

Bachelor Project



**Czech
Technical
University
in Prague**

F3

**Faculty of Electrical Engineering
Department of Computer Graphics and Interaction**

Rendering Detailed Models in Unreal Engine

Dan Juříček

**Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2022**

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Juříček** Jméno: **Dan** Osobní číslo: **495555**
Fakulta/ústav: **Fakulta elektrotechnická**
Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**
Studijní program: **Otevřená informatika**
Specializace: **Počítačové hry a grafika**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Zobrazování detailních modelů v Unreal Engine

Název bakalářské práce anglicky:

Rendering Detailed Models in Unreal Engine

Pokyny pro vypracování:

Zmapujte existující metody zobrazování detailních modelů v reálném čase. Seznamte se s technologií Nanite implementovanou v Unreal Engine 5 a podrobně ji popište.

Vytvořte vizualizace, které budou ilustrovat vnitřní fungování technologie Nanite. Konkrétně vizualizujte proxy mesh a cluster v závislosti na vzdálenosti od kamery a parametrech dostupných v nastavení Nanite. Získané poznatky podrobně popište.

Vytvořte nejméně čtyři různé testovací scény, které budou demonstrovat možnosti technologie Nanite. Scény vytvořte jak ve formě klasické LOD reprezentace tak v reprezentaci Nanite. Porovnejte vytvořené varianty scén z hlediska paměťové náročnosti a rychlosti zobrazování na nejméně dvou různých platformách. Nejméně pro jednu scénu vytvořte škálovatelnou variantu, která umožní vyhodnotit závislost rychlosti a kvality zobrazování na celkovém množství trojúhelníků ve scéně.

Seznam doporučené literatury:

- [1] Lindstrom, Peter, et al. Real-time, continuous level of detail rendering of height fields. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. 1996.
- [2] De Floriani, Leila, Leif Kobbelt, and Enrico Puppo. A survey on data structures for level-of-detail models. Advances in multiresolution for geometric modelling. Springer, Berlin, Heidelberg, 49-74, 2005.
- [3] Dietrich, Andreas, Enrico Gobbetti, and Sung-Eui Yoon. Massive-model rendering techniques: a tutorial. IEEE Computer Graphics and Applications 27.6, 20-34, 2007.
- [4] Yoon, Sung-Eui, Christian Lauterbach, and Dinesh Manocha. R-LODs: fast LOD-based ray tracing of massive models. The Visual Computer 22.9, 772-784, 2006.
- [5] Áfra, Attila T. Interactive ray tracing of large models using voxel hierarchies. Computer Graphics Forum. Vol. 31. No. 1. Oxford, UK: Blackwell Publishing Ltd, 2012.
- [6] Nanite Virtualized Geometry. <https://docs.unrealengine.com/5.0/en-US/RenderingFeatures/Nanite/>

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **09.02.2022**

Termín odevzdání bakalářské práce: **20.05.2022**

Platnost zadání bakalářské práce: **30.09.2023**

doc. Ing. Jiří Bittner, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Podpis studenta

Acknowledgements

I would firstly like to express my thanks to my parents and family, for the only reason I was able to study until now, is thanks to them. I would then like to thank my friends, with whom I could complain or talk about any problems I experienced when writing this thesis. I would namely want to thank my good friend Dominik Dinh, for having discussions with me throughout the creation of this thesis and for letting me use his models for my thesis. Lastly I would like to thank doc. Ing. Jiří Bittner, Ph.D for his patience, guidance, and for motivating me to finish this project.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Abstract

The aim of this thesis is to analyze the newly released technology from Unreal Engine 5 called Nanite Virtualized geometry and to compare its capabilities with the traditional optimization technique known as discrete level of details. This thesis presents a brief introduction to optimization techniques and an analysis of the functionality of Nanite. Several test scenes were created with results comparing Nanite technology in with various settings ...

Keywords: Nanite, Unreal Engine, Level of Details

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Cílem této práce je analýza nově vytvořené technologie Unreal Engine 5 nazývaný Nanite Visualed Geometry. A následné porovnání rozdílů mezi použitím tradiční optimalizační technikou nazývanou diskrétní úrovně detailů. Tato práce představuje různé optimalizační techniky a analyzuje funkcionalitu technologie Nanite. Vytvořilo se několik testovacích scének s porovnáváním Nanite technologie s různými nastavení ...

Klíčová slova: Nanite, Unreal Engine, Level of Details

Překlad názvu: Zobrazování detailních modelů v Unreal Engine

Contents

1 Introduction	1		
1.1 Goal	1		
2 Rendering	3		
2.1 Polygonal Mesh	3		
2.1.1 Triangle mesh	3		
2.2 Model optimization	5		
2.2.1 Types of LOD's	5		
2.2.2 Discrete LOD	5		
2.2.3 Continuous LOD	5		
2.2.4 View-Dependent LOD	6		
2.2.5 Overview of LODs	6		
2.2.6 Normal maps	6		
2.3 Mesh simplification	7		
2.3.1 Edge collapse	7		
2.3.2 Triangle collapse	8		
2.4 Error Metrics	8		
2.4.1 Quadric Error Metrics	8		
2.4.2 Space-screen error	9		
2.5 Imposters	9		
2.6 Culling	9		
2.6.1 View Frustum Culling	9		
2.6.2 Occlusion Culling	10		
2.6.3 Instancing	10		
3 Nanite	13		
3.1 Rendering Pipelines	13		
3.2 Forward Rendering	14		
3.3 Deferred Rendering	14		
3.4 Nanite culling	14		
3.5 Visibility Buffer	15		
3.6 Nanite LOD creation	16		
3.6.1 Cluster Hierarchy	17		
3.7 Cluster Creation	17		
3.8 Runtime LOD selection	19		
3.9 Two-Pass Occlusion Culling	20		
3.10 Rasterization	21		
3.11 RenderDoc	21		
4 Nanite in practice	25		
4.1 Nanite support	25		
4.1.1 Unsupported Nanite settings	25		
4.1.2 Aggregate geometry	25		
4.1.3 Concerning foliage	26		
4.2 Nanite inside Unreal Engine 5	26		
4.3 Viewport	27		
4.4 Details Panel	27		
4.5 Nanite mesh settings	27		
4.5.1 Overdraws	29		
4.6 Standard optimization workflow in Unreal Engine	30		
4.7 Testing scenes	31		
5 Statistics	33		
5.1 Data graphs	33		
5.2 User feedbacks	33		
5.3 Drawcall test	34		
5.4 Outdoor scene	35		
5.5 Indoor scene	37		
5.6 Foliage scene	39		
5.7 Scene Playground	43		
5.8 Test results	45		
6 Conclusion	47		
Bibliography	49		
A Electronic Appendix	53		
B User Manual	55		

Figures

2.1 Image of a polygon mesh using quads in Blender.	4
2.2 Difference between detailed geometry and a simplified geometry with a normal map. (Source: [4]) ..	7
2.3 A model captured with 3 different LODs. (Source: [26]).....	7
2.4 Representation of an edge collapse operation (Source: [3]).....	8
2.5 Difference between occlusion culling in Nanite (right) and Static Mesh (left).	11
3.1 Visualization of the GBuffer in Indoor scene with Scene visualisation (top left) using RenderDoc.	15
3.2 Difference in occlusion between HZB (left) and Hardware Occlusion Queries(right).	16
3.3 Visualization of a simple build operation for a cluster.	18
3.4 Difference between choosing the children group of clusters(left) and choosing the parent group of clusters (right). The boundaries drawn with white colors do not change, however the number of clusters, therefore triangles do.....	20
3.5 Screenshot of the RenderDoc application.	22
3.6 One frame capture of the indoor scene in RenderDoc.	22
3.7 Simple visualization of the Nanite render pipeline. This pipeline has to pass two times.	23
4.1 Visualization of the static mesh editor with a hand made model without optimization.	27
4.2 Table with trim error applied. ...	29
4.3 A scene with Nanite Visualization turned on.	30
5.1 Graph depicting the frametime with increasing objects that have high drawcalls.	34
5.2 Image of the plane with 64 unique meshes. Every square represents one unique material.	35
5.3 Image of the outdoor scene.	37
5.4 Graph depicting the difference in time for rendering a frame in a camera flythrough in the Outdoor scene.	37
5.5 Graph depicting the difference in time for rendering a frame in a camera flythrough in the Indoor scene.	40
5.6 Image of the Indoor scene.	41
5.7 Graph depicting the difference in time for rendering a frame in a camera fly-through in the Foliage scene.	41
5.8 Image of the Foliage scene with the difference of Nanite(left) and distance culled LOD(right)	42
5.9 Graph depicting the framerate according to the increasing number of objects.....	43
5.10 Image of the stress scene with 1 million objects being rendered. ...	44

Tables

4.1 Table of hardwares used to produce data in the test scenes....	31
5.1 Table of meshes present in Drawcall scene.	34
5.2 Table of performance for Drawcall scene. The values were captured after all the meshes had been rendered..	35
5.3 Table of meshes present in Outdoor scenes.	36
5.4 Table of performance for Outdoor scene.	36
5.5 Table of meshes present in Indoor scenes.	38
5.6 Table of performance for Indoor scene.	39
5.7 Table of meshes present in Foliage scenes.	40
5.8 Table of performance for Foliage scene.	41
5.9 Table of performance for Stress scene.	43
5.10 Table of meshes present in Playground scenes.....	44
5.11 Table of performance for Stress scene.	44



Chapter 1

Introduction

Since the beginning of computer graphics, programmers have struggled with balancing complexity and representation for several years up until date. With better hardware coming out each year, the boundary of model representations in scenes has been pushed to its limits with more detailed models coming out. However, even to this date, hardware cannot keep up with the demands that detailed models need. In order to make up for this, several optimization techniques were invented to reduce the demands made on hardware.

With the recent release of the early access for Unreal Engine 5 in May 2021, they also introduced a new technology called Nanite virtualized geometry. According to Unreal Engines Documentation, to quote Nanite's definition: "Nanite is Unreal Engine 5's new virtualized geometry system that uses a new internal mesh format and rendering technology to render pixel scale detail and high object counts. It intelligently does work on only the detail that can be perceived and no more. Nanite's data format is also highly compressed, and supports fine-grained streaming with automatic level of detail." [9]. This would mean that artists no longer need to be concerned about polycounts, drawcalls, or memory.



1.1 Goal

In this thesis, we will be researching how Unreal Engine represents mesh and how Unreal Engine represents mesh with Nanite. We will also look at how Nanite fairs, comparing the differences between using Nanite-ready models and manually prepared optimized models and unoptimized models, and how it affects the framerate, visualization, and memory of a scene. For testing purposes, several scenes will be created using Unreal Engine 5, where Nanite will be used, and traditional optimization techniques will be used on a larger scale scene and a smaller scale scene.

Chapter 2

Rendering

Nanite is a complex technology that uses several optimization techniques to work. In this thesis, we shall only touch upon the very basics and then show some results comparing the capabilities of Nanite with the most popular and widely used optimization technique known as "Level of detail." In order to further examine these methods, we first must have a basic understanding of how models are represented.

2.1 Polygonal Mesh

There are several ways how to render a model in 3D graphics. On a broader scale, representations can be divided into boundary representation, discrete representation, and procedural modeling. However, due to a pretty voluminous topic, we will expand only on boundary representations, more precisely polygonal meshes, since Unreal Engine renders objects using triangles that make up polygonal meshes. For more information concerning other representations, please refer to this book [28]

Polygonal meshes, which will be referred to as meshes for short, use polygons as a base. Polygons, which we will later refer to as faces, are shapes created from sets of vertices. The minimum number of vertices needed to create a face is three, which creates a triangle shape, and four vertices create a quad. We can create more complex models by grouping and connecting several polygons together. By connecting triangular polygons, we create something known as triangle meshes.

2.1.1 Triangle mesh

Triangle meshes are probably the most used due to their simplicity and efficiency. Several efficient algorithms exist that can rapidly render triangles onto a screen, and hardware natively supports triangle rasterization. Furthermore, since triangles are the simplest form of a polygon, any general, more complex polygon can be reduced to a triangle form, which also ensures planar surfaces.

A downside to triangles is that texturing and modeling using triangles is very unintuitive. It is much easier to create models using quads than triangles in a 3D modeling program. However, since quads take longer to render and

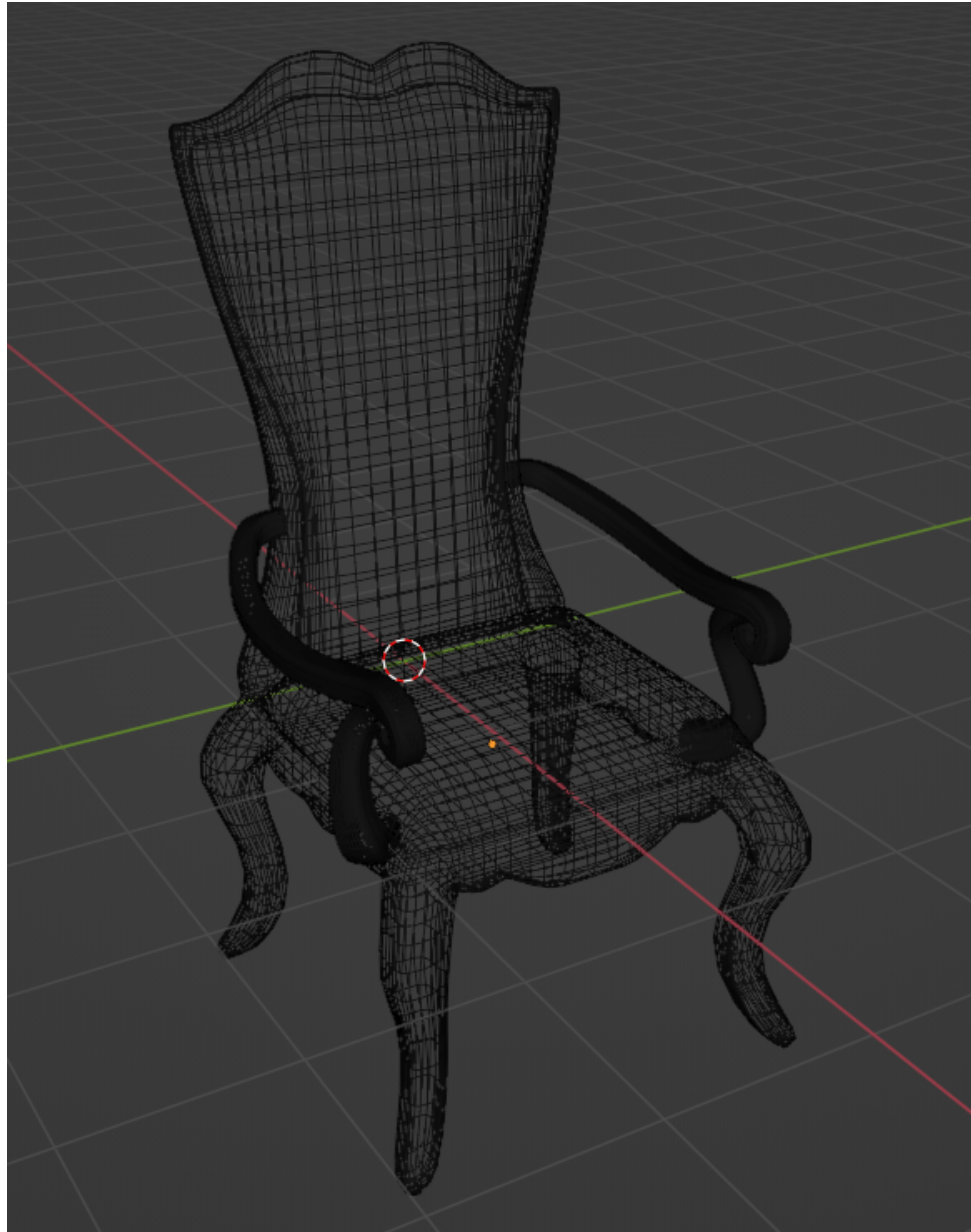


Figure 2.1: Image of a polygon mesh using quads in Blender.

some drivers might not even support quad rendering, it is much faster and easier to split the quads into triangles and render the triangles instead. This applies to any general polygon.

■ 2.2 Model optimization

Model optimization is a relatively large area as several techniques are used simultaneously to achieve the best results. Because of this, only the most used techniques shall be discussed here, such as level of details, occlusion culling, imposters, and instancing.

■ 2.2.1 Types of LOD's

This chapter will briefly review one of the most used model optimization techniques known as "Levels of detail" or "LOD" for short. For simplicity, level of details will be referred to as LODs throughout this paper. LODs can be generally categorized into three main methods.

■ 2.2.2 Discrete LOD

A traditional and widely used method is a discrete level of detail (DLOD for short) which James H. Clark proposed in 1976 [5]. The basic concept of DLOD lies in creating several objects with varying data complexion (with several level of details) pre-render. Depending on their distance, adequate LODs would be loaded into the scene during run-time. Since more distant objects would have fewer polygons to process, the rendering speed and performance would increase. In the past, this would mean an artist would have to create several models with different complexions. However, throughout the years, various simplification algorithms have been made to ease the workload of artists.

A downside to DLOD is that due to ready-made sets of models, there is no way to know from which angle the model will be viewed. Therefore the model simplification has to be made on the whole model. This might result in a phenomenon called "visual popups" if mishandled. Visual pop-ups happen when the more detailed model is switched to the less detailed model. If either the model has been too simplified or the distance was not far enough, the viewer might be able to distinguish between different models.

■ 2.2.3 Continuous LOD

The idea of Continuous LODs is to quickly render coarse meshes that, through a series of reconstructions, progressively recreate the initial mesh.

The most famous continuous LOD is a progressive mesh [19]. Unlike DLOD, continuous LOD functions at run-time. This method creates a data structure offline, also known as the progressive mesh. Taking in the highest detailed mesh, a series of suitable decimate algorithms, known as the edge-collapse[19]

is then applied to continually remove vertices and faces, resulting in a much simpler model, until the simplest desired model is left, also known as the base mesh. Then through a series of inverse operations for edge-collapse, known as vertex-splits, the model is then reconstructed from its simple shape to the formal mesh.

Therefore, for a progressive mesh, we store the simplest model and the sequence of vertex-splits, that create a more complex mesh from the previous simpler mesh. This forms a hierarchical structure that helps to create a model in the chosen level of detail. This, in return, provides better granularity and better fidelity since the level of detail can be specified precisely and not through a set of ready-made models. As a result, lesser polygons are needed, allowing said polygons to be used for other objects.

■ 2.2.4 View-Dependent LOD

View-Dependent LODs [20] is an abstraction of Continuous LODs. It can dynamically select the most appropriate level of detail for the current view by only using the vertex split operations where it is needed. For example, one object can be sectioned into several parts, each with its own level of detail that will vary according to what the person sees in the current view. This further increases granularity and fidelity and is especially adequate for massive objects with many details. This sort of technique is mostly used for terrains due to its drawbacks.

■ 2.2.5 Overview of LODs

Despite several advantages of Continuous and View-Dependent LODs, the traditional, discrete LOD still remains the most common method. The static method of creating several level of details beforehand is simple and best for most graphical hardware. The extra processing overhead and memory needed to create a Continuous or View-Dependent LOD is not suitable for simple objects. However, Nanite has found a way to effectively and cheaply use View-Dependent LODs.

■ 2.2.6 Normal maps

A normal map is a texture mapping technique used to add details to meshes by faking how light bounces on the object. It is a texture map where each pixel represents the X,Y,Z vectors in an RGB value. This helps add details to the mesh without the need to add extra layers of geometry. A common technique used is to create a low-poly model first that will then use the normal map. Afterward, create a high-poly version of this model and bake the normal map. The baking process is to recalculate and bend the normals of the low-poly model to the high-poly models. This method can have very impressive results, as can be seen in Figure 2.2

The downside to normal maps is that it becomes very distinguishable when observed from up close that a texture has been used.

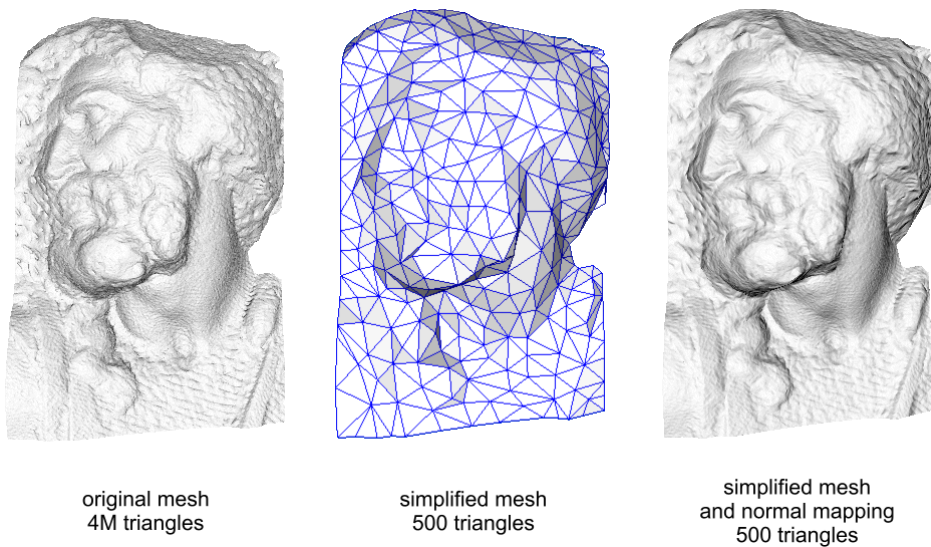


Figure 2.2: Difference between detailed geometry and a simplified geometry with a normal map. (Source: [4])



Figure 2.3: A model captured with 3 different LODs. (Source: [26]).

2.3 Mesh simplification

Early practitioners created LODs manually by either recreating the model with fewer polygons or deleting certain mesh parts. This proved to be a laborious task, and as such, several simplification algorithms were invented to speed up this process. Only two widely used operations will be introduced here. For more information and a detailed description of different operations, it is highly recommended to read *Level of Detail for 3D Graphics* [22].

2.3.1 Edge collapse

A widely used method was proposed by Hoppe [19] called edge collapse. This method takes in two vertices v_a , v_b that create an edge and collapses them into a single new vertex v_{new} . By doing so, it removes two vertices and also removes the triangle that was made up of said vertices. The inverse operation

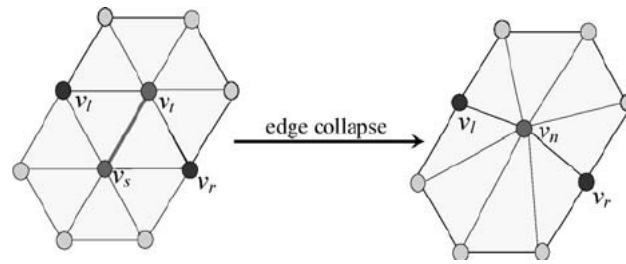


Figure 2.4: Representation of an edge collapse operation (Source: [3])

is called vertex split, which adds two vertices that make up a triangle. (2.4) The method can be further extended to variations known as half-edge collapse and full-edge collapse. In the half-edge collapse, one of the vertices that create the edge is considered an end point, to where the edge collapses to.

■ 2.3.2 Triangle collapse

A very similar method to edge collapse is a triangle collapse, which simplifies a mesh by collapsing a whole triangle, made of out vertices v_a , v_b , v_c , to a new single vertex v_{new} [27]. The v_{new} is either one of the three older vertices or a newly computed vertex. A triangle collapse can be replicated by creating two edge collapses. However, it requires less memory. Due to collapsing more vertices at once, triangle collapses are less adaptable than edge collapses.

■ 2.4 Error Metrics

When using simplification algorithms or rendering level of detail, the visual aspect of the output quality is also essential to maintain visual fidelity. Error metrics are used to determine how much is a simplified model different from the original main model.

This used to be done by the modeler, who could determine how simple an object might look and how far it should be from the camera to make it indistinguishable from the highly detailed object. This approach worked perfectly since it relied on the visual feedback of a human. However, for larger scenes, this laborious would take too long.

Therefore several autonomous methods were created to ease the workload of modelers.

■ 2.4.1 Quadric Error Metrics

QEM, for short, is a method introduced by Garland and Heckbert [12] that measures how much a simplified mesh has changed as opposed to the original mesh. Briefly, QEM uses an abstraction of the edge-collapse simplification method called iterative vertex contraction. Like edge-collapse, it takes in two vertices and contracts them into one new vertex. However, this does not apply to vertices with edges only. This method also contracts non-edge pairs

of vertices that will remove one vertex and then join previously unconnected regions. The error is then calculated as a sum of squared distances from a vertex for each vertex.

■ 2.4.2 Space-screen error

This method is used for determining which LOD should be rendered into a screen. It can be viewed as a method to calculate screen pixel difference when an LOD is switched. In order to roughly estimate a screen space error, a set of viewing parameters and a mesh with its calculated error metrics is needed. From all this, we can calculate an error on the screen space in pixels and according to a threshold, determine if the object in a lower detail is suitable. [7]

■ 2.5 Imposters

Another interesting concept in computer graphics concerning LODs is image-based LODs, also known as either stand-ins or imposters[22]. Their main goal is to replace geometric mesh LOD with a singular image of the object placed onto a flat polygon, usually a quad. Then by using an alpha map texture, we can achieve a semi-transparent image mimicking an object. For most purposes, we would want to have the imposter facing the camera. This technique is called billboarding and is generally used for foliage, for example, grass or tree leaves. Transparency in computer graphics is expensive work and time-consuming, but due to the convenience of imposters, several pipelines were built to render transparency as fast as possible.

■ 2.6 Culling

Culling is another helpful method for increasing game performance. Unreal Engine has methods for visibility and occlusion enabled by default. For visibility, the method is called View Frustum Culling, and for occlusion, it is called Dynamic Occlusion. The basic concept of these methods is to reduce the number of objects being drawn on a scene, thus increasing performance and decreasing memory usage. In order to test the objects faster, each object has a simple bounding box and sphere that automatically scales to the size of the object. The bounding boxes are used since it is better to check for a box than a complicated mesh.

■ 2.6.1 View Frustum Culling

In order to visualize a scene, a virtual camera is used that, through various functions, determines what is visible on the screen. The view frustum is the volume created by a near and far plane that contains any potentially visible objects. This volume usually takes a pyramid shape-like appearance. The near plane is the closest point to the camera where objects may appear, and

the far plane is the furthest. Since the view frustum is what the user will see, anything that is not even partially within the frustum does not need to be rendered. The goal of the view frustum culling method is to identify if the objects are inside the view frustum, even partially. If they are not, they are not rendered, saving a lot of time and memory for the GPU.

■ 2.6.2 Occlusion Culling

View frustum culling optimizes the scene by not rendering objects that are not within the frustum of the camera. However, in some instances, the object is within the camera but is fully occluded behind another object. Nevertheless, this object is unnecessarily rendered, even though the viewer cannot see the object. Occlusion culling [6] methods are used to identify if an object is being occluded by another object. If it is, it will not be rendered. To do this, methods check for the visibility of objects within the frustum and if another object occludes them. This is done by issuing a query to the GPU / CPU to check for the visibility state of each object. This task is very memory and time-consuming. Therefore occlusion culling is usually used after all the other culling methods have culled the objects, thus reducing the number of objects that need to be checked.

■ Hardware Occlusion Queries

The main dynamic occlusion culling used by Unreal Engine is Hardware Occlusion Queries and is enabled by default. This method issues a query to the GPU for each frame for each object. The result of the object's visibility is returned one frame later. Due to this latency, this might sometimes cause visual pop-ups if the camera moves too fast. The cost of occlusion scales with the number of objects to process. This cost can be optimized by other, faster culling methods like distance culling so that there will be fewer objects to process. Like frustum culling, if even a tiny portion of the object is visible, it will not be occluded.

■ 2.6.3 Instancing

Sometimes several copies of a single mesh need to be used when creating a scene. The foliage could be considered a prime example. Rendering several same meshes at once tends to be taxing since this will result in several drawcalls. Drawcalls contain all the information such as textures, shaders, and buffers that a CPU needs to process to send to the GPU. Therefore, several properties need to be prepared for rendering a single mesh, resulting in at least one drawcall per mesh. A different way to look at drawcalls is as a group of polygons sharing the same property.

For this reason, Unreal Engine can use instancing, where there is only one template for all copies of a mesh. This can prove to be very cheap since all that is needed is to store various transformations of the copies into an array and send this to the GPU with the cost of only one drawcall. This, however,

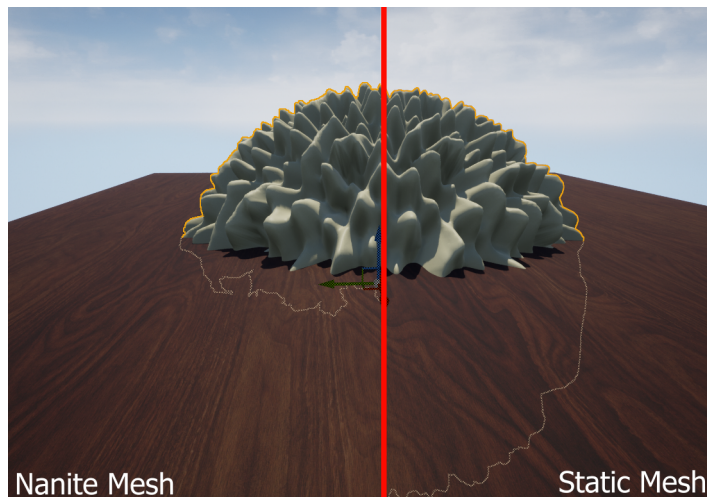


Figure 2.5: Difference between occlusion culling in Nanite (right) and Static Mesh (left).

also does not work well with LODs. In order to use LODs for instanced meshes, Hierarchical Instanced meshes will have to be used [21].

Even with LODs available, instanced meshes have the same functionality as a single grouped mesh. This means occlusion culling does not affect Instanced meshes. Therefore even if a single object is visible, all the instances will be rendered. It is advised to use instances for objects in close vicinity. Instancing meshes work both for Static meshes and for Nanite meshes resulting in a significant boost in performance seen in the test, which can later be seen in Section 5.11.

I encountered an interesting observation when testing out hierarchical instancing. When I tried to render several triangles at once, the instance would disappear completely whenever a certain bound of frametime had been reached. Due to minimal to no information on Hierarchical instancing, I can only assume that when a scene in Unreal Engine takes too long to render thanks to instancing, it instead decides not to render the instance altogether.

Chapter 3

Nanite

3.1 Rendering Pipelines

Nanite meshes use triangles for rendering. Therefore one of the main goals is to create a pipeline that renders triangles as fast as possible. Rendering pipelines are the backbone of rendering 3D models on 2D computer screens. CPU/GPU rendering pipelines used to be the norm, where the CPU handles most of the data that it then sends to the GPU to render on the screen. In contrast to the CPU, a GPU has more computational power meaning it is able to process several tons of data faster than a CPU. However, the problem was that GPU used to have less memory available than the CPU. This could prove to be a problem since highly detailed models will take up much memory. Another reason is that most of the rendering commands need data that comes from the CPU.

Due to technological advancements, however, that is no longer the case, and with DrawIndirect, GPU rendering commands are now supported for the GPU. Therefore a workaround was created to have GPU draw without getting a draw command from the CPU. This meant utterly bypassing the CPU-GPU roundtrip approach, eliminating the bottleneck, where either CPU/GPU had to wait for each other to finish their process. This also means that the CPU will be available to do other work. This is also known as the GPU-Driven Rendering pipeline [17].

The high-level idea of GPU-driven pipelines is to use a DrawIndirect, which takes its parameters from the GPU buffer. Then based on the position in the GPU buffer, the GPU will execute the draw in that buffer. In order to make this happen, the main idea is to have the whole scene with all of its objects stored in the GPU. This goes well and does well when the amount of binds is minimized, which is also known as "Bindless" Rendering [8].

Since Unreal Engine 4.22 was released, the mesh rendering pipeline has been redesigned from immediate mode to retained-mode. [9] Briefly, this means that except of drawing everything visible in the scene every frame, all scene draws are prepared beforehand and stored in the video memory (GPU), where they are only slightly updated when things change. Unreal Engine 5 extended on this. Nanite meshes are also neatly stored in a large resource. If we were to draw depth only, it would only take one DrawIndirect. This will

be very useful in the future for culling.

3.2 Forward Rendering

The two core techniques for shading a scene are deferred lighting and forward lighting [23]. Lighting calculations tend to be very costly, especially for forward rendering. The idea of forward rendering is to render an object in one pass, meaning taking in the data, creating triangles, transforming and splitting it into fragments that are then used to calculate lighting for each light present in the scene. This method scales by the number of (objects) fragments * number of lights in a scene. Also, fully calculating the lighting per object means that if part of an object is later occluded, the light calculations would be wasted.

3.3 Deferred Rendering

Deferred Rendering [23] takes a different path to forward lighting by decoupling geometry from lighting. This is also the rendering technique set by default in Unreal Engine 5 that Nanite uses. As the name suggests, the light rendering is delayed after all the geometry has been prepared. It first goes through the geometry pass, where the scene is rendered once, with all the important information for light calculations stored in a collection of buffers (images) called a GBuffer. Visualization of a GBuffer can be seen in Figure 3.1. As can be seen in the Scene color (top left), it is entirely black, as the lighting has not been calculated yet, only the geometries have been rendered.

With the GBuffers now ready, the delayed calculation of lighting can begin. It takes the GBuffer, which it iterates over pixel by pixel and with the information provided, calculates the lighting for the scene. As such, complexity only scales with screen resolution and the number of lights.

This, however, comes with certain disadvantages. GBuffers are relatively large. Therefore, they take up much memory and bandwidth. Another disadvantage is that it cannot render transparent objects. In order to have transparent objects, a combination of Forward and Deferred rendering can be used for just those objects. Anti-aliasing is also not supported for deferred rendering.

3.4 Nanite culling

As was already stated, culling is one of the main points to consider when optimizing a game. As opposed to the old system, where the model wasn't rendered only when it was fully occluded, Nanite's culling system is more practical, as it doesn't render parts of the model that are occluded and renders the ones that can be seen as can be seen in Figure 2.5. This is done by grouping triangles into "clusters" and giving each cluster a bounding box

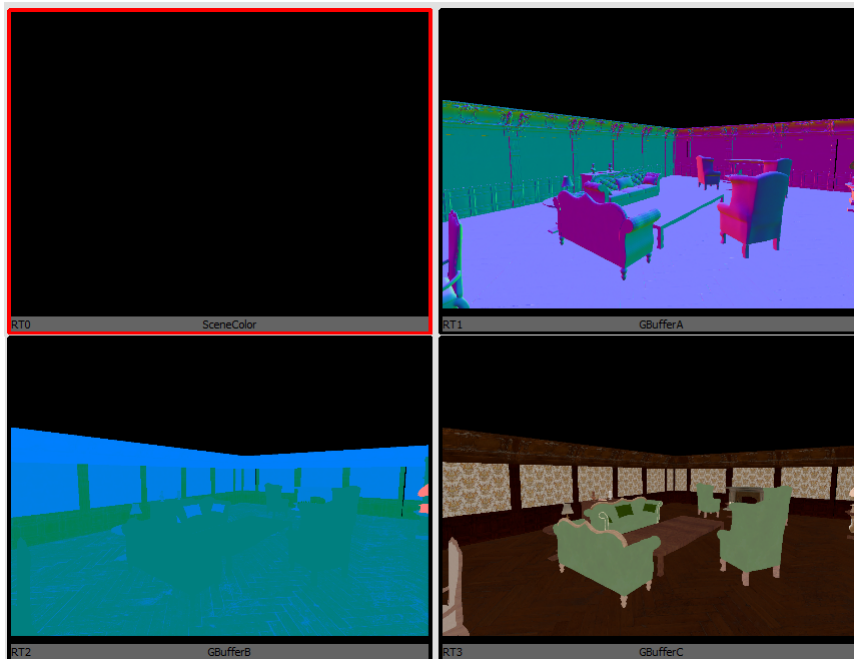


Figure 3.1: Visualization of the GBuffer in Indoor scene with Scene visualisation (top left) using RenderDoc.

for cull testing. It is important to note that each cluster is made out of 128 triangles. A more detailed look into the cluster will be in 3.6.1.

As opposed to the older pipeline, where the occlusion culling was done by Hardware occlusion culling, Nanite culls against a Hierarchical Z-Buffer or HZB for short [14]. An HZB is a mipmapped, down-sampled version of a depth buffer used to check the bounds of objects that also use fewer texture fetches, making them faster. However, due to its more conservative nature and loss of information, it sometimes does not occlude objects that should be occluded, as can be seen in Figure 3.2. However, because Nanite does not occlude whole objects but just the clusters that have their own bounding bounds, this technique proves to be more viable. We will explore the Nanite culling section a little more in Section 3.9.

In order to create an HZB, something has to be rendered first. The core idea is that what was visible last frame will most probably be visible again in this frame. With this idea, a "Two-pass occlusion culling" was created. First, draw what was visible in the last frame, build the HZB from this, then draw what is visible now but was not in the last frame. This creates an almost perfect occlusion culling that only fails during high visibility changes.

3.5 Visibility Buffer

With the base setup, it can be seen that drawing depth can be done very fast. Nanite wants to decouple visibility from materials when adding materials into the mix. This is done by using a visibility buffer[2] [18]. A visibility

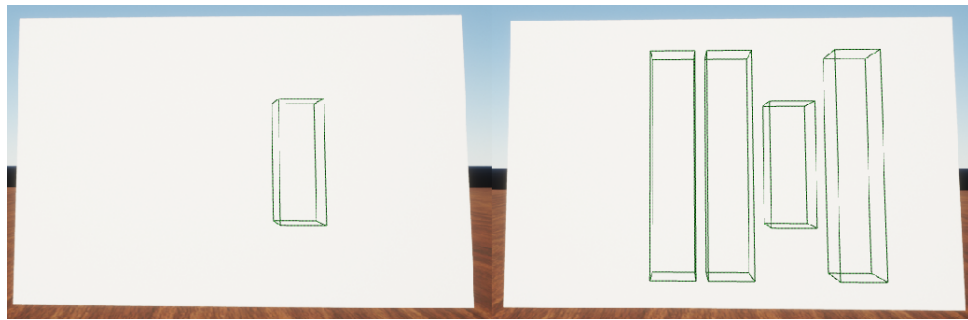


Figure 3.2: Difference in occlusion between HZB (left) and Hardware Occlusion Queries(right).

buffer in the past was meant to substitute the GBuffer, which has a very large bandwidth and takes much memory, by reducing these disadvantages. It only saves TriangleID and InstanceID, where parameters can be easily fetched from here.

Nanite uses visibility buffer differently, however. It uses it to create a GBuffer faster and more efficiently. The visibility buffer stores the depth, triangles and instance ID's. With all of this information saved, the material shaders load the visibility buffer and calculate the data for GBuffer per pixel. However, it sounds like much work has to be done, but due to several cache hits, it is not as slow as it seems. Minor speculations on my side, but one of the reasons for several cache hits is probably due to the fact that several pixels will have the same material shader in their vicinity. Therefore a new material shader does not have to be loaded for each pixel.

This has several benefits. With faster GBuffer creation, it also does not need to switch shaders during rasterization and eliminates pixel quad inefficiencies with dense meshes. Pixel quad inefficiencies happen when a mesh has tiny triangles. Pixel shaders work on a 2x2 group of pixels that are called quads. The problem happens when pixel-sized triangles are being rendered. Except for rendering only one pixel for one triangle, additional 3 pixels are being rendered. Article [18] has an in-depth explanation and shows that visibility buffers generating GBuffers can operate faster with smaller triangles as opposed to forward and deferred rendering.

3.6 Nanite LOD creation

Being able to draw depth within one draw call speeds things up a lot. However, this still scales linearly with instances and with triangles. Instance scaling for Unreal Engine does not prove to be a problem as it can process several multitudes of instances without any problem. Linear scaling in triangles, however, proves to be a problem as processing a multitude of triangles will slow down the GPU.

The idea of Nanite is to render as many triangles as there are pixels. For clusters, this means there should be a constant amount of clusters for every

frame, regardless of scene complexity. To do this, LODs are needed.

■ 3.6.1 Cluster Hierarchy

Since Nanite's written presentation [1] provides quite a certain amount of information, I will only briefly introduce how Nanite does LOD and later on delve into their code to have a deeper understanding of the idea. I would also recommend these to papers, as Nanite used the approach stated there as a building block with some minor tweaks [24]. Unreal Engine is open-source. Therefore I will be commenting on snippets of code in this section. The core codes of Nanite can be found by navigating through: Engine > Source > NaniteBuilder > Private. In order to access these codes, a GitHub account is needed with a registered Epic Games Account. The GitHub repository can be found here [13].

■ 3.7 Cluster Creation

Nanite approaches LOD in a view-dependent way. The same clusters that are made out of 128 triangles that were used for culling are used to calculate LODs for each cluster of an object. Nanite clusters consist of two parts, a preprocess, where the clusters are created and LODs calculated, and a runtime, where loading LODs and culling are done. Clusters can be used for LODs, by creating a hierarchy tree of them, where the leaves are the most detailed segments of the original mesh and their parents are the simplified versions. At runtime, we can find a cut of the tree that has the LOD needed. This is calculated by the screen space error projection of the cluster. Another good perk is streaming, meaning the whole tree does not need to be in the memory at all times. When the children nodes are needed, they can be requested and streamed. When we use parents and have not used their child nodes for a while, they can be removed from the memory.

It is important to note that if a series of simplifications were executed on every cluster of a mesh regardless of their neighbors, then it could cause several topological problems as the edges that connect two or more clusters would not match. In order to bypass this problem, locked boundaries of edges are created by grouping clusters together and performing simplifications within this boundary. These boundaries are then alternated with each LOD simplification.

The LOD creation process for clusters can be seen in Figure 3.3 and can be described as such:

- 1. Create clusters** Creates clusters out of 128 triangles
- 2. Group clusters** Selects a group of N clusters to create. Clusters with most boundary edges are grouped together.
- 3. Merge clusters** Merge the triangles of clusters to a shared list
- 4. Simplify** Simplify the new cluster by 50% of the number of triangles

5. Split Split the simplified triangle list to $N/2$ clusters of 128 triangles.

This process continues by having the newly made clusters be able to form groups again, until only one single cluster remains as the root.

The split and merge operations create DAG instead of a normal tree. This has a suitable property, that locked boundaries cannot stay locked. It is also important to note that in Figure 3.3 the 2. Select cluster and 5. Split has the same boundaries. This will be important later on, as it allows to choose between different LODs per cluster.

■ Nanite simplification

Nanite uses most of the simplification methods that have been briefly introduced in Section 2.3. For simplifying, Nanite uses edge-collapse, where the error metric is calculated using the Quadric Error Metrics, QEM. The vertex position is optimized for minimal error, which returns an estimate of an error of a simplification. This estimated error is then used for screen space projected error to calculate the number of pixel error that determines which LOD is going to be used. The pixel error is not entirely accurate, also due to the fact that by this point, only the triangles have been rasterized. There is no information about materials, texture or colors and others. According to Unreal's statement, their code concerning simplification has been optimized for quality and speed to beat any commercially available option.

This would briefly encompass the creation of clusters and LODs during preprocess.

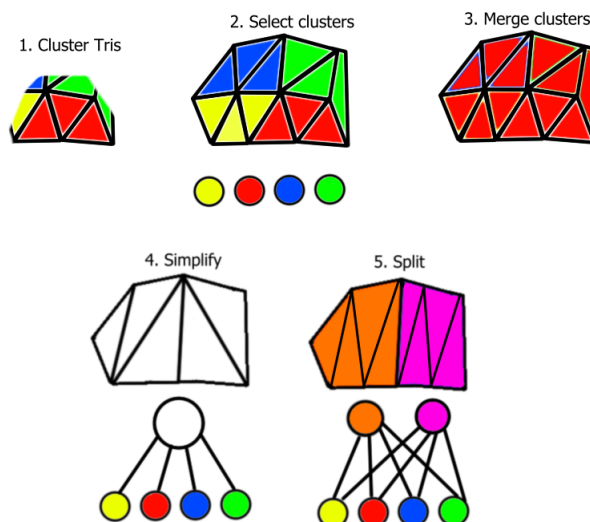


Figure 3.3: Visualization of a simple build operation for a cluster.

3.8 Runtime LOD selection

Runtime comes after all the clusters have been built (in the editor). Later, when the program is running, we want it to be able to select for each frame which cluster will be rendered based on the view. When choosing a cluster to be rendered, two clusters with the same boundaries but different LODs are chosen (refer to Figure 3.3). Then, based on the estimated screen projected error, a LOD is selected at runtime. However, it also depends on which LOD is selected since there may be more LODs with lower errors than the threshold. Another thing to keep in mind is that for larger-scale levels, many clusters will be too detailed to even be considered since it would just waste time. Therefore, some sort of cluster culling for LODs has to be created.

Listing 3.1: Cluster groups structure implemented in Unreal engine 5

```

struct FClusterGroup
{
    FSphere3f          Bounds;
    FSphere3f          LODBounds;
    float              MinLODError;
    float              MaxParentLODError;
    int32              MipLevel;
    uint32             MeshIndex;
    bool               bTrimmed;

    uint32             PageIndexStart;
    uint32             PageIndexNum;
    TArray< uint32 >  Children;

```

This code snippet is a structure of a ClusterGroup. The MinLODError and MaxParentLODError play an essential part in LOD selections. When a group of clusters is formed, they store the minimum error of a cluster in the group and the highest error of its parent. This is to ensure that all the clusters always make the same decision. These errors are calculated at a point that will maximize the projected error inside a bounding sphere around the cluster. That is why Clusters and ClusterGroups have FSphere3F Bounds / LODBounds. The process of LOD selection is then simple. An error threshold of 1 pixel is set to minimize visible pop-ups when the child/parent changes. This is why accurate error estimates have to be made during LOD cluster creation. Then we want to only render cluster groups with an error smaller or equal to the threshold. Figure 3.4 shows how one view chooses the parent and another chooses the children after a slightly closer look. The boundaries stay the same. However, the number of clusters decreases. It also accurately depicts the essence of View-dependent LOD, since all the surrounding clusters stay the same. Therefore in order to choose between the two boundaries, a simple check is run as so:

Render if : ParentError > error threshold && ClusterError <= error

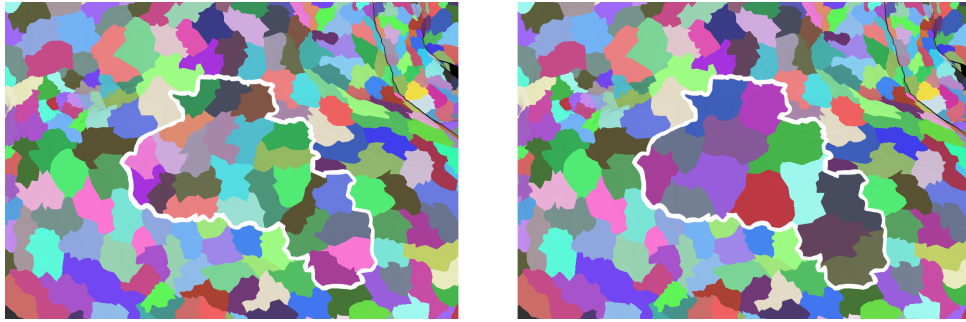


Figure 3.4: Difference between choosing the children group of clusters(left) and choosing the parent group of clusters (right). The boundaries drawn with white colors do not change, however the number of clusters, therefore triangles do.

threshold

Cull if : ParentError \leq error threshold

With all this, the LOD selection should always select the most accurate representation of parts of an object. Moreover, since it does not depend on the full path from the root to the node, it can be processed in parallel. However, there is still too much work. Since Nanite is made to process several millions of triangles, millions of clusters will be made, resulting in a very large DAG. Several clusters will be too detailed to be selected if the scene is very large, so there is no point in checking them. The same rules apply for the culling and the LOD selection, as can be seen in this condition 3.8. In order to ensure parallel and faster traversal, a Bound Volume Hierarchy or a BVH [15] is constructed, where nodes are ParentErrors. This can also ensure hierarchical culling, which means popping a node from the queue, checking the condition, and pushing back its children if they pass the condition, until the queue is empty. Therefore large amounts of unwanted clusters can be culled since they will not even enter the queue. The BVH is traversed using Persistent Threads [16]

3.9 Two-Pass Occlusion Culling

Since LOD culling is being done, it would also be good to do visibility culling. This can be done at the same time the LOD culling is done. However, some problems might rise up such as: LOD Selection from frame to frame will most likely be different, therefore some clusters might be occluded unlike last frame and some clusters don't even have to be present in the memory anymore. Moreover, we don't even have the current HZB buffer yet.

The Two-Pass occlusion culling method should solve these problems. The core idea is to test the current clusters, if they would be visible in the last frame, by testing their bounds against the last frame's HZB with the previous transforms. In order to correctly render and occlude everything, the render goes as so:

1. Test if the clusters would be visible from last frame's HZB.
2. Draw what would be visible and save the occluded for later.
3. Build the HZB for this from from the depth buffer.
4. Using the current HZB, test what was occluded from the last frame again.
5. Draw what is visible (from the current HZB), but was occluded.
6. Build a complete HZB from this and use it for the next frame.

Since we have to redraw the scene two times, the Nanite pipeline has two passes, where the second pass just rerenders the previously occluded clusters or instances, that are supposed to be visible.

■ 3.10 Rasterization

Nanite wants to have pixel-scaled detail and in order to achieve that, pixel-sized triangles are needed. However, hardware rasterizers are not optimized to work with microtriangles since they work on a 2x2 pixel block. Therefore it would still compute 4 pixels even if only one pixel is supposed to be computed. A very nice statistic can be seen here [11]. Therefore, Nanite has its own GPU software rasterizer, specifically built to render tiny triangles around three times faster than a hardware rasterizer. So every cluster is chosen to either be rasterized by the software or hardware based on the length of the triangle. If the triangles in a cluster are less than 32 pixels long, they are software rasterized.

■ 3.11 RenderDoc

In order to get a broader view of how Nanite's pipeline worked, I opted to use RenderDoc. RenderDoc is a free graphics debugger that allows developers a single-frame capture, and detailed inspection of applications created with Vulkan, DirectX, OpenGL and more. Using RenderDoc, we can see what Unreal does in order to render one frame. Unreal Engine also supports RenderDoc in the form of an in-editor plugin. The program layout can be seen in 3.5. As can be seen, it captures each state of the program with the timestamp of how long each action took. There are also several visualization tabs on the right side. When writing about what RenderDoc does, it will be applied knowledge without any backing facts.

I decided to capture a frame from the Indoor scene, as the objects are more clustered together and it has more unique objects than other scenes. The frame capture can be seen in Figure 3.6.

3. Nanite

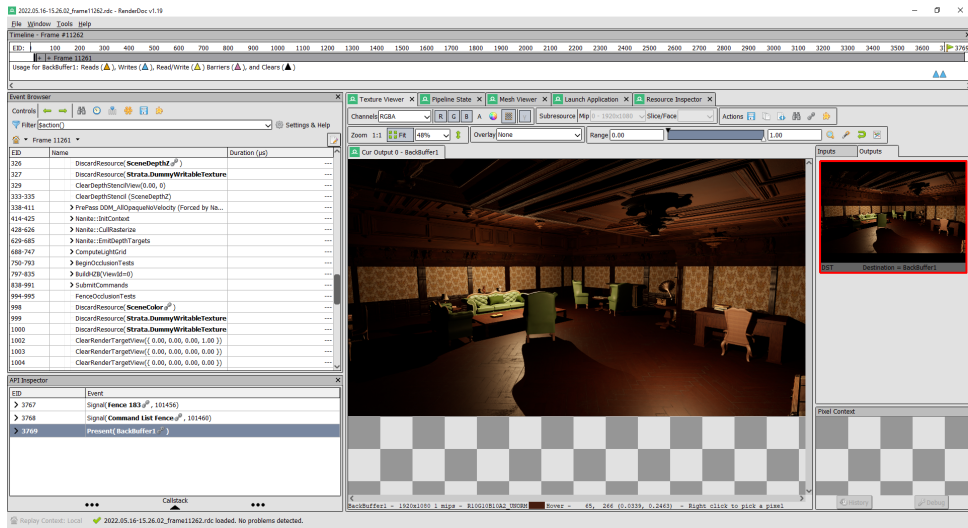


Figure 3.5: Screenshot of the RenderDoc application.

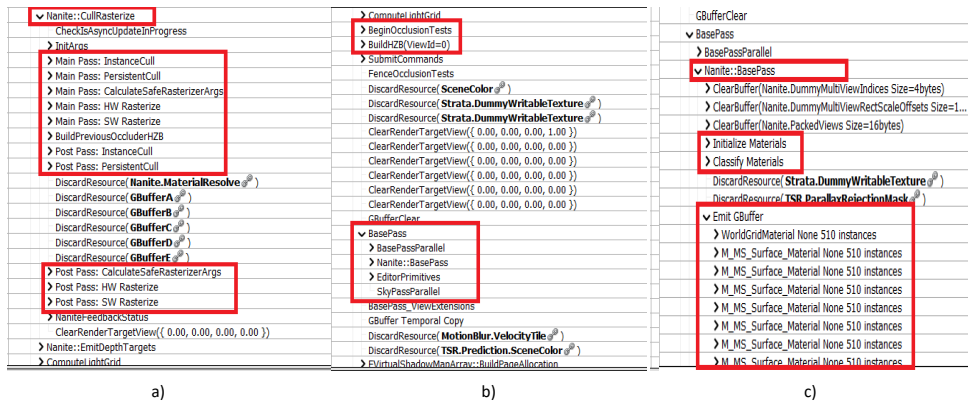


Figure 3.6: One frame capture of the indoor scene in RenderDoc.

Nanite Cull and Rasterize

The first important part is the `Nanite::CullRasterize`, where as the name suggests, Nanite meshes first cull each other and then rasterizes what needs to be rasterized. These are the parts that were mentioned in Section 3.6.1. As can be seen in Figure 3.6a), the `Nanite::CullRasterize` has two passes. The main pass and the post pass. This is the two pass occlusion culling, coupled with rendering that we talked about in Section 3.9. Having the HZB built from objects that were visible the last frame, first perform the `InstanceCull`, where instances that fully occluded are saved for later testing, then perform the `PersistentCull`, where it culls clusters of an instance that are not visible and saves them for later.

After the work has been HW Rasterize and SW Rasterize are used for drawing differently sized triangles. Very smaller triangles are drawn using the SW(software) Rasterize and larger triangles are drawn using the

HW(hardware) Rasterize. This can further be confirmed by opening the drop down window. HW Rasterize calls a function `IndirectDraw(<384>, <1769>)` which is the traditional way of using the hardware rasterization. The issued command wants the hardware to draw 1769 instances of 384 triangles per instance. The SW Rasterize calls a function `IndirectDispatch(<2825,1,1>)`. This function calls 2825 compute shaders to rasterize tiny triangles.

With this, the first pass is done. Rebuild the HZB, which can be seen as `BuildPreviousOccluderHZB` and start testing the occluders. The Post Pass is the second step that does the same, except with the newly made HZB (Figure 3.6b) `BuildHZB`).

■ BasePass

Up until now, the whole process was focused on only rendering the geometry without materials. BasePass (Figure 3.6c)) is where the materials are processed and rendered on screen by applying the materials and transforming the Visibility buffer to a GBuffer. The BasePass is separated into two parts: Classify Materials and EmitGbuffer. Classify Materials takes in a Visibility Buffer as input however does not seem to have a visible output through RenderDoc. Since EmitGBuffer looks like a series of drawcalls for each Material ID, my guess that Classify Material classifies materials present in the frame in a texture and EmitGbuffer then goes through the texture for each drawcall and renders the material, if present.

Now with this general knowledge we can visualize the Nanite RenderPipeline as can be seen in Figure 3.7.

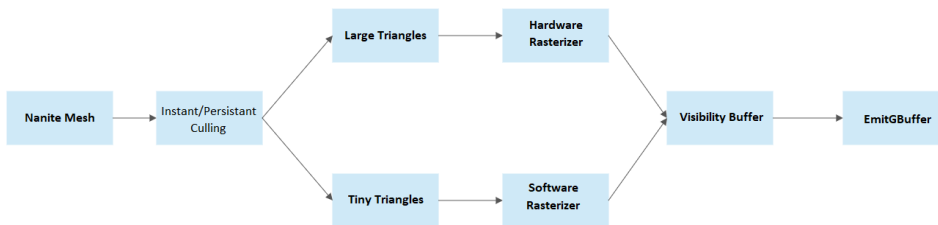


Figure 3.7: Simple visualization of the Nanite render pipeline. This pipeline has to pass two times.

Chapter 4

Nanite in practice

4.1 Nanite support

4.1.1 Unsupported Nanite settings

Nanite can be used for Static meshes and joined static meshes also called Geometry collections. However, it is very limited when it comes to rigid meshes. Any form of mesh deformation is not supported by Nanite technology. This for example includes:

1. Skeletal Animation
2. Spline meshes

Additionally, certain settings for materials are also not supported. Using such settings will usually result in either an error or will not be an option to begin with. These settings include for example:

1. Any Blend mode setting, excluding Opaque
2. Wireframe
3. World position offset
4. Two sided material

The examples presented are only a selected few, that are more usually used, when creating a scene. For all examples, I recommend reading the official Unreal Engine documentation[9].

4.1.2 Aggregate geometry

Furthermore, it is important to note that even though Nanite works on any kind of Static mesh, it is advised to not use it everywhere. The main example would be aggregate geometry. Aggregate geometry stands for several smaller, disjoint objects, that together create a larger volume for example hair, grass, leaves and other. Nanite's main principle is to try and draw no more triangles than there are pixels and it does so by using two primary

techniques: fine-grained level of detail and occlusion culling. However with meshes that have aggregate geometry, it is harder for Nanite to determine how occlusion culling or LOD should be used, usually resulting in a lot more triangles being drawn, than is needed.

The most obvious case of this would be foliage. Not only does it have several disjointed meshes, that make up a volume, but also usually scenes would want to apply wind effects or some other effects, that could affect the position of the grass. This all the more supports the fact, that Nanite should not be used for foliage. However, this does not mean Nanite is completely inapplicable for foliage. For example using Nanite for leaves of a tree would certainly not work, using Nanite for the tree trunk might be useful. However, wind support for Nanite is planned in the future.

■ 4.1.3 Concerning foliage

When it comes to foliage, there are three general methods of how to render foliage into a scene. One uses a technique called billboarding and the other uses a mesh. The third method is a form of hybrid, using both techniques for different scenarios. Usually, when using foliage such as leaves, bushes etc. the most used approach would be with billboards, as it has very little amount of vertices and realistic results. However, when rendering grass, this approach might be debatable. An interesting article surfaced, where several tests were done using billboards and using geometric meshes for rendering grass. Visually the result were similar, but surprisingly for the speed tests, geometric meshes were faster and took less memory.

Since we primarily want to test Nanite, tests using grass billboards will be completely omitted as Nanite will have no effects at all. It is also important to note, that Unreal Engines foliage tools uses Hierarchical Instancing for grass by default.

■ 4.2 Nanite inside Unreal Engine 5

Unreal Engine is a complex engine with several features spanning from animation creation to blueprint programming. One of the features we will be mostly using will be the Static Mesh editor User Interface and the scene viewport to test out Nanite capabilities. The Static Mesh Editor can be divided to four parts as can be seen in Figure 5.1 consisting of:

1. Menu Bar
2. Toolbar
3. Viewport Panel
4. Details Panel

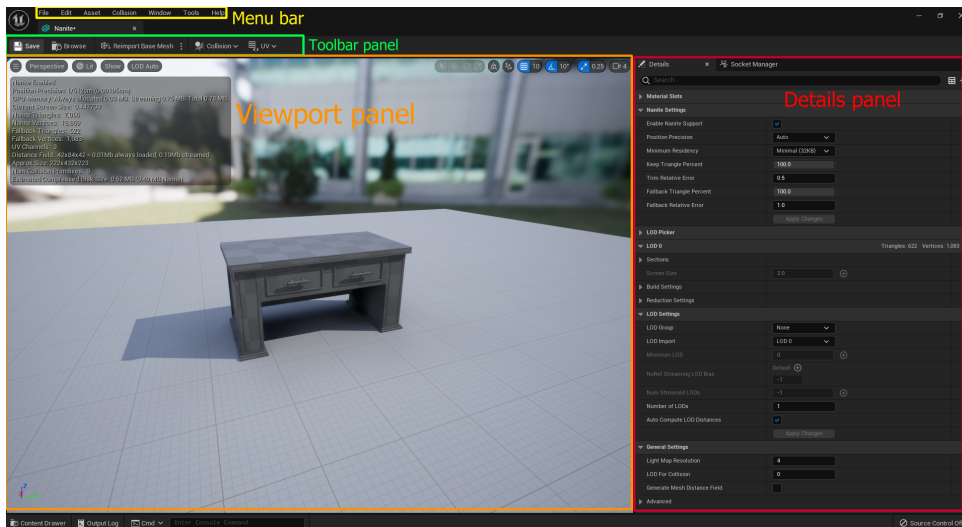


Figure 4.1: Visualization of the static mesh editor with a hand made model without optimization.

4.3 Viewport

The most important part for testing LOD and Nanite meshes will be the Viewport panel and Details panel. The Viewport gives us a visual representation of a rendered object as it would be rendered in-game. It also has a viewmode, that lets us see the representation in different views such as wireframes, unlit, with or without lightning and more. The view mode also gives us access to "Nanite Overview". Since Nanite meshes are different from Static meshes, this overview needs to be turned on in order to see more details for Nanite meshes. Viewport also has simple statistics for each individual models, where our main focus will be directed to the number of triangles and vertices.

4.4 Details Panel

The Details panel shows specific properties and settings for the static mesh such as materials, LOD settings and more importantly Nanite settings. This is where the Nanite meshes are generated. The generation of a Nanite mesh in Unreal Engine is really simple. Import a Static Mesh, open the static mesh in the Static Mesh Editor, navigate to the Details Panel and in the Nanite settings, check the "Enable Nanite".

4.5 Nanite mesh settings

When enabling Nanite, several settings are available. One pair of settings concern the Nanite mesh itself and another consists of its proxy mesh, also called a Fallback mesh. These settings allow users to control the representation

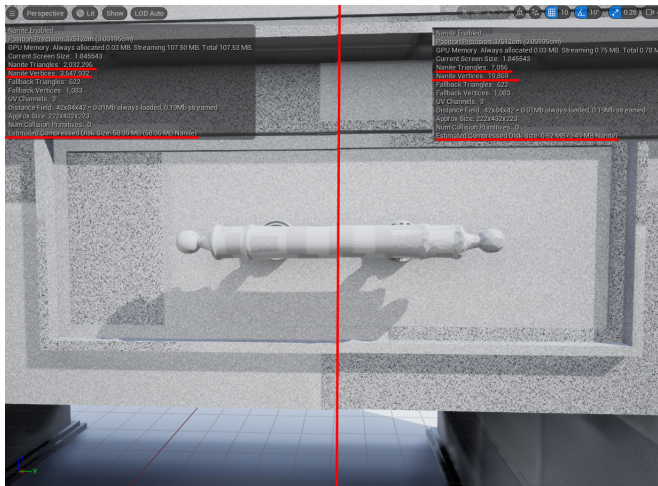


Figure 4.2: Table with trim error applied.

fallback mesh is when a certain hardware doesn't support Nanite. When this happens, except of rendering the Nanite mesh, it renders the fallback mesh.

Fallback meshes have the same settings as the Nanite meshes that are **Fallback Triangle percent** and **Fallback Relative Error**. By default Fallback triangle percent is 100 (no decimation) and Fallback relative error is 1.00 (minor decimation).

After enabling Nanite and applying the settings, Unreal Engine will automatically do all the calculations and create the mesh all by itself in a matter of a few seconds. The Nanite mesh is now ready to use. By disabling the Nanite mesh, it will revert to its original static mesh form.

Unreal Engine offers a Nanite Visualization mode, that lets us visualize different aspects of a Nanite mesh. When enabling Nanite Visualization there are several different options that can be used to visualize Nanite meshes(4.3), where the most important one would be:

1. Mask - visualization that colors Nanite meshes green and Non-Nanite meshes red
2. Triangle - displays all triangles of the Nanite meshes in the current scene
3. Clusters - displays colored representations of grouped triangles that form a cluster
4. Overdraw - displays the amount of overdraw for a scene geometry. Closely stacked objects cause overdraw that can be seen as heat signatures.

4.5.1 Overdraws

When creating a scene with Nanite objects, it is important to not cause too much overdraws for Nanite to work correctly. Overdraws happen, when

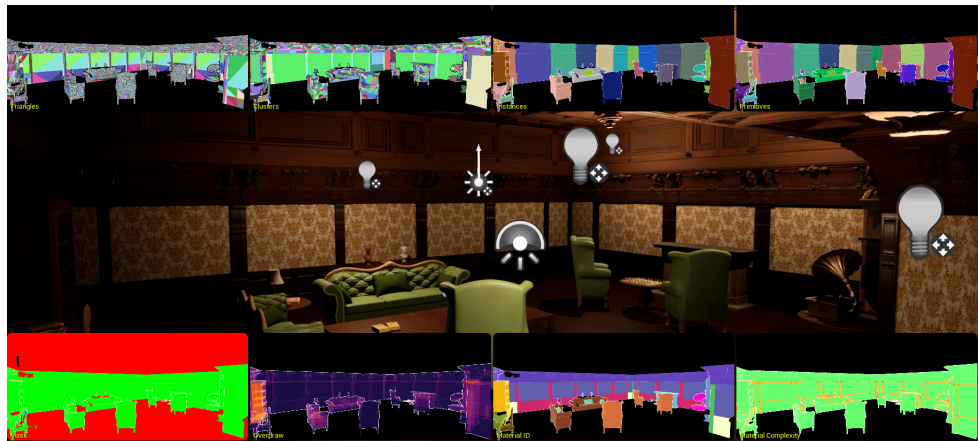


Figure 4.3: A scene with Nanite Visualization turned on.

several Nanite meshes are closely stacked together and overlap each other. Due to this Nanite has problems with occlusion cullings and will render objects even though they are hidden. In the Nanite visualization, they can be perceived as heat signatures, where the higher temperature is equaled to higher overdraws, which negatively affects performance.

4.6 Standard optimization workflow in Unreal Engine

Having now introduced the essential optimization techniques when rendering meshes. A standard workflow for optimizing meshes will be introduced to show the difference between Nanite and Static mesh optimization.

Before a mesh can be imported into the Unreal Engine, it needs to be first created. The general process of model creation in a 3D modeling software is first to create a low-poly mesh and then create a detailed high poly mesh. Additional textures would be baked from the high-poly model, such as normal maps, which would then be applied to the original low-poly mesh to cheaply fake various intricate details of a mesh. This results in fewer vertices and triangles needed to create a detailed model.

Afterward, the mesh would then go through several phases of mesh simplification to create several levels of detail, which will then be applied in a game engine. This can be either done in the desired 3D modeling software or directly in Unreal Engine after the mesh has been imported. Unreal Engine not only allows the users to control how simplified each LOD can be, but they can also choose when to render each LOD based on the distance. It is, however, advised not to set it manually since, primarily, the algorithm will pick the best possible solution.

Since Occlusion culling is enabled by default, having several objects might be demanding on the hardware. Therefore the best practice would be to instance several copies of the same mesh (if there are any), thus decreasing

Hardware	GPU	CPU	Memory	Resolution
Machine 1	NVIDIA GeForce RTX 2070	Intel Core i7-9750H	2x 8GB DDR4 SDRAM	1920x1080
Machine 2	NVIDIA GeForce GTX 970	Intel(R) Core(TM) i7-4790K	2x 8GB DDR4	1920x1080

Table 4.1: Table of hardwares used to produce data in the test scenes.

draw calls and workload for occlusion testing. Due to frustum culling being enabled by default, only cull distance might need to be set up additionally in order to cull objects far away from the viewer.

If even after these standard optimizations, the scene is still slow, it is advised to do a CPU / GPU profiling to find out where the problems might be.

This is laborious work that sometimes does not work very well compared to Nanite. Nanite completely changes this workflow. Modelers no longer need to create high poly models for normal maps. They can import the high poly model right away without worrying about the number of triangles or drawcalls. After importing a high poly mesh, all that needs to be done is to enable Nanite the mesh, and it will automatically process all the optimization for them. Additionally, according to statistics and tests in Section 5, it does it substantially better in some cases.

4.7 Testing scenes

In order to compare the difference between Nanite and LOD, test scenes were created. Each scene has a Nanite and LOD variant with cameras capturing the same scene. This is crucial since the same conditions have to be applied to both variants in order to gain the most exact data used for comparison as possible. The scenes were tested on two machines with these hardware properties:

Chapter 5

Statistics

In order to compare the scenes with different meshes, several data, such as draw time, number of draw calls, frames per second, triangle count need to be gathered. We can give Unreal Engine a command that will start gathering all the data we need. By using the command "stats startfile", Unreal creates its own data format, where it stores everything that happened during rendering the scene. After we accumulated enough data, we can stop it by issuing another command "stats stopfile". By doing so, an Unreal data format will be stored into the project. This data format can then be viewed inside of Unreal Engine using a "Session Frontend" tool. The data format stores all information about everything needed to render the scene, from commands used to number of triangles drawn.

Each scene has cameras that are binded to a keyboard key that will then start a camera animation and start gathering data. This is done in order to have the same conditions when we start gathering data to have it as precise as can be.

5.1 Data graphs

The data graphs were created using Unreal Engine to gather the information with the StartFPSChart and StopFPSChart commands. This command gathers data concerning RenderThread (CPU), GameThread, and GPU each millisecond. Due to the huge amount of data gathered, I shortened it by writing a small python program to convert the data to seconds, averaging them for each second. I then plotted graphs with data relevant to their time in seconds or relevant to the number of objects in the scene.

5.2 User feedbacks

User feedbacks were done in person, as a lot of people do not own an appropriate hardware to run the tests and in order for me to portray the same experience to every tester. As a result I was able to have 7 testers fully test the scenes and give feedback. The testers background knowledge about computer graphics are diverse, from completely none, to very experienced.

They were tested without knowing which scene is which in order to get the most accurate result. For each scene I will summarize the feedbacks I had gained.

5.3 Drawcall test

In Section 2.6.3 we stated that a lot of drawcalls / materials tend to be worse than a lot of triangles for modern GPU. This test scene was done to confirm what was previously stated and to see how Nanite deals with drawcalls.

As can be seen in Table 5.1, a simple plane was created with only 128 triangles. This plane however had 64 unique materials, that will increase drawcalls and overhead. 225 planes were then added each second for 15 seconds, until 3375 planes were rendered in one scene.

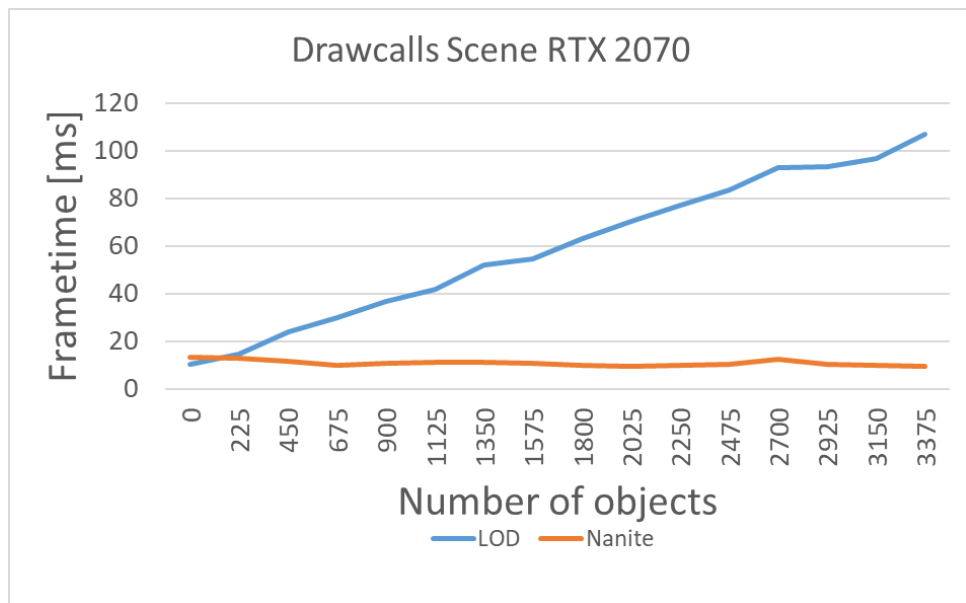


Figure 5.1: Graph depicting the frametime with increasing objects that have high drawcalls.

Mesh Name	LOD Triangles	Nanite Triangles	Instances
Plane	LOD0: 128	128	3375

Table 5.1: Table of meshes present in Drawcall scene.

In Figure 5.1 and Table 5.2 we can see how a lot of drawcalls can negatively affect performance. In Section 5.5 over 3 million more triangles are being rendered, yet the framerate is still stable. The difference is in drawcalls. The CPU spends a lot of time, in order to get 4500 drawcalls ready every frame, so the GPU has to wait for the CPU to finish.

Test Drawcall	AVG. GPU	AVG. CPU	AVG. FPS	AVG. Tris
With Nanite (RTX)	9.05 ms	6.55 ms	85.72 FPS	100k Tris
With LOD (RTX)	110.12 ms	95.56 ms	9.45 FPS	1 mil Tris
LOD Drawcalls	4500			
Nanite Drawcalls	44			

Table 5.2: Table of performance for Drawcall scene. The values were captured after all the meshes had been rendered.

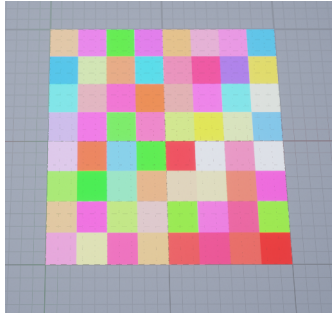


Figure 5.2: Image of the plane with 64 unique meshes. Every square represents one unique material.

Since Nanite decouples visibility and materials, the materials need to only be processed once, which reduces the amount of drawcalls needed.

5.4 Outdoor scene

For this scene, only models from Megascans [25] were used. Megascans make highly detailed models from 3D scanning objects in real life. These models usually have thousands of triangles in order to correctly represent a highly detailed model from a 3D scan. However, the model can be significantly simplified without losing most of its visual fidelity. The model of a cliff that was used here originally had over 4 million triangles. After simplification the final number of triangles for the most detailed LOD0 was around 25,000 triangles. The Nanite version of this model was created from the robust, original model that had over 4 million triangles, more can be seen in Table 5.3.

Interestingly, for a scene made out of Megascans objects, traditional LODs proves to be somewhat better, outperforming a little in all cases. Taking in account, that several Nanite meshes derive from the original source, that has a lot more triangles than the LOD as can be seen in Table 5.3, they both have very stable performance, with Nanite having the opportunity to show more detailed parts. It is also important to note, that the scene itself averages only at around 1.5 million triangles for LODs, which for today's modern GPU, isn't a lot.

However, even with several LODs, visual popping was still a little evident in the case of LODs, to the point, that some of the testers, who were a little more knowledgeable about LODs were able to find them even without

Mesh Name	LOD Triangles	Nanite Triangles	Instances
Boulder	LOD0: 13,104 LOD1: 6552 LOD2: 3276 LOD3: 1638 LOD4: 818	2 million	40
RockFormation	LOD0: 25,168 LOD1: 12,584 LOD2: 6291 LOD3: 3146 LOD4: 1573	2 million	37
RockGround	LOD0: 11,854 LOD1: 5927 LOD2: 2964 LOD3: 1482 LOD4: 740	2 million	45
HugeCliff	LOD0: 24,943 LOD1: 12,471 LOD2: 6236 LOD3: 3117 LOD4: 1559	4 million	7
MassiveCliff	LOD0: 25,163 LOD1: 12,582 LOD2: 6290 LOD3: 3146 LOD4: 1572	4 million	15

Table 5.3: Table of meshes present in Outdoor scenes.

Test Outdoor	AVG. GPU	AVG. CPU	AVG. FPS	AVG. Tris
With Nanite (RTX)	14.67 ms	6.18 ms	67.93 FPS	2.5mil Tris
With LOD (RTX)	13.39 ms	4.81 ms	74.47 FPS	1.5mil Tris
With Nanite (GTX)	29.07ms	9.30 ms	34.30 FPS	2.5mil Tris
With LOD (GTX)	23.91ms	7.55 ms	41.76 FPS	1.5mil Tris
LOD Drawcalls	430			
Nanite Drawcalls	320			

Table 5.4: Table of performance for Outdoor scene.

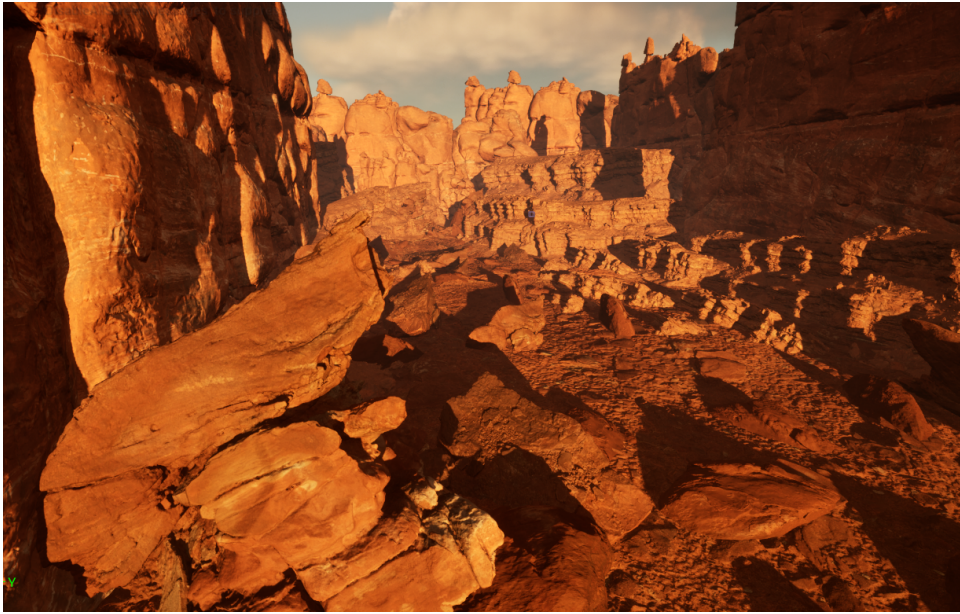


Figure 5.3: Image of the outdoor scene.

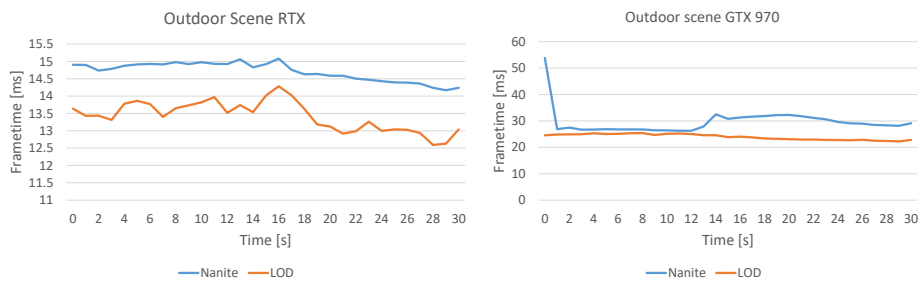


Figure 5.4: Graph depicting the difference in time for rendering a frame in a camera flythrough in the Outdoor scene.

being prompted to. Though for normal viewers, the visual pop ups are imperceptible. In Nanites case, no visual pop-ups were visible at all.

Most of the people who tested this scene individually agreed, that the Nanite version of the scene looked better, than the LOD version. Most of them pointed out, that the shadows and some parts of the rock formations seemed more natural. The more experienced ones even found visual pop-ups in the LOD scene, while the less experienced ones said there wasn't any difference.

5.5 Indoor scene

The second scene is made out of meshes created by amateurs without the applications of LODs. It is meant to be a representation of an indoor of a

Mesh Name	LOD Triangles	Nanite Triangles	Instances
FrontWall	LOD0: 20,309	20,309	18
Chess_Bishop	LOD0: 46,648	46,648	4
Chess_Pawn	LOD0: 8192	8192	16
Chess_King	LOD0: 11,109	11,109	2
Chess_Knight	LOD0: 26,095	26,095	4
Chess_Queen	LOD0: 51,602	51,602	2
Chess_Rook	LOD0: 35,302	35,302	4
Armchair	LOD0: 83,331	83,331	3
Chessboard	LOD0: 958	958	1
Book	LOD0: 1520	1520	79
Bookshelf	LOD0: 13,438	13,438	3
Chair	LOD0: 45,916	45,916	2
Drawer	LOD0: 79,421	79,421	2
Table	LOD0: 744	744	2
Grandfathers Clock	LOD0: 8952	8952	1
Gramophone	LOD0: 90,960	90,960	1
BankLamp	LOD0: 17,633	17,633	1
Newspaper	LOD0: 1880	1880	1
Pipe	LOD0: 39,232	39,232	1
RoundTable	LOD0: 6840	6840	3
Female Statue White	LOD0: 9256	9256	1
Female Statue Orange	LOD0: 10,094	10,094	1
Fireplace	LOD0: 7978	7978	1
Wooden Trim	LOD0: 19,682	19,682	5
Wooden Ceiling1	LOD0: 5376	5376	9
Wooden Ceiling2	LOD0: 15,730	15,730	19
Sofa	LOD0: 57,738	57,738	2
Cushion	LOD0: 7076	7076	4
Chest	LOD0: 19,996	19,996	1
Wooden Lamp	LOD0: 19,996	19,996	1
Living Table	LOD0: 504	504	1
Writing Table	LOD0: 17,210	17,210	1
Telephone	LOD0: 69,672	69,672	1

Table 5.5: Table of meshes present in Indoor scenes.

Test Indoor	AVG. GPU	AVG. CPU	AVG. FPS	AVG. Tris
With Nanite (RTX)	12.49 ms	4.43 ms	79.91 FPS	400k Tris
With LOD (RTX)	12.70 ms	6.02 ms	78.58 FPS	4 mil Tris
With Nanite (GTX)	20.93 ms	6.15 ms	47.75 FPS	400k Tris
With LOD (GTX)	20.33 ms	8.36 ms	49.15 FPS	4 mil Tris
LOD Drawcalls	750			
Nanite Drawcalls	300			

Table 5.6: Table of performance for Indoor scene.

household. Because of this, the triangles will be closely clustered resulting in several triangles being rendered at once. This is to see how Nanite will be able to process several abundant triangles of unoptimized models and how the scene will be compared with having non-Nanite and non-optimized models. It is also a test of Nanite being used in game ready made scene. Another reason is to see, how Nanite would fair with some models that are topologically incorrect.

An overview of all meshes present in this scene can be seen in Table 5.5. The reason why Nanite triangles are same as LOD0 triangles is due to the fact, that the Nanite mesh was created from the LOD0. This is an issue I will later discuss.

In Figure 5.5 Nanite seems to be fairing slightly better than meshes without LODs for the RTX hardware. Looking at Table 5.6 the GPU times seem to be the same, however, the CPU is slightly slower for the LOD. This is most likely because of drawcalls, since this since has several materials and meshes that need to be processed.

The graph for the GTX hardware and Table 5.6 seem to be fairly similar.

All of the testers agreed, that the scene looked completely the same, without any distinguishable difference. One tester however found out, that the bookshelves in the Nanite scene had an overlap problem, where one shelf kept switching textures. Upon closer inspection in a 3D modeling software, the shelf had doubled faces. It seems that since the faces were overlapped, Nanite had trouble deciding which face to actually render. Meanwhile in the LOD, no such artifact could be seen.

5.6 Foliage scene

The goal of this test was to see how Nanite would fair with aggregate geometry, specifically foliage. The best candidate for this would be to try Nanite on grass as it's pretty normal for grass to be made out of geometric mesh unlike other foliage.

The scene was simplistic with the main focus on grass only. Unreal Engine has its own inbuilt foliage system, which makes placing foliage easier. With this system, developers can select different types of foliage and place several meshes around a circled area with adjustable width and density. Upon placing, Unreal Engine automatically creates several instances according to

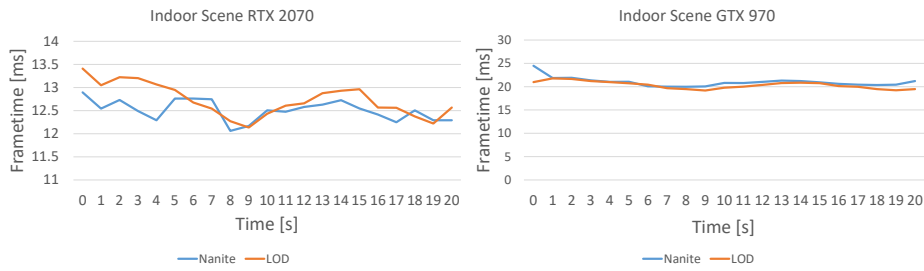


Figure 5.5: Graph depicting the difference in time for rendering a frame in a camera flythrough in the Indoor scene.

Mesh Name	LOD Triangles	Nanite Triangles	Instances
Grass	LOD0: 1152	1152	1 million (culled for LOD)

Table 5.7: Table of meshes present in Foliage scenes.

the settings. [10] This simple tool makes it possible to create more test results. For this purpose, Nanite and Non LOD meshes will be tested with one million instances. The model consists of several grass blades per one instance with 1152 triangles in total.

Since Nanite cannot handle world position offsets, a simple Wind shader will be applied to the grass that will change its world position offset. This is to test what will happen if a world position offset is forcibly applied to a Nanite mesh.

However, even though using a foliage tool for grass is the standard procedure, it has one problem that was stated in Section 2.6.3. Since Foliage tools create Hierarchical instances, it completely disappears after a certain framerate threshold is reached. In order to still be able to continue with the testing, cull distance was added for the LOD grass. This means that after a certain distance has been reached from the camera, the grass beyond this region will not render, unless the viewer gets closer. Therefore LOD test will not be rendering one million instances at once. I decided to keep the one million instances for Nanite in order to show the huge difference between the two.

Another problem with Instanced meshes is that the profiler cannot correctly calculate how many triangles are being rendered and how many drawcalls are being used for Instanced meshes, therefore they will not be included in the tables.



Figure 5.6: Image of the Indoor scene.

Test Foliage	AVG. GPU	AVG. CPU	AVG. FPS	AVG. Tris
With Nanite (RTX)	19.13 ms	9.15 ms	52.25 FPS	25 mil Tris
With LOD (RTX)	45.56 ms	8.39 ms	21.93 FPS	-
With Nanite (GTX)	43.79 ms	10.86 ms	22.85 FPS	25 mil Tris
With LOD (GTX)	67.32 ms	11.95 ms	14.87 FPS	-

Table 5.8: Table of performance for Foliage scene.

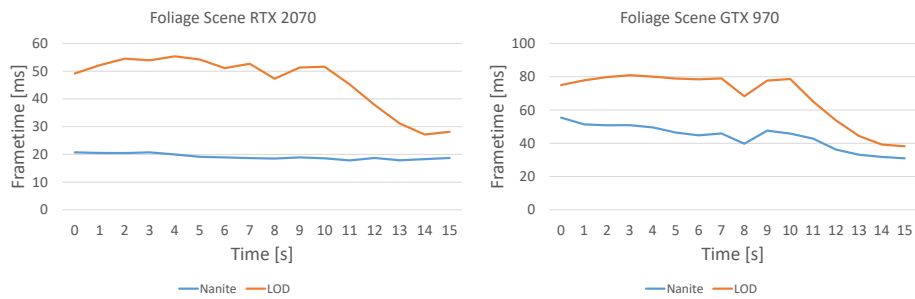


Figure 5.7: Graph depicting the difference in time for rendering a frame in a camera fly-through in the Foliage scene.

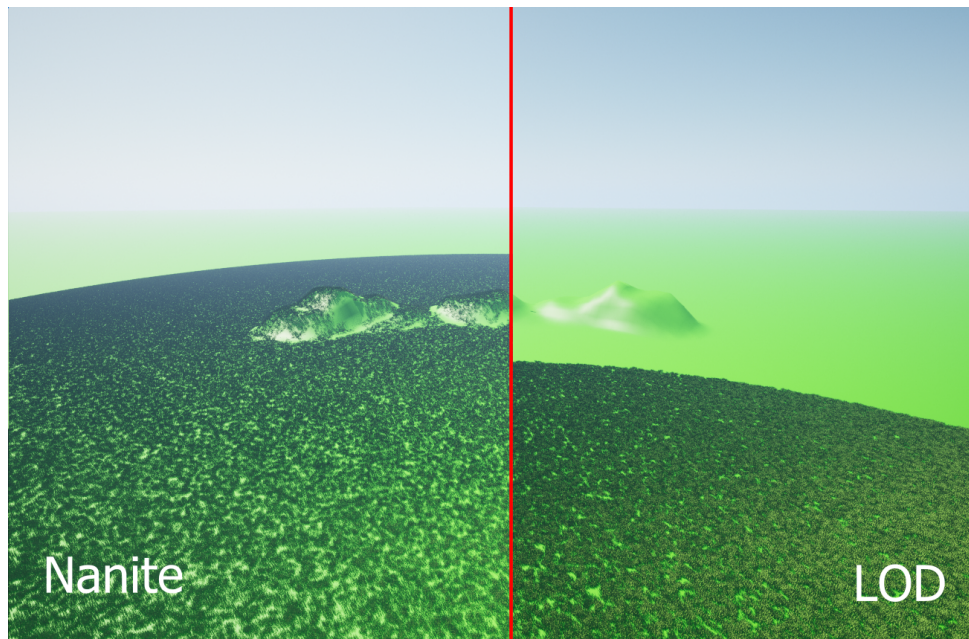


Figure 5.8: Image of the Foliage scene with the difference of Nanite(left) and distance culled LOD(right)

As was already stated, adding World offset (grass wind) doesn't work with Nanite. Since World Offsets are added through materials, when assigned the material, the grass will turn black without any world offset. This alone might be enough to encourage others to not use Nanite for foliage. However, for non-moving foliage, Nanite bears exceptionally better results.

When using over one million instances, Nanite can effortlessly render all the instances at once with pretty high and stable FPS. Static mesh completely falls short on this part. It is unable to render all the instances at once, as it would take too long to process. Even with cull distance, the performance for large amounts of grass is poor. According to the Table 5.8, the bottleneck is the GPU. This is most probably due to the fact, that for instances, any visible instance, all the instances will be rendered. That means a lot of triangles have to be rendered at once. For Nanite this would mean around one billion triangles, but since Nanite only scales with screen resolution, it actually renders only around 25 million triangles.

The reason sudden drop, that can be seen in the graph Figure 5.7 is because towards the end there is no more grass.

All the tester agreed, that the Nanite foliage scene looked better, which was to be expected as grass didn't suddenly appear as was the case of the LOD scene. Some testers have noticed a small problem with Nanite. Due to its problems with aggregate geometry, some patches of grass would disappear if the camera was far enough, even though it should still be seen. This was quite a big problem with the Early Access of Unreal Engine 5, that was supposedly fixed, but for some cases, it still seems to be a problem.

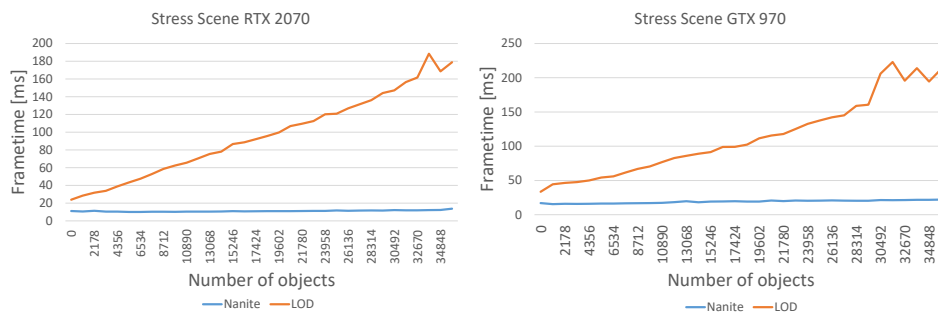


Figure 5.9: Graph depicting the framerate according to the increasing number of objects.

Test StressScene	AVG. End GPU	AVG. End CPU	End FPS	Total Tris
With Nanite (RTX)	11.82 ms	8.31 ms	90.17 FPS	25 mil Tris
With LOD (RTX)	72.53 ms	56.10 ms	5FPS	250 mil Tris
With Nanite (GTX)	20.75 ms	16.02 ms	40.47 FPS	25 mil Tris
With LOD (GTX)	230.42 ms	162.03 ms	4.21FPS	250 mil Tris
LOD Drawcalls	39,500			
Nanite Drawcalls	44			

Table 5.9: Table of performance for Stress scene.

5.7 Scene Playground

The last scene made is a playground for testing the capabilities of Nanite with various settings. There is an option to choose from the editor how many objects will be rendered for a column, row and height.

The mesh used is a simple sphere with several layers of subdivisions and deformations, adding up to 250,000 triangles per mesh. These meshes can be tested only with either Nanite or LOD. Up to 3 LODs were created with Unreal Engine. The main test for this scene is to stress test Nanite and LOD by having several instances with a lot of triangles at once.

Table 5.9 is a little different from the previous tables as the tests were done per added objects. Therefore the table is made out of values that have been captured after all the objects had been rendered, in this case 35,937 instances.

The graphs in Figure 5.10 perfectly depicts the problem with the normal standard rendering pipeline. It accurately depicts that poor performance linearly scales in number of triangles. The more triangles present, the worse performance gets. As can be seen in Table 5.9, Nanite renders only 25 million triangles, without losing visual fidelity, while LOD renders around 250 million triangles in the end. Another reason that can be seen in the table is the number of drawcalls. While Nanite has only 40 drawcalls, the CPU time is really low. For LOD, that has 39,500 drawcalls, the CPU time increases substantially and makes the GPU wait.

In order to test the limits of Nanite, I tired spawning as many objects as possible, to see how the framerate will change.

Mesh Name	LOD Triangles	Nanite Triangles	Instances
StressMesh	LOD0: 250,880	250,880	35,937
	LOD1: 62,720	62,720	
	LOD2: 31,360	31,360	

Table 5.10: Table of meshes present in Playground scenes.

Test Limit	AVG. GPU	AVG. CPU	AVG. FPS	AVG. Tris
100k objects	25.58ms	17.25 ms	40 FPS	25-50 mil Tris
200k objects	43.40ms	24.32 ms	22.99 FPS	25-75 mil Tris
1 mil objects (Instanced)	21.31 ms	14.98 ms	47.03 FPS	75 mil Tris

Table 5.11: Table of performance for Stress scene.

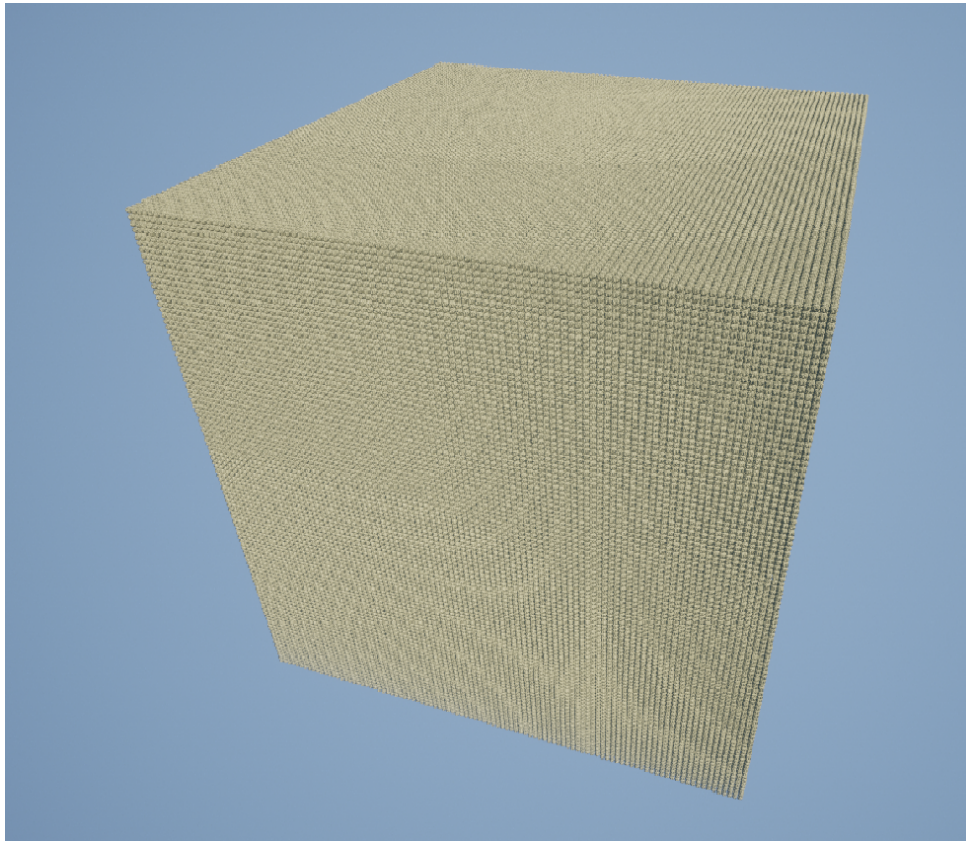


Figure 5.10: Image of the stress scene with 1 million objects being rendered.

Table 5.11 shows that with an RTX 2070, over 100,000 objects with 250,000 triangles can be rendered with still stable framerate. That is around 25 billion triangles without any form of optimization. This however linearly drops as more instances are added. This is most likely because Nanite meshes still scale with instance. After creating instanced meshes and adding one million instanced meshes, the performance improved. However, when trying 2 millions instances, the same problem happened as stated before, where instances started to disappear.

■ 5.8 Test results

The test results have shown, that in terms of performance, Nanite performs similarly to the the standard optimization technique (Section 5.4) and outperforms when working with larger scenes, where a lot of triangles and drawcalls are present (Section 5.7) and that it isn't only dependent on newer generation GPU's. Although according to Unreal Engine documentation that Nanite should not be used for aggregate geometry, the test results in Section 5.6 have shown, that Nanite can work on static grass.

It is my opinion, that the test results were not completely in favor of Nanite. Nanite's potential starts to show, when a lot of details has been added through geometry. However, because abundance of geometry used to be expensive, the standard pipelines used models with less geometry and applied normal maps to fake the details. Because of this, most free models online today are made this way, therefore I had no means to test a geometry detailed models where the difference could be more apparent.



Chapter 6

Conclusion

We have briefly introduced different optimization techniques and how they are used to render detailed objects in Unreal Engine 5. We have then studied how Nanite works and how it is used in Unreal Engine 5 with all its settings. Test results have shown, that Nanite has either similar or better performance than the standard optimization technique. It had substantially better results, when several millions of triangles had to be rendered it once even with older generation GPUs. Furthermore, the tests confirm, that drawcalls are no longer a concern with Nanite meshes. We have also shown how Nanite might completely revolutionize the workflow of creating models. However, the tests were not done to accommodate this new workflow.

Nevertheless, we have also shown, that Nanite isn't a perfect replacement for all optimization techniques, as it cannot accurately represent foliage and the mesh has to be static.

Nanite is still under development and in the future, will have more functionality, such as animations, VR compatibility, foliage, world offset and more. This all could be an initiative for more tests and research in the future.



Bibliography

- [1] Graham Wihlidal Brian Karis, Rune Stubbe. Nanite a deep dive. https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf, 2021.
- [2] Christopher A. Burns and Warren A. Hunt. The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT)*, August 2013.
- [3] Hung-Kuang Chen, Chin-Shyurng Fahn, Jeffrey Tsai, Rong-Ming Chen, and Ming-Bo Lin. Generating high-quality discrete lod meshes for 3d computer games in linear time. *Multimedia Syst.*, 11:480–494, 05 2006.
- [4] Paolo Cignoni. Normal map example. https://commons.wikimedia.org/wiki/File:Normal_map_example.png.
- [5] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10), oct 1976.
- [6] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. *Proceedings of the Symposium on Interactive 3D Graphics*, 01 2001.
- [7] Patrick Cozzi and Kevin Ring. *3D Engine Design for Virtual Globes*. A. K. Peters, Ltd., USA, 1st edition, 2011.
- [8] NVIDIA Documentation. Nvidia bindless rendering. <https://www.nvidia.com/en-us/drivers/bindless-graphics/>.
- [9] Unreal Engine documentation. Unreal engine 5. <https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/>.
- [10] Unreal Engine documentation. Unreal engine foliage. <https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/Foliage/>.
- [11] G-Truc. How bad are small triangles on gpu and why? <https://www.g-truc.net/post-0662.html>.

- [12] Michael Garland and Paul Heckbert. Surface simplification using quadric error metrics. *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 1997, 07 1997.
- [13] Unreal Engine GitHub. Unreal engine github. <https://github.com/EpicGames/UnrealEngine>.
- [14] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. 1993.
- [15] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. Efficient bvh construction via approximate agglomerative clustering. 2013.
- [16] Kshitij Gupta, Jeff A. Stuart, and John D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. 2012.
- [17] Ulrich Haar and Sebastian Aaltonen. Gpu-driven rendering pipelines. http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_ 2015.
- [18] John Hable. Visibility buffer rendering with material graphs. <http://filmicworlds.com/blog/visibility-buffer-rendering-with-material-graphs/>.
- [19] Hugues Hoppe. Progressive meshes. *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, August 1996.
- [20] Liang Hu, Pedro V. Sander, and Hugues Hoppe. Parallel view-dependent level-of-detail control. *IEEE Transactions on Visualization and Computer Graphics*, 16(5), 2010.
- [21] Intel. Unreal engine 4 optimization tutorial, part 2. <https://www.intel.com/content/www/us/en/developer/articles/training/unreal-engine-4-optimization-tutorial-part-2.html>.
- [22] David Luebke, Martin Reddy, Jonathan D. Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
- [23] Vera Mommersteeg. The differences between deferred and forward rendering. January 2015.
- [24] Federico Ponchio. Multiresolution structures for interactive visualization of very large 3d datasets. 2009.
- [25] Megascans Quixel. Quixel, megascans. <https://quixel.com/megascans>.
- [26] Francisco Ramos, Miguel Chover, Oscar Ripolles, and Carlos Granel. Continuous level of detail on graphics hardware. *Discrete Geometry for Computer Imagery, 13th International Conference, DGCI 2006, Szeged, Hungary, October 25-27, 2006, Proceedings*, 4245:460–469, 2006.

- [27] Issac Trotts, Bernd Hamann, Kenneth Joy, and David Wiley. Simplification of tetrahedral meshes. *Proc. IEEE Visualization*, pages 287–295, 01 1998.
- [28] J. Žára, B. Beneš, J. Sochor, and P. Felkel. *Moderní počítačová grafika*. Computer Press, 2004.




Appendix A

Electronic Appendix

The project and build files can be find in the README, which has the link to the repository. The repository has a folder with the build inside that has an .exe file and a project folder, that can be downloaded and then launched with Unreal Engine 5.

All images are included in the images folder. Latex files can be found in the Latex folder.



Appendix B

User Manual

The build can only be run with GPU's that support DirectX12, as Nanite needs it.

Controls for the build version: There are 7 scenes in total. 6 scenes are pairs of Nanite / LOD. Last scene is the playground and stress test scene.

In order to cycle between scenes, use the 1-7 keys.

To know which scene is currently displayed, press the "G" key.

Use the "WASD" key to move and the mouse to look around.

Use the "L" key to increase speed and "K" key to decrease speed.

When not in the playground scene, you may enable the camera mode, that automatically captures data by the "C" key. The data will then automatically open in the folder.

When in the Playground scene, press "N" to either enable or disable Nanite. You will be prompted by a true(Enable Nanite) / false (Disable Nanite) text in the top right section. Press "M" to enable or disable data gathering. You will be prompted by the true / false text again