Unreal Engine's Nanite

Rui Armada
Visualização em Tempo Real
Universidade do Minho
pg50737@alunos.uminho.pt

Abstract—In the present document it is presented the state-of-the-art of one of Unreal Engine's latests technologies, the *Unreal Engine's Nanite*. Being a new virtualized geometry technology, *Nanite* is leading the gaming and entertainment industry by making it possible to create photorealistic environments with massive amount of detail and visual fidelity. Furthermore, some experimentation will be performed in order to verify if this technology is worthwhile and if it really can augment the performance of a scene.

Index Terms—Unreal Engine 5, Nanite, Virtualized Geometry, Level of Detail, Performance

1 Introduction

One of the biggest struggles of real time rendering engines, currently, is how to display a lot of detail without a big impact on the overall performance of the scene or game.

Usually, during the pipeline of asset development, after the construction of the high-poly version of a model, there is the need to apply optimization techniques to the asset in question. For example, surface subdivision can be used to give the viewer the idea that the model has more geometry than it actually has. The same can be said about normal mapping, usually used to "fake" bumps or even protrusions in the model, giving the illusion of texture and detail without incrementing the poly count of the model.

To the untrained eye, all these techniques mentioned above offer the illusion of detail without the viewer ever noticing that, in reality, the model has a very small poly-count compared to what can be perceived. But, to the trained eye, it can be noticed some clues that are clear giveaways of the methods that were used during the asset optimization stage. For instance, in normal mapping, if the model is a sphere, and it's zoomed in immensely, the original geometry of the model can be noticed breaking the illusion that the model is fully smoothed. Furthermore, when it comes to monetary resources and time spent on the creation of the asset, it's much simpler, cheaper, and quicker to simply create a highly detailed model with a much higher polygon count; however, without proper optimization techniques, the final product will ultimately affect the performance of the actual scene being rendered.

In a brief overview, Unreal Engine is one of the most popular game development engines around. It has become one of the industry standard engines for creating high-quality, interactive and immersive multimedia across various platforms.

That being said, there was the need to develop a technology capable of handling assets with a high poly-count without tanking the engine's overall performance, and thus *Unreal Engine's Nanite* was born. The need for a technology akin to *Nanite* arose because the cost of using a different team of people to apply optimization techniques to the asset was higher than just importing the highly detailed mesh directly into the engine and using *Nanite* to optimize the mesh without the labour, and time-consuming, cost attached to it.

2 Evolution of Real-Time Graphics

Fig. 1. Capcom's Monster Hunter World was one of the games that set the bar for how to create a photorealistic, real-time 3D graphics, fantasy worlds.

The first edition of a 3D Game Engine Design was written in the late 1990s [1] and for a long time all computer graphics remained stalled in time and looked akin to games like *Tron*, *Mega-Man* and even the first *Super Mario World*. At the time, all these games were a technological marvel in the 3D Graphics world but far away from the realism of the real world. Then, pre-rendered 3D started to look good enough for objects like spaceships, and now it is possible to, almost, claw all the way out of the *uncanny valley* in which CG humans have often resided. So, in recent years *Nvidia*, *AMD* and the people interested in the CG field have been pushing the limits in order to make video games look as good as movies or other photorealistic media.

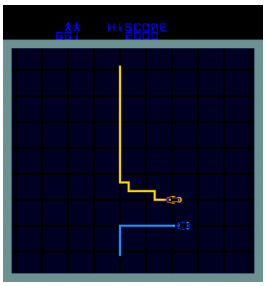

Fig. 2. SBally Midway's Tron, released in 1982 [2].

That being said, through the years we get closer to the objective of full immersive and photorealistic games. It's possible nowadays to write code to draw video games in engines like *Unreal Engine* or *Unity* to produce graphics for LED volumes that are capable and are expected to achieve photorealism. Recent releases are, occasionally, capable of producing scenes capable of provoking a fleeting impression of a real world on the other side of the display screen.

Until very recently, video games invariably used what was fundamentally a development of techniques going decades. First, triangles were used to make 3D objects. Why triangles? Because no matter where three points are in space, they always describe a two-dimensional flat area. At first, the triangles weren't even filled in – Wireframe graphics can be seen straight through. After this, it was concluded that the triangles would look better filled in. Then we started to attach bits of image data onto these triangles, and thus textures were born, and about the same time we started to take notice of where virtual light sources were in the scene to make the triangles brighter or darker.

This level of technology worked out fine through the 2000s, but there are obvious things it couldn't achieve, things that pre-rendered 3D could. Until recently, video games weren't capable of displaying accurate reflections, they could sometimes do something that looked like a reflection of the environment on a chromed out surface, but it was just a fixed image that didn't represent changes in the environment that's being reflected. They also couldn't achieve an accurate shadow system; without a lot of workarounds, usually the shadow of an unmoving object was a pre-rendered image pasted onto the triangles of the floor to simulate light and shadow.

Fig. 3. Without any type of ray-tracing, the image reflected is a static image of the city to the point that it doesn't capture the character [3].

This creates a good example of the sort of limitations real-time 3D engines often impose in order to achieve what they do. With this, there was the need to build newer technologies capable of bypassing these issues and replicating accurate images with the maximum of visual fidelity possible, and so *Unreal Engine's Nanite* is one of those technologies, capable of bypassing the problem of loss of mesh detail during the process of asset optimization and producing meshes with huge amounts of detail without plummeting the engine's performance.

3 What is Nanite?

Nanite is the solution to the problem of importing assets, with immense detail, into real time rendering engines just like Unreal. For starters, *Nanite* is a virtualized micropolygon[1] geometry system introduced in Unreal Engine 5. It is a rendering technology that enables the rendering of extremely detailed and complex 3D environments in real-time. Furthermore, Unreal Engine's *Nanite* follows the following core concepts and principles:

- **Detail preservation:** *Nanite*'s core principle is to preserve details and fidelity in the scene to be rendered. It achieves this by dynamically adapting the level of detail based on the camera's proximity to objects. As the camera gets closer, *Nanite* seamlessly loads higher-resolution micropolygons to maintain sharpness and detail, resulting in a more realistic and immersive experience.
- **Micropolygon Geometry:** *Nanite* uses a micropolygon geometry system in order to render highly detailed meshes and environments. Instead of relying on traditional triangle-meshes, *Nanite* breaks down geometry into tiny pieces, micropolygons, which are small surface patches that can be as small as a pixel. This approach allows for highly detailed surfaces and eliminates the need for pre-baked LODs or manual optimization of geometry.
- **Virtualized Geometry:** *Nanite* uses a virtualized geometry approach, where only the visible portions of the geometry are loaded into memory and processed. This allows the rendering of massive environments with billions of polygons while keeping memory usage in check. It leverages virtual texturing and hardware virtual memory to efficiently stream and manage the geometry data involved in the scene.
- **Hierarchical Level of Detail (LODs):** *Nanite* also incorporates a hierarchical level of detail system to manage the complexity of large scenes. It automatically generates LODs based on the level of detail required for a given area or camera distance. This helps the optimization of performance by dynamically reducing the number of polygons rendered in real-time.
- **Scalability and Performance:** *Nanite* is designed to scale across different hardware configurations. It leverages modern GPU features like hardware-accelerated ray tracing and variable rate shading to deliver real-time performance, even with highly detailed scenes. *Nanite* also supports other rendering features like global illumination, dynamic lighting and advanced materials to enhance overall visual quality.

1. Micropolygons are tiny polygons that represent the smallest visible details in a scene, such as individual pebbles or grains of sand.

In summary, *Nanite* revolutionized real-time rendering by enabling the rendering of incredibly detailed and complex 3D environments. By utilizing all the core principles mentioned above, it can offer a high level of visual fidelity while maintaining real-time performance.

4 Building Nanite

In this section, it shall be examined all the technical and artistic aspects that enabled the creation of *Nanite*. Firstly, it will be examined the artistic points that were taken into account, including the main strategies that were considered in order to build a solid foundation for this technology. Then, it will be examined the technical points and implementation of *Nanite*.

4.1 Artistic Assumptions

If the geometry is virtualized, the problem of budgets for geometry would cease to exists. So, in order to build *Nanite*, there was the need to take into account some strategies that would possibly help in achieving this goal.

Firstly, there was the idea of using Voxels[2] in order to build this technology. A lot of considerations needed to be made about the use of voxelization to achieve a virtualized geometry system, for instance when a certain *mesh* is re-sampled into Voxels it can look malformed. This happens because voxelization is a form of uniform resampling, and this induces loss in mesh detail and form.

In short, the issue with data size when voxelization is used remains too problematic in order to utilize this strategy for the virtualized geometry system. In order to replace explicit surfaces completely, many years and effort would be necessary to achieve the desired results [4].

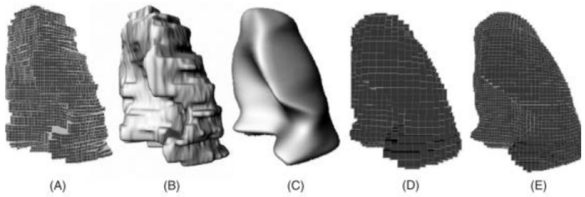

Fig. 4. Voxelization example [5].

Secondly, subdivision surfaces were considered. Subdividing a surface by definition is amplification only. This method gives great results up close, but the wireframe of the mesh doesn't get more simplified than the original one. In fact, it gets more complex the more subdivided the surface is, and it gets a higher poly count than the typical game low poly meshes, which affects the performance of the renderer. More polygons instigate more memory use that affects negatively the performance, which can be seen through low fps values and incrementally high draw calls.

2. A Voxel is a single sample, or data point, on a regularly spaced, three-dimensional grid.

Fig. 5. Subdivision levels 0 to 3, without and with Smooth Shading [6].

In addition, Displacement maps were taken into consideration. Basically, a displacement map can be viewed the same as a bump or even a normal map but far more powerful. A displacement map is, in essence, a texture map made to displace and manipulate actual geometry, meaning that shadows and silhouettes are also affected, and the lighting will also look as accurate as having handmade displacements. Naturally, this method comes with its costs: if the mesh is very basic in terms of poly count, there is no room for simple displacements because the genus of a surface can't increase. In other words, it's impossible to turn a cube into a torus using displacement mapping. It is also worth to mention that displacement mapping is a form of uniform resampling – this technique works well on organic surfaces that are already uniformly sampled, but proves really destructive on hard surfaces if not carefully controlled. It is great for up close and amplification, but not good enough for general purpose simplification.

Fig. 6. Displacement Mapping example [7].

After a long period of consideration, there was no faster or higher quality solution other than using triangles as the core of *Nanite*. In conclusion, if it is necessary to establish an artistic style based on these other approaches, taking into account their strengths and weaknesses, they can be highly effective and yield satisfactory results, but Unreal itself **cannot dictate** an artistic style.

4.2 Technical Assumptions

It is necessary to approach the technical points of *Nanite* from the perspective of requirements. In the past ten years, the development of 3A games has gradually tended to two main points: **interactive film narrative** and **open world**. In order to cutscene realistically, there is the need for exquisite character models and sufficiently flexible and rich open worlds. The map size and the number of objects have increased exponentially, both of which have greatly increased the requirements for the fineness and complexity of a scene - **usually the number of objects is much larger and each object is much more detailed**.

With this, there are usually two major bottlenecks in the rendering of complex environments:

1) The CPU-side verification and communication overhead between CPU-GPU brought by each Draw Call;
2) Overdraw caused by inaccurate elimination and the resulting waste of GPU computing resources;

In recent years, the optimization technology has often revolved around these two problems, and some technical consensus in the industry has been formed.

In view of the overhead caused by CPU-side verification and status switching, there is now a new generation of graphics APIs[3], which are designed for the following optimizations:

- Allow graphics drivers to do less verification work on the CPU side;
- Being able to dispatch different tasks through different Queues For GPU (Computer/Graphics/DMA Queue);
- Synchronization between CPU and GPU is required to be handled only by the developers;
- Make full use of the advantages of multicore CPU and multi-thread to submit commands to GPU;

Thanks to these optimizations, the number of Draw Calls of the new generation graphics API has increased by an order of magnitude compared with the previous generation graphics API[4].

Another optimization direction is to reduce the data communication between the CPU and GPU, and more accurately remove triangles that do not contribute to the final picture. Based on this idea, GPU Driven Pipelines were born.

Thanks to the more and more widespread application of GPU driven Pipelines in games, the vertex data of a model is further divided into more fine-grained clusters[5] so that the granularity of each cluster can better adapt to the Vertex Processing stage. Cache size and various types of culling (*Frustum Culling*, *Occlusion Culling* and *Backface Culling*) have gradually recognized this new vertex processing flow.

However, the traditional GPU Driven Pipeline relies on Compute Shader (CS) culling. The removed data needs to be stored in the GPU buffer via APIs; then, the removed Vertex/Index Buffer is fed back to the GPU's Graphics Pipeline, which has increased the invisible overhead of reading and writing. In addition, the data will be read repeatedly (CS reads before culling and the pipeline reads through Vertex Attributes when drawing).

Based on the previous reasons, in order to further improve the flexibility of vertex processing, Nvidia first introduced the concept of Mesh Shader, hoping to gradually remove some fixed units in the traditional vertex processing stage.

3. *Vulkan*, *DX12* and *Metal*
4. *DX11*, *OpenGL*.
5. Also known as "meshlets".

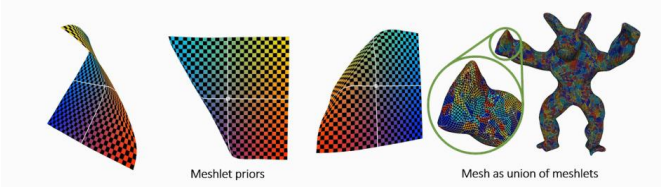

Fig. 7. Mesh representation in Clusters/Meshlets [8].

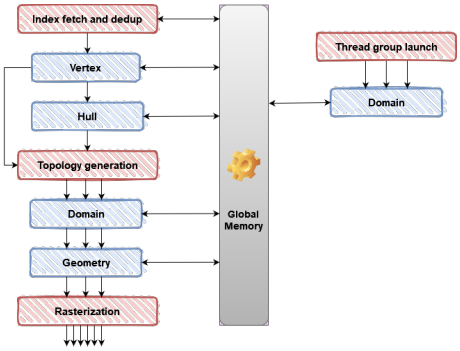

Fig. 8. Traditional GPU Driven Pipeline eliminates dependence on CS, and the eliminated data is passed to the vertex processing pipeline through VRAM.

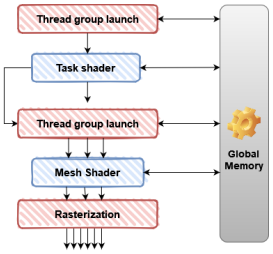

Fig. 9. Pipeline Based on the Mesh Shader, Cluster culling becomes part of the vertex processing stage, reducing unnecessary Vertex Buffer Load/Store.

4.2.1 Soft Rasterization vs Hard Rasterization

In order to clarify this issue, it is imperative to comprehend the fundamental functions of hard rasterization and the broad application scenarios it envisages. To put it simply: **At the beginning of the design of traditional rasterization hardware, the size of the input triangle envisaged is much larger than one pixel**. Based on this assumption, the process of hardware rasterization is usually hierarchical.

Taking the Nvidia Rasterizer as an example, a triangle usually undergoes two stages of rasterization: *Coarse Raster* and *Fine Raster*. The former takes a triangle as input and 8×8 pixels as a block, and the triangle is rasterized into several blocks.

At this stage, the occluded block will be completely eliminated through the low-resolution `Z-Buffer` which is called ***Z Cull*** on the Nvidia card. After the *Coarse Raster*, the block that passes through the *Z Cull* will be sent to *Fine Raster*, which finally generates pixels for shading. In the *Fine Raster* stage, begins the familiarity with ***Early Z***. Due to calculation needs of Mip-Map sampling, the information of the adjacent pixels of each pixel must be known. The difference

of the sampled UV must be used as the calculation basis for the Mip-Map sampling level. For this reason, the final output of *Fine Raster* is not a pixel, but a small 2×2 **Pixel Quad**.

For triangles close to the pixel size, the waste of hardware rasterization is obvious. Firstly, the *Coarse Raster* stage is almost useless, since these triangles are usually smaller than 8×8. This situation is even worse for the long and narrow triangles, not only because a triangle often spans multiple blocks and cannot be removed by *Coarse Raster*, but also because it adds additional computational burden. Furthermore, for larger triangles, the *Fine Raster* stage based on Pixel Quad will only generate a small number of useless pixels on the edge of the triangle, which is only a small part of the area of the entire triangle. However, for small triangles, pixel quad will generate pixels four times the area of the triangle at worst.

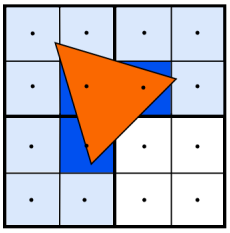

Fig. 10. Rasterization waste of small triangles due to pixel quad.

For these reasons, soft rasterization based on Compute Shader does have a chance to overthrow hard rasterization under the specific premise of pixel-level small triangles.

4.2.2 Deferred Material

In the evolution of the rendering pipeline, some researchers have proposed *Z-Prepass*, *Deferred Rendering*, *Tile Based Rendering*, and *Clustered Rendering* to solve the problem of redrawing. These different rendering pipeline frameworks that are used to answer the same question: **When a light source exceeds a certain value and the complexity of the material increases, how to avoid numerous rendering logic branches in the Shader and reduce redrawing?**

Generally speaking, the deferred rendering pipeline needs a set of *Render Targets* called ***G-Buffers***. These textures store all the material information needed for lighting calculations. In today's 3A games, the types of materials are often complex and changeable, and the *G-Buffer* information that needs to be stored is increasing year by year.

Fig. 11. Guerrilla's *Killzone* 2 *G-Buffer* layout. [9]

Excluding *Lighting Buffer*, in fact, the number for *G-Buffer* is 4, totalling 16 Bytes/Pixel.

On the other hand, in recent years **due to the increased material complexity and fidelity, the bandwidth required by the G-Buffer has been doubled**.

For scenes with high Overdraw, the read and write bandwidth generated by the drawing of *G-Buffer* will overcome a performance bottleneck. With this, a new rendering pipeline was introduced: the ***Visibility Buffer***. Algorithms based on *Visibility Buffer* no longer generate bloated *G-Buffers* alone, but instead use *Visibility Buffers* with lower bandwidth overhead. The *Visibility Buffer* usually needs the following information:

1) **Instance ID**, which indicates which Instance ($16 \approx 24$ bits) the current pixel belongs to;
2) **Primitive ID**, which indicates which triangle ($8 \approx 16$ bits) of Instance the current pixel belongs to;
3) **Barycentric Coord**, which represents the position of the current pixel in the triangle, expressed in barycentric coordinates (16 bits);
4) **Depth Buffer**, which represents the depth of the current pixel ($16 \approx 24$ bits);
5) **Material ID**, which indicates which material the current pixel belongs to ($8 \approx 16$ bits);

It is only needed to store about $8 \approx 12$ Bytes/Pixel to represent the material information of all geometries in the scene. At the same time, a global vertex and texture map must be maintained. The table stores the vertex data of all geometries in the current frame, as well as material parameters and textures.

In the lighting shading stage, it's only needed to index to the relevant triangle information from the global *Vertex Buffer* according to the Instance ID and Primitive ID. Furthermore, according to the centre of gravity coordinates of the pixel, the vertex information in the *Vertex Buffer* (UV, Tangent Space, etc.) performs interpolations to obtain pixel-by-pixel information. In addition, according to the Material ID to index the relevant material information, the *Vertex Buffer* performs operations such as texture sampling and input to the lighting calculation link to complete the colouring. This type of method is sometimes called **Deferred Texturing**.

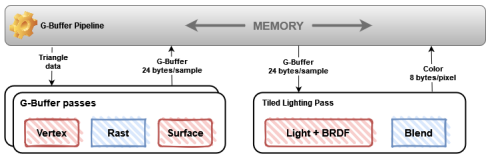

Fig. 12. Rendering pipeline based on *G-Buffer*.

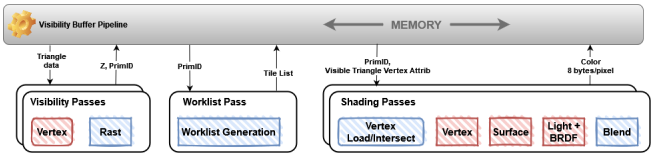

Fig. 13. Rendering pipeline based on *Visibility Buffer*.

Intuitively, the *Visibility Buffer* reduces the storage bandwidth of the information needed for shading. In addition, it delays the reading of geometric information and texture information related to lighting calculations to the shading stage, so those non-visible pixels on the screen do not need

to read this data, but only need to read the vertex position. For these two reasons, **the *Visibility Buffer* bandwidth overhead is greatly reduced compared to traditional *G-Buffer* in complex scenes with higher resolution**. However, maintaining global geometry and material data at the same time increases the complexity of engine design and reduces the flexibility of the material system[6].

4.3 Implementation

So, with the diverse array of papers, presentations, and documentation about *Nanite* it's possible to deduce the way it was implemented. In addition, a good foundation about the concepts and ideas that led to this technology was necessary, and so the previous sections offer a good insight on the subject. With this, *Nanite* at its core can be simply disassembled in two parts:

1) Optimization of vertex processing and optimization of pixel processing. The optimization of vertex processing is mainly based on the idea of GPU Drivel Pipeline;
2) Optimization of pixel processing with the help of *Visibility Buffer* combined with soft rasterization;

4.3.1 Instance Cull & Persistent Cull

In *Nanite*, each *Nanite Mesh* is cut into several Clusters in the preprocessing stage. Each cluster contains 128 triangles and the entire Mesh is organized into a tree structure in **Bounding Volume Hierarchy (BVH)** and each leaf node represents a cluster of triangles. After this, there are two steps of culling, including *frustum culling* and *occlusion culling* based on **Hierarchical Depth Buffer (HZB)**. Among them, there are two prevalent types of culling that *Nanite* uses: **Instance culling** and **Occlusion culling**.

Instance culling is one of the first things that happens in the at this stage of *Nanite*. It seems akin to a GPU form of *frustum* and *occlusion culling*. There is an instance of data and primitive data that is bound at this stage, this means that *Nanite* culls at the instance level first, and if the instance survives it starts culling at a finer-grained level, in this case the instances are the root nodes of the BVH tree of each Mesh. In addition, the `Nanite.Views` buffer provides camera info for *frustum culling* and the HZB is used for *occlusion culling*. With this, the HZB is generated in the current frame with the previous frame's visible objects [10]. The HZB is tested with the previous objects as well as anything new, and the visibility is updated for the next frame.

Both visible and non-visible instances are written into buffers. For the latter, it's needed to inform the CPU that a certain entity is occluded, and it should stop the processing until it becomes visible. The visible instances are also written out into a list of candidates.

The **Persistent culling** seems to be related to streaming. It is a fixed number of compute threads, suggesting it is unrelated to the complexity of the scene and instead checks some spatial structure of occlusion. In this stage, each mesh will be sent the root node of its BVH for the hierarchical elimination of the *Persistent culling* through the *Instance culling*. If a BVH node is eliminated its child nodes will not be processed.

6. To solve this it's possible to use certain Graphics APIs but some of them are yet not supported by all hardware.

4.3.2 Rasterization

After the culling stage, each cluster will be sent to different rasterizers, mentioned in subsection 4.2.1. In *Nanite*, large triangles and non-Nanite Meshes are still based on hardware rasterization, and small triangles are based on soft rasterization written by compute shaders. *Nanite's Visibility buffer* is, in essence, a texture (8 Bytes/Pixel) that stores two distinct channels, each with a size of 32 bits. These channels store the following information:

1) Triangle ID, $0 \approx 6$ bits of the R channel;
2) Cluster ID, $7 \approx 31$ bits of the R channel;
3) Depth, 32 bits of the G channel;

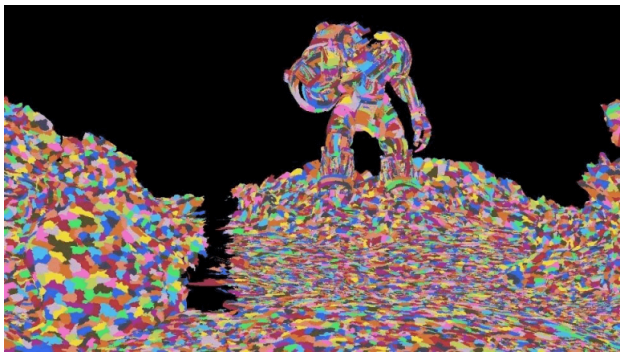

Fig. 14. Cluster ID channel is one of the information textures contained in the *Visibility Buffer* [11].

With this, the entire logic of the *soft* rasterization becomes clear.

1) Based on the scan line algorithm, each cluster starts a separate Compute Shader, calculates and caches all Clip Space Vertex positions to shared memory in the initial stage of the Compute Shader;
2) Each thread in the Compute Shader reads the corresponding Index Buffer of the triangle and the transformed Vertex Position;
3) The side of the triangle is calculated according to the Vertex Position;
4) Backside elimination and small triangle culling is performed;
5) Utilization of atomic operations to complete the `Z-Test`, and write the data into the *Visibility Buffer*;

It's also worth mentioning that, in order to ensure that the entire *soft rasterization* logic is concise, the Mesh **cannot have** any kind of deformation or skeletal animation.

4.3.3 Emit Targets

In order to ensure that the data structure is as compact as possible as well as reduce the read and write bandwidth, all the data required for *soft rasterization* is stored in a *Visibility Buffer* in order to mix the pixels generated by hardware rasterization on a scene. With this, it's finally needed to write the additional information within the *Visibility Buffer*

into the unified *Depth/Stencil Buffer* and *Motion Vector Buffer*. So, in this stage, there are several passes applied to the screen:

1) **Scene Depth/Stencil, Nanite Mask and Velocity Buffer**: This phase outputs three important quantities: Depth, Motion Vectors and Nanite Mask. The first two are standard quantities that are later used for things like TAA, reflections, etc. On the other hand, the *Nanite Mask* is used to indicate whether the current pixel is a Normal Mesh or a *Nanite Mesh*.

Fig. 15. Nanite Mask [11].

2) **Material Depth**: This is essentially a *Material ID* turned into a unique depth value and stored in a depth-stencil target. Effectively, there is one shade of grey per material. This is going to be used next as an optimization that takes advantage of *Early Z*.

Fig. 16. Material Depth Buffer [11].

4.3.4 Classify Materials & Emit G-Buffer

Now that the geometry pipeline has been explained in detail, it's time to talk about materials. Up until the *Visibility Buffer* generation and now, the frame spends a great amount of runtime doing other tasks: light grid, sky atmosphere, and rendering the *G-Buffer* as it would normally. This really drives home the separation between geometry and materials that the *Visibility Buffer* aims for. With this, the most important steps of this stage are inside the **Classify Materials** and **Emit G-Buffer**.

In the **Classify Materials** stage, the material classification pass runs a Compute Shader that analyses the full-screen *Visibility Buffer*. This is very important for the next pass. The output of the process is $20 \times 12 (= 240)$ pixels in a `R32G32_UINT` texture called *Material Range* that encodes the range of materials present in the 64×64 region represented by each tile.

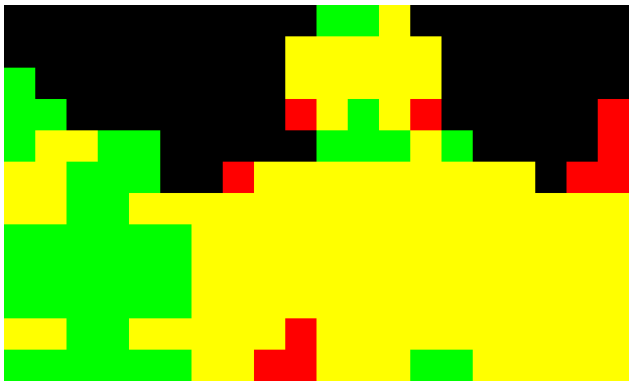

Fig. 17. Material ID Range [11].

At this point, *Nanite* finally joins visibility and material together. The objective of the *Visibility Buffer* is turning triangle information into surface properties. Unreal allows users to define arbitrary materials to surfaces, so how efficiently it manages that complexity? It uses an **Emit G-Buffer**.

That being said, *Nanite* does one draw call per material ID, and every draw call is a full-screen quad chopped up into 240 squares rendered across the screen. As it was mentioned before, the material range texture is 240 pixels, so every quad of this full-screen draw call has a corresponding texel[7]. The quad vertices sample this texture and confirm whether the tile is a relevant tile; for example, whether any pixel in the tiles has the material corresponding to the material that's going to be rendered. If not, the x coordinate will be set to NaN and the whole quad is discarded.

According to the documentation, the system uses 14 bits for *Material IDs*, for a total of 16384 maximum materials. A constant buffer sends the *Material ID* to the vertex shader so that it can check whether it's in range.

Furthermore, it's necessary to remember that a material depth texture was created. This texture is where every *Material ID* is set to be a certain depth. These quads are outputted to the depth represented by their material and the depth mode is set to equal, so the hardware can then very quickly discard any pixels that aren't relevant. As an extra step, the engine previously marks the stencil buffer as pixels that have *Nanite Geometry* and regular pixels, also used for *Early Z/Stencil* optimizations.

7. A texel is a term used in computer graphics to refer to a texture element.

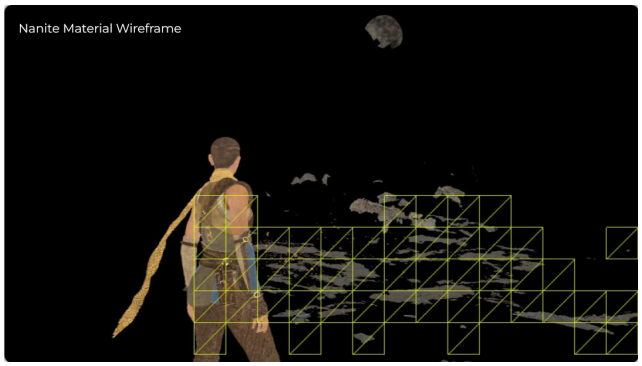

Fig. 18. Nanite Material Wireframe [11].

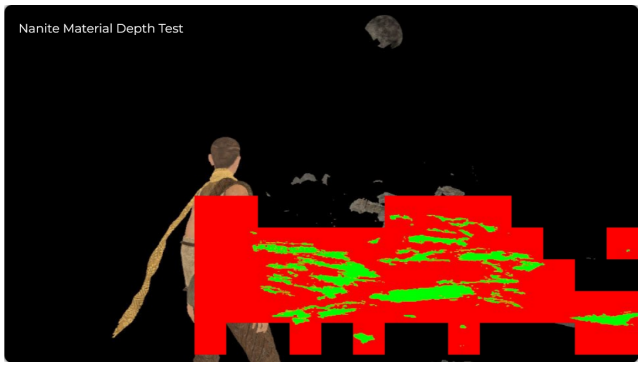

Fig. 19. Nanite Material Depth Test [11].

As it can be noticed, some quads are completely red, which would lead to the conclusion that they are completely discarded by the vertex shader. However, it seems that the material range texture is exactly what it says: a range of materials covered by that tile. If a material happens to be "in the middle" but none of the pixels have it, it will be considered as a candidate even though the depth test will discard it completely later. In any case, that's the main idea – the same process as shown in the images is repeated until all materials are processed. The final *G-Buffer* is show as tiles in the image below.

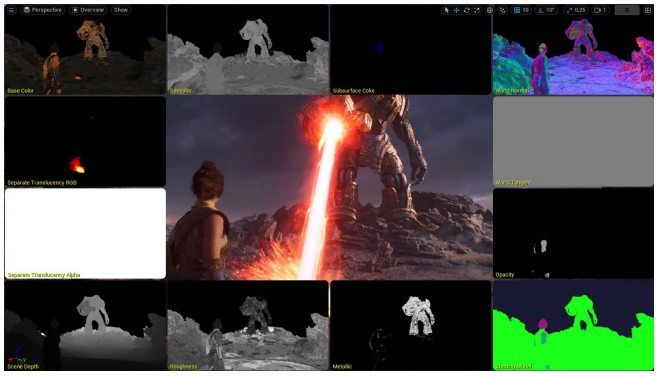

Fig. 20. *G-Buffer* shown as tiles [11].

5 The Power of Nanite

In this section, we will explore some features of Unreal Engine's Nanite. By leveraging the power of virtualized textures and advanced algorithms, *Nanite* allows developers to create highly detailed and realistic environments without worrying about the limitations of traditional polygon-based approaches. Let us delve into the key features of this cutting-edge technology and discover how it allows developers to achieve unprecedented levels of visual fidelity and immersion.

5.1 Basic features

Geometry: in terms of geometry, *Nanite* can be enabled on Static Meshes and Geometry Collections[8]. Furthermore, a *Nanite*-enable mesh can be used with the following Component types:

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

One of the limitations *Nanite* has in regard to geometry is that it can only be used on rigid meshes. Basically it can't be used on animated models like playable characters but works great on static objects that are often representative of a larger sample of geometry present in a scene.

Materials: in regard to this matter, *Nanite* is only capable of supporting Blend Mode set to `Opaque`. In computer graphics and image editing, blend modes are used to determine how pixels of one layer or object are combined with the pixels of another layer or background. When it comes to opaque materials, the blend mode typically used is `Normal`[9] or `Normal(Blend)`.

5.2 Hierarchical Dynamic Level of Detail - LODs

Traditionally, renderers use techniques like pre-determined levels-of-detail[10] and prebaked textures to optimize performance and manage memory, usually artists are heavily involved in this process. However, these methods often involve compromises in visual fidelity or require significant manual effort.

In *Nanite*, **LOD**s are instead determined at a sub-mesh level, with much less artist involvement. That being said, *Nanite*, instead of relying on pre-defined LOD meshes and textures, it allows the rendering of virtualized micropolygons.

So, a whole tree, or a directed acyclic graph (**DAG**), of 128-triangle clusters is computed for each mesh at asset import time. The tree has $<= 128$ triangles per node and the children of each node give a more in-depth look at that node. Then a "cut" of this tree is determined at runtime on the GPU and based on this "cut" a set of triangle clusters

8. Data structure and system used for handling and simulating complex physics interactions involving multiple pieces of geometry.

9. Usually, the normal mode, or the default mode, replaces the pixels of the underlying layer with the pixels of the top layer, without any blending or transparency effects therefore it stays opaque.

10. A good example is the different levels of subdivision when making the model. Each subdivision has a level of detail different from others.

will be rendered. Culling also happens on a per-cluster, not per-mesh, basis.

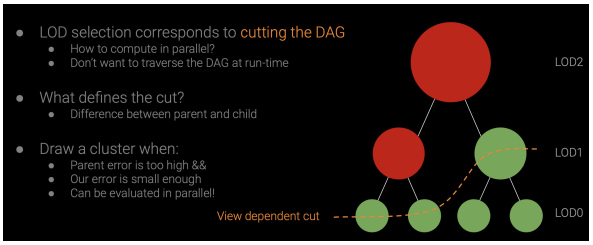

Fig. 21. Hierarchical Level of Detail [12]

When all this comes together well, a system with a very consistent number of triangles rendered to the screen per-frame and very little overdraw, also known as wasted shader work, is achieved. This efficiency is what lets *Nanite* achieve high detail at decent `FPS` counts.

If not planned ahead, there might be problems with gaps between clusters, especially when they are next to each other in *LOD* level > 1. There might also exist visual popping when switching between *LOD* levels. It is also possible to have problems with memory since there is a massive amount of data (each mesh has many tree clusters) and it needs a lot of GPU work to pre-shade them.

With this, how does this cluster-based visibility culling happen on a GPU?

5.2.1 Mesh simplification and hierarchy construction

Firstly, a ton of work went into determining how triangles should be clustered and hierarchically arranged at asset-import-time so that cracks can't be observed, and yet it's still possible to efficiently determine a cut of this hierarchy at runtime. This involves complex graph partitioning and multidimensional optimization.

It is also worth pointing out that the most detailed view of a mesh in *Nanite*: the leaf nodes in the cluster graph are exactly the same triangles present in the original asset; *Nanite* isn't capable of optimizing away detail, instead it displays simplified triangle clusters, higher nodes, only when the detail change isn't visually perceptible.

5.2.2 LOD N vs LOD N+1

One of the most novel contributions of *Nanite* is the calculation of good perceptual error metrics between *LODs*. Basically, it is needed to have the knowledge of how much worse a simpler triangle cluster is than its child (i.e. more detailed) clusters in order to do good *LOD* selection. Amazingly, if the error difference is < 1 pixel and some type of temporal antialiasing is present, there won't be any popping noticeable to the naked eye.

5.2.3 Cutting the cluster hierarchy DAG (LOD Selection)

A good cluster hierarchy and *LOD* error calculations come together at runtime with view-based information[11] to determine the "cut" of the cluster *DAG*, which is the *LOD* selection of that frame.

11. In GPU compute, using a bounding volume hierarchy (BVH) and custom parallel task system.

5.3 Fallback Mesh

When *Nanite* is enabled for a Static Mesh, it creates a simplified version of the intricate mesh that can be accessed and utilized in situations where *Nanite* data is not applicable. This simplified mesh, known as the **Fallback Mesh**, serves as a substitute when *Nanite* rendering is not supported. It is particularly useful in scenarios where it is impractical to use the high-detail mesh, such as when complex collision detection is necessary, for example.

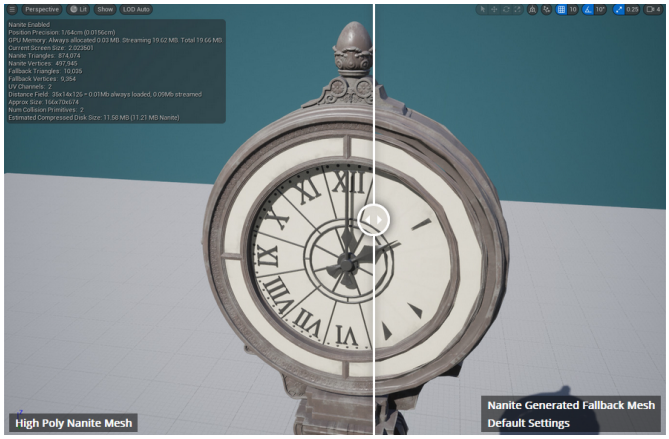

Fig. 22. *Nanite* Generated Fallback Mesh [13].

It's worth noting that there is the possibility of adjusting values like the **Fallback Relative Error** to specify how much of the original detail is retained from the original source mesh, and Fallback Percentage to set how much of that detail is used.

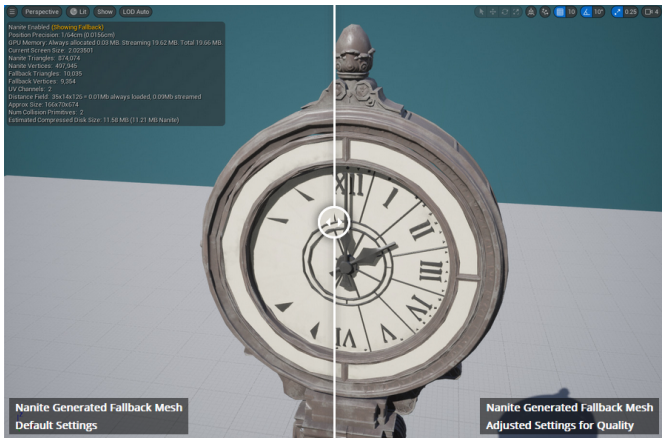

Fig. 23. Fallback Mesh with default settings vs adjusted settings [13].

5.4 Streaming virtual geometry

For memory management, *Nanite* aggressively ejects unused clusters from working memory and stream in new ones from disk. For this reason, a decent SSD is basically required for *Nanite* to work. This is what's called "virtual geometry", analogous to "virtual texturing". Formatting, compressing and deciding what to stream is complex. About approximately 1M input asset triangles become almost $11Mb$ compressed *Nanite* data on disk [14].

6 EXPERIMENTATION

After this extensive analysis on the technology, it's worth seeing it in action. With this, some test scenarios were developed in order to show the true power of *Nanite*.

Firstly, all tests were made on a machine with the following specifications:

- **CPU:** AMD Ryzen 7 5800H with Radeon Graphics 3.20 GHz
- **RAM:** 32.0 GB (31.4 GB usable)
- **GPU:** NVIDIA GeForce RTX 3080 Laptop GPU

With the help of UE5, a new empty scene was created with the Unreal Started Content contained in it. After creating a new landscape using the mode `Landscape`, in order to place any object on the scene, it was chosen the following model as the test model.

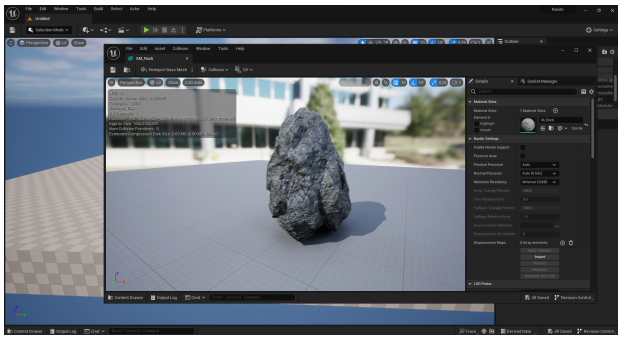

Fig. 24. Test_1 Model – Starter Content Rock.

After this, the mode `Foliage` was used to place numerous instances of the same model onto the landscape. With this, the testing scene contained **160k instances** of the chosen mesh for the tests. It's also worth mentioning that the command `Stats fps` was used in order to check the fps count of the scene and comparing results. Without *Nanite*, the scene obtained a fps count of **20.64 fps**. So, it can be seen that, even the machine being fairly powerful, the number of instances reduces greatly the frames per second of this scene.

To solve this problem, *Nanite* was enabled and applied to the test mesh. This will apply *Nanite* to all the instances of the mesh that are present in the terrain. To check if *Nanite* was enabled properly, it was inspected if one of the information textures that *Nanite* creates was present.

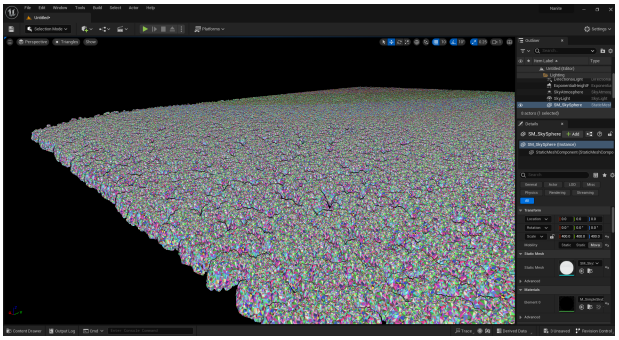

Fig. 25. Test_1 Triangle ID texture is present, confirming that *Nanite* was enabled properly.

Now, with *Nanite* enabled, the scene achieves an astounding fps count of **60 fps** (possibly capped by UE5 system definitions).

To further prove that *Nanite* can do what it sets out to do, a second Test scenario was made. This time it was used a third party static mesh in order to prove that *Nanite* works on external meshes and not just the ones made available by Epic Games.

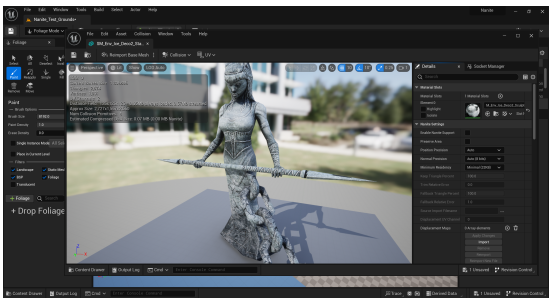

Fig. 26. Test_2 Model – Infinity Blade: Ice Lands Statue.

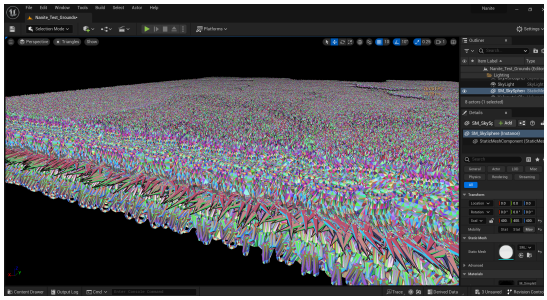

Fig. 27. Test_2 Triangle ID texture is present, confirming that *Nanite* was enabled properly.

In this test scenario, the geometry of the mesh is much more detailed than the previous one. With this, without *Nanite*, it can only achieve **14.56 fps**, having an instance count of $89.5k$. With *Nanite*, it achieves a fps count of **58.20 fps**.

This confirms that *Nanite* can indeed improve the performance of a scene with numerous static meshes contained in it. To further illustrate this enhancement in performance, the subsequent graph is presented.

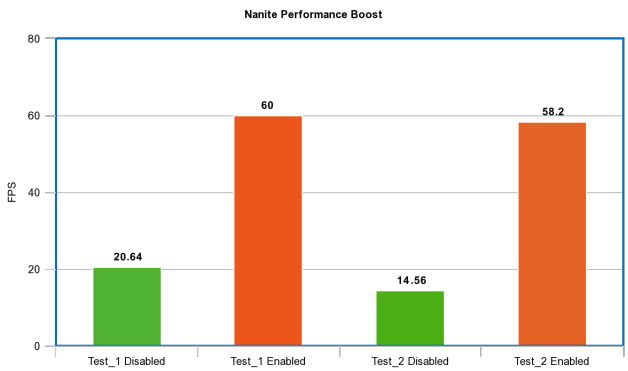

Fig. 28. *Nanite* performance comparison.

7 Conclusions

Unreal Engine's Nanite represents a significant advancement in real-time rendering technology, pushing the boundaries of what is possible in terms of visual fidelity and scalability. By introducing virtualized micropolygon geometry, *Nanite* eliminates the limitations of traditional rasterization techniques, allowing for the rendering of highly detailed scenes with unprecedented levels of geometry complexity. This breakthrough technology not only revolutionizes the way developers create and optimize assets, but also offers a new level of immersion and realism for players.

The state-of-the-art nature of *Nanite* is evident in its ability to render massive scenes without sacrificing performance or compromising on visual quality. Leveraging the power of virtualized micropolygons and material streaming, *Nanite* provides an efficient solution for rendering geometry-rich environments, making it an invaluable tool for game developers, filmmakers, and other industries that require high-fidelity rendering.

The implementation of *Nanite* showcases the remarkable engineering and design efforts undertaken by Epic Games. The integration of virtualized micropolygon geometry into the existing Unreal Engine ecosystem demonstrates a thoughtful and comprehensive approach to delivering cutting-edge technology to developers. *Nanite*'s seamless integration with other features such as *Lumen*, the dynamic global illumination system, further enhances its capabilities and opens up new possibilities for creating visually stunning experiences.

While *Nanite* undoubtedly represents a breakthrough in real-time rendering technology, there are still areas for further exploration and refinement. As the technology evolves, addressing potential challenges such as memory management and optimizing performance for a broader range of hardware configurations will be crucial. Additionally, continued research and development efforts can expand *Nanite*'s potential applications beyond gaming, into fields such as architectural visualization, virtual reality, and simulations.

In conclusion, *Unreal Engine's Nanite* is a groundbreaking technology that redefines the state of the art in real-time rendering. With its ability to handle massive scenes and deliver unprecedented levels of detail, *Nanite* empowers developers to create immersive and visually stunning experiences. As the technology continues to advance, it can be expected *Nanite* to play an increasingly significant role in shaping the future of real-time graphics.

References

[1] D. Eberly, "3d game engine design: A practical approach to real-time computer graphics," 2006.

[2] "Tron – retro gamer," last accessed: 2023-06-19. [Online]. Available: https://www.retrogamer.net/retro_games80/zaxxon/

[3] "Insights: How reflections in games are made," last accessed: 2023-06-19. [Online]. Available: https://80.lv/articles/insights-how-reflections-in-games-are-made

[4] K. Rune, B. Wihlidal, and S. Graham, "Advances in real-time rendering in 3d graphics and games," 2021.

[5] E. Mora Ramirez, "Radiopharmaceutical dosimetry in targeted radionuclide therapy," 03 2019.

[6] "Subdivision surface modifier," last accessed: 2023-06-19. [Online]. Available: https://docs.blender.org/manual/en/latest/modeling/modifiers/generate/subdivision_surface.html

[7] "Normal vs displacement vs bump maps: Differences and when to use which," last accessed: 2023-06-19. [Online]. Available: https://www.cgdirector.com/normal-vs-displacement-vs-bump-maps/

[8] "Meshlet priors for 3d mesh reconstruction," last accessed: 2023-06-19. [Online]. Available: https://research.nvidia.com/sites/default/files/styles/wide/public/publications/teaser_1.jpg?itok=j8XqbjbM

[9] M. Valient, "Deferred rendering in killzone 2," p. 18, last accessed: 2023-06-19. [Online]. Available: https://www.slideshare.net/guerrillagames/deferred-rendering-in-killzone-2-9691589

[10] K. Rune, B. Wihlidal, and S. Graham, "Advances in real-time rendering in 3d graphics and games," 2021.

[11] IGN, "Unreal engine 5 – official valley of the ancient tech showcase," May. 2021. [Online]. Available: https://www.youtube.com/watch?v=SeY-ousiCPg

[12] K. Rune, B. Wihlidal, and S. Graham, "Advances in real-time rendering in 3d graphics and games," p. 65, 2021.

[13] "Nanite virtualized geometry," last accessed: 2023-06-19. [Online]. Available: https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/

[14] K. Rune, B. Wihlidal, and S. Graham, "Advances in real-time rendering in 3d graphics and games," p. 144, 2021.