Congratulations! Our Future Game Engine Talents

- **300+** students submitted their homework
- **40** students are qualified for graduation certification
- **10** of them scored full points in all homework

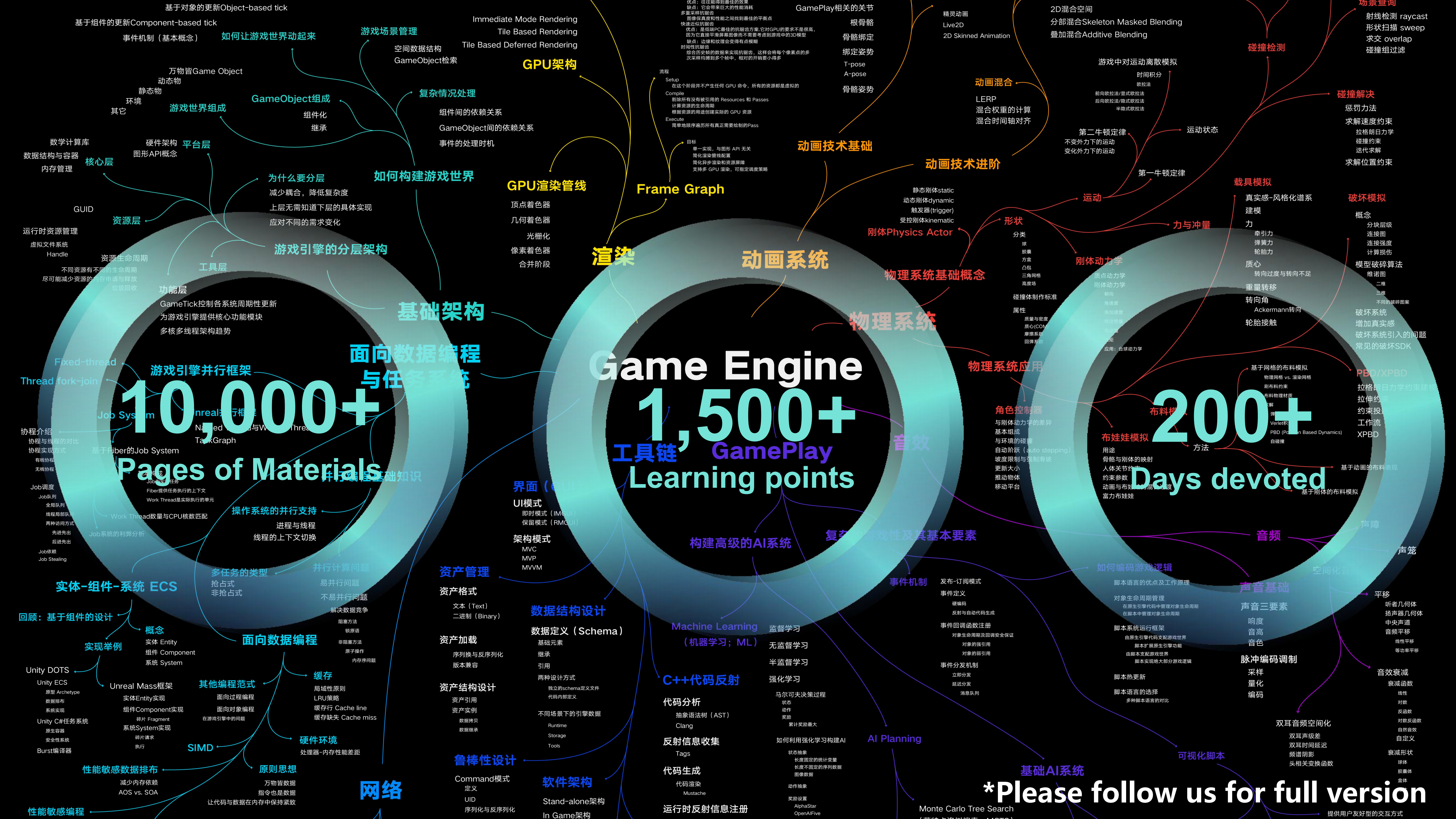
林宇杰	柳　航	贺文宁	蔡经浩	李方钏	管康诚
李进文	周一帆	梁嘉辉	涂 强	马晟杰	於怿丰
肖 翱	王鑫	彭盛林	庄鸣真	朱江涛	周宇东
冉高成	路程	陆俊壕	吴永力	鲁瀚洋	孙蒙蒙
张皓	朱威远	郑宇光	龙虎	胡浩宇	席圣渠
郇迪	刘浩天	胡树彦	翁开宇	方超伟	侯瑞涛
丁雨航	杨汶昊	祁志铭	林辉	潘安	

Set sail for the endless ocean of knowledge

We will continue to share everything we know about game engine

Follow us on WeChat
Reply【入群】to join our community

- Text version of lectures will be released soon
- Videos & Streaming are coming



10,000+
Pages of Materials
Game Engine
1,500+
Learning points
200+
Days devoted
*Please follow us for full version
基础架构
面向数据编程与任务系统
渲染
动画系统
物理系统
工具链
GamePlay
音效
如何让游戏世界动起来
基于对象的更新Object-based tick
基于组件的更新Component-based tick
事件机制（基本概念）
游戏场景管理
空间数据结构
GameObject检索
Immediate Mode Rendering
Tile Based Rendering
Tile Based Deferred Rendering
GPU架构
万物皆Game Object
动态物
静态物
环境
其它
游戏世界组成
GameObject组成
组件化
继承
复杂情况处理
组件间的依赖关系
GameObject间的依赖关系
事件的处理时机
数学计算库
数据结构与容器
内存管理
核心层
硬件架构
图形API概念
平台层
为什么要分层
减少耦合，降低复杂度
上层无需知道下层的具体实现
应对不同的需求变化
如何构建游戏世界
GUID
资源层
运行时资源管理
虚拟文件系统
Handle
资源生命周期
工具层
游戏引擎的分层架构
功能层
GameTick控制各系统周期性更新
为游戏引擎提供核心功能模块
多核多线程架构趋势
GPU渲染管线
顶点着色器
几何着色器
光栅化
像素着色器
合并阶段
Frame Graph
动画技术基础
根骨骼
骨骼绑定
绑定姿势
T-pose
A-pose
骨骼姿势
精灵动画
Live2D
2D Skinned Animation
2D混合空间
分部混合Skeleton Masked Blending
叠加混合Additive Blending
动画混合
LERP
混合权重的计算
混合时间轴对齐
动画技术进阶
静态刚体static
动态刚体dynamic
触发器(trigger)
受控刚体kinematic
刚体Physics Actor
形状
物理系统基础概念
物理系统应用
游戏中对运动离散模拟
时间积分
第二牛顿定律
第一牛顿定律
运动状态
运动
力与冲量
刚体动力学
碰撞检测
场景查询
射线检测 raycast
形状扫描 sweep
求交 overlap
碰撞组过滤
碰撞解决
惩罚力法
求解速度约束
求解位置约束
载具模拟
真实感-风格化谱系
建模
力
质心
重量转移
转向角
轮胎接触
破坏模拟
概念
模型破碎算法
破坏系统
增加真实感
破坏系统引入的问题
常见的破坏SDK
角色控制器
布娃娃模拟
布料模拟
PBD/XPBD
游戏引擎并行框架
Fixed-thread
Thread fork-join
Job System
Unreal并行框架
TaskGraph
基于Fiber的Job System
协程介绍
并行编程基础知识
Job调度
操作系统的并行支持
进程与线程
线程的上下文切换
Work Thread数量与CPU核数匹配
Job系统的利弊分析
多任务的类型
抢占式
非抢占式
并行计算问题
易并行问题
不易并行问题
实体-组件-系统 ECS
回顾：基于组件的设计
概念
实体 Entity
组件 Component
系统 System
实现举例
Unity DOTS
Unreal Mass框架
面向数据编程
其他编程范式
面向过程编程
面向对象编程
缓存
局域性原则
LRU策略
缓存行 Cache line
缓存缺失 Cache miss
硬件环境
处理器-内存性能差距
SIMD
性能敏感数据排布
减少内存依赖
AOS vs. SOA
原则思想
万物皆数据
指令也是数据
让代码与数据在内存中保持紧凑
性能敏感编程
界面（GUI）
UI模式
即时模式（IMGUI）
保留模式（RMGUI）
架构模式
MVC
MVP
MVVM
资产管理
资产格式
文本（Text）
二进制（Binary）
资产加载
序列换与反序列化
版本兼容
资产结构设计
资产引用
资产实例
数据结构设计
数据定义（Schema）
基础元素
继承
引用
两种设计方式
不同场景下的引擎数据
Runtime
Storage
Tools
鲁棒性设计
Command模式
定义
UID
序列化与反序列化
软件架构
Stand-alone架构
In Game架构
网络
构建高级的AI系统
Machine Learning
（机器学习；ML）
监督学习
无监督学习
半监督学习
强化学习
马尔可夫决策过程
如何利用强化学习构建AI
C++代码反射
代码分析
抽象语法树（AST）
Clang
反射信息收集
Tags
代码生成
代码渲染
Mustache
运行时反射信息注册
复杂游戏性及其基本要素
事件机制
发布-订阅模式
事件定义
事件回调函数注册
事件分发机制
AI Planning
Monte Carlo Tree Search
基础AI系统
如何编码游戏逻辑
脚本语言的优点及工作原理
对象生命周期管理
脚本系统运行框架
脚本热更新
脚本语言的选择
可视化脚本
音频
声音基础
声音三要素
响度
音高
音色
脉冲编码调制
采样
量化
编码
双耳音频空间化
双耳声级差
双耳时间延迟
频谱阴影
头相关变换函数
空间化音
声障
声笼
平移
听者几何体
扬声器几何体
中央声道
音频平移
音效衰减
衰减函数
自定义
衰减形状
提供用户友好型的交互方式

Q&A

- Q1: How does ECS handle destruction of entities?
- Q2: How can we measure Cache miss?
- Q3: How should we provide tools for designers to design functions under DOP architecture?
- Q4: How does Lumen handle emissive materials?

Lecture 22

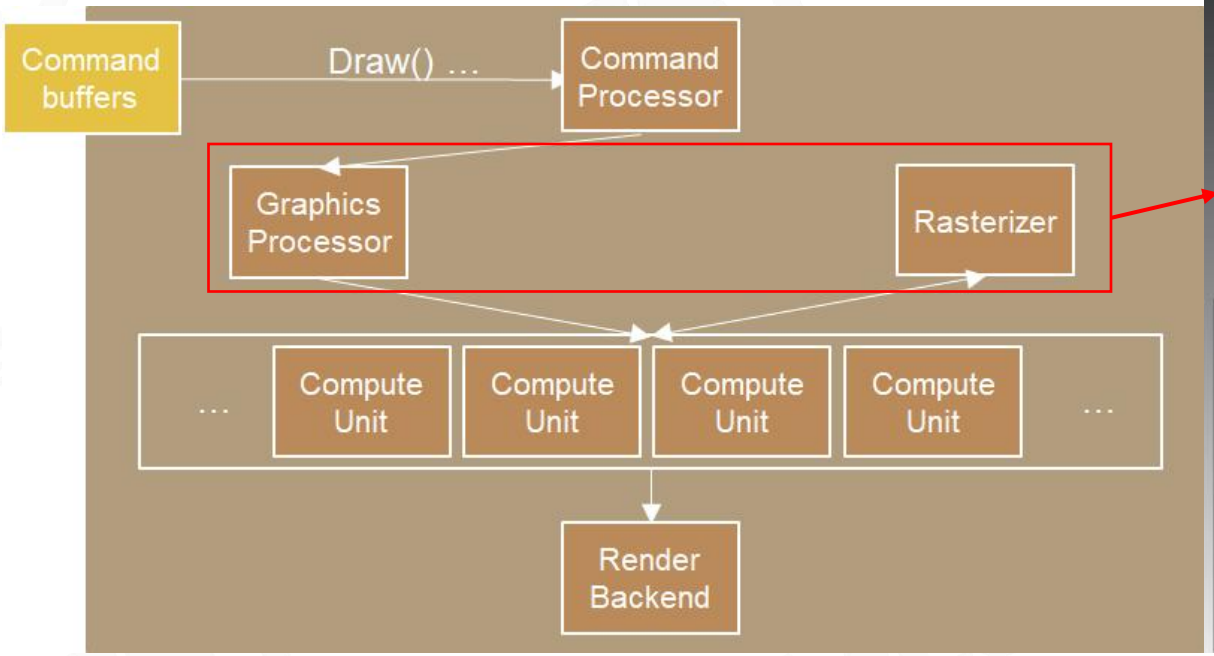
GPU-Driven Geometry Pipeline and Nanite

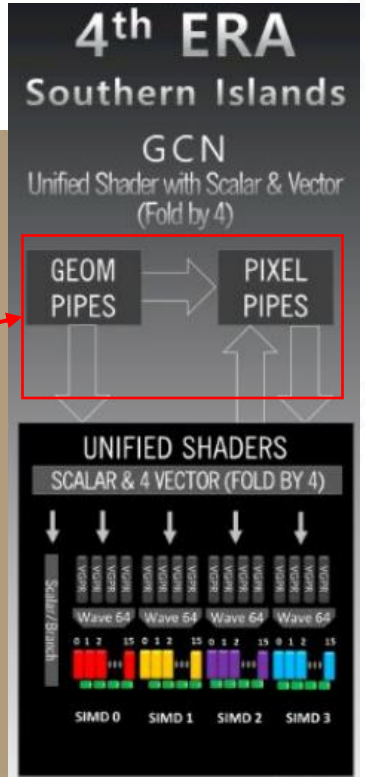
Advanced Topics

WANG XI

“Long” Pipeline of Traditional Rendering

- Compute unit works with graphics processor and rasterizer
- It's a series of data processing units arranged in a chain like manner
- Difficult to fully fill the GPU

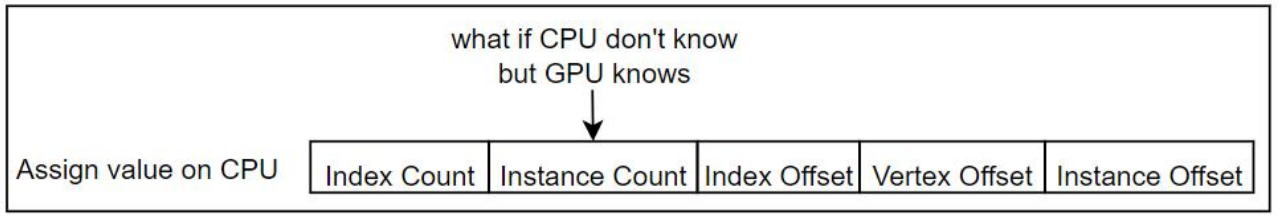




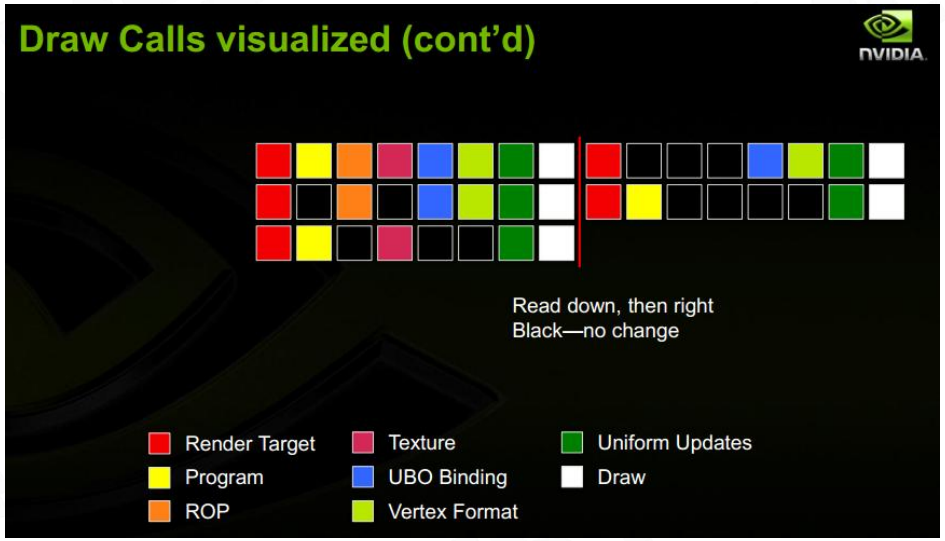
Jungle of Direct Draw Graphics API

Explosion of DrawCalls:

- Meshes x RenderStates x LoDs x Materials x Animations



Problem 1:A traditional DrawIndexedInstanced command requires 5 arguments assigned on CPU

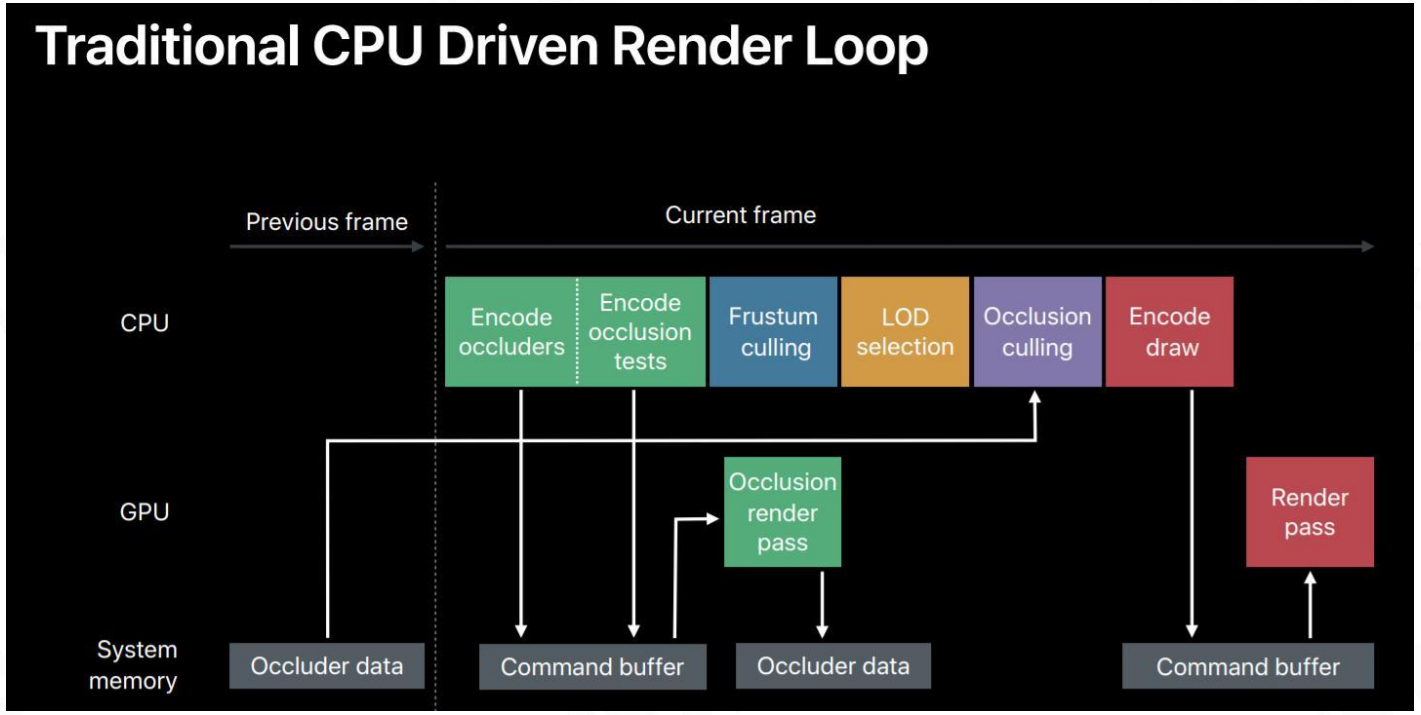


Problem 2: Driver state switching overhead between amount of draw commands

Bottleneck of Traditional Rendering Pipeline

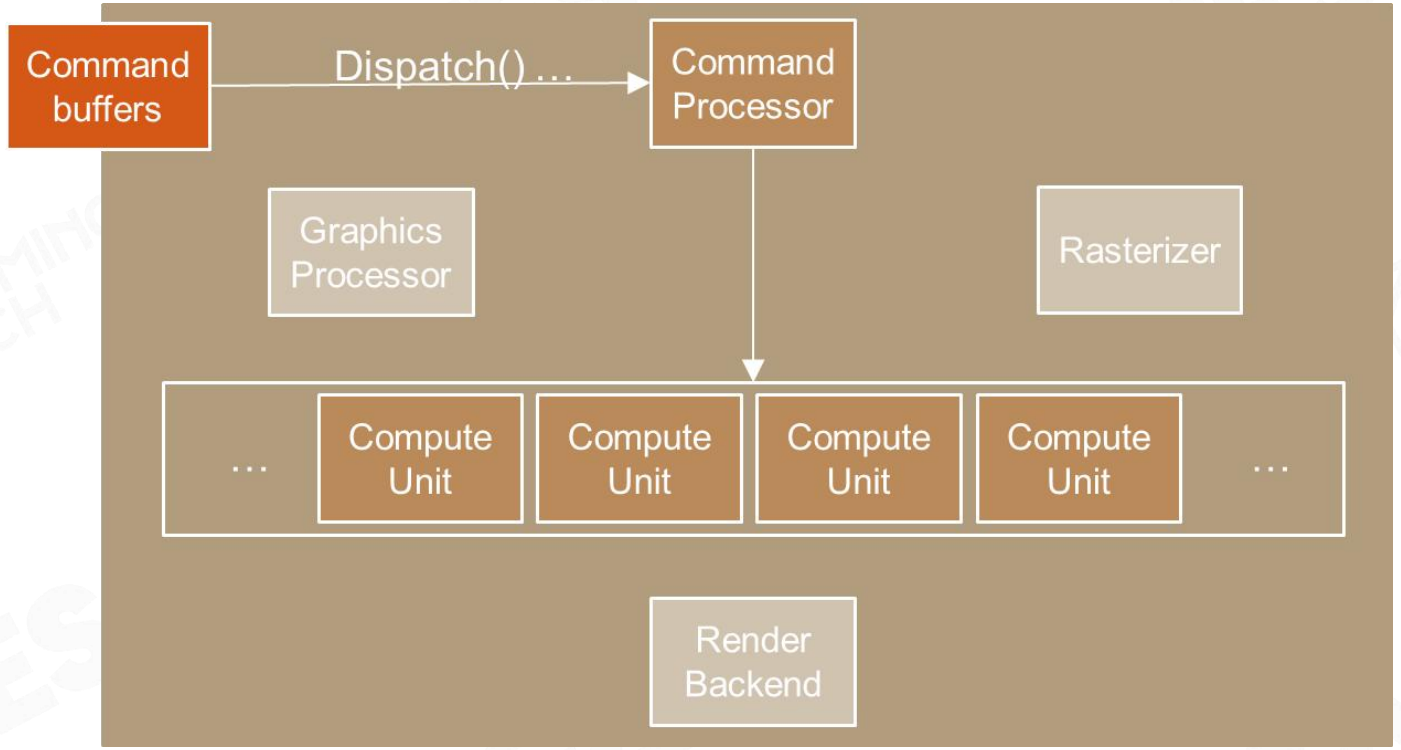
When rendering complicated scene with dense geometries and many materials

- High CPU overload
  - Frustum/Occlusion Culling
  - Prepare drawcall
- GPU idle time
  - CPU can not follow up GPU
- High driver overhead
  - GPU state exchange overhead when solving large amount of drawcalls



Compute Shader - General Computation on GPU

- High-speed general purpose computing and takes advantage of the large numbers of parallel processors on the GPU
- Less overhead to graphics pipeline
- Just one stage in pipeline



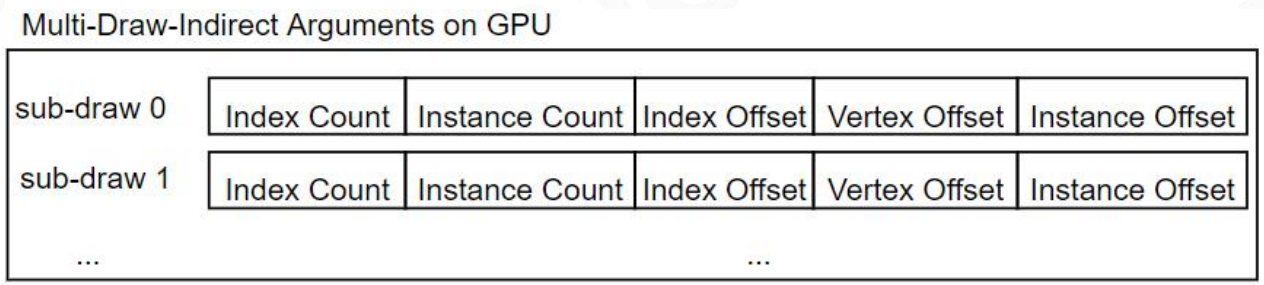
Draw-Indirect Graphics API

Advantage:

- Allow you to specify parameters to draw commands from a GPU buffer, or via GPU compute program
- "Draw-Indirect" command can merge a lot of draw calls into one single draw call, even with different mesh topology

Notice:

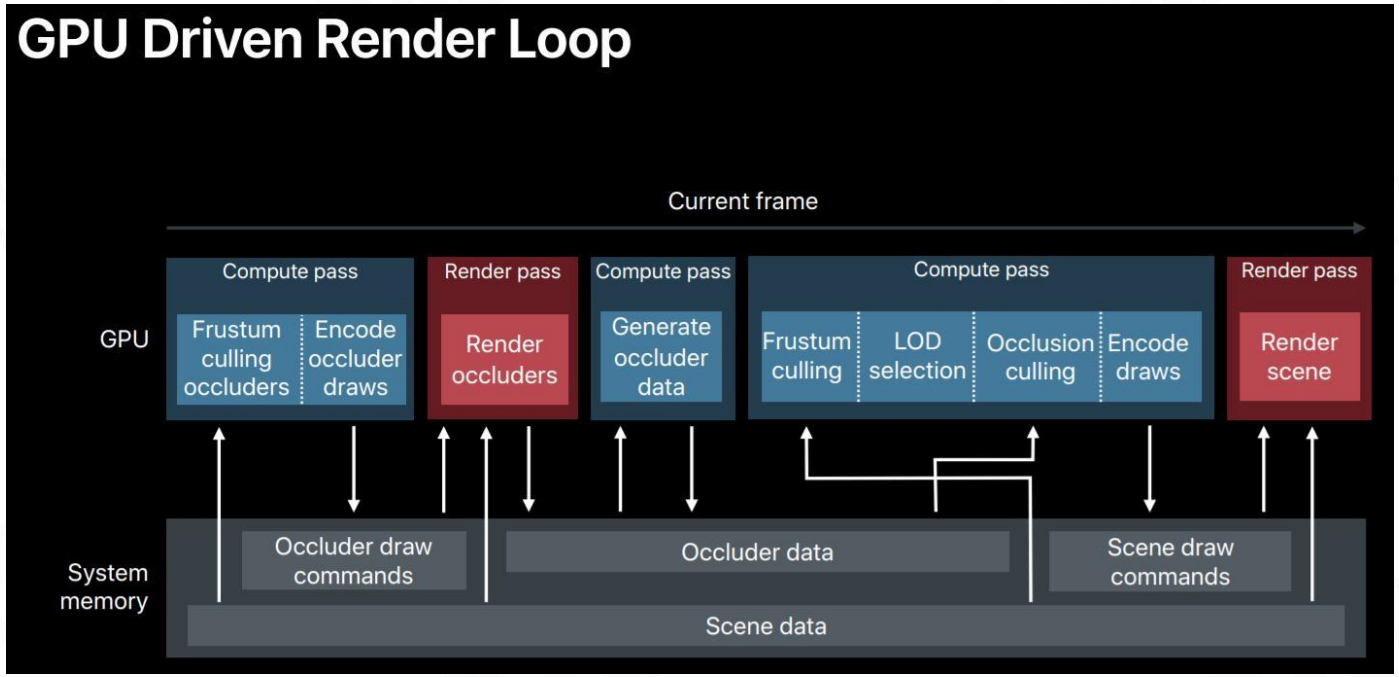
- The actual name of "Draw-Indirect" is different in each graphics platform, but they act as the same role. (e.g. vkCmdDrawIndexedIndirect(Vulkan), ExecuteIndirect(D3D12), ...)



GPU Driven Render Pipeline – DrawPrimitive vs. DrawScene

- GPU controls what objects are actually rendered
  - Lod selection, visibility culling on GPU
- No CPU/GPU roundtrip
  - CPU do not touch any GPU data
- N viewports/frustums

Frees up the CPU to be used on other things, ie. AI



GPU Driven Pipeline in Assassins Creed

GPU Driven Pipeline in Assassins Creed

Motivation

- Massive amounts of geometry: architecture, seamless interiors, crowds

Use mesh cluster rendering to

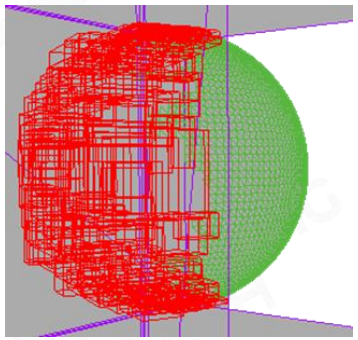
- Allow much more aggressive batching and culling granularity
- Render different meshes efficiently with a single indirect draw command

Mesh Cluster Rendering

- **Require**
  - Fixed cluster topology (E.g. 64 triangles in Assassin Creed or 128 triangles in Nanite)
  - Split & rearrange all meshes to fit fixed topology(insert degenerate triangles)
  - Fetch vertices manually in VS

- **Key Implementation**
  - Cull clusters by their bounding on GPU (usually by compute shader)
  - GPU outputs culled cluster list & drawcall args
  - Draw arbitrary number of visible clusters in single drawcall

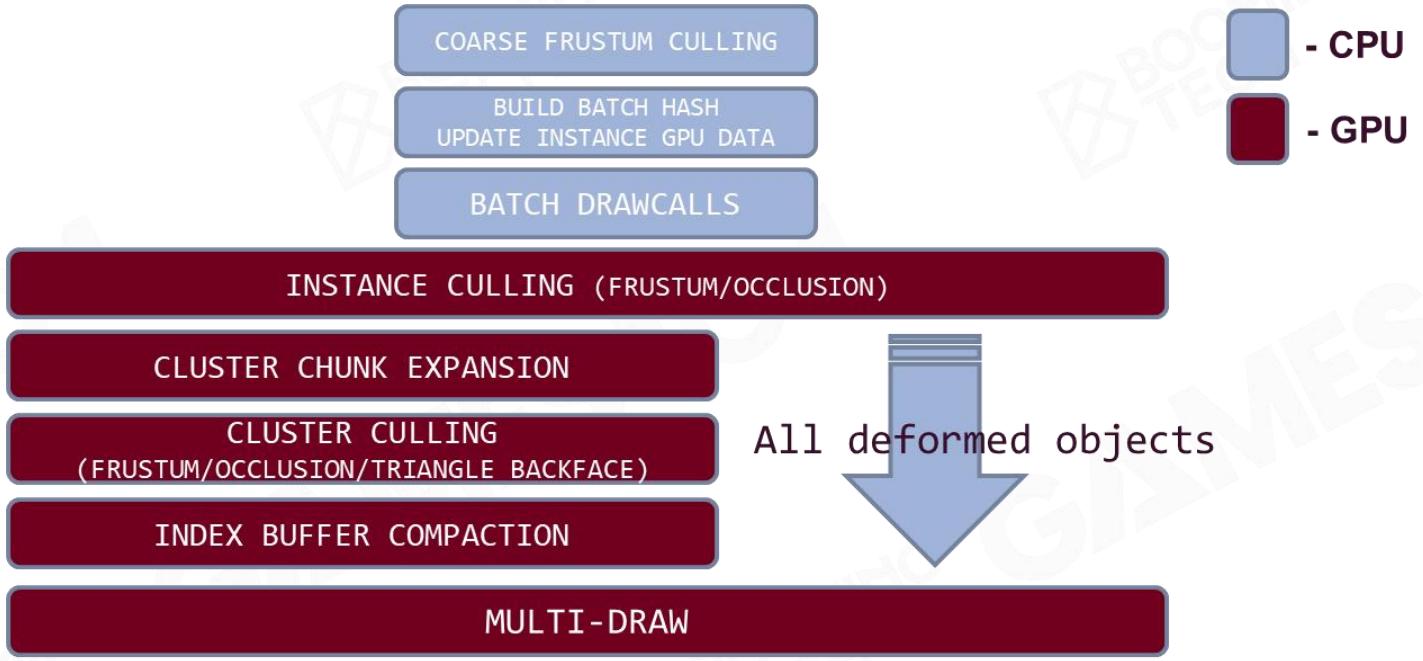
Clusters (64 triangles)



Culled by cluster bouding box

GPU Driven Pipeline in Assassins Creed

- **Overview**
  - Offload more work from CPU to GPU
  - But not perfectly "draw scene" command, can only draw objects with the same material together

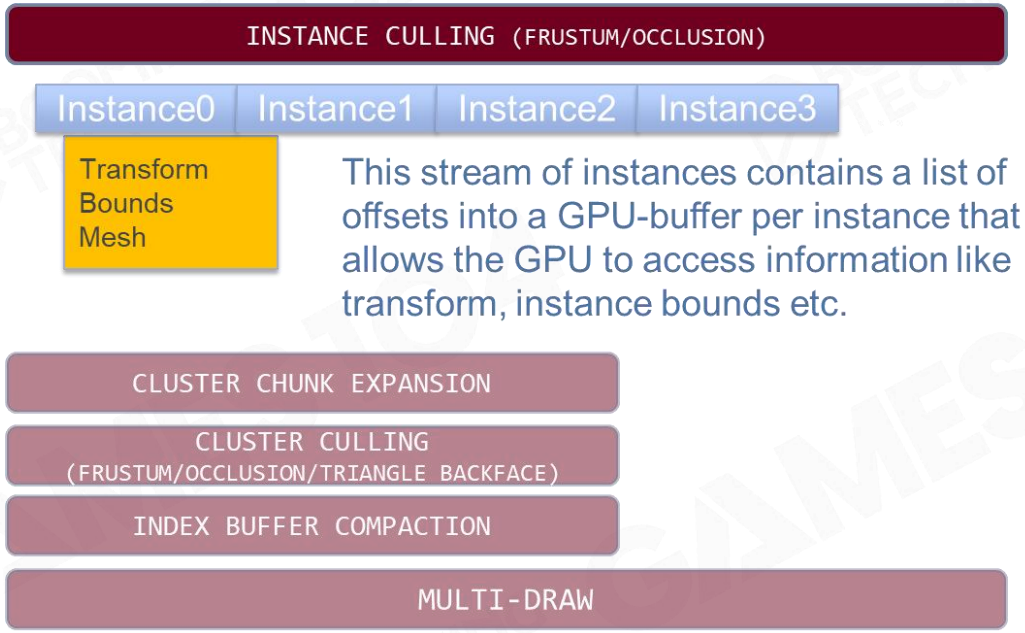


Works on CPU side

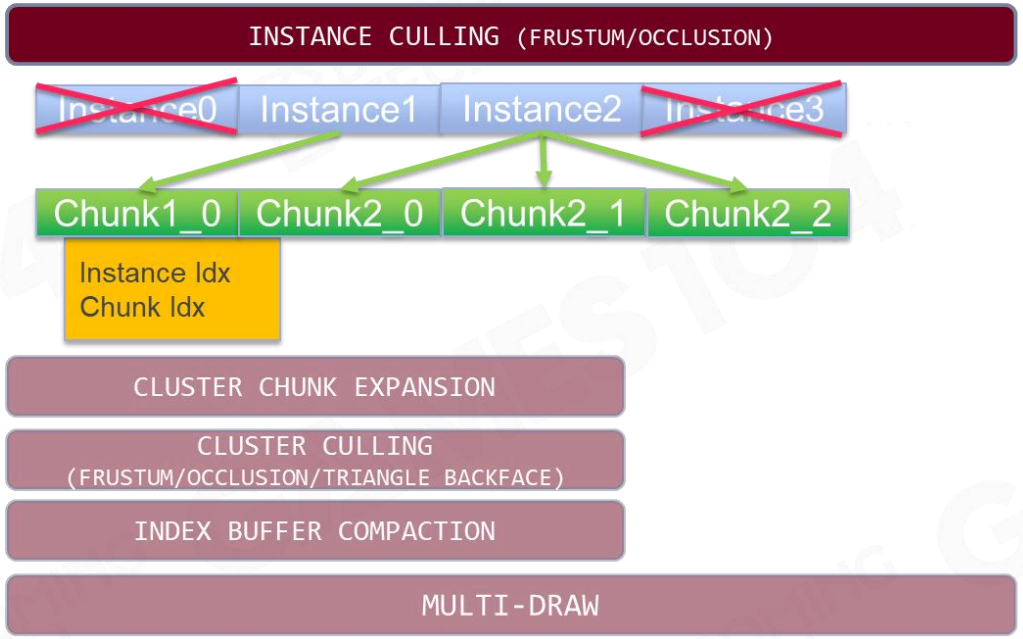
- Perform very coarse frustum culling and then batch all un-culled objects together by material
  - CPU quad tree culling
  - Drawcalls merged based on hash that build on non-instanced data:(e.g. material, renderstate, …).
- Update per instance data(e.g. transform, LOD factor...),static instances are persistent

GPU Instance Culling

- Output cluster chunks after instance culling
- Use the cluster chunk expansion (64 cluster in a chunk) to balance GPU threads within a wavefront.



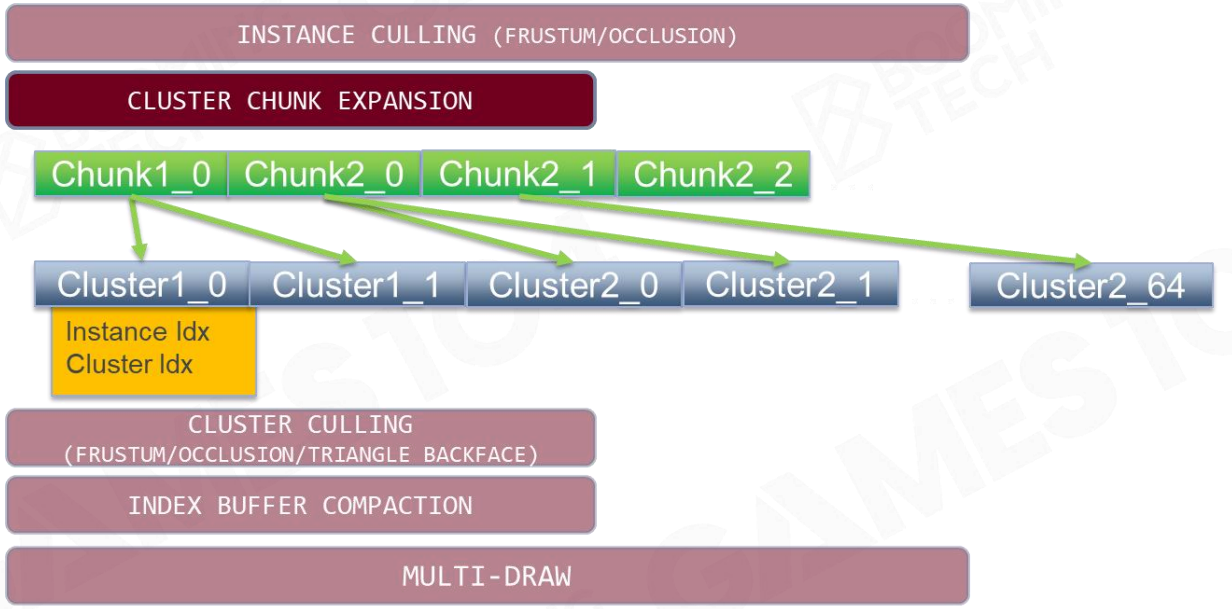
Initial State



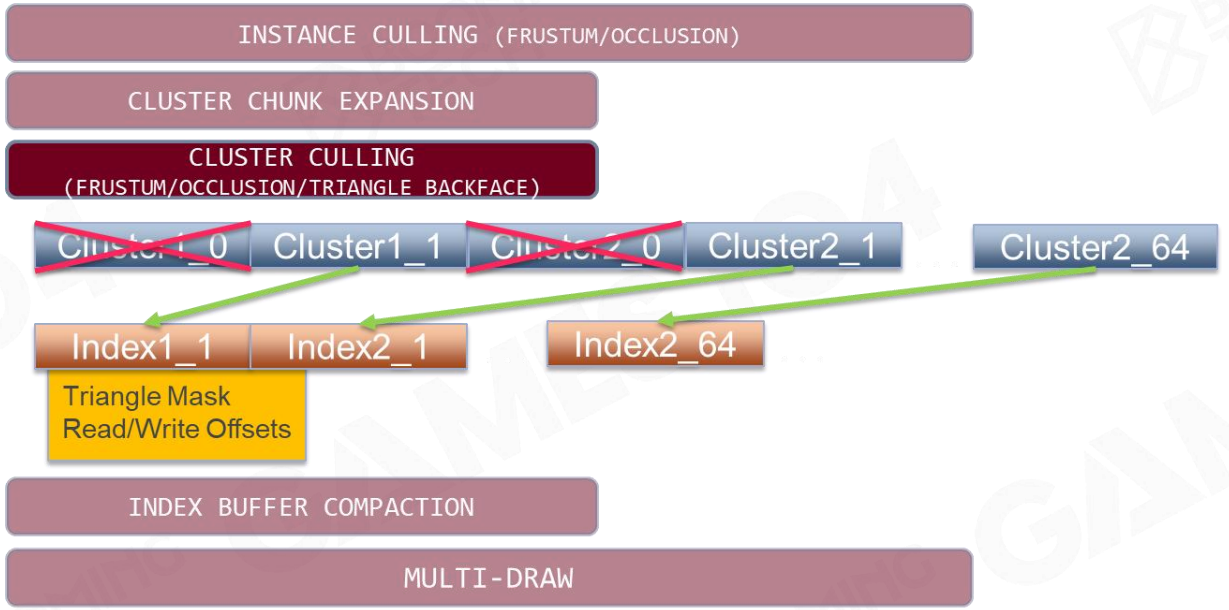
Step 1

GPU Cluster Culling

- Cluster culling by cluster bounding box
  - output cluster list
- Triangle backface culling
  - output triangle visibility result and write offsets in new index buffer



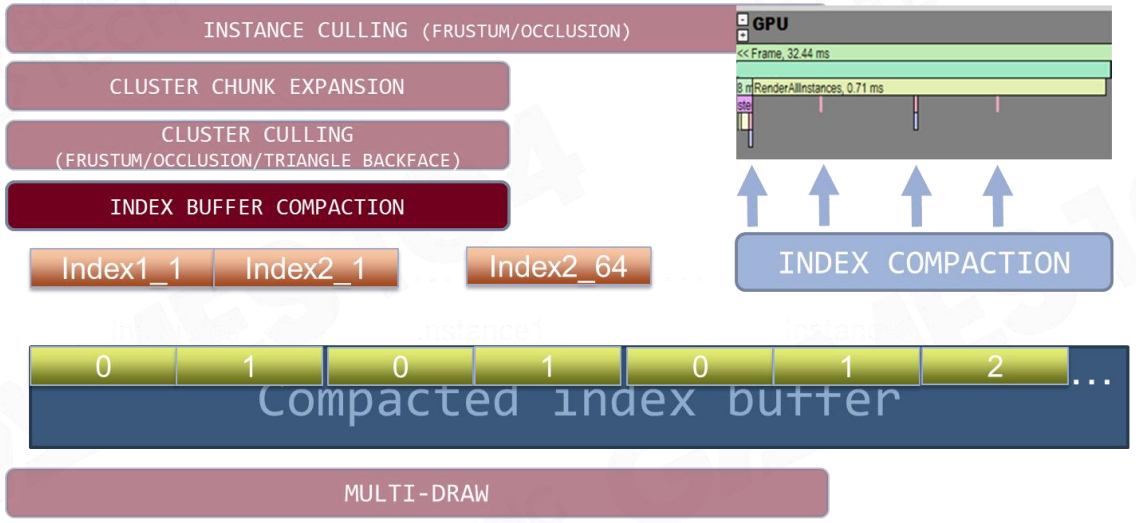
Step 2



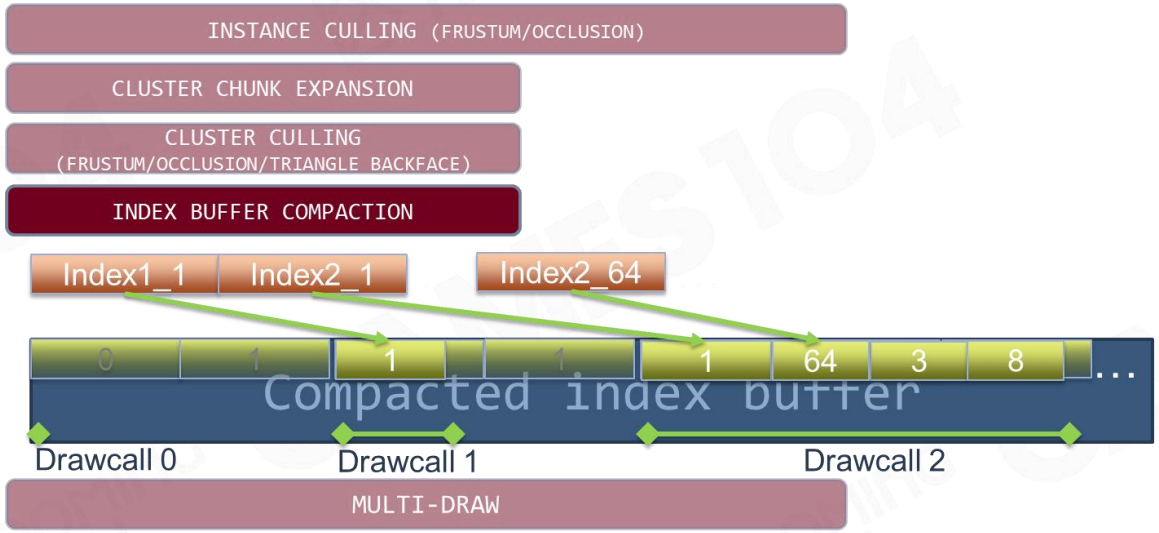
Step 3

Index Buffer Compaction

- Prepare a empty index buffer(8Mb) and per-assign space for each mesh instance
- Parallel copy the visible triangles index into the new index buffer
- Index buffer compaction and multi-draw rendering can be interleaved because of fixed size of new index buffer(8Mb)



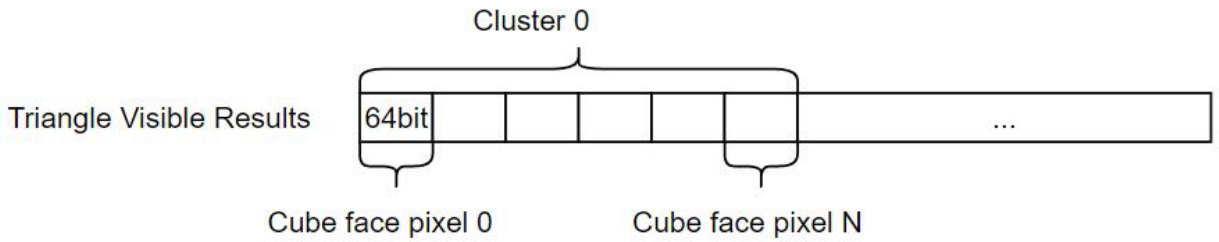
Step 4

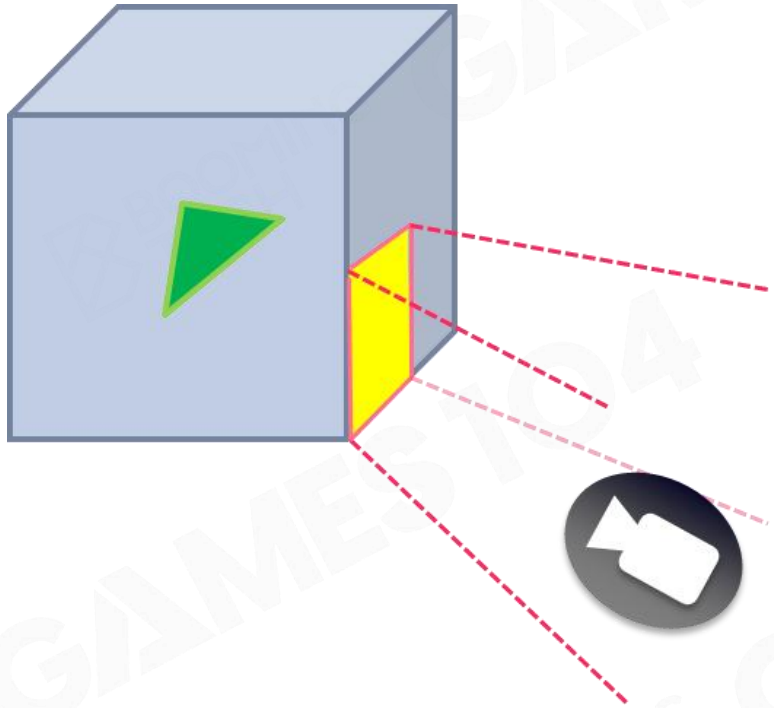


Step 5

Codec Triangle Visibility in Cube : Backface Culling

- Bake triangle visibility for pixel frustums of cluster centered cubemap
- Cubemap lookup based on camera
- Fetch 64 bits for visibility of all triangles in cluster





Occlusion Culling for Camera and Shadow

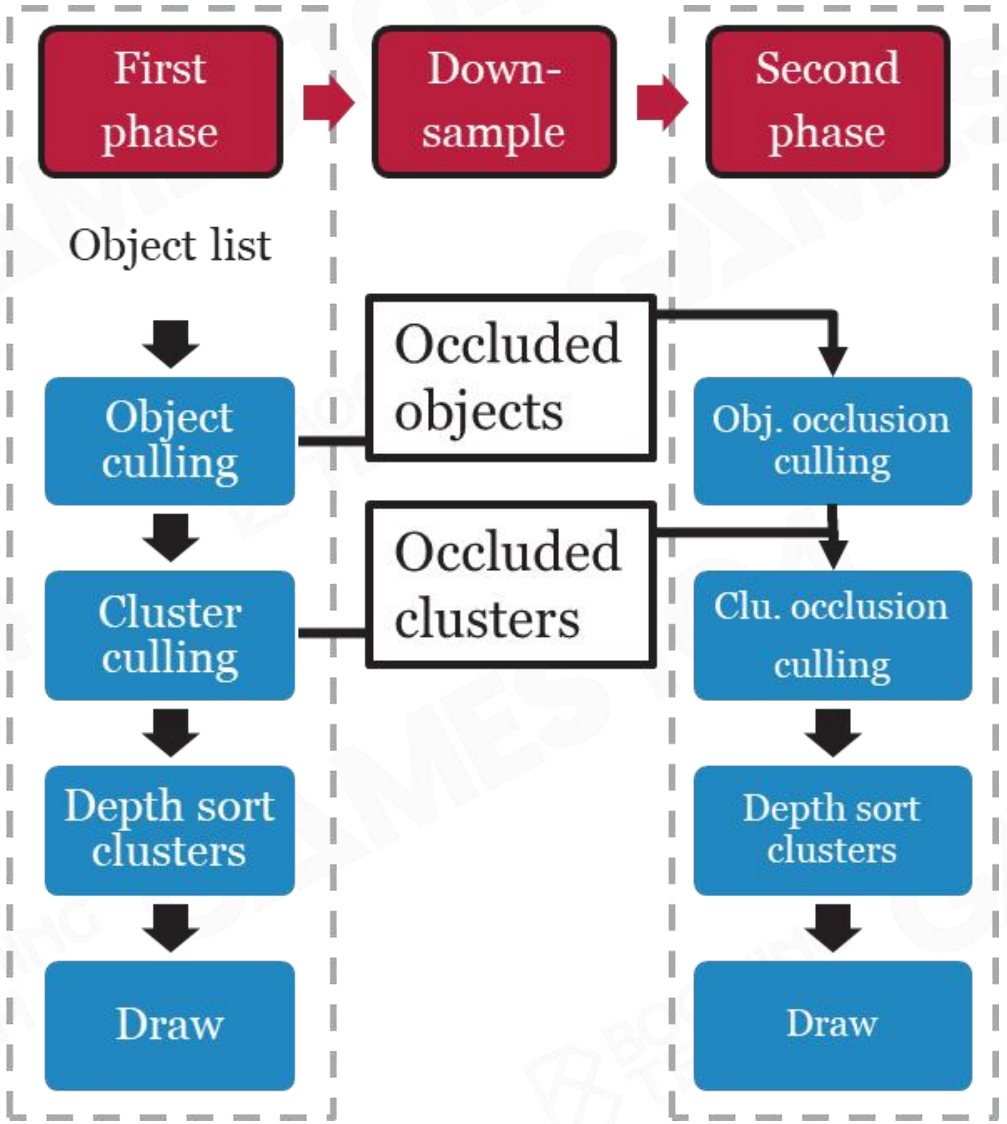
Occlusion Depth Generation

- Depth pre-pass with best occluders in full resolution
  - Choose best occluders by artist or heuristic (e.g. 300 occluders)
  - Holes can be from rejected occluder(bad occluder selection or alpha-tested geometry)
- Downsampled best occluders depth to 512x256
- Then combined with reprojection of 1/16 low resolution version of last frame's depth
  - Last frame's depth can helped to filled with holes.
  - False occlusion is possible from large moving objects in the last frame's depth, but works in most cases.
- Generate hierarchy-Z buffer for GPU culling

Two-Phase Occlusion Culling

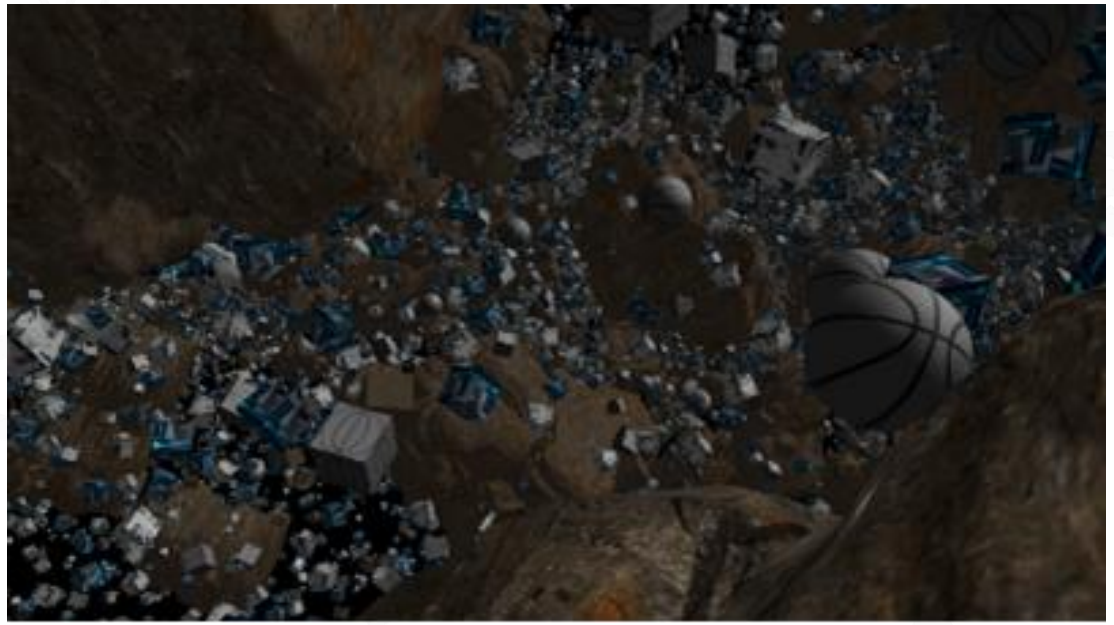
1st phase
Cull objects & clusters using last frame's depth pyramid
Render visible objects

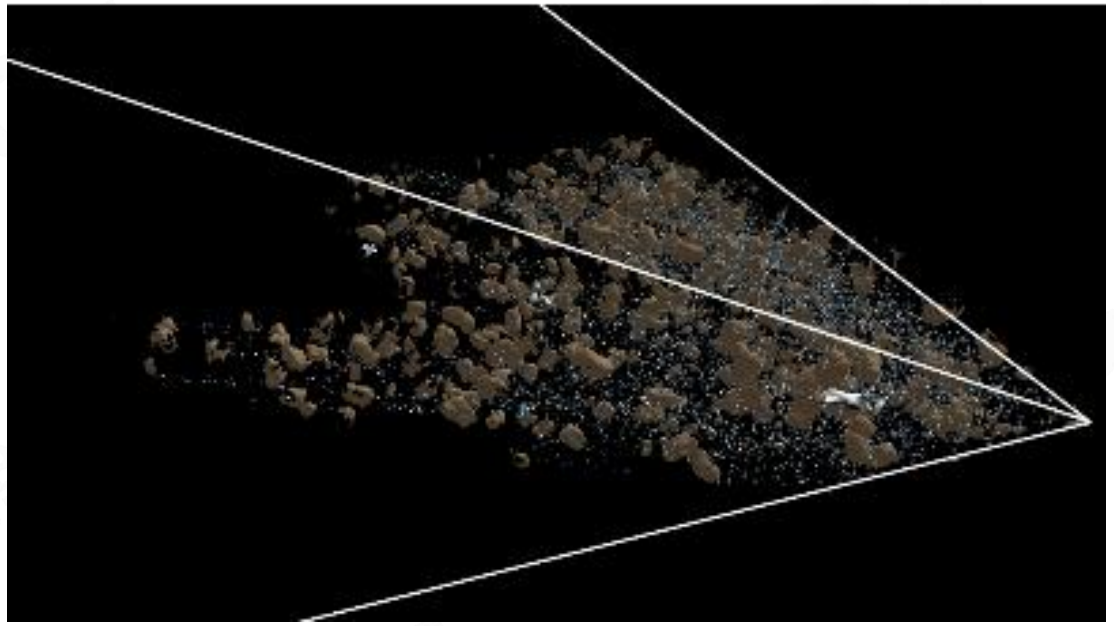
2nd phase
Refresh depth pyramid
Test culled objects & clusters
Render false negatives



Crazy Stressing Cases

- “Torture” unit test scene 250,000 separate moving objects
- 1 GB of mesh data (10k+ meshes)
- 8k2 texture cache atlas
- DirectX 11 code path
- 64 vertex clusters (strips)
- No ExecuteIndirect / MultiDrawIndirect
- Only two DrawInstancedIndirect calls





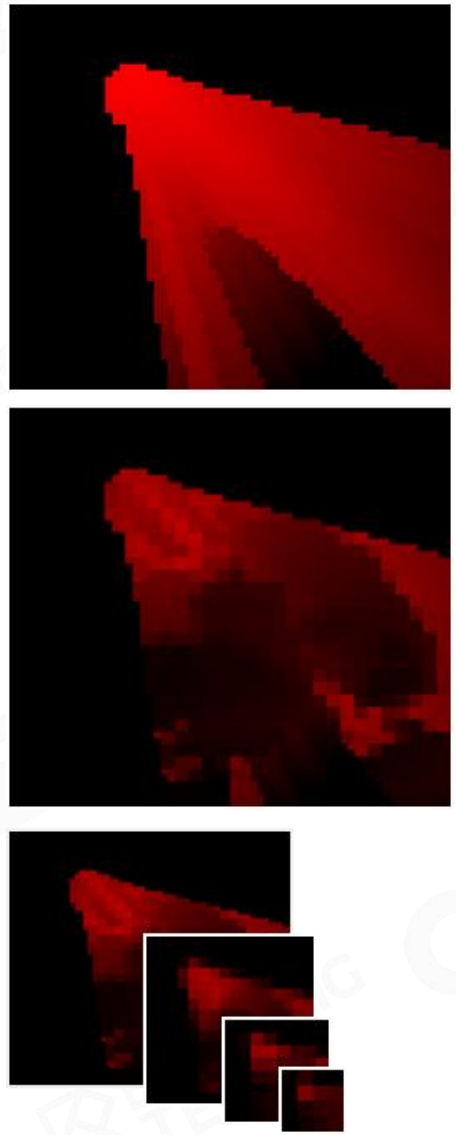
Xbox One, 1080p

GPU time	1st phase	2nd phase
Object culling + LOD	0.28 ms	0.26 ms
Cluster culling	0.09 ms	0.04 ms
Draw (G-buffer)	1.60 ms	< 0.01 ms
Pyramid generation	0.06 ms	
Total	2.3 ms	

CPU time: 0.2 milliseconds (single Jaguar CPU core)

Fast Occlusion for Shadow

- For each cascade
  - Generate camera depth reprojection (64x64 pixel)
  - Then combine with last frame's shadow depth reprojection
  - Generate hierarchy-Z buffer for GPU culling



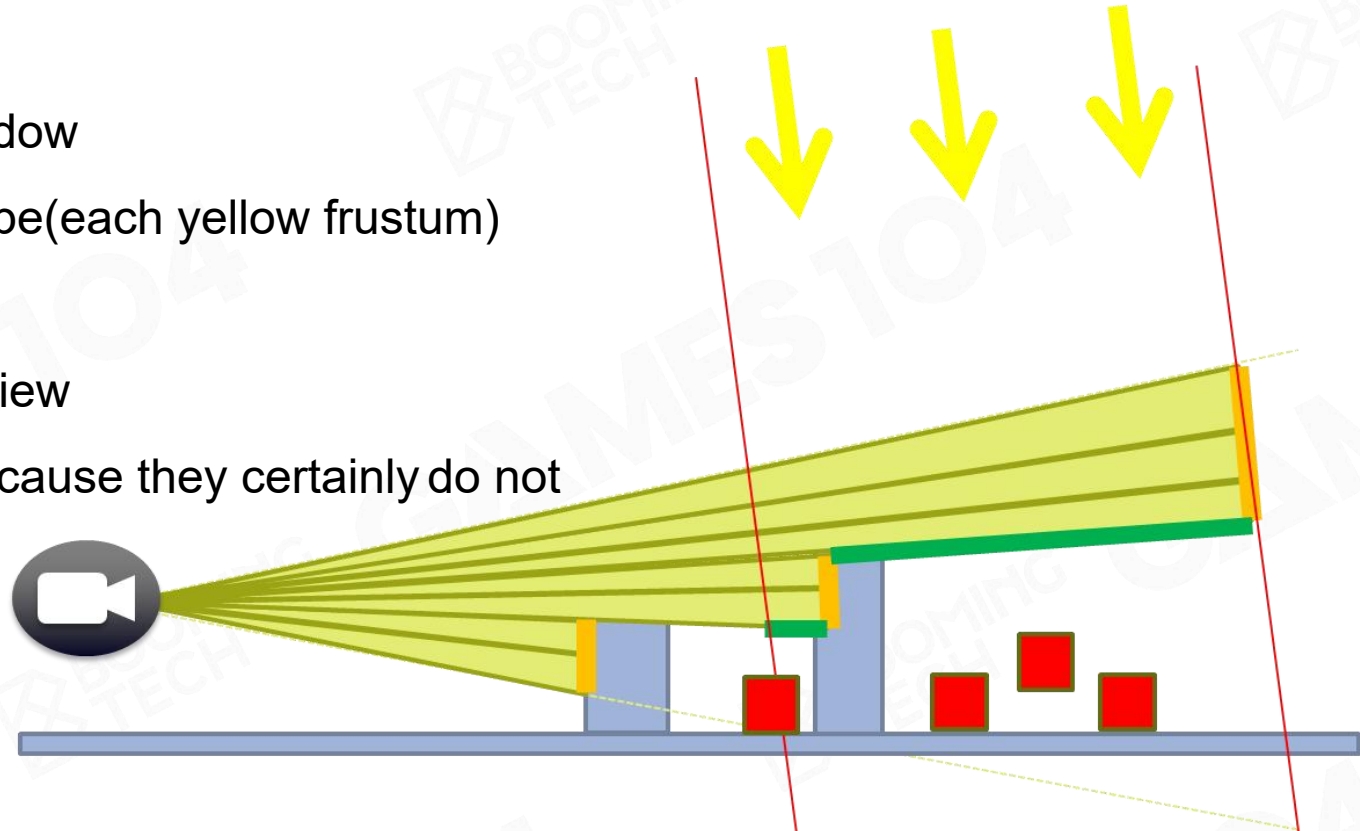
Camera Depth Reprojection for Shadow Culling

Motivation

- It is essential to cull objects in light view, which does not cast a visible shadow

Implementation

- Get camera visible areas that may appear shadow
  - For each 16*16 screen tile, construct a cube(each yellow frustum) according to min/max depth in this tile.
- Render max depth of these cubes in the light view
- All objects that far from depth can be culled because they certainly do not contribute to visible shadow



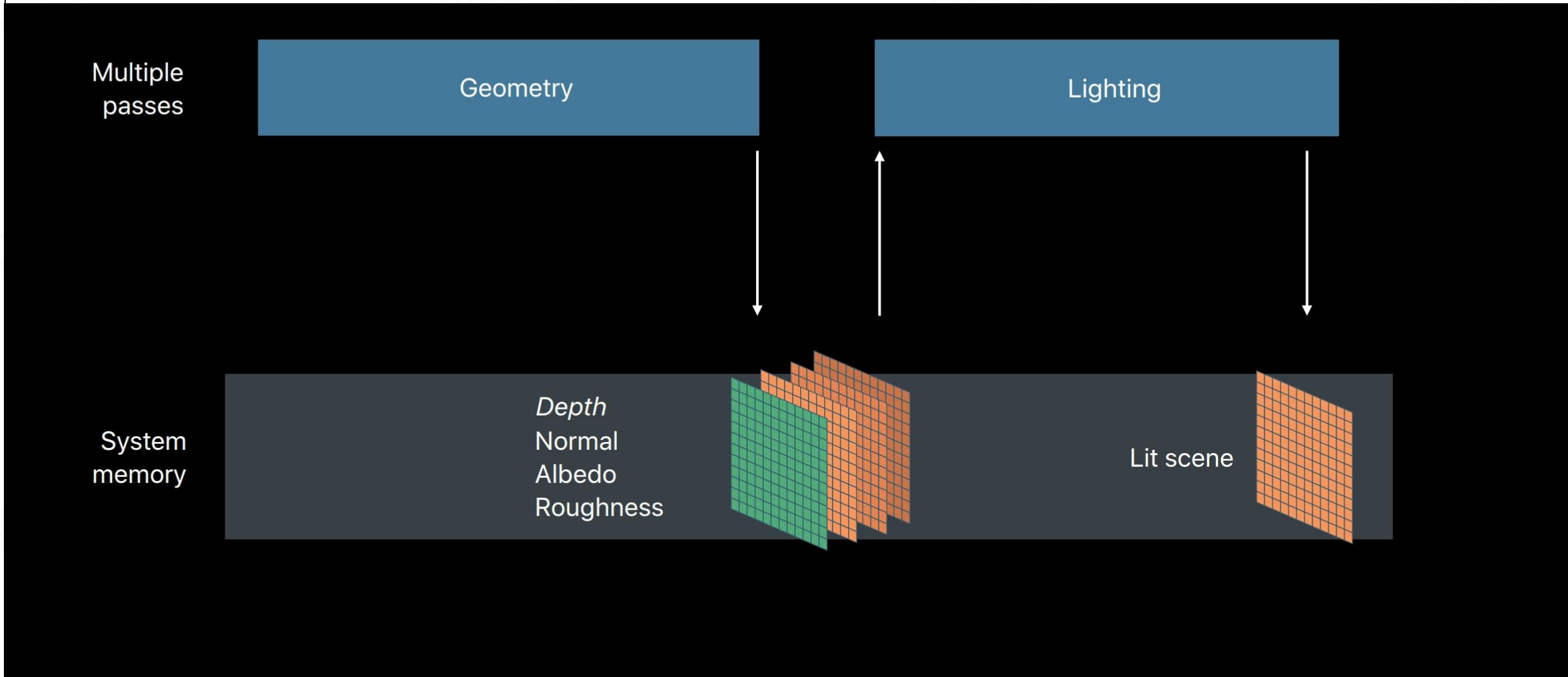
Best Cases of Camera Depth Reprojection

Visibility Buffer

Recap - Deferred Shading, G-Buffer

- Forward rendering shades all fragments in triangle- submission order
  - Wastes rendering power on pixels that don’t contribute to the final image
- Deferred shading solves this problem in 2 steps:
  - First, surface attributes are stored in screen buffers -> G-Buffer
  - Second, shading is computed for visible fragments only

Deferred Shading



Fat G-Buffer of Deferred Shading

- However, deferred shading increases memory bandwidth consumption:
  - Screen buffers for: normal, depth, albedo, material ID,…
  - G-Buffer size becomes challenging at high resolutions

	R	G	B	A
GB0	Normal (10:10)		Smoothness	MaterialId (2)
GB1	BaseColor			MatData(5)/Normal(3)
GB2	---------	MetalMask	Reflectance	AO
GB3	Radiosity/Emissive			

GBuffer layout for Disney deferred material (Ref. Frostbite Engine)

Challenges of Complex Scene

Journal of Computer Graphics Techniques Vol. 2, No. 2, 2013
The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading http://jcgt.org

The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading

Christopher A. Burns
Intel Labs

Warren A. Hunt
Intel Labs

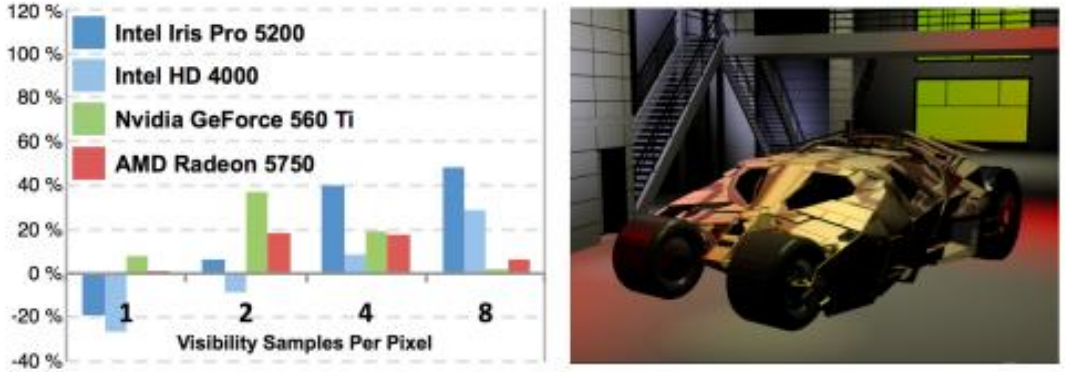


Figure 1. Compared to a conventional g-buffer pipeline, our technique demonstrates out-performance with large sample counts, especially on micro-architectures with deep cache hierarchies.

Visibility Buffer - Filling

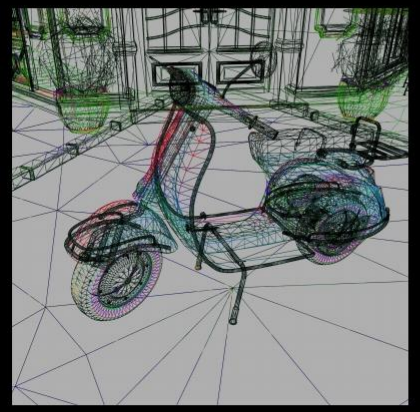
- Visibility Buffer generation step
  - For each pixel in screen:
    - Pack (alpha masked bit, drawID, primitiveID) into one 32-bit UINT
    - Write that into a screen-sized buffer
  - The tuple (alpha masked bit, drawID, primitiveID) will allow a shader to access the triangle data in the shading step

Visibility
(Object/Triangle)

Visibility Buffer - Shading

- For each pixel in screen-space we do:
  - Get drawID/triangleID at pixel pos
  - Load data for the 3 vertices from the VB
  - Compute triangle gradients
  - Interpolate vertex attributes at pixel pos using gradients
    - Attribs use w from position to compute perspective correct interpolation
    - MVP matrix is applied to position
  - We have all data ready: shade and calculate final color

Pipeline of Visibility Buffer



Geometry

Barycentrics

Primitive ID

Geometry reconstruction

Material shading

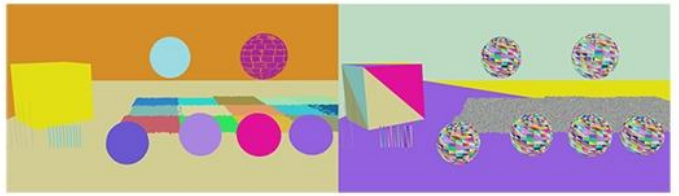
Lighting

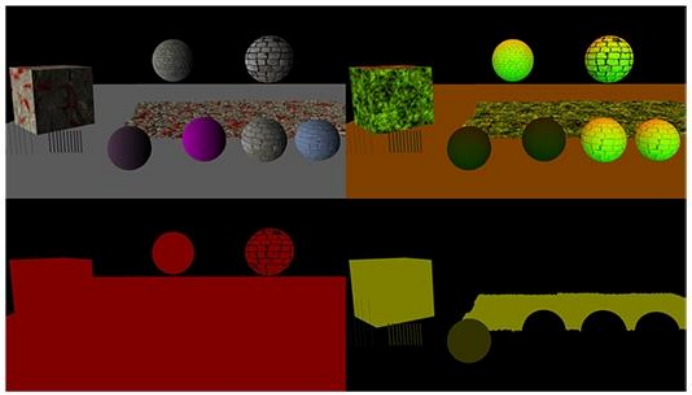
Geometry

Lit scene

Visibility Buffer
+
Deferred Shading

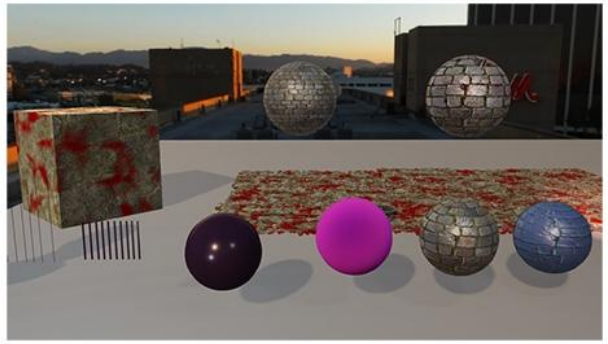
Visibility
(Object/Triangle)





GBuffer

Final



Correct Texture Mipmap with Gradient

Without SampleGrad

$$SampleGrad(sample\_state, uv, (\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}), (\frac{\partial v}{\partial x}, \frac{\partial v}{\partial y}))$$

Results

Total

- 8 Million Triangles
- 5 Million Vertices

Visibility Buffer

GPU AMD RADEON R9 380	1080p	1440p	2160p
No MSAA	8.57	10.72	15.19
No MSAA – No Culling	14.52	15.86	20.45
2x MSAA	11.44	16.38	25.87
4x MSAA	15.27	20.82	37.86

Deferred Shading

GPU AMD RADEON R9 380	1080p	1440p	2160p
No MSAA	9.75	12.30	20.19
No MSAA – No Culling	14.16	16.66	24.06
2x MSAA	16.16	23.09	42.68
4x MSAA	24.90	36.37	69.64

Virtual Geometry - Nanite


Challenges of Realistic Rendering

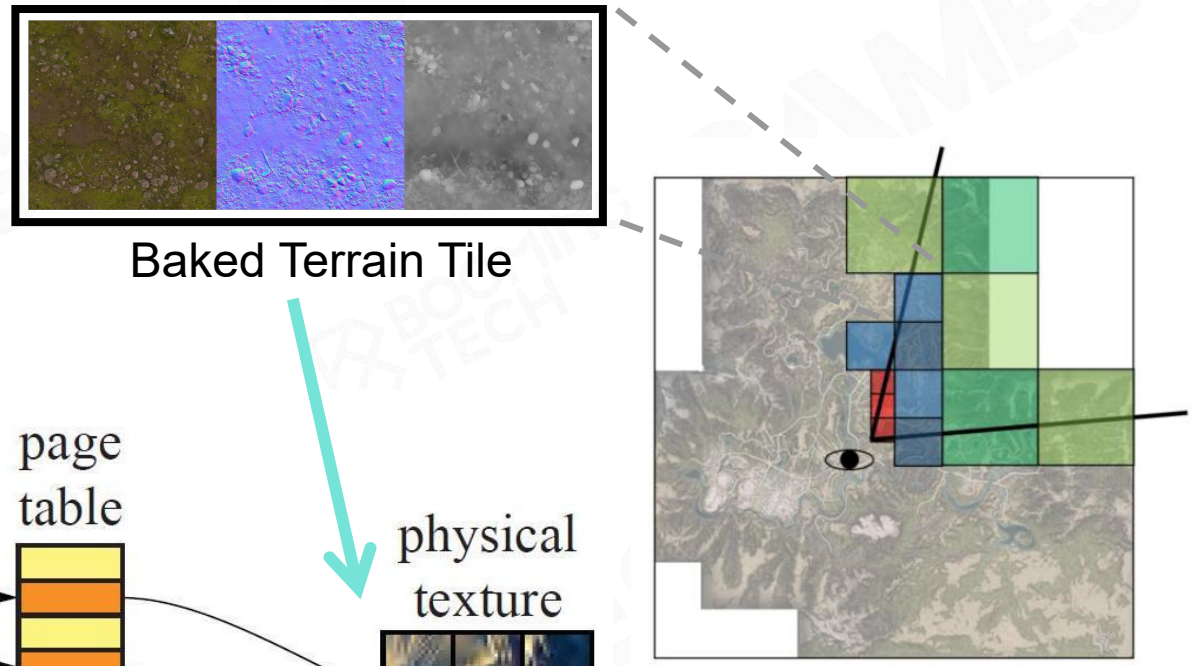
Challenges of Realistic Rendering

Nanite Overview

- Overview
- **Geometry Representation**
  - Cluster-based LoD
  - BVH and runtime LoD
- Rendering
  - Software and Hardware Rasterization
  - Visibility Buffer
  - Deferred Materials
  - Tile-based Acceleration
- Virtual Shadow Map
- Streaming and Compression

Virtual Texture

- Build a virtual indexed texture to represent all blended terrain materials for whole scene
- Only load materials data of tiles based on view-depend LOD
- Pre-bake materials blending into tile and store them into physical textures



The Dream

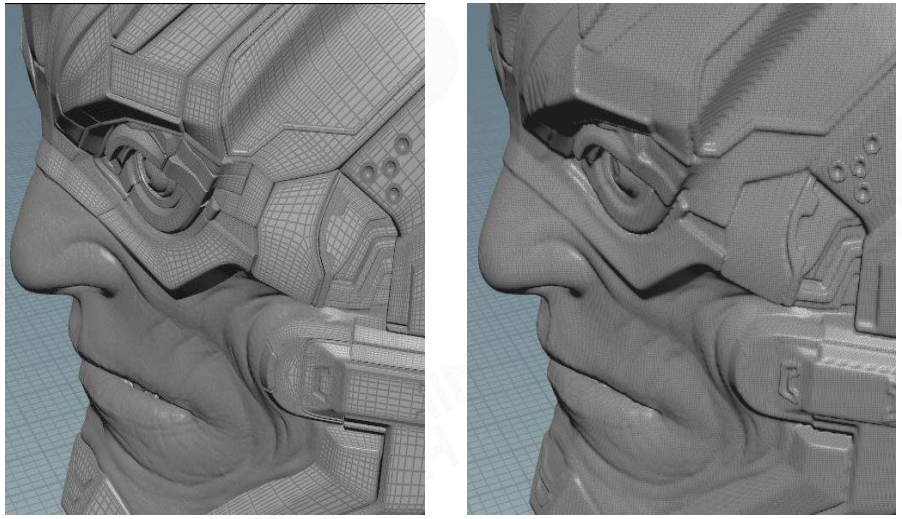
- Virtualize geometry like we did textures
  - No more budgets
    - Poly count
    - Draw calls
    - Memory
- Directly use film quality source art
  - No manual optimization required
- No loss in quality

Reality

- **MUCH** harder than virtual texturing
  - Not just memory management
  - Geometry detail directly impacts rendering cost
  - Geometry is not trivially filterable (**SDF, Voxels, Point Clouds)**

Voxels?

- Spatially uniform data distribution
- Big memory consumption
- Attribute leaking

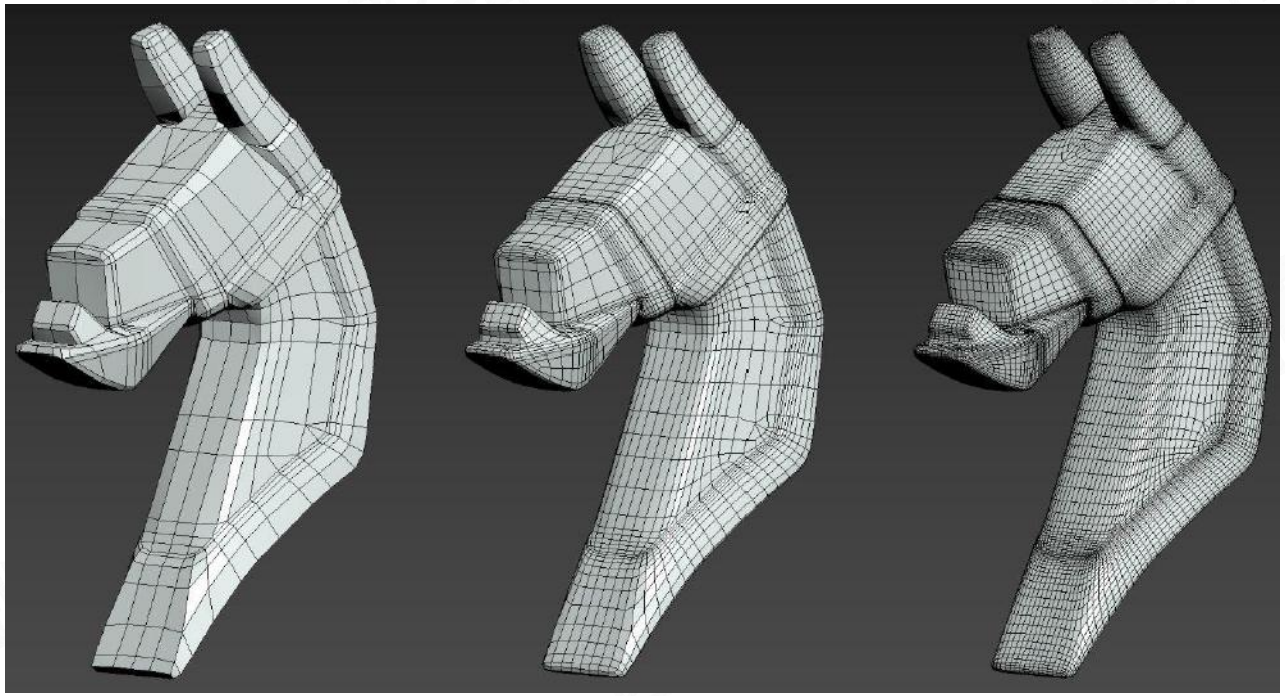


2M poly bust resampled to 13M voxels

- 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

Subdivision Surfaces?

- Subdivision by definition is amplification only
- Great for up close but doesn't get simpler than base mesh
- Sometimes produces an excessive number of triangles



Maps-based Method?

- Works well for organic surfaces that already are uniformly sampled
- Difficult to control hard surface features
- Sometimes object surface is not connected

Point Cloud?

- Massive amounts of overdraw
- Requires hole filling

https://highperformancegraphics.org/slides22/Software_Rasterization_of_2_Billion_Points_in_Real_Time.pptx

Texture details

Point cloud details

Foundation of Computer Graphics

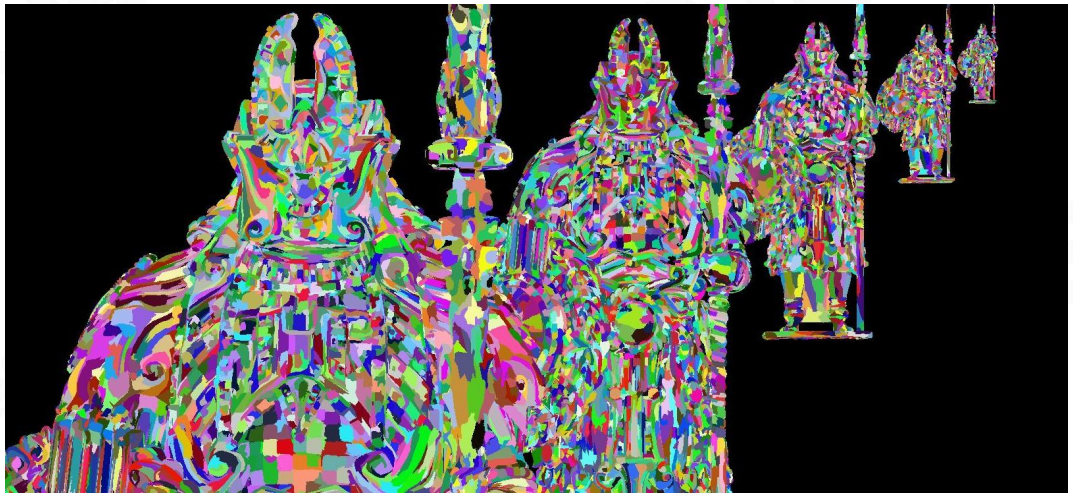
- The most elemental, atomic unit of surface area in 3D space
- Every surface can be turned into triangles

Nanite Geometry Representation

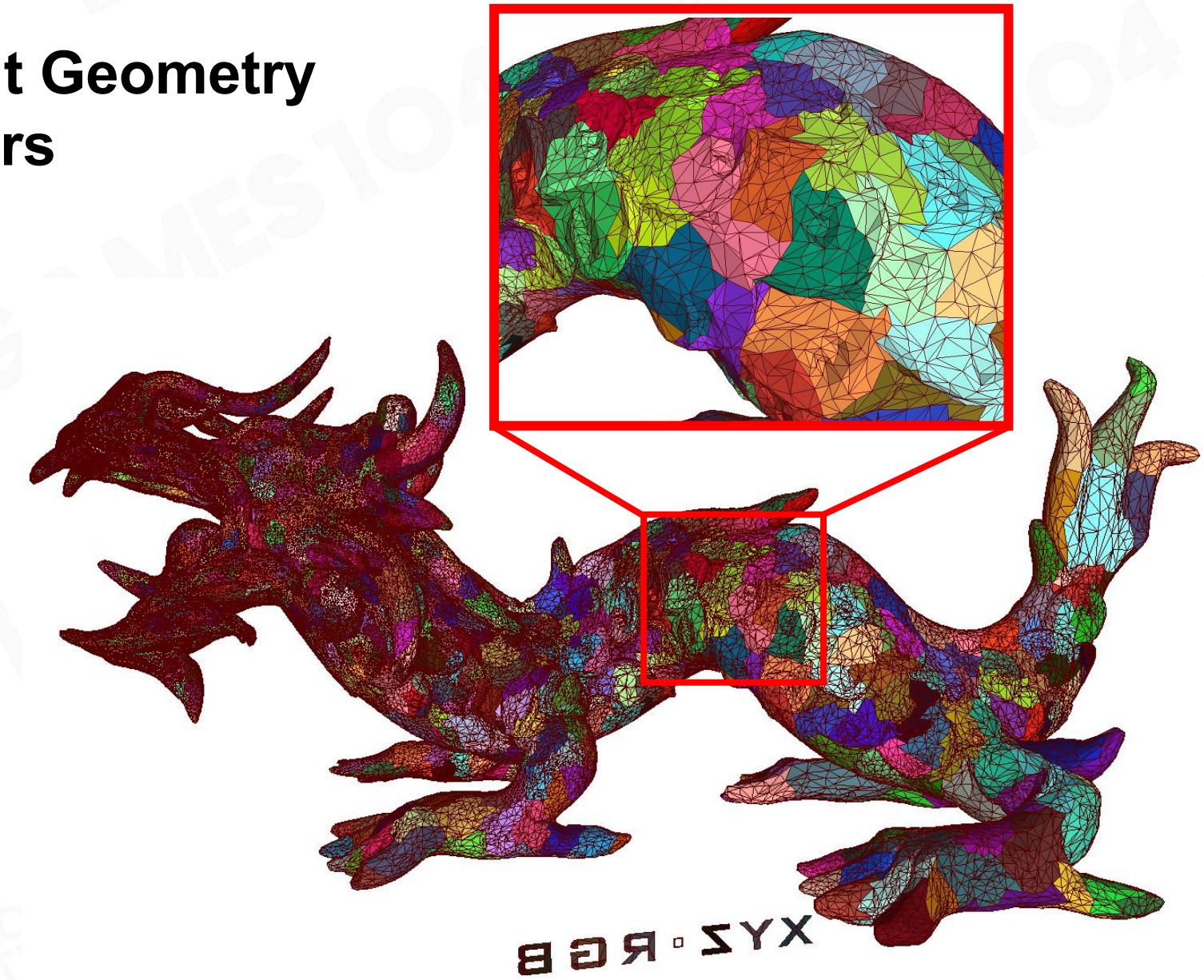
Screen Pixels and Triangles

- Linear scaling in instances can be ok
- Linear scaling in triangles is not ok

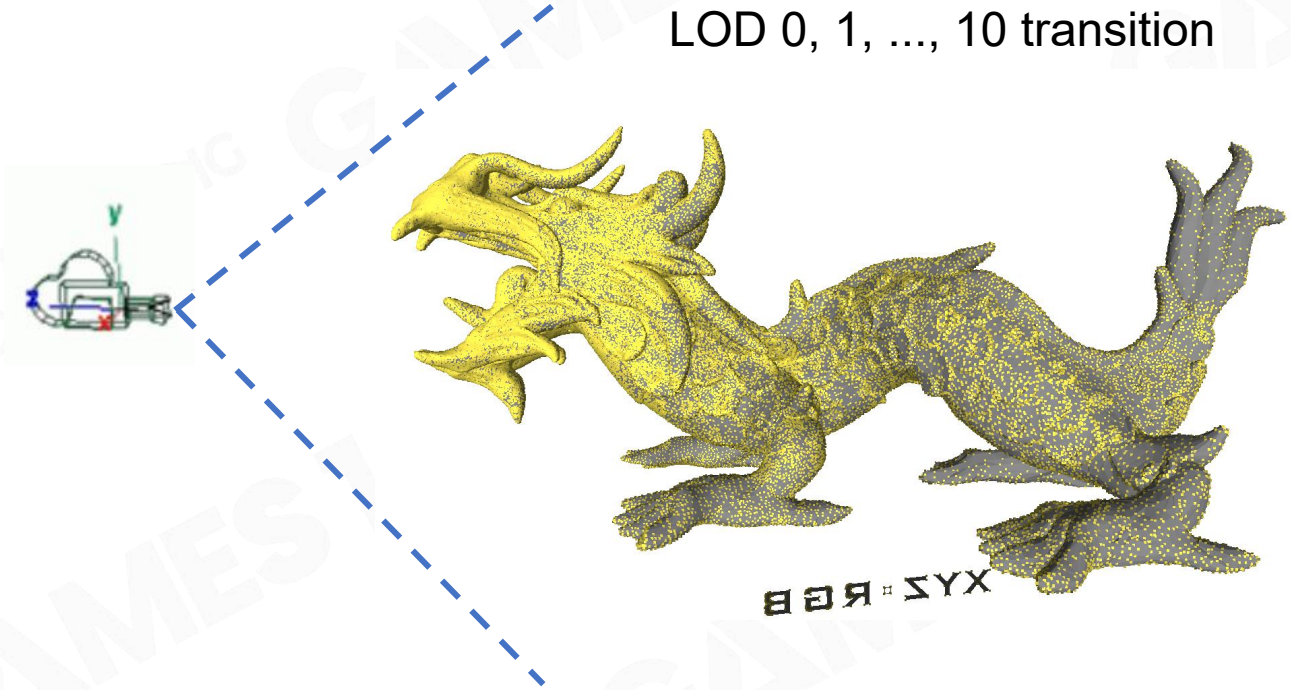
Why should we draw more triangles than screen pixels?

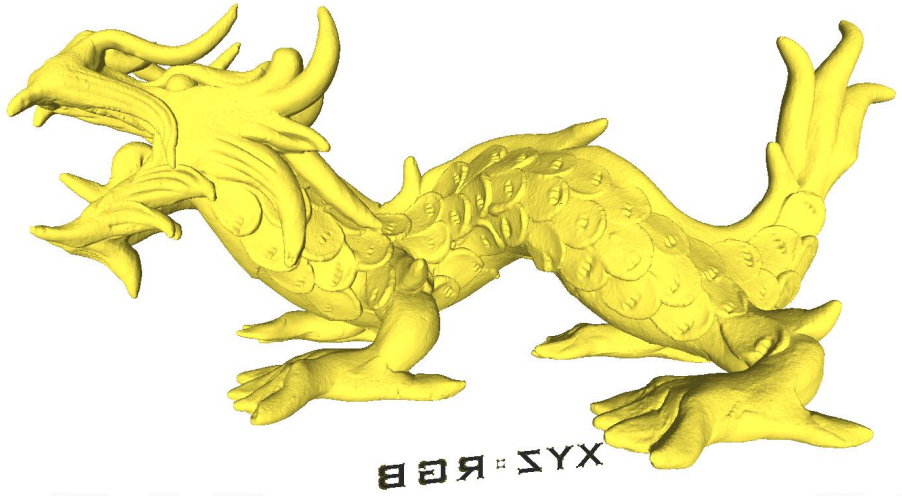


Represent Geometry by Clusters



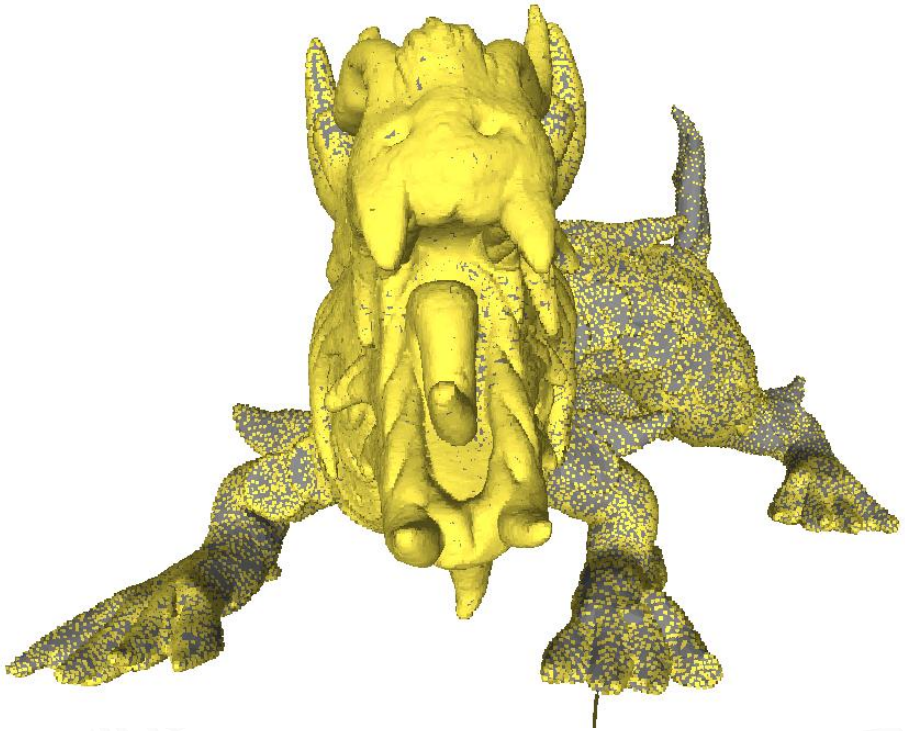
View Dependent LOD Transitions – Better than AC Solutions





Similar Visual Apperance with 1/30 Rendering Cost!

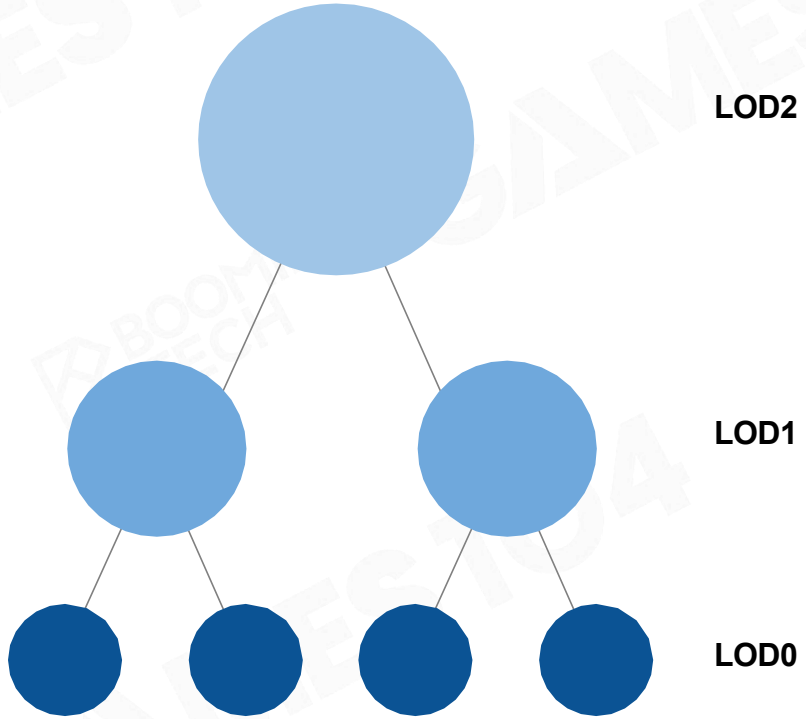
721M triangles



24M triangles

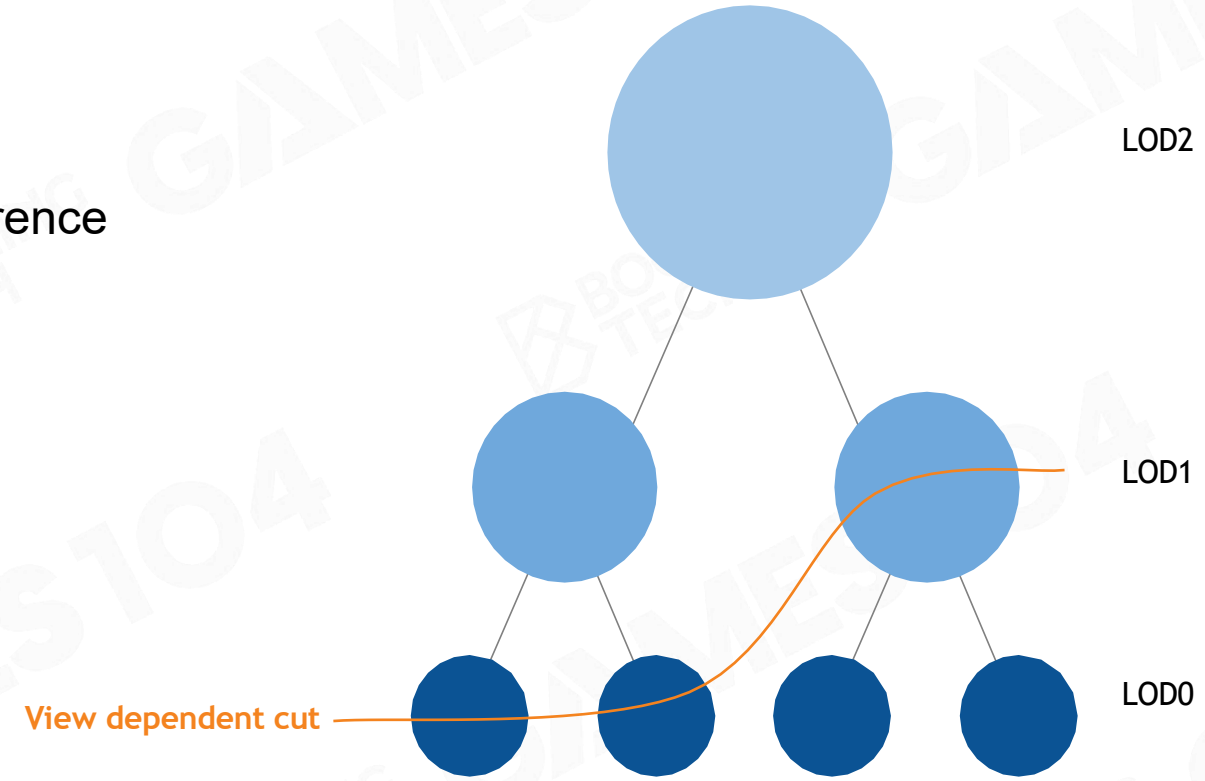
Naïve Solution - Cluster LoD Hierarchy

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



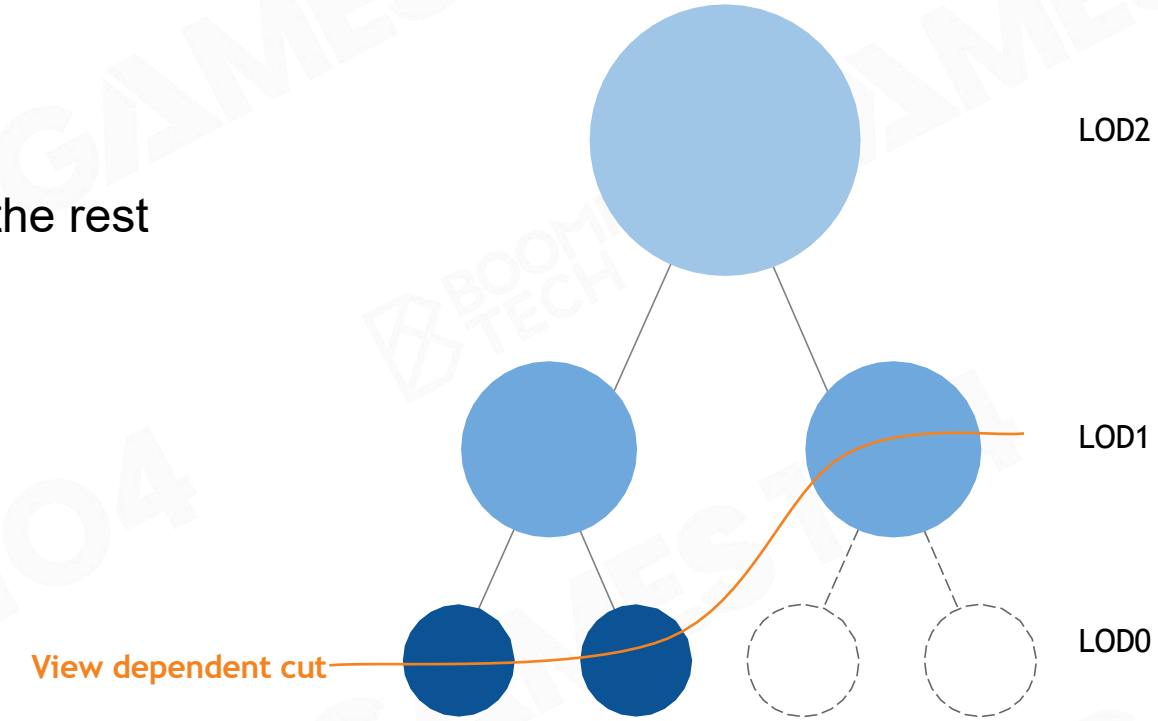
Naïve Solution - Decide Cluster LOD Run-time

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



Naïve Solution – Simple Streaming Idea

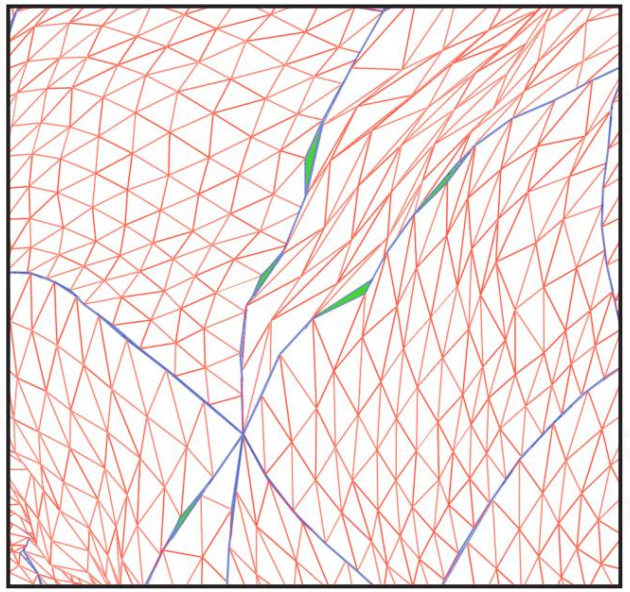
- Entire tree doesn't need to be in memory at once
- Can mark any cut of the tree as leaves and toss the rest
- Request data on demand during rendering
  - Like **virtual texturing**

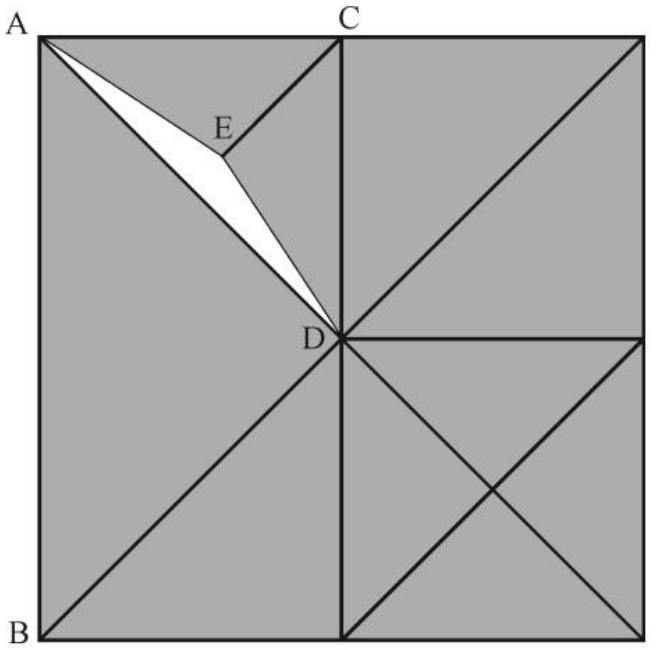


But, How to Handle LOD Cracks

- If each cluster decides LOD independent from neighbors, cracks!

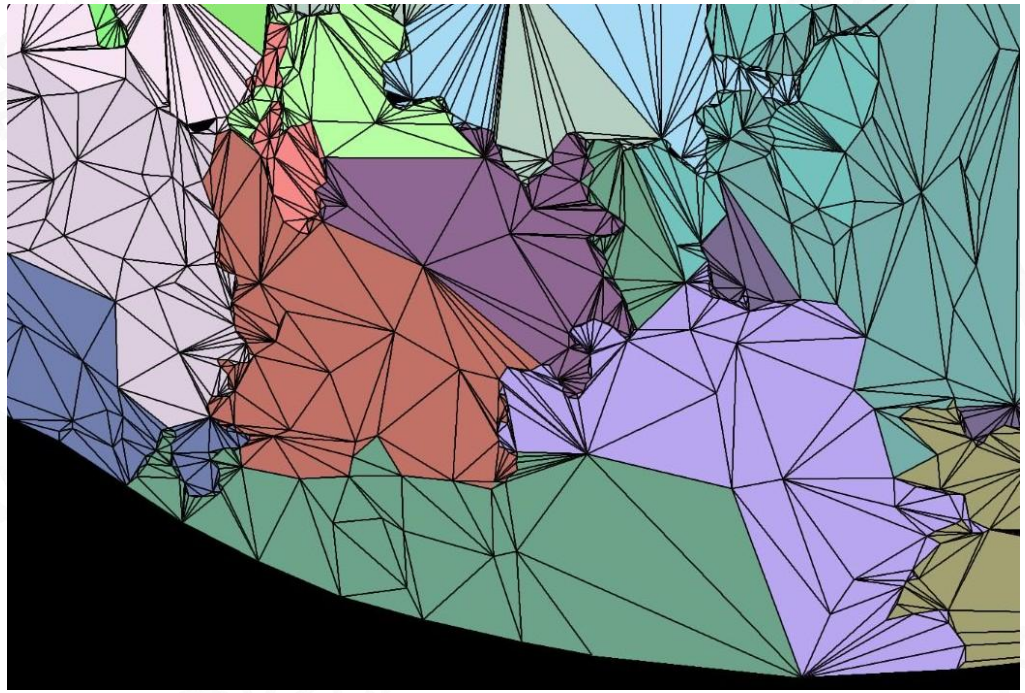
- Naive solution:
  - Lock shared boundary edges during simplification
  - Independent clusters will always match at boundaries

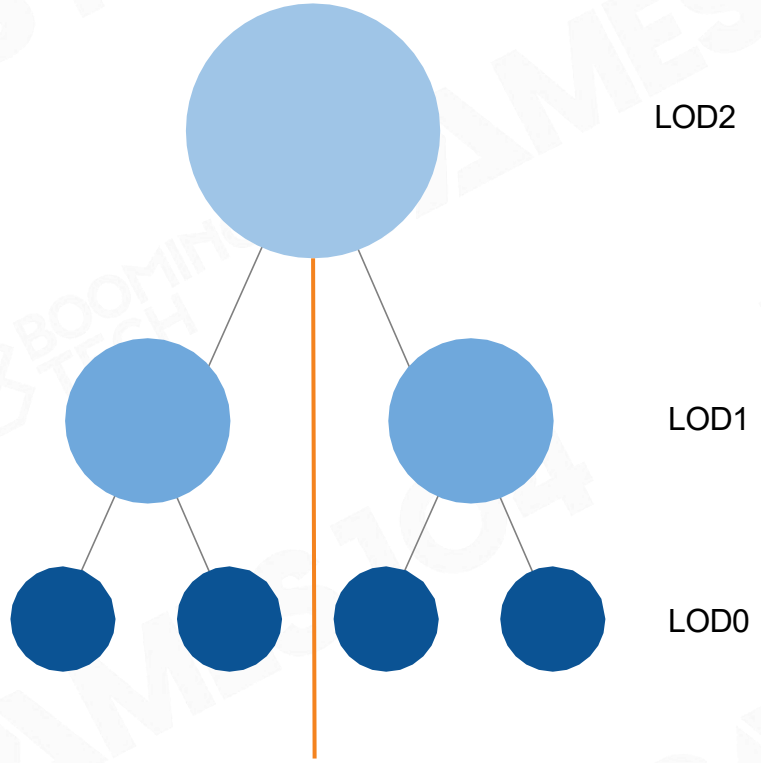




Locked Boundaries? Bad Results

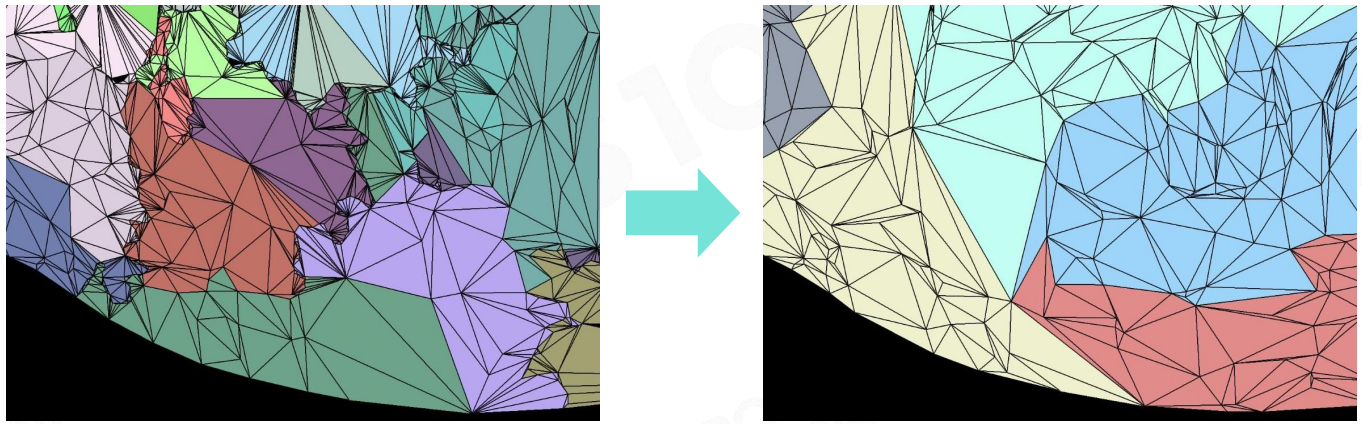
- Collects dense cruft
- Especially between deep subtrees





Nanite Solution - Cluster Group

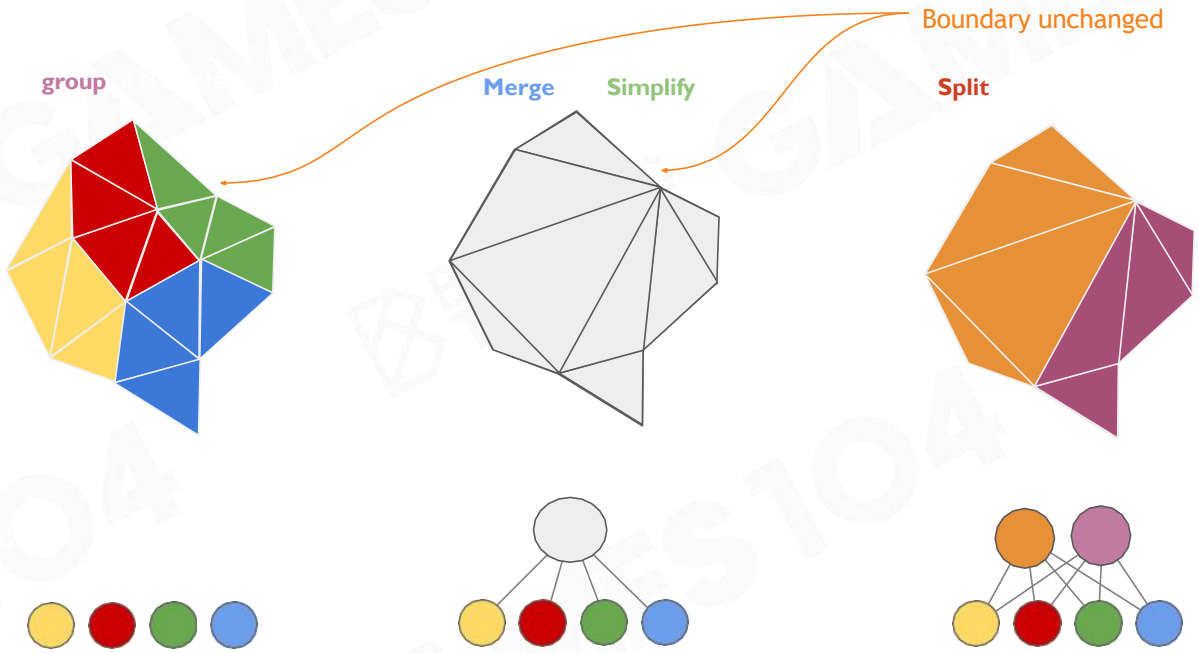
- Can detect these cases during build
- Group clusters
- Force them to make the same LOD decision
- Now free to unlock shared edges and collapse them



Build Operations

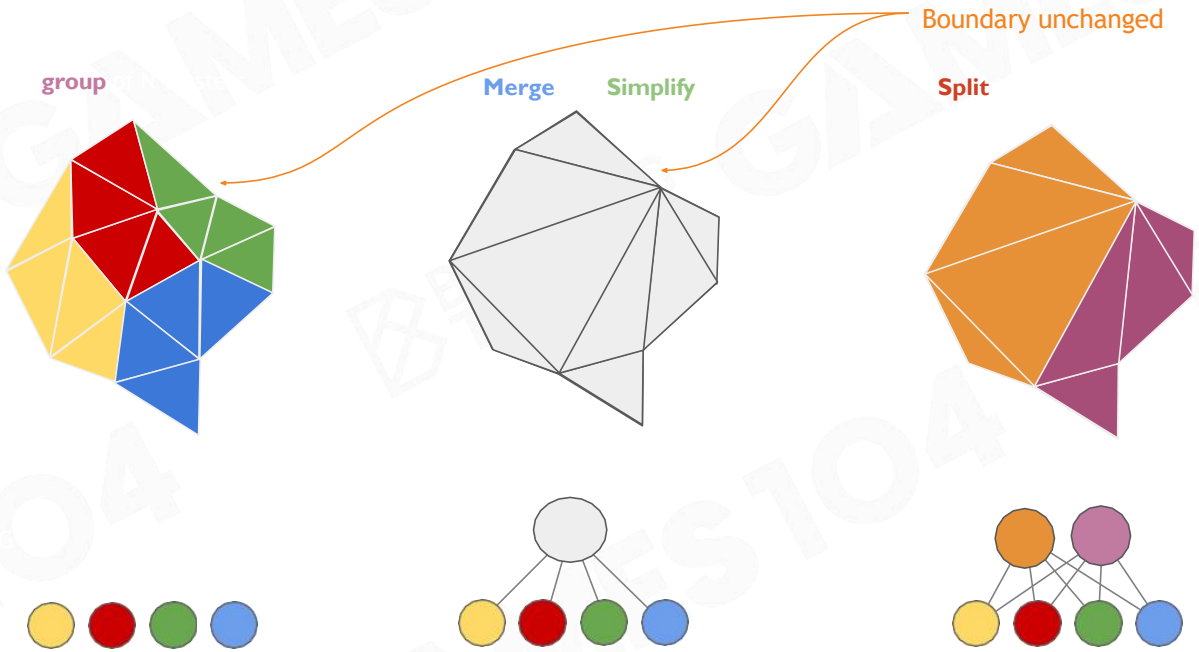
Here is an illustration of the process

- Pick grouped these 4 adjacent clusters
- Merge and Simplify the clusters to half the number of triangles
- Split simplified triangle list back into 2 new clusters
- We now have reduced 4 4-triangle clusters to 2 4-triangle clusters

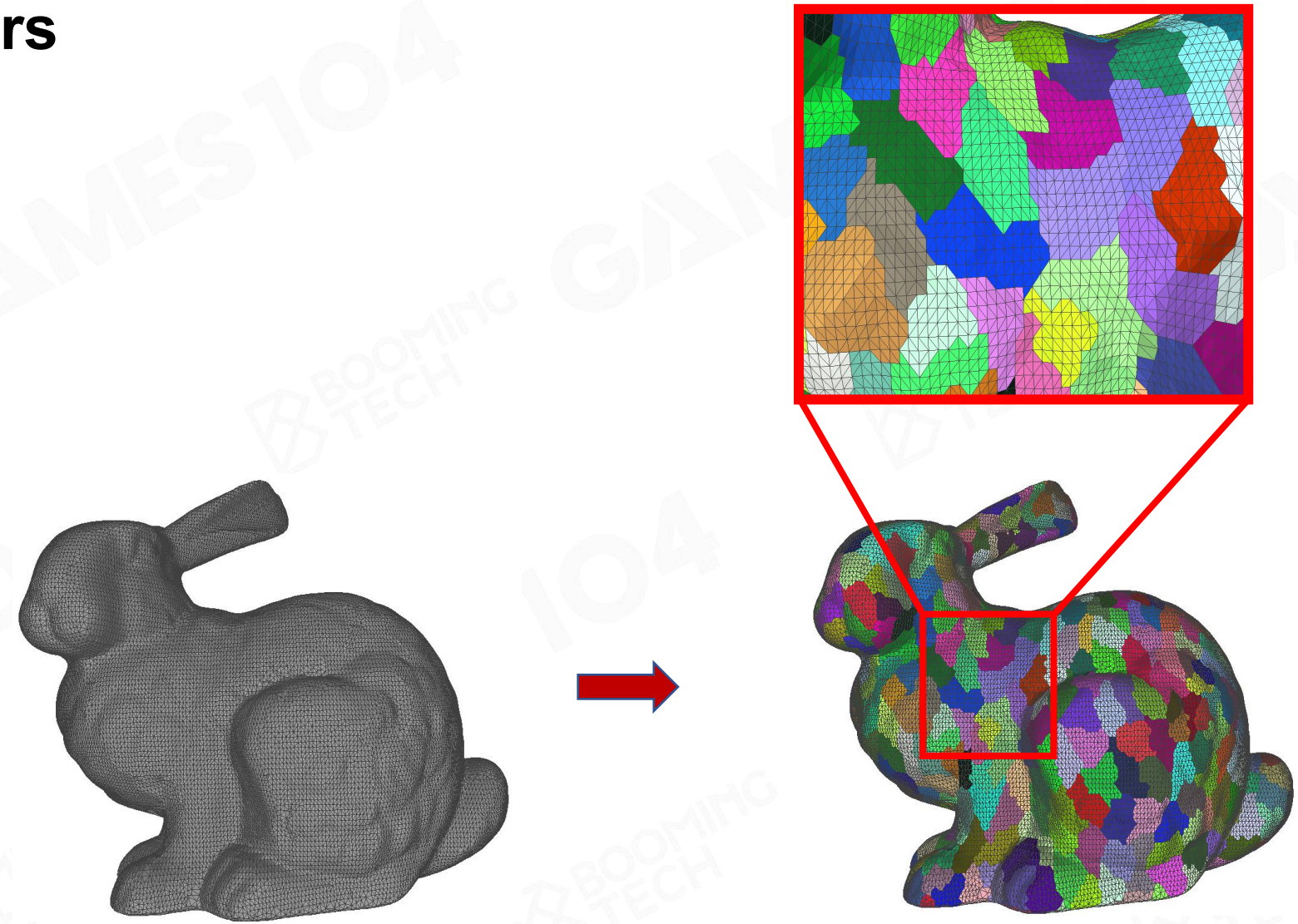


Build Operations

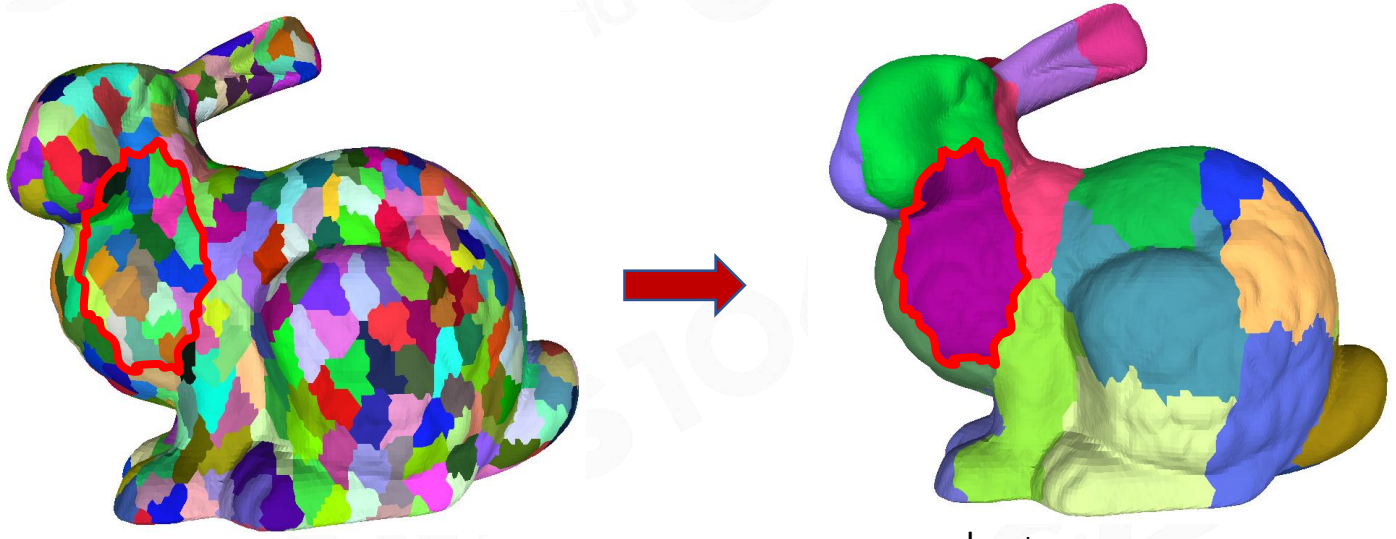
- Cluster original triangles
- While NumClusters > 1
  - Group clusters to clean their shared boundary
  - Merge triangles from group into shared list
  - Simplify to 50% the number of triangles
  - Split simplified triangle list into clusters (128 tris)



Build Clusters

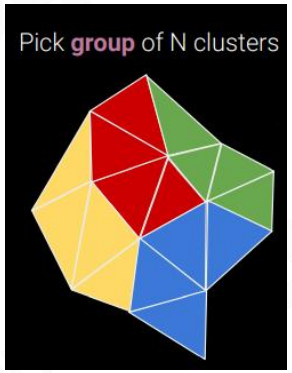


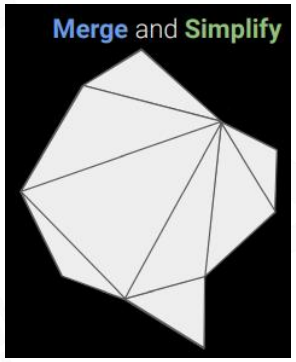
Simplification on Cluster Group

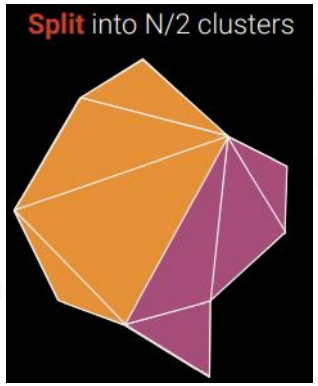


18 clusters (LOD_0)

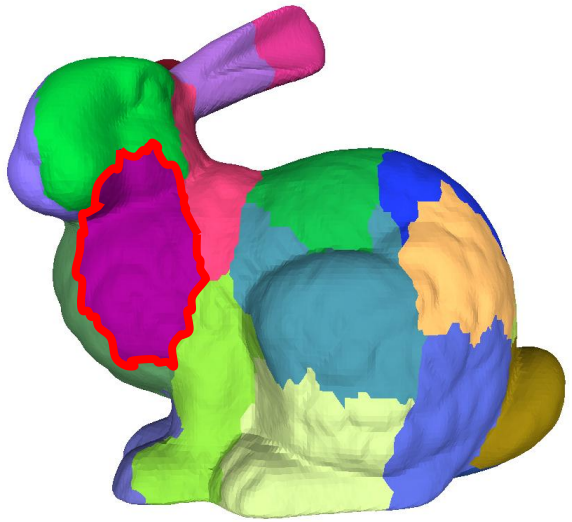
cluster group
(LOD_0)







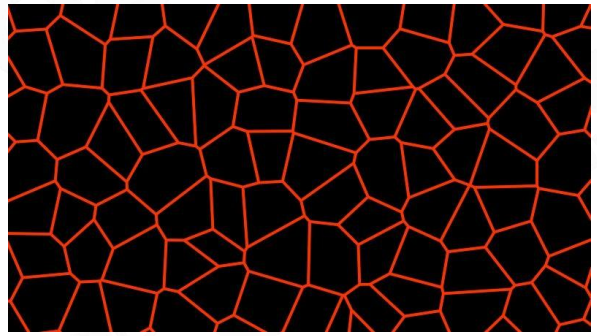
18 clusters (LOD_0)

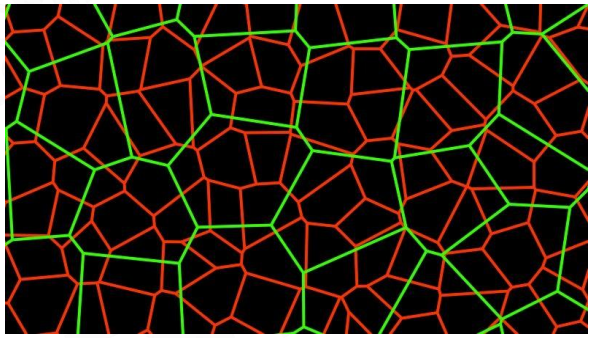


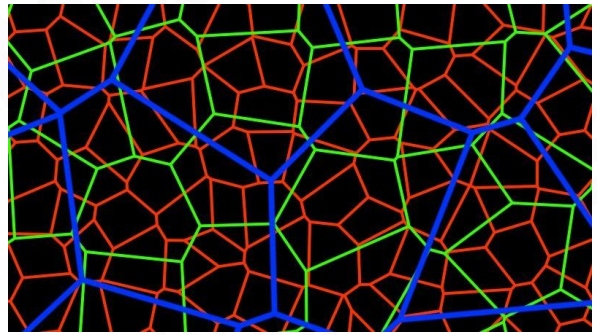
cluster group
(LOD_0)

Alternate Group Boundaries between Levels

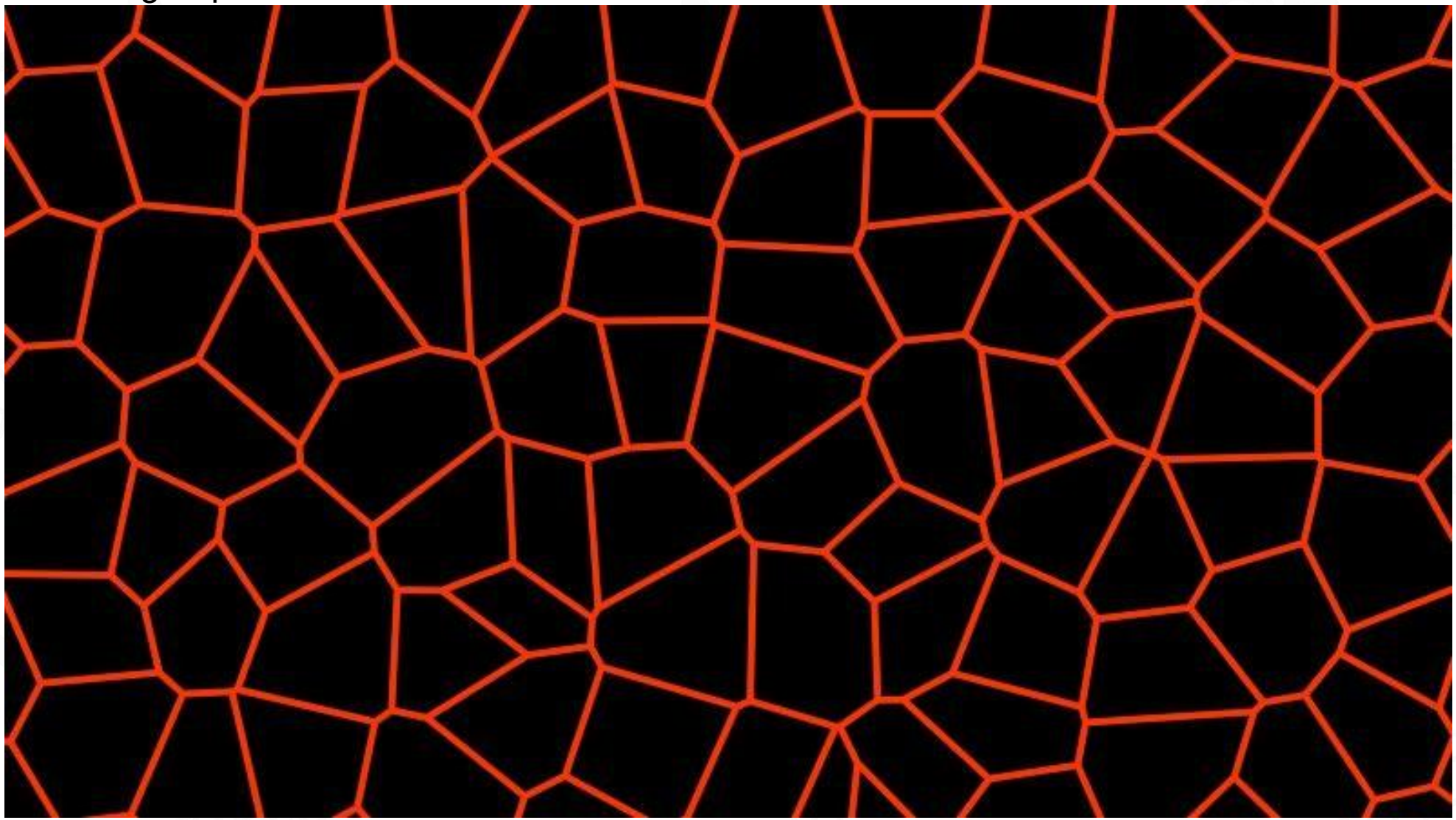
- The key idea is to alternate group boundaries from level to level by grouping different clusters.
- A boundary in one level becomes the interior in the next level
- Locked one level, unlocked the next



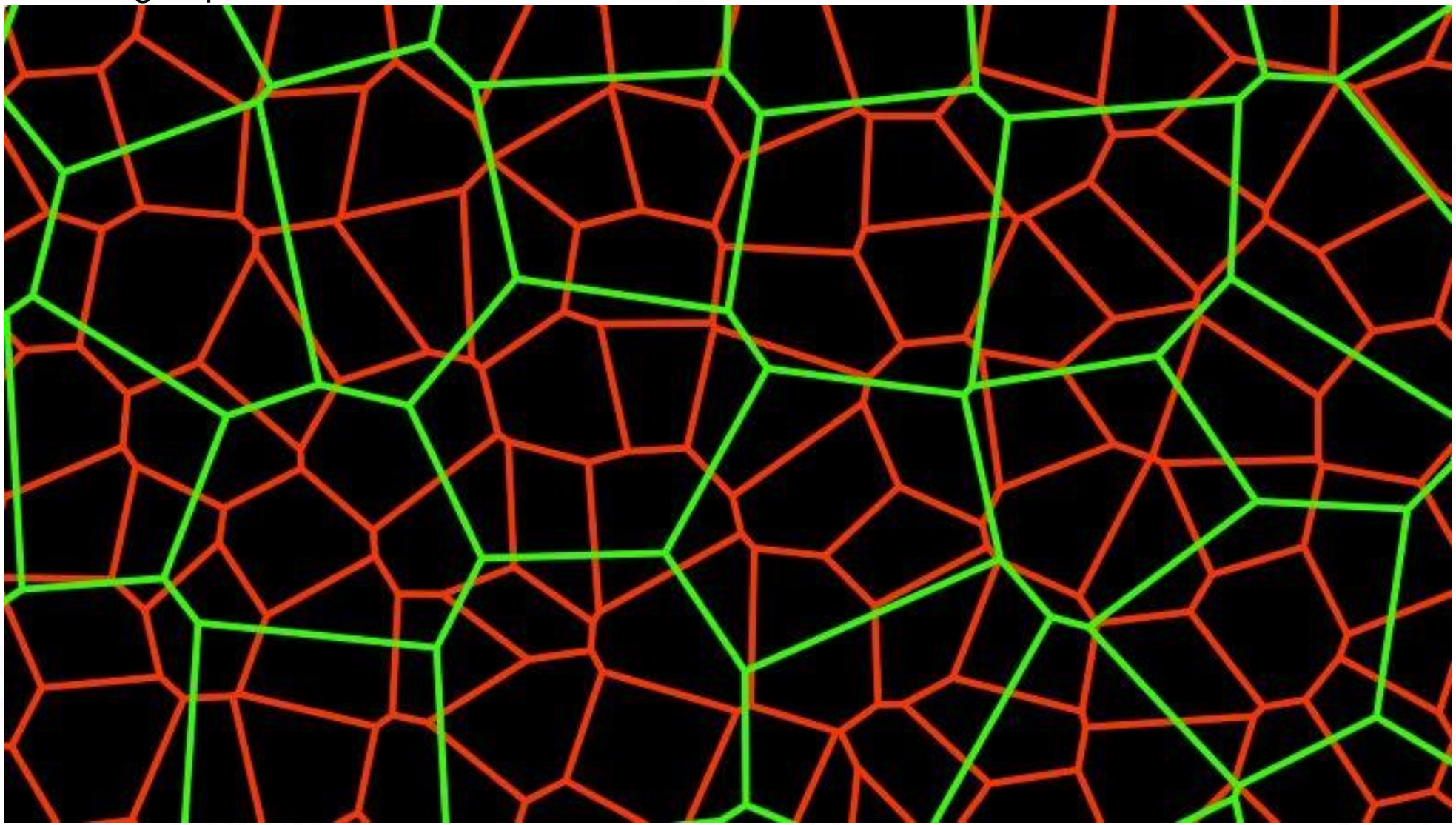




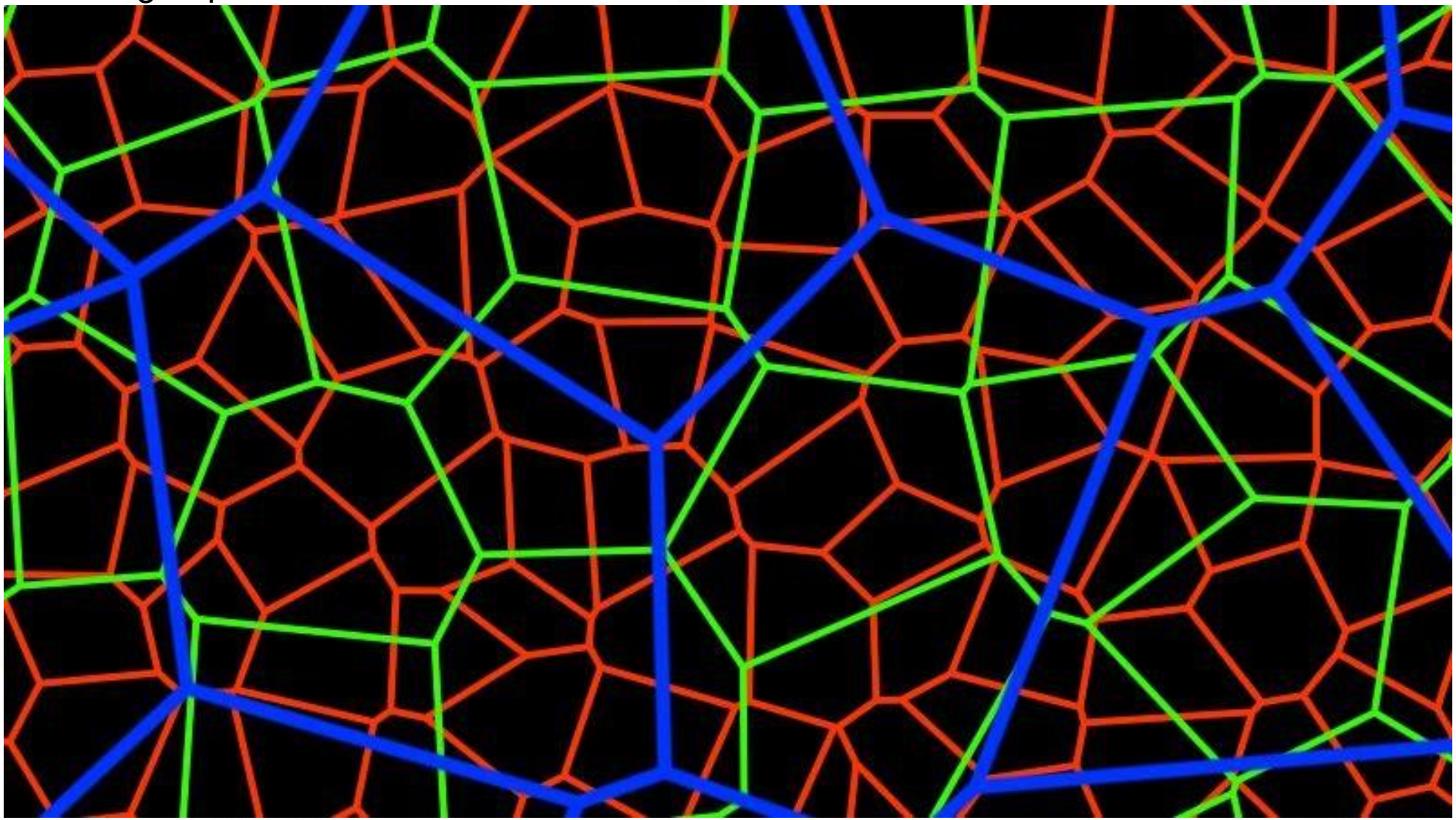
Cluster group boundaries for LoD0



Cluster group boundaries for LoD1

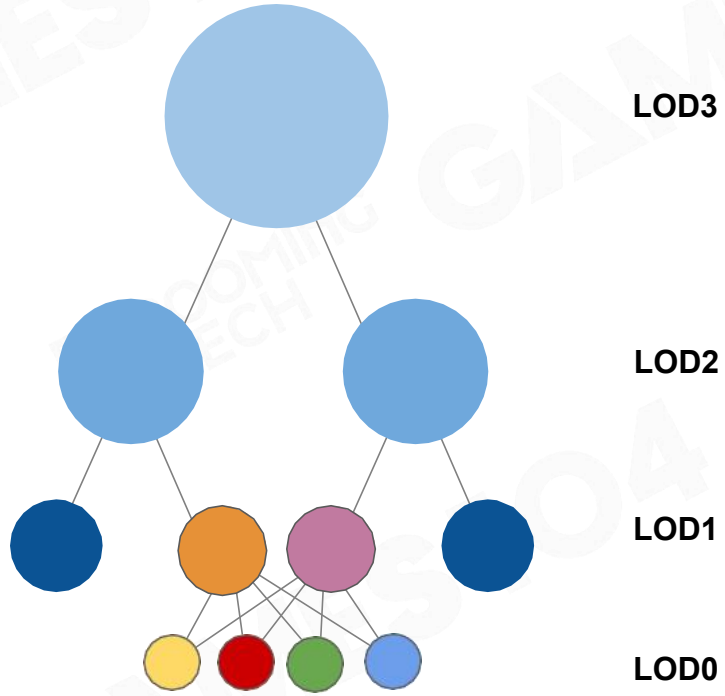


Cluster group boundaries for LoD2



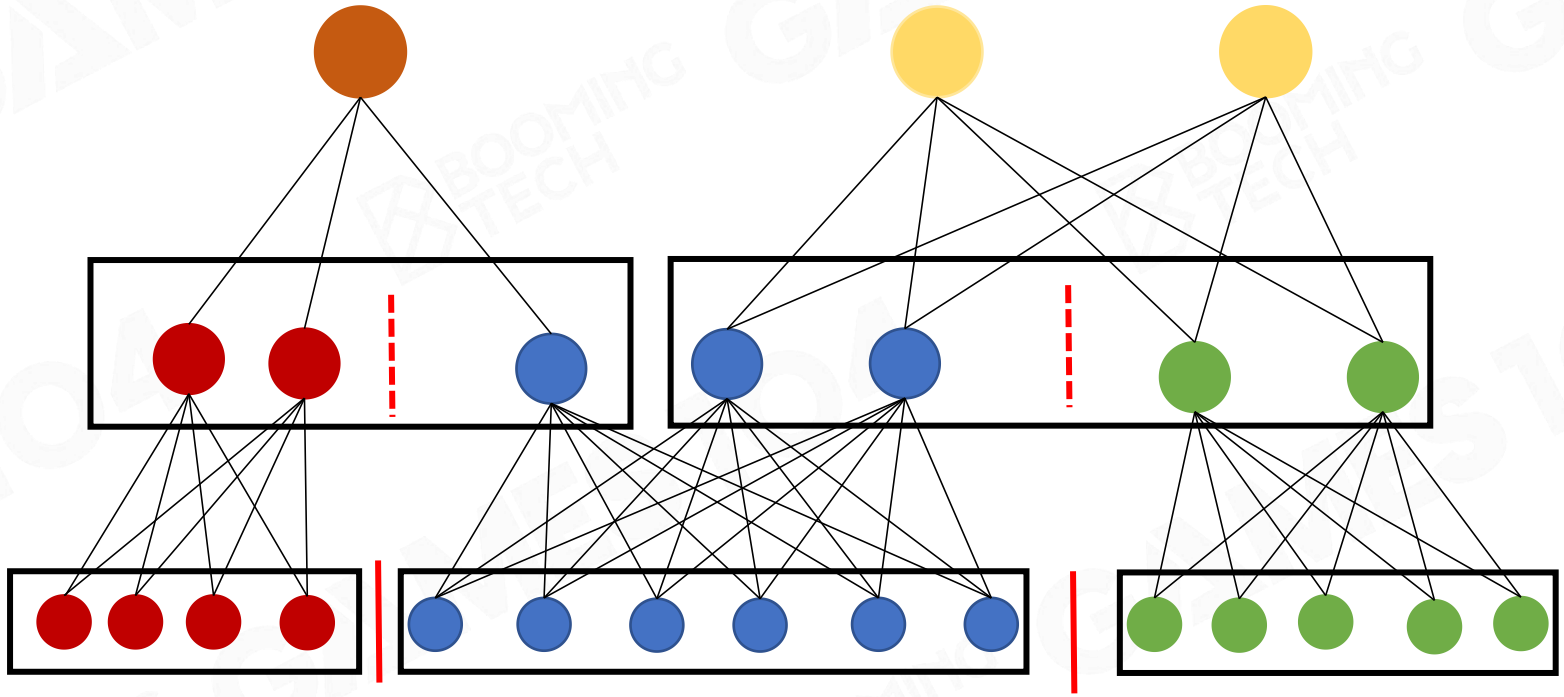
DAG for Cluster Groups

- **Merge** and **split** 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

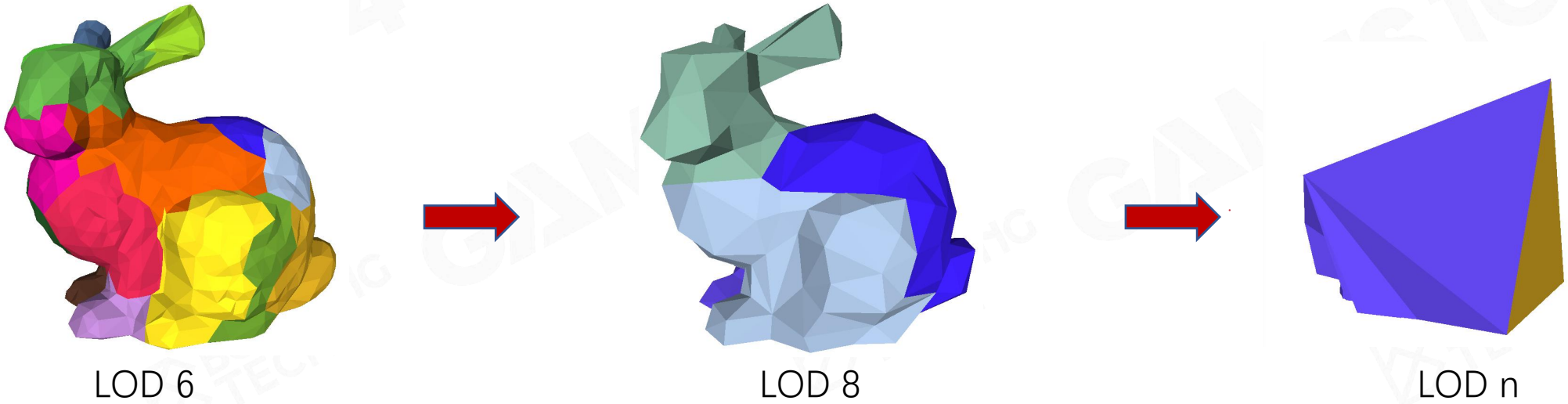


Why DAG, not Tree (Trap!)

Jungle of clusters, group and their links

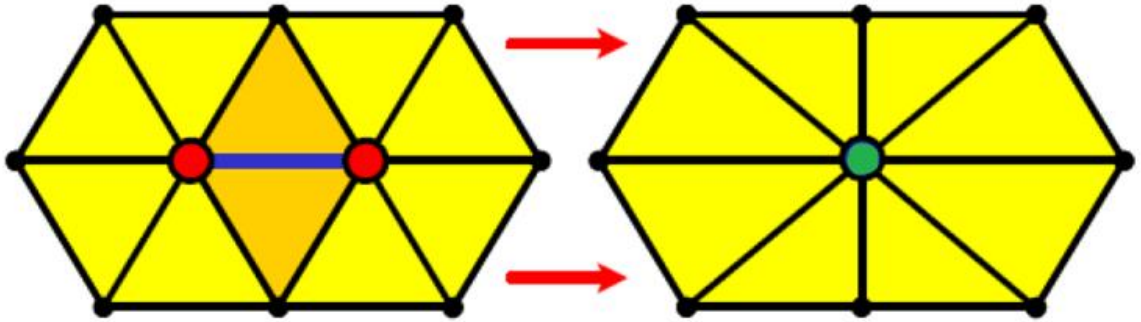


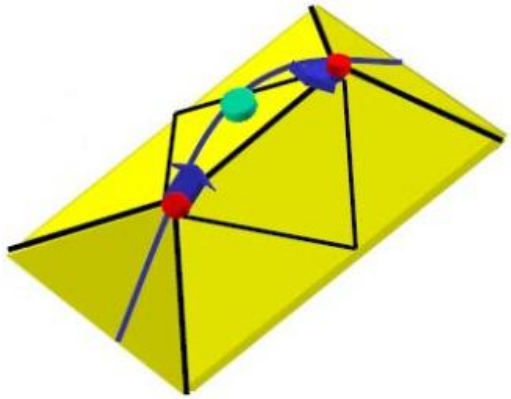
Let's Chop the Lovely Bunny



Detail of Simplification - QEM

We need to consider geometry, normal, color and UV





$$\mathbf{n}^{\mathsf{T}}\mathbf{v}+d=0$$

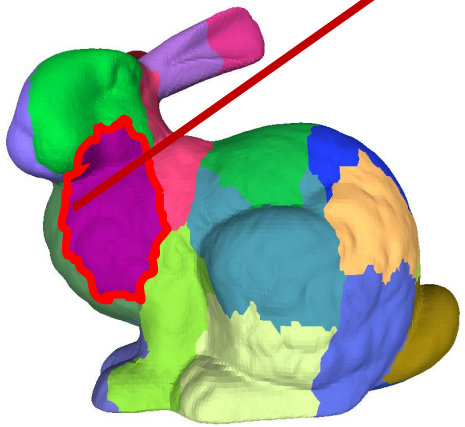
LODError

$$\mathbf{v}=[x\ y\ z]^{\mathsf{T}}$$

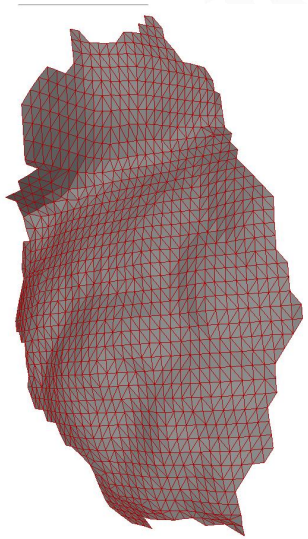
$$D^2=(\mathbf{n}^{\mathsf{T}}\mathbf{v}+d)^2=(\mathbf{v}^{\mathsf{T}}\mathbf{n}+d)(\mathbf{n}^{\mathsf{T}}\mathbf{v}+d)=\mathbf{v}^{\mathsf{T}}(\mathbf{n}\mathbf{n}^{\mathsf{T}})\mathbf{v}+2d\mathbf{n}^{\mathsf{T}}\mathbf{v}+d^2.$$

18 clusters (LOD_0)

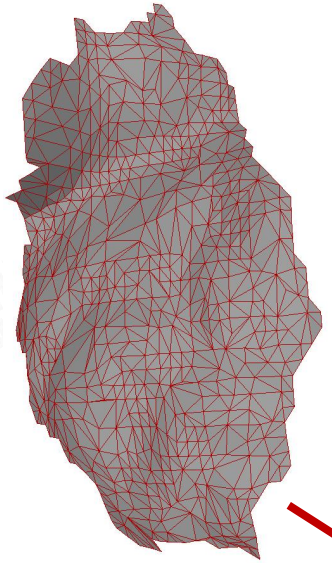
METIS



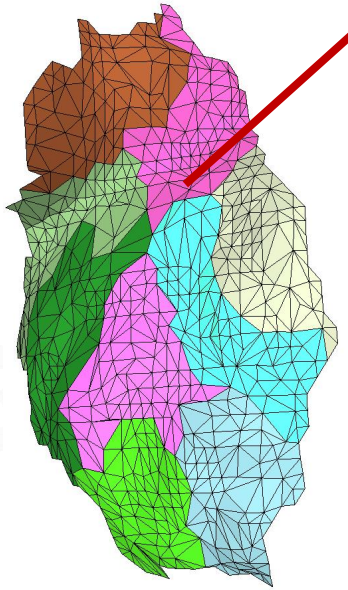
cluster group (LOD_0)

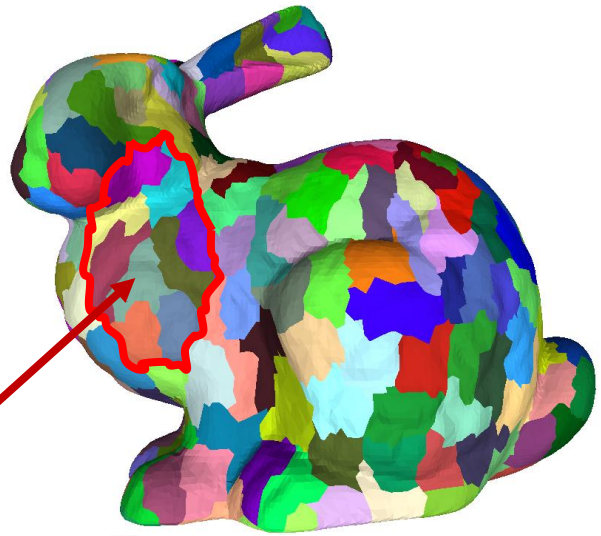


QEM



METIS



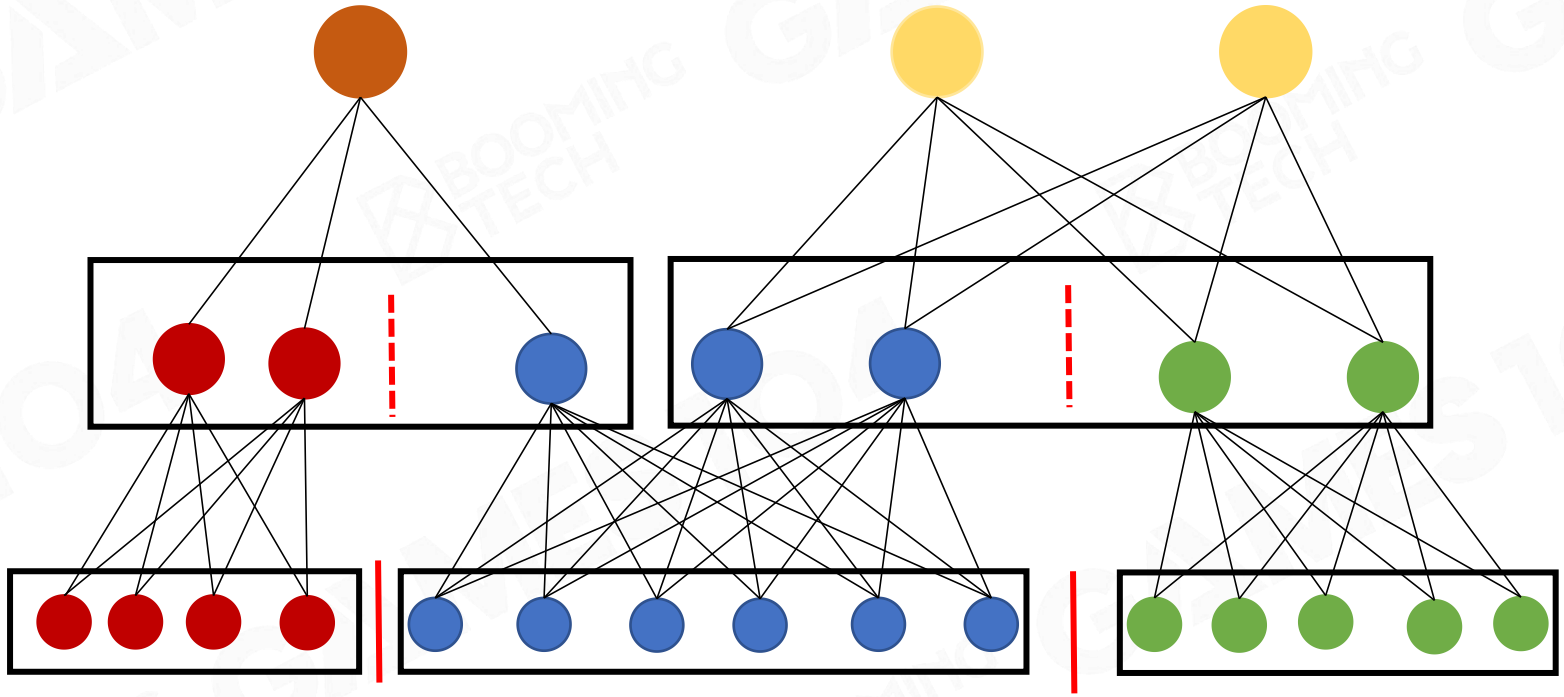


9 clusters (LOD_1)

Runtime LoD Selection

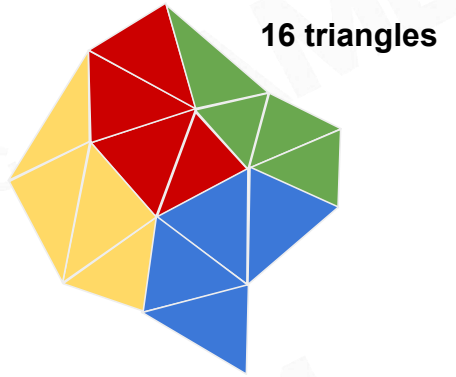
View-Dependent LoD Selection on DAG?

Group is faster than cluster, but DAG is still very complicated

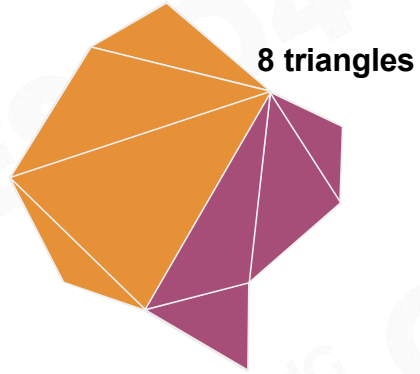


LOD Selection for Cluster Group

- Two submeshes with same boundary, but different LOD



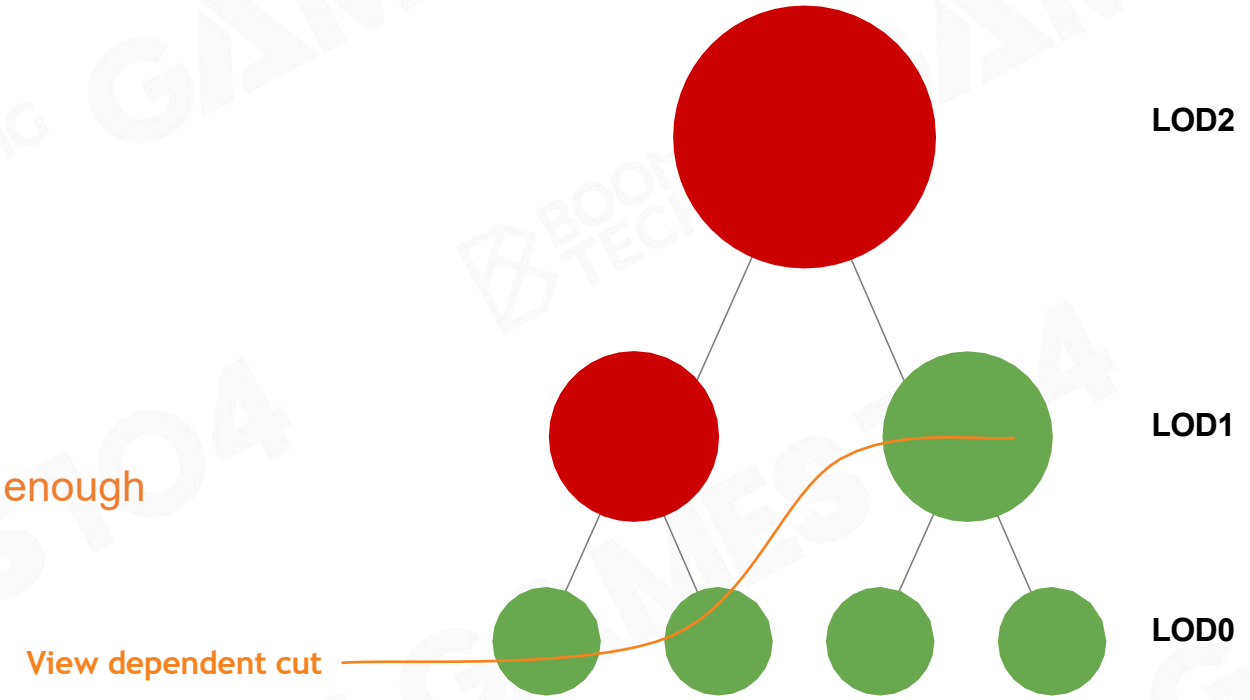
- Choose between them based on screen-space error
  - Error calculated by simplifier projected to screen
  - Corrected for distance and angle distortion at worst-case point in sphere bounds



- All clusters in group must make same LOD decision
  - How? Communicate? No!
  - Same input => same output

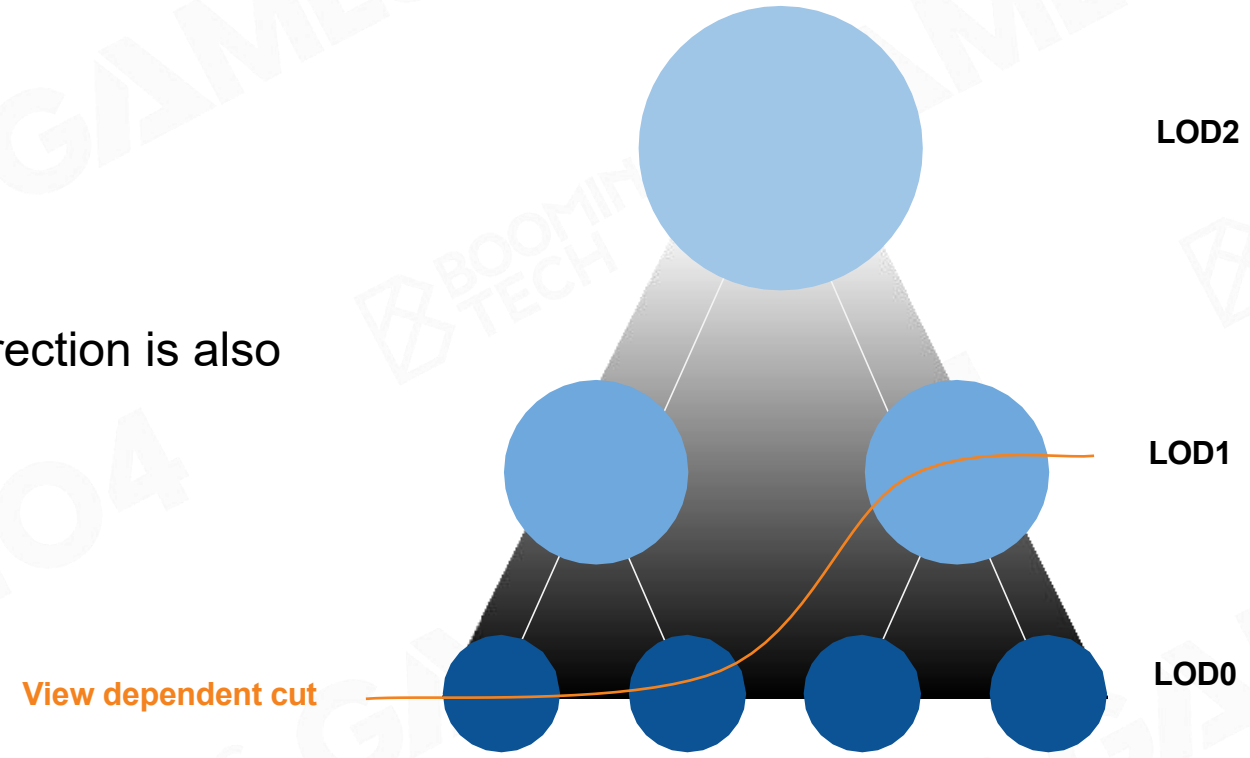
LOD Selection in Parallel

- 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!



LOD Selection in Parallel

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

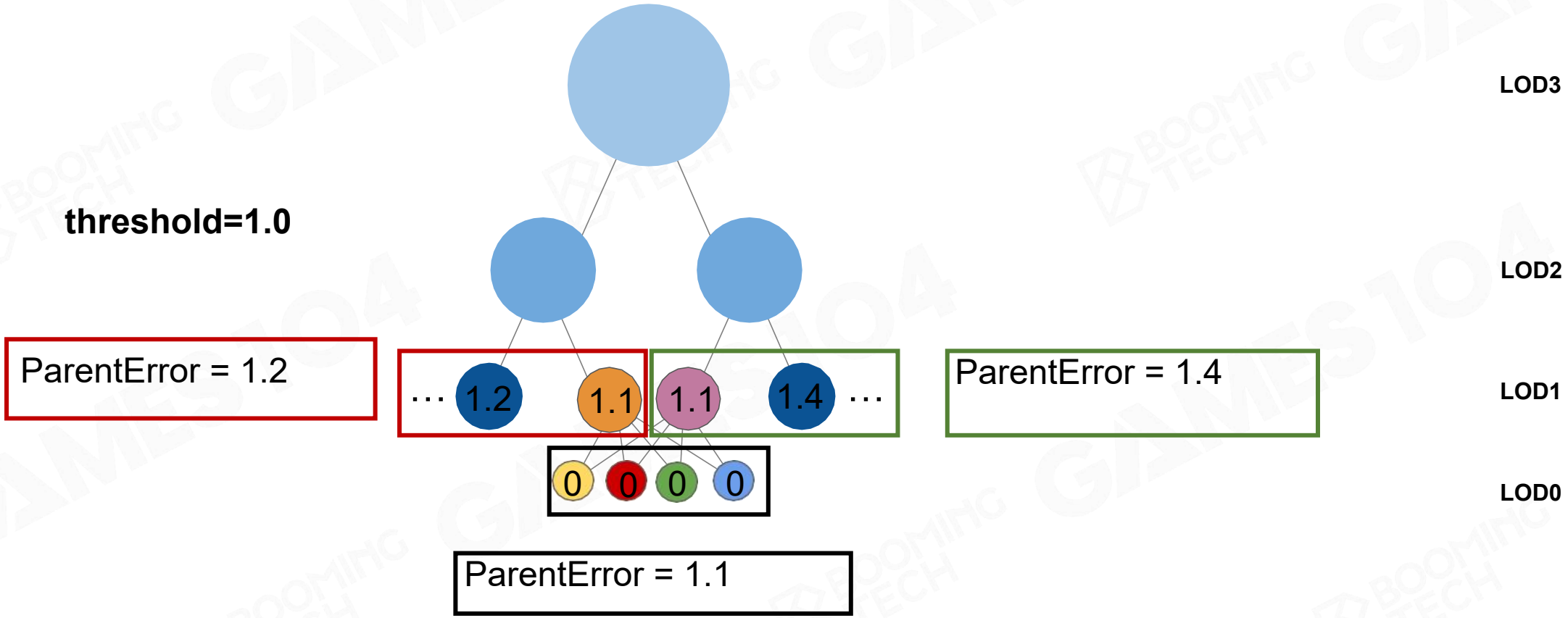


Core Equation of Parallel LoD Selection for Cluster Groups

- 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!

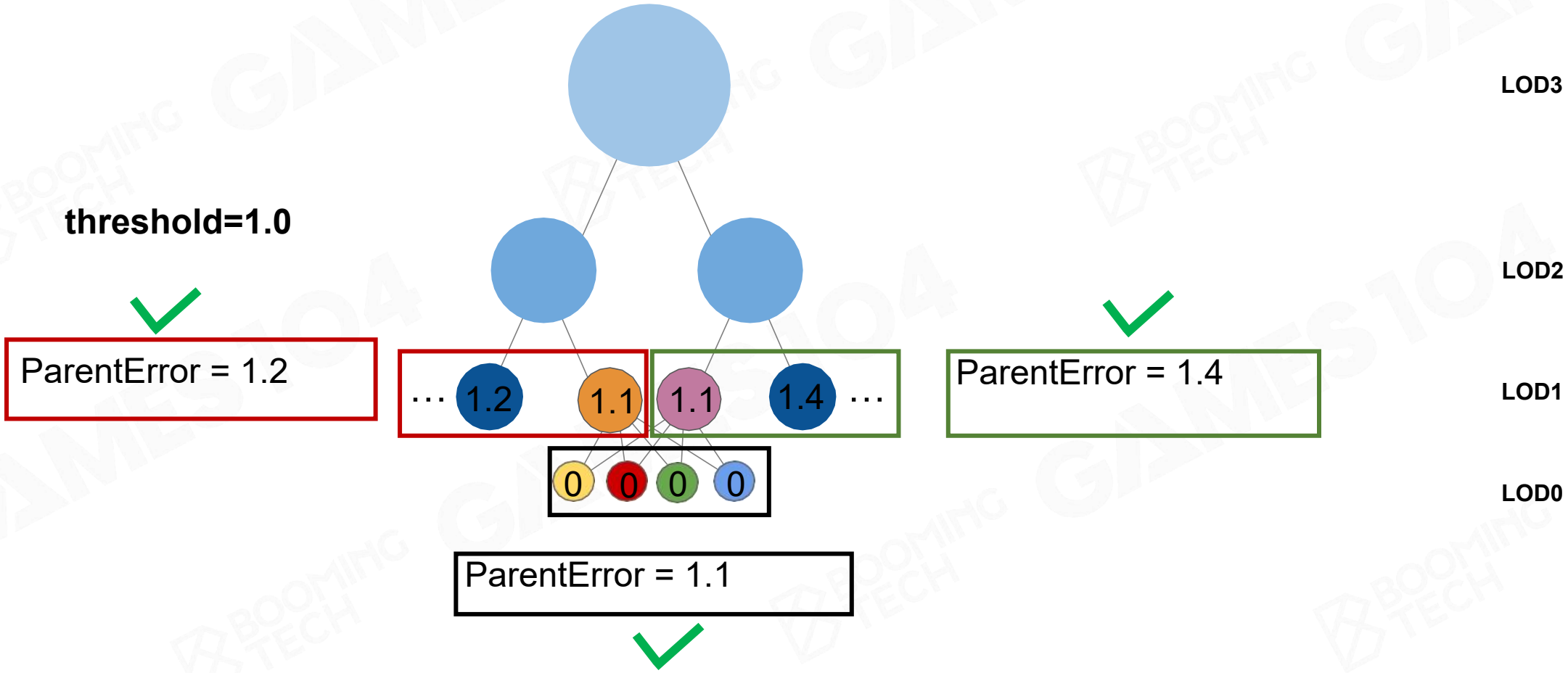
Isolated LoD Selection for Each Cluster Group (1/3)

- Render: ParentError > threshold && ClusterError <= threshold
- Cull: ParentError <= threshold || ClusterError > threshold



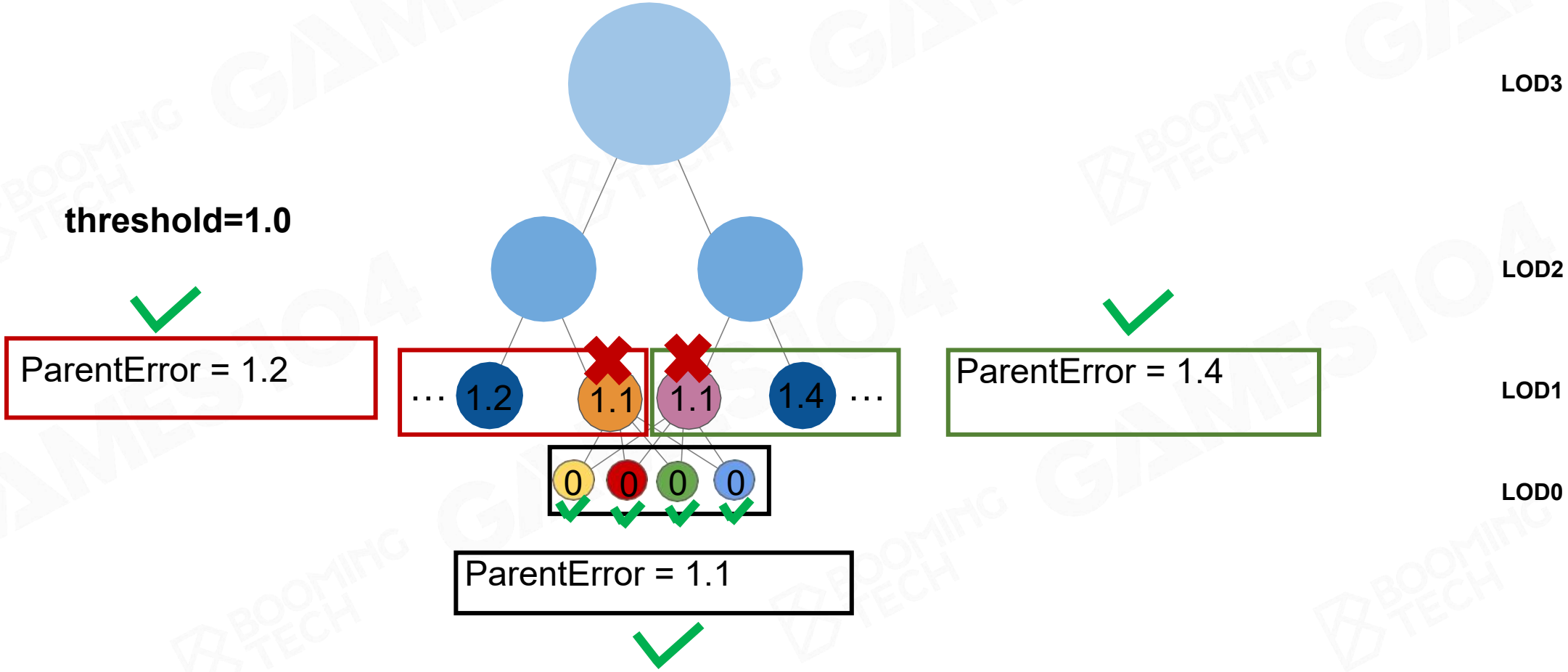
Isolated LoD Selection for Each Cluster Group (2/3)

- Render: **ParentError > threshold** && ClusterError <= threshold
- Cull: ParentError <= threshold || ClusterError > threshold



Isolated LoD Selection for Each Cluster Group (3/3)

- Render: ParentError > threshold && **ClusterError <= threshold**
- Cull: ParentError <= threshold || ClusterError > threshold



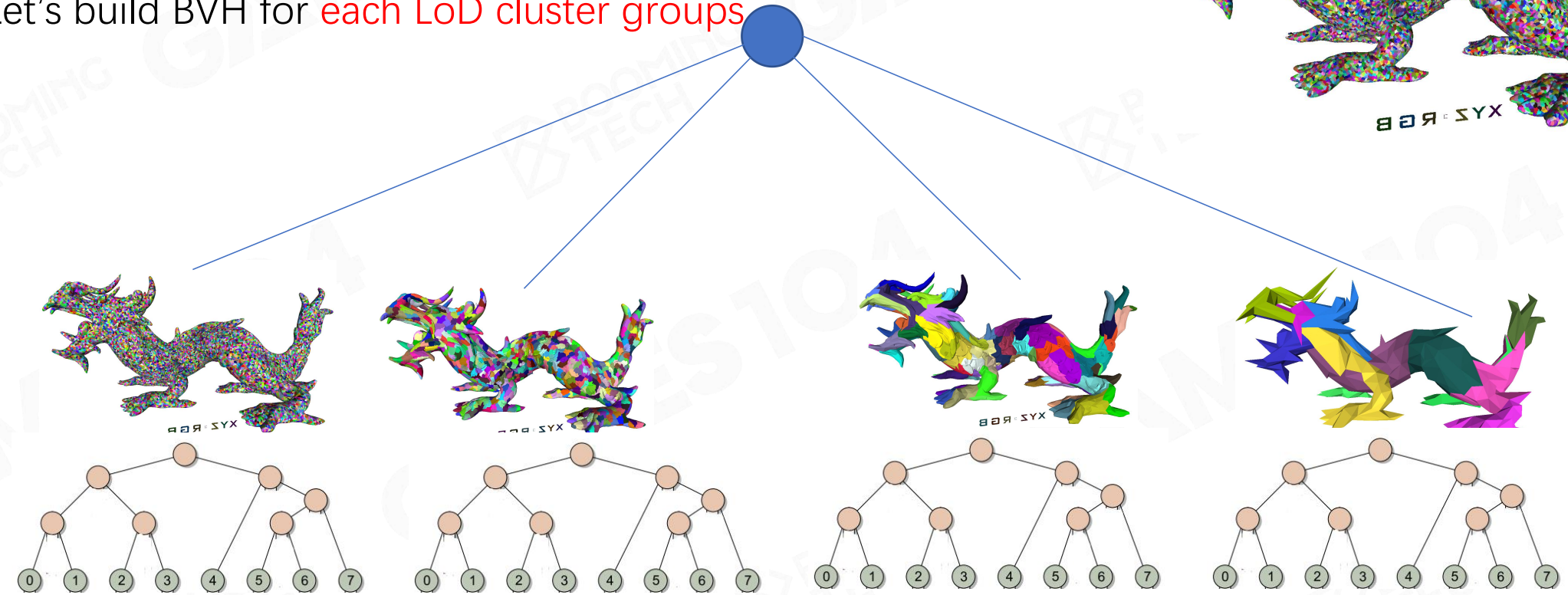
BVH Acceleration for LoD Selection

Really Bad Explanation of Why and How about BVH

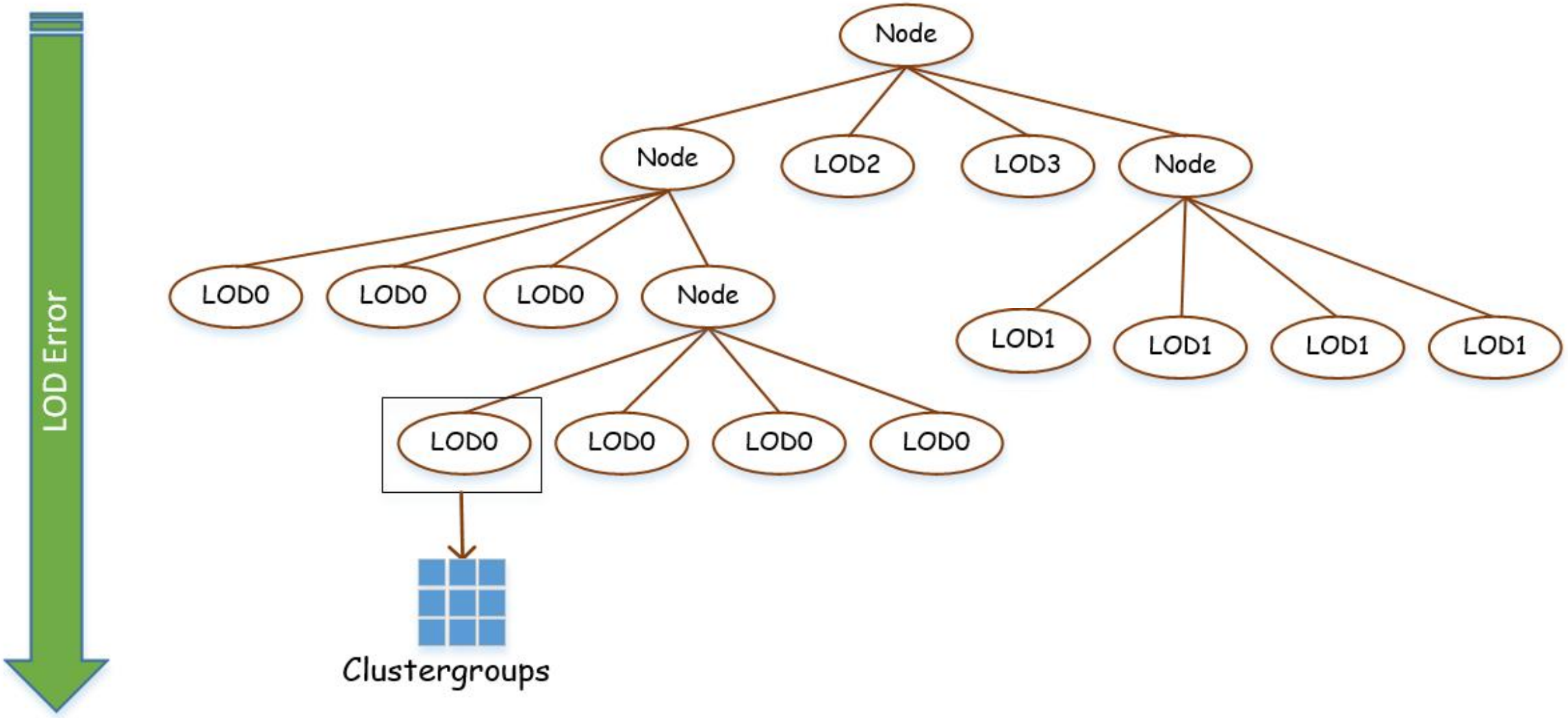
- BVH4
  - Max of children's ParentError
  - Internal node: 4 children nodes
  - Leaf node: List of clusters in group

Build BVH for Acceleration of LoD Selection

- 7,000,000 triangles will create 110,000 clusters
- Iterating all cluster/cluster groups is too slow
- Let's build BVH for each LoD cluster groups



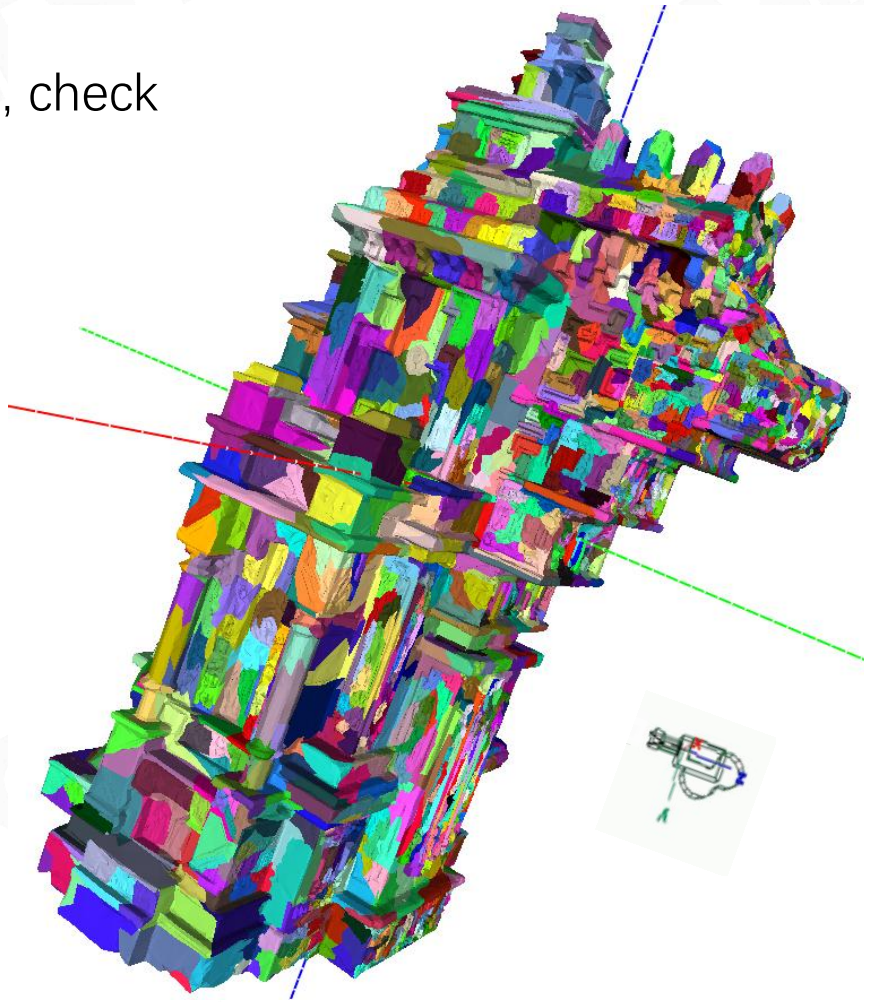
Balance BVH for 4 Nodes



Detail of BVH Acceleration

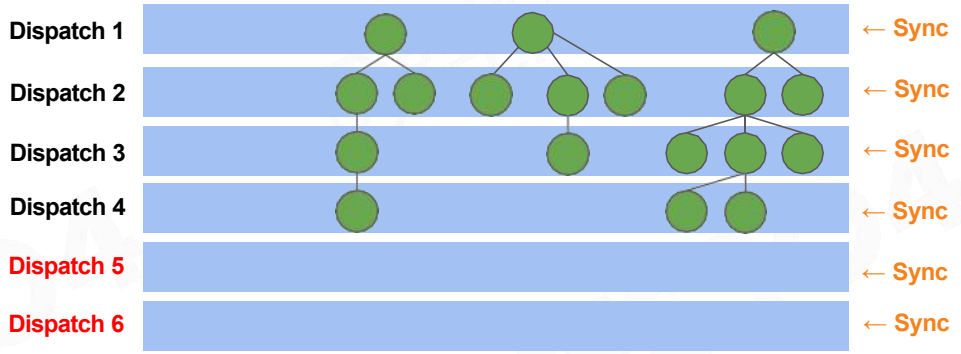
- total 110437 clusters,
- check bvh node = 107, check cluster = 4240,
- select cluster = 2175

6,400,000 tris -> 260,000 tris



Hierarchical Culling - Naive Approach

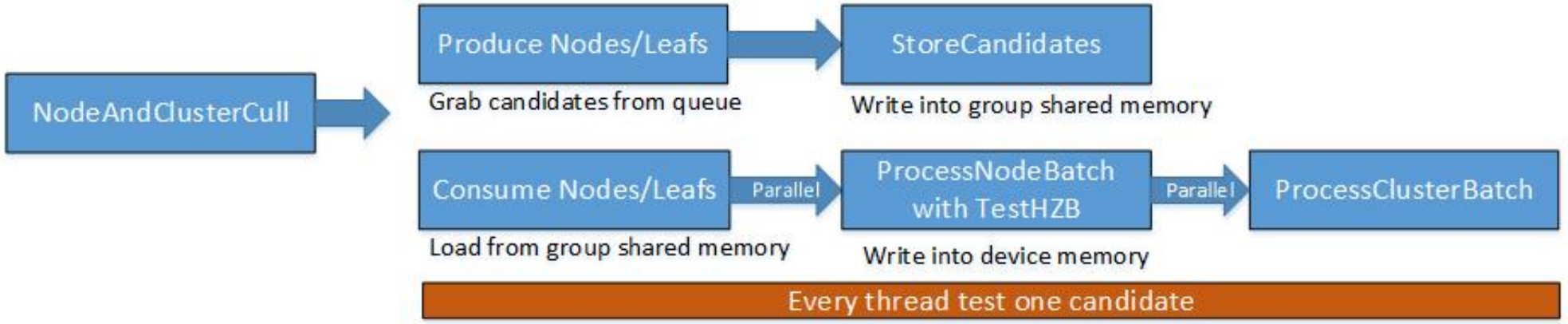
- Dependent DispatchIndirects
  - One per level
- Global synchronization
  - Wait for idle between every level
- Worst case # of levels
  - Empty dispatches at the end
- Can be mitigated by higher fanout
  - Wasteful for small/distant objects



Persistent Threads

- Ideally
  - Start on child as soon as parent finished
  - Spawn child threads directly from compute

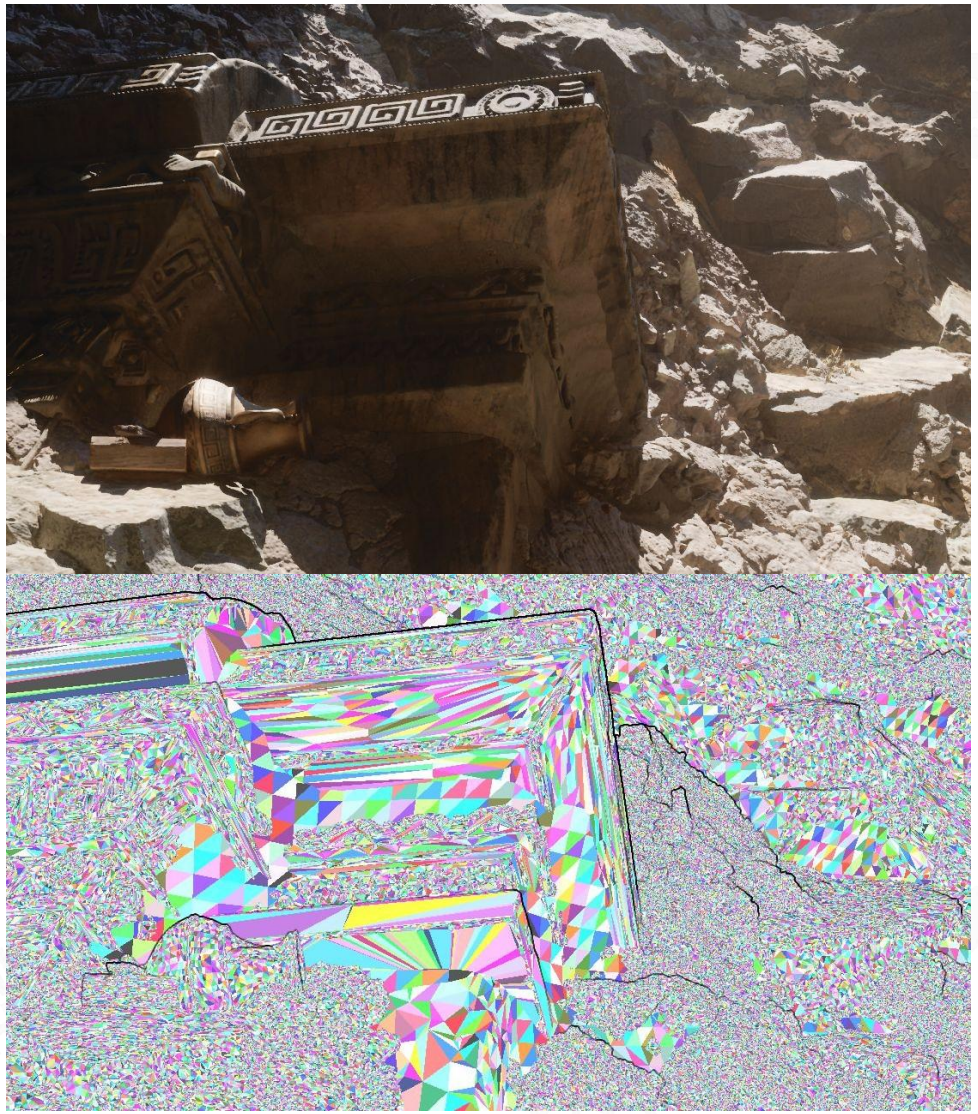
- Persistent threads model instead
  - Can't spawn new threads. Reuse them instead!
  - Manage our own job queue
  - Single dispatch with enough worker threads to fill GPU
  - Use simple multi-producer multi-consumer (MPMC) job-queue to communicate between threads



Nanite Rasterization

Pixel Scale Detail

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



Hardware Rasterization

- HW Rasterization unit is quad(2x2 pixels) for ddx and ddy
- Need help pixels(yellow) to form quads

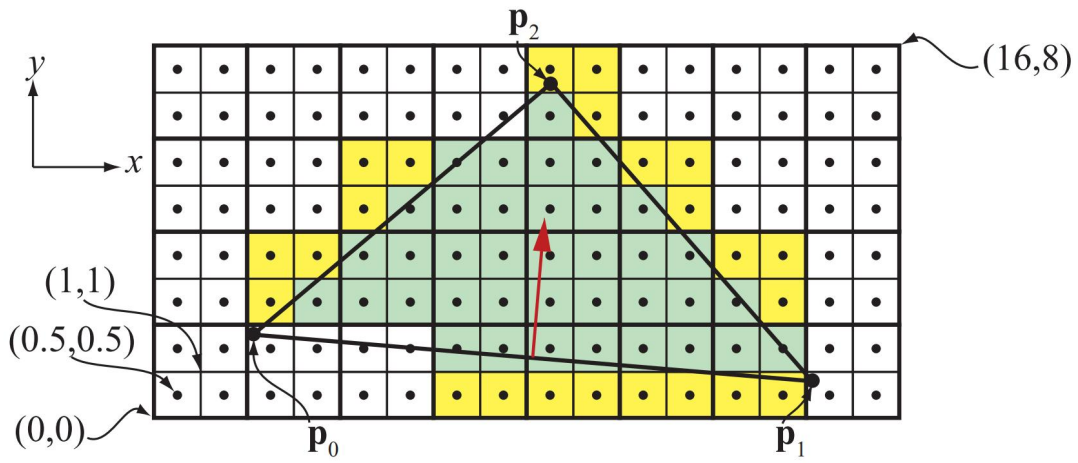


Figure 23.1. A triangle, with three two-dimensional vertices $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ in screen space. The size of the screen is 16×8 pixels. Notice that the center of a pixel (x, y) is $(x + 0.5, y + 0.5)$. The normal vector (scaled in length by 0.25) for the bottom edge is shown in red. Only the green pixels are inside the triangle. Helper pixels, in yellow, belong to quads (2×2 pixels) where at least one pixel is considered inside, and where the helper pixel's sample point (center) is outside the triangle. Helper pixels are needed to compute derivatives using finite differences.

From Real-time rendering Chapter 23 Graphics Hardware

Hardware Rasterization

- Use 4x4 tiled traversal to accelerate

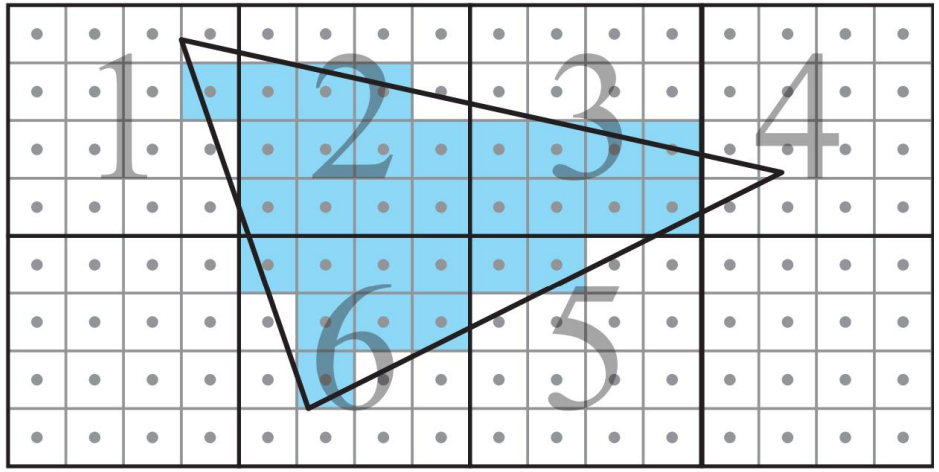
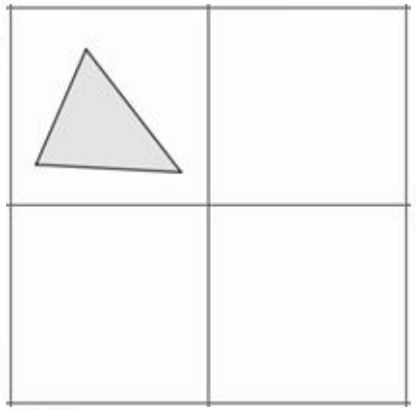


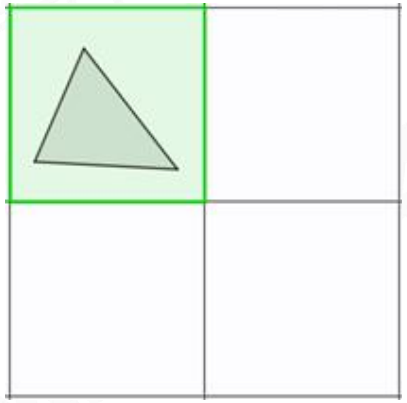
Figure 23.3. A possible traversal order when using tiled traversal with 4×4 pixel tiles. Traversal starts at the top left in this example, and continues to the right. Each of the top tiles overlap with the triangle, though the top right tile has no pixels it. Traversal continues to the tile directly below, which is completely outside, and so no per-pixel inside tests are needed there. Traversal then continues to the left, and the following two tiles are found to overlap the triangle, while the bottom left tile does not.

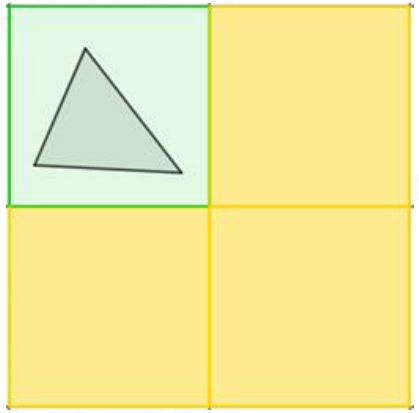
From Real-time rendering Chapter 23 Graphics Hardware

Hardware Rasterization

- A lot of wasting for small triangle
- tiled traversal stage is useless
- quad generate 4x pixels than its really covered







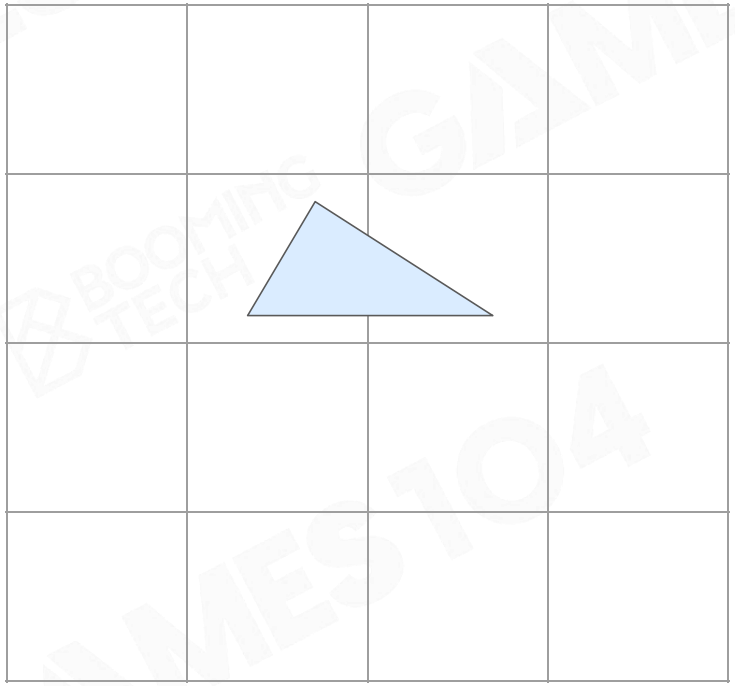
Software Rasterization for Tiny Triangles

- 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

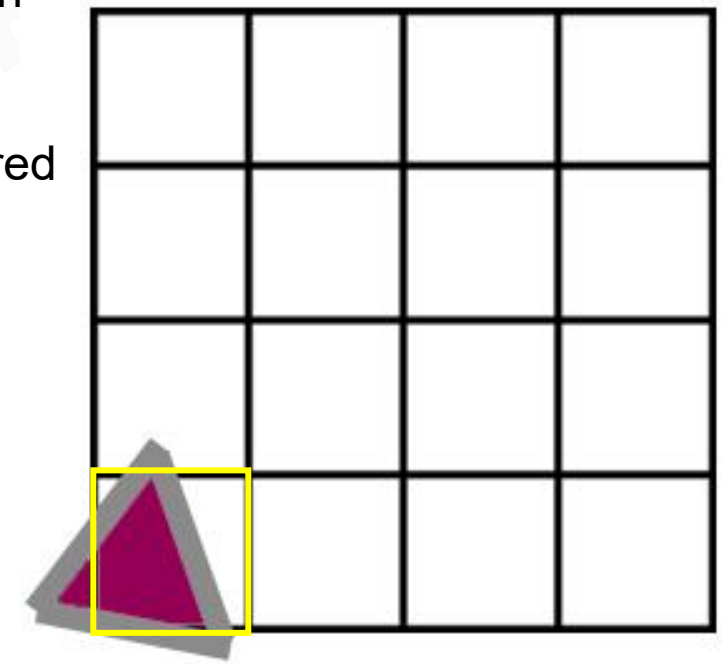
- Can we beat the HW rasterizer in SW?



3x faster!

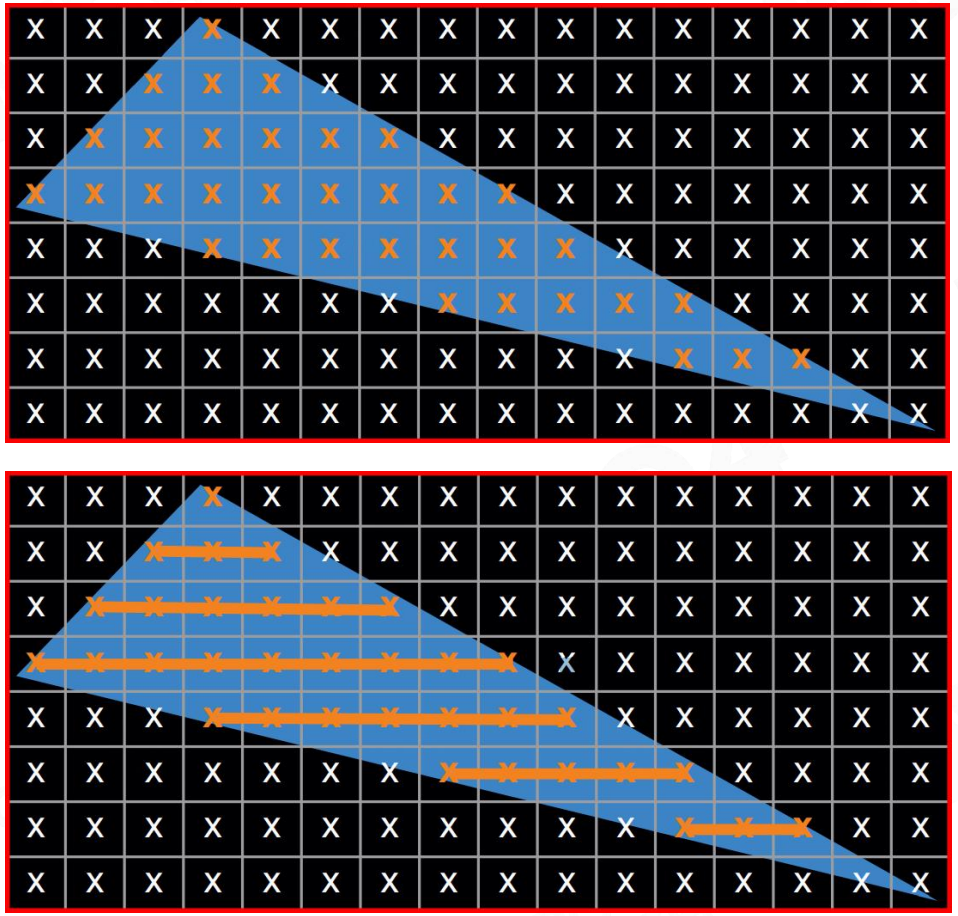
Nanite – Rasterization

- Only rasterize 1 pixel when the triangle size smaller than 1 pixel in Shader function
- We will save 3 pixels compute resources if the triangle only covered in 1 pixel
- Reconstruct derivatives for ddx/ddy



Scanline Software Rasterizer

- Per-cluster based rasterization selection
  - All edges of cluster <18 pixels are SW rasterized
- Iterate over the rect tests a lot of pixels
- Best case half are covered
- Worst case none are
- Scanline method is a choice



How To Do Depth Test?

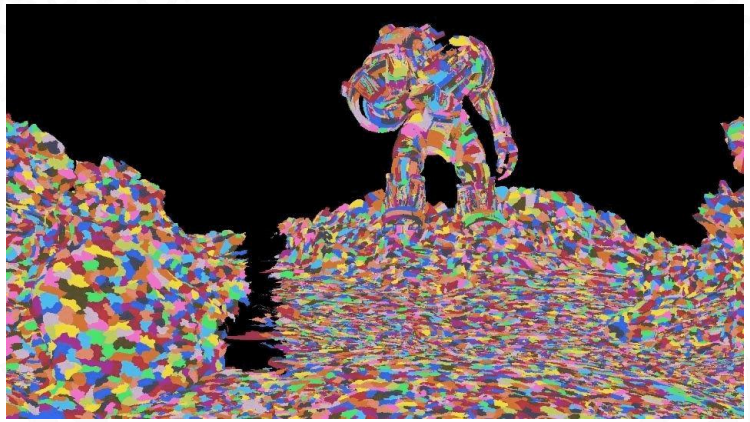
- Don't have ROP or depth test hardware
- Need Z-buffering
  - Can't serialize at tiles
  - Many tris may be in parallel for single tile or even single pixel
- Use 64 bit atomics!
- InterlockedMax
  - Visibility buffer shows its true power

32	25	7
Depth	Visible cluster index	Triangle index

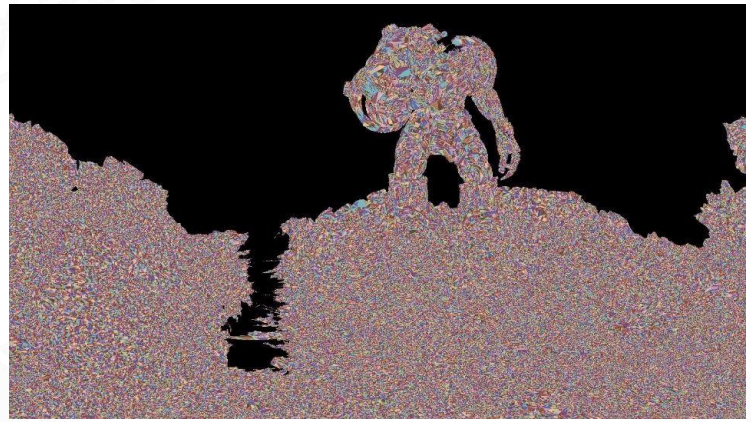
Nanite Visibility Buffer

NumberBits	32	25	7
Type	Depth	Visible cluster index	Triangle index

Depth



Clusters



Triangles

Nanite Visibility Buffer

- Write geometry data to screen
  - Depth : InstanceID : TriangleID

32	25	7
Depth	Visible cluster index	Triangle index

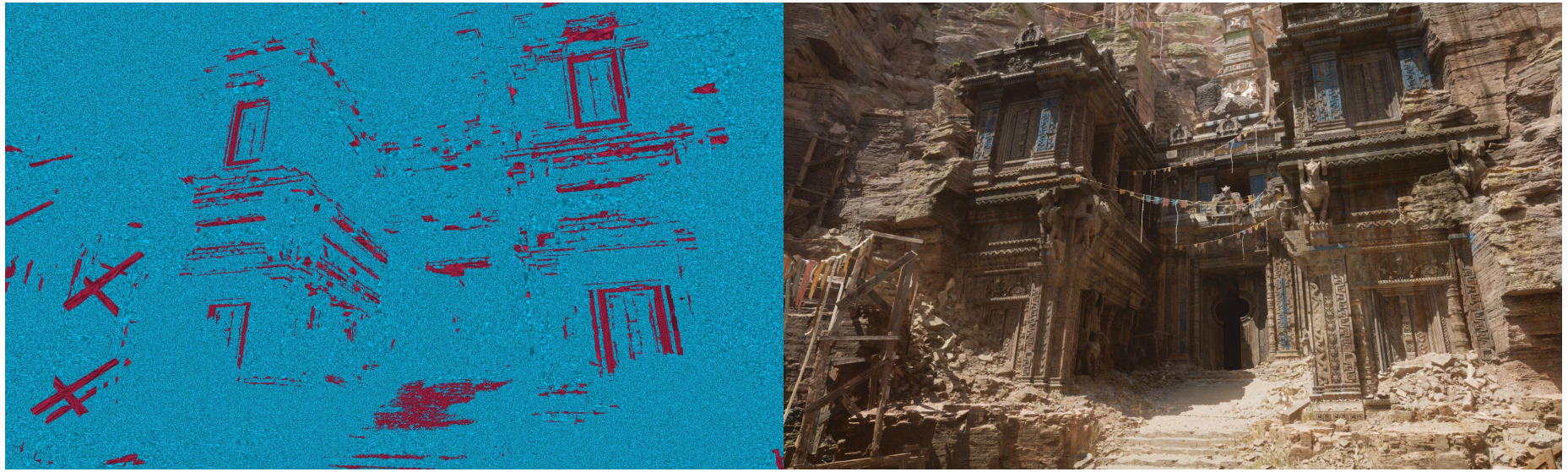
- Material shader per pixel:
  - Load VisBuffer
  - Load instance transform
  - Load 3 vert indexes
  - Load 3 positions
  - Transform positions to screen
  - Derive barycentric coordinates for pixel
  - Load and lerp attributes

Nanite 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
- Draw all opaque geometry with 1 draw
  - Completely GPU driven
  - Not just depth prepass
  - Rasterize triangles once per view

Hardware Rasterization

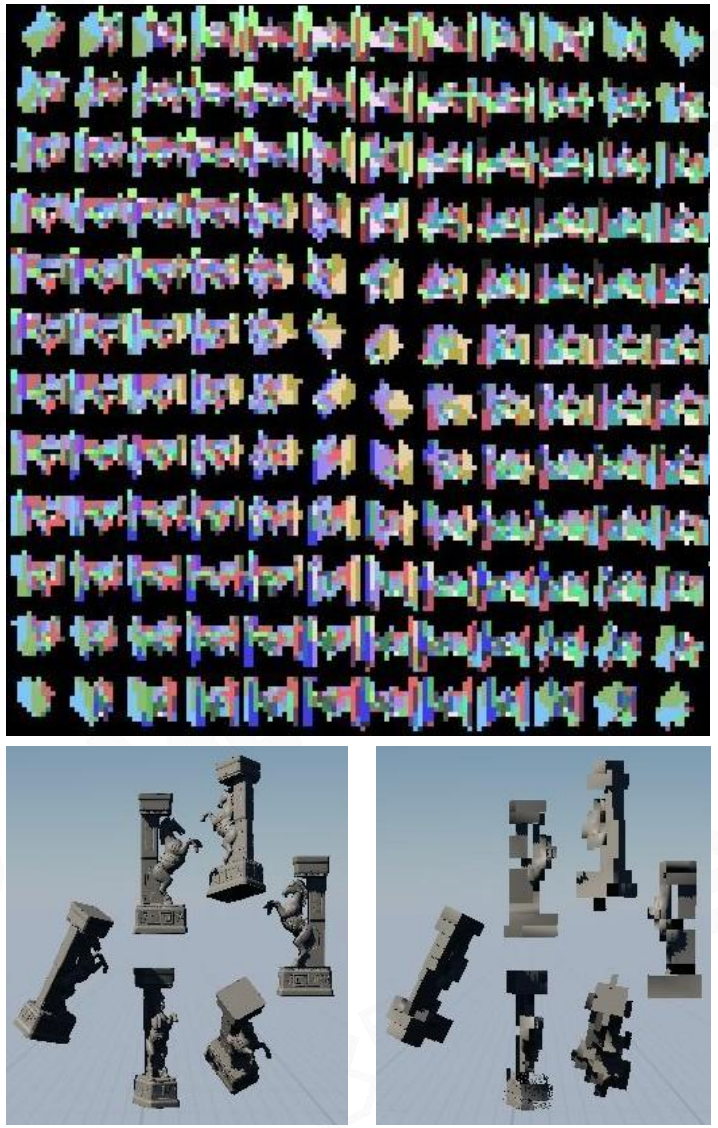
- What about big triangles?
- Use HW rasterizer
- Choose SW or HW per cluster
- Also uses 64b atomic writes to UAV



Blue: software rasterized, Red: Hardware rasterized

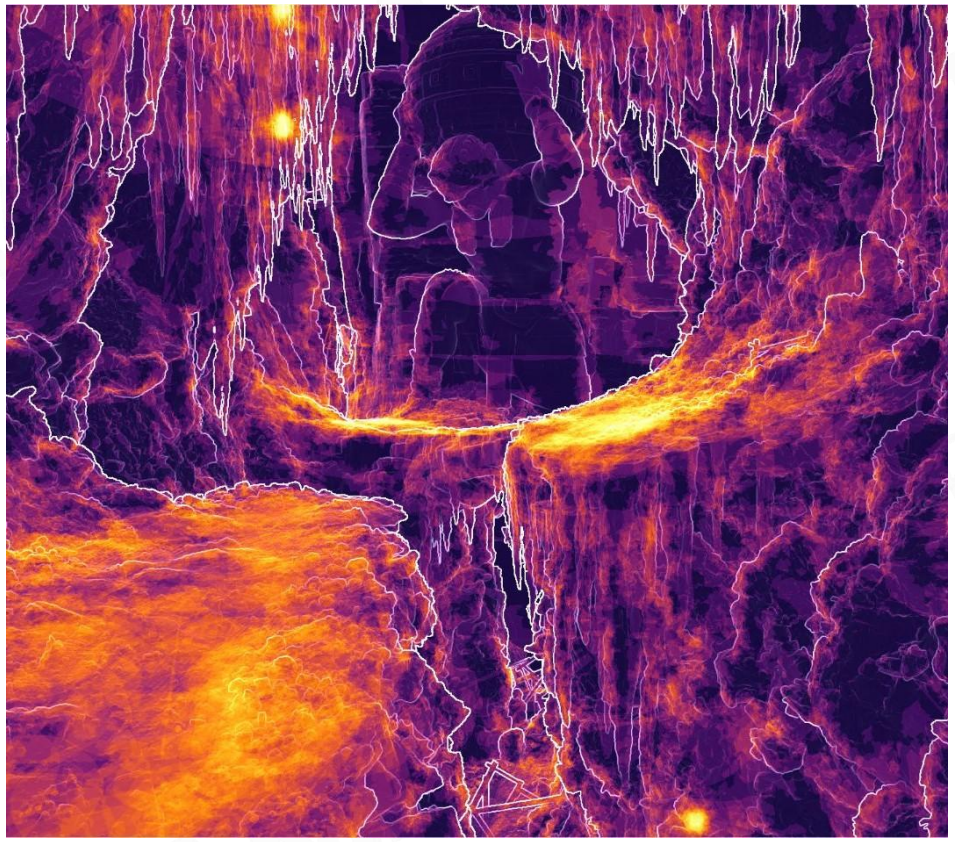
Imposters for Tiny Instances

- 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



Rasterizer Overdraw

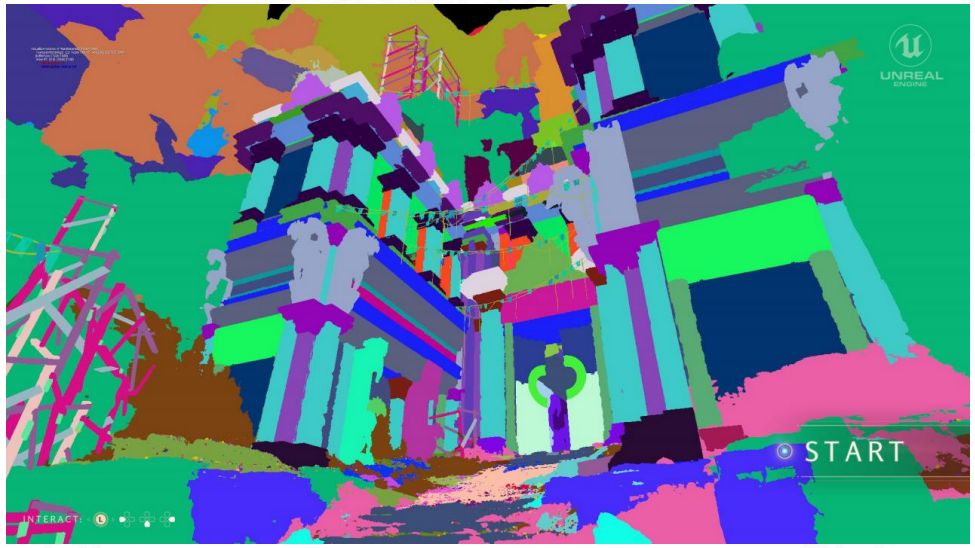
- No per triangle culling
- No hardware HiZ culling pixels
- Our software HZB is from previous frame
  - Culls clusters not pixels
  - Resolution based on cluster screen size
- Excessive overdraw from:
  - Large clusters
  - Overlapping clusters
  - Aggregates
  - Fast motion
- Overdraw expense
  - Small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



Nanite Deferred Material

Deferred Material

- Nanite want to support full artist created pixel shaders
- In theory, all materials could be applied in a single pass, but there are complexities and inefficiencies there



Different color blocks indicate different materials

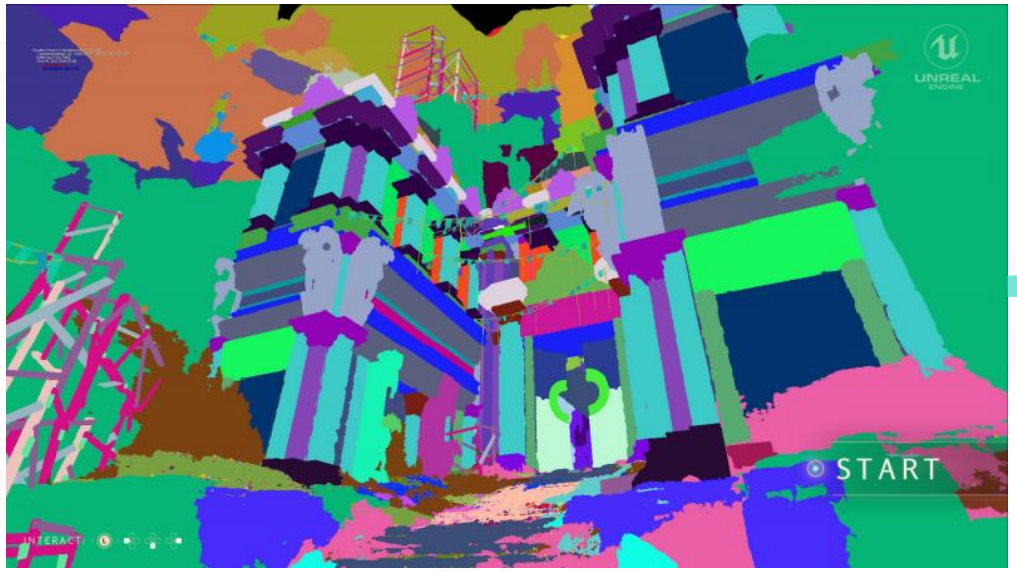
Material Shading

- **Common method**
  - Draw a full screen quad per unique material
  - Skip pixels not matching this material

- **Disadvantages**
  - CPU unaware if some materials have no visible pixels (unfortunate side effect of GPU driven)
  - So unnecessary drawing instructions will be committed

Shading Efficiency

- **Hardware depth test!**
  - Convert material ID to depth value

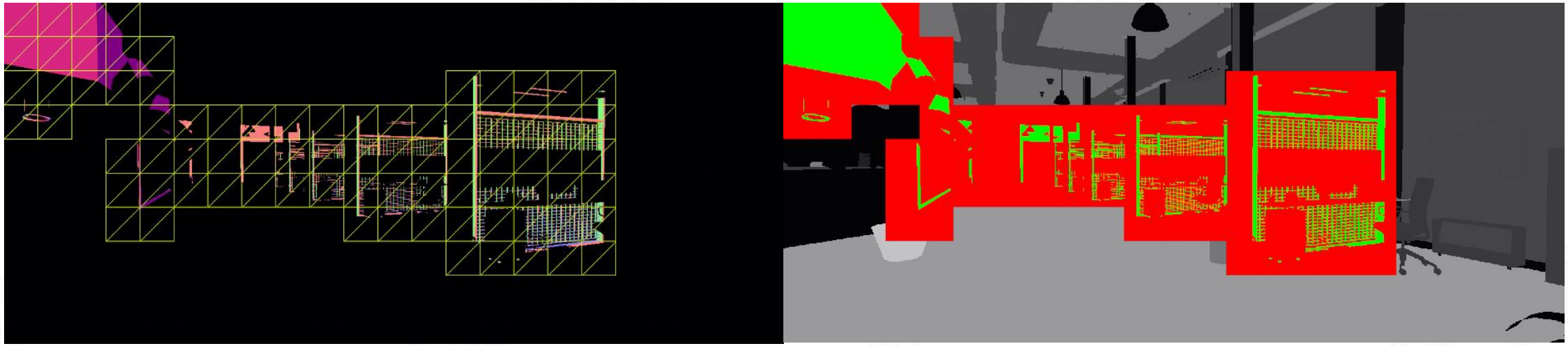


Shading

- Then draw a full screen quad and set depth test function to "equal", so unmatched pixels will be discarded
- But full screen quad **is not necessary and can be improved**!

Material Sorting with Tile-Based Rendering

- We can do a screen tile material classification
- For a certain material, exclude tiles that do not contain this material



Specific material tiles and depth test result (green: pass depth test, red: failed in depth test)

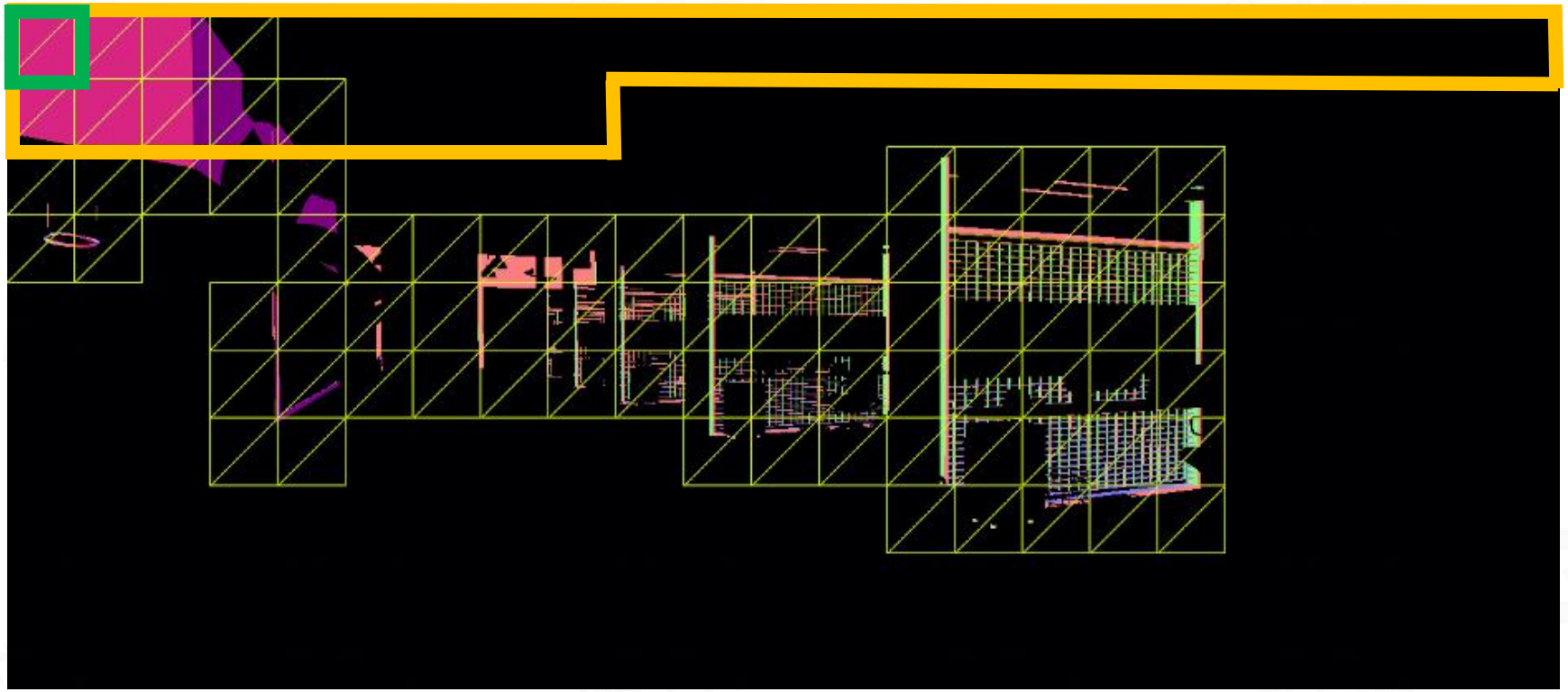
Material Classify

32 tiles in 1 tile group

For each material, use 32 bits to indicate the presence in a tile group

64x64 pixels 1 tile

16x16 threads for 1 tile



Material Classify - Material Tile Remap Table

- Finally forms a material and tile remap table
  - Get the number of tiles based on the screen resolution and pack 32 tiles into a group
  - 'MaterialRemapCount' means the number of groups
  - Record the tiles in which a material is located by marking it by bit
  - This table can be used to calculate the tile position to render to

Material Tile Remap Table

Material Slot \Tile Group	0	1	2	3	...	MaterialRemap Count-1
0	<32 bits>					
1						
...						
MaterialSlotCount-1						

Deferred Material Overall Process

- Generate material resolve texture
- Generate material depth texture
- Classify screen tile materials
- Generate G-Buffer
  - This will be output to the g-buffer to match with the rest of the pipeline
  - Commit drawing commands per material

```
void UnpackMaterialResolve(
    uint Packed,
    out bool IsNanitePixel,
    out bool IsDecalReceiver,
    out uint MaterialSlot)
{
    IsNanitePixel   = BitFieldExtractU32(Packed,  1,  0) != 0;
    MaterialSlot    = BitFieldExtractU32(Packed, 14,  1);
    IsDecalReceiver = BitFieldExtractU32(Packed,  1, 15) != 0;
}
```

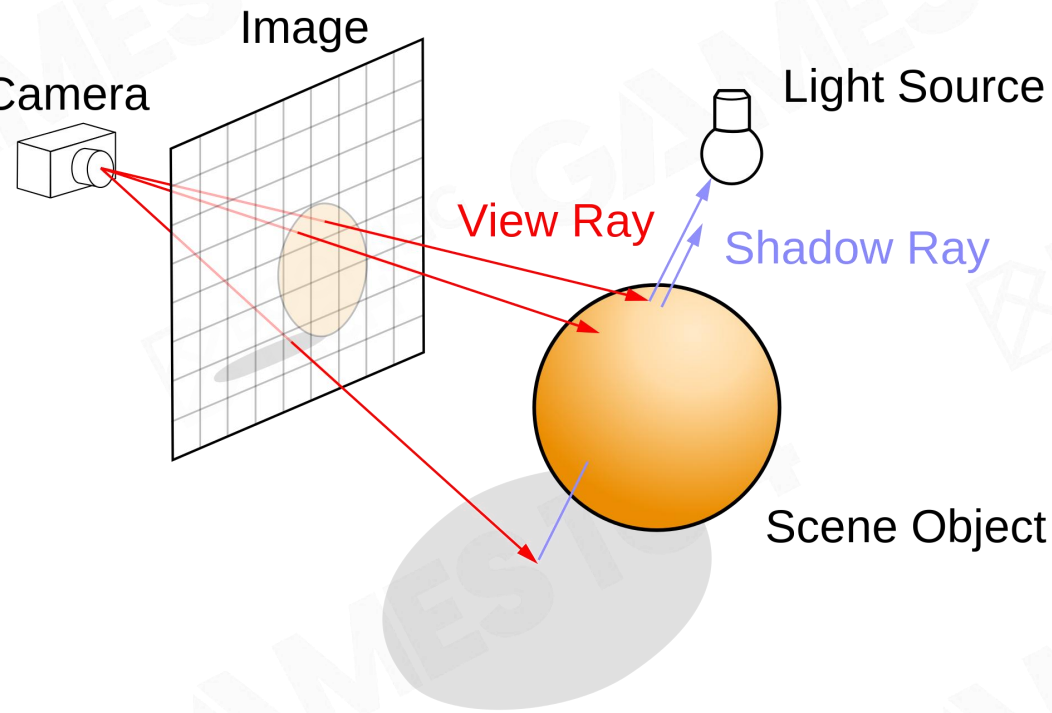
MaterialSlot is the material index in material id array


Shadows

Micropoly Level Detail for Shadows

Nanite Shadows - Ray Trace?

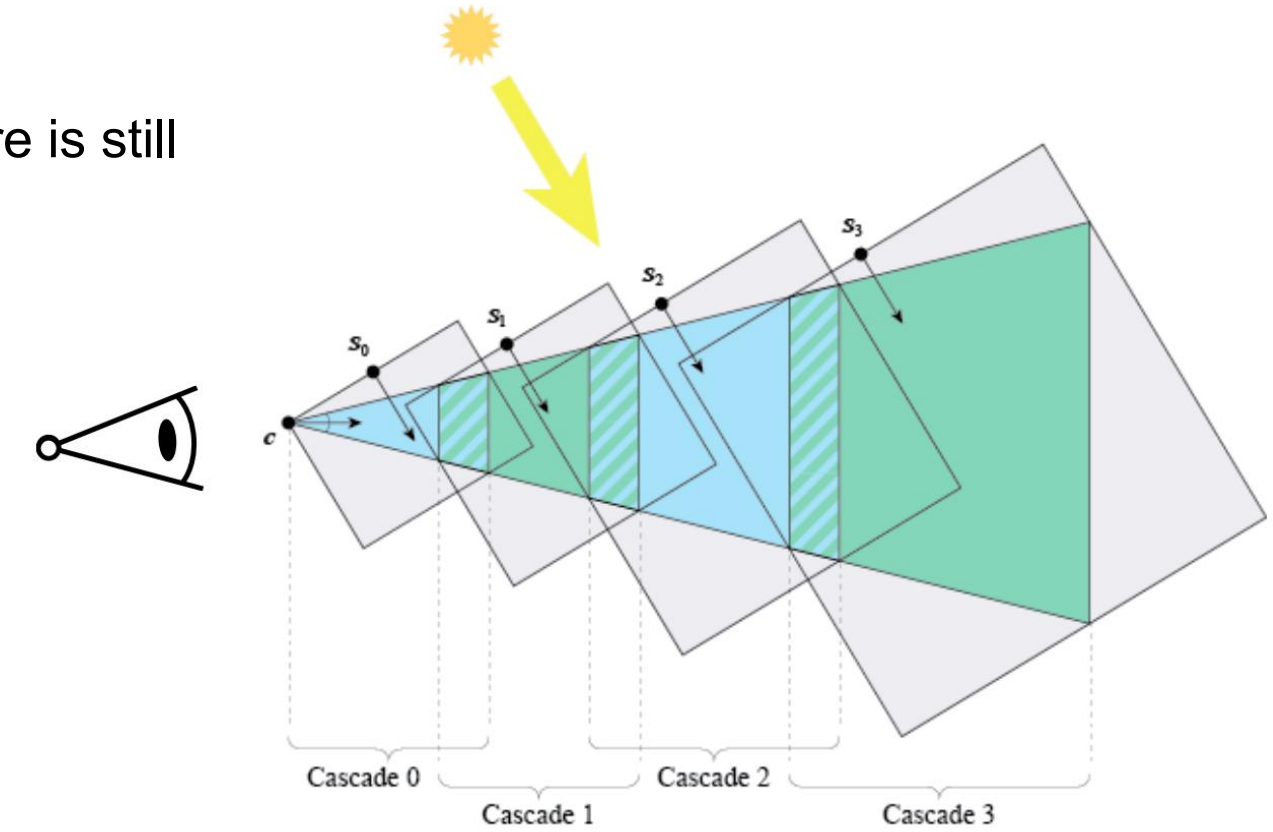
- Ray trace?
  - There are more shadow rays than primary since there are on average more than 1 light per pixel
  - Custom triangle encoding
  - No partial BVH updates
  - HW triangle formats + BLAS (bottom level acceleration structure) currently are 3-7x the size of Nanite data



RTX 40XX，50XX? Radeon RX 70XX...?

Recap Cascaded Shadow Map

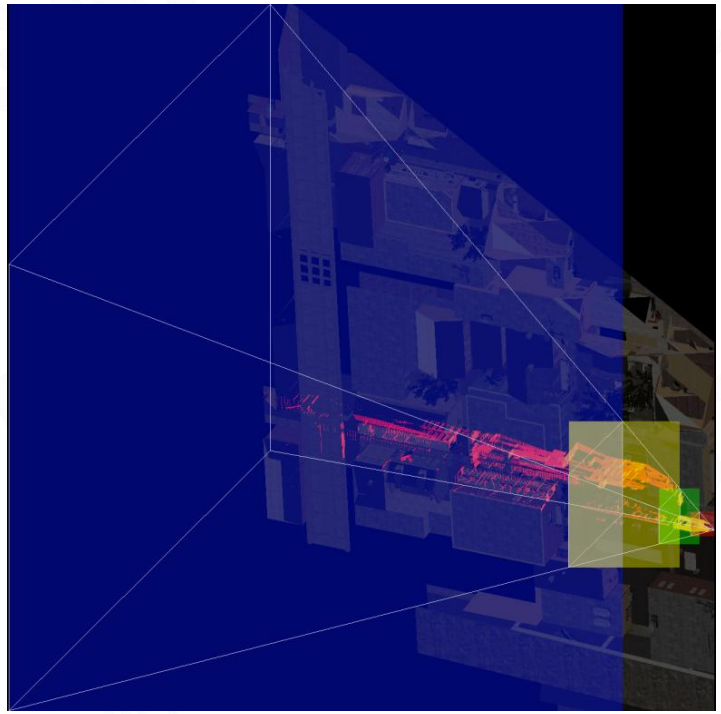
- Relatively coarse LOD control
- If better shadow detail is desired, there is still significant memory consumption

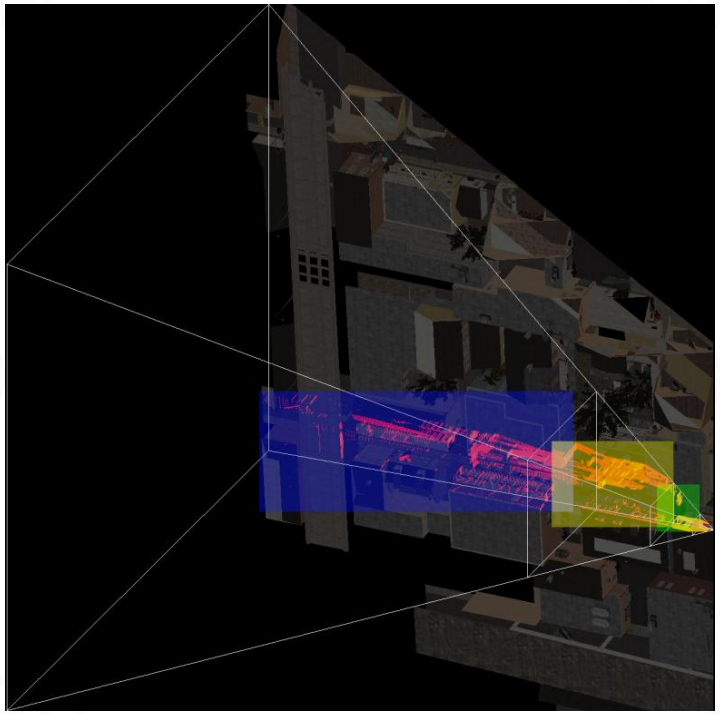


Sample Distribution Shadow Maps

- Gives a better cascaded map coverage by analysing the range of screen pixel depths
- An optimized cascaded shadow map but still has coarse LOD control

Cascaded shadow ranges





Sample distribution shadow ranges

White wireframe:camera frustum, red/yellow regions: where shadow samples are required

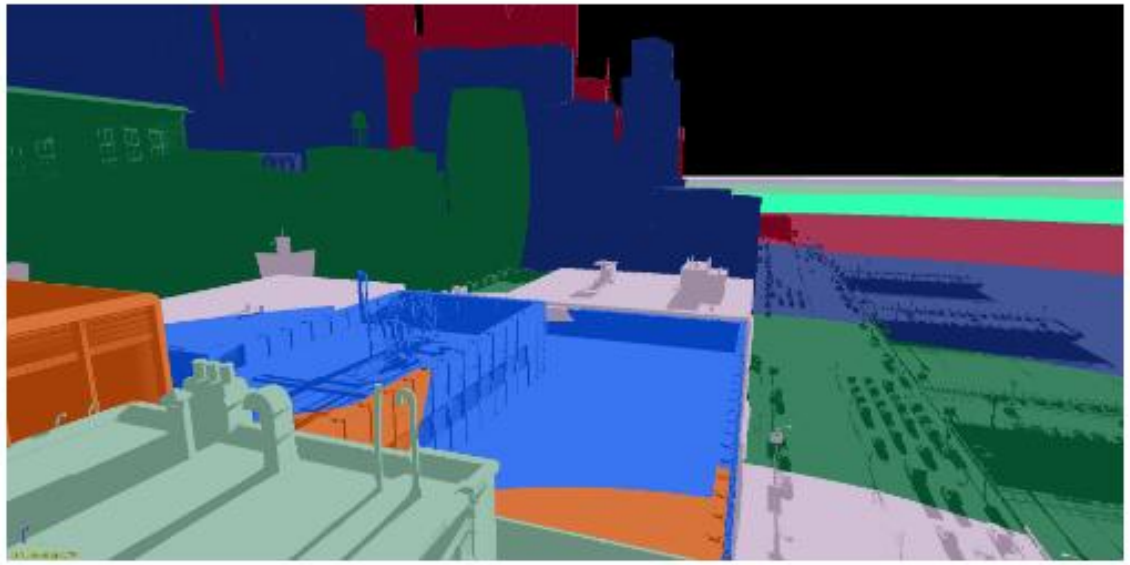
Sample Distribution Shadow Maps

Cascaded Shadow Maps

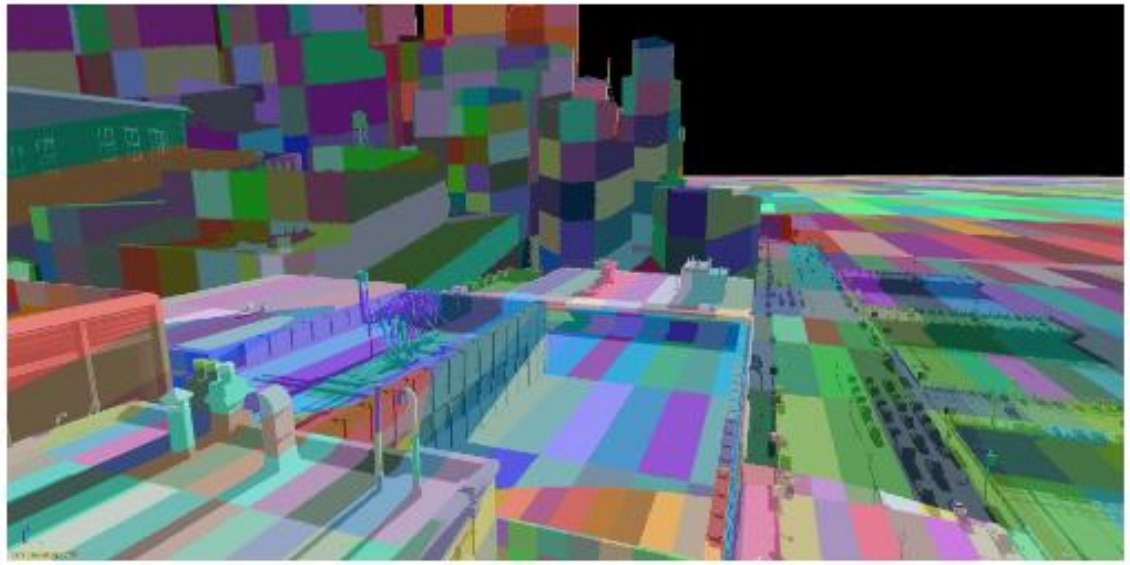
Sample Distribution Shadow Maps

Virtual Shadow Map - A Cached Shadow System!

- Most lights don't move, should be cached as much as possible



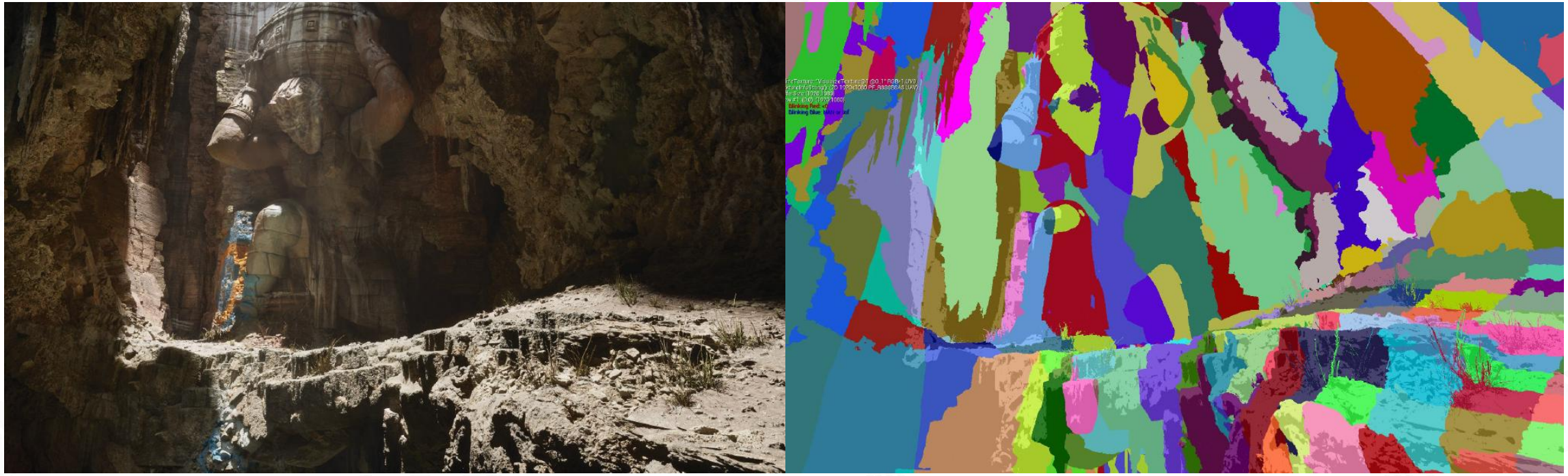
Directional Light Clipmap Visualization



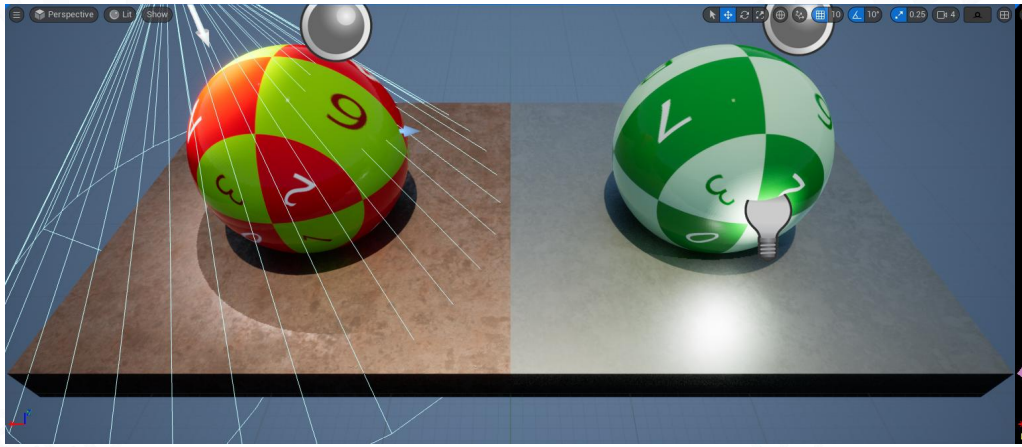
Virtual Shadow Map Pages Visualization

Virtual Shadow Maps

- 16k x 16k virtual shadow map for each light (exception, point light with 6 VSMs)

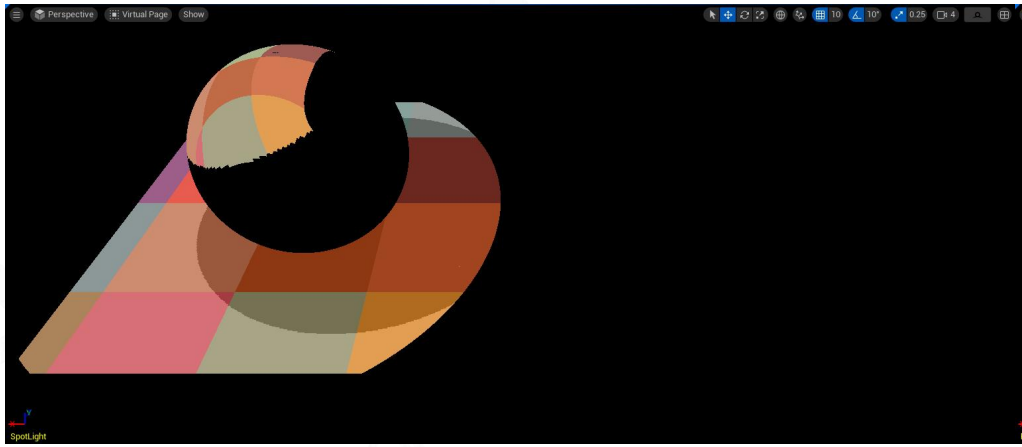


Different Light Type Shadow Maps

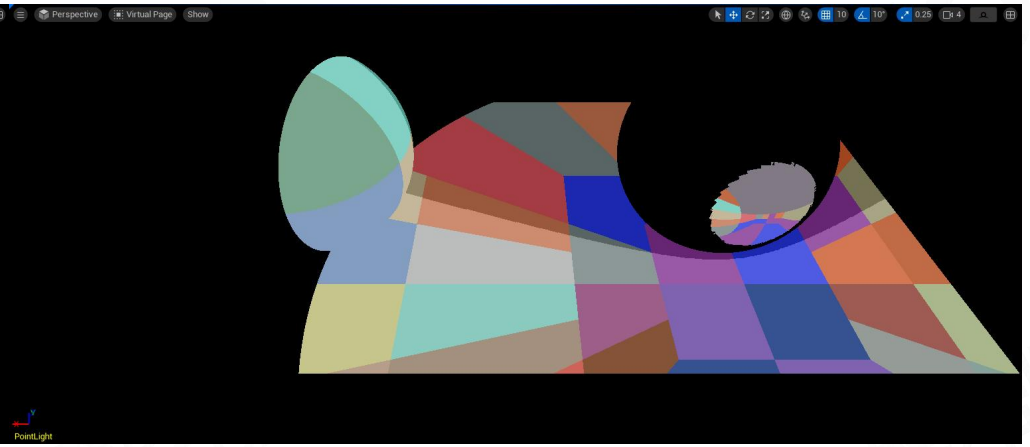


Scene with 3 lights

1. Directional light shadow pages (N level clipmaps)



2. Spot light shadow pages (1 projection map)

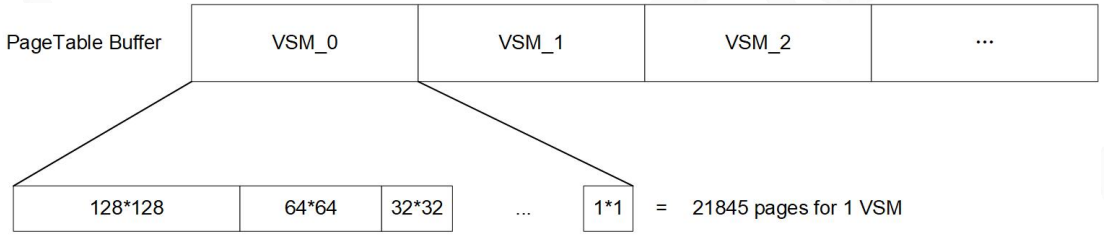


3. Point light shadow pages (6 cube face maps)

Shadow Page Allocation

- Only visible shadow pixels need to be cached
  - 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 the page as needed
  - Allocate physical page space for uncached pages

Shadow Page Table and Physical Pages Pool

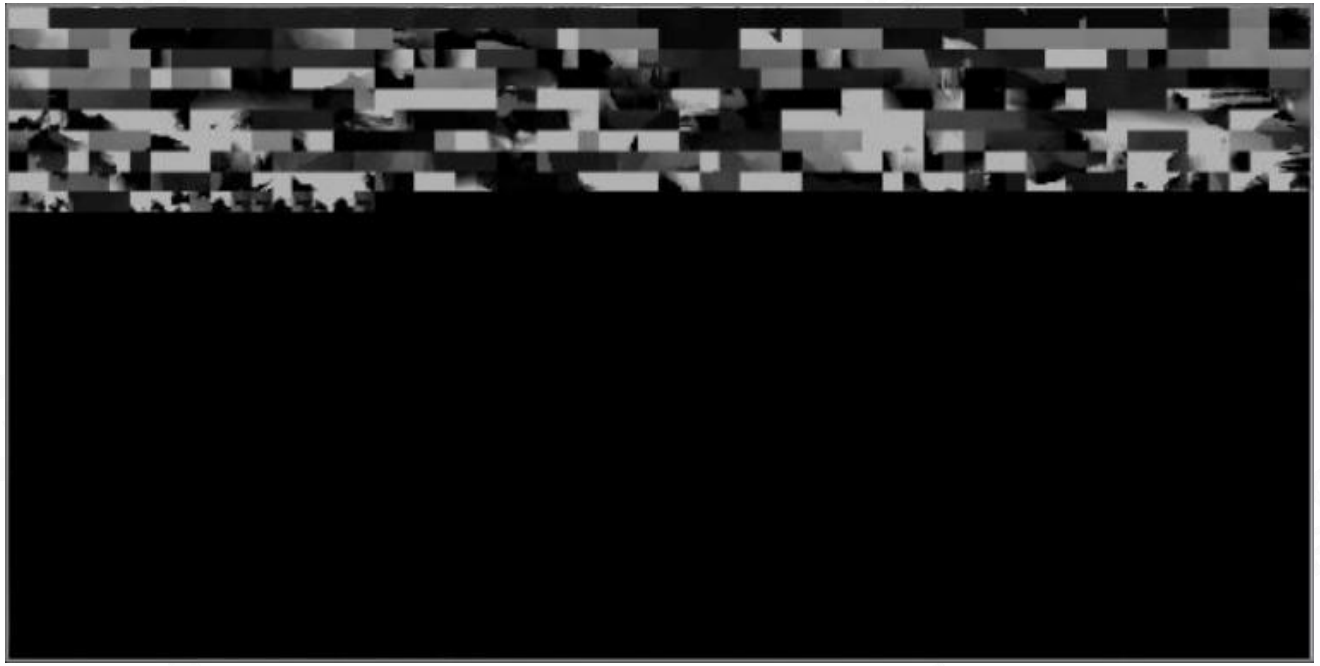



```
int2 LevelPos = UV_shadowmap * LevelDim
int LevelOffset = GetLevelOffset(MipLevel)
PageIndex = VSM_ID * PageTableSize + LevelOffset + LevelPos.x + LevelPos.y * LevelDim
```

```
PageTableEntry bits/value
[0:9]   PageAddress.x
[10:19] PageAddress.y
[20:25] LODOffset
[26:30] (currently unused)
[31]    bAnyLODValid
```

Indexing

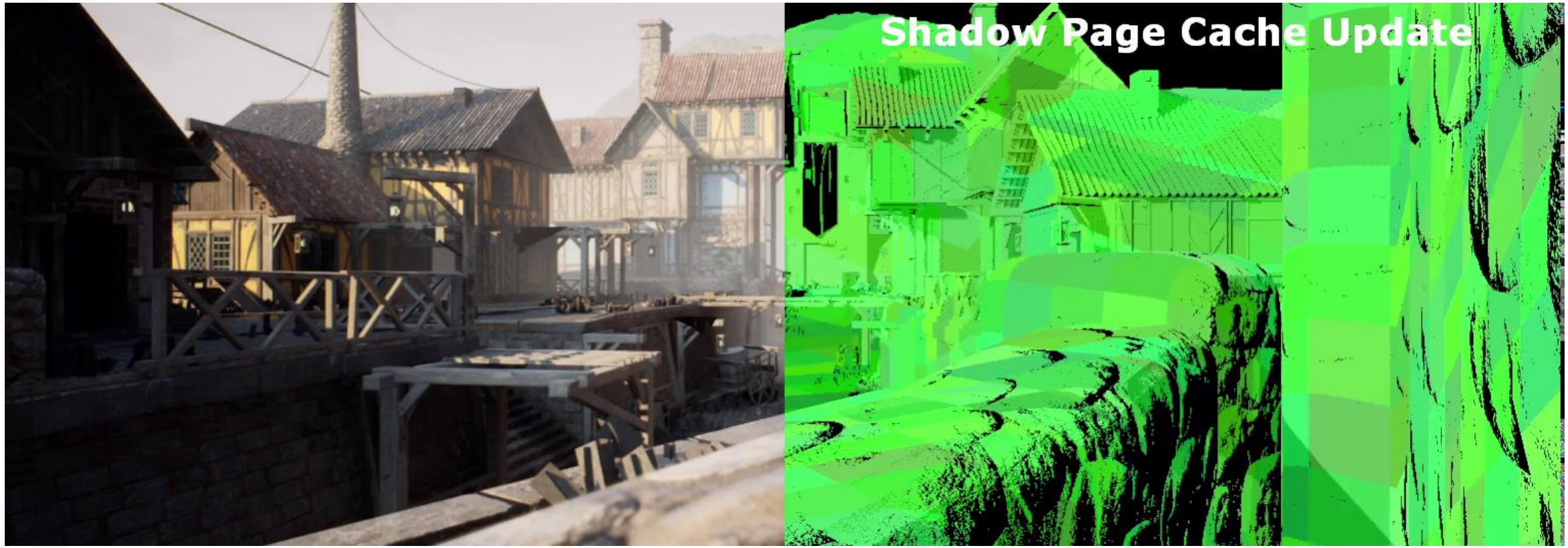
Physical shadow pages cache



Shadow Page Cache Invalidation

- Camera movement, if the movement is relatively smooth, there will not be many pages to update
- Any light movement or rotation will invalidate all cached pages for that light
- Geometry that casts shadows moving, or being added or removed from the scene will invalidate any pages that overlap its bounding box from the light's perspective
- Geometry using materials that may modify mesh positions
- ...

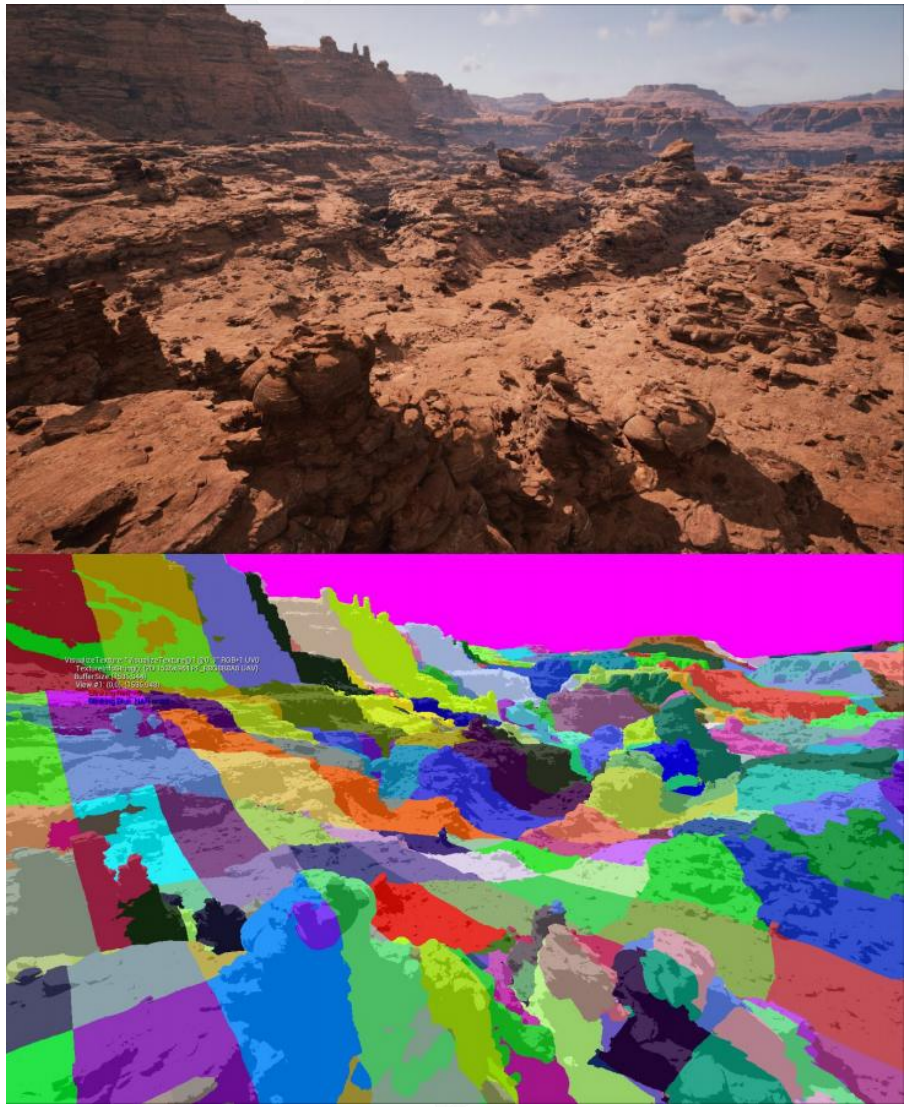
Shadow Demo



Conclusions

- Number of shadow pages proportional to screen pixels

- Shadow cost scales with resolution and number of lights per pixel



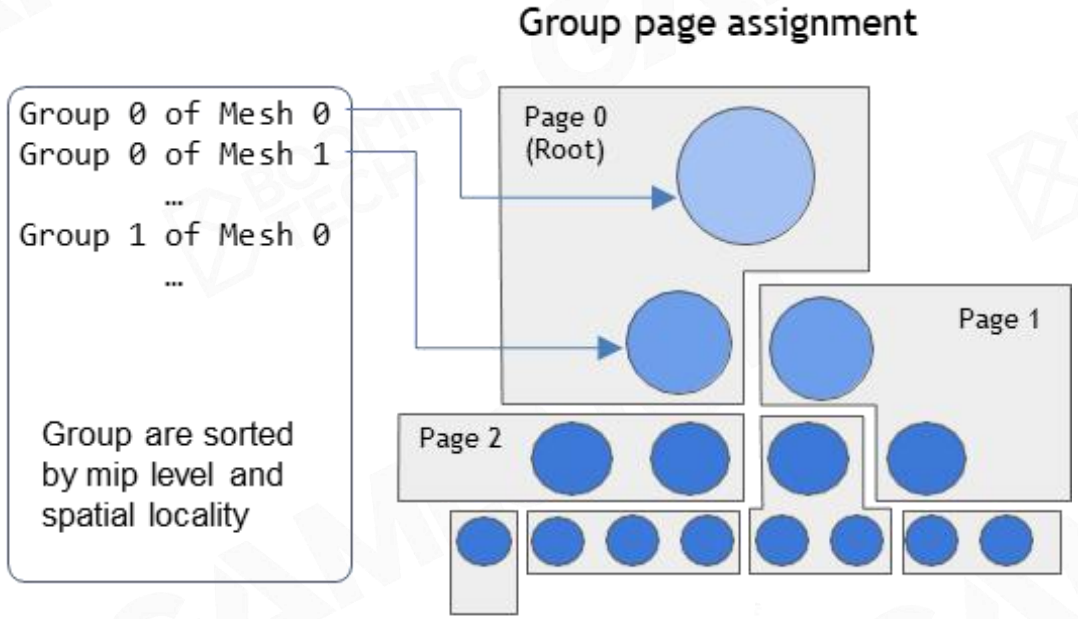
Streaming and Compression

Streaming

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

Paging

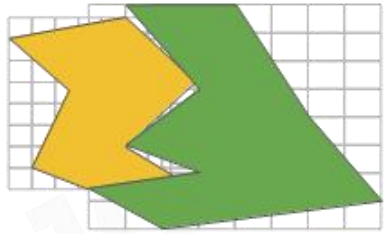
- Fill fixed-sized pages with groups
- Based on spatial locality to minimize pages needed at runtime
  - Sort groups by mip and spatial locality
- Root page(64k)
  - First page contains top lod level(s) of DAG
  - Always resident on GPU so we always have something to render
- Streaming Page(128k)
  - Other lod levels of cluster groups
  - Life time is managed by LRU on CPU
- Page contents:
  - Index data,Vertex data, Bounds, LOD info, Material tables, etc.



Memory representation

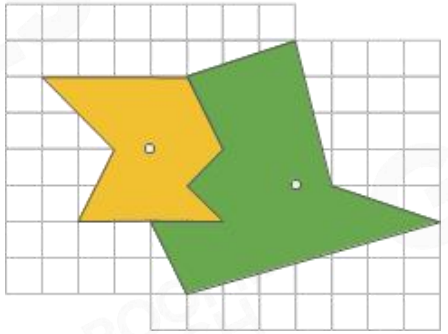
Vertex quantization and encoding

- Global quantization
  - A combination of artist control and heuristics
  - Clusters store values in local coordinates that is relative to value min/max range



- Per-cluster custom vertex format
  - Uses minimum number of bits per component: ceil(log2(range))
  - Just a string of bits, not even byte aligned

- Decoded using GPU bit-stream reader because of divergent encode format between clusters

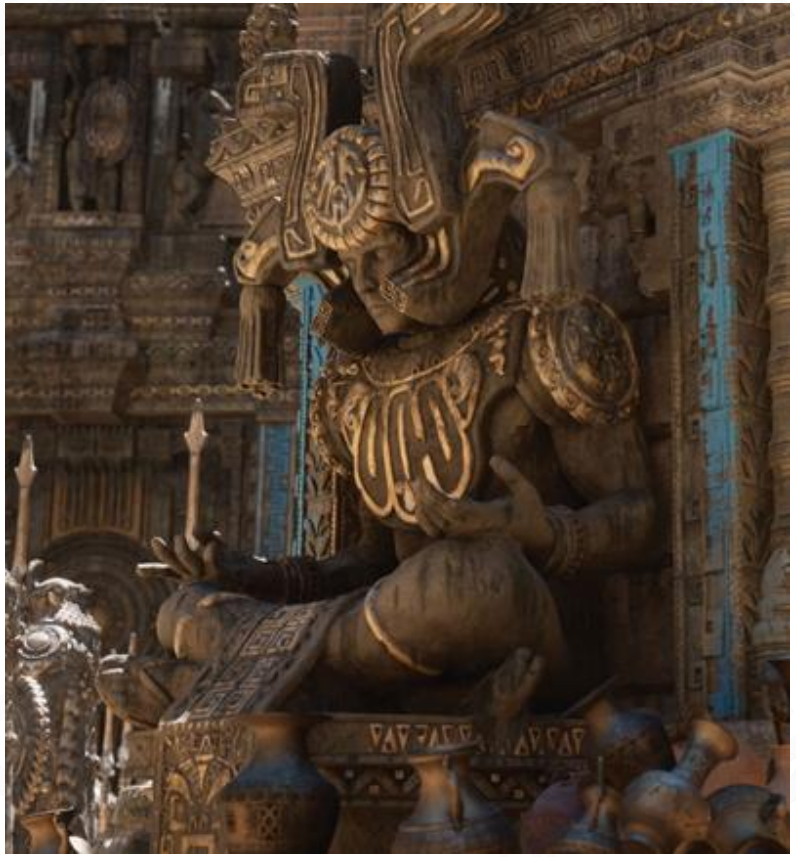


Disk Representation

- Hardware LZ decompression
  - In consoles now and on its way to PC with DirectStorage
  - Unbeatably fast, but general purpose
  - String deduplication and entropy coding
- For better compression
  - Domain-specific transforms
  - Focus on redundancies not already captured by LZ and massaging the data to better fit how LZ compression
- Transcode on the GPU
  - High throughput for parallel transforms, currently runs at ~50GB/s with fairly unoptimized code on PS5
  - Powerful in combination with hardware LZ
  - Eventually stream data directly to GPU memory

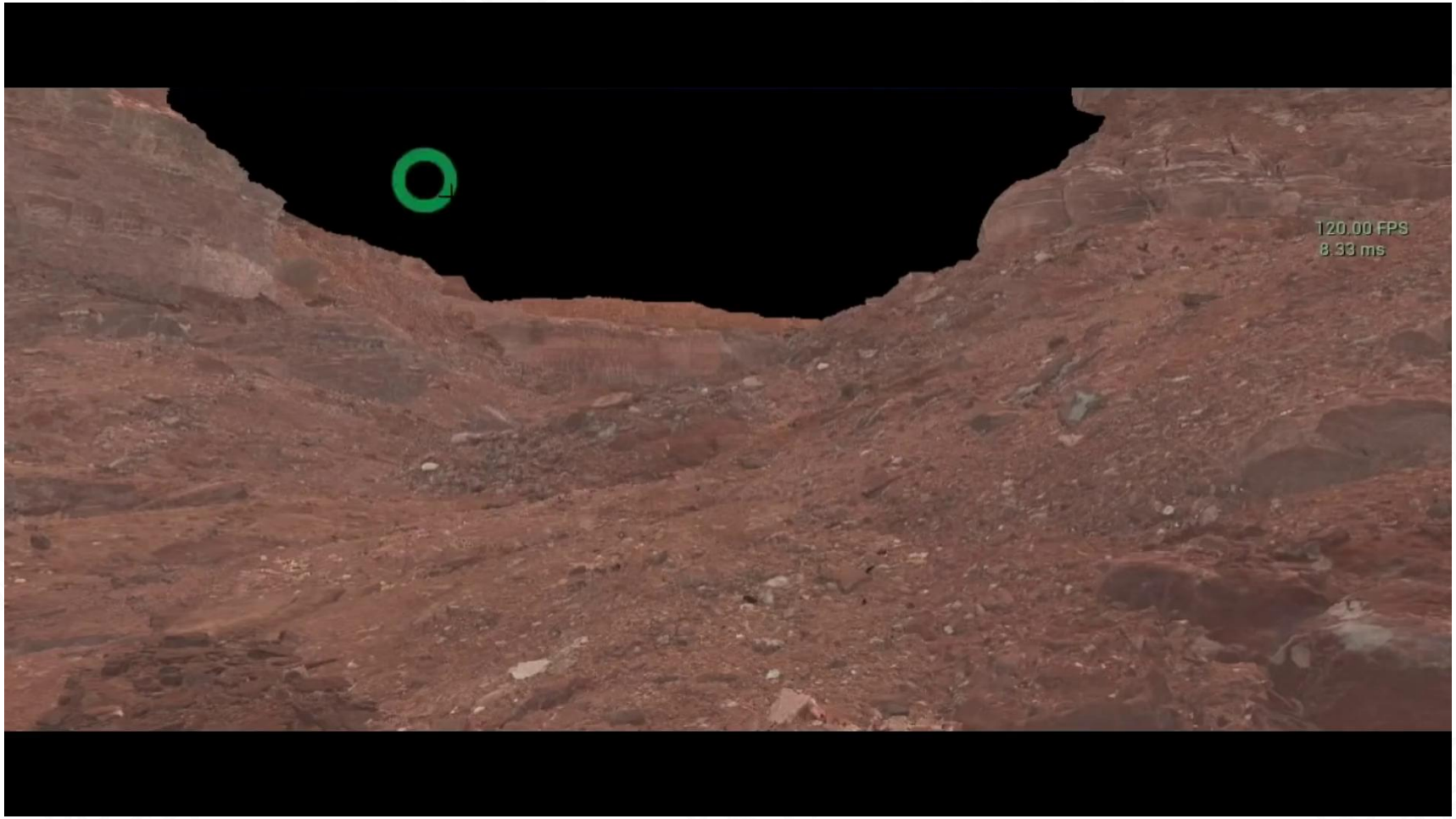
Results: Lumen in the Land of Nanite

- 433M Input triangles, 882M Nanite triangles
- Raw data: 25.90GB 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



Welcome to Billions of Triangles World

Jungle of Nanite Geometries



References

References

- The Nanite 2021: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final
- Journey to Nanite: https://www.highperformancegraphics.org/slides22/Journey_to_Nanite
- GPU-Driven Rendering Pipelines: https://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final
- The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading https://jcgt.org/published/0002/02/04/
- The filtered and culled Visibility Buffer: http://www.conffx.com/Visibility_Buffer_GDCE
- Optimizing the Graphics Pipeline with Compute: https://frostbite-wp-prd.s3.amazonaws.com/wp-content/uploads/2016/03/

Lecture 22 Contributors

- 砚书
- 光哥
- 炯哥
- 坤
- 超裕

Q&A

Enjoy ;)
Coding

Course Wechat

Please follow us for further information