MASTER'S THESIS 2025

Real-Time Geometry Reconstruction with Compute Shader Tessellation

Jintao Yu

DEPARTMENT OF COMPUTER SCIENCE
DEPARTMENT OF DESIGN SCIENCES
LTH | LUND UNIVERSITY

Electronic Arts

EXAMENSARBETE
Datavetenskap

Real-Time Geometry Reconstruction with Compute Shader Tessellation

Geometrisk rekonstruktion i realtid med Compute Shader Tessellation

Jintao Yu

Real-Time Geometry Reconstruction with Compute Shader Tessellation

Master's Thesis

Jintao Yu

jintaoyuemail@gmail.com

June 11, 2025

Master's thesis work carried out at Electronic Arts.

Supervisors: Christian Kindahl, ckindahl@ea.com
Michael Doggett, michael.doggett@cs.lth.se

Examiner: Mattias Wallergård, mattias.wallergard@design.lth.se

Abstract

Rendering high-fidelity geometry in real time poses significant challenges due to the large data size, increased computational demands, and memory bandwidth limitations associated with dense meshes. While several techniques have been proposed to address these challenges — such as discrete level-of-detail (LOD) systems, hardware-based tessellation, and virtualized geometry solutions like Nanite. However each comes with its own limitations and constraints which become more pronounced when adapting to modern rendering demands such as dynamic displacement, animated meshes, flexible LOD strategies, and memory-aware mesh reconstruction. To overcome these limitations, we introduce a compute-shader-based geometry reconstruction framework that leverages precomputed tessellation patterns to dynamically refine coarse meshes in real time.

The proposed system is fully programmable, supports arbitrary topologies, and enables GPU-side generation of high-density primitives with efficient memory usage. Through experiments, we compare our method against widely adopted approaches. We demonstrate that our method reduces mesh loading times and improves rendering efficiency while maintaining comparable visual quality. This work contributes a practical, scalable tessellation solution, and offers a step toward flexible, shader-driven geometry pipelines suited for future real-time graphics applications.

Keywords: Computer Graphics, Geometry Processing, GPU-based Tessellation, Mesh Refinement

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Electronic Arts and the Frostbite team for generously providing office space, essential hardware resources, and a supportive and welcoming work environment. Working alongside such talented and kind colleagues has enriched my experience and created many memorable moments throughout this thesis journey.

I am especially thankful to my supervisor Christian Kindahl and my manager Andreas Buller, from the Frostbite team at EA. Their continuous guidance, valuable advice, and constructive feedback were crucial in shaping my research direction and overcoming various challenges. Their support significantly contributed to the progress and quality of this work.

I would also like to extend my heartfelt appreciation to my supervisor at LTH, Michael Doggett. His steady insightful comments, and timely feedback helped me to clarify my ideas and improve the structure and content of this thesis.

Finally, I am deeply grateful to my family and friends, whose unwavering support, understanding, and encouragement helped me persevere through the most difficult and stressful periods of working on this thesis alone. Their belief in me was a constant source of motivation and strength, without which this accomplishment would not have been possible.

Jintao Yu

Contents

List of abbreviations

GLSL	OpenGL Shading Language
CUDA	Compute Unified Device Architecture
LOD	Level of Detail
SSBO	Shader Storage Buffer Object
UBO	Uniform Buffer Object
GPGPU	General-Purpose Computing on Graphics Processing Units
OpenCL	Open Computing Language
VR	Virtual Reality
AR	Augmented Reality
OpenGL	Open Graphics Library
CS	Compute Shader

Chapter 1
Introduction

This chapter briefly introduces the background of geometry refinement techniques, discusses the main challenges in the current field, and outlines the objectives of this thesis.

1.1 Background

Reproducing high fidelity geometry in real-time play a vital role in computer graphics. In recent years, the demand for real-time photorealistic rendering has driven a surge of interest in high-resolution geometry synthesis, particularly in the context of the entertainment industry, AR/VR and virtual production pipelines.

While providing benefits such as better visual effects, high-fidelity geometry also brings several challenges due to its massive data size. These challenges include reduced editability due to complex mesh topologies that are difficult to animate or manipulate in real-time engines. Increased computational burden, for examples more expensive cost of lighting calculations, ray tracing, accurate collision detection, and animation of dense meshes, as well as greater memory requirements for vertices, data and textures. Higher bandwidth is required between CPU and GPU and IO overhead among different shader stages and GPU memory [1].

Generating additional triangles in real-time on the GPU, so-called tessellation, is gradually becoming one of the solutions for rendering a large number of triangles in real-time. In the development of geometry processing technology in these decades, this concept has given rise to a wealth of research results and technical solutions [2–8], especially Hardware Tessellation [9].

However, in terms of flexibility and utilization of GPU resources, Hardware Tessellation in the traditional GPU rendering pipeline is no longer suitable for the iteration and development of modern GPU architectures. With the introduction of the compute shader

[10] and the concept of GPU-driven pipeline [11], it has become possible to use GPU resources flexibly and efficiently. Based on this trend, this thesis prototyped a customized compute shader pipeline to control the segmentation behavior and detail sampling process, with the intention of exploring and implementing a rendering pipeline which utilizes the compute shader tessellation with displacement mapping to restore high-precision models in real time.

1.2 Aim

This thesis explores the latest real-time surface tessellation techniques, with a specific focus on compute shader implementations. Building upon this foundation, it further develops and prototypes a rendering pipeline aimed at supporting high-fidelity geometries. Additionally, the advantages and disadvantages of compute shader-based subdivision techniques are evaluated, including their impact on rendering performance, frame rates, etc. compared to the traditional hardware tessellation and even more advanced technique Nanite [12].

1.3 Research Questions

Based on the objectives and challenges outlined above, this thesis aims to address the following research questions:

1. How can compute shader be effectively utilized to implement an efficient and flexible real-time surface tessellation method?

2. What are the trade-offs between computational complexity and rendering performance when employing a compute shader-based tessellation solution compared to other solutions?

3. Is it feasible to achieve high-fidelity geometry reconstruction in real-time rendering using software-based tessellation combined with displacement mapping? What are the technical challenges involved in this approach?

1.4 Contribution

This thesis presents a real-time geometry reconstruction pipeline based on compute shader-driven tessellation, enabling dynamic surface refinement and displacement mapping directly on the GPU. It introduces a reusable, pattern-based tessellation method implemented entirely in compute shader, capable of generating high-density geometry with arbitrary input topology.

The proposed pipeline is designed for practical use, with particular attention to memory efficiency and visual fidelity. Finally, the system is evaluated through detailed comparisons with traditional approaches such as hardware tessellation and Nanite, demonstrating its effectiveness in both performance and visual quality.

1.5 Sustainable Development Goals

The Sustainable Development Goals, short for SDGs, are 17 global development goals put forward by the United Nations in 2015 [13], aiming to eradicate poverty, protect the planet, and promote peace and prosperity for all humanity by 2030. Our project meet the targets of SDG 4: Quality Education and SDG 9: Industry, Innovation and Infrastructure.

Quality Education

Figure 1: Quality Education

My project has been able to significantly enhance the immersive experience of education and training systems through in-depth research and promotion of real-time high-fidelity graphic rendering technologies, especially in VR/AR teaching platforms [14]. In areas such as medical training, engineering simulation, and historical re-enactment, high-quality graphics rendering combined with virtual reality and augmented reality can create more realistic and interactive learning environments that meet the goals of high-quality education.

Industry, Innovation and Infrastructure

Figure 2: Industry, Innovation and Infrastructure

This project explores flexible and efficient graphical segmentation solutions under GPU architecture, and promotes the innovation of graphical rendering infrastructure for industries such as VR/AR, game engines and digital content creation [14]. The research results help strengthen industrial digitalization capacity building and support the development of emerging creative industries, which is in line with industry, innovation and infrastructure development goals.

1.6 Ethics

Risks of Misleading and False Information

High-fidelity rendering technology in Virtual Reality and Augmented Reality is prone to cause an "Illusion of reality", especially in scenes such as avatars and digital twins. Users may misjudge the authenticity of the images, or even make wrong emotional or behavioral judgments in the virtual world, affecting mental health or social cognition. There is a particular need to be wary of technologies being used to mislead, deceive or manipulate user behavior, e.g., in marketing or disinformation dissemination.

Inclusivity and Accessibility

While high-quality graphics technologies enhance the visual experience, there is also a risk that they may exacerbate the digital divide. The dependence of some advanced technologies on high-end hardware devices may result in only a few users with high-performance devices or large enterprises being able to enjoy the advantages they bring, thus limiting the popularization of the technologies and their accessibility to the general public.

To address this problem, exploring how to achieve high-fidelity visual effects on low-end devices through reasonable algorithmic degradation or approximation has become an important direction to enhance technology penetration and promote educational equity.

Chapter 2
Theory and Related Work

This chapter explains several key theories which are crucial to this thesis and provide an overview of geometry refinement techniques on the GPU, ranging from early vertex shader–based pattern instancing, hardware tessellation pipelines, to recent developments such as Micro Mesh and Mega Geometry.

2.1 Tessellation

In computer graphics, tessellation describes the process of dividing existing primitives into smaller ones thereby achieving higher realism, see Figure 3. The primitives are usually triangles or quads, depending on the tessellation algorithm used. For example, Catmull-Clack [3]. The main advantage of this process is that the GPU can generate a higher density of small tuples in real-time during the rendering phase while only transmitting a coarse mesh which is a model with simplified topology and a lower number of primitives counts. While enriching the detail of the model, improves the visual realism, it also decreases the bandwidth of data transfers between the CPU and the GPU.

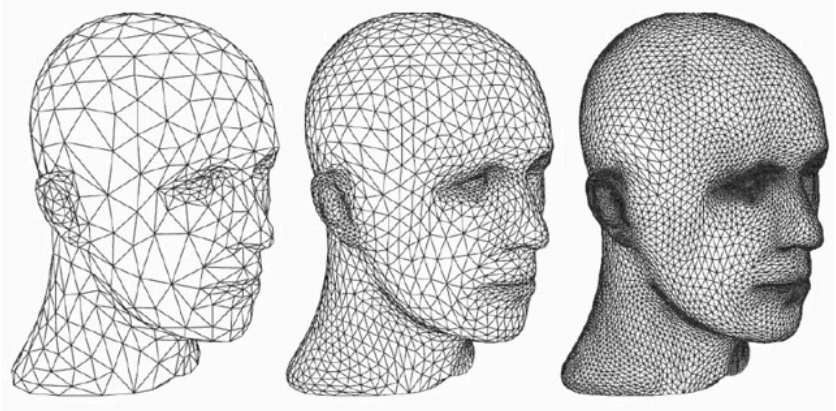

Figure 3: A human head model rendered under multiple tessellation levels. [15]

The widespread use of surface tessellation techniques has led to the development of various derivative technologies, including GPU refinement kernels [5, 6], Hardware Tessellation [9], and GPGPU-based refinement methods [16]. These technologies enable efficient real-time model refinement, allowing the GPU to adaptively tessellate and generate the necessary smaller primitives on-the-fly during runtime. This capability has been widely applied in real-time model refinement, procedurally generated terrains, and ocean simulations, see Figure 4, providing greater flexibility and scalability in the asset creation process.

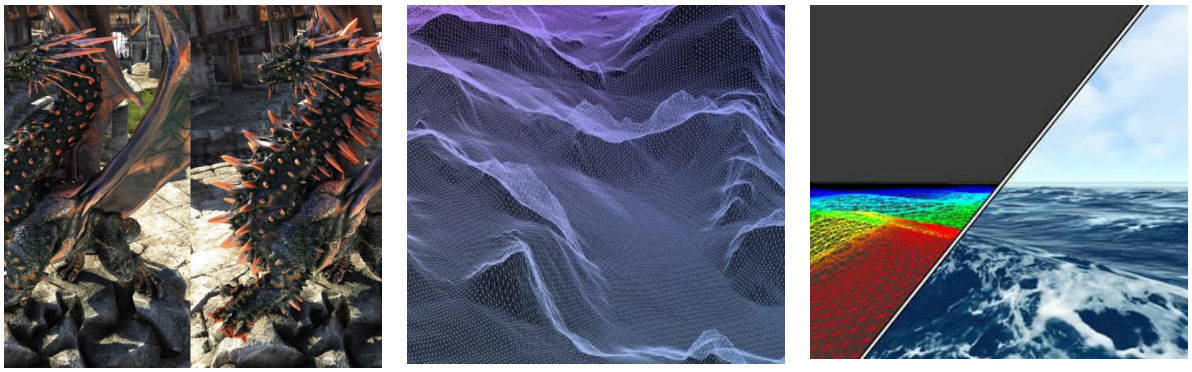

Figure 4: Dragon(left) from m Unigine Heaven Benchmark, procedural terrain(middle) and ocean (right) [17]

2.2 Geometry representation

Geometry representation is the fundamental concept in computer graphics when constructing and processing 3D models. Different model representations in different scenes can have a significant impact in terms of complexity, editability, and rendering effects. Traditional geometric representations such as polygons, parametric surface, and subdivision surface have been widely used in modeling and rendering. However, with the continuous development of Artificial Intelligence and Machine Learning technologies, new geometric expressions, such as point cloud and neural radiance field, have emerged.

2.2.1 Topology

Topology is defined differently in different domains. In graphics geometry, topology describes the connections and organization of model data. In real-time rendering, the topology results from a combination of points, lines and surfaces that describe the connectivity of the model, see Figure 5.

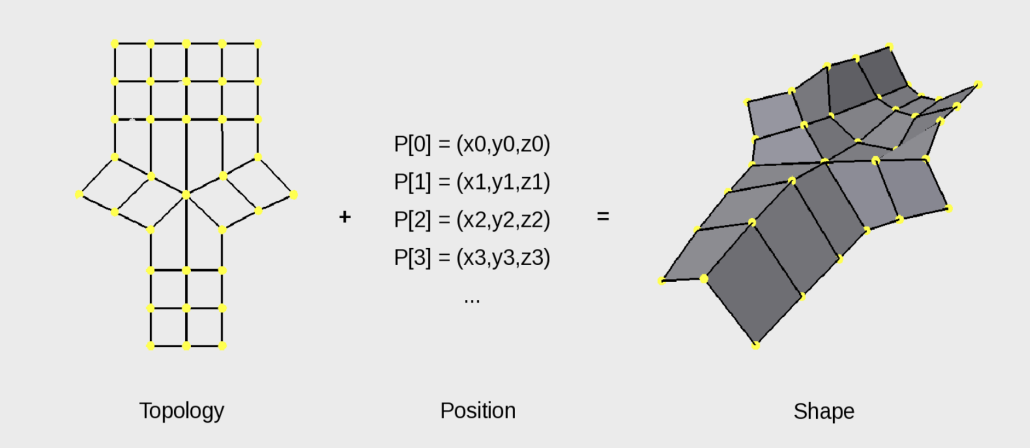

Figure 5: Topology in mesh shaping [18]

In 3D modeling, topology affects the deformation behavior of the model, the subdivision process, and even the lighting results. In subdivision algorithms [3, 4, 19], a good topology result, while reducing the generation of unnecessary vertices, also ensures the correct position and connection relationship of the generated vertices during the subdivision process. Particularly in the entertainment industry like game and movie, the topology result also directly affects the effect of animation deformation and the accuracy of physics simulation.

It is worth noting that sometimes a model that looks exactly the same on the outside may have a completely different topology behind the scenes, see figure 6.

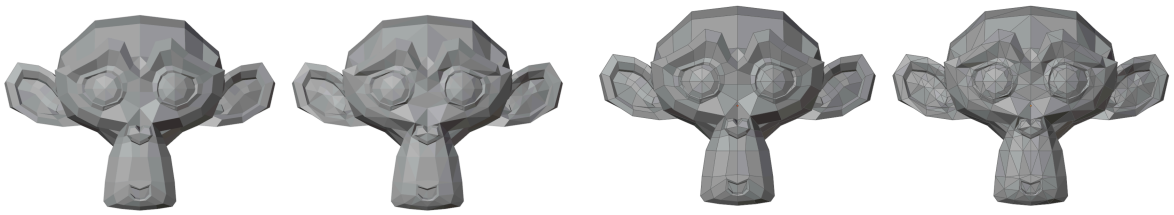

Figure 6: Same visual look(left) but different topology(right)

2.2.2 Polygonal Representations

Because triangles consist of three vertices forming a unique minimal plane, they naturally avoid coplanar problems that can occur with quadrilaterals, such as twisting or folding.

This property has enabled triangles to be established as a fundamental unit in geometric modeling and graphics rendering from the very beginning of computer graphics and has remained a central component in the development of graphics technology in the ensuing decades. Particularly in real-time rendering, triangles have become a core graphical primitive designed into GPU architectures due to their simple data structure, the lack of ambiguity in interpolated results, and the ease of rasterization.

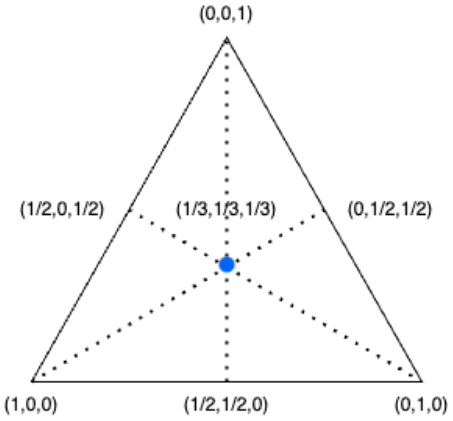

Figure 7: Barycentric coordinates of a triangle

In the GPU, vertex attributes such as position, normal or uv coordinates can be easily computed by simply using barycentric interpolation with the following formula

$$M = \alpha \cdot A + \beta \cdot B + \gamma \cdot C \tag{1}$$

which:

- M represent the interpolated attribute value
- A, B and C are the attributes at the three vertices of the triangle
- α, β, γ are the barycentric coordinates of the triangle satisfying $\alpha + \beta + \gamma = 1.0$

2.3 Compute Shader

To fulfill the demand for arbitrary computational tasks, compute shader [10] as a shader stage, was introduced to execute massively parallel, general-purpose computation on the GPU. Unlike languages designed for GPGPU programming such as CUDA [20], compute shaders naturally benefit from tight integration with graphics APIs, allowing them to directly utilize graphics-related functions

2.3.1 Pipeline and Architecture

Compute shader has a very subtle relationship with the other stages in the overall graphic pipeline. The separate pipeline architecture means that the compute shader is able to operate independently of the traditional rendering pipeline, which provides flexibility but also creates a situation where additional cost being necessary for the compute shader to

interact with the traditional rendering pipeline, which is a limitation of this article and will be explained in detail in section 6.

During the execution of the Compute Shader, the execution model [21] is defined by Work Groups and Invocations, which control how tasks are scheduled in parallel on the GPU. Each workgroup consists of multiple threads, and each thread is an Invocation; both the workgroup and the number of threads define the layout of the computational tasks in a three-dimensional way, and the determination of the dimension is controlled by the dimension of the data you're put in.

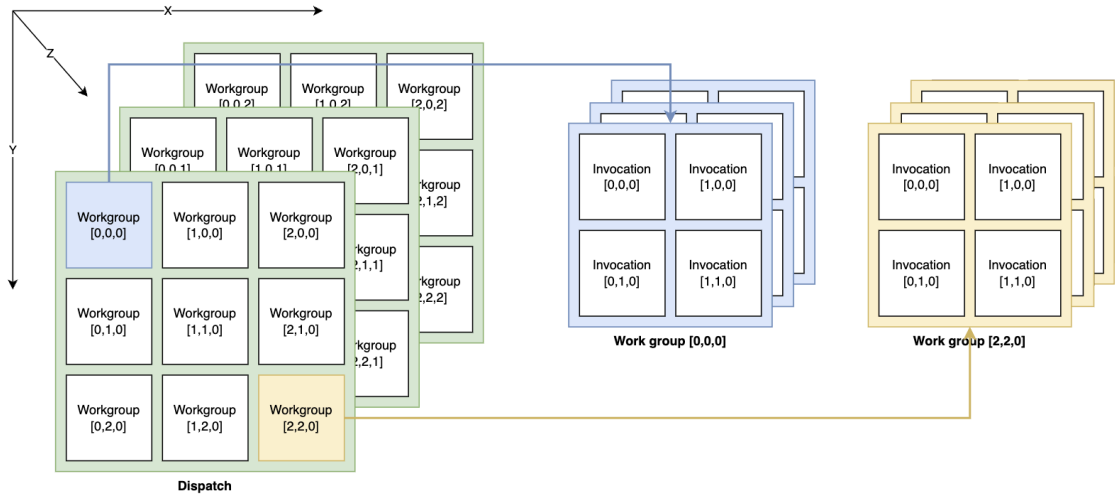

Figure 8: Compute shader execution model

The number of workgroups and the number of threads in each group determine the total number of calls to the Compute Shader. For example, dispatching work groups with dimensions of (3,3,3) including (2,2,1) invocations inside one work group will invoke the compute shader $3 \times 3 \times 3 \times 2 \times 2 \times 1 = 108$ times.

2.3.2 Data Access and Manipulation

In order to enable Compute Shaders to read and write data flexibly, modern graphics APIs provide mechanisms such as Shader Storage Buffer Object (SSBO). and Storage Image. SSBO, as a general-purpose buffer object, is commonly used to store and manipulate structured data, while storage image is used more for accessing image resources and is applicable to various tasks related to image processing. Through these two approaches, a compute shader can efficiently interact with internal and external data to support complex computation processes.

Typically, a compute shader can access data with a thread ID, but during parallel execution, multiple threads may access the same memory address at the same time, especially when reading and writing to shared buffers. To avoid data contention without introducing expensive locking mechanisms, Atomic operations ensure that only one thread can read or write to a target address at a given time, thus avoiding data conflicts while ensuring parallelism.

2.4 Displacement mapping

Displacement Mapping [22] was originally a technique used to generate natural textures, and later it gained widespread application in computer graphics. As an alternative way to restore the details of the original model during real-time rendering, displacement mapping samples pre-baked or mathematically calculated vertex displacement data to modify the positions of the surface vertices of an object.

This allows for more precise detail by actually changing the shape of the object's surface, instead of just cheating the surface normals which normal maps do. Because the surface shape is really modified, displacement mapping also naturally fixes problems like self-shadowing and self-occlusion that normal or bump maps can't handle well.

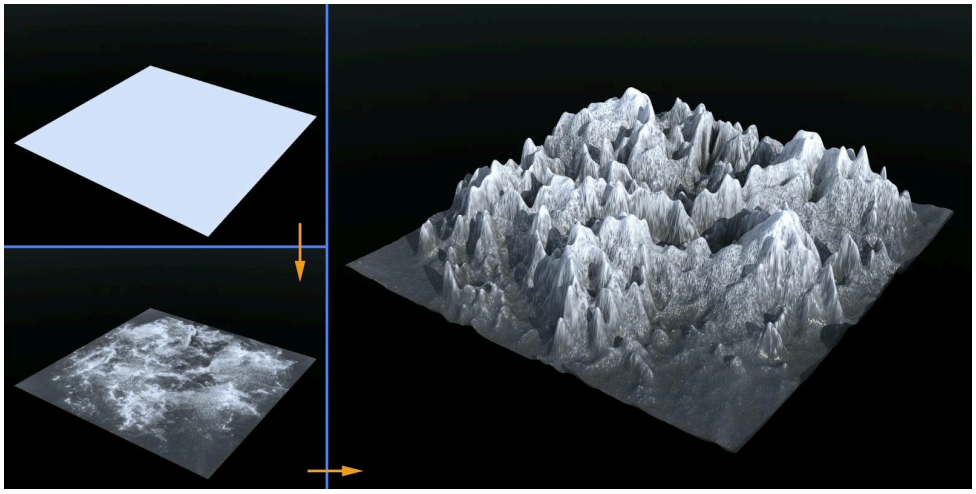

Figure 9: Displacement mapping of a quad(top left) with a height map(bottom left) [23]

Scalar Displacement and Vector Displacement are common in most use cases where the former has a relatively simple data format, using a single-channel grayscale map to describe the displacement magnitude of each point along a certain direction—usually the model's normal direction. It is commonly used on models with simple topology, such as terrain generation see Figure 9. The latter is a more advanced representation. It records the displacement direction and magnitude of vertices in 3D space using the three RGB channels, allowing displacement in any direction: normal, tangent, or even opposite directions. Therefore, it is better suited for complex surface deformations, see Figure 10. However, the increased data size leads to higher storage and bandwidth costs, which limits its application.

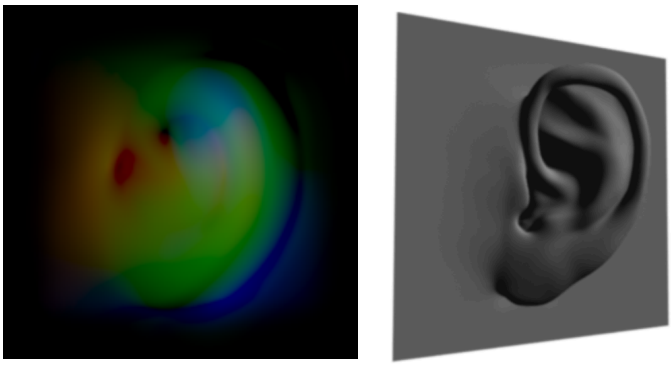

Figure 10: Vector displacement mapping from Autodesk [24]

2.5 Related work

Compute shader tessellation [25], *GPGPU tessellation* [16] are the main focus of the initial research. Later, inspired by Mega Geometry [26] and related techniques, the direction of the study was then shifted to its cluster-based tessellation technique, the subsequent exploration focused on *GPU generic refinement schemes* [5, 27], *mesh shader pipeline* [28], and *micro-triangle* [29]. In addition, to deepen our understanding of the geometry representation, keywords *subdivision surface* [30] and *polygonal representations* [18] are used. Finally to enrich the visual quality, we also studied techniques such as *subdivision surface approximation* [4, 19, 31] and *displacement mapping* [32–34].

2.5.1 Prior Refinement Schemes

Before Hardware Tessellation technique came out, Boubekeur et al. [5, 27] proposed a practice of using a vertex shader to instantiate triangle patterns later Lenz et al. [6] presented an improved version with efficient data storage, and many of the concepts in this article were borrowed from these articles. They saved the required patterns for different tessellation factors in a 3D array on the CPU in advance, and uploaded these data to the GPU memory to allow the vertex shader to directly perform transformations on vertices in different patterns to achieve the generation of more primitives.

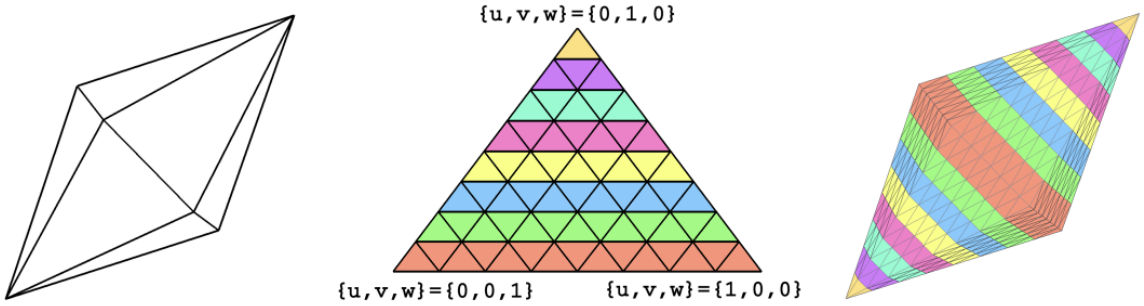

Figure 11: Coarse mesh(left), Refine Pattern(middle), Refined Mesh(right) [5]

Since this method is based on vertex shaders, there is no way for vertex shaders to communicate with each other, and only one vertex can be processed at a time. The original articles don't give the implementation details, so it's not clear how the coarse triangle and pattern are mapped.

Later, interpolation-based techniques were proposed such as Phong Tessellation [7] and PN-Triangle [8, 35], which generate smoother subdivided surfaces by interpolating existing vertices and fitting curvature. These methods primarily aim to improve visual quality. Although they do not directly increase new topological structures, their visual effects closely resemble those of traditional geometric subdivision.

Meanwhile, GPGPU-based methods were also being developed to maximize the utilization of the GPU's parallel computing power. Schwarz et al. [16] introduct an adaptive tessellation method based on CUDA, other GPU based tessellation methods like [25, 36] were presented in recent years utilizing compute shader to generate massive amount of triangles in the GPU.

2.5.2 Hardware Tessellation

To address the CPU-GPU data transfer bottleneck associated with high-precision modeling, Direct3D 11 [37] equipped with a new hardware segmentation technology, laid the groundwork for the development of a new dynamic LOD system.

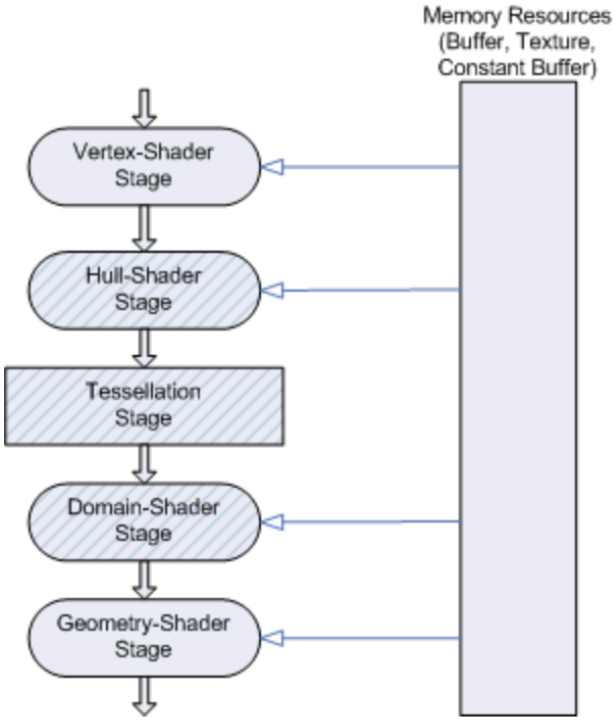

Figure 12: Hardware Tessellation pipeline from Direct3D 11 [37]

Hardware tessellation in [9] provides a more flexible tessellation object. In the rendering pipeline, primitives are abstracted into patches, which are collections of vertices. A patch with 3 vertices represents a triangle, while one with 4 vertices represents a quad. Adaptive effects are achieved by controlling the subdivision level of individual edges. The system also supports fractional subdivision factors and various spacing strategies, enabling more detailed and precise tessellation.

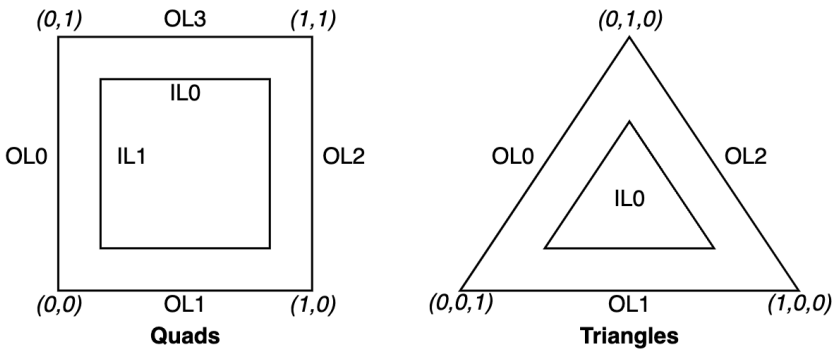

Figure 13: Common patches in Hardware Tessellation

Hardware tessellation exposes programmability through programmable shaders such as the hull shader and domain shader, offering developers a certain degree of flexibility and extensibility. This enables efficient hardware-accelerated tessellation computations while maintaining strong performance advantages.

Hardware tessellation improves real-time detail processing but has a few limitations. It operates on patches independently without shared topology, which causes duplicate vertices to be generated on shared edges. Additionally, tessellation output is directly sent to the pipeline without writing to GPU memory, and since tessellation is done in a non-programmable GPU stage, manual removal of duplicates is difficult.

2.5.3 Nanite

With the release of Unreal Engine 5, Nanite [12] offers a geometry detail management approach different from traditional surface tessellation. Unlike conventional refinement schemes that use a coarse mesh and dynamically generate more triangles on the GPU to restore detail, Nanite partitions the original high-precision model into clusters—each a collection of a certain number of triangles—and dynamically loads geometry data at different levels of detail during rendering.

Nanite excels in static scenes by preserving high-detail geometry with efficient draw calls and GPU scheduling. However, it struggles with dynamic geometry like skinning or procedural deformation due to the absence of a coarse mesh. Animations require processing large numbers of high-resolution triangles, leading to heavy computational overhead.

2.5.4 Micro-Mesh and Mega Geometry

Unlike the previously mentioned refinement patterns, Micro Mesh [29, 38–40] adopts a fundamentally different approach to reduce GPU memory usage and rendering costs. It converts a mesh into a set of compressed micro-meshes, which are encoded using barycentric coordinates and compiled into a compact binary format.

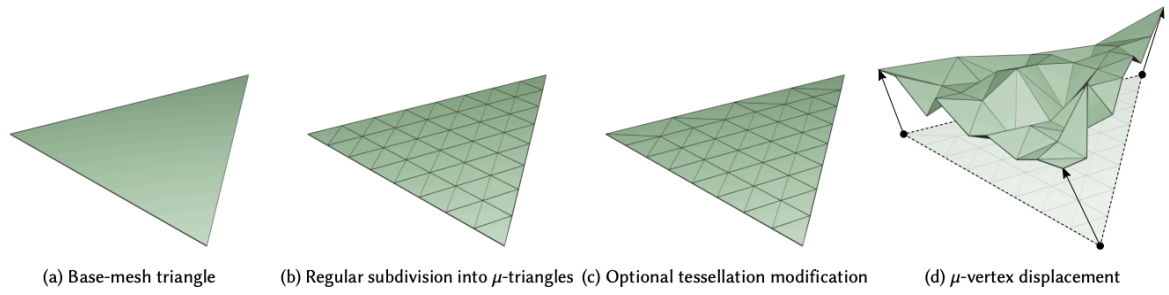

Figure 14: Micro Mesh construction [29]

On the GPU side, NVIDIA's modern GPU architectures [41] introduce a new Geometry Engine that provides native hardware acceleration for Micro Meshes. This allows micro-triangle data to be efficiently unpacked and traversed during real-time rendering.

However, the construction of Micro Meshes involves a series of complex preprocessing steps—such as determining tessellation rates, casting rays from each micro vertex to compute detailed vertex offsets, building acceleration structures, and compressing the generated data. As a result, Micro Meshes are primarily used in offline rendering or high-quality ray tracing tasks.

Compared to Micro Triangles, Mega Geometry [26] also adopts the concept of adaptive refinement patterns from early GPU-based methods, using a cluster-based structure for geometry management. It performs efficient dynamic tessellation on the GPU in real time through Task and Mesh Shaders, but the process is still triangle-based, leading to duplicate vertices on shared edges.

Chapter 3
Methodology

This chapter aims to introduce the methodology involved to address the research questions and challenges, including the implementation overview as well as tools and framework used in this project.

3.1 Implementation Overview

The implementation can be broadly divided into two main phases: offline and runtime. We precompute all the required triangle patterns in advance and upload them to GPU memory so that the compute shader can use them later, see Figure 15.

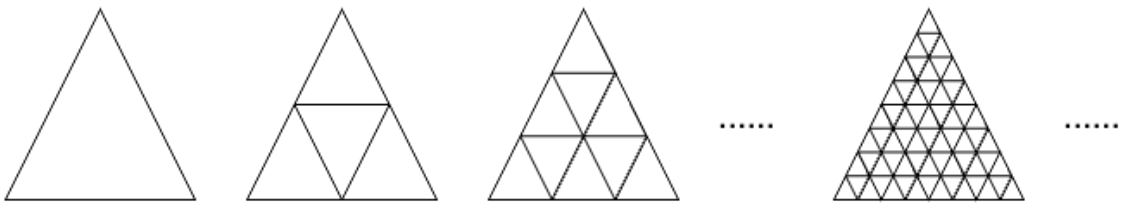

Figure 15: Pre-computed patterns

During runtime, for each visible triangle from the input coarse mesh, we select a triangle pattern based on various metrics such as a fixed tessellation rate or the distance to the camera. The selected pattern is then applied in the compute shader to generate smaller triangles. If a displacement texture is available, displacement mapping is performed afterward, see Figure 16.

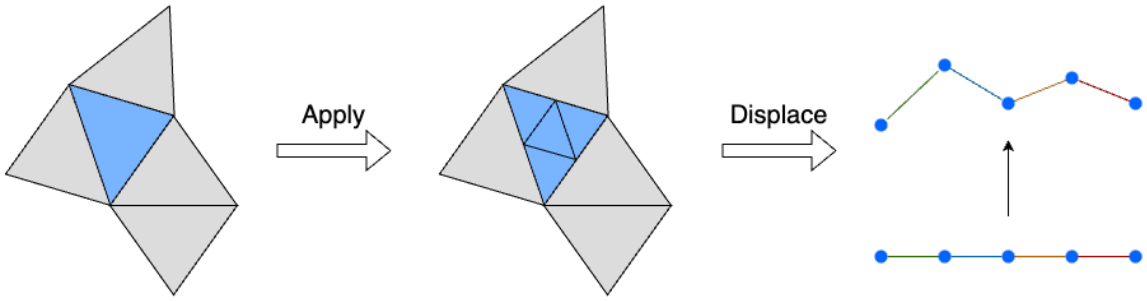

Figure 16: Short version of real-time process

3.2 Tools and Framework

The development of our system is based on several key tools and frameworks. The graphics backend is built on Vulkan with the shading language GLSL. *nvpro* [42] from Nvidia is used to fast build the Vulkan application and provide several useful utilities, including convince UI framework and profiling tools. The C++ library *tinyobjloader* [43] is used to load models in wavefront format, the open-source *stb* [44] library is employed for image loading. *ktx2* [45] is the primarily format of the texture used in this project to reduce GPU memory usage. For graphics debugging, we rely on Nvidia *Nsight* and *RenderDoc*. The 3D model used in this project was created in ZBrush based on the *Big Guy* character.

Chapter 4
Implementation

In this chapter, we present the implementation details of the compute shader based tessellation prototype.

4.1 Pipeline Overview

Building upon the insight from the previous studies, we provide a compute shader based pipeline to demonstrate how we can leverage the GPU's massive parallelism to reconstruct high-fidelity geometry in real time. To better illustrate the structure and functionality of our system, we present detailed explanations of several crucial stages in our pipeline in the following sections.

4.1.1 Resources Overview

This section summarizes the GPU resources used throughout the pipeline. Each resource plays a specific role in different stages of the rendering process, ranging from visibility determination to normal recalculation. Table 1 provides a detailed breakdown of these resources, including their types, usage stages, and descriptions.

Table 1: Resources Overview

Name	Type	Stages	Description
Input Mesh data	SSBO	All Compute Stages	Mesh data of the coarse mesh
Pattern Table	SSBO	All Compute Stages	Different levels of refined pattern
Lookup Table	SSBO	All Compute Stages	Entry point to Pattern Table
Triangle Visibility	SSBO	All Compute Stages	ID of the visible triangles
Data Counters	UBO	All Compute Stages	All atomic counters
Vertex Tessellation Rate	SSBO	All Compute Stages	Tess factor of each vertex
Indirect Commands	SSBO	Indirect Setup	Setup indirect commands
Scene Information	SSBO	All Compute Stages	Scene Configuration
Frame Constants	UBO	Vertex Shader Fragment Shader	Constants for each frame
Refined Vertices	SSBO Vertex Buffer	Tessellation Stage Vertex Shader	Generated vertices
Refined Indices	SSBO Index Buffer	Vertex Shader	Generated triangles
Normal	SSBO	Fragment Shader	Normal of deformed mesh

In Table 2, we present the mesh data we used in this project.

Table 2: Input Mesh Data Review

Name	Triangle Counts	Vertices Counts	Edges Counts	Disk Size	Memory Size
Big guy coarse mesh	2900	1452	2900	145 KB	292 KB
Big guy detail mesh	2,969,600	1,484,802	2,969,600	241 MB	363.7 MB
Big guy displacement texture	N/A			1.2 MB	13.2 MB

4.1.2 Data Pre-Processing

In the offline phase, two key components are generated: the tessellation patterns set and a lookup table that indexes them in the GPU memory. Each pattern consists of a set of vertices and triangle indices, which are stored in two large, contiguous memory blocks.

The size of each block is determined by the total number of vertices and indices across all supported tessellation levels (from level 0 to the maximum), as shown in left side of Figure 17. For example, the first three vertices in the memory block belong to the first pattern, the next six to the second pattern, and so on. The same layout applies to the index buffer.

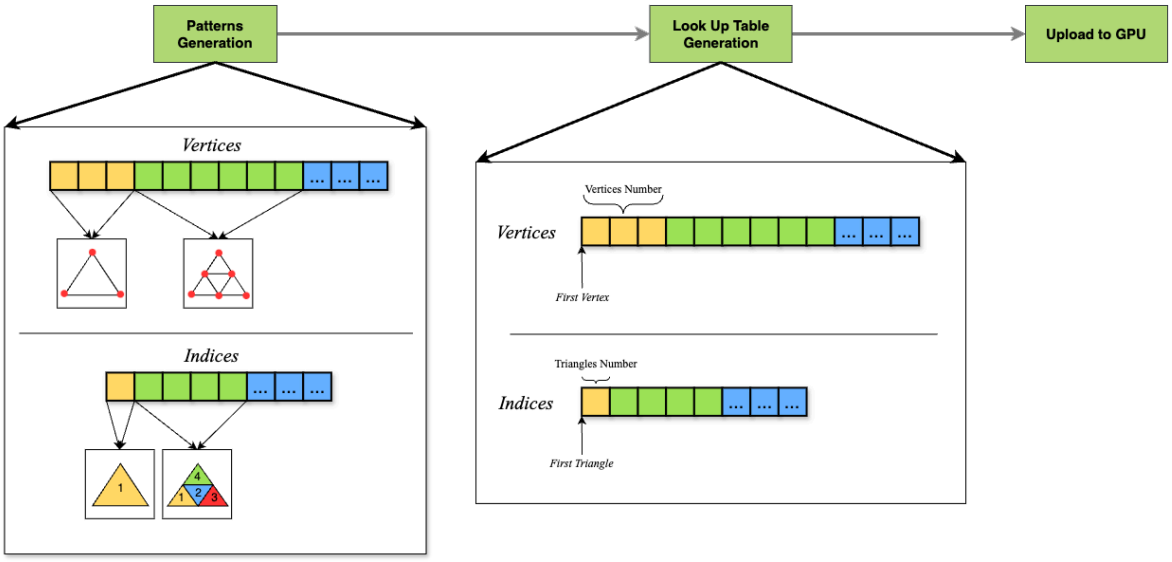

Figure 17: Offline processes

The lookup table stores the entry points for each pattern, making it efficient to access the appropriate pattern directly in GPU memory during execution. Additional information such as the indices of edge vertices on the coarse triangle is also stored to facilitate vertex reuse when applying patterns.

4.1.3 Real-Time Framework

During the runtime phase, the system runs a sequence of compute-shader-based stages: resource cleanup, triangle visibility determination, tessellation level computation, pattern-based tessellation, and normal recalculation, see Figure 18. Since compute shaders operate within an isolated pipeline and cannot directly render to the screen, we must transfer the generated vertex and index data to the traditional rendering pipeline using SSBO, which are then bound to the vertex and fragment shaders.

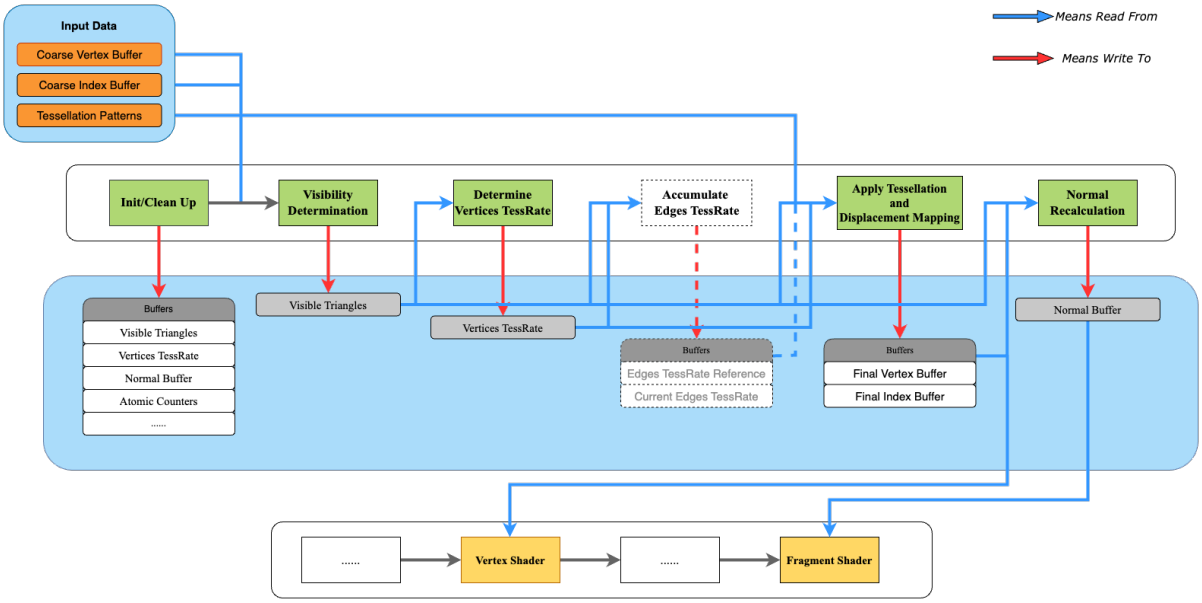

Figure 18: Realtime compute shader based framework

At the very beginning of the pipeline, several resources must be reset, because they varied every frame, such as the buffer stores the triangles that are visible, the atomic counters that track generated vertices and triangles, see Table 1. Once the resources are properly reset, the system proceeds to determine which triangles in the input mesh are visible to the current camera, then we compute the tessellation level for each vertices from the visible triangles, based on that, the tessellation compute shader will fetch the corresponding pattern from the tessellation patterns set with the lookup index, generated vertices and indices data are then written to two final buffer and subsequently bound to vertex shader to finalize the rendering.

If there are displacement textures available, vertex positions are adjusted during the tessellation stage and one additional compute pass is needed to re-calculate the normals from the the displaced vertices.

4.2 Patterns Generation

In order to achieve flexible and efficient GPU-based tessellation, this section presents algorithms for generating reusable uniform triangle patterns. With the help of these patterns, smaller, finer triangles can be quickly generated on the GPU side by simple and efficient interpolation of the barycentric coordinates.

It's crucial to note that the ordering of the vertices and triangles matters as it affects the efficiency of the subsequent vertex reuse algorithms. Therefore, the vertices and triangles in each mode are arranged in left-to-right and bottom-to-top order, as detailed in Figure 19.

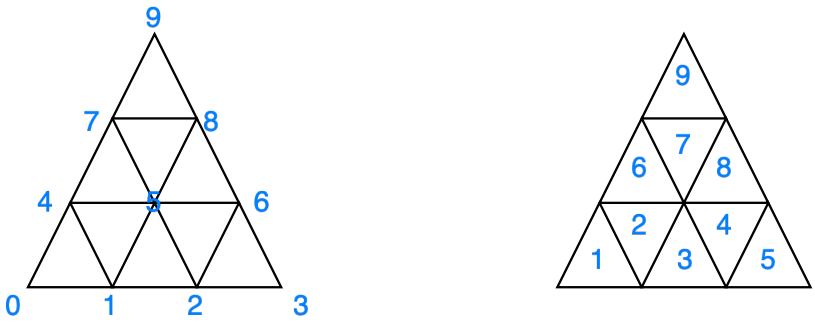

Figure 19: Vertices order(left), Triangle order(right)

For each vertex, we store its corresponding barycentric coordinates. For each vertex, we store its corresponding barycentric coordinates. As shown in Figure 20, the triangle is divided horizontally from top to bottom into multiple layers, and each level can be regarded as a linear interpolation along u, with v increasing by one over the depth value, i.e. tessellation rate in each iteration. Due to the properties of barycentric coordinates and to reduce memory usage, we only store the u and v components of the barycentric coordinates for each vertex, since w can be calculated with $w = 1.0 - u - v$.

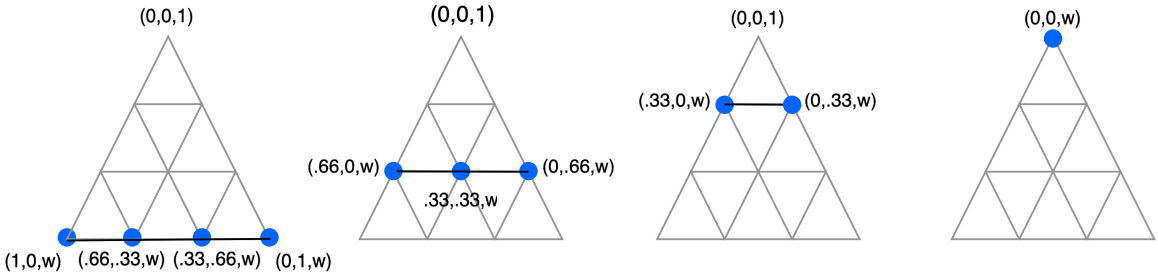

Figure 20: Vertices interpolation with gradually increasing depth value

Algorithm 1 Pattern Vertices Generation

```
function PatternVertices(max_tess_rate, pattern_vertices)
    idx ← 0
    cur_tess_rate ← 1
    for cur_tess_rate ≤ max_tess_rate do
        i ← 0
        j ← cur_tess_rate
        for i ≤ cur_tess_rate do
            for j ≥ cur_tess_rate − i do
                fracs ← i/cur_tess_rate, j/cur_tess_rate
                bary_coord ← Lerp[fracs]
                pattern_vertices[idx] ← bary_coord
                idx ← idx+1
                j ← j-1
            end
```

```
15:         i ← i+1
16:       end
17:       cur_tess_rate ← cur_tess_rate + 1
18:     end
19: end
```

To efficiently build reusable Uniform Patterns, we precompute the triangle topology, i.e. index order of each pattern. We make the current triangle a quadrilateral by adding a virtual triangle along its right side, so that two simple loops can compute the order we want, see Figure 21. We use the depth of the pattern as the loop variable.

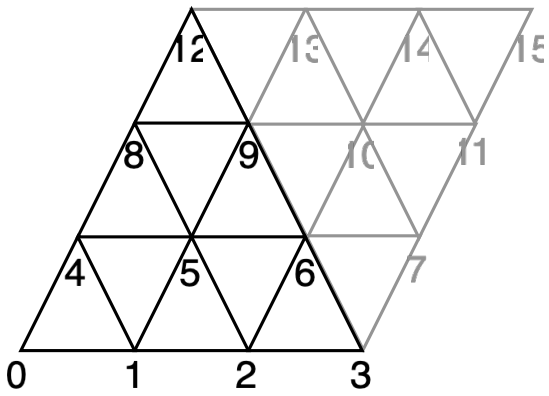

Figure 21: A example pattern for computing triangle indices

We use the depth of the pattern as the basis for the loop, starting at the bottom, and constructing the triangles in between, using ID information about the vertices of the current layer and the next layer in turn. To avoid unnecessary memory traversal and index out of bounds, we don't need the extra vertex information on the virtual triangles, so we need to calculate the maximum index value for each layer by using the partial sum of an arithmetic sequence and the depth value, and give up on generating the index of the current triangle when there are vertices exceeding the maximum index value computed by formula (2) for the corresponding layer.

$$S_n = \frac{n * (a_n - a_1)}{2} \tag{2}$$

Algorithm 1 Pattern Indices Generation

```
1: function PatternVertices(max_tess_rate, pattern_indices)
2:     idx ← 0
3:     prev_row_len ← 0
4:     cur_tess_rate ← 1
5:     for cur_tess_rate ≤ max_tess_rate do
6:         depth ← cur_tess_rate
7:         j ← 0
8:         for j ≤ depth do
9:             max_idx ← (j+2)*((cur_tess_rate +1)*(cur_tess_rate −j)) / 2 − 1
10:            num_row_quad ← cur_tess_rate - j
11:            cur_row_len ← cur_tess_rate + 1 - j
```

```
        i ← 0
        for i < num_row_quad do
            start_point ← i + (j*(cur_tess_rate +1)+(cur_tess_rate −j)) / 2
            fir_point ← start_point
            sed_point ← start_point + 1
            thi_point ← start_point + cur_row_len
            pattern_indices[idx] ← vec3(fir_point, sed_point, thi_point)
            idx ← idx+ 1
            fir_point ← start_point + 1
            if fir_point + cur_row_len > max_idx then
                sed_point ← 0
                else
                    sed_point ← fir_point + cur_row_len
                end
            end
            if sed_point == 0 then
                continue
            end
            thi_point ← start_point + cur_row_len
            pattern_indices[idx] ← vec3(fir_point, sed_point, thi_point)
            idx ← idx+ 1
            i ← i+ 1
        end
        j ← j + 1
    end
    cur_tess_rate ← cur_tess_rate + 1
  end
end
```

Using the above method, we can efficiently pre-generate triangle uniform patterns at arbitrary resolutions on the CPU side to support high-concurrency tessellation on the GPU. For detailed pseudocode, please refer to Algorithm 1 and Algorithm 2.

4.3 Determination of Visibility and Tessellation Rate

In addition to resource initialization, the formal execution of a real-time rendering pipeline begins with two key steps: rejection of non-visible triangles and surface tessellation rate assignment. Discarding invisible triangles at the initial stage of the pipeline is essential because these triangles do not contribute to the final result in subsequent computations, but consume valuable computational and memory resources, resulting in wasted performance.

To this end, we employ a strategy similar to fixed function culling in the graphics hardware, i.e. back culling to determine the visibility of triangles. Specifically, we first construct a face normal for each triangle and compute the dot product between it and the camera view direction. If the dot product is greater than zero, the triangle is facing the camera and is visible; conversely, if the dot product is less than or equal to zero, the

triangle has its back to the camera and should be culled. This approach is both simple and efficient, and is especially suitable for parallel computing environments.

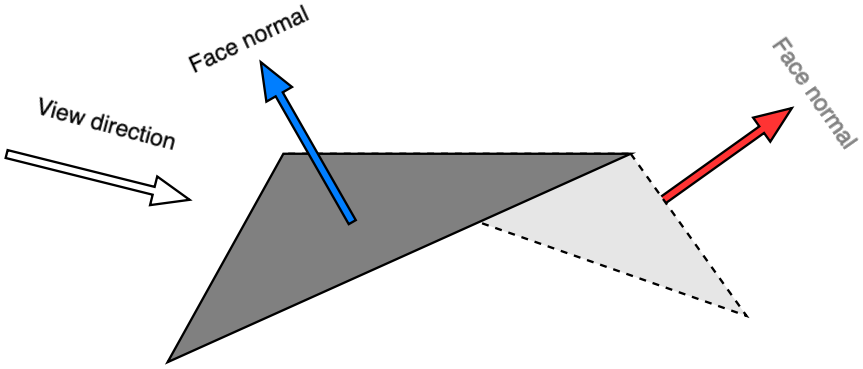

Figure 22: Illustration of backface culling

However, the focus of this paper is on the design of the rendering pipeline framework based on the Compute Shader. In order to simplify the implementation and avoid the T-Junction problem caused by the inconsistency of the tessellation levels of neighboring triangles, we choose to uniformly assign the same level to all vertices. The possible impact of this strategy and how we further deal with the T-Junction problem will be discussed in detail in the following sections.

4.4 Tessellation and Displacement

After the preparatory work, the actual execution of Tessellation itself is not complicated. The core of Tessellation is to use the barycentric coordinates of the pre-generated pattern and the vertex data of the original coarse triangle to calculate the real vertex positions after refinement by the interpolation formula (1).

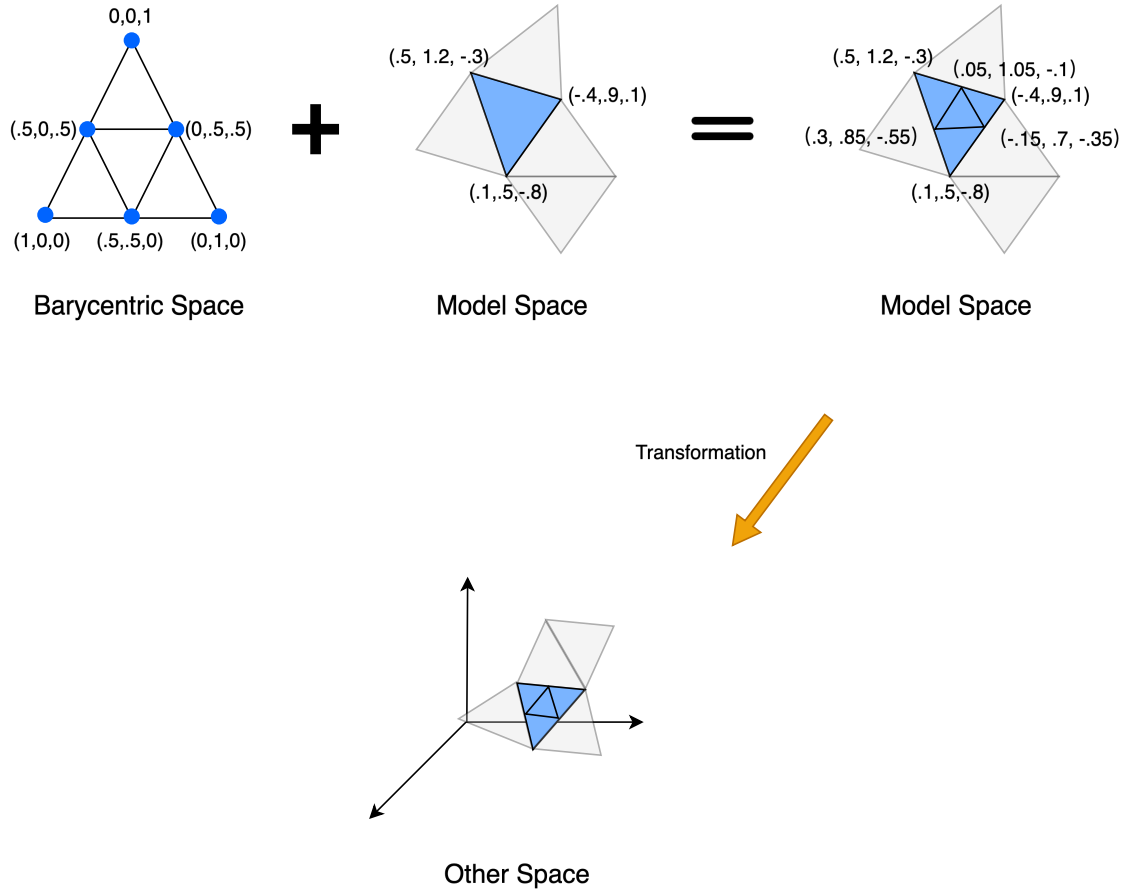

Figure 23: Tessellation process

To further enhance geometric detail, displacement mapping can be applied on top of the interpolated surface. Once the base position is computed through barycentric interpolation, we offset it along the original surface normal using a displacement value sampled from a displacement texture. This value is typically fetched using the corresponding interpolated texture coordinates.

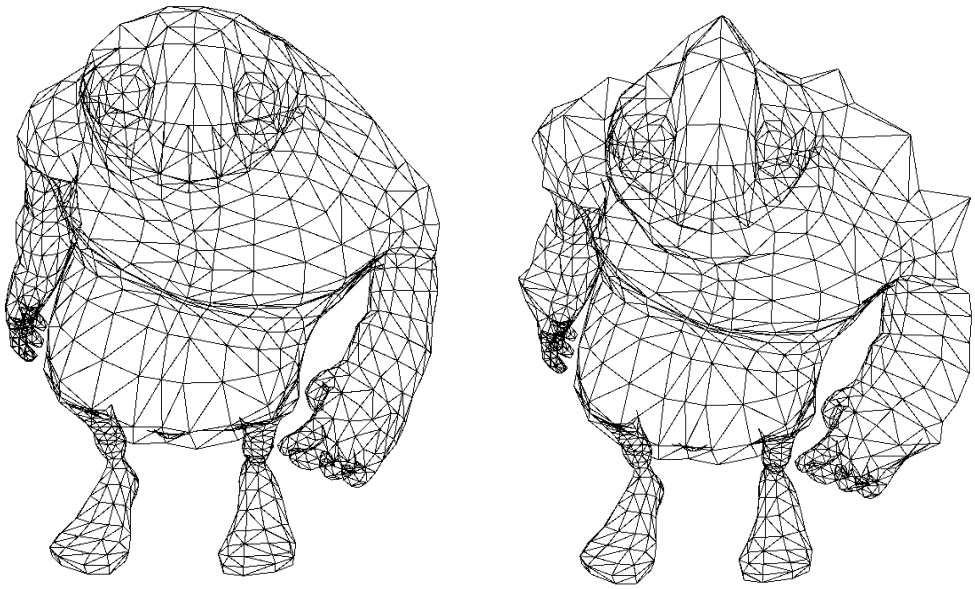

Figure 24: Displacement Mapping

4.5 Normal Re-Calculation

In the case of dynamic water or procedurally generated terrain, where vertex positions are offset by mathematical functions or noise functions such as Perlin noise [46], normals can also be directly calculated from the corresponding mathematical expressions. Even for terrain, whose normals naturally point mostly upwards, reasonably accurate lighting results can still be achieved without extensive processing.

However, when dealing with complex 3D models, the situation is different. If we don't rely on a normal map, we need to re-calculate the displaced normals in some other way. Computing derivatives from scalar displacement texture gives us the normal in tangent space, but it requires extra data to build a TBN matrix. Calculating partial derivatives per point also requires access to neighboring points, which is not straightforward for triangle primitives, see Figure 25.

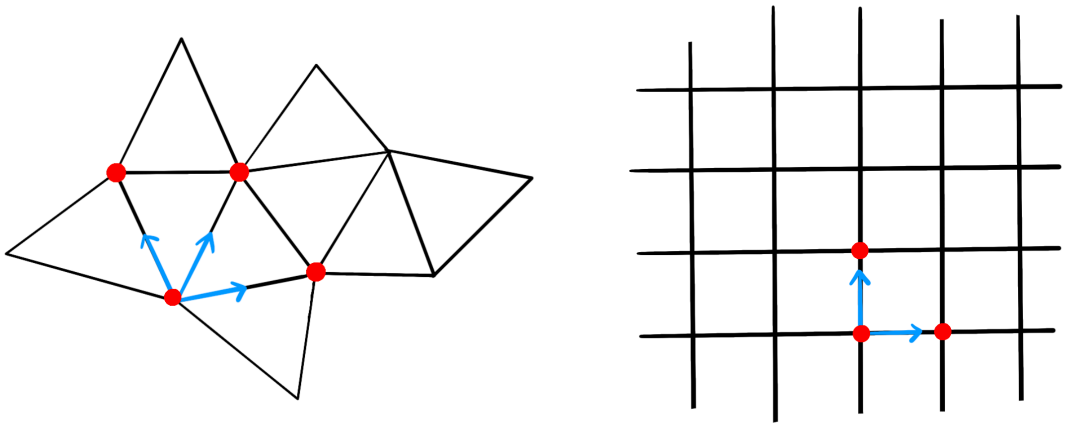

Figure 25: Quad has better topology to calculate derivatives than triangle

If only face normal is used, each triangle will appear distinctly flat and the overall effect will be stiff. Therefore, it is more desirable to compute normals for each vertex, so that during the rasterization phase, the GPU can interpolate the normals inside the triangles, resulting in a smoother surface effect. Specifically, for each tessellated triangle, we first construct its face normals and accumulate them to each of the three vertices constituting the triangle. Taking the topology in Figure 26 as an example, the normal at the red vertex is the average of the face normals of all triangles connected to the red vertex, the final normal of the red vertex is jointly determined by the face normal of accumulated by all its neighboring triangles.

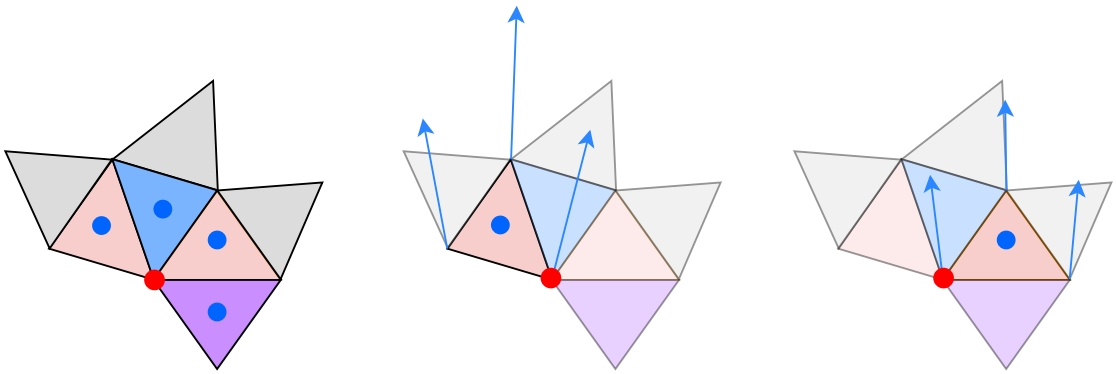

Figure 26: Normal recalculation at the red vertex based on connected triangles' face normal (left), and the process of accumulating face normals (middle and right).

Since this process is executed in parallel in the Compute Shader, the final normals of all vertices are not immediately available until the end of the calculation. Therefore, these normals need to be normalized to ensure they are in the correct range before they are subsequently used.

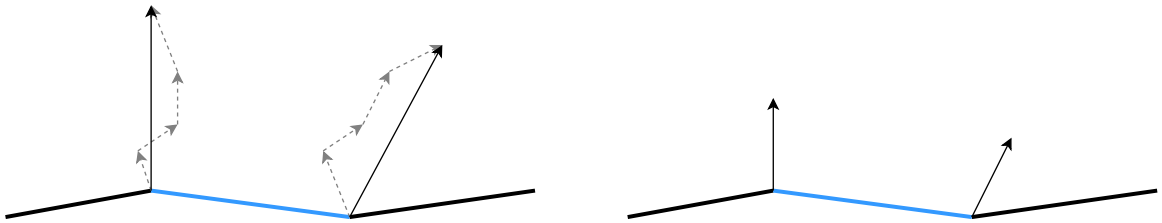

Figure 27: accumulated normals(left) and normalized normal(right)

Chapter 5
Evaluation and Results

In this chapter, we present a comprehensive evaluation of our pattern-based compute shader tessellation framework. We adopt a combination of quantitative and qualitative criteria, including most of the stages in the pipeline. In terms of performance evaluation, we compare our approach against existing methods including hardware tessellation, other GPU tessellation schemes and virtualize geometry system Nanite.

5.1 Visual Quality Analysis

The evaluation of visual quality mainly focuses on geometric fidelity, attribute coherence, shading, and visual consistency.

5.1.1 Visibility and Tessellation

Here in Figure 28 and 29, we show how the same tessellation pattern with level 10 can be reused across models with completely different topologies. Despite variations in vertex connectivity and surface curvature, the pattern-based method maintains geometric correctness and visual consistency.

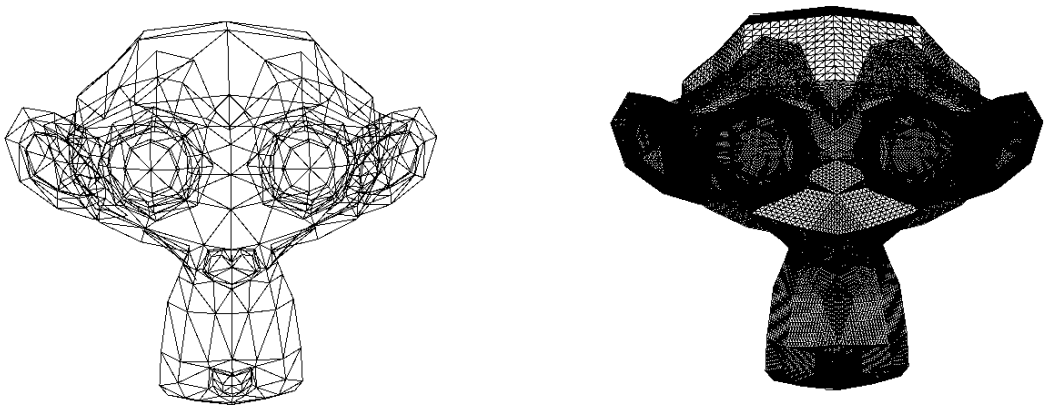

Figure 28: Suzzane coarse mesh(left) and tessellated Suzzane mesh

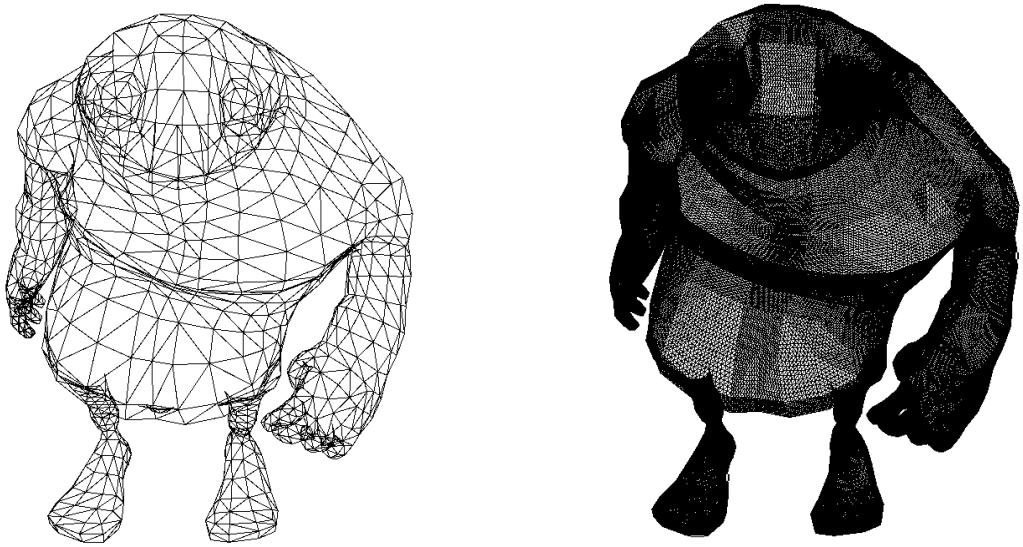

Figure 29: Big guy coarse mesh(left) and tessellated Big guy mesh

However, it is possible to increase the tessellation level to 100, the following image demonstrate what is the model looks like with pattern 100 tessellation level.

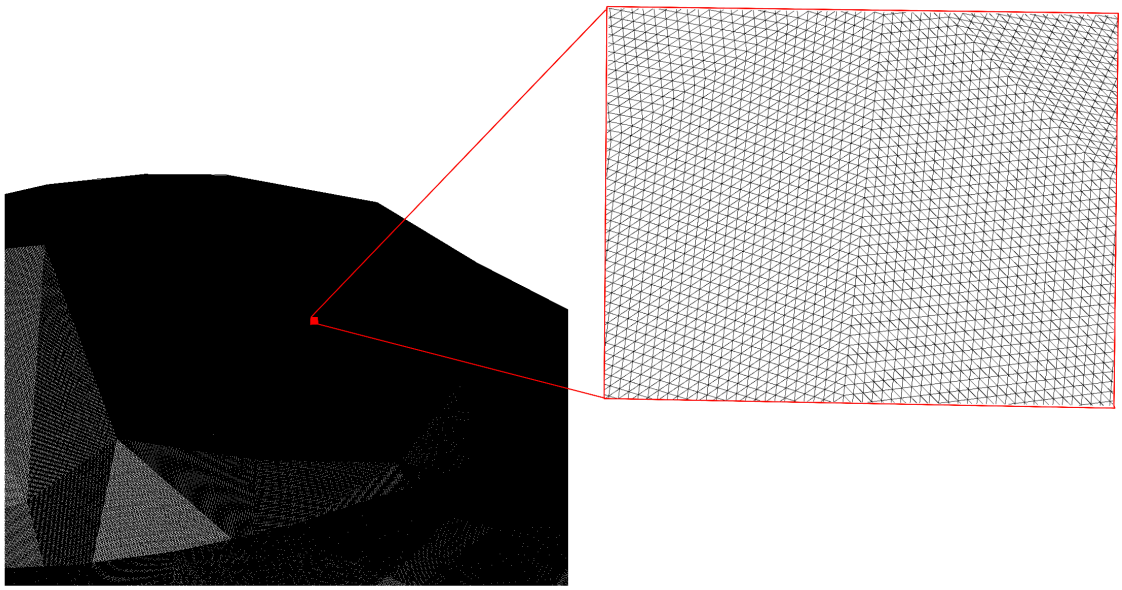

Figure 30: Big guy with 100 tessellation level

Figure 31 compares the impact of culling operations on visibility at a fixed camera view. As can be seen in the second image, even a simple culling algorithm can help us save almost 50% of the computation and memory overhead.

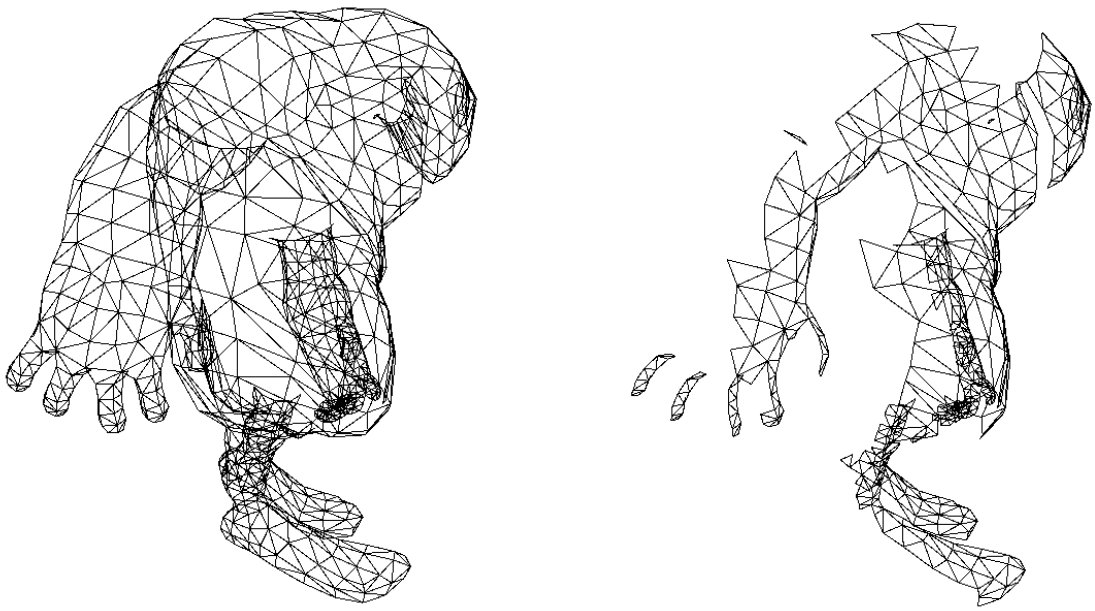

Figure 31: Side view of disable culling(left) and enable culling(right)

5.1.2 Displacement Mapping

Tessellation and displacement mapping have always played an important role in procedural terrain generation, and our method can also be applied to common terrain generation scenarios, see Figure 32.

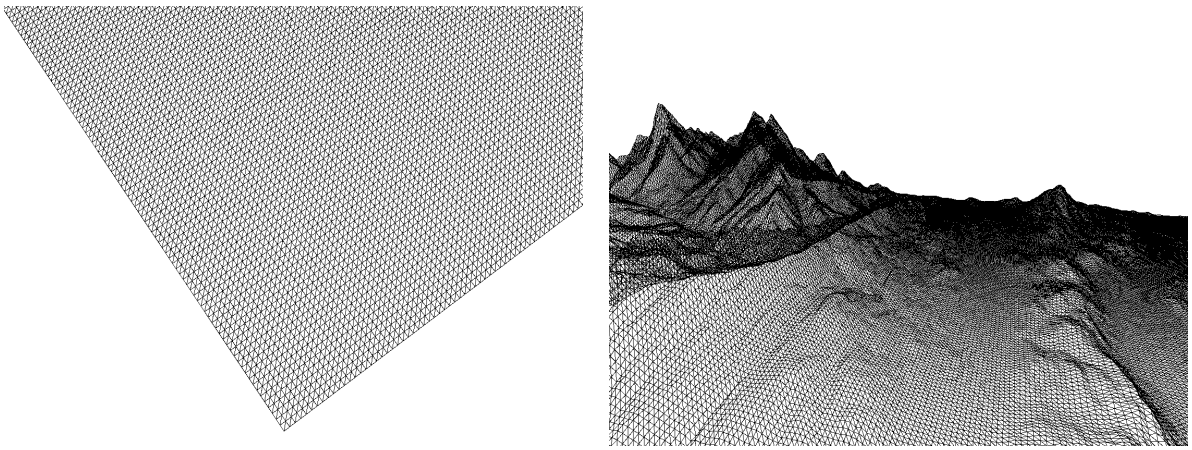

Figure 32: Terrain with 50 tessellation level(left) and with displacement mapping applied(right).

To evaluate the generality of our method, we further apply displacement to complex 3D models. Figure 33 shows that, even with arbitrary topology, our framework effectively refines the surface, extending beyond traditional terrain applications.

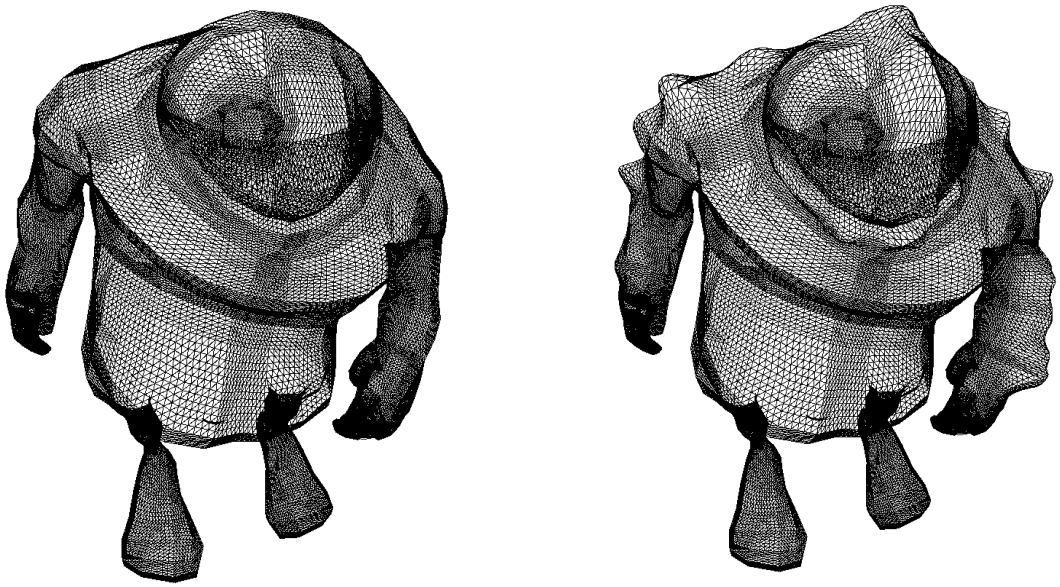

Figure 33: Big guy with 10 tessellation level(left) and with displacement mapping applied(right).

5.1.3 Normal Recalculation

In Figure 34, we can see that the lighting calculation after displacement is not particularly accurate.

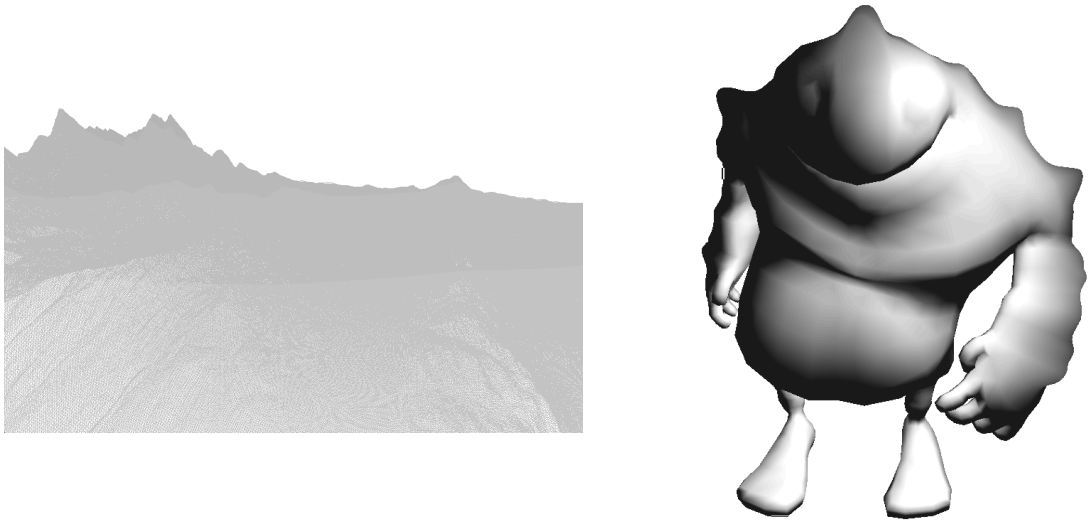

Figure 34: Lighting results with original terrain normal(left) and model normal(right)

This is because we have not re-calculated the normals of the displaced vertices. Generally speaking, when a mesh is deformed, the curvature of the surface changes, and so do the corresponding tangents and normals, see Figure 35. While this may not be obvious on a terrain, the results in Figure 36 show that it is easier to see on more complex 3D models.

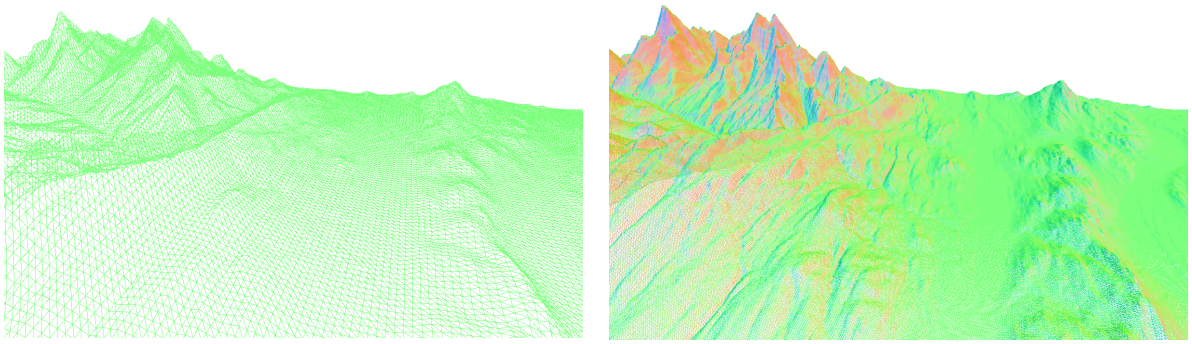

Figure 35: Terrain with input normal(left) and re-calculated normal(right) after displacement mapping

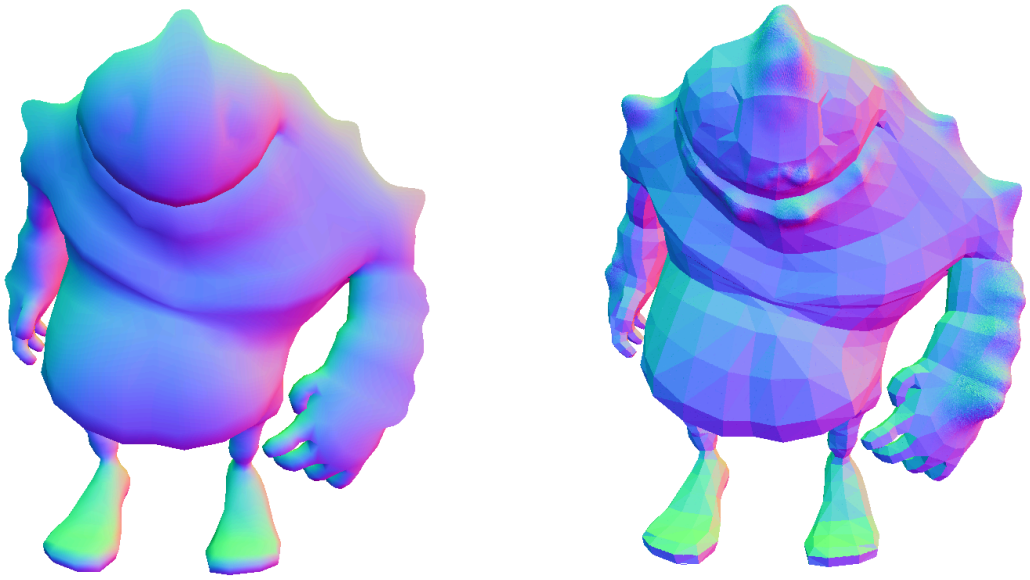

Figure 36: Big guy with input normal(left) and re-calculated normal(right) after displacement mapping

Figure 37 demonstrates the impact of different normals on the lighting of the 3D model.

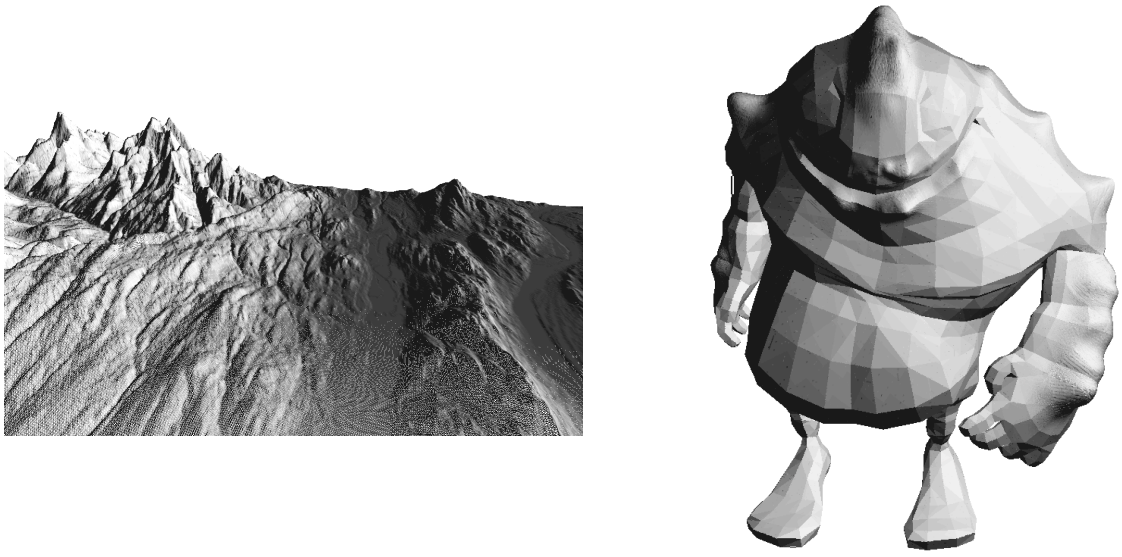

Figure 37: Lighting results with re-calculated normals

5.2 Performance Analysis

As we mentioned in the previous chapter, the compute shader pipeline also incurs some additional performance overheads. The following table and figure describes in detail the overhead incurred in our framework.

Table 3: Resources in GPU Memory

Buffer Name	Size in memory
Triangles Visibility	0.285 MB
Refine Patterns	5.47 MB
Indirect Commands	160 B
Refined Vertices	403 MB
Refined Indices	403 MB
Scene Configuration	64 B
Re-calculated Normal	403 MB
Frame Constants	176 B

It's worth noting that most of the memory here is pre-allocated in order to meet the needs of dynamically generated vertices, they might vary depending on the maximum number of triangles you ultimately wish to generate. In the following analysis, we focus solely on the actual memory used by the generated geometry, excluding the pre-allocated buffer space reserved for dynamic vertex generation.

Following figure show the GPU time of each stage in milliseconds with the tessellation factor of 50, which will generate 7,250,000 triangles from the input coarse mesh with 2900 triangles.

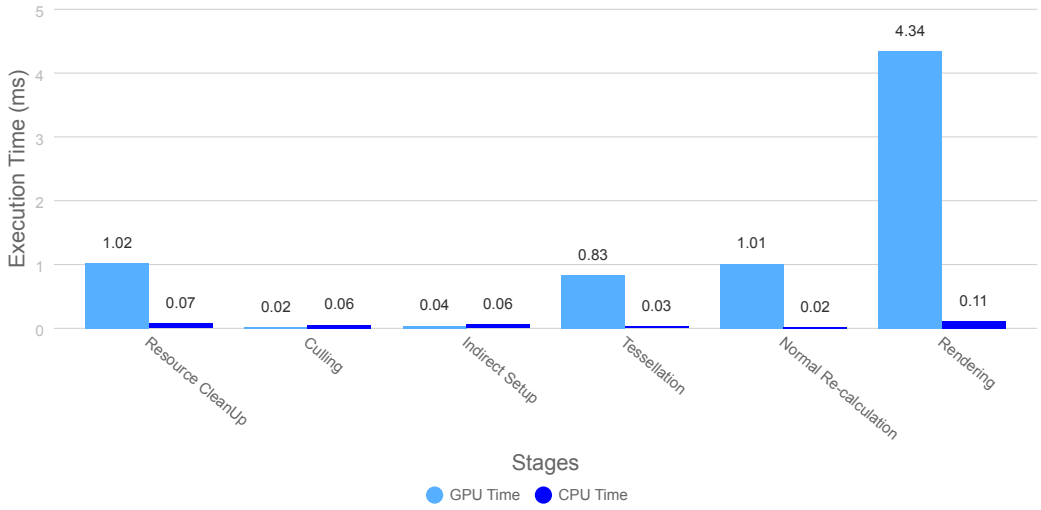

Figure 38: GPU and CPU execution time of different stages

While documenting the performance overhead of our own framework is necessary, in order to fully assess the utility of the approach, we must also compare it to several existing mainstream schemes. In the next section, we will show the performance results in turn against state-of-the-art techniques such as direct rendering of highly detailed models, Hardware Tessellation, and Nanite.

Compared to original detailed mesh

In Table 2, we present comprehensive information about the input detailed mesh data. This section we present the subsequent analysis focusing on model loading time, rendering time and memory consumption, see Figure 39. To get the same triangle amount, input coarse mesh will apply pattern with tessellation level 32 to generate $2900 * 32 * 32 = 2,969,600$ triangles.

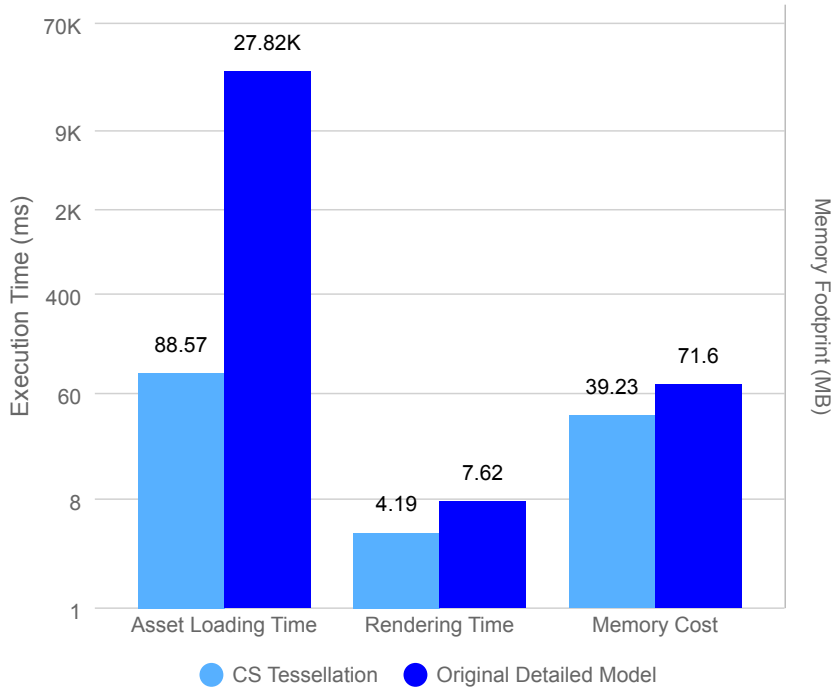

Figure 39: Comparison between Compute Shader Tessellation and the original detailed model. Execution time is shown on the left Y-axis, and memory usage on the right Y-axis.

As the generated vertices must inevitably be written back to GPU memory, the actual memory usage is not ideal. However, the model loading time is significantly reduced from 27,820 ms to 88 ms, and the rendering performance also improves by approximately 45%, with the rendering time decreasing from 7.62 ms to 4.19 ms. Thanks to the early discard of invisible triangles, we actually render only about half the number of triangles compared to the original detailed mesh while maintaining the same visual quality. Similarly, the memory consumption for the vertex and index buffers is also reduced by half, even taking displacement texture into account, the total memory usage amounts to only about 55% of the original model.

Compared to Hardware Tessellation

Since it is impossible to precisely measure the GPU time consumed by the tessellator, and unlike compute shaders which explicitly write data back to GPU memory, Hard Tessellation directly streams data to the GPU cache for subsequent processing, although it is possible to estimate memory usage depending on the amount of generated vertices and triangles, we only consider the rendering consumption as hardware implementation detail such as data compression is unknown.

Therefore, comparing memory usage in this context is unnecessary. Instead, we evaluate the methods based on their overall rendering time.

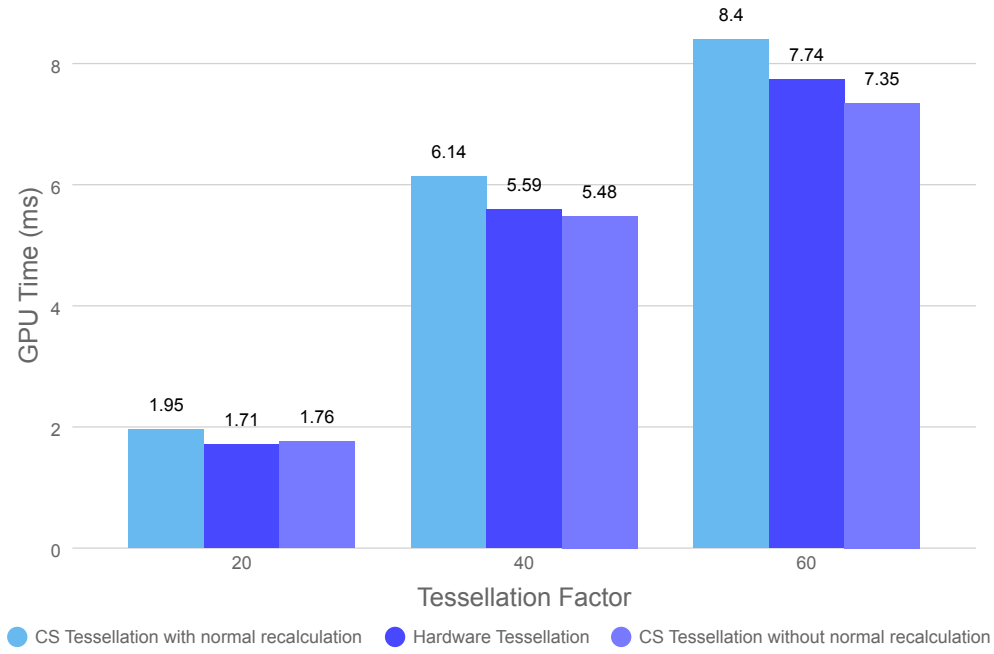

Figure 40: Comparison between compute shader tessellation with hardware tessellation in different tessellation factors

Figure 40 compares the rendering performance of the two approaches using the same tessellation factors. As we can see, the overall rendering time based on hardware tessellation is slightly more efficient than the compute-shader-based pipeline. However, since we did not re-calculate the normals in the tessellation evaluation shader, if that process is omitted, the compute shader tessellation actually performs better overall.

Chapter 6
Discussion and Future Work

This chapter discuss the challenges and issues we meet throughout the whole process. We also outline several possible solutions and related research for furture work.

6.1 Data Optimization

GPUs have relatively limited memory resources available compared to host systems. In this memory-constrained environment, any redundant data storage can negatively impact system performance.

Although it was mentioned earlier that we only store u and v in the barycentric coordinates of each vertex, they are currently each still stored as 32-bit floats, which means that they take up 8 bytes per vertex. To further consolidate their memory footprint, we can encode v as 16-bit integers respectively, combining them into a single 32-bit, i.e. 4 bytes data structure, thus halving the memory overhead per vertex. Similarly, different bit-widths can be used for the storage of triangle indices [26].

Besides the compression strategies mentioned above, AMD proposes a lossy compression format for small meshlets patches by quantizing vertex coordinates and encoding topology compactly, achieving efficient and hardware-friendly geometry compression [47]. [48] focuses on triangle strips, optimizing rendering efficiency by reordering vertex sequences and constructing the longest possible strips to reduce redundant vertex transmission. [6] improve the pattern-based mesh refinement method [5], storing only essential refinement patterns and local information, significantly reducing the storage and transmission overhead of refined meshes while enabling efficient GPU parallel processing.

6.2 Adaptive Patterns

The patterns we have used so far are uniform patterns, i.e., the same tessellation rate is applied to all three sides of the triangle. Although this approach is simple to implement and to some extent can avoid the crack problem caused by T-junction by uniform tessellation rate, it still has many limitations in practical application.

From the perspective of rendering efficiency, different regions often have different geometric complexity or viewpoint importance. Uniform tessellation is similar to the common discrete LOD method in games, which is unable to flexibly adjust the tessellation density of each region according to the difference in geometric complexity or viewpoint, thus introducing a large number of redundant vertices. Compared to adaptive patterns [26, 27] based on curvature or viewing angle, uniform patterns generate too many invalid vertices in flat areas or areas far away from the viewing angle, adding unnecessary rendering and memory overhead.

6.3 Generic Vertices Deduplication

Based on my limited research and investigation, I found that all techniques, including mega geometry [26], or slightly earlier GPU tessellation [9, 16, 25], or even hardware tessellation, they all inevitably generate duplicate vertices on the shared edge, because they are single triangle or quad as the object of tessellation, and do not consider the overall topological information. This results in an adjacent triangle suddenly becoming two separated triangles, with overlapping vertices on their shared edges, see Figure 41.

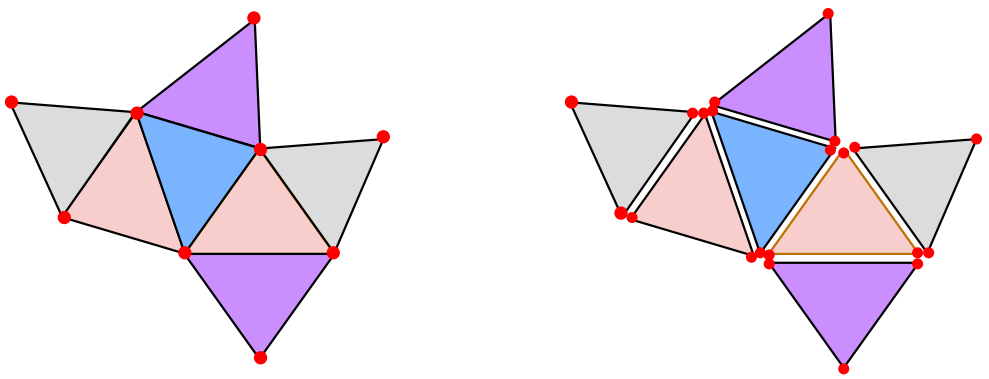

Figure 41: Adjacent triangles(left), Separated triangles(right)

That's why during the normal re-calculation stage, due to the presence of duplicated vertices, the face normals of original adjacent triangles used to contribute to the same shared vertex to ensure smooth normal interpolation, now only affects the three vertices explicitly defining its own triangle, as shown in Figure 42.

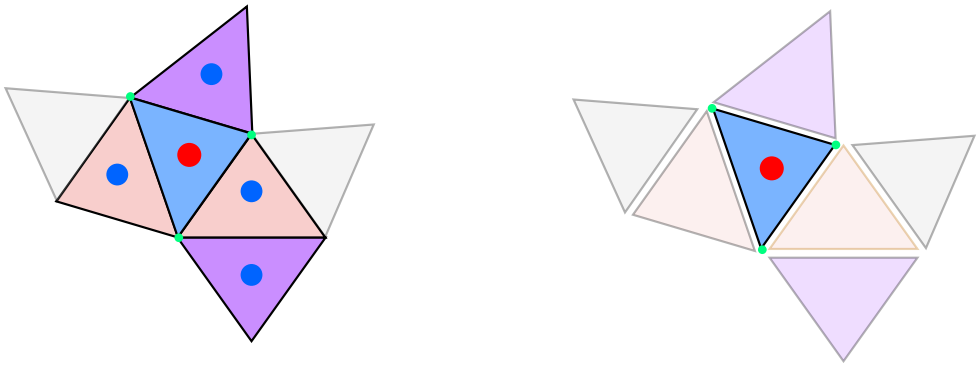

Figure 42: Triangles involved normal accumulation before tessellation(left) and after tessellation(right)

If the curvature of the two neighboring triangles is too large, then they will have a large converging normal direction, and the shared edges of the two triangles will start to fight for the right to render this edge, because they appear to be one edge, but they are actually composed of different but overlapped vertices, see Figure 43.

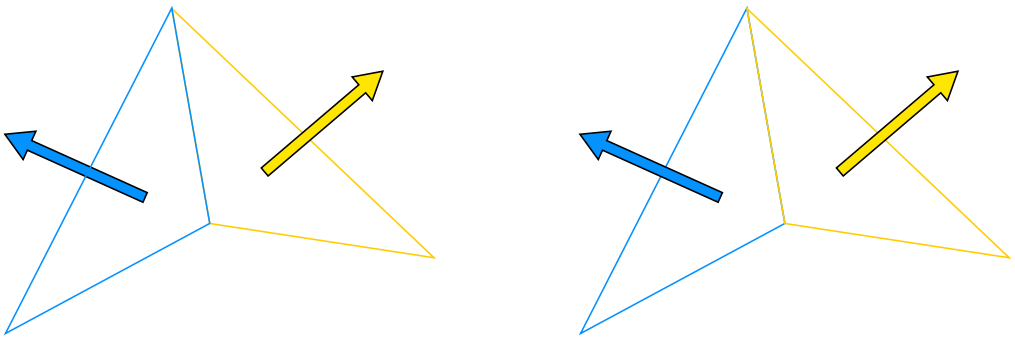

Figure 43: Conflicting normal on the shared edge where vertices' normal are assigned by the blue triangle(left) in some frame or are assigned by the yellow triangle(right) in some frame

But causing such a visual flaw is not the worst result, the GPU memory consumed by generating redundant vertices is what we need to optimize more. In order to minimize the redundant vertices generated during tessellation, there is no very generic solution in the academic world so far, and since deduplication has a highly sequential nature, it is actually more suitable for CPU execution, and removing the redundant vertices in parallel in a compute shader is a relatively difficult task.

Wald et al. [49] present a highly parallelized vertices removal algorithm using cuda was proposed, but cuda is a different language, and forcing it to manipulate the data generated by the compute shader would introduce many additional operations such as memory mapping.

As shown in Figure 44, A pattern with tessellation level ten results in approximately half of the vertices being duplicated. Although the proportion of duplicated vertices decreases

as the tessellation factor increases, the absolute number of duplicate vertices continues to grow.

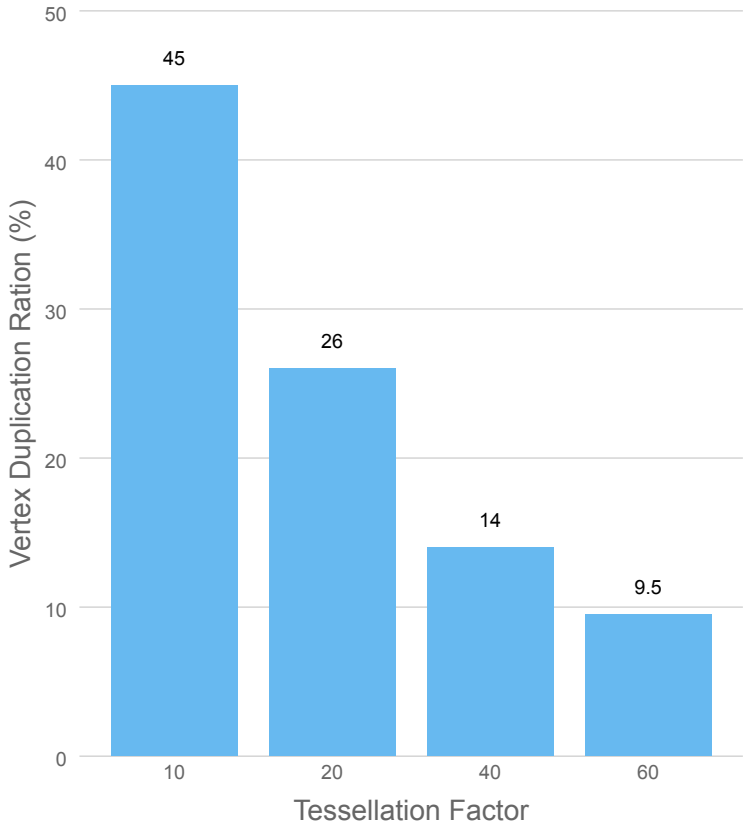

Figure 44: Proportion of duplicate vertices in each tessellation pattern

However, the duplicate vertices removal strategy will differ for different tessellation strategy. For example, if the tessellation is a wide range tessellation, say based on the whole cluster, the duplicate vertices to be removed will only exist at the edges of the cluster. As a result, providing a universal solution for duplicate vertex removal remains difficult.

6.4 Next-Gen Geometry Pipeline

Although Compute Shader-based tessellation offers greater flexibility than the traditional hardware pipeline, it introduces notable overhead. Since it can't pass vertex data directly into the graphics pipeline, the output must be written to GPU memory, causing extra I/O and memory bandwidth pressure. In contrast, hardware tessellation writes vertex data directly to the GPU cache, avoiding this step and improving overall rendering efficiency.

Mesh Shader [28] somehow allow us to avoid unnecessary overhead while maintaining flexibility and high parallel computational capability. As shown in Figure 45, mesh shader, similar to hardware tessellation, can directly pass the output data to the subsequent

rendering work, and due to the existence of task shader, it can dynamically dispatch the number of mesh shaders, thus fully utilizing the GPU's highly parallel computing capability.

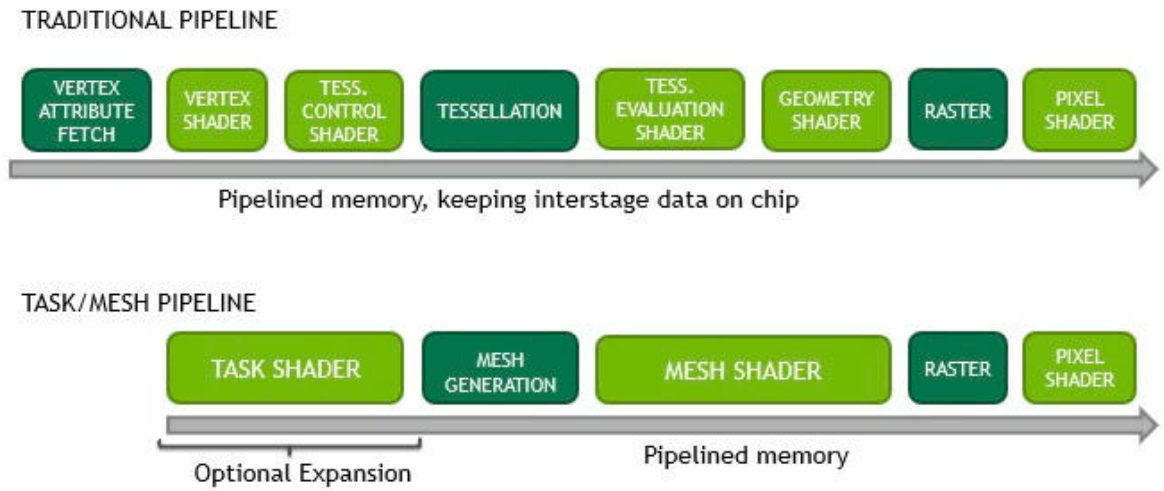

Figure 45: Traditional graphics pipeline vs. Mesh shader pipeline

In addition, the Mesh Shader Pipeline can effectively solve some of the performance bottlenecks that exist in traditional rendering pipelines. For example, in a traditional pipeline, the same vertex may be executed by multiple vertex shaders, resulting in completely unnecessary wasted computation [50]. At the same time, the fixed phases of the graphics pipeline lack sufficient flexibility and make it difficult to fully utilize the advantages of modern GPUs in massively parallel computation.

However, the output of mesh shader is limited by hardware constraints—typically up to 256 vertices and 128 primitives per workgroup [28]. As a result, the size of reusable patterns is also constrained. To handle cases where a triangle's tessellation level exceeds available patterns, we can iteratively split it until it fits within a supported pattern [26].

6.5 Visual Effect

Even when using a 4K resolution displacement texture, at very high tessellation rates each triangle can become smaller than a single pixel. In such cases, the limited bit depth used to store scalar values may not provide enough precision, leading to artifacts as shown in Figure 46. Given the limitations of traditional displacement textures, both Henry Schäfer et al. [51] and Vinod Melapudi et al. [34] propose new approaches that achieve more detailed displacement data by enabling finer control over the projection of vertices from a coarse mesh to a detailed mesh.

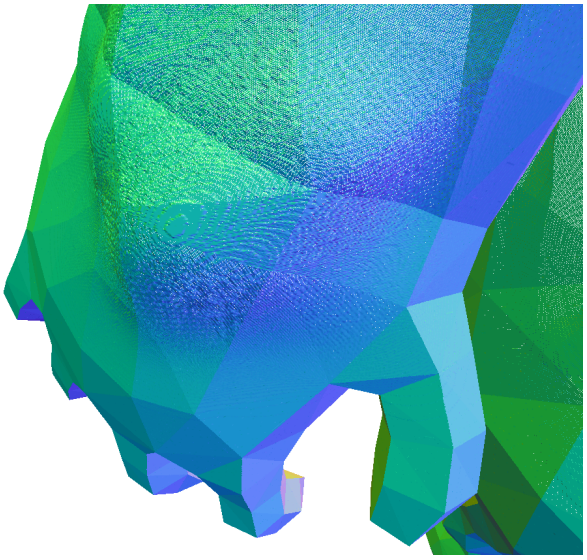

Figure 46: Contour-like artifacts resulting from excessive tessellation in displacement mapping.

As you can see from the previous results, since we only increased the density of the triangles and did not do any smoothing like the subdivision surface [30], visually, even though we interpolated and smoothed the vertex normal inside the triangles with the rasterizer, the silhouette of the model still looks sharp and angular, see Figure 47.

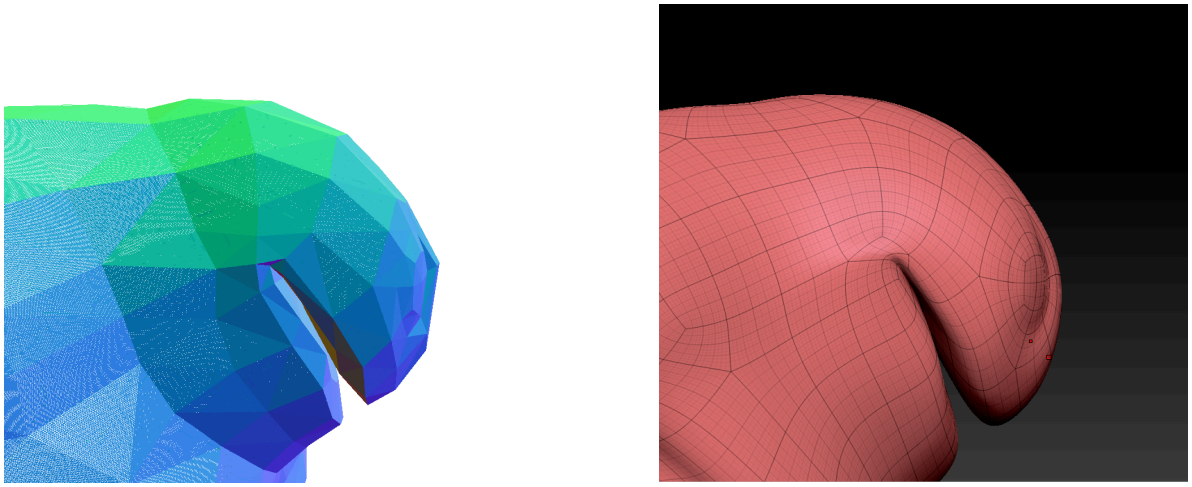

Figure 47: Side view from the same model of tessellation(left) and subdivision(right)

Common approaches to improve this are generally divided into two categories: one is directly rendering subdivision surfaces, and the other is constructing higher-order smooth surfaces based on flat triangles.

Subdivision Surface is actually the iterative application of Subdivision Schemes [3, 19, 31, 52] on top of a coarse mesh to achieve surface smoothing by continuously generating and weighting vertex positions. These approaches are particularly computationally intensive in the case of adaptive subdivision. Especially for animated mesh [53], the Subdivision Level of each frame may be different due to the change in geometric complexity over time,

which not only requires the dynamic generation of a large number of vertices, but also needs to offset the positions of existing vertices according to the weighting rule to ensure the surface smoothness. In this regard, studies such as Brainerd et al. [54] have proposed to pre-generate the Subdivision plan by using the adaptive quadtree structure, thus realizing efficient GPU rendering of the Subdivision Surface in the GPU.

Another representative method is PN Triangle [8, 35, 55], which constructs higher-order surfaces based on planar triangles to achieve smoother visual effects with less computational cost. PN Triangle interpolates vertex positions and normals to construct quadratic Bezier surfaces, which significantly improves the angularity of planar triangles and is lighter in computation compared to the full Subdivision.

It is also lighter than a full Subdivision, making it suitable for real-time rendering. In addition to PN Triangle, recent years have seen a rise in shader-based surface subdivision and fitting techniques, such as the use of Bezier Patch, Gregory Patch [56], and other higher-order surface models.

Chapter 7
Conclusion

In this section we present the summary of this thesis and answer the research questions.

7.1 Key Findings

This thesis focuses on presenting a prototype framework of how we can apply compute shader to reconstruct geometry's fidelity in realtime, especially for static mesh. Even without targeted optimization, its performance is already comparable to hardware tessellation. More importantly, its programmable and pattern-driven nature provides a level of flexibility that fixed-function pipelines lack, allowing developers to tailor the tessellation process for different use cases. Furthermore, the framework's design shows potential for handling animated geometry, where animation can be applied to a coarse base mesh while finer detail is generated at render time.

While our approach offers both performance gains and greater flexibility in engine-level, its practical application still show several important limitations and challenges. Since it is not a true subdivision surface technology, simply increasing the number of triangles cannot effectively improve the smoothness of the model surface and lacks the desired visual effect in character rendering.

In the case of static models, we found that this method is also far less efficient in terms of memory efficiency and detail reduction than the current mainstream virtualized geometry technique. The main reasons are as follows.

Model detail reduction using displacement mapping faces several challenges. On the one hand, high-precision mapping imposes significant memory pressure, which can be partially mitigated by virtual texturing, but the effect is still limited. On the other hand, obtaining accurate displacement values often requires additional processing, and highly modeled geometry often contains tens or even hundreds of millions of triangles, resulting

in extremely high-resolution textures and even multiple maps covering different parts of the character's body.

In addition, topological limitations cannot be ignored. Displacement mapping is not a one-size-fits-all approach, for instance, a sphere cannot be transformed into a circle by displacement, which requires strict control of the structural layout during the creation of the base coarse model to ensure that subsequent displacement mapping can effectively reconstruct the target shape. Finally, since the compute shader needs to write the generated vertex data back to the GPU memory, keeping the duplicate vertices generated at the common edges will not only waste memory, but also reduce the rendering efficiency, e.g. triggering a lot of unnecessary vertex shader calls. Therefore, additional de-duplication algorithms are needed.

Overall, we find that compute-shader-based tessellation presents significant potential for real-time geometry reconstruction. Its programmable flexibility and competitive performance make it a promising alternative to traditional hardware tessellation. We hope this work can inspire further exploration into leveraging compute shaders for efficient, high-fidelity rendering, particularly in scenarios involving complex surface detail and potentially animated assets.

7.2 Research Questions

With all the insights gathered throughout whole thesis work, we present our answer to the research questions mentioned in the previous chapter as follows.

1. How can compute shader be effectively utilized to implement an efficient and flexible real-time surface tessellation method?

Due to the SIMD nature of modern GPUs, we find that a pre-computed, pattern-based tessellation approach effectively leverages the parallel architecture of the GPU. With minimal memory overhead, a high volume of primitives can be generated on the fly entirely on the GPU by large numbers of compute shader threads performing simple barycentric interpolation operations in parallel. This method is highly programmable and pattern-oriented, giving developers fine-grained control over the tessellation process and enabling adaptation to a wide range of rendering scenarios.

2. What are the trade-offs between visual fidelity and rendering efficiency when using a compute shader-based tessellation method compared to alternative approaches?

For direct rendering of high-precision models, using this method has obvious advantages in terms of rendering efficiency and memory consumption. However, there is still a gap between the visual effects reconstructed by relying only on displacement mapping and the original high-precision model.

Compared with hardware-level tessellation, compute shader approach is close to the same performance in rendering. However, since hardware tessellation is a fixed feature deeply integrated into the GPU rendering pipeline, the smaller primitives generated by

hardware tessellation are able to directly stream into the GPU cache, thus avoiding the I/O consumption and extra memory overhead caused by writing back to memory, which is still an advantage in terms of efficiency and resource scheduling.

In contrast to Nanite, the latter is able to preserve high-fidelity information, including the original model's topology and normals, by dynamically streaming different segments of the mesh on demand during static mesh rendering, offering a clear advantage in visual fidelity. However, compute shader tessellation has potential performance advantages when dealing with animated models. Our approach allows us to perform animation computation only on the simplified base mesh, and then complement the details in real-time, thus significantly reducing the burden of animation computation on high dense vertices.

3. Is it feasible to achieve high-fidelity geometry reconstruction in real-time rendering using software-based tessellation combined with displacement mapping? What are the technical challenges involved in this approach?

Rather than simply answering "yes" or "no" to this question, it is more accurate to say that this combination has a clear potential for restoring high-fidelity geometry in real-time rendering. As demonstrated earlier, the framework proposed in this thesis has been able to efficiently and dynamically generate a large number of triangles on the GPU, showing good performance and flexibility.

However, there are still a number of key issues mentioned above that need to be addressed in order to fully capitalize on the benefits of this approach. These challenges include the lack of surface smoothness due to the lack of realistic subdivision, the memory and accuracy pressure from displacement mapping, topology constraints, and the additional processing required to remove redundant vertices. While these issues do not negate the feasibility of the approach, they indicate that the scheme still needs further optimization and refinement before it can be widely applied in real production environments.

References

[1] H. Hoppe, “Progressive meshes,” *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. pp. 111–120, 2023.

[2] J. H. Clark, “Hierarchical geometric models for visible surface algorithms,” *Communications of the ACM*, vol. 19, no. 10, pp. 547–554, 1976.

[3] E. Catmull and J. Clark, “Recursively generated B-spline surfaces on arbitrary topological meshes,” *Seminal graphics: pioneering efforts that shaped the field*. pp. 183–188, 1978.

[4] C. Loop, “Smooth subdivision surfaces based on triangles,” 1987.

[5] T. Boubekeur and C. Schlick, “Generic mesh refinement on GPU,” in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2005, pp. 99–104.

[6] R. Lenz, J. B. Cavalcante-Neto, and C. A. Vidal, “Optimized pattern-based adaptive mesh refinement using GPU,” in *2009 XXII Brazilian Symposium on Computer Graphics and Image Processing*, 2009, pp. 88–95.

[7] T. Boubekeur and M. Alexa, “Phong tessellation,” *ACM SIGGRAPH Asia 2008 papers*. pp. 1–5, 2008.

[8] A. Vlachos, J. Peters, C. Boyd, and J. L. Mitchell, “Curved PN triangles,” in *Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001, pp. 159–166.

[9] Microsoft Docs, “Direct3D 11 Tessellation.” Accessed: January 24, 2025. [Online]. Available at: https://learn.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-features#tessellation

[10] Khronos Group, “OpenGL 4.3 Specification - Compute Shaders.” Accessed: January 24, 2025. [Online]. Available at: https://www.khronos.org/opengl/wiki/Compute_Shader

[11] S. A. Ulrich Haar, "GPU-Driven Rendering Pipelines." Accessed: February 4, 2025. [Online]. Available at: https://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf

[12] B. Karis, R. Stubbe, and G. Wihlidal, "A Deep Dive into Nanite Virtualized Geometry." Accessed: January 23, 2025. [Online]. Available at: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

[13] T. Hák, S. Janoušková, and B. Moldan, "Sustainable Development Goals: A need for relevant indicators," *Ecological indicators*, vol. 60, pp. 565–573, 2016.

[14] M. Slater *et al.*, "The ethics of realism in virtual and augmented reality," *Frontiers in Virtual Reality*, vol. 1, p. 1, 2020.

[15] D. Zorin and P. Schröder, *Subdivision for Modeling and Animation*. 1999. Accessed: February 25, 2025. [Online]. Available at: https://multires.caltech.edu/pubs/sig99notes.pdf

[16] M. Schwarz and M. Stamminger, "Fast GPU-based adaptive tessellation with CUDA," in *Computer Graphics Forum*, 2009, pp. 365–374.

[17] H. Bowles, D. Zimmermann, G. Noris, and B. Wang, "Crest: Novel Ocean Rendering Techniques in an Open Source Framework," in *Siggraph 2017 Advances in Real-Time Rendering in Games*, 2017.

[18] P. A. Studios, "Subdivision Surfaces." Accessed: February 19, 2025. [Online]. Available at: https://graphics.pixar.com/opensubdiv/docs/subdivision_surfaces.html

[19] L. Kobbelt, "√ 3-subdivision," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, 2000, pp. 103–112.

[20] NVIDIA Corporation, "CUDA C++ Programming Guide." May 2025. Accessed: January 29, 2025. [Online]. Available at: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[21] A. Overvoorde, "Compute Shader." Accessed: January 21, 2025. [Online]. Available at: https://vulkan-tutorial.com/Compute_Shader

[22] R. L. Cook, "Shade trees," in *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 1984, pp. 223–231.

[23] Wikipedia contributors, "Displacement mapping." Accessed: March 1, 2025. [Online]. Available at: https://it.wikipedia.org/wiki/Displacement_mapping

[24] Autodesk, "Vector Displacement Maps." Accessed: March 1, 2025. [Online]. Available at: https://download.autodesk.com/global/docs/softimage2014/en_us/userguide/index.html

[25] J. Khoury, J. Dupuy, C. Riccio, and W. Engel, "Adaptive GPU tessellation with compute shaders," *GPU Zen: Advanced Rendering Techniques*, vol. 2, pp. 3–17, 2019.

[26] NVIDIA, "NVIDIA® RTX Mega Geometry." Accessed: February 14, 2025. [Online]. Available at: https://github.com/NVIDIA-RTX/rtxmg.git

[27] T. Boubekeur and C. Schlick, "A flexible kernel for adaptive mesh refinement on GPU," in *Computer Graphics Forum*, 2008, pp. 102–113.

[28] C. Kubisch, "Introduction to Turing Mesh Shaders." Accessed: March 21, 2025. [Online]. Available at: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/

[29] A. Maggiordomo, H. Moreton, and M. Tarini, "Micro-mesh construction," *ACM Transactions on Graphics (TOG)*, vol. 42, no. 4, pp. 1–18, 2023.

[30] B. Sharp, "Subdivision surface theory," *Game Developer*, vol. 7, no. 1, pp. 34–42, 2000.

[31] D. Doo, "A subdivision algorithm for smoothing down irregularly shaped polyhederons.," *Computer Aided Design*, pp. 157–165, 1978.

[32] M. Nießner and C. Loop, "Analytic displacement mapping using hardware tessellation," *ACM Transactions on Graphics (TOG)*, vol. 32, no. 3, pp. 1–9, 2013.

[33] M. Stuchlık, "Extraction of displacements between mesh and Basemesh," 2017.

[34] V. Melapudi and P. Havaldar, "Time and Memory Efficient Displacement Map Extraction," *ACM SIGGRAPH 2021 Talks*. pp. 1–2, 2021.

[35] M. Schwarz, M. Staginski, and M. Stamminger, "GPU-based rendering of PN triangle meshes with adaptive tessellation," in *Proceedings of Vision, Modeling, and Visualization*, 2006, pp. 161–168.

[36] J. Dupuy, "Concurrent Binary Trees (with application to longest edge bisection)," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, pp. 1–20, 2020.

[37] Microsoft Corporation, "Direct3D 11 Graphics." [Online]. Available at: https://learn.microsoft.com/en-us/windows/win32/direct3d11/atoc-dx-graphics-direct3d-11

[38] NVIDIA Corporation, "Micro-Mesh Basics." Accessed: January 23, 2025. [Online]. Available at: https://developer.download.nvidia.com/ProGraphics/nvpro-samples/slides/Micro-Mesh_Basics.pdf

[39] C. Kubisch, "Micro-Mesh Rasterization." Accessed: January 25, 2025. [Online]. Available at: https://developer.download.nvidia.com/ProGraphics/nvpro-samples/slides/Micro-Mesh_Rasterization.pdf

[40] NVIDIA Corporation, "Micro-Mesh Asset Pipeline." Accessed: January 25, 2025. [Online]. Available at: https://developer.download.nvidia.com/ProGraphics/nvpro-samples/slides/Micro-Mesh_Asset_Pipeline.pdf

[41] NVIDIA Corporation, "NVIDIA Ada GPU Architecture." Accessed: February 19, 2025. [Online]. Available at: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[42] NVIDIA Corporation, "nvpro_core." Accessed: February 27, 2025. [Online]. Available at: https://github.com/nvpro-samples/nvpro_core

[43] Takahiro Harada, "tinyobjloader." Accessed: March 7, 2025. [Online]. Available at: https://github.com/tinyobjloader/tinyobjloader

[44] S. Barrett, "Single-File Public Domain Libraries for C/C++." Accessed: March 13, 2025. [Online]. Available at: https://github.com/nothings/stb

[45] E. Chadwick, "More with Less: KTX2 for Creatives." Accessed: March 18, 2025. [Online]. Available at: https://www.khronos.org/assets/uploads/developers/presentations/3D_Formats_Wayfair_KTX2_SIGGRAPH_Aug21.pdf

[46] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[47] J. Barczak, C. Benthin, and D. McAllister, "Dgf: A dense, hardware-friendly geometry format for lossily compressing meshlets with arbitrary topologies," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 7, no. 3, pp. 1–17, 2024.

[48] F. Evans, S. Skiena, and A. Varshney, "Optimizing triangle strips for fast rendering," in *Proceedings of Seventh Annual IEEE Visualization'96*, 1996, pp. 319–326.

[49] I. Wald, "GPGPU-parallel re-indexing of triangle meshes with duplicate-vertex and unused-vertex removal," *arXiv preprint arXiv:2109.09812*, 2021.

[50] M. Oberberger, B. Kuth, and Q. Meyer, "From Vertex Shader to Mesh Shader." [Online]. Available at: https://gpuopen.com/learn/mesh_shaders/mesh_shaders-from_vertex_shader_to_mesh_shader/

[51] H. Schäfer, M. Prus, Q. Meyer, J. Süßmuth, and M. Stamminger, "Multiresolution attributes for hardware tessellated objects," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 9, pp. 1488–1498, 2013.

[52] M. Research, "Smooth Subdivision Surfaces Based on Triangles," 2016. Accessed: March 28, 2025. [Online]. Available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/thesis-10.pdf

[53] T. DeRose, M. Kass, and T. Truong, "Subdivision surfaces in character animation," *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. pp. 801–810, 2023.

[54] W. Brainerd, T. Foley, M. Kraemer, H. Moreton, and M. Nießner, "Efficient GPU rendering of subdivision surfaces using adaptive quadtrees," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–12, 2016.

[55] T. Boubekeur, P. Reuter, and C. Schlick, "Scalar tagged PN triangles," in *EUROGRAPHICS Short Papers*, 2005.

[56] C. Loop, S. Schaefer, T. Ni, and I. Castano, "Approximating subdivision surfaces with Gregory patches for hardware tessellation," *ACM SIGGRAPH Asia 2009 papers*. pp. 1–9, 2009.

Appendices