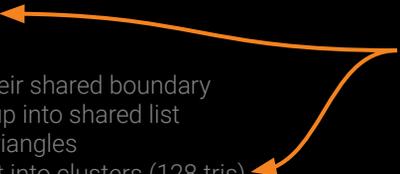
=====

Bonus:

Better results should be possible with custom graph partitioning code which takes bounds into account for cost metric.

Best of breed graph partitioning algorithms are fairly complex and writing our own would be a large undertaking.

Build operations

- **Cluster** original triangles
 - While NumClusters > 1
 - **Group** clusters to clean their shared boundary
 - **Merge** triangles from group into shared list
 - **Simplify** to 50% the # of triangles
 - **Split** simplified triangle list into clusters (128 tris)
- Same thing
- 

It turns out that the **Split** step is actually the same thing as the initial clustering process

Take a list of triangles and build clusters of 128

Comparison to prior work

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

An older version of Nanite was more a mix of those and Tetrapuzzles in that the grouping followed Voronoi partitions but they were spatial partitions like tetrapuzzles. In space simple Worley noise is an irregular Voronoi partition so grouping for a level is scale the coordinates to the resolution for that level and anything that lands in the same cell is grouped.

The problem with any spatial based grouping like Tetrapuzzles uses, and in particular ones with rigid shapes, sizes, and correspondence with levels is the unevenness in the number of triangles assigned to each group. Only one triangle (or one cluster in our case) could fall in a tetrahedron and there's little you can do about it.

Treating the relationships between clusters as an unstructured graph frees you to get at more of the properties that really matter and optimize for them directly.

Batched Multi-triangulation saw the problem with the most general framework of group, merge, simplify, split such that there's no need for the clusters in the group to have anything to do with the clustering of triangles after split. That is a really nice property and actually simplifies implementation. It is unclear to me whether QuickVDR did the same but they didn't call it out or explain this principal as clearly.

QuickVDR saw that the goals of grouping could be optimized for directly. In terms of building the binary tree they realized graph partitioning could optimize for minimal shared edges. But funny enough for deciding dependent nodes, which was their solution to locking boundaries, they did a greedy optimization with a priority queue

disconnected and after building the binary tree through graph partitioning.

Combining the two forms becomes what I believe is the most general and optimal framework. Grouping isn't stochastic from irregular Voronoi partitions, it is directly optimized for by expressing the cluster relationships as a graph with weighted edges. Grouping and hierarchy building is a unified operation expressed as a graph that we can divide using graph partitioning. Nodes that are dependent on one another are no different than siblings which means the structure is really a DAG. The number of triangles being worked with is always in multiples of some fixed granularity meaning we can exactly fill our 128 triangle clusters. We do not form groups with individual triangles. The merge and split formulation is symmetrical and understands clearly that no correspondence or constraints in terms of triangle to cluster assignment needs to be made between parents and children. All siblings are connected to all their parents. The structure ultimately being built is clearly a DAG which enables reasoning about it correctly.

Mesh **simplify**

- Edge collapsing
- Picks smallest error edge first
- Error calculated using Quadric Error Metric (QEM)
- Optimizes position of new vertex for minimal error

- Highly refined
- Returns **estimate of error** introduced
 - Later projected on screen to number of pixels error
 - The **hardest part!**

The last step we'll discuss is **simplification**.
For this we use typical edge collapsing decimation.

Error is calculated using the Quadric Error Metric
We optimize for minimal error in new positions and attributes.

Although not very novel this code is very highly refined at this point for quality and speed and bests any other commercially available option out there to my knowledge.

It returns an estimate of the error introduced by simplifying and this has been the most challenging aspect to get right.

This estimated error is later projected on screen to a number of pixels of error to determine which LOD to select so is foundational to both quality and efficiency.

Perceptual heuristic for pixel error in a highly efficient to evaluate form and one that can be directly optimized for has consumed an **enormous** amount of time.

If I were to guess, probably a man year worth of effort has gone into this problem in one form or another.

In some ways it is an impossible task because at this stage we don't know what material will be applied to this mesh.

For example we don't know how shiny it is to know the exact impact of normal error
Or how much UV error is apparent based on the texture

Or if there are vertex colors, what they are even used for.

What we have definitely isn't perfect and can fail although it's rare to be noticeable. It certainly could be more aggressive by factoring in more aspects about human perception.

Error metric

- Basic quadrics are integral of distance² error over area
- Quadrics with attributes mix all errors in one with weights
 - Complete heuristic hack
- Can do better?
 - Hausdorff mesh distance?
 - Render results and use image based perceptual?
 - No concept of rate distortion optimization
- Import and build time matters too
 - All of the Nanite builder code is highly optimized too
- Want collapses to optimize for same metric as returned pixel error

Error metric

- Scale independence
- Need to know size on screen to know weights
 - Chicken and egg problem
- Assume most clusters draw at constant screen size
- Surface area normalized
- Edge length limits
- Lots of tuning
- Great care with floating point precision
 - Many places in quadrics with inherent catastrophic cancellation

=====

Bonus:

I'll explain a realization and trick I found in mixing position and attribute error that I haven't seen published before.

There has been a question that has been a thorn in my side for the simplify for a long time. The balance of weight for attributes vs position has always been weird. What I want is to know how far does the camera needs to be such that the error generated by simplification is imperceptible. The math to project world space size to size in pixels is easy. That means I need to know in world space how much error was generated such that I can transform it into number of pixels. Perception is so much more complicated than this but this is the framework I'm working under. Position deviation error is exactly in the right units and framework to do this with. Attribute error such as difference in normals is not. I've looked at countless papers for what to do about that and I've found no solutions beyond what I have been doing, which is to mix position and attribute errors together as if they were the same units but with weights for how important each is relatively. There is no theoretical foundation for this. The only defense for it is experimentally. It's a dumb heuristic to mix them as if they are the same thing. They aren't but I still don't have anything better.

What's even worse is the balance dictated by the weights isn't scale invariant. Attribute deviation, particularly normals, is scale independent. How perceptible is a 1 degree change in normal? How about if it was a mile away? Doesn't matter. All that matters is how many pixels that error covers. Position deviation on the other hand is

scale dependent. 1 unit position offset a mile away can't be perceived no matter if it was the entire empire state building that moved.

Time for some thought experiments. If I have a mesh that is a sculpture of a standing man and I have another mesh that is identical except it is scaled by 100x they shouldn't have a different balance of normals vs position. Both should simplify the same. So rescale the bounds to be the same? This is what the literature suggests and what I was doing before. All meshes are scaled to have similar extents. What if I have a mesh with 2 copies side by side? Now what? Normalizing the bounds doesn't work. What if you have the same 2 copies but they are really far apart? Same geo but different bounds. Normalizing average surface area of a triangle instead of bounds accounts for this. What if the 2 copy mesh has one of the copies tiny enough to stand on the shoulder of the other? Why should the smaller copy simplify any differently than the larger? Any sort of global rescaling doesn't work anymore, even with surface area. What granularity does? Feels like it should be taken to the limit as granularity goes to zero down to a triangle. I'm not exactly sure how that would work though. Needs more thought. Probably this operation should happen on the edge collapse triangle neighborhood. In practice it's probably fine to bring the granularity low enough.

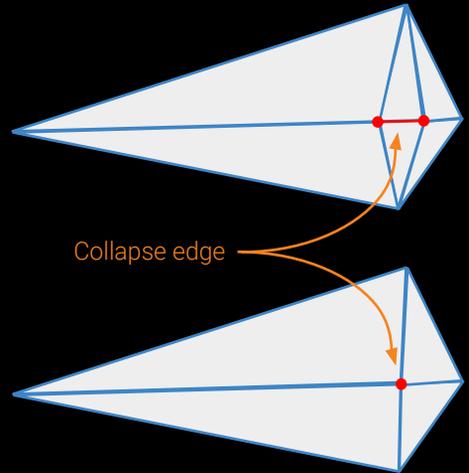
Per group of clusters I pass to the simplifier is small enough that the same sort of behavior roughly happens in both the big and small copy case. Something that Nanite's structure enables but doesn't exist in the general simplifier. Both will be clustered similarly, clusters should group similarly. From a LODing perspective those most often will project on screen to the same average triangle size. So I calculate the average surface area per triangle and rescale the mesh such that that is roughly a constant I chose as a good mix of position to attribute balance. With that it is now scale invariant even down to the subpart.

That small guy on the shoulder of the big guy is actually a really important case to get right. It means when a mesh is very far and thus very small, it should simplify similarly to a small part of a big mesh that is up close. A cluster that covers N pixels should simplify and calculate its error the same as any other that covers N pixels.

Getting this right isn't just a theoretical nicity. Making this fix changed the number of triangles drawn in our demos from being somewhat variable to basically constant. For example the room of statues used to be 2-3x as expensive as the cave scene. After this change both scenes rasterized the same number of triangles.

Prefiltering (future work)

- Can't prefilter BxDF
 - Material assignment is unknown
- Normal distribution could be filtered
 - Similar to normal map prefiltering to roughness
 - SGGX (volumetric)
 - Needs to apply to diffuse as well
 - Fed to BxDF
- Non-uniform triangle sizes cause issues
 - Vertex footprint isn't symmetrical
 - Lerps across all adjacent triangles
 - Filter kernel won't match pixels
 - Minification and magnification simultaneously
 - Unsolved



Prefiltering (future work)

- Visibility prefiltering means expressing partial coverage
 - Triangle mesh stops making sense
- Best solution is a hybrid
 - Superpixel features as mesh
 - Subpixel features expressed volumetrically
 - [Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets](#)
- How to render partial coverage volumetric data?
 - Stochastic?
 - Ray march?
 - Point scatter?
 - OIT?

Subpixel partial coverage and prefiltering is especially important for aggregate geometry like leaves and grass.
Solving that case is still an open question.

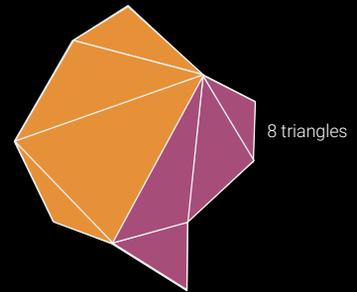
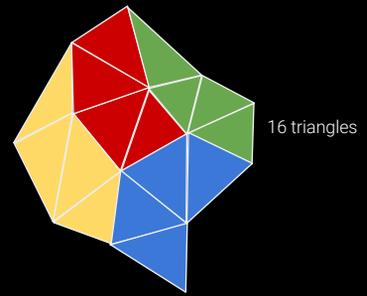
Run-time view dependent LOD

Moving on to the run-time portion

We've built this cluster hierarchy. Each frame we need to select which clusters we are going to draw based on the view.

LOD selection

- Two submeshes with same boundary, but different LOD
- Choose between them based on screen-space error
 - Error calculated by simplifier projected to screen
 - Corrected for distance and angle distortion at worst-case point in sphere bounds
- All clusters in group must make same LOD decision
 - How? Communicate? No!
 - Same input => same output



The interesting thing is that for each iteration of the build operations we produced two sets of clusters with a different number of triangles, but with the same boundary. So they are interchangeable in the original mesh, without causing any cracks. This is the essence of the level-of-detail system.

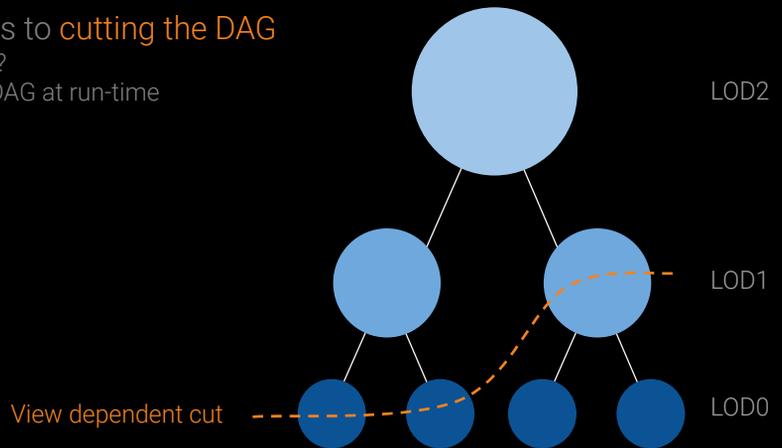
At runtime we choose between them based on how much screen space error we estimate they have. Error calculated by the simplifier is projected to the screen accounting for distance and projection angle distortion. This is calculated at the point that maximizes the projected error within a bounding sphere around the cluster.

As explained before, grouped clusters must make the same LOD decision. How do we do that in parallel? One to many expansion? Communicate between them?

No it's very simple. If they use the same data to decide they will decide the same thing. Same input, same output. So all clusters in the group store the same unioned error value and sphere bounds used to project that error to the screen.

LOD selection in parallel

- LOD selection corresponds to **cutting the DAG**
 - How to compute in parallel?
 - Don't want to traverse the DAG at run-time
- What defines the cut?



But all that gives us is whether this cluster has small enough error that we could draw it.

We don't want to draw all low error clusters. Many of them represent the same area but at different detail.

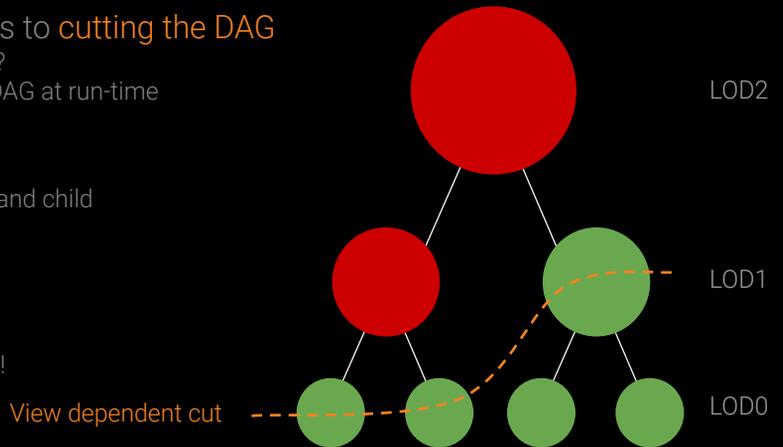
LOD selection involves finding a view dependent **cut** of the hierarchy and we now know that is actually a DAG.

We want to determine this on the GPU, so how can we compute the cut in parallel efficiently?

It is worth considering what defines the cut.

LOD selection in parallel

- LOD selection corresponds to **cutting the DAG**
 - How to compute in parallel?
 - Don't want to traverse the DAG at run-time
- What defines the cut?
 - Difference between parent and child
- Draw a cluster when:
 - Parent error is too high &&
 - Our error is small enough
 - Can be evaluated in parallel!



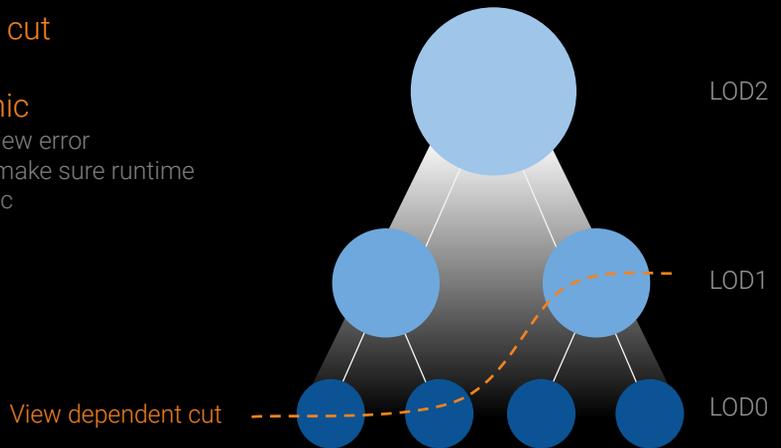
The cut happens at the point where a parent's error is too high, but it's child's error is small enough to be valid to draw.

Parent says no but the child says yes.

This is entirely local, does not depend on the entire path to this node, and thus can be evaluated in parallel.

LOD selection in parallel

- Only if there is **one unique cut**
- Force error to be **monotonic**
 - Parent view error \geq child view error
 - Careful implementation to make sure runtime correction is also monotonic



But that is only true assuming there is 1 unique cut. If there wasn't we'd need to walk the path and find the first one.

There will be a unique cut if the selection function for all paths from the root to the leaves switches from no to yes once and never switches back.

Our selection function is a thresholding of the view dependent error function.

Guaranteeing a single transition along the path is the same as forcing the error function of every path to be monotonic.

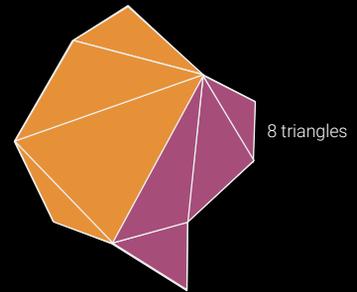
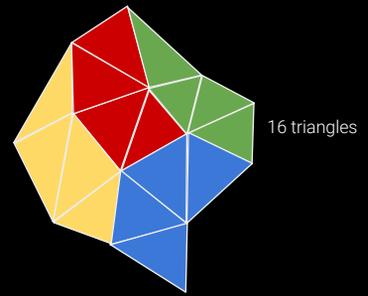
That sounds complicated but really means we need to make the error calculation for parents to always be at least as large as their children.

This is forced during the offline DAG building by modifying the parent's stored error and bounds used for projecting it.

Both need to be as big as their children.

Seamless LOD

- Binary choice either parent or child
- Won't this visibly pop?
- Need smooth transitions?
 - Geomorphing
 - Cross fading
- If < 1 pixel error, they are imperceptibly different
- Temporal AA sees any difference as aliasing



So in each frame we are selecting either the parent or the child.
Won't this visibly pop when it switches?

Do we need to smooth the transitions with geomorphing or cross fading?

That would be expensive at render time
or require significant additional data
or both.

But if we only draw clusters that are less than 1 pixel of error they are imperceptibly different and temporal antialiasing smoothes out any change.
TAA is built to blend subpixel differences over time.

It does our work for us so long as the error is subpixel.
This is the reason why getting an accurate error estimate is so important.

Surface angle based LOD

- Cluster error from simplification is object space scalar
 - Unknown direction
 - Position error could be directional
 - Attribute error mixed in makes this hard
- Projection to screen does not account for surface angle
 - Similar to if mipmaps only were purely a function of distance
 - Common for tessellation factor calculation as well
- Means glancing angle surfaces over tessellate
- Solving requires anisotropic LOD
 - Not possible through cluster selection
 - Cluster selection must be isotropic like mip selection
- Glancing angle costs happen with other schemes too
 - Overdraw in point based
 - Surface skimming in SDFs and SVOs

Hierarchical LOD selection

- Visible clusters could be
 - Close: All from single instance
 - Far: All root clusters from different instances
- Needs to be hierarchical
 - But DAG traversal is complicated!
- Remember: LOD decision entirely local
 - We can use any data structure we want to accelerate this!

As explained, we can make cluster selection perfectly parallel, but it turns out to be extremely wasteful when scaled up.

For large scenes the vast majority of clusters are too detailed to be selected. We shouldn't even be considering most of them.

For quick rejection, we need a hierarchy. The most natural thing would be to base it on the DAG structure.

But traversing a DAG is more complicated than we would like, especially if it has to happen in parallel.

Luckily, as we have just shown. The LOD decision can be evaluated locally using just the cluster error and the parent error, so we don't need to use the DAG at all.

We can build any data structure we want to accelerate these tests.

Hierarchical culling

- When can we LOD cull a cluster?
 - Render: `ParentError > threshold && ClusterError <= threshold`
 - Cull: `ParentError <= threshold || ClusterError > threshold`

Parent is already precise enough. No need to check child

- `ParentError <= threshold`
 - Tree based on `ParentError`, not `ClusterError`!
- BVH8
 - Max of children's `ParentError`
 - Internal node: 8 children nodes
 - Leaf node: List of clusters in group

The clusters we can cull are exactly the ones that fail the LOD selection test from before.

Any cluster whose `ParentError` is already small enough can be culled.

Interestingly, this means that an acceleration structure for LOD culling should be based on `ParentError`, not the `ClusterError` itself.

With that we build a BVH over the clusters.

As with any BVH, the parents conservatively bound their children which in this case also includes `ParentError`.

Because parents are shared in a group, the leaves of the tree are group-sized lists of clusters.

=====

Bonus:

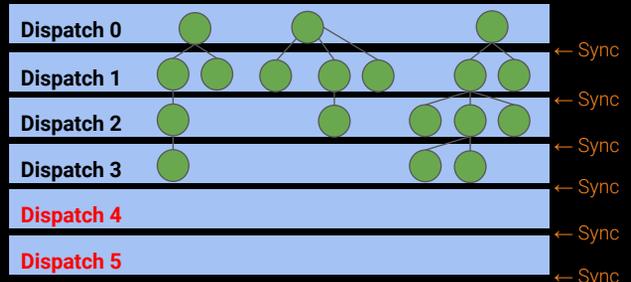
A cluster can be culled both if its error is too large or if its parent error is already small enough.

Typically, for a given view, there are orders of magnitude more clusters that are too detailed than there are clusters that are not detailed enough.

We should focus on accelerating the culling of too detailed clusters. That would be any cluster with a `ParentError` that is already small enough below the threshold.

Hierarchical culling - naive approach

- Dependent DispatchIndirects
 - One per level
- Global synchronization
 - Wait for idle between every level
- Worst case # of levels
 - Empty dispatches at the end
- Can be mitigated by higher fanout
 - Wasteful for small/distant objects



Traversing this tree is a classic parallel expansion work scheduling problem. Implemented naively looks like this with many passes, each processing a single level of the tree, appending any passing children nodes to a buffer to be processed by the next pass.

Each pass depends on the previous, so the GPU is completely drained at every level of the tree.

Because the CPU doesn't know how deep the recursion will go, enough dispatches have to be issued to cover the worst case.

This means we can very easily end up with empty dispatches that don't do any processing at all!

This can be mitigated somewhat by choosing a higher branch factor, but this also results in inefficiencies.

=====

Bonus:

Small or distant objects might only need to render a few clusters, but will always have to evaluate all of their children.

Persistent threads

- Ideally
 - Start on child as soon as parent finished
 - Spawn child threads directly from compute
- Persistent threads model instead
 - Can't spawn new threads. Reuse them instead!
 - Manage our own job queue
 - Single dispatch with enough worker threads to fill GPU
 - Use simple multi-producer multi-consumer (MPMC) job-queue to communicate between threads

Really we'd like to start processing the child as soon as the parent passes, not wait for every other node of this level to finish.

Ideally, we would be able to just spawn new threads for the children directly from compute. But we don't currently have any way to do that.

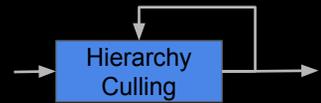
But, instead of spawning new threads, we can reuse the threads we already have and distribute work to them using our own job queue.

This is known as persistent threads and is basically like writing your own mini job system.

Spawn just enough worker threads to fill the GPU and then keep reusing those threads, pulling work from the queue until there's none left.

Persistent Threads - Hierarchy Culling

- While work queue isn't empty
 - Dequeue a node
 - Test
 - Enqueue any passing children
- Single dispatch
 - No recursion-depth or fanout limits
 - No need to repeatedly drain the GPU
 - 10-60% saving (typically ~25%), depending on scene complexity
- Relies on scheduling behavior
 - Requires that once a group has started executing, it will not be starved indefinitely
 - Scheduling behavior not defined by D3D or HLSL
 - Works on consoles and all relevant GPUs we have tested
 - An optimization. Not a requirement for Nanite



For hierarchical culling this means:

Repeatedly, threads pop a node from the queue, process it and push back the children that pass the tests, until the queue is empty.

There is now just a single dispatch

There are no more limits to the number of levels and the GPU doesn't have to be drained repeatedly.

The persistent threads approach is 25% faster on average than the naive one.

Unfortunately, blocking algorithms like this rely on scheduling behavior that is not defined by D3D or HLSL.

The crucial property that needs to be guaranteed is that once a thread group has started executing, and thus could have taken a lock, it should continue to be scheduled and not be starved indefinitely.

Although undefined, this approach works on consoles and all relevant GPUs we have tested so far.

=====

Bonus:

Hopefully, PC programming models will catch up, so we can use optimizations like this with confidence on PC in the future.

Persistent Threads - Cluster Culling

- Leaves are clusters with shared parents
 - Go through similar culling checks as nodes
 - Outputs visible clusters
- Cluster culling in same persistent shader
 - Might not be enough active BVH nodes at a time to fill the GPU
 - Execution time can end up being determined by depth of deepest traversal
 - Start cluster culling work early and use it to fill holes
- Two queues
 - Process from cluster queue while waiting for nodes to appear in node queue
 - Coalesced into batches of 64



Nodes in the tree represent ParentError, so the leaves are lists of clusters that share that parent.

These clusters need to go through similar culling checks as the nodes.

Because the BVH traversal can go deep and the number of active nodes at a given time can be small relative to the width of the GPU,

the BVH culling phase will not always be able to fill the GPU.

When that is the case, then the latency of a few deep traversals can end up dominating the culling execution time.

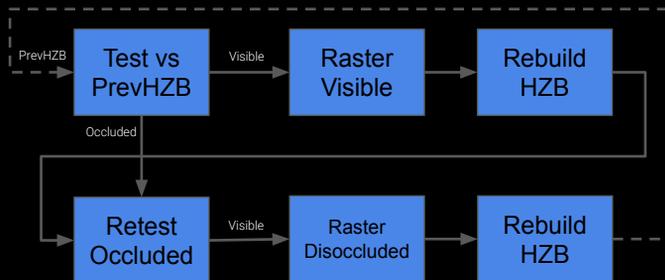
To mitigate this issue, we have integrated the the cluster culling into the persistent hierarchy culling shader, so cluster culling can be started early and fill out holes in the BVH culling.

In practice, this is implemented with an additional queue for clusters. While a worker is waiting for new nodes to appear in the node queue, it can process already found clusters from the cluster queue.

To avoid divergence this is done in batches..

Two-Pass Occlusion Culling

- Explicitly tracking previously visible state gets complicated
 - LOD selection likely different
 - Visible clusters from previous frame might not even be in memory anymore!
- Test if currently selected clusters **would** have been visible last frame
 - Test previous HZB using previous transforms



Now obviously while we are doing all this culling for LOD we should also cull based on visibility.

There is an issue though with the 2 pass occlusion method I previously explained.

Tracking the previously visible set gets complicated
LOD selection from frame to frame is likely different
and visible clusters from last frame might not even be in memory anymore due to streaming.

So instead we test if the currently selected clusters **would** have been visible in the last frame.

To do that we test their bounds against the previous frame's HZB using the previous transforms.

So the two pass solution now looks like this:

- Test the previous HZB with the previous transform
- Draw what is visible, save the occluded for later
- Then build the initial HZB for this frame from the depth buffer
- Using this HZB, test what we thought was occluded again
- Draw what is now visible, but was previously occluded
- Finally, build the complete HZB from the now complete depth buffer to use in the next frame.

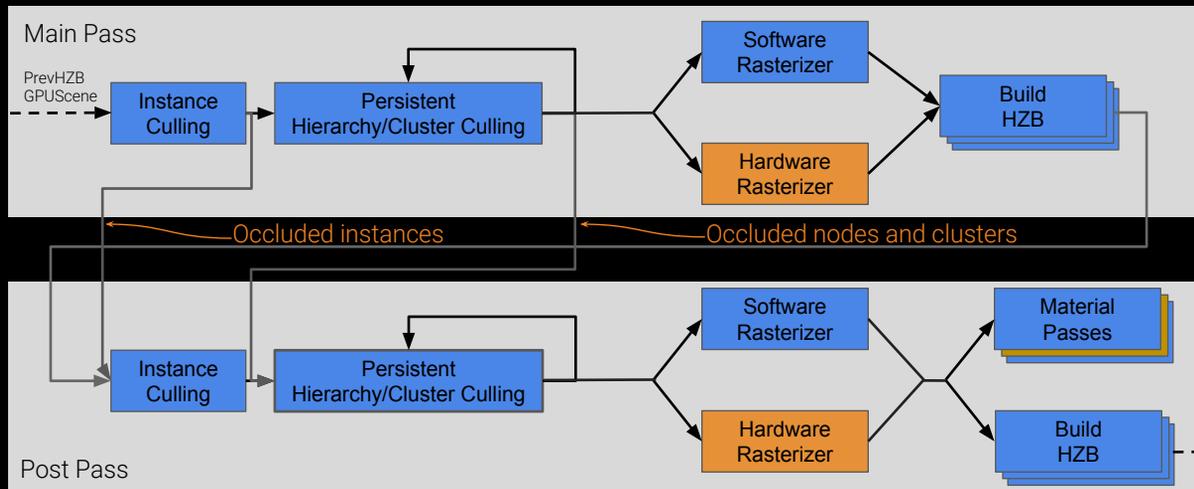
Because of all this we end up running almost the entire Nanite pipeline twice,

where the second pass is essentially just cleaning up the disoccluded regions.

This is not quite as bad as it sounds though, as the second pass is usually a tiny fraction of the main pass.

Note that any culling not related to occlusion only needs to evaluate once, So frustum and LOD culling is only done in the first pass

Culling Dataflow



Putting it all together this is what the culling flow looks like:

The first pass evaluates occlusion using the previous frames transforms testing against the previous frame's HZB. It starts with instances from GPUScene evaluating visibility on a per instance basis.

(=>)

Visible instances then go to the Persistent threads hierarchical cluster culling. This does both LOD and visibility and outputs visible clusters.

(=>)

Which are then rasterized to the visibility buffer.

(=>)

HZB is built for the current frame based on what has just been rasterized.

(=>)

Then all the culling stages are repeated, where instances, nodes and clusters found to be occluded based on the previous frame information are retested with the current frame HZB and current frame transforms.

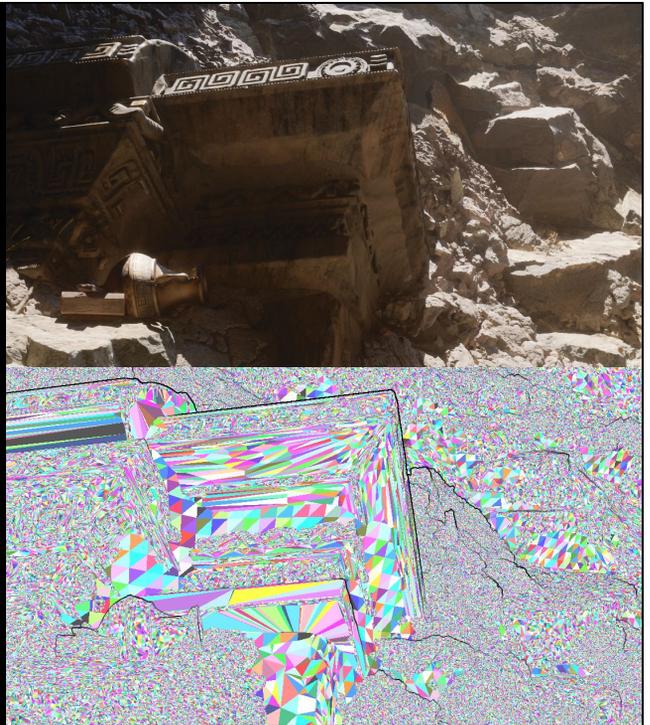
(=>)

Now with the complete visibility buffer we build the HZB for next frame and apply the deferred material passes.

Rasterization

Pixel scale detail

- Can we hit pixel scale detail with triangles > 1 pixel?
 - Depends how smooth
 - In general no
- We need to draw pixel sized triangles



Now that we have fine grained view dependent LOD and we are mostly scaling with screen resolution, how many triangles do we need to draw?
Remember we want zero perceptual loss of detail.

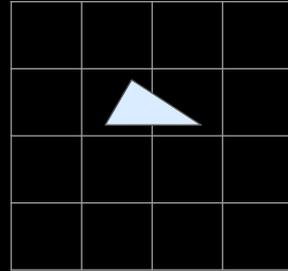
How small do the triangles need to be such that the error is less than a pixel big and is effectively lossless?
Can we do that with triangles that are larger than pixels?

In a lot of cases yes. Triangles are adaptive and can go where they're needed.

But in general no. Pixel sized features need pixel sized triangles to represent them without visible error.

Tiny triangles

- Terrible for typical rasterizer
- Typical rasterizer:
 - Macro tile binning
 - Micro tile 4x4
 - Output 2x2 pixel quads
 - Highly parallel in pixels not triangles
- Modern GPUs setup 4 tris/clock max
 - Outputting SV_PrimitiveID makes it even worse
- Can we beat the HW rasterizer in SW?



Is that practical?

Tiny triangles are terrible for a typical rasterizer, HW rasterizers included
They are designed to be highly parallel in pixels not triangles since that's the typical workload.

Modern GPUs setup 4 tris/clock max and outputting primitive ID needed for vis buffer makes this even worse.

Primitive shaders or mesh shaders can be faster but are still bottlenecked and not designed for this.

Could we possibly beat the hardware with a software rasterizer?

Software Rasterization

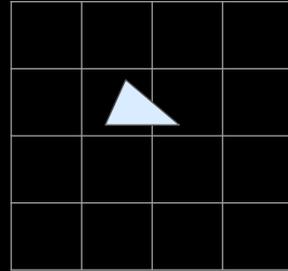
3x faster!

YES!

We can do a lot better. 3x faster than hardware on average compared to our fastest primitive shader implementation. Even more for pure micropoly cases and quite a bit more if compared to the old VS/PS path.

Tiny triangles

- Binning triangles is as much work as just writing the final pixels
- Even a single vector stamp does wasteful tests for small tris
 - Basic bounding box is faster
- Serialization at tile level to handle depth and ROP
- Outputs 2x2 pixel quads
- General purpose
 - VS+PS scheduling
 - Output formats, ordering, blending
 - Clipping
 - ...

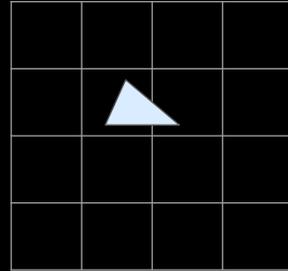


Why? How?

I'm not a hardware engineer but just looking at all the operations a typical rasterizer does, a ton of it doesn't apply to our case or is incredibly inefficient to do for tiny triangles.

Tiny triangles

- Optimized for larger triangles covering many pixels
 - Run wide over pixels
- We want many triangles with few pixels each
 - Run wide over triangles



They are optimized for larger triangles which cover many pixels.
The key thing to parallelize is running wide over pixels

We have many triangles which only cover a few pixels each.
We want to run wide over triangles, not pixels.

Certainly hardware could be built to do this at lower power than us but it's questionable if that's the best use of transistors versus giving us more general compute units we could use for this or anything else.

How to depth test?

- Don't have ROP or depth test hardware
- Need Z-buffering
 - Can't serialize at tiles
 - Many tris may be in parallel for single tile or even single pixel
- Use **64 bit atomics!**
- InterlockedMax

30	27	7
Depth	Visible cluster index	Triangle index

- Visibility buffer shows its true power

If we forego hardware rasterization we also lose the ROP and depth test hardware but we still need to Z-buffer.

We certainly don't want to lock tiles like some SW rasterizers do.

Many triangles may be in flight trying to write to a tile at once, or even a single pixel

We really don't want to lock at all.

Instead we use 64b atomics!

Specifically a global image InterlockedMax to the visibility buffer

This 64b integer has

Depth in the high bits which is what gives us the depth test,

And the payload in the low bits.

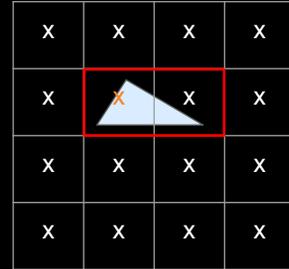
In our case the payload is the visible cluster index and triangle index.

With that detail the visibility buffer shows its true power. The payload needs to be small enough to pack in 34 bits or less.

Without that we wouldn't be able to do fast software rasterization.

Micropoly software rasterizer

- 128 triangle clusters => threadgroup size 128
- 1 thread per vertex
 - Transform position
 - Store in groupshared
 - If more than 128 verts loop (max 2)
- 1 thread per triangle
 - Fetch indexes
 - Fetch transformed positions
 - Calculate edge equations and depth gradient
 - Calculate screen bounding rect
 - For all pixels in rect
 - If inside all edges then write pixel



Here's a basic view of our micropoly software rasterizer.

All the fancy hierarchical tiling and stamps gets thrown out the window. We are left with a super basic half space rasterizer that has been instruction level micro-optimized.

It is kind of similar in structure to mesh shaders. It shares vertex work without any need for a post transform cache.

Threadgroup size is 128

In the first phase a thread is mapped to a vertex from the cluster's vertex buffer

Fetch the vertex position, transform it, and store in groupshared.

If there are more than 128 verts fetch and transform another to support up to 256 per cluster.

Then in the 2nd phase switch to a thread mapped to each triangle with a max of 128 triangles per cluster

Fetch the indexes for this triangle.

Use those to fetch the transformed positions from groupshared.

Calculate the edge equations and depth gradient for the triangle.

Then for all pixels inside the rect bounding the triangle.

Test if inside the triangle and if so write the pixel.

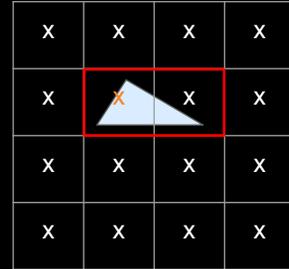
Micropoly software rasterizer

```
for( uint y = MinPixel.y; y < MaxPixel.y; y++ )
{
    float CX0 = CY0;
    float CX1 = CV1;
    float CX2 = CV2;
    float ZX = ZY;

    for( uint x = MinPixel.x; x < MaxPixel.x; x++ )
    {
        if( min3( CX0, CX1, CX2 ) >= 0 )
        {
            WritePixel( PixelValue, uint2(x,y), ZX );
        }

        CX0 -= Edge01.y;
        CX1 -= Edge12.y;
        CX2 -= Edge20.y;
        ZX += GradZ.x;
    }

    CY0 += Edge01.x;
    CV1 += Edge12.x;
    CV2 += Edge20.x;
    ZY += GradZ.y;
}
}
```



Here's the inner loop in a slightly less optimized state.

This iterates over the pixels in the bounding rect, tests if the center is inside all 3 edges

If so it writes the pixel which is simply pack the depth with the payload and atomic max it with to the screen.

Very few iterations of this loop are expected so we shouldn't add any fixed overhead trying to reducing it.

=====

Bonus:

Been explored before in the context of real-time REYES [72]

Didn't use the 64b atomic trick or visibility buffer.

Although they advocate pairing triangles for the coverage tests and claim better performance from doing so I haven't found this to be the case.

Hardware implementations have even been proposed [73]

Hardware Rasterization

- What about big triangles?
 - Use HW rasterizer
- Choose SW or HW per cluster
- Also uses 64b atomic writes to UAV



How about big triangles?

Use the HW rasterizer. It's good at that.

Do the same for any other cases we wouldn't be faster at such as clipping

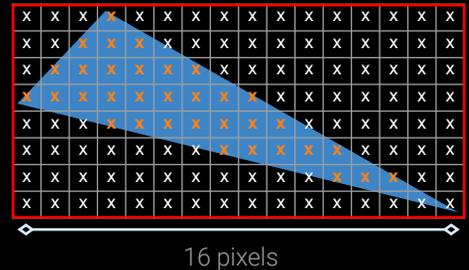
We choose SW or HW per cluster based on which will be faster. The vast majority are SW rasterized in our demos.

We were initially nervous about cracks between SW and HW clusters but thankfully DirectX has a very strict spec for rasterization rules. By following that we can exactly match HW meaning no pixel cracks.

We could use the depth test hardware for HW raster but then we'd have to merge the UAV version and render target version and couldn't async overlap them which we do. Instead the HW rasterizer doesn't bind any color or depth targets and atomic writes to the UAV exactly the same as SW raster.

Scanline software rasterizer

- How big is too big?
 - **Much** bigger than expected
 - Clusters with edges <32 pixels are SW rasterized
- Iterate over the rect tests a lot of pixels
 - Best case half are covered
 - Worst case none are



How big of triangles are too big?

Turns out we can beat the hardware with triangles much bigger than expected, far past micropoly.

We software rasterize any clusters whos triangles are less than 32 pixels long.

That turns into a lot of pixels in the rect to iterate over.

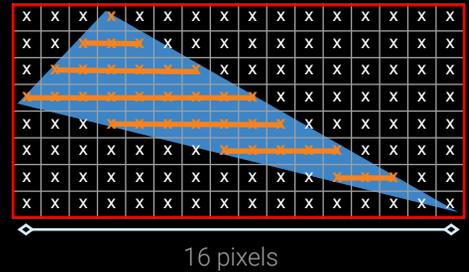
Best case only half of them are covered

Worse case none are

Surely there's something we can do between these extremes.

Scanline software rasterizer

- Scanline could be faster?
- Traditional trapezoid is complicated
 - Lots of setup and edge walking



Could scanline be faster?

A traditional trapezoid approach with edge walking and all that is complicated. Let's not get crazy..

We purposefully are trying to keep the inner loop simple and reduce any extra triangle setup cost.

Micropoly software rasterizer

```
for( uint y = MinPixel.y; y < MaxPixel.y; y++ )
{
    float CX0 = CV0;
    float CX1 = CV1;
    float CX2 = CV2;
    float ZX = ZY;

    for( uint x = MinPixel.x; x < MaxPixel.x; x++ )
    {
        if( min3( CX0, CX1, CX2 ) >= 0 )
        {
            WritePixel( PixelValue, uint2(x,y), ZX );
        }

        CX0 -= Edge01.y;
        CX1 -= Edge12.y;
        CX2 -= Edge20.y;
        ZX += GradZ.x;
    }

    CY0 += Edge01.x;
    CY1 += Edge12.x;
    CY2 += Edge20.x;
    ZY += GradZ.y;
}
```

Can't we know what x interval this will pass?

Let's take a look at that code again.

Why do we need to test each pixel individually to see if it is covered or not?
Can't we know what X interval the test will pass?

Easy, just solve for X.

Scanline software rasterizer

```
float3 Edge012 = { Edge01.y, Edge12.y, Edge20.y };
bool3 bOpenEdge = Edge012 < 0;
float3 InvEdge012 = Edge012 == 0 ? 1e8 : rcp( Edge012 );

for( uint y = MinPixel.y; y < MaxPixel.y; y++ )
{
    float3 CrossX = float3( CY0, CY1, CY2 ) * InvEdge012;

    float3 MinX = bOpenEdge ? CrossX : 0;
    float3 MaxX = bOpenEdge ? MaxPixel.x - MinPixel.x : CrossX;

    float x0 = ceil( max3( MinX.x, MinX.y, MinX.z ) );
    float x1 = min3( MaxX.x, MaxX.y, MaxX.z );
    float ZX = ZY + GradZ.x * x0;

    x0 += MinPixel.x;
    x1 += MinPixel.x;
    for( float x = x0; x <= x1; x++ )
    {
        WritePixel( PixelValue, uint2(x,y), ZX );
        ZX += GradZ.x;
    }
}
...
}
```

This isn't fixed point anymore meaning it's not exactly the same.

Solve for x interval that passes

Now iterate only over filled pixels

So, here is our scanline rasterizer.

Instead of the inner loop iterating from rect min to max testing whether this pixel is in or out, we solve for the x interval that passes and only iterate over those.

Although this isn't exact fixed point math anymore due to a divide we haven't found any issues in practice.

We choose the scanline version if the X loop is >4 pixels for any triangle in the wave

=====

Bonus:

The same approach is taken in [71] but I was unaware of it at the time.

It is entirely possible that some form of work distribution for pixel writes for larger triangles would be faster than what we have, either faster in the range that our scanline rasterizer wins or moving the threshold higher for choosing SW raster. Without a doubt there is a significant amount of divergence with a wave having backfacing triangles that early out, small triangles that only cover a few pixels, and larger triangles that might trigger the scanline rasterizer.

Distributing the work evenly could result in very large gains if the overhead of doing so was low. Unfortunately the triangle setup state is a decently large amount of data. Small in terms of registers but large in terms of groupshared.

It's also important to keep in mind that the inner loop does barely anything, a couple ALU instructions and the atomic max.

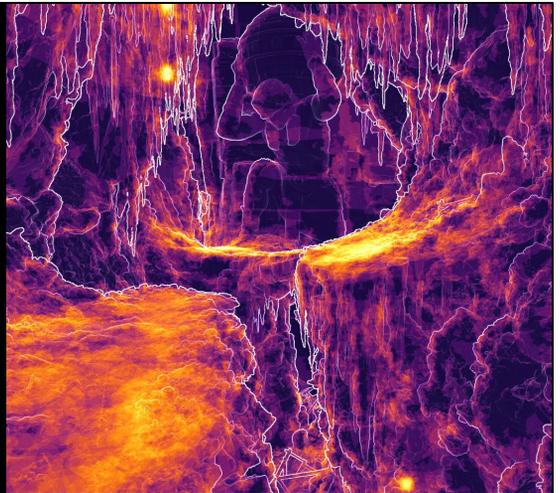
Potentially impactful but untested future work.

Since writing this I actually have tested this and it is not a win in my tests. The point at which triangles become large enough that distributing the work is faster, the HW rasterizer is faster yet.

Basically there is no middle ground between the two. That isn't to say it isn't possible or could vary with hardware architectures. Just reporting my latest tests on the matter.

Rasterizer **overdraw**

- No per triangle culling
- No hardware HiZ culling pixels
- Our software HZB is from previous frame
 - Culls clusters not pixels
 - Resolution based on cluster screen size
- Excessive overdraw from:
 - Large clusters
 - Overlapping clusters
 - Aggregates
 - Fast motion
- Overdraw expense
 - Small tris: Vertex transform and triangle setup bound
 - Medium tris: Pixel coverage test bound
 - Large tris: Atomic bound



You must construct additional polygons

It is interesting to compare our approach with the depth cull hardware.

We don't run material shaders during rasterization so ignore anything having to do with those. We'll get to materials next.

Although there aren't expensive pixel shaders to worry about, in the micropoly case triangles are like pixel work.

We do not have per triangle culling which is like not having hardware EarlyZ. Now this typically isn't an issue but cases where surfaces are overlapped closer than the size of their cluster bounds like the hot spots in this image, or are riddled with small holes, would require culling at a much finer granularity to be efficient.

Riddled with holes sounds like a content problem but actually describes most aggregate geometry cases like leaves and grass. Overdraw is one of many reasons Nanite doesn't perform as well with those.

We also don't have hardware HiZ although we do have HZB culling at a cluster granularity. For dense meshes clusters tend to be similar in size to HiZ tiles so the analogy is a decent one. You can think of it as if we only have HiZ to cull work.

So far we've only discussed triangle work but pixel fill can be expensive too.

Meshes aren't **always** dense. Large triangles mean large clusters.
The larger the clusters, the coarser the culling granularity is in terms of pixels.

This means as triangles get larger pixel overdraw increases.
Interestingly that means there are situations where drawing more triangles is actually faster.

=====

Bonus:

Per triangle occlusion culling isn't done because of the two pass occlusion we do.

Any cluster with an occlusion culled triangle would need to be evaluated and rasterized again which would nullify any savings.

There is also a divergence issue in that we have mapped 1 thread per triangle.

Unless all triangles in a wave are culled or we significantly reduce the longest running thread we won't save much.

Like hardware HiZ we've found more data such as plane equations stored in the HZB can improve culling rate.

We can not exploit the plane equations of triangles as we should assume that many tiny triangles contribute to that tile.

This makes building HiZ data as a streaming operation very hard because to be effective we need occluder fusion.

We don't currently need it to be streaming since we work off of previous frame's HZB.

We can fit depth planes or whatever we want

but if we wanted to improve the issues explained here we really need some form of streaming HiZ data.

The reliance on previous frame depth for occlusion culling is one of Nanite's biggest deficiencies.

Tiny instances

- What happens when entire mesh only covers a few pixels?
- DAG ends at 1 root cluster
 - 128 triangles
 - Stops scaling with resolution
- Cull when too small?
 - Can't if structural building block

We've optimized for tiny triangles but how about tiny instances?
What happens when an entire mesh only covers a few pixels?

The cluster hierarchy only goes so far. It ends at 1 root cluster of 128 triangles.
At that point cost stops scaling with resolution. It stops scaling all together.

That's really small though, we could just cull instances when they get tiny.
Except not if they are a structural building blocks, say a wall section of a building seen
from very far away.

Culling small instances could mean entire buildings vanish.

How often does this happen? Is it even worth addressing?

Since getting their hands on Nanite our artists have pushed instance count just as
much if not more so than polycount.

Joke around the office is instances are the new triangles.

This stresses numerous things we plan to improve but tiny instances were absolutely
a problem.

Tiny instances

- Obviously need to merge at some point
 - Even if rendering scales sublinearly, memory doesn't
- Instance memory adds up fast
 - $10M * \text{float}4 \times 3 = 457\text{MB}$
- Hierarchical instancing desired for the future
 - Instances of instances of instances
- Nanite has no special solution for merging
 - Merged unique proxies must replace instances at extreme distances
 - Key improvement is pushing that distance far away

We will need to merge eventually. Obviously distant lands need to stream out to a cheaper proxy at some point.

Even if we could make the rendering cost for instances scale perfectly, the memory to store them won't.

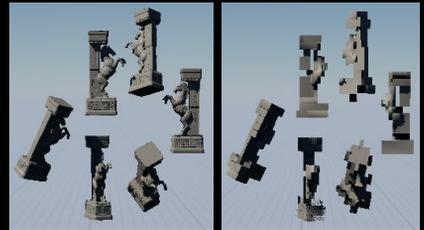
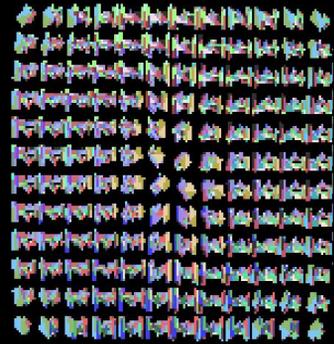
Just the transforms themselves will become too much at some point.

We hope to support hierarchical instancing in the future to change this equation but that doesn't apply to all cases.

We need general purpose merging in the distance but that means unique data. And high resolution unique data gets out of control really fast. We don't want megatexture or megageometry sort of limitations on world scale. Ideally geometry is pixel perfect with no perceptible loss at all distances. So merged proxies need to be pushed as far away as possible and we need a solution before that.

Visibility buffer imposters

- 12 x 12 view directions in atlas
 - XY atlas location octahedral mapped to view direction
 - Dithered direction quantization
- 12 x 12 pixels per direction
 - Orthogonal projection
 - Minimal extents fit to mesh AABB
 - 8:8 Depth, TriangleID
 - 40.5KB per mesh always resident
- Ray march to adjust parallax between directions
 - Few steps needed due to small parallax
- Drawn directly from instance culling pass
 - Bypassing visible instances list
- Would like to replace with something better



Our current solution for that is visibility buffer imposters.

This is your basic static imposters with the small tweak that they don't store color or gbuffer attributes

They store depth and triangle ID from the root cluster.

That can be directly injected into the screen's visibility buffer and supports remapping materials

Non-uniform scale and everything else Nanite supports.

There is definitely a quality loss that can be noticed in some cases meaning switching to it can sometimes be noticed as a pop.

Usually that only happens when many of the same mesh are next to one another like repeating wall sections and the switch is noticeable because there is a bulk change and a directly neighboring thing to compare it to.

Most of the time it works pretty well though.

I would love to replace this with something that is just as fast but consumes less memory and is truly seamless.

Deferred Material Evaluation

Let's move onto materials

Visibility Buffer recap

- Rasterization writes visibility buffer
 - Depth : VisibleClusterID : TriangleID
- VisBuffer decode preamble for material pixel shader:
 - Load VisBuffer
 - Load VisibleCluster => InstanceID, ClusterID
 - Load instance transform
 - Load 3 vert indexes
 - Load 3 positions
 - Transform positions to screen
 - Derive barycentric coordinates for pixel
 - Load and lerp attributes

This is the same thing we covered before but with more detail relative to Nanite

After decoding the visibility buffer we basically have everything a pixel shader normally has to execute.

So we could draw a quad covering the screen, decode the vis buffer and evaluate the material pixel shader almost like it was bound during rasterization.

Material ID

- What material is this pixel?
- VisBuffer decode:
 - VisibleCluster => InstanceID, ClusterID
 - ClusterID + TriangleID => MaterialSlotID
 - InstanceID + MaterialSlotID => MaterialID



Vertex transform might be fixed function in Nanite but we want to support full artist created pixel shaders.

So we don't want just 1 material for all pixels.
How do we know what material this pixel has?
We can derive that from the vis buffer too.

With callable shaders we could in theory apply all materials in a single pass this way but there are complexities and inefficiencies there.

Material shading

- Full screen quad per unique material
- Skip pixels not matching this material ID
- CPU unaware if some materials have no visible pixels
 - Material draw calls issued regardless
 - Unfortunate side effect of GPU driven
- How to do efficiently?
 - Don't test every pixel for matching material ID for every material pass



Instead we could draw a full screen quad per unique material
And skip the pixels not matching this material ID.

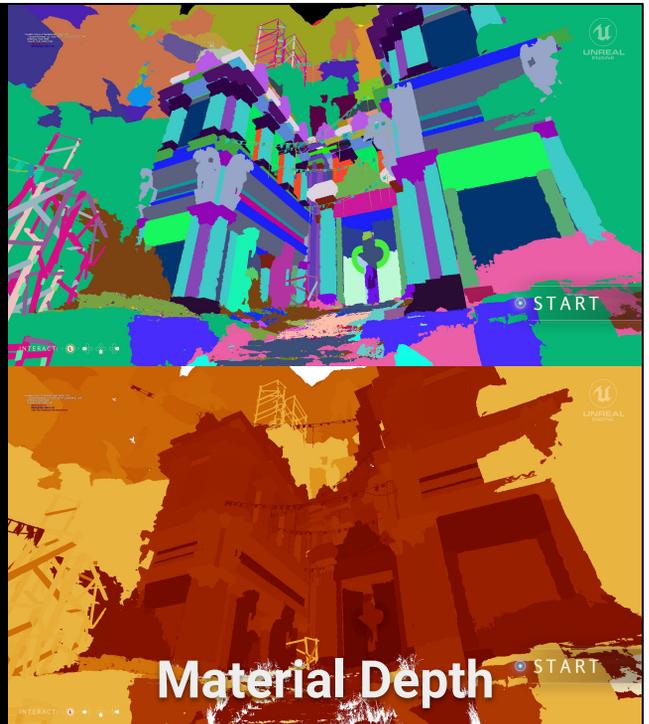
Unfortunately because all culling is GPU driven, the CPU isn't aware of what materials are visible.

Material draw calls have to be issued regardless.

It would be awfully inefficient to test every pixel for every material.

Material culling

- Stencil test?
 - Don't want to reset for every material
- Exploit the depth test hardware!
 - Material ID -> depth value
- Build Material Depth buffer
 - CS also outputs standard depth and HTILE for both depth buffers
- For all materials:
 - Full screen quad
 - Quad Z = Material Depth
 - Depth test set to equals



There's hardware to cull tons of pixels and efficiently pack the surviving pixels in waves.

Stencil sounds like the closest but we don't want to reset the test for every material.

Instead we exploit the depth test hardware.

Material ID becomes the depth value.

On console we can alias memory and know the layouts such that we can make this efficient.

A compute shader outputs both material depth and standard depth that we'll use in later passes.

Along with depth buffers it also outputs HTILE so we get HiZ acceleration.

So for all materials we draw a full screen quad where the quad's Z value is the material depth.

The depth test is set to equals so we only draw pixels with matching ID.

=====

Bonus:

The idea of using depth equals tests for deferred materials comes from Dawn engine [50].

They reduce the pixels covered by the screen quad by atomic min/max the objects using the material's screen rect.

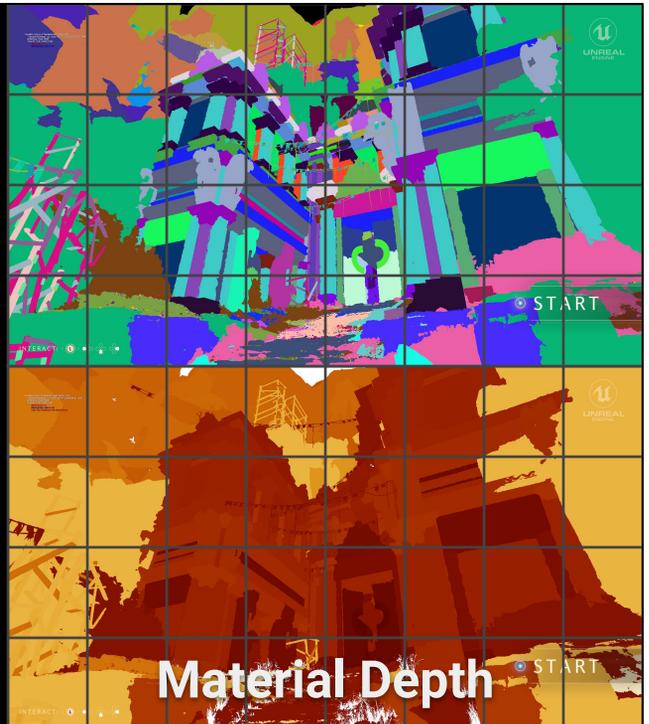
As you can see in these images that won't be very effective in our cases as it is

extremely common for the same material to be assigned to objects on opposite sides of the screen.

The tile approach we explain on the next slide doesn't have this issue.

Material culling

- Material covers small portion of the screen
 - HiZ handles this OK
 - We can do better
- Coarse tile classification / culling
 - Render 8x4 grid of tiles per material
 - Same shading approach as full screen quads
- Tile killed in vertex shader from 32b mask
 - $X=NaN$



Most materials cover only a small fraction of the screen.

HiZ culls does a pretty good job of coarse culling but we can do better.

Instead of a full screen quad we draw a grid of tiles where the tiles are culled according to a 32b mask that was built when material depth was.

This is an area that is being heavily reworked right now and we will likely switch to completely compute in the future.

What I've explained describes how the console path works for the two demos we've shown.

=====

Bonus:

Tiles marked for culling are killed in the vertex shader by snapping X to NaN , preventing any pixel shader waves from spinning up.

Where available, we use rect primitives to avoid diagonal overshade inefficiencies. Unfortunately (and surprisingly), PC APIs do not elegantly and consistently support this yet, but consoles takes advantage of it.

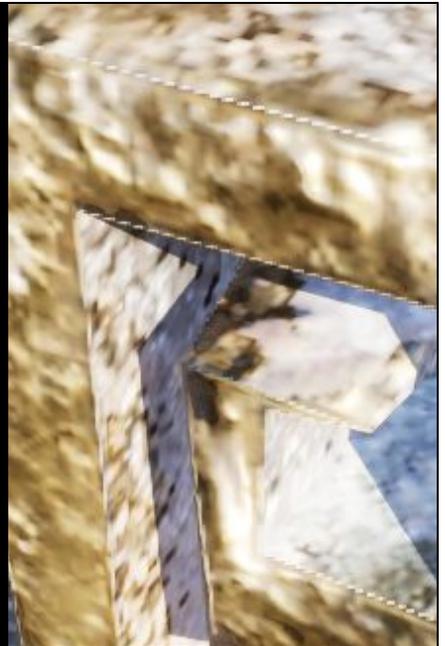
The tile grid resolution varies depending on whether or not wave intrinsics are available.

If wave ops are supported, we make an 8x4 grid per material, represented by a 32bit mask, where each bit corresponds to one of these tiles.

The portable (no wave intrinsic) version has a grid of 64x64 that computes a 64bit mask of materials. This scheme can alias bins, so it can be defeated, though it has worked surprisingly well all things considered. Aliasing would just lose a bit of culling efficiency, though we're still culling by depth at least.

UV derivatives

- Still a coherent pixel shader so we have finite difference derivatives
- Pixel quads span
 - Triangles  Good!
- Also span
 - Depth discontinuities
 - UV seams
 - Different objects } Not good!



Because material are still coherent pixel shaders we still have finite difference based derivatives to use for texture filtering.

Unlike traditional rasterization the pixel quads span across triangles. This is a very good thing because with tiny triangles quad overdraw can get out of hand very quickly.

But quads also span depth discontinuities, UV seams, and even different objects. And that is not good.

Finite differences of discontinuities are nonsense and often huge which results in high mip levels being used like the artifacts in this image.

Analytic derivatives

- Compute analytic derivatives
 - Attribute gradient across triangle
- Propagate through material node graph using chain rule
- If derivative can't be evaluated analytically
 - Fall back to finite differences
- Used to sample textures with SampleGrad

- Additional cost tiny
 - <2% overhead for material pass
 - Only affects calculations that affect texture sampling
 - Virtual texturing code already does SampleGrad



Instead we compute analytic derivatives of the attributes across the triangle.

These are propagated through the artist created node graphs automatically using the chain rule.

If we encounter an operation that isn't analytically differentiable we fall back to finite differences.

We could instead resort to something like ray differentials to be correct but we haven't had issues so far.

All Sample calls are replaced with SampleGrad using our gradients.

All this would appear to add a significant cost but we've measured the overhead as less than 2% for our materials.

The reason this is small is the additional work is only for operations affecting texture sampling

And all virtual texture samples already use SampleGrad to handle tile discontinuities.

=====

Bonus:

Unreal has artist authored material shaders so we need a systematic approach to propagate these.

Mathematically it is a simple matter of applying the chain rule for each operation. Ideally this would be part of the shader compiler like OSL.

Instead this is done at the point we translate the node graph into HLSL.

[77], [78]

Pipeline numbers

Main pass

Instances pre-cull	896322
Instances post-cull	3668
Cluster node visits	39274
Cluster candidates	1536794
Visible clusters SW	184828
Visible clusters HW	6686

Post pass

Instances pre-cull	102804
Instances post-cull	365
Cluster node visits	19139
Cluster candidates	458805
Visible clusters SW	7370
Visible clusters HW	536

Total rasterized

Clusters	199,420
Triangles	25,041,711
Vertices	19,851,262

We've now covered the entire rendering pipeline from start to finish for the primary view.

Here's a look at the sort of numbers that go through each stage

If we rendered this frame using the standard rendering path in UE4 it would have to rasterize over a billion triangles.

Nanite on the other hand rasterizes 25M triangles due to its intelligent LODing and culling.

That 25M is consistent throughout the demo, regardless of how complex the scene is.

And it does this all really fast.

Performance

- Average ~2496x1404 upsampled to 4k
 - TAAU at the time
 - Would use TSR now
- ~2.5ms to draw entire VisBuffer
 - View + GPU scene => complete VisBuffer
 - Nearly zero CPU time
- ~2ms deferred material pass
 - VisBuffer => GBuffer
 - Small CPU cost. 1 draw per material.

Nanite::CullRasterize

Clear VisBuffer	66us
Main Pass: InstanceCull	108ms
Main Pass: ClusterCull	406us
Main Pass: Rasterize	1148us
BuildHZB	99us
Post Pass: InstanceCull	125us
Post Pass: ClusterCull	102us
Post Pass: Rasterize	183us

Nanite::BasePass

DepthExport	217us
Emit GBuffer	2084us

We use dynamic resolution and temporal upsampling to 4k for this demo. The average frame hovers at about 1400p before upsample.

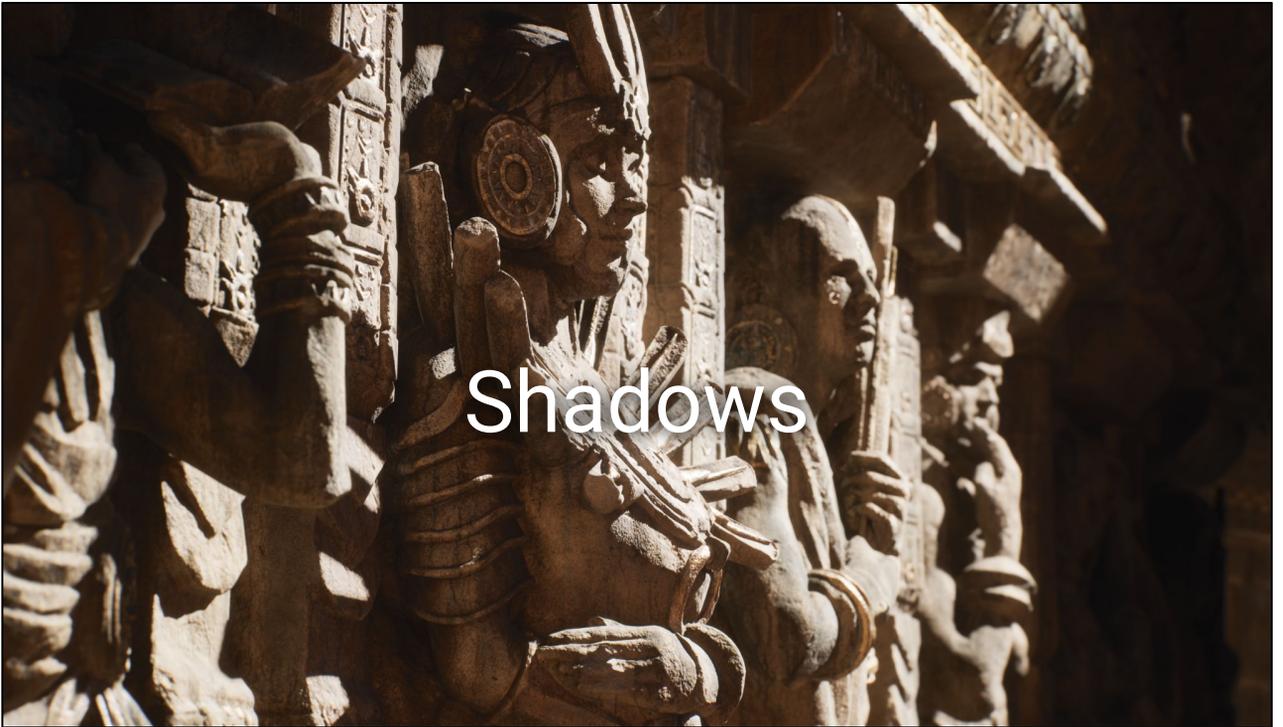
All geometry culling and rasterization, starting from the GPU scene and a view to a complete rasterized VisBuffer takes on average 2.5ms.

Because this is all GPU driven there is negligible CPU cost.

To apply the materials and transform the VisBuffer into a GBuffer takes on average 2ms.

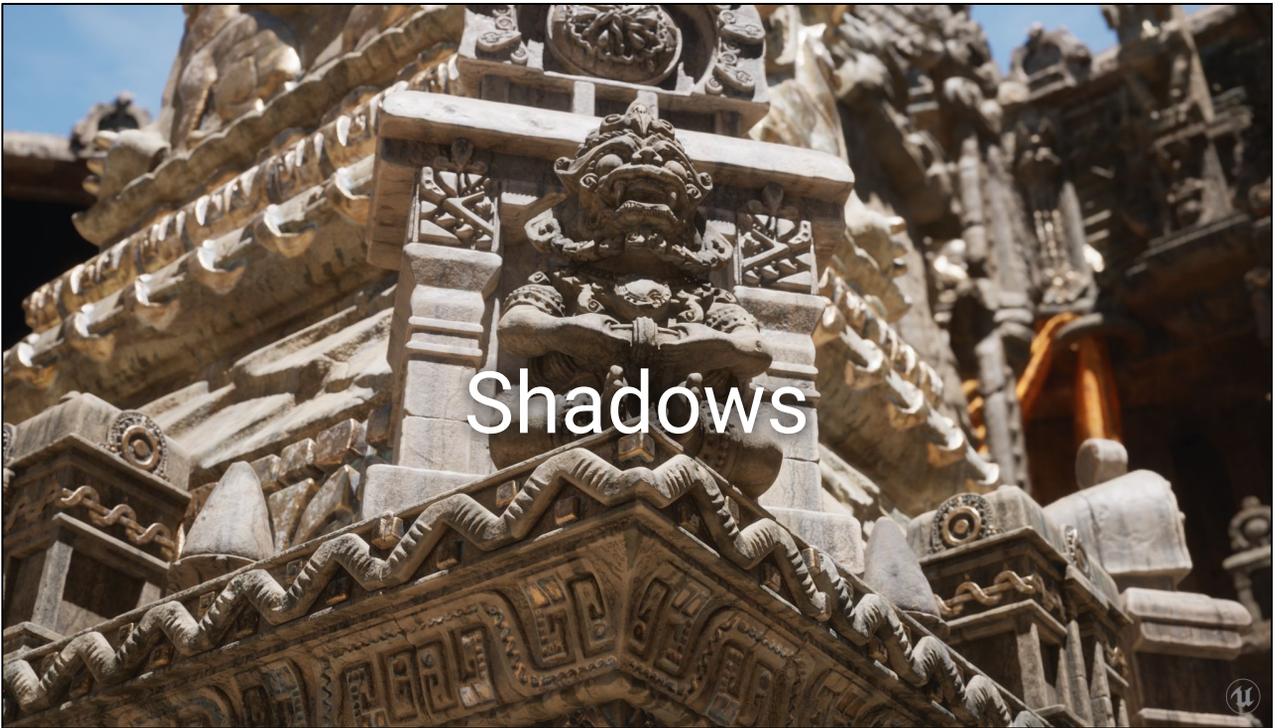
This has a small CPU cost with 1 draw per material.

Well within budget for a 60hz game.



Primary view isn't the only place we need to draw geometry. We need shadows too.

Do we really need micropoly level detail for secondary views?



Maybe not for indirect lighting but shadows need detail.
In fact the largest visual difference between real geometry and normal maps most often comes from detailed self shadowing.

Nanite shadows

- Ray trace?
 - DXR isn't flexible enough
 - Complex LOD logic
 - Custom triangle encoding
 - No partial BVH updates
- Want a raster solution
 - Leverage all our other work
- Most lights don't move
 - Should cache as much as possible

Can we ray trace them?

Unfortunately there are more shadow rays than primary since there are on average more than 1 light per pixel.

We need something at least as fast as what we have for primary.

Even if tracing was fast enough, hardware ray tracing APIs aren't currently flexible enough to

- Evaluate our complex LOD logic
- To not massively bloat memory to conform to their specific triangle format
- And they lack the ability to partially update BVHs to avoid expensive from scratch million element BVH builds

We'll be looking more into ray tracing Nanite in the future but for now we want a raster solution to leverage all our other work.

That number of lights per pixel really needs to be taken seriously.

It is very important Nanite's costs don't multiply out of control due to shadows.

Thankfully most lights and most geometry casting shadows from them don't move.

If we can cache this work there's a chance we can keep this under control.

=====

Bonus:

To make some predictions, ray tracing's advantage with coherent rays comes from

culling irrelevant work.

Compared to a traditional raster pipeline, ray tracing can process significantly less triangles to make up for the significant work it no longer can share.

That culling advantage will be **far** less dramatic when up against Nanite, such that tracing seems almost certain to be slower for coherent rays.

The only way to know is to try of course.

HW triangle formats + BLAS currently are 3-7x the size our memory format which includes attributes. A direct comparison without attributes the ratio is ~60% worse. Compressed triangle and BVH structures are really important to develop.

Virtual shadow maps

- Nanite enables new techniques
- 16k x 16k shadow maps everywhere
 - Spot: 1x projection
 - Point: 6x cube
 - Directional: Nx clipmaps
- Pick mip level where 1 texel = 1 pixel
- Only render the shadow map pixels that are visible
- Nanite culled and LODed to the detail required



Nanite supports normal shadow map drawing but this new architecture enables new techniques that weren't practical before.

It allowed us to implement efficient virtual shadow maps.

We use 16k shadow maps for everything now.

Depending on the light type there might be one or more shadow maps

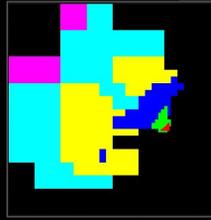
The resolution we rasterize into the shadow map is made to match the screen pixels that those triangles cast onto.

If that region of the shadow map doesn't cast onto anything on screen we don't draw to it.

Compared to more common approaches to optimize shadow rendering, not only are we culling the rasterization work, we don't even allocate memory to shadow map space that we aren't going to sample.

Virtual shadow maps

- Page size = 128 x 128
- Page table = 128 x 128, with mips
- Mark needed pages
 - Screen pixels project to shadow space
 - Pick mip level where 1 texel = 1 pixel
 - Mark that page
- Allocate physical pages for all needed
- If cached page already exists use that
 - And wasn't invalidated
 - Remove from needed page mask



These 16k shadow maps are virtualized and sparse.
Page size is 128 so mip0 is 128 by 128 pages

Allocation of the virtual memory every frame is fairly straightforward:

- For each pixel on screen
- For all lights affecting this pixel
- Project the position into shadow map space
- Pick the mip level where 1 texel matches the size of 1 screen pixel.
- Mark that page in that level as needed

We support caching too so we avoid drawing anywhere in the shadow map that we've already covered in a previous frame.

This means for the most part the only regions of the shadow maps that are updated each frame are those where objects are moving or edges of the frustum as the camera moves.

=====

Bonus:

The calculation of texel size only accounts for the perspective warp, not the projection (which is unbounded).

This is to say we don't use the slope of the surface to the shadow map projection.

That is normally done for mipmap level calculation but at least with the content we've tested so far is too noisy in practice.

Since it is unbounded it needs to be clamped and craggy rocks like these push all

page selection to the limit, hence the limit basically becomes a global mipbias. If most surfaces were smooth it would probably be a good idea though.

Because we don't account for projection warp in mip selection we suffer from projection aliasing.

Multi-view rendering

- Deep pipeline with significant synchronization overhead
- $\text{NumShadowViews} = \text{NumLights} * \text{NumShadowMaps} * \text{NumMips}$
- No reason different views can't be culled and rasterized simultaneously
 - Amortize the cost!
 - Tag elements with view id

Although we support all meshes rendering into virtual shadow maps, combined with Nanite is really where it shines.

But first we needed to make some modifications to the Nanite rendering to make it efficient.

The Nanite pipeline is actually fairly deep with significant synchronization overhead. That's tiny proportional to the cost of a full resolution primary view but spinning up the pipeline for a minor amount of work can be very inefficient.

So we don't want to call Nanite render many times, like once for each light
Or worse,
for each mip level,
for each shadow map,
for each light.

Instead we added multiview support to Nanite.

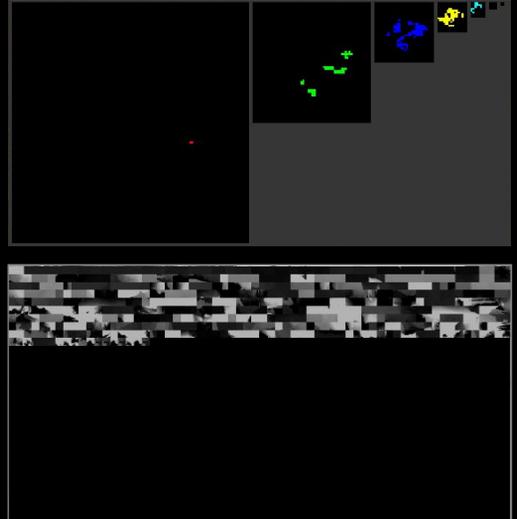
The Nanite pipeline gets an array of views to render into instead of just one.

Now not only can Nanite draw the entire scene with a single chain of dependent dispatch indirects. It can render all shadow maps for every light in the scene, to all of their virtualized mipmaps at once.

In extreme cases we've seen a 100x speedup compared to individual calls.

Culling and addressing to pages

- Cull if doesn't overlap needed pages
 - Similar to HZB test
- SW raster:
 - Emit cluster per overlapped page
- HW raster:
 - Page table indirection per pixel before atomic write



To cull work that would draw to unmapped regions of the shadow map we slot in an extra test next to where we normally test the bounding rect against the HZB. For shadows it also tests against the needed pages mask. If an instance or cluster doesn't overlap a needed page it is culled.

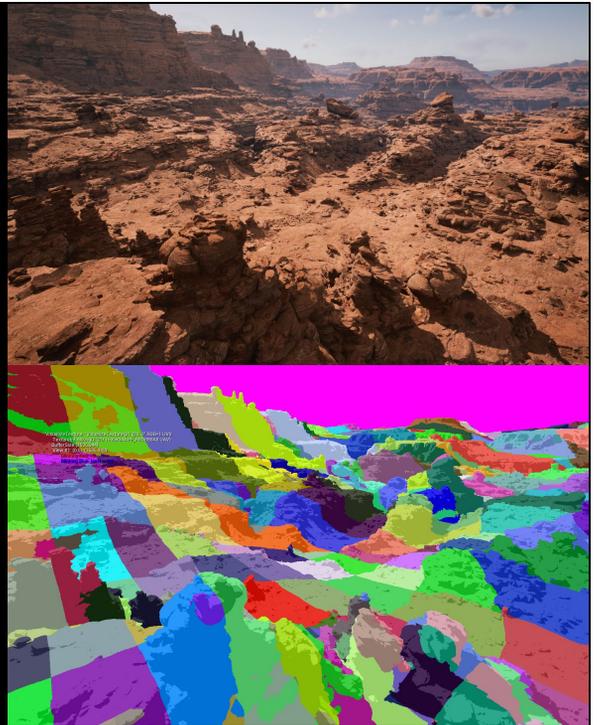
The physical texture we are writing to isn't contiguous in virtual space. This means clusters that overlap multiple pages can't expect the addressing of a pixel to be direct. We have multiple approaches to handle that.

For the software rasterizer it is best to keep the inner loop as simple as possible. We've found even a single additional shift in the inner loop is measurable. So instead we emit 1 visible cluster to the rasterizer per overlapped page do the page translation once for the cluster and scissor to the page pixels. SW clusters are small so most overlap a single page.

Hardware clusters are bigger, often overlap multiple pages and duplicating the vertex and triangle cost doesn't make sense. Instead we do the virtual to physical page table translation per pixel. Because we are doing atomic UAV writes, even in the HW path, we are free to scatter them.

Nanite shadow LOD

- Render pages with 1 texel = 1 pixel
 - NumPages proportional to screen pixels
- LOD matching < 1 pixel of error
 - NumTriangles proportional to shadow pixels
- Shadow cost scales with resolution!
 - Not scene complexity
 - * NumLights per pixel



Nanite's fine grained culling is hugely helpful for efficient virtual shadow maps but so is the level of detail.

Just like in the primary view, Nanite picks the LOD matching 1 pixel error. In the case of shadows this means the pixels of the mip level it is rasterizing to. And remember the mip level was chosen such that 1 texel = 1 pixel. This maintains the property of roughly scaling cost with screen resolution not scene complexity.

That does not mean the triangles drawn to the shadow map are exactly the same as those drawn in the primary view.

That mismatch can cause incorrect self shadowing. We address that discrepancy with a short screen space trace to span the zone where they could differ.

=====

Bonus:

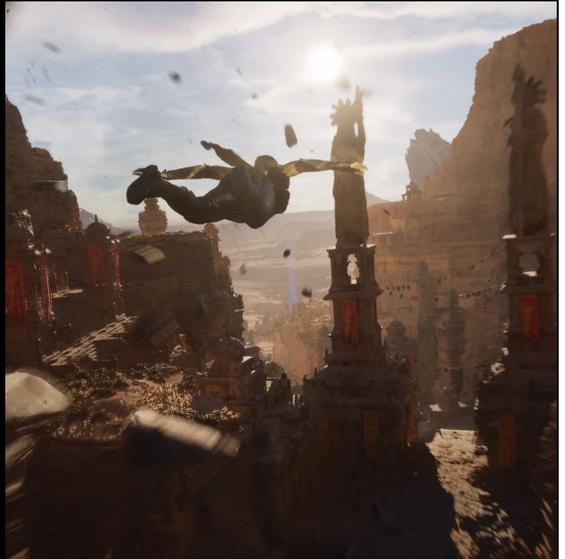
We actually apply a Nanite LOD bias for shadow rendering to account for the multiplier of shadow rays over primary. Defaults to 2 pixels of error.

Because we already have to handle a mismatch between primary view triangles and shadow map triangles with the screen trace this reduction of detail is barely noticeable and worth the savings.

Streaming

Streaming

- Virtualized geometry
 - Unlimited geometry at fixed memory budget
- Conceptually similar to virtual texturing
 - GPU requests needed data. CPU fulfills them.
 - Unique challenges
- Cut DAG at runtime to only loaded geometry
 - Needs to always be a valid cut of full DAG
 - Similar to LOD cutting. No cracks!



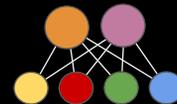
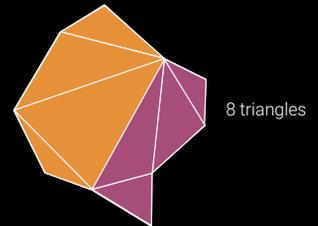
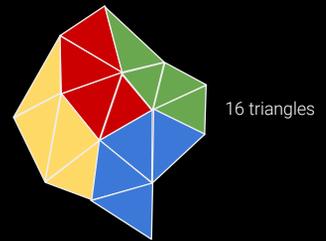
The virtual memory analogy for geometry is conceptually very similar to virtual texturing, although the details are different and there are some unique challenges.

It is similar in that the GPU requests data where it finds the quality is insufficient and the CPU asynchronously fulfills those requests by loading the data from disk.

It is different in that when we load and unload data we need to make sure it is always a valid cut of the full DAG, so there will be no cracks in the geometry.

Streaming unit

- A cluster can overlap any parent cluster
 - Rendering cluster => None of the parents should render
 - Parent not rendered => All siblings (or their descendants) need to render to fill the hole
 - Full group needed before a cluster in the group can be rendered
- We should stream at **group** granularity
- Geometry has **variable size**
 - Use fixed-sized pages to avoid memory fragmentation
 - => Variable amount of geometry per page



So, what do we stream? What is the smallest streamable unit?

Remember how simplification happens at the group level.

By construction, a cluster group is exactly replaced by its parents.
Conversely the parents can only be replaced by the full group of children.

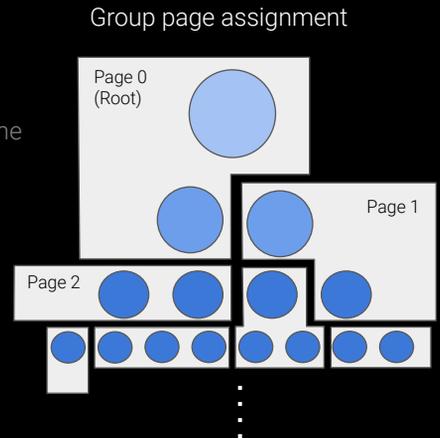
So we can't start rendering a cluster before all the siblings in the group are also loaded otherwise it won't be a complete replacement.
Thus there is no reason to stream at finer-than group granularity. Doing so could even result in incorrect rendering!

Another difference compared to virtual texturing is that geometry has variable size, even before compression. Clusters have a variable number of vertices, attributes, and so on.

To avoid memory fragmentation, we want to use fixed sized pages, but this also means we have to fit a variable amount of geometry per page.

Paging

- Fill fixed-sized pages with groups
 - Based on spatial locality to minimize pages needed at runtime
- Root page
 - First page contains top level(s) of DAG
 - Always resident, so we always have something to render
- Page contents:
 - Index data
 - Vertex data
 - Meta: Bounds, LOD info, Material tables, etc.
- Resident pages stored in one big GPU page buffer



So we fill fixed-sized pages with cluster groups.

We account for both spatial locality and level in the DAG when deciding which groups to put on the same page.

We do this to try to minimize the number of pages that are likely to be needed at runtime.

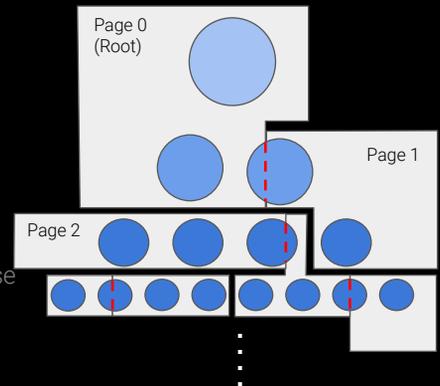
The first page, the root page, is always resident and contains as much of the top of the DAG as we can fit in a page.

Always having the root page means we always have something to render, no matter what.

The resident pages are stored in one big ByteAddressBuffer on the GPU.

Group parts

- Slack
 - Clusters are small (~2KB)
 - Groups can be large (8-32 clusters)
 - Significant slack if allocating whole groups
- Split groups
 - Group can span multiple consecutive pages
 - Splitting happens at cluster granularity based on memory use
 - Group not activated before all parts are loaded
 - Page now filled at cluster granularity (~2KB)
 - ~1KB average slack per page! (~1% slack for 128KB page)



Unfortunately, filling pages with whole groups can result in a lot of slack, because groups can be quite large.

They range from 8 to 32 clusters and each cluster is up to about 2KB.

Our solution is conceptually simple. We split the groups into multiple parts at cluster granularity.

Because, as previously mentioned, a cluster cannot be drawn before all of its siblings are loaded, a split group is only considered active when all of its parts are loaded.

By filling pages at cluster granularity instead of groups, we only incur around 1% waste.

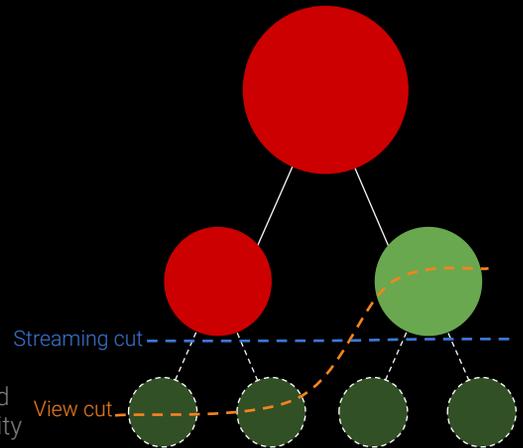
=====

Bonus:

For convenience these parts are always allocated to consecutive pages, so that a group's pages can always be requested as a single range of pages.

Deciding what to stream

- Easy for Virtual Texturing
 - Given directly by UVs and gradient/LOD level
- For Nanite this requires hierarchy traversal
 - Find clusters that would have been drawn
 - Requires looking beyond the streaming cut
- Full culling hierarchy always resident
 - It is metadata about cluster groups (tiny)
 - Traversal can go several levels beyond what is loaded
 - Immediately request all levels needed for target quality



So how do we decide what to load?

For Virtual Texturing this is easy. The data that needs to be loaded can be determined directly from the UV coordinates and the selected LOD level or UV gradient.

For Nanite, we can't find the data directly from the query position and LOD error, we have to traverse the hierarchy to find it.

During the traversal we have to look at nodes beyond the current streaming cut and determine if we would have drawn them if they were resident.

This means that the culling hierarchy needs to cover more than what is currently streamed in. The hierarchy is small as it only contains meta-information about cluster group frequency, so for simplicity, we always keep the full hierarchy resident.

This has the added benefit that traversal is independent of the current streaming cut and can go several levels beyond it.

A new object can immediately request data from all the levels it needs for target quality.

Had the hierarchy itself also been streamed, this would have to be happen level by level over several frames.

IO latency would be multiplied by the number of levels we are off, which would translate to more visible pop in.

Streaming requests

- Persistent shader outputs page requests during culling traversals
 - Requests page ranges with priority based on LOD Error
 - Updates priorities of already loaded pages
- Asynchronous CPU readback of requests
 - Add any missing DAG dependencies
 - Issue IO page requests for pages with highest total priority
 - Evicts low-priority pages
- Handle completed IO requests
 - Install pages on GPU
 - Patch GPU-side pointers
 - Fix pointers to groups in new loaded/unloaded pages
 - Fix pointers to split groups that are now complete or no longer complete
 - Mark/Unmark clusters as leaves

In practice, the streaming requests are generated as part of the hierarchical cluster culling traversal.

A request consists of a range of pages with a priority based on the LOD Error.

They are generated not only for the primary view, but also any active shadow views although those are considered lower priority.

It is also worth noting that the requests are emitted even for resident pages to update their priorities.

The request are read back asynchronously by the CPU. It adds any missing DAG dependencies and issues IO requests for the highest priority pages.

It also evicts low-priority pages to make room in memory.

Finally, when the IO requests are ready, the page data is installed on the GPU and the GPU-side data structures are updated.

This includes fixing pointers for loaded/unloaded pages,

fixing pointers for split groups that have been completed or are no longer complete.

And it involves marking and unmarking clusters as leaves.

Compression

The last topic I'll be talking about is compression

Two representations

- Memory representation
 - Used directly for rendering
 - Near-instant decode time
 - Needs to support random access from visibility buffer
 - Quantized and bit-packed
 - Goal: Save memory and/or bandwidth
- Disk representation
 - Transcoded to memory representation as data is streamed in
 - Can afford significantly higher decode cost
 - Doesn't need random access
 - Assumes data will be compressed by (hardware) byte-based LZ
 - Goal: Reduce compressed disk size

For Nanite, we actually have two compressed geometry formats. They represent the same data but are optimized for different goals.

The memory representation is what is used directly by the rendering code, during rasterization and for the deferred material passes. This needs to be near-instant to decode and random-access as any triangle can be requested from a visibility buffer lookup.

The goal here is to keep the memory required for the streaming pool down. Even if we had the memory to burn, fitting more data in the cache means less cache misses which means less IO and chance for pop in.

We have a separate disk representation that gets transcoded into the memory representation as data is streamed in. This format doesn't require random access and because streaming happens at lower frequency than rendering, we can afford more much advanced techniques here. We assume that data on disk will be compressed by some sort of byte-based LZ algorithm. So the goal with this representation is to minimize the disk footprint AFTER compression.

Vertex quantization and encoding

- Global quantization
 - A combination of artist control and heuristics
- Clusters store values in local coordinates
 - Relative to value min/max range
- Per-cluster custom vertex format
 - Uses minimum number of bits per component: $\text{ceil}(\log_2(\text{range}))$
 - Vertex bitstream requires vertex declaration to be decoded
 - Just a string of bits, not even byte aligned.
- Decoded using GPU bitstream reader
 - Divergent formats => Refill on every read?
 - Reads specify a compile-time bound on read size
 - Only refill when accumulated compile-time read size overflows

For the memory representation, first the positions and the various attributes are first quantized globally based on a combination of explicit artist control and heuristics.

Clusters store these quantized values in local coordinates relative to the min/max range of values in the cluster.

The values are stored using the minimum number of bits required to represent the value range of that component.

This means that instead of using a fixed vertex format for the entire mesh or for all clusters, every cluster uses a vertex format specialized to just what it needs based on the values it spans.

Each vertex is just a fixed length string of bits. There is no need for alignment, even to whole bytes.

Because sizes are still fixed-width inside a cluster, this still supports random access and is still relatively simple to decode, although it now requires a compact vertex declaration to be fetched to make sense of the bits.

=====

Bonus:

A few notes on this from the decoding side.

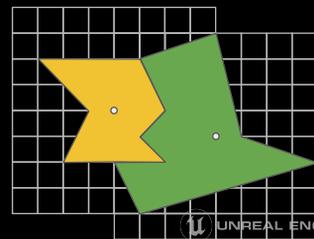
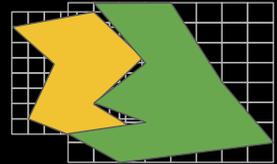
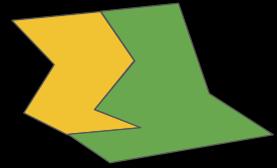
As this is the format used directly for rendering, this has to be decoded with a bitstream reader on the GPU.

To avoid having to check if the bitstream needs to be refilled at every read, a compile-time upper bound for the bitsize of every read is specified.

We know that only when the accumulated compile-time upper bound overflows, could the bitstream be running out of bits and only in that case does it need to be refilled. Especially with divergent lanes, this can end up being a significant saving.

Vertex positions

- Inconsistent quantization leads to cracks!
 - Easy to avoid for single object
- How do we avoid cracks between objects?
 - Common to build level geometry out of modules
- Quantize to object space grid
 - Per-object absolute power of two step size (e.g. 1/16cm)
 - Centered at object origin
 - Don't(!) normalize to bounds
- Vertices fall on same grid when:
 - Quantization level is the same
 - Translation between objects is also a multiple of the step size
- Only leaf level is perfectly aligned
 - Simplifier decisions are not consistent across objects
 - Runtime LOD decisions are not synchronized



When quantizing, you have to be particularly careful when it comes to positions as inconsistencies can and will lead to very visible crack in the geometry.

This is fairly manageable when just considering single objects, but in practice we also have to deal with cracks that appear between objects.

This is a particularly common problems when level geometry is built out of modular pieces, which is something we encourage with Nanite because of its support for very high instance counts.

Because we have no information about where meshes are placed in the level when the geometry is built, there can be no negotiation between the objects.

In practice, this means that the objects have to be quantized to the same global grid.

We achieve this by performing quantization in object space using a user-selectable power of step size centered around the object origin.

It is crucial that the step size is not normalized to the bounds of the object or in other ways tied to its dimensions.

While this doesn't solve all problems, this sort of quantization guarantees that as long as objects use the same quantization level, the same scale and that the translation between them is also a multiple of the step size, they are quantized to the same grid and their vertices will be aligned perfectly. This actually also works for rotations as long as they are by some multiple of 90 degrees.

This ends up covering most cases we care about in practice.

Unfortunately, this only holds for the leaf level of the hierarchy. At higher LODs the simplifier could have made different decision along the shared boundary and at runtime LOD decisions will not be perfectly synchronized.

This still works fairly well in practice as only the leaf data can be observed up close and the higher LOD levels already target at most pixel error.

Memory - Triangles and attributes

- Triangles
 - Base Index and two positive 5 bit offsets
 - Builder guarantees triangle indices never span more than 5 bits.
 - Typically ~17 bits per triangle (7+5+5)
- Texture coordinates
 - Needs to handle UV seams
 - Excludes largest gap from U and V ranges
- Normals:
 - Octahedral coordinates
- Tangents:
 - Implicit. 0 bits!
 - Derived from UV gradients of the triangle

Triangle indices are rotated, so the first index of the triangle is the one with the lowest index.

That is stored using the minimum number of bits required for the cluster, typically 7. The remaining two indices are stored as positive 5 bit deltas from the first index. This comes from the fact that our builder is guaranteeing that triangles never span more than a 32 index window.

Because seams are common for UV data, they use an encoding that excludes the largest gap in the range, effectively making it two ranges per component.

Normals use an encoding based on octahedral coordinates.

Tangents are not stored at all. They are implicitly derived per pixel from the UV gradients of the triangle instead.

This works particularly well for the type of very-highly tessellated geometry often used with Nanite, but we are planning to also support explicit tangents for the cases where it is necessary.

Implicit tangent space

- Implicit tangent space: 0 bits!
- Tangent/bitangent are the U/V directions in the view-space normal plane. Derive them!
- Similar to screen-derived tangent space (Mikkelsen)
 - Calculated directly on triangle coordinates instead of using screen ddx/ddy
 - Reuse local triangle uv/position deltas already calculated for barycentrics and texture LOD calculations in material pass
- Traditional explicit tangent frames are relaxed by averaging with neighbors.
 - Less important for high-poly meshes.
 - We will also support explicit tangents in the future.

Instead of storing and interpolating an explicit tangent frame, we derive it at runtime. The tangent and binormals are just the U and V directions on the normal plane after all.

This is a very similar to the screen space derived tangent space proposed by Morten Mikkelsen and others.

Except we don't actually use screen space derivatives. Instead we use the triangle deltas we already need to calculate for the barycentric interpolation and texture LOD calculations in the material pass.

There are potentially some continuity problems with this approach compared to explicit tangent space.

But for high-poly models it is generally much less of a problem. We will also support explicit tangents in the future.

=====

Bonus:

Specifically, ours is based on the derivation from Schlüler [84]

Material Tables

- Each cluster stores material table
 - Specifies triangle ranges of material assignment
- Two encodings alias the same 32 bits
 - Fast path encodes 3 ranges
 - Slow path points to memory, max 64 materials

A mesh can have multiple materials assigned to it.
We need to store which triangles are assigned to which materials.

To keep this small we could have forced clusters to only have a single material assignment and basically treated them like separate meshes.
This is too restrictive for simplification as well as very complex to handle in terms of cracks in the build process.

It is best to keep this per triangle data of a cluster.
To keep it small triangles are sorted by material and the ranges associated with each material are stored.

Most clusters have 3 or less materials assigned to them and the table fits in 32bits.
Clusters with more than 3 materials follow an indirection to a variable sized table that fits up to 64 materials.

In either case the ranges are searched for which contains the triangle index in question.

Disk representation

- Hardware LZ decompression
 - In consoles now
 - On its way to PC with DirectStorage.
 - Unbeatably fast, but general purpose
 - String deduplication and entropy coding
- For better compression
 - Domain-specific transforms
 - Assume data will be LZ compressed
 - Focus on redundancies not already captured by LZ
 - Transcode to more compressible format

For the disk representation we leverage hardware LZ decompression on the platforms that support it.

It is already available in consoles today and it will be making its way to PC soon with DirectStorage.

Hardware LZ decompression is unbeatably fast at what it does, which is essentially entropy coding and strip deduplication.

As we expect that hardware decompression will increasingly become the norm, we want to make sure to design our formats to leverage it.

Designing new formats that use their own custom entropy coding back ends feels like a waste of time at this point.

So instead of building a special purpose compressor we instead specialize the data to the compressor by applying domain specific transforms to it, focusing on the redundancies not already captured by the LZ compression, and massaging the data to better fit how LZ works.

GPU transcoding

- Transcode on the GPU
 - High throughput for parallel transforms
 - Lots of (async) compute per byte as long as it is parallel
 - Currently runs at ~50GB/s with fairly unoptimized code on PS5
 - Eventually stream data directly to GPU memory
- Powerful in combination with hardware LZ
 - LZ handles inherently serial entropy coding and string matching work
 - Likely to become a common pattern this generation
- Nanite: GPU has the context
 - Pages can reference data from parent pages, without requiring a CPU copy.

With the inherently serial entropy coding and string matching work handled by LZ, it is now much more practical to do compression work on the GPU, because it only has to do transforms which are often more parallel in nature.

As long as it is parallel, much more advanced transformations are practical compared to what would be possible on the CPU, especially if it runs in async compute, although we haven't had the time or need to do this yet.

Our current transcoding scheme already runs at about 50GB/s on PS5 with fairly unoptimized code.

In the specific case of Nanite, transcoding on the GPU also has the benefit that the GPU transcoder can reference parent data in already resident pages, without having to keep an additional CPU copy around.

=====

Bonus:

Doing transcoding on the GPU also opens the door to potentially streaming the data directly from the drive to GPU memory in the future, without involving the CPU at all.

Generally, we think that the combination of hardware decompression and GPU transcoding is a powerful one and a pattern we expect to see more and more this generation.

General LZ optimization

- Memory representation is compact, but not ideal for LZ
- Pad data up to byte alignment
 - Matches will be missed when the bit-alignment doesn't match
 - Byte statistics garbled by misalignment
 - Increases raw size, but decreases compressed size
- Reorder to minimize match offsets
 - References are typically to data of the same type
 - Reorder data by type
- Favor small byte values
 - More skewed byte statistics => More effective entropy coding

So how are we massaging the data to better fit how LZ works?

LZ compression is byte-based, which is a poor fit for our completely unaligned bitstream data.

Padding data up to byte alignment can help LZ find much better matches as it would otherwise only be able to match strings that happen to have the same bit-alignment. Although this increases the uncompressed size, it can be a noticeable win for compressed size.

Padding can also help entropy coding as unaligned data can easily end up garbling the byte statistics.

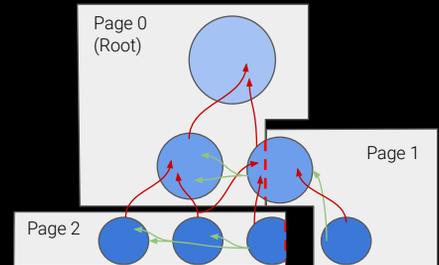
Generally, we make sure to order data of the same type close together so match offsets are minimized.

Whenever we have the choice, we favor byte values that are smaller to make the byte statistics as skewed as possible.

The more skewed the statistics, the more the effective entropy coding is going to be.

Disk - Vertex deduplication

- Clustering generates duplicate vertices
 - Shared boundary vertices
 - Vertices not affected by simplification
- Not recognized by LZ
 - Clusters use different encodings
 - Parent page data only visible to GPU
- Store references
 - Only to same or parent pages
 - Parent page always installed before child
 - Topology sort. Batched to minimize syncs
- Bitstream translated to local vertex format
- ~30% of vertices coded as references!
- Future: Big opportunity to use parents for prediction!



And now on to the more domain-specific transforms.

The process of clustering the data and generating the cluster hierarchy ends up generating a lot of redundancy.

Breaking the mesh into independent clusters generates duplicated vertices along all the shared edges.

Also, during simplification it is not uncommon that some vertices are unaffected, so they are identical to the source vertex in the child.

These types of redundancies are generally not recognized by the LZ compression as the duplicate vertex likely has a different encoding in the other clusters.

In the case of a parent page reference, LZ can't even see it as the data is only available in the GPU page pool.

Instead of redundantly storing the duplicate vertices, references are stored instead. Because of streaming, we can't just assume any page data is available for referencing, but we can rely on the parent pages being loaded as it is already required by rendering that the loaded data is a valid cut of the DAG.

So we can reference data both from clusters within the current page or any of its parents.

Typically about 30% of vertices in a cluster can be encoded as references.

Because the parent data is essentially just a simplified version of the child data, we expect that there are still significant wins ahead by extending this sort of mechanism to prediction and not just exact matching.

=====

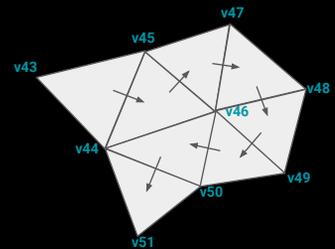
Bonus:

When multiple levels are streamed in at once the streamer has to perform a sort of topology sort to ensure parents are installed before the children.

When a reference is decoded the vertex bitstream has to be decoded and reencoded into the format of the current cluster.

Disk - Topology encoding

- Custom topology encoding
 - Based on generalized triangle strips
 - Vertices reordered so first reference to every vertex can be omitted
 - Coded References are deltas from highest seen vertex (5 bits)
 - Random access using bitscan/popcount of control bitmasks
 - ~5 bits/tri vs ~17 bits/tri for memory format
- Originally intended for memory format
 - Moved to transcoding because it wasn't quite fast enough
 - Faster encodings possible for memory
 - Denser encodings possible for disk
 - Future work



IsReset	IsLeft	IsRef	Ref Value
...
1	0	0	-
0	0	0	-
0	1	0	-
0	0	0	-
0	0	0	-
0	0	0	-
0	0	1	6
0	1	0	-
...

Only explicitly stored reference
Reference to v44. Highest seen vertex v50
Delta: 6

Finally, the disk encoding also uses a more compact topology encoding.

Instead of storing 3 indices per triangle, the triangles are rearranged into a number of triangle strips.

A trisrip requires 3 indices for the first triangle, but any subsequent triangles in the strip only require a single new index.

The longer the strips are, the fewer indices are necessary on average.

We are using generalized trisrips that, instead of alternating between left and right, allow you to explicitly choose between left and right at every step.

We have found that this allows us to form much longer strips and that ends up being a net win even if it requires a bit per triangle to choose between left and right.

We also reorder vertices by first use, so the index on the first reference to a vertex will always just be the total number of vertices seen so far and thus it doesn't need to be stored explicitly.

References to already seen vertices still need to be explicitly stored, but here again, the builder guarantees that these can never be more than a 5-bit offset from the highest seen vertex.

The resulting structure is a number of bitmasks that indicate when to reset the strip, when to go left or right and which references are explicit.

At ~5bits/tri, this encoding ends up being significantly more compact than the memory

encoding at ~17bits/tri.

=====

Bonus:

Using bitscan and popcount the structure supports random access, so any triangle can be decoded in constant time.

The encoding was originally intended for the runtime memory format, but the decoding ended up not quite being fast enough, so it was moved to the transcoding stage where it is plenty fast.

For this reason, it ends up not being quite the right trade-off for either.

We have some ideas that already look promising.

Results: Lumen in the Land of Nanite

- 433M Input triangles, 882M Nanite triangles
- Raw data: 25.90GB
 - Full floats, byte indices, implicit tangents
- Memory format: 7.67GB
 - Compressed: 6.77GB
- Compressed disk format: **4.61GB**
 - ~20% improvement since Early Access
 - 5.6 bytes per Nanite triangle
 - 11.4 bytes per input triangle
- **1M triangles = ~10.9MB** on disk
- Increased focus going forward



And now some concrete results. In the “Lumen in the Land of Nanite” demo there were ~433M source triangles.

Which ends up generating about twice as many Nanite triangles because of the cluster hierarchy.

The raw uncompressed data, assuming positions, normals, and UVs are stored in floats and indices in bytes, while still using implicit tangents would be about 26GB. Using half precision coordinates, this would be closer to half, but still significantly more than what we get with our memory representation, which gets us down to 7.7GB.

If we were to just compress and store this on disk directly, it would only be about 10% smaller.

The exact same data represented using the disk format gets compressed down significantly further to just 4.6GB.

This is a footprint of about 5.6 bytes per Nanite triangle, including all the necessary cluster and hierarchy metadata.

Or in terms of source triangles this is about 11.4bytes / triangle.

These are all numbers from latest code which has improved since the early access build.

Compression will be an increased focus going forward. We believe there is a lot left to squeeze.

=====

Bonus:

Compressed sizes are on PC using Kraken at Compression Level 5 as the backend LZ compressor

Future

- Focused on rigid geometry first
 - >90% of the geometry
 - Object movement supported
- Does not yet support:
 - Translucent or masked materials
 - Non-rigid deformation, skeletal animation, etc
- Not great with aggregates
 - Many tiny things becoming a porous volume
 - Grass, leaves, hair

Nanite is ready to use today. The full source code is available in the UE5 early access release.

Although it doesn't yet perfectly realize every aspect of the initial virtualized geometry dream it gets pretty damn close.

But we aren't done yet.

We consciously limited what we wanted to support with the first version of Nanite. We focused on rigid geometry first as it comprises the majority of most scenes and was the most straightforward to support.

There are big chunks of the support matrix we do not cover yet.

Nanite does not support

Translucent or masked materials or

Non-rigid deformation whether that's static or animating

There are also types of geometry where our solution doesn't perform as well.

The property of scaling cost with screen resolution doesn't hold with aggregates such as grass and leaves.

Future

- Nanite everything
- Ray tracing
- Tessellation
 - Displacement
 - Higher order surfaces
- Variable rate shading
- Many view rendering
- Massive instance counts
- Foliage
- Animation
- Terrain

This is just a small glimpse of where we hope to go from here.

Ultimately we want Nanite everything, such that there is no concept for what is and is not Nanite in the engine. It will just be the way we render geometry.

In addition to that we think there many exciting places we can bring aspects of this tech and do things that are currently impractical.

Out of core ray tracing

Micropoly tessellation

Pixel scale displacement mapping

Fractal instancing

We are interested in seeing what can be achieved by applying this mindset and what we've learned to foliage, animation, and terrain.

I'm really excited about where things go from here.

Thank you!

- Nanite co-authors:
 - Rune Stubbe
 - Graham Wihlidal
- Virtual Shadow Maps:
 - Ola Olsson
 - Andrew Lauritzen
- The UE5 rendering team
- Epic artists



I'd like to thank Nanite's co-authors: Rune and Graham.
The authors of the virtual shadow maps Ola and Andrew.
The rest of the UE5 rendering team for helping support this tech
The artists who make all the pretty stuff with it.
And lastly Epic for giving me the opportunity to explore this dream of mine.

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://pcner.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/RLod.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/docs/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 Tessellation" <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://nhoppe.com/proj/gim/>
22. Niski et al. 2007, "Multi-grained Level of Detail Using a Hierarchical Seamless Texture Atlas" <https://www.cs.jhu.edu/~cohen/Publications/HSTA.pdf>
23. 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/>
24. Lee et al. 2000, "Displaced subdivision surfaces" <http://nhoppe.com/proj/dss/>
25. Lee et al. 1998, "MAPS: Multiresolution Adaptive Parameterization of Surfaces" <https://cm-bell-labs.cit.ubc.ca/who/wim/papers/sig98/>
26. Khodakovskiy et al. 2003, "Globally Smooth Parameterizations with Low Distortion" <http://multires.caltech.edu/pubs/global.pdf>
27. Maximo et al. 2013, "Adaptive multi-chart and multiresolution mesh representation" https://www.researchgate.net/publication/259096285_Adaptive_multi-chart_and_multiresolution_mesh_representation
28. Liu et al. 2017, "Seamless: Seam erasure and seam-aware decoupling of shape from mesh resolution" <https://cracl.cs.cmu.edu/seamless/>
29. 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-pba2007.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. Kumar et al. 1996, "Hierarchical BackFace Culling" <https://www.cs.unc.edu/~geom/papers/documents/technicalreports/tr96014.pdf>
38. Shopf et al. 2008, "March of the Froblins: Simulation and rendering massive crowds of intelligent and detailed creatures on GPU" https://developer.amd.com/wordpress/media/2013/01/Chapter03-SBOT-March_of_The_Froblins.pdf
39. Hill and Collin 2011, "Practical, Dynamic Visibility for Games" <https://blog.selfshadow.com/publications/practical-visibility/>
40. Haar and Aaltonen 2015, "GPU-Driven Rendering Pipelines" http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf
41. 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
42. Chajdas 2016, "AMD GeometryFX" <https://github.com/GPUOpen-Effects/GeometryFX/>

GPU work queue

43. Kerbl et al. 2018, "The Broker Queue: A Fast, Linearizable FIFO Queue for Fine-Granular Work Distribution on the GPU"

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/>
48. Stachowiak 2015, "A Deferred Material Rendering System" <https://onedrive.live.com/view.aspx?resid=EBE7DEDA70D06DA0115&app=PowerPoint&authkey=IAP-nDh4IMUJug6vs>
49. Aaltonen 2016, "Modern textureless deferred rendering techniques" <https://forum.beyond3d.com/threads/modern-textureless-deferred-rendering-techniques.57611/>
50. Doghramachi and Bucci 2017, "Deferred+: Next-Gen Culling and Rendering for Dawn Engine" <http://gpuzen.blogspot.com/>

View dependent progressive meshes

51. Ulrich 2002, "Chunked LOD" <http://tulrich.com/geekstuff/chunklod.html>
52. Yoon et al. 2004, "Quick-VDR: Interactive View-Dependent Rendering of Massive Models" <http://gamma.cs.unc.edu/OVDR/>
53. 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>
54. Cignoni et al. 2005, "Batched Multi Triangulation" <http://publications.crs4.it/pubdocs/2005/CGGMPS05a/ieeviz2005-gpumont.pdf>
55. Ponchio 2008, "Multiresolution structures for interactive visualization of very large 3D datasets" http://vca.isti.cnr.it/~ponchio/download/ponchio_phd.pdf
56. Sander and Mitchell 2005, "Progressive Buffers: View-dependent, Geometry and Texture LOD Rendering" <http://www.cse.ust.hk/~psander/docs/progbuffer.pdf>
57. 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
58. Hu et al. 2010, "Parallel View-Dependent Level-of-Detail Control" <http://www.cse.ust.hk/~psander/docs/pmstreami.pdf>
59. 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>
60. 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

Graph partitioning

61. Karypis and Kumar 1999, "A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs" <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>

Simplification

62. Garland and Heckbert 1997, "Surface Simplification Using Quadric Error Metrics" <http://www.cs.cmu.edu/~garland/Papers/quadrics.pdf>
63. Garland and Heckbert 1998, "Simplifying Surfaces with Color and Texture using Quadric Error Metrics" <https://www.cs.cmu.edu/~garland/Papers/quadric2.pdf>
64. Hoppe 1999, "New quadric metric for simplifying meshes with appearance attributes" <http://hhoppe.com/proj/newqem/>
65. Hoppe and Marschner 2000, "Efficient minimization of new quadric metric for simplifying meshes with appearance attributes" <http://hhoppe.com/proj/minqem/>

Pre-filtering

66. Heitz and Neyret 2012, "Representing Appearance and Pre-filtering Subpixel Data in Sparse Voxel Octrees" <https://hal.inria.fr/hal-00704461>
67. Loubet and Neyret 2017, "Hybrid mesh-volume LoDs for all-scale pre-filtering of complex 3D assets" <https://hal.archives-ouvertes.fr/hal-01468817>

Temporal AA

68. Karis 2014, "High-Quality Temporal Supersampling" <http://advances.realtimerendering.com/s2014/epic/TemporalAA.pptx>

Rasterization

69. Abrash 2009, "Rasterization on Larrabee" http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/abrash09_lrbrast.pdf
70. Laine and Karras 2011, "High-Performance Software Rasterization on GPUs" <https://research.nvidia.com/publication/high-performance-software-rasterization-gpus>
71. Kenzel et al. 2018, "A High-Performance Software Graphics Pipeline Architecture for the GPU" <https://markussteinberger.net/papers/cuRE.pdf>
72. Fatahallan et al. 2009, "Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur" <http://graphics.stanford.edu/papers/mprast/>
73. Brunhaver et al. 2010, "Hardware Implementation of Micropolygon Rasterization with Motion and Defocus Blur" <http://graphics.stanford.edu/papers/hwrast/>
74. Weber 2014, "Micropolygon Rendering on the GPU" <https://www.co.tuwien.ac.at/research/publications/2015/WEBER-2015-PRA1/WEBER-2015-PRA1-thesis.pdf>
75. Giesen 2013, "Triangle rasterization in practice" <https://fgiesen.wordpress.com/2013/02/08/triangle-rasterization-in-practice/>

References

Imposters

76. Brucks 2018, "Octahedral Impostors" <https://shaderbits.com/blog/octahedral-impostors>

Analytic derivatives

77. Piponi 2004, "Automatic Differentiation, C++ Templates and Photogrammetry" <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.7749&rep=rep1&type=pdf>
78. Gritz et al. 2010, "Open Shading Language" <https://dl.acm.org/doi/abs/10.1145/1837026.1837070>

Virtual shadow maps

79. Fernando et al. 2001, "Adaptive Shadow Maps" <https://www.cs.cornell.edu/~kb/publications/ASM.pdf>
80. Lefohn et al. 2007, "Resolution-matched shadow maps" <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.165.779&rep=rep1&type=pdf>
81. Olsson et al. 2014, "Efficient Virtual Shadow Maps for Many Lights" <https://efficientshading.com/2014/01/01/efficient-virtual-shadow-maps-for-many-lights/>
82. Olsson et al. 2015, "More Efficient Virtual Shadow Maps for Many Lights" <https://efficientshading.com/2015/06/01/more-efficient-virtual-shadow-maps-for-many-lights/>

Compression

83. Meyer 2012, "Real-Time Geometry Decompression on Graphics Hardware" <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.299.1553&rep=rep1&type=pdf>
84. Schlüder 2013, "Followup: Normal Mapping Without Precomputed Tangents" <http://www.thetenthplanet.de/archives/1180>
85. Mikkelsen 2020, "Surface Gradient-Based Bump Mapping Framework" <http://icgt.org/published/0009/03/04/>

