Rendering

Nanite Virtual Geometry UE5

CS 415: Game Development

Professor Eric Shaffer

Some material courtesy of Luther Tychonievich and Brian Karis

What is Nanite?

Nanite is Unreal Engine 5's virtualized geometry system which uses a new internal mesh format and rendering technology to render pixel scale detail and high object counts. It intelligently does work on only the detail that can be perceived and no more. Nanite's data format is also highly compressed, and supports fine-grained streaming with automatic level of detail.

Goal is to have artists not worry about polygon budgets…

ILLINOIS

Nanite Features

- Multiple orders of magnitude increase in geometry complexity
- Higher triangle and objects counts than has been possible before in real-time
- Frame budgets no longer constrained by polycounts and mesh memory usage
- Now possible to directly import film-quality source arts
- Use high-poly detailing rather than baking detail into normal map textures
- Level of Detail (LOD) is automatically handled
  - no longer requires manual setup for individual mesh's LODs
- Loss of quality is rare or non-existent, especially with LOD transitions

But…

NOTE

Although the advantages can be game-changing, there are practical limits that still remain. For example, instance counts, triangles per mesh, material complexity, output resolution, and performance should be carefully measured for any combination of content and hardware. Nanite will continue to expand its capabilities and improve performance in future releases of Unreal Engine.

Nanite is designed to handle a mesh that:

- Contains many triangles, or has triangles that will be very small on screen
- Has many instances in the scene
- Acts as a major occluder of other Nanite geometry
- Casts shadows using Virtual Shadow Maps

ILLINOIS

Occluded Means What?

Occluded geometry…objects (meshes, etc) that will not be visible

This can happen for several reasons

Being outside the viewing frustum. If the camera is in the center of the scene, this is commonly the case for between 75% and 95% of all geometry. Frustum clipping will mean this geometry is not rasterized nor sent to the fragment shader, but doing additional per-instance frustum culling can save significant vertex shader time, particularly in highly detailed scenes.

Being a “back face”…these are polygons that are hidden by the front side of an object (eg the back side of a sphere). These can be culled after vertex shader but before rasterization.

Being behind another object…the hidden surface removal stage takes care of this but that happens after rasterization and fragment shading. If it is possible to perform this operation more efficiently in some way…as nanite does…that’s very useful too.

Not Everything Can be Nanited

Geometry

Nanite can be enabled on Static Meshes and Geometry Collections.

A Nanite-enabled mesh can be used with the following Component types:

- Static Mesh
- Instanced Static Mesh
- Hierarchical Instanced Static Mesh
- Geometry Collection

Nanite is currently limited to rigid meshes, which represent greater than 90% of the geometry in any typical scene. Nanite supports dynamic translation, rotation, and non-uniform scaling of rigid meshes, but does not support general mesh deformation, whether it is dynamic or static. This means any position of a Nanite mesh in a way that is more complex than can be expressed in a single 4x3 matrix multiply applied to the entire mesh.

Deformation is not supported, but not limited to, the following:

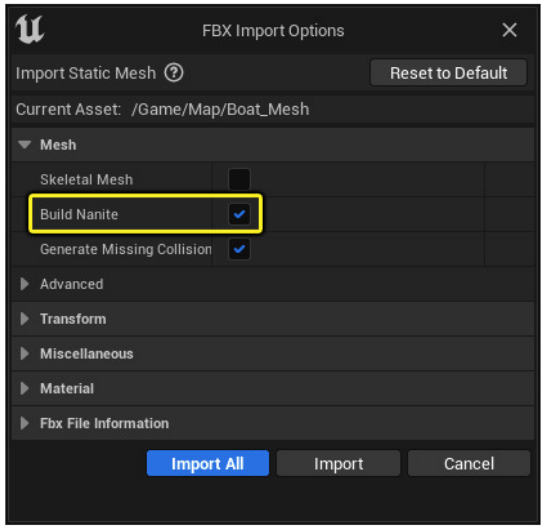
- Skeletal animation
- Morph Targets
- World Position Offset in materials
- Spline meshes

ILLINOIS

How to Use Nanite

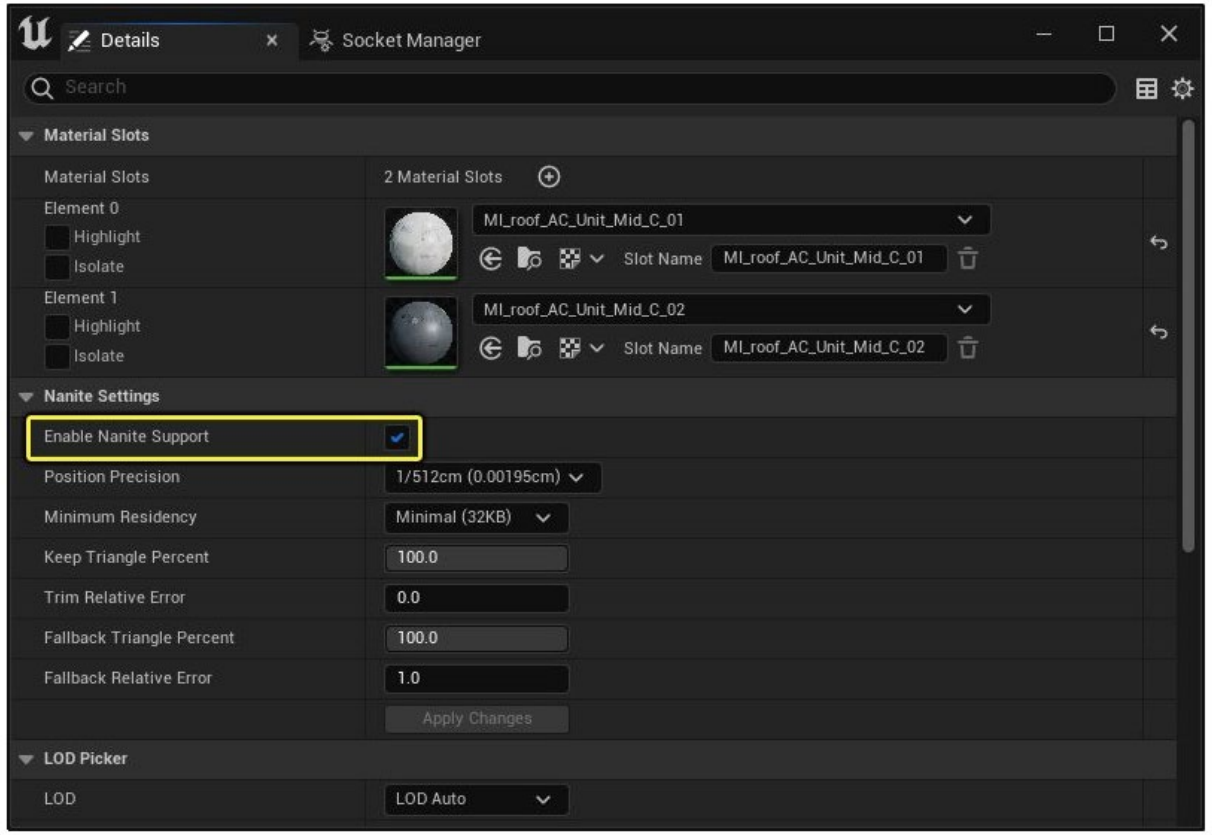
Importing a Static Mesh

When importing a mesh intended for us with Nanite, check the box for **Build Nanite**.



Or

In the Static Mesh Editor, locate the **Nanite Settings** and check the box for **Enable Nanite Support**.



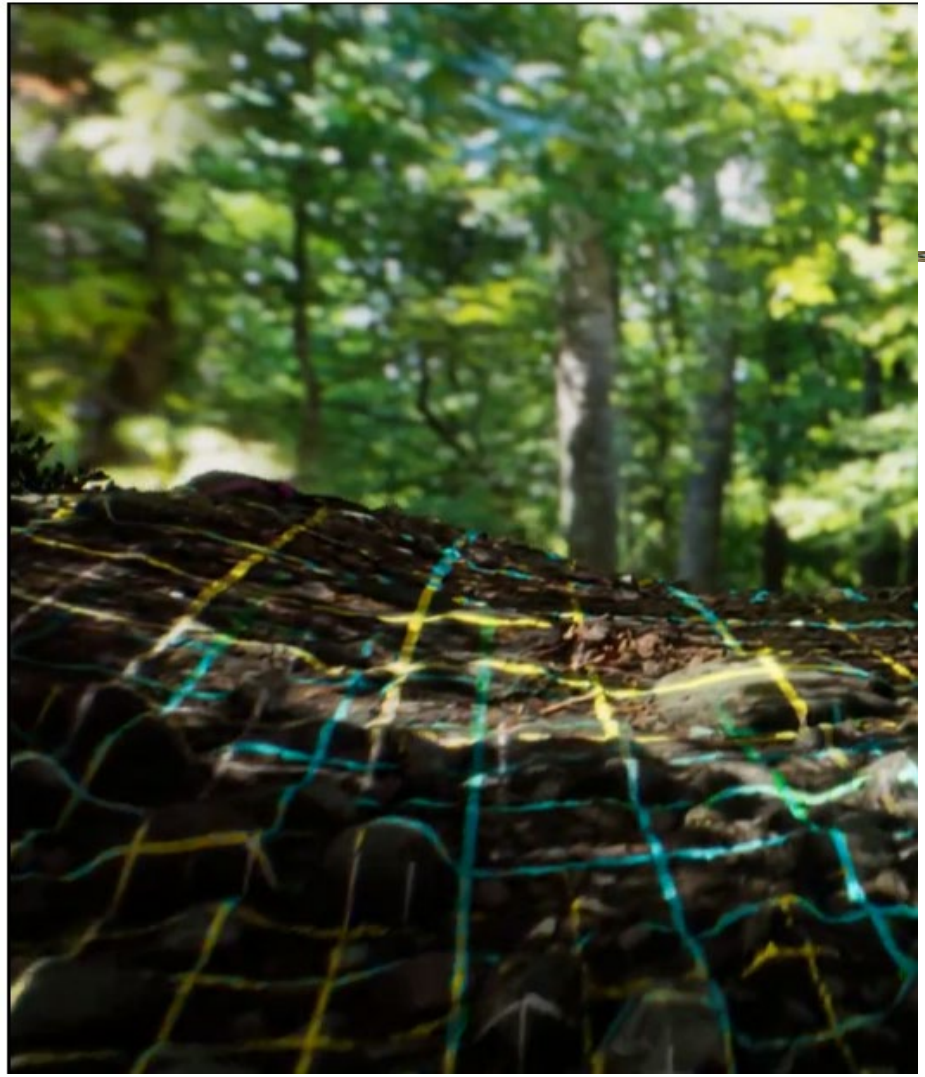
ILLINOIS

An Aside: Virtual Textures

Virtual Texturing Methods

Unreal Engine supports two virtual texturing methods: **Runtime Virtual Texturing** (RVT) and **Streaming Virtual Texturing** (SVT).

Runtime Virtual Textures	Streaming Virtual Textures
▪ Supports larger texture resolutions. ▪ Texel data cached in memory on demand. ▪ Texel data generated by the GPU at runtime. ▪ Well suited for texture data that can be rendered on demand, such as procedural textures or composited layered materials.	▪ Supports larger texture resolutions. ▪ Texel data cached in memory on demand. ▪ Texel data cooked and loaded from disk. ▪ Well suited for texture data that takes time to generate, such as lightmaps or large, detailed artist created textures.



Textures are images mapped onto geometry

Virtual textures divide image up into tiles of a fixed size

- Typically 128x128 pixels

As player moves throughout the world, UE streams in required tiles

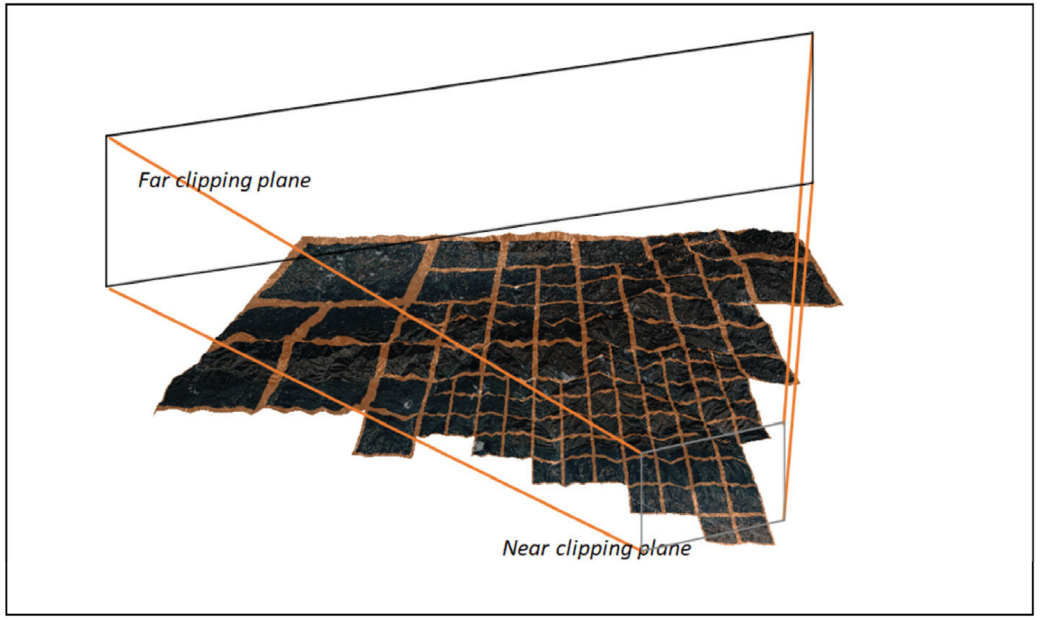
- Enables use of really large detailed textures

Sometimes, texture can be generated directly on GPU as needed

ILLINOIS

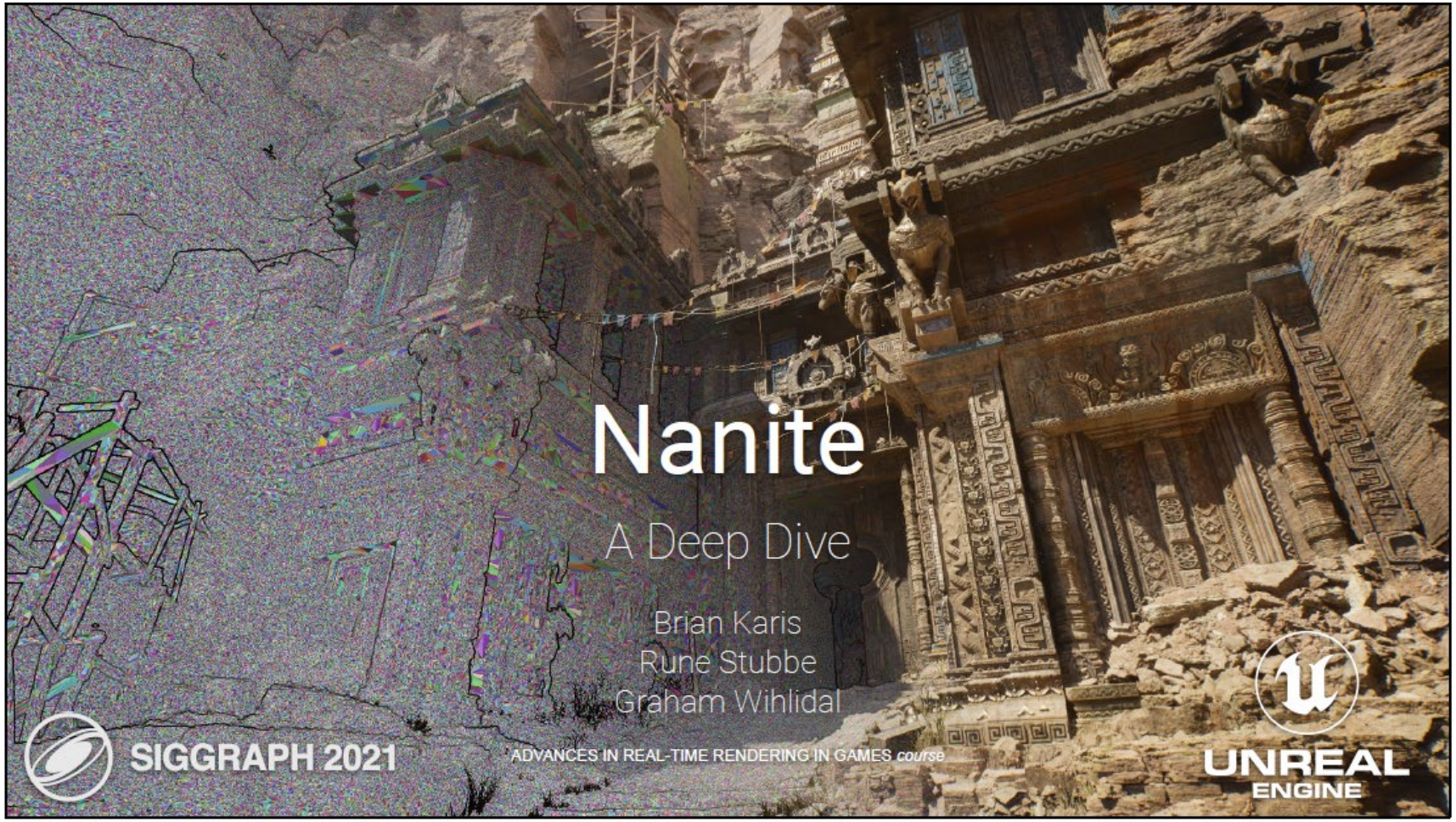
Streaming Means What

Streaming is a common term for dynamically keeping just a subset of visible geometry in memory, storing the rest of it in some larger but slower medium like disk or the cloud, and updating what's in memory as the scene changes. Streaming is primarily used to overcome memory limitations.



ILLINOIS

Nanite: A Quick Dive



ILLINOIS

What Does Virtual Geometry Mean?

- 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

==========
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 arcanely 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.

Brian Karis, Engineering Fellow at Epic Games

ILLINOIS

Rendering in Unreal Engine

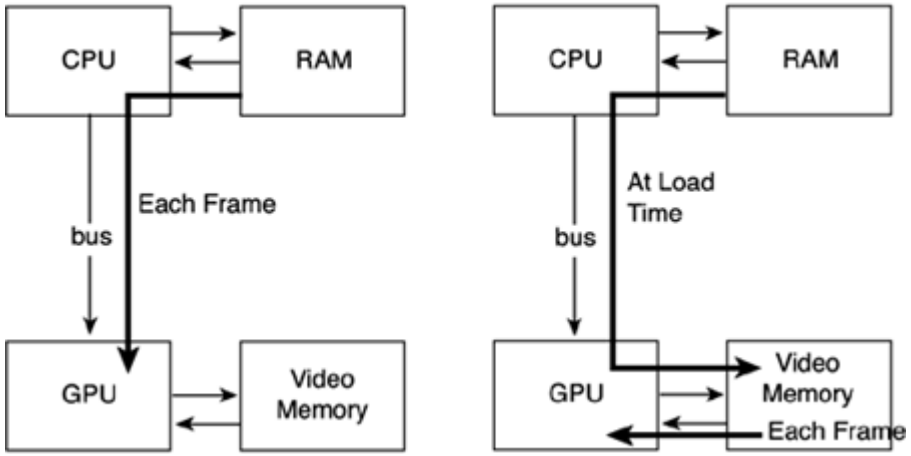
All the scene geometry is stored on the GPU

- *Retained Mode*

Persists across frames

Sparsely updated when scene changes

Most frame-to-frame changes are minor



Immediate Mode vs. Retained Mode

ILLINOIS

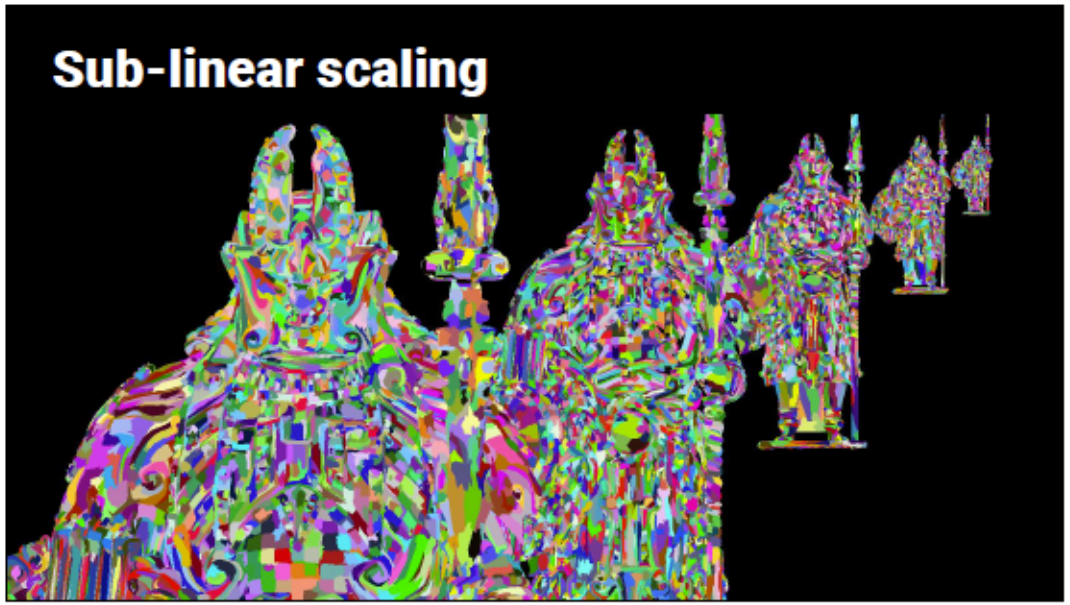
Nanite: Do Not Draw What You Do Not See

Hidden surface removal occurs late in rendering

- Lots of time wasted processing geometry with no visual impact

ILLINOIS

The Goal



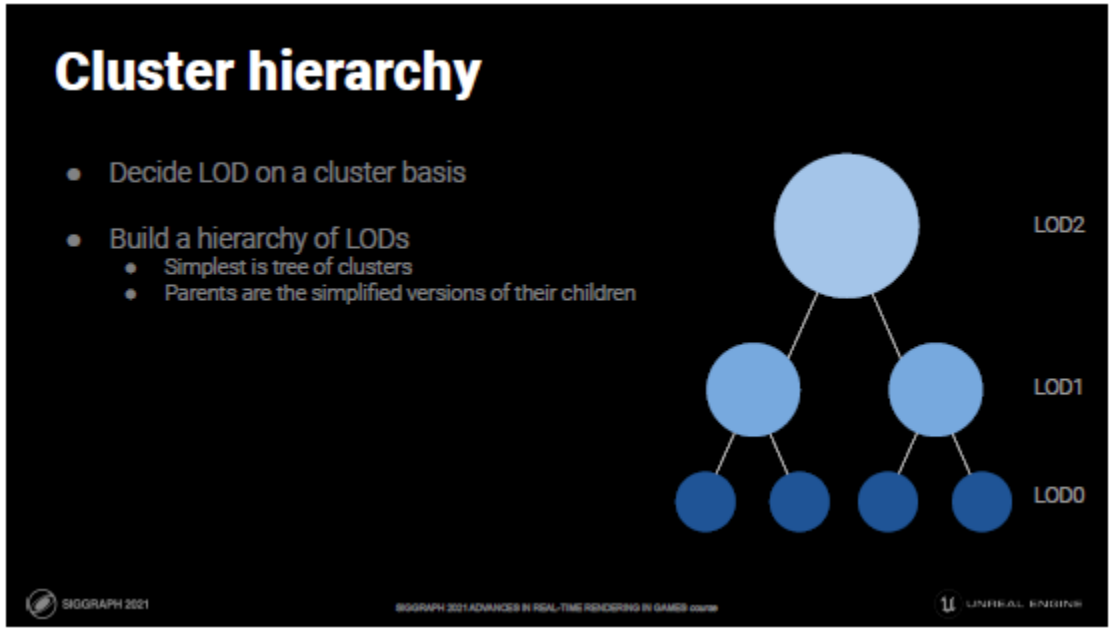
To think about it another way, there are only so many pixels on screen. Why should we draw more triangles than pixels?
In terms of clusters, we want to draw the same number of clusters every frame regardless of how many objects or how dense they are.

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.

==========
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.

ILLINOIS

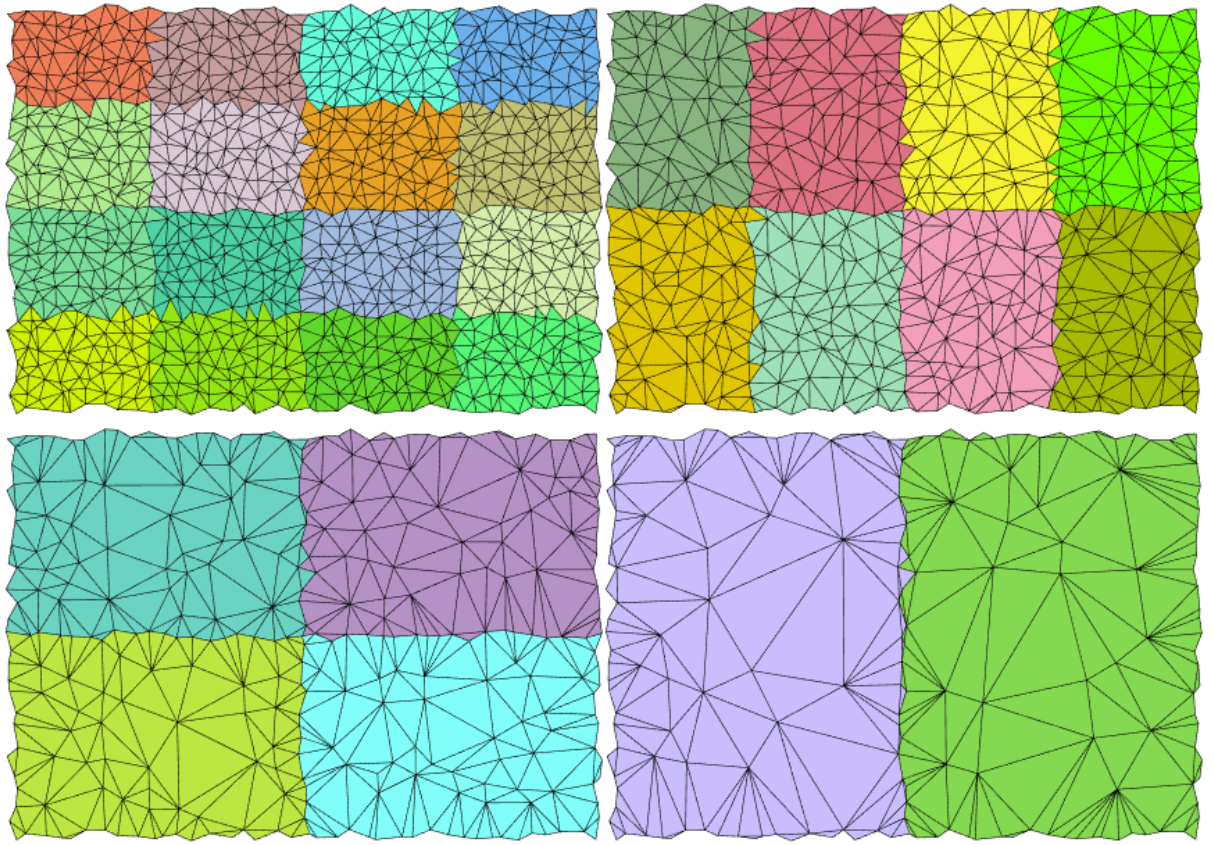
Building Triangle Cluster Hierarchy



- 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.

ILLINOIS

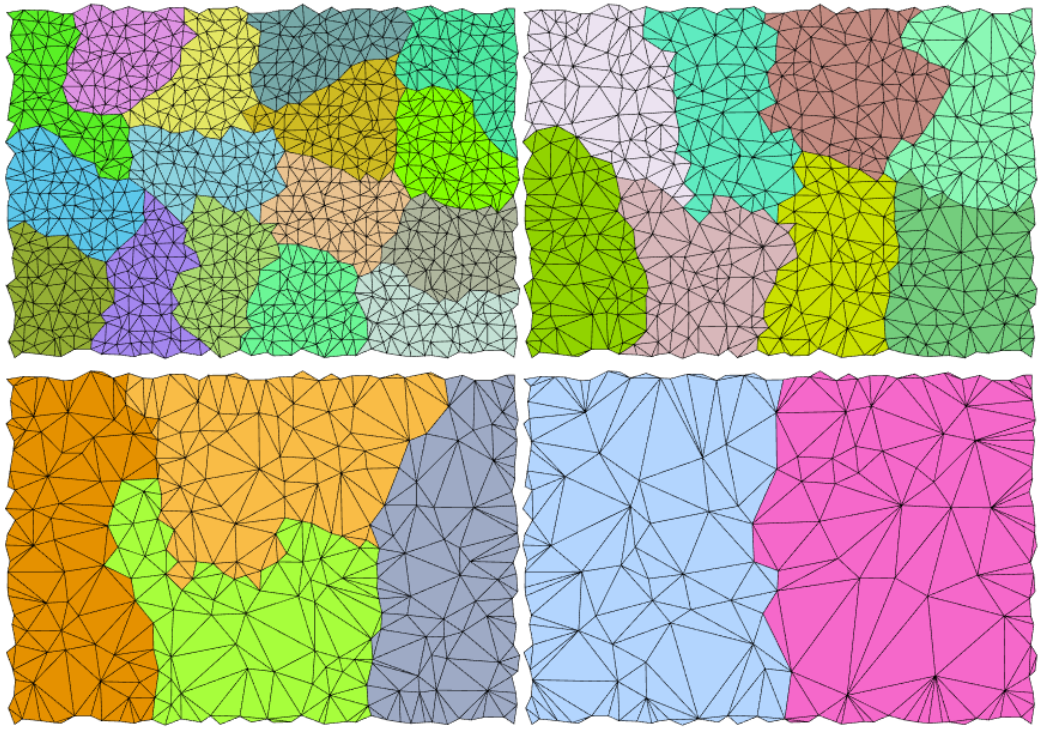
Building Triangle Cluster Hierarchy



- Naïve approach: merge 2 clusters and simplify
- What issue arises along the boundary?

ILLINOIS

Building Triangle Cluster Hierarchy

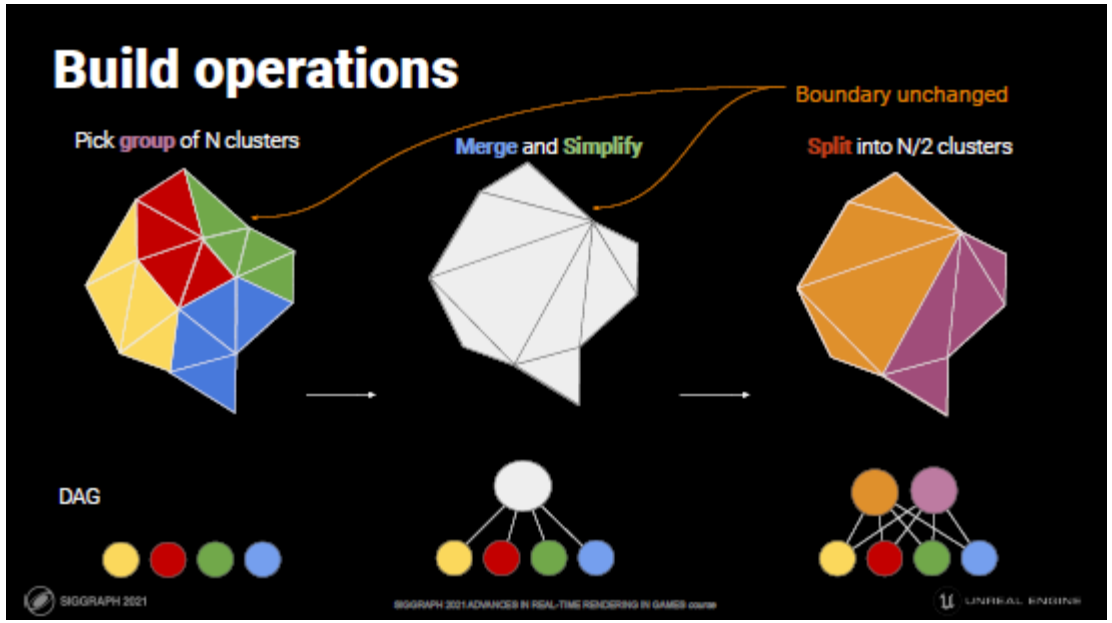


Merges four clusters and simplify them to half as many triangles.
Then splits them into two new clusters.

Note that boundaries only survive two or three steps of the algorithm.
The last image does not have dense vertices along its central boundary.

ILLINOIS

Build Operations



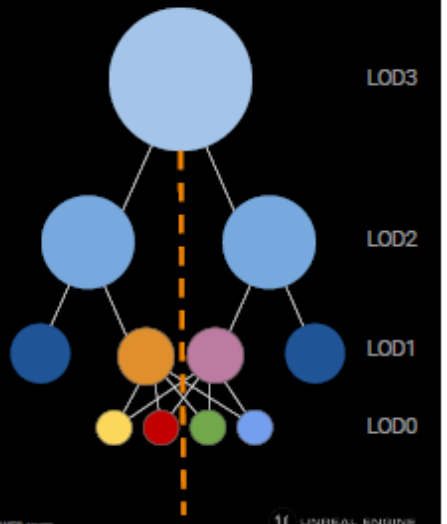
- First we need to build the leaf clusters
- In our case a cluster is 128 triangles
- Then for each LOD level
  - We group clusters to clean their shared boundary
  - Merge the triangles from the group into a shared list
  - Simplify to half the number of triangles
  - Then split the simplified list of triangles back into clusters
  - Repeat this process until there is only 1 cluster left at the root.

ILLINOIS

DAG

DAG

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

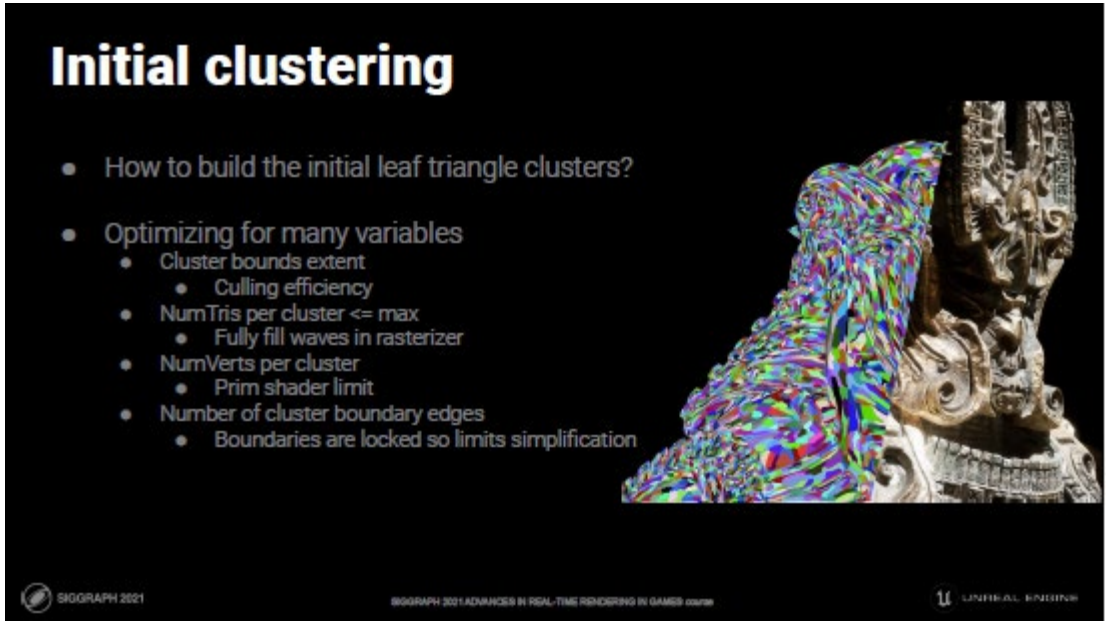


DAG: Which clusters to group?

- **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

ILLINOIS

Initial Clustering

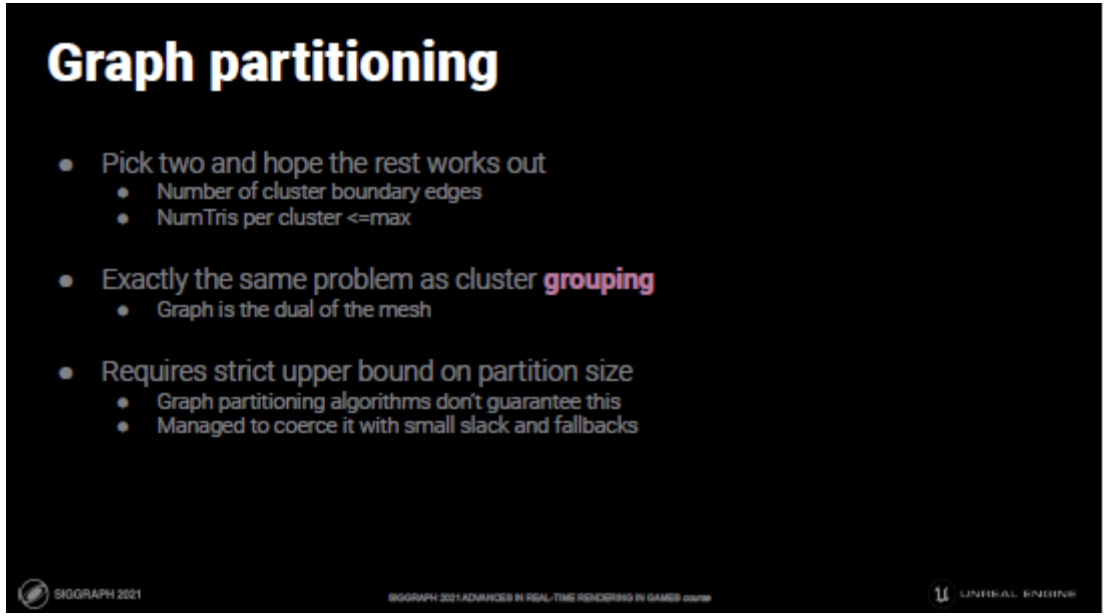


- But to start building this DAG we need the leaf clusters.

  How do we build those?
- Unfortunately this is a multi-dimensional optimization problem.
  - We'd like to minimize cluster bounds extent for culling efficiency
  - Number of triangles per cluster needs to be as close to but no greater than 128
  - 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

ILLINOIS

Initial Clustering



We can't really optimize for all dimensions simultaneously so pick 2

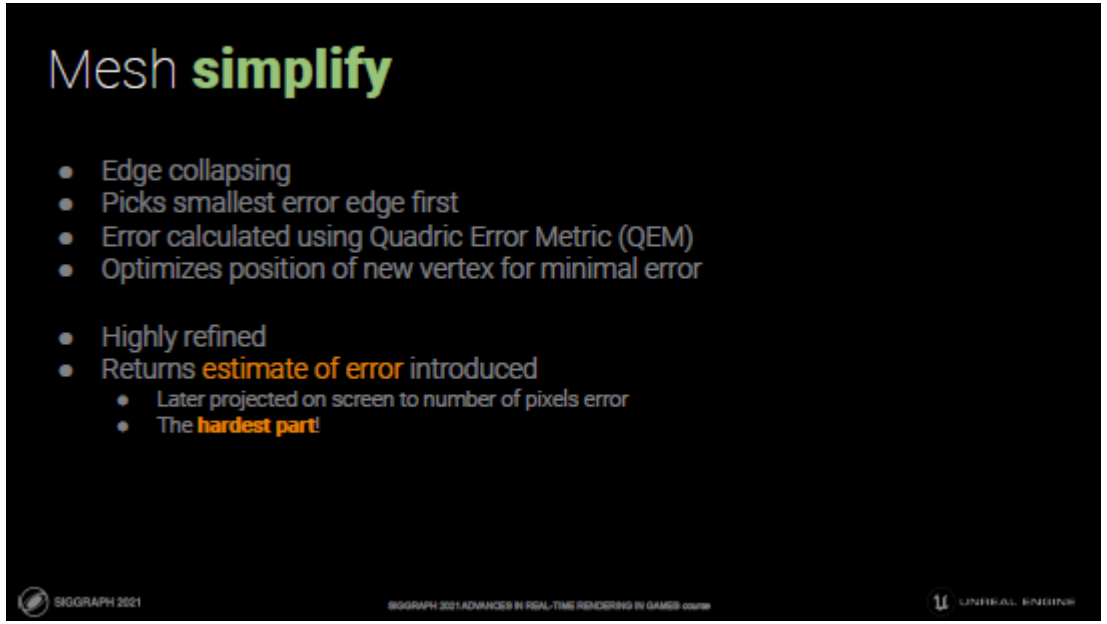
- Hope the rest works out because they are correlated.
- We optimize for the number of boundary edges and number of triangles per cluster

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

- Once again we are graph partitioning
- This time the graph is the dual of the mesh

ILLINOIS

Mesh Simplify



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.

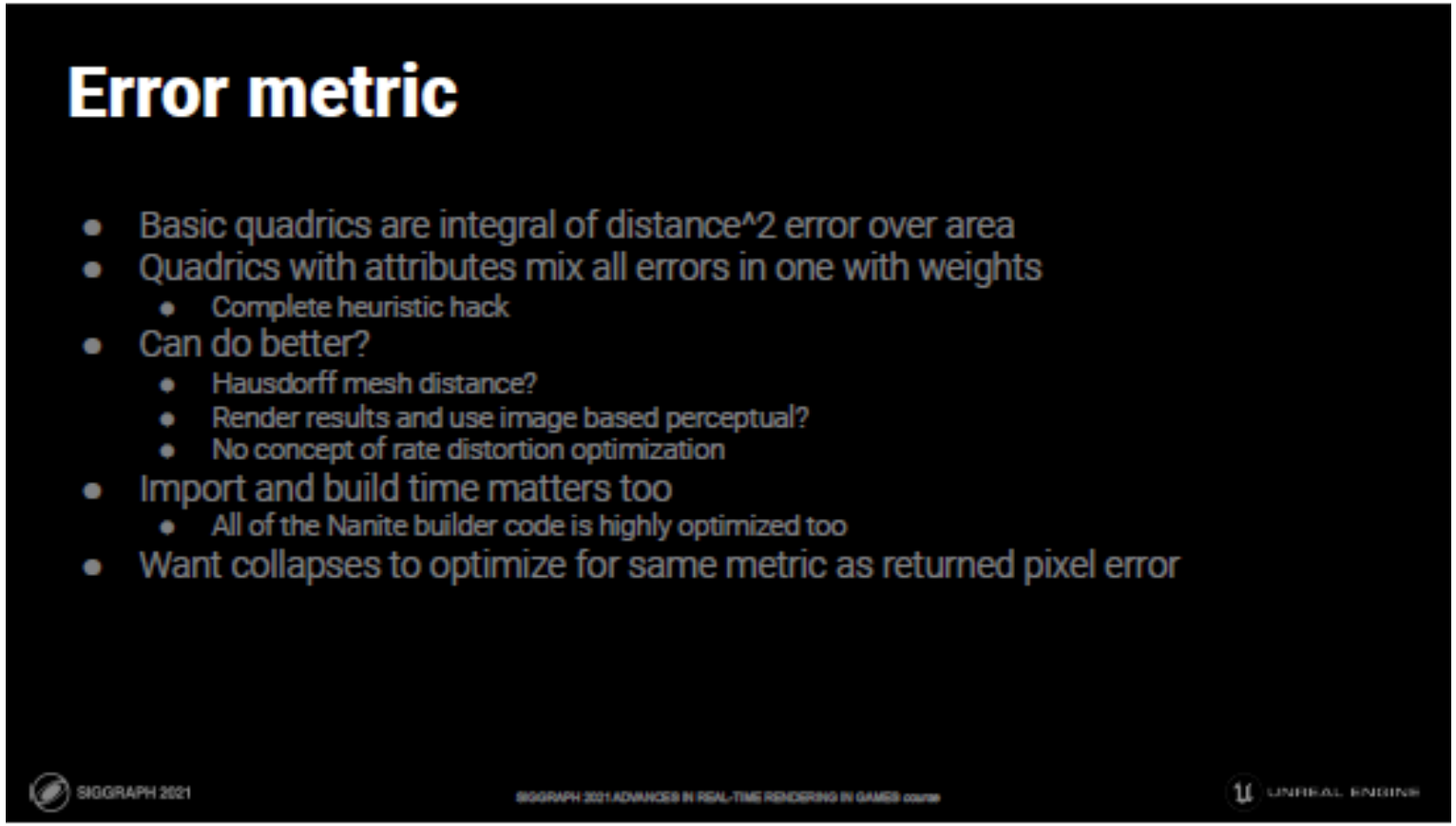
Perceptual heuristic for pixel error in a highly efficient to evaluate form
and one that can be directly optimized for
has consumed an **enormous** amount of time.
If I were to guess, probably a man year worth of effort has gone into this problem in one form or another.

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

- 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."

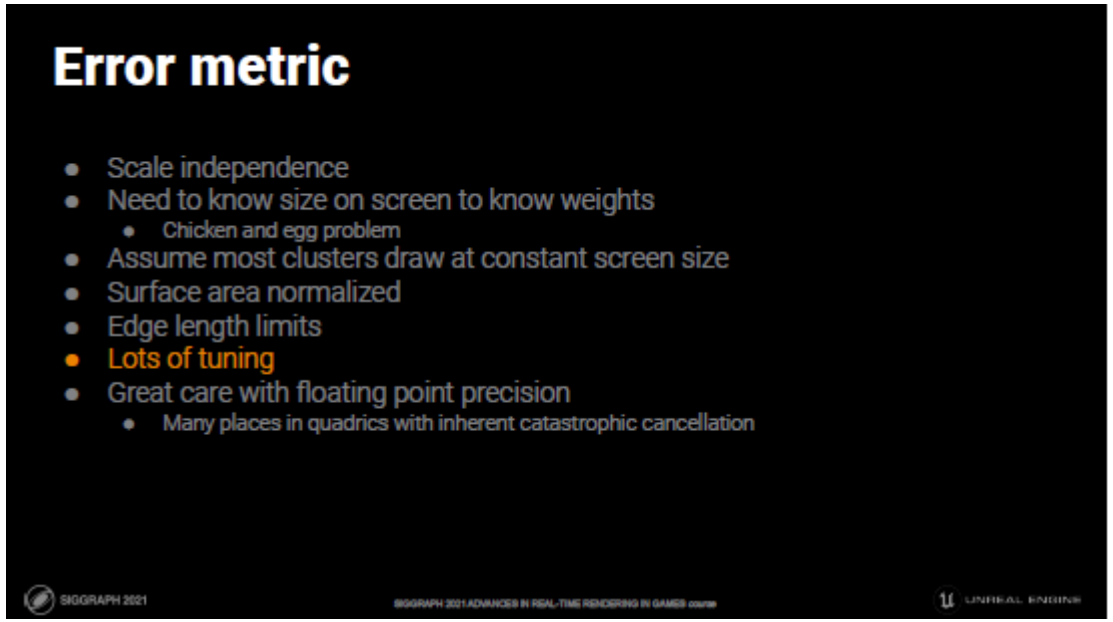
ILLINOIS

Error Metric



ILLINOIS

Error Metric



==========

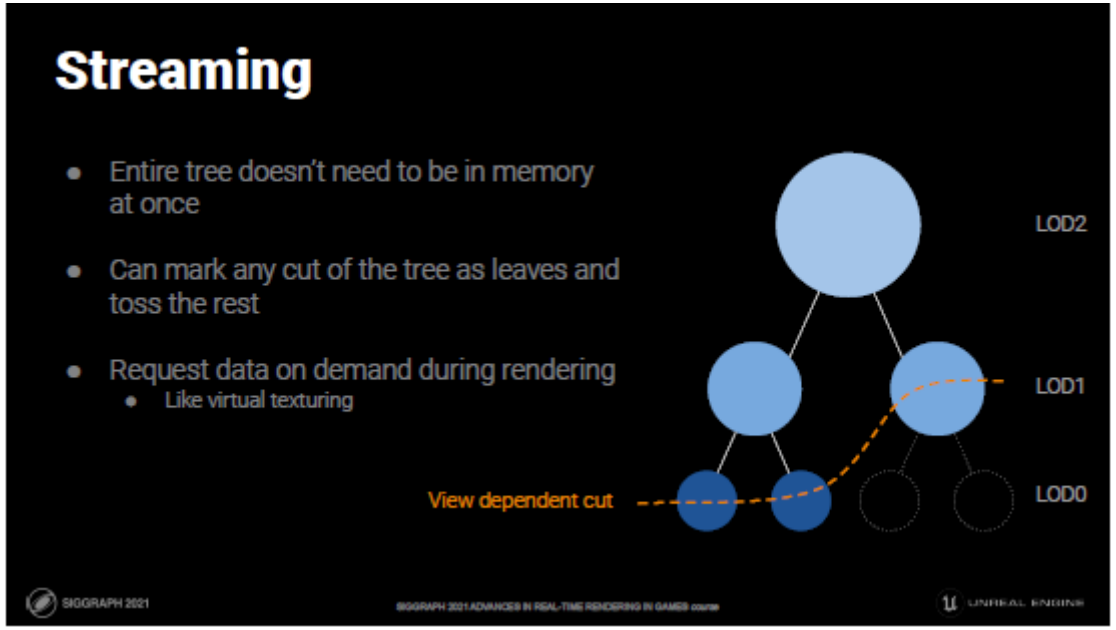
Bonus:

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.

ILLINOIS

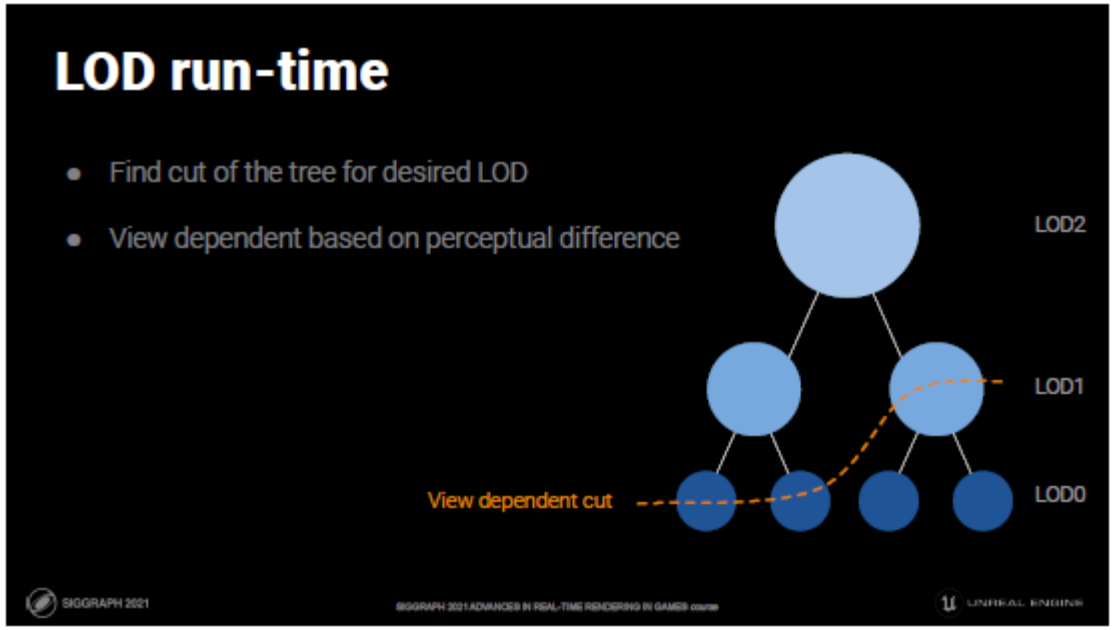
Streaming



- 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

ILLINOIS

LOD Run-Time



At run time we find a cut of the tree for the desired LOD.
Different parts of same mesh can be at different levels of detail

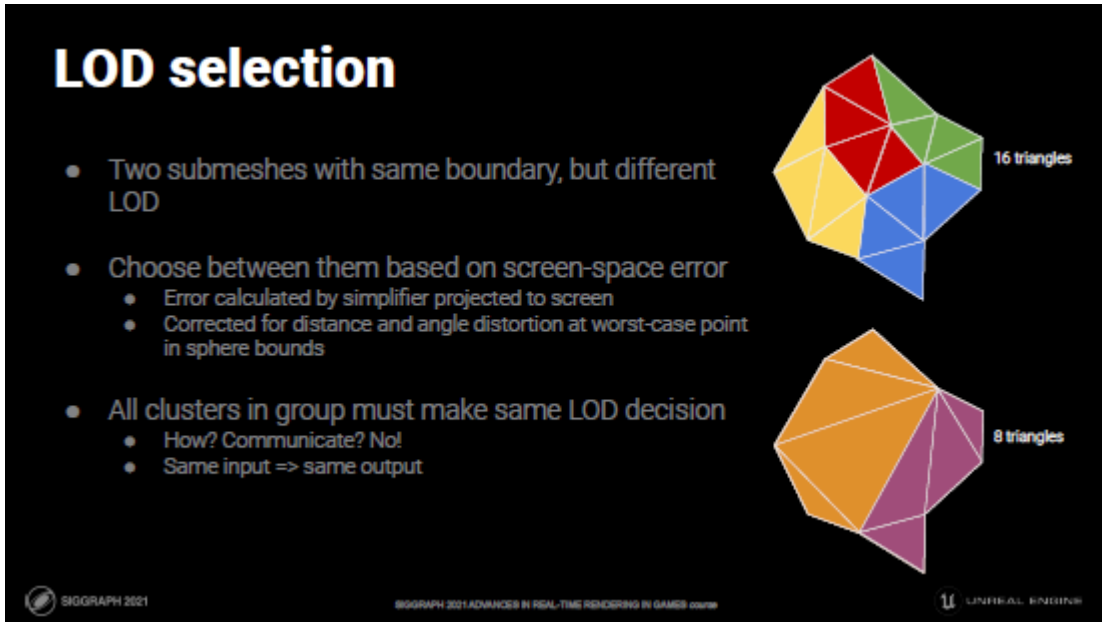
This is done in a view dependent way

- 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.

ILLINOIS

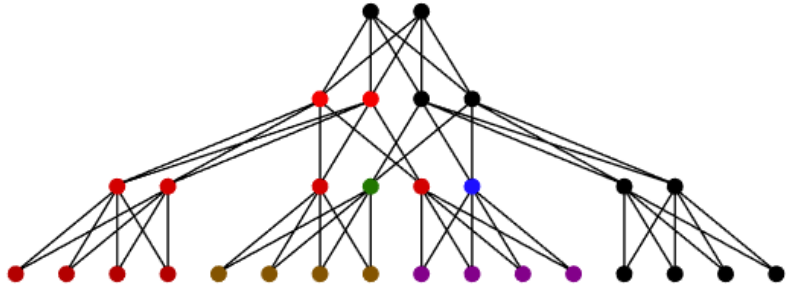
Run-time LOD Selection



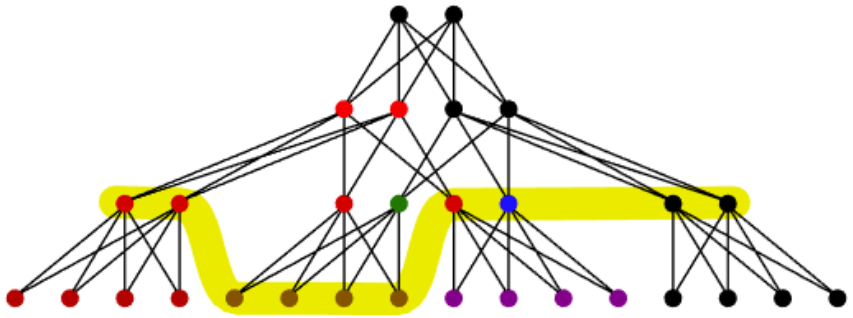
ILLINOIS

Selecting a LOD

- At least one ancestor of each leaf node is included. This ensures there is no part of the model omitted.
- If a node is selected, none of its ancestors or descendants are. This ensures only one LOD is selected for each part of the mesh.
- The selected nodes can be connected by a cut that does not cross any edges. This ensures cluster boundaries do not have cracks.



Group-and-split results in a DAG: each group of clusters shares four child clusters but groups of children can span multiple parent



A valid cut of a DAG, satisfying the three criteria above.

ILLINOIS

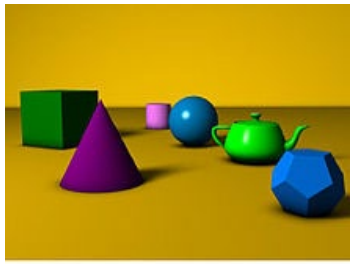
Occlusion Culling

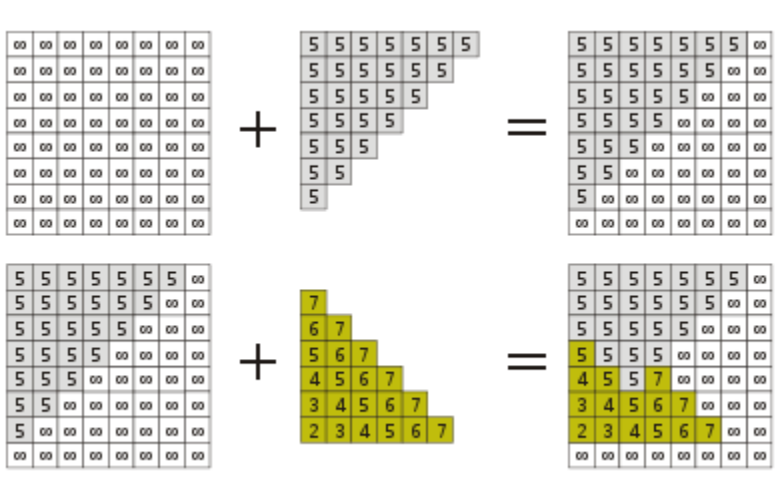
Two pass occlusion culling

- 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

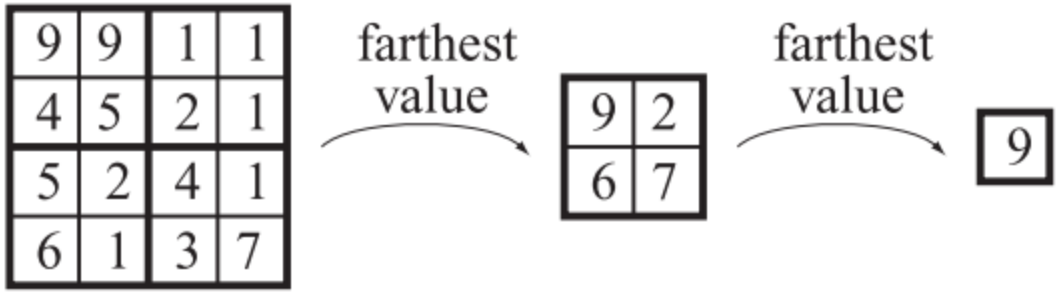
How does a hierarchical z-buffer work? Let's just look at a standard z-buffer…

…HZB uses LOD again to allow quick rejection of large blocks of pixels as being behind existing geometry





Hierarchical Z-Buffer



On the left, a 4 × 4 piece of the z-buffer is shown.
The numerical values are the depths of each pixel.

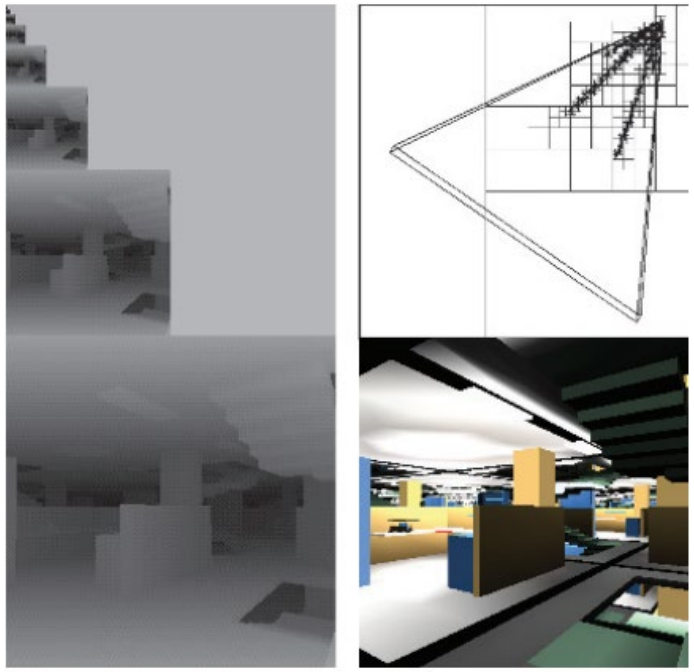
This is downsampled to a 2 × 2 region
Each value is the farthest (largest) of the four 2×2 regions on the left.

Finally, the farthest value of the remaining four z-values is computed.

These three maps compose an image pyramid that is called the hierarchical z-buffer.

ILLINOIS

Hierarchical Z-Buffer



A scene with high depth complexity (lower right)
The corresponding z-pyramid (on the left),
An octree subdivision of the scene (upper right).

By traversing the octree from front to back and culling occluded octree nodes as they are encountered, this algorithm visits only visible octree nodes and their children (the nodes portrayed at the upper right) and renders only the triangles in visible boxes.

Akenine-Moller, Tomas; Haines, Eric; Hoffman, Naty.
Real-Time Rendering, Fourth Edition

In general, most occlusion culling algorithms based on HZB work like this:

1. Generate a full hierarchical z-pyramid
2. To test if an object is occluded, project its bounding volume to screen space and estimate the level in the z-pyramid.
3. Test occlusion against the selected level. Optionally continue testing using a finer level if results are ambiguous.

ILLINOIS