Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

GPU-Driven LOD Rendering in Vulkan

Bc. Richard Smělý

Supervisor: Ing. Petr Felkel Ph.D.
Field of study: Computer Graphics
May 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Smělý Richard**	Personal ID number:	**484903**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Graphics**		

II. Master’s thesis details

Master’s thesis title in English:

GPU-Driven LOD Rendering in Vulkan

Master’s thesis title in Czech:

Vykreslování LOD řízené GPU ve Vulkanu

Name and workplace of master’s thesis supervisor:

Ing. Petr Felkel, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **12.02.2025** Deadline for master's thesis submission: **23.05.2025**

Assignment valid until: **20.09.2026**

Head of department’s signature

prof. Mgr. Petr Páta, Ph.D.
Vice-dean´s signature on behalf of the Dean

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work.
The student must produce his thesis without the assistance of others, with the exception of provided consultations.
Within the master’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt Student’s signature

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Smělý Richard**	Personal ID number:	**484903**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Graphics and Interaction**		
Study program:	**Open Informatics**		
Specialisation:	**Computer Graphics**		

II. Master’s thesis details

Master’s thesis title in English:

GPU-Driven LOD Rendering in Vulkan

Master’s thesis title in Czech:

Vykreslování LOD řízené GPU ve Vulkanu

Guidelines:

Design and implement a Level of Detail (LOD) rendering system using Vulkan. The system will optimize the handling of large-scale geometry by utilizing GPU-driven techniques for LOD selection, virtualized mesh streaming, and culling.
Focus on developing a hierarchical LOD structure with virtualized geometry streaming while ensuring configurability and real-time navigation around the scene.
Preprocess polygonal 3D meshes into a hierarchical LOD structure. At runtime select the correct LOD for all parts of the mesh based on a view dependent error metric.
Compare the proposed approach with traditional LOD techniques. Benchmark the performance based on frame time, memory usage, visual impact of using LODs, and scalability across five scenes of different complexity.

Bibliography / sources:

[1] Federico Ponchio. “Multiresolution structures for interactive visualization of very large 3D datasets”. PhD thesis. Clausthal University of Technology, 2009. ISBN: 9783940394781.
[2] Yingwei Ge et al. “A Novel LOD Rendering Method With Multilevel Structure-Keeping Mesh Simplification and Fast Texture Alignment for Realistic 3-D Models”. In: IEEE Transactions on Geoscience and Remote Sensing 62 (Sept. 2024), p. 5640519. DOI: 10.1109/TGRS.2024.3457796.
[3] Paolo Cignoni et al. “Adaptive TetraPuzzles: Eficient out-of-core construction and visualization of gigantic multiresolution polygonal models”. In: ACM Trans. Graph. 23 (Aug. 2004), pp. 796–803. doi: 10.1145/1015706.1015802.
[4] Michael Lujan et al. “Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server”. In:2019 International Conference on Computing, Networking and Communications (ICNC). IEEE, Feb. 2019, pp. 777–781. doi: 10.1109/ICCNC.2019.8685588

Acknowledgements

I would like to thank my advisor, Ing. Petr Felkel Ph.D., for his guidance and support throughout this thesis. I am also grateful to the Czech Technical University in Prague and its teachers for sharing their knowledge and expertise with me. Special thanks also belongs to my family and friends for their encouragement and understanding during this work.

Declaration

I hereby declare that this thesis is the result of my own independent work and investigation, except where otherwise stated. All sources of information and data have been appropriately acknowledged. This thesis has not been submitted, either in whole or in part, for any other degree or qualification at this or any other institution.

Abstract

This text examines efficient rasterization of large-scale 3D models in real time with the main focus being on levels of detail (LOD). After a brief summary of existing techniques, the work contains a design and implementation of a 3D renderer for polygonal meshes that uses a hierarchical LOD structure. The renderer preprocesses an input OBJ mesh into a graph of meshlets. Using modern GPU features such as task and mesh shaders, the renderer enables real-time navigation and view-dependent LOD selection. The renderer is then evaluated on 20 meshes of various complexity and the results show that frame time is approximately halved on average without causing any major visual artifacts. Further investigation suggests that with a better simplification and meshlet grouping algorithm, the performance increase could be even greater.

Keywords: 3D, rendering, lod, Vulkan, meshlet, view-dependent

Supervisor: Ing. Petr Felkel Ph.D.
Praha,
Karlovo náměstí 293,
120 00 Nové Město

Abstrakt

Tento text se zabývá efektivní rasterizací rozsáhlých 3D modelů v reálném čase, přičemž hlavní pozornost je věnována úrovním detailů (LOD). Po stručném shrnutí existujících technik práce obsahuje návrh a implementaci 3D rendereru pro polygonální sítě, který využívá hierarchickou strukturu LOD. Renderer předzpracovává vstupní OBJ model do grafu meshletů. Pomocí moderních funkcí GPU, jako jsou task a mesh shadery, umožňuje renderer navigaci v reálném čase a výběr LOD v závislosti na pozici kamery. Renderer je poté vyhodnocen na 20 modelech a výsledky ukazují, že se doba na jeden snímek v průměru zkracuje přibližně o polovinu, aniž by docházelo k výraznějším vizuálním artefaktům. Další zkoumání ukazuje, že s lepším decimačním a algoritmem seskupování meshletů by mohl být nárůst výkonu ještě větší.

Klíčová slova: 3D, vykreslování, lod, Vulkan, meshlet, pohledově závislé

Překlad názvu: Vykreslování LOD řízené GPU ve Vulkanu

Contents

Figures

Tables

Introduction

Rendering large-scale models is essential in fields ranging from engineering and architecture to entertainment and scientific research. As datasets grow in complexity and detail, the demand for efficient, high-quality rendering techniques has become more important than ever. Large models bring with them the promise of unprecedented realism and accuracy, yet they also pose significant challenges in computation, memory management, and rendering performance [13][20].

This work will focus primarily on techniques and implementation of a 3D renderer for polygonal meshes. Several techniques for increasing the performance of 3D scenes will be explained. In particular, algorithms for rendering *levels of detail* (*LOD*) will be explored, and a deeper dive will be done for adaptive mesh level of detail generation and selection, which aims to enable rendering of arbitrarily sized models without compromising efficiency or quality.

The work will contain a design and an implementation of a 3D rendering engine which is able to preprocess a 3D triangle mesh provided as an OBJ file into a hierarchical representation in the form of submeshes – meshlets. The design of the Application will be centered around using new capabilities of modern GPUs in the form of *task* and *mesh* shaders and the low-level GPU API *Vulkan*. During runtime, the Application will be able to display this hierarchy and the user will be able to navigate around the 3D scene in real time. Based on a view-dependent error metric, the Application will switch between different levels of detail of the meshlets. The Application will also provide multiple configuration options which will be available either from a configuration file, command line, or directly through a user interface.

The performance of the Application will then be evaluated through rigorous measurements over several different meshes. The performance of different configuration options will be compared and reported. The measurements will be visualized using plotted graphs and discussed. Possible future improvements to the algorithm, its usage, and the Application in general will be outlined.

Chapter 1
Visualization of large models

New techniques like photo-realistic model acquisition, computer simulation, or CAD software increased the size of rendered datasets multiple times. To render these big datasets, we need to use more advanced algorithms, because a naïve triangle mesh rendering is infeasible for these large models. Although these algorithms can have different approaches and ideas, the main traits can still be divided into a few categories [38].

Algorithms can perform *filtering*, which in some way limits the data that needs to be processed to only the needed subset. The performance advantage of leaving out some data is obvious, but care must be taken not to leave out too much data, so the final rendered image does not have unwanted artifacts. The exact specifications of what is unwanted and what is acceptable depend on the use case.

A wide range of algorithms deals with *data management*. For any GPU rendering, there must be some way to transfer data from the CPU to the GPU. Some algorithms can utilize data locality to utilize the CPU or GPU cache. If the data exceeds the size of RAM, some dynamic loading and unloading from disk is needed. For even larger datasets, there is also the option of loading data over the network.

After the data is loaded and available come the *rendering* algorithms, which can utilize modern computer hardware (most notably the GPU) to render this data efficiently. Renderers often need to balance the trade-off between performance and visual quality.

Depending on the complexity of the algorithm, the algorithm can encompass one or more of the traits mentioned above. The following chapters will explore some of these algorithms.

1.1 View-dependent filtering

The main idea of *view-dependent* filtering is that anything that is not seen in the 3D scene by the camera at render time can be discarded.

The most basic but still effective approach to filtering is discarding geometry, which cannot possibly be visible in the 3D scene due to being out of the viewing frustum – *frustum culling*, or due to having a polygon normal facing away from the camera – *backface culling*.

In a scene with multiple objects, there can also be employed *occlusion culling* which discards objects that are behind other objects (when viewed from the camera). However, the overhead and algorithm complexity for occlusion culling are noticeably higher when compared to frustum or backface culling. An illustration of these three culling methods can be seen in Figure 1.1.

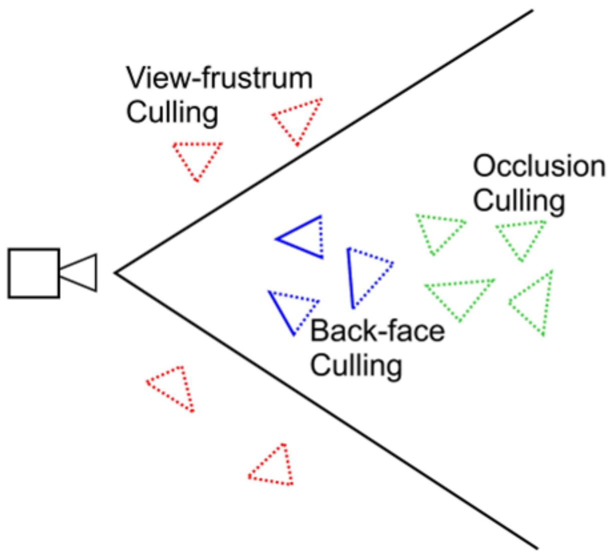


Figure 1.1: Example of different culling methods [14]

Another way to reduce the number of polygons that must be rendered is to use *levels of detail* (or commonly referred to as *LOD* or *LoD*). This paper focuses mainly on this approach of increasing 3D rendering performance, which is why levels of detail is a standalone chapter that will explain this technique in more detail.

Chapter 2
Levels of detail

The technique called levels of detail[1] is mainly a filtering method with some overlap to data management. The LOD filtering method decimates the mesh into differently coarse mesh representations. Ideally, the rendered mesh should contain only enough detail that every rendered pixel is the same as if the original – most detailed – mesh had been used. In reality, there is often a direct trade-off between increased performance and visual quality.

Typically, there are two distinct tasks for the LOD system:

1. Generate the different LODs
2. Choose which LOD to render

Generation of the different LODs is usually done as a precomputation step and can often even be saved to disk as a form of "cache". The choice of which LOD to render is made during rendering based on some criteria set by the algorithm used. Although advantages of a fast precomputation step exist, in this paper only the runtime performance of the algorithm will be the main focus.

When choosing which algorithm to use for levels of detail, one must know how it deals with the following common problems of LODs.

- Too coarse mesh representation resulting in visual artifacts
- *Visual popping* when switching between different levels
- Visible *cracks* in the model
- High memory usage

2.1 Discrete level of detail

The simplest method to create a LOD structure is to generate a discrete set of decreasing resolution models at a predefined accuracy [11][17]. The

[1] Also sometimes called *multi-resolution*.

application then selects, based on a criterion (usually distance from camera or screen space of the model bounding box), the best level for current use. The selection is quick and simple. But when switching to another level of detail due to camera or object crossing the selection criterion, the whole model changes, which often causes visible popping. An example of multiple discrete levels of detail can be seen in Figure 2.1.

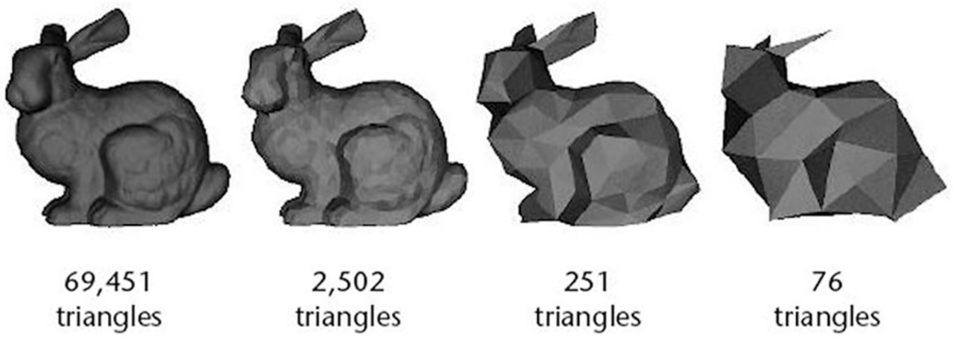


Figure 2.1: Example of different discrete levels of detail [46]

Memory usage can be managed by dynamically loading only the closest levels of detail to the currently selected with the assumption the selection criterion will change continuously without jumps (for example, the camera gets closer to the object). The property of memory manageability is important for rendering large datasets, and the idea of keeping only the closest to currently used levels of detail is common for other LOD algorithms as well.

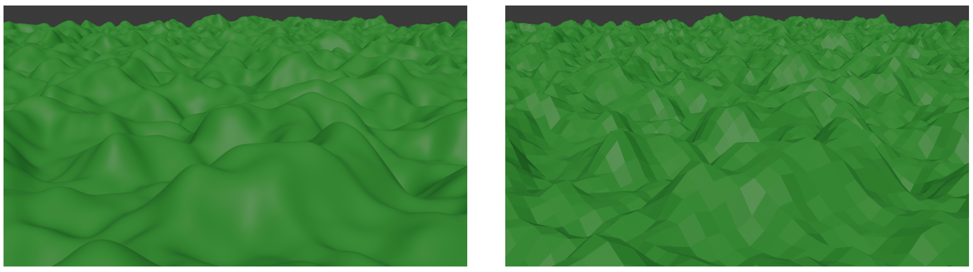
2.2 Discrete level of detail with smooth transitions

The problem of visual popping can be solved by interpolating the two closest LODs. The interpolation can be done either at the geometry level (before rasterization) or pixel level (after rasterization). Pixel interpolation is just a simple weighted alpha blending of the closest two LODs [18], but that means essentially rendering the model twice[2]. Geometry interpolation is a bit more complex as the model needs to store a mapping of vertices of the lower resolution model onto the higher resolution model [23].

For rendering large models, this discretization could be used, but for some models, as seen in Figure 2.2 the mesh representation will be too detailed or too coarse by definition, as the decision of what LOD to use is made for the whole model. Some parts of the mesh may be significantly closer to the camera than others, so the renderer either wastes performance rendering the far parts of the mesh in much more detail than needed, or the closer parts of

[2]Which may not be as bad as it sounds, because only the simplified meshes are rendered for less important objects based on criterion.

the model lack detail.



(a) : 2.5 million vertices **(b) :** 40 thousand vertices

Figure 2.2: Comparison of different landscape LOD levels

2.3 Nested models

The *nested models* approach is based on recursive subdivision and simplification of the initial mesh. This enables the renderer to decide what parts of a mesh to render in coarser detail and what parts to render in finer detail so, in theory, the renderer keeps its performance and quality regardless of mesh size and orientation.

However, when using this tree-like idea, another problem arises. When the renderer chooses different LODs adjacent to each other, the boundary can have "cracks" due to the simplification process – which does not preserve boundaries by default. This can be solved in three ways [38].

The most straightforward solution is *marking* the boundary vertices as read-only. Marking has one big disadvantage of creating dense geometry near the marked edges, as the simplification algorithm can never touch the boundary.

Cracks can also be fixed in a post-processing step by *stitches* which connects the cracks to form a connected mesh again. Storing this data for every combination of neighboring LODs is infeasible. Stitches can also be computed during rendering, but that incurs significant performance overhead.

Geomorphing interpolates vertex attributes between different levels, so similarly to the "Discrete level of detail with smooth transitions" there needs to be additional complexity to map vertices between levels.

2.3.1 Batched multi-triangulation

Processing whole subsets of a mesh instead of individual triangles can be beneficial from the data transfer and processing point of view. The loss of fine control is almost always outweighed by the performance gains of using CPU and GPU cache. So [38] proposed *batched multi-triangulation* – a solution to the edge cracks by *temporary* grouping the meshlets[3] and marking only the group edges as read-only. The edges are marked only for one (or in general several, but limited) level of the simplification process. After simplifying the grouped meshlets in one level, the marked edges are unmarked, different meshlets are grouped (which are now simplified), their group edges are marked, and simplification continues. You can see this illustrated in Figure 2.3 — marked edges are red.

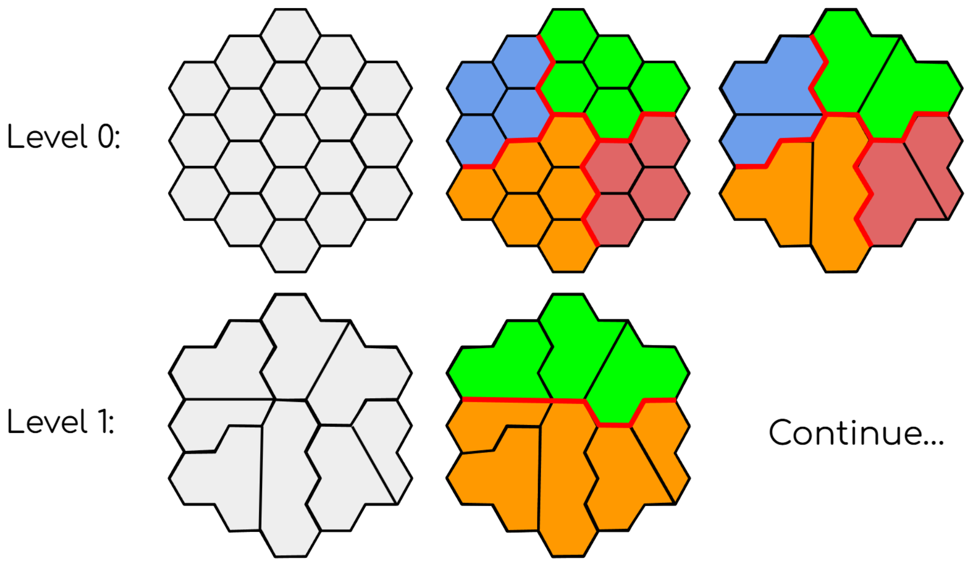


Figure 2.3: Illustration of marking different edges between levels

The progressive moving of the marked edges creates dependencies so that one child may depend on multiple parents, so a directed acyclic graph is created instead of a tree. This approach combines the possibility of displaying different resolutions on different parts of the model and, at the same time, there should be no visible cracks on the borders of the patches.

Even though the preprocessing is a bit more complicated and performance-heavy than for the other approaches, the runtime overhead of choosing the right nodes in the DAG should be manageable in real-time if implemented correctly. If the patches are small enough and the error criterion is so strict that LOD changes are subpixel large, there would also be no visual popping. Inspiration from this algorithm idea by [38] is apparent in Unreal Engine [28].

2.4 Impostors

The basic idea of *impostors* is to render some of the objects to texture buffers and instead of rasterizing the actual 3D object an *impostor* (the texture) is displayed over a simple quad mesh. These impostors can be pre-computed for

[3]Meshlets were called *patches* in the original text by [38].

different angles and then shown as billboards. Many precomputed impostor angles can be concerning memory-wise, while too few may limit the usefulness or cause visual artifacts in the rendered image. Impostors can also be rendered dynamically and changed upon the camera passing some kind of threshold angle value. This technique is often used in games for objects that are far away [12].

The impostor approach has two drawbacks. It can be used only for distant objects because when the camera is closer to the object, even slight angle changes would force rerendering of the whole impostor. Another drawback is that, even with this technique, the model has to be rendered from "full" resolution into the impostor texture.

Both of these drawbacks can be counteracted by combining other previously mentioned LOD ideas with impostors based on some kind of threshold to switch from rendering a lower resolution LOD to just an impostor[4].

2.5 Levels of detail summary

In summary, each of the aforementioned techniques has drawbacks, but offers advantages. The succinct comparison can be seen in Table 2.1. The Application in this work will implement the approach of *nested models – batched multi-triangulation*.

Technique	Strengths	Weaknesses
Discrete level of detail	simplicity	visual popping
DLOD with smooth transitions	no popping	quality/performance trade-off
Nested models – Batched multi-triangulation	no popping, no quality/performance trade-off	complex preprocessing and runtime handling
Impostors (dynamic)	good performance, low memory requirements	only suitable for far away objects

Table 2.1: Summary of LOD techniques

2.6 Mesh simplification

In the previous section during the description of the techniques used for levels of detail, the simplification of the mesh (either as a whole or as submeshes)

[4]Threshold can be any metric, but screen size or distance from camera is most common.

is skipped and treated as a black-box. In this section, the black box will be opened and explored.

The mesh simplification algorithm receives a dense/high-resolution 3D mesh as its input. Then the goal is to generate some kind of coarser/lower-resolution representation that is faster to process and render[5]. For some applications that use mesh simplification, it is also beneficial when the chosen mesh simplification algorithm reports either a computed or at least an approximated error value of the simplification – the amount of "difference" between the original mesh and the simplified mesh. An example of mesh simplification can be seen in Figure 2.4.

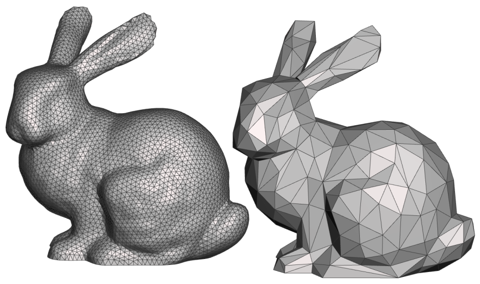


Figure 2.4: Example of mesh simplification [41]

The mesh simplification algorithms generally only account for vertex positions, but some of them can be extended to also account for vertex normals and/or texture coordinates. Simplification strategies may be broadly grouped into two categories: local strategies that iteratively simplify the mesh by the repeated application of some local operator and global strategies that are applied to the input mesh as a whole. Local strategies are by far the most common [45].

The most commonly used algorithms are explained in more detail in the following text.

2.6.1 Local vertex decimation

The algorithm known as *vertex decimation* is working locally on individual vertices. It classifies each vertex whether it can be deleted. Then based on some criterion – in the original paper [42] it was distance to a plane created by averaging all normals that the vertex touches – the vertex is deleted. The resulting hole from the vertex deletion is then re-triangulated and the algorithm continues until some stopping condition is reached. Illustration of one step of the vertex decimation algorithm is shown in Figure 2.5.

[5]There exist algorithms for other kinds of mesh simplification, but they are not relevant to this work.

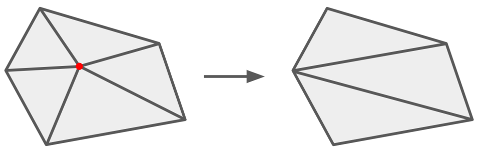


Figure 2.5: Illustration of one step of vertex decimation

2.6.2 Local edge contraction

One of the most used algorithms for mesh simplification falls under the category of *edge contraction* [45] – also referred to as *edge collapse*.

The algorithm also operates locally. In each step (as the name suggests) it finds an edge and collapses it into a single vertex. An illustration of one step of edge contraction can be seen in Figure 2.6 [24].

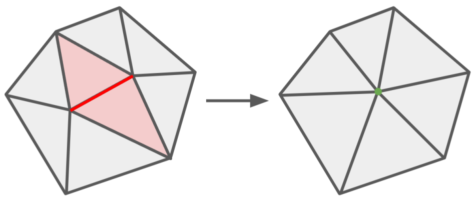


Figure 2.6: Illustration of one step of edge contraction

For a given edge collapse, it is not immediately clear where the resulting vertex should be placed. Obvious choices such as the position of the start, end of the edge, or their average are convenient but can easily be shown to be non-optimal. Rather than arbitrarily placing the resulting vertex, it is sensible instead to consider an error function associated with the contraction operation and attempt to minimize its value in the space of possible vertex placements [45].

In [19], the use of *Qaudric Error Metric* was proposed. The algorithm assigns a symmetric 4×4 matrix Q to each vertex, the error function $\Delta(v)$ of the *vertex* $v = [v_x, v_y, v_z, 1]^T$ is then computed as $\Delta(v) = v^T Q v$. For computing the error of an *edge contraction* $(v_1, v_2) \to \bar{v}$ the error matrix is then simply computed as $Q_{\bar{v}} = Q_{v_1} + Q_{v_2}$. The error function $\Delta(\bar{v})$ is then minimized to find the best position for the contracted vertex. If this minimization fails (for example, due to the matrix not being invertible), the fallback can choose the lowest error of the aforementioned positions v_1, v_2, or $(v_1+v_2)/2$. After the removal of an edge, the error matrices of the neighboring vertices are updated, and the algorithm continues. The mesh simplification made by the Meshoptimizer library is based on this algorithm [27].

2.6.3 Global simplification strategies

There are also algorithms that consider the model as a whole and perform simplifications based on global error metrics. For the Application implemented in this work, the global aspect is a problem, because using the *batched* multi-triangulation inherently means that the simplification must be localized to the submesh. As the global simplification strategies are not suitable, they will be omitted from this overview.

Chapter 3
Application

This chapter contains the design and implementation of an Application[1], which uses some of the techniques from previous chapters to enable viewing and real-time exploration of a 3D polygonal scene.

3.1 Requirements

The Application should have some necessary features and a level of performance. The following are the formal requirements for the implemented software. For the *functional requirements* the software must:

- dynamically select the appropriate resolution of a model based on the error metric,
- support automatic generation and management of different levels of detail (LOD) for 3D polygonal models,
- update the displayed model resolution in real-time during user interactions such as zooming, panning, and rotating,
- be able to import models in standard OBJ format and preprocess them for multi-resolution rendering,
- handle dynamically changing scenes where objects may move,
- include user-configurable settings to visualize the LODs.

And for the *non-functional requirements* the software also must:

- be capable of handling scenes with up to 1 million polygons without significant degradation in performance,
- run on Windows and Linux devices,

[1]Referred to in this text as *Application* with capital A.

- have a codebase that should follow modular design principles to facilitate updates and addition of features[2],
- ensure minimal perceptible loss in visual quality when transitioning between different resolutions,
- provide an intuitive interface for users to interact with and configure rendering settings.

3.2 Chosen tools and dependencies

The Application was developed using multiple technologies. The goal was to limit the number of external dependencies to a reasonable amount in order to simplify the codebase, so that it can serve as a universal example. Using fewer dependencies also increases the chance that the user will be able to compile the Application without too many additional steps. The dependencies the Application actually uses are abstracted into interfaces as much as possible to enable swapping a potentially obsolete dependency for another one.

For this reason, an *event system*, a rudimentary *OBJ file loader*, a *JSON file writer*, a *configuration handler* and a *binary serializer* were self-implemented. The choice of technologies and external dependencies is discussed in the following.

3.2.1 Engine programming language

The choice of the programming language most often comes down to the preference of the developer, and there is no clear-cut best option. For real-time rendering, compiled languages are preferred, as they usually provide superior performance.

The Application will be written mostly in C++ as this language provides the performance needed for real-time rendering. The widespread usage of C++ in computer graphics development, mature documentation, type safety, and a robust standard library are other advantages of this language which simplify development.

It could be argued that C++ is quite complex, especially in comparison with C, for example. The main disadvantage of C is the lack of a standard library and less type safety.

One other candidate for a rendering engine is the language Rust, which is very similar to C++. The disadvantage of Rust is that it is a relatively new language, so some features might not be as well documented and/or polished.

[2]Mostly meaning extensibility with different windowing and GPU APIs other than GLFW and Vulkan.

3.2.2 GPU API

In GPU APIs, the options are a bit limited. This work is intended to be as “open” and widely available as possible, so in this context, the APIs that are bound to a single platform (for example, DirectX and Metal) are omitted.

If only cross-platform APIs are considered, only three viable candidates remain: *OpenGL* [29], *Vulkan* [30], and *WebGPU* [49].

OpenGL was historically the usual choice, but as [34] states, there are several shortcomings in performance predictability and energy efficiency compared to Vulkan.

Vulkan is a more modern alternative to OpenGL, which offers very fine control over the graphics hardware resources, so the developer can utilize the computing power of the GPU more fully. This comes at the cost of a more complex implementation that has to take care of basically everything in the rendering process.

WebGPU is an interesting alternative that is designed to run in the browser. This has the advantage of being truly universal and multi-platform. This comes at the cost of bigger runtime overhead, so the non-browser-based APIs are more performant. As this work focuses on the performance and real-time usability of the rendering algorithms, it could be an interesting comparison between running functionally identical code in the browser with WebGPU and locally with Vulkan. One thing to take into consideration is that for WebGPU running in the browser, the serving of the large models can in some cases be non-trivial. Also, at the time of writing, the WebGPU standard is behind in development and does not offer the usage of task and mesh shaders.

From this comparison, Vulkan was chosen as the main GPU API. Specifically, the implementation compiles with VulkanSDK version 1.3.296. But all the code that involves Vulkan is as encapsulated as possible, so a later switch to or an addition of another GPU API would not be a problem. For example, specializations could be made for Windows (by using DirectX) or Apple devices (by using Metal).

3.2.3 Windowing API

Two of the most commonly used multiplatform windowing APIs at the time of writing are *SDL* [43] and GLFW [21]. SDL has more features involving, for example, managing audio and game controllers. On the other hand, GLFW is very well documented. As the newly created Application does not have much need for the extra features that SDL provides, GLFW was chosen as the windowing API used. Also, no advanced functionalities are to be expected – the Application really just needs a cross-platform way to expose a *surface* to the Vulkan API and basic keyboard and mouse input.

3.2.4 Other dependencies

For simplifying math operations with 3D vectors and transformation matrices, the library *GLM* [22] was used.

Another dependency used is for the user interface. During runtime, the Application needs a user interface for changing various configuration values and displaying statistics (most notably the frames per second). There are many frameworks that can satisfy this rather simple use-case. For the Application, the framework *Dear ImGUI* (commonly referred to as just *ImGUI*) [16] was chosen for its simplicity. The integration of this framework is also made easy by the fact that the framework repository provides headers specifically for Vulkan and GLFW.

The last dependency of the Application is a 3D mesh library *Meshoptimizer* [27] which has many features regarding 3D mesh manipulation, but the Application uses only the mesh simplification algorithm. See Section 2.6 for more details.

3.3 Application architecture

The Application updates in an infinite loop till it is explicitly shut down by the user. The Application is divided into several logical components. The main ones are the following.

Scene contains all information about models in the 3D scene, such as their position, orientation, scale, and the 3D mesh that they are represented by. Scene also contains the camera properties.

Input handles the user's interaction with keyboard and mouse and exposes *Input Events* that can be listened to by the Scene.

Renderer takes the Scene structure and renders it every frame from the camera perspective (with 3D perspective projection). Currently, the Renderer is using a Vulkan implementation called *VulkanHandler*, but the Renderer is an API-agnostic interface that could be implemented by any other GPU API.

Platform Framework is also an interface-like component which abstracts away the GLFW API (so it could be swapped with SDL, for example) and provides a window surface to the Renderer and hardware (keyboard and mouse) callbacks for Input.

Config makes use of the *singleton* pattern to provide configurable values to the entire application. It is split into a compiled and a loaded part. Configuration values which are either performance-critical or it does not make sense to load them (for example a boolean for debug build) are defined as `constexpr` constants. The loaded part of configuration has some sensible default values hard-coded into the Application, but they can be overridden by a simple text file named “config.cfg” which can contain any number of configuration key-value pairs. The loaded configuration can then be overridden once more by specifying command line parameters in the same way.

Mesh contains logic for preprocessing and other operations with 3D meshes. This logical component also includes the logic for the preprocessing and representation of meshlets.

3.4 Core renderer components

As already stated, the Application is using Vulkan API to access and utilize the GPU for rendering images onto the screen. When implementing the Vulkan renderer, the book Vulkan Tutorial [37] was followed for the initial Vulkan setup. Later, the implementation was heavily modified to suit the intended usage of the Application to render 3D meshes with LOD using task and mesh shaders, as the Vulkan Tutorial was only for a very basic renderer.

3.4.1 Rendering pipeline

When rendering with Vulkan, the Application can choose to use the *vertex shader pipeline*, which has the optional stages of tessellation and geometry shader before passing its output to the hardware rasterizer and then fragment shader. The vertex shader runs on a predefined vertex buffer which defines vertex positions and an index buffer which defines primitives – triangles.

Alternatively, the Application can choose the fairly new (first introduced by NVIDIA in 2018) *mesh shader pipeline* [31]. The mesh shader works more similarly to a non-graphics oriented *compute shader*. The mesh shader runs in a predefined number of workgroups. Each of the workgroup threads (or invocations) has very few basic "own" variables like its index. The majority of the processed data is loaded from uniforms and storage buffers. The mesh shader then outputs an upper bounded number of vertices and primitives to the hardware rasterizer.

Optionally, the mesh shader pipeline can also utilize a *task shader* (called *amplification shader* in DirectX). This shader runs before the mesh shader, and its layout is very similar, but it does not emit vertices and primitives. Instead, it schedules mesh shader workgroups with a very small data *payload* that can be read by the mesh shader threads. An illustration of the pipeline differences can be seen in Figure 3.1.

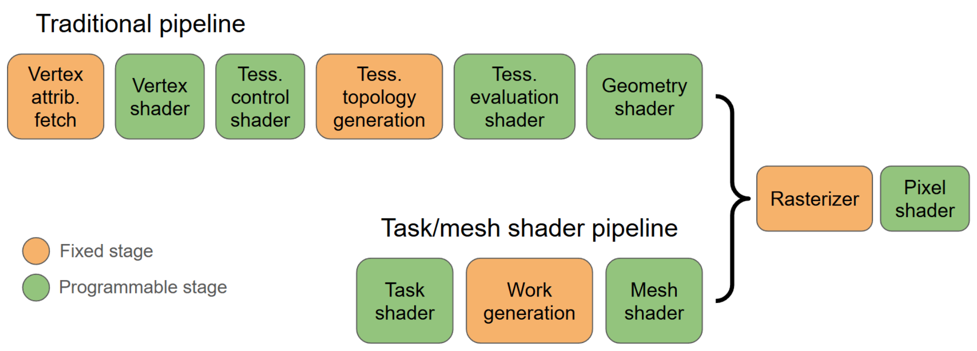


Figure 3.1: Vertex and mesh pipeline comparison [32]

For the usage of the LOD hierarchy outlined in Section 2.3.1, the mesh shader pipeline seems like a perfect fit – the mesh is divided into submeshes called *meshlets* (as seen in Figure 3.2) which are then rendered independently. Not all GPUs support the new mesh shader pipeline yet, so a traditional fallback pipeline would be necessary for those GPUs. While the Application can render meshes through the vertex shader pipeline, the rest of the text is focused on the mesh shader pipeline, as that is the pipeline that uses the LOD hierarchy.

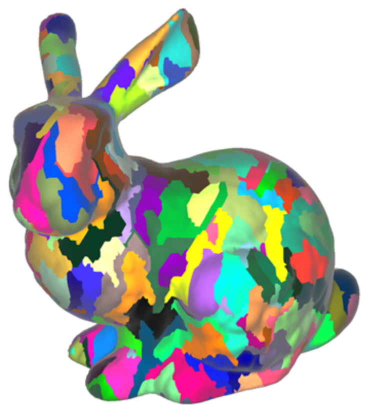


Figure 3.2: Mesh split into meshlets [31]

3.4.2 Shaders

The three shaders in the LOD hierarchy rendering pipeline are, in order of execution:

1. `lod.task` – task shader which decides which meshlets to render
2. `lod.mesh` – mesh shader which assembles the meshlet vertices and primitives and sends them to the rasterizer

3. `phong.frag` – fragment shader which uses basic Blinn–Phong reflection model to shade each fragment/pixel

3.4.3 User interface

As the user interface is also a rendered component of the screen, the renderer also handles the rendering of the UI. Fortunately, the ImGUI library provides a simple way to hook the UI rendering to Vulkan. The user of ImGUI just needs to provide some Vulkan elements to the library so it can internally initialize all necessities (like fonts, et cetera) and then just add two calls to ImGUI in the Vulkan command buffer recording.

The UI elements are then added separately in a class. Most of the UI elements are directly linked to the configuration values. So, setting something up in the configuration file or command line arguments automatically loads and synchronizes itself in the UI.

3.5 Data structures and algorithms

This section contains the various data structures and algorithms used in the Application. Higher-level concepts are shown as pseudo-code and implementation details (which are also often important) are shown directly as C++ or GLSL.

As mentioned in Section 2.5, the Application will implement the LOD hierarchy using the batched multi-triangulation based on [38] with the addition of using the new capabilities of the GPU – using mesh shaders and task shaders to accelerate the batched nature of the algorithm.

The LOD hierarchy algorithm can be logically split into two parts:

1. The preprocessing – building of the LOD hierarchy
2. The runtime usage of the LOD hierarchy to render the mesh

3.5.1 The LOD hierarchy building

In the constructor, the `HierarchyBuild` class receives a `Mesh` consisting of an array of *vertices* and *triangles* (and a *bounding sphere*). Note that the triangles are just arrays of three `uint32_t` to reference the index of a vertex in the vertex array.

The LOD hierarchy itself also contains the array of vertices (which is just `std::moved` from the input `Mesh`. In addition to that, the LOD hierarchy contains `Nodes`. These `Nodes` are basically meshlets with some additional attributes such as their *bounding sphere*, *group center and error* and *parent group center and error*. The meaning of these additional attributes will be explained later.

To conserve space and reduce CPU-GPU bandwidth, the triangle indices inside a meshlet are remapped so that the vertices inside a single meshlet can be referenced by a single byte (`uint8_t`) – meshlet local index. A visual diagram of this compression can be seen in Figure 3.3 [31].

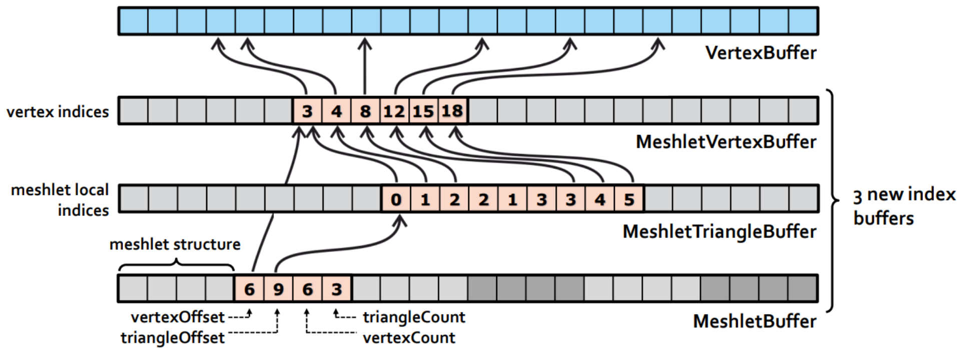


Figure 3.3: Diagram of buffers needed for efficient meshlet/node storage [1]

Then the final `Node` structure looks like in the Listing 3.1. The entire LOD hierarchy has the necessary members listed in Listing 3.2.

Listing 3.1: Compressed Node structure

```
struct Node
{
    uint32_t vertexOffset = 0;
    uint32_t triangleOffset = 0;
    uint32_t vertexCount = 0;
    uint32_t triangleCount = 0;

    // Bounding sphere center (xyz) and its radius (w).
    glm::vec4 boundingSphere{};

    glm::vec3 groupCenter{};
    float groupError{};
    glm::vec3 parentGroupCenter{};
    float parentGroupError = FLT_MAX;
};
```

For convenience and simplicity, the structures used for the *preprocessing* stage are not compressed so they are easier to work with. The overhead of using a bigger structure is not really important as the preprocessing is intended to be done only once and then saved to disk. Only at the end of preprocessing is the LOD hierarchy compressed into the form above for runtime usage (and serialization to disk).

Listing 3.2: Class members of the LOD hierarchy structure

```
class LodHierarchy
{
    std::vector<Vertex> vertices{};
    std::vector<uint32_t> vertexIndices{};
    // MeshletTriangleBuffer in Figure 3.3
    std::vector<std::array<uint8_t, 3>> nodeTriangles{};
    // MeshletBuffer in Figure 3.3
    std::vector<Node> nodes{};
    glm::vec4 meshBoundingSphere{};
}
```

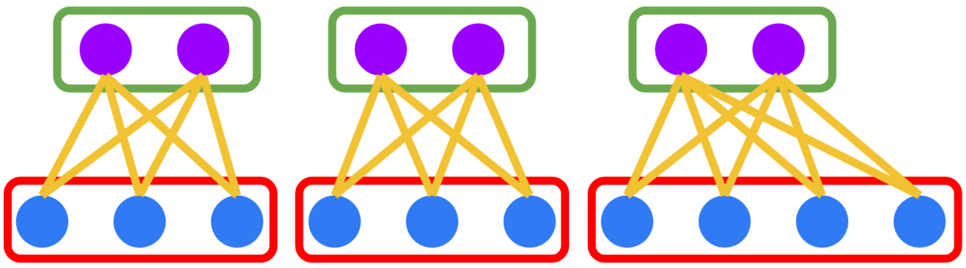
With the data structures described, the building of the whole LOD hierarchy is done according to the pseudocode in Listing 3.3 by calling `BuildHierarchy`.

Listing 3.3: High-level pseudocode of LOD hierarchy build

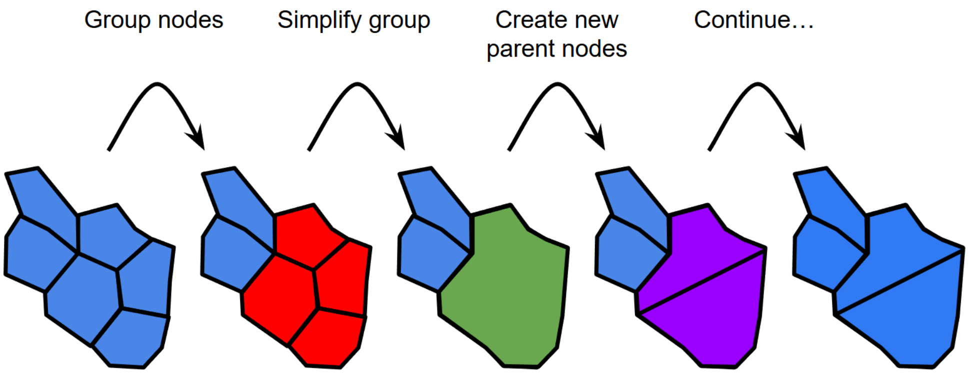
```
def AddLevel(nodesToProcess):
    // Nodes to process in next level
    nextToProcess = []

    while (group = GroupNodes(nodesToProcess)):
        simplGroup, simplError = Simplify(group)
        newNodes = CreateNodes(simplifiedGroup, simplError)
        nextToProcess.append(newNodes)

    return nextToProcess

def BuildHierarchy(mesh):
    nodesToProcess = CreateNodes(mesh.triangles, 0)
    while (nodesToProcess not empty):
        nodesToProcess = AddLevel(nodesToProcess)
```

The `CreateNodes` function clusterizes the input into meshlets/nodes. The `GroupNodes` function groups these nodes into even bigger "meshlets," which are then simplified and then reclusterized.

For a visualization of the node graph, see Figure 3.4a. After the first `CreateNodes` call, the mesh is split into nodes (*blue*). Then these nodes are grouped (*red*). The triangle set of each group is simplified individually. That creates a simpler (*green*) triangle set with the same boundary as had the



(a) : Graph visualization of node grouping in LOD hierarchy



(b) : Mesh visualization of group simplification

Figure 3.4: Visualization of node grouping in LOD hierarchy

group before[3]. The green triangle set is then split again into nodes (*purple*). To better understand what is happening with the actual mesh, see Figure 3.4b which displays one iteration of the group simplification. Color meaning is matching between Figure 3.4a and Figure 3.4b.

The nodes do not actually retain any references to each other after building is complete, but for each parent-child relation (*orange*) it must hold that (see 3.1):

- The parent's `groupError` is higher than its child. So, any sequence from root nodes to leaf nodes is always monotone.
- When a parent node is created, its `groupError` is the same as all the other parents' in the group. This is never adjusted in the future.
- When a parent node is created, its `groupCenter` is the average center of the group. This is never adjusted in the future.
- The children's `parentGroupError` is set to the `groupError` value of the parent.
- The children's `parentGroupCenter` is set to the `groupCenter` value of the parent.

[3]Very important note is that the green simplified groups come *before* the purple nodes!

The monotonicity is ensured by taking the maximum of `groupErrors` of the children and adding the simplification error of the group.

The first level[4] differs in that it always has exactly `0.0f` simplification error, because it is the original most detailed mesh representation.

Then, each parent of these zero-error children has more and more simplification error the higher the graph goes. Note that the topmost parent(s) have `parentGroupError` set to infinity so that their non-existent parent is never chosen during runtime.

CreateNodes

To create nodes from a triangle list, the function first computes a vector of *triangle adjacencies* – one `TriangleAdjacency` for each triangle. The `TriangleAdjacency` holds up to three (one triangle can have at most three edge-neighbors) neighbors of each triangle. With this adjacency information, the algorithm creates the nodes by calling `CreateNodes` according to the pseudocode in Listing 3.4.

From the pseudocode, it can be seen that each node starts with one triangle and grows to its neighbors until it reaches a constant cap of either `MAX_TRIANGLES` or `MAX_VERTICES`. These constants can be fine-tuned for best performance (for example, NVIDIA recommends up to 64 vertices and 126 triangles [31]).

The scoring is done by using a slightly modified version of lazy bounding sphere scoring introduced in [26]. In short, the scoring of a neighboring triangle is done by priority from the most important to the least important[5]:

NEGATIVE How many additional unique vertices would appending this triangle add to the currently created node – that prioritizes triangles which already have some vertices in the node. Note that because the neighbors are always edge-adjacent, the number of additional vertices can be either 1 or 0.

POSITIVE If the triangle (its centroid) is already in the node bounding sphere. With this, the node fills in, so it is as compact as possible.

NEGATIVE How much will the bounding sphere of the node enlarge when this neighbor is added. If the node does not have any neighbors already in its bounding sphere, it must enlarge it. This ensures the enlargement of the bounding sphere is as low as possible.

[4]Note that indexing is from zero, so first level is actually level zero.

[5]POSITIVE means bigger number is better, NEGATIVE vice versa.

Listing 3.4: Creation of the hierarchy nodes

```
def ComputeNeighborScore(unusedTriangles, neighbor, node):
    if (!unusedTriangles.contains(neighbor))
        return -infinity

    // Score based on priority
    score[0] = -HowManyVerticesWillNeighborAdd(neighbor, node)
    score[1] = IsInNodeBoundingSphere(neighbor, node)
    score[2] = -BoundingSphereEnlargesBy(neighbor, node)

    return score

def CreateNode(unusedTriangles, adjacencies):
    node = Node()
    node.insert(unusedTriangles.pop())
    while (true):
        bestTriangle = None
        for (nodeTriangle : node.triangles):
            // The following for uses adjacencies
            for (neighbor : adjacencies.From(nodeTriangle)):
                score = ComputeNeighborScore(neighbor)
                if (score > bestTriangle.score):
                    bestTriangle = neighbor

        node.insert(bestTriangle)

        if (node.triangles.size > MAX_TRIANGLES
            or node.vertices.size > MAX_VERTICES):
            node.remove(bestTriangle)
            return node

        unusedTriangles.remove(bestTriangle)

def CreateNodes(triangles):
    nodes = []
    adjacencies = ComputeAdjacencies(triangles)

    while (triangles not empty):
        nodes.append(CreateNode(triangles, adjacencies))

    return nodes
```

GroupNodes

The grouping of nodes and clusterization of triangles into nodes are different algorithms, but in many aspects, they are similar, so this section will describe the grouping in relation to Section 3.5.1.

The `GroupNodes` function also creates a vector of adjacencies between nodes, but in this case the number of neighbors is not limited to three. Also, the `NodeAdjacency` structure holds information about how many vertices are shared – on the boundary with the neighbor. The created group does not have a limit on the total count of vertices or triangles; instead, it simply has a target count of nodes that it should contain.

Scoring of neighboring nodes is done by dividing the number of common/shared vertices between the currently examined node and its neighbor by the triangle count of the neighbor. So, in short:

$$score = \frac{sharedVertices}{neighborTriangles}$$

This "encourages" the group to add neighbors which have a high amount of shared vertices, but also tries to promote neighbors with lower triangle counts as they would be left out otherwise – as nodes with lower triangle counts can never have as many shared vertices.

Simplify

For the step of simplification of the group, the library *Meshoptimizer* is used. The user of the library (in this case the Application) can use the option `meshopt_SimplifyLockBorder` to enforce that the simplification algorithm does not move the border vertices. That is essential for the simplification of a group because otherwise there could be visible cracks as described in Chapter 2.

The implementation is abstract enough, so the library call could be replaced by another mesh simplification library or a hand-crafted solution.

Compression and saving

After the preprocessing, the whole LOD hierarchy is compressed into the form seen in Listing 3.2. The resulting data is then serialized to disk so the hierarchy is built only once and then simply loaded for each subsequent Application launch. In configuration, the user of the Application can force rebuilding of the hierarchy by using `ForceReserialization=true`.

The full source of the lod hierarchy preprocessing can be seen in the file `src/mesh/lod.cpp`.

3.5.2 LOD hierarchy runtime handling

Although preprocessing is arguably complex and not very optimized, it serves to improve the performance of the *runtime* rendering as much as possible. At runtime, when a scene contains a not-yet-loaded model, the Application deserializes the `LodHierarchy` from disk or builds it if it is the first time the model is contained in a scene. After that, the LOD hierarchy is copied from CPU to GPU. This does not cause as much overhead as modern GPUs with PCIe can transfer multiple GB of data per second [9].

During the rendering of a frame, these operations happen (in order of execution):

1. Copy per-model uniforms (transforms) from CPU to GPU.
2. Check if a LOD hierarchy is present on the GPU – skip this model if not.
3. Based on a bounding sphere, frustum cull the whole mesh on the CPU.
4. Push index of the model (to access correct uniform) from CPU to GPU.
5. Bind LOD hierarchy descriptor set.
6. Execute `drawMeshTasksEXT(count)` to launch the task shader. The count of mesh tasks is computed as `(nodeCount / TASK_THREADS) + 1`.
7. Task shader invocations fill *payloads* with data specifying which nodes to render, and then schedule mesh shader invocations.
8. Mesh shader invocations fill the vertex and primitive arrays for the rasterizer, and then schedule rasterizer work.
9. The hardware rasterizer rasterizes and schedules fragment shader work.
10. The fragment shader shades each pixel with Blinn-Phong shading.

When using multiple frames in a *sequence*, the CPU can start processing another frame as soon as it executes all the `drawMeshTasksEXT` commands. However, if the CPU is too fast and processes more frames than are in the swapchain faster than the GPU can render a single frame, it must wait for the GPU to finish rendering the first frame.

Most of the steps are common and do not warrant a deeper inspection, but the *task shader* and *mesh shader* directly work with the LOD hierarchy, so they deserve a closer look. Inspiration for the shader implementation was drawn from [1].

The task and mesh shaders also contain synchronization primitives and logic, so it is recommended that a curious reader look directly into the source code of the Application for details.

Task shader

The responsibility of the task shader is to choose which nodes should be drawn. There are two main checks that determine whether the node should be drawn: culling by *error* and culling by *frustum*.

Culling by frustum is not really an innovative algorithm, but the task shader has a noteworthy possibility of doing effective frustum culling *on a per-node level*. That means, for example, if the camera is close to an object and only a subset of nodes is visible from the whole mesh, the task shader can cull many nodes that would have to be rasterized otherwise. An example of moderate meshlet frustum culling can be seen in Figure 3.5.

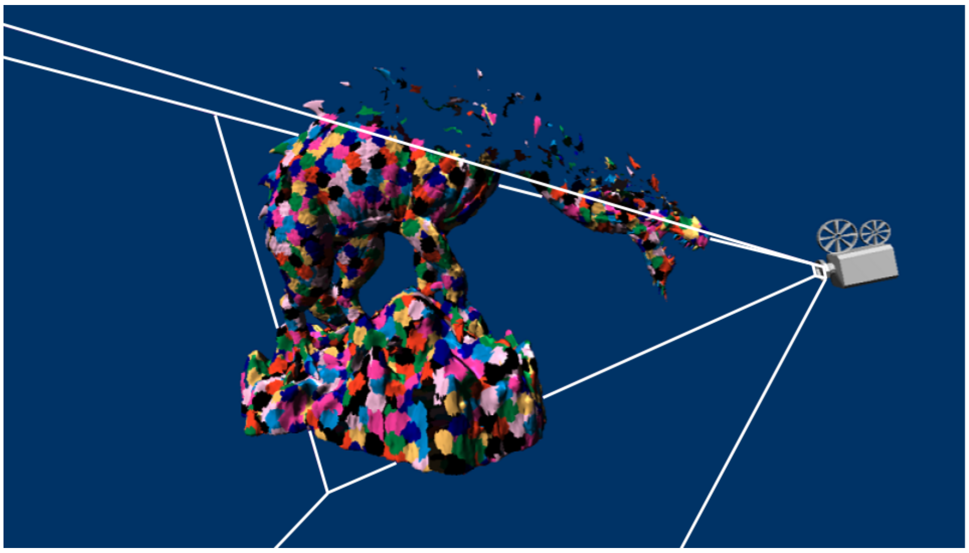


Figure 3.5: Task shader frustum culling example [36]

The other culling – culling by error stems from the LOD hierarchy nature. The goal is to cull too detailed *and* too coarse nodes. Due to the way the LOD hierarchy is built (see Section 3.5.1) the `groupErrors` in the node "graph"[6] are always monotone. By taking inspiration in [28], it is not necessary to traverse the graph. Each node can be evaluated by itself using the GLSL code in Listing 3.3[7].

The most important part of the `cullByError` is the line containing the comparison with error `THRESHOLD`. In other words, it means that if the parent error is lower than the error threshold, then the parent node should be drawn as it is detailed enough and cheaper to draw (so do not draw this node). And if the current node error is too high, then it means that the current node is too coarse and a finer level – a child will be drawn (so do not draw this

[6]Remember that nodes do not have any references to each other, only the parent group error and center.

[7]Uniform transform access was omitted for simplification.

Listing 3.5: Snippet from the task shader

```
float computeError(vec3 center, float error)
{
    if (error == 0.0)
        return 0.0;

    vec3 centerInCamera = (VM_MATRIX * vec4(center, 1.0)).xyz;
    float centerDistance = length(centerInCamera);
    return error / (centerDistance * centerDistance);
}

bool cullByError(Node node)
{
    float parentError = computeError(
        node.parentGroupCenterAndError.xyz,
        node.parentGroupCenterAndError.w);

    float error = computeError(
        node.groupCenterAndError.xyz,
        node.groupCenterAndError.w);

    if (parentError <= THRESHOLD || error > THRESHOLD)
        return true;

    return false;
}
```

node).

The threshold is a uniform value that is set for the entire scene at the start of rendering. When it is set to zero, only nodes with the simplification error of exactly zero – so only the maximally detailed nodes will be drawn. The user can then set the error threshold for the whole scene. With this one variable, the performance/quality trade-off can be adjusted.

The final error metric is currently computed just as:

$$finalError = \frac{error}{distanceToCamera^2}$$

But it could very well be experimented with and fine-tuned.

The full source of the task shader can be seen in the file `shader/lod.task`.

Mesh shader

The mesh shader receives data from the task shader (called *payload*). The data simply contain the indices of the nodes that should be rendered. Then the mesh shader reconstructs the true 3D vertex positions and primitive/triangle indices from the compressed `Node` structure and transforms them by the per-model uniform.

Optionally, it can also do per-primitive culling and other operations, but the effectiveness of that is lowered if the task shader is already doing that type of culling on a per-meshlet basis. The culling computation would have to be done for *all* the primitives, even though only a fraction of them could be culled – because the task shader already culled most of them.

The full source of the task shader can be seen in the file `shader/lod.mesh`.

3.6 Example

This section will showcase the usage of the Application with some screenshots and explanations. For the full Application guide, see the Appendix A. The example uses a 3D mesh created by subdividing the default *suzanne* in Blender called *massive-suzanne* (8 million vertices) [10]. For this example, the mesh will be stored on the path "*C:/data/objs*".

After using the command "*./<executable path> Model=massive-suzanne ResourcesPath=C:/data/objs*", the Application first preprocesses the 3D mesh into the LOD hierarchy. As already mentioned, this LOD hierarchy is then saved to disk, so on subsequent launches, it is only loaded. Once the LOD hierarchy is available, the Application starts rendering the 3D scene onto a window surface. The user interface will be collapsed on the first launch, as can be seen in Figure 3.6.

The expandable user interface boxes are *Options* which enable the user to adjust various runtime configuration options, *Stats* which display a rolling average of frame time and frames per second. The last box *How to use this app* basically copies Section A.2. By default, the Application displays the meshlets as random-colored, which aids in visualizing the switching between different nodes. By increasing the "*Options » LOD hierarchy » Error threshold*" from 0 to approximately 0.001, the Application applies more aggressive node selection – meaning less detailed nodes are rendered closer to the camera. The error threshold of exactly zero means that the most detailed nodes are always visible.

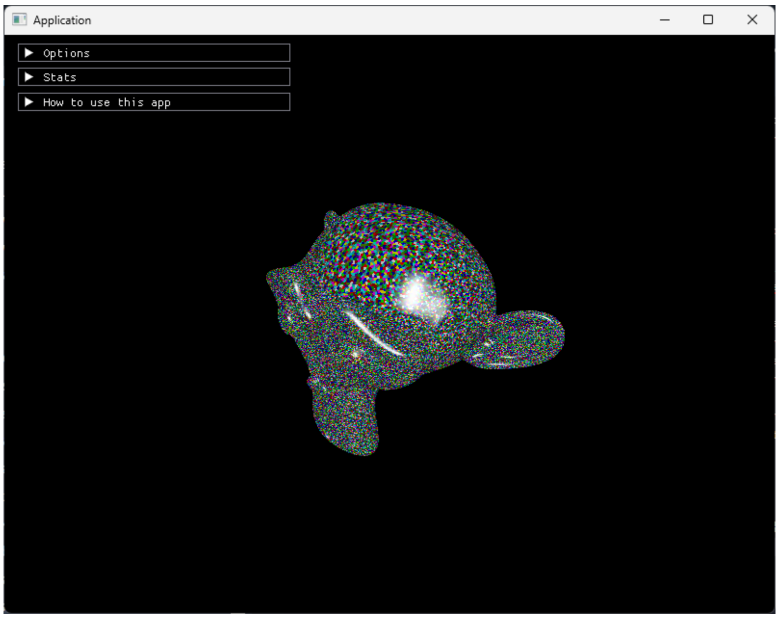
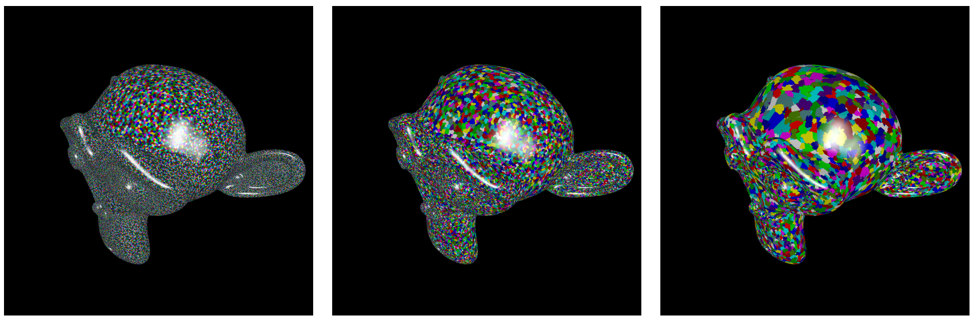
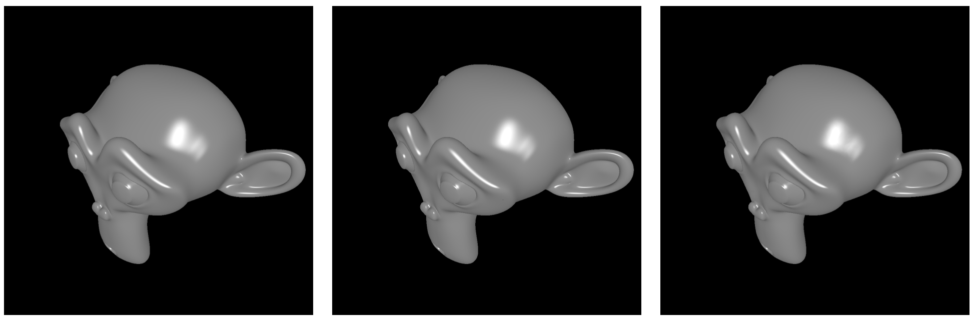


Figure 3.6: The Application upon launch



(a) : $ET = 0$ **(b) :** $ET = 0.00001$ **(c) :** $ET = 0.001$

Figure 3.7: Colored comparison of different error thresholds (ET)



(a) : $ET = 0$ **(b) :** $ET = 0.00001$ **(c) :** $ET = 0.001$

Figure 3.8: Colored comparison of different error thresholds (ET)

(a) : $ET = 0$ **(b) :** $ET = 0.00001$ **(c) :** $ET = 0.001$

Figure 3.9: Zoomed comparison of different error thresholds (ET)

The difference is very noticeable (as seen in Figure 3.7) when drawing meshlets with different colors. When the color is switched to a uniform color by unchecking *"Options » Fragment shader » Draw colored"* and shaded smoothly by the Blinn-Phong reflection model, the difference is practically non-existent, as seen in Figure 3.8. For this mesh, the frame time is approximately halved for the threshold 0.00001 with no perceptible difference. For the error threshold 0.001, the frame time is about one fifth, but on the very shiny part of the head, there is a slight visual difference, as seen in Figure 3.9.

Chapter 4

Evaluation

The performance of the Application can be evaluated based on multiple criteria. When comparing the raw performance of the renderer, the simplest measurement is the frame render time. The application has the capability of rendering the same scene with the classic vertex shader pipeline. So apart from just absolute frame time values, the frame times can be compared between the task/mesh shader pipeline (with and without the LOD hierarchy) and the vertex shader pipeline. The vertex shader pipeline cannot use the implemented LOD hierarchy properly, so it is evaluated without it.

LOD techniques should also be evaluated based on perceptible difference when a lower-resolution representation is used. With the LOD technique used, the visual difference between the original mesh and the adaptive lower resolution representation can be measured with *Structural Similarity Index Measure* (or *SSIM*)[48]. The resulting value of SSIM is in the range $[0, 1]$ with 1 meaning the absolute similarity (image identity) and 0 meaning the lowest similarity.

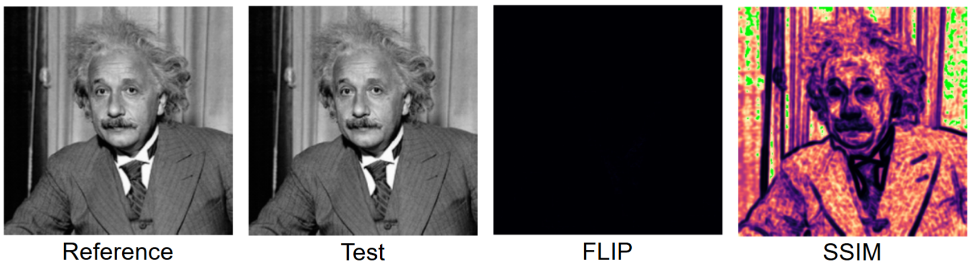


Figure 4.1: Comparison of FLIP and SSIM metrics

There exists a newer comparison algorithm called *FLIP* (also can be seen as ꟻLIP) [2]. This algorithm aims to better model the *perceived* difference between images. In this case, the value of 0 means zero difference/error (images perceived as identical), there is no specific upper bound on the error, but the higher the value goes, the more the images are perceived as different. The ability to better model human perception can be seen in Figure 4.1.

Both SSIM and FLIP can output a “difference” image which shows where the compared images differ. The output similarity/error value is just an aggregation of these differences over the whole image. For an example of this difference image see the *SSIM* image in Figure 4.1. The images rendered by the Application will be compared both by SSIM and FLIP.

Another option regarding measurement is the choice of the 3D meshes that will be rendered. For this evaluation, only publicly available 3D meshes are used. Some of the meshes are common in 3D mesh rendering papers like the Stanford Bunny, Buddha Statue, or the Chinese Dragon. It is expected that the LOD hierarchy generation algorithm of the Application will be best suited to meshes which contain “one” object, as then the algorithm can iteratively simplify the mesh. For meshes (for example, architectural or vehicular) that contain multiple distinct objects, it may happen that the individual objects are so simple that further simplification does not make sense, but the overall complexity of the scene is not low enough. Both of these types of scenes are tested and evaluated.

When viewed from different angles, the Application culls different meshlets, so a seeded randomized sequence of camera positions and orientations is used. The statistics of how many meshlets are culled are reported.

The Application also has a range of configuration options that slightly change its behavior. In particular, the *error threshold* can be adjusted to alter the aggressiveness of node culling. Different combinations and comparisons of these are also evaluated.

The time of preprocessing of the original meshes into the LOD hierarchy is measured, but the results of the preprocessing specifically are just for completeness, as the preprocessing would be done once in a production-level application and then saved to disk. At runtime, the application would only load the already computed LOD hierarchy from a hard-drive.

The Python script handling the automatic launching and evaluation of SSIM and FLIP is also in the repository of the Application in the folder `py-stats`. More detail is added for each individual measurement in their respective sections.

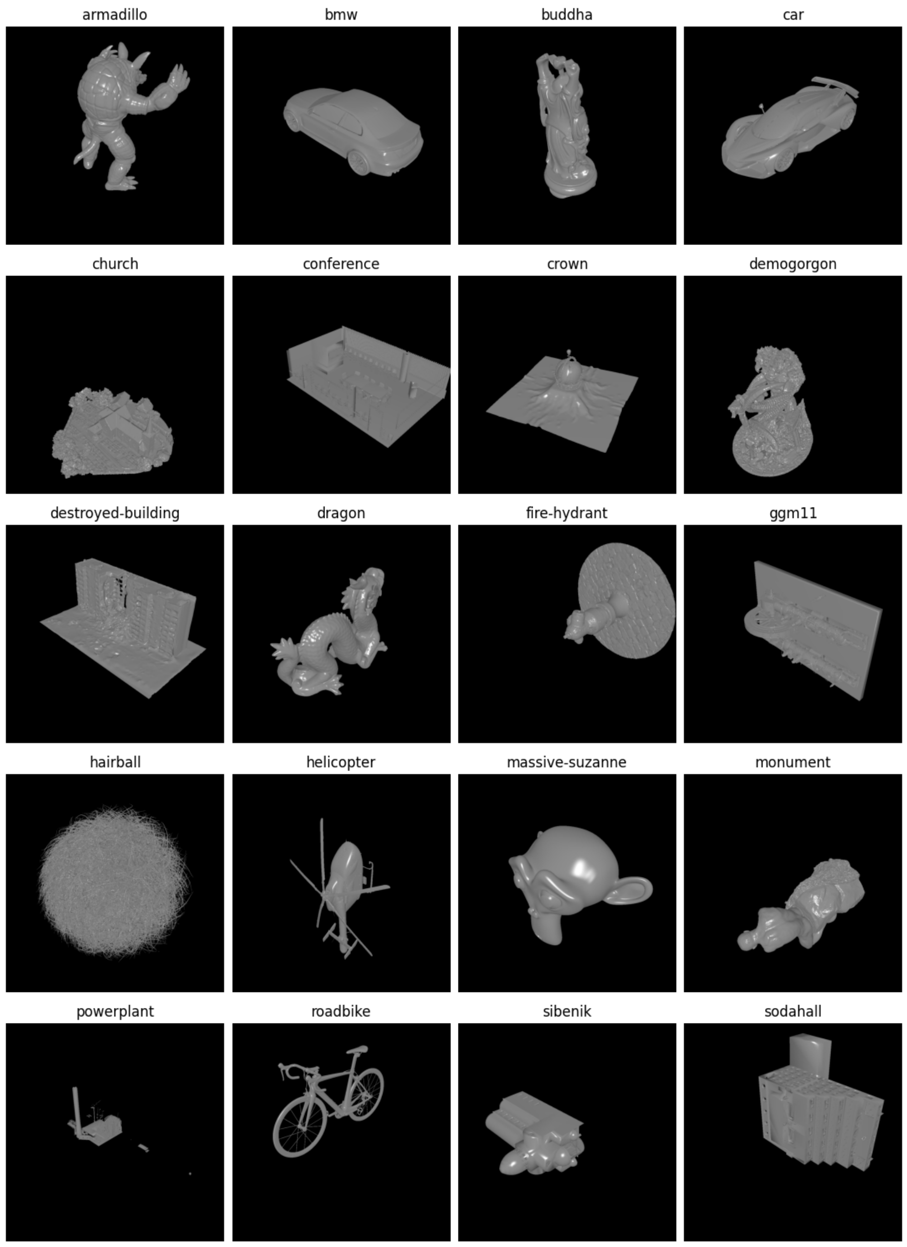


Figure 4.2: Meshes used during evaluation

In the evaluation, 20 different meshes were used. The vertex and triangle counts are listed in Table 4.1. The default view of the scene of these meshes can be seen in Figure 4.2. The computer on which the Application was measured had the following specifications:

- Operating system: Windows 11
- Compiler: MSVC
- GPU: NVIDIA GeForce RTX 3070 Laptop

- CPU: Intel i7-10870H @ 2.20GHz
- RAM: 16 GB (2933 MT/s)
- Disk: KXG60ZNVV512G KIOXIA (SSD)

mesh	vertices	triangles
armadillo [44]	172 974	345 944
bmw [35]	198 895	384 893
buddha [44]	543 439	1 087 300
car [15]	219 463	420 643
church [5]	1 237 460	2 470 330
conference [35]	189 033	331 179
crown [39]	2 465 980	4 868 920
demogorgon [6]	1 254 260	2 508 610
destroyed-building [40]	44 931	89 932
dragon [44]	435 491	871 198
ggm11 [4]	7 715 960	15 373 900
hairball [35]	1 440 000	2 850 000
helicopter [3]	11 727 100	23 454 800
massive-suzanne [10]	8 063 620	16 121 900
monument [7]	3 588 860	7 177 020
fire-hydrant [8]	1 523 670	3 043 230
powerplant [35]	5 424 570	12 759 000
roadbike [35]	848 715	1 676 780
sibenik [35]	42 962	80 125
sodahall [33]	1 285 510	2 169 130

Table 4.1: Vertex and triangle counts for the evaluated meshes

For all tests and measurements, the ImGUI user interface was turned off. The Application was compiled in release mode with maximum optimization enabled. Unless stated otherwise, the measurements were done on fullscreen FullHD (1920x1080) resolution.

4.1 First measurement

The first measurement was done with the aim to compare an approximated use-case of displaying multiple models in a scene from different camera angles.

The configuration of the Application can influence how the `Scene` is initialized. Most notably, the mesh (in the format of an OBJ file). By default, this loaded mesh is uniformly rescaled so that it fits into the space $[-1, 1]$. Another option the configuration provides is arranging the scene meshes into a grid – enabling direct rendering of the mesh at multiple distances/angles. Camera position can be randomized by a specific seed (so the rendered frames can be later compared between runs of the Application)[1].

So in the first measurement, the mesh was placed in a grid of 10 by 10 instances (so 100 meshes in total) with the grid spacing left on the default `2.0f` as can be seen in Figure 4.3[2]. For the frame times, the time was measured over 1000 frames with the camera positions randomized in the extents (in each axis) $[-20, 20]$ with instant position changes. Out of these 1000 frames, the first 20 were saved to disk – for performance reasons in a different Application run – to compare them with the SSIM and FLIP error metric.

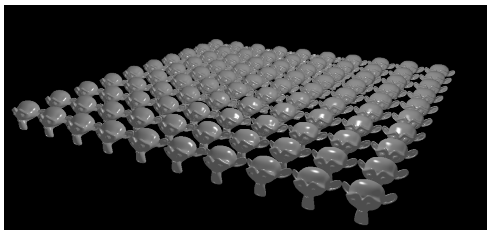


Figure 4.3: 3D scene for the first measurement

The raw data from the first measurement can be seen in an appended Table B.1. Each model was measured using 8 different setup configurations. The first configuration used a traditional *vertex* shader pipeline, the following *lod* configurations were the implemented LOD hierarchy with default settings using an error threshold of 0, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, and 1 respectively. That means that the *lod_0* setup rendered maximum resolution nodes and subsequent *lod_N* rendered coarser and coarser nodes – ideally thus increasing performance.

The SSIM and FLIP comparison of lod setups was done both against the traditional vertex pipeline setup to measure the absolute difference from the original and against the previous (as ordered in the table) lod to measure an approximated perceptible "popping" when the rendered nodes are switched.

From this data, several graphs are plotted by the library Matplotlib [25]. Figure 4.4 displays the absolute times to render 1000 frames for each mesh. The absolute times of the vertex shader pipeline and zero error threshold LOD (*lod_0*) are dominated by the meshes with most vertices, with the highest one being the *helicopter*[3]. It can be seen, though, that when the error

[1]Camera view is always directed to the center of the scene – $[0, 0, 0]$ in world coordinates.

[2]Grid spacing of `2.0f` means that after normalization the meshes in the grid could be exactly touching.

[3]Note that the *helicopter* mesh has over 23 millions of triangles so the full scene with

threshold is increased, the render time decreases as expected.

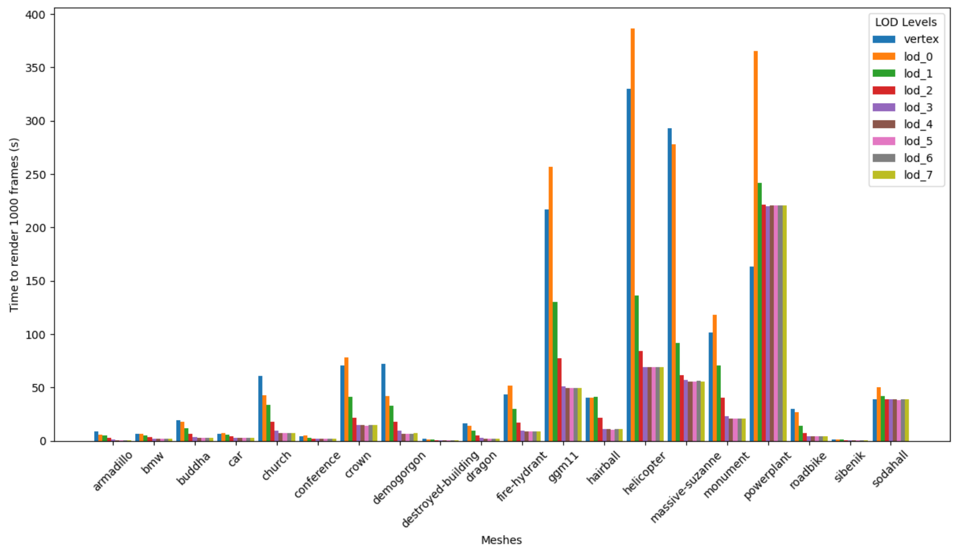


Figure 4.4: Absolute times to render 1000 frames

To further examine the relative differences, Figure 4.5 was plotted. It takes the time to render the mesh through the vertex pipeline as a baseline to contrast the performance of different LOD error thresholds against it.

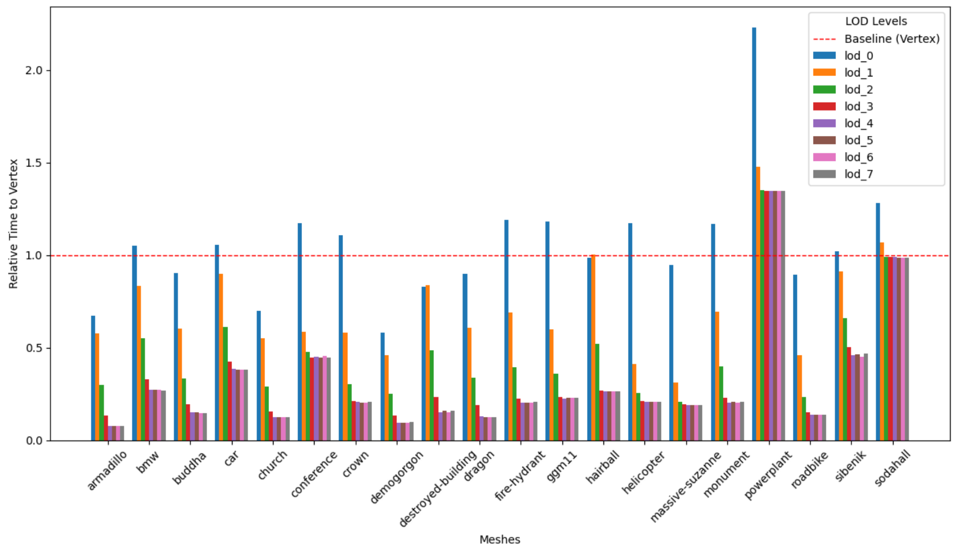


Figure 4.5: Times to render 1000 frames relative to the vertex pipeline

For 11 of the measured meshes, the vertex shader pipeline performs better than the *lod_0* setup. However, for most meshes, this difference is relatively small. The cause of this difference is most likely the fact that the traditional

100 of these contains 2.3 *billion* triangles.

vertex shader pipeline always renders only the highest resolution representation of the mesh, while the mesh shader pipeline (even for zero threshold) still performs the node selection process, which incurs this overhead. For the meshes where the mesh shader pipeline beats the performance of the vertex shader pipeline, the performance increase is most likely gained through the meshlet culling, which can discard parts of the mesh (which is impossible for the vertex shader pipeline).

The performance gain when using a higher error threshold to render coarser nodes when the camera is further away can be clearly seen in Figure 4.5 for most of the measured meshes. It can also be seen that the error threshold reaches a "maximum value" where further increase has very little to no effect on rendering performance. This is the logical consequence of the nodes having a finite absolute simplification error estimate. If the error threshold is so high that the most simplified – "root" nodes are always chosen, there is no more room for performance gains.

Ideally, there would be only a few root nodes, so that the renderer would have to rasterize a very limited number of triangles. From the measurement, it seems that the node simplification process terminates too early. By default, if the simplification does not simplify the meshlet vertices by at least a factor of 0.7, the simplification is rejected (and no new node is created) to keep the number of nodes bounded to a reasonable amount.

One exception where the mesh shader pipeline performs much worse than the vertex shader pipeline is on the *powerplant* mesh. It can also be seen on the graph with absolute values in Figure 4.4, that even though *helicopter* has more than twice as many vertices, the render times of the *lod_0* setup are almost the same. This can suggest that the nodes in *powerplant* are poorly distributed/filled. This can happen when the input mesh contains a lot of disconnected components, because the node generation algorithm does not contain any metric to automatically connect close-by vertices. The overhead of processing near-empty nodes then outweighs the other benefits of the mesh shader pipeline. Although the maximum resolution nodes for *powerplant* offer very poor performance, the simplified nodes remedy this at least somewhat. The *sodahall* mesh did not have such poor performance for *lod_0*, but the simplified nodes in further configuration setups did not bring too much of a performance gain either. That is most likely again because the mesh contains many disconnected objects which the simplification algorithm cannot handle.

When the render times are averaged across all the evaluated meshes in Figure 4.6, it can be seen that even though the *powerplant* and *sodahall* are involved, the frame times are on average lower for lod_1 and further. The overhead of the task shader when rendering with zero error threshold is more visible here, but in real use-case scenarios, the error threshold would never be exactly zero.

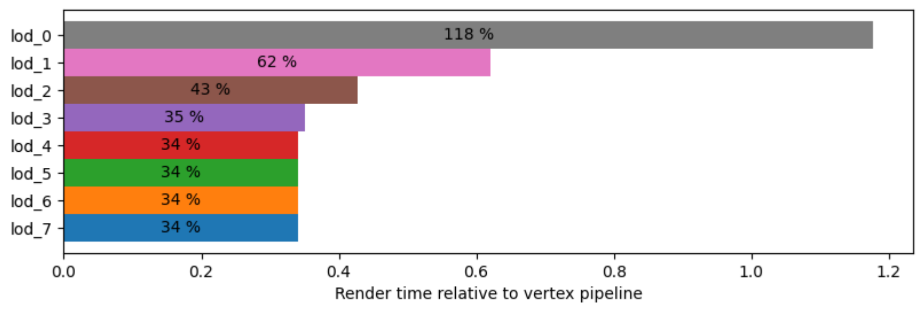


Figure 4.6: Render times relative to vertex pipeline averaged across meshes

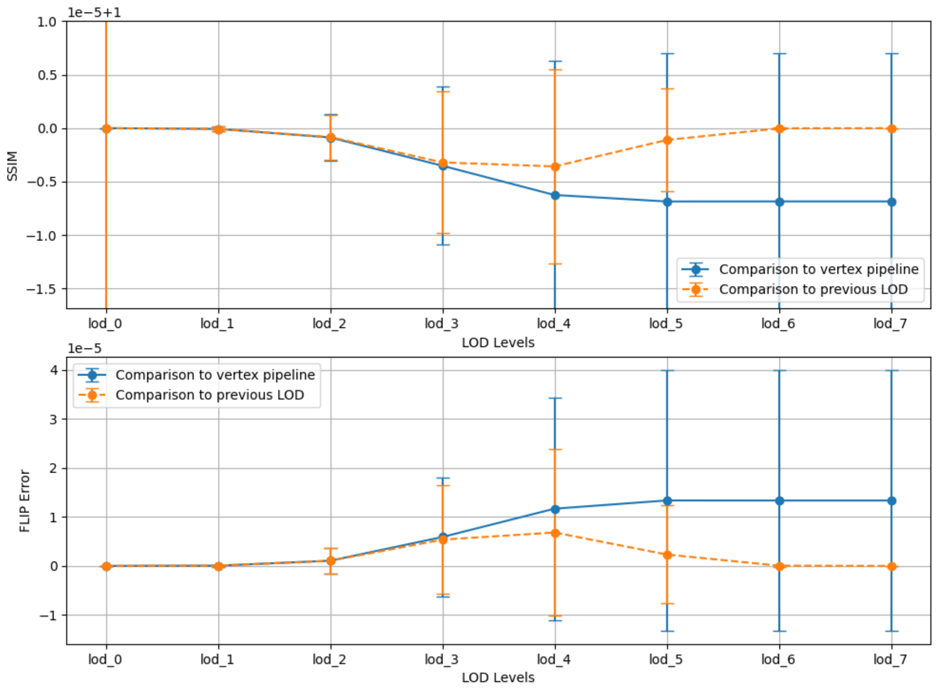


Figure 4.7: Plot of SSIM and FLIP errors averaged across all models

The SSIM and FLIP errors are plotted in Figure 4.7. Both SSIM and FLIP separately plot the errors compared to the “original” mesh rendered by the traditional vertex pipeline (blue) and compared to the previous LOD configuration in the sequence[4]. The values are averaged across all meshes and the standard deviation *of frame comparisons for each specific model averaged across all models* is also plotted.

As expected with increasing LOD error threshold, the SSIM decreases and FLIP error increases – but note the $1e-5+1$ and $1e-5$ exponents in Figure 4.7 – meaning the errors are still quite small. The plot for the most part supports the conclusions drawn from the frame rendering times figures. The error threshold almost reaches its usable maximum around the value set for

[4]Hence lod_0 has zero FLIP error and SSIM of exactly one by definition.

lod_4 at which the renderer already renders the most simplified nodes for *most* camera positions, but there can still be seen a small “nudge” (more noticeable on the orange lines) by increasing the LOD threshold to lod_5 value at which the renderer *always* renders the most simplified nodes.

As a side note, the similarity/error values obtained from the evaluation of the two metrics of SSIM and FLIP are surprisingly symmetrical, and neither one seems to provide any additional information over the other.

4.2 Second measurement

For the second measurement, the goal was to look deeper into the LOD hierarchy and its runtime node selection. This time, the setup did not use the grid, so only one mesh per measurement was rendered. The randomized camera extents were slightly increased to $[-25, 25]$ for every axis to better ensure that the camera can reach positions further away from the mesh. To test only the LOD hierarchy node selection, the node frustum culling was turned off.

The number of configuration setups was increased to 12 for better granularity of the evaluation. From the first measurement, it was obvious that increasing the error threshold more than for the configuration setup of *lod_5* does not bring any benefits, so for the second measurement, the configuration setups were rescaled accordingly by adding more lower error thresholds and adding midpoint values between the existing ones – the comparisons can be seen in Table 4.2.

Error threshold	First measurement	Second measurement
0	lod_0	lod_0
0.0000001	—	lod_1
0.0000005	—	lod_2
0.000001	lod_1	lod_3
0.000005	—	lod_4
0.00001	lod_2	lod_5
0.00005	—	lod_6
0.0001	lod_3	lod_7
0.0005	—	lod_8
0.001	lod_4	lod_9
0.005	—	lod_10
0.01	lod_5	lod_11

Table 4.2: Measured error thresholds comparison

The raw data from the measurement is appended in Table B.2. Each mesh has its own header with information about how many nodes in total are in the LOD hierarchy (*nodes*), how many of the nodes have zero simplification error (*zero nodes*), how many of the nodes do not have any parent (*root nodes*) and how many triangles are in the root nodes (*root tris*). The columns then contain the counts of rendered nodes and triangles averaged over all 1000 frames. The Application exports these stats for each frame separately, so the standard deviation is also calculated (as well as the minimum and maximum can be retrieved).

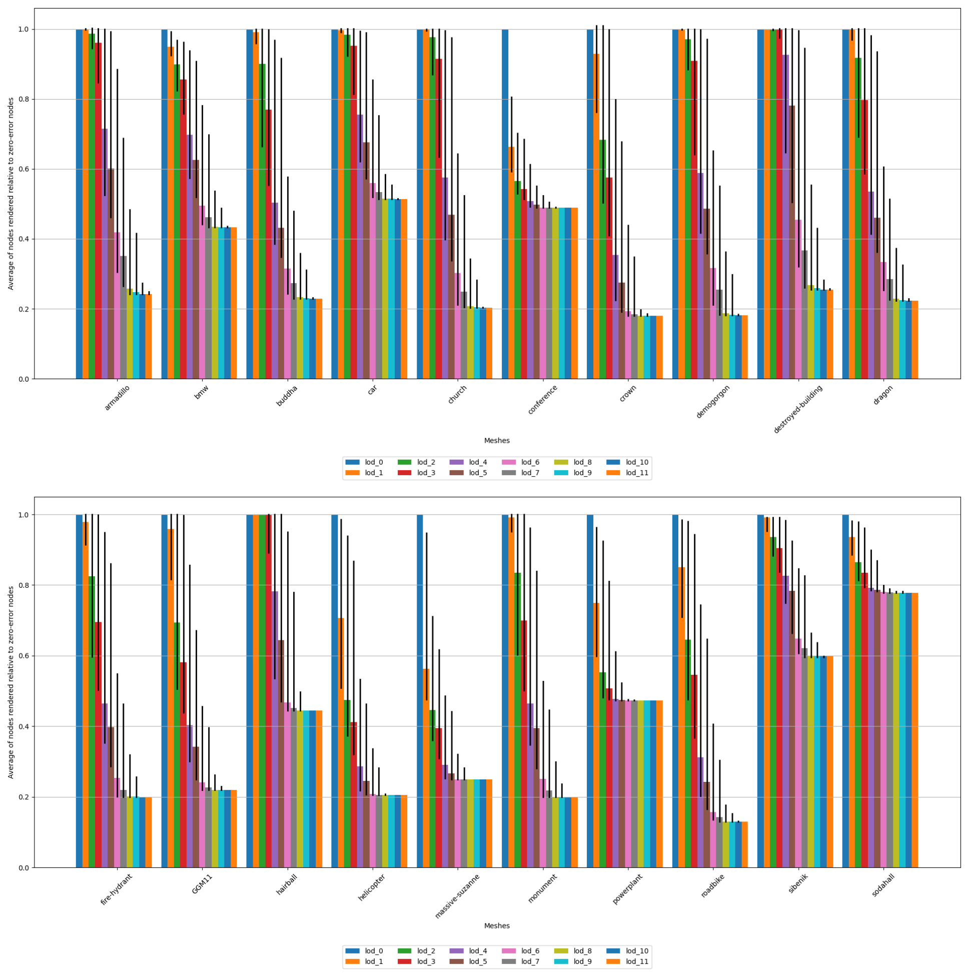


Figure 4.8: Average nodes rendered relative to the count of zero-error nodes

To visualize how increasing the threshold reduces the number of nodes selected for rendering, Figure 4.8 was plotted. It displays the number of rendered nodes per frame averaged across all frames relative to the number of zero error nodes for each error threshold configuration. The black lines above some of the colored bars denote the minimum and maximum of rendered nodes.

When compared with Figure 4.5, the data shows very similar trends for each model. That means the number of selected nodes for rendering is very strictly tied to the render time. Due to rescaling the measured error threshold values, a more gradual increase in performance can be seen for each one.

It is actually very good to see that the minimum and maximum values for rendered nodes are fairly far apart for the middle error threshold values (lod_4 to lod_7 approximately), because it means that the camera distance influences the rendered quality greatly – when the camera is very close to the mesh almost all max-resolution nodes are rendered, and when the camera is far away only coarse nodes are rendered. An observant reader can also spot that for some of the meshes, the lower thresholds (for example *lod_1* and *lod_2* for the *crown* mesh) can actually render more nodes than the baseline zero error nodes. This is a consequence of the greedy nature of the node-building process, which may create less triangle-filled nodes during preprocessing. Then even though the nodes are less detailed, there are slightly more of them.

To examine the exact "sweet spot" of the error threshold, Figure 4.9 was plotted, which averages these relative amounts of rendered nodes across all meshes. Additionally, it contains a measure of the range between minimum and maximum of the rendered nodes (red dash-dotted line). It can be clearly seen that the maximum range is reached for the configurations *lod_4* and *lod_5* which, according to Table 4.2, correspond to error threshold values of 0.000005 and 0.00001 respectively. The exact best error threshold could be expected to be somewhere between these two values. When looking back at Figure 4.6 (note that the error threshold configurations were rescaled according to Table 4.2), the frame time can be expected to be approximately halved when using this best error threshold value.

Also, with the assumption that the simplification algorithm would be fixed to generate also the more aggressively simplified nodes, it could be reasonable to expect that the performance would be even better, because from the figures it seems that the main rendering overhead of the LOD hierarchy is still the rasterization and processing of the nodes in the *mesh* shader (because the amount of nodes rendered directly affects frame time) not the node selection in the *task* shader.

4.3 Hierarchy build times

Although the performance of the preprocessing step is not the focus of this work, the build times of the LOD hierarchy were also measured and reported in appended Table B.3.

When the LOD hierarchy build times are compared against the mesh vertex counts in Figure 4.10, it can be seen that the *helicopter* and *powerplant*

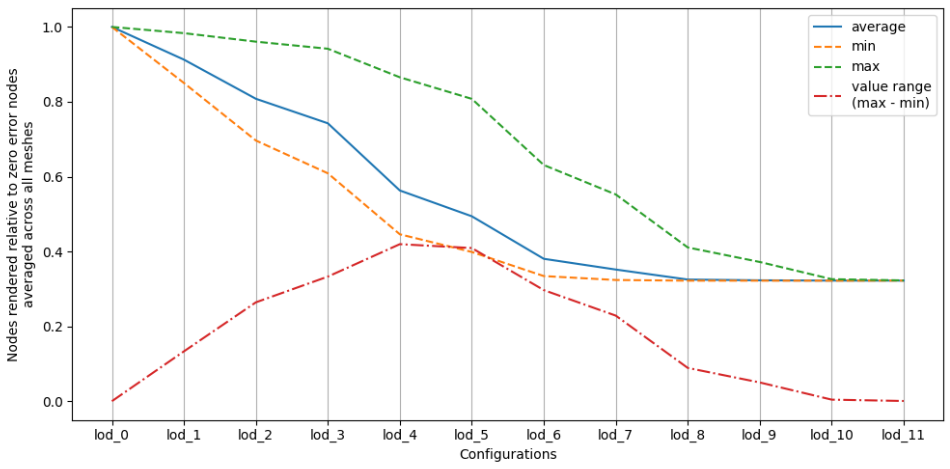


Figure 4.9: Relative number of nodes rendered averaged across all meshes

especially stand out. After a debugging session, it was confirmed that this is due to the fact that after failing to simplify a group (as discussed in Section 4.1), the Application keeps the nodes "open" and tries to simplify them in the next iteration. This behavior is intentional because, when neighboring nodes are changed, it may happen that simplification would be possible. However, if many of the groups fail to simplify, the overhead of processing these nodes multiple times can substantially increase the build time.

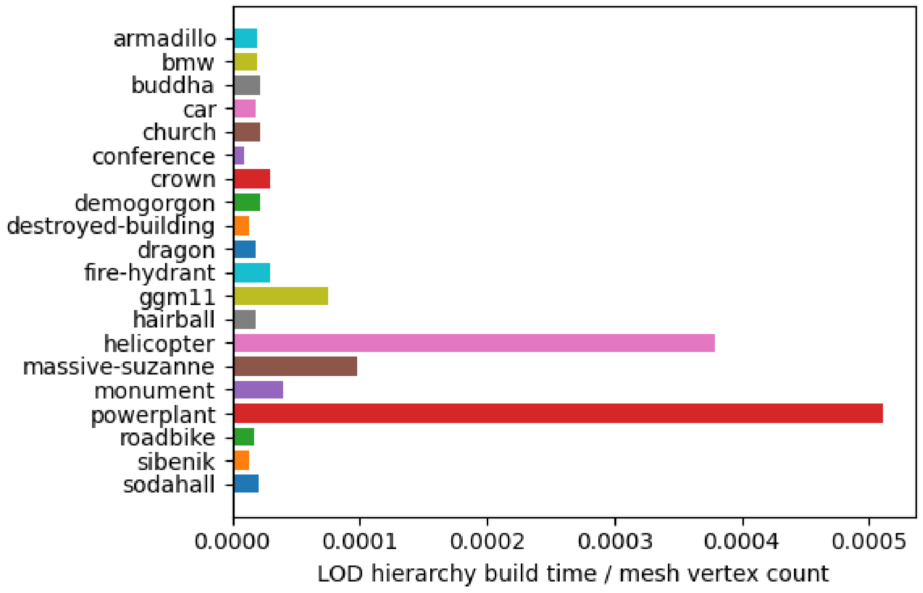


Figure 4.10: LOD hierarchy build times relative to mesh vertex count

4.4 Limitations of current implementation

The Application has several limitations. Some of the limitations were expected and were out-of-scope of this work. The ability to handle textures and other file types apart from OBJ is not implemented but would be fairly easy to add. Another simple limitation is that if the GPU hardware or driver does not support the (at the time of writing this work fairly new) task/mesh shader pipeline, the Application cannot be used at all. The Application is also missing any runtime editing of the 3D scene (apart from turning on mesh rotation).

The initial testing of the Application which was then complemented by the measurements has shown another big limitation of the current implementation; the simplification and node grouping algorithm can simplify only connected components of the mesh and after a finite simplification steps gets "stuck" unable to simplify the grouped nodes more. The algorithm iterates over the triangle neighbors by their edge-connectivity, and although there is a spatial-aware metric of the bounding sphere to rank these neighbors, the neighbors which are not connected by an edge are not even considered. The node grouping algorithm then works similarly. The problem is most apparent for meshes with many disconnected objects like *powerplant* or *sodahall*. This subsequently leads to the LOD hierarchy having too many *root nodes* – nodes which are displayed when the camera is far away. Ideally, the count of root nodes would be very low – could be as low as a single root node.

4.5 Future work

The Application is suited to be extended in multiple ways. The design of the application was done so that the main blocks are abstract enough so that they can be swapped or iterated on with future versions.

The application is currently CPU single-threaded, but there are multiple opportunities for multi-threading. The basic and most influential would be the loading of the meshes in a separate thread, as currently when a mesh is loading, the application thread is blocked. The Application is designed so that only the `MeshBank` would need to be adjusted. Currently, the Vulkan renderer just prompts the `MeshBank` for a mesh and if the `MeshBank` returns a `nullptr`, it skips the model. So, the multithreaded implementation would be just to continuously return `nullptr`s till the mesh is loaded and ready.

Another opportunity for distributing work to multiple threads is in the Vulkan implementation. For example, the transfer of data to and from the GPU could be done – with some proper synchronization – in a separate thread.

From the user interface standpoint, it may be beneficial to implement some kind of basic runtime 3D scene editing for adding models, moving models around the scene, or scaling models. While not very useful for rigorous testing, for quick experimentation, this way could be much faster than packing the whole scene into an OBJ file.

As the OBJ format was mentioned, it would also be possible to implement other loaders for different 3D scene descriptions (for example *FBX*). The current `ObjLoader` simply returns a `Mesh` structure. So, if any other loader was implemented, it could also return a `Mesh` structure and the `MeshBank` would just decide which loader to use based on the file suffix. Alternatively, it would be possible to plug in the quite well-known library *Open Asset Import Library* (usually called *Assimp*)[47] into the code.

4.5.1 More aggressive simplification

Focusing on the LOD hierarchy technology/algorithm, there are also options for future work and improvements. Probably the most influential of them would be to implement a more aggressive node grouping and simplification algorithm. This aggressive variant could be used as a fallback when the current connectivity-preserving algorithm fails or simplifies the node group too little.

In the Application, there already exists an implementation of a Kd-tree, which was used for early mesh shader testing during development. The aggressive simplification algorithm could use this structure to find close-by nodes/triangles, which would enable the Application to have many fewer root nodes and thus increase the performance even further when the camera is far away.

As a bonus heuristic, the information from the original mesh could be used while simplifying nodes throughout all of the hierarchy. The current implementation performs the grouped node simplification in isolation (just with the locked border) without any knowledge about the child nodes or the original mesh.

4.5.2 Unloading of nodes

To handle a large number of different meshes, the Application would have to have a system which adaptively loads and unloads currently unused `Nodes` from GPU/RAM. The compressed structure of the `Node` (see Listing 3.1) would have to be slightly altered to contain all the information necessary to render the Node.

The system would then have to keep only the currently rendered Nodes (from the camera viewpoint) in memory. The task shader would then simply add a small additional code, which would write to a buffer any new “requested" `Nodes` based on the error metric. The requested `Nodes` would then be asynchronously loaded (first to RAM, then copied to GPU memory) and rendered.

This proposed approach would have one assumption about continuous camera movement – as when the camera would move too rapidly, the additional `Nodes` might not have enough time to get loaded and visual artifacts could occur. To combat this, the system could have a buffer of `Nodes` in levels above and below (parents and children in the graph) of the currently rendered `Nodes`.

For instantaneous camera movements (like teleporting) some small subset of coarse `Nodes` could be loaded so the Application would have a fallback.

4.5.3 Textures

Another feature that would have to be added for many use-cases of 3D renderers is the ability to display textures on the mesh. Currently, the most commonly used texture mapping is through UV coordinates, which map each vertex to a point in a texture. The UV data is saved separately for each vertex.

Some kind of texture file handling would have to be implemented into the Application. Otherwise, a rudimentary texture mapping should be rather simple to implement.

Currently, the implementation of the simplification uses the *Meshoptimizer*s `meshopt_simplify` [27] which does not create new vertices. That means that for coarser nodes, the texture coordinates would have more and more artifacts. As no new vertices are created, the vertex texture coordinates cannot be properly interpolated for the simplified mesh.

When combined with the proposal in Section 4.5.2, new vertices could be created, which would enable the simplification algorithm to also compute better simplified texture coordinates.

4.5.4 Pipeline fallback

When the Vulkan implementation during its initialization fails to find a device that supports the task/mesh shader pipeline, there could be some kind of fallback pipeline implemented. The fallback pipeline could either use traditional LOD techniques (see Chapter 2) or implement the LOD hierarchy using compute shaders, which are supported on the most commonly used GPUs at this point in time.[5]

[5]Or the simplest alternative of not using any LOD technique at all.

4.5.5 Error metric adjustments

In Section 3.5.2, it is mentioned that the error threshold used in the task shader to cull too coarse / too detailed nodes is set for the whole scene. While this gives the user the tool to extremely simply adjust the performance/quality, some additional information could be added to further improve or modify this error threshold. For example, the user could adjust this error threshold per object to prioritize higher quality rendering for more important objects.

Some automatic heuristics could also be employed based on the material information or structural complexity of the object. As an example, the error threshold could automatically be decreased for shiny objects, as generally the visual artifacts caused by the node simplification are more noticeable on those. Another simple heuristic would be to adjust the error threshold over time to meet some preset target framerate. Even simpler would be to also scale the error threshold with viewport resolution and field of view.

Chapter 5
Conclusion

The theoretical part of this work gave an overview of the existing techniques to increase the performance of 3D polygonal renderers. The main focus was increasing performance through the usage of *levels of detail* (*LOD*). An algorithm that uses a hierarchy of meshlets to enable rendering parts of a mesh at different resolutions while maintaining good performance and visual quality was explained. The different LOD techniques were compared.

In the practical part, an Application for meshlet rendering was designed and implemented. The Application is able to load and preprocess a 3D mesh into a *hierarchy of meshlets* which is then serializable onto the hard-drive. At runtime, the Application can take this preprocessed hierarchy and do effective culling of these meshlets based on a *view-dependent metric* to achieve increased performance without sacrificing visual quality. The Application uses the GPU API *Vulkan* to enable low-level control over the hardware. Instead of the traditional vertex shader pipeline, the Application uses mesh/task shaders for the selection and processing of the hierarchy. The selected meshlets are then rasterized and rendered onto the screen. The application is able to switch between these meshlets seamlessly without visual artifacts.

The user can navigate around the rendered 3D scene in real-time using the mouse and keyboard. The Application also provides configuration options that are available either from a configuration file, as a command line argument, or directly through a simple user interface.

The text explained the design decisions, the architecture, and the inner workings of the Application in detail. Chosen implementation details of the LOD hierarchy preprocessing and runtime handling were highlighted and provided as code snippets.

The performance of the Application was then measured using an implemented Python script. The measurements were done on 20 meshes of various complexity and nature ranging from 80 thousand triangles to 23 million triangles. The results were visualized in multiple plotted graphs.

The results showed that frame time is approximately halved on average without causing any major visual artifacts. Further investigation suggested that with a better simplification and meshlet grouping algorithm, the performance increase could be even greater. The limitations of the current implementation were critically assessed and possibilities for future improvements and extensions were outlined.

Appendix A

Application installation and usage

A.1 Installation

The application is available as a Git repository hosted on faculty GitLab here: `https://gitlab.fel.cvut.cz/smelyric/lod-o-matic`.

After cloning the repository with the command

```
git clone --recurse-submodules <the repository url>
```

the external dependencies of GLM, GLFW, Meshoptimizer, and ImGUI will also be cloned and initialized to the exact commit this application was originally compiled with.

One other needed dependency is VulkanSDK which can be accessed through `https://www.vulkan.org/tools`. The site offers the SDK for multiple operating systems. The application was compiled and tested with the version `1.3.296`, but future versions should be compatible as long as they maintain backward compatibility. Also, the SDK should contain a command line tool `glslc` which transpiles the GLSL shader files into SPIR-V – Vulkan readable files.

The Application is compiled using CMAKE, so any tool or environment that can handle CMAKE projects should now be able to compile the Application[1]. The executable binary will be called *Lod-o-Matic.exe*.

A.2 Usage

By default, upon launch, the Application will display a single *Suzanne* model near the middle of the screen.

For navigation around the scene, it is recommended to use the mouse and keyboard. The keybinds are as follows:

W/S To move the scene camera forward and backwards, respectively, in the direction of the view.

[1]Of course some kind of C++ compiler is needed.

A/D To move the scene camera left and right, respectively, in the horizontal direction perpendicular to view.

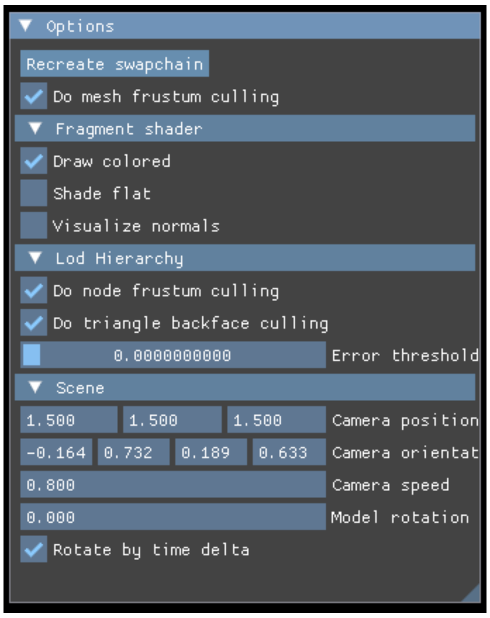
Q/S To move the scene camera up and down, respectively.

RIGHT MOUSE To look around, the user can hold the right mouse button, then by moving the mouse, the camera/view rotates around the camera origin.

MOUSE WHEEL By rotating the mouse wheel up, the speed at which the camera is moving by the previously mentioned keys increases. Rotating the mouse wheel down decreases the speed.

C Resets the *c*amera orientation to default.

K *K*ills the Application.



▼ Options

Recreate swapchain

✓ Do mesh frustum culling

▼ Fragment shader

✓ Draw colored

☐ Shade flat

☐ Visualize normals

▼ Lod Hierarchy

✓ Do node frustum culling

✓ Do triangle backface culling

0.0000000000 Error threshold

▼ Scene

1.500 1.500 1.500 Camera position

-0.164 0.732 0.189 0.633 Camera orientat

0.800 Camera speed

0.000 Model rotation

✓ Rotate by time delta

Figure A.1: Application Options

The user can influence the behavior of the Application by changing the values in *Options* in the user interface as seen in Figure A.1. More extended configuration is available through a configuration file `config.cfg` or command line arguments. The complete list of configuration options is listed in Table A.1 with their default values and descriptions. The configuration is changed by *<key>=<value>* separated by newlines for the configuration file and spaces for the command line.

Table A.1: Application configuration options

Key	Default value	Description
RunTests	true	Runs tests at the start of the Application.
PromptAfterEveryFrame	false	Needs you to press N after each rendered frame.
DoBenchmarkRun	false	Does a benchmark run which renders only a limited amount of frames before exiting.
BenchmarkFrameCount	1000	The amount of frames to render in a benchmark run.
RandomCamera	false	Positions the camera randomly each frame
RandomCameraExtent	50	The extent of the random camera position (for all axes).
InterpolateCamera	0	Over how many frames to interpolate the random camera positions.
RenderIntoFiles	false	Whether to also render each frame into a TGA file. Note that this option decreases performance drastically.
RenderedFilesDirectory	rendered	Where to save the rendered TGA images.
WindowStartMaximized	false	Whether to start the window maximized.
WindowStartFullscreen	false	Whether to start the window in fullscreen.
WindowWidth	800	The initial width of the window.
WindowHeight	600	The initial height of the window.
WindowTitle	App	The title of the window.
VulkanPrintQueueFamilies	false	Whether to print found GPU queues.

Continued on the next page...

Key	Default value	Description
MaxFramesInFLight	2	Maximum amount of submitted frames.
GPU	-1	Which GPU to use when there are multiple valid ones. Setting this to -1 prompts the user on command line.
DoBackfaceCulling	true	Whether to do triangle backface culling.
DoMeshFrustumCulling	true	Whether to do whole-mesh frustum culling.
DrawColored	true	Whether to draw the meshlets/meshes colored. Draws grey when false.
ShadeFlat	false	Whether to shade the mesh flat.
VisualizeNormals	false	Whether to render normals as color.
ViewAngle	45	The view angle of the projection.
NearPlane	0.01	Position of the near clipping plane.
FarPlane	1000	Position of the far clipping plane.
UseMeshShader	true	Whether to use mesh shaders (but not LOD hierarchy).
UseLodHierarchy	true	Whether to use mesh shaders with LOD hierarchy.
ErrorThreshold	0	The error threshold for node selection.
DoNodeFrustumCulling	true	Whether to do node frustum culling.
DoTriangleBackfaceCulling	true	Whether to do triangle backface culling in the mesh shader.
LodMaxNodesInGroup	4	Maximum amount of nodes in group.

Continued on the next page...

Key	Default value	Description
LodMinNodesToProcess	8	Minimum amount of nodes to continue processing.
LodRestartRandom	false	Whether to restart from random point or continue from last during adjacencies.
LodSimplifySingleNodes	false	Whether to try to simplify singular nodes.
ForceReserialization	false	Whether to enforce LOD hierarchy preprocessing (even when serialized lod was found).
LogKeyStrokes	false	Whether to log every keystroke to console.
LogMousePosition	false	Whether to log mouse movement to console.
ComputeMeshletStats	true	Whether to compute meshlet stats.
OutputInfo	true	Whether to output info level messages to console.
OutputStats	false	Whether to output statistics to console when exitting the Application.
OutputRuntimeHierarchyStats	false	Whether to collect runtime hierarchy stats. This decreases performance significantly.
MouseSensitivity	0.01	Camera turning sensitivity.
MovementSpeed	0.8	Initial camera movement speed.
ShowUi	true	Whether to show user interface.
ShowDemoUi	false	Whether to show sample ImGUI user interface window.
Model	suzanne.obj	Which mesh to load and display.

Continued on the next page. . .

Key	Default value	Description
ModelRotationSpeed	0	Speed of the model rotation.
RotateByTimeDelta	true	Whether to multiply the model rotation by time delta.
ModelGridSize	1	How many models in a grid (in horizontal axes) to render.
ModelGridSpacing	2	How far apart should the models be in the grid.
CameraPosX	1.5	Initial camera position on the X axis.
CameraPosY	1.5	Initial camera position on the Y axis.
CameraPosZ	1.5	Initial camera position on the Z axis.
ResourcesPath	resources	Path to OBJ files.
LoadNormals	true	Whether to load normals from the OBJ.
LoadTextureCoordinates	false	Whether to load texture coordinate from the OBJ.
NormalizeMeshSize	true	Whether to normalize the size of the mesh into [-1, 1] in all axes. Must be turned on for the error threshold to be consistent.

Appendix B

Measured data

This appendix contains only raw data. Descriptions can be found in the main text of the work.

Table B.1: Raw data from the first measurement (see Section 4.1)

First measurement					
setup	time per 1000 frames in sec.	1 - ssim to vertex ($\times10^{-5}$)	flip error to vertex ($\times10^{-5}$)	1 - ssim to prev. lod ($\times10^{-5}$)	flip error to prev. lod ($\times10^{-5}$)
armadillo					
vertex	9.0193				
lod_0	6.0663	0.00000	0.00000		
lod_1	5.1959	0.00552	0.01344	0.00552	0.01344
lod_2	2.7031	0.06064	0.07501	0.05360	0.06157
lod_3	1.2130	0.31991	0.43242	0.25127	0.35741
lod_4	0.7129	1.00410	1.86105	0.82498	1.61180
lod_5	0.6974	1.15776	2.27487	0.17306	0.41936
lod_6	0.6841	1.15776	2.27487	0.00000	0.00000
lod_7	0.6870	1.15776	2.27487	0.00000	0.00000
bmw					
vertex	6.4475				
lod_0	6.7852	0.00000	0.00000		
lod_1	5.3663	0.66854	0.59626	0.66854	0.59626
lod_2	3.5425	0.84324	0.78513	0.17452	0.25132
lod_3	2.1178	1.15698	0.90021	0.31842	0.34481
lod_4	1.7649	1.36513	1.36800	0.25123	0.45128
lod_5	1.7519	1.41562	1.82002	0.12265	0.55132
lod_6	1.7583	1.41562	1.82002	0.00000	0.00000
lod_7	1.7394	1.41562	1.82002	0.00000	0.00000
buddha					

First measurement (continued)					
setup	time per 1000 frames in sec.	1 - ssim to vertex ($\times 10^{-5}$)	flip error to vertex ($\times 10^{-5}$)	1 - ssim to prev. lod ($\times 10^{-5}$)	flip error to prev. lod ($\times 10^{-5}$)
vertex	19.5355				
lod_0	17.6522	0.00000	0.00000		
lod_1	11.8157	0.00216	0.00000	0.00219	0.00000
lod_2	6.4829	0.02601	0.02308	0.02421	0.02308
lod_3	3.8005	0.11017	0.19403	0.09422	0.17358
lod_4	2.9143	0.21291	0.26819	0.13701	0.12585
lod_5	2.9137	0.27365	0.52139	0.11480	0.26124
lod_6	2.8931	0.27365	0.52139	0.00000	0.00000
lod_7	2.8953	0.27365	0.52139	0.00000	0.00000
car					
vertex	6.7681				
lod_0	7.1542	0.00000	0.00000		
lod_1	6.0975	0.00135	0.00000	0.00135	0.00000
lod_2	4.1288	0.03912	0.03814	0.03726	0.03814
lod_3	2.8755	0.23410	0.15504	0.21289	0.13347
lod_4	2.6018	0.42653	0.49279	0.24947	0.34437
lod_5	2.5692	0.52249	0.67167	0.11131	0.17888
lod_6	2.5768	0.52249	0.67167	0.00000	0.00000
lod_7	2.5750	0.52249	0.67167	0.00000	0.00000
church					
vertex	60.6307				
lod_0	42.4796	0.00000	0.00000		
lod_1	33.4828	0.00784	0.03198	0.00784	0.03198
lod_2	17.5898	0.12181	0.36259	0.12738	0.39457
lod_3	9.4527	0.79530	2.18432	0.77432	2.07182
lod_4	7.5508	1.42559	3.68640	0.92848	2.31949
lod_5	7.5743	1.62493	4.21458	0.56162	1.22499
lod_6	7.5699	1.62493	4.21458	0.00000	0.00000
lod_7	7.5322	1.62493	4.21458	0.00000	0.00000
conference					
vertex	4.2443				
lod_0	4.9803	0.00089	0.00000		
lod_1	2.4885	0.00785	0.00000	0.00696	0.00000
lod_2	2.0311	0.17729	0.00000	0.16971	0.00000
lod_3	1.9050	0.28577	0.03569	0.34180	0.03569
lod_4	1.9221	0.42412	0.07434	0.14304	0.03866

First measurement (continued)					
setup	time per 1000 frames in sec.	1 - ssim to vertex ($\times 10^{-5}$)	flip error to vertex ($\times 10^{-5}$)	1 - ssim to prev. lod ($\times 10^{-5}$)	flip error to prev. lod ($\times 10^{-5}$)
lod_5	1.8996	0.43162	0.07434	0.04214	0.00000
lod_6	1.9280	0.43162	0.07434	0.00000	0.00000
lod_7	1.9002	0.43162	0.07434	0.00000	0.00000
crown					
vertex	70.6058				
lod_0	78.1575	0.00018	0.00000		
lod_1	40.9428	0.00126	0.00000	0.00111	0.00000
lod_2	21.5414	0.02585	0.03485	0.02518	0.03485
lod_3	14.9189	0.07755	0.07754	0.07109	0.08294
lod_4	14.5405	0.09990	0.19833	0.08806	0.20217
lod_5	14.4786	0.12294	0.31183	0.05491	0.13487
lod_6	14.4988	0.12294	0.31183	0.00000	0.00000
lod_7	14.7427	0.12294	0.31183	0.00000	0.00000
demogorgon					
vertex	71.7990				
lod_0	41.8420	0.00000	0.00000		
lod_1	32.9122	0.00842	0.00000	0.00846	0.00000
lod_2	17.9013	0.04127	0.00786	0.03461	0.00786
lod_3	9.6451	0.17885	0.15903	0.15520	0.15177
lod_4	6.8726	0.44594	0.62335	0.38332	0.54841
lod_5	6.8545	0.49743	0.82735	0.14129	0.26581
lod_6	6.8060	0.49743	0.82735	0.00000	0.00000
lod_7	7.0497	0.49743	0.82735	0.00000	0.00000
destroyed-building					
vertex	1.7673				
lod_0	1.4684	0.00000	0.00000		
lod_1	1.4814	0.00000	0.00000	0.00000	0.00000
lod_2	0.8618	0.13519	0.11007	0.13522	0.11007
lod_3	0.4160	0.39306	0.38836	0.38697	0.40542
lod_4	0.2700	0.93568	1.44103	0.82573	1.27198
lod_5	0.2844	1.03399	1.81769	0.30256	0.61165
lod_6	0.2681	1.03399	1.81769	0.00000	0.00000
lod_7	0.2841	1.03399	1.81769	0.00000	0.00000
dragon					
vertex	16.0649				
lod_0	14.4507	0.00000	0.00000		

First measurement (continued)					
setup	time per 1000 frames in sec.	1 - ssim to vertex ($\times 10^{-5}$)	flip error to vertex ($\times 10^{-5}$)	1 - ssim to prev. lod ($\times 10^{-5}$)	flip error to prev. lod ($\times 10^{-5}$)
lod_1	9.7756	0.00514	0.00000	0.00514	0.00000
lod_2	5.4401	0.04303	0.05303	0.04084	0.05303
lod_3	3.0585	0.22771	0.31724	0.18211	0.26636
lod_4	2.0615	0.52601	1.02078	0.35961	0.81614
lod_5	2.0400	0.64807	1.21765	0.12820	0.20359
lod_6	2.0381	0.63644	1.21765	0.01519	0.00000
lod_7	2.0337	0.63644	1.21765	0.00000	0.00000
fire-hydrant					
vertex	43.5083				
lod_0	51.8932	0.00001	0.00000		
lod_1	29.9826	0.01586	0.00000	0.01584	0.00000
lod_2	17.1873	0.13999	0.06950	0.11191	0.06950
lod_3	9.7831	0.45013	0.36549	0.40540	0.37934
lod_4	8.9039	0.53999	0.51040	0.18559	0.22692
lod_5	8.8979	0.55364	0.53809	0.01564	0.04140
lod_6	8.9050	0.55364	0.53809	0.00000	0.00000
lod_7	8.9858	0.55364	0.53809	0.00000	0.00000
ggm11					
vertex	217.1840				
lod_0	256.8200	0.00055	0.00000		
lod_1	129.9210	0.05301	0.01030	0.05348	0.01030
lod_2	77.7935	0.06725	0.02922	0.06866	0.01892
lod_3	51.2127	0.35439	0.13117	0.31037	0.10457
lod_4	49.3321	0.35128	0.13117	0.02225	0.00000
lod_5	49.4529	0.35128	0.13117	0.00000	0.00000
lod_6	49.3745	0.35128	0.13117	0.00000	0.00000
lod_7	49.4795	0.35128	0.13117	0.00000	0.00000
hairball					
vertex	40.8569				
lod_0	40.3641	0.00174	0.00000		
lod_1	41.0625	0.00174	0.00000	0.00000	0.00000
lod_2	21.3512	0.53769	0.88797	0.53580	0.88797
lod_3	11.0209	2.50129	5.48866	2.33432	4.99302
lod_4	10.7799	3.44592	7.80369	1.17841	2.42404
lod_5	10.7057	3.43679	7.80369	0.02517	0.00000
lod_6	10.7883	3.43679	7.80369	0.00000	0.00000

First measurement (continued)					
setup	time per 1000 frames in sec.	1 - ssim to vertex ($\times 10^{-5}$)	flip error to vertex ($\times 10^{-5}$)	1 - ssim to prev. lod ($\times 10^{-5}$)	flip error to prev. lod ($\times 10^{-5}$)
lod_7	10.8221	3.43679	7.80369	0.00000	0.00000
helicopter					
vertex	329.6850				
lod_0	386.6950	0.00378	0.00000		
lod_1	136.4950	0.00490	0.00855	0.00253	0.00855
lod_2	83.9002	0.01130	0.02321	0.01200	0.01465
lod_3	69.2237	0.02883	0.06057	0.01849	0.03736
lod_4	68.9055	0.03602	0.06057	0.00674	0.00000
lod_5	68.9250	0.03602	0.06057	0.00000	0.00000
lod_6	68.9293	0.03602	0.06057	0.00000	0.00000
lod_7	68.9244	0.03602	0.06057	0.00000	0.00000
massive-suzanne					
vertex	292.7830				
lod_0	277.7680	0.00013	0.00000		
lod_1	91.7137	0.00020	0.00000	0.00007	0.00000
lod_2	61.2992	0.00639	0.01182	0.00619	0.01182
lod_3	56.7070	0.01608	0.04755	0.00969	0.03573
lod_4	55.9219	0.02361	0.06162	0.00810	0.01636
lod_5	55.9318	0.02361	0.06162	0.00000	0.00000
lod_6	55.9986	0.02361	0.06162	0.00000	0.00000
lod_7	55.9118	0.02361	0.06162	0.00000	0.00000
monument					
vertex	101.2710				
lod_0	118.2960	0.00000	0.00000		
lod_1	70.4387	0.01039	0.00000	0.01037	0.00000
lod_2	40.4608	0.04706	0.01103	0.03400	0.01103
lod_3	23.0155	0.13918	0.05093	0.09677	0.03990
lod_4	20.7299	0.18185	0.07513	0.06253	0.02633
lod_5	20.8325	0.20105	0.10000	0.02682	0.02488
lod_6	20.7797	0.20105	0.10000	0.00000	0.00000
lod_7	20.8092	0.20105	0.10000	0.00000	0.00000
powerplant					
vertex	163.5360				
lod_0	364.9820	0.00000	0.00000		
lod_1	242.0000	0.00490	0.01741	0.00490	0.01741
lod_2	221.0780	0.02507	0.03487	0.02028	0.01747

First measurement (continued)					
setup	time per 1000 frames in sec.	1 - ssim to vertex ($\times 10^{-5}$)	flip error to vertex ($\times 10^{-5}$)	1 - ssim to prev. lod ($\times 10^{-5}$)	flip error to prev. lod ($\times 10^{-5}$)
lod_3	220.1250	0.11842	0.10763	0.10030	0.07367
lod_4	220.2540	0.19463	0.26941	0.08540	0.16554
lod_5	220.3270	0.19463	0.26941	0.00000	0.00000
lod_6	220.4010	0.19463	0.26941	0.00000	0.00000
lod_7	220.2900	0.19463	0.26941	0.00000	0.00000
roadbike					
vertex	29.9004				
lod_0	26.7265	0.00017	0.00000		
lod_1	13.7705	0.00200	0.01440	0.00203	0.01440
lod_2	7.0151	0.04763	0.16805	0.04527	0.15365
lod_3	4.4898	0.19907	0.84412	0.16159	0.75523
lod_4	4.1400	0.67196	2.68623	0.50001	2.02395
lod_5	4.1571	0.82298	3.38420	0.20553	0.87591
lod_6	4.1635	0.82159	3.35801	0.01469	0.08440
lod_7	4.1615	0.82159	3.35801	0.00000	0.00000
sibenik					
vertex	1.2862				
lod_0	1.3150	0.00000	0.00000		
lod_1	1.1721	0.00000	0.00000	0.00000	0.00000
lod_2	0.8472	0.08114	0.06137	0.08114	0.06137
lod_3	0.6495	0.14689	0.08690	0.08470	0.02552
lod_4	0.5894	0.40042	0.30018	0.32936	0.22584
lod_5	0.5988	0.40273	0.30684	0.01063	0.00881
lod_6	0.5829	0.40273	0.30684	0.00000	0.00000
lod_7	0.6041	0.40273	0.30684	0.00000	0.00000
sodahall					
vertex	39.1147				
lod_0	50.1412	0.00035	0.00000		
lod_1	41.8677	0.00036	0.00000	0.00001	0.00000
lod_2	38.7508	0.01724	0.00831	0.01696	0.00831
lod_3	38.7845	0.10414	0.07593	0.09681	0.06762
lod_4	38.6787	0.53229	0.62805	0.48111	0.56874
lod_5	38.5816	0.68946	0.79393	0.19231	0.16587
lod_6	38.6244	0.68946	0.79393	0.00000	0.00000
lod_7	38.6493	0.68946	0.79393	0.00000	0.00000

Table B.2: Raw data from the second measurement (see Section 4.2)

Second measurement					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
armadillo					
nodes: 16 644, zero nodes: 4788, root nodes: 1160, root tris: 14 212					
lod_0	0.1447	4788.00	0.00	345944.00	0.00
lod_1	0.1742	4788.04	1.00	345812.71	469.64
lod_2	0.1858	4726.10	63.84	297125.87	47214.91
lod_3	0.1441	4603.58	120.16	226878.90	61567.00
lod_4	0.1464	3423.31	591.18	111311.56	42959.50
lod_5	0.1460	2881.89	528.66	83235.09	30158.17
lod_6	0.1503	2002.97	341.03	41276.30	15037.04
lod_7	0.1524	1680.89	300.29	29135.95	11765.71
lod_8	0.1531	1236.60	149.62	15980.67	4278.05
lod_9	0.1542	1183.88	80.93	14743.24	2095.50
lod_10	0.1496	1160.79	8.27	14226.65	151.16
lod_11	0.1519	1160.08	1.29	14213.48	23.54
bmw					
nodes: 18 354, zero nodes: 6665, root nodes: 2884, root tris: 46 316					
lod_0	0.1573	6665.00	0.00	384893.00	0.00
lod_1	0.1432	6331.88	72.83	337302.86	19898.29
lod_2	0.1454	5991.64	187.55	255455.57	38939.32
lod_3	0.1488	5709.71	260.36	206756.23	44651.95
lod_4	0.1447	4654.57	478.40	124527.63	30639.97
lod_5	0.1506	4167.18	464.33	100191.65	23979.82
lod_6	0.1517	3301.14	312.90	62012.66	13515.87
lod_7	0.1520	3081.88	225.09	53330.69	9182.80
lod_8	0.1461	2900.05	59.41	46849.92	2117.95
lod_9	0.1538	2888.17	24.15	46449.14	792.91
lod_10	0.1483	2884.04	0.80	46317.42	24.99
lod_11	0.1516	2884.00	0.13	46316.10	3.29
buddha					
nodes: 51 354, zero nodes: 14 749, root nodes: 3383, root tris: 37 845					
lod_0	0.2983	14749.00	0.00	1087304.00	0.00
lod_1	0.2976	14610.68	117.59	1004540.31	79791.17
lod_2	0.2482	13276.61	1060.31	564677.71	177209.62
lod_3	0.2254	11357.11	1744.52	407745.51	156439.91

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_4	0.1783	7426.63	1465.99	200983.64	69855.42
lod_5	0.1689	6362.01	1086.84	153588.38	48554.90
lod_6	0.1663	4652.40	667.49	76371.89	27701.20
lod_7	0.1488	4024.39	586.30	55583.72	20789.96
lod_8	0.1508	3433.29	196.59	39126.56	5411.04
lod_9	0.1492	3396.30	85.62	38156.82	2109.40
lod_10	0.1473	3383.03	0.78	37846.27	19.74
lod_11	0.1583	3383.00	0.00	37845.00	0.00
car					
nodes: 18 839, zero nodes: 6726, root nodes: 3459, root tris: 70 024					
lod_0	0.1970	6726.00	0.00	420643.00	0.00
lod_1	0.1617	6708.53	16.17	413640.85	6263.66
lod_2	0.1394	6621.40	67.21	317669.60	57082.51
lod_3	0.1510	6400.94	258.42	253665.53	58940.58
lod_4	0.1574	5082.72	562.03	153965.68	36101.35
lod_5	0.1487	4547.93	522.01	128323.15	26177.14
lod_6	0.1464	3762.01	270.61	87147.12	14620.23
lod_7	0.1552	3594.33	179.53	77344.75	10231.92
lod_8	0.1491	3468.16	40.56	70540.42	2355.33
lod_9	0.1433	3461.25	15.58	70144.14	865.67
lod_10	0.1522	3458.99	0.31	70024.70	11.26
lod_11	0.1552	3459.00	0.00	70024.00	0.00
church					
nodes: 116 635, zero nodes: 34 538, root nodes: 7025, root tris: 86 547					
lod_0	0.6602	34538.00	0.00	2470327.00	0.00
lod_1	0.6177	34530.85	14.90	2467423.16	6898.81
lod_2	0.5668	33758.81	627.02	1784388.62	474261.57
lod_3	0.4892	31587.48	2777.25	1336079.72	441805.40
lod_4	0.3263	19851.47	4574.53	598570.72	252539.36
lod_5	0.2925	16199.46	3765.06	429712.60	173548.57
lod_6	0.1958	10419.45	2133.13	184133.48	83385.43
lod_7	0.2019	8574.66	1740.01	127616.39	57757.30
lod_8	0.1682	7135.43	473.98	89142.88	12036.58
lod_9	0.1529	7049.19	172.76	87074.01	3937.30
lod_10	0.1576	7025.04	0.69	86547.55	11.60

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_11	0.1785	7025.00	0.00	86547.00	0.00
conference					
nodes: 10 778, zero nodes: 5512, root nodes: 2694, root tris: 81 637					
lod_0	0.1659	5512.00	0.00	331179.00	0.00
lod_1	0.1606	3657.45	124.28	160286.71	13949.54
lod_2	0.1536	3119.76	216.15	114132.42	17855.14
lod_3	0.1569	2990.22	150.58	102161.47	13007.70
lod_4	0.1584	2800.17	69.55	86818.54	5127.90
lod_5	0.1540	2747.36	51.43	83933.18	3166.04
lod_6	0.1514	2699.42	15.78	81859.78	644.38
lod_7	0.1456	2695.45	7.35	81701.39	266.63
lod_8	0.1453	2694.01	0.39	81638.97	14.30
lod_9	0.1491	2694.00	0.09	81637.23	3.87
lod_10	0.1551	2694.00	0.00	81637.00	0.00
lod_11	0.1519	2694.00	0.00	81637.00	0.00
crown					
nodes: 223 070, zero nodes: 70 972, root nodes: 12 781, root tris: 268 585					
lod_0	1.1496	70972.00	0.00	4868924.00	0.00
lod_1	0.8906	65964.47	4005.84	2869391.21	639116.90
lod_2	0.6599	48454.35	7529.62	1586636.52	469804.37
lod_3	0.5697	40854.66	7307.73	1220503.28	367372.08
lod_4	0.3840	25077.72	6176.27	607714.43	221528.29
lod_5	0.3327	19540.56	5292.13	442661.56	165621.88
lod_6	0.2624	13675.95	1910.09	290768.16	49089.32
lod_7	0.2557	13091.09	924.43	276013.68	22446.11
lod_8	0.2568	12793.53	69.55	268890.13	1762.21
lod_9	0.2528	12783.58	20.61	268644.36	504.12
lod_10	0.2458	12781.00	0.03	268585.04	1.20
lod_11	0.2528	12781.00	0.00	268585.00	0.00
demogorgon					
nodes: 117 236, zero nodes: 34 030, root nodes: 6160, root tris: 68 784					
lod_0	0.6721	34030.00	0.00	2508606.00	0.00
lod_1	0.6183	34029.11	5.83	2508250.54	967.92
lod_2	0.5420	33058.29	813.93	1838307.76	516350.46
lod_3	0.5127	30964.56	2714.88	1373507.91	469492.32

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_4	0.3221	20045.08	4185.99	621005.16	257185.90
lod_5	0.2695	16533.61	3575.55	454374.19	172481.62
lod_6	0.1926	10796.55	2178.94	200992.15	89267.49
lod_7	0.2025	8664.89	1988.24	131952.85	68000.51
lod_8	0.1502	6378.12	737.51	73637.04	18529.32
lod_9	0.1646	6218.92	326.86	69983.45	7302.92
lod_10	0.1490	6160.20	3.91	68787.59	74.22
lod_11	0.1699	6160.00	0.00	68784.00	0.00
destroyed-building nodes: 4224, zero nodes: 1370, root nodes: 349, root tris: 5078					
lod_0	0.1756	1370.00	0.00	89932.00	0.00
lod_1	0.1791	1370.00	0.00	89932.00	0.00
lod_2	0.1587	1368.61	1.79	89916.76	34.57
lod_3	0.1550	1367.77	3.26	89522.58	1102.66
lod_4	0.1522	1270.23	86.16	53736.60	17710.92
lod_5	0.1526	1071.01	192.16	38134.69	15664.33
lod_6	0.1480	622.83	166.50	16558.34	7713.10
lod_7	0.1607	502.80	124.12	11532.43	5119.22
lod_8	0.1512	366.06	43.15	5744.05	1816.91
lod_9	0.1476	354.20	22.29	5278.33	916.87
lod_10	0.1518	349.11	1.51	5081.47	54.93
lod_11	0.1495	349.00	0.09	5078.03	0.82
dragon nodes: 41 069, zero nodes: 11 851, root nodes: 2640, root tris: 30 683					
lod_0	0.2523	11851.00	0.00	871198.00	0.00
lod_1	0.2450	11831.60	51.06	838411.00	47569.55
lod_2	0.2022	10881.73	741.74	485461.14	148035.16
lod_3	0.1806	9442.88	1327.47	352241.52	133031.61
lod_4	0.1481	6338.63	1168.83	176177.41	60555.86
lod_5	0.1603	5451.94	896.26	134153.75	42307.97
lod_6	0.1460	3958.22	578.41	67858.99	23753.85
lod_7	0.1466	3376.12	539.04	49477.99	18186.62
lod_8	0.1536	2709.67	216.20	32302.52	5480.84
lod_9	0.1501	2659.87	102.89	31125.52	2403.86
lod_10	0.1496	2640.16	2.86	30686.46	58.49

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_11	0.1477	2640.00	0.00	30683.00	0.00
fire-hydrant					
nodes: 149 271, zero nodes: 42 254, root nodes: 8459, root tris: 92 609					
lod_0	0.7737	42254.00	0.00	3043226.00	0.00
lod_1	0.7034	41355.38	679.67	2474445.50	371907.73
lod_2	0.5442	34842.64	3976.27	1329895.25	435672.49
lod_3	0.4367	29361.63	4940.64	974489.04	355622.78
lod_4	0.3187	19614.08	3574.49	483898.05	173362.64
lod_5	0.3013	16765.53	2876.89	351349.55	132078.19
lod_6	0.2371	10733.93	2277.96	147126.10	70455.96
lod_7	0.2025	9280.16	1584.87	111419.85	43523.96
lod_8	0.1921	8505.45	331.65	93566.37	7424.95
lod_9	0.2000	8468.01	107.21	92780.50	2148.17
lod_10	0.1882	8459.00	0.00	92609.00	0.00
lod_11	0.1875	8459.00	0.00	92609.00	0.00
ggm11					
nodes: 771 023, zero nodes: 222 694, root nodes: 48 758, root tris: 545 820					
lod_0	3.3159	222694.00	0.00	15373918.00	0.00
lod_1	2.6453	213433.04	7907.86	9697086.13	2727190.44
lod_2	1.8558	154654.35	27764.94	4776824.29	1831579.89
lod_3	1.5904	129309.24	24768.82	3561377.52	1266045.15
lod_4	1.1250	89717.54	14711.27	1802713.23	625672.84
lod_5	0.9589	76079.88	12872.71	1292590.00	493231.63
lod_6	0.7082	53522.60	7585.44	653959.82	206356.28
lod_7	0.6812	50309.20	4408.07	579725.69	108583.71
lod_8	0.6633	48813.48	498.76	546879.34	9953.36
lod_9	0.6759	48763.94	90.22	545927.60	1647.69
lod_10	0.6589	48758.00	0.00	545820.00	0.00
lod_11	0.6511	48758.00	0.00	545820.00	0.00
hairball					
nodes: 90 997, zero nodes: 29 747, root nodes: 13 233, root tris: 246 410					
lod_0	0.6499	29747.00	0.00	2850000.00	0.00
lod_1	0.6001	29747.00	0.00	2850000.00	0.00
lod_2	0.5977	29748.40	1.50	2844681.07	7873.94
lod_3	0.5942	29693.94	246.98	2817778.54	55567.67

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_4	0.4150	23268.88	3483.40	1549079.77	657627.50
lod_5	0.3011	19162.90	3771.14	885530.60	624610.58
lod_6	0.2052	13912.26	1778.93	298009.74	203074.10
lod_7	0.1985	13431.14	901.26	259920.31	88648.27
lod_8	0.1885	13235.99	54.36	246451.99	939.75
lod_9	0.1881	13233.00	0.03	246410.01	0.38
lod_10	0.1937	13233.00	0.00	246410.00	0.00
lod_11	0.1932	13233.00	0.00	246410.00	0.00
helicopter					
nodes: 1 165 470, zero nodes: 324 342, root nodes: 66 458, root tris: 675 594					
lod_0	4.8363	324342.00	0.00	23454798.00	0.00
lod_1	2.7292	229055.10	39202.85	7683371.31	3206091.57
lod_2	1.8838	154028.07	27040.70	3719334.80	1335829.59
lod_3	1.6661	133589.75	20571.70	2792288.45	927318.16
lod_4	1.1813	92968.73	15726.49	1322715.18	517591.87
lod_5	1.0284	79508.32	13028.23	968290.32	367365.34
lod_6	0.8877	67545.76	3932.09	697637.06	87567.71
lod_7	0.8839	66739.42	1641.35	680996.70	33612.43
lod_8	0.8952	66460.17	39.37	675632.88	684.63
lod_9	0.8863	66458.02	0.51	675594.59	18.52
lod_10	0.8713	66458.00	0.00	675594.00	0.00
lod_11	0.8858	66458.00	0.00	675594.00	0.00
massive-suzanne					
nodes: 836 677, zero nodes: 228 097, root nodes: 56 743, root tris: 592 066					
lod_0	3.4463	228097.00	0.00	16121856.00	0.00
lod_1	1.5813	128402.09	16066.02	3268756.76	809392.91
lod_2	1.2620	101804.07	10377.03	1964077.75	488453.77
lod_3	1.1300	90158.50	10839.95	1507807.87	432898.34
lod_4	0.8460	66357.82	8632.22	806018.07	239371.34
lod_5	0.7783	60866.69	5962.72	678238.50	148060.27
lod_6	0.7266	57034.33	1235.14	597573.45	25424.07
lod_7	0.7318	56812.77	445.18	593322.38	8470.28
lod_8	0.7409	56743.66	9.39	592077.05	152.90
lod_9	0.7189	56743.02	0.50	592066.38	8.66
lod_10	0.7460	56743.00	0.00	592066.00	0.00

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_11	0.7362	56743.00	0.00	592066.00	0.00
monument					
nodes: 355 297, zero nodes: 99 369, root nodes: 19 762, root tris: 211 371					
lod_0	1.6911	99369.00	0.00	7177016.00	0.00
lod_1	1.4532	98472.53	1074.06	6138919.21	1041895.17
lod_2	1.0840	82957.17	10325.26	3118677.30	1150943.70
lod_3	0.9125	69503.76	12225.18	2254294.69	881850.44
lod_4	0.6357	46186.18	8456.14	1104103.12	397045.56
lod_5	0.5721	39246.27	6839.68	791258.26	304314.34
lod_6	0.3704	24870.85	5301.60	327730.94	155250.00
lod_7	0.3325	21622.95	3599.67	251794.42	92444.57
lod_8	0.3450	19856.22	630.46	213167.88	12708.11
lod_9	0.3215	19776.71	165.01	211635.37	3002.62
lod_10	0.3382	19762.00	0.00	211371.00	0.00
lod_11	0.3292	19762.00	0.00	211371.00	0.00
powerplant					
nodes: 1 607 432, zero nodes: 909 810, root nodes: 429 813, root tris: 2 656 012					
lod_0	6.0371	909810.00	0.00	12758950.00	0.00
lod_1	4.4721	682073.48	72214.89	7312419.90	2053774.92
lod_2	3.2124	503511.61	67834.84	3649797.84	1291875.15
lod_3	2.9515	461691.05	46185.38	3044410.28	748005.97
lod_4	2.7428	433831.62	9302.33	2706464.75	109821.78
lod_5	2.7565	431383.57	3001.04	2677312.60	29106.60
lod_6	2.7577	429918.65	384.44	2657663.47	5853.12
lod_7	2.7229	429840.98	172.52	2656476.88	2716.38
lod_8	2.7298	429813.23	2.13	2656018.23	33.19
lod_9	2.7390	429813.02	0.19	2656013.42	8.80
lod_10	2.7440	429813.00	0.00	2656012.00	0.00
lod_11	2.7386	429813.00	0.00	2656012.00	0.00
roadbike					
nodes: 76 952, zero nodes: 25 359, root nodes: 3272, root tris: 73 011					
lod_0	0.4147	25359.00	0.00	1676776.00	0.00
lod_1	0.3562	21571.24	1650.96	804733.98	143176.00
lod_2	0.2628	16366.03	2239.93	510620.20	113215.98
lod_3	0.2321	13830.84	2391.60	404060.57	102147.96

Second measurement (continued)					
setup	time per 1000 frames in sec.	rendered nodes average	rendered nodes standard deviation	rendered triangles average	rendered triangles standard deviation
lod_4	0.1908	7920.68	2274.36	207476.55	72293.09
lod_5	0.1684	6152.44	1783.99	153715.81	54738.32
lod_6	0.1553	3986.48	733.12	90001.54	21201.44
lod_7	0.1589	3615.48	432.42	80337.50	11987.59
lod_8	0.1511	3304.14	98.26	73572.52	2153.14
lod_9	0.1581	3281.15	42.70	73157.82	794.66
lod_10	0.1612	3272.19	2.69	73013.24	33.61
lod_11	0.1539	3272.00	0.06	73011.02	0.51
sibenik					
nodes: 3648, zero nodes: 1620, root nodes: 969, root tris: 19 671					
lod_0	0.1689	1620.00	0.00	80125.00	0.00
lod_1	0.1671	1605.17	4.16	78765.64	412.54
lod_2	0.1653	1516.14	44.61	72001.16	3188.53
lod_3	0.1529	1464.95	52.34	67216.88	5242.36
lod_4	0.1489	1338.91	47.51	44499.24	9216.34
lod_5	0.1492	1268.34	74.67	37162.99	7600.49
lod_6	0.1532	1049.66	76.37	25703.41	3962.06
lod_7	0.1478	1006.87	52.51	22745.59	2949.95
lod_8	0.1502	970.99	10.19	19958.26	879.36
lod_9	0.1617	969.52	4.37	19748.06	374.82
lod_10	0.1529	968.99	0.23	19671.93	14.04
lod_11	0.1464	969.00	0.00	19671.00	0.00
sodahall					
nodes: 117 038, zero nodes: 77 214, root nodes: 60 056, root tris: 993 222					
lod_0	0.7230	77214.00	0.00	2169132.00	0.00
lod_1	0.6687	72216.13	1603.84	1671994.86	192228.17
lod_2	0.6321	66808.18	2368.54	1294539.10	144331.09
lod_3	0.6108	64407.04	2265.75	1182307.12	112003.39
lod_4	0.5635	61167.68	1066.50	1041856.98	45243.89
lod_5	0.5540	60748.85	590.53	1023120.60	25017.09
lod_6	0.5534	60323.59	138.75	1003236.67	5999.13
lod_7	0.5604	60198.05	116.92	998905.58	4335.71
lod_8	0.5693	60068.60	42.56	993812.61	1595.05
lod_9	0.5646	60059.58	20.22	993396.04	803.85
lod_10	0.5557	60056.04	0.60	993224.92	39.76

lod_11	0.5589	60056.00	0.06	993222.38	5.93

Table B.3: Raw data from measurement of hierarchy build times (see Section 4.3)

Hierarchy build times	
mesh	build time in sec.
armadillo	3.279
bmw	3.766
buddha	11.746
car	3.953
church	27.064
conference	1.789
crown	71.844
demogorgon	27.080
destroyed-building	0.579
dragon	7.910
fire-hydrant	45.040
ggm11	582.783
hairball	25.872
helicopter	4440.857
massive-suzanne	794.357
monument	141.453
powerplant	2770.908
roadbike	14.505
sibenik	0.542
sodahall	27.461

FAKULTA ELEKTROTECHNICKÁ
FACULTY OF ELECTRICAL ENGINEERING
Technická 2
166 27 Praha 6

DECLARATION

I, the undersigned

Student's surname, given name(s):	Smělý Richard
Personal number:	484903
Programme name:	Open Informatics

declare that I have elaborated the master’s thesis entitled

GPU-Driven LOD Rendering in Vulkan

independently, and have cited all information sources used in accordance with the Methodological Instruction on the Observance of Ethical Principles in the Preparation of University Theses and with the Framework Rules for the Use of Artificial Intelligence at CTU for Academic and Pedagogical Purposes in Bachelor’s and Continuing Master’s Programmes.

I declare that I used artificial intelligence tools during the preparation and writing of this thesis. I verified the generated content. I hereby confirm that I am aware of the fact that I am fully responsible for the contents of the thesis.

In Prague on 22.05.2025

Bc. Richard Smělý
....
student's signature

Appendix C

Bibliography

[1] David Ambroz. “MESH SHADERS”. In: Presented at the Computer Graphics 2 lecture at the Czech Technical University in Prague, 2024.

[2] Pontus Andersson et al. “FLIP: A Difference Evaluator for Alternating Images”. In: *Proc. ACM Comput. Graph. Interact. Tech.* 3.2 (Aug. 2020). DOI: 10.1145/3406183. URL: https://doi.org/10.1145/3406183.

[3] Artec Studio. *Air rescue helicopter.* URL: https://www.artec3d.com/3d-models/air-rescue-helicopter (visited on 05/20/2025).

[4] Artec Studio. *Blast furnace gas engine.* URL: https://www.artec3d.com/3d-models/ggm11 (visited on 05/20/2025).

[5] Artec Studio. *Church.* URL: https://www.artec3d.com/3d-models/church (visited on 05/20/2025).

[6] Artec Studio. *Demogorgon.* URL: https://www.artec3d.com/3d-models/demogorgon (visited on 05/20/2025).

[7] Artec Studio. *Fisherman statue.* URL: https://www.artec3d.com/3d-models/fisherman-sculpture (visited on 05/20/2025).

[8] Artec Studio. *Hydrant.* URL: https://www.artec3d.com/3d-models/hydrant-ai-photogrammetry (visited on 05/20/2025).

[9] Eric Bainville. “Memory operations”. In: *GPU Benchmarks* (Nov. 2009). URL: https://www.bealto.com/gpu-benchmarks_mem.html (visited on 05/20/2025).

[10] Blender Development Team. *Blender (Version 4.4).* Blender Foundation. 2025. URL: https://www.blender.org (visited on 05/20/2025).

[11] Rita Borgo, Paolo Cignoni, and Roberto Scopigno. “An easy-to-use visualization system for huge cultural heritage meshes”. In: Jan. 2001, pp. 121–130. DOI: 10.1145/585009.585013.

[12] Kenneth Rohde Christiansen. “The use of Imposters in Interactive 3D Graphics Systems”. In: 2005. URL: https://api.semanticscholar.org/CorpusID:6939631.

[13] Paolo Cignoni et al. “Adaptive TetraPuzzles: Eficient out-of-core construction and visualization of gigantic multiresolution polygonal models”. In: *ACM Trans. Graph.* 23 (Aug. 2004), pp. 796–803. DOI: 10.1145/1015706.1015802.

[14] Icaro L. L da Cunha and Luiz M. G. Goncalves. “An Adaptive and Hybrid Approach to Revisiting the Visibility Pipeline”. en. In: *CLEI Electronic Journal* 19 (Apr. 2016), pp. 8–8. ISSN: 0717-5000. URL: http://www.scielo.edu.uy/scielo.php?script=sci_arttext&pid=S0717-50002016000100008&nrm=iso.

[15] Ddiaz Design. *2026 Zenvo Aurora Agil*. 2025. URL: https://sketchfab.com/3d-models/2026-zenvo-aurora-agil-7659b0982c9f4550a674bc73e6d3497e (visited on 05/20/2025).

[16] Dear ImGUI Development Team. *Dear ImGUI*. 2024. URL: https://github.com/ocornut/imgui (visited on 05/20/2025).

[17] Thomas Funkhouser and Carlo Sequin. “Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments”. In: *Proc ACM SIGGRAPH 93 Conf Comput Graphics* 27 (May 2000). DOI: 10.1145/166117.166149.

[18] Thomas Funkhouser, Carlo Sequin, and Seth Teller. “Management of Large Amounts of Data in Interactive Building Walkthroughs”. In: (Mar. 2000). DOI: 10.1145/147156.147158.

[19] Michael Garland and Paul S. Heckbert. “Surface Simplification Using Quadric Error Metrics”. In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9798400708978. URL: https://doi.org/10.1145/3596711.3596727.

[20] Yingwei Ge et al. “A Novel LOD Rendering Method With Multilevel Structure-Keeping Mesh Simplification and Fast Texture Alignment for Realistic 3-D Models”. In: *IEEE Transactions on Geoscience and Remote Sensing* 62 (Sept. 2024), p. 5640519. DOI: 10.1109/TGRS.2024.3457796.

[21] GLFW Development Team. *GLFW*. 2024. URL: https://www.glfw.org/ (visited on 05/20/2025).

[22] GLM Development Team. *OpenGL Mathematics*. 2024. URL: https://github.com/g-truc/glm (visited on 05/20/2025).

[23] Hugues Hoppe. “Progressive meshes”. In: SIGGRAPH ’96. New York, NY, USA: Association for Computing Machinery, 1996. ISBN: 0897917464. DOI: 10.1145/237170.237216. URL: https://doi.org/10.1145/237170.237216.

[24] Hugues Hoppe et al. “Mesh optimization”. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’93. Anaheim, CA: Association for Computing Machinery, 1993, pp. 19–26. ISBN: 0897916018. DOI: 10.1145/166117.166119. URL: https://doi.org/10.1145/166117.166119.

[25] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[26] Mark Bo Jensen, Jeppe Revall Frisvad, and J. Andreas Bærentzen. “Performance Comparison of Meshlet Generation Strategies”. English. In: *Journal of Computer Graphics Techniques* 12.2 (2023). ISSN: 2331-7418.

[27] Arseny Kapoulkine. *Meshoptimizer*. 2024. URL: https://meshoptimizer.org/ (visited on 05/20/2025).

[28] Brian Karis, Rune Stubbe, and Graham Wihlidal. *A Deep Dive into Nanite Virtualized Geometry*. Youtube. 2021. URL: https://youtu.be/eviSykqSUUw (visited on 05/20/2025).

[29] Khronos Group. *OpenGL API*. 2024. URL: https://www.opengl.org/ (visited on 05/20/2025).

[30] Khronos Group. *Vulkan API*. 2024. URL: https://www.vulkan.org/ (visited on 05/20/2025).

[31] Christoph Kubisch. “Introduction to Turing Mesh Shaders”. In: (). URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/.

[32] Christoph Kubisch. “Mesh Shading for Vulkan”. In: (). URL: https://www.khronos.org/blog/mesh-shading-for-vulkan.

[33] Rick Lewis and Carlo H. Séquin. “Generation of 3D building models from 2D architectural plans”. In: *Computer-Aided Design* 30.10 (1998). The Soda Hall 3D mesh was generated as part of this work, pp. 765–779. URL: https://www.sciencedirect.com/science/article/pii/S0010448598000311.

[34] Michael Lujan et al. “Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server”. In: *2019 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, Feb. 2019, pp. 777–781. DOI: 10.1109/ICCNC.2019.8685588.

[35] Morgan McGuire. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data (visited on 05/20/2025).

[36] Giuseppe Modarelli et al. "Direct3D 12 mesh shader samples". In: *Microsoft Learn* (Dec. 2022).

[37] Alexander Overvoorde. *Vulkan Tutorial*. URL: `https://vulkan-tutorial.com/resources/vulkan_tutorial_en.pdf`.

[38] Federico Ponchio. "Multiresolution structures for interactive visualization of very large 3D datasets". PhD thesis. Clausthal University of Technology, 2009. ISBN: 9783940394781.

[39] Czech Technical University in Prague. *Crown*. This mesh was provided for testing by the Czech Technical University in Prague. 2025.

[40] scanforge. *Ukraine War House Destroyed Drone Scan*. 2024. URL: `https://www.cgtrader.com/free-3d-models/scanned/various/ukraine-war-house-destroyed-drone-scan-free` (visited on 05/20/2025).

[41] Ryan Schmidt. "Mesh Simplification with g3Sharp". In: *Gradientspace* (Aug. 2017). URL: `https://www.gradientspace.com/tutorials/2017/8/30/mesh-simplification` (visited on 05/20/2025).

[42] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. "Decimation of triangle meshes". In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '92. New York, NY, USA: Association for Computing Machinery, 1992, pp. 65–70. ISBN: 0897914791. DOI: `10.1145/133994.134010`. URL: `https://doi.org/10.1145/133994.134010`.

[43] SDL Development Team. *Simple DirectMedia Layer*. 2024. URL: `https://www.libsdl.org/` (visited on 05/20/2025).

[44] Stanford Computer Graphics Laboratory. *The Stanford 3D Scanning Repository*. 1996. URL: `https://graphics.stanford.edu/data/3Dscanrep/` (visited on 05/20/2025).

[45] Jerry O Talton. "A short survey of mesh simplification algorithms". In: *University of Illinois at Urbana-Champaign* (2004).

[46] Siim Tiigimagi. "What is LOD: Level of Detail". In: *3D Studio* (2023). URL: `https://3dstudio.co/3d-lod-level-of-detail/` (visited on 05/20/2025).

[47] Kim Kulling Vilmring. *Open Asset Import Library*. Sept. 2024. URL: `https://assimp.org/` (visited on 05/20/2025).

[48] "Visual quality assessment: recent developments, coding applications and future trends". In: (). URL: `https://www.nowpublishers.com/article/OpenAccessDownload/SIP-010`.

[49] W3C. *WebGPU API*. 2024. URL: `https://www.w3.org/TR/webgpu/` (visited on 05/20/2025).