Geometry Processing

Extra Session

The Nanite System in Unreal Engine 5

Ludwig-Maximilians-Universität München
Prerequisite
Recap: Rasterization Pipeline

init frame buffer
init depth buffer
for each triangle \( t \) in scene {
    \( tp = \text{project}(t) \) // MVP
    for each pixel \( p \) in frame buffer {
        if \( tp \) covers \( p \) { // culling
            if \( z \) passes depth test at \( p \) { // depth-test
                update \( z \) buffer and frame buffer // interpolation & update
            }
        }
    }
} flush frame buffer to monitor

Memory
- Uniforms
- Textures
- Buffers
- ...

Vertex Generation
Vertex Processing (Vertex Shader)
Primitive Generation (Tessellation Shader)
Primitive Processing (Geometry Shader)
Fragment Generation (Rasterization)
Fragment Processing (Fragment Shader)
Frame Buffer Ops
Deferred Rendering [Deering et al 1988]

Core idea: Two rendering pass! Use a geometry-buffer to store geometric information

Limitation: Screen-space information only

GBuffer: Depth, Albedo, Normal, Specular, Shadow, etc...
Deferred Rendering \[\text{Deering et al 1988}\]

Core idea: Two rendering pass! Use a geometry-buffer to store geometric information.

Limitation: Screen-space information only.

```plaintext
init frame buffer
init depth buffer
for each triangle $t$ in scene {
    for each pixel $p$ in frame buffer {
        if project($t$) covers $p$ {
            if $z$ passes depth test at $p$ {
                write(depth); shade(pixel)
            }
        }
    }
}
flush frame buffer to monitor
```

```plaintext
init frame buffer
init $G$ buffer
init depth buffer
for each triangle $t$ in scene {
    for each pixel $p$ in frame buffer {
        if project($t$) covers $p$ {
            if $z$ passes depth test at $p$ {
                write($z$); write($G$)
            }
        }
    }
}
for each pixel $p$ in frame buffer {
    shade($p$)
}
flush frame buffer to monitor
```

Forward Shading

Deferred Shading
The Concept of "Virtual"

- **Virtual Memory**: allows transparent memory access to a larger address space than the physical memory.
- **Virtual Texture**: allows mip-mapped texture used as cache to allow a much higher resolution texture to be emulated for RTR, while only partly residing in texture memory.
- **Virtual Geometry**: ???
GPU Drawing Mode: Immediate vs. Retained Mode

Retained Mode: Scene representation *persistence between frames on GPU*
- All vertex/index data in single large resource

Immediate Mode: Primitive vertex attributes may inserted *from CPU memory per frame*
- Vertex/index data are separately distributed
Extra Session: The Nanite System in Unreal Engine 5

● An Overview of The Nanite System in UE5

● Mesh Building
  ○ Clustering Hierarchy
  ○ Culling, LOD, Visibility Buffer

● Mesh Rendering
  ○ Rasterization
  ○ Material, Shading, Shadowing

● Summary
What is Nanite?

Each statue has more than 33 million triangles
Nanite: An Overview

Nanite is a GPU-driven deferred rendering pipeline for rigid meshes.

Diagram:
- **Nanite**
  - **Building**
    - FBuilderModule::Build
    - Build Clusters and Hierarchy Encode Pages
  - **Streaming**
    - FStreamingManager::RequestNanitePages
    - Page-based Streaming
  - **Rendering**
    - Nanite::CullRasterize
    - Culling, Rasterizing, Exporting Material Attributes, Shading
Nanite: An Overview

Nanite is a GPU-driven deferred rendering pipeline for rigid meshes.
Key Challenges in Building Nanite: Data and Transfer

High-resolution models: 1M Triangles ≈ 140MB (including idx, pos, nor, uv, nor, and etc.)

- consume large amounts of space (including disk and memory)
- read/write IO consumption between disk and memory
- bandwidths between CPU and GPU when copying memory

Solutions: Compression and Visibility buffer (optimized down to 13.8 MB on disk, not today’s topic :(
Costs and Solutions in Nanite

When data is large, there will also be huge rendering costs:

- **CPU processing cost**: Entirely GPU driven, hence not much serious
- **CPU-GPU communication cost**: Parallel command buffer commit; GPU-Driven pipeline hence not much costs too
- **Vertex shader cost**: LOD clusters and Culling to remove non-visible clusters to reduce vertex shaders
- **Primitive rasterization cost**: sub-pixel primitives are rasterized using *software rasterization*
Nanite GPU Pipeline

Nanite::Streaming: Read cluster rendering information from last frame

Nanite::InitContext: Initialize Culling context

Nanite::CullRasterize: Execute culling and rasterization

- Nanite::InitArgs
- Nanite::InstanceCull: Remove invisible instances
- Nanite::PersistentCull: Remove invisible clusters on BVH
- Nanite::CalculateSafeRasterizerArgs: Sanity checks for SW/HW render regions
- Rasterization (Hardware+Software, but all on GPU): Build visibility buffer
- Build HZB

Nanite::EmitDeptTargets

Nanite::BasePass: Render G-Buffer

Nanite::Shadows: Render Shadow Maps

Nanite::Readback
Extra Session: The Nanite System in Unreal Engine 5

● An Overview of The Nanite System in UE5

● Mesh Building
  ○ Clustering Hierarchy
  ○ LOD, Visibility Buffer

● Mesh Rendering
  ○ Culling
  ○ Rasterization
  ○ Material, Shading, Shadowing

● Summary
Nanite Mesh Preprocessing

Mesh process is computed offline, and triggered by any model update or activation.
Levels of Detail (LOD) Solutions

- Level of Detail (LOD), a.k.a "Cascaded", is the key to performance
  - Recall: texture MipMapping
  - Choosing the right level of detail to use can speed up computation
- Multiple ways of applying levels of details. Examples:
  - Cascaded shadow maps [Dimitrov et al. 2007]
  - Cascaded light propagation volume [Kaplanyan 2009]
  - Progressive Meshes [Hoppe 1996] [Garland et al 1997]
- Key challenges
  - Transition between different levels: Handling discrete levels
  - Usually need some overlapping and blending near boundaries: Handling cracks
Nanite organizes a static mesh by the following hierarchy:

- Each mesh generates different LODs
- Each LOD points to different groups
- Each group is a collection of multiple clusters
- A cluster can only be in a single LOD
Cluster Hierarchy
Cluster Hierarchy

LOD0

G0_0
Cluster L0_0 Cluster L0_1 ... Cluster L0_7

Merge

G1_1
Cluster L1_0 Cluster L1_1 ... Cluster L3_3

LOD1

G0_1
Cluster L0_0 Cluster L0_1 ... Cluster L0_7

Merge

G1_1
Cluster L1_0 Cluster L1_1 ... Cluster L3_3
**View-dependent Selection** [Hoppe 1997]

Two submeshes contain the same boundary but in different LOD.

Choose between them based on screen-space error.

All clusters in the group must make the same LOD decision.
Cluster Generation

- Use original mesh (LOD 0) to generate graph partition
- Similar to Meshlets from NVIDIA
- Neighbor triangles are treated as a cluster, using *Metis* (cache-miss)
- Each cluster contains 128 triangles (to fit vertex processing memory cache)
- Cluster can be grouped together, each group contains 8~32 clusters
- Each LOD needs to repeat this process and each group belongs to one mesh
Handling LOD Cracks by Graph Partitioning [Ponchio 2009]
Handling LOD Cracks by Graph Partitioning in Nanite

Pick Group  Merge and Simplify  Re-partitioning
Mesh Simplification

Per Cluster Group Simplification

Simplification by QSim [Garland 1997] (discussed in Geometry Processing: 5 Remesh)
Nanite Mesh Build Process

Cluster original triangles

While NumCluster > 1, do:

- **Group** clusters to clean their shared boundary
- **Merge** triangles from group into shared list
- **Simplify** to 50% of triangles
- **Split** simplified triangle list into clusters (per 128 triangles)
Extra Session: The Nanite System in Unreal Engine 5

- An Overview of The Nanite System in UE5
- Mesh Building
  - Clustering Hierarchy
  - Culling, LOD, Visibility Buffer
- Mesh Rendering
  - Rasterization
  - Emit Targets and Material Classification
- Summary
The Rendering Pipeline with Nanite

Conventional mesh can go through the traditional pipeline to render a GBuffer, whereas a nanite mesh is using a compute shader for culling then clustering triangles to do either HW or SW rasterization.
Culling Pipeline

A culling pipeline requires two passes. The first pass requires Hierarchical Z-Buffer (HZB) from last frame.
(Two-Pass) Culling Pipeline
Instance Culling

Instance culling removes invisible instances.

Visible instances are stored in a candidate BVH node list for persistent cull.
Hierarchical (Persistent) Culling

Each level of BVH is depending on its parents, the naive approach by dispatch per level is slow (obviously)

Using a persistent thread to consume all dispatched tasks (work-stealing algorithm)
Sub-pixel Triangle Rasterization [Kenzel et al 2018]

After culling, depending on the screen space size, each cluster will be dispatched to different SW rasterizers.

Large triangles are sent into HW rasterizer

Sub-pixel triangles are sent to SW rasterizer that runs on compute shader

The rasterizer is based on Scanline algorithm (discussed in Computer Graphics 1: Rasterization I)

- Each cluster run an individual compute shader, and cache all clip space vertex position to shared memory
- Each thread reads index buffer and transformed vertex position, run backface culling and write depth atomically into visibility buffer.
Visibility Buffer

A general visibility buffer is similar to GBuffer but stores much less information on memory:

1. Instance ID
2. Primitive ID
3. Barycentric coordinates (for interpolation)
4. Depth buffer
5. Material ID (for shading)

Except screen resolution buffer data, visibility buffer require two additional structure:

1. Global vertex buffer
2. Global material map

But Nanite is different:


**Emit Targets**

When visibility buffer is prepared, all information will be write into depth/stencil buffer and motion vector buffer. This consists of multiple screen-passes:

1. Emit scene depth/stencil/nanite mask/velocity buffer
   - Nanite Mask: 0/1 to indicate if mesh is conventional or nanite
   - Scene Depth Buffer: converted from Visibility buffer
   - Velocity buffer: for motion vectors

2. Emit Material Depth

![Nanite Mask](image1)
![Velocity Buffer](image2)
![Scene Depth/Stencil Buffer](image3)
![Material Depth Buffer](image4)
Classify Materials & Emit G-Buffer

In the shading phase, visibility buffer maintains a global material table that stores material parameters and texture index. Each pixel uses material ID to find the material, and parse material information then work with virtual texturing. In a complex scene, the number of materials are huge and searching a correct material is costly.

In the base pass, screen space is split into 8x8 segments, each segment counts material depths range.
Performance

- Average ~2496x1404 upsampling to 4K
- Culling => Rasterizer => Visibility Buffer ~2.5ms
- Visibility Buffer => GBuffer ~2ms

<table>
<thead>
<tr>
<th>Nanite::CullRasterize</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear VisBuf</td>
<td>66us</td>
</tr>
<tr>
<td>InstanceCull</td>
<td>108ms</td>
</tr>
<tr>
<td>PersistentCull</td>
<td>406us</td>
</tr>
<tr>
<td>Rasterize</td>
<td>1148us</td>
</tr>
<tr>
<td>Build HZB</td>
<td>99us</td>
</tr>
<tr>
<td>Post InstanceCull</td>
<td>125us</td>
</tr>
<tr>
<td>Post PersistentCull</td>
<td>102us</td>
</tr>
<tr>
<td>Post Rasterize</td>
<td>183us</td>
</tr>
<tr>
<td>DepthExport</td>
<td>217us</td>
</tr>
<tr>
<td>Emit GBuffer</td>
<td>2084us</td>
</tr>
</tbody>
</table>

Changkun Ou, Prof. Butz | Universität München | mimuc.de/gp
Extra Session: The Nanite System in Unreal Engine 5

- An Overview of The Nanite System in UE5
- Mesh Building
  - Clustering Hierarchy
  - Culling, LOD, Visibility Buffer
- Mesh Rendering
  - Rasterization
  - Material, Shading, Shadowing
- Summary
Summary

- Nanite is an extreme clever engineering that utilizes large amounts of research from 2000s
- The key engineering insights from Nanite
  - Software rasterization is powerful in processing subpixel triangles (but it is likely that future hardware will add additional support for this :)
  - "Perceptually lossless" real-time rendering on large high-resolution rigid models made possible (no more polygon budgets)
- Limitations
  - No Forward Rendering
  - No Alpha Blending/Mask Textures
  - No Deformable Meshes (including skinning, etc)
  - ...
- Future Work
  - Non-rigid models?
  - Tessellation and displacement?
- Uncovered topics: Virtual shadow map (VSM), paging, streaming, disk compression, TAA, etc.
Nanite Source Code

- Search Strategy: "namespace Nanite"
- https://github.com/EpicGames/UnrealEngine/tree/ue5-main/Engine/Source/Developer/NaniteBuilder
- https://github.com/EpicGames/UnrealEngine/tree/ue5-main/Engine/Source/Runtime/Renderer/Private/Nanite
- https://github.com/EpicGames/UnrealEngine/blob/ue5-main/Engine/Source/Runtime/Engine/Private/Rendering/NaniteStreamingManager.cpp
Further Readings: Geometry


Further Readings: Rendering

Further Readings: **Official Sources from EpicGames**


Additional Slides
Compression and Encoding

Compression: Memory & disk representations

- High-resolution cluster use high-precision floating numbers
- Low resolution cluster uses low-precision floating numbers
- Cracks between clusters can align high precision to low precision

Page Encoding:

- 128kb size page
- Each page stores multiple groups
- Spatial neighbors are stored in the same page, and use Moton Code to order cluster groups then use LZ4 compression
More about Virtual Texture

Hardware Virtual Texture: Hardware sampling, address mapping, no more runtime memory costs, etc.

- Direct X: Tiled Resource
- Vulkan: Sparse Partially-Resident Images
- OpenGL: ARB_sparse_texture
- Metal: Sparse Texture
Global Illumination Solutions

When would screen space ray tracing (SSR) fail?

No single GI solution that is perfect for all cases, except RTRT

But complexity using RTRT is still costly in the current generation

Therefore industry trends to use hybrid solutions

A possible solution:

SSR for a rough GI approximation

Upon SSR failure, switch to more complex ray tracing

Either hardware (RTRT) or software (?)
Global Illumination Solutions

Software ray tracing

- HQ SDF for individual objects that are close-by
- LQ SDF for the entire scene
- RSM if there are strong directional / point lights
- Probes that stores irradiance in a 3D grid (Dynamic Diffuse GI, or DDGI)

Hardware ray tracing

- Doesn't have to use the original geometry, but low-poly proxies
- Probes (RTXGI)

Highlighted solutions are mixed to get Lumen in UE5