TU WIEN Informatics

Schnelles Rendern hochdetailierter Geometrie in Echtzeit mit modernen GPUs

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Johannes Unterguggenberger, BSc
Matrikelnummer 00721639

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

Diese Dissertation haben begutachtet:

Marc Stamminger

Jiří Bittner

Wien, 7. März 2025

Johannes Unterguggenberger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Fast Rendering of Ultra-Detailed Geometry in Real Time on Modern GPUs

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Dipl.-Ing. Johannes Unterguggenberger, BSc
Registration Number 00721639

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer

The dissertation has been reviewed by:

Marc Stamminger	Jiří Bittner

Vienna, March 7, 2025

Johannes Unterguggenberger

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Johannes Unterguggenberger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 7. März 2025

Johannes Unterguggenberger

Acknowledgements

What does it take to do a PhD in computer graphics?

As a start, it might require a Michael Wimmer, a Peter Kán, and a Károly Zsolnai-Fehér to spark the fire of pursuing a PhD, which would not have been ignited otherwise. It might take some nice colleagues like Hiroyuki Sakai and Christian Freude to turn your workplace into a pleasant environment. Speaking of which, nice and motivated students might spark some motivation in return so that one finds meaning in the teaching duties that might be required alongside the research efforts.

Maybe a world-class supervisor like Bernhard Kerbl is required so that one stops struggling in the sea of endless research opportunities, and gets some direction and clarity. Maybe such a world-class supervisor sometimes has to go to extraordinary lengths to provide the supervision that one so direly missed.

In a more holistic view, maybe it requires a mother who still makes the right calls in bleak times when things have already gone sideways. Maybe some financial support from your parents is required during your extended studies. And money isn't only money—it means that someone believes in your course. Should you find yourself standing on shaky mental ground with your thoughts going all over the place, you'll need some new stability, clarity, and confidence. Maybe these can be found in a partner in life who wants the best for you. Maybe these can be given from unexpected sources and even by people, you don't know personally, like Horst Lüning, Gunter Dueck, and Jordan B. Peterson on YouTube. For a PhD, you need good literature. However, the most helpful literature may not be found in your own field. Maybe you need to find a book like *12 Rules for Life* [Pet18], that lets you make peace with religion, society, and yourself. Maybe you need to draw some inspiration from people like like Elon Musk to overcome procrastination: If he can build rockets that fly back to earth and land there in an upright manner as if it were normal, why shouldn't you be able to complete a PhD? Maybe you need to draw some inspiration from a game developer like Naughty Dog and see that some things are greater than the sum of their parts. Maybe you need a Jeff Cavaliere on YouTube, a Dr. Martin Gruber, and a physio like Bernhard Walter in real life to fix a damaged body.

Maybe the realization is required that life is give-and-take—and that this is true, especially at university. Maybe if you gave something to students and colleagues, you'd get something back. Maybe you'd even get very helpful co-authorships back for your papers like such by Jakob Pernsteiner and Lukas Lipp.

Maybe in my case, all the aspects mentioned above were required so that I was able to do a PhD in computer graphics, even being able to focus on my beloved field of freaking *real-time rendering* (So cool!!). And speaking of love, I am grateful beyond description to my partner in life, Theresa, who not only stood by my side through hard and stressy times but also gave me the most invaluable gift of them all: Our son Oskar has enriched my life in so many ways. With him having just turned one, I am very excited for this new chapter in my life and I'll make sure it's going to be a good one.

I am deeply grateful for everything mentioned above and for many more things that I haven't mentioned here. I will *Pursue What is Meaningful (Not What is Expedient)* [Pet18] now more than ever, so that things around me may get better and better forever into the future.

Thank you!

Kurzfassung

Graphikpipelines basierend auf Rasterisierung stellen weiterhin die Basis für das Rendering in modernen Echtzeitgraphikapplikationen dar. Mit alternativen Renderverfahren, wie hardwarebeschleunigtem Raytracing, bleibt es schwierig, 60 oder mehr Bilder pro Sekunde zu berechnen bzw. 90 oder mehr Bilder pro Sekunde, um ruckelfreie Wahrnehmung in einigen Virtual Reality (VR) Applikationen zu gewährleisten.

In den vergangenen Jahren kamen einige Trends auf, die zu hoher Geometrielast in rasterisierungsbasierten Graphikpipelines führen. Einer dieser Trends ist VR-Rendering, welches manchmal nicht nur das möglichst schnelle Rendern aus zwei Kameraperspektiven erfordert, sondern für manche Applikationen oder Konfigurationen auch das schnelle Rendern aus zusätzlichen Kameraperspektiven – alles zur Erzeugung eines einzigen Bildes. Ein weiterer Trend wurde primär von Epic Games mit der Veröffentlichung ihrer Nanite-Technologie initiiert. Sie ermöglicht es, statische 3D-Modelle zu rendern, deren geometrische Details feiner sind als die Größe eines einzelnen Pixels im final erzeugten Bild. Eine naheliegende Konsequenz daraus ist, dass Anwender auch animierte 3D-Modelle und andere Szenenobjekte in ähnlich hohem Detailgrad erwarten werden.

Diese Dissertation beschreibt neue fundamentale Methoden und Evaluierungen zum Themengebiet der hohen Geometrielast im rasterisierungsbasierten Echtzeitrendering. Sie leistet folgende Beiträge, um Qualitäts- und Performanzanforderungen moderner Echtzeitapplikationen und Spielen zu erreichen:

Wir präsentieren detaillierte Analysen zum Stand der Technik hinsichtlich "Multi-View Rendering" – dem gleichzeitigen Erzeugen von Bildern aus mehreren Blickwinkeln. Wir beschreiben unter anderem eine Pipelinekonfiguration basierend auf Compute-Shadern, die gute Kompatibilität und Performanz in einigen anspruchsvollen Konfigurationen zeigt.

Weiters beschreiben wir einen fundamentalen Algorithmus zum artefaktfreien "Culling" – dem Verwerfen von Teilen der Szene, die außerhalb des Sichtbereiches der aktuellen Kameraperspektive sind oder deren Vorderseite nicht sichtbar ist – beim Rendering von animierten 3D-Modellen, die in kleine Gruppen von Dreiecken ("Cluster") aufgeteilt wurden. Damit ermöglicht unsere Technik ähnlich präzises Culling für animierte Modelle, wie es Nanite für statische Modelle ermöglicht. Das Berechnen der einzelnen Cluster-Aumaße von animierten Modellen ist dabei im Gegensatz zu statischen 3D-Modellen nicht trivial.

Weiters beschreiben wir eine generelle Methode zum Rendern von Szenenobjekten, die parametrisch beschrieben werden können und in weiterer Folge mit ähnlich hohem geometrischen Detailgrad gerendert werden:

Nachdem ein Compute-Shader-basierter Schritt den nötigen Detailgrad bestimmt, erfolgt das Rendering entweder punktbasiert oder die Geometrie wird direkt auf der GPU mittels des Hardwaretessellators erzeugt.

In unserer Forschung berücksichtigen wir technologische Entwicklungen wie hardwarebeschleunigtes Multi-View Rendering, die neuen Task- und Mesh-Shader in Graphikpipelines, effiziente Nutzung von klassischen Shadern (wie z.B. Tessellation-Shader) und generell die effiziente Nutzung von Hardwarekonfigurationen unter Berücksichtigung der Charakteristiken von modernen Graphikprozessoren – mit dem Ziel, rasterisierungsbasiertes Echtzeitrendering in Szenarien mit hoher Geometrielast und für hochdetailierte Geometrie zu beschleunigen.

Abstract

Rasterization-based graphics pipelines are still essential for rendering today's real-time rendering applications and games. We generally see high demand for efficient rasterization-based rendering techniques. With alternative approaches, such as hardware-accelerated ray tracing, it remains challenging to render more than 60 frames per second (FPS) for many real-time applications across different GPU models, or more than 90 FPS in stereo as often demanded for a smooth experience in Virtual Reality (VR) applications.

In recent years, some trends emerged which put pressure on rasterization-based graphics pipelines with high geometry loads. One of these trends is VR rendering, which sometimes not only requires rendering a given scene faster and two times in every frame but some applications or settings require even more than two views to be rendered for the creation of one single frame. Another trend was mainly initiated by Epic Games' Nanite technology, which enables the rendering of static meshes with sub-pixel geometric detail in real time. As a consequence, skinned models and other scene objects might well be expected to be rendered in similar geometric detail, increasing the geometry load even further.

With this dissertation, we contribute fundamental methods and evaluations to high geometry-load scenarios in the context of real-time rendering using rasterization-based graphics pipelines to help reach the performance or quality requirements of modern real-time rendering applications and games:

We contribute an in-depth analysis of the state of the art in multi-view rendering and introduce geometry shader-based pipeline variants that can help to improve compatibility and performance in challenging multi-view rendering scenarios. We describe a fundamental approach for artifact-free culling when rendering animated 3D models divided into clusters for ultra-detailed geometry scenarios. With our approach, also parts of skinned models can be culled in a fine-grained manner to match Nanite's fine-grained culling of static clusters. In contrast to static meshes, finding conservative bounds for clusters of animated meshes is non-trivial, but is achieved with our approach. Finally, in order to render other scene objects—such as, e.g., items or generally shapes which can be described with a parametric function—in similar geometric detail, we describe a method to generate ultra-detailed geometry on the fly: After compute shader-based level of detail (LOD) determination, the resulting parametrically defined shapes are either rendered point-wise or geometry is generated on-chip using the hardware tessellator.

In our research, we regard new technological developments such as hardware-accelerated multi-view rendering, new task and mesh shader stages, efficient usage of classical shader stages (such as tessellation shaders), and generally efficient usage of the vast set of features, stages, and peculiarities of modern GPUs, with the goal to accelerate real-time rendering of ultra-detailed geometry.

Contents

CHAPTER 1

Introduction

Real-time rendering has a wide range of application areas. While video games are its archetypal area of use, it is generally relevant whenever 3D models or scenes shall be explored in real time, which traditionally means that one frame must be rendered so fast that 60 or more frames per second (FPS) can be calculated. While graphics processing units (GPUs) are getting faster with each new generation, new challenges have emerged in recent years which can make it hard to render scenes fast. One such challenge is rendering for virtual reality (VR), for which 60 FPS are typically not enough. 90 FPS are seen as a requirement for a user experience that provides good quality and minimizes unwanted side-effects [Vla15], while two views of the scene must be calculated within one frame—one for each eye. Another challenge is more detailed input geometry. It can arise from more detailed modeling to satisfy raised user expectations, or from scanned models (turned into triangle meshes), stored in high geometric detail.

Fast rendering of highly detailed models with rasterization-based graphics pipelines—which are still the standard tool of GPUs to render triangle models—often requires optimized techniques or rendering configurations to achieve the desired performance. This dissertation describes techniques that can be used in conjunction with rasterization-based graphics pipelines to improve rendering speed in such situations.

1.1 Motivation

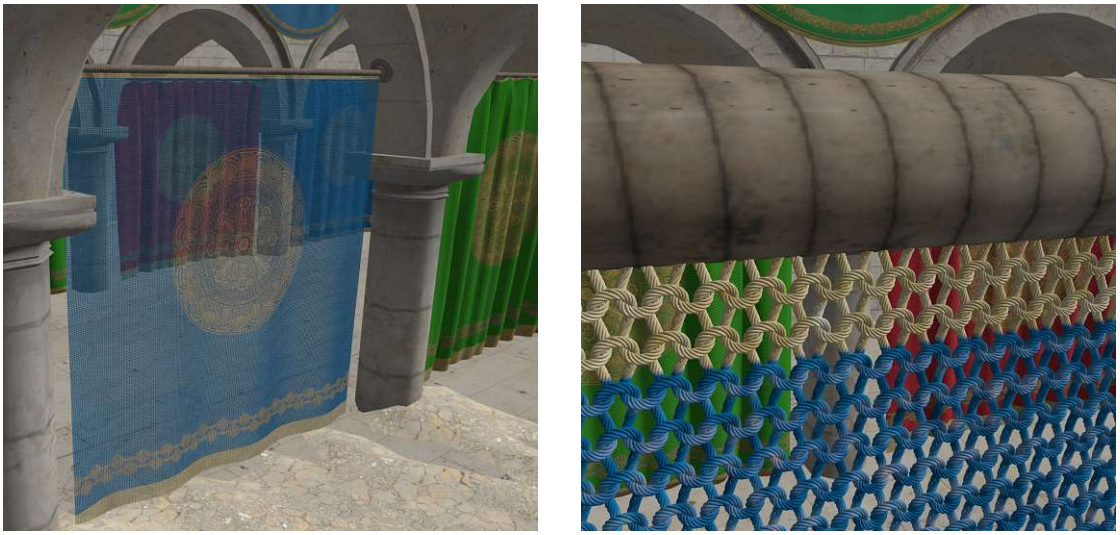
In recent years, the trend of using ultra-detailed geometry emerged in the field of real-time rendering as most prominently heralded by Epic Games' Nanite technology [KSW21]. While previously, artists were required to produce 3D models suitable for the particular requirements of real-time rendering—which typically meant limiting the polygon count—Nanite enables the usage of models with such high geometric detail that would have been considered to be infeasible to be rendered at real-time frame rates before. In a sense, these kinds of 3D models—like large scanned landscapes, detailed scans of

Figure 1.1: *Ultra-detailed landscape model "Valley of the Ancient" [Epi24b], rendered with Unreal Engine 5 using Nanite [Epi24a]. Nanite cleverly selects geometry LODs so that close to pixel-perfect geometric precision is rendered, if the original model provides enough geometric detail. This can be observed well with the smaller stones in the image.*

sculptures or buildings, or such that were modeled with ultra-high geometric detail—would still be infeasible to be rendered in real time if it were not for clever level-of-detail (LOD) approaches and efficient implementations tailored to modern GPUs, and efficiently utilizing their features.

We use the term *ultra-detailed geometry* to describe situations and setups that either take large amounts of input geometry or produce detailed geometry on the fly so that rendering produces—typically after LOD selection—geometry with approximately pixel-level detail in the rendered output. The amount of output geometry in such cases always depends on the rendering and scene setup, such as camera position, camera parameters, and screen resolution. Examples for ultra-detailed geometry are shown in Figure 1.1, which shows the case of large amounts of input geometry, and Figure 1.2 showing the case of generating geometry on the fly.

While modern GPUs are very powerful, and their computational capacities have been increasing strongly with each new GPU generation as shown in Figure 1.3, the fundamental approach of enabling the rendering of ultra-detailed geometry still is to render geometric detail only in that LOD which is for sufficient for achieving desired quality criteria in the rendered output. This typically means to cull as much of the geometry as possible with respect to the viewing frustum and camera position, and selecting a suitable geometry



(a) *A curtain model showing ultra-detailed geometric detail.*

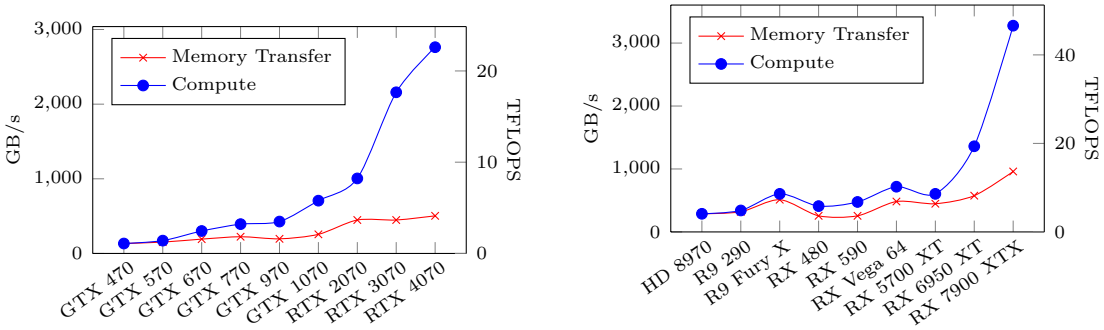
(b) *The same curtain model viewed from a closer camera position.*

Figure 1.2: *Ultra-detailed geometry that has been produced on the fly with one of our techniques [Unt+24] within a rasterization-based rendering pipeline. The curtain model shown in Figure 1.2b is comprised of many small parts—namely fiber curves shown in Figure 1.2b—producing even sub-pixel level detail in the rendering output.*

LOD for ultra-detailed geometry. The performance trends shown in Figure 1.3 suggest that trading increased computational load with decreased memory transfer load can be helpful to achieve these goals.

The focus of our research is to contribute algorithms, techniques, results, and deduced guidelines for efficient rendering in the broader field of handling high geometry loads on modern GPUs, rendered with rasterization-based graphics pipelines. To this end, we have identified three focus areas of research and present solutions for each one to help improve rendering performance in such scenarios:

1. We have conducted in-depth analyses of rendering pipeline configurations for multi-view scenarios since they often add additional high demands by rendering two or more views within one frame—and maybe even the 90 FPS requirement if this is required in a VR setup.
2. We extend fine-grained geometry culling to animated meshes—which are currently not supported by Nanite [Epi24a]—enabling partial culling of highly detailed skinned 3D models.
3. Finally, we have investigated the generation of ultra-detailed geometry on the fly through parametric functions and developed a method to render them fast, carrying forward the strategy of trading increased computational load (parametric function evaluation for roughly every pixel) with reduced memory load (the entire geometry description is given through code).



(a) *The data of the last nine generations of NVIDIA GPUs shows that the compute performance of the RTX 4070 increased to 2077% of the GTX 470, while memory transfer speed increased to only 379%.*

(b) *Comparing nine generations of AMD GPUs paints a similar picture than Figure 1.3a: While compute performance of the RX 7900 XTX rose to 5439% of the HD 8970, memory transfer speed increased to only 569% in comparison.*

Figure 1.3: *Figures 1.3a and 1.3b show the memory transfer speeds in gigabytes per second (GB/s) and the maximum compute performance in billion floating point operations per second (TFLOPS) for different GPU generations. Both axes are scaled so that they reflect the same amount of performance increase.*

1.2 Problem Statement

Rendering a sufficiently high amount of FPS in ultra-detailed geometry scenarios can be very challenging. A lot of factors need to be considered for managing to render at least 60 FPS for a typical real-time application, or at least 90 FPS for a typical VR application, requiring at least two views to be produced—one for each eye—while some effects require even more views to be produced. Furthermore, rasterization-based graphics pipelines offer a vast number of configuration options in current graphics application programming interfaces (APIs), which have been strongly extended in recent years, so that it has become a challenge by itself just maintaining an overview of all the options and features.

Challenges for creating an efficient and fast rendering setup include:

- How to configure rasterization-based graphics pipelines API-wise to achieve good rendering performance?
- How to make use of culling—which is discarding unnecessary rendering work—within rasterization-based graphics pipelines or as separate, preceding steps in frame generation?
- How can input geometry be used efficiently with rasterization-based graphics pipelines?
  - E.g., can data be shared when rendering multiple views simultaneously?
  - E.g., can we discard unnecessary rendering work upfront?
- Can geometry be generated on the fly, relieving pressure on early rendering stages within rasterization-based graphics pipelines?

High geometry loads often lead to bottlenecks when its data passes through graphics pipeline stages during rendering. For this reason, discarding work—typically called *culling*—as early as possible in the process of rendering one frame can be essential for reaching performance requirements. While applying culling to static geometry is oftentimes straightforward, and auxiliary culling information can be pre-computed—like bounding boxes for parts of the geometry—animated models add further challenges to culling decisions. With geometry possibly changing every frame, useful culling decisions—like bounding boxes—must be adapted or re-evaluated every frame, requiring solutions tailored to given rendering scenarios.

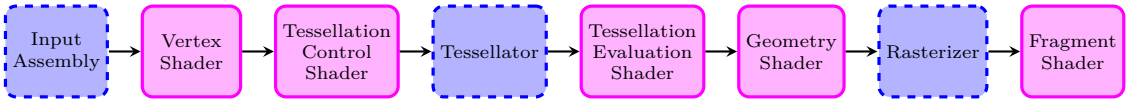
To this end, this dissertation contributes insights of fundamental research in this context, re-evaluating traditional graphics pipeline configuration options, taking new options into account—like hardware extensions or new shader stages—and describing approaches and algorithms to achieve fast rendering performance for challenging scenarios like rendering of ultra-detailed geometry.

1.3 Background

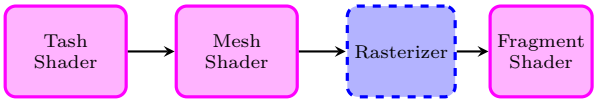
GPUs have come a long way and have undergone major changes with respect to their features over time. Some 20 years ago, the trend to deviate from fixed but fully hardware-accelerated functionality towards general programmability started with consumer-grade GPUs and consequently in the field of real-time rendering. By introducing programmable shader stages into the previously fixed-function rasterization-based graphics pipelines, countless possibilities were enabled.

Instructions are sent to GPUs through a so-called graphics API, one example of which is OpenGL [Khr22gl]. Its version 1.0 was released in 1992. Since then it has been subject to major restructurings and has seen a huge amount of new features being added. Some of the most notable feature additions are the support for hardware tessellation with OpenGL 4.0 in 2010, and the addition of compute shaders with OpenGL 4.3 in 2012 [Khr22h]. The former is an example of the addition of a modern fixed-function element—the latter is a prime example of general programmability.

A schematic view of a classical rasterization-based graphics pipeline is shown in Figure 1.4a. It shows several stages: some are mandatory, some are optional, some are programmable, and some are fixed-function. Input geometry potentially passes through all active stages and is processed in them either according to (programmed) shader code or in a fixed-function manner. From Figure 1.4a it can be seen that *graphics pipelines* have many stages and a well-defined, but also rigid structure. The other extreme in terms of stages are *compute pipelines*: They only have one single, but very flexible stage, which can perform freely defined computations without any fixed functionality. This contrast can be seen in Figure 1.4c. However, using them for the same workloads as graphics pipelines typically constitutes a performance degradation, or it requires huge efforts to achieve similar performance as classic rasterization-based graphics pipelines as shown by Kenzel



(a) *Stages of a classical graphics pipeline*



(b) *Stages of a graphics mesh pipeline*

(c) *Single stage of a compute pipeline*

Figure 1.4: *Rasterization-based graphics pipelines, as shown in Figures 1.4a and 1.4b, consist of several stages, some of which are fixed-function (colored blue with dashed border) and others are programmable through shader programs (colored magenta with solid border). Compute pipelines feature only one single stage, as shown in Figure 1.4c, and are clear of any fixed-function stages. Task and mesh shader stages, as shown in Figure 1.4b, are two compute shader-style stages within a graphics pipeline. A task shader can spawn multiple mesh shader invocations with fast data passing between those two stages. Mesh shaders can output data directly to the fixed-function hardware rasterizer.*

et al. [Ken+18]. The implementation of fixed functionality in hardware can be expected to be highly optimized, both algorithmically and also implementation-wise.

With NVIDIA's Turing microarchitecture, rasterization-based graphics pipelines were made more flexible by the introduction of so-called *task* and *mesh shaders* [NVI18a]. They completely replace all the geometry processing stages of rasterization-based graphics pipelines (vertex shaders, tessellation shaders, and geometry shaders) with two programmable stages, which are very similar to compute shaders and allow for efficient data sharing from task to mesh shader stages, as shown in Figure 1.4b. Only the stages from the rasterizer onwards remain fixed-function within such *graphics mesh pipelines*. In this case, we can observe a trend towards introducing more flexibility to replace fixed functionality.

A different trend that can be observed in modern GPUs is that highly optimized functionality is made accessible by means of extensions. Such functionality may or may not rely on the presence of additional hardware units and oftentimes has a well-defined but strict interface. From this perspective, it can be argued that this constitutes a trend opposite to introducing more flexibility, by re-introducing small pieces of fixed functionality. Hardware tessellation is a prime example of this. It requires the GPU to have a hardware unit that amplifies input geometry within rasterization-based graphics pipelines in an efficient manner: Geometry being tessellated typically stays in fast level 1 (L1) and level 2 (L2) caches and doesn't need to be transferred to global memory. Tessellation can

create large amounts of geometry "in the middle" of graphics pipeline execution, taking memory pressure off of early pipeline stages. Another example of the re-introduction of fixed functionality is the *OVR_multiview* extension [Cas18], which is supported by some GPUs of NVIDIA, ARM, Qualcomm, and Imagination Technologies [Wil24]. Its purpose is to accelerate the rendering of multiple views of a scene, where its concrete implementation is vendor-specific. One more example of fixed functionality, which has become very popular in recent years, is hardware-accelerated real-time ray tracing. It was introduced with NVIDIA's Turing microarchitecture in 2018 [NVI18a]. While some of its stages are programmable, a large part of its functionality is fixed-function and severely hardware-accelerated on some modern GPUs, enabling real-time frame rates for ray-traced games and applications.

Using the appropriate hardware features can help solve a given problem or reach performance goals. Using the appropriate algorithms or techniques can help just as much if not more. There are some actions that are beneficial in almost every case. One example is visibility culling, which means discarding parts of the geometry that will not be visible in the final rendering of a frame. Overviews of early and foundational visibility culling algorithms are given by Bittner and Wonka [BW03], and Cohen-Or et al. [Coh+03] The latter states the three different types of visibility culling:

- *View frustum culling (VFC)*: Primitives which are outside of the viewing frustum—which is defined by a camera's projection matrix, and positioned by a camera's view matrix—can be discarded.
- *Back-face culling (BFC)*: Primitives that face away from the camera—so that only their back sides are seen by a camera—can be discarded.
- *Occlusion culling*: Primitives fully occluded by other geometry can be discarded.

VFC and BFC are automatically performed by GPUs in the context of rasterization-based graphics pipelines on a per-triangle basis. However, it can be beneficial to compute VFC and BFC culling decisions manually for clusters of primitives, or VFC for entire 3D models. A simple, yet effective approach to computing the VFC culling decision is to test a bounding sphere or a bounding box against all frustum planes defined and positioned by a camera's projection and view matrices: If the bounding volume is entirely outside of *one* frustum plane, the cluster or 3d model can safely be culled. For BFC, all faces within a cluster must be facing away from the camera, so that the entire cluster can be safely culled. It is typically desirable for each culling approach to have little performance overhead and to be conservative—meaning that false negatives shall be avoided in terms of their culling decision.

With graphics mesh pipelines, it is common practice to divide geometry into small clusters, which are often called *meshlets*. Optimal division of geometry into meshlets is a research strain on its own: Meshoptimizer [Kap21] implements different strategies for meshlet building, like a vertex-cache optimized approach, or optimizing for triangle and vertex locality. Kim and Ha Lee [KL22] describe a clustering method based on normal locality,

so that deviations between normals assigned to the same meshlet are minimized. Jensen et al. [JFB23] assign triangles to a growing bounding sphere that starts at a certain vertex while striving for a minimal bounding sphere radius. They compare different meshlet generation strategies.

Another fundamental and widely used technique is using different levels of detail (LOD), which is typically complementary to other techniques, like culling, to help improve rendering speed. Which LOD of, e.g., a 3D model shall be used during rendering is typically dependent on parameters such as distance to the camera, or resulting polygon size projected to screen space. Once determined, the selected LOD is then rendered for the current frame. There are different LOD frameworks and switching strategies: LOD frameworks can be discrete, continuous, or view-dependent [Lue+02]. Switching between different levels can be done instantaneously (hard switching) or gradually (e.g., through blending) [GW07; SW08]. Also for LOD selection, different approaches have been described: A LOD can be selected statically (based on distance or on the projected area), reactively with information from the previous frame [FS93], or predictively [FS93]. Cignoni et al. [Cig+05] describe an approach for enabling gradual transitions based on a directed acyclic graph (DAG) which encodes the dependencies between surface patches of different levels of detail of the mesh. Operating on patches of triangles—instead of individual triangles—for view-dependent LOD decisions is key to achieving good performance with their GPU algorithm. This is confirmed by Ponchio [Pon09], who compared different patch-based multi-resolution frameworks for terrains, static meshes, and animated models.

One very prominent technique that combines all the above-mentioned considerations to enable ultra-detailed rendering of meshes with a huge amount of geometry in real time is Epic Games' Nanite technology [Epi24a]. It supports virtually arbitrarily high input geometry levels and achieves fast rendering speeds for so-called *micro-poly* geometry. It has become famous since it was integrated into Unreal Engine 5 and heralded a new era for artists when it was presented in 2021: They no longer need to stick to polygon budgets when creating assets for applications rendered in real time. Instead, the rendering engine takes care of handling arbitrary geometric detail in input meshes and enables rendering them with up to pixel-perfect geometric precision after level of detail (LOD) selection at cluster-granularity, and data streaming on demand. Yoon et al. [Yoo+05] have described a similar approach previously. Nanite processes detailed input models into clusters of 128 triangles each. At runtime, LOD switching in the way described by Cignoni et al. [Cig+05] enables seamless transitions between different LODs on a per-cluster basis and ensures that relevant clusters are streamed to GPU memory on demand. Clusters are selected so that rendering their triangles for a given camera position produces approximately pixel-sized triangles if the input mesh provides enough explicit detail.

Nanite employs both hardware and software rasterization. They report that the latter is up to three times faster for rendering small triangles. Only if a rendered triangle's projected edge length spans at least 32 pixels, is it rendered with the hardware rasterizer [KSW21].

Software rasterization techniques have been subject to extensive research. Frolov et al. [FGB20] compare CPU-based rasterizers. Laine and Karras [LK11] describe a software rasterization implementation that guarantees hole-free results and supports multisample anti-aliasing (MSAA). Kenzel et al. [Ken+18] show that an optimized, fully concurrent, multi-stage, load-balancing implementation of a software rasterizer can reach performance that is within one order of magnitude of the hardware graphics pipeline. In contrast to Nanite, their evaluations focus on typical game scenes, which would not qualify as micro-poly scenes. Schütz et al. [SKW22] achieve over 80 percent memory utilization with a point rendering technique that is able to render more than 2 billion points with over 60 FPS on previous-generation high-end GPU (NVIDIA RTX 3090).

To combine software rasterization and hardware rasterization into the same render target, Nanite uses a single-channel 64bit integer texture, where each entry stores depth (first 30bit), cluster index (next 27bit), and triangle index (last 7 bits). Since fragment shaders also write their results via atomic operations, their associated graphics pipeline's color output, depth test, and depth writes are disabled [KSW21]. Storing depth in the most significant bits of the written 64-bit integer values is crucial: this way, it serves as an equivalent to the depth test. While Nanite can render impressively detailed 3D meshes, even in mid-2024 the documentation of Unreal Engine 5.5 states that it is still limited to static geometry [Epi24a].

Nanite has given rise to various lines of follow-up work. Benthin and Peters have cultivated Nanite's fundamental approach of supporting micro-poly geometry for ray tracing pipelines [BP23].

An alternative to streaming geometry from memory is to generate it on the GPU—most notably by means of the hardware tessellator, which can potentially reduce memory bandwidth usage significantly. The tessellator is a specialized hardware unit that produces geometry on-chip within the context of a draw call, based on input patches [Khr23tesb]. The hardware tessellator can be used in many ways as reported by Nießner [Nie+16]. It is often used to subdivide Catmull-Clark subdivision surfaces until a certain error metric is satisfied or an ultra-detailed geometry scenario has been established [NL13; Bra+16]. Not always is this approached by using the tessellation units of a GPU. Some solutions are compute-based [SS09; PEO09] or use task and mesh shaders [Kut+23].

In the context of VR rendering, it is always required to produce at least two views. Some setups or use cases may even require the generation of more than two views, such as renderings for ultra-wide field of view (FOV) displays [NVI18b], piecewise projections for cylindrical projection volumes [LD09] or ultra-wide monitors [Rum23], or the generation of multiple shadow maps. Failure to maintain high frame rates and low latency in VR systems can not only impair visual quality but lead to motion sickness and discomfort [RK15]. For such applications, it is crucial to render as efficiently as possible and to distribute workload and utilize GPU caches optimally.

1.4 Contributions to the State of the Art

For rendering ultra-detailed geometry, it is unclear which functionality of graphics APIs and GPUs to use in which scenario—whether going for a more flexible approach or a fixed-function feature is the right choice to solve performance requirements. Furthermore, those decisions might have to be made and evaluated depending on the concrete usage scenario. In this dissertation, we provide solutions for some relevant, selected use cases and evaluate them thoroughly: Fast multi-view rendering can help to achieve sufficiently high frame rates in VR scenarios, or generally for applications that require the simultaneous rendering of multiple views of the same scene. Graphics mesh pipelines can help to enable the rendering of ultra-detailed animated 3D models, utilizing the flexible compute shader-style task and mesh stages for fine-grained culling, while still exploiting hardware-accelerated rasterization. A similar combination of flexible compute shaders and hardware-accelerated fixed functionality—but using multiple layers of compute pipeline invocations, storing and then using the results in graphics pipelines—can be beneficial for more complex application setups, such as the evaluation, on-the-fly generation, and rendering of parametric functions in real time, for which use case the two-stage nature of task and mesh shaders before the rasterizer turned out to be too inflexible.

The following sections provide more details on these selected topics and point to our respective published research.

1.4.1 Optimal Pipeline Configurations for Different Geometry Loads in Multi-View Scenarios

The impact of high geometry loads on render times when producing multiple views can be linear with the number of simultaneously rendered views in the worst case. This is an undesirable effect and can be avoided in many cases where parts of the different views' frusta overlap to some degree. The caching behavior of GPUs cannot be directly configured, but different pipeline configurations lead to different caching behavior. In our research, we have analyzed over 50 different pipeline configurations and describe their performance characteristics for many different scenarios: different numbers of simultaneously rendered views, different scenes with different geometry loads, different resolutions, and different shader loads for various use cases. Besides the meanwhile well-supported *OVR_multiview* [Cas18] extension, we were able to identify some geometry shader-based pipeline configurations that outperform *OVR_multiview* on some GPUs, tend to show favorable performance characteristics with higher geometry loads and still offer support for the tessellation stages, which *OVR_multiview* does not offer. Details, results, and findings of our research are presented in Chapter 2.

Publication: Johannes Unterguggenberger, Bernhard Kerbl, Markus Steinberger, Dieter Schmalstieg, and Michael Wimmer. "Fast Multi-View Rendering for Real-Time Applications". In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Steffen Frey, Jian Huang, and Filip Sadlo. Eurographics. online, May 2020, pp. 13–23. isbn: 978-3-03868-107-6. [Unt+20] The paper is avail-

able for download from: https://www.cg.tuwien.ac.at/research/publications/2020/unterguggenberger-2020-fmvr, DOI: 10.2312/pgv.20201071, and source code is provided on GitHub: https://github.com/cg-tuwien/FastMVR.

Individual contributions: The first author, Johannes Unterguggenberger, implemented and analyzed the different pipeline variants, and discovered the positive effects on performance by certain geometry shader-based variants, most notably those producing four views at a time. Bernhard Kerbl provided the research idea, initiated our research, and provided thorough supervision during implementation and analyses. Markus Steinberger contributed in-depth analyses of the observed performance characteristics across different GPUs. Dieter Schmalstieg and Michael Wimmer provided general guidance and feedback. Johannes Unterguggenberger, Bernhard Kerbl, and Markus Steinberger were involved in writing the paper.

1.4.2 Using a Modern, Low-Level Graphics API for Research and Teaching

While working on our paper about fast multi-view rendering [Unt+20] (see also Section 1.4.1) we used OpenGL, since it appeared to be more relevant at the time of our research and evaluations, and it provided better tooling. However, we also felt its age at some points during implementation. We were also puzzled by the absence of support for some new features that were supported in Khronos' newer graphics API Vulkan—perhaps most prominently the absence of a real-time ray tracing API in OpenGL, which is still missing today. For our papers on fine-grained culling for clustered 3D models [Unt+21] and for our work on fast rendering of parametric objects, we, therefore, decided to use the Vulkan API. We also transitioned teaching in introductory and advanced graphics courses at the Institute of Visual Computing and Human-Centered Technology at TU Wien. Our reasons for the transition from OpenGL to Vulkan, students' perceptions, and further details are described in the following two papers:

Publication: Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer. "The Road to Vulkan: Teaching Modern Low-Level APIs in Introductory Graphics Courses". In: Eurographics 2022 - Education Papers. Reims: The Eurographics Association, Apr. 2022 [UKW22]. The paper is available at: https://www.cg.tuwien.ac.at/research/publications/2022/unterguggenberger-2022-vulkan/, DOI: 10.2312/eged.20221043, and the first assignment for introductory graphics courses is available at the ACM SIGGRAPH Education Committee's cgSource: https://education.siggraph.org/cgsource/content/road-vulkan-teaching-vulkan-introductory-graphics-courses.

Publication: Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer. "Vulkan all the way: Transitioning to a modern low-level graphics API in academia". In: Computers and Graphics 111 (Apr. 2023), pp. 155–165. [UKW23]. The paper is available for download from: https://www.cg.tuwien.ac.at/research/publications/2023/unterguggenberger-2023-vaw/, DOI: 10.1016/j.cag.2023.02.001.

Individual contributions: The first author, Johannes Unterguggenberger, implemented the Vulkan frameworks, conducted the user study, and was the main author of the paper. Bernhard Kerbl and Michael Wimmer provided supervision and helped in writing.

1.4.3 Fine-Grained Conservative Culling for Ultra-Detailed Animated Meshes

While Nanite describes a sophisticated approach for rendering ultra-detailed static geometry [KSW21], it does not support animated meshes [Epi24a]. The problem with animated meshes is that their geometry—which is divided into small clusters with a Nanite-like approach—changes under animation. One key element of achieving good rendering performance for ultra-detailed geometry is fine-grained culling on a per-cluster basis. The main challenge in this context is to compute bounds for a given cluster that are reasonably tight but still conservative. Failing to compute conservative bounds leads to visual artifacts of prematurely culled clusters, which are typically very noticeable to a viewer. In our method, we analyze how clusters transform under animation and compute both spatial bounds for a cluster to enable view-frustum culling and bounds for the resulting normal directions, enabling backface culling. This enables fine-grained and conservative culling for clusters along the frustum planes, and also conservative backface culling for cases where all the triangles assigned to a certain cluster are known to be facing away from the camera. Details, results, and findings of our research are presented in Chapter 4.

Publication: Johannes Unterguggenberger, Bernhard Kerbl, Jakob Pernsteiner, and Michael Wimmer. "Conservative Meshlet Bounds for Robust Culling of Skinned Meshes". In: Computer Graphics Forum 40.7 (Oct. 2021), pp. 57–69. `issn`: 1467-8659. [Unt+21] The paper is available for download from: `https://www.cg.tuwien.ac.at/research/publications/2021/unterguggenberger-2021-msh/`, DOI: 10.1111/cgf.14401.

Individual contributions: The first author, Johannes Unterguggenberger, and Bernhard Kerbl devised the main algorithm. Johannes Unterguggenberger implemented the method in Vukan and performed performance evaluations under constant supervision by Bernhard Kerbl, who also provided the research idea. Jakob Pernsteiner implemented additional rendering effects. Michael Wimmer provided general guidance and feedback.

1.4.4 Creating Ultra-Detailed Geometry On-the-Fly for Parametric Objects

The absence of a general method to render parametrically defined objects combined with the observation of the general performance trends outlined in Figure 1.3 gave rise to our research about rendering parametric objects in ultra-high geometric detail. Such highly detailed rendering of parametric objects generally fits well with other ultra-high geometric detail approaches such as Nanite [KSW21] or our own work on fine-grained culling for ultra-detailed animated meshes [Unt+21]. A parametric object's surface is described only with a parametric function, which is a very compact description that

requires virtually no memory initially and is at a later point in the rendering process turned into a high geometry load on-the-fly, just before being rendered. Since we describe a general method, it is able to handle a variety of different types of parametric functions, such as ones describing seashell surfaces, plain-knit yarn curves, or spherical harmonics (SH) glyphs. While for many of these parametric objects, no published method exists that could render them at high frame rates, our method outperforms a state-of-the-art SH glyph method substantially, producing better quality and more FPS for higher-order SH glyphs. Details, results, and findings of our research are presented in Chapter 5.

Publication: Johannes Unterguggenberger, Lukas Lipp, Michael Wimmer, Bernhard Kerbl, and Markus Schütz. "Fast Rendering of Parametric Objects on Modern GPUs". In: Eurographics Symposium on Parallel Graphics and Visualization. Ed. by Guido Reina and Silvio Rizzi. The Eurographics Association, 2024 [Unt+24]. The paper is available for download from: `https://www.cg.tuwien.ac.at/research/publications/2024/unterguggenberger-2024-fropo/`, DOI: 10.2312/pgv.20241129, the source code is available on GitHub: `https://github.com/cg-tuwien/FastRenderingOfParametricObjects`.

Individual contributions: The first author, Johannes Unterguggenberger, and Markus Schütz devised the main algorithm. Johannes Unterguggenberger created the Vulkan implementation and performed the performance evaluations. Markus Schütz provided the research idea and constant supervision. Lukas Lipp assisted with the implementation of SH visualization. All authors were involved in writing the paper.

CHAPTER 2

Fast Multi-View Rendering for Real-Time Applications

The contents of this chapter are largely based on our paper "Fast Multi-View Rendering for Real-Time Applications", presented at the Eurographics Symposium on Parallel Graphics and Visualization 2020 [Unt+20].

2.1 Motivation

In real-time rendering, some effects require the rendering of multiple views of the same scene. For example, for VR applications and games, at least two views are required: one for each eye. Some setups require even more views to be rendered, also some algorithms—like shadow mapping for many light sources—might require multiple views of the same scene to be rendered efficiently in order to satisfy the performance goals of 90 FPS.

Modern GPUs offer a wide range of configuration options for graphics pipelines, and they have many stages as illustrated in Figure 1.4a. In some programmable stages, culling mechanisms—such as VFC and BFC as described in Section 1.3—can be implemented to preempt hardware culling. Previous work uses some of these approaches to describe and propose certain configurations, but an in-depth analysis and evaluation of the most relevant options was still lacking. In this chapter, we perform these analyses and evaluations and come up with guidelines for implementers. This is valuable in ultra-high geometry scenarios, where it is typically even more challenging to reach high numbers of FPS, especially in a multi-view rendering scenario.

2.2 Introduction

Consumer-grade head-mounted displays (HMDs) have become popular for VR in recent years, and new VR games are being released regularly. In 2020, Valve's Steam Store already listed more than 4000 games tagged as "VR Only" [Val03]. Inherent to VR games are increased requirements on the rendering performance of a PC or gaming console because every frame has to be rendered at least twice—i.e., at least once for each eye—with view positions slightly offset. However, two views might not be sufficient for an HMD with a wide field of view and non-coplanar displays[BS18]. Four or more views can be required for such setups.

Efficient rendering of multiple views does not only have its applications in VR rendering or in rendering for multi-monitor/multi-projector setups. Multiple ID buffers containing primitive IDs can be evaluated in order to determine which primitives are visible from a range of viewpoints, i.e., a *potentially visible set* (PVS). Such a PVS can be used to e.g. shade all triangles which may become visible under head movement [Mue+18]. Another application scenario is shadow mapping for multiple light sources. Each light source represents the origin of at least one view frustum that corresponds to the region that is illuminated by that light. For omnidirectional lights and algorithms like cascaded shadow mapping [Dim07], multiple views must be rendered per light source.

Producing multiple views per frame while maintaining frame rates of at least 60Hz can be challenging. Rendering effort depends heavily on the scene representation and the GPU that renders the scene. For real-time VR applications, usually, the requirements are even higher. Vlachos [Vla16] recommends staying below 10ms time per frame to achieve stable frame rates at 90Hz. In general, it can be stated that there is a need for multi-view rendering (MVR) techniques that enable efficient processing of several viewpoints and are versatile enough to be used for arbitrary scene setups and across different GPUs.

Hardware manufacturers such as Oculus [Eve16] and NVIDIA [NVI18b] have shown increased interest in the efficient rendering of multiple views. Hardware-accelerated MVR is commonly exposed as an extension for existing graphics APIs. For OpenGL and OpenGL ES, the extension is called *OVR_multiview* [Cas18] and has been implemented by NVIDIA, ARM, Qualcomm, and Imagination Technologies. Hardware-accelerated MVR is likely to outperform any other approach for MVR, including software techniques that make intelligent use of shader programs to minimize the number of draw calls and memory transfer [Wil15; DNS10].

However, to the best of our knowledge, there is little information available that lets one quantify the actual benefits of using one MVR method over another on recent GPU models. An additional caveat for desktop systems is that hardware-accelerated MVR on consumer-grade NVIDIA GPUs is limited in its applicability to real-time graphics, due to its lack of geometry shader and tessellation shader support as can be seen though the low number of devices supporting *GL_EXT_multiview_tessellation_geometry_shader* [Wil24]. A detailed analysis of available MVR methods with modern graphics APIs would enable

developers to make informed choices in their design without resorting to a trial-and-error process.

To gain clarity about the performance of hardware-accelerated MVR and software-based methods for GPUs, we provide an exhaustive evaluation of various techniques for rendering with multiple viewpoints in different scenes on a range of recent GPU models, for three distinct MVR applications. Specifically, we evaluate relevant MVR methods in the context of ID buffer generation for PVS, light-field G-buffer rendering, and shadow mapping. For our performance tests, we implement and test more than 50 different rasterization pipeline configurations, including the techniques of Sorbier et al. [DNS10], Wilson [Wil15], Vlachos [Vla15], different variants of hardware-accelerated MVR pipelines, and entirely new variants. To facilitate the identification and imparting of individual methods, we introduce a formalized syntax to describe custom MVR pipelines. In summary, our contributions include the following:

- We introduce a symbol-based description language to declare specific pipeline configurations for MVR in a concise manner.
- We examine the emergent performance characteristics of available hardware-accelerated MVR and compare them to other pipeline variants, including previously published techniques.
- We analyze and interpret performance trends for the most relevant MVR pipeline variants across different GPUs and scenes. In comparison to previous work, we also consider much larger configurations with up to 32 simultaneously rendered views.
- We describe two optimized, geometry shader-based MVR variants and identify applications where they can be used as viable alternatives to hardware-accelerated MVR. In contrast to the latter, these general variants preserve full support for custom tessellation and geometry shader routines on consumer-grade devices.

In the following, we summarize related work and previous efforts to achieve efficient MVR in hardware and software (Section 2.3). In Section 2.4, we introduce our symbol-based parameter syntax for describing different pipeline variants that are suitable for MVR. Our setup and full evaluation, along with obtained results, are described in Section 2.5. We analyze emergent performance trends and give interpretations, as well as additional important insights in Section 2.6. Summary and outlook are provided in Section 2.7.

2.3 Related Work

A considerable body of previous work has addressed the problem of multi-view rendering in computer graphics. As long as view positions only change in terms of rotation, textured impostor rectangles can be used as stand-ins for actually transformed scene geometry[SS96]. Halle et al. present an alternative scene representation and rendering algorithm that enables significant speedup of view-dependent computations by enforcing restrictions w.r.t. the discrepancies between views[Hal98]. Specifically, all camera positions must lie on a single translational axis along which views can be sampled. Sitthi-Armon et al.[Sit+08] describe how to make use of reprojection to avoid shading computations

for slightly differing views by using cached results from previous frames. A particular application of *decoupled sampling* is the smooth generation of visibility for multiple new views, yielding superior results to caching approaches[Rag+11].

Beyond straightforward implementations, there are several aspects of image synthesis with MVR that bear potential for optimization. Adelson et al.[Ade+91] provide a detailed analysis of this topic in the context of stereoscopic projections. The authors propose several methods to avoid duplicate attribute computations, efficiently cull geometry that is invisible to both eyes, and resolve visibility by combining Z-buffers with BSP trees for depth testing. Based on these ideas, several methods and mechanisms for modern graphics APIs have been proposed to improve the performance of MVR over simple multi-pass rendering. Marbach [Mar09], as well as Beck et al.[BSF10], provide basic evaluations on the benefits of geometry shaders and layered rendering for MVR, with mixed results. The techniques of Marbach [Mar09] and Sorbier et al. [DNS10] have in common that they aim to reduce driver overhead and increase GPU utilization by supplying all active views with a single draw call: rendered geometry is amplified in a geometry shader loop. Each view's pixel values are written to a separate layer of an array texture [Mar09] or to an exclusive region in a single texture, where the single texture contains all views to be rendered [DNS10]. An aspect of the technique by Sorbier et al. is that culling and clipping cannot be performed implicitly by the rasterizer, which the authors address by discarding all "out of bounds" writes in the fragment shader.

A more recent approach by Wilson [Wil15] also relies on the single-texture approach, but uses instanced rendering to achieve geometry amplification. Furthermore, they define custom clip planes in the vertex shader to avoid out-of-bounds writes, thus saving on potentially expensive fragment discards. To achieve efficient MVR with point type primitives, Marrs et al. [MWH18] avoid the rendering pipeline altogether and use compute shaders instead. Unfortunately, previous work on software GPU rasterizers has shown that similar performance gains cannot be expected for triangle meshes [Ken+18].

Hasselgren et al.[HA06] have conceived and simulated their prototype of a complete VR-oriented architecture that aims to maximize exploitation of coherence between views. Starting with the Pascal microarchitecture, NVIDIA has added built-in hardware support for MVR that is exposed in VRWorks [NVI18b] and OpenGL by the Oculus Virtual Reality (OVR) multi-view extension [Cas18]. Driven by the need for fast stereoscopic projection in VR, the *Single Pass Stereo* functionality optimizes rendering to two separate viewpoints. With the Turing microarchitecture, NVIDIA has further expanded on this feature set by adding support for accelerated rendering of up to four separate viewpoints[NVI18a].

Recently, streaming rendering techniques for VR have been proposed [Mue+18; HSS19a]. Inspired by early work on optimizing VR applications pioneered by Regan et al.[RP94], these approaches require the computation of a potentially visible set (PVS) of geometry to be shaded on a server and then streamed for framerate upsampling to a client, e.g., a head-mounted display. For PVS computations, these approaches render four to eight frames along the predicted head movement, leading to a typical MVR problem: generating multiple primitive ID buffers quickly. As an alternative to sampled visibility,

Hladky et.al [HSS19b] proposed a conservative single-pass PVS computation. While this avoids MVR, it requires up to a hundred ms for typical scenes, raising the question of whether efficient MVR rendering may not be a better solution to the problem.

2.4 Classification

In order to exhaustively analyze the properties of different MVR techniques and discuss their mechanics, we first establish a method classification catalog that enables us to capture all relevant properties with a compact, intuitive representation. To this end, we introduce a formal notation to represent an arbitrary MVR technique that processes N different views as a pipeline function $\mathcal{P}(\ldots)$ whose parameters define its implementation specifics. We propose a parameter set that is based on the variety of pipelines presented in previous work, as well as additional attributes that we found to facilitate their classification in practice during our experiments. In our current model, we consider four essential properties:

- **Pipeline invocation count:** The number of times the pipeline must be run from start to finish in order to process all N views.
- **Geometry amplification:** The mechanism used for producing sufficient copies of the input geometry to provide each view.
- **Custom culling:** Required or supplemental steps included in the pipeline to perform culling and/or clipping of triangle primitives.
- **Framebuffer layout:** The layout and configuration for the framebuffer object that the fragment shader writes its output to.

In the following, we elaborate on the significance of each individual parameter and its effects on technique configuration. In addition, we provide illustrations of selected pipeline examples. The list of symbols used to describe the properties of specific configurations are listed in Table 2.1.

2.4.1 Pipeline Invocation Count

When executing an MVR pipeline, it may be desired or necessary to run the entire pipeline multiple times. Let us consider the most straightforward method to achieve MVR, which is to invoke multiple draw calls that write the result for each of the N different views to a separate target texture. In this case, the graphics pipeline must run from start to end N times per scene entity to produce N views. Figure 2.1a illustrates this basic pipeline setup and the necessary steps for each invocation.

Running the full pipeline multiple times may also be required to circumvent hardware restrictions for specific techniques. For instance, the OVR extension enables efficient hardware acceleration on NVIDIA Turing models only when using four target views or fewer [NVI18a]. One way to evaluate hardware-accelerated MVR for a larger number of views is thus to split the N views into groups of four and invoke the entire pipeline

Table 2.1: *List of symbols used for describing the different configuration variants of graphics pipelines.*

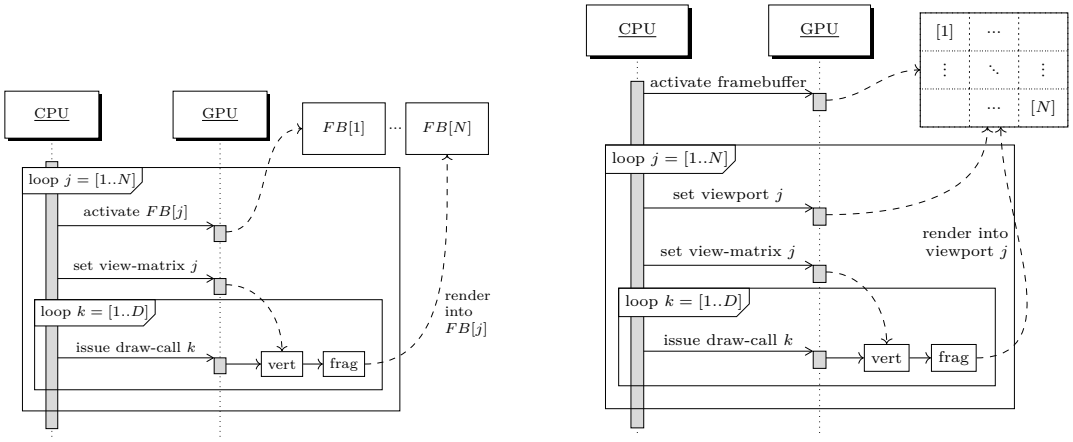
List of Symbols	
Geometry Amplification	
↦	Direct forwarding
⇶	Amplification by instanced rendering
↬	Amplification by geometry shader loop
⇉	Amplification by geometry shader instancing
≋	(Accelerated) OVR geometry amplification
Framebuffer Layout	
▭▭	Separate framebuffer objects
▦	Single large, partitioned framebuffer
▣	Layered framebuffer
▭▦	Multiple partitioned framebuffers
▣▣	Multiple layered framebuffers
Culling & Clipping	
$\texttt{CLIP}_{\texttt{VP}}$	Clipping with reduced viewport
$\texttt{CLIP}_{\texttt{\|\|}}$	Clipping with clip planes
$\texttt{CLIP}_{\texttt{FS}}$	Clipping in fragment shader
$\texttt{VFC}_{\texttt{GS}}$	Frustum culling in geometry shader
$\texttt{BFC}_{\texttt{GS}}$	Backface culling in geometry shader

multiple times. We indicate such an approach by setting the first parameter of a pipeline function to $\lceil \frac{N}{4} \rceil$. On the NVIDIA Pascal microarchitecture, only two views can be hardware-accelerated [NVI16], therefore, e.g., a test with $\lceil \frac{N}{2} \rceil$ invocations would be of particular interest in such a scenario.

2.4.2 Geometry Amplification

The key requirement for achieving MVR is the amplification of the input geometry, cuing the rasterizer to render multiple instances of each primitive—once for each view. Modern graphics APIs offer various ways to achieve this amplification at different access points in the pipeline. The choice of access point affects the quantity and nature of work that the GPU must handle. The earlier amplification occurs, the more stages must process an amplified amount of data.

At its earliest, geometry amplification can be done at the very beginning of a rasterization pipeline, which effectively means invoking the pipeline multiple times, each time with a different viewpoint location. In this case, all geometry that enters the pipeline is simply forwarded. We indicate this behavior by the ↦ symbol in the pipeline's geometry amplification parameter field. The simple pipelines in Figure 2.1 both use this setting.



(a) *Sequence diagram showing the processing of multiple multi-view draw calls with the pipeline configuration* $\mathcal{P}(N, \mapsto, \square\cdots\square)$

(b) *Sequence diagram showing the processing of multiple multi-view draw calls with the pipeline configuration* $\mathcal{P}(N, \mapsto, \boxplus, \texttt{CLIP}_{VP})$

Figure 2.1: *Examples of MVR configurations corresponding to our definition syntax. (a) A straightforward MVR pipeline uses N invocations to write each view into a different framebuffer $FB[1] \dots FB[N]$. (b) A multi-pass variant with a single, partitioned framebuffer and varying viewports for clipping.*

In order to reduce the number of draw calls without losing any flexibility, API calls that perform *instanced rendering* (signified by $\Rightarrow$) can be used instead.

For MVR, only view-dependent computations need to be duplicated. By moving the amplification to the end of the geometry stage, we can thus avoid redundant invocations of vertex shaders that, e.g., compute skeletal animation, which is uniform across all views in a given frame. In the geometry shader, output primitives can either be emitted in a loop ($\mapsto\!\!\!\!\!\!\Rightarrow$) or, if the number of duplicates is fixed, via geometry shader instancing ($\mapsto\!\!\!\!\Rightarrow$).

Finally, specific extensions for MVR have been added to rendering APIs. A powerful example is the OVR extension: on modern NVIDIA GPUs, it enables hardware-accelerated geometry amplification, which exploits re-usability of shading results across multiple views. On architectures that have no built-in support, the OVR functionality will usually fall back to a looping behavior. We indicate this type of geometry amplification with the $\rightrightarrows$ symbol. The declared target application for this functionality is stereoscopic rendering for VR, where the majority of geometry computations and visibility tests are valid for both eyes [NVI18b]. Unfortunately, using the OVR extension also prohibits the use of any custom tessellation or geometry shaders on all consumer-grade GPUs [Wil24].

Note that for all amplification methods, the number of copies generated is implicitly given by the number of times a pipeline is run. If it is called only once, geometry amplification must generate N copies. For $\lceil \frac{N}{4} \rceil$, each run must amplify the input geometry by a factor of up to $\times 4$. A pipeline that does not generate sufficient geometry for all N desired views is not valid in our definition.

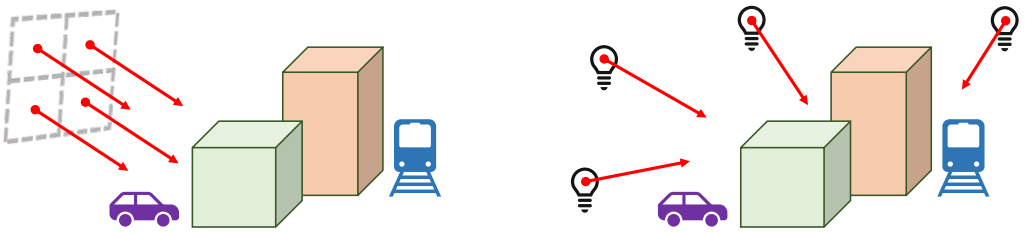
2.4.3 Custom Culling and Clipping

Some MVR techniques require additional steps after geometry amplification and before storing each view's result into its target framebuffer. When using a partitioned framebuffer, i.e., a framebuffer that contains multiple views, we must ensure that triangles are adequately clipped against the current target region and do not protrude into regions that correspond to a different view. This can be achieved in several ways: For one, graphics APIs provide methods for reconfiguring the viewport against which culling and clipping are performed between draw calls. We indicate that this feature is being used by $\texttt{CLIP}_{\texttt{VP}}$. Alternatively, we can avoid this additional API command and implied synchronization dependencies by defining custom clip planes ($\texttt{CLIP}_{||}$) in the vertex shader instead [Wil15]. A third method exploits the `discard` instruction in the fragment shader ($\texttt{CLIP}_{\texttt{FS}}$) to achieve correct clipping[DNS10]. In addition to purely functional clipping, we also consider the impacts of performing fine-grained view frustum culling and backface culling in the geometry shader to reduce its output, which we denote with the symbols $\texttt{VFC}_{\texttt{GS}}$ and $\texttt{BFC}_{\texttt{GS}}$, respectively. If multiple methods are used in the same pipeline, they are concatenated by the | symbol.

2.4.4 Framebuffer Layout

There are several possible choices w.r.t. the layout for storing the collective results generated for all N processed views. In our cases, we consider all framebuffer objects to contain at least one depth buffer and an arbitrary number of color targets. In the simplest case, N separate framebuffer objects and associated textures are allocated and each one is bound directly before rendering a particular view. We indicate this layout with the □·□ symbol, which is also used to describe the simple pipeline setup in Figure 2.1a. Note that this layout is extremely restrictive, as it implies that no API- or hardware-backed geometry amplification can be used since framebuffer bindings cannot be changed while a graphics pipeline runs. Consequently, a common suggestion in previous work is to use a single large framebuffer object instead of multiple smaller ones, and specify the target write window for each individual view [DNS10; Wil15]. We use the ⊞ symbol to represent such a pipeline configuration. Since ⊞ pipeline configurations eventually contain multiple view-results in one single framebuffer, one of the clipping methods described in Section 2.4.3 is indispensable for producing correct results. A third option for the framebuffer layout is to exploit layered rendering capabilities to write each view's content to a separate layer of an array texture [Mar09]. When an array texture is part of the framebuffer object, the rasterizer is executed in a special mode that allows setting the built-in layer ID that each primitive is assigned to. A layered framebuffer is indicated by the ❑ symbol.

As already noted in Section 2.4.1, there are cases where we would like to partition the rendering of N views into chunks of, e.g., four views each and, as a consequence, issue $\lceil \frac{N}{4} \rceil$ draw calls per scene entity to produce the entirety of N view results. For such variants, we are using either multiple separate framebuffer objects of the principal ⊞ type or multiple separate framebuffer objects of the principal ❑ type. To indicate a



(a) *Rendering four ID buffer / G-buffer views* **(b)** *Rendering four shadow map views*

Figure 2.2: *Different configurations for a single four-view setup in our applications. (a) To generate ID buffers for a PVS, we sample a rectangle to obtain visibility information under small camera motion as done, e.g., by [Mue+18]. The same samples can be used for generating a G-buffer for a small lightfield. (b) For shadow maps, samples are arbitrarily distributed, since light source positions are generally independent.*

set of conjugate framebuffer objects, we use the symbol. To indicate a set of conjugate framebuffer objects, we use the symbol.

2.5 Evaluation

In order to thoroughly evaluate and identify the conditions that influence an MVR technique's performance, we have collected timing results for several scenes, GPU models, and setup configurations in three different applications. We have evaluated more than 50 MVR variants in the context of ID buffer rendering for PVS computation, G-buffer generation for lightfields, and shadow mapping (see Figure 2.2). For rendering a single ID buffer, we use multiple viewpoints on the surface of a rectangle to sample the scene visibility. For the lightfield G-buffer, we use the same sample positions as for ID buffer rendering but set multiple color targets to store all fragment shader outputs. With shadow mapping, the discrepancy between viewpoints in MVR is random, since the positions of light sources in a scene are mostly independent. We evaluate our MVR applications at up to 100 PVS/light source origins and orientations which are uniformly distributed in each scene. Each application's run time is recorded for a varying number of target views in 6 scenes listed in Table 2.2. In contrast to most previous work, our evaluation also considers large MVR setups and ranges from 2 to 32 simultaneous target views. We consider two common framebuffer resolutions for LQ/HQ purposes: 800×600 and $1080p$.

For our evaluation, we have implemented a testbed that enables users to quickly define and run a wide range of MVR techniques. In our implementation, a particular MVR variant is configured in the source code. Different variants can be easily composed of predefined components. For example, vertex and fragment shaders in the same category have the same inputs and outputs. A geometry shader can be added in-between since they are implemented so that their inputs and outputs are compatible with the outputs

Table 2.2: *Scenes used to generate test results, along with geometry properties and the usual range of draw calls needed to produce one view of each scene. Due to frustum culling, the number of draw calls varies depending on the active view and pipeline configuration.*

	#vertices	#triangles	avg. #drawcalls
Bistro	2.52M	2.83M	43–71
Gallery	0.65M	1.00M	29–48
Robot Lab	0.38M	0.47M	46–111
San Miguel	9.02M	9.98M	866–1446
Sponza	0.29M	0.44M	84–137
Viking Village	2.87M	4.26M	195–384

of vertex shaders and the inputs of fragment shaders. Based on the other configuration parameters of a certain variant, our implementation automatically binds the appropriate buffers and render targets, and issues the proper number of draw calls. □-□ variants, for example, get a new frame buffer bound before each draw call, while for ⊞ or ⧉ variants, only one framebuffer might have to be bound before issuing all draw calls.

When we conducted our research, we decided to use OpenGL 4.6 over Vulkan as the target rendering API, for two reasons: First, Vulkan had not yet fully penetrated the industry. Therefore, results obtained with OpenGL better reflected the expected impact in graphics applications at that time. Second, several helpful tools that allow for in-depth analysis (such as the Nsight Graphics range profiler) were incompatible with the Vulkan API [NVI20]. Furthermore, given that our test applications do not show high CPU workload or rely on complex input resource management, we expect deviations to be minor. We have recorded our results for the following GPUs: NVIDIA's GTX 980, GTX 1060, GTX 1650 SUPER, RTX 2080, RTX 2800 Ti, and AMD's RX 580. Scenes undergo CPU-side frustum culling and have backface culling enabled in the rendering API. While we performed the measurements on different machines, our timings record only the portion of the GPU-side frame time required for rendering all N views. Before timing, we ran a warmup phase of 15 frames. For a single measurement, we uploaded the required resources to the GPU, waited for the completion of previous commands, and recorded the multi-view rendering time using GPU timer queries. For evaluation, we consider the average frame times across all measurements per configuration. Due to the vast size of the parameter space for this problem, we must restrict our evaluation to cases that are of particular interest. In the following, we first identify a subset of the most robust techniques out of the possible combinations that result from the parameters in Section 2.4. We consider ID buffer rendering as our base use case since it requires minimal effort w.r.t. to vertex and fragment shading (i.e., it generates a single integer output) and should enable hardware-accelerated techniques to achieve peak performance due to the strong correlation of visible geometry between views [BS18]. This subset of techniques is then further refined to yield the most promising ones, for which we analyze trends and explore the impact of changing load parameters by applying them to G-buffer rendering (three vector-valued outputs) and shadow mapping (depth-only).

2.5.1 Identifying Robust MVR Techniques

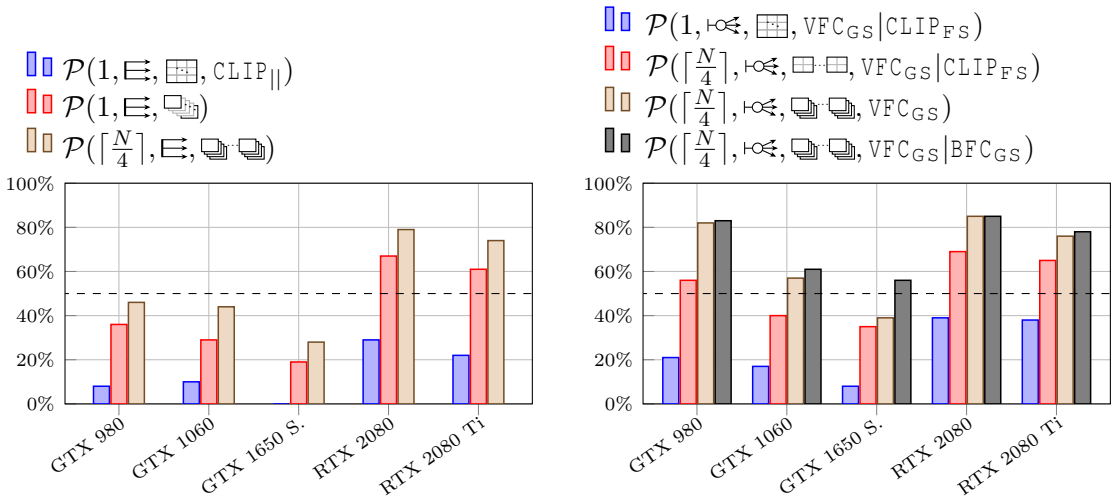
For the sake of brevity, we do not include results for all techniques that were part of our initial experiments. Instead, we provide a comparison of techniques that are based on previous work, as well as hardware-accelerated variants and promising combinations of API features that have not been proposed before. In order to identify the top-performing pipeline variants per category (where the categories are described in Sections 2.5.1 to 2.5.1), we compare them to the simplest possible MVR baseline: $\mathcal{P}(N, \rightarrow, \square\square)$, which describes a pipeline that requires N pipeline invocations, performs simple geometry forwarding, and renders the results into separate framebuffer objects. A technique is considered robust on a given GPU if it performs faster than our baseline in at least 50% of all ID buffer rendering setups, which include different scenes, view counts, and framebuffer resolutions. Note that we skip this comparison for the AMD RX580 entirely; no MVR technique performed significantly better than $\mathcal{P}(N, \rightarrow, \square\square)$ on that particular GPU model.

Instanced Rendering

Based on the method proposed by Wilson [Wil15], $\mathcal{P}(1, \rightrightarrows, \boxplus, \texttt{CLIP}_{||})$ describes an MVR pipeline that uses a single pass and instanced rendering for geometry amplification, forwarding the output to a large, partitioned framebuffer. To avoid writing outside each view's bounds, custom vertex clip planes are used, which allows skipping the geometry shader stage altogether. While testing this approach in our setup, we found that its overall performance can be improved by using a layered framebuffer instead of a partitioned one. Note that in this case, we must set the layer ID for each primitive, which theoretically requires the presence of a geometry shader. However, for such constant-time efforts, modern NVIDIA GPUs support *pass-through* geometry shaders, which almost completely avoid the overhead caused by this stage. The $\mathcal{P}(1, \rightrightarrows, \square)$ variant employs this particular setup and consistently outperforms the original version on all tested NVIDIA GPUs—in most cases even by a large margin. We further found that the performance of this approach can be improved by restricting the number of views that are rendered at the same time. Our empirical tests have shown that limiting the number of simultaneously processed views to 4 works best, which will be a recurring theme in the following sections. An interpretation of this trend is provided in Section 2.6. The resulting instanced rendering-based pipeline variant is described with the symbol $\mathcal{P}(\lceil \frac{N}{4} \rceil, \rightrightarrows, \square\square)$ and shows the most favorable ratio across the pipeline variants considered in this section when compared to our baseline (see Figure 2.3a). Since none of these variants performed better than our baseline on weaker GPU models for 4 or more views, they are excluded from our detailed analysis in this chapter. Corresponding results can be found in our supplemental material.

Geometry Shader-Based Techniques

Like Wilson's approach, the approach by Sorbier et al.[DNS10] targets a single, partitioned framebuffer and uses a single invocation to produce N views. However, their geometry



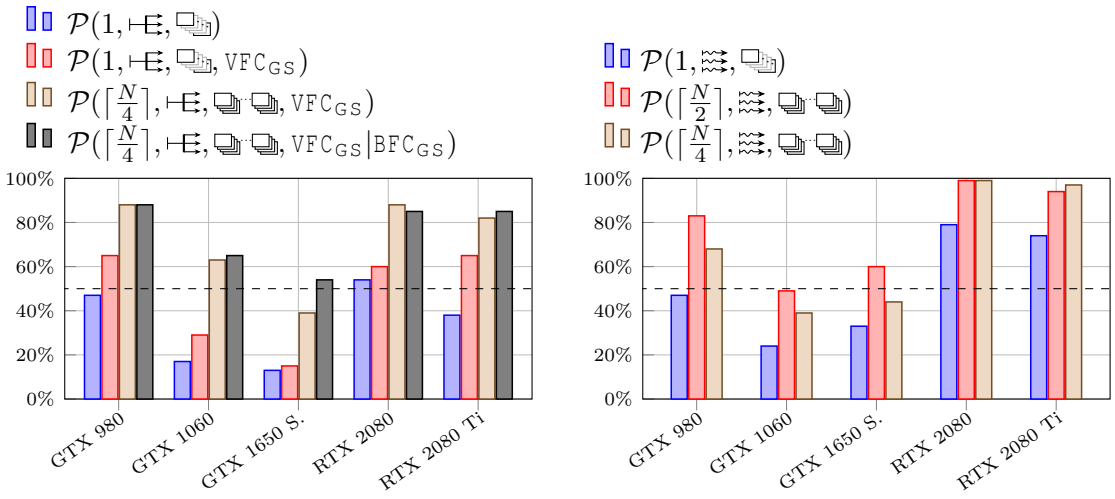
(a) *Bars represent the percentage of configurations where a particular instanced rendering variant showed better performance than* $\mathcal{P}(N, \rightarrowtail, \square\square)$ *for the same scene configuration.*

(b) *Bars represent the percentage of tests where a particular geometry shader-based variant showed better performance than our baseline* $\mathcal{P}(N, \rightarrowtail, \square\square)$ *for the same scene configuration.*

Figure 2.3: *These bar charts show performance comparisons between instancing-based variants and our baseline in Figure 2.3a, and between variants using a loop in geometry shaders in Figure 2.3b.*

amplification occurs in a geometry shader loop. The geometry shader further applies frustum culling to reduce the input to the rasterizer and performs clipping in the fragment shader to restrict rendering to each view's framebuffer region. Their pipeline variant can thus be denoted by $\mathcal{P}(1, \mapsto\!\!\lessdot, \square, \texttt{VFC}_{\texttt{GS}}|\texttt{CLIP}_{\texttt{FS}})$. Once again, restricting the number of simultaneous views provides a significant performance boost. We further saw that switching the large framebuffers for layered ones yields overall better performance. This is partly due to the fact that the fragment shader stage must no longer perform `discard` operations to achieve clipping and can take advantage of early depth testing. However, we also found that keeping the frustum culling routine in the geometry shader is beneficial; since the geometry shader is executed in a loop, testing each triangle against a frustum incurs only a small overhead which can be amortized by the reduced output of the geometry shader. We denote this pipeline as $\mathcal{P}(\lceil \frac{N}{4} \rceil, \mapsto\!\!\lessdot, \square\square, \texttt{VFC}_{\texttt{GS}})$. To further relieve the rasterization stage, we have added backface culling to the geometry shader routine, which led to a relative speedup in more than 60% of all cases. Figure 2.3b shows that overall, geometry shader loop-based variants that render into layered framebuffers outperform our baseline.

Instead of a simple geometry shader loop, we also consider methods based on fixed geometry shader instancing. This type of geometry amplification can be achieved by defining a fixed number of geometry shader invocations, which is part of OpenGL's core functionality since version 4.0. While previous literature does not mention any



(a) *Percentage of cases where a geometry shader instancing-based variant showed better average performance than $\mathcal{P}(N, \rightarrow, \square\square)$ for the same scene configuration.*

(b) *Bars represent the percentage of configurations where an OVR variant showed better performance than $\mathcal{P}(N, \rightarrow, \square\square)$ for the same scene configuration.*

Figure 2.4: *These bar charts show performance comparisons between geometry shader instancing-based variants and our baseline in Figure 2.4a, and between variants using OVR hardware acceleration over our baseline in Figure 2.4b.*

comparable variants, we found this amplification method to work particularly well on NVIDIA GPUs in our use case. Similar to the loop-based pipelines, we have tested combinations with custom frustum and backface culling routines in the geometry shader. On average, we found that the most effective techniques include both of these traits and target layered framebuffers. Based on previous impressions, we also constrained the number of views rendered per invocation down to 4, resulting in an appreciable performance increase. The percentages of cases where these variants outperform our multi-pass baseline are plotted in Figure 2.4a. Of all MVR variants that do not rely on hardware acceleration, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \mapsto\!\!\!\!\equiv, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ performed best across all tested NVIDIA GPU models.

OVR and Hardware-Acceleration

The OVR extension allows defining multiple target views for which vertex shader outputs can be written. On NVIDIA Pascal and Turing architectures, choosing 2 or 4 target views respectively allows the extension to exploit the underlying MVR hardware features for maximal efficiency. If more views are defined than the hardware supports, a slower fallback mechanism will be triggered instead. Hence, restricting the number of simultaneous views may again be beneficial to performance in this particular geometry amplification mode. For instance, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \rightrightarrows, \square\square)$ describes a variant of MVR that partitions the N views into chunks of four views each and is thus accelerated on Turing. OVR may

Table 2.3: *Results of ID buffer generation for different resolutions, scenes, and view counts per GPU. Each table cell represents averaged frame times from 102 measurements per configuration in milliseconds. For brevity, we use the following shorthand to reference our MVR techniques: M.Pass = $\mathcal{P}(N, \rightarrow, \square\square)$, $GSL_4 = \mathcal{P}(\lceil \frac{N}{4} \rceil, \bowtie, \square\square, VFC_{GS}|BFC_{GS})$, $GSI_4 = \mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\!\equiv, \square\square, VFC_{GS}|BFC_{GS})$ and $OVR_X = \mathcal{P}(\lceil \frac{N}{X} \rceil, \approx, \square\square)$.*

			AMD RX580			GTX 980				GTX 1060				RTX 2080			
	Scene	#Views	M.Pass	GSL_4	GSI_4	M.Pass	GSL_4	GSI_4	OVR_2	M.Pass	GSL_4	GSI_4	OVR_2	M.Pass	GSL_4	GSI_4	OVR_4
1920 × 1080	Bistro	2 × 2	**2.24**	4.48	3.65	2.56	1.95	**1.74**	2.50	2.46	2.50	**2.44**	2.69	1.11	0.96	0.89	**0.88**
		4 × 4	**8.97**	17.88	14.58	10.26	7.81	**7.03**	9.96	8.96	7.85	**7.20**	8.90	4.22	3.67	3.46	**3.32**
		8 × 4	**17.84**	35.66	29.01	20.34	15.60	**14.03**	19.77	17.89	15.51	**14.28**	17.65	8.31	7.16	6.65	**6.62**
	San Miguel	2 × 2	**5.34**	9.74	6.95	6.21	3.87	**3.21**	5.27	5.40	3.75	**3.30**	4.83	3.21	2.07	1.87	**1.68**
		4 × 4	**21.66**	38.58	27.25	27.90	15.61	**13.15**	22.03	26.12	15.20	**13.64**	20.14	16.36	7.03	6.25	**6.09**
		8 × 4	**43.17**	76.76	54.03	57.80	31.33	**26.37**	44.20	54.09	30.50	**27.04**	40.81	34.39	14.59	13.13	**12.6**
	Sponza	2 × 2	**0.72**	1.89	1.79	0.78	0.65	0.63	**0.61**	0.87	1.16	1.01	**0.95**	0.48	0.35	0.37	**0.34**
		4 × 4	**2.99**	7.64	7.21	3.21	2.68	2.58	**2.54**	3.03	3.03	**2.97**	3.08	1.83	1.41	1.37	**1.32**
		8 × 4	**6.02**	15.28	14.40	6.57	5.38	5.17	**5.03**	5.97	5.15	5.01	**4.91**	3.45	2.89	2.91	**2.77**
	Viking Village	2 × 2	**2.62**	4.17	3.40	2.94	2.04	**1.78**	2.82	2.57	**2.11**	2.13	2.68	1.75	1.06	**0.88**	0.96
		4 × 4	**10.41**	16.77	13.54	12.47	8.07	**7.14**	11.57	11.04	7.78	**7.24**	10.38	5.97	3.82	**3.48**	3.55
		8 × 4	**20.65**	33.34	27.17	25.24	16.16	**14.33**	22.99	22.75	15.77	**14.63**	20.89	12.06	7.69	**7.17**	7.32

only be used with array textures and may not be combined with tessellation or geometry shading, which fairly limits the amount of OVR-based MVR pipeline variants. The relative speedup of OVR-based methods over our multi-pass baseline is significant on all NVIDIA GPUs. On older microarchitectures, $\mathcal{P}(\lceil \frac{N}{2} \rceil, \approx, \square\square)$ performs better in at least 50% of our tests. Turing GPUs can exploit acceleration for $\mathcal{P}(\lceil \frac{N}{4} \rceil, \approx, \square\square)$, which has a significant impact on the results since most of our configurations render >4 views. On recent models, both hardware-accelerated variants clearly outperform our baseline in >90% of all test cases. This is also shown in Figure 2.4b.

2.5.2 Exploring MVR Setup Parameters and Analysis

We provide a detailed analysis of how setup parameters affect performance of the most promising MVR methods, based on our initial ID buffer evaluation: The geometry shader loop-based configuration $\mathcal{P}(\lceil \frac{N}{4} \rceil, \bowtie, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$, the geometry shader instancing-based configuration $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\!\equiv, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ and OVR-based methods. Tables 2.3 to 2.5 list the run time results for selected, representative setups. For the full list of timings, please refer to the supplemental material or our original paper [Unt+20].

Impact of Scene Size

In the results from the NVIDIA models, we could observe that some pipelines are better suited to particular scenes than to others. For the larger scenes ("Bistro", "San Miguel", and "Viking Village"), geometry shader-based pipelines outperform other pipeline variants on Maxwell and Pascal in all test cases and stay within a 20% performance margin to OVR-based techniques on Turing in most cases. $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\!\equiv, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ turns out to be one of the top-performing techniques in all configurations for the ID buffer tests. Only on

high-end Turing GPUs, $\mathcal{P}(\lceil\frac{N}{4}\rceil, \approx, \square\square)$ is often showing significantly better performance. (compare with Table 2.3). For the remaining, smaller scenes, performance generally varies less across different pipelines. The overall picture shows relative performance gains in favor of OVR-based techniques and our baseline $\mathcal{P}(N, \rightarrow, \square\square)$, compared to geometry shader-based techniques. OVR-based techniques exhibit good performance across all test scenes, but geometry shader-based techniques stay within a performance margin of 20% even on the 2080 Ti in the majority of test cases. While $\mathcal{P}(N, \rightarrow, \square\square)$ falls behind OVR-based techniques in most cases on NVIDIA GPUs, it is the fastest technique on the AMD RX 580 in 100% of tests for small and large scenes alike.

Differences Across GPU Architecture

Geometry shader-based pipelines show consistent performance characteristics across different NVIDIA GPU microarchitectures. A GPU's performance tier has more impact on the render times than the microarchitecture. This effect is less pronounced with techniques that partition rendering into sets of 4 views ($\lceil\frac{N}{4}\rceil$) and becomes obvious for geometry shader-based "all-in-one" techniques of the type $\mathcal{P}(1, ...)$. On low-tier models (i.e. GTX 1060, and GTX 1650 SUPER), they performed worse than $\mathcal{P}(N, \rightarrow, \square\square)$ in 73% of all tests. The optimized pipeline variants of type $\mathcal{P}(\lceil\frac{N}{4}\rceil, ...)$ show better performance across all NVIDIA GPUs, surpassing $\mathcal{P}(N, \rightarrow, \square\square)$ in virtually all test cases, as can be seen in Table 2.3.

Since NVIDIA supports the hardware-accelerated creation of up to four views on Turing, we expected performance gains to reflect the doubled number of simultaneous views compared to the Pascal microarchitecture. Indeed, the number of test cases where an OVR variant outperforms other techniques increases on Turing. However, the effect is most noticeable on high-tier NVIDIA GPUs. On the GTX 1650 SUPER, we cannot report an overall preference for OVR-based techniques, since $\mathcal{P}(\lceil\frac{N}{4}\rceil, \mapsto\!\!\!\!\approx, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ shows better performance in 43% of test cases and roughly equal performance in the others. These relations change drastically with the high-tier models: $\mathcal{P}(\lceil\frac{N}{4}\rceil, \approx, \square\square)$ performs better in one third of all test cases by a large margin (>20%). Comparing $\mathcal{P}(\lceil\frac{N}{4}\rceil, \approx, \square\square)$ to $\mathcal{P}(\lceil\frac{N}{2}\rceil, \approx, \square\square)$, the former showed advantages on the RTX 2080 and RTX 2080 Ti in 49% and 63% of all tests, respectively. On all other GPUs, the differences between those two OVR-based variants are marginal. Aside from the advantageous performance of $\mathcal{P}(\lceil\frac{N}{4}\rceil, \approx, \square\square)$ on high-tier Turing GPUs, $\mathcal{P}(\lceil\frac{N}{4}\rceil, \mapsto\!\!\!\!\approx, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ seems to be the best choice for other NVIDIA GPUs across a multitude of different configurations.

Varying Number of Views

A very consistent observation across our test results is the dominance of OVR-based techniques for stereo rendering on the Turing microarchitecture. This comes as no surprise since stereo rendering is the declared purpose of NVIDIAs hardware MVR support. No other pipeline variant was able to outperform $\mathcal{P}(\lceil\frac{N}{4}\rceil, \approx, \square\square)$ on the RTX 2080 and the RTX 2080 Ti in any of our test cases. On the GTX 1650 SUPER and on the

Table 2.4: *Results of G-buffer generation for different resolutions, scenes, and view counts per GPU. Each table cell represents averaged frame times from 60 measurements per configuration in milliseconds. For brevity, we use the following shorthand to reference our MVR techniques: M.Pass* $= \mathcal{P}(N, \rightarrow, \square\square)$*, GSL*$_4$ $= \mathcal{P}(\lceil \frac{N}{4} \rceil, \mapsto\!\!\lessdot, \square\square, \texttt{VFC}_{GS}|\texttt{BFC}_{GS})$*, GSI*$_4$ $= \mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\equiv, \square\square, \texttt{VFC}_{GS}|\texttt{BFC}_{GS})$ *and OVR*$_X$ $= \mathcal{P}(\lceil \frac{N}{X} \rceil, \approx, \square\square)$*.*

			AMD RX580			GTX 980				GTX 1060				RTX 2080			
	Scene	#Views	M.Pass	GSL$_4$	GSI$_4$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_4$
1920 × 1080	Bistro	2 × 2	**2.76**	5.07	4.35	2.40	2.22	**2.15**	2.27	**2.54**	3.07	2.88	2.65	1.57	1.39	1.42	**1.33**
		4 × 4	**11.07**	20.31	17.40	9.46	8.99	**8.61**	9.04	8.33	8.96	8.90	**8.22**	5.74	4.95	4.93	**4.76**
		8 × 4	**22.06**	40.44	34.73	18.92	17.90	**17.20**	18.00	17.03	17.98	17.94	**16.60**	10.80	9.76	9.59	**9.49**
	San Miguel	2 × 2	**6.64**	23.03	15.39	7.43	7.02	**5.82**	6.67	6.59	7.15	6.70	**6.08**	3.97	3.97	3.72	**3.19**
		4 × 4	**26.28**	91.60	61.07	31.06	28.08	**23.23**	26.73	28.50	28.41	26.49	**24.15**	17.67	12.39	11.29	**9.47**
		8 × 4	**52.78**	182.8	121.68	62.36	55.83	**46.11**	53.33	57.48	56.14	52.22	**48.26**	35.97	24.56	22.46	**18.84**
	Sponza	2 × 2	**1.89**	3.32	3.09	1.55	1.50	1.40	**1.35**	**1.73**	1.86	2.49	2.17	1.09	**1.03**	1.08	**1.03**
		4 × 4	**7.61**	13.38	12.44	6.35	5.78	5.70	**5.54**	5.76	5.51	5.48	**5.11**	4.20	**3.70**	3.71	3.72
		8 × 4	**15.22**	26.63	24.77	12.91	11.60	11.27	**10.98**	12.07	11.18	11.11	**10.49**	8.93	7.38	7.42	**7.22**
	Viking Village	2 × 2	**3.82**	10.95	7.84	4.22	3.90	**3.40**	3.94	**3.62**	4.06	3.91	3.73	1.97	1.97	2.01	**1.96**
		4 × 4	**15.08**	43.37	31.40	17.25	15.54	**13.71**	15.61	15.00	16.20	15.72	**14.11**	9.01	7.32	6.95	**6.11**
		8 × 4	**29.98**	86.69	62.71	34.65	31.23	**27.40**	31.13	30.79	32.21	31.35	**28.30**	17.22	14.62	13.63	**12.25**

GTX 1060, OVR-based techniques still performed very well in all test cases with two views. For four or more views, geometry shader-based pipeline variants start to show competitive performance characteristics. For 16 and 32 views, they stay within a 10% performance margin to OVR-based techniques in most test cases and often outperform them on pre-Turing microarchitectures. Due to the consistent performance characteristics for $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$ MVR techniques with four or more views, we argue that they scale comparably well with the number of simultaneously rendered viewpoints. On the AMD RX580, $\mathcal{P}(N, \rightarrow, \square\square)$ again yields the best performance for all view counts (see Table 2.3).

Added Vertex & Fragment Load

To establish performance trends of applications with higher vertex load—e.g., vertex skinning—we have simulated highly expensive vertex stages by adding a loop that performs 15k fused multiply-add instructions to the shader. Techniques that amplify geometry before the geometry shader stages are impacted more severely by increased vertex load. Amplifying geometry *in* the geometry shader potentially saves up to $N-1$ expensive vertex shader invocations. The reduced number of vertex shader-invocations of $\mathcal{P}(1, ...)$-types compared to $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-types are reflected in our measurements: While $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-types outperform $\mathcal{P}(1, ...)$-types in 100% of test cases with light vertex load by a huge margin, the latter perform better than the former for many tests with high vertex load across NVIDIA GPUs. OVR-based techniques cope well with high vertex load on Turing: On the GTX 1650 SUPER, they outperform all other techniques in 100% On the RTX 2080 Ti, OVR-based techniques perform worse than $\mathcal{P}(1, \vdash\!\!\equiv, \square, \texttt{VFC}_{GS})$ in 50% of test cases, the same percentage where $\mathcal{P}(1, \vdash\!\!\equiv, \square, \texttt{VFC}_{GS})$ outperforms $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\equiv, \square\square, \texttt{VFC}_{GS}|\texttt{BFC}_{GS})$. On Maxwell and Pascal, the top-performing techniques are $\mathcal{P}(1, \mapsto\!\!\lessdot, \square, \texttt{VFC}_{GS})$ and

Table 2.5: *Results of LQ shadow mapping for different resolutions, scenes, and view counts per GPU. Each table cell represents averaged frame times from 30 measurements per configuration in milliseconds. For brevity, we use the following shorthand to reference our MVR techniques: M.Pass* $= \mathcal{P}(N, \rightarrowtail, \square\square)$*, GSL*$_4$ $= \mathcal{P}(\lceil \frac{N}{4} \rceil, \multimap, \square\square, \texttt{VFC}_{GS}|\texttt{BFC}_{GS})$*, GSI*$_4$ $= \mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\!\equiv, \square\square, \texttt{VFC}_{GS}|\texttt{BFC}_{GS})$ *and OVR*$_X$ $= \mathcal{P}(\lceil \frac{N}{X} \rceil, \approx, \square\square)$*.*

			AMD RX580			GTX 980				GTX 1060				RTX 2080			
	Scene	#Views	M.Pass	GSL$_4$	GSI$_4$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_2$	M.Pass	GSL$_4$	GSI$_4$	OVR$_4$
800 × 600	Bistro	4	**1.18**	2.15	3.98	**1.37**	2.54	2.44	2.45	**1.35**	3.01	3.02	2.34	**0.50**	0.98	1.11	1.27
		16	**6.00**	11.95	22.47	**6.30**	11.36	11.44	11.60	**5.31**	11.77	11.73	8.96	**2.42**	4.45	5.60	5.32
		32	**12.18**	23.96	43.17	**12.65**	21.54	21.54	21.65	**9.84**	22.05	22.37	16.68	**4.61**	8.42	8.08	10.61
	San Miguel	4	**3.23**	7.83	13.64	**3.39**	5.97	5.85	6.41	**2.93**	5.98	6.08	4.75	4.05	**3.12**	3.47	4.60
		16	**13.39**	34.23	61.54	**14.95**	26.62	26.88	26.58	**13.19**	27.19	27.54	21.73	11.49	**9.66**	11.49	12.29
		32	**25.78**	66.07	118.65	**30.41**	51.61	51.90	51.97	**27.77**	52.95	53.92	40.45	11.49	**9.66**	11.49	12.29
	Sponza	4	**0.46**	0.76	1.12	**0.35**	0.56	0.57	0.62	**0.35**	0.64	0.66	0.52	**0.17**	0.22	0.24	0.24
		16	**1.58**	3.19	4.64	**1.33**	2.23	2.28	2.17	**1.45**	2.75	2.86	2.05	**0.73**	0.95	1.03	1.03
		32	**2.59**	5.70	8.22	**2.62**	4.28	4.43	4.05	**2.63**	5.03	5.12	3.66	**1.43**	2.02	2.32	1.86
	Viking Village	4	**3.44**	7.38	13.00	**3.80**	5.86	5.82	6.22	**3.36**	5.80	5.86	4.92	**1.97**	2.25	2.44	2.81
		16	**7.53**	14.04	25.30	**7.88**	13.42	13.35	14.25	**6.38**	13.91	14.13	10.99	**3.08**	4.67	5.09	5.92
		32	**16.08**	30.91	54.57	**16.82**	28.58	28.35	29.87	**13.16**	29.15	29.17	22.73	**6.40**	10.64	10.44	12.07

$\mathcal{P}(1, \vdash\!\!\!\equiv, \square, \texttt{VFC}_{GS})$. For increased fragment load, we use the results obtained from our G-buffer rendering application (see Table 2.4). Here, geometry shader-based techniques lose performance compared to OVR-based techniques and—especially on low-tier GPUs—also to $\mathcal{P}(N, \rightarrowtail, \square\square)$. While OVR-based techniques also showed good performance characteristics with light fragment load, they show a clear lead with increased fragment load by means of G-buffer rendering: Their performance is better than the otherwise well-performing geometry shader-based techniques in 40% of test cases on Maxwell and in up to 90% of test cases on Pascal and Turing. However, the advantage is substantial (>20%) only on high-end Turing GPUs.

Influence of Viewpoint Discrepancy

The fundamental difference between our ID buffer/G-buffer tests and our shadow mapping is the scene setup w.r.t. the view frusta. While for ID buffer tests, view frusta have strong coherence, for shadow mapping, the view frusta might not overlap at all (see Figure 2.2). The resulting performance measurements for shadow mapping (a selection of which is provided in Table 2.5) draw a highly interesting—and consistent—picture: $\mathcal{P}(N, \rightarrowtail, \square\square)$ outperforms OVR-based techniques in the majority of test cases on all NVIDIA GPUs. For all non-high-tier Turing GPUs, we can even observe more than 40% faster frame times in at least half of all test cases (varying per GPU). On high-tier Turing, the performance differences are a bit less pronounced but still substantial (>20% faster in half of all test cases on the RTX 2080 Ti) Geometry shader-based techniques are not faring better with shadow mapping either. We observed similar relations to $\mathcal{P}(N, \rightarrowtail, \square\square)$ as with OVR-based techniques. While OVR-based techniques generally show slightly better performance than geometry shader-based techniques, they stay within a 20% margin on most GPUs except for the GTX 1060.

These performance relations are in stark contrast to the results from ID buffer generation, where $\mathcal{P}(N, \mapsto, \square\square)$ shows worse performance than both, OVR-based techniques and geometry shader-based techniques across all NVIDIA GPUs—especially on high-tier models. It appears that NVIDIA GPUs can take advantage of cases where geometry is visible in multiple views that are rendered with the same draw call. With views that do not share geometry, this advantage vanishes, leading to worse performance than $\mathcal{P}(N, \mapsto, \square\square)$. A tentative explanation for this phenomenon is given in Section 2.6.

2.6 Discussion

While some of the results in Section 2.5 turned out as expected—e.g., OVR's stereo rendering performance—other results were more surprising: Geometry shader-based pipeline variants of the $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-type showed very competitive performance for many test cases on NVIDIA GPUs. $\mathcal{P}(N, \mapsto, \square\square)$ remains the overall strongest technique for shadow mapping with incoherent view frusta on NVIDIA models. To provide a deeper understanding of the performance characteristics that we observed for the different MVR approaches, we used the NVIDIA Nsight Graphics profiler on an RTX 2080 and analyzed frame captures for test cases that generate the maximal number of target views in our three applications.

Due to the relatively low load on vertex and fragment shading throughout all tested applications, streaming multi-processor (SM) utilization is never a limiting factor. The $\mathcal{P}(N, \mapsto, \square\square)$ approach is limited by viewport culling (VPC) in all three applications, with raster operations (ROPs) and memory access (VRAM) being high or close to the physical limit. While G-buffer rendering shows the highest ROP utilization, no memory operation reached the hardware limits. The SM utilization caps at 15%, with most load stemming from vertex shading. Geometry shader-based $\mathcal{P}(1, ...)$-type pipelines increase SM utilization: it reaches 30% with $\mathcal{P}(1, \mapsto\!\!\!\!\!\!\diagup, \square, \texttt{VFC}_{\texttt{GS}})$ and 85% with $\mathcal{P}(1, \mapsto\!\!\!\!\!\!\equiv, \square, \texttt{VFC}_{\texttt{GS}})$. All other GPU units show low utilization. Most interestingly, VPC is reduced to 15%–30%, which is owed to culling in the geometry shader. In some instances, we found a sudden VRAM overload, although the technique should in theory reduce VRAM access the most. We attribute this effect to temporary scheduling/memory management issues after geometry shading, which may lead to L2 cache thrashing or to overflow of internal work queues. We only observed this issue in some test cases, and only with 32 views being generated. As this issue rarely arises, the generally low performance of geometry shader-based $\mathcal{P}(1, ...)$-types cannot be attributed to this effect. We believe the main issue is a typical geometry shader problem: if many outputs may be generated, memory management and scheduling become challenging, leaving all profiled units underutilized.

Geometry shader-based $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-type methods show an overall better balance. Custom culling in the geometry shader reduces VPC pressure significantly compared to our baseline $\mathcal{P}(N, \mapsto, \square\square)$. Also, VRAM is in general significantly lower than with our baseline, while it is slightly higher than with $\mathcal{P}(1, ...)$-type methods (when it does not spike VRAM). Geometry shader-based $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-type pipeline variants do not show any

sudden VRAM spikes and SM utilization is overall low with 10%–12%. Thus, there is overall no clear bottleneck and it seems that geometry shader output scheduling or simply missing workload may again be the limiting factor for these approaches. In contrast to shadow mapping, the geometry shader output is more coherent in ID buffer and G-buffer test cases, as most often triangles are either culled for all four views or emitted four times. VPC load caps at about 40% to 50%. Shadow mapping on the other hand typically emits one or two triangles, which makes scheduling a lot harder and thus leads to lower performance in this application. Also, VPC pressure remains higher for shadow mapping (up to 75%). However, when able to exploit geometry reuse, $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-type methods find a sweet spot as they reduce the load on the $\mathcal{P}(N, \mapsto, \square\square)$ bottleneck and do not lead to too complicated scheduling/memory management issues for the hardware scheduler.

$\mathcal{P}(\lceil \frac{N}{4} \rceil, \approx, \square\square)$ shows the highest load on VPC of all tested methods in all test cases (90–100%) All other components see little load, recording the lowest load on everything but SM at 14%–17%. In the ID buffer and G-buffer applications, $\mathcal{P}(\lceil \frac{N}{4} \rceil, \approx, \square\square)$ yields very competitive performance, which we attribute to scheduling also being efficient, as again culling is very consistent and rasterizer queue fill rates are similar. For shadow mapping, the performance is less competitive although the profiling characteristics do not show a vastly different behavior. Our assumption is that scheduling may be an issue, again, as culling clogs the pipeline and triangles trickle into the rasterizer queues of the different views. $\mathcal{P}(N, \mapsto, \square\square)$ is also limited by VPC, but generates all the load for each disjoint view at a time and thus achieves better scheduling and overall utilization.

Our evaluation results indicate that OVR-based techniques show clear advantages with stereo rendering, increased vertex and/or fragment load, and in general on high-tier Turing GPUs. For low vertex and fragment loads, higher numbers of views, and especially on previous NVIDIA microarchitectures, we often found $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\!\equiv, \square\square, \mathtt{VFC_{GS}}|\mathtt{BFC_{GS}})$ to show slightly superior and consistent performance. It also exhibits relatively good performance with the bigger scenes tested. We recommend this particular pipeline variant in general for use cases that require geometry or tessellation shaders, which are not supported by OVR-based techniques. For non-overlapping view frusta and in general, for AMD GPUs $\mathcal{P}(N, \mapsto, \square\square)$ shows the best performance across all tests. We found that overall rendering performance for producing multiple views can drastically be improved by splitting the workload into packages of four views at a time, which is utilized by $\mathcal{P}(\lceil \frac{N}{4} \rceil, ...)$-type variants. Although requiring $\lceil \frac{N}{4} \rceil$ times the number of draw calls compared to $\mathcal{P}(1, ...)$-type variants, they enable better load distribution and balance as detailed above. Only in combination with very high vertex load or excessive numbers of draw calls, performance shifts in favor of $\mathcal{P}(1, ...)$-type pipeline variants. For non-hardware-accelerated MVR techniques, we can state as a general rule of thumb that performing geometry amplification as late as possible is advantageous, and that custom frustum and backface culling in geometry shaders further increases performance. Using layered framebuffers comes with the advantages that clipping can be performed by the rasterizer and early depth tests can be utilized.

2.7 Conclusion and Future Work

In this chapter, we have examined a wide range of different techniques that are available today on modern GPU hardware to achieve multi-view rendering. In order to facilitate the concise description of relevant properties, we have introduced a general and extensible pipeline catalog. Our evaluation spans multiple GPU generations and use cases, and provides a basis for making informed decisions w.r.t. the applicability of different pipelines, as well as the main factors that impact their performance. In contrast to most available material on this topic, we go beyond stereoscopic projection and analyze multi-view setups that target more than two views. We have shown that, with the help of widely available rendering API features, we can achieve a performance improvement of 5–6× over earlier methods that were explicitly recommended for such scenarios.

While we found NVIDIA's hardware support and the general OVR extension for multi-view rendering to work well across a wide range of setups, we also observed that it can be outperformed in applications with low shading load, particularly on weaker GPU models. Furthermore, the lack of support for tessellation and geometry shading in this feature motivates the question of which alternatives can be used if these pipeline stages are required. Even in 2024, support for the *EXT_multiview_tessellation_geometry_shader* extension is very low across all GPUs, with only 0.88% of devices supporting it [Wil24]. For those cases when using OpenGL, we have identified suitable, optimized methods based on geometry shader instancing that are usually within 15–20% of the fastest OVR-based techniques, and are even outperforming them for some scene configurations.

Future work might find it worthwhile to extend our catalog by additional pipeline variants that are enabled by mesh shaders [KB19; Kub18b]. Based on the improvements obtained by the application of custom clipping and culling steps in various pipeline variants, we are confident that the enhanced programmability of the geometry stages through mesh shaders can facilitate further performance gains while preserving the ability to perform custom subdivision steps as part of rasterization-based graphics pipelines. All techniques are available as part of our testbed, which we provide for download and further experiments on GitHub [1].

With the more modern Vulkan API, support for the combination of hardware-accelerated MVR and geometry and tessellation shaders is much better. While the early multiview extension *VK_KHR_multiview* [Khr24vsp] shows low support across various GPUs for the additional shader stages, the promotion of the multiview feature into Vulkan's core features led to wide support: In 2024, 67.39% of devices support the *multiviewGeometryShader* feature, and 73.94% support the *multiviewTessellationShader* feature (Core 1.1 features) [Wil24]. Therefore, an application targeting Khronos' new graphics API seems to not be severely affected by the absence of the combination of hardware-accelerated MVR and support for tessellation shaders on most devices. These significantly different numbers in hardware support can be seen as another justification for preferring the Vulkan API over OpenGL, as reasoned in our papers about transitioning to Vulkan [UKW22; UKW23],

[1] https://github.com/cg-tuwien/FastMVR

and in Chapter 3. Also with the Vulkan API, geometry shader instancing-based graphics pipeline configurations are likely to lead to different caching behavior and utilization of a GPU's modules compared to the hardware-accelerated multiview extension. Therefore, new applications might benefit from exploring the performance characteristics of ⊢⇶ pipeline variants.

CHAPTER 3

Using a Modern, Low-Level Graphics API for Teaching and Research

The contents of this chapter are largely based on our paper "Vulkan all the way: Transitioning to a modern low-level graphics API in academia", published in Computers and Graphics, volume 111, pages 155–165, 2023 by Elsevier [UKW23].

3.1 Motivation

The OpenGL API has provided users with the means for implementing versatile, feature-rich, and portable real-time graphics applications for many years. Consequently, it has been widely adopted by practitioners, researchers, and educators alike and is deeply ingrained in many curricula that teach real-time graphics for higher education. Over the years, the architecture of GPUs incrementally diverged from OpenGL's conceptual design, which might be one of the reasons why not all GPU features see such a high adoption rate in OpenGL as in other graphics APIs. One example is the *OVR_multiview* extension, as described in Section 2.7. Another example is the missing support for real-time ray tracing with OpenGL.

The more recently introduced Vulkan API provides a more modern, fine-grained approach for interfacing with the GPU, which allows a high level of controllability and, thereby, deep insights into the inner workings of modern GPUs. This property makes the Vulkan API especially well-suitable for research—especially in the field of real-time rendering, where performance is essential—and also teaching in university education.

3.2 Introduction

For over two decades, OpenGL has remained the default choice for real-time rendering research at universities, and for teaching undergraduate students the use of graphics APIs. Its high portability as well as an extensive body of documentation, guides, and tooling options (e.g., open-source software emulators) made it the logical choice for university usage. However, there are clear indicators that we are at a juncture where using OpenGL for research and teaching is no longer adequate: Its API design as a state machine is often considered bothersome and, in many cases, no longer reflects the underlying hardware architecture. More severely, several interesting and desirable features of modern APIs, such as push constants or hardware-accelerated ray tracing, are simply not supported by OpenGL. The practical reasons for and against using OpenGL today are succinctly illustrated by our experience using it in research. In our work on fast multi-view rendering [Unt+20], we already felt the age of OpenGL. Its usage turned out to be more error-prone due to the lack of proper error messages when compared to the modern low-level graphics API of our choice: Vulkan [Khr22vk]. For further research, we decided to switch to Vulkan, since it abstracts the hardware on a lower level than OpenGL, offering more insights and much more fine-grained control over the actual work that is carried out by GPUs, leading to better and more productive development experience once learned. Consequently, our goal was set towards making the transition from OpenGL to Vulkan also in teaching in academia.

Besides Vulkan, there are two other major, modern, low-level APIs: DirectX 12 [Mic22] and Metal [Apl22]. While most modern graphics APIs are similarly aligned in terms of usage principles and their level of hardware abstraction, only Vulkan is usable across all major desktop operating systems and across device categories (albeit only through an intermediate layer [Bre22] on Apple platforms). Furthermore, it is an open industry standard defined by the members of the Khronos group, which includes all major GPU manufacturers, operating system manufacturers, and other individual, academic, and industry members [Khr22me]. They all contribute to shaping and maintaining the Vulkan API, while DirectX and Metal are proprietary standards, defined and controlled by a single company each. Vulkan appears to be a future-proof API. Thanks to vendor-specific extensions, new hardware features are accessible in a timely manner. E.g., hardware-accelerated ray tracing was available through an NVIDIA-specific extension [Khr18ray] only one month after its availability in DirectX and has later been standardized [Khr20ray]. Given these conditions, Vulkan is the sensible choice in higher education in our opinion.

The challenge of learning Vulkan is revealed when comparing source code and descriptive text for two of the most famous tutorials for drawing a single triangle to the screen: The OpenGL tutorial at *LearnOpenGL.com* requires fewer than 150 lines of code (LOC) on the host side [Vri22]. In contrast, the de facto entry point for learning Vulkan at *vulkan-tutorial.com* ends up with approximately 700 LOC for achieving the same task and requires a much more extensive description for explaining the necessary setup leading up to this point [Ove22]. The tutorials illustrate how Vulkan is indeed an API that operates on a much lower abstraction level than OpenGL. This implies that there are many more

factors and talking points with Vulkan that must be addressed (at least to some degree) if taught to students. On the other hand, a potential upside thereof is that students receive a more fundamental knowledge about the inner workings of a modern GPU—and conveying fundamental knowledge constitutes a primary goal of higher education.

In this chapter, we give a summary of our experiences of transitioning teaching from OpenGL to Vulkan. Details and questionnaire evaluations can be found in our published work [UKW22; UKW23], where we describe the changes that we have made to transition an introductory graphics course to Vulkan in detail, and how we manage to keep workload in manageable bounds in an advanced graphics course that exclusively uses Vulkan. We can conclude that transitioning to Vulkan in university education was much less bumpy for our students than we initially anticipated. We propose the usage of tailored programming frameworks to keep the learning effort limited and focused.

Furthermore, we make the point that low-level access to a GPU's features is beneficial during real-time rendering research. Since Vulkan is very explicit and verbose, we suggest the usage of suitable frameworks—two of which we introduce in the following sections: We use *Vulkan Launchpad [Res22]* for teaching in introductory courses and *Auto-Vk [Res24a]* and *Auto-Vk-Toolkit [Res24b]* for higher-level courses and research. The latter two proved useful in our own research since they serve as the technological basis for the techniques presented in Chapters 4 and 5.

3.3 Related Work

Based on the analysis performed by Balreira et al. [BWF17], it can be concluded that OpenGL was the most widely used graphics API in university education in 2017, given the absence of any mention of other graphics APIs. We consider our suggestions and experiences described in this chapter as being potentially relevant to every department that is thinking about migrating from teaching OpenGL to teaching Vulkan but also to those who have already migrated. Experiences with the transition from legacy OpenGL to modern OpenGL in university education are described by Reina et al. [RME14]. They point to increased learning efforts in modern OpenGL due to reduced out-of-the-box convenience compared to legacy OpenGL. A similar point could be made when comparing Vulkan to modern OpenGL.

While Vulkan may provide a reasonable learning curve for developers who are proficient with various other APIs, it is notoriously difficult for students without prior experience. To fully appreciate Vulkan, users require an in-depth understanding of the underlying GPU hardware. The fine-grained control over work generation and scheduling necessarily makes Vulkan verbose. Hence, students are confronted with the task of implementing a significant amount of boilerplate code for leveraging hardware features they may not yet fully understand. This situation is further aggravated by the absence of in-depth teaching material for Vulkan: comprehensive, thoroughly researched hands-on guidebooks, such as OpenGL's famed "Red Book" [Shr+13] or the "OpenGL Superbible" [SWH15] are not yet available for Vulkan. Early attempts to provide additional abstraction or simplify

the interface had only limited success [AMD18]. However, Vulkan as an API is still evolving. Recent additions to the SDK, such as the `VK_KHR_dynamic_rendering` and `VK_KHR_synchronization2` extensions [Khr24vsp], aim to alleviate neuralgic pressure points of the API.

3.4 Vulkan in Introductory Graphics Courses

For teaching Vulkan in an introductory graphics course, we resort to using a suitable programming framework, so that students do not have to write all the boilerplate code themselves and instead, are able to focus on API usage and graphics development. I.e., the main purpose of this programming framework is to help students in learning the Vulkan API usage efficiently without getting lost. The programming framework that we have developed for this purpose is called *Vulkan Launchpad* [Res22].

There are five assignments in our introductory graphics course that serve different didactic purposes. In the first assignment, we let students create a Vulkan instance, a surface, select a physical device, create a logical device, queue, and swap chain manually. They directly interface with the Vulkan API for these tasks, because we consider it valuable to let students get in touch with each one of these fundamental types. The remainder of the required initial application setup is abstracted by the framework, namely installing a debug callback, framebuffer, and render pass creation, as well as the creation of the synchronization primitives (semaphores and fences) for swap chain handling [Khr22vsa]. If these had to be set up by students, complex concepts like image layout transitions and synchronization would have to be learned for the first assignment at the beginning of the course already, which we deemed to constitute a too-steep learning curve. Students must provide the created handles with additional configuration parameters (e.g., clear color values) to an initialization function. The resulting render loop implementation after completing Assignment 1 leads to C/C++ source code as shown in Listing 3.1.

Listing 3.1: *Render loop implementation after completing the first assignment. The parameters to the framework initialization function refer to handles of types* `VkInstance` *,* `VkSurfaceKHR` *,* `VkPhysicalDevice` *,* `VkDevice` *,* `VkQueue` *, and a custom configuration struct containing required swap chain parameters.*

```
// Instance, surface, physical device, logical device, queue, and swap chain
// configuration must be passed to the framework initialization function:
vklInitFramework(inst, srf, phd, dev, q, swpcfg);

while (!glfwWindowShouldClose(window)) { // Render loop
  glfwPollEvents(); // <- Handle user input
  // Wait until the next swapchain image becomes available for usage:
  vklWaitForNextSwapchainImage();
  vklStartRecordingCommands(); // <- Begin recording into a command buffer
  vklDrawTeapot();             // <- Record the draw calls for a teapot
  vklEndRecordingCommands();   // <- End command buffer recording
  vklPresentCurrentSwapchainImage(); // <- Present rendered image on screen
}
```

With this approach, we manage to defer teaching image layout transitions and synchronization to a much later point in the course. Not before Assignment 5, students have to use these for image loading and mipmap creation. The downside is that students do not interface with Vulkan directly in terms of swap chain handling and command buffer recording. Instead, they use framework utility functions (those with “vkl" prefixes). The code of the abstracted functionality in Listing 3.1 amounts to 300 LOC (not counting the functionality of graphics pipeline creation). Tasking students with implementing these functions on their own during Assignment 1 would have required bigger restructurings of the assignments and most likely would have required the removal of some tasks in later assignments. While it would not be strictly required to draw something to the screen for fulfilling the tasks of Assignment 1, letting the framework draw a red teapot to the current swap chain image provides students with additional feedback on whether their setup code is in a proper state, in addition to possible framework or Vulkan validation error messages.

Listing 3.2: *Auxiliary configuration struct with required parameters for custom graphics pipeline creation.*

```
struct VklGraphicsPipelineConfig
{
  const char* vertexShaderPath;
  const char* fragmentShaderPath;
  std::vector<VkVertexInputBindingDescription> vertexInputBuffers;
  std::vector<VkVertexInputAttributeDescription> inputAttributeDescriptions;
  VkPolygonMode polygonDrawMode;
  VkCullModeFlags triangleCullingMode;
  std::vector<VkDescriptorSetLayoutBinding> descriptorLayout;
};
```

The creation of custom graphics pipelines is the subject of Assignment 2. The required Vulkan code constitutes another 80 LOC just for graphics pipeline creation, which is why we decided to provide a framework function for it with hard-coded configuration values for many parameters. A few parameters can be configured via a custom struct, which is shown in Listing 3.2. It only supports vertex and fragment shader stages for the creation of graphics pipelines. Vertex input attribute descriptions translate directly to Vulkan’s `VkPipelineVertexInputStateCreateInfo` [Khr24vsp]. It is supposed to be set up for streaming vertex positions during Assignments 2 and 3, to be extended by vertex normals during Assignment 4, and by texture coordinates during Assignment 5. Further configurable parameters are the polygon drawing mode and the primitive culling mode, both relevant for Assignment 3. The last member is a set of descriptors stating all resources that are used in custom vertex or fragment shader programs, which is internally required for pipeline layout creation. For simplicity, we support only one descriptor set, but other than that, we do not abstract or simplify descriptor handling. Instead, students must handle descriptor set layout creation, descriptor set allocation, and descriptor writes manually in Assignments 2 to 5. Several uniform buffers have to be created for storing per-frame uniform data, such as colors and transformation matrices for different objects.

We refrain from introducing SPIR-V [Khr22spv], and from letting students compile shader modules on their own, but handle these parts internally in the framework using glslang [Khr22gsl]. This further reduces student workload so that they can focus on shader development. Compilation errors are displayed conveniently in the console. Further functionality that is abstracted by the framework is memory handling for buffers and images. Listing 3.3 shows the declarations of the relevant framework functions, the implementations of which amount to another 100 LOC. To make students aware of the fact that memory must actually be handled explicitly in Vulkan, we have chosen corresponding expressive function names (including the word “memory") and described them in detail in our documentation.

Listing 3.3: *Convenience functions for creating buffers and images, provided by the framework, which handle their associated device memory internally, opaquely to the user.*

```
VkBuffer vklCreateHostCoherentBufferWithBackingMemory(
    VkDeviceSize, VkBufferUsageFlags);

void vklCopyDataIntoHostCoherentBuffer(
    VkBuffer, const void*, size_t);

void vklDestroyHostCoherentBufferAndItsBackingMemory(
    VkBuffer);

VkImage vklCreateImageWithBackingMemory(
    uint32_t, uint32_t, VkFormat, VkImageUsageFlags);

void vklDestroyImageAndItsBackingMemory(
    VkImage);
```

3.5 Didactic Advantages of Using Vulkan

One side effect of Vulkan’s verbosity is that it necessarily reveals more and more underlying hardware details as students progress. Investigative minds are not easily satisfied by following a list of instructions they cannot comprehend. In order for them not to be deterred, Vulkan forces its users to deal with several important concepts discussed in this section that OpenGL does not. Consequently, instructors must address the underlying processes and hardware modules, while in OpenGL, the same use cases may “just work" because the details are handled by drivers internally. Therefore, OpenGL is less likely to encourage investigations of what is going on under the hood. Just by using a low-level graphics API like Vulkan *correctly*, educators are forced to convey more fundamental knowledge about modern GPUs and their architecture.

Vulkan implicitly conveys that switching between different shader programs is never free, as one might come to believe when developing applications based on OpenGL exclusively. Instead, whenever a different shader program is used, a whole new graphics pipeline must be created with all its bulk of configuration parameters. The extensive code blocks required to achieve this in Vulkan reflect that changing shaders is rather invasive to the

rendering setup and implies potentially heavy-weight changes. Users are encouraged by the design of the API to prepare all potentially required pipelines upfront, selecting the appropriate one during render-loop execution. In OpenGL, the driver usually hides this complexity and instead instantiates pipelines dynamically on-demand, reducing the amount of control the user has over the application's runtime performance. E.g., when the primitive culling mode is changed, that change can occur during render-loop execution, which might lead to an expensive operation being performed within the render loop.

When a uniform buffer is used to store per-frame data specific to a certain object (e.g., transformation matrices), the same buffer cannot be used for storing per-frame data of another object to be drawn in the same frame in most situations. Recording the drawing of multiple objects into the same command buffer requires the usage of different uniform buffers for the objects' per-frame data since otherwise unwanted effects occur. If, for example, the same host-visible uniform buffer was used for two objects, only the last write to this uniform buffer would succeed due to data being written at queue submission time [Khr24vsp]. These factors force users of the API to think about the reasons why this occurs. Developing these thoughts further, it becomes clear that modern GPUs work in a massively parallel way, which can also mean that both objects from our example are processed in parallel. As such, there must be different uniform buffers—one for each object—accessible during parallel processing of the objects. In OpenGL, again, users do not have to think about these aspects. Uniforms can just be set and used, and rendering "just works", producing the correct result. Users are not forced to think about the modus operandi of modern GPUs and, in the worst case, might think that draw calls are processed in a sequential or host-synchronous manner.

Synchronization, in general, is largely hidden from the API user with OpenGL, bearing the danger of drawing false conclusions about the actual command processing on the hardware. Vulkan, on the other hand, does not hide the responsibility of properly synchronizing commands from its API users, putting the massively parallel nature of modern GPUs into the spotlight. The necessary synchronization must not only be explicitly defined within shaders or between pipeline stages but also between the individual GPU queues that may receive and schedule incoming work. For students who desire to understand and exploit synchronization on a fundamental level, Vulkan provides an additional benefit over older graphics APIs, namely a clearly-defined memory consistency model, which is similar to the well-established C++ memory model [Hec18].

Another area where OpenGL hides vital concepts that affect virtually all modern GPUs is command buffer recording. Commands are simply issued on the fly in OpenGL, completely concealing the possibility of recording and reusing chains of commands, let alone the possible performance implications of command recording. In order to remain efficient, the driver usually caches and organizes these commands, again performing vital work in the user's stead. In Vulkan, all of these concepts are revealed to users so that they are forced to think about the motivation for their presence. From a didactic point of view, achieving a better insight into the inner workings of modern graphics devices can never be wrong.

3.6 Vulkan in Advanced Graphics Courses and Research

Advanced graphics courses and research have in common that the student or researcher, respectively, shall have a tool that helps completing a certain task or goal, while assuming that the concepts of the underlying graphics API have already been well understood. The structure of a programming framework being such a tool is totally different from a programming framework whose main purpose is helping to learn the API—such as *Vulkan Launchpad*, used for introductory graphics courses as described in Section 3.4. Desirable properties are that the framework for advanced graphics courses and research shall reduce development times, provide debug mechanisms and convenience features (such as, e.g., shader hot-reloading).

With our framework *Auto-Vk-Toolkit*, we have aimed for the perfect balance between productivity improvements and avoiding unnecessary programming overhead, while still maintaining full flexibility in terms of graphics API usage. *Auto-Vk-Toolkit* is rather tightly coupled to the Vulkan API, providing convenience functions and helpers to access it, and offers to fall back to direct Vulkan usage in cases where *Auto-Vk-Toolkit* does not provide a suitable abstraction. It internally uses another framework, namely *Auto-Vk*, which provides a low-level abstraction layer over the Vulkan API (more precisely over Vulkan-Hpp [VkHpp22]), enabling code that is much more concise and possibly faster to write.

Auto-Vk-Toolkit adds a multitude of useful functionality to it like asset management and deployment, window system integration, input handling, render-loop handling, model and image loading, predefined animation handling code for skinned meshes, code for dividing given geometry into meshlets, serialization, shader hot-reloading, and it includes a library for drawing user interfaces. These functionalities help to reduce development and project setup efforts, letting users focus on rendering algorithm development with the Vulkan API in a more efficient and time-saving manner than using the Vulkan API directly.

For example, a graphics pipeline can be created like shown in Listing 3.4 with just a few LOC, while code that uses the raw Vulkan API for the same task can easily require 80 LOC or more. Listing 3.4 shows that *Auto-Vk* manages to require fewer LOC by establishing some default configuration but still enables further configuration options, which can be added to the function call through C++ variadic templates. Another example is attachment declaration for renderpass creation as shown in Listing 3.5. It allows expressively specifying the attachment usage across different subpasses and also where a resolve operation should happen precisely. The setup code is arguably as concise as it can be for that purpose while not hiding any conceptual low-level details. A final example presented in this chapter is the code for establishing a global memory barrier in a concise manner as shown in Listing 3.6. Also in this case, the corresponding code when interfacing with the raw Vulkan API would require many more LOC.

Listing 3.4: *C++ source code for creation of a graphics pipeline using Auto-Vk. The parameters refer to shader files, specify vertex buffer input binding locations matched with a data format and shader input location, a renderpass, front-facing configuration, and several descriptor bindings to various resources.*

```
using namespace avk; using namespace glm;

// Create a new graphics pipeline with vertex and fragment shader:
auto p = context().create_graphics_pipeline_for(
  // Specify shaders for this pipeline:
  vertex_shader("my_shader.vert"),
  fragment_shader("my_shader.frag"),

  // Specify buffer bindings to target locations:
  from_buffer_binding(0)->stream_per_vertex<vec3>()->to_location(0),
  from_buffer_binding(1)->stream_per_vertex<vec2>()->to_location(1),

  // Use a renderpass created previously:
  renderpass,

  // Further config parameters:
  cfg::front_face::define_front_faces_to_be_counter_clockwise(),

  // Define resource bindings:
  descriptor_binding(0, 0, mMaterials),
  descriptor_binding(1, 0, mUniformsBuffer),
  descriptor_binding(1, 1, mLightsBuffer)
);
```

Listing 3.5: *C++ source code using Auto-Vk for declaring that a framebuffer attachment in a certain format shall be cleared on load, used as depth/stencil attachment in the first subpass, used as input attachment bound to location 1 in the second subpass, used as depth/stencil attachment in the third subpass and resolved to the attachment at index 3 after the third subpass.*

```
using namespace avk;

attachment::declare(vk::Format::eD32Sfloat, on_load::clear,
    usage::depth_stencil >> usage::input(1)
                         >> usage::depth_stencil+usage::resolve_to(3),
    on_store::dont_care)
```

Listing 3.6: *C++ source code using Auto-Vk for creating a global memory barrier that synchronizes a transfer operation (making its write accesses available) with a compute pipeline (ensuring the memory is visible for read access).*

```
using namespace avk;

auto barrier = sync::global_memory_barrier(
  stage::transfer        >> stage::compute_shader,
  access::transfer_write >> access::shader_read
)
```

In research, we have extensively used *Auto-Vk-Toolkit*'s shader hot-reloading feature during development, which saved many hours of development time. The ability to have graphics pipelines created anew while the application is running allows for quick tests and fixes at run time. This is especially helpful if the application performs a precomputation step like dividing input geometry into meshlets and computing bounds per meshlet—both of which are performed in our work on computing conservative bounds for robust culling of meshlets [Unt+21]. Another framework feature that helped with precomputations is the serialization feature provided by *Auto-Vk-Toolkit*: Data is serialized to a file after precomputation and can be deserialized at the next run. Serialization can even be used to speed up model loading: Once loaded successfully, a model and all its textures can be serialized to a binary file. Loading it is much faster than parsing the same model again and loading all its required textures. Our work on fast rendering of parametrically defined objects [Unt+24] uses different graphics pipelines and compute pipeline configurations. The convenience functions for creating them (like shown in Listing 3.4) proved to be valuable for this purpose. We have also used renderpasses with multiple sub passes, attachment usage for which we could conveniently describe as shown in Listing 3.5. Further notable framework features used were GPU timestamp queries and reading back data from them, updating images and the swapchains after window resizes, and automatically inferred synchronization stages and memory accesses for pipeline barriers. The most useful tool during our research was, once again, shader hot-reloading, since it allowed for quick creation of parametric models on-the-fly during run time.

In our advanced graphics course *Algorithms for Real-Time Rendering*, which targets graduate students, we also use *Auto-Vk-Toolkit* for the assignments. The students doing this course are already experienced with graphics API usage. Therefore, using a framework that allows more time-efficient usage of the underlying graphics API allows to focus more on specific course contents instead of graphics API usage—which was learned in introductory courses already. Exercise topics in our advanced course include proper implementation of normal mapping using tangent space, handling a large number of light sources, hardware tessellation, view-frustum and backface culling in tessellation control shaders, dynamically adaptive levels of tessellation, deferred shading, deferred shading in combination with multisample anti-aliasing, manual multisample resolve in compute shaders, tile-based deferred shading, physically based shading, screen-space ambient occlusion, tone mapping, temporal anti-aliasing, and real-time ray tracing using ray query and ray tracing pipelines. While using OpenGL previously for all tasks (except ray tracing), then starting the transition to Vulkan for some tasks while leaving some in OpenGL, we switched to Vulkan completely for all tasks in 2022. This allowed us to address some advanced low-level GPU topics in teaching that were impossible to cover with the OpenGL API: Real-time ray tracing is not supported at all by the OpenGL API, but it is with Vulkan. On a more fundamental level, fine-grained synchronization and its effects can only be analyzed and discussed properly with a low-level API like Vulkan since all the synchronization primitives that would allow fine-grained synchronization are missing from OpenGL. Having the possibility to discuss them enables us to teach low-level concepts which we consider to be very important for acquiring in-depth knowledge about

GPU programming. A further benefit of using the Vulkan API is the explicit and precise specification of when multisample resolve operations are performed by the GPU along with appropriate framebuffer attachment usage and synchronization. While the inclusion of more low-level concepts like buffer sub-allocation would have been interesting topics to cover additionally, we were unable to fit more such topics into the scope of this course—not least because its main focus is on rendering algorithms.

Besides being well suited for teaching purposes, the two frameworks are furthermore intended to be used for rapid prototyping and general Vulkan development. We have successfully used them in our published research, like for our work on exploiting conservative meshlet bounds using graphics mesh pipelines [Unt+21]. Some of their advantageous properties are reflected in student feedback on our advanced graphics courses.

3.7 Conclusion

We successfully employed Vulkan for teaching the use of a real-time graphics API in an introductory course and for teaching selected state-of-the-art techniques and low-level GPU concepts in advanced courses. Abstracting some functionality of early assignments was key to enabling a manageable and fair workload. Flattening the learning curve of Vulkan for first-time graphics API users enabled us to provide a similar challenge as previously established OpenGL assignments. But also in advanced courses, the use of a framework that abstracts the Vulkan API and reduces implementation effort turned out to be crucial to stay within sensible boundaries in terms of student workload. The biggest hurdle for many students in introductory graphics courses was C/C++ usage. More efficient C/C++ learning resources and lectures should allow students to focus better on graphics API usage. The biggest hurdles for students with respect to graphics API usage in our advanced courses seemed to be some of the advanced low-level concepts like synchronization. Since the vast majority (presumably all) of students in advanced courses did not learn graphics programming using the Vulkan API, but instead using a higher-level API like OpenGL, many of them had to learn concepts like fine-grained synchronization at a later point in time in the context of our courses. We think that teaching Vulkan from the start will have a positive effect on our students to become proficient users of modern graphics APIs and, thereby, in more advanced courses when they encounter Vulkan again. Using a low-level API enables students to learn about the massively parallel operation mode of modern GPUs early in their visual computing education. We provide detailed results and evaluations of user studies for both, the introductory and the advanced graphics course in our published work on transitioning to a modern low-level graphics API in academia [UKW23]. Our evaluation has shown that students appreciate the skills and knowledge they picked up through using the Vulkan API. We believe that teaching Vulkan is both viable and beneficial to students who aim to become competent practitioners of visual computing. While the transition may be challenging, it appears to be a worthwhile investment to provide students with future-proof education.

Introductory and advanced courses have different requirements in terms of Vulkan API abstraction level. Therefore, we propose to use *Vulkan Launchpad [Res22]* for the former to hide several complex concepts for first-time users of Vulkan. For the latter, we propose using *Auto-Vk [Res24a]* and *Auto-Vk-Toolkit [Res24b]*, which do not hide any fundamental Vulkan concepts but just aim to make development more efficient and code more concise. These two frameworks also constitute the technological basis for our further contributions in the context of fast rendering of ultra-detailed geometry in real time, i.e., the techniques presented in Chapters 4 and 5.

CHAPTER 4

Conservative Meshlet Bounds for Robust Culling of Skinned Meshes

The contents of this chapter are largely based on our paper "Conservative Meshlet Bounds for Robust Culling of Skinned Meshes", presented at Pacific Graphics 2021, and published in Computer Graphics Forum 40.7 (Oct. 2021) [Unt+21].

4.1 Motivation

While Chapter 2 is concerned with optimal rendering performance of multiple views, we shift the focus to achieving optimal rendering performance for one single view for a previously unsolved problem: Fine-grained culling for animated meshes.

As we describe in Section 1.3, Nanite enables rendering of ultra-detailed static geometry with fine-grained view frustum and backface culling (VFC and BFC) to reach optimal rendering performance. However, support for animated meshes is still lacking in 2024 in the official release of Unreal Engine 5.5 [Epi24a]. The same level of detail is bound to become relevant for animated models in the near future to match the static environment, which is the motivation for the research conducted and described in this chapter.

We describe the essential parts to enable fine-grained culling of geometry clusters for animated models, enabling conservative VFC and BFC on a per-cluster basis. Our technique avoids false positive culling decisions by calculating conservative spatiotemporal bounds for positions and normal orientations of primitive clusters—so-called "meshlets". The precomputed information can be used for efficient VFC and BFC, for example during rendering through a compute shader and software rasterization-centric approach like Nanite [KSW21], but also applies to hardware rasterization on modern GPUs with support for task and mesh shaders [Kub18b].

Figure 4.1: *Skinned models* Gawain[1] *(198k vertices, 261k triangles, 136 bones, 9044 meshlets),* Giant Worm *(169k vertices, 329k triangles, 80 bones, 10382 meshlets),* Butcher *(287k vertices, 477k triangles, 224 bones, 14915 meshlets), and* Wyvern *(267k vertices, 512k triangles, 88 bones, 16015 meshlets), divided into color-coded meshlets, fit for rendering with task and mesh shaders.*

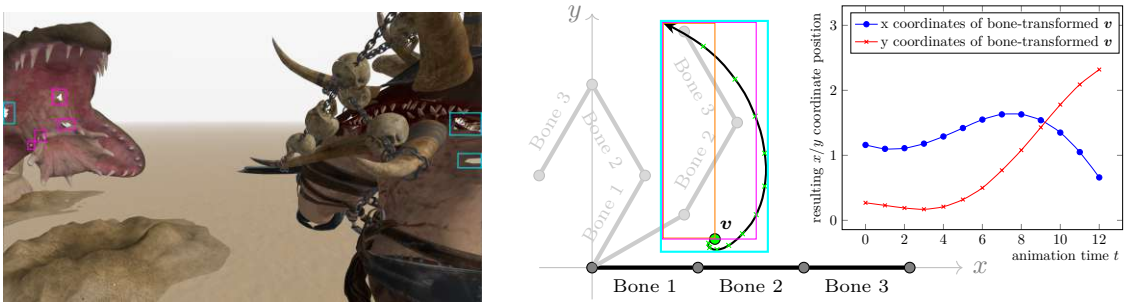
4.2 Introduction

Following recent advances in GPU hardware development and newly introduced rendering pipeline extensions, the segmentation of input geometry into meshlets has emerged as an important practice for efficient rendering of complex 3D models. Meshlets can be processed efficiently using mesh shaders [Kub18b] on modern graphics processing units, in order to achieve streamlined geometry processing in just two tightly coupled shader stages that allow for dynamic workload manipulation in between. The additional granularity layer between entire models and individual triangles enables new opportunities for fine-grained visibility culling methods.

In contrast to static models, VFC and BFC on a per-meshlet basis for skinned, animated models are difficult to achieve while respecting the conservative spatiotemporal bounds that are required for robust rendering results. In this chapter, we describe a solution for computing and exploiting relevant conservative bounds for culling meshlets of models that are animated using linear blend skinning (LBS). By enabling visibility culling for animated meshlets, our approach can help to improve rendering performance and alleviate bottlenecks in the notoriously performance- and memory-intensive skeletal animation pipelines of modern real-time graphics applications.

Task and *mesh* shaders (named *amplification* and *mesh* shaders in DirectX [Mic21], respectively) have been introduced as new shader stages with NVIDIA's Turing microarchitecture to replace the multi-layered geometry processing of a conventional graphics

[1] Provided by Unity Technologies through their "The Heretic: Digital Human" package



(a) *On the borders of the screen, the effect of premature frustum culling can be observed during animations (cyan rectangles). Inside* Giant Worm*'s mouth, some meshlets are prematurely backface-culled as a result of disregarding how the normals of triangles assigned to meshlets change through animation (marked in magenta).*

(b) *The path of a single vertex* $\boldsymbol{v}$*, animated via LBS from one keyframe to the next. The vertex is strongly weighted toward Bone 3. Sampling intermediate positions underestimates the spatial bounds (orange box: start and end, magenta box: two intermediate samples). As multiple rotations are chained, the path of* $\boldsymbol{v}$ *can become arbitrary and coordinate extrema difficult to predict.*

Figure 4.2: *Premature culling can result in very noticeable visual artifacts, as shown in Figure 4.2a. Through animation, the shape of a meshlet can change significantly. Underestimating the vertex bounds can lead to premature view frustum culling, while not taking into account the possible changes in surface normals of a meshlet's triangles can lead to erroneous backface culling. For both, vertex positions and surface normals, merely sampling along an animation interval can be insufficient for calculating conservative bounds, as illustrated in Figure 4.2b.*

pipeline with a more streamlined alternative [NVI18a]. The key point of task and mesh shaders is to allow more fine-grained control over primitive processing and dynamic workload distribution via two tightly coupled compute shader-style geometry processing stages within rasterization-based graphics pipelines, while still utilizing their later hardware-accelerated fixed-function stages. The new setup encourages the division of geometry workload into smaller packages that can be efficiently processed by the GPU. For Nvidida's Turing microarchitecture, optimal efficiency can be achieved by dividing indexed triangle meshes into parts that consist of no more than 64 vertices and 126 triangles [Kub18a]. Each one of these small geometry packets is referred to as a *meshlet*. Since the new shader stages and meshlet-based rendering see wide-spread support in recently released GPUs—including the GPUs utilized in the gaming consoles PlayStation 5, and Xbox Series S and X, which are based on AMD's RDNA 2 microarchitecture [Bla20]—we expect developer adoption and research interest in task and mesh shader-based solutions to rise considerably. The task shader runs early in the graphics pipeline and can serve as a work scheduler for the mesh shader, similar to the dynamic work generation performed by tessellation control shaders in a conventional pipeline. Only meshlets that are not culled by the task shader are processed in the later shader stages. Precomputation of positional and normal bounds for static models can be achieved easily [Wih16]. However,

generation and usage of such information are significantly more difficult to achieve for animated models, since a meshlet's shape can change under animation. Dividing modern skinned 3D models into meshlets can yield results like those shown in Figure 4.1.

Vertex skinning is one of the most popular and widely used animation methods for 3D models. With vertex skinning, vertices are assigned to one or multiple bones of a skeleton in a weighted manner. When the skeleton's bones move into different positions over the course of an animation, assigned vertices move according to their weighting. With respect to meshlets—each of which represents a part of the skin—this means that geometry primitives associated with a meshlet can change their shape since the referenced vertices will have different bone assignments and weights in general. For example, a meshlet could be stretched in one or multiple directions, thus increasing its bounds w.r.t. a resting or configuration pose. Also, since bone assignment and weighting can differ between the vertices of a triangle, animation can produce face normal directions that were not present in the initial pose. Figure 4.2 visualizes artifacts of VFC and BFC that can occur if these bounds are underestimated.

Our proposed method provides a solution for computing conservative spatiotemporal vertex bounds under animation for arbitrary animation clips. We show how conservative bounds for a meshlet's extents and also for its face normals distribution can be computed from the vertex bounds of its associated vertices in a CPU-based precomputation step. Evaluating the precomputed data in task shaders allows for robust VFC and BFC on a per-meshlet basis, extending the advantages of the additional visibility culling granularity of task shaders in rasterization-based graphics pipelines from static meshes to models with skeletal animation. Exploiting the computed bounds to perform robust culling can accelerate rendering and reduce bottlenecks in skeletal animation pipelines on modern GPU architectures while preserving the fidelity of the rendered scenes. In summary, our contributions include the following:

- We derive an adaptive procedure to compute spatiotemporal axis-aligned bounding boxes (AABBs) on a per-meshlet basis for a given interval of an animation clip that are suitable for LBS. Our approach yields conservative bounds over all continuous joint orientations within a given animation and can be parameterized to produce arbitrarily tight bounds.
- We describe a method for efficiently computing the maximum extents described by a given rotation quaternion, based on a derivative of Rodrigues' rotation formula.
- Given spatial bounds for individual mesh vertices, we show that we can obtain a conservative estimate on the maximum deviation of a triangle's surface normal from an initial state during a given animation interval. We apply this concept to entire meshlets to obtain normal cones for robust backface culling with LBS.
- We evaluate our spatiotemporal meshlet bounds for animated models in task and mesh shader-based rasterization, enabling VFC and BFC on a per-meshlet basis. Our methods incur negligible memory overhead and improve run-time performance.

This chapter is structured as follows: in Section 4.3, we discuss relevant background and related work our approach builds on. In Section 4.4, we derive our solution for the adaptive computation of arbitrarily tight vertex bounds. In Section 4.5, we show how this information can be exploited to compute conservative surface normals distributions for animation intervals. Implementation details are described in Section 4.7. Sections 4.6 and 4.8 assess the usage of the computed bounds for visibility culling and resulting performance impact. We discuss possible extensions of the presented approach as well as open problems in Section 4.9.

4.3 Related Work

Vertex skinning plays an integral role in the animation of characters in visual applications. The fundamental concept of connecting vertices with manually or automatically defined weights to an underlying skeleton can be implemented in a variety of ways [MLT89; BP07]. Arguably, one of the most widespread methods to this day is LBS, in which the final animated position of each vertex is the result of a linear combination of independently computed results. Obvious shortcomings and frequent artifacts (e.g., the "candy-wrapper" effect) gave rise to alternative approaches, such as spherical blend skinning [KŽ05b] and log-matrix skinning [CM04], which manage to eliminate some of these artifacts, but introduce others and exhibit a higher performance penalty. In contrast, dual-quaternion skinning is comparable to LBS in terms of performance, while resolving most of its issues [Kav+07]. Instead of opting for mathematically involved solutions at runtime, the selection of optimized centers of rotation as an isolated preparatory step for animation has been suggested by Le and Hodgins [LH16]. While pursuing either of these methods would be worthwhile, we will only consider the fastest of these methods, LBS, for the derivation of our conservative bounds.

The task of computing conservative spatiotemporal bounds for meshes—or more generally the vertices and faces of its meshlets—undergoing skeletal animation is more challenging than it may seem at first glance. A key requirement for conservative bounds which are suitable for visibility culling is that they encompass all possible positions that vertices can occupy, as well as all possible face normal orientations that can emerge during an animation clip or subintervals thereof. Clearly, these challenges are related to the field of collision detection. A wide range of efficient solutions exists for this topic, which requires computing positional bounds for particular instants or ranges. *Reduced deformation* solutions effectively decouple bound computations from the geometry and perform them on influencing factors only (e.g., bones) before applying them to entire clusters of primitives. Several such approaches present solutions that target individual animation frames [KŽ05a; SBT06; SOG08], but not the interval in-between. Furthermore, their application usually entails a non-negligible run-time cost for computing and updating bound information, which impedes complex animated scenes with many differently animated (e.g., temporally offset) models. The same is true for established reduced deformation approaches for bounding normals, such as normal trees [SGO09]. Temporally continuous collision detection (e.g., swept volume approaches) have been applied to rigid objects [AA00;

Kim+03; RLM04], but remain a challenging problem for deformable meshes, which we target in our work. Specialized methods for reduced deformable models, such as BD-Trees [JP04] provide excellent opportunities to accelerate bound queries for entire primitive groups. However, they require precomputed displacement fields and imply the creation and maintenance of hierarchical data structures, which we strive to avoid in order to minimize run-time overhead in complex animated scenes. Although spatial bounds may be approximated from analysis of the model data, available proprietary solutions make no claim about computed bounds being conservative [Uni21], while sampling-based methods can easily miss extreme positions and orientations and provide no guarantee for robustness [Gün+06], so that they can still lead to undesirable artifacts such as those described in Figure 4.2a.

Task and mesh shaders were first introduced with the NVIDIA Turing architecture [NVI18a]. The new shader stages can be used as alternative geometry processing stages within rasterization-based graphics pipelines. Consequently, the usage of task and mesh shaders, and the usage of classical geometry processing shaders (vertex, tessellation, and geometry shaders) are mutually exclusive. The data structures of uniforms and buffers to be used with task and mesh shaders can be freely defined. It is common practice to prepare auxiliary information about a meshlet's bounds and normals distribution and evaluate that information in task shaders for visibility culling. The standard usage conventions and restrictions regarding possible input geometry clusters (meshlets) are defined by the corresponding Khronos conventions [Kub18b; KB19]. Similar mechanisms and rule sets are set to become adapted by competing hardware vendors in the near future.

With Turing, NVIDIA also introduced GPU acceleration structures and real-time ray tracing pipelines (meanwhile standardized and defined by the Khronos conventions [Koc20a]). While task and mesh shaders are strictly limited to rasterization-based graphics pipelines and hence not usable with ray tracing pipelines, the usage of acceleration structures (enabled for usage in any shader stage through Khronos convention [Koc20b]) might appear as a possible option. Current real-time graphics APIs, however, only support ray-based access to the hardware-accelerated ray tracing data structures [Koc21] and hence do not suit rasterization.

Although meshlets as input for the hardware rendering pipeline have only been recently introduced as a topic for computer graphics, similar concepts have been proposed previously. The idea of clustering geometry is a fundamental theme in high-performance rendering of complex scenes, along with possible opportunities for optimization of their visualization at runtime [Ura19; HA15; Sho+08]. Before the introduction of mesh shaders, Kerbl et al. documented the exact rule sets used by modern GPUs to partition triangle meshes into batches before processing them in bulk [Ker+18]. In addition to rendering, the generation of meshlets ahead of time is a relevant topic in itself. Common solutions include the application of mesh optimizers, which can be easily adapted to produce basic meshlets instead [LY06; HS17; Hop99; Kap21; Wal21]. While these can account for basic qualities of meshlets, such as high connectivity and spatial compactness, they cannot

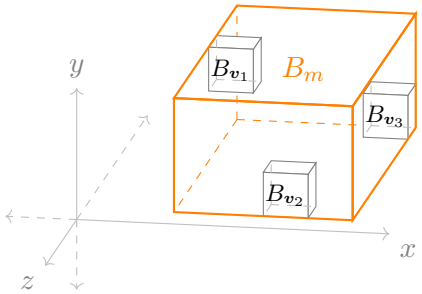


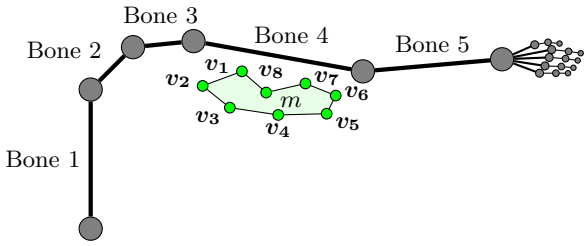
Figure 4.3: *Individual per-vertex AABBs B_{v_1}, B_{v_2}, and B_{v_3} are combined into a common bounding box B_m by taking the minimum and maximum coordinates from all AABBs' corners. B_m represents the conservative bounds of meshlet m.*

ensure specialized target criteria, such as tight bounds under animation, which is one of the targeted characteristics in this chapter. The idea of using geometry clusters to perform efficient visibility culling has been previously pursued, though usually with a clear focus on static geometry and applications in industry [AHA15; HC11]. Position bounds for view frustum and occlusion culling are trivially derived from the extrema found in the set of vertices in each meshlet. For backface culling, a normal cone is required, which can easily be built from the set of surface normals present in the meshlet, although finding the optimal cone is a more challenging task. Recently, Wihlidal proposed methods for generating static geometry clusters to maximize their likelihood of getting culled for visibility [Wih16]. In this chapter, we will expand on this concept to enable high-performance visibility culling for meshlets of animated models.

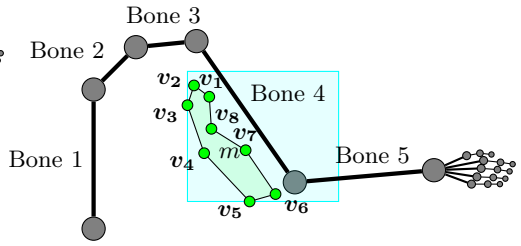
4.4 Meshlet Bounds Computation

In this section, we describe an algorithm to analytically compute conservative bounds per meshlet. At first, vertex bounds are calculated. Subsequently, the bounds of a meshlet can be easily computed by combining all its associated vertices' bounds into a common bounding box, which is exemplarily shown in Figure 4.3.

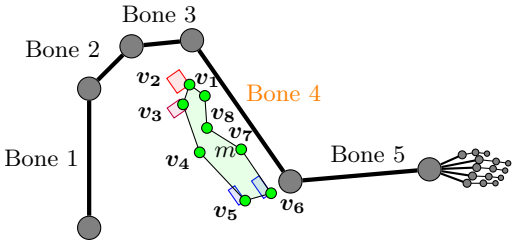
Vertex skinning transforms a vertex $\boldsymbol{v}$ according to its weighted assignment to the bones of an underlying skeleton. As multiple bones can have influence on $\boldsymbol{v}$, we first compute one AABB $B_{\boldsymbol{v}b_i}$ per influencing bone b_i and combine them in a second step into an AABB $B_{\boldsymbol{v}}$ which represents **conservative spatiotemporal bounds** for $\boldsymbol{v}$ under LBS. The temporal aspect of $B_{\boldsymbol{v}}$ refers to the animation time interval that we compute it for. One natural choice for the animation time interval is the span between two keyframes. We assume the transformation between two successive keyframes to be specified with a triple of translation, rotation, and scaling values which are interpolated in between. The skeleton depicted in Figures 4.4a and 4.4b shall serve as an example for two different keyframe times. In Figure 4.4b, Bone 4 has rotated 45° clockwise (CW) w.r.t. its parent bone, and Bone 5 has rotated 45° counter-clockwise (CCW) w.r.t. Bone 4 compared to



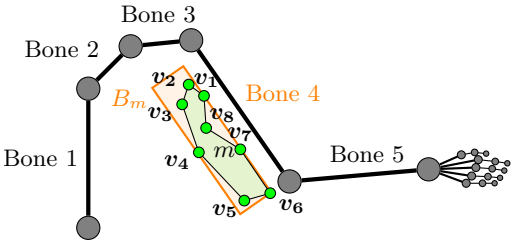
(a) *The vertices of a skinned mesh are assigned to one or multiple bones with different individual weightings each. Vertices* $\boldsymbol{v_1} \dots \boldsymbol{v_8}$ *are combined to form meshlet* m*.*



(b) *Bones are transformed by an animation (Bone 4 rotated by 45° CW, Bone 5 by 45° CCW w.r.t. 4.4a), associated vertices are transformed according to their weights. As vertex weights vary, the shape of* m *changes. The cyan box encompasses all vertex positions between the poses in 4.4a and 4.4b.*



(c) *Each vertex' bounding box encompasses all vertex positions between the two different poses, relative to the coordinate system defined by Bone 4. The boxes of vertices* $\boldsymbol{v_1}$*,* $\boldsymbol{v_4}$*,* $\boldsymbol{v_7}$*, and* $\boldsymbol{v_8}$ *are infinitely small, which indicates that their weight w.r.t. Bone 4 is one. Vertices* $\boldsymbol{v_5}$ *and* $\boldsymbol{v_6}$ *are influenced by Bones 4 and 5 (blue bounding boxes). Bone 3 influences the position of* $\boldsymbol{v_2}$ *and* $\boldsymbol{v_3}$*, and Bone 2 has minor influence on* $\boldsymbol{v_3}$ *(red and purple boxes).*



(d) *Combining all vertex bounding boxes which are axis-aligned w.r.t. the coordinate system of Bone 4 to a common bounding box can be trivially computed like described in Figure 4.3. Performing this procedure yields conservative spatio-temporal bounds* B_m *of meshlet* m *for the animation between the two different skeleton poses when animated from 4.4a to 4.4b.*

Figure 4.4: *In this example, meshlet* m *represents a part of a skinned mesh's skin. It consists of vertices* $\boldsymbol{v_1} \dots \boldsymbol{v_8}$ *each of which has one or multiple weighted bone assignments. According to those, vertices are moved when the skeleton's pose changes between the state in Figure 4.4a and the state in Figures 4.4b to 4.4d. The final bounding box* B_m *is computed relative to the coordinate system of the bone which has the highest combined influence on the vertices—which is Bone 4 in this case, the principal bone of meshlet* m*.*

the state in Figure 4.4a. All child bones of Bone 5 inherit their parent's transformation and therefore change their *global* position, too. Their *local* transformations, however, stay constant between the two keyframes. Furthermore, Figure 4.4 shows meshlet m which has several vertices assigned to it. Each vertex has weighted assignments to one or multiple bones. Whenever the skeleton is animated into a certain position, the vertices get transformed accordingly, leading to different shapes of m in Figures 4.4a and 4.4b.

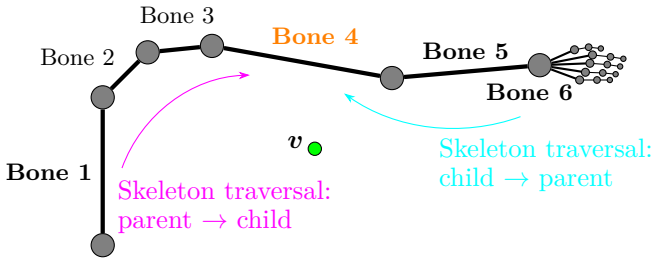


Figure 4.5: *Assume a vertex $\boldsymbol{v}$ with non-zero weights w.r.t. four bones: Bone 1, Bone 4, Bone 5, and Bone 6. To compute the combined bounds of $\boldsymbol{v}$ w.r.t. a given target bone (Bone 4), we compute a bounding box for each bone that influences $\boldsymbol{v}$ during our algorithm's first step, accumulating transformations along the bone hierarchy towards the target bone. To compute bounds w.r.t. a single bone, we start at that bone and traverse the skeleton until we reach the target. When computing the bounds of B one 1, we must regard the transformations of Bones 2 and 3 in the given animation interval, even if they have no direct influence on $\boldsymbol{v}$. Similarly, when computing the bounds of Bone 6, we must include the transformations of Bone 5. Even though Bone 5 is contained in the path from Bone 6 to Bone 4, separate bounding boxes for Bone 5 and Bone 6 must be computed.*

We propose to compute and store the bounding box B_m of meshlet m in the space of the most influential bone. We find this bone on a per-meshlet basis by summing all the normalized weightings of all vertices that are assigned to m per bone. The bone with the highest sum of normalized weights is deemed to be the most influential bone and thus we assign it to m as its **principal bone**. The reasoning behind this approach is that B_m can be assumed to be of minimal extent if it is computed in the space that has the most influence on the assigned vertices. Relative to the principal bone's space (PBS), the bounds of the majority of the associated vertices can be assumed to be smaller than in other spaces. This is illustrated in Figure 4.4 where it can be observed that the bounds in m's PBS (depicted in Figure 4.4d) are smaller than the bounds computed relative to mesh space or world space (depicted in Figure 4.4b).

4.4.1 Vertex Bounds Computation

The most computation-intensive part of our algorithm is conservative vertex bounds calculation in a given target bone space—referring to the PBS of a meshlet, which is the most influential bone's coordinate system as described in Section 4.4. Our algorithm consists of the following major steps for computing the bounds of a specific vertex $\boldsymbol{v}$ in the target bone's coordinate system (referred to by bone b_t) between two animation times t_1 and t_2. Animation times must not stretch over keyframe time boundaries but must be limited to keyframe bounds, otherwise sudden changes in transformations—which must be expected for subsequent keyframed intervals—run the risk of missing extreme positions and thus, fail to remain conservative. For sub-keyframe intervals, t_1 and t_2 can be chosen arbitrarily narrowly. Furthermore, $\boldsymbol{v}$ has a list of associated bones $I_{\boldsymbol{v}} = \{b_0, \ldots, b_n\}$,

where each one of these mappings has a respective weight $W_{\boldsymbol{v}} = \{w_0, \ldots, w_n\}$ assigned. They satisfy the conditions $w_i \in [0, 1]$ and $\sum_{w \in W_{\boldsymbol{v}}} w = 1$.

1. For each bone $b_i \in I_{\boldsymbol{v}}$, compute $\boldsymbol{v}$'s conservative spatiotemporal bounding box $B_{\boldsymbol{v}b_i}$ between t_1 and t_2 with full weight (i.e., as if $w_i = 1$) in the coordinate system of b_t. In more detail, for each bone $b_i \neq b_t$ the procedure is like follows:
   a) Transform $\boldsymbol{v}$ into the coordinate system of b_i and apply b_i's *local* scale, rotation, and translation transformations at animation time t_1. Initialize $B_{\boldsymbol{v}b_i}$ by setting its minimum and maximum coordinates to the result, yielding an AABB with zero volume in the local space of b_i.
   b) Extend $B_{\boldsymbol{v}b_i}$ by the b_i-*local* scale, rotation, and translation transformation *differences* between t_1 and t_2.
   c) Traverse to node b_j which is one node closer towards b_t from b_i. Break if b_t has been reached, otherwise loop as follows:
      i. Transform every corner of $B_{\boldsymbol{v}b_i}$ into the coordinate system of b_j, and construct a new AABB $B_{\boldsymbol{v}b_j}$ there.
      ii. Apply b_j's *local* scale, rotation, and translation transformations at animation time t_1 to every corner of $B_{\boldsymbol{v}b_j}$. Use the transformed corners to construct a new AABB $B'_{\boldsymbol{v}b_j}$.
      iii. Extend $B'_{\boldsymbol{v}b_j}$ by the b_j-*local* scale, rotation, and translation transformation *differences* between t_1 and t_2.
      iv. Assign $b_i = b_j$ and $B_{\boldsymbol{v}b_i} = B'_{\boldsymbol{v}b_j}$, let b_j refer to the next node which is one step closer towards b_t. Break if b_t has been reached, otherwise loop.
2. Combine all $B_{\boldsymbol{v}b_i}$ based on their respective weights w_i into the vertex' conservative spatiotemporal bounding box $B_{\boldsymbol{v}}$ that represents all positions that $\boldsymbol{v}$ can occupy between t_1 and t_2.

Traversing the bone hierarchy step-wise is a crucial property of our algorithm and a necessity for computing conservative vertex bounds for an animation interval. Decomposition of a global, affine bone matrix would not be a viable solution since extreme positions and orientations from intermediate steps could be missed. Care must be taken regarding the skeleton traversal direction when moving bone-by-bone towards b_t as illustrated in Figure 4.5. If Equation (4.1)

$$\boldsymbol{v'} = \boldsymbol{P_{b_i}}\ \boldsymbol{T}\ \boldsymbol{R}\ \boldsymbol{S}\ \boldsymbol{v} \tag{4.1}$$

transforms from a child's bone space into its parent's bone space (with $\boldsymbol{T}$ referring to the translation, $\boldsymbol{R}$ to the rotation, $\boldsymbol{S}$ to the scaling, and $\boldsymbol{P_{b_i}}$ being a constant matrix that positions bone b_i relative to its parent bone), the transformation in the inverse direction must be reformulated as stated in Equation (4.2)

$$\boldsymbol{v'} = (\boldsymbol{P_{b_i}}\ \boldsymbol{T}\ \boldsymbol{R}\ \boldsymbol{S})^{-1}\ \boldsymbol{v} = \boldsymbol{S}^{-1}\ \boldsymbol{R}^{-1}\ \boldsymbol{T}^{-1}\ \boldsymbol{P}_{\boldsymbol{b_i}}^{-1}\ \boldsymbol{v} \tag{4.2}$$

to correctly apply the separate transformations step-wise. We chose matrix notation for the sake of brevity and clarity, but different forms are practicable as well—most notably using unit quaternions for applying rotations.

If an AABB $B_{\boldsymbol{v}b_j}$ is represented by two vectors—one for its minimum coordinates and the second for its maximum coordinates—it can be transformed conservatively as follows:

- $B_{\boldsymbol{v}b_j}$ is translated by adding the translation vector to its minimum and maximum coordinates.
- $B_{\boldsymbol{v}b_j}$ is rotated by rotating each of its corners and constructing a new AABB $B'_{\boldsymbol{v}b_j}$ from the results.
- $B_{\boldsymbol{v}b_j}$ is scaled by component-wise multiplication of its minimum and maximum coordinates with the scaling vector.
- $B_{\boldsymbol{v}b_j}$ is transformed by a matrix by constructing a new AABB $B'_{\boldsymbol{v}b_j}$ from the matrix-transformed corners.

Extending the bounds by translation, rotation, and scale values as required in steps 1b and 1(c)iii of our algorithm can be trivially computed for translation and scaling, but not for rotations:

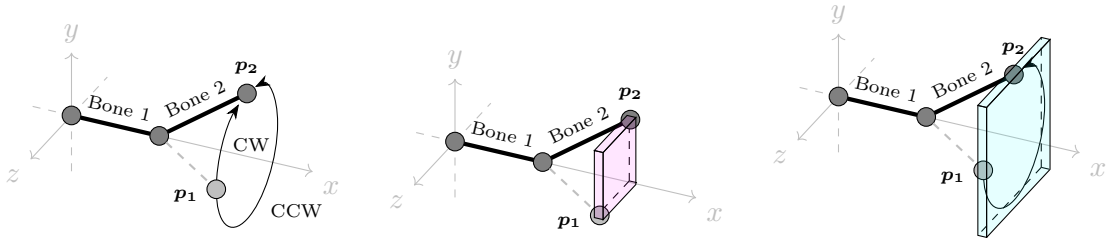
- $B_{\boldsymbol{v}b_j}$ is extended by translation through extending the bounding box by both, the translated minimum and maximum coordinates.
- $B_{\boldsymbol{v}b_j}$ is extended by scaling through extending the bounding box by both, the scaled minimum and maximum coordinates.
- $B_{\boldsymbol{v}b_j}$ is extended by rotation through the method described in Section 4.4.2.

4.4.2 Computing the Maximum Rotation Extents

For rotations, we need to take situations like those depicted in Figure 4.2b into consideration and prevent missing any possible location that a rotated point could occupy. Figure 4.6 illustrates this problem with a different example: Rotating Bone 2 about the axis described by its parent Bone 1 lets a rotated point $\boldsymbol{p_1}$ end up in a certain end position $\boldsymbol{p_2}$. Spanning a bounding box only over the initial and final positions, $\boldsymbol{p_1}$ and $\boldsymbol{p_2}$, can lead to incorrect boundaries. Figure 4.6c shows the correct bounding box for the given scenario which can only be computed by regarding all possible locations along the circular segment described by the rotation, or—in the case of axis-aligned data structures—by considering those particular rotations that lead to maximum extents in each of the principal axes' directions.

To find and efficiently compute the maximum extents of a given rotation, we present a solution based on *Rodrigues' rotation formula* [Rod40], which computes the result $\boldsymbol{v'}$ of rotating a vector $\boldsymbol{v}$ by a given angle θ about a given (normalized) axis of rotation $\boldsymbol{n}$. Rotation transforms within a skeleton are often specified via unit quaternions which can be converted into angle-axis representation [Sho85]. Thus, Rodrigues' rotation formula is applicable. It is stated in Equation (4.3):

$$\boldsymbol{v'} = \boldsymbol{v}\cos\theta + (\boldsymbol{n} \times \boldsymbol{v})\sin\theta + \boldsymbol{n}(\boldsymbol{n} \cdot \boldsymbol{v})(1 - \cos\theta). \tag{4.3}$$



(a) *When rotating a certain point from an initial position* $\boldsymbol{p_1}$ *to its target position* $\boldsymbol{p_2}$*, the rotation can be performed either in CW direction or in CCW direction about the axis along Bone 1.*

(b) *To compute a conservative bounding box that encompasses all possible positions that the rotated point can take along the rotation path, it is insufficient to consider its initial/-final positions.*

(c) *If an animation is defined s.t.* $\boldsymbol{p_1}$ *is rotated to* $\boldsymbol{p_2}$ *in CCW rotation direction, its actual conservative spatiotemporal bounding box is much larger than the incorrect one depicted in Figure 4.6b.*

Figure 4.6: *These illustrations show that it is essential to regard the actual rotation paths for computing conservative spatiotemporal AABBs. It is insufficient to only consider initial and final positions since the maximum rotation extents might occur elsewhere.*

We use its first-order derivative by θ to find those angles that lead to maximum extents in each of the principal axes' directions. Setting that first-order derivative of Equation (4.3) by θ to zero in order to find the extrema results in Equation (4.4)

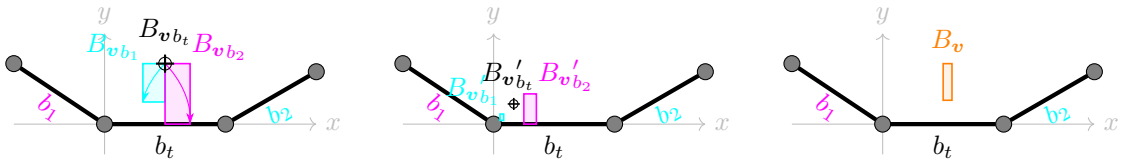
$$\boldsymbol{x}_\theta = -\tan^{-1}\frac{\boldsymbol{n}\times\boldsymbol{v}}{\boldsymbol{n}(\boldsymbol{n}\cdot\boldsymbol{v})-\boldsymbol{v}}, \tag{4.4}$$

which yields a vector of angles $\boldsymbol{x}_\theta$ in radians that represents the rotation angles that lead to maximum extents in each principal axis direction. Please note that the operations in Equation (4.4) mean component-wise application of the division and $\tan^{-1}$.

An AABB $B_{\boldsymbol{v}}$ can be extended to encompass all the possible positions of $\boldsymbol{v}$ rotated by angle θ about axis $\boldsymbol{n}$ by calculating nine rotation angles:

- $\phi_1 = \text{clamp}(\boldsymbol{x}_{\theta x}\quad\quad, \min(\theta,0), \max(0,\theta))$
- $\phi_2 = \text{clamp}(\boldsymbol{x}_{\theta x}-\pi, \min(\theta,0), \max(0,\theta))$
- $\phi_3 = \text{clamp}(\boldsymbol{x}_{\theta x}+\pi, \min(\theta,0), \max(0,\theta))$
- $\phi_4 = \text{clamp}(\boldsymbol{x}_{\theta y}\quad\quad, \min(\theta,0), \max(0,\theta))$
- $\phi_5 = \text{clamp}(\boldsymbol{x}_{\theta y}-\pi, \min(\theta,0), \max(0,\theta))$
- $\phi_6 = \text{clamp}(\boldsymbol{x}_{\theta y}+\pi, \min(\theta,0), \max(0,\theta))$
- $\phi_7 = \text{clamp}(\boldsymbol{x}_{\theta z}\quad\quad, \min(\theta,0), \max(0,\theta))$
- $\phi_8 = \text{clamp}(\boldsymbol{x}_{\theta z}-\pi, \min(\theta,0), \max(0,\theta))$
- $\phi_9 = \text{clamp}(\boldsymbol{x}_{\theta z}+\pi, \min(\theta,0), \max(0,\theta))$

and extending $B_{\boldsymbol{v}}$ by the results of Equation (4.3), calculated with vector $\boldsymbol{v}$, (normalized) axis of rotation $\boldsymbol{n}$, and each of these nine rotation angles. The angles are clamped to the



(a) *All AABBs have been transformed into the coordinate system of bone b_t. The arrows indicate that the bounding boxes were created based on a certain rotation relative to b_t. $B_{\boldsymbol{v}b_t}$ is an AABB with zero volume.*

(b) *Based on bone weights $w_1 = \frac{1}{2}$, $w_t = \frac{1}{3}$, and $w_2 = \frac{1}{6}$, intermediate AABBs $B'_{\boldsymbol{v}b_1}$, $B'_{\boldsymbol{v}b_t}$, and $B'_{\boldsymbol{v}b_2}$ are calculated by multiplying each original AABB's minimum and maximum coordinates by its respective weight.*

(c) *For an AABB that represents the combined conservative spatio-temporal bounds $B_{\boldsymbol{v}}$ for vertex $\boldsymbol{v}$ under LBS, all the intermediate bounds' ($B'_{\boldsymbol{v}b_1}$, $B'_{\boldsymbol{v}b_t}$, and $B'_{\boldsymbol{v}b_2}$) resp. minimum and maximum coordinates are summed up.*

Figure 4.7: *Three bone-specific spatiotemporal AABBs are shown which were created by our algorithm to include all possible positions vertex $\boldsymbol{v}$ can occupy between two animation times. Bone-specific vertex AABBs $B_{\boldsymbol{v}b_1}$, $B_{\boldsymbol{v}b_t}$, and $B_{\boldsymbol{v}b_2}$ are combined into AABB $B_{\boldsymbol{v}}$ by taking the bone weights of vertex $\boldsymbol{v}$ into account and following the steps described in 4.7a, 4.7b, and 4.7c in that order. $B_{\boldsymbol{v}}$ represents conservative bounds for LBS.*

range $[\theta, 0]$ or $[0, \theta]$, depending on the sign of θ, to keep the resulting bounding box as tight as possible—yet conservative—around the original position $\boldsymbol{v}$.

4.4.3 Vertex Bounds Combination for LBS

The steps described in Section 4.4.1 yield a number of conservative spatiotemporal vertex bounds $B_{\boldsymbol{v}b_i} \dots B_{\boldsymbol{v}b_n}$, each representing all possible positions between two animation times t_1 and t_2 as if the respective bone was the single bone of influence (i.e., if it had a weighting of 1) on $\boldsymbol{v}$. Each $B_{\boldsymbol{v}b_i}$ is given in the same space, namely the coordinate system of a uniformly selected principal bone for all the vertices associated to meshlet m, which we called PBS.

We propose the approach depicted in Figure 4.7 for computing the combined, weighted vertex bounds $B_{\boldsymbol{v}}$ that are suitable for LBS:

1. For each vertex' AABB $B_{\boldsymbol{v}b_i}$, multiply its minimum and maximum coordinates with its corresponding bone-weighting w_i.
2. Add the resulting minimum and maximum coordinates of all $B_{\boldsymbol{v}b_i} \dots B_{\boldsymbol{v}b_n}$ as computed in step 1, which yields $B_{\boldsymbol{v}}$.

The final step of our algorithm for computing conservative spatiotemporal meshlet bounds B_m is the combination of its associated vertices' AABBs $B_{\boldsymbol{v}_1} \dots B_{\boldsymbol{v}_n}$ as illustrated in Figure 4.3.

4.5 Normals Distribution of Meshlets

In the previous Sections 4.4.1 to 4.4.3, we have addressed the computation of conservative spatiotemporal meshlet bounds which can be used to enable view frustum culling. In this section, we describe how these bounds can be utilized to compute a conservative estimation for a meshlet's normals distribution to ultimately enable backface culling. Our algorithm consists of the following steps:

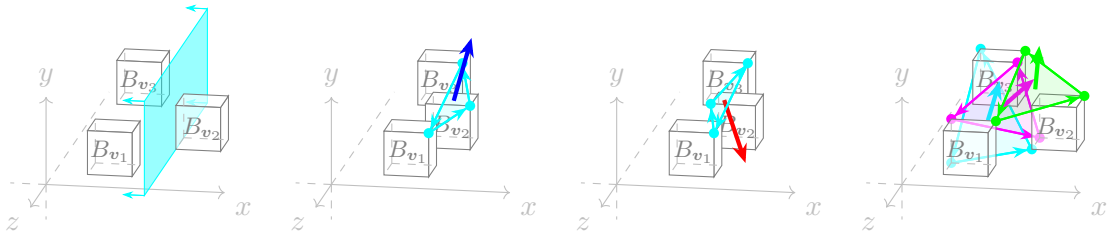
1. Determine an initial normal $\boldsymbol{n}_m$ and an initial normals distribution angle α w.r.t. $\boldsymbol{n}_m$ for meshlet m.
2. For each triangle associated to m, consider its vertices' conservative spatiotemporal AABBs $B_{\boldsymbol{v}_1}$, $B_{\boldsymbol{v}_2}$, and $B_{\boldsymbol{v}_3}$.
3. Optionally: Test if a plane can be found which divides space s.t. $B_{\boldsymbol{v}_i}$ and the respective other two AABBs lie on opposite sides of it. If such a plane cannot be found, abort normals distribution calculation for m, otherwise continue.
4. For each combination of corners $c_{\boldsymbol{v}_1} \in B_{\boldsymbol{v}_1}$, $c_{\boldsymbol{v}_2} \in B_{\boldsymbol{v}_2}$, $c_{\boldsymbol{v}_3} \in B_{\boldsymbol{v}_3}$:
   a) Compute $\boldsymbol{n}_i = (c_{\boldsymbol{v}_2 i} - c_{\boldsymbol{v}_1 i}) \times (c_{\boldsymbol{v}_3 i} - c_{\boldsymbol{v}_2 i})$
   b) Compute the angle between $\boldsymbol{n}_i$ and $\boldsymbol{n}_m$, and store the maximum angle α_{max} from all combinations of corners.
5. Store α_{max} and use it for backface culling computations for m.

Initial $\boldsymbol{n}_m$ and α values (step 1) are computed from a resting or configuration pose. By taking the maximum angle during step 4b, a conservative normals distribution is calculated for a given animation interval. As an optional, potentially performance-improving step 3, we propose the approach outlined in Figure 4.8: If a set of $B_{\boldsymbol{v}_1}$, $B_{\boldsymbol{v}_2}$, and $B_{\boldsymbol{v}_3}$ does not fulfill the requirement described in Figure 4.8a, their normals distribution might encompass the whole sphere of normals. If the requirement is fulfilled, we can be sure that a useful α_{max} can be found as illustrated in Figure 4.8d.

4.6 Rendering Strategies Using Meshlet Bounds for Culling

Computing meshlet bounds as described in the previous sections is a computationally elaborate task that requires computation times in the order of milliseconds to seconds per time interval for models of similar detail to our test models. It is parallelizable and suitable for multithreaded CPU implementations as well as GPU implementations. Computation times for calculating vertex and normal bounds are given in Table 4.1 for each model.

Hence, we propose using a precomputation step for computing the meshlet bounds. However, another essential question is how the precomputed bounds (defined per meshlet, animation clip, and time interval) shall be used during rendering. One possible application is storing the AABBs in GPU buffers and computing the correct lookup index in a task



(a) *We can compute a useful normals distribution if, for each AABB, we can find a plane so that it lies on one side of the plane, and the other two AABBs lie on the other side.*

(b) *If vertex AABBs are positioned in an unfavorable way w.r.t. each other, we cannot find a useful normals distribution (compare with Figure 4.8c).*

(c) *For suboptimally positioned bounding boxes, different combinations of $B_{\boldsymbol{v}_1}$'s, $B_{\boldsymbol{v}_2}$'s, and $B_{\boldsymbol{v}_3}$'s corners lead to normals pointing in opposite directions.*

(d) *If AABBs are positioned favorably, we can use the 8^3 combinations of $B_{\boldsymbol{v}_1}$'s, $B_{\boldsymbol{v}_2}$'s, and $B_{\boldsymbol{v}_3}$'s corners to compute extreme normal deviations for conservative meshlet bounds.*

Figure 4.8: *A conservative normals distribution of a meshlet can be found by analyzing each of its triangles separately. For each triangle, a total number of 8^3 possible normal directions are created by computing the face normals of each triangle that can be constructed from a combination of the bounding box corners from the vertices that describe the triangle, as illustrated in Figure 4.8d. In this way, extreme normal deviations are computed and put in relation to a reference normal $\boldsymbol{n}_m$. The triangle-specific order among $B_{\boldsymbol{v}_1}$, $B_{\boldsymbol{v}_2}$, and $B_{\boldsymbol{v}_3}$ must be maintained for these computations. A conservative test of whether a useful normals distribution can be calculated is presented in Figure 4.8a. If the vertices' AABBs are positioned in an unfavorable manner w.r.t. each other, the normals distribution might encompass the whole sphere of directions as described in Figures 4.8b and 4.8c.*

shader. Taking the current animation time into account, we may then read the AABB from the buffer and evaluate the bounds against the current view frustum. While this might appear like a feasible idea at first glance, it is a strategy that we advise against for several reasons: Animated models can contain a large number of keyframes, easily ranging in the hundreds or thousands. Since memory consumption would be in linear relation with the number of keyframes, this would lead to extensive memory usage and incur additional delays in task shader executions for the memory transfer.

As a better strategy, we propose to use a precomputation step to answer a simple question for each meshlet, namely: "Across the entirety of an animation clip, what is the maximum deviation of a meshlet's bounds w.r.t. certain predefined reference bounds?". This approach is illustrated in Figure 4.9 for spatial bounds. We choose the bounding sphere that encompasses all of a meshlet's vertices in its initial "bind pose" or "T-pose" as the fixed reference per meshlet. When bone-animated, the sphere's center and radius are transformed with the principal bone's transformation matrix, which in general can lead to states where the transformed bounding sphere no longer encompasses all its assigned vertices as illustrated in Figure 4.9b. Based on our conservative spatiotemporal

Table 4.1: *This table shows the average computation times for computing AABBs for all meshlets for a given time interval per model. The computations were performed with 24 parallel threads on an AMD Zen 2 CPU at 3.8 GHz. Times are reported in seconds for the creation of vertex and normal bounds.*

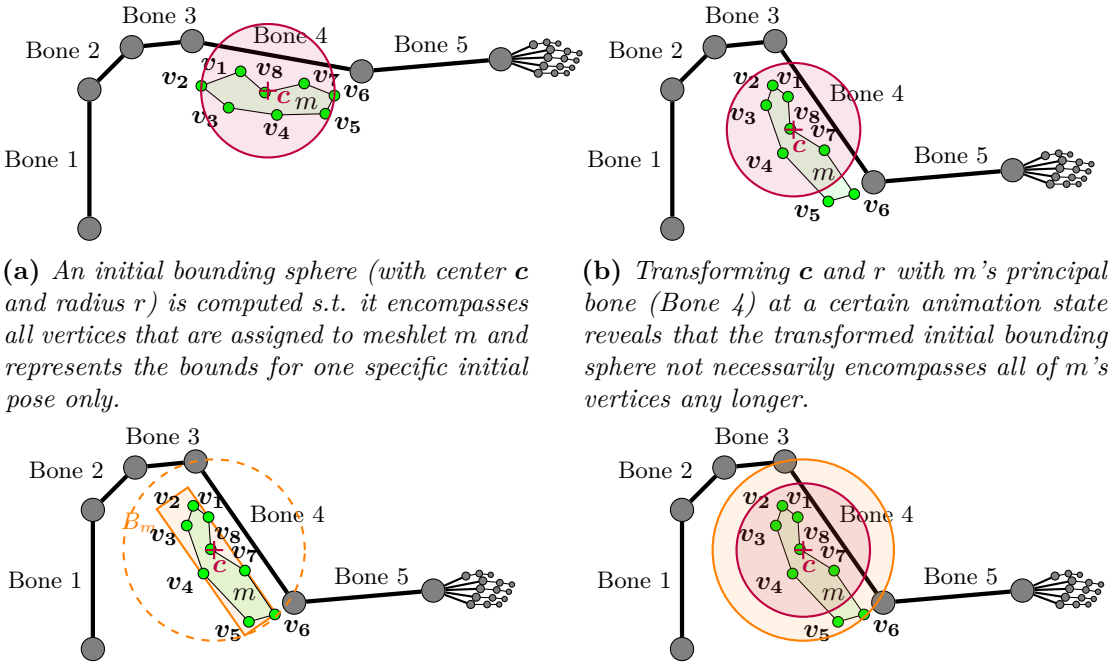
Model	Spatial Extents	Orientations	Total Time
Gawain	0.25s	0.19s	0.44s
Giant Worm	0.37s	0.28s	0.65s
Butcher	0.19s	0.40s	0.59s
Wyvern	0.56s	0.45s	1.00s

AABB B_m for that specific animation state (computed as described in Figure 4.4), we can compute a factor by how much the transformed reference bounding sphere's radius has to be extended in order to also encompass all positions of B_m. Having computed the maximum required radius across all time intervals of an animation clip, it suffices to store a meshlet's initial bounding sphere's center point $\boldsymbol{c}$ and the extended radius per meshlet.

During rendering, very little computational overhead is required: $\boldsymbol{c}$ and its extended radius are transformed by the meshlet's principal bone matrix for the current animation state, further transformed into the same space of the view frustum's planes, and tested against the frustum planes. We might also use AABBs for spatial bounds during rendering, but spheres incur significantly less computational overhead when culling against a view frustum. Using spheres, only six plane-to-point distance computations (one for each of the six view frustum planes) have to be performed in a task shader, whereas bounding boxes demand computation of 6×8 plane-to-point distances.

The quality of the radius extension factor can be further improved by analyzing ever-smaller time intervals. The largest possible time interval to evaluate is from one keyframe time t_1 within a certain animation clip to its subsequent keyframe time t_2, because we may not jump over diverging transformations in order to remain conservative. If we determine that the spatiotemporal bounds between the two keyframes do not satisfy given quality requirements, we can start to adaptively subdivide the time interval for bounds computation. In general, we can assume that if a meshlet contains vertices that are influenced by bones other than the meshlet's principal bone, subdividing the time intervals between t_1 and t_2 leads to smaller vertex bounds and consequently to tighter meshlet bounds. The maxim of our algorithm is to compute *conservative* bounds, which is why we cannot disregard the temporal influence of an animation on the bounds. The subdivision approach allows to close in on the theoretical minimum bounds, while remaining conservative. This way, the generation of vertex bounds naturally adapts to the shape of the position function of seemingly arbitrarily moving vertices, such as the one illustrated in Figure 4.2b, placing more emphasis on ranges with a strong variation.

For the normals distribution (as described in Section 4.5), a similar strategy can be employed: The mean normal direction of an initial "bind pose" or "T-pose" serves as



(a) *An initial bounding sphere (with center* $\boldsymbol{c}$ *and radius* r*) is computed s.t. it encompasses all vertices that are assigned to meshlet* m *and represents the bounds for one specific initial pose only.*

(b) *Transforming* $\boldsymbol{c}$ *and* r *with* m*'s principal bone (Bone 4) at a certain animation state reveals that the transformed initial bounding sphere not necessarily encompasses all of* m*'s vertices any longer.*

(c) *By using* m*'s conservative spatio-temporal bounds* B_m*—computed as described in Figure 4.4—we compute a factor by how much* r *must be extended to also encompass the entirety of* B_m*.*

(d) *By computing* m*'s extended radii for all animation intervals and storing the maximum radius together with the original* $\boldsymbol{c}$*, a conservative spatiotemporal bounding sphere can be computed for* m*.*

Figure 4.9: *In order to minimize computational overhead during rendering, we propose to use a meshlet* m*'s spatio-temporal AABB* B_m *for computing a scaling factor which tells how much* B_m*'s extents w.r.t. an initially created reference bounding sphere has grown or shrunk. The maximum scaling factor across all possible poses represents the conservative bounding sphere of meshlet* m*.*

the reference normal $\boldsymbol{n}_m$ for meshlet m. In the same vein of finding a maximum radius extension factor, for the normals distribution a maximum angle-deviation α_{max} can be determined by analyzing all keyframe or sub-keyframe intervals. Smaller animation time intervals generally lead to smaller values for α_{max}. During rendering, the computation of whether or not m can be conservatively backface culled can be evaluated using $\boldsymbol{c}$, the extended radius, $\boldsymbol{n}_m$, α_{max}, and the camera's position.

A variation of the strategy described above is to not store individual values for the extended radius and α_{max} per meshlet but to define constant values for them, and determine in a precomputation step which meshlets satisfy these quality requirements. I.e., this strategy would answer the questions: "Which meshlets satisfy the requirement that their bounding sphere's radius does not have to be extended by more than a constant scaling factor s_r s.t. all of its vertices stay within the extended bounding sphere?", and regarding the normals distribution: "Which meshlets satisfy the requirement that their

α_{max} is smaller than a constant α_T?" This approach has the advantage that constant values are used for s_r and α_T and do not have to be read per meshlet, thus helping to reduce the required memory bandwidth. Low-overhead shaders can be used to render those meshlets which do not fulfill any of the two criteria. Meshlets that fulfill one criterion can be rendered with the appropriate culling code. Shaders including both, BFC and VFC code, can be used for meshlets that fulfill both requirements. The subdivision approach described above can be used to determine more meshlets as being of sufficient quality for both or either of the criteria. We employed this approach for the setup of our benchmarks presented in Section 4.8.

4.7 Implementation Details

We have implemented the algorithms presented in the previous sections using C++ and Vulkan. Computing conservative vertex bounds, combining them into conservative meshlet bounds, and computing conservative normal bounds per meshlet are implemented in a CPU-based precomputation step. Our current implementation uses the same amount of parallel threads for this step as the number of logical processors reported by the operating system.

During our precomputation step, we evaluate each meshlet m_i against predefined maximum values s_r (referring to a maximum scaling factor w.r.t. m_i's initial radius r_i as described in Figure 4.9) and α_T (referring to a maximal threshold for m_i's α_{max}, which is computed as described in Section 4.5). Based on this evaluation, we assign m_i to one of the following three categories:

- m_i is both, view-frustum cullable and backface cullable, if it satisfies both limits for all animation intervals of interest.
- m_i is view-frustum cullable but not backface cullable, if it satisfies the requirements w.r.t. s_r for all animation intervals of interest, but not the requirements w.r.t. α_T.
- m_i is neither view-frustum cullable nor backface cullable if it does not fulfill at least the requirement w.r.t. s_r.

Based on these categorizations, we issue a total number of three draw calls using different pipeline configurations for each of the three categories: Meshlets that are suitable for culling are rendered with pipelines that include culling code. The meshlets that have been deemed to not be cullable are rendered with a pipeline that does not include culling code, thus not suffering from the potential computational overhead caused by the additional culling instructions. In trying to minimize runtime overhead of our culling code we chose to go for the approach with constant values for s_r and α_T instead of storing and evaluating individual thresholds per meshlet.

Our GPU implementation is based on Vulkan [Kub18b] and GLSL [KB19]. Both types of culling are performed in task shaders. Task shaders operate in groups of 32 threads per warp [NVI18a], where each of these threads tests a different meshlet in parallel. We use

(a) *Overview of our benchmark scene in view direction* **(b)** *Camera setup/camera position during benchmarks* **(c)** *Effect of using VFC* **(d)** *Effect of using BFC*

Figure 4.10: *In the test scene that we are using for our benchmarks, we position multiple instances of our test models, three of which are duplicated along the frustum planes of our camera, which is positioned as shown in 4.10b. We benchmark different geometry loads according to the model duplication scheme indicated in 4.10a. Benchmarks with view frustum culling enabled produce effects as shown in 4.10c along the frustum planes. 4.10d shows the effects of enabled backface culling on a per-meshlet basis (same camera positioning as in 4.10b).*

ballot shader instructions to synchronize the threads before passing on the information of how many and which meshlets have not been culled and are to be further processed by later shader stages. In the subsequent mesh shader stage, vertex skinning is performed for the meshlets that have survived culling. The vertices assigned to such meshlets are transformed with 32 parallel threads per meshlet. The mesh shader constitutes the final geometry processing stage, which means that its output is forwarded to the fixed-function rasterizer stage in graphics pipelines for further processing.

4.8 Results

We have evaluated the performance characteristics of our implementation (as described in Section 4.7) with different scene compositions consisting of multiple instances of the models shown in Figure 4.1 and arranged according to the scheme presented in Figure 4.10a. For all performance benchmarks, we measure the milliseconds of the relevant draw calls with GPU timer queries for 1000 frames after a warmup phase of 100 frames. The query results of the 1000 measured frames are averaged for the results. The camera remains stationary during our benchmarks at the position that Figure 4.10b has been captured from, while the models constantly animate, thus constantly varying positions and orientations of meshlets.

Figure 4.11 presents the general picture of the performance characteristics from our benchmarks, comparing culling-enabled pipelines to pipelines without culling code. Including culling code in task shaders constitutes a certain computational overhead compared to pipelines that do not include such code. The additional overhead can be more than made up for across all of our test cases and across different GPUs. Table 4.2 lists the average performance increases for the different benchmarks and shows the percentages of meshlets that could be culled. Assuming that the maximum reduction in render

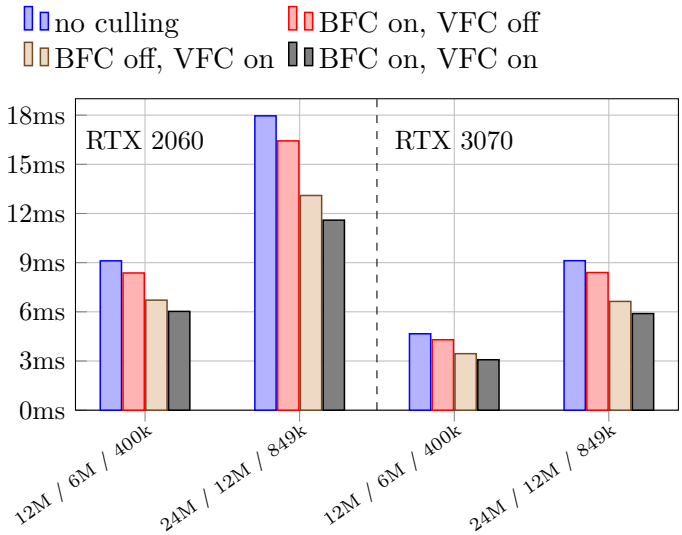


Figure 4.11: *GPU performance measurements (average milliseconds from GPU timer queries) are shown for two different GPUs and four different scene configurations. The x axis labels represent triples of number of vertices, number of triangles, and number of meshlets in that order. Each set of bars compares the results of a pipeline without culling code to the results of shader pipelines that include code for BFC only, VFC only, or both in their respective task shaders.*

time is bounded by the percentage of culled meshlets, we can state that the pipelines implementing our technique stay within a margin of only a few percent to the theoretical optimum in our tests. With BFC and VFC enabled, we measured a culling ratio of approximately 40% for the scene described in Figure 4.10, achieving reductions of render times of up to 35.4%. Consistent performance patterns can be observed across both tested GPUs. To generate the data for the results presented in Figure 4.11, we employed a precomputation step which we configured with a run-time limit of 15 minutes per model. It gradually refines the bounds within the given time limit, starting with keyframe boundaries and subdividing them until the time limit has been reached. The resulting meshlet classification details per model are shown in Table 4.3. The data that we used for our benchmarks are stated under the columns to $\alpha_T = 20°$, listing the amounts of meshlets which were rendered with the "BFC on, VFC on" pipeline ("both"), with the "BFC off, VFC on" pipeline (s_r), and with a "no culling" pipeline ("none").

Performance analyses of backface culling only are presented in Table 4.4. The performance increase rises with the number of meshlets that are classified to be backface-cullable. If only as little as approximately 20% of meshlets are backface cullable, the additional overhead of the included culling code counteracts its potential benefits. Backface culling in task shaders has shown to have the potential of an additional reduction of render times by 11.4% in our test scenes. In Section 4.6 we have described that our precomputation step can be used for gradual refinement of meshlet classification. By evaluating smaller

Table 4.2: *This table shows the average percentage of culled meshlets (columns headed "Culled") during the benchmarked scene configurations from Figure 4.11 and the performance increase that resulted from culling them, which means the reduction of render time in percent (columns headed "Faster"). "400k" refers to the 12M / 6M / 400k config, and "849k" refers to the 24M / 12M / 849k config (numbers of vertices, triangles, and meshlets, respectively).*

GPU	Scene	BFC only		VFC only		BFC+VFC	
		Culled	Faster	Culled	Faster	Culled	Faster
RTX 2060	400k	11.4%	8.1%	31.3%	26.3%	39.9%	33.8%
	849k	11.5%	8.5%	31.4%	27.1%	39.7%	35.4%
RTX 3070	400k	11.4%	7.8%	31.3%	26.1%	39.9%	34.0%
	849k	11.5%	7.9%	31.4%	27.2%	39.7%	35.4%

Table 4.3: *Classification percentages from different models using a 15 minute time limit for each. The percentage values in columns labeled with "both" refer to meshlets that fulfill the requirements to be both, view-frustum cullable and backface cullable—i.e., stay below a radius scale factor s_r and within a normal deviation threshold of α_T. The values in "s_r" columns represent the number of meshlets that only fulfill the requirement of staying below the radius scale factor. The number of meshlets that do not fulfill the requirements are listed in column "none". Using higher values for the normal deviation threshold α_T results in more meshlets satisfying "both" requirements at the cost of less optimal backface culling performance during rendering. $s_r = 3$ was used for generating this classification.*

	$\alpha_T = 10°$		$\alpha_T = 20°$		$\alpha_T = 30°$		
Model	both	s_r	both	s_r	both	s_r	none
Gawain	82%	17%	89%	10%	92%	7%	1%
Giant Worm	66%	29%	71%	24%	74%	21%	5%
Butcher	88%	6%	90%	4%	91%	3%	6%
Wyvern	52%	42%	60%	34%	65%	29%	6%

animation subintervals, steadily tighter conservative bounds can be found for meshlets, eventually leading to a higher number of meshlets being cullable during run time. As our results in Tables 4.2 and 4.4 attest, performance increases proportionally to the number of meshlets that could be culled in task shaders. Therefore, increasing the number of meshlets that *can* be culled benefits render times accordingly. The trade-offs between precomputation time and resulting meshlet classification percentages are shown in Figure 4.12. It can be seen that different outcomes must be expected from different animated models and their animation clips. While the initial classification values of the Butcher model show high percentages of cullable meshlets already after relatively short precomputation times (i.e., small to no subdivisions of keyframe intervals), different

Table 4.4: *Average render times of scenes composed of different ratios of meshlets which are backface-cullable to meshlets that do not satisfy that requirement. For these measurements, view frustum culling was disabled. Only backface culling code is active in task shaders, producing the results listed under the "BFC on" column, while the pipeline used to create the results found under the "BFC off" column does not include any culling code. The average percentage of meshlets that were culled by BFC code is listed in the column "Culled" and the resulting reduction of render time in percent is listed under "Faster". A scene setup with 731k meshlets was used.*

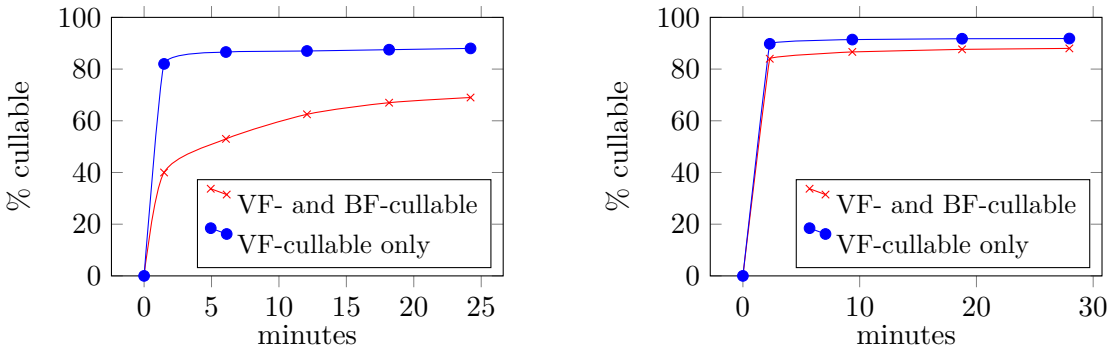
Cullable	BFC off	BFC on	Culled	Faster
100%	7.79ms	**6.90ms**	15.7%	11.4%
80%	7.80ms	**7.12ms**	12.5%	8.7%
60%	7.80ms	**7.35ms**	9.2%	5.7%
40%	7.79ms	**7.51ms**	7.0%	3.6%
20%	7.79ms	**7.77ms**	3.8%	0.3%
0%	**7.80ms**	8.07ms	0.0%	-3.4%

characteristics can be observed for the Giant Worm model. Increased computational effort in the precomputation step leads to significantly higher numbers of backface cullable meshlets for Giant Worm. Tighter target bounds for s_r and α_T have been chosen to emphasize the effects of gradual refinement during precomputation.

We further evaluate the effects of added culling for animated meshlets in a scenario that also includes static models. Figure 4.13 shows a scene configuration where in addition to 400k animated meshlets, 399k static-geometry meshlets are rendered. In these tests, static meshes are also drawn via task and mesh shaders and always culled with established VFC and BFC methods. The resulting performance measurements of the *combined* render times are shown in Figure 4.13b. The additional static meshlets raise the total render time on an RTX 3070 from 4.66ms to 7.23ms if animated meshlets are not culled. The same scene configuration with VFC and BFC enabled for animated meshlets reduces the total render time to 5.69ms, which constitutes an average reduction of combined render time by 21.3%. Culling percentages remain the same as stated in Table 4.2 for 400k animated meshlets. While we kept vertex processing effort at the minimum for both types of geometry, animated models still require significantly more vertex processing than static models due to skinning code in mesh shaders, highlighting the benefit of our approach for scenes with moderate to high amounts of animation.

4.9 Discussion and Future Work

We have presented an algorithm to compute conservative spatiotemporal bounds on a per-meshlet basis. Using the spatiotemporal vertex bounds of its assigned vertices, also a conservative estimate for a meshlet's normals distribution can be computed. Bounds and normals distributions are intended to be computed during a flexible precomputa-



(a) *Effect of gradual bounds refinement for* Giant Worm *clips.*

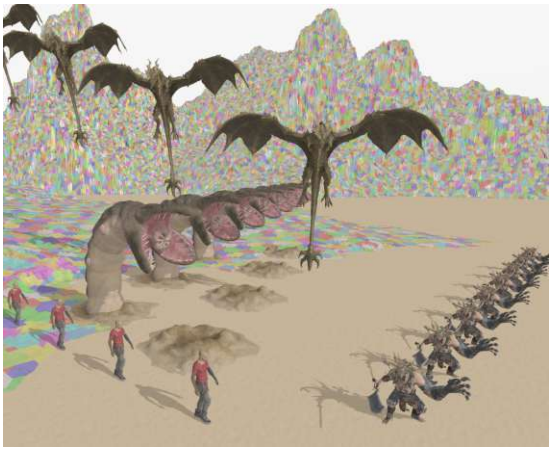
(b) *Effect of gradual bounds refinement for* Butcher *clips.*

Figure 4.12: *Effect of different time targets for our precomputation step on the classification into meshlets that are both view-frustum cullable and backface cullable, view-frustum cullable only, or neither.* $s_r = 2$ *and* $\alpha_T = 10°$ *were chosen for these measurements.*

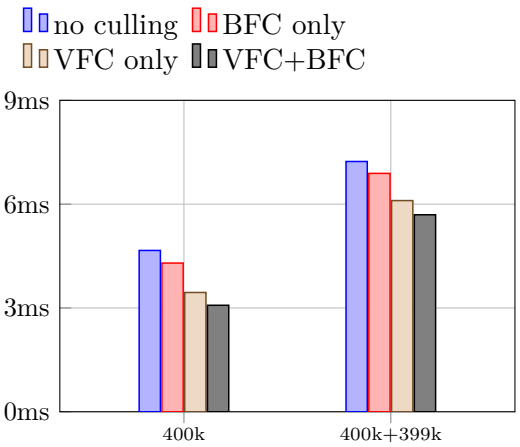
tion step which allows to trade tighter bounds or normal deviation angles for reduced precomputation time. In all cases, our algorithm ensures conservative results.

Adding culling to task shaders incurs some additional computational overhead of a few percent during rendering. This disadvantage can in general be more than made up for in our tests. VFC on a per-meshlet basis enables fine-grained culling of meshlets outside of the view frustum and can lead to significant reductions of render time. The benefit of including BFC in task shaders depends on the quality of meshlets insofar as many of them should be backface cullable. If the precomputation step manages to compute conservative normals distributions for close to 100% of meshlets, render time reductions of up to 11.4% are possible through BFC in graphics pipelines with very light vertex processing load and can be expected to be significantly higher with graphics pipelines that feature complex vertex processing load. The benefit of combined BFC and VFC was close to the theoretical optimum in our tests when comparing the relative reduction of render time to the percentage of culled meshlets.

Naturally, model animation is a far-reaching and complex application field. In this work, we have derived and presented a solution suitable for bounding individual animation clips. However, we note that our basic approach may easily be extended for use with a variety of techniques. For example, inverse kinematics (IK) is a common method in modern real-time animation. For pipelines that involve IK, we can reuse the same techniques presented in this chapter, but instead of subdividing and bounding vertex motion across time intervals, we can instead bound a different parameter space, such as the solid angles representing ranges of possible orientations for a set of joints. Other important techniques, such as the blending of animation clips, can be addressed by not computing bounds for individual clips, but instead for the full repertoire of possible animations. If intermediate vertex states are produced from linearly blending between animations, conservative vertex



(a) *Additional static-geometry meshlets (terrain, meshlets colored) are rendered for this benchmark.*



(b) *Render times of 400k animated meshlets compared to render times of 400k animated + 399k static meshlets.*

Figure 4.13: *Mixed scenario with static and animated meshlets. 4.13b shows average render times in milliseconds, comparing the results of the 400k measurement on RTX 3070 from Figure 4.11 with the measurements of the same setup, plus additional 399k static-geometry meshlets. Again, our technique leads to significant render time reductions.*

bounds are then easily obtained from the union of all animations, and the bounding of the normal cone can be performed as previously described.

With the addition and ongoing development of hardware-accelerated ray-tracing, the use of already-computed spatial acceleration structures for ray-tracing might be considered as a viable, hierarchical alternative in the future. In contrast to *currently* available data structures, our approach serves to compute conservative bounds over arbitrary time intervals and does not require random access to meshlet data, as it must be expected with ray tracing. Instead, meshlet data is accessed in a strictly contiguous manner, not dissimilar to vertex attribute streaming in conventional rasterization-based graphics pipelines, hence the available data structures with logarithmic access times are unfavorable in this case. In a similar vein, hierarchical data structures such as bounding sphere trees [JP04; KŽ05a; SBT06] could represent a possible avenue for increasing the performance of our precomputation step by decreasing its computational cost from $\#vertices \times \#joints$ down to $\#meshlets \times \#joints$. However, since bounding spheres are a less accurate representation than bounding boxes, and since the meshlet-focused approach would overestimate bounds even more, we decided on sticking with the more accurate approach of computing bounding boxes per vertex.

As far as future work is concerned, investigating options for accelerating the precomputation step and allowing further tradeoff options could lead to helpful improvements of our algorithm. Also adding support for further skinning methods—such as dual-quaternion skinning—besides LBS would be a natural pathway for future research, since applications

are likely to require other skinning methods as well. However, we note that the dependency on a particular skinning method is comparably small with our presented approach: the only missing piece for enabling different skinning techniques is the derivation of conservative positional bounds for a single vertex between two successive keyframes. Once derived, the corresponding methods can be supplied as a drop-in replacement for the current solution for LBS. Our approach for robust meshlet bounds can therefore work with any skinning technique for which such bounds can be found.

Overall, we have presented a fundamental algorithm for computing conservative bounds of clusters of ultra-detailed input geometry. The higher the geometric detail of a given input mesh, the more relative performance gain can be expected since generally, both spatial bounds and normals spread can be expected to shrink if more triangles are used to represent a given shape, assuming that cluster/meshlet sizes stay the same. Our evaluations have shown that through fine-grained culling, a lot of rendering time can be saved—showing almost linear performance improvement w.r.t. the percentage of culled meshlets. Thus, is can be stated that our method provides optimizations that are crucial for maintaining real-time frame rates for rendering ultra-detailed animated models.

In the next chapter, we take the fundamental idea of rendering ultra-detailed static or animated models one step further by generating geometry directly on the GPU by means of parametric functions. This reduces memory load in early graphics pipeline stages to almost zero and amplifies geometry in later pipeline stages.

CHAPTER 5

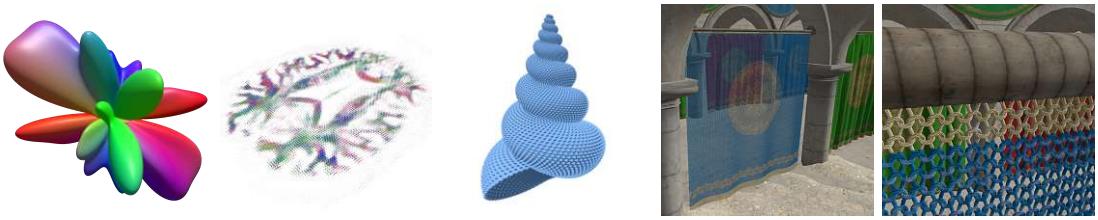
Fast Rendering of Parametric Objects in Real Time on Modern GPUs

The contents of this chapter constitute an extended edition of our paper "Fast Rendering of Parametric Objects on Modern GPUs", which was presented at the Eurographics Symposium on Parallel Graphics and Visualization 2024 [Unt+24].

5.1 Motivation

Parametric functions are an extremely efficient representation for 3D geometry, capable of compactly modeling highly complex objects, some examples of which are shown in Figure 5.1. Once specified, parametric 3D objects allow for visualization at arbitrary levels of detail, at no additional memory cost, limited only by the number of evaluated samples and by floating point accuracy. This makes them perfectly suitable for being integrated into ultra-detailed geometry scenarios, offering the same (or even higher) geometric precision than the input meshes, like the one shown in Figure 1.1.

In this chapter, we describe a technique that renders objects described by parametric functions. Our technique evaluates the required levels of detail (LOD) in every frame and culls parts of parametrically-defined objects in a fine-grained manner—not dissimilar to our technique from Chapter 4, just now on a per-patch level instead of working with meshlets. A similarity between the two techniques is that only a relatively little amount of data is loaded for making the VFC decision before handling the associated (possibly ultra-detailed) geometry: The approach in Chapter 4 loads a meshlet's metadata in the task shader and spawns mesh shaders only if the meshlet is not culled. The approach in this chapter operates on patches and uses a separate compute shader-based LOD stage,



(a) *Single SH glyph order 12* **(b)** *19,600 SH glyphs* **(c)** *Parametric seashell* **(d)** *358k fiber curves (curtain)* **(e)** *358k fiber curves close-up*

Figure 5.1: *Our technique is able to render a variety of parametrically defined objects with diverse properties—all in real time. The frames per second for Figures 5.1a to 5.1e on a mid-range previous-generation NVIDIA RTX 3070 GPU are 248 FPS, 270 FPS, 349 FPS, 98 FPS, 138 FPS, respectively (Figures 5.1a and 5.1b rendered at 1440×1440 resolution, Figures 5.1c to 5.1e at 1920×1080 with 4×SSAA and 8×MSAA).*

in the context of which VFC is performed on a per-patch level. Geometry is subsequently only generated for those patches that were not culled in that LOD stage.

Our method has the potential to reduce memory load on a GPU since geometry is generated on-chip through tessellation or is directly point-rendered—only for patches that were not culled. It achieves high rendering performance for many parametric functions. It even outperforms a root finding-based technique for spherical harmonics (SH) glyph rendering by utilizing modern GPUs' hardware features efficiently.

5.2 Introduction

Procedural content can be created from coarse inputs by computing transient data for fine-grained details on the fly. At the application level, we can exploit suitable procedural geometry representations, like parametric functions [PDG21; Cra23; Wil22]. These can efficiently yield detailed 3D shapes by evaluating surface or volume samples according to the function's mathematical definition, as illustrated in Figure 5.2. However, mapping the sample evaluation to the hardware rendering pipelines of modern graphics processing units (GPUs) is not trivial. This has given rise to several specialized solutions, each targeting interactive rendering of a constrained set of parametric functions.

In this chapter, we propose a general method for efficient rendering of parametrically defined 3D objects. Our solution is carefully designed around modern hardware architecture. Our method adaptively analyzes, allocates, and evaluates parametric function samples to produce high-quality renderings. Geometric precision can be modulated from few pixels down to sub-pixel level, enabling real-time frame rates of several 100 FPS for various parametric functions. We propose a dedicated LOD stage, which outputs patches of similar geometric detail to a subsequent rendering stage that uses either a hardware tessellation-based approach or performs point-based software rasterization. Our method requires neither preprocessing nor caching, and the proposed LOD mechanism is fast

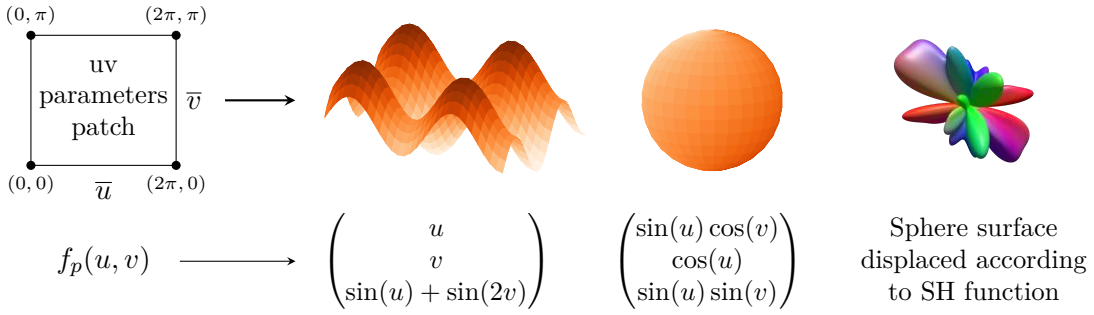


Figure 5.2: *We consider vector-valued parametric functions $f_p(u, v)$ with $f_p : \mathbb{R}^2 \to \mathbb{R}^3$ mapping independent 2D variables called parameters to 3D positions in Euclidean space. The arithmetic definition enables a compact representation of geometric shapes with varying complexity and desirable properties, such as C^∞ continuity. We illustrate the input to such $f_p(u, v)$ with parameter patches that range from lower bound parameter values u_{min}, v_{min} to upper bound parameter values u_{max}, v_{max}, where the notations $\overline{u}$ and $\overline{v}$ refer to the whole ranges, i.e., $\overline{u} = [u_{min}, u_{max}]$ and $\overline{v} = [v_{min}, v_{max}]$.*

enough to run each frame. Hence, our approach also lends itself to animated parametric objects. We demonstrate the benefits of our method over a state-of-the-art spherical harmonics (SH) glyph rendering method, while showing its flexibility on a range of other demanding shapes.

At the hardware level, procedural content generation is facilitated by modules like the tessellation shader [Khr23tesb]. Coarse base geometry is processed in compute units, where the tessellation engine produces new geometry primitives before passing them to the rasterizer. Hence, fundamentally tessellation trades increased computational load for reduced memory transfers, resulting in overall increased rendering speed compared to not using the tessellator and transferring geometry in high detail [Nie+16]. However, their historic orientation toward triangle meshes makes it unclear how to exploit these modules for other representations, including parametric functions.

With our proposed method, we consolidate the use of procedural geometry representations—specifically, parametric functions—with recent hardware trends to achieve fast, high-quality rendering of complex mathematical shapes. Our proposed technique takes as input only a parametric function. Compared with previous work, our solution makes few assumptions about the sampled functions, supporting a wide range of complex shapes (see Figure 5.1). It requires neither derivatives nor preprocessing. We propose a multi-step pipeline: a dedicated LOD analysis stage adaptively determines the sampling density to be used by a subsequent rendering stage. Depending on the chosen sampling resolution, our technique can render highly tessellated patches (where each resulting triangle spawns approximately one or only a few screen pixels) or directly render the parametric function point-wise, typically sampling the function once or multiple times per screen pixel. Our point-based method draws inspiration from recent advances in point cloud

rendering [SKW21]. Common to both rendering approaches is their emphasis on utilizing the compute capabilities of GPUs and their ease of integration into existing rendering applications. We also account for the emerging trend in recent years toward ultra-high geometric detail in real time, as heralded by Epic Games' Nanite [Epi24a]. Nanite can render static geometry at such high detail that, after a LOD selection step, rasterization is usually performed on triangles not much larger than a single pixel[KSW21]. Similarly, our technique can render almost pixel-perfect geometric detail for parametric functions that show limited variance at a sub-pixel level when rendering at screen resolution. With super-sampled (SS) configurations, our tessellation-based or point-based rendering variants are able to capture and render sub-pixel geometric detail in real time.

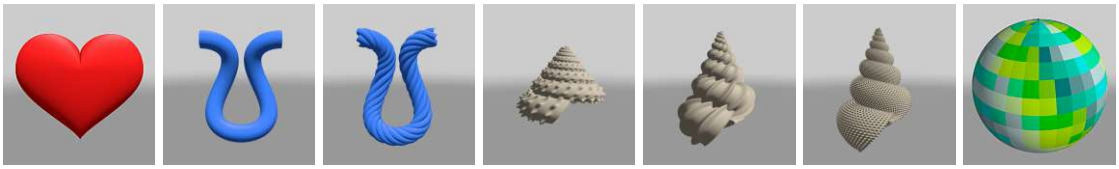
In summary, our contributions include the following:

1. We describe a general method to render a wide range of parametric functions with close to pixel-perfect geometric accuracy, which is fast enough to be used in conjunction with SS.
2. We describe an efficient compute shader-based LOD selection algorithm that generates view-dependent parameter patches for generic parametric functions, leading to approximately uniform geometric detail in the rendered output across the entire parametric object.
3. We describe different variants to render the patches from Contribution 2, including point-based rendering, rasterization using the hardware tessellator, and a hybrid technique to select the optimal rendering variant per patch.
4. We evaluate our method on a range of demanding parametric shapes and compare it to the state of the art in terms of SH glyph rendering by Peters et al. [Pet+23], showing that our method surpasses it in terms of rendering speed and rendering quality for higher SH orders; in large datasets already for SH order 4.

5.3 Related Work

Most previous work on rendering parametric curves or surfaces focuses on specific parametric shapes, such as the efficient rendering of rational Bèzier patches [EML09; SS09], Catmull-Clark subdivision surfaces [PEO09; NL13; Kut+23; WA23], or similar patches such as B-Spline or NURBS curves and surfaces [WA23]. Poirier et al. [PDG21] focus on rendering Spherical Harmonics (SH) glyphs for visualizing measurements from diffusion magnetic resonance imaging (dMRI) scans. Some of these techniques utilize specialized data structures [PEO09; Kut+23]. All of them rely on a triangulated representation of a given type of parametric function for rendering. Some techniques create triangle primitives in software to be rendered by the hardware rasterizer or perform tessellation in software [PEO09; SS09; EML09; WA23], while others make use of hardware tessellation units [NL13], which are standard features on modern desktop GPUs. The technique by Kuth et al. [Kut+23] uses mesh shaders, which were first introduced to desktop GPUs with NVIDIA's Turing architecture [NVI18a] in 2018.

Especially older techniques do not try to produce ultra-detailed geometry but instead



(a) *Johi's Heart* **(b)** *Yarn Curve* **(c)** *Fiber Curve* **(d)** *Seashell variant 1* **(e)** *Seashell variant 2* **(f)** *Seashell variant 3* **(g)** *Sphere*

Figure 5.3: *These images show various parametrically defined objects. Slight variations in the parametric functions can lead to different shapes, as can be seen in Figures 5.3d to 5.3f, which all use the same underlying parametric function with slight variations in auxiliary parameter values. Figure 5.3g shows a possible set of patches produced by our* Patch Subdivision *stage for a parametrically defined sphere.*

optimize for a visual error metric to be less than a pixel [EML09; PEO09], which is also the approach taken by the more recent technique by Worchel et al. [WA23]. The method of Eisenacher et al. [EML09] is conceptually similar to our approach, insofar as it subdivides patches until a certain metric is satisfied. In contrast to our approach, they use uniform subdivision of patches, and their technique is tailored to Bèzier patches. Their method could be incorporated into the structure of our method by using the same error metric from their "oracle" step in our Patch Subdivision stage instead of our generally applicable screen distance-based metric. Our tessellation-based rendering variant would be perfectly suitable for rendering Bèzier patches, possibly requiring a crack avoidance procedure between neighboring patches.

The work by Poirier et al. [PDG21] on SH glyph rendering takes a more pragmatic approach in trying to evaluate and set suitable tessellation levels, but fails to prevent visual inaccuracies. Another recent approach is able to produce and render ultra-detailed geometry at real-time frame rates on modern GPUs [Kut+23], but focuses on Catmull-Clark subdivision surfaces only. In terms of SH glyph rendering, Peters et al. [Pet+23] achieve pixel-perfect geometric detail by intersecting a ray with the SH glyph through polynomial root finding for each screen pixel. Their visual results constitute a significant improvement over the results from Poirier et al. [PDG21], but suffer from exceedingly decreasing performance with increasing SH order and visual artifacts with SHs of orders 10 and higher. In terms of rendering configuration, our proposed method is more similar to the approach by Poirier et al. [PDG21] insofar as we also use the graphics pipeline. However, in terms of rendering quality and in terms of its performance characteristics, our method is much more similar to the ray tracing-based approach by Peters et al. [Pet+23]: both methods scale performance-wise relative to the number of rendered pixels: The method of Peters et al. traces one ray per pixel, while our solution selects suitable levels of geometric detail using a screen distance-based metric.

Further usages of glyphs in the context of medical or scientific visualization include comparisons between healthy and infected persons [Zha+17b; Zha+17a; Zha+15] or comparisons between ensembles of stress tensor fields [Abb+15].

Point rendering can arguably also be described as an approach to render ultra-detailed geometry, under the condition that sufficiently many samples are used. Recent work shows that modern GPUs are capable of processing and rendering 50 to 144 billion points per second [SKW21; SKW22], with the former being limited by memory bandwidth and the latter improving the throughput through compression. By sampling points on parametric functions, we avoid memory fetches as the bottleneck but potentially trade it for a compute-based bottleneck.

Similar to Nanite [KSW21], our approach combines the strengths of the graphics and compute pipelines of modern GPUs to produce geometry with close-to-pixel detail. However, we instead target parametric functions, which can be sampled with arbitrary fidelity, limited only by machine floating point precision. Furthermore, our solution involves no preprocessing and recomputes the levels of detail from scratch in each frame; thus, it also lends itself to animated shapes with erratic changes in appearance.

5.4 Parametric Function Definition

Our method considers parametric functions that transform two input parameters (u, v) into Cartesian coordinates (x, y, z) of the three-dimensional Euclidean space, i.e., $\mathbb{R}^2 \to \mathbb{R}^3$, or intuitively a transformation of a quad/rectangle into a three-dimensional surface as illustrated in Figure 5.2. Interpreting $\mathbb{R}^2$ as, for example, spherical coordinates and $\mathbb{R}^3$ as the cartesian coordinates of the corresponding points on a sphere allows us transforming the quad into a sphere. We can then further transform the sphere into spherical harmonic glyphs using the respective SH functions.

Listing 5.1: *GLSL source code of a parametric function which produces a heart shape based on two input parameters u and v.*

```
#define PI        3.14159265359
// Creates the surface "Johi's Heart".
// Input: u ... first  parameter in range [0, PI  )
//        v ... second parameter in range [0, 2*PI)
vec3 sampleJohisHeart(float u, float v) {
    // Start with a sphere shape:
    vec3 p = vec3(sin(u) * cos(u), cos(u), sin(u) * cos(v));
    // Distort it into a heart shape:
    if (u < PI / 2.0) {
        p.y *= 1.0 - (cos(sqrt(sqrt(abs(p.x * PI * 0.7)))) * 0.8);
    } else {
        p.x *= sin(u) * sin(u);
    }
    p *= vec3(0.9, 1.0, 0.4);
    return p;
}
```

In the context of our method, a *parametric function* is expressed directly in source code; in addition to mathematical elements, it may also contain logical operators and flow

control (conditional statements, loops, and recursions). This facilitates the intuitive generation of features that are harder to express mathematically, such as creases or discontinuities. An example of such a parametric function is given in Listing 5.1. It uses `if` statements to scale parts of a sphere base shape such that the resulting surface forms the heart shape shown in Figure 5.3a.

5.5 Method

In this section, we describe our approach, which is able to render a wide variety of parametric functions in real time with controllable precision. Depending on the current frame's camera position, orientation, projection, and screen coordinates, a LOD stage produces patches of similar size in screen space, which are subsequently rendered with one of two different rendering variants. Figure 5.4 depicts the overview of our method and its major stages:

1. Patch Initialization: A compute shader stores one patch per parametric object in the buffer of patches to be evaluated, in particular the parameter ranges $\overline{u_i}$ and $\overline{v_i}$ along with some auxiliary data, such as the type of object—which refers to a particular $f_{p_i}(u, v)$—and which material shall be used for shading. For objects that are known to be very detailed, $\overline{u_i}, \overline{v_i}$ can already be uniformly subdivided in this stage, which ensures a minimal number of output patches to be forwarded to the rendering stage.
2. Patch Subdivision: The second stage executes up to a predefined number of n LOD steps, which subdivide the parameter ranges $\overline{u_i}, \overline{v_i}$ until their approximated screen-space extents e_{u_i}, e_{v_i} when evaluated with $f_{p_i}(u, v)$ no longer exceed screen-space thresholds t_u, t_v. The point of this procedure is to create patches of similar sizes to be forwarded to the Rendering stage in order to optimally utilize GPUs.
3. Rendering: All the patches which have been scheduled for rendering in one of the preceding LOD steps during Patch Subdivision are rendered in one of two manners:
   a) Tessellation-based: Patches are rendered with a tessellation-enabled graphics pipeline with either fixed or adaptive tessellation levels. Vertices generated by the tessellator are positioned according to $f_{p_i}(u, v)$.
   b) Point-based: Patches are sampled by $f_{p_i}(u, v)$ at fixed steps across parameter ranges $\overline{u_i}$ and $\overline{v_i}$. Results are projected into screen space and written to a 64-bit integer target image.

5.5.1 Patch Subdivision Stage

To determine whether and how a patch i should be subdivided, its parameter range $\overline{u_i}, \overline{v_i}$ is sampled and evaluated using its associated function $f_{p_i}(u, v)$. The evaluated samples are projected to screen space and analyzed separately to determine the subdivision pattern. In order to achieve adaptive subdivision, we sample along lines in u- and v-direction

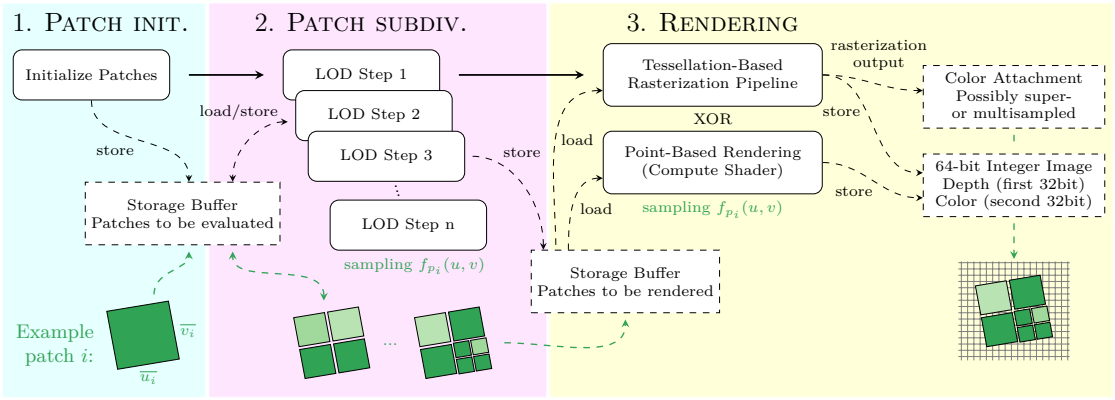


Figure 5.4: *Overview of the stages of our method, and the buffers and textures which are accessed in each stage. Patches are evaluated and subdivided and then re-evaluated up to n times in the n LOD steps before they are stored in the "Patches to be rendered" buffer. The rendering stage reads all the patches from that buffer and either generates a triangle mesh to render them via hardware tessellation, or performs point-based rendering. The latter variant cannot be used with a renderpass that rasterizes into a framebuffer. Instead, it must perform* `atomicMin` *writes into a 64-bit integer buffer or image.*

independently. Concretely, we take 8 samples in u-direction at 4 fixed v-values $v_1, \ldots, v_4$ (for a total of 32 samples), and the same for the v-direction at 4 fixed u-values $u_1, \ldots, u_4$. For each line, we compute the sum $e_{u_{ik}}$ or $e_{v_{ik}}$ respectively of screen-space extents between the projected sample points, and compare them to user-defined thresholds t_u, t_v. If for any line, $e > t$, then the patch half in which the line is located is split along the direction of the line. For example, if this happens to at least one of the lines in u-direction at v_1 or v_2, then the first half of the v-range needs to be split into two u-intervals and its parts are scheduled for re-evaluation by the subsequent LOD step. The resulting subdivision patterns are shown in Figure 5.5.

The 32 samples correspond to a typical subgroup size on NVIDIA and Intel GPUs. After a subgroup has taken 2×32 samples for the patch splitting decisions, the same subgroup takes a further 25 samples of the parametric function, which is crucial to prevent false-positive culling decisions for cases where only a small part of a patch corner reaches into the viewing frustum. Our evaluation scheme is illustrated in Figure 5.6, showing the lines used for patch evaluation along u and v, and the 25 extra samples.

A fixed number of compute shader invocations is dispatched every frame to perform these evaluations in multiple subsequent LOD steps. We use 12 dispatch calls, which is enough to subdivide a (perfectly screen-aligned) 32k pixels-wide patch down to 8 pixels and should suffice for the vast majority of cases. After sufficient patch subdivision has been achieved in a certain LOD step, no further LOD step will store any more patches into the "Patches to be evaluated" storage buffer and only store sufficiently subdivided patches into the "Patches to be rendered" storage buffer—how both buffers are accessed

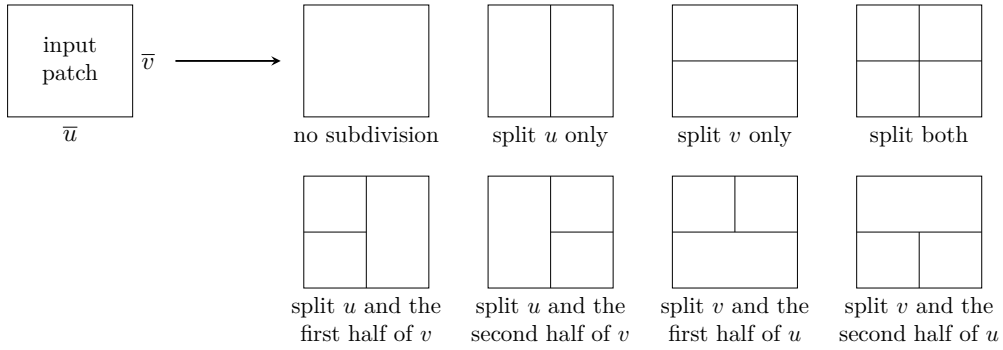


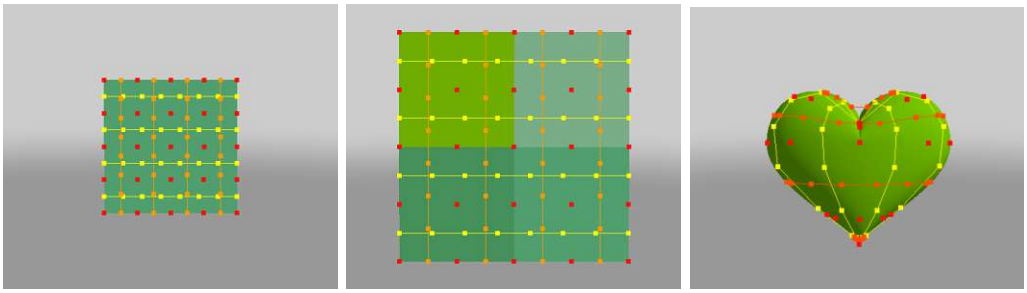
Figure 5.5: *During an LOD step, an input patch can either not be subdivided, or split according to the seven patterns shown in this figure when being scheduled for re-evaluation.*

is shown in Figure 5.4. All of the dispatch calls in the Patch Subdivision stage are indirect dispatch calls. For those LOD steps for which no patches are left to be evaluated this means that the GPU still has to process the respective dispatch commands, but will find that the number of workgroups to be processed is zero and therefore, no compute invocations will be executed [Khr24vsp]. The empty dispatch calls did not incur any noticeable or measurable overhead in our tests. Alternatively, the number of dispatch calls could be adapted based on the previous frame's highest required LOD step, if latencies of a few frames to reach the appropriate number of dispatch calls are acceptable by an application.

In many scenarios, the Patch Subdivision stage is very fast, often taking less than 10% of the total frame time, which is typically the case for the tests presented in Figures 5.12 and 5.16 even when the camera is near the parametric object so that many screen pixels are covered. The exact percentage depends on the particular object and scene setup. In the test presented in Figure 5.15, the LOD stage has to determine patch sizes for 358k fiber curves (i.e., 358k initial patches), which leads to the LOD stage taking up to 50% of the frame time for this particular rendering configuration.

One key factor of the Patch Subdivision stage's low impact on frame times in our implementation is its heavy use of subgroup operations. They allow sharing data between compute invocations [Khr24sub]. For example, in order to compute the screen distance between two adjacent samples, the parametric function does not have to be sampled twice in the same thread—instead, the other sample's screen-space coordinates are retrieved from the thread which computed the neighboring sample by accessing this neighboring thread's invocation index through subgroup operations.

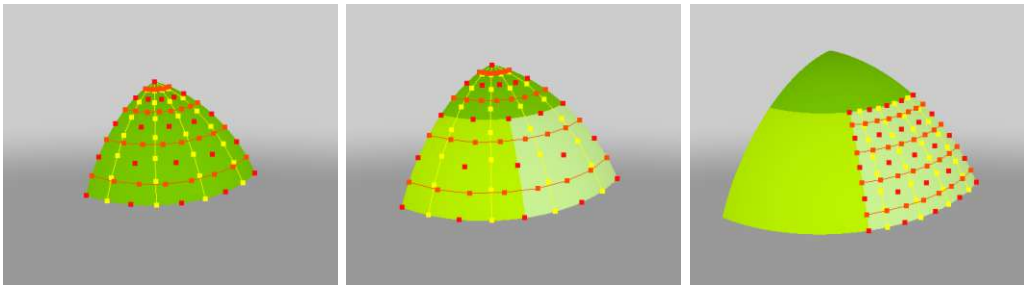
The second key factor for achieving fast rendering performance is non-uniform patch subdivision, which is illustrated in Figure 5.5. There are eight possible cases when a patch is evaluated in the LOD step: If the approximated screen-space extents $e_{u_{ik}}, e_{v_{ik}}$ do not exceed thresholds t_u, t_v across $\overline{u_i}, \overline{v_i}$ of patch i, no subdivision is performed. In this case, patch i is not scheduled for re-evaluation but is instead stored in the buffer for



(a) *Patch evaluation found that this patch does not need to be split.*

(b) *Patch evaluation found that this patch shall be split across $\overline{u}$ and $\overline{v}$.*

(c) *Patch evaluation pattern on the example of Johi's Heart.*



(d) *Evaluation of one patch of a sphere segment (LOD step 1)*

(e) *Non-uniform subdivision of a sphere segment (LOD step 1)*

(f) *Evaluation of one of the now subdivided patches (LOD step 2)*

Figure 5.6: *Each of our LOD steps analyzes a given $\overline{u_i}, \overline{v_i}$ patch by sampling $f_{p_i}(u, v)$ at 89 locations to determine if and which splits are necessary. Samples to approximate the screen-space extents across $\overline{u_i}$ are drawn in yellow. Samples to approximate the screen-space extents across $\overline{v_i}$ are drawn in orange. The 25 extra samples to help with the frustum culling decision are indicated by red dots. Figure 5.6c shows why multiple evaluations across a patch are crucial in many cases: The first and fourth measurements in v-direction ($e_{v_{i_1}}$ and $e_{v_{i_4}}$) are very near to the poles, where the parameters are very condensed. The middle two measurements ($e_{v_{i_2}}$, $e_{v_{i_3}}$) fall into usable positions. Figures 5.6d to 5.6f show an example of evaluating a sphere segment. Assuming a resolution of 512×512 and user-defined screen thresholds of $t_u = t_v = 256$, LOD step 1 finds that the patch in Figure 5.6d does not exceed the thresholds and hence, no subdivisions are required. With the camera closer to the object in Figure 5.6e, the lower parts of the sphere segment exceed t_v. The patch is non-uniformly subdivided and the new parts are scheduled for evaluation in LOD step 2. Figure 5.6f shows the evaluation pattern on one of these patches during LOD step 2.*

patches to be rendered. If, however, an exceedance of t_u, t_v was detected for any $e_{u_{ik}}$ or $e_{v_{ik}}$, patch i is split according to the seven patterns shown in Figure 5.5—except in LOD step n, where (possibly split) patches are scheduled for rendering in any case. Non-uniform patch subdivision doesn't influence the performance of the Patch Subdivision stage too much, but to a greater degree leads to better performance in the Rendering

stage, since the resulting patches are more similar in terms of their screen-space extents. The reduction in total frame time when comparing non-uniform patch subdivision to uniform patch subdivision—that is, a constant patch subdivision scheme of one patch into four—amounts to already 15% for a simple parametrically defined sphere like shown in Figure 5.3g, and can be as high as 50% for the close-up view of a SH glyph like shown in Figure 5.12c.

5.5.2 Rendering Tessellated Patches

Rendering patches that have been scheduled for rendering with our TESSELLATION-BASED variant is very straightforward: Each scheduled patch i is rendered as a quad with fixed inner and outer tessellation levels l [Khr23tesb]. Each vertex produced by the tessellator is set to the position produced by $f_{p_i}(u_x, v_y)$, where u_x and v_y refer to the interpolated parameter locations within $\overline{u_i}$ and $\overline{v_i}$ ranges, produced by the tessellator. In GLSL, these can be computed with the help of `gl_TessCoord.xy` in tessellation evaluation shaders [Khr23tesa]. The maximum tessellation level supported on modern GPUs is typically 64, which means that an edge (of a triangle or quad to be tessellated) is subdivided into 64 parts. A perfectly screen-aligned edge with a screen-space extent of 64 (which could be the result of using a threshold value $t = 64$) would therefore be subdivided so that there is one segment for each pixel. In many cases, choosing such fine subdivisions is counter-productive due to the resulting impact on rendering performance [KDR18; Ker+22]. Therefore, we propose to use fixed tessellation levels only for parametric functions that are expected to produce sub-pixel geometric detail, or resort to adaptive tessellation with SS, which might capture sub-pixel detail even better and produce more uniform geometry density, enabling continuous LOD selection.

Adaptive tessellation levels are based on the actual approximated maximum screen-space extents e_{u_i}, e_{v_i} of a given patch i. These extents are measured during PATCH SUBDIVISION by taking the maximum of the four measurements $e_{u_{i_k}}$ and the maximum of the four measurements $e_{v_{i_k}}$, as illustrated in Figure 5.6. For example, splitting parameter range $\overline{u_1}$ with approximated screen extents e_{u_1} based on threshold t_u can lead to split patch sizes with extents $e_{u_2} \approx \frac{t_u}{2}$ if e_{u_1} is just slightly less than t_u. Adaptive tessellation in that context means to scale a tessellation level l as follows:

$$l = \text{clamp}(\frac{e_{u_2} l}{\min(t_u, l)}, l_{min}, l_{max}).$$

The result is clamped to a minimum tessellation level l_{min} (e.g., $l_{min} = 8$) and a maximum tessellation level l_{max} (e.g., $l_{max} = 64$).

5.5.3 Point-Based Rendering

With the POINT-BASED variant, a patch i of range $\overline{u_i}, \overline{v_i}$ to be rendered is sampled point-wise by $f_{p_i}(u, v)$ in equidistant steps along each parameter direction. The resulting color values are stored in an image at the respective screen-space coordinates. Since

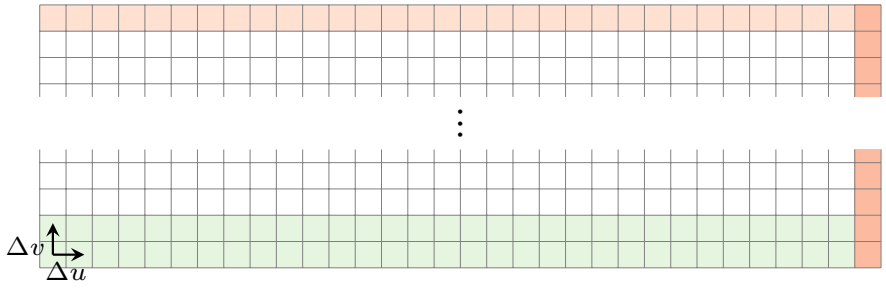


Figure 5.7: *With the Point-based variant, parameter range $\overline{u_i}$ is processed in "columns" of 32 samples. Within such a column, subgroups sample the patch to be rendered "row-wise" in v parameter direction. A subgroup always keeps the data of two rows in registers, which are marked in green. Data of neighboring u samples is shared through subgroup operations. The samples that do not write pixels are marked in red—they take auxiliary samples that are used for tangent and bitangent calculations.*

color attachments are incompatible with this rendering variant, we use a 64-bit image as the target to receive the rendering output as described in Section 5.5.2. Samples of an input patch are produced in the following manner: The screen-space threshold parameters t_u, t_v determine the size of a patch. Given a defined workgroup size of N, we divide the parameter range $\overline{u_i}$ by the smallest multiple of N that is larger than t_u, i.e. by $\lceil \frac{t_u}{N} \rceil$, and use that as the total number of samples taken across $\overline{u_i}$ for each parameter v. Parameter range $\overline{v_i}$ is sampled in steps of size $\Delta v = \frac{v_{i_{max}} - v_{i_{min}}}{t_v} - \varepsilon$, where ε can be used to decrease the step size in order to prevent holes in the rendered output. We use $N = 31$, which is not arbitrary, but rather tied to our compute shader-based implementation: 32 subgroups—which is a typical subgroup size on NVIDIA and Intel GPUs, while it is typically 64 on AMD GPUs—sample the patch "row-wise" in $\lceil \frac{t_u}{31} \rceil$ "columns" (if we call parameter direction u a "row" and parameter direction v a "column" for the sake of tangible description) and share data among neighboring subgroups. Our approach is illustrated in Figure 5.7. Each subgroup keeps the data of two rows in registers so that neighboring values in v parameter direction are available. Neighboring values in u parameter direction are read from neighboring subgroups through `subgroupShuffle` operations. The values of the neighboring samples in u and v directions are used to calculate the normal vector for the current sample u_x, v_y through the cross product of tangent vector $\mathbf{t} = f_p(u_{x+1}, v_y) - f_p(u_x, v_y)$ and bitangent vector $\mathbf{b} = f_p(u_x, v_{y+1}) - f_p(u_x, v_y)$. The normal $\mathbf{n} = \mathbf{t} \times \mathbf{b}$ is required for shading computations. We avoid sampling $f_p(u, v)$ multiple times with the same parameters. The last u column does not produce points since it does not have a neighbor with higher subgroup index. Therefore, only 31 subgroups write pixel values, while the last subgroup only provides its data to the second to last subgroup.

Our Point-based rendering variant also features a continuous LOD configuration based on **adaptive sampling**, where the number of samples in u and v directions are not calculated based on the threshold parameters t_u, t_v, but instead on the approximated

screen distance extents e_{u_i}, e_{v_i}, which are measured during Patch Subdivision. The adaptive sampling configuration processes patch i in $\left\lceil \frac{e_{u_i}}{31} \right\rceil$ columns, taking row-wise steps of size $\Delta v = \frac{v_{imax} - v_{imin}}{e_{v_i}} - \varepsilon$. The adaptive sampling variant assumes $f_{p_i}(u, v)$ to distribute samples somewhat uniformly across $\overline{u_i}, \overline{v_i}$, since it takes approximately one sample across the screen space-based extents e_{u_i} and e_{v_i}. This variant often leads to much faster rendering speeds but runs the danger of producing small holes (often the size of 1×1 pixel) in the rendered output. In some cases, they can be prevented by increasing the ε values.

5.5.4 Render Targets

Since our Tessellation-based rendering variants are rendered with classical graphics pipelines, their outputs are typically rasterized into a framebuffer's color attachment. This is the only option to profit from hardware-accelerated multi-sampling (MS) and its resolve operation. MS evaluates the depth test for multiple subsamples of one fragment while limiting the color samples—or fragment shader invocations—to one per fragment. I.e., it enables higher depth resolution in a hardware-accelerated manner, which is especially beneficial for fine-structured geometry such as the curtains shown in Figures 5.1d and 5.1e, which consist of many small yarn or fiber curves like those shown in Figures 5.3b and 5.3c. Rendering those with SS or MS produces much less Moiré patterns in the rendered output thanks to the higher geometry sampling density. Eight subsamples per pixel are currently the maximum on modern NVIDIA and AMD GPUs [Wil24]. If more evaluations are desired for a single fragment, a combination of SS and MS has to be established. For many of our evaluations presented in Section 5.6, we utilized a configuration of 4xSS (i.e., resolution doubled in each dimension) combined with 8xMS framebuffers. In total, this leads to 32 depth evaluations, and four color evaluations for each fragment. The final rendering results are produced by first resolving MS through hardware resolve [Khr24ras], and then averaging the resulting four corresponding fragments of the high-resolution color attachment into one.

We use color attachments as render targets for all our tests presented in Section 5.6 that use a Tessellation-based rendering variant since it is generally faster than the alternative—which is storing the rendering output into a 64-bit integer image. The latter is fundamentally the same approach as taken by Nanite [KSW21]. Depth writes and also the depth test of a graphics pipeline can be disabled in this case. The resulting depth and color values are instead written in software through atomic operations from fragment shaders. By combining depth and color values into a single 64-bit integer value, as illustrated below, `atomicMin` operations ensure that parallel writes will produce the correct rendering result.

Since our Point-based rendering variants operate within compute shaders, they cannot use the hardware's depth testing functionality, which is only accessible when rendering via graphics pipelines into framebuffers with an attached depth attachment. Hence, we have to use a 64-bit integer image and store their rendering results via atomic operations. In general, it is undesirable to leave holes in the rendered result of a connected parametrically defined surface. Therefore, useful rendering settings will rather try to oversample a parametric function to some degree, so that at least one sample is taken per output fragment. This leads to some amount of overdraw—meaning multiple `atomicMin` operations accessing the same fragment in the 64-bit output image—which is generally a tradeoff that must be taken to avoid holes with our implementation.

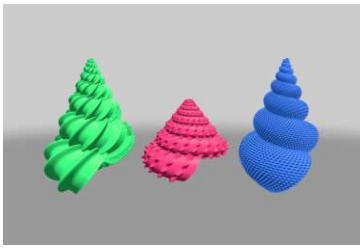
5.5.5 Anti-Aliasing with Point-Based Rendering

The Point-based rendering approach described in Section 5.5.3 writes its results into a 64-bit image, as described in Section 5.5.4. A potential problem with this approach is that even when a given pixel is oversampled, one of the samples—the one with the smallest depth value—overwrites all the other samples, potentially producing aliased results. In this section, we describe an alternative variant that first gathers samples in small, virtual, local framebuffers in shared memory and then resolves them in software, producing anti-aliased results. We refer to this variant by Point-based$_{\text{local FB}}$, and denote the variant which writes directly into the 64-bit image as Point-based$_{\text{direct}}$.

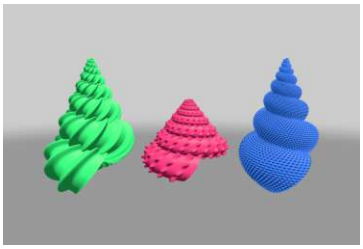
With the Point-based$_{\text{local FB}}$ variant, the render target is subdivided into 16×16 pixels sized local framebuffers which reside in shared memory. Since it would be highly inefficient to render each patch into each of these tiles, we propose a patch-to-tile assignment step that selects all the potentially relevant patches per tile based on each patch's axis-aligned bounding box (AABB). Whenever a patch's AABB intersects a tile's bounds, it is assigned to this tile. Dividing the screen into 16×16 tiles can look like shown in Figure 5.8d. Depending on the chosen screen distance threshold values t_u and t_v, a heatmap showing how many patches have been selected per tile can look like shown in Figure 5.8e for the different seashell variations from Figures 5.3d to 5.3f.

The patch to tile assignment step must be scheduled after the Patch Subdivision stage and, naturally, before the Rendering stage (i.e., between stages 2 and 3 in Figure 5.4). It is implemented in a compute shader with one thread per patch that has been scheduled for rendering by the Patch Subdivision stage. This thread assigns the patch to all tiles which overlap with the patch's AABB. In our tests, 16×16-sized tiles showed favorable performance over larger tile sizes. Overall, this intermediate step has a very low impact on performance: Even in a relatively demanding scenario—namely 60k yarn curves, 358k fiber curves, and six seashell models, covering the entire screen so that all tiles get patches assigned—this step does not take longer than 0.14ms for a resolution of 1920×1080 on an RTX 3070. Further performance evaluations are presented in Section 5.6.1.

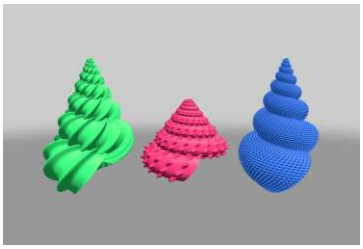
Local framebuffers have the same format as the global render target: 64-bit integer as described in Section 5.5.4. For 2×2 super-sampling, 32×32-sized local framebuffers must



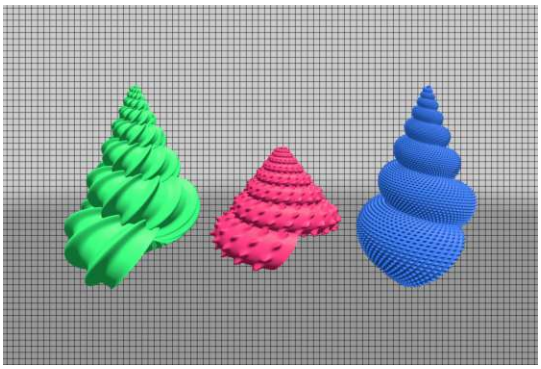
(a) *Point rendering, aiming for 1spp*



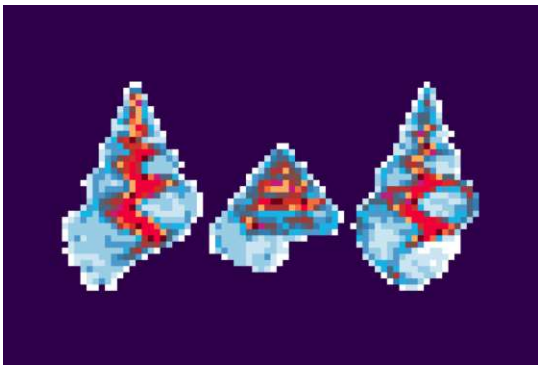
(b) *Point rendering, aiming for 9spp (oversampling 3× in both, u and v directions)*



(c) *Point rendering into local framebuffers, 3×3 px per output px, enabling up to 9xSS if all pixels get values assigned.*



(d) *There is one local framebuffer for each 16×16 region of the screen.*



(e) *Heatmap indicating the numbers of patches assigned to each 16×16 tile. Red regions represent higher patch counts.*

Figure 5.8: *These images show different results of our point rendering variants. Figures 5.8a and 5.8b are results from directly rendering into the 64-bit integer image, while Figure 5.8c shows a result from first rendering into small, virtual, local framebuffers per 16×16 tile (shown in Figure 5.8d). Relevant patches are first assigned to each tile. A corresponding heatmap illustrating the assigned patch counts per tile is shown in Figure 5.8d.*

be allocated; for 3×3 they are sized 48×48 accordingly. For each tile, the following steps are performed:

1. Initialize each pixel in the local framebuffer to the clear color.
2. Point-render each selected patch into the local framebuffer in the same manner as described in Section 5.5.3.
   - Discard points outside of the local framebuffer bounds
   - `atomicMin` operations now access shared memory
3. Average each 2×2/3×3 region in the local framebuffer, producing one output pixel. Write the latter into the global render target.
   - Regard only entries that have been set, i.e., are different from the clear value

The main benefit of POINT-BASED$_{\text{local FB}}$ are super-sampled results, which are not possible with our POINT-BASED$_{\text{direct}}$ variant. The effect can be observed Figure 5.8: While just

increasing the sample count (result shown in Figure 5.8b compared to Figure 5.8a) succeeds in filling some holes, it is unable to get rid of Moiré patterns. Only Point-based$_{\text{local FB}}$ is able to reduce them as can be seen in Figure 5.8c—most notably on the blue seashell on the right, which has a lot of tiny geometric details on the surface.

5.5.6 Rendering Variants and Configurations

In the previous sections, we have presented two fundamentally different rendering approaches: One uses the rasterization-based graphics pipelines in conjunction with the hardware tessellator to amplify geometry, and the other uses point-rendering from compute shaders. Each of these variants can be configured in different manners, adapting them to the rendering requirements of different parametric objects.

Major configuration options for the Tessellation-based rendering variants are:

- Using 1 sample per pixel (noAA), MS, SS, or a combination of MS and SS, to increase the sample count per pixel
- Using different tessellation levels
- Varying screen distance thresholds t_u and t_v, affecting patch sizes

The latter has an effect of approximate screen-space extents of the patches output by the Patch Subdivision stage. In combination with the tessellation levels, patches can be generated which lead to sub-pixel-sized triangles, or to triangles covering multiple pixels in the resulting framebuffer. Adaptive tessellation levels, as described in Section 5.5.2, lead to more homogeneous triangle sizes, while fixed tessellation levels can help with avoiding artifacts on patch borders and for generating patches with sub-pixel geometric detail for the Rendering stage. The latter is especially useful in combination with MS or SS framebuffers, so that multiple samples can be captured per pixel, leading to super-sampled and anti-aliased results. In terms of MS, the typical maximum number of subsamples on modern GPUs is eight [Wil24]. If more subsamples per pixel are desired, SS framebuffers or a combination of MS and SS must be utilized. For example, combining a 4xSS framebuffer with an 8xMS framebuffer format, gives 32 samples for one pixel, allowing to capture a lot of sub-pixel detail. As can be seen in our evaluations in Section 5.6, our rendering method still renders at high FPS even for 4xSS+8xMS configurations.

Major configuration options for the Point-basedrendering variants are the following:

- Using the Point-based$_{\text{direct}}$ or Point-based$_{\text{local FB}}$ rendering approach
- Varying the sample density per patch
- Varying screen distance thresholds t_u and t_v, affecting patch sizes

The differences between Point-based$_{\text{direct}}$ and Point-based$_{\text{local FB}}$ are mainly that the former generally leads to better rendering performance, while only the latter is able to produce anti-aliased rendering results. More technical details are described in Section 5.5.5. Varying t_u and t_v typically has an impact on performance: While decreasing them leads to higher computational load in the Patch Subdivision stage (more subdivisions, more

LOD steps), it often has a positive impact on rendering performance since workload per compute thread (pixel filling) decreases and workload can potentially be distributed better across the GPU's computing clusters. Increasing the sample density per patch also affects the computational workload during rendering and leads to more memory transfers due to overdraw, but is often necessary to fill holes in the rendered output.

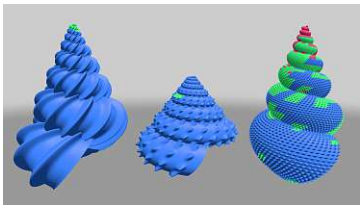
For our evaluations presented in Section 5.6, we mainly focus on the configurations TESSELLATION-BASED with noAA, TESSELLATION-BASED with 4xSS+8xMS, and POINT-BASED$_{\text{local FB}}$ with 4xSS. We leave out POINT-BASED$_{\text{direct}}$ from most comparisons because it is unable to produce anti-aliased rendering results.

5.5.7 Hybrid Rendering

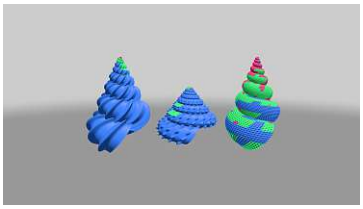
Some of the rendering variants described in Section 5.5.6 are better suited to certain scenarios than others. Tessellation with noAA or POINT-BASED$_{\text{direct}}$ generally lead to better performance than the variants producing anti-aliased results. Depending on the properties of a parametric object, one or the other variant would be better suited for rendering it. For some parametric objects, it depends on factors such as distance to the camera and framebuffer resolution whether or not its parametric description produces sub-pixel features for certain rendered areas. A parametric yarn curve (like shown in Figure 5.3b or a parametrically defined sphere (like shown in Figure 5.3g) will never have sub-pixel details on its surface. However, super-sampling might still be desirable when rendering them to get higher resolution for these objects' boundaries to the background or boundaries of other objects. The curtains consisting of several thousands of yarn or fiber curves shown in Figures 5.1d and 5.1e would be an example of such a situation, especially when the yarn or fiber curves are so small in screen space that they cover only a few pixels or even less than a pixel. Rendering fiber curves (like shown in Figure 5.3c) or the different seashell variants (shown in Figures 5.3d to 5.3f) often leads to situations where the surface itself leads to sub-pixel detail. Such a situation is established by the rendered seashells shown in Figures 5.8a to 5.8c: These seashell models are fundamentally a self-repeating shape, gradually scaling their geometric features smaller towards the top region. Consequently, also their surface features get scaled ever smaller, eventually leading to sub-pixel geometric detail at some point. This is most obvious with the blue seashell model on the right, which has small bumps across its surface. Pixel-level sampling in Figures 5.8a and 5.8b leads to noticeable aliasing, while sub-pixel-level sampling and averaging leads to an anti-aliased result in Figure 5.8c.

In this section, we propose a hybrid rendering variant, which performs a spot-check on each patch that is about to be scheduled for rendering during the PATCH SUBDIVISION stage. If this spot-check detects sub-pixel variations, the following actions are performed:

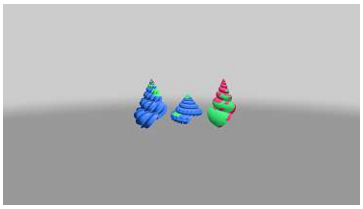
1. Do not schedule the patch in question for rendering.
2. Mark the patch to indicate that it has sub-pixel details that shall be rendered in an anti-aliased manner.
3. Re-schedule the patch for evaluation in the next LOD step.



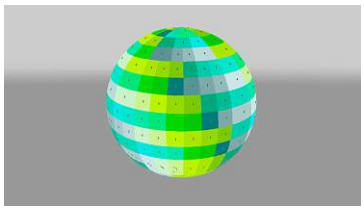
(a) *With the camera close to the seashell models, it can be seen that only few patches are scheduled for super-sampled rendering.*



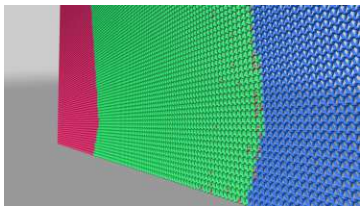
(b) *With the camera moving away, more patches are deemed to require super sampling.*



(c) *With the camera further away, all the patches of the seashell model on the right have been found to show sub-pixel detail.*



(d) *Spot-checks are performed in the middle of patches for a region of 7×3 pixels.*



(e) *Sub-pixel feature detection applied to a curtain model consisting of 60k yarn curves*

Figure 5.9: *Figure 5.9d shows the positions of the spot-checks for sub-pixel feature detection during the Patch Subdivision stage. In Figures 5.9a to 5.9c and 5.9e, the effects of a two-level sub-pixel feature detection approach are visualized: The patches colored in blue have not been found to have sub-pixel features in native resolution and are consequently scheduled for rendering with noAA. The patches colored in green have been found to have sub-pixel features w.r.t. native resolution and have been scheduled for re-evaluation once, but haven't been found to have sub-pixel features when tested against an 8xMS framebuffer configuration. The patches colored in red have been detected to have sub-pixel features also w.r.t. the increased resolution of 8xMS and have consequently been scheduled for rendering with a 4xSS+8xMS rendering variant.*

More details about the approach are given in Listing 5.2. It shows a part of a possible GLSL compute shader implementation for the LOD steps with hybrid rendering enabled. The code assumes that the configured workgroup size is equal to the subgroup size so that one workgroup processes one patch, and can do so efficiently using subgroup operations. Subgroup operations are used for estimating patch extents as described in Section 5.5.1 and Figure 5.6. They are also used for the aforementioned spot-check for detecting sub-pixel variations. In Listing 5.2, this check is performed within the function `hasSubpixelFeatures` (invoked in Line 25) and evaluates a 7×3 pixel large region in the middle of a patch to be scheduled for rendering in order to detect if it has sub-pixel variations or not. These spot-check regions are highlighted in Figure 5.9d.

The fundamental task of one workgroup in the context of a LOD step is to analyze a patch (via the call to `analyzePatch` in Line 12) and then either schedule its split parts for

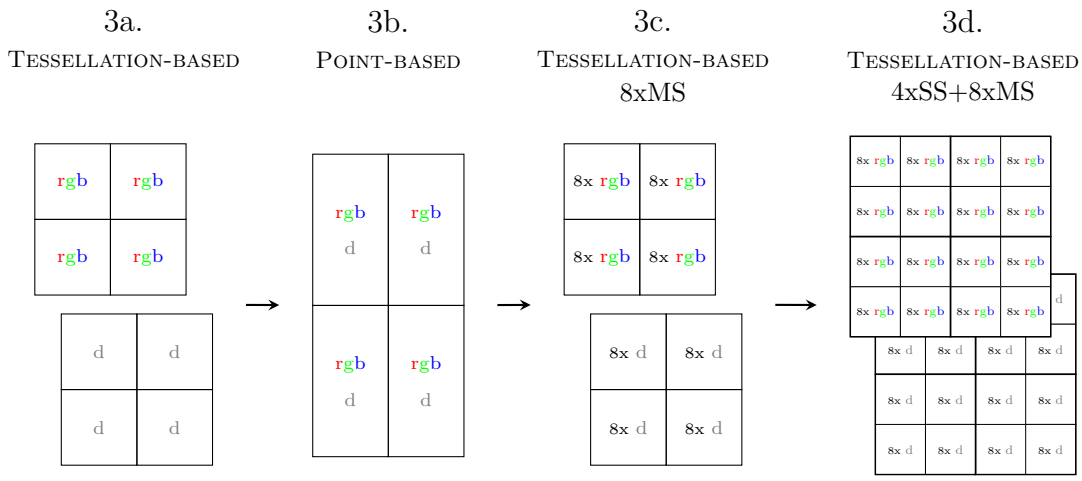


Figure 5.10: *Different rendering variants require different render target formats. Tessellation-based performs best in conjunction with framebuffers with color and depth attachments, and they enable MS, providing multiple sub-samples per pixel (e.g., eight depth and color samples). Point-based rendering targets 64-bit integer images which store both, color and depth, in one 64-bit integer value as described in Section 5.5.4. The order of rendering variants indicated in this figure is important to not lose any detail: Lower resolution targets are rendered first. The highest resolution comes last. Depth and color data must be transferred between the different render target formats.*

re-evaluation (in Line 19), or schedule a part that does not need to be split for rendering (in Line 46). The hybrid rendering approach adds the code from Line 24 to Line 44, introducing another decision on whether to schedule the patch for re-evaluation or for rendering. If `hasSubpixelFeatures` in Line 25 detects that normals sampled at subpixel locations differ from their respective interpolated normals—interpolated bi-linearly from the corner points of an approximately 1×1 pixel large region—significantly, then the patch is scheduled for re-evaluation at a higher effective resolution within an MS or SS scenario in Line 35. If most of the subsamples' normals are approximately equal to their interpolated counterparts, the patch is assumed to not have severe sub-pixel variations and therefore, scheduled for rendering with the rendering method currently set for the patch in question in Line 40. Evaluating a patch at an effective higher resolution corresponds to decreasing the screen distance thresholds t_u and t_v with respect to the original resolution, as it is done in Line 32. For example, rasterizing into an 8xMS framebuffer provides eight times more samples than in the noAA case, which corresponds to a resolution increase by a factor of $\sqrt{8}$ in each direction; a 4xSS framebuffer corresponds to a resolution increase by a factor of 2, and a 4xSS+8xMS framebuffer corresponds to a resolution increase by a factor of $\sqrt{32}$. It shall be noted that implementations do not have to stick to these exact adaptions of the screen distance threshold values, but they can serve as a rough guideline. The point is that these factors correspond to approximately how much additional geometric detail can be captured by such configurations.

Listing 5.2: *Partial compute shader code for an LOD step, focussing on the part of how patches are processed for hybrid rendering.*

```
// Main function of a compute shader that runs for every LOD step
void main()
{
  uint patchId = getPatchIdOfCurrentWorkgroup();
  vec4 uvParamRange = getPatchParameterRange(patchId);
  int  rv = getRenderVariantForPatch(patchId);
  vec2 tutv = getScreenDistanceThresholdsForPatch(patchId);
  // ...
  // Evaluate whether or not a patch shall be split according
  // to the approach shown in Figure \ref{fig:patch-eval-vis}:
  vec4 outFromToUV[4];
  int  splitParts = analyzePatch(patchId, tutv, outFromToUV);

  if (splitParts > 0) {
    // Schedule each part for re-evaluation in the
    // subsequent LOD step:
    if (subgroupElect()) {
      for (int i = 0; i < splitParts; ++i) {
        scheduleForNextLodStep(patchId, outFromToUV[i]);
      }
    }
  }
  else {
    if (useHybridRendering(patchId) && hasMoreDetailedRenderVariant(rv)) {
      if (hasSubpixelFeatures(middleOf(patchId))) {
        if (subgroupElect()) {
          int rvNext = getNextMoreDetailedRenderVariant(rv);
          // Will become effective in the subsequent LOD step:
          setMoreDetailedRenderVariant(patchId, rvNext);
          // Based on the render variant, adapt t_u and t_v:
          float d = getDivisorForScreenDistanceThresholds(rvNext);
          tutv = tutv / d;
          setScreenDistanceThresholdsForPatch(patchId, tutv);
          // Schedule for re-evaluation, not for rendering:
          scheduleForNextLodStep(patchId, uvParamRange);
        }
      }
      else {
        if (subgroupElect()) {
           scheduleForRendering(patchId, uvParamRange, rv);
        }
      }
    }
    else {
      if (subgroupElect()) {
         scheduleForRendering(patchId, uvParamRange, rv);
      }
    }
  }
}
```

All operations that schedule a patch either for re-evaluation in the subsequent LOD step or for rendering are enclosed by `if (subgroupElect())` statements [Khr24sub] to ensure that only one single thread schedules the patch for re-evaluation or rendering. Without these statements, each thread would schedule the same patch, which would be redundant and immensely wasteful in terms of performance. The rest of the code in Listing 5.2 is supposed to remain executed in parallel on 32 threads (of an 8×4 sized workgroup), which is the maximum subgroup size on NVIDIA and Intel GPUs. The subgroup size of 8×4 is also the reason why we chose to evaluate 7×3 pixels for each patch: It can be done efficiently in one subgroup iteration. The respective last "rows" and "columns" do not evaluate a separate pixel but merely provide data—similar as our point rendering approach described in Figure 5.7. By keeping most of this shader's implementation parallel, threads can work together. This can be especially beneficial during `analyzePatch` (invoked in Line 12) and during `hasSubpixelFeatures` (invoked in Line 25), where an efficient implementation can make use of subgroup operations [Khr24sub] to share data between threads: For distance computations between samples during `analyzePatch` , or for computing normals through the cross product of tangents and bitangents, which can be computed by getting neighboring threads' positions, during `hasSubpixelFeatures` .

In our implementation of `hasSubpixelFeatures` we take a total of nine sub-sample positions, sample the parametric function at these offsets, compute normals, and compare them with the bi-linearly interpolated normals from the ≈1px sized region. In an attempt to capture a wide range of frequencies, we select the sub-sample positions at all u, v-offset combinations of dividing each ≈1px parameter range by the first three prime numbers: $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{5}$. The effect of our hybrid rendering approach can be observed in Figure 5.9: It shows a three-tier hybrid approach, first evaluating patches against the native resolution with noAA, then against an 8xMS increased resolution, and finally rendering the patch with 4xSS+8xMS if evaluation at the 8xMS-level still shows sub-pixel variations.

Care must be taken when creating an application that shall be able to dynamically use different rendering variants since they might require different render target formats and the application needs to switch between them during a frame. While Tessellation-based render variants preferably render into a framebuffer's color attachment (as described in Section 5.5.2), Point-based render variants write into 64-bit integer images (as described in Section 5.5.3). Depth and color information must be transferred between these two formats—for these cases, compute shaders must be used to read and extract color and depth values from 64-bit integer images, or to combine and write them to 64-bit integer images. For MS framebuffers, transferring color and depth values from previous rendering variants into all the relevant samples must be ensured as well. Figure 5.10 shows a setup supporting different render variants and indicates the order of rendering them within the Renderingstage: It is important to start with the render variants that use the coarsest render targets and always progress to render targets that provide finer detail. Failing to follow this order will have the effect of losing resolution. Keeping this order is essential to get correct rendering results at the geometry borders since framebuffers of higher resolution capture more depth samples. In our implementation, we

transfer the the previously rendered noAA results into an 8xMS framebuffer by rendering a full-screen quad. Writing color and depth values in a fragment shader ensures that all the subsamples get the already rendered color and depth values assigned. Also, color and depth transfer from the 8xMS framebuffer into the 4xSS+8xMS framebuffer attachments is performed through a full-screen quad for the same reason. We show in Section 5.6.2 that such a hybrid rendering setup can help to keep frame rates constant, but it must be noted that our implementation also incurs some significant overhead from transferring the data between the different render targets as shown in the following table:

	3a. → 3b.	**3b. → 3c.**	**3c. → 3d.**	**Total**
Values written	2.1M	16.6M	66.4M	85.0M
Bytes transferred	17MB	133MB	530MB	680MB
Time	0.05ms	0.11ms	0.78ms	0.94ms

The values written refer to color or depth values that have to be written by the render output units for a resolution of 1920×1080. Color values of 8bit per channel + 32bit depth values amount to the stated bytes transferred. The milliseconds have been measured on an NVIDIA RTX 3070 using GPU timestamp queries, without rendering any patches. It shall be noted that timestamp queries synchronize between measurements and that on a modern GPU, the actual data transfer overhead might be less severe since a GPU might be able to schedule other work in parallel. However, the overhead is not insignificant—especially the data transfer of the 8xMS data into the 4xSS+8xMS framebuffer.

5.6 Results

We evaluate our method and its rendering variants based on the following parametric functions, which have different characteristics and therefore pose different challenges to a rendering method:

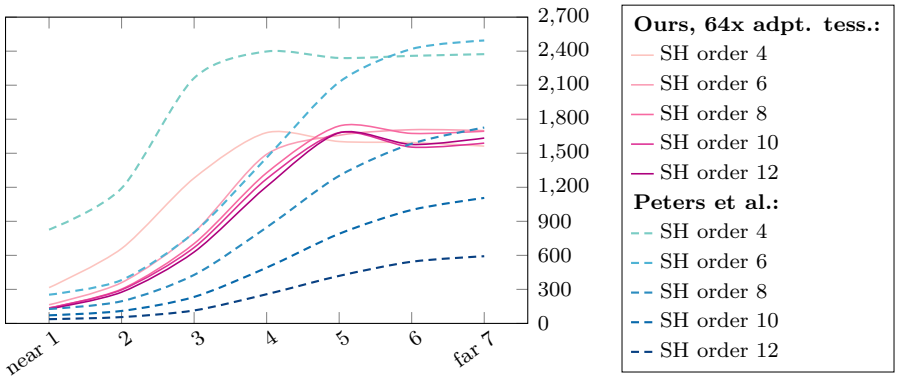
- SH glyphs
- Parametric plain-knit yarn curves
- Parametric seashells

One challenge when rendering SH glyphs—especially such of higher order—is that each sample is computationally expensive. Furthermore, parameters are distorted very non-uniformly across the entire parameter range. The challenge with rendering plain-knit yarn curves is the vast amount that is required for a typical scene. While these two parametric functions produce smooth surfaces, the parametric seashell model has very small-scale surface features, leading to sub-pixel geometric detail for most of our test setups. The results presented in Figures 5.12, 5.13, 5.15 and 5.16 were gathered after a GPU warm-up phase of 2 seconds from multiple camera positions. The camera is located

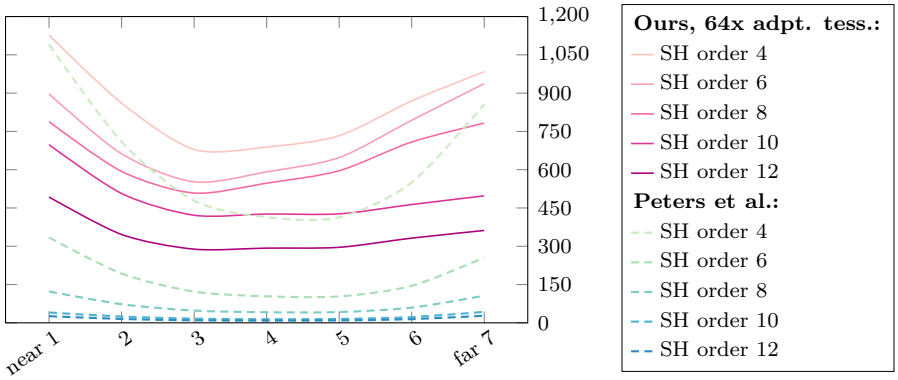
close to the object or center of the dataset for the first measurement and moves away with every further measurement. Each measurement result (such as FPS, number of pixels written, or number of render patches) represents the averaged data over a 5 second time span, during which the camera moved around the center of the object or dataset in one full circle—requiring the Patch Subdivision stage to generate and forward a different set of patches to the Rendering stage every frame.

SH Glyphs: Spherical harmonics are mathematical functions defined on the surface of a sphere and are parameterized using spherical coordinates (θ, φ) with $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$. These functions result from a linear combination of a set of orthonormal basis functions, an important property that makes them useful in a wide range of fields. They are described via band index ℓ and parameter m. ℓ also states the *order* of an SH. Higher orders allow the representation of higher frequencies but also come with an increase in computational complexity, making them expensive to evaluate and challenging to visualize. Thanks to their spherical parametrization, one way of representing them is by using a sphere with the distance from its surface to its center set to the result of the evaluated SH at the corresponding location (θ, φ), which aligns them with the definition of parametric functions.

The SH glyphs we use represent measurements from a so-called high angular resolution diffusion imaging (HARDI) [Tuc+02] dataset of a brain scan [Has+22], captured via dMRI and shown in Figure 5.1b. We compare our method with the state-of-the-art SH glyph rendering method by Peters et al. [Pet+23], which uses ray tracing to render high-quality SH glyphs in real time. Figure 5.11 shows the effect of using different SH orders for rendering. Our method shows remarkably consistent performance for varying SH orders, while the method by Peters et al. suffers from strongly decreasing performance with each SH order increase. While it renders more than 2000 FPS on an NVIDIA RTX 3070 for SH order 2, our method outperforms it for SH orders 8 and higher in single SH glyph rendering (Figure 5.11a) and already for SH orders 4 and higher for the large dataset (Figure 5.11b). Since higher SH orders reveal more detail about the measured data they represent, we have focused further tests on SH order 12: For single glyph rendering of SH order 12, Figure 5.12e shows that our method shows better performance across all GPUs. The Tessellation-based variant is optimal for rendering SH glyphs due to their smooth surface. Our recommended configuration is shown in Figures 5.12c and 5.12e, which produces almost pixel-perfect geometric detail. Even a configuration of our method with fixed tessellation factors outperforms the method by Peters et al. across all GPUs but does not improve rendering quality despite the higher geometric detail produced. Our method avoids the artifacts produced by the method by Peters et al., which are shown in Figure 5.12b, but it also introduces small, much less noticeable artifacts near the poles of the scaled sphere, an example of which is shown in Figures 5.21a and 5.21b. 5.12f to 5.12i show how Patch Subdivision increases the number of patches uniformly during the first four LOD steps to create sufficient geometric detail. Steps 5 and higher subdivide fewer patches or subdivide non-uniformly for larger distances to the camera since the subdivisions suffice already in more cases. Rendering the HARDI



(a) *FPS comparison of using different SH orders for rendering a single SH glyph, measured on an NVIDIA RTX 3070.* $t_u = t_v = 84$*, noAA, adaptive tessellation, and* $l_{max} = 64$ *have been used as the configuration for our method.*

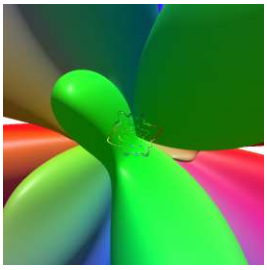


(b) *FPS comparison of using different SH orders for rendering the HARDI dataset containing 19,600 SH glyphs on an NVIDIA RTX 3070.* $t_u = t_v = 84$*, noAA, adaptive tessellation, and* $l_{max} = 64$ *have been used for our method.*

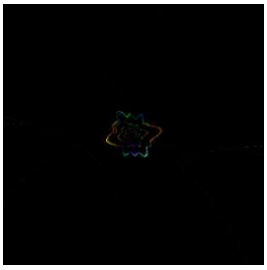
Figure 5.11: *These diagrams show the impact of using different SH orders for rendering the SH glyphs on rendering performance. The method by Peters et al. [Pet+23] is much more affected by varying SH orders in that regard.*

dataset, our method outperforms the method by Peters et al. even more strongly, as shown in Figure 5.13g. It even provides headroom for a 4xSS and 8xMS variant, as shown in Figures 5.13e and 5.13h. The image quality of this configuration is superior, which is especially noticeable during camera movements and can also be observed in Figure 5.13f.

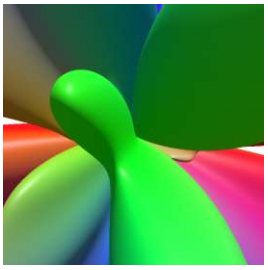
Plain-Knit Yarn: Gröller et al. [GRS95] provided an early parametric description of knitwear, but we use the more recent parametric plain-knit yarn curves described by Crane [Cra23]. Its fundamental shapes of yarn curves and fiber curves are shown in Figures 5.3b and 5.3c. We use an extruded version of them for our evaluations, namely one which takes the yarn direction ("tangent") as parameter u, and constructs a circle around each point along that direction via parameter v. We get an orthogonal vector ("normal")



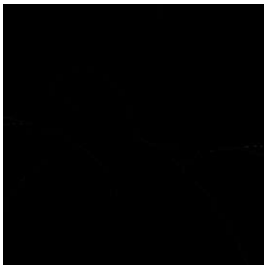
(a) *Rendered with Peters et al.*



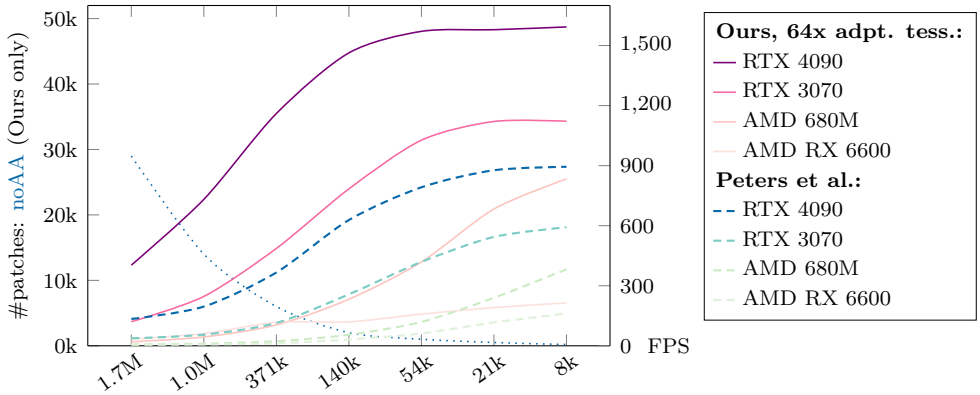
(b) *Diff. between reference and 5.12a*



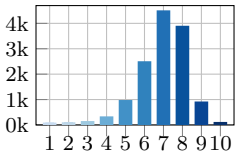
(c) *Renderd with ours with adaptive tess.*



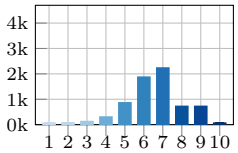
(d) *Diff. between reference and 5.12c*



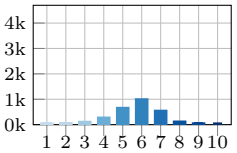
(e) *FPS rendered by different GPUs for different views of one single SH glyph, comparing our method ($t_u = t_v = 84$, noAA, adaptive tessellation, $l_{max} = 64$) to the method of Peters et al. The x axis shows the average number of pixels written (excluding overdraw). The dotted line represents the number of patches to be rendered, output by* PATCH SUBDIVISION.



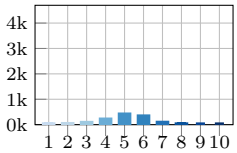
(f) *#patches out per* LOD *step (1.7M)*



(g) *#patches out per* LOD *step (1.0M)*



(h) *#patches out per* LOD *step (371k)*



(i) *#patches out per* LOD *step (140k)*

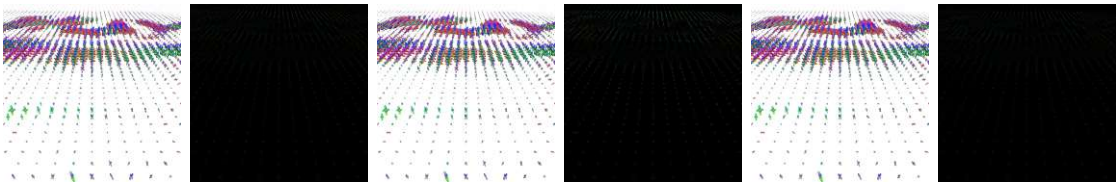
Figure 5.12: *For an SH glyph of order 12, Figures 5.12a to 5.12d show qualitative comparisons of the rendered results compared with a reference image produced with 16xSS and 8xMS. For the first measurement, the camera starts at the view producing Figures 5.12a and 5.12c and moves away for subsequent measurements. Figure 5.12e presents the performance of our variants in comparison to the method by Peters et al. Ours avoids their closeup artifacts. It also shows the number of patches to be rendered, produced in our* PATCH SUBDIVISION *stage—while Figures 5.12f to 5.12i show the average numbers of patches to be evaluated during each LOD step. No patches remain to be evaluated after LOD step 10 in any of these test setups.*

to the tangent and rotate it using Rodrigues' rotation formula [Rod40] around the tangent by an angle θ, which is parameter v in the range $[0, 2\pi)$. The source code repository containing the code for Crane's plain-knit yarn curves lists several implementations, none of which is able to render a large number of yarn curves in real time with good rendering quality and offers real-time changes to the code—our method enables this for at least 358k curves, as our results in Figure 5.15c show.

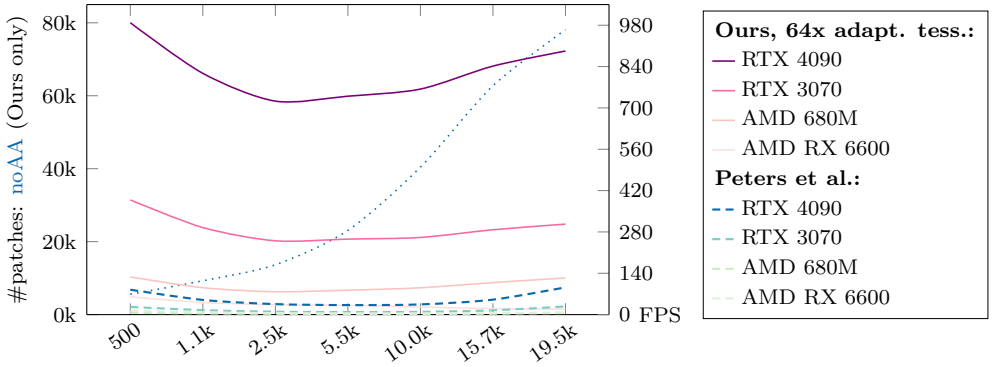
We show the performance comparisons of three different variants of our method on the example of a blue curtain composed of 60k yarn curves or 358k fiber curves in Figure 5.15. The configurations compared are a Tessellation-based variant with adaptive tessellation, $t_u = t_v = 62$, $l_{max} = 19$, and no anti-aliasing (noAA), the same configuration with 4xSS and 8xMS, and a Point-based variant with 4xSS. In both test cases, the noAA configuration suffers from severe aliasing artifacts such as Moiré patterns, which is due to the high number of yarn curves or fiber curves and the sub-pixel geometric detail produced by them, especially for larger camera distances. The configurations with SS produce visually satisfying results like those shown in Figures 5.1d and 5.1e. In most situations, Tessellation-based shows better performance numbers, but there's also a noticeable sweet spot for the Point-based variant, where it outperforms the former—namely when rendering a limited number of yarn curves or fiber curves at a relatively small scale. However, our implementation of the Point-based variant suffers from the problem that it produces a rendering result equivalent to that of conservative rasterization, which can impair visual quality especially when rendering thin geometry.

In contrast to the results of single SH glyph rendering, where all the geometry is raised from one single initial patch, Figures 5.15d to 5.15k show different subdivision characteristics since Patch Initialization already creates 358k patches. Many of them are culled for near-camera positions. Hence, only ≈12k remain after the first LOD step in Figure 5.15d, all of which need to be subdivided due to the camera distance and screen resolution. As the camera moves away, more fiber curves become visible but relatively fewer patches need to be subdivided. The SS variants in Figures 5.15h to 5.15k generally require more patch subdivisions due to the higher effective resolution.

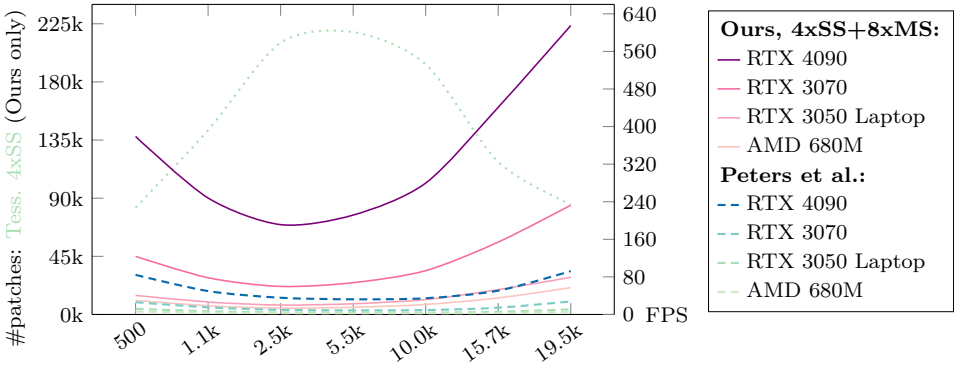
Seashell: The parametric seashell model follows roughly the construction guidelines by Wilson [Wil22], which allows the creation of a myriad of seashell variants by parameter variations, such as the ones shown in Figures 5.3d to 5.3f. Achieving satisfying rendering results for the variant shown in Figure 5.3f is challenging since its parametric model creates many tiny bumps on the surface, resulting in sub-pixel geometric detail when viewed from a distance or rendered in low resolution. For our tests with the parametric seashell function, we use the same configurations that we have used for the yarn curves tests, except for $l_{max} = 64$ to capture sub-pixel detail produced by the parametric seashell model for many view positions. Also in this case, configurations with SS produce much more satisfying visual results. The Point-based performance trails behind Tessellation-based configurations. Results of different configurations are shown in Figure 5.16.



(a) *Rendered with Peters et al.* **(b)** *Difference between 5.13a and reference* **(c)** *Rendered with our method, noAA* **(d)** *Difference between 5.13c and reference* **(e)** *Rendered with ours, 4xSS+8xMS* **(f)** *Difference between 5.13e and reference*

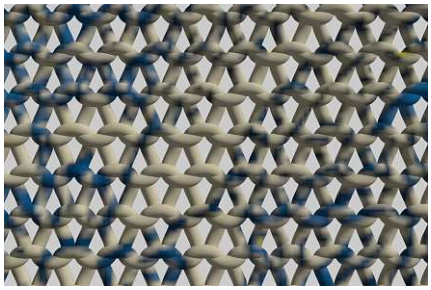


(g) *FPS rendered by different GPUs for different views of the HARDI dataset, comparing our method ($t_u = t_v = 84$, noAA, adaptive tessellation, $l_{max} = 64$) to the method of Peters et al. The x axis shows the average number of SH glyphs rendered for a certain view distance. The dotted line represents the patches output by our* PATCH SUBDIVISION *stage.*

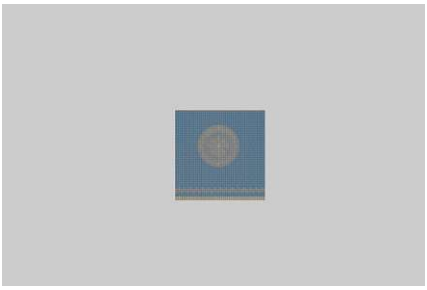


(h) *Comparing our method ($t_u = t_v = 84$, 4xSS, 8xMS, adaptive tessellation, $l_{max} = 64$) to the method of Peters et al. The x axis shows the average number of SH glyphs rendered for a certain view distance. The dotted line represents the number of patches to be rendered, generated by our* PATCH SUBDIVISION *stage, which is much higher than in Figure 5.13g due to 4xSS.*

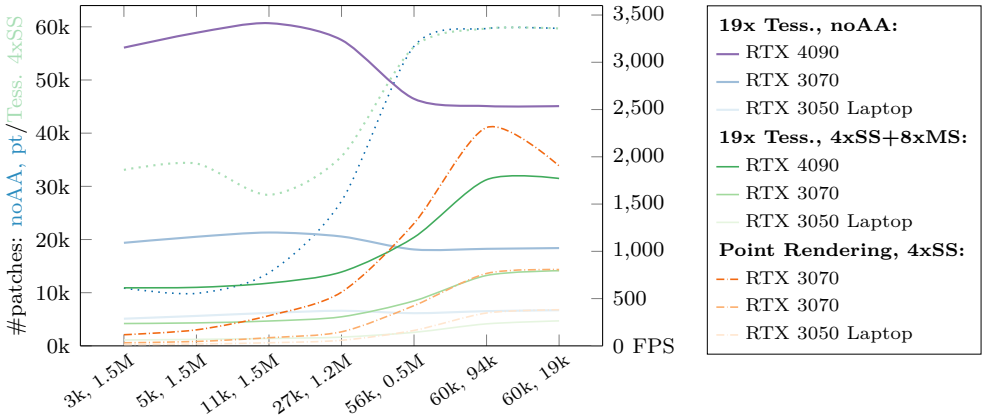
Figure 5.13: *For 19,600 SH glyphs of order 12, Figures 5.13a to 5.13f show qualitative comparisons between the results and a reference image which has been produced with 16xSS and 8xMS. Figures 5.13g and 5.13h compare the performance of our variants with the method by Peters et al. The camera starts at a view distance similar to the one producing Figures 5.13a, 5.13c and 5.13e and moves away for subsequent measurements.*



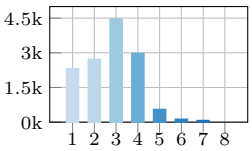
(a) *Initial camera position*



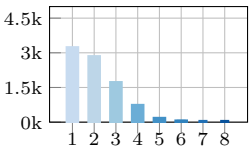
(b) *Final camera position*



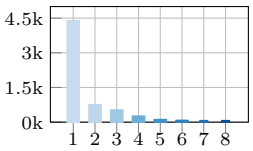
(c) *Comparing the performance of three variants on multiple GPUs. The x axis states tuples of number of yarn curves rendered, and pixels written (not counting overdraw) for the different camera distances between Figures 5.14a and 5.14b.*



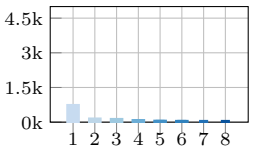
(d) *#patches out per step (3k, 1.5M)*



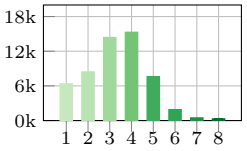
(e) *#patches out per step (5k, 1.5M)*



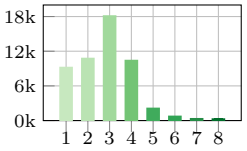
(f) *#patches out per step (11k, 1.5M)*



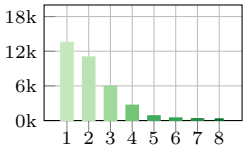
(g) *#patches out per step (27k, 1.2M)*



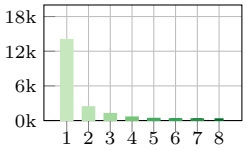
(h) *#patches out per LOD step Tess. 4xSS (3k, 1.5M)*



(i) *#patches out per LOD step Tess. 4xSS (5k, 1.5M)*

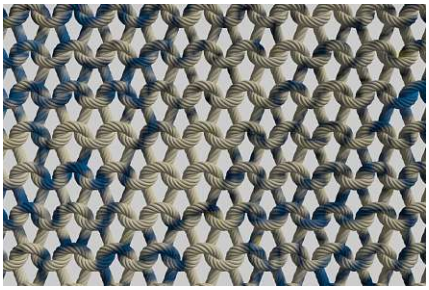


(j) *#patches out per LOD step Tess. 4xSS (11k, 1.5M)*

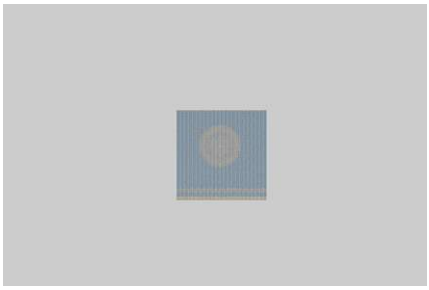


(k) *#patches out per LOD step Tess. 4xSS (27k, 1.2M)*

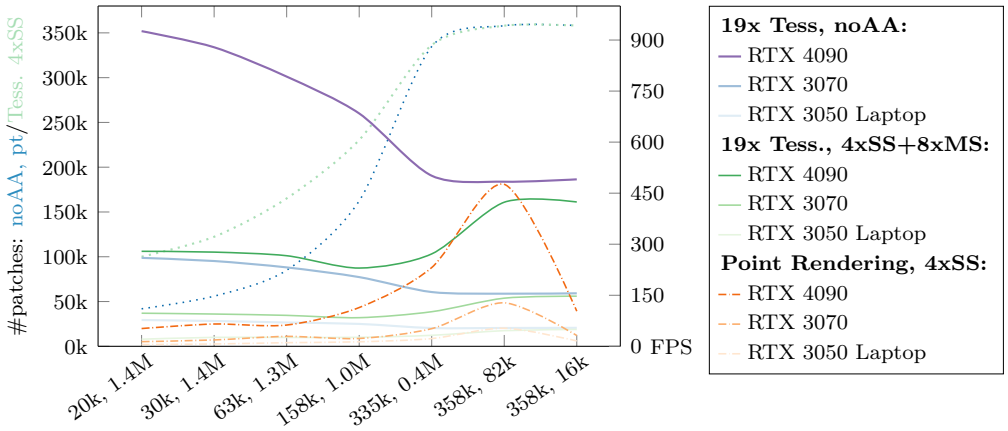
Figure 5.14: *Performance results and number of patches to be rendered of different configurations for 60k yarn curves. Figures 5.14d to 5.14g show the numbers of patches to be evaluated for the noAA and point rendering variants for each LOD step, while Figures 5.14h to 5.14k show these numbers for the tessellation 4xSS configuration.*



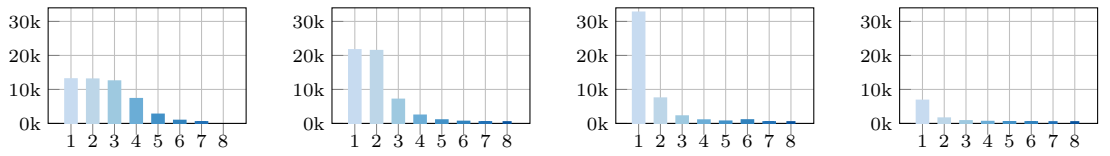
(a) *Initial camera position*



(b) *Final camera position*



(c) *Comparing the performance of three variants on multiple GPUs. The x axis states tuples of number of fiber curves rendered, and pixels written (not counting overdraw) for the different camera distances between Figures 5.15a and 5.15b.*

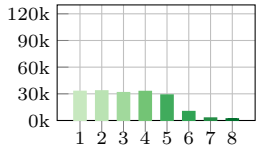


(d) *#patches out per step (20k, 1.4M)*

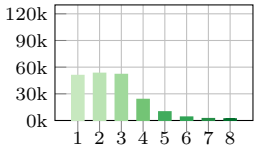
(e) *#patches out per step (30k, 1.4M)*

(f) *#patches out per step (63k, 1.3M)*

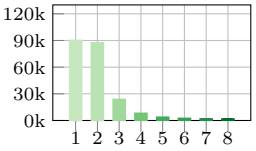
(g) *#patches out per step (158k, 1.0M)*



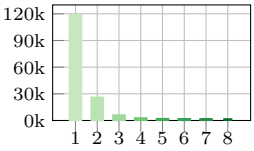
(h) *#patches out per LOD step Tess. 4xSS (20k, 1.4M)*



(i) *#patches out per LOD step Tess. 4xSS (30k, 1.4M)*

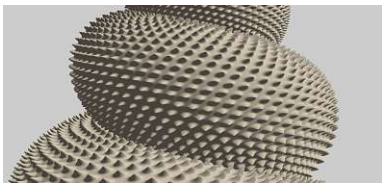


(j) *#patches out per LOD step Tess. 4xSS (30k, 1.4M)*



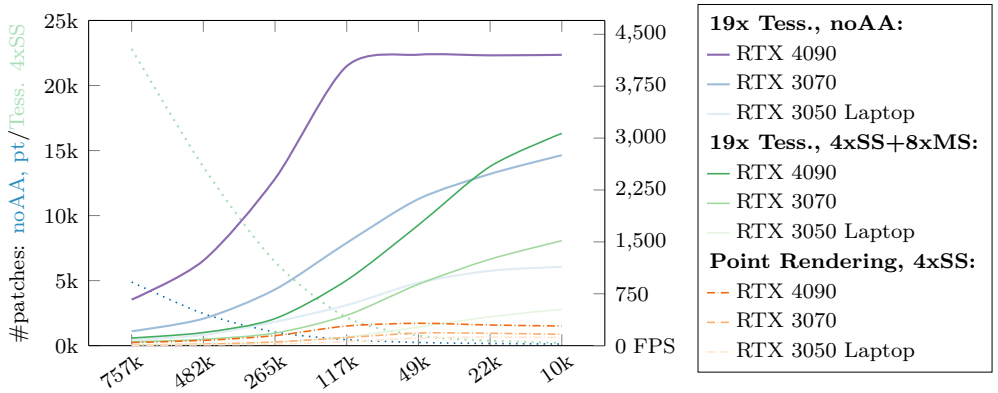
(k) *#patches out per LOD step Tess. 4xSS (30k, 1.4M)*

Figure 5.15: *Performance results and number of patches to be rendered of different configurations for 358k fiber curves. Figures 5.15d to 5.15g show the numbers of patches to be evaluated for the noAA and point rendering variants for each LOD step, while Figures 5.15h to 5.15k show these numbers for the tessellation 4xSS configuration.*

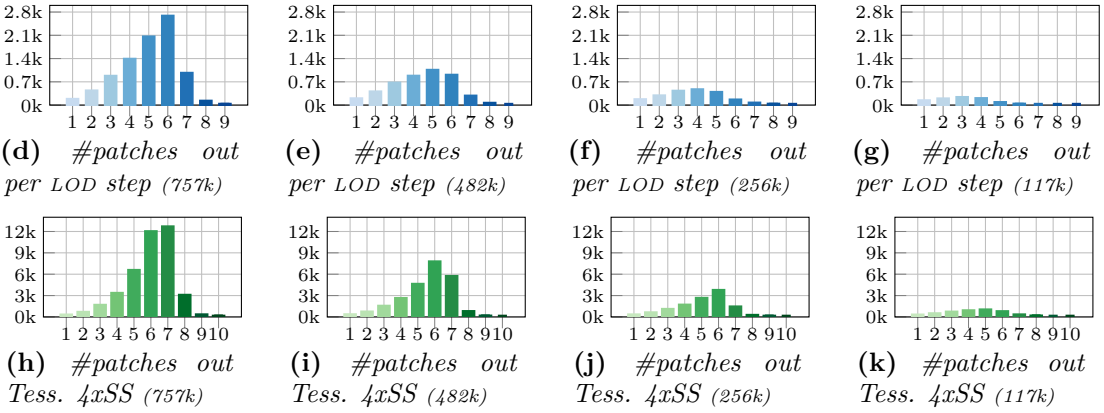


(a) *Initial camera position*

(b) *Final camera position*



(c) *Comparing the performance of three variants on multiple GPUs. The x axis represents different camera distances between Figures 5.16a and 5.16b, stating the number of pixels written on average (not counting overdraw).*



(d) *#patches out per LOD step (757k)*

(e) *#patches out per LOD step (482k)*

(f) *#patches out per LOD step (256k)*

(g) *#patches out per LOD step (117k)*

(h) *#patches out Tess. 4xSS (757k)*

(i) *#patches out Tess. 4xSS (482k)*

(j) *#patches out Tess. 4xSS (256k)*

(k) *#patches out Tess. 4xSS (117k)*

Figure 5.16: *Results of a parametric seashell with sub-pixel detail. Figures 5.16d to 5.16g show the number of patches output to be evaluated for noAA and point rendering configurations, Figures 5.16h to 5.16k for Tess. 4xSS configurations.*

5.6.1 Comparing Point-based Render Variants

Since POINT-BASED$_{\text{local FB}}$ first writes its results into small, virtual, local framebuffers, the overdraw with this variant is first captured by the local framebuffers in contrast to the global framebuffer. The single resulting color value after the software-resolve operation leads to only one single `atomicMin` operation into the global 64-bit integer target. While it would be reasonable to expect some performance improvement from this approach—due to the atomic operations targeting shared memory instead of global memory—this did not show in our measurements. The performance results of different configurations are presented in Figure 5.17. They reveal that the super-sampled rendering quality of POINT-BASED$_{\text{local FB}}$ variants takes a toll on performance. This can mainly be attributed to increased computational load within the point rendering shaders, requiring the additional clearing and resolve steps (as described in Section 5.5.5). The additionally required patch-to-tile assignment step never took more than 0.4ms in our measurements.

5.6.2 Hybrid Rendering Performance

Performance reslts of our hybrid rendering approach are presented in Figures 5.18 and 5.19. Both test cases show that a hybrid approach can lead to much more stable frame rates as the PATCH SUBDIVISION selects a suitable render method which ultimately keeps render load more stable. In contrast, fixed rendering with the 4xSS+8xMS configuration shows varying frame rates and severe FPS drops when the camera is near the parametric objects. The LOD stage takes even less time for the hybrid approach in the seashell test shown in Figure 5.18. This is a result of fewer patch subdivisions required with the hybrid approach, as the fixed 4xSS+8xMS configuration unconditionally aims for much smaller patch sizes. This behavior can be observed well in Figures 5.18h and 5.18i: As the camera is close to the parametric objects, they get subdivided into huge numbers of small patches. In contrast, the hybrid approach determines that super-sampled rendering is not required for many patches and hence, stops subdividing earlier as can be seen in Figures 5.18d and 5.18e, scheduling much fewer patches for rendering. Overall, this leads to a much less pronounced FPS drop when the camera moves closer to the parametric seashells. However, the chart also shows the fixed overhead of a hybrid rendering setup, as described in Section 5.5.7. This, unfortunately, leads to the hybrid rendering technique not being able to outperform the fixed 4xSS+8xMS configuration in all cases.

The results of 60k yarn curves reveal a general weakness of the hybrid rendering approach in situations with a high number of small initial patches—each one of the 60k yarn curves is scheduled as an initial patch during PATCH INITIALIZATION. Since the yarn curves are very small in screen space, many of them are found to better be rendered in higher detail, i.e., either 8xMS or 4xSS+8xMS. Therefore, many of them are rescheduled for re-evaluation. One such situation is shown in Figure 5.9e and as the camera moves further away, more patches are re-scheduled for re-evaluation. Since there is such a high number of initial patches, this scenario already constitutes a relatively high workload for the PATCH SUBDIVISION stage. As many patches are rescheduled, the workload increases even more. Figures 5.19b to 5.19d reveal that almost all input patches are scheduled

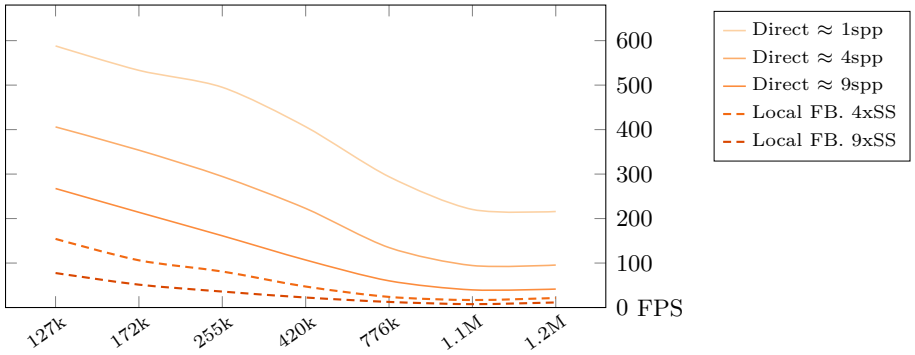


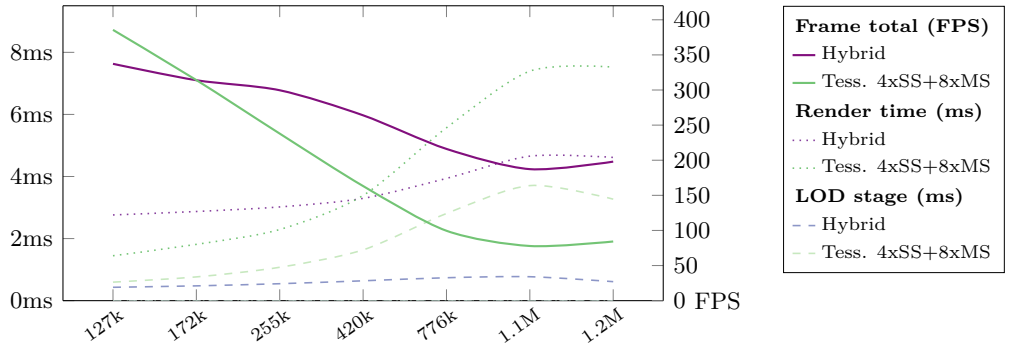
Figure 5.17: *Comparison of different point-based rendering variants, including three variants of* Point-based$_{direct}$ *with varying amounts of oversampling (and hence, overdraw), and two variants of* Point-based$_{local\ FB}$ *with different amounts of super-sampling, which also require different local framebuffer sizes. The scene used for testing is the one shown in Figure 5.8 with varying camera distances. The x-axis represents states the number of pixels written (not counting overdraw).*

three times for evaluation with the hybrid approach. Only as the camera moves closer in Figure 5.19e, the workload in the Patch Subdivision stage declines. As a consequence, we see high frame times of the LOD stage in Figure 5.19a. Only in the camera position which is nearest to the curtain, the hybrid rendering approach is able to outperform fixed 4xSS+8xMS rendering. The latter shows optimal patch output characteristics when the camera is further away, but subdivides vigorously as the camera moves closer. This can be observed well in Figures 5.19h and 5.19i.

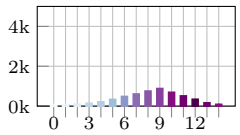
Overall, it can be stated that the hybrid rendering variant is well-suited for parametric objects which do not consist of a huge amount of initial patches so that the LOD step code presented in Listing 5.2 can figure out how much rendering detail is required. The hybrid rendering approach can help to keep frame rates much more stable in such situations.

5.7 Construction of Parametric Objects

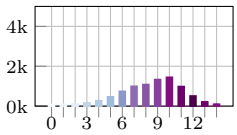
In this section, we give some general guidelines on the construction of parametric objects. While it is rather straightforward to model simple shapes like a sphere or a heart shape parametrically, things get more complicated when more complex shapes are constructed in code. An example of a more complex and animated parametric object is the "giant worm" model which is shown in Figure 5.20. It is constructed by several geometric shapes: Its main body is basically a distorted cylinder that follows the path of a Bèzier curve with added geometric detail on its skin. The skin detail is added via multiplication of several sin functions, parameterized on u and v multiplied with factors, in order to get a repeating pattern across the parameter ranges. The thickness of the worm's body is scaled via $\frac{1}{u+o}$, where o is a small offset so that the thickness does not evaluate to inf. The jaws consist of six patches (three for the skin on the outside, and three for



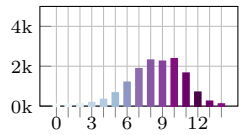
(a) *Performance results for rendering three seashell models, as shown in Figures 5.9a to 5.9c, comparing a three-tier hybrid variant with the performance of* Tessellation-based *with fixed 4xSS+8xMS. The x axis shows the number of pixels covered on the screen as the camera moves from a distance towards the parametric objects. The dashed lines show the milliseconds for rendering all the patches, and the dotted lines represent the milliseconds how long all the LOD steps of the* Patch Subdivision *took.*



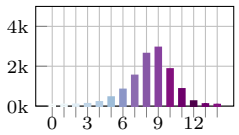
(b) *#patches out three-tier hybrid (127k)*



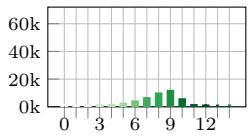
(c) *#patches out three-tier hybrid (255k)*



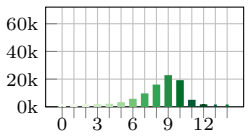
(d) *#patches out three-tier hybrid (776k)*



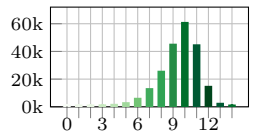
(e) *#patches out three-tier hybrid (1.2M)*



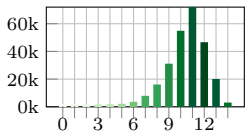
(f) *#patches out fixed 4xSS+8xMS (127k)*



(g) *#patches out fixed 4xSS+8xMS (255k)*



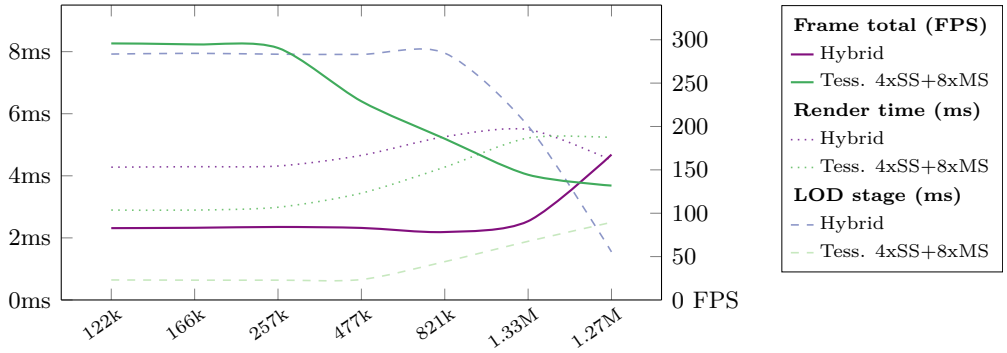
(h) *#patches out fixed 4xSS+8xMS (776k)*



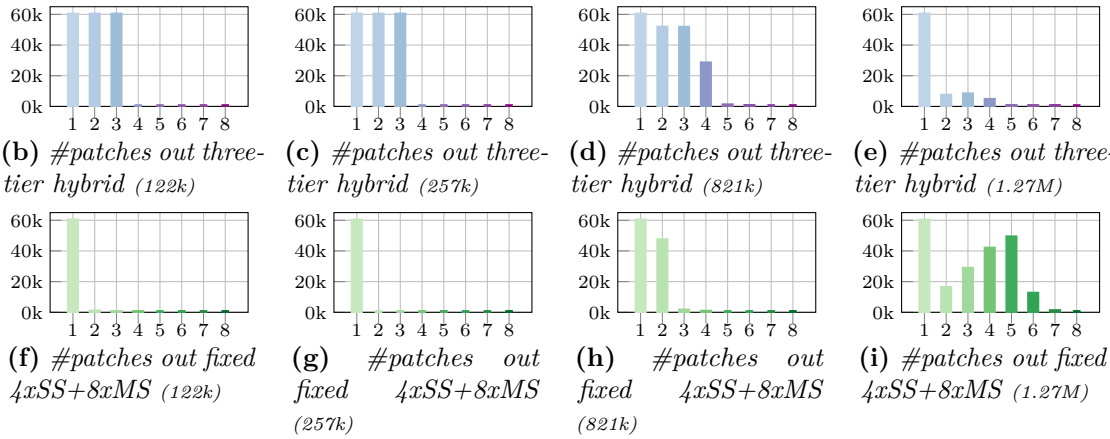
(i) *#patches out fixed 4xSS+8xMS (1.2M)*

Figure 5.18: *These diagrams show performance results achieved with our three-tier hybrid rendering technique which dynamically schedules patches for rendering with Tess. noAA, Tess. 8xMS, or Tess 4xSS+8xMS rendering variants for a scene with three seashells as shown in Figures 5.9a to 5.9c. Figures 5.18b to 5.18e put the numbers of output patches after each LOD step in perspective to Figures 5.18f to 5.18i.*

the inside of the mouth) that are brought into shape through trigonometric functions. They are positioned relative to the body shape, which can be seen as its parent. By sampling the parent shape at certain u, v parameter values, one gets a reference point for positioning child patches. The tongue is a partially distorted sphere. The distortion is performed based on its u and v parameter values. The teeth are many small cones, each one positioned relative to a certain position of a parent jaw patch. By changing parameters of, e.g., the Bèzier curve or by rotating the jaws, animations can be performed and child objects move along with their parent objects, since they are positioned relative to them.



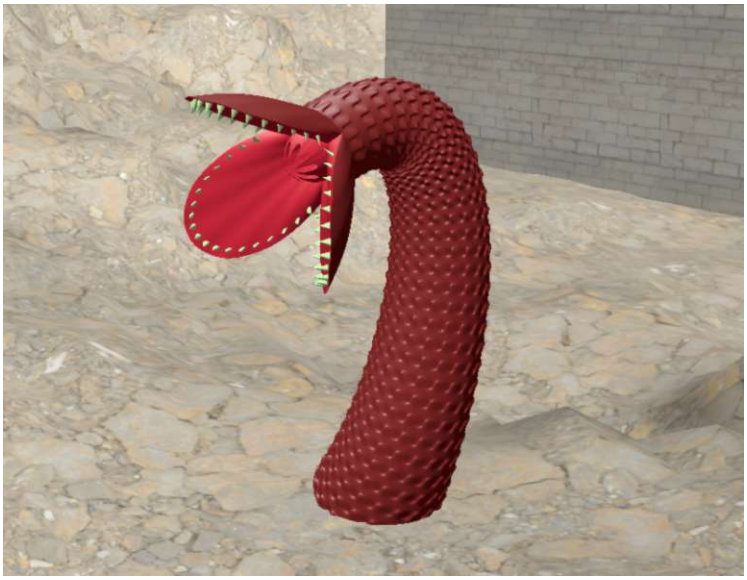
(a) *Performance results for rendering a curtain consisting of 60k yarn curves, such as the one shown in Figure 5.9e, comparing a three-tier hybrid variant with the performance of Tessellation-based with fixed 4xSS+8xMS. The x axis shows the number of pixels covered on the screen as the camera moves from a distance towards the parametric objects. The dashed lines show the milliseconds for rendering all the patches, and the dotted lines represent the milliseconds how long all the LOD steps of the Patch Subdivision took.*



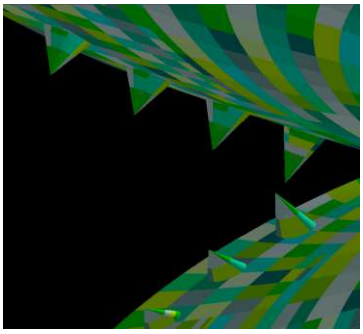
(b) *#patches out three-tier hybrid (122k)*
(c) *#patches out three-tier hybrid (257k)*
(d) *#patches out three-tier hybrid (821k)*
(e) *#patches out three-tier hybrid (1.27M)*
(f) *#patches out fixed 4xSS+8xMS (122k)*
(g) *#patches out fixed 4xSS+8xMS (257k)*
(h) *#patches out fixed 4xSS+8xMS (821k)*
(i) *#patches out fixed 4xSS+8xMS (1.27M)*

Figure 5.19: *These diagrams show performance results achieved with our three-tier hybrid rendering technique which dynamically schedules patches for rendering with Tess. noAA, Tess. 8xMS, or Tess 4xSS+8xMS rendering variants for a scene with 60k yarn curves as shown in Figures 5.14a and 5.14b. Figures 5.19b to 5.19e put the numbers of output patches after each LOD step in perspective to Figures 5.19f to 5.19i.*

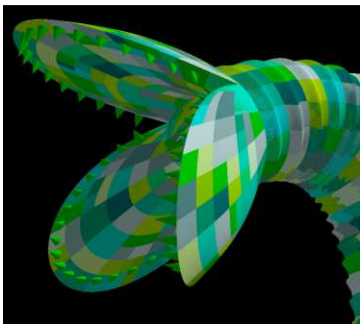
While this model does not satisfy the quality requirements of modern digital entertainment, it shall be noted that it can be rendered with arbitrary precision thanks to its parametric description. Figures 5.20b to 5.20d show that for different camera positions, our Patch Subdivision stage always generates patches so that close to pixel-perfect geometric detail is rendered. This aspect would be a perfect fit for a micro polygon scenario like Nanite [KSW21], where pixel-sized triangles are not uncommon but rather the norm. An additional benefit of using parametric models in such a scenario are the non-existent memory requirements for a parametric model's geometry. They can help to relieve some of the high demands on storage requirements and memory transfers of typical



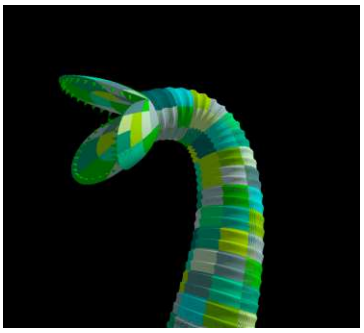
(a) *"Giant worm" modeled with parametric functions.*



(b) *Close-up view of the "giant worm" model, 1207 patches have been generated, rendering at 90 FPS with the point-based rendering variant.*



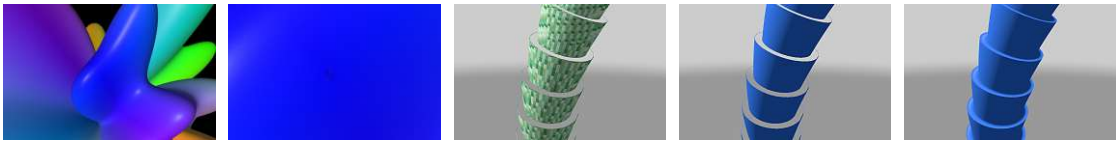
(c) *A view of the "giant worm" model where 949 patches have been generated, rendering at 109 FPS with the point-based rendering variant.*



(d) *Viewing the "giant worm" model from further away, 238 patches have been generated, rendering at 120 FPS with the point-based rendering variant.*

Figure 5.20: *This parametrically modeled "giant worm" shape consists of multiple parametrically defined and distorted geometric shapes. Figures 5.20b to 5.20d show the resulting patches after the* Patch Subdivision *stage, ensuring that the object is rendered with close to pixel-perfect geometric detail from all camera positions. The FPS have been measured on an RTX 3050 Laptop.*

ultra-high geometry scenarios, which often require loading or maintaining huge amounts of geometry—sometimes even swapping geometry from GPU memory to main memory in order to free space for other highly detailed meshes.



(a) *Artefacts at SH poles* **(b)** *Closeup of Figure 5.21a* **(c)** *Jump discontinuities* **(d)** *Rendered via point rendering* **(e)** *Rendered via triangles*

Figure 5.21: *Current limitations of our method*

5.8 Conclusion and Future Work

We have presented a general method for sampling and rendering parametric functions in real time. Although a general method, it outperforms recent solutions for rendering individual, high-detail SH glyphs and large glyph datasets in terms of rendering speed and quality. For large datasets, our noAA configuration achieves higher frame rates on a *mobile* AMD 680M GPU than the method by Peters et al. [Pet+23] on a dedicated *desktop* NVIDIA RTX 4090 GPU. Even at the higher-quality 4xSS and 8xMS configurations, our method running on the much weaker RTX 3050 Laptop is competitive with theirs on an RTX 4090. We found non-uniform patch subdivision to be a key factor for good performance, contrary to the recommendation of Eisenacher et al. [EML09] to always split patches 1:4. Additional experiments for organic shapes and fabric yield several hundred FPS while avoiding prominent artifacts. Our hybrid rendering approach is another tool that can help to keep frame rates stable by supersampling only those patches that have been found to show subpixel variations.

There are multiple avenues for future work: We intend to investigate better handling of holes in parametric objects. Our Patch Subdivision ignores the concept of holes and would currently subdivide patches strongly near the discontinuity as illustrated in Figure 5.21c. An early exit criterion could improve performance in such cases. Furthermore, the rendered output of a parametrically defined palm tree trunk shown in Figures 5.21d and 5.21e currently differs between Point-based and Tessellation-based since point rendering does not fill gaps, which are closed by the triangles produced by the tessellator. Besides the proposed screen distance-based metric for deciding whether to subdivide a patch, we hope to explore further metrics based, e.g., on the derivatives of $f_p(u, v)$. We also plan to investigate relevant performance factors for our Point-based variant, the speed of which we found to depend heavily on the test setup used. A more optimized implementation might be able to increase the performance of Point-based$_{\text{direct}}$ and Point-based$_{\text{local FB}}$ rendering variants. Assuming better performance would make the latter perfectly suitable for the supersampling variant in a hybrid rendering approach, allowing to avoid the expensive data transfers between different framebuffer formats. A new feature of modern graphics APIs called *Work Graphs* [PR23; Adv23] would allow the number of dispatch calls for the Patch Subdivision stage to be determined and scheduled within a frame, eliminating the latency of an adaptive approach like described in Section 5.5.1.

Since Patch Subdivision is fast enough to run every frame, our method is well suited for use with animated objects or time-varying data. Although flowing curtains consisting of hundreds of thousands of fiber curves have their appeal, we have only scratched the surface: we believe that our general method can unlock a wide range of elaborate, animated parametric functions, and enable glyph-based visualization of time-varying medical data with high frame rates, or smooth morphing between data sets in real time.

Finally, the development of parametric modeling tools, or investigation into export formats from existing modeling tools would be worthwhile. Constructing a model like a "giant worm" in code can get exhausting. However, it is still notable that building such a shape in code is entirely possible—and probably much easier than trying to construct a similar model as a triangle mesh in code. Nevertheless, we assess suitable tooling as a necessary stepping stone for creating more complex parametric objects in high fidelity.

The source code for this method is available on GitHub [1].

[1] https://github.com/cg-tuwien/FastRenderingOfParametricObjects

CHAPTER 6

Conclusion

This dissertation presents in-depth descriptions and analyses of techniques that help applications and games achieve better rendering performance in certain selected scenarios, which are relevant in the context of achieving fast rendering of ultra-detailed geometry on modern GPUs. While our work on fast rendering of multiple views provides rather general recommendations for notoriously performance-intensive VR and multi-view applications—which are arguably even more crucial with ultra-detailed geometry—our techniques for enabling fine-grained culling of animated models' clusters and rendering of parametrically-defined objects in pixel-level or even sub-pixel-level detail directly target ultra-detailed geometry scenarios, scenes, or parts of a scene.

Our extensive evaluations of MVR scenarios in Chapter 2 revealed that hardware-accelerated MVR is not always faster than alternative pipeline configurations. We describe two geometry shader-based variants that can outperform the *OVR_multiview*-based $\mathcal{P}(\lceil \frac{N}{4} \rceil, \approx, \square\square)$ pipeline variant: $\mathcal{P}(\lceil \frac{N}{4} \rceil, \vdash\!\!\equiv, \square\square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ shows better performance in some test cases on older GPUs and is sometimes affected less by higher geometry loads. $\mathcal{P}(1, \vdash\!\!\equiv, \square, \texttt{VFC}_{\texttt{GS}}|\texttt{BFC}_{\texttt{GS}})$ outperforms all other pipeline variants for scenarios with very high vertex shader load. Another notable finding of our evaluations was that view frusta need to overlap to some degree for a positive effect on performance. Otherwise, simple multi-pass rendering $\mathcal{P}(N, \rightarrow, \square\square)$ might still be faster.

The technique we have described in Chapter 4 constitutes a fundamental algorithm for preventing artifacts in rendering when using fine-grained culling of skinned models that have been divided into meshlets. Our technique is guaranteed to yield conservative bounds, which can be used to prevent premature VFC (at the borders of the screen) or premature BFC (with deformed clusters w.r.t. their initial positions). Our evaluations show that fine-grained culling results in almost linear savings in frame times in relation to the number of clusters culled, which is especially helpful in scenarios with ultra-high geometric detail.

Our general method for rendering parametrically-defined objects described in Chapter 5 enables rendering of parametric objects with almost pixel-perfect precision, achieving several hundred FPS for various parametric objects. Objects defined in such a manner fit well into an ultra-detailed geometry scenario, such as a typical Nanite scene. The advantages of this approach are revealed with the simple example of a sphere model: Instead of storing a sphere model of ultra-high geometric detail in memory and streaming it through the graphics pipeline each frame, it can be defined entirely in code using a parametric function. Its geometry can be created at a later point in the graphics pipeline, avoiding most of its high geometry load in early pipeline stages, while rendering it in arbitrary geometric precision. Our method enables the rendering of a variety of different shapes, and our evaluations revealed that our method outperforms a previous state-of-the-art method for rendering SH glyphs in terms of speed and quality.

In conclusion, we can draw three general observations from our research:

First, some old wisdom remains still true and is now at least as important as it always has been, and it is crucial to achieving good rendering performance with high geometry loads: In ultra-detailed geometry scenarios, the key to achieving high FPS is rendering not more than necessary by using culling (VFC and BFC) and the right amount of detail (by applying a LOD strategy). We use fine-grained culling in geometry shader instancing-based variants which performed well for the simultaneous rendering of multiple views [Unt+20]. Culling is also the key to reducing render times significantly for our technique for rendering ultra-detailed skinned meshes. Levels of detail further help to render not more than necessary [Unt+21]. This is especially evident in our work on fast rendering of parametrically-defined objects, where we create the geometry necessary for almost pixel-perfect detail on the fly—but not more detailed than that thanks to precise evaluation in each frame in our Patch Subdivision stage [Unt+24].

Second, a more recently applicable rule of thumb that can be stated is: It can be beneficial to trade increased compute load for reduced memory transfers. This is especially true on the latest GPU models, as shown in Figure 1.3. We follow this approach by computing culling decisions in geometry shaders for simultaneous rendering of multiple views [Unt+20], by computing culling decisions in task shaders to cull meshlets [Unt+21], and most notably by evaluating parametric functions in compute, tessellation, and fragment shaders during our method on fast rendering of parametric objects [Unt+24].

Lastly, modern GPUs offer a multitude of classical pipeline stages and features, as well as a multitude of new pipeline stages and features. Maintaining an overview of them all can be overwhelming but also worthwhile for achieving good performance in certain application scenarios. We have tested hardware-accelerated MVR by means of the *OVR_multiview* extension and found it to perform very well in many scenarios, but not in all. In some, it was outperformed by classical geometry shader-based pipeline configurations, leading to different caching and scheduling behavior on modern GPUs [Unt+20]. We have used new task and mesh shader stages for fine-grained culling and rendering of skinned meshes, which allowed for efficient scheduling of mesh shaders and data passing between task and mesh shader stages, while still forwarding non-culled meshlets directly to the

rasterizer [Unt+21]. While we attempted to use task and mesh shaders also for our method for fast rendering of parametric objects, their fixed two-step nature turned out to be too inflexible for our Patch Subdivision stage. Therefore, we resorted to multiple compute shader invocations for this stage. While point rendering seemed to be beneficial for rendering almost pixel-perfect detail, modern GPUs' tessellation units led to much better performance in almost all cases that we have tested. We found tessellation units to perform their task very efficiently, which made them the variant of choice for fast rendering of parametric objects in most cases [Unt+24].

With the contributions of this dissertation to the state of the art in rendering ultra-detailed geometry in real time, we look forward to the techniques that other researchers will develop in the future and hope that our research proves to be useful to them. There are multiple opportunities for follow-up work.

Future Work

Each of the techniques presented has its own individual scope for possible follow-up research, but especially combinations of the techniques presented in Chapters 2, 4 and 5 might provide some interesting research opportunities.

For multi-view rendering, evaluations should be performed with an implementation that uses a low-level graphics API—such as Vulkan, as reasoned in Chapter 3—and are performed on modern GPUs. Two newer NVIDIA GPU generations have been released since we conducted our research presented in Chapter 2. The widely supported *VK_KHR_multiview* extension should be included in the tests when using Vulkan. It can be expected—but should not be taken for granted—that this extension outperforms other graphics pipeline configurations.

Natural follow-up work to our research on conservative meshlet bounds, as described in Chapter 4 for animated meshes, would be to extend our algorithm to further skinning methods. Meaningful performance results could be gathered by integrating and testing our algorithm in a Nanite-style micro-poly scene setup, including seamless transitions between LODs on a per-cluster basis and cluster streaming. It would allow gathering more insights into the performance savings achievable with fine-grained culling for clusters of animated meshlets in such setups.

Follow-up work on our method on fast rendering of parametric functions, as described in Chapter 5, should investigate the performance characteristics of our point-based rendering variant since it seems to show subpar performance. Karis [KSW21] reported better performance of the software rasterizer over the hardware rasterizer when rendering small triangles. Such behavior was not observable in our evaluations shown in Section 5.6 and should, therefore, be subject to further investigation.

We assume potential for relative performance gains for both, our techniques on meshlet bounds for animated meshes and fast rendering of parametric objects, in combination with multi-view rendering: Both techniques perform view frustum culling. From our

evaluations in Section 2.5 we know that many GPUs require view frusta to overlap for achieving performance improvements when simultaneously rendering multiple views over just rendering each view sequentially. For overlapping views, the culling steps could be merged and not be run separately for each view. Meshlet bounds could be used for view frustum culling against an all-encompassing multi-view frustum in the task shader before scheduling the rendering of multiple views in the mesh shader. The latter can be achieved via the *GLSL_EXT_mesh_shader* extension, which interacts with *VK_KHR_multiview* [KB22]. Applying the same concept to the rendering of parametric objects bears the potential for substantial performance savings since the Patch Subdivision stage—which can take up to 50% of the frame time for some scenes, as described in Section 5.5.1—only has to run once for all the views.

One possible way to combine our technique on fast rendering of parametric objects with rendering meshlets of static or animated models would be to add geometric surface detail onto objects using parametric functions. Since parametric functions do not require any GPU memory but generate geometric detail on the fly, they constitute a powerful tool to add geometric detail—especially useful in ultra-detailed geometry setups.

For the same reason—requiring no additional GPU memory—we believe that it could be worthwhile in general to express as many objects as possible with parametric functions in a game or application that strives to achieve almost pixel-perfect geometric precision in its rendered output. Items or power-ups are prime examples of objects that could be modeled in code with parametric functions. For expressing more complex objects as parametric functions, specialized modeling tools might be required. Trying to learn a parametric expression for given 3D models could constitute an interesting path of research within the field of machine learning.

We can't wait to marvel at the visual fidelity of future games and applications rendering ultra-high geometric detail and hope that the techniques presented in this dissertation can play a part in helping them reach their rendering performance goals.

Overview of Generative AI Tools Used

None.

List of Figures

List of Tables

Bibliography

[AA00] Steven Abrams and Peter K. Allen. "Computing swept volumes". In: *The Journal of Visualization and Computer Animation* 11.2 (2000), pp. 69–82.

[Abb+15] Amin Abbasloo et al. "Visualizing tensor normal distributions at multiple levels of detail". In: *IEEE transactions on visualization and computer graphics* 22.1 (2015), pp. 975–984.

[Ade+91] Stephen J Adelson et al. "Simultaneous generation of stereoscopic views". In: *Computer Graphics Forum*. Vol. 10. 1. Wiley Online Library. 1991, pp. 3–10.

[Adv23] Advanced Micro Devices, Inc. *Announcing GPU Work Graphs in Vulkan*. https://gpuopen.com/gpu-work-graphs-in-vulkan. [Accessed 08-April-2024]. 2023.

[AHA15] Magnus Andersson, Jon Hasselgren, and Tomas Akenine-Möller. "Masked Depth Culling for Graphics Hardware". In: *ACM Trans. Graph.* 34.6 (Oct. 2015). ISSN: 0730-0301. DOI: 10.1145/2816795.2818138.

[AMD18] AMD. *V-EZ*. https://github.com/GPUOpen-LibrariesAndSDKs/V-EZ. [Accessed 22-Oct-2022]. 2018.

[Apl22] Apple Inc. *Metal. Accelerating graphics and much more*. https://developer.apple.com/metal. [Accessed 21-Mar-2022]. 2022.

[Bla20] Alexander Blake-Davies. *Next-Generation Gaming with AMD RDNA 2 and DirectX 12 Ultimate*. https://community.amd.com/t5/blogs/next-generation-gaming-with-amd-rdna-2-and-directx-12-ultimate/ba-p/427032. [Accessed 12-April-2021]. 2020.

[BP07] Ilya Baran and Jovan Popović. "Automatic Rigging and Animation of 3D Characters". In: *ACM Trans. Graph.* 26.3 (July 2007), 72–es. ISSN: 0730-0301. DOI: 10.1145/1276377.1276467.

[BP23] Carsten Benthin and Christoph Peters. "Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail". In: *Computer Graphics Forum*. Vol. 42. 8. Wiley Online Library. 2023, e14868.

[Bra+16] Wade Brainerd et al. "Efficient GPU rendering of subdivision surfaces using adaptive quadtrees". In: *ACM Transactions on Graphics (TOG)* 35.4 (2016), pp. 1–12.

[Bre22] The Brenwill Workshop Ltd. *KhronosGroup/MoltenVK*. https://github.com/KhronosGroup/MoltenVK. [Acc. 21-Mar-2022]. 2022.

[BS18] Swaroop Bhonde and Mahalakshmi Shanmugam. *Turing Multi-View Rendering in VRWorks*. https://devblogs.nvidia.com/turing-multi-view-rendering-vrworks. [Accessed 19-February-2020]. 2018.

[BSF10] Stephan Beck, Mathias Schneider, and Bernd Fröhlich. "Multiple View Generation for Auto-stereoscopic Displays". In: *Proceedings of the 7th Workshop of Bauhaus-Universität Weimar on Virtuelle und Erweiterte Realität der GI-Fachgruppe VR/AR"*. Weimar, Germany, 2010, pp. 21–23.

[BW03] Jiří Bittner and Peter Wonka. "Visibility in computer graphics". In: *Environment and Planning B: Planning and Design* 30.5 (2003), pp. 729–755.

[BWF17] Dennis Giovani Balreira, Marcelo Walter, and Dieter W Fellner. "What we are teaching in Introduction to Computer Graphics." In: *Eurographics (Education Papers)*. 2017, pp. 1–7.

[Cas18] Cass Everitt. *Oculus Virtual Reality Multi-View Extension*. https://www.khronos.org/registry/OpenGL/extensions/OVR/OVR_multiview.txt. [Accessed 6-March-2020]. 2018.

[Cig+05] P. Cignoni et al. "Batched multi triangulation". In: *VIS 05. IEEE Visualization, 2005.* 2005, pp. 207–214. DOI: 10.1109/VISUAL.2005.1532797.

[CM04] Frederic Cordier and Nadia Magnenat-Thalmann. "A Data-Driven Approach for Real-Time Clothes Simulation". In: *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*. PG '04. USA: IEEE Computer Society, 2004, pp. 257–266. ISBN: 0769522343.

[Coh+03] Daniel Cohen-Or et al. "A survey of visibility for walkthrough applications". In: *IEEE Transactions on Visualization and Computer Graphics* 9.3 (2003), pp. 412–431.

[Cra23] Keenan Crane. *A Simple Parametric Model of Plain-Knit Yarns*. https://github.com/keenancrane/plain-knit-yarn. [Accessed 29-February-2024]. Mar. 2023.

[Dim07] Rouslan Dimitrov. "Cascaded shadow maps". In: *Developer Documentation, NVIDIA Corporation* (2007).

[DNS10] Frnnçois De Sorbier, Vincent Nozick, and Hideo Saito. "GPU-based multi-view rendering". English. In: *CGAT 2010 - Computer Games, Multimedia and Allied Technology, Proceedings*. 2010, pp. 273–279. ISBN: 9789810854799.

[EML09] Christian Eisenacher, Quirin Meyer, and Charles Loop. "Real-time view-dependent rendering of parametric surfaces". In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 137–143.

[Epi24a] Epic Games, Inc. *Nanite Virtualized Geometry in Unreal Engine | Unreal Engine 5.5 Documentation | Epic Developer Community*. https://dev.epicgames.com/documentation/en-us/unreal-engine/nanite-virtualized-geometry-in-unreal-engine?application_version=5.5. [Accessed 09-Oct-2024]. 2024.

[Epi24b] Epic Games, Inc. *Valley of the Ancient in UE Game Samples - UE Marketplace*. https://www.unrealengine.com/marketplace/en-US/product/ancient-game-01. [Accessed 25-Sep-2024]. 2004–2024.

[Eve16] Cass Everitt. "Multiview Rendering". In: *ACM SIGGRAPH 2016, Moving Mobile Graphics - SIGGRAPH 2016 Course*. 2016.

[FGB20] VF Frolov, VA Galaktionov, and BH Barladyan. "COMPARATIVE STUDY OF HIGH PERFORMANCE SOFTWARE RASTERIZATION TECHNIQUES." In: *Mathematica Montisnigri* 47 (2020).

[FS93] Thomas A Funkhouser and Carlo H Séquin. "Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments". In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 1993, pp. 247–254.

[GRS95] Eduard Gröller, René T Rau, and Wolfgang Straßer. "Modeling and visualization of knitwear". In: *IEEE Transactions on Visualization and Computer Graphics* 1.4 (1995), pp. 302–310.

[Gün+06] Johannes Günther et al. "Interactive ray tracing of skinned animations". In: *The Visual Computer* 22.9 (Sept. 2006), pp. 785–792. ISSN: 1432-2315. DOI: 10.1007/s00371-006-0063-x.

[GW07] Markus Giegl and Michael Wimmer. "Unpopping: Solving the image-space blend problem for smooth discrete LOD transitions". In: *Computer Graphics Forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 46–49.

[HA06] Jon Hasselgren and Tomas Akenine-Möller. "An efficient multi-view rasterization architecture". In: *Proceedings of the 17th Eurographics conference on Rendering Techniques*. Eurographics Association. 2006, pp. 61–72.

[HA15] Ulrich Haar and Sebastian Aaltonen. "GPU-Driven Rendering Pipelines". Siggraph 2015: Advances in Real-Time Rendering in Games. 2015.

[Hal98] Michael Halle. "Multiple viewpoint rendering". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. 1998, pp. 243–254.

[Has+22] SeyyedKazem HashemizadehKolowri et al. "Jointly estimating parametric maps of multiple diffusion models from undersampled q-space data: A comparison of three deep learning approaches". In: *Magnetic Resonance in Medicine* 87.6 (2022), pp. 2957–2971. DOI: https://doi.org/10.1002/mrm.29162. eprint: https://onlinelibrary.wiley.com/

doi/pdf/10.1002/mrm.29162. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.29162.

[HC11] Stephen Hill and Daniel Collin. *Practical, Dynamic Visibility for Games*. https://blog.selfshadow.com/publications/practical-visibility/. 2011.

[Hec18] Tobias Hector. *Comparing the Vulkan SPIR-V memory model to C++'s*. https://www.khronos.org/blog/comparing-the-vulkan-spir-v-memory-model-to-cs. [Accessed 21-Mar-2022]. 2018.

[Hop99] Hugues Hoppe. "Optimization of Mesh Locality for Transparent Vertex Caching". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. 1999, pp. 269–276. ISBN: 0201485605. DOI: 10.1145/311535.311565.

[HS17] Songfang Han and Pedro Sander. "Triangle Reordering for Efficient Rendering in Complex Scenes". In: *Journal of Computer Graphics Techniques (JCGT)* 6.3 (Sept. 2017), pp. 38–52. ISSN: 2331-7418.

[HSS19a] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. "Tessellated Shading Streaming". In: *Computer Graphics Forum*. Vol. 38. 4. Wiley Online Library. 2019, pp. 171–182.

[HSS19b] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. "The Camera Offset Space: Real-Time Potentially Visible Set Computations for Streaming Rendering". In: *ACM Trans. Graph.* 38.6 (Nov. 2019). ISSN: 0730-0301. DOI: 10.1145/3355089.3356530. URL: https://doi.org/10.1145/3355089.3356530.

[JFB23] Mark Bo Jensen, Jeppe Revall Frisvad, and J Andreas Bærentzen. "Performance Comparison of Meshlet Generation Strategies". In: *Journal of Computer Graphics Techniques* 12.2 (2023).

[JP04] Doug L. James and Dinesh K. Pai. "BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models". In: *ACM Transactions on Graphics (SIGGRAPH 2004)* 23.3 (Aug. 2004).

[Kap21] Arseny Kapoulkine. *meshoptimizer, Mesh optimization library*. https://github.com/zeux/meshoptimizer. [Accessed: 13-April-2021]. 2016-2021.

[Kav+07] Ladislav Kavan et al. "Skinning with Dual Quaternions". In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. Seattle, Washington: Association for Computing Machinery, 2007, pp. 39–46. ISBN: 9781595936288. DOI: 10.1145/1230100.1230107.

[KB19] Christoph Kubisch and Pat Brown. *GLSL Mesh Shader Extension*. https://www.khronos.org/registry/OpenGL/extensions/NV/NV_mesh_shader.txt. [Accessed 2-June-2021]. 2019.

[KB22] Christoph Kubisch and Pat Brown. *GLSL Mesh Shader Extension*. https://github.com/KhronosGroup/GLSL/blob/main/extensions/ext/GLSL_EXT_mesh_shader.txt. [Accessed 13-Oct-2024]. 2022.

[KDR18] Jad-Nicolas Khoury, Jonathan Dupuy, and Christophe Riccio. "Adaptive GPU Tessellation with Compute Shaders". In: 2018. URL: https://api.semanticscholar.org/CorpusID:198116975.

[Ken+18] Michael Kenzel et al. "A High-Performance Software Graphics Pipeline Architecture for the GPU". In: *ACM Trans. Graph.* 37.4 (Nov. 2018). DOI: 10.1145/3197517.3201374.

[Ker+18] Bernhard Kerbl et al. "Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the Modern GPU". In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.2 (Aug. 2018). DOI: 10.1145/3233302.

[Ker+22] Bernhard Kerbl et al. "An Improved Triangle Encoding Scheme for Cached Tessellation". In: *Eurographics 2022 - Short Papers*. Ed. by Nuria Pelechano and David Vanderhaeghe. The Eurographics Association, 2022. ISBN: 978-3-03868-169-4. DOI: 10.2312/egs.20221031.

[Khr18ray] The Khronos® Group Inc. *VK_NV_ray_tracing(3) Manual Page*. https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_NV_ray_tracing.html. [Accessed 21-Mar-2022]. 2018.

[Khr20ray] The Khronos® Group Inc. *VK_KHR_ray_tracing_pipeline(3) Manual Page*. https://www.khronos.org/registry/vulkan/specs/1.3-extensions/man/html/VK_KHR_ray_tracing_pipeline.html. [Accessed 21-Mar-2022]. 2020.

[Khr22gl] The Khronos® Group Inc. *OpenGL - The Industry's Foundation for High Performance Graphics*. https://www.opengl.org. [Accessed 21-Jan-2022]. 2022.

[Khr22gsl] The Khronos® Group Inc. *KhronosGroup/glslang: Khronos-reference front end for GLSL/ESSL, partial front end for HLSL, and a SPIR-V generator*. https://github.com/KhronosGroup/glslang. [Accessed 24-Jan-2022]. 2022.

[Khr22h] The Khronos Group Inc. *History of OpenGL — OpenGL Wiki*. [Online; accessed 1-October-2024]. 2022. URL: https://www.khronos.org/opengl/wiki/History_of_OpenGL.

[Khr22me] The Khronos® Group Inc. *Khronos Members - The Khronos Group Inc.* https://www.khronos.org/members/list. [Accessed 21-Mar-2022]. 2022.

[Khr22spv] The Khronos® Group Inc. *SPIR Overview - The Khronos Group Inc.* https://www.khronos.org/spir. [Accessed 24-Jan-2022]. 2022.

[Khr22vk] The Khronos® Group Inc. *Vulkan | Cross platform 3D Graphics*. https://www.vulkan.org. [Accessed 21-Jan-2022]. 2022.

[Khr22vsa] The Khronos® Group Inc. *KhronosGroup/Vulkan-Samples: One stop solution for all Vulkan samples*. https://github.com/KhronosGroup/Vulkan-Samples. [Accessed 24-Jan-2022]. 2022.

[Khr23tesa] Khronos Group. *gl_TessCoords - OpenGL 4 Reference Pages*. https://registry.khronos.org/OpenGL-Refpages/gl4/html/gl_TessCoord.xhtml. [Accessed 08-March-2024]. 2011-2014.

[Khr23tesb] The Khronos Group Inc. *Tessellation :: Vulkan Documentation Project*. https://docs.vulkan.org/spec/latest/chapters/tessellation.html. [Accessed 03-March-2024]. 2022-2023.

[Khr24ras] The Khronos Group Inc. *Rasterization :: Vulkan Documentation Project*. https://docs.vulkan.org/spec/latest/chapters/primsrast.html. [Accessed 12-September-2024]. 2022-2024.

[Khr24sub] The Khronos Group Inc. *subgroups :: Vulkan Documentation Project*. https://docs.vulkan.org/guide/latest/subgroups.html. [Accessed 01-March-2024]. 2022-2023.

[Khr24vsp] The Khronos Group Inc. *Vulkan® 1.3.279 - A Specification (with all registered Vulkan extensions)*. https://registry.khronos.org/vulkan/specs/1.3-extensions/html. [Accessed 11-July-2024]. 2024.

[Kim+03] Young Kim et al. "Fast swept volume approximation of complex polyhedral models". In: Jan. 2003, pp. 11–22. DOI: 10.1145/781611.781613.

[KL22] Sungjin Kim and Chang Ha Lee. "Mesh clustering and reordering based on normal locality for efficient rendering". In: *Symmetry* 14.3 (2022), p. 466.

[Koc20a] Daniel Koch. *Vulkan Acceleration Structure Device Extension*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_acceleration_structure.html. [Accessed 2-August-2021]. 2020.

[Koc20b] Daniel Koch. *Vulkan Ray Query Device Extension*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_query.html. [Accessed 2-August-2021]. 2020.

[Koc21] Daniel Koch. *GLSL Ray Query Extension*. https://github.com/KhronosGroup/GLSL/blob/master/extensions/ext/GLSL_EXT_ray_query.txt. [Accessed 2-August-2021]. 2021.

[KSW21] Brian Karis, Rune Stubbe, and Graham Wihlidal. "A Deep Dive into Nanite Virtualized Geometry". In: *ACM SIGGRAPH 2021 Courses, Advances in Real-Time Rendering in Games, Part 1*. https://advances.realtimerendering.com/s2021/index.html [Accessed 10-September-2021]. 2021.

[Kub18a] Christoph Kubisch. *Introduction to Turing Mesh Shaders*. https://developer.nvidia.com/blog/introduction-turing-mesh-shaders. [Accessed 12-April-2021]. 2018.

[Kub18b] Christoph Kubisch. *Vulkan Mesh Shader Device Extension*. https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_NV_mesh_shader.html. [Accessed 2-June-2021]. 2018.

[Kut+23] Bastian Kuth et al. "Edge-Friend: Fast and Deterministic Catmull-Clark Subdivision Surfaces". In: *Computer Graphics Forum* 42.8 (2023), e14863. DOI: https://doi.org/10.1111/cgf.14863. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14863. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14863.

[KŽ05a] Ladislav Kavan and Jiří Žára. "Fast Collision Detection for Skeletally Deformable Models". In: *Computer Graphics Forum* 24.3 (2005), pp. 363–372.

[KŽ05b] Ladislav Kavan and Jiří Žára. "Spherical Blend Skinning: A Real-Time Deformation of Articulated Models". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. Washington, District of Columbia: Association for Computing Machinery, 2005, pp. 9–16. ISBN: 1595930132. DOI: 10.1145/1053427.1053429.

[LD09] Haik Lorenz and Jürgen Döllner. "Real-time Piecewise Perspective Projections." In: *GRAPP*. 2009, pp. 147–155.

[LH16] Binh Huy Le and Jessica K. Hodgins. "Real-Time Skeletal Skinning with Optimized Centers of Rotation". In: *ACM Trans. Graph.* 35.4 (July 2016). ISSN: 0730-0301. DOI: 10.1145/2897824.2925959.

[LK11] Samuli Laine and Tero Karras. "High-performance software rasterization on GPUs". In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. 2011, pp. 79–88.

[Lue+02] David Luebke et al. *Level of detail for 3D graphics*. Elsevier, 2002.

[LY06] Gang Lin and Thomas P. -Y. Yu. "An Improved Vertex Caching Scheme for 3D Mesh Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 12.4 (July 2006), pp. 640–648. ISSN: 1077-2626. DOI: 10.1109/TVCG.2006.59.

[Mar09] Jonathan Marbach. "GPU Acceleration of Stereoscopic and Multi-View Rendering for Virtual Reality Applications". In: *Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*. VRST '09. Kyoto, Japan: Association for Computing Machinery, 2009, pp. 103–110. ISBN: 9781605588698. DOI: 10.1145/1643928.1643953. URL: https://doi.org/10.1145/1643928.1643953.

[Mic21] Microsoft Corporation. *DirectX-Specs*. https://microsoft.github.io/DirectX-Specs. [Accessed 02-August-2021]. 2021.

[Mic22] Microsoft Corporation. *DirectX graphics and gaming*. https://docs.microsoft.com/en-us/windows/win32/directx. [Accessed 21-Mar-2022]. 2022.

[MLT89] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. "Joint-Dependent Local Deformations for Hand Animation and Object Grasping". In: *Proceedings on Graphics Interface '88*. Edmonton, Alberta, Canada: Canadian Information Processing Society, 1989, pp. 26–33.

[Mue+18] Joerg H Mueller et al. "Shading atlas streaming". In: *ACM Transactions on Graphics (TOG)* 37.6 (2018), pp. 1–16.

[MWH18] Adam Marrs, Benjamin Watson, and Christopher Healey. "View-warped Multi-view Soft Shadows for Local Area Lights". In: *Journal of Computer Graphics Techniques (JCGT)* 7.3 (July 2018), pp. 1–28. ISSN: 2331-7418.

[Nie+16] M. Nießner et al. "Real-Time Rendering Techniques with Hardware Tessellation". In: *Computer Graphics Forum* 35.1 (2016), pp. 113–137. DOI: https://doi.org/10.1111/cgf.12714. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.12714. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12714.

[NL13] M. Nießner and C. Loop. "Analytic Displacement Mapping using Hardware Tessellation". In: *ACM Transactions on Graphics (TOG)* 32.3 (2013), p. 26.

[NVI16] NVIDIA Corporation. "NVIDIA GeForce GTX 1080". In: (2016).

[NVI18a] NVIDIA Corporation. *NVIDIA Turing GPU Architecture*. https://images.nvidia.com/aem-dam/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. [Accessed 12-April-2021]. 2018.

[NVI18b] NVIDIA Corporation. *VRWorks - Multi-View Rendering (MVR)*. https://developer.nvidia.com/vrworks/graphics/multiview. [Accessed 19-February-2020]. 2018.

[NVI20] NVIDIA Corporation. *Nsight Graphics User Guide*. https://docs.nvidia.com/nsight-graphics/UserGuide. [Accessed 6-March-2020]. 2020.

[Ove22] Alexander Overvoorde. *Vulkan Tutorial*. https://vulkan-tutorial.com. [Accessed 22-January-2022]. 2022.

[PDG21] Charles Poirier, Maxime Descoteaux, and Guillaume Gilet. "Accelerating Geometry-Based Spherical Harmonics Glyphs Rendering for dMRI Using Modern OpenGL". In: *Computational Diffusion MRI*. Ed. by Suheyla Cetin-Karayumak et al. Cham: Springer International Publishing, 2021, pp. 144–155. ISBN: 978-3-030-87615-9.

[PEO09] Anjul Patney, Mohamed Ebeida, and John Owens. “Parallel view-dependent tessellation of Catmull–Clark subdivision surfaces”. In: Aug. 2009, pp. 99–108. DOI: 10.1145/1572769.1572785.

[Pet+23] Christoph Peters et al. “Ray Tracing Spherical Harmonics Glyphs”. In: *Vision, Modeling, and Visualization*. Ed. by Michael Guthe and Thorsten Grosch. The Eurographics Association, 2023. ISBN: 978-3-03868-232-5. DOI: 10.2312/vmv.20231223.

[Pet18] Jordan B Peterson. *12 rules for life: An antidote to chaos*. Random House Canada, 2018.

[Pon09] Federico Ponchio. *Multiresolution structures for interactive visualization of very large 3D datasets*. Univ.-Bibliothek, 2009.

[PR23] Amar Patel and Tex Riddell. *D3D12 Work Graphs Preview*. https://devblogs.microsoft.com/directx/d3d12-work-graphs-preview. [Accessed 08-April-2024]. 2023.

[Rag+11] Jonathan Ragan-Kelley et al. “Decoupled sampling for graphics pipelines”. In: *ACM Transactions on Graphics (TOG)* 30.3 (2011), pp. 1–17.

[Res22] Research Unit of Computer Graphics | TU Wien. *Vulkan Launchpad*. https://github.com/cg-tuwien/VulkanLaunchpad. 2022.

[Res24a] Research Unit of Computer Graphics | TU Wien. *Auto-Vk*. https://github.com/cg-tuwien/Auto-Vk. 2024.

[Res24b] Research Unit of Computer Graphics | TU Wien. *Auto-Vk-Toolkit*. https://github.com/cg-tuwien/Auto-Vk-Toolkit. 2024.

[RK15] Kjetil Raaen and Ivar Kjellmo. “Measuring latency in virtual reality systems”. In: *Entertainment Computing-ICEC 2015: 14th International Conference, ICEC 2015, Trondheim, Norway, September 29-Ocotober 2, 2015, Proceedings 14*. Springer. 2015, pp. 457–462.

[RLM04] S. Redon, M. C. Lin, and D. Manocha. “Fast Continuous Collision Detection for Articulated Models”. In: *Solid Modeling*. Ed. by Gershon Elber, Nicholas Patrikalakis, and Pere Brunet. The Eurographics Association, 2004. ISBN: 3-905673-55-X. DOI: 10.2312/sm.20041385.

[RME14] Guido Reina, Thomas Müller, and Thomas Ertl. “Incorporating Modern OpenGL into Computer Graphics Education”. In: *IEEE computer graphics and applications* 34.4 (2014), pp. 16–21.

[Rod40] Olinde Rodrigues. “Des lois géométriques qui régissent les déplacements d’un système solide dans l’espace, et de la variation des coordonnées provenant de ces déplacements considérés indépendants des causes qui peuvent les produire”. In: *Journal de Mathématiques Pures et Appliquées*. 1st ser. 5 (1840), pp. 380–440.

[RP94] Matthew Regan and Ronald Pose. “Priority rendering with a virtual reality address recalculation pipeline”. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, pp. 155–162.

[Rum23] Wolfgang Rumpler. “Real-Time Distortion Correction Methods for Curved Monitors”. MA thesis. Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria: Research Unit of Computer Graphics, Institute of Visual Computing and Human-Centered Technology, Faculty of Informatics, TU Wien, Jan. 2023, p. 102. URL: https://www.cg.tuwien.ac.at/research/publications/2023/rumpler-2023-dcmcm/.

[SBT06] J. Spillmann, M. Becker, and M. Teschner. “Efficient Updates of Bounding Sphere Hierarchies for Geometrically Deformable Models”. In: *Vriphys: 3rd Workshop in Virtual Realitiy, Interactions, and Physical Simulation*. Ed. by Cesar Mendoza and Isabel Navazo. The Eurographics Association, 2006. ISBN: 3-905673-61-4. DOI: 10.2312/PE/vriphys/vriphys06/053-060.

[SGO09] Sara C. Schvartzman, Jorge Gascón, and Miguel A. Otaduy. “Bounded Normal Trees for Reduced Deformations of Triangulated Surfaces”. In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. SCA ’09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 75–82. ISBN: 9781605586106. DOI: 10.1145/1599470.1599480.

[Sho+08] Jeremy Shopf et al. “March of the Froblins: Simulation and Rendering Massive Crowds of Intelligent and Detailed Creatures on GPU”. In: *ACM SIGGRAPH 2008 Games*. SIGGRAPH ’08. Los Angeles, California: Association for Computing Machinery, 2008, pp. 52–101. ISBN: 9781450378499. DOI: 10.1145/1404435.1404439.

[Sho85] Ken Shoemake. “Animating rotation with quaternion curves”. In: *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*. 1985, pp. 245–254.

[Shr+13] Dave Shreiner et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. 8th. Addison-Wesley Professional, 2013. ISBN: 0321773039.

[Sit+08] Pitchaya Sitthi-amorn et al. “An improved shading cache for modern GPUs”. In: *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*. 2008, pp. 95–101.

[SKW21] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. “Rendering Point Clouds with Compute Shaders and Vertex Order Optimization”. In: *Computer Graphics Forum* 40.4 (2021), pp. 115–126. ISSN: 1467-8659. URL: https://www.cg.tuwien.ac.at/research/publications/2021/SCHUETZ-2021-PCC/.

[SKW22] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. "Software Rasterization of 2 Billion Points in Real Time". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5.3 (July 2022), pp. 1–17.

[SOG08] Denis Steinemann, M. Otaduy, and M. Gross. "Tight and efficient surface bounds in meshless animation". In: *Comput. Graph.* 32 (2008), pp. 235–245.

[SS09] Michael Schwarz and Marc Stamminger. "Fast GPU-based Adaptive Tessellation with CUDA". In: *Computer Graphics Forum* 28.2 (Proceedings of Eurographics 2009) (Mar. 2009), pp. 365–374.

[SS96] Gernot Schaufler and Wolfgang Stürzlinger. "A three dimensional image cache for virtual reality". In: *Computer Graphics Forum*. Vol. 15. 3. Wiley Online Library. 1996, pp. 227–235.

[SW08] Daniel Scherzer and Michael Wimmer. "Frame sequential interpolation for discrete level-of-detail rendering". In: *Computer Graphics Forum*. Vol. 27. 4. Wiley Online Library. 2008, pp. 1175–1181.

[SWH15] Graham Sellers, Richard S. Wright, and Nicholas Haemel. *OpenGL Superbible: Comprehensive Tutorial and Reference*. 7th. Addison-Wesley Professional, 2015. ISBN: 0672337479.

[Tuc+02] David S. Tuch et al. "High angular resolution diffusion imaging reveals intravoxel white matter fiber heterogeneity". In: *Magnetic Resonance in Medicine* 48.4 (2002), pp. 577–582. DOI: https://doi.org/10.1002/mrm.10268. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/mrm.10268. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/mrm.10268.

[UKW22] Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer. "The Road to Vulkan: Teaching Modern Low-Level APIs in Introductory Graphics Courses". In: *Eurographics 2022 - Education Papers*. Reims: The Eurographics Association, Apr. 2022, pp. 31–39. ISBN: 978-3-03868-170-0. DOI: 10.2312/eged.20221043. URL: https://www.cg.tuwien.ac.at/research/publications/2022/unterguggenberger-2022-vulkan/.

[UKW23] Johannes Unterguggenberger, Bernhard Kerbl, and Michael Wimmer. "Vulkan all the way: Transitioning to a modern low-level graphics API in academia". In: *Computers and Graphics* 111 (Apr. 2023), pp. 155–165. ISSN: 1873-7684. DOI: 10.1016/j.cag.2023.02.001. URL: https://www.cg.tuwien.ac.at/research/publications/2023/unterguggenberger-2023-vaw/.

[Uni21] Unity Technologies. "Unity Documentation". https://docs.unity3d.com/ScriptReference/AnimationClip-localBounds.html. Animation Clip Local Bounds. 2021.

[Unt+20] Johannes Unterguggenberger et al. “Fast Multi-View Rendering for Real-Time Applications”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Steffen Frey, Jian Huang, and Filip Sadlo. Eurographics. online, May 2020, pp. 13–23. ISBN: 978-3-03868-107-6. DOI: 10.2312/pgv.20201071. URL: https://www.cg.tuwien.ac.at/research/publications/2020/unterguggenberger-2020-fmvr/.

[Unt+21] Johannes Unterguggenberger et al. “Conservative Meshlet Bounds for Robust Culling of Skinned Meshes”. In: *Computer Graphics Forum* 40.7 (Oct. 2021), pp. 57–69. ISSN: 1467-8659. DOI: 10.1111/cgf.14401. URL: https://www.cg.tuwien.ac.at/research/publications/2021/unterguggenberger-2021-msh/.

[Unt+24] Johannes Unterguggenberger et al. “Fast Rendering of Parametric Objects on Modern GPUs”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Guido Reina and Silvio Rizzi. The Eurographics Association, 2024. ISBN: 978-3-03868-243-1. DOI: 10.2312/pgv.20241129.

[Ura19] Yury Uralski. *Mesh Shading: Towards Greater Efficiency of Geometry Processing, Advances in Real-time Rendering*. Siggraph Course. 2019.

[Val03] Valve Corporation. *Steam Store*. https://store.steampowered.com. [Accessed 26-February-2020]. 2003.

[VkHpp22] The Khronos® Group Inc. *KhronosGroup/Vulkan-Hpp, Open-Source Vulkan C++ API*. https://github.com/KhronosGroup/Vulkan-Hpp. [Accessed 18-Sep-2022]. 2022.

[Vla15] Alex Vlachos. “Advanced VR Rendering”. In: *Game Developers Conference*. Vol. 1. 2015.

[Vla16] Alex Vlachos. “Advanced VR rendering performance”. In: *Game Developers Conference*. Vol. 2016. 2016.

[Vri22] Joey de Vries. *LearnOpenGL - Hello Triangle*. https://learnopengl.com/Getting-started/Hello-Triangle. [Accessed 22-Jan-2022]. 2022.

[WA23] Markus Worchel and Marc Alexa. “Differentiable Rendering of Parametric Geometry”. In: *ACM Transactions on Graphics (TOG)* 42.6 (2023), pp. 1–18.

[Wal21] Chuck Walbourn. *DirectXMesh geometry processing library*. https://github.com/microsoft/DirectXMesh. [Accessed: 13-April-2021"]. 2014-2021.

[Wih16] Graham Wihlidal. “Optimizing the Graphics Pipeline with Compute”. Game Developers Conference. 2016.

[Wil15] Timothy Wilson. “High performance stereo rendering for VR”. In: *San Diego Virtual Reality Meetup*. Vol. 2015, January 20. 2015.

[Wil22] Brianna Wilson. *Building a Parametric Seashell*. https://observablehq.com/@bronna/parametric-seashell. [Accessed 29-February-2024]. 2022.

[Wil24] Sascha Willems. *GPU Hardware Info Database*. http://gpuinfo.org. [Accessed 17-July-2024]. 2016–2024.

[Yoo+05] S.-E. Yoon et al. "Quick-VDR: out-of-core view-dependent rendering of gigantic models". In: *IEEE Transactions on Visualization and Computer Graphics* 11.4 (2005), pp. 369–382. DOI: 10.1109/TVCG.2005.64.

[Zha+15] Changgong Zhang et al. "Glyph-based comparative visualization for diffusion tensor fields". In: *IEEE transactions on visualization and computer graphics* 22.1 (2015), pp. 797–806.

[Zha+17a] C. Zhang et al. "Overview + Detail Visualization for Ensembles of Diffusion Tensors". In: *Computer Graphics Forum* 36.3 (2017), pp. 121–132. DOI: https://doi.org/10.1111/cgf.13173. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13173. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13173.

[Zha+17b] Changgong Zhang et al. "Comparative Visualization for Diffusion Tensor Imaging Group Study at Multiple Levels of Detail". In: *Eurographics Workshop on Visual Computing for Biology and Medicine*. Ed. by Stefan Bruckner et al. The Eurographics Association, 2017. ISBN: 978-3-03868-036-9. DOI: 10.2312/vcbm.20171237.