TU WIEN Informatics

Real-Time Distortion Correction Methods for Curved Monitors

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Wolfgang Rumpler, BSc
Matrikelnummer 01526299

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger, BSc

Wien, 4. Dezember 2022

Wolfgang Rumpler

Michael Wimmer

TU WIEN Informatics

Real-Time Distortion Correction Methods for Curved Monitors

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Wolfgang Rumpler, BSc
Registration Number 01526299

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Johannes Unterguggenberger, BSc

Vienna, 4th December, 2022

Wolfgang Rumpler | Michael Wimmer

Erklärung zur Verfassung der Arbeit

Wolfgang Rumpler, 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.

Wien, 4. Dezember 2022

Wolfgang Rumpler

Danksagung

Ich möchte mich an dieser Stelle bei allen Menschen bedanken, die mich während meines Studiums und dieser Arbeit unterstützt und begleitet haben. Meinen Dank möchte ich vor allem meiner Familie und meinen Freunden aussprechen, die meine Zeit an der Universität maßgeblich beeinflusst und zu einer unvergesslichen und schönen Zeit gemacht haben. Darüber hinaus wäre diese Arbeit ohne die Beratung und Unterstützung durch meine Betreuer nicht möglich gewesen. Ich bin dankbar für die Hilfe und Unterstützung, die ich zu jeder Zeit erhalten habe.

Vielen Dank!

Acknowledgements

I would like to take this opportunity to thank all the people who have supported and accompanied me during my studies and this thesis. Thanks to my family and friends, who greatly influenced my time at the university and made it a memorable and joyful venture. Furthermore, this thesis would not have been possible without the guidance and support of my supervisors. I am thankful for the help and support I received at all times.

Thank you!

Kurzfassung

In den letzten Jahren sind gekrümmte Computermonitore zu einer plausiblen Option im Privatbereich geworden. Herkömmliche Echtzeit-Grafikpipelines gehen jedoch davon aus, dass die Oberfläche des Bildschirms flach ist, und die meisten Echtzeitanwendungen, wie Spiele und Simulationen, berücksichtigen die tatsächliche Geometrie des Bildschirms während des Renderingvorgangs nicht. Folglich erscheinen die synthetisierten Bilder verzerrt und unnatürlich, wenn sie auf einem gekrümmten Bildschirm betrachtet werden.

Methoden zur Korrektur der Linsenverzerrung in Kameras und Head-Mounted Displays können auch für die Entzerrung auf gekrümmten Monitoren verwendet werden. Die auf dem gekrümmten Bildschirm zu sehende Verzerrung hängt jedoch auch von dem Blickwinkel ab. Mit Head-Tracking können perspektivisch korrekte Bilder erzeugt und eine genaue Krümmungskorrektur durchgeführt werden.

In dieser Arbeit analysieren wir verschiedene Methoden der Krümmungskorrektur für Echtzeit-Anwendungen in Abhängigkeit des Blickwinkels und der Geometrie des Monitors. Die durchgeführten Experimente bestätigen, dass bildbasierte Methoden die kürzeste Rechenzeit und akzeptable Bildqualität erreichen. Echtzeit-Raytracing und geometriebasierte Implementierungen stellen bei der Verwendung aktueller Hardware praktikable Alternativen dar und werden dabei nicht von Resampling-Artefakten beeinträchtigt. Zusätzlich beschreiben wir ein zur Hardware-Tessellierung alternatives Unterteilungsschema für geometriebasierte Ansätze. Dabei wird die Geometrie entlang eines am Bildschirm ausgerichteten Gitters unterteilt. Die dadurch generierte Geometrie approximiert die Oberfläche des Bildschirms besser als die vordefinierten Tessellierungsmuster der Hardware-Tessellierung. Wir präsentieren eine Implementierung dieses Schemas, welche mit einem einzelnen Renderpass in der neuen Graphics Mesh Pipeline auskommt. Obwohl die Performanz unserer softwarebasierten Implementierung des Unterteilungsschemas weiter verbessert werden muss, werden damit bei grober Geometrie weniger Dreiecke im Vergleich zur Hardware-Tesselierung generiert und eine gleichwertige Bildqualität erzielt.

Abstract

In recent years, curved computer monitors have become a viable option for consumers. However, traditional real-time graphics pipelines expect a flat display surface, and most real-time applications, such as games and simulations, do not consider the actual geometry of the monitor during rendering. As a result, the synthesized images appear distorted and unnatural when viewed on a curved display.

Distortion correction methods for correcting the lens distortion in cameras and head-mounted displays can also be utilized in real-time rendering software for curved monitors. However, the final distortion observed on the curved display depends on the user's viewpoint. With head-tracking, accurate distortion correction can be performed, and perspective-correct projections can be produced.

In this thesis, we analyze various methods for generating correct renderings based on the user's viewpoint and the geometry of the monitor. Our experiments confirm that image-based methods provide the best overall performance with acceptable image quality. However, real-time ray tracing and geometry-based implementations are practicable alternatives when using current hardware, and these methods do not suffer from image resampling artifacts. Additionally, we present and evaluate a custom subdivision scheme as an alternative to hardware tessellation for geometry-based solutions that can be implemented in a single render pass using the recently introduced graphics mesh pipeline. In our subdivision scheme, the geometry is split along a screen-aligned grid that reflects the geometry of the display more accurately than the fixed tessellation patterns of hardware tessellation. While the performance of our software-based subdivision scheme has to be improved further, it produces fewer triangles for coarse geometry and, at the same time, achieves similar image quality to hardware tessellation.

Contents

CHAPTER 1

Introduction

In recent years, consumer-grade ultra-wide curved monitors have become available to a broad user base and are increasing in popularity in the gaming and simulation sectors. Compared to traditional flat screens, one significant advantage of these displays is a considerably increased field of view (FOV).

For example, a flat monitor with a 49-inch diagonal, an aspect ratio of 32:9, and a viewing distance of 50 cm provides a horizontal FOV of 100°. A screen with the same dimensions yet featuring a curved surface with a curvature radius of one meter provides an increased horizontal FOV of 120°. This example is illustrated as a top-down view in Figure 1.1, where both monitor variants with the same surface dimensions are depicted with their respective FOV. Furthermore, from this figure, it is apparent that a flat monitor covering the same horizontal FOV as a curved one is considerably larger. The width of the flat monitor has to be increased by 44% from 1.2m to 1.73m to provide the same FOV.

The increased FOV positively affects immersion and can reduce simulator sickness in virtual environments [LDP+02]. At the same time, monitors do not suffer from the same challenges as other immersive display technologies such as head-mounted displays (HMDs). In contrast to desktop displays, these devices require a lightweight design for a comfortable user experience and displays with high resolution for both eyes [XHH+21, ZYX+20]. As a result, ultra-wide monitors constitute an attractive and comfortable solution for everyday content consumption in a desktop environment.

1.1 Problem Statement

Rendering virtual scenes for curved displays poses some interesting challenges, which are widely ignored by off-the-shelf games and real-time rendering engines. Traditional real-time rendering algorithms produce incorrect results for curved monitors if they are not adapted. From a technical point of view, rasterization graphics hardware is designed

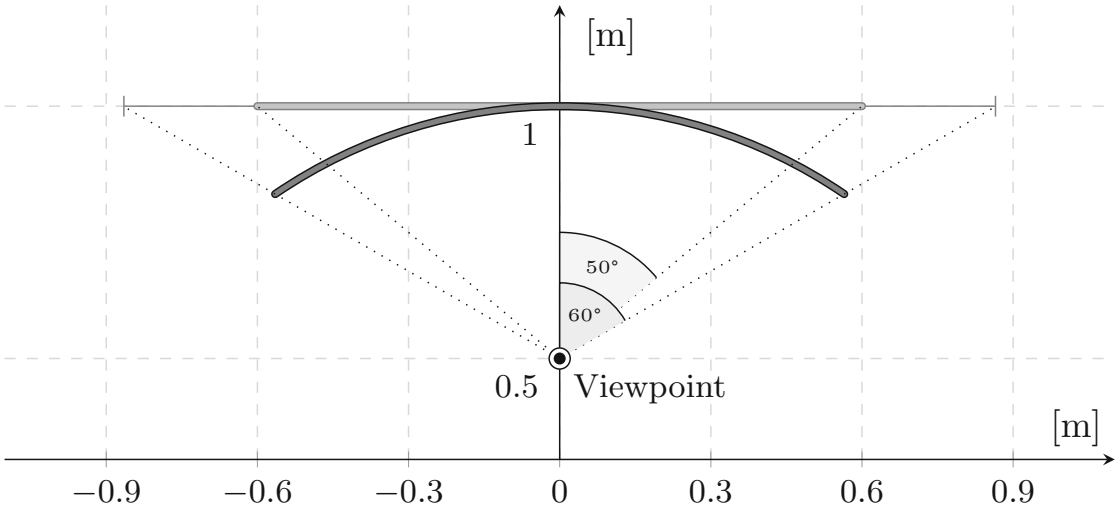


Figure 1.1: The FOV of a curved monitor surpasses that of a flat counterpart. Both monitors have the same surface dimensions with a width of 1.2m.

for rendering geometry onto a planar surface called the *image plane*. The resulting images, therefore, are presented correctly on flat screens only. On curved displays, images rendered in the same way are distorted since these monitors do not resemble the planar surface that is assumed by the hardware. For example, straight lines in the rendered image will be perceived to be curved [PRO19]. This distortion of straight lines on a curved monitor with a curvature radius of one meter is visible in Figure 1.2. A checkerboard pattern is displayed on the monitor and then captured with a camera. The checkerboard appears warped, and the horizontal lines bend towards the center of the image. These distortion artifacts are visible to a human observer in any content not generated according to the geometry of the display. This issue may reduce the appeal of heavily curved monitors.

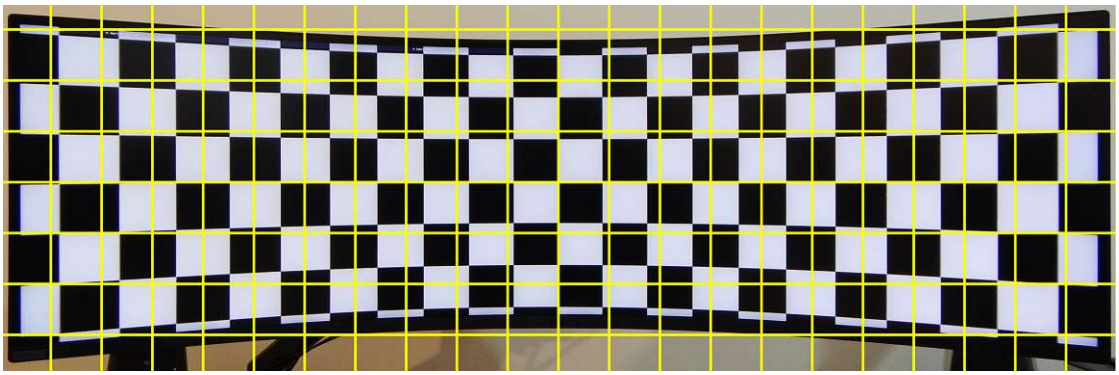


Figure 1.2: The distortion of straight lines on the curved monitor is clearly visible in the photo. The horizontal lines in the checkerboard appear to bend towards the center of the image. A rectangular grid is superimposed to highlight the deviation from a regular grid.

Besides the curvature of the monitor, the distortion also depends on the user's position relative to the display. Salomon [Sal06] and Bourke [Bou04] discuss this problem in the context of off-axis fisheye projections. Additionally, moving the viewpoint closer to the screen or further away will result in variation of the FOV [Bou04]. Since users typically do not sit in the exact same position in front of a monitor and move slightly during usage, it is crucial to consider their position for distortion correction. Furthermore, the wide FOV of ultra-wide monitors presents the opportunity to use the natural FOV of the display during image synthesis such that the projection corresponds to the viewpoint of the user. These properties naturally motivate head-coupled perspective—also referred to as Fish Tank Virtual Reality (FTVR)—where the scene is rendered based on the user's exact viewpoint [WAB93]. With head-coupled perspective, a display appears like a window into the virtual world and reflects the user's head movements. For example, looking around corners in the virtual world becomes possible by simply moving the head. Head-coupled perspective also enables the perception of motion parallax, a prominent depth cue that helps interpret geometry even in the absence of other cues [RG79, WAB93, HB16]. The exact position of the user's viewpoint relative to the monitor must be determined accurately to implement an FTVR effect. Fortunately, readily available webcam-based software and consumer-grade hardware for head tracking exist for this purpose [HTB+, Tob].

Distortion correction combined with head-coupled perspective is arguably an improvement over current practices for curved screens, as only this combination renders scenes correctly for the user's viewpoint. Unfortunately, this combination is not the standard implementation found in real-time applications in a desktop environment. This status quo is the source of our interest in this topic and the incentive for this thesis.

1.2 Aim of the Work

The core motivation for this work is the lack of a contemporary and detailed comparison of the various distortion correction methods in a modern off-the-shelf desktop setup with an ultra-wide curved monitor. Especially with the addition of head-coupled perspective, the best approach in such a setup has yet to be determined. In contrast to most work on distortion correction in real-time rendering, we are interested in their applicability for desktop applications on curved monitors rather than projectors or HMDs. Hence, our research aims to identify a practical technique in the presented context while achieving high image quality and reasonably low impact on rendering performance. Identifying and comparing the advantages and disadvantages of different methods and evaluating image quality and performance for curved displays are important considerations. Therefore, we analyze and compare multiple methods for distortion correction with sufficiently complex scene geometry, representing a realistic workload, with the combination of a head-coupled perspective in a real-world experiment. Readily available head-tracking solutions allow us to retrieve the user's viewpoint and determine the correct FOV for an accurate distortion correction and perspective-correct rendering.

We expect that the proposed methods to generate correct images for the monitor used in our experiments will suffer from noticeable artifacts due to its sheer size and significant curvature radius of one meter. Compensating these artifacts with additional subdivisions of the geometry for geometry-based methods or rendering at higher resolutions for image-based methods will significantly impact the performance in geometrically complex scenes. Analyzing this impact on quality and performance is one part of our work. Additionally, we test and evaluate an alternative implementation of a geometry-based solution, which has recently been enabled with the introduction of *Mesh Shaders* [Kub18]. We investigate whether subdivision of the geometry along a grid in screen space translates to a better distortion correction compared to using the tessellation pipeline. We are confident that investigating a method based on mesh shaders leads to relevant research findings for this particular problem. Furthermore, we expect to identify additional potential research topics for future work.

1.3 Contributions

Based on our experiments and the analysis of the quality and performance of different methods, we provide recommendations for applications with support for perspective-correct rendering on curved monitors. Besides the reevaluation of solutions described in the literature, we also present the implementation of a custom grid-based subdivision scheme for geometry-based methods.

In our subdivision scheme, the geometry is split along a regular grid in screen space to reflect the geometry of the screen more accurately compared to the fixed subdivision patterns of hardware tessellation. We show that this approach generates fewer triangles for coarse geometry while achieving similar image quality compared to an implementation using hardware tessellation. Furthermore, our subdivision scheme can be implemented in a single pass in the new *graphics mesh pipeline* [The22] with performance suitable for real-time applications.

1.4 Structure of the Work

In the following Chapter 2, we provide some background on linear perspective, describe the state of the art, and present various pieces of related work. Additionally, we briefly describe recent hardware developments relevant to the implementation of our experiment. In Chapter 3, we describe our scientific method and vital consideration to arrive at sound results during our evaluation. Chapter 4 contains detailed descriptions and technical details of the implemented methods. Results of our measurements, the analysis, and the discussion of the advantages and disadvantages of the implemented methods are provided in Chapter 5. Following the analysis, we summarize our findings, explain the limitations of our work, and present potential future research topics in Chapter 6.

CHAPTER 2

Background & Related Work

Linear perspective has a long history in arts and mathematics. Already the ancient Greeks used some form of perspective in their drawings, but the first experiments with a central perspective where straight lines converge towards a single vanishing point are attributed to Filippo Brunelleschi in 1425 [Smi87, p.351-353].

As popular as the linear perspective projection is for generating the appearance of depth in artworks, it is also often subject to debate due to unsatisfactory results in certain situations [Reh03, Fle95, Der99]. One simple downside of perspective projection is its limitation to a FOV smaller than 180°. Another criticism is that images created with a linear perspective can provide an incorrect impression of the size of objects. The size of the columns projected onto the image plane, depicted in Figure 2.1, does not correspond to the distance from the viewpoint. While both columns have the same size, the one nearest to the viewpoint has a smaller image than the second one [Der99]. This exact problem was also already documented by Leonardo da Vinci [Smi87, p.356]. The effect becomes even more pronounced with a wider FOV. Artists try to hide this unnatural widening effect by limiting the maximum angle of the visual field in their artworks [Der99]. Unfortunately, no reasonable angular limit prevents a viewer from perceiving a wrong impression as the perspective depends on the observer's viewpoint [Gib60]. In order to get a realistic impression of an art piece, the work must be viewed from the same viewpoint that the artist chose for its creation.

Regardless of its limitations, linear perspective is an important subject in mathematics and is the dominating model in computer graphics [Bay95]. It is the appropriate choice to picture three-dimensional content as a flat image since the human vision begins with a perspective projection onto the retina [Fle95]. In other words, the image should provide the illusion of the same perspective projection that is sensed by the eye of the observer. Consequently, since most displays are flat, current hardware for computer graphics is designed with a linear perspective in mind. Unfortunately, displaying an image created with linear perspective projection on a curved monitor is fundamentally incorrect as the

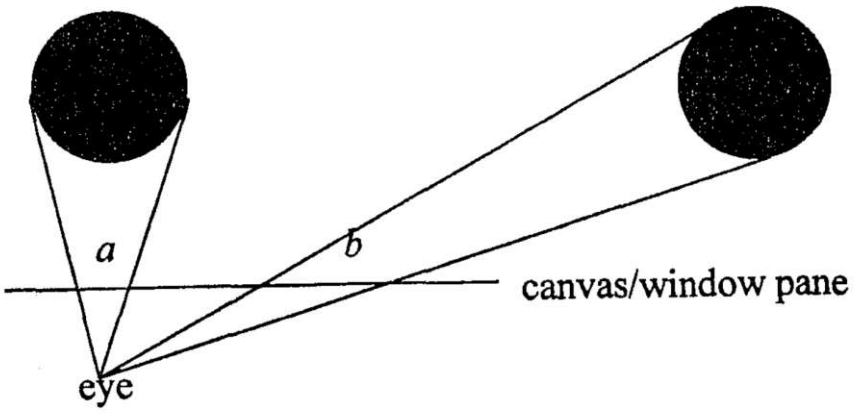


Figure 2.1: Perspective projection can produce an inaccurate impression of the size of objects (reprint from Derksen [Der99]). The equally sized columns have differently sized images a and b. The column nearest to the viewpoint has a smaller image than the column further away.

model for computation and the display surface in the real world do not match. In these cases, it is necessary to correct the images to create the desired impression of depth. Similar situations where distortion must be corrected in images come up regularly when lenses or curved display surfaces are involved. Various correction methods for lenses in cameras, for projections onto curved surfaces, or for the lenses in HMDs have been developed and are discussed in the literature.

2.1 Real-Time Correction Methods

The methods for implementing distortion correction in real-time on modern hardware can be divided into three categories. They all have different characteristics, advantages, and disadvantages. The first option is to use ray tracing to generate non-linear projections [AG93, GHFP08]. Ray tracing generally comes with a high computational cost and, despite recent hardware developments for accelerated ray tracing [NVI18], it still is considerably slower than traditional rasterization. An alternative is to warp a linearly projected image in a post-processing pass to generate the desired projection [WH95, PJB13, TNAM16]. This method generally reduces the sharpness of the rendered images due to the inevitable image resampling involved [PJB13]. The third solution is to project the geometry with a non-linear projection such that the desired projection is approximated during rasterization [GG99, SBGS06, JMN+08, ALMM14, PRO19]. Projecting the geometry with a non-linear projection preserves image sharpness but produces inaccuracies and artifacts for coarse geometry. The vertices are non-linearly transformed during projection but then linearly interpolated by the graphics hardware, which is incorrect w.r.t. the non-linear transformation [PRO19]. The artifacts can be reduced with

additional subdivisions of the geometry [PJB13, PRO19]. Another technique assigned to the same category is to approximate the desired projection with many linear projections [LD09]. In this work, we refer to these three categories as *ray tracing*, *image-based*, and *geometry-based* methods, respectively. In the following paragraphs of this section, we briefly describe related pieces of work.

Acquisto and Gröller [AG93] present how ray tracing can be used to generate a wide range of projections. For ray tracing various projections, only the initial ray direction and origin have to be adapted accordingly. The authors note that the calculation time for these two vectors accounts for a small part of the total time of the image synthesis only.

An efficient method for distortion correction for HMDs is presented by Watson and Hodges [WH95]. The authors use texture mapping hardware to map the linear projection of a scene onto a distorted mesh and thereby generate the desired image. Their distortion mesh is carefully generated with a parametric model such that it exactly counteracts the effects of the lenses used in the HMD.

When a parametric model is not applicable for describing the distortion, textures describing the displacement can be used instead. For image rectification, this approach allows capturing local deformations of lenses that a global model does not capture well [RVSS10]. In the context of rendering, a similar concept known as camera textures can also be applied directly to the geometry of a scene to produce images with non-linear projections [SBGS06].

Glaeser and Gröller [GG99] discuss differences between various curved perspectives for use in virtual reality applications. They use a geometry-based method to generate their images and subdivide every edge based on the length in the final image. Whether and how the inside of faces of the geometry is subdivided is not mentioned. They conclude that their solution is feasible for real-time animation in reasonably complicated scenes.

Jo et al. [JMN+08] present an implementation of a geometric distortion correction in a game engine for a cylindrical immersive environment. However, they require a fixed viewpoint position and, therefore, do not compensate for the off-axis distortion when the viewpoint changes.

Gascuel et al. [GHFP08] implement non-linear projections on standard graphics hardware in the fragment shader. They calculate the screen-space bounds for every primitive and render a bounding primitive using rasterization. In the fragment shader, they cast rays to the actual primitives to discard any unnecessary fragments from the bounding region. They also present how the screen space bounds of a triangle are found for a select set of non-linear projections.

Trapp and Döllner [TD08] describe Projection Tile Screens as an approach for creating arbitrary projections with a single center of projection for real-time applications. Their solution is based on rendering dynamic cube maps of the scene and then distorting it based on a texture describing the desired projection. The authors explain how the texture

can be created and note that combining linear and non-linear projections is possible with their solution.

Bourke [Bou09] discusses projection onto a spherical dome, presented as *iDome*, via a spherical mirror. He adapts the rendering pipeline of the Unity[1] game engine to render correct projections for this particular hardware. In his approach, four sides of a cube are rendered to generate the desired FOV. These images are then distorted into the necessary shape by applying them to a mesh with carefully constructed texture coordinates. Besides the geometry of the dome, the geometry of the mirror, which is used to redirect the projection onto the surface, has to be taken into account. The author describes a performance overhead of a factor of 2–3 compared to rendering a single side of the cube instead of four.

Lorenz and Döllner [LD09] present Piecewise Perspective Projections (PPP) where they perform a large number of linear projections to approximate non-linear projections. They present their implementation for the two use cases of cylindrical projections and an adaptation of camera textures. Their approach avoids image resampling and, therefore, achieves better image quality for the cylindrical projection while simultaneously achieving superior performance compared to an image-based implementation where they distort a cube map. However, they observe a significant overhead for the camera texture use case, where they use 128×128 projections for the approximation. The authors contribute this overhead to the fact that every triangle has to be replicated for every projection containing it. Furthermore, the authors note that they have to perform two projections for every grid cell—one for each triangular half of the projection plane—as hardware rasterizers do not support arbitrarily shaped quadrilaterals as image plane [LD09]. Based on the data provided in their paper, the PPP method with a 128×128 grid reduces the frames per second (FPS) by more than a factor of 10 when compared to an image-based method.

Díaz Hernández [DH11] implements two geometry-based methods in a game engine for a large cylindrical immersive environment with stereoscopic 3D and head tracking. The implemented solutions are a slice-based one similar to PPP and projecting the geometry with the non-linear projection. The author concludes that their implementation with the non-linear projection is more efficient than PPP. However, no additional subdivision of the geometry to avoid artifacts from linear interpolation is mentioned.

An effective improvement for image-based methods and a detailed comparison is presented by Toth et al. [TNAM16]. The authors first approximate the distortion with a few piecewise projections, warp these views with an image-based approach, and stitch them together for the final result. Using multiple projections reduces the total resolution required to compensate for the loss of quality during warping. Furthermore, their method can be implemented efficiently on modern hardware using extensions for current rendering pipelines [BS18].

[1] `https://unity.com/` (last accessed on July 2, 2022)

Pohl et al. [PJB13] improve the sharpness of image-based distortion correction by using bicubic texture filtering instead of bilinear filtering. They compare both texture filtering methods with ray-traced images using an edge detector to quantify the difference in sharpness and an image similarity index based on human perception. For both metrics, they achieve superior results with cubic texture filtering over linear filtering. They also present performance differences when performing image warping using a distortion mesh compared to distortion in the fragment shader. Their measurements show that using a distortion mesh with 7200 triangles achieves comparable performance to deforming the image in the fragment shader.

Ardouin et al. [ALMM14] solve artifacts for triangles spanning discontinuities in wide-angle projections with a FOV greater than 180° for geometry-based methods. They implement an additional clipping stage to split problematic triangles along the discontinuity such that the rasterizer can process them without generating artifacts.

Muszyński et al. [MGN18] present a comparison of various non-linear projection methods in a software rasterizer. Unfortunately, they do not evaluate the image quality in detail, and the frame time achieved by their software rasterizer can not be translated to the performance of an implementation on a graphics processing unit (GPU) directly.

Martschinke et al. [MMSB19] address the distortion in HMDs resulting from the displacement of the projection center caused by the rotation of the eyeball. This displacement, similar to head movements in our experiment, has to be considered for distortion correction. They utilize a correction method without a parametric model based on a displacement map describing the distortion of their lenses. It is important to note that they only consider eye movement on a plane.

The work done by Pérez et al. [PRO19] presents an implementation of geometric distortion correction for FTVR on curved monitors using hardware tessellation. The authors observe a performance impact of about 20% in their implementation with tessellation compared to a traditional linear projection. They do not provide a comparison to an image-based solution. Furthermore, their setup uses a display with a curvature radius of 1.8 meters. We expect that a more substantial curvature requires a higher tessellation and, therefore, results in a higher impact on performance.

2.2 Recent Hardware Developments

Recent advances in graphics hardware allow the implementation of new variations on methods for generating non-linear projections in real time. We briefly describe the most significant changes to the traditional graphics pipeline relevant to this work.

2.2.1 Real-Time Ray Tracing

Ray tracing is a standard method for image synthesis. The fundamental idea is to trace back light rays reaching the camera to find the origin of the rays. This process

is computationally expensive and was not feasible for real-time applications in the past [NVI18]. However, with *NVIDIA RTX*, as a part of NVIDIA's GPU architecture *Turing*, the necessary hardware and software components to generate ray-traced images in real time are introduced [NVI18]. The main hardware components of NVIDIA RTX accelerate the traversal of the hierarchical data structure containing the scene geometry and the ray-triangle intersection calculation. The new hardware is accessed with a new ray tracing pipeline supported in current graphics application programming interfaces (APIs) such as Vulkan [LGB22]. We use this real-time ray tracing support of NVIDIA's graphics architecture to generate corrected renderings in real-time.

The scene geometry is structured in a hierarchical manner using two types of data structures [NVI18, LGB22]. The first data structure is the top-level acceleration structure (TLAS). The geometry of an object may be included in a scene multiple times with different positions. These object instances are stored in the TLAS. An entry in the TLAS points to the second type of data structure, the bottom-level acceleration structure (BLAS), that contains the geometric primitives. As noted earlier, traversing these data structures and finding the intersection of a ray with a triangle in the BLAS is done in hardware.

The execution of the ray tracing pipeline follows the flow diagram shown in Figure 2.2. The colord stages, *ray generation*, *miss*, *closest hit*, *intersection*, and *any hit*, in the figure represent programmable shader stages [LGB22]. The ray generation stage is the entry point of the pipeline and is used to cast the initial rays into the scene. A payload can be attached to the traced rays to communicate information between the shaders. The miss shader is executed if the ray does not intersect any geometry. The closest hit shader is executed for the intersection closest to the ray origin. The intersection shader can be used to implement custom intersection logic to, for example, intersect rays with procedural geometry. Finally, the any hit shader is executed for every intersection along the ray.

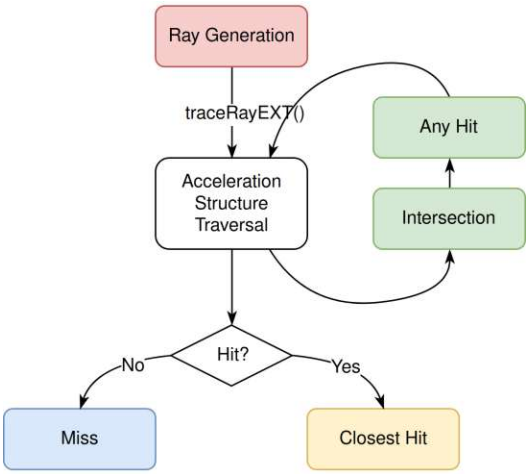


Figure 2.2: The pipeline structure of the ray tracing pipeline in Vulkan (reprint from Lefrançois et al. [LGB22])

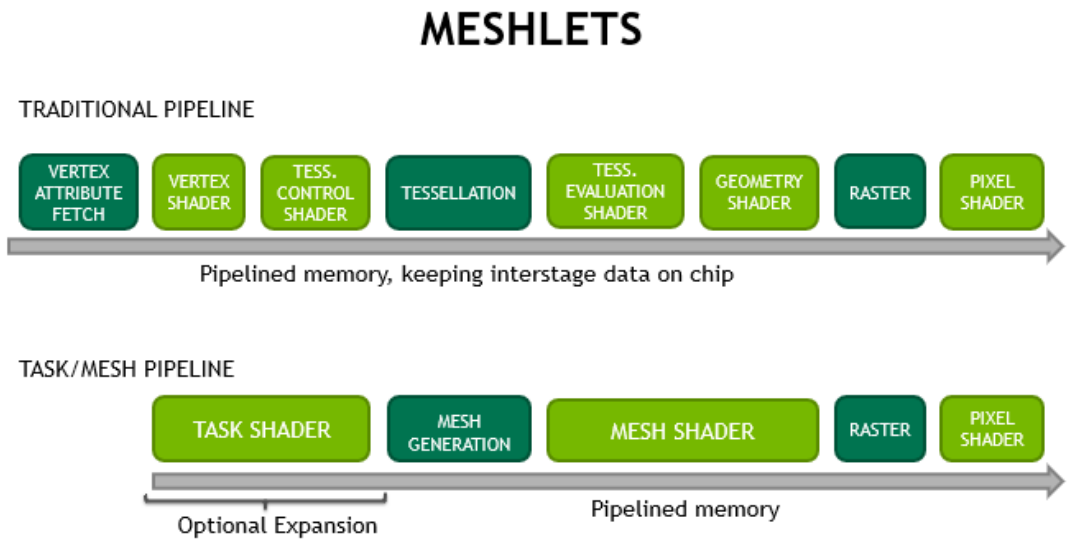


Figure 2.3: The stages of the traditional graphics pipeline compared to the graphics mesh pipeline (reprint from Kubisch [Kub18]).

2.2.2 Graphics Mesh Pipeline

With the release of NVIDIA's Turing GPU architecture *Mesh Shading* was introduced [NVI18]. It is meant as a more performant and flexible alternative to the traditional graphics pipeline [Kub18]. In the Vulkan API specification the pipeline is identified as *graphics mesh pipeline* [The22]. We utilize the flexibility of the new pipeline in our work to implement a custom subdivision scheme for a geometry-based implementation.

Figure 2.3 depicts the stages of the traditional pipeline and the new mesh pipeline. The dark green stages of the pipeline represent fixed stages that are configurable, and the light green stages are programmable. One structural difference to the traditional pipeline is the consolidation of stages. The vertex and tessellation control stages appear to be combined in a *task shader* stage. Tessellation evaluation and geometry shader are superseded by the *mesh shader* stage.

The programmable task shader is used to modify the number of mesh shader workgroups spawned and to supply additional information to the mesh shader. Every task shader workgroup emits the built-in value `gl_TaskCountNV` to define the number of mesh shader workgroups required. User-defined data is passed on to the mesh shader in a buffer defined in the shader code. The mesh shader generates the final geometry for the rasterizer by writing it into the built-in buffers `gl_MeshVerticesNV` and `gl_PrimitiveIndicesNV`. These buffers are shared between the invocations of a workgroup and contain the vertices and triangle indices, respectively. To quantify the final number of triangles written to the buffers, `gl_PrimitiveCountNV` has to be specified. The size of the output buffers is defined with a constant in the shader code

of the mesh shader [Kub18]. After executing the mesh shader, the generated geometry is rasterized as usual. A linear interpolation of the vertex attributes occurs as a fixed function step during rasterization right before the fragment shader is executed.

The new pipeline stages use the same programming model as compute shaders [Kub18]. This programming model allows using the invocations of the work groups for arbitrary computations. Data can be passed to the shaders in any arbitrary form using standard GPU buffers. Typically, geometry is represented in the form of *meshlets*, which are small sets of the vertices and primitives of objects.

2.2.3 Vulkan Subgroups & Group Operations

Task, mesh, and compute shaders are executed as a collection of work groups. Every workgroup consists of a fixed number of invocations specified in the shader code. Workgroups and invocations are identified by their respective coordinates.

Subgroups in Vulkan are a subset of invocations of a work group that can be synchronized, and data can be shared efficiently between them [The22]. Within a subgroup, group operations can be performed to communicate between the invocations. Essentially, the operations allow synchronizing, voting, performing arithmetic operations, and sharing data across the active invocations. Depending on the program flow, an invocation may be considered active or inactive [Hen18]. Active invocations contribute to the result of an operation. For example, an invocation that exits from the shader right before a group operation is executed is inactive and does not contribute to the outcome.

Generally, group operations perform conceptually simple tasks. The function `bool subgroupAll(bool value)` returns *true* when all invocations provide the same boolean value and *false* otherwise. For arithmetic operations, there are two different variants. For example, a sum operation can return the cumulative result of all contributing invocations to every invocation or only the sum up to the index of the respective invocation. These two variants are called reduction and scan operations, respectively [The22]. Furthermore, scan operations can be inclusive or exclusive, describing whether they include the contribution of the calling invocation or not. For example, an inclusive sum over multiple invocations will return the sum of the first n invocations for the n^{th} invocation. In the case of an exclusive scan, the n^{th} invocation will receive the sum over the first $n-1$ invocations.

We use group operations in our geometry-based method that is implemented using the graphics mesh pipeline. The different operations are used to distribute the work and efficiently communicate auxiliary information between shader invocations.

CHAPTER 3

Methodology

Our methodology is based on experimentation with various real-time correction methods and their evaluation using a current hardware setup. We investigate the impact of the method-specific parameters on image quality and performance to find appropriate configurations for every method. The analysis across the implemented methods is then based on the determined configurations.

We use the recently introduced ꟻLIP difference evaluator, an image difference evaluator based on human perception, to quantify the perceptual image differences between the methods and reference images [ANAM+20]. The perceptual image difference is interpreted as an approximation of the quality achieved by a specific configuration. The performance is evaluated by measuring and comparing the frame time of the distortion correction methods in scenes with sufficiently complex geometry. Various parameters like the FOV, rendering resolution, and tessellation level are investigated. These measurements provide an overview of the performance penalty that has to be expected for the various methods in the context of current real-time applications.

Finally, we also implement support for consumer-grade head-tracking solutions and provide a subjective assessment of the performance and quality of current tracking devices in the context of FTVR. While a user study would be appropriate to evaluate the tracking performance, it is considered out of the scope of this work as we concentrate on the technicalities of the different rendering methods.

3.1 Implemented Methods

We compare four methods for rendering a corrected perspective onto a curved monitor: ray tracing with hardware acceleration, image-based correction, a geometry-based variant using tessellation shaders, and a geometry-based variant with a custom subdivision scheme implemented using the new graphics mesh pipeline. These solutions have different

characteristics in terms of performance and quality, and their implementations enable the creation of a comparative overview of their advantages and disadvantages. Except for the texture warping method, all selected approaches are executed in a single pass. Only basic information about the user's viewpoint and the geometry of the monitor has to be supplied to the shaders, and no additional data structures are required. These properties make the methods reasonably easy to add to existing software.

Hardware-Accelerated Ray Tracing. A basic implementation of the correct perspective for a curved screen is trivial when using ray tracing. Ray tracing is a viable option for real-time rendering on modern GPUs where hardware acceleration is present. Since hardware support for ray tracing is relatively new at the time of writing, it is currently not the standard way to render entire scenes in real-time. Therefore, the rendering performance of the implementation using ray tracing can generally be expected to be lower than the implementations based on rasterization. Typically, the primary use case is the implementation of effects that are difficult to achieve with a rasterizer, such as accurate reflections on arbitrary surfaces.

Image-Based. The image-based method renders the scene into a texture using a single linear perspective projection described by a particular projection matrix. Then the image is warped into the desired perspective in a post-processing step. Many rendering engines already utilize additional post-processing passes for various visual effects. Therefore, the image-based method for correcting the perspective is considered a trivial extension in most cases. Only the projection matrix must be modified, and the corresponding post-processing pass must be added.

Geometry-Based Using Tessellation Shaders. One of our geometry-based methods uses hardware-supported tessellation and is based on the approach presented by Pérez et al. [PRO19]. The standard rasterization-based graphics pipeline is utilized with tessellation shaders for subdividing the geometry for accurate rendering. It uses a simple heuristic to determine the required tessellation level of a triangle based on the midpoint of every edge. We extend this heuristic to reduce artifacts in certain situations where the heuristic fails to produce sufficient tessellation levels.

Geometry-Based Using Mesh Shaders. Our second geometry-based method utilizes the graphics mesh pipeline to implement a custom subdivision scheme. The geometry in the view volume is subdivided based on a regular grid on the display surface. A grid cell corresponds to a segment of the view volume. Triangles intersecting the bounds of this volume are split and triangulated prior to the rasterization. We argue that this subdivision corresponds to the required distortion closely since the subdivision is based on the actual display surface. Furthermore, no matter how large a triangle is on screen, only the visible portion is split into smaller triangles.

3.2 Accuracy of Rendered Results

We capture the display with a physical camera from multiple positions to verify that our implementations are accurate for varying viewpoints. For this purpose, a scene consisting of a single plane with a checkerboard texture is rendered on the screen. The virtual camera is placed such that the view direction is orthogonal to the plane. If the distortion of the screen is corrected accurately, then the images captured with the physical camera depict a rectangular checkerboard pattern. In other words, a rectangular grid can be superimposed onto the photo and aligned with the checkerboard pattern displayed on the screen. For incorrect projections, the lines remain curved in the image.

Note that the superimposed grid must be transformed according to the perspective of the physical camera to match the rendered checkerboard. The positions of the virtual checkerboard and the virtual camera are constant during the measurements. Still, the rendering is corrected for the viewpoint of the physical camera. Therefore, the checkerboard appears fixed in space, similar to a painting hanging on a wall. Typical foreshortening effects of linear perspective are visible for viewpoints at an angle to the monitor.

While, in theory, a superimposed grid and the checkerboard pattern should match perfectly, we expect minor errors. Contributing sources of errors are inaccuracies during the measurements of the camera position relative to the monitor, deviation of the display surface from a perfect cylinder, and inaccuracies during the transformation of the grid. However, we do not investigate these errors further since they are relatively small, and the main purpose of these measurements is to verify the accuracy of our distortion correction methods. Also, it is sufficient to test a single implementation as all other implementations fundamentally produce the same projection.

3.2.1 Camera Calibration

An important consideration for the photos is that camera lenses can also introduce distortions. Such lens distortions would skew any measurements, and lines on the photos may look bent even when the rendering itself is correct. One example of such distortions, although typically desired, is visible in photos taken with fisheye lenses. Therefore, we need to test the accuracy of the camera by photographing a printed checkerboard pattern and again comparing it to a grid in the previously described way. If the checkerboard pattern in the image does not align with a superimposed grid, we know that the camera optics introduce an additional distortion. In this case, the distortion parameters of the camera have to be estimated, and the image corrected to a linear perspective to get an unbiased measurement. Software solutions for camera calibration exist and can be used to estimate the distortion parameters of the optics [Bou15]. However, cameras with fixed lenses are typically calibrated by the manufacturers already. In Figure 3.1, an image of a printed checkerboard superimposed with a regular grid shows that the camera used for our measurements is accurate enough for our purpose by default. Therefore, no further calibration is performed.

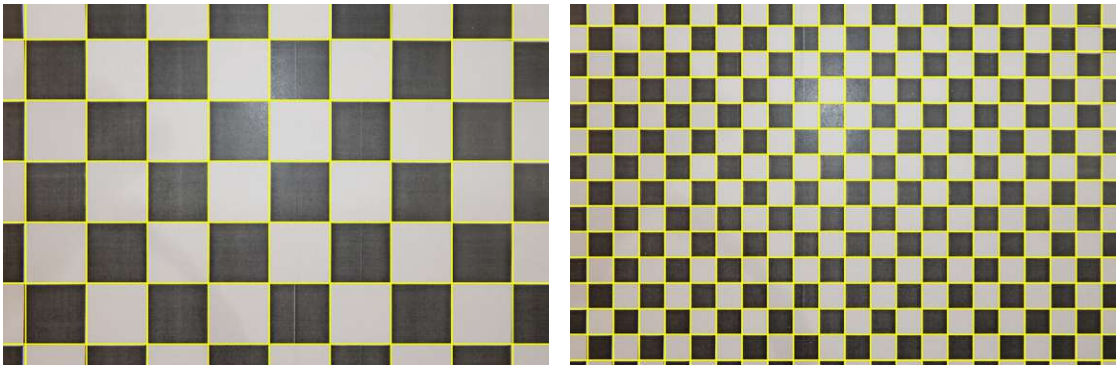


Figure 3.1: We capture two photos of differently sized checkerboard patterns to test our camera for distortion artifacts that may bias measurements of our implementation. The superimposed grid matches the captured checkerboard well. The images verify that the camera used for measuring has been calibrated sufficiently by the manufacturer for our purpose.

3.3 Quality Evaluation

While the implemented methods produce valid results, they all suffer from some artifacts. These artifacts are subject to our interest and are investigated in detail. To quantify the perceivable difference to a reference image, we utilize the difference evaluator ꟻLIP [ANAM+20]. ꟻLIP is designed to quantify the perceptual difference between two alternating images regarding human vision. The algorithm is based on a combination of image differences in colors and differences in image features such as edges and points. A low image difference suggests that the two images cannot be distinguished easily by a human, and the renderings can potentially be used interchangeably. With ꟻLIP, we can accurately and efficiently capture the perceptual difference for a large data set.

Evidently, choosing a proper reference image is an essential consideration for arriving at reasonable conclusions based on ꟻLIP as the difference metric. The primary concern regarding the choice of the reference image is that we can not expect to arrive at perfectly identical images due to the different rendering methods. For our measurements, the differences resulting from the various rendering methods have to be reduced as much as possible. In other words, the noise from using different rendering methods has to be as low as possible to measure the actual flaws of the methods accurately. The differences between the methods are influenced by the factors described in the following paragraphs.

Mipmap-based texture filtering [Cro84], a common practice in real-time rendering, makes comparing different methods difficult. The geometry- and image-based methods would use different Mipmap levels since the projection used during the respective rasterization stages generate different screen-space footprints for the same geometry. Furthermore, texture filtering for ray tracing also has to be considered. During ray tracing, texture filtering can be performed by utilizing supersampling or ray differentials [Ige99]. However, we are not aware of any standard solution for ray tracing that generates results equivalent

to the rasterizer. Entirely leaving out textures would resolve these problems but would, at the same time, hide errors due to the linear interpolation of texture coordinates for geometry-based methods. Consequently, we do not use Mipmap-based texture filtering in our measurements.

Generally, we have noticed a good correspondence between rasterized geometry and ray-traced results when a single ray per pixel is cast through the pixel center, and no texture filtering is used during rasterization. Unfortunately, taking a single sample per pixel and disabling texture filtering results in significant aliasing in the images. As a result, image differences from resampling an aliased image in the image-based method cannot be attributed conclusively to the method itself since the sampled signal was noisy in the first place. To reduce aliasing across all images, we employ Supersampling Anti-Aliasing (SSAA), where multiple samples per pixel (SPP) are calculated and combined. Supersampling can be implemented easily for ray tracing and can be configured on rasterization hardware. We generate all rasterization-based images with 8 SPP using hardware-supported SSAA and 9 SPP distributed on a uniform grid for ray tracing to reduce aliasing in the images.

Regarding the selection of the reference images, ray tracing seems like the obvious candidate for generating reasonable reference images since it is essentially artifact-free. However, we refrain from this option to reduce the baseline difference for the rasterization-based methods, such as the geometry-based and image-based implementations. While, in theory, it is possible to create the exact same sampling of a scene with ray tracing compared to that produced by the rasterization pipeline, in practice, this objective is a notable effort and depends on the implementation details of the specific hardware. We avoid this effort by using our grid-based method to generate the reference images. When a grid resolution of 2560×720, i.e., half the render resolution, is used, the grid produces a vertex per pixel for every triangle during subdivision. Due to the high geometric density produced, the reference images do not suffer from geometric artifacts similar to ray tracing. At the same time, we can use the same pipeline configuration for all rasterization-based methods to ensure a consistent baseline.

With the described configuration to reduce the base differences between the methods, we evaluate the ꟻLIP error of the implemented methods at 100 uniformly distributed sample points along camera paths through various scenes. These samples are rendered in the native resolution of the test monitor at 5120×1440 pixels. The mean error over all samples provides insight into the ability to recreate the reference images for each method. Furthermore, ꟻLIP allows us to present difference maps to highlight and discuss recognized artifacts of the methods in detail.

Another consideration regarding ꟻLIP is that the angular resolution that the observer experiences is taken into account. This value depends on the user's viewpoint and has to be supplied in Pixels Per Degree (PPD) to the algorithm for accurate perception-based results. The authors provide a formula to calculate the PPD [ANAM+20]. Their formula calculates the PPD at the closest point for a flat display surface with a given width, resolution, and viewing distance. We can use the same formula with the shortest distance

to the curved display surface as long as the ray corresponding to this distance intersects the cylinder at a right angle. A comparable flat display at the same distance would yield the same PPD. This property is provided for viewpoints inside the cylinder sector that the display surface forms. Therefore, we restrict the viewpoints during image difference evaluation to the cylinder sector and use the same formula with the shortest distance to the cylinder surface as viewing distance and the arc length of the display as width.

3.4 Performance Evaluation

We are interested in the performance of the different methods for diverse and geometrically demanding scenes comparable to the workloads presented by modern real-time applications. Therefore, we utilize three different scenes for evaluation to represent varying complexity. These scenes are *UE4 Sun Temple* (0.6M triangles) [Gam17], *Amazon Lumberyard Bistro* (2.8M triangles) [Lum17], and *NVIDIA Emerald Square* (10M triangles) [NHB17] from the Open Research Content Archive (ORCA) [NVI22].

In order to perform a comparative evaluation of the performance, we measure two basic metrics. The first metric, and arguably the one of main interest, is the processing time on the GPU. Since the selected methods are implemented entirely on the GPU, we do not need to consider any time spent on the central processing unit (CPU). The time on the GPU is measured using timer queries placed right before and after the execution of the relevant draw calls. The second metric is the final number of triangles sent to the rasterizer after clipping. This metric enables the investigation of the amount of tessellation performed for the geometry-based methods. We take 1000 uniformly distributed measurements of these metrics during camera tours through the respective scenes to get insight into the behavior over time. Note that in contrast to the quality evaluation, we do not employ any SSAA during the performance evaluation. This measure is taken since SSAA is not representative of current practices in real-time rendering and would bias the performance measurements.

To minimize differences in performance due to different implementations, we reuse the same shading code and, where possible, use the exact same functions. One difference in the implementations of the methods is the application of view culling for the geometry-based methods. Geometry outside the visible volume is discarded to avoid the tessellation of geometry that is not visible on the screen. This optimization is a trivial extension when tessellation shaders are used. Similarly, by the nature of our proposed implementation in the graphics mesh pipeline, view culling is considered a side effect. Performing view culling for the geometry-based methods and not for the image-based method is no disadvantage for the later implementation. By pointing the camera in a direction where no geometry is visible in each scene, we observe higher performance for the image-based method. This observation shows that our triangle-based view culling implementation is less efficient than solely utilizing the clipping hardware for discarding triangles outside the view. Generally, this may not be the case when entire objects are culled instead of every triangle individually or when acceleration structures are used. Another notable aspect is

that we do not use instanced rendering as there is no direct support for instancing when the graphics mesh pipeline is used. This kind of functionality has to be programmed in software using the task and mesh shaders. Finally, note that in the case of hardware-accelerated ray tracing, the hierarchical acceleration structures provided by the API have to be used.

We have performed our evaluation on the consumer-grade GPU *NVIDIA RTX 3070 Ti* using the official driver with the version number *522.25*. All benchmarks have been executed in full-screen with the native resolution of the display at 5120×1440 pixels. Every benchmark consists of various configurations depending on the parameter tested. For example, to test the influence of the FOV on performance, the various methods are executed with different viewing distances to the monitor. Every viewpoint represents a separate benchmark, with every method representing a single configuration in the benchmark. We have executed 100 iterations for every configuration in a benchmark to get a reliable performance estimate. All iterations of all configurations in a benchmark have been run in a random order to avoid bias due to the built-in dynamic frequency scaling of the GPU. Otherwise, a long benchmark may favor the first configurations over later ones as the GPU clock speed can be lowered during the benchmark due to thermal constraints or other metrics.

CHAPTER 4

Correction Methods

In the following sections, we provide details about the four methods implemented for correcting the perspective on a curved monitor. Additionally, we describe how head tracking is implemented in our experiment. All implementations assume that the user's viewpoint is inside the cylinder volume. Therefore, the methods may not work for viewing positions outside the cylinder without further adaptations.

In general, we perform all computations, if not stated otherwise, in the same space where the origin corresponds to the center of the cylinder matching the display surface. The same coordinate system also corresponds to our view space and tracking space. This coordinate system is shown in Figure 4.1, where the x-axis points to the right, $-z$ points into the center of the screen, and y points upwards.

None of the methods requires any special data structures or resources managed by the application. Therefore, we do not describe the general application logic and concentrate on the concepts used in the shader programs that are executed on the GPU. The only additional information that has to be provided to the shaders by the application is basic information about the display surface and the user's viewpoint in the tracking space. In our implementations, the corresponding values are sent to the shaders using a standard Uniform Buffer Object (UBO).

Furthermore, all methods share the same shader code where applicable. For example, the fragment shading code and the cylindrical projection function are identical for all methods. Also, the function used for view culling is the same for both geometry-based methods.

The illumination quality is not a priority. We, therefore, use a simple Phong Shading Model [Pho75] for calculating the surface illumination in the scenes. However, when implementing visual effects that depend on the view position, the exact viewpoint of the observer has to be used for a correct appearance. In our case, the specular reflection component in the shading model is the only view-dependent effect in our implementations.

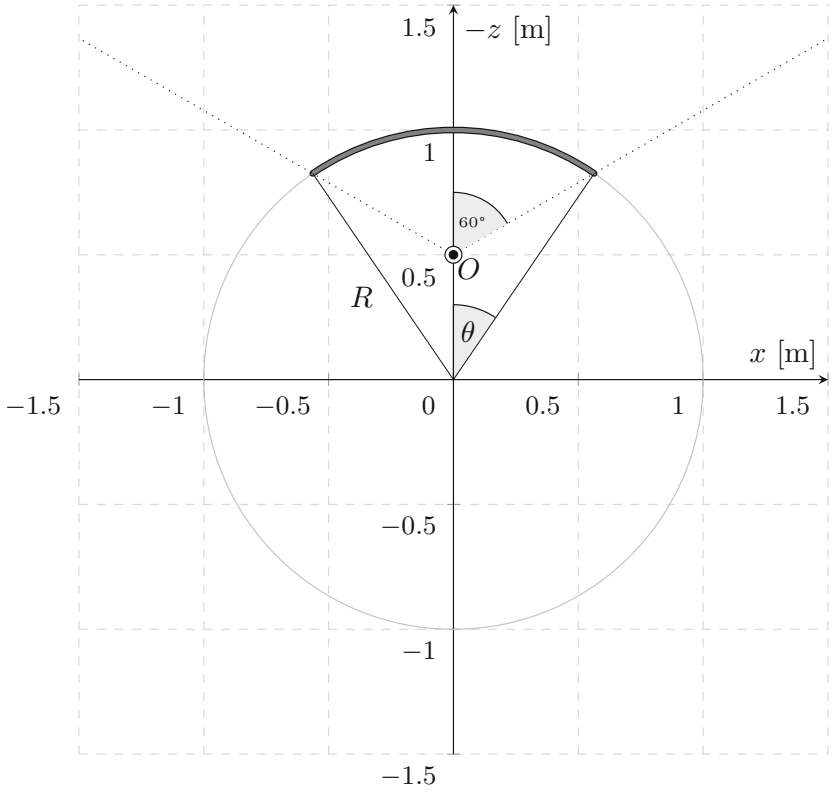


Figure 4.1: A view from $+y$ towards the origin of the coordinate system used in the presented implementations. The circular base of the cylinder is described by the curvature radius R of the monitor. The angle θ is half the angle of the cylinder sector enclosed by the monitor, and the user's viewpoint is represented by the position O.

4.1 Hardware-Accelerated Ray Tracing

Ray tracing allows rendering a wide variety of projections by only modifying how the initial rays are generated [AG93]. The rays are cast from the eye-point into the scene through the display surface. For example, in the case of linear projection, the initial rays are cast through the center of every pixel on a planar screen. The remaining process of ray tracing is not modified at all. Hence, existing ray tracing pipelines can be easily adapted, and the impact of various projections on the performance is expected to be negligible.

To perform a cylindrical projection, as suggested by the geometry of a curved monitor, the initial rays are cast through a pixel grid on the surface of a cylinder. The planar pixel grid of the framebuffer is mapped onto the surface of a cylinder by applying Equation 4.1. The constants H, R, and θ in the equation denote the height of the display, its curvature radius, and half the angle of the cylinder sector that the display forms, respectively. The coordinates u and v describe the center coordinates of a pixel in the framebuffer and have

values in the range of $[-1, 1]$, where the limits correspond to the edges of the image. We divide the y component by two since H represents the entire height of the monitor, and the range of v is relative to the center of the framebuffer. Dividing by two is unneeded for x and z since θ already represents half the angle of the cylinder sector. The additional minus sign for the y component is used for flipping the texture in Vulkan and may not be necessary, depending on the implementation.

$$P_s = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} sin\,(u \cdot \theta) \cdot R \\ -0.5 \cdot v \cdot H \\ -cos\,(u \cdot \theta) \cdot R \end{bmatrix} \tag{4.1}$$

The position P_s is used as the ray origin for the ray tracing process. Using P_s directly as the ray direction of the initial ray is correct for viewpoints in the center of the cylinder only. The observer-dependent direction of the initial ray is calculated by subtracting the viewpoint O from the calculated ray origin P_s and normalizing the result.

To utilize the hardware acceleration on our GPU in Vulkan, we store the entire geometry of the scene in the TLAS and BLAS. We do not need to think about updating these acceleration structures, as our scenes are entirely static. Besides the generation of the initial rays in the ray generation shader, we only utilize the closest hit shader to evaluate the lighting model and the miss shader to color the background. For rendering an anti-aliased image, we perform SSAA by averaging multiple rays per pixel distributed on a regular grid instead of casting only a single ray through the pixel center.

4.2 Image-Based Method

Our implementation of an image-based method is mainly inspired by the works of Watson and Hodges [WH95] and Pohl et al. [PJB13]. Fundamentally, the implementation is divided into two render passes. In the first pass, the scene is rendered with a perspective-correct linear projection. The second pass then warps the generated image into the desired cylindrical projection and displays it on the screen. The warping can be done by either texture mapping the linear projection onto a distorted mesh, referred to as the distortion grid, and then rendering this distortion grid, or by directly displacing every pixel in the image. We implement three variations of the second pass to experiment with the different options.

- The first variant is to displace the vertex coordinates of the distortion grid in the vertex shader.
- For the second variant, we modify the texture coordinates of the distortion grid in the vertex shader and leave the vertex positions in the form of a regular grid.
- The third variant is to displace the texture coordinate in the fragment shader for every texture lookup.

The advantage of the first two variants, where the vertex attributes of the distortion grid are manipulated, is that the cylindrical projection is evaluated for every vertex only. Unfortunately, these attributes are subject to linear interpolation during rasterization, and errors will become visible when the grid resolution is too low. The fragment shader-based solution requires the evaluation of the cylindrical projection for every fragment and does not suffer from the same artifacts. Furthermore, the fragment shader solution allows the implementation of more sophisticated texture filtering not supported by the hardware.

The two main concerns for the first pass are rendering the scene with a perspective-correct projection corresponding to the viewpoint and ensuring that the projected volume contains the required information for the second pass. These two topics are described in the next sections. Thereafter, we describe how the vertex coordinates and texture coordinates for the second pass are calculated. Finally, we present our implementation of cubic texture filtering to improve the sharpness of the warped image as suggested by Pohl et al. [PJB13].

4.2.1 Perspective-Correct Projection

The classical linear projection assumes a flat image plane. Hence, we need to define such a plane onto which the scene is projected. We approximate the cylindrical screen with a plane that is spanned from the left edge of the screen to the right edge such that the entire FOV is covered. The dimensions of the near plane can be calculated based on the surface diagonal D, aspect ratio A, and curvature radius R of the monitor. In our implementation, the user has to provide the diagonal and curvature radius, which are the basic properties describing the size and shape of the monitor. The aspect ratio is calculated from the screen resolution. Note that we assume the pixels of the screen to be quadratic and that the application covers the entire screen. Otherwise, the presented formulas for the dimensions have to be adapted accordingly.

First, we calculate the dimensions of the display surface as if it was flat. The formulas for the width and height, depicted in Equation 4.2, are found by substituting the corresponding side in the Pythagorean theorem with the definition of the aspect ratio $A = W/H$.

$$H = \frac{D}{\sqrt{1 + A^2}}, \; W = \frac{DA}{\sqrt{1 + A^2}} \tag{4.2}$$

The properties of the near plane, which are necessary for the definition of the projection matrix, are now calculated with basic trigonometric functions in our coordinate system. Figure 4.2 shows a view from the positive y-axis towards the center of the cylinder. In this figure, the mentioned properties, which are described in the following paragraphs, are depicted.

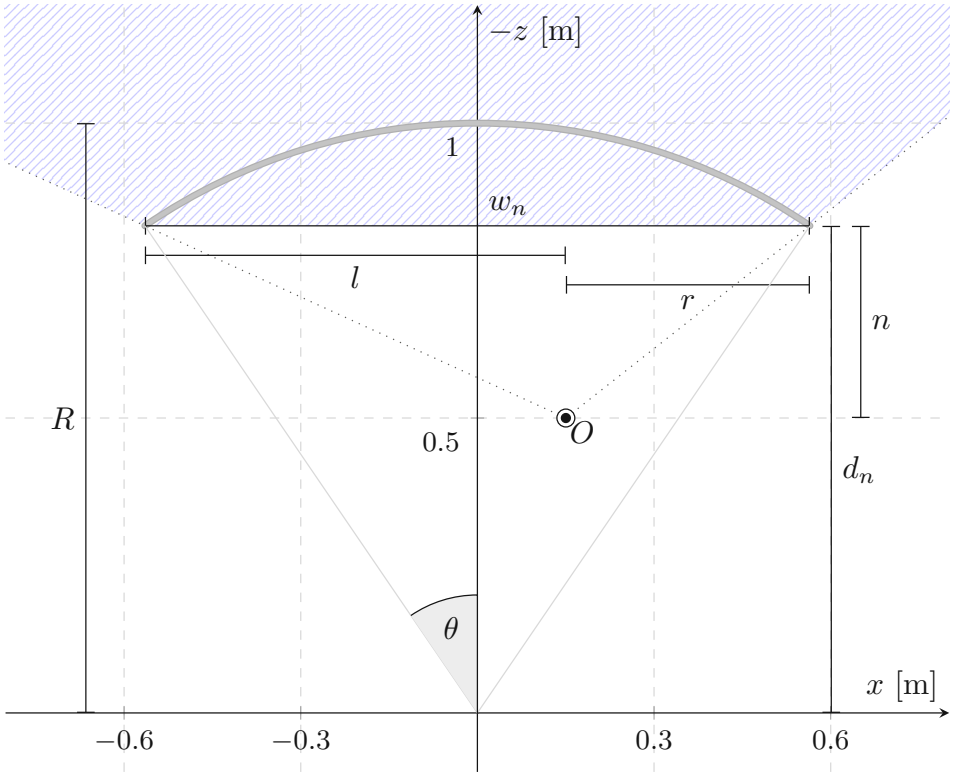


Figure 4.2: The illustration represents a top view of the curved monitor and the variables required for the definition of the projection matrix. The hatched area corresponds to the view volume captured by the linear projection.

From the width of the screen W, which is equivalent to the arc length of the screen, the angle θ can be calculated with Equation 4.3. The width of the near plane w_n and the distance from the origin to the near plane d_n are calculated with the formulas in Equation 4.4.

$$\theta = \frac{W}{2R} \tag{4.3}$$

$$\begin{aligned} w_n &= 2R\sin(\theta) \\ d_n &= R\cos(\theta) \end{aligned} \tag{4.4}$$

The values l, r, b, and t describe the signed extents (left, right, bottom, and top) of the near plane relative to the viewpoint O. While Figure 4.2 does not depict the values b and t, they are calculated analogous to l and r. The values n and f are the distance from

the viewpoint to the near and far planes, respectively. The formulas for these values are listed in Equation 4.5, where F is the depth of the frustum.

$$\begin{aligned} l = -\frac{w_n}{2} - O_x,\ r = \frac{w_n}{2} - O_x, \\ b = -\frac{H}{2} - O_y,\ t = \frac{H}{2} - O_y, \\ n = d_n + O_z,\ f = n + F \end{aligned} \tag{4.5}$$

With all described variables, we finally define our projection matrix P, depicted in Equation 4.6. It is based on the comprehensive description of generalized perspective projection from Kooima [Koo09]. The projection matrix represents an off-axis projection where the center of the projection is generally not aligned with the middle of the image plane. This form is necessary since we want the projection center to correspond to the user's viewpoint, which is not limited to the central axis of the screen. Also, an additional translation matrix based on the observer position is applied. This translation can be interpreted as an extension of the typical view matrix and is necessary for a correct parallax effect. Note that we do not apply a rotation to the projection matrix as presented by Kooima [Koo09] since we approximate the screen with a single plane that is already XY-aligned.

$$P = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-f}{f-n} & \frac{-(fn)}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -O_x \\ 0 & 1 & 0 & -O_y \\ 0 & 0 & 1 & -O_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

4.2.2 Corrected Near Plane Extents

One essential requirement of the linear projection is to cover the user's entire FOV through the curved monitor. Otherwise, information outside the texture that the scene is rendered into may be required during warping. In the case of viewing positions above or below the upper or, respectively, the lower edge of the monitor, a perspective projection based on a near plane connecting the four corner points of the monitor does not cover the entire vertical FOV of the monitor. This problem is depicted in Figure 4.3, where an elevated viewpoint is pictured from the side looking along the $-x$-axis. The frustum enclosed by the four rays from the viewpoint to the corners of the monitor does not cover the entire display surface. Note that only two of these rays are visible in the figure as dotted lines. To resolve this issue and ensure that the required information is rendered, we increase the vertical extents b and t of the near plane such that the entire view volume is captured. Since these bounds are part of the projection matrix, they must be recalculated every time the user's viewpoint changes. For the horizontal bounds, no adaptation is required since the vertical edges of the near plane limit the FOV correctly.

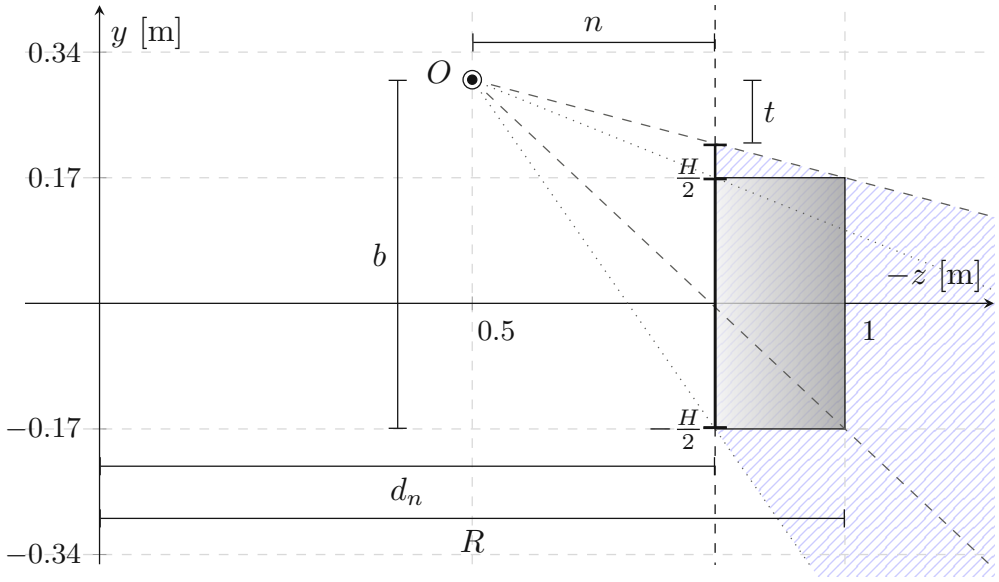


Figure 4.3: The illustration depicts a side view of the monitor with a viewpoint requiring corrected near plane extents for the linear projection. In the depicted example, the near plane is extended vertically to cover the entire view volume and does not correspond to the height of the monitor. Also, note that both values of t and b are negative in this example since they are below the viewpoint. The hatched area represents the view volume captured by the linear projection.

To calculate the corrected extents for the near plane, we cast rays, depicted as dashed lines in Figure 4.3, to the top and bottom of the curved screen at the center of the monitor. The center of the monitor represents the most distant point from the viewpoint on the $-z$-axis. The y-values at the intersections of the rays with the near plane are additional vertical limits to the FOV covered by the monitor. The final vertical extents for the near plane are the union of the extents defined in Equation 4.5, corresponding to the dotted lines in Figure 4.3, and the mentioned additional limits. In the shader code, these extended limits of the near plane also have to be considered during the warping of the texture.

Note that we do not adapt the rendered resolution if the near plane dimensions change. Thereby the image quality may be slightly decreased for viewpoints requiring increased extents as more image information is projected into the same resolution while less of it will be visible on screen after warping. One workaround to this problem is to initially allocate a framebuffer with increased resolution and adjust the resolution of the viewport dynamically. This approach works for a predetermined range of viewing positions only as the near plane extents can become arbitrarily large. Generally, it is reasonable to assume that the near plane extents are close to the initially calculated dimensions and the loss in quality is small since viewpoints far above or below the screen are uncommon.

4.2.3 Calculating Vertex Coordinates

The first variant of the warping pass that we describe adapts the vertex positions of a two-dimensional screen-filling grid, referred to as the distortion grid. The image rendered with a linear projection is then mapped onto this distortion grid, and the grid is rasterized to generate the desired correction.

Initially, the two-dimensional grid resembles the near plane, onto which the scene was rendered in the first pass. The target position of the grid vertices is found by projecting the grid vertices onto the cylinder, which corresponds to the monitor surface. For this projection, we utilize the same cylindrical projection used for the geometry-based methods, which is described in detail in Section 4.3. In short, the key is to cast a ray from the viewpoint through a grid vertex and find the intersection with the cylinder surface. The position of the intersection represents the new grid vertex positions. However, only the x and y coordinates of the intersection are used, and we set the z component to zero to produce an orthographic projection of the cylinder surface. Mapping the rendered image to the generated distortion grid and rasterizing the grid without any further transforms creates the desired correction.

Since the exact vertex positions depend on the viewpoint, we must adapt the vertex positions in the vertex shader for every frame. If the viewpoint is static, the transformation can be applied to the grid in advance, and there is no need for updating the vertex positions in real-time.

4.2.4 Calculating Texture Coordinates

The second variation of the warping pass is to adapt the texture coordinates instead of the vertex positions. In contrast to modifying the vertices, the distortion grid fills the screen precisely, and no geometry is clipped. Consequently, the better utilization of the distortion grid may influence image quality positively compared to the first variant.

The coordinates are found with the inverse function of the vertex coordinate distortion described in Subsection 4.2.3. First, we distribute the vertex coordinates of the grid uniformly on the cylinder surface with Equation 4.1, where u and v correspond to the grid coordinates in the range of $[-1, 1]$. The generated mesh resembles the surface of the physical monitor. Then we intersect rays from the viewpoint to the vertices on the cylinder surface with the near plane. The new texture coordinates correspond to these intersections with the near plane. Finally, the x and y components of the intersections are mapped to the range $[0, 1]$ using the near plane dimensions.

The same calculations are used in the third variant, where we distort the texture coordinates inside the fragment shader. The only difference is that the texture coordinates are corrected on a fragment basis. In this case, the resolution of the grid is irrelevant, and rasterizing a single screen-filling quad is sufficient for the distortion. In other words, this variant is not subject to linear interpolation errors from the rasterizer. The potential downside is an additional computational overhead since the texture coordinate

transformation has to be performed on a fragment basis instead of for usually significantly fewer vertices.

4.2.5 Cubic Texture Interpolation

Pohl et al. [PJB13] show that the loss in image sharpness from the texture resampling can be reduced by using cubic texture filtering instead of a linear one during warping. An extension to the Vulkan API for cubic texture filtering exists [The22]. However, by consulting the *Vulkan Hardware Database* [Wil22], it is apparent that, as of the time of writing, this feature is found mainly in mobile graphics processors rather than desktop GPUs. Therefore, we implement the cubic texture filtering in the fragment shader in software by following the specification of the Vulkan extension. Our unoptimized implementation for cubic texture filtering is listed in Listing 4.1.

Listing 4.1: Cubic Texture interpolation written in OpenGL Shading Language (GLSL)

```
vec4 textureCubic(sampler2D tex, vec2 texCoords) {
    vec2 size = textureSize(tex, 0);
    vec2 unnormalized = texCoords * size;
    vec2 w = fract(unnormalized - 0.5);
    vec2 w2 = w * w;
    vec2 w3 = w2 * w;
    mat4 f = mat4(0, -1, 2, -1,
                  2, 0, -5, 3,
                  0, 1, 4, -3,
                  0, 0, -1, 1);
    vec4 catmulRomWeightsI = 0.5 * vec4(1, w.x, w2.x, w3.x) * f;
    vec4 catmulRomWeightsJ = 0.5 * vec4(1, w.y, w2.y, w3.y) * f;

    vec4 interpolated = vec4(0);
    for(int i = 0; i < 4; i++) {
        for(int j = 0; j < 4; j++) {
            ivec2 intCoords = ivec2(floor(unnormalized - 3.0/2.0))
                + ivec2(i, j);
            vec4 texelData = texelFetch(tex, intCoords, 0);
            interpolated += catmulRomWeightsI[i]
                * catmulRomWeightsJ[j] * texelData;
        }
    }
    return interpolated;
}
```

4.3 Geometry-Based Method Using Tessellation Shaders

Our implementation of the geometry-based distortion correction is based on the work of Pérez et al. [PRO19]. Similarly to performing a linear projection, the fundamental idea is transforming the view volume into a cubic volume that is then fed to the rasterizer. This

transformation, while typically done by applying a projection matrix to every vertex and performing the perspective divide, is done with a custom non-linear projection in the shader. To reduce artifacts due to linear interpolation during rasterization, the authors suggest using tessellation shaders to subdivide the geometry prior to the projection. They present a simple yet effective heuristic for calculating the required tessellation levels. We recognize cases where this heuristic fails to generate satisfactory results and extend it slightly at the cost of higher tessellation. In the following sections, we describe our implementation of the cylindrical projection presented by Pérez et al. [PRO19], our extension to the tessellation heuristic, and our implementation of view culling to discard triangles outside the view volume.

4.3.1 Cylindrical Projection

The cylindrical projection presented by Pérez et al. [PRO19] essentially transforms the cylindrical view volume into a cube and can be summarized in two steps. First, vertices of the scene geometry are projected onto the surface of the display cylinder as illustrated in Figure 4.4, where the vertex V is projected to V_p. Then the projected coordinates are used to map the original vertex into the cubic clip space. Minor adaptations to their presented implementation are described in the following paragraphs to support our setup and improve the results.

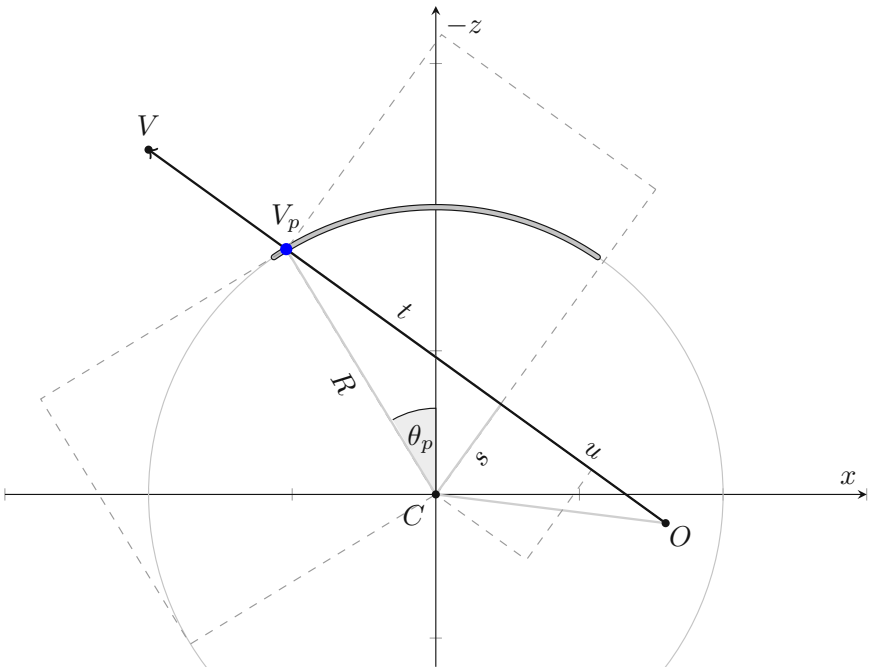


Figure 4.4: The figure depicts a top view of the projection of a vertex V onto the cylinder surface at V_p for the viewpoint O and the respective auxiliary variables involved in the projection.

We do not use the exact formula discussed by Pérez et al. [PRO19] for the projection of vertices onto the circular base of the cylinder since they use a slightly different coordinate system with the x-axis pointing to the left. Instead, we calculate a ray-circle intersection to find the distance d from the viewpoint O to the projected point V_p. This distance from the viewpoint O in the direction of the original vertex V is calculated with the function listed in Listing 4.2. The symbols in the code correspond to the symbols in Figure 4.4. The ray direction `dir` is the normalized vector pointing from O to V, and the circle is described by its center C and radius R. First, the vector $\overline{OC}$ is projected onto the ray direction to get the length of the segment u. In the remaining steps, the squares of the edges R, t, and s of the right triangle are calculated. The distance from the viewpoint to the ray-circle-intersection is provided by $d = t + u$. Note that this function returns the distance in the forward direction only. Further, we assume that the viewpoint is inside the cylinder and that the ray direction is normalized.

Listing 4.2: Ray-Circle Intersection written in GLSL

```
1 float rayCircleIntersect(vec2 O, vec2 dir, vec2 C, float R) {
2     vec2 OC = C - O;
3     float u = dot(OC, dir);
4     float s2 = dot(OC, OC) - u * u;
5     float t2 = R * R - s2;
6
7     float t = sqrt(t2);
8     return u + t;
9 }
```

The final coordinates of V_p are based on the described distance d and are calculated with Equation 4.7. Note that the formula for the y component used by Pérez et al. [PRO19] is based on the slope of the ray $m = (y_p - O_y)/(z_p - O_z)$ along the z direction only. This approach is acceptable as long as $z_p \neq O_z$, but with a FOV greater than 180°, vertices can be precisely to the left or right of the viewpoint. In these cases, the denominator of the slope can get arbitrarily close to zero, and we observe artifacts. Therefore, we calculate the value of y_p together with the other components based on the xz-distance to avoid this division by zero.

$$V_p = \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = d \frac{\begin{bmatrix} x - O_x \\ y - O_y \\ z - O_z \end{bmatrix}}{\left\| \begin{bmatrix} x - O_x \\ z - O_z \end{bmatrix} \right\|} + \begin{bmatrix} O_x \\ O_y \\ O_z \end{bmatrix} \tag{4.7}$$

The next step is to transform the vertices into the cubic clip space used by the hardware rasterizer. The corresponding clip space coordinates are calculated as described by Pérez et al. [PRO19]. The x coordinate of the vertex is transformed based on the angle of its

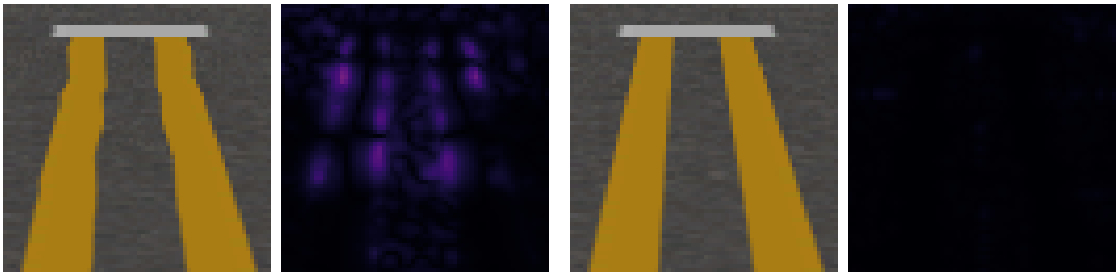
projected point θ_p on the cylinder. For the y coordinate, the vertical position on the cylinder surface is mapped to the height of the display. Finally, the z component is based on the distance from the origin to the vertex in the xz-plane and the depth of the view volume. The formulas for all three clip space components are listed in Equation 4.8. In contrast to the formulas used by Pérez et al. [PRO19], the y coordinate is flipped to account for the screen space coordinate system in Vulkan, and the z component is mapped to the range $[0, 1]$ instead of $[-1, 1]$. The angle used for the calculation of x is obtained with $\theta_p = sign(x_p) \cdot cos^{-1}(-z_p/R)$. For values of z_p close to -1 (e.g., $z_p < -0.9$) we use $\theta_p = sin^{-1}(x_p/R)$ instead to mitigate artifacts in the middle of the screen. These artifacts are presumably due to inaccurate results for values approaching the limits of the domain of cos^{-1}. The symbol H depicts the height of the display, n corresponds to the curvature radius of the monitor, and f is equal to $n + F$, where F is the depth of the view volume.

$$\begin{aligned} x_c &= \frac{\theta_p}{\theta} \\ y_c &= \frac{-2y_p}{H} \\ z_c &= \frac{-sign(z)\sqrt{x^2 + z^2} - n}{f - n} \end{aligned} \tag{4.8}$$

Note that the depicted formulas for the clip space coordinates do not support a FOV greater than 180°. The reason for this is the use of $-sign(z)$ in the calculation of z_c in Equation 4.8. Any vertex with a positive z component produces a value smaller than zero and, therefore, is considered behind the camera and will be clipped in the clipping stage of the graphics pipeline. This also means that the scene will not be rendered correctly when the viewpoint is in front of the plane spanned by the corners of the monitor. It is possible to leave out the $-sign(z)$ when calculating z_c and thereby allow the rendering of viewpoints where a FOV greater than 180° is observed. However, we do not utilize this adaptation in the method using hardware tessellation as artifacts for triangles spanning the discontinuity behind the camera, as discussed by Ardouin et al. [ALMM14], can be the result. An example of the resulting errors is shown in Figure 4.5, where triangles spanning the discontinuity at $\pm\pi$ are rasterized over the entire width of the screen. To ensure correct rasterization of these triangles, they can be split along the discontinuity in an additional clipping stage [ALMM14].

For perspective-correct texture interpolation, the w-component plays an important role [HM91]. Pérez et al. [PRO19] do not mention this subject, and we assume they do not address it. The authors set the fourth component to 1, hiding information necessary for correct texture interpolation from the hardware. Díaz Hernández [DH11] suggests multiplying the unit-cube-position with the distance from the viewpoint to the vertex. We proceed similarly and arrive at our final clip space coordinates V_c by multiplying the homogeneous vector with the distance from the viewpoint to the vertex in the xz-plane as depicted in Equation 4.9.

Figure 4.5: Severe rasterization artifacts in a 360° rendering where triangles spanning the discontinuity at $\pm\pi$ are rasterized over the entire width of the screen.



(a) Incorrect Texture Interpolation (b) Correct Texture Interpolation

Figure 4.6: In the first pair of images (a), the road markings appear warped since the homogeneous coordinate is set to one, and the clip space coordinates are used as is. The ꟻLIP difference map highlights the differences to a reference image. The second pair of images (a) are the result when the clip space coordinates are multiplied by the distance from the viewpoint to the vertex in the xz-plane. The ꟻLIP difference map shows that the error from texture interpolation is reduced significantly.

$$V_c = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \left\| \begin{bmatrix} p_x - O_x \\ p_z - O_z \end{bmatrix} \right\| \tag{4.9}$$

The idea is that a classical projection matrix moves the z coordinate into the fourth coordinate of the result vector. When considering an infinitely narrow frustum pointed at the vertex, then the z-coordinate of a linear projection would correspond to the distance in the xz-plane in our coordinate system. An example of the error when not performing this adaptation is displayed in Figure 4.6. The road markings in the images are distorted when the texture coordinates are not corrected based on the distance.

4.3.2 Extended Tessellation Heuristic

Tessellating the geometry using tessellation shaders is suggested by Pérez et al. [PRO19] to mitigate errors due to linear interpolation during rasterization. For this, the authors present a heuristic to determine the necessary tessellation level. Higher tessellation allows more accurate deformation of the geometry. The idea is similar to approximating a curved surface with one large rigid plane compared to multiple smaller rigid pieces.

Their heuristic identifies the dislocation of the midpoint of an edge in screen space after the cylindrical projection is performed. An example of this dislocation is provided in Figure 4.7. The midpoint m_1 of an edge is projected as described in Subsection 4.3.1 and then transformed into screen space. The resulting coordinate represents the correct position of the midpoint in screen space. When the edge is not subdivided, only the end vertices of the edge are projected correctly. The midpoint of the edge in screen space is then represented by the linearly interpolated midpoint m_2 of the corrected vertices, which does not necessarily correspond to the correct position. The distance in pixels between the two screen space coordinates of the correct midpoint m_1 and the interpolated midpoint m_2 is used as the tessellation level of the edge. For the inner tessellation level of a triangle, the average of the tessellation levels of all edges is suggested.

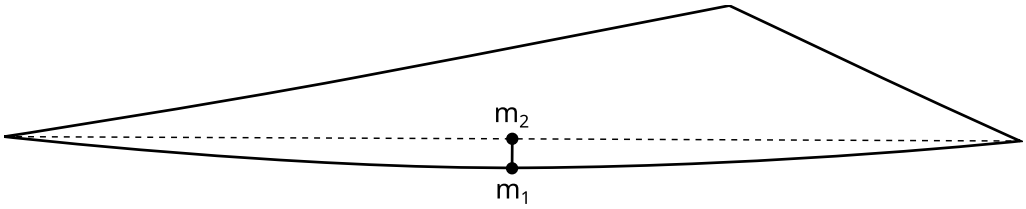
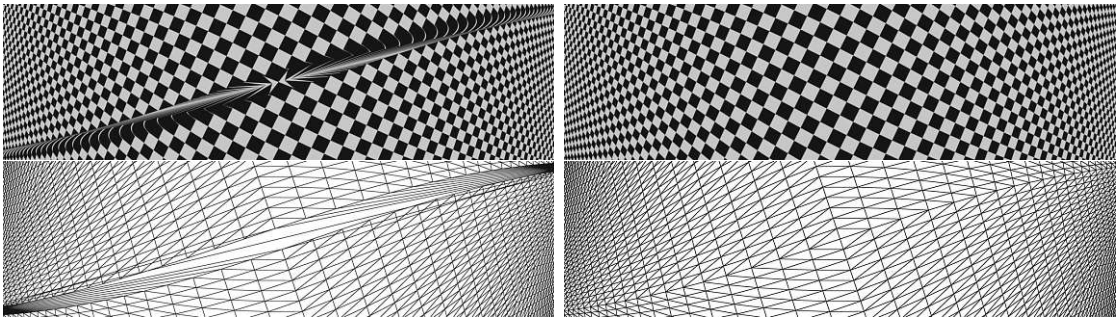


Figure 4.7: The tessellation heuristic from Pérez et al. [PRO19] is based on the distance of the linearly interpolated midpoint of the edge vertices in screen space and the correct midpoint of the distorted edge.

While their heuristic generally produces good results, it yields insufficient tessellation in some cases. One problem is that it is based on a single sample point, namely the midpoint of an edge. An example of the resulting artifacts is depicted in Figure 4.8. In the presented rendering, the diagonal edge of a quad is centered on the optical axis, and the viewpoint is placed at the center of the display cylinder. After the correction, the diagonal edge represents a symmetric S-curve with its inflection point at the center of the screen. The midpoint of the diagonal falls precisely onto the optical axis regardless of whether it is transformed correctly or interpolated from the edge vertices. Therefore, it is determined that no tessellation is necessary at all, and the rendering will contain artifacts. Similar issues are observed for vertical edges since they are not distorted at all and for horizontal edges at the same height as the viewpoint, which resemble straight lines in screen space.

We resolve these issues by calculating the edge length in screen space and increasing the



(a) Heuristic used by Pérez et al. [PRO19] (b) Extended Heuristic

Figure 4.8: The left image (a) shows an example where the tessellation heuristic suggested by Pérez et al. [PRO19] fails to produce satisfactory results together with the wireframe rendering. The midpoint of the diagonal edge projected into screen space is the same point as the linear interpolation of the projected edge vertices. The result with our extension to the heuristic is depicted in the right image (b), where these errors are resolved. Note that the checkerboard was rendered with a curvature radius of 0.5m to emphasize the S-curve form of the diagonal.

tessellation level for every N pixels. A similar solution with an additional vertex for every 10 pixels of an edge is also proposed by Glaeser and Gröller [GG99]. By experimentation, we found a threshold value of $N = 50$ to be sufficient for our implementation. A lower threshold will increase the overall tessellation level considerably. We use the maximum of the edge length heuristic and the original heuristic as the final tessellation level. Additionally, we use the maximum edge tessellation level for the inner tessellation level of a triangle instead of taking the average over all edges. This adaptation results in a higher tessellation of a triangle face when at least one of the edges is determined to require a high tessellation level.

4.3.3 View Culling

A common performance optimization in computer graphics is view culling, where geometry outside the view volume is not processed. Pérez et al. [PRO19] mention view culling in their implementation to discard triangles outside the view before the tessellation is applied. Unfortunately, they do not elaborate on how they implement it. Unsurprisingly, view culling requires additional consideration when performed for non-linear projections.

When using a linear perspective, culling can be performed by testing the vertices of a triangle with every plane of the view frustum separately. If all vertices lie on the outer side of at least one plane that encloses the frustum, then the triangle does not intersect with the frustum. This rejection criterion can also be performed after applying the projection matrix by comparing the values of the xyz-coordinates with the value of the fourth component $\pm w$. A downside of this simple approach is that triangles outside the frustum are not discarded when they cross two planes outside the volume, as depicted

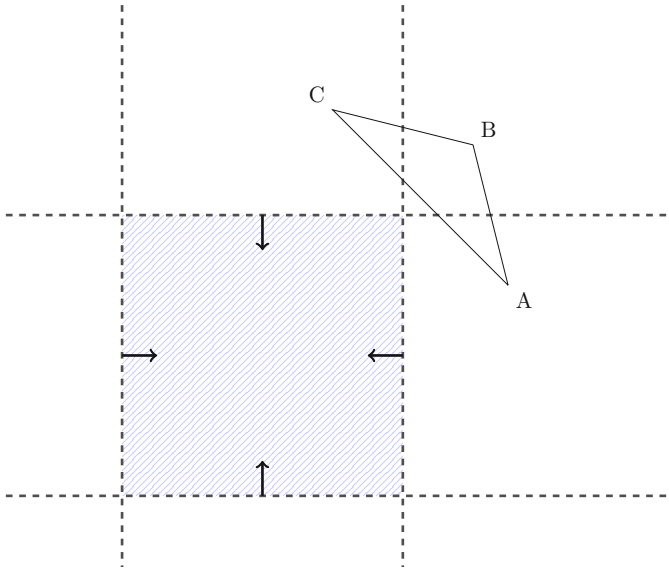
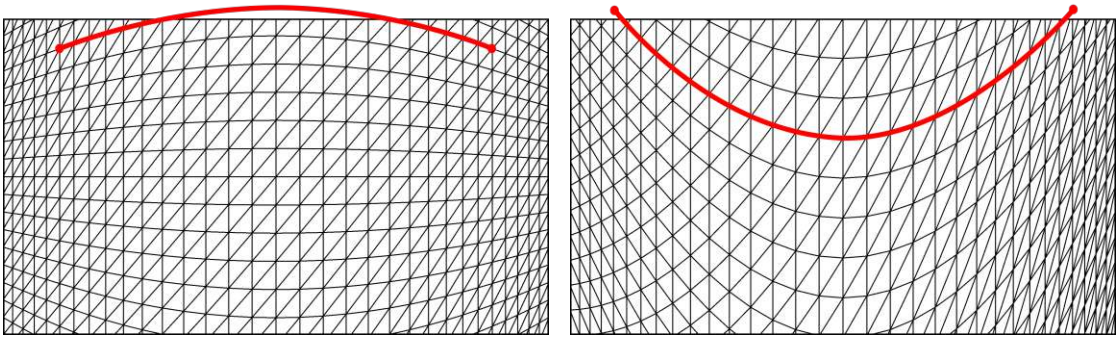


Figure 4.9: The triangle $\overline{ABC}$ will not be rejected when testing the vertices against all four planes separately since they are not consistently on the outside of any plane.

in Figure 4.9. In other words, this solution is correct but not optimal. Optimized and accurate methods exist [AM01], but adapting an optimized approach for the cylindrical projection is outside the scope of this work, and the simple rejection criterion described performs sufficiently well for our use case.

Fundamentally, the simple rejection test also works for the cylindrical projection. However, there are two special cases that we need to handle. Firstly, viewpoints above or below the display surface, equivalent to the viewpoints considered in Subsection 4.2.2, need special handling since, for those viewpoints, triangles outside the clipping volume can bend into the clipping volume after tessellation. Secondly, while only utilized in the method using mesh shaders, the rejection criterion has to be adapted to support a FOV greater than 180°. A workaround for each of these issues is discussed in the following paragraphs.

For viewpoints between the upper and lower edge of the screen, the distortion of straight lines on the screen is convex, i.e., lines bulge outwards, as depicted in Figure 4.10a. In this case, we can use the plane intersection tests in clip space to reject triangles based on the y-component since the y-value of the vertices present valid bounds for the entire edge. In other words, if the vertices of an edge are vertically outside the clip space, the edge is also entirely outside. However, this property of convex edges on the screen is generally not guaranteed for non-linear projections and has to be considered during culling. For the cylindrical projection, horizontal edges generally bulge away from the vertical position of the viewpoint. Edges above the viewpoint bend up, and edges below the viewpoint bend down. As a result, for viewpoints above the top edge of the screen, a triangle edge can bulge into the view volume while the respective endpoints are outside, as depicted in Figure 4.10b. Unfortunately, using the plane intersection



(a) The distortion of straight lines on the screen when corrected for a view point centered in front of the screen.

(b) The distortion of straight lines on the screen when corrected for a view point above the top edge of the screen.

Figure 4.10: The distortion of the geometry on the screen depends on the user's viewpoint and has to be considered when performing view culling. For the cylindrical projection, edges generally bulge away from the vertical position of the viewpoint. With a viewpoint inside the screen bounds (a), all horizontal lines bulge outwards on the screen, and testing the vertices of a triangle is sufficient. A viewpoint above the screen (b), produces edges that bulge into the screen while the vertices are outside the view volume.

test will discard triangles where all vertices are outside the view volume while a portion of it should be visible on the screen. Our workaround for this matter is similar to the solution discussed in Subsection 4.2.2. We extend the screen dimension during culling to the vertical position of the viewpoint, where a horizontal line resembles a straight line in screen space. While this increases the tested volume and includes more triangles than necessary, it also guarantees that the rejection tests do not discard triangles that are visible on the screen. Note that for a FOV smaller than 180° it is also possible to directly use the planes of the extended frustum for the rejection test instead.

The simple plane-based rejection test does not work for the horizontal component when the FOV is greater than 180°. In this case, the infinite planes bounding the volume will intersect with the view volume itself. This self-intersection results in false rejections for triangles on the sides that must be rendered to the screen. While such a wide FOV is not supported by the tessellation shader approach without further modification, it is supported by our mesh shader implementation, described in Section 4.4, where we utilize the same implementation of view culling to discard unnecessary triangles early. To ensure that the culling is correct for a FOV up to 360°, we combine two conditions for the rejection test against the bounding planes on the horizontal axis. The first test is the standard rejection test of calculating the dot product between the normalized vector to the vertex $\hat{p}$ and the normal vector of the plane n. Since the normal vector points towards the inside of the view volume, the vertex is considered outside that plane when $\hat{p} \cdot n < 0$ holds. If the FOV exceeds 180° we not only require that the triangle lies on the outside of one plane but that it is contained entirely in the outer volume. For this requirement, we add the second condition. Let the normalized vector pointing along a plane in the xz

direction be denoted as s. Then $\alpha = s_1 \cdot s_2$ describes the angle of the cone outside the view volume when the FOV exceeds 180°. A vertex is contained in the outside volume when the first condition is true, and additionally, the angle between vectors s_1 and $\hat{p}$ is smaller than α. The final condition for detecting vertices horizontally outside the view volume is $n \cdot \hat{p} < 0 \wedge (fov_x < \pi \vee s_1 \cdot \hat{p} >= s_1 \cdot s_2)$. Note that the complete condition requires lazy evaluation of the *OR*-clause to enable the second condition only if the FOV is greater than 180°. In other words, the second expression of the *OR* clause must not be evaluated when the FOV is smaller than 180°. Again, we reject the triangle if all vertices are outside the view volume.

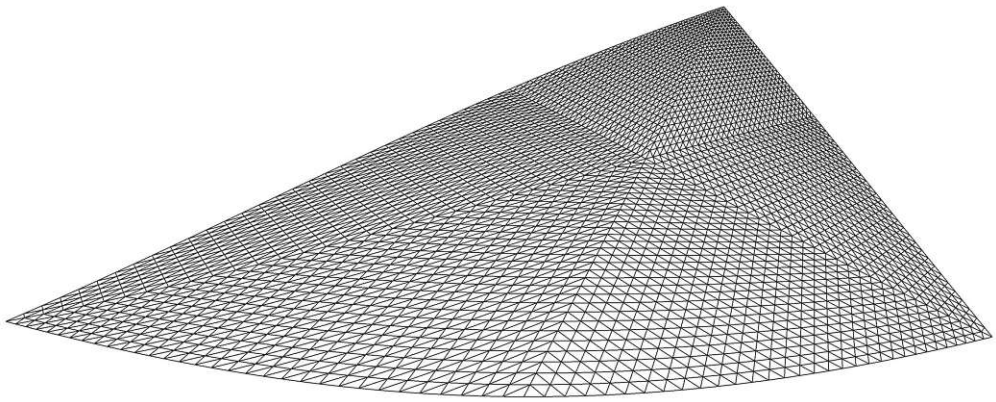
Note that there remain potential issues at the far distance where an edge may intersect with the frustum while the vertices are not contained in the volume. Since we do not observe this case in our scenes with a view volume depth of 1000 units, we choose not to handle this case. However, this problem is easily fixed when necessary by additionally testing the bounding box of the triangle for intersection with the cylinder at the maximum distance and rejecting the triangle only if no intersection is found.

4.4 Geometry-Based Method Using Mesh Shaders

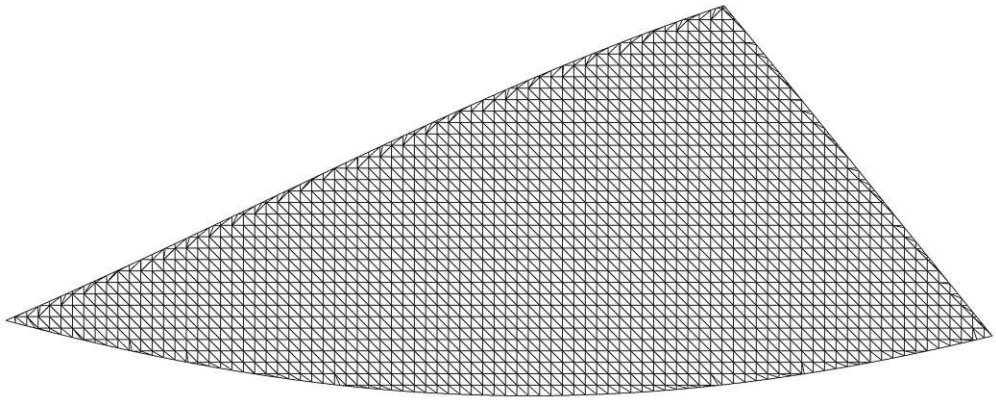
Our geometry-based method is motivated by the observation that the tessellation from the previously discussed geometry-based approach does not produce satisfactory results for thin and large triangles. We observe more tessellation than necessary for elongated thin triangles where the edges have to be subdivided more often than the area of the triangle. Large triangles can reach the hardware limit of the number of subdivisions and therefore achieve poor quality close to the camera. To some degree, these issues may be addressable with better heuristics for the tessellation levels. However, the fundamental tessellation patterns used by the hardware can not be modified to match the requirements better. Similarly, the hardware limit for the maximum number of subdivisions can not be circumvented easily with tessellation shaders. These limitations can only be compensated with a custom tessellation implementation in programmable shaders.

For our implementation in the graphics mesh pipeline, the core idea of distorting the vertices remains the same, but we take a different approach for the tessellation. We divide the view volume along a regular rectangular grid in screen space. The triangles are then subdivided along the edges of this grid, resulting in a uniform subdivision in the final rendering. Figure 4.11 illustrates a triangle tessellated with the hardware tessellation and our custom subdivision scheme for comparison.

This approach has some notable advantages over hardware tessellation. First, the subdivision is not based on the properties of triangles or their edges but on the display surface itself. Triangles covering the same screen area are guaranteed to be subdivided at the same locations in screen space. Hence, the error from linear interpolation during rasterization is more consistent across all triangles and distributed uniformly over the screen. Furthermore, the tessellation naturally becomes less the further away triangles are from the camera resulting in a smooth continuous level of detail. In other words, there



(a) Geometry generated with tessellation shaders



(b) Geometry generated with our grid-based subdivision

Figure 4.11: The images show a triangle tessellated and projected with hardware tessellation (a) and the result with our custom grid-based subdivision scheme (b).

are no discrete changes in how the triangles are subdivided, and the geometry changes fluidly with the camera movement. Finally, large triangles are subdivided only against the grid cells in screen space. Therefore, no computations and memory are wasted for the tessellation of parts of triangles that are outside the view volume.

A fundamental requirement for the subdivision scheme is that the resulting geometry is watertight and artifact-free. Especially "T" junctions, where two edges meet without a shared vertex, must be avoided. These "T" junctions would produce holes in the geometry after the non-linear projection since the vertex at the junction will be projected more accurately compared to the interpolated edge. This issue is depicted in Figure 4.12, where the red quad is subdivided more than the gray one and approximates the desired curvature more closely. The result is a hole between the two quads. A positive aspect of our proposed approach is that no "T" junctions are generated with our subdivision scheme since shared edges between triangles are split along the same grid lines and at

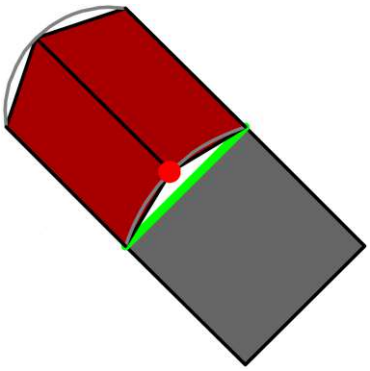


Figure 4.12: The image illustrates a “T” junction between the red and gray quads and shows the resulting crack in the geometry since the red quad approximates the desired surface more closely (adapted from Moreton [Mor01]).

the same locations. It is important to note that this aspect is valid only when the entire geometry of a continuous surface is tessellated with the same grid resolution and no triangles are left out. Surfaces of different objects may still be subdivided with different grid resolutions or other algorithms.

Another consideration is that logically equal vertices have to be represented by the exact same values since otherwise, crack-free rasterization is not guaranteed [Mor01]. Unfortunately, this is a potential downside of our method. Calculating the intersections of a triangle with the planes defining the grid may produce slightly different vertices for neighboring cells due to the finite precision of floating-point numbers. While we do not observe any noticeable artifacts in that regard, crack-free rasterization is not guaranteed by our implementation and is subject to future work. One potential solution to this issue, which we do not implement, is eliminating duplicate vertices at the crossings of grid cells. Furthermore, this enhancement could reduce the computational cost of our implementation since fewer vertices have to be computed and passed to the rasterizer in total.

In the following sections, we discuss the main topics of our implementation in detail. The description is split into the following four central subjects.

- Determining the number of mesh shader workgroups required and spawning the workgroups in the task shader.
- Identifying the triangle and grid cell that is processed by every invocation of a mesh shader workgroup.
- Calculating the bounding box of a triangle in the grid.
- Clipping a triangle against the planes representing the cell boundaries to generate the final triangles in the mesh shader.

4.4.1 Task Shader

The primary purpose of the task shader is to spawn the necessary amount of mesh shader workgroups. Unfortunately, finding the exact amount of mesh shader workgroups required is non-trivial. A triangle may cover any number of cells in the grid, and the final number of generated vertices depends on how the triangle intersects with the grid in screen space. The final intersections with the grid are calculated in the mesh shader only. Therefore, the number of vertices generated is unknown in the task shader. This is a problem since the maximum number of output primitives is defined as a constant in the shader code of the mesh shader [Kub18]. A solution is to estimate the number of workgroups conservatively so that every mesh shader workgroup is guaranteed to have enough space for storing the generated primitives.

Finding a conservative bound is trivial when a single invocation in the mesh shader clips one triangle against one cell in the grid. With this constraint, a single invocation can produce at most seven vertices during the clipping of a triangle against a cell. Four vertices can be generated from the corners of the cell and three from the triangle itself. This fact, combined with the constant buffer size V_{out} for the output vertices, dictates how many invocations of one workgroup can be used for evaluating grid cells. Namely, $I = \lfloor V_{out}/7 \rfloor$ invocations may contribute to the output of a single workgroup and can process one cell in the grid. All other invocations are not allowed to write to the output as it is not guaranteed that there is enough space left.

Knowing the number of invocations that will process cells in a mesh shader workgroup, we can identify the total number of workgroups required. For this, the number of grid cells C_T covered by a triangle T has to be determined. A detailed description for calculating the number of covered cells is provided in Subsection 4.4.3. The number of workgroups $G = \lceil C_T/I \rceil$ is guaranteed to be sufficient for fully processing the triangle.

In Figure 4.13, an example with a single triangle is shown. In this example, we assume that $V_{out} = 49$ and the output buffer, therefore, is guaranteed to be large enough for all vertices generated by seven invocations. The total coverage of the triangle in the grid is $C_T = 24$. As a result, we need $G = 4$ workgroups in the mesh shader stage, each processing seven cells of the triangle in the example. Note that the additional four invocations left over in the fourth workgroup are used to process cells of the next triangle in the geometry.

A meshlet typically consists of more than a single triangle, and a single task shader workgroup can process multiple meshlets. Every invocation in the task shader calculates the bounds of all triangles in the grid of one meshlet. Additionally, the maximum number of cells covered by any triangle in the meshlet is determined. Based on these two numbers, we identify three different cases for each meshlet.

- If the total number of cells covered is zero, then the meshlet does not need to be processed further, and no meshlet workgroup will be spawned. This effectively resembles view culling on a meshlet basis.

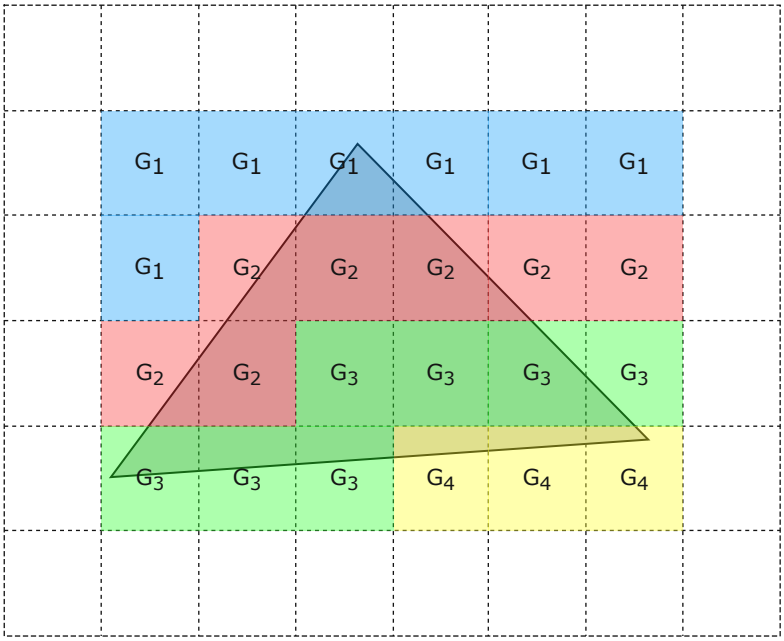


Figure 4.13: The figure illustrates an example of a triangle that has to be processed by four mesh shader workgroups when the size of the output vertex buffer per workgroup is 49. The buffer is guaranteed to fit the generated primitives of 7 invocations using the trivial bound of at most 7 vertices per cell. The four workgroups, represented by different colors, are required to process all 24 grid cells covered by the bounding box of the triangle.

- If the maximum number of cells covered by any triangle of the meshlet is determined to be exactly 1, then no triangle of the meshlet intersects with the grid, and the entire meshlet can be forwarded to the rasterizer without subdivision. In this case, only a single mesh shader workgroup is spawned for copying the data. Note that the triangles do not need to be contained in the same cell, and the total number of cells covered by the meshlet might be greater than 1.
- In all other cases, the conservative bound for the number of generated triangles, based on the total number of cells covered, determines how many workgroups are spawned.

The total number of required workgroups is accumulated over all task shader invocations using `subgroupAdd(...)` and written to `gl_TaskCountNV`. This variable is a built-in variable for the number of workgroups spawned by the pipeline.

Before executing the mesh shader, some additional information about the meshlets processed in the task shader workgroup has to be saved for later access by the related mesh shader workgroups. This information is necessary to determine what meshlet and triangle a mesh shader invocation has to process. We need to pass two arrays with a

length of the workgroup size and two integers to the mesh shader. The first array contains the number of workgroups spawned for every meshlet processed by the task shader so that a mesh shader workgroup can identify which meshlet it is supposed to process. The first integer represents the global index of the first meshlet processed by the task shader. This base index is used to resolve the meshlet indices in the mesh shader, which are relative to the spawning task shader workgroup. The second array contains the relative indices of the meshlets that can be copied without subdividing the geometry. The second integer is the number of meshlets that have to be copied and represents the length of the second array. Further details on how these values are used in the context of the mesh shader are described in Subsection 4.4.2.

4.4.2 Mesh Shader

The mesh shader is responsible for generating the vertices and indices that are sent to the rasterizer. For this, we need to define the size of the output buffers into which the final primitives are written.

Recall that I denotes the number of invocations allowed to process a cell and to write the corresponding triangles to the output. Maximizing the number of the contributing invocations I of a workgroup is favorable as it makes the best use of the available GPU resources. In theory, we only need to increase the output buffer size to provide enough space for all invocations. Nevertheless, by experimentation, we found that the performance decreases for more than $I = 13$ invocations when doing so. This is probably due to a less efficient mapping of the increasingly sized output buffer to the underlying hardware. Another option to increase the ratio of contributing invocations to workgroup size is to decrease the workgroup size. However, we found that this does not improve performance, and it is explicitly advised against using a workgroup size smaller than 32 invocations for NVIDIA's Turing and Ampere architectures [Kub20]. Consequently, our final number of mesh shader invocations performing the clipping of triangles against the cell boundaries is $I = 13$. The remaining 19 invocations are still used for computations regarding work distribution and will not be entirely dormant. Based on the fixed number of invocations $I = 13$, the upper bound for the number of generated vertices is $V_{out} = \max(7I, V_{in})$ where V_{in} is the maximum number of vertices in a meshlet. In the case where seven vertices are generated, five triangles will be created. Therefore, the maximum number of output primitives is $T_{out} = \max(5I, T_{in})$, where T_{in} is the maximum number of primitives in a meshlet. The dependency on the size of the input meshlet is necessary since a workgroup may copy the entire meshlet directly to the rasterizer if it does not intersect with the grid and does not require any subdivision. In this case, the output buffers must still be sufficiently large to fit the input meshlet.

The program logic in the mesh shader is more complex than in the task shader. Therefore a rough overview of the mesh shader is illustrated in Figure 4.14. The steps of this flow diagram are explained in the following paragraphs.

The first step in the mesh shader is to check whether the group has to copy a meshlet.

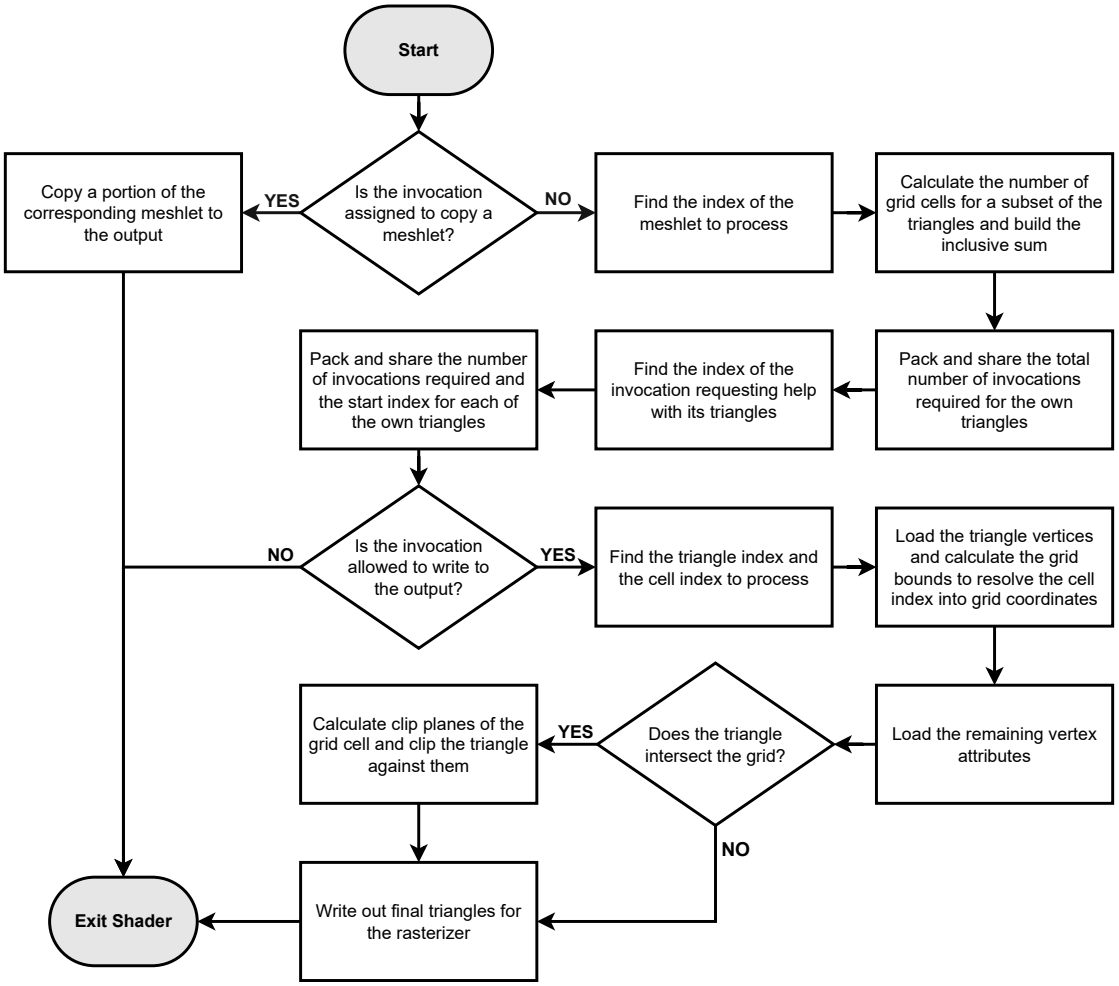


Figure 4.14: Overview of the program flow for a single mesh shader invocation.

Recall that the number of meshlets that have to be copied and their respective identifiers are provided by the task shader. The group index points into the second array passed from the task shader if the group index is smaller than the number of meshlets that must be copied. The respective value in the array identifies the relevant meshlet. Every invocation of the group copies a portion of the meshlet, corresponding to the local invocation index, to the output and exits the shader.

If the group is not responsible for copying a meshlet, the first task is to identify the meshlet the group should process. Unfortunately, the meshlet index is not apparent from the group index itself since it is unknown how many groups have been spawned for other meshlets. The first array from the task shader provides this information and is used to identify the correct meshlet index. The invocations find the correct meshlet index by summing up the values in the array with an exclusive scan summation. Then they vote to find the last invocation regarding the scan operation with a sum smaller than

or equal to the group index. The index of this invocation is the relative index of the meshlet to process. The final meshlet index is acquired by adding the base index from the task shader. Since multiple groups may work on the same meshlet, we also need the index of the workgroup relative to the meshlet to determine the index of the first triangle and cell to process. In Listing 4.3, the described function for finding the meshlet index and the workgroup index relative to the meshlet is presented. Note that the parameter `groupIdx` is the workgroup index reduced by the number of groups reserved for copying.

Listing 4.3: Finding the correct meshlet index in the mesh shader.

```
uint find_meshlet_index(uint groupIdx, out uint meshletRelGroupIdx) {
    uint preceedingGroups = 0;
    uint relativeMshIdx = 0;

    uint groupSum = subgroupExclusiveAdd(
        INPUT.groupsPerMeshlet[gl_LocalInvocationID.x]);
    uvec4 vote = subgroupBallot(groupSum <= groupIdx);
    relativeMshIdx = subgroupBallotFindMSB(vote);
    preceedingGroups = subgroupBroadcast(groupSum, relativeMshIdx);

    meshletRelGroupIdx = groupIdx - preceedingGroups;
    return INPUT.baseIdx + relativeMshIdx;
}
```

With the correct meshlet identified, every invocation of the workgroup calculates the grid bounds for a portion of triangles of the meshlet. This step essentially repeats the work of the task shader but avoids sharing the data for all triangles between the two shader stages. The number of grid cells per triangle and the relative index of the workgroup to the meshlet are necessary to find the triangle and the grid cell that every invocation has to process.

The number of cells every triangle covers is accumulated over all invocations with an inclusive scan summation. With the inclusive sum and the relative workgroup index to the meshlet, every invocation can calculate how many cells of the own triangles are processed by other workgroups. Thereby, the invocations collectively know how many invocations of the current workgroup have to help in processing each triangle. An example of this approach is illustrated in Figure 4.15 for one meshlet. In the example, every invocation calculates the bounds for three triangles of the meshlet. The grey area represents the amount of work processed by other mesh shader workgroups with a smaller index relative to the current meshlet. The triangles $T1$ to $T4$ are fully processed, and $T5$ is a partially finished triangle with 12 cells left to process. In the example, *Invocation 2* recognizes the fifth cell of $T5$ as the first cell to process based on the inclusive sum and the number of cells processed by other groups. This information must be shared with all other invocations so they can identify their own cell to process.

To share the necessary data about each triangle, we pack it into unsigned integers and make them available to the other invocations with subgroup operations. To do so

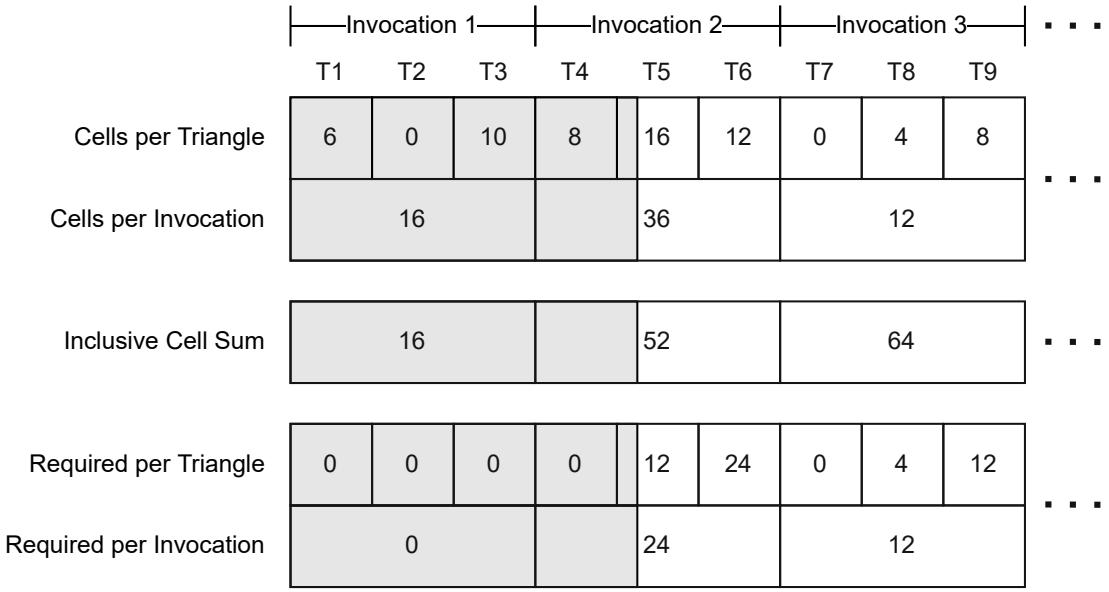


Figure 4.15: Every mesh shader invocation calculates the cells covered by the assigned triangles and shares the information on how many invocations are required with the other invocations in the workgroup. The gray area depicts the cells processed by mesh shader workgroups with a lower index. The last two rows represent the number of required invocations per triangle and per invocation. Since the number of required invocations is limited to the workgroup size, it can be represented by only a few bits. These numbers are packed into unsigned integers and shared efficiently using subgroup operations.

efficiently, we need to reduce the range of the shared values to fit into a few bits. Every invocation can request help from all the invocations in the workgroup at most. Our workgroups have a size of 32 invocations. Therefore, 6 bits for every triangle would be enough to specify the 33 values in the range of $[0, 32]$ for specifying the number of required invocations. However, we choose to use only 5 bits. Every invocation processes at most six triangles in our implementation, and using 5 bits per triangle allows us to pack the values of all triangles into a single 32-bit unsigned integer. Using one bit less is not a problem as only $I = 13$ invocations are allowed to write out triangles in the end, as discussed earlier. A single unsigned integer now contains the number of cells left for each of the six triangles. Unfortunately, this information is not accessible to all other invocations right away. Every invocation must first identify the invocation holding the information about the triangle that must be processed next to access the correct integer.

In our example, in Figure 4.15, both *Invocation 1* and *Invocation 3* do not know the first triangle and need this information from *Invocation 2*. In other words, *Invocation 2* requires help from the other invocations. The last row in the figure depicts this information as the first non-zero segment. Again, we pack the amount of help required by every invocation in total into a few integers and share it with all invocations. Each of

the 32 invocations writes at most 5 bits to specify the total help required. Therefore, 32 invocations need 160 bits, or five unsigned integers, to contain the information. The function `subgroupOr(...)` that is used to combine and simultaneously share the values between the invocations supports the `uvec3` type. Hence, we only need two subgroup operations, each operating on a `uvec3`, to share the data. Every invocation then scans the five integers and searches for the first invocation requesting help.

The index of the requesting invocation is used to acquire the previously packed information about the individual triangles. We use `subgroupShuffle(uint data, uint index)` to access the data of a specific invocation, where `data` is the packed triangle information, and the `index` corresponds to the invocation requesting help. Again, the packed data is scanned for the first unfinished triangle. To find the final index of the first cell that must be processed, we need additional information on how many cells of the first unfinished triangle are processed by other workgroups. For this, we use a second `subgroupShuffle(...)` call to the same invocation, which can provide this information. With the gathered data, each invocation can identify the triangle and the exact grid cell it has to process.

Any invocation that is not allowed to write to the output buffer since their index is too large or when no cell is left to process exits the mesh shader now. All remaining invocations get the vertices of their triangle and calculate the grid bounds of this triangle one last time. The bounds are required since the grid cells have been identified with a linear index up to this point. With the grid bounds, the cell index i is translated into coordinates in the 2D grid with Equation 4.10. Additionally, the exact bounds of the target triangle provide information on whether it has to be clipped in the first place. If the triangle does not intersect with the boundaries of a cell, then the triangle can be copied without calculating the clipping planes and executing the clipping algorithm. Otherwise, the clipping algorithm is executed, and the resulting vertices and triangles are passed to the rasterizer.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_{min} \\ y_{min} \end{bmatrix} + \begin{bmatrix} i \bmod (x_{max} - x_{min})) \\ \lfloor i/(x_{max} - x_{min}) \rfloor \end{bmatrix} \tag{4.10}$$

4.4.3 Triangle Grid Bounds

Calculating the bounding box for every triangle in the grid, also referred to as the grid bounds in this work, is an essential step for achieving acceptable performance. Otherwise, the clipping process must be performed for every grid cell, even when a triangle is contained inside a single cell. A similar optimization is done by Lorenz and Döllner in their PPP method [LD09] as well as by Gascuel et al. [GHFP08]. While the first impression might suggest similarities to PPP, our implementation has more in common with the solution of Gascuel et al., where the triangles are bound in screen space after they are distorted. This approach is used since our goal is to achieve a uniform and regular grid in screen space after the distortion, as shown in Figure 4.16a. The idea behind

this decision is to distribute the generated errors more evenly over the display surface. Lorenz and Döllner assume a uniform grid when determining the triangle coverage. While this allows for a straightforward coverage determination by only considering the triangle coordinates after a linear projection, the grid cells do not have a uniform size and shape after the view deformation, as shown in Figure 4.16b. Using the same approach would subdivide the geometry less near the center of the image, where it is magnified compared to the linear projection, and more on the sides.

Similar to Gascuel et al. [GHFP08], we calculate and combine the bounds of each edge of a triangle to arrive at the final bounds in screen space. The exact bounds are then reduced to the minimum and maximum coordinates in the screen space grid. Additionally, we perform view culling as described in Subsection 4.3.3 prior to calculating the bounds to avoid unnecessary calculations for triangles outside the view.

The horizontal bounds are trivial to determine. Vertical lines remain straight in the cylindrical projection, and the bounds of the projected triangle vertices can be directly used for the horizontal grid bounds. Finding the vertical screen space bounds is more involved. An edge $\overline{P_1P_2}$ of a triangle projected onto the cylindrical surface resembles a segment of an ellipse, as illustrated in Figure 4.17. This ellipse is found by intersecting the plane spanned by the two edge endpoints and the viewpoint O with the cylinder surface. Based on this ellipse, we calculate the vertical bounds.

The standard plane equation $ax + by + cz - d = 0$ is used to describe the plane spanned by the three points. The values of a, b, and c correspond the components of the normal vector obtained from the normalized cross product of $\overline{OP_1}$ and $\overline{OP_2}$. The value of d is then calculated with the dot product of the plane normal and O. Two of the three remaining unknowns in the plane equation are provided by the circular base of the display, which is described with Equation 4.11 in our coordinate system.

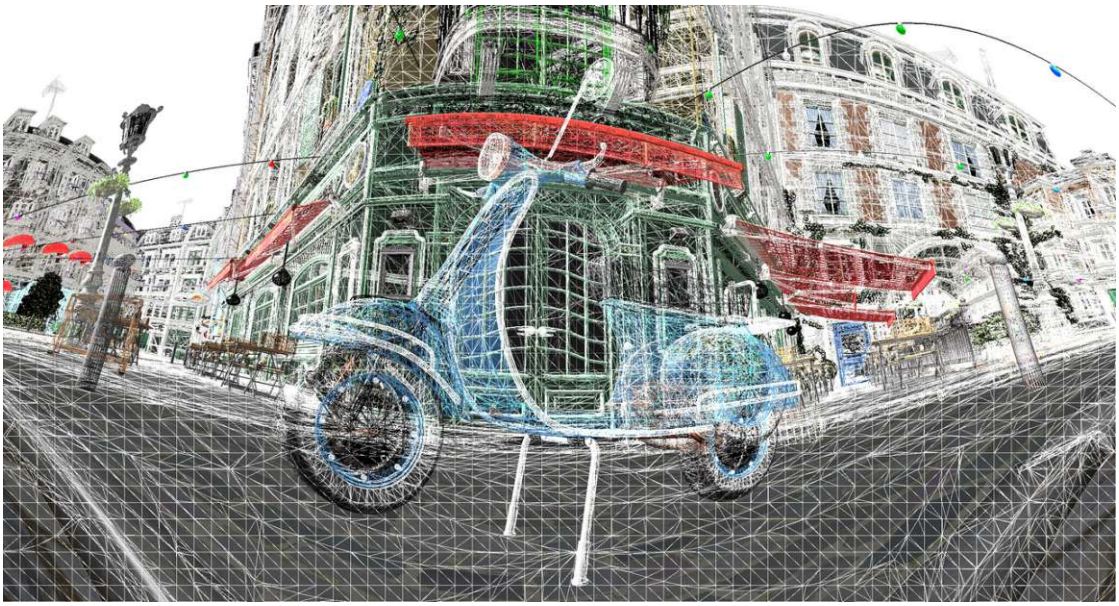
$$\begin{bmatrix} x \\ z \end{bmatrix} = \begin{bmatrix} R\,sin(t) \\ -R\,cos(t) \end{bmatrix} \tag{4.11}$$

Substituting the equation of the circular base into the plane equation yields the formulation of the y component of the ellipse. Based on this formula, depicted in Equation 4.12, we analyze the ellipse for extrema of the y component.

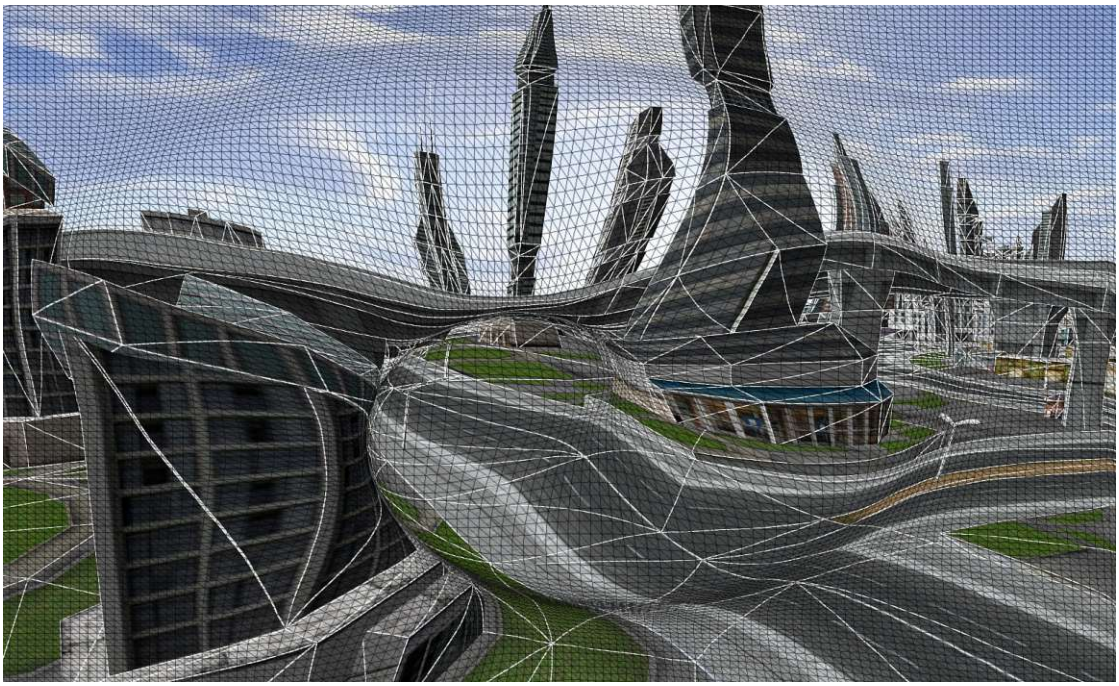
$$y = \frac{c\,R\,cos(t) - a\,R\,sin(t) + d}{b} \tag{4.12}$$

The extrema are found using standard analysis methods. First, the function is differentiated regarding the parameter t—the angle on the circular base—as shown in Equation 4.13.

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \frac{-c\,R\,sin(t) - a\,R\,cos(t)}{b} \tag{4.13}$$



(a) Rendering with superimposed wireframe using our grid-based subdivision



(b) Rendering with superimposed borders of the projection pieces using PPP (reprint from [LD09])

Figure 4.16: Our grid-based subdivision produces uniformly sized grid cells after the correction regardless of the distortion (a). The individual cells in PPP have varying sizes after the view deformation is performed (b).

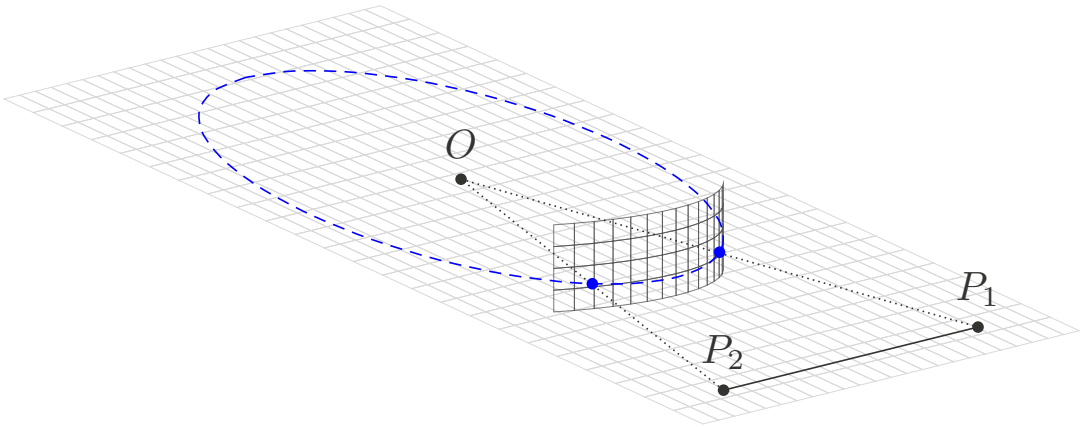


Figure 4.17: An edge $\overline{P_1P_2}$ projected onto the display surface resembles a segment of an ellipse. This ellipse is the intersection of the plane, formed by the points O, P_1, and P_2, with the cylinder of the monitor.

By setting the derivative equal to zero and rearranging the formula as shown in Equation 4.14, we arrive at the formula describing the parameter t at the critical points. The highest and lowest points on the ellipse are found at the resulting positions of t. The exact values of these extrema are calculated by inserting the values of t into Equation 4.12

$$\begin{aligned} & c\,sin(t) + a\,cos(t) = 0 \\ \Longrightarrow\ & \frac{c\,sin(t)}{a\,cos(t)} = -1 \\ \Longrightarrow\ & \frac{c}{a}\,tan(t) = -1 \\ \Longrightarrow\ & t = n\pi + atan(-\frac{a}{c}) \qquad n \in \mathbb{Z} \end{aligned} \tag{4.14}$$

It is important to note that the value of t found may describe a position on the ellipse outside the projected edge. Therefore, we clamp t to the angular values of the projected endpoints t_1 and t_2 corresponding to P_1 and P_2, respectively. Either an extremum is between the two endpoints and not clamped, or the y-value of the arc between the endpoints is monotonically de- or increasing. In the second case, the projected y values of the vertices correspond to the vertical minima and maxima in screen space. Therefore, clamping the value does not remove the extrema regarding the edge. However, there is an exception for edges that cross the discontinuity at $\pm\pi$ since clamping the values requires that the edges exist in the interval of $[-\theta, \theta]$. This issue is addressed together with other special cases in the following paragraphs. A final note regarding Equation 4.14 is that the division by zero must be considered. If the value of c, corresponding to the z-component of the plane normal, is zero, then the extrema lie on $t = \pm\pi/2$.

Figure 4.18: The image shows the geometric errors resulting for a low horizontal grid resolution of 2×50 when the bounds are not corrected.

Some special cases, similar to the ones found by Lorenz and Döllner for their cylindrical projection [LD09], must be handled separately. First, all triangle edges are tested for intersection with the planes limiting the view horizontally. If an intersection is detected, the horizontal bounds are set to the minimum or maximum bound, respectively. As a result, triangles wrapping around the cylinder—extending out from the left side of the view volume and entering the right side—are bound by the entire width of the grid. While this may introduce large regions that do not contain the actual triangle, we select this solution as it is reasonable to assume that the number of such triangles is small. More exact boundaries can be calculated by splitting the triangle and separately determining bounds. Extrema outside the view do not need to be considered, and we clamp the angle t of the vertices, used for determining the vertical bounds, to the horizontal FOV. Finally, we consider triangles intersecting with the cylinder axis to cover the entire grid, similar to Lorenz and Döllner [LD09]. This convention ensures the correct rendering of triangles that surround the user.

The discussed calculation for the bounds is accurate for high grid resolutions. However, we noticed severe errors when using a low grid resolution on the horizontal axis. The resulting artifacts are apparent as holes in the geometry visible in Figure 4.18, where a grid resolution of 2×50 is used. The underlying problem is that we assume the grid cells to be rectangular in screen space after projection, which is not necessarily the case. Any edge in the geometry is subject to distortion, and so are the edges introduced by our subdivision scheme. The result is that the geometry generated on the border of the grid cells bends in screen space, and the bounds calculated will not match the actual position and shape of the cells. Note that the artifacts are not visible for high grid resolutions since the grid approximates the display geometry and approaches a cylinder surface in the limit.

To solve this problem, we calculate distorted bounds by approximating the screen with planes according to the horizontal resolution of the grid. Then we intersect rays from the viewpoint to the calculated extrema and the edge vertices with these planes. The vertical component of the intersections with the planar approximation corresponds to the actual location in the grid. This method is essentially identical to calculating the

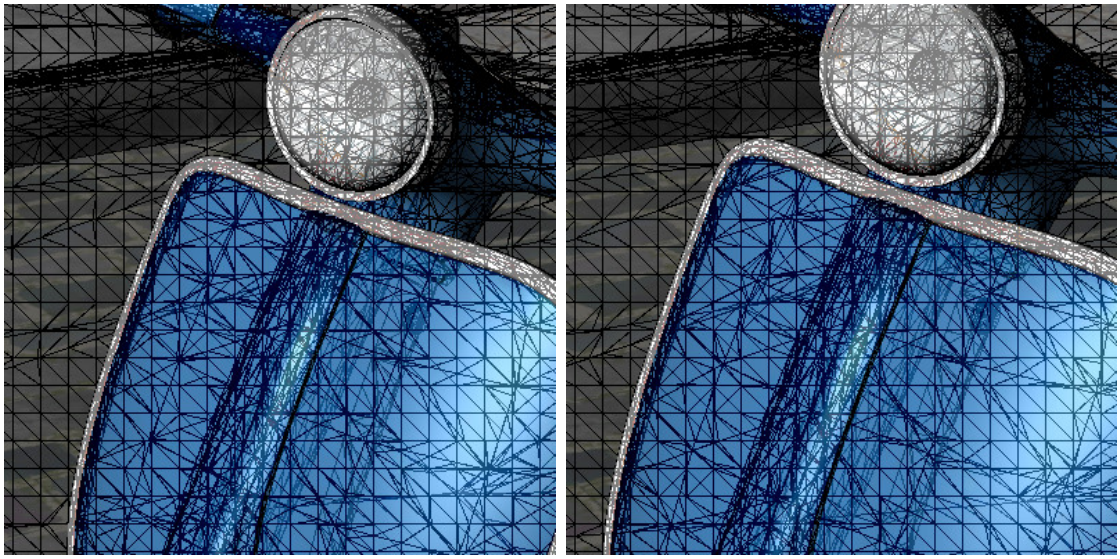


Figure 4.19: The images show the changes in the triangulation of our subdivision scheme for two slightly different camera positions.

distorted texture coordinates in Subsection 4.2.4, where the image plane of the linear projection corresponds to one planar approximation of the curved screen. The union of the corrected bounds with the bounds of the vertices in screen space are the final artifact-free grid bounds.

4.4.4 Clipping

The triangles are clipped against the planes of every relevant cell in the mesh shader. One invocation of a mesh shader workgroup processes one cell in the grid for a single triangle. During clipping, edges intersecting with the cell boundaries are split, and new vertices are introduced. Note that all vertex attributes have to be interpolated accordingly in this step. The generated triangles are different for every camera position since the grid is fixed to the camera. The resulting triangulations of the geometry for two slightly different camera positions are depicted in Figure 4.19.

We use the Sutherland–Hodgman algorithm—named after the inventors—for the clipping process. The algorithm is described in the publication “Reentrant polygon clipping” from 1974 by Sutherland and Hodgman [SH74]. A critical property of the algorithm for our use case is that the order of the original vertices remains unchanged. Thereby the face direction is preserved. Also, this property makes it trivial to generate the index list of the clipped triangle by connecting two consecutive vertices in the output list with the first one. Another essential property of the algorithm for our use case is that it poses no restrictions on the clipping planes defining the cell boundaries. This property is vital as the planes defining a cell boundary represent an oblique frustum in 3D space. The lines superimposed onto the display surface in Figure 4.20 correspond to the 2D section of the

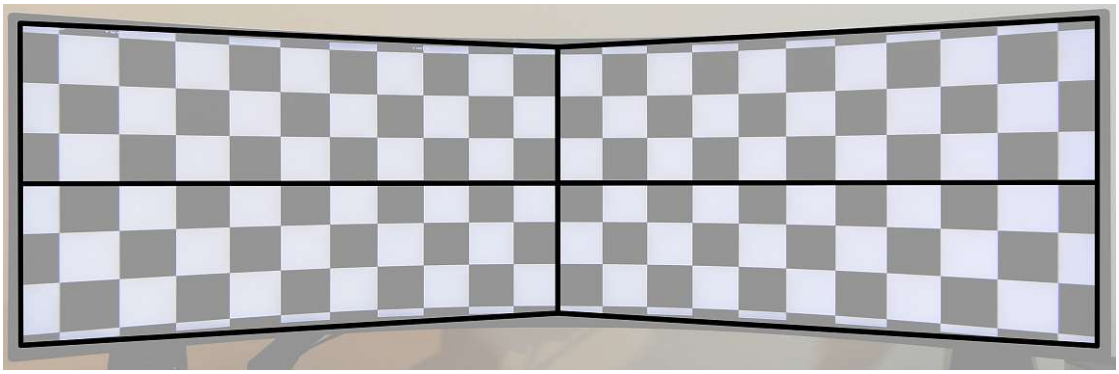


Figure 4.20: The lines superimposed onto the image represent the 2D section of the cell boundaries with a 2-by-2 grid resolution. In general, the planes corresponding to the cell boundaries represent oblique frustums.

frustum planes using a 2-by-2 grid resolution. The resulting grid cells approximate the curved screen, and the top and lower bounding planes are generally not parallel. After applying the cylindrical projection, the boundaries of the cells become rectangular in screen space.

The planes bounding a cell can be constructed from the coordinates of the cell in the grid, the monitor dimensions, and the viewpoint. The vertical planes correspond to straight vertical lines, as visible in Figure 4.20. This observation suggests that they only depend on the horizontal coordinate in screen space. For the normal of these planes, we calculate the vector from the viewpoint to the cylinder surface at the corresponding position in the xz-plane and take the cross product with the up vector in view space. The distance to the origin is calculated with the dot product of the normal and the viewpoint position. On the other hand, the images of the horizontal planes represent piecewise approximations of arcs and therefore depend on the horizontal and vertical coordinates in the grid. For the normal vector of a horizontal plane, we calculate the two vectors from the viewpoint to the points on the cylinder where the horizontal plane intersects with the adjacent vertical planes and take the cross product of these vectors. The distance to the origin is calculated analogously to the vertical planes.

4.5 Head Tracking

Tracking the user's head allows updating the viewpoint to the correct position for rendering. Thereby, the correct perspective is always rendered for the user. This concept is also known as FTVR [WAB93] and, for example, allows looking around corners by moving the head.

We experiment with two different readily available head-tracking solutions to implement an elementary FTVR experience. The first variant is webcam-based and can therefore

be used without special hardware. The second solution utilizes the consumer-grade eye tracker *Tobii Eye-Tracker 5* [Tob].

For the webcam-based implementation, we utilize the open source project *opentrack* [HTB+]. The software translates the output of various head-tracking solutions into a different format for various applications. In our implementation, we use the output as a User Datagram Protocol (UDP) stream that we directly feed into our application. As an input source, *opentrack* supports multiple tracking solutions, including webcam-based tracking using neural networks.

For the integration of the *Tobii Eye-Tracker 5*, we directly interface with the proprietary API of the device. While the eye tracker can identify the exact position of the user's eyes and the corresponding view directions, we only utilize the head-tracking capabilities. The main advantage of the head-tracking data is that a head pose is also provided when the eyes are partially occluded, resulting in more stable tracking.

The data of the selected tracking source is captured in a secondary thread and concurrent with the rendering. In every frame, the latest available head-tracking data is fetched and used during rendering. A constant user-specified viewpoint offset can be added to the position. This offset is used to adapt the viewpoint when no head tracking is utilized and for manually correcting small offsets from the tracking source.

The tracked viewpoint position is used to build the perspective-correct projection matrix as discussed in Subsection 4.2.2. Then the projection matrix, together with the viewpoint position, is provided to the shaders using a standard UBO and used in the methods to perform the perspective-correct rendering, as discussed in the prior sections.

CHAPTER 5

Results & Comparison

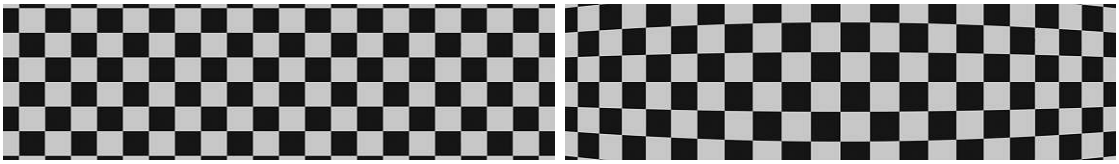
In this chapter, we present our measurements and discuss our findings in detail. The first section demonstrates the effects of the corrected rendering compared to the traditional linear perspective and illustrates that our implementation is accurate for various viewpoints. The following sections present details about the various parameters of our implemented methods and their influence on performance and quality. Analyzing the different methods in isolation allows us to identify sensible parameters for the final comparison. Finally, we directly compare the methods, discuss their visual quality, and provide a short assessment of the head-tracking experience.

Recall that all quality measurements are done with SSAA using 8 SPP to reduce aliasing in the images, as described in Section 3.3. The ꟻLIP error is produced by comparing the images with the reference images generated using our grid-based method with a grid resolution of 2560×720 if not stated otherwise. The performance is measured without any SSAA to avoid biasing the results and to measure a reasonable workload. In the case of ray tracing, we use 9 SPP on a uniform grid for anti-aliasing and 1 SPP through the center of the pixel for the performance evaluation.

To present the values of our measurements, we mainly use box plots to consolidate all sample points of a benchmark. These plots convey a general impression of the spread of the respective metric over the benchmark.

5.1 Accuracy of Rendered Results

In Figure 5.1, the rendered results of a checkerboard with linear and cylindrical projection for a monitor with a curvature radius of one meter, a 49-inch diagonal, and an aspect ratio of 32:9 are depicted. The viewpoint used for rendering is located in the center of the cylinder and conforms to a viewing distance of 100cm from the monitor. In the case of the linear projection, the checkerboard remains a regular checkerboard since the



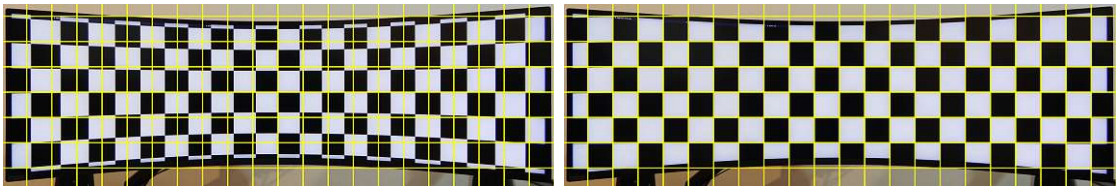
(a) Linear Projection (b) Cylindrical Projection

Figure 5.1: The images show the results of a checkerboard rendered with a linear (a) and a corrected projection (b).

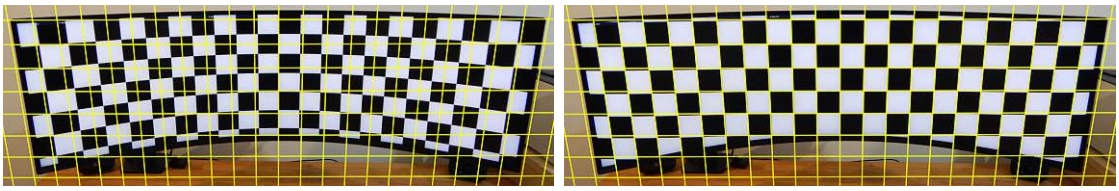
image plane is parallel to the checkerboard plane. The corrected rendering is distorted according to the geometry of the monitor and the user's viewpoint. Horizontal lines are curved, and the vertical lines are distributed non-linearly over the horizontal axis to counteract the curvature. When viewed on the curved monitor with the corresponding viewing distance, the corrected rendering will resemble a rectangular checkerboard, while the traditional rendering appears warped.

Displaying the rendered results on the physical monitor produces the expected results. The corresponding measurements are depicted in Figure 5.2. The screen is captured with a calibrated camera from six different viewpoints. The matching viewpoint positions for the camera are configured in the application to generate a perspective-correct rendering of the checkerboard. We render an image using a traditional perspective-correct linear projection and one corrected for the curved screen for each viewpoint. A rectangular grid fitted to the perspective of the physical camera is superimposed onto every pair of images to highlight the differences between the corrected and linear projection. In the presented images, it is apparent that the corrected rendering recreates the impression of a linear projection of the checkerboard pattern significantly better than using a linear projection. In other words, the corrected projection counteracts the curvature of the monitor, and a user perceives a linear projection when the viewpoint used for rendering is configured correctly.

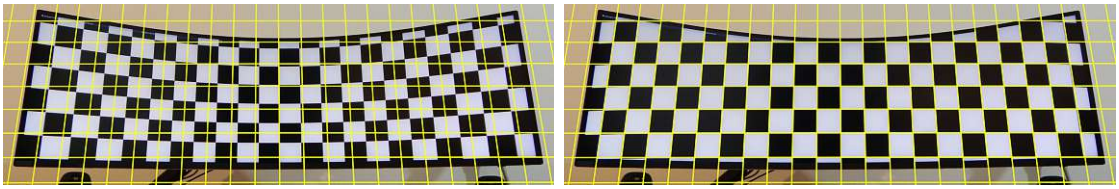
In the case of a more realistic scene, as depicted in Figure 5.3, the improvement of the perspective is less evident in the pictures but still noticeable. We contribute this to the fact that continuous straight lines spanning the entire view are rare in most scenes. Nevertheless, the corrected result improves the impression of a scene by reducing distortions considerably. For example, the facade of the restaurant and the signs appear bent in the linear projection. Furthermore, the motorcycle and light bulbs on the right side of the image appear horizontally stretched due to the perspective distortion in the linear projection. Especially during camera movements, the distortions from incorrect rendering are noticeable as geometry is stretched in the periphery of the vision and contracted in the center of the vision. Considering the improvement of the renderings, we are convinced that the rendering of 3D scenes should be corrected in general when comparable monitors become more common.



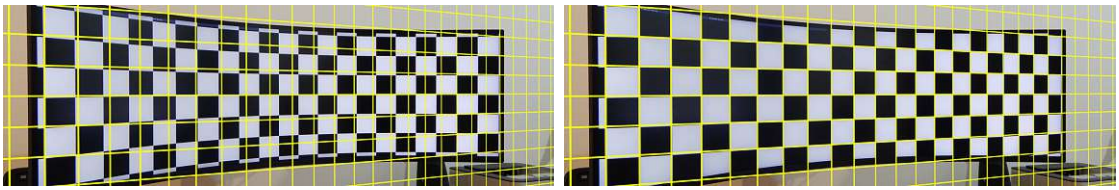
(a) Linear Projection, Viewpoint $(0, 0, 0)$ (b) Corrected Projection, Viewpoint $(0, 0, 0)$



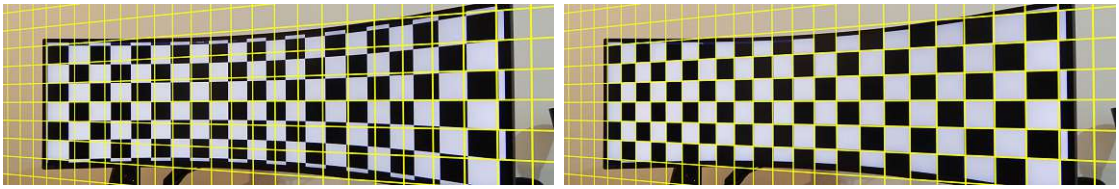
(c) Linear Projection, Viewpoint $(0, 0.25, 0)$ (d) Corrected Projection, Viewpoint $(0, 0.25, 0)$



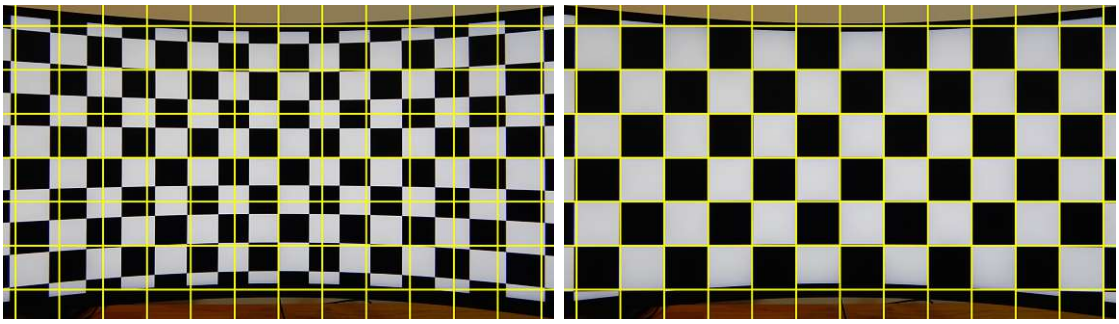
(e) Linear Projection, Viewpoint $(0, -0.25, 0)$ (f) Corrected Projection, Viewpoint $(0, -0.25, 0)$



(g) Linear Projection, Viewpoint $(-0.5, 0, 0)$ (h) Corrected Projection, Viewpoint $(-0.5, 0, 0)$



(i) Linear Projection, Viewpoint $(0.5, 0, 0)$ (j) Corrected Projection, Viewpoint $(0.5, 0, 0)$

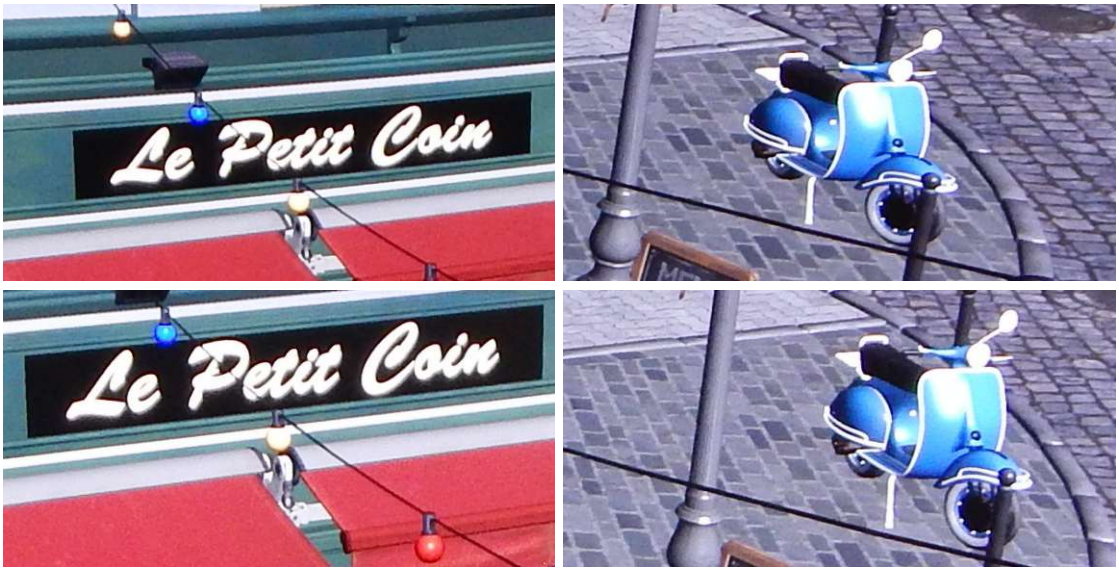


(k) Linear Projection, Viewpoint $(0, 0, -0.5)$ (l) Corrected Projection, Viewpoint $(0, 0, -0.5)$

Figure 5.2: The photos show the results of the linear projection (left) and the correct rendering (right) of a checkerboard on the physical monitor for six different viewpoints. A rectangular grid transformed according to the perspective is superimposed and matches the cylindrical projection closely.

(a) Linear Projection, Viewpoint (0, 0, 0)

(b) Corrected Projection, Viewpoint (0, 0, 0)



(c) Zoomed-in regions of the linear (top) and corrected projection (bottom)

Figure 5.3: The photos show the bistro scene rendered with a linear projection (a) and the corrected rendering (b) on the physical monitor. Distortions such as the curved facade or elongated objects on the sides are visible for the linear projection. Two enlarged regions are depicted in (c) for detailed comparision of the two projections.

	Vertex Coordinates	Texture Coordinates	Fragment Shader
Avg. Frame Time	1.056ms	1.054ms	1.065ms

Table 5.1: The average frame time for the implemented variants of image-based correction.

5.2 Image-Based Method

We investigate the influence of the three implemented variants of the warping pass for the image-based method. Furthermore, we test the effect of cubic texture filtering in the warping pass and rendering the linear projection at higher resolutions. Our experiments suggest that the best configuration for the 100cm viewing distance is using texture warping in the fragment shader with cubic filtering and 1.25× the native resolution of the monitor. In this section, we argue why we select this configuration for the final comparison with the other methods in Section 5.5.

5.2.1 Image Warping Variants

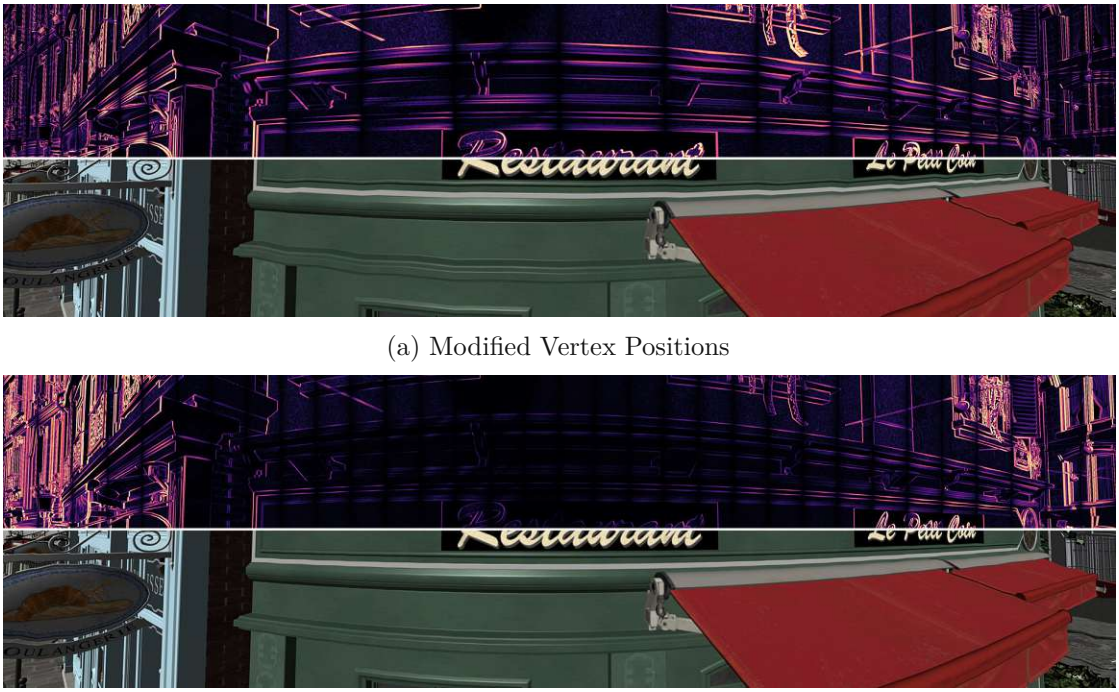
In Section 4.2, we discuss three different variants of the warping pass that generate the desired projection from a rendering with a linear perspective. The first two variants distort the vertex coordinates and the texture coordinates of the distortion grid, and the third variant modifies the texture coordinates in the fragment shader. Table 5.1 lists the average frame time for each of the three warping variants. This average is calculated over all samples along a camera tour through the bistro scene. The average value over all samples is representative of the differences since the performance overhead of each variant is constant. The distortion grid used for the measurements of the first two variants has a resolution of 128×36. For the fragment shader variant, only a screen-aligned quad is rendered. All three implementations are rendered with the native resolution of 5120×1440 and using the same viewpoint located in the center of the cylinder.

In terms of performance, our measurements confirm the results found by Pohl et al. [PJB13]. Whether texture warping is performed by transforming vertex or texture coordinates in the vertex shader or by transforming texture coordinates in the fragment shader does not impact performance considerably on the hardware used and with a reasonably sized distortion grid. In other words, the performance overhead of calculating the cylindrical projection for over $7 \cdot 10^6$ fragments instead of $4 \cdot 10^3$ vertices is negligible on current desktop hardware. Consequently, using a distortion grid likely provides no significant benefit when using powerful hardware. Still, a distortion grid might be a reasonable alternative on less powerful hardware or when the distortion geometry was measured in advance instead of using an analytical function.

We do not recognize visual artifacts for the variants using the distortion grid with the grid resolution used in the experiments. All variants suffer mainly from the expected resampling artifacts. However, for low distortion grid resolutions, or in the case of considerable distortion, as is required for close viewpoints, the methods in the vertex

shader may produce visible artifacts. These errors are a result from the linear interpolation of the vertex attributes of the distortion grid during the rasterization.

In Figure 5.4, we show the artifacts of a low grid resolution of 32×9 with a viewing distance of 25cm. Displacing the vertex coordinates results in a more significant error near the center where the grid cells are magnified. Modifying the texture coordinates does not alter vertex positions and produces a higher error on the sides of the image where more geometry is projected into the uniformly sized grid cells. These observations suggest that the displacement of vertex and texture coordinates can be combined and optimized to reduce the overall error for low grid resolutions. However, we do not investigate this option further and use the correction in the fragment shader in all following benchmarks as it does not suffer from these interpolation artifacts, and the performance difference is negligible, as depicted in Table 5.1.



(a) Modified Vertex Positions

(b) Modified Texture Coordinates

Figure 5.4: The images show artifacts for the image-based variants in the vertex shader when a low grid resolution of 32×9 is used with a viewing distance of 25cm from the monitor. A ꟻLIP difference map, using the fragment shader variant as the reference, is shown in the upper half, and the rendered result is visible in the lower half of the images. The linear interpolation of the vertex attributes of the distortion grid results in noticeable artifacts when displacing vertex positions (a) or texture coordinates (b). The correction in the fragment shader (not shown in this figure) does not suffer from these artifacts since every fragment is warped correctly.

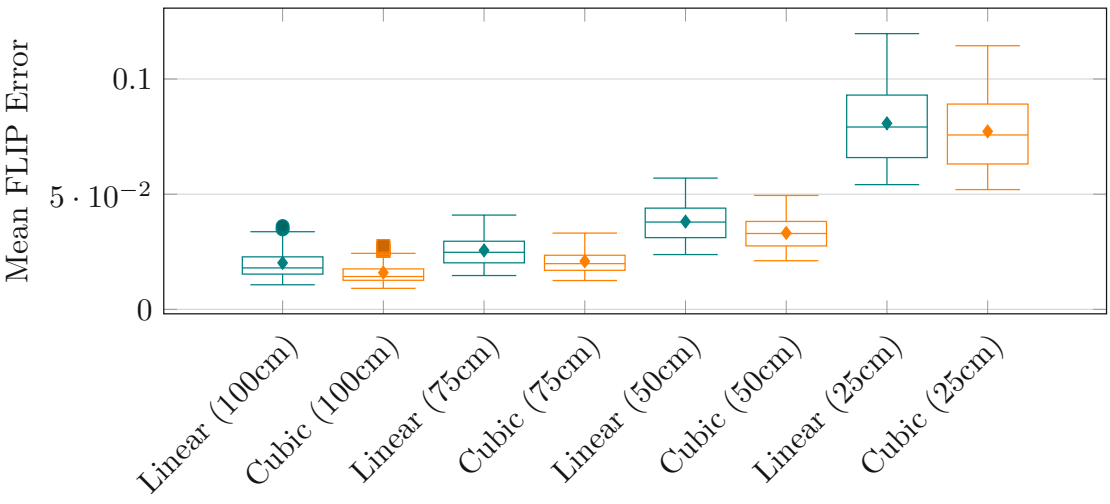


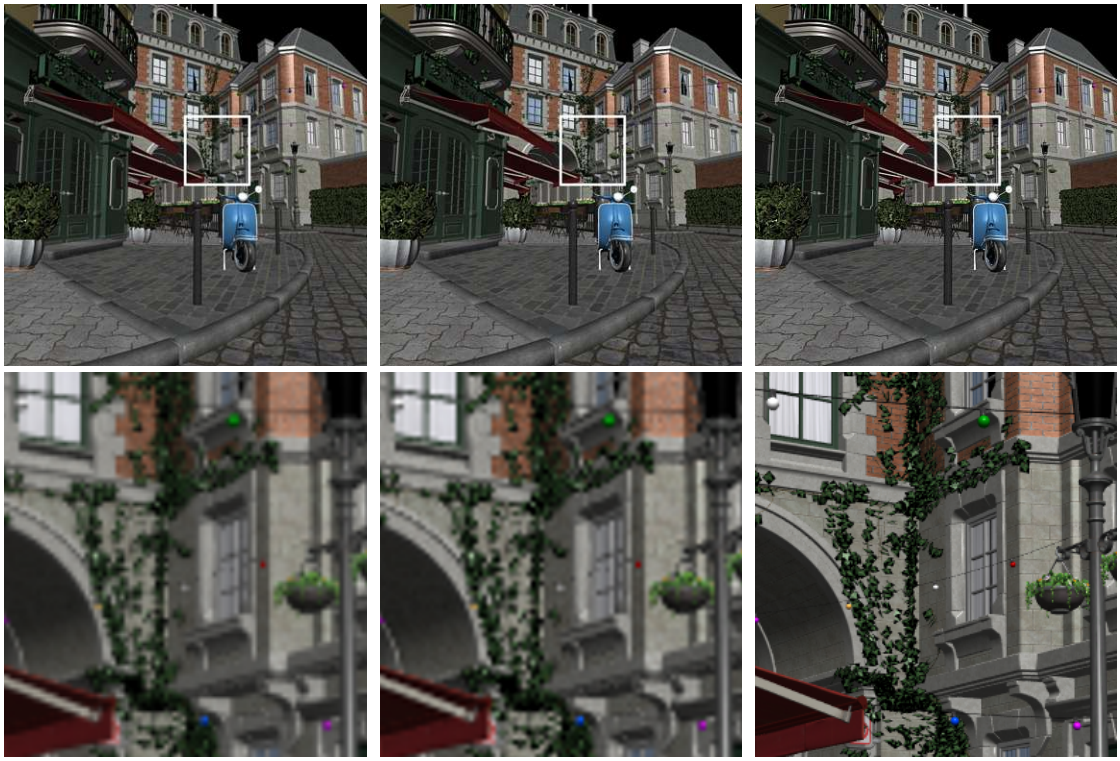
Figure 5.5: The boxplots picture the mean ꟻLIP error of 100 samples taken for cubic texture filtering versus linear filtering in the image-based method. A pair of boxplots for each of the four tested viewing distances from the monitor is depicted.

5.2.2 Cubic Texture Filtering

We confirm the noticeable improvement in the sharpness of the corrected image when using cubic texture filtering during warping as presented by Pohl et al. [PJB13]. The mean ꟻLIP error, using the mesh shader reference images, during the camera tour through the bistro scene is depicted in Figure 5.5 for linear and cubic filtering. The 100 samples from the benchmark are consolidated in a boxplot. Cubic texture filtering has a positive impact on the perceptual differences to the reference images according to the ꟻLIP error. Furthermore, the plots show that the relative improvement of the image differences decreases for viewpoints near the monitor. While the error is reduced by about 21% for the 100cm viewing distance, it is reduced only by about 4% for the close viewing distance of 25cm.

The smaller decrease in image difference for the viewpoint near the monitor is a direct consequence of the strong magnification of the image. The closer the viewpoint is to the monitor, the stronger the image has to be warped, and the more information is missing in the render texture. During warping, the cubic filtering can not reconstruct this missing information that was not present in the linearly projected image in the first place. This issue is visible in Figure 5.6, where a section of the warped image is shown for the linear and cubic filtering using a 25cm viewing distance. Both filtering methods show significant differences to the reference image and look blurry. However, the image using cubic filtering shows slightly increased sharpness and contrast. Note that the ꟻLIP error may not adequately capture the subjective image quality improvement using cubic filtering since the difference to the reference is already significant in both cases.

In terms of performance, linear texture filtering outperforms cubic filtering. Rendering



(a) Linear Texture Filtering (b) Cubic Texture Filtering (c) Reference

Figure 5.6: The images show a section of the image-based correction with linear filtering (a), cubic texture filtering (b), and the geometry-based reference image (c). A zoomed-in section shows the differences in detail. The loss in sharpness is the result of resampling the linear projection for a close viewing distance from the monitor of 25cm. With cubic texture filtering, the contrast of edges is more pronounced, and the image appears slightly sharper compared to linear filtering.

the scene with a linear perspective only and without any distortion correction requires about 0.99ms on average across all viewing distances. The frame time when using the image-based correction method with linear filtering is, on average, 1.07ms. Therefore, the post-processing pass adds about 0.08ms to the overall frame time. With cubic texture filtering, we measure about 1.36ms in total, increasing the post-processing time from 0.08ms to 0.37ms. Therefore, our implementation of cubic filtering is slower by a factor of 4.63 compared to the hardware-supported linear filtering. The performance impact may be considerably lower when hardware support for cubic texture interpolation is available on the hardware. Furthermore, our shader implementation of cubic texture interpolation is not optimized, and there may be room for improvement.

In summary, cubic texture filtering improves the sharpness of the warped images and reduces the perceptual difference to the reference images. However, our software implementation bears a considerable performance overhead compared to hardware-supported

linear filtering. Whether cubic filtering is more efficient than linear filtering also depends on the render resolution of the linear projection, which is analyzed in the following section.

5.2.3 Render Texture Resolution

As expected, increasing the render texture resolution improves the overall image quality. However, our measurements show that an optimal resolution in terms of FLIP error using reference images generated with the grid-based subdivision exists. Therefore, changes to the resolution should be done with care. In other words, increasing the resolution can also negatively affect image quality.

When rendering an image with a viewing distance of 100cm in our setup, a pixel in the center is magnified by a factor of about 1.21 during warping. Increasing the resolution by a factor of three provides more than two samples per warped pixel and should be sufficient for generating good results. Therefore, we test different resolutions up to 3× the native resolution. Furthermore, cubic texture filtering should be considered when testing the various resolutions. Hence, the tests are done with and without cubic filtering.

In Figure 5.7, the mean FLIP error for the various resolutions is plotted for the 100cm viewing distance. The plots show an optimum regarding the FLIP error at around 2× the native resolution for linear filtering and around 1.25× for cubic filtering. In Figure 5.8, we show the warped images when rendering at the native resolution, 2× the native resolution with linear filtering, and 1.25× the native resolution with cubic filtering together with the reference image. The images reflect the findings from the diagram, and the images using 2× and 1.25× the native resolution provide similar sharpness and appear similar to the reference image. This observation shows that increasing the resolution to an arbitrary degree is inefficient. Contributing factors are that the previously calculated magnification factor is only valid for a pixel in the center of the image and that the filter kernel used during the resampling is constant in the number of pixels. For example, the linear interpolation done by the hardware is limited to a four-pixel neighborhood when sampling a 2D texture[The22]. This fixed neighborhood size results in under-sampling of the linear projection when the sample locations in the linear projection are further apart than their respective neighborhoods are wide. Ideally, the filter would adapt to the magnification at every sample location and use the correct size for the filter kernel.

Analogous to the FLIP error, the frame time is plotted in Figure 5.9. Trivially, increasing the resolution increases the frame time. The measurements show that multiplying the resolution by a factor of two increases the frame time by about the same factor of two in our benchmark. The plot also shows that the overhead due to cubic filtering remains approximately constant for all resolutions.

Based on the FLIP error results and the measured performance, we select the resolution of 1.25× with cubic filtering for all further benchmarks. The overhead from the cubic filtering is still smaller than rendering at 2× the native resolution and provides similar

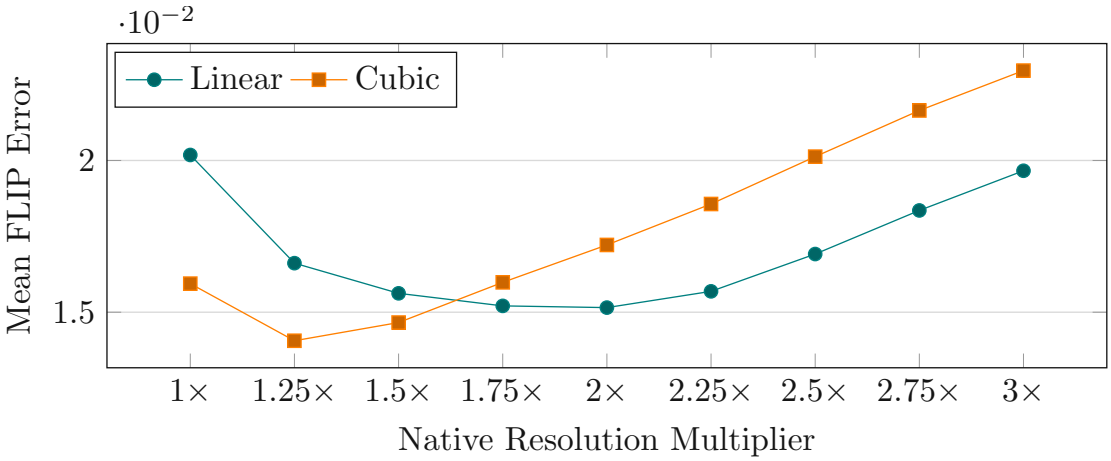


Figure 5.7: The chart shows the mean FLIP error when rendering at resolutions up to 3 times the native resolution with linear and cubic filtering for a 100cm viewing distance. The FLIP error is minimal at around 2× for linear filtering and 1.25× for cubic filtering.

(a) Linear (1.00×) (b) Linear (2.00×) (c) Cubic (1.25×) (d) Reference

Figure 5.8: Rendering the linear projection with 2× the native resolution and resampling with linear filtering improves image quality to a similar degree as rendering with 1.25× and resampling the image with cubic filtering.

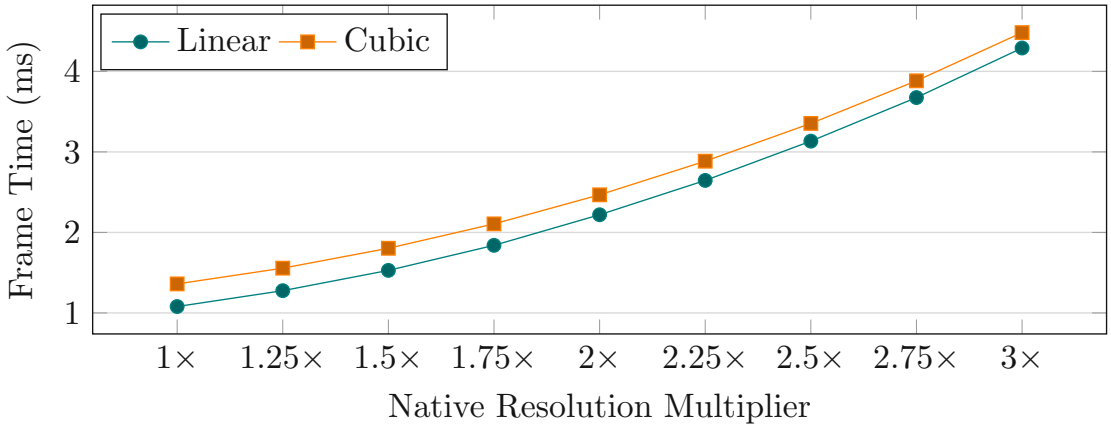


Figure 5.9: The chart shows mean frame time in milliseconds when rendering at resolutions up to 3 times the native resolution with linear and cubic filtering for a 100cm viewing distance.

quality. Note that this selected configuration is optimal for the 100cm viewing distance, and a different configuration might provide the best results for a different viewing distance.

5.3 Geometry-Based Method Using Tessellation Shaders

For the geometry-based implementation using tessellation shaders, we evaluate the quality and performance of both tessellation heuristics described in Section 4.3. We compare the two heuristics regarding their frame time and their mean ꟻLIP error using the reference images generated with the grid-based subdivision. Based on these metrics, we settle upon one of the heuristics with appropriate characteristics for comparing the different methods in Section 5.5.

In Figure 5.10, we show the mean ꟻLIP error over the 100 samples during the bistro benchmark for both heuristics. A pair of box plots is depicted for each of the four evaluated viewing distances. The first observation is a reduced error across all viewing distances when our extended tessellation heuristic is used. The error with our extended heuristic is about 36% lower on average. An important aspect is that the spread of the mean error is also scaled down to a similar degree with our modification. The error produced, therefore, is consistently lower. The reduced error is also visible in Figure 5.11, where two renderings with their respective ꟻLIP error maps for the two heuristics are displayed. The extended heuristic produces a noticeably darker error map corresponding to the lower error.

Similar to the error, the frame time is depicted in Figure 5.12. A performance overhead due to higher tessellation and additional computations for the extended heuristic is visible in the figure. With the extended heuristic, the frame time is increased by about 7% on average. The slight increase in the frame time of less than 10% is noteworthy, considering

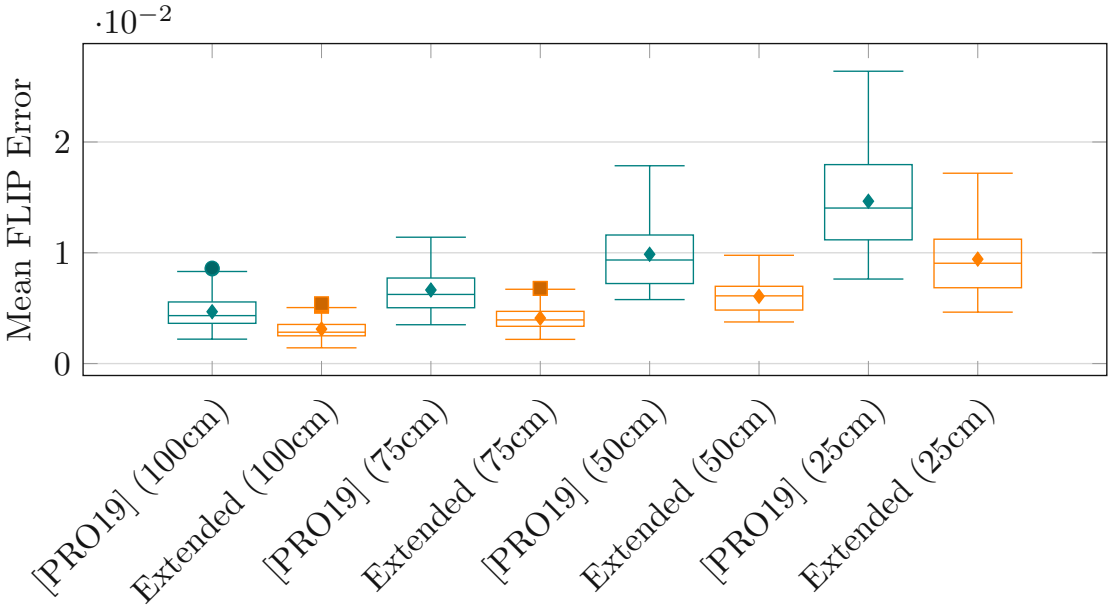
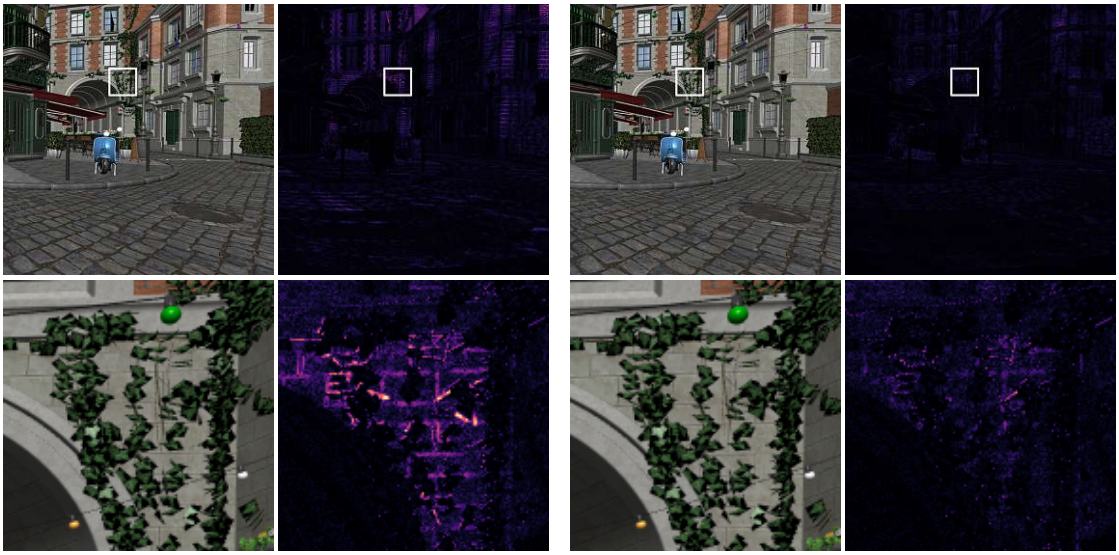


Figure 5.10: The boxplots show the mean ꟻLIP error of 100 frames during the bistro benchmark for the two tessellation heuristics and four viewing distances from the monitor.



(a) [PRO19] Heuristic (b) Extended Heuristic

Figure 5.11: The images show the results of the two discussed tessellation heuristics and their respective ꟻLIP error maps. A zoomed-in section is provided to show the differences in details. The extended heuristic in (b) generates a darker error map that corresponds to the lower error compared to the original heuristic in (a). The renderings correspond to a viewing distance of 25cm from the monitor.

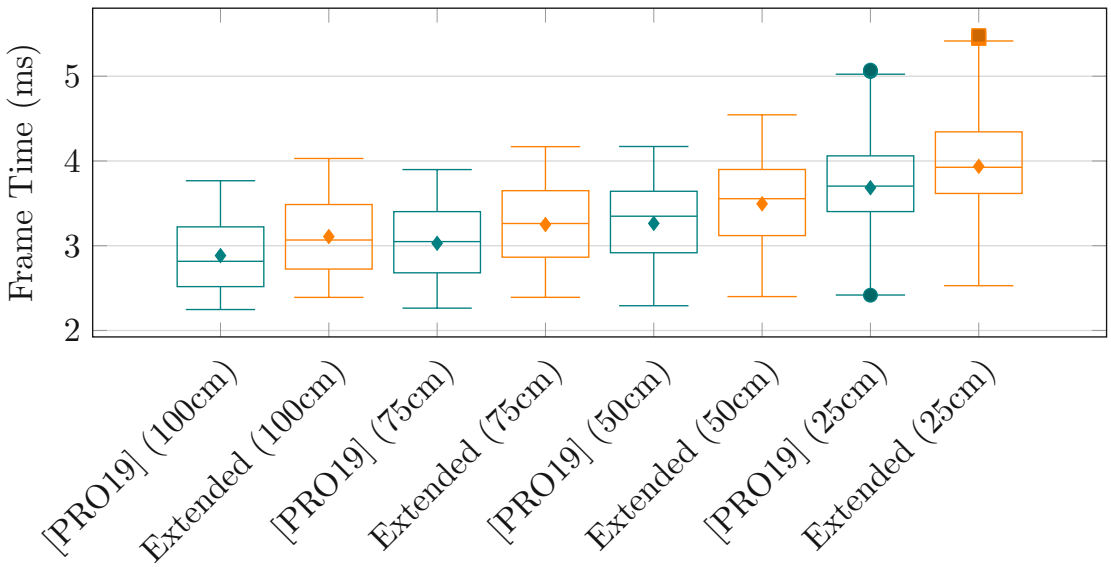


Figure 5.12: The boxplots present the frame time across the 1000 samples during the bistro benchmark for the two tessellation heuristics and four viewing distances from the monitor.

the significant increase in triangles generated with our heuristic. We measure an increase in the number of rasterized triangles of about 84% for the 100cm and 41% for the 25cm viewing distances. This observation highlights the efficiency of the rasterizer and shows that significantly increasing the triangle count does not necessarily increase the frame time considerably. Our extension to the heuristic does not generate as much additional geometry for the closer distances since the geometry becomes generally smaller in screen space with increasing FOV. Accordingly, the shorter edges of the triangles in screen space have to be subdivided less.

Based on the measurements, our extended tessellation heuristic is considered an improvement over the original heuristic from Pérez et al. [PRO19] when the same importance is attributed to the frame time and ꟻLIP error. The reduction of the error of roughly 36% is more substantial than the frame time overhead of about 7% when comparing the extended heuristic with the original heuristic. Therefore, the benchmarks in Section 5.5 displaying results of the geometry-based method with tessellation are generated using our extended heuristic.

5.4 Geometry-Based Method Using Mesh Shaders

In this section, we analyze our implementation of the geometry-based solution using the graphics mesh pipeline. The main configurable parameter of our method is the resolution of the grid that subdivides the geometry. We show that in our implementation, the grid resolution of 128×36 provides the best tradeoff between quality and performance

for the viewing distance of 100cm from the monitor. This grid resolution is also used in all further benchmarks. Another parameter we investigate is the influence of the number of invocations allowed to write to the output buffer. As explained in Section 4.4, only a fixed number of invocations of every mesh shader work group is allowed to write out triangles to the rasterizer. This limitation is necessary to guarantee enough space in the output buffer for all generated triangles. The influence on performance due to this restriction is an important consideration as it provides additional insight into the performance overhead of our implementation.

5.4.1 Grid Resolution

The resolution of the grid on which geometry is subdivided has a well-behaved and proportional impact on performance and quality. Generally, frame time and the number of generated triangles will increase with the number of grid cells. Figure 5.13 presents the frame time during the bistro benchmark for five grid resolutions with a viewing distance of 100cm from the monitor. From this figure, it is apparent that the frame time is monotonically increasing with the grid resolution. The grid resolution of 128×36 is the highest resolution that can achieve frame rates consistently above 60 FPS. With 256×72 cells, the frame rate occasionally dips below 60 FPS.

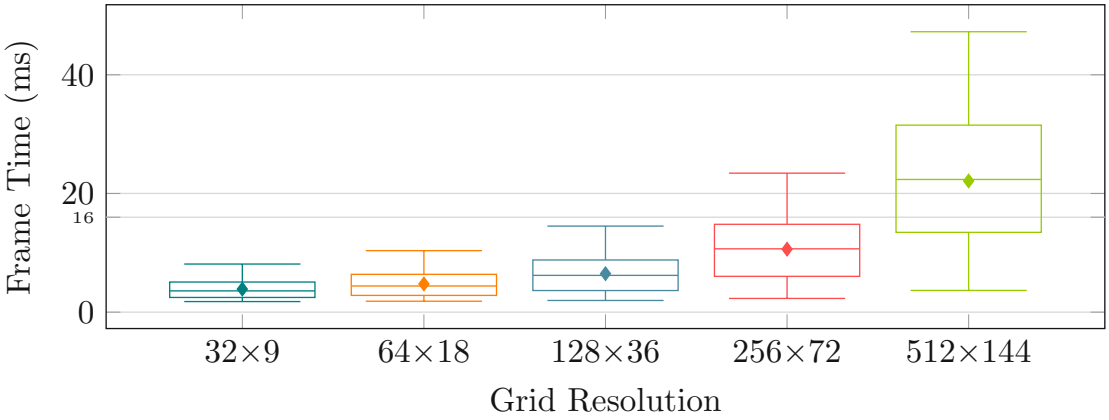


Figure 5.13: The boxplots show the frame time in milliseconds for five grid resolutions in the mesh shader implementation during the bistro benchmark with a viewing distance of 100cm from the monitor.

Figure 5.14 depicts the ꟻLIP error for the different grid resolutions using the reference images generated with a grid resolution of 2560×720. As expected, the error decreases consistently when the resolution is increased. Note that a low grid resolution such as 32×9 is generally insufficient to get an acceptable image quality, as shown in Figure 5.15. Geometric errors can occur inside the cell boundaries. Only at the crossings of the grid, the geometry is guaranteed to be free from artifacts. At a grid resolution of 128×36, every cell has a size of 40×40 pixels when the native resolution of the monitor is used. With this resolution, we are also subjectively satisfied with the overall results.

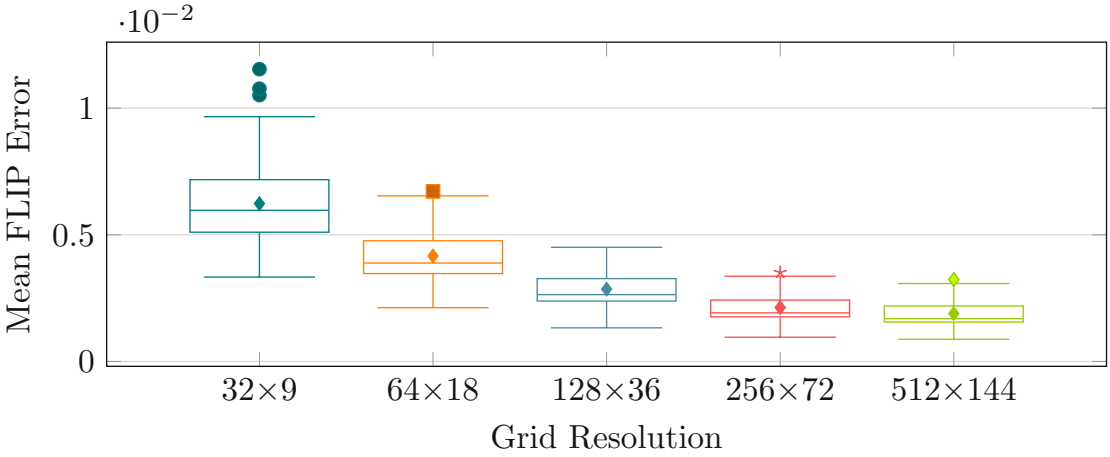
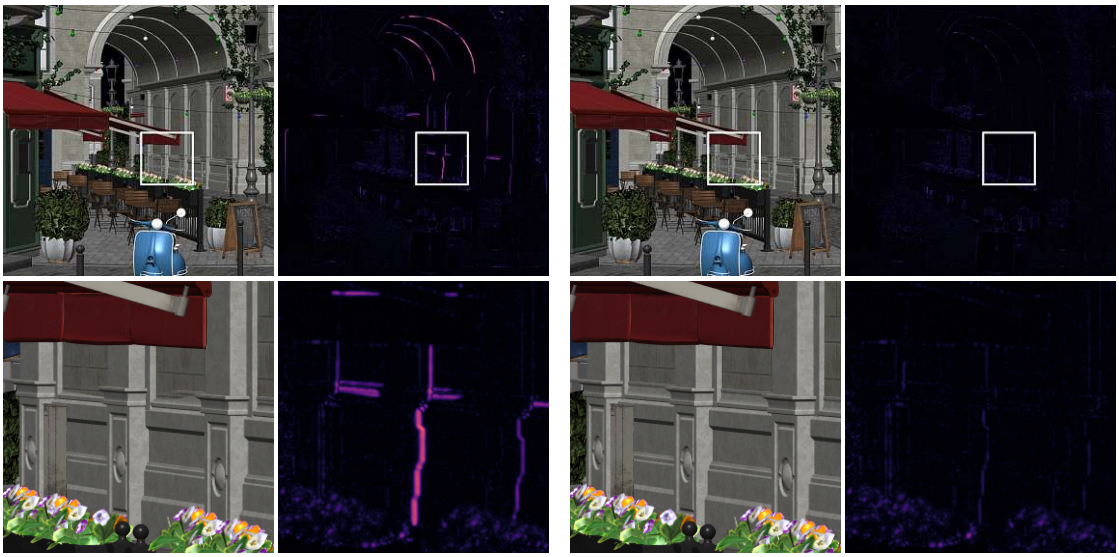


Figure 5.14: The boxplots depict the mean FLIP error for 100 samples of the mesh shader implementation for five different grid resolutions using the reference images generated with a grid resolution of 2560×720.



(a) Grid Resolution: 32×9 (b) Grid Resolution: 128×36

Figure 5.15: Noticeable artifacts appear for our grid-based method when the grid resolution is too low. With a grid resolution of 32×9, the coarse geometry of the depicted archway suffers from artifacts at a 100cm viewing distance. Most artifacts are resolved at a higher grid resolution, such as 128×36.

To identify the best grid resolution in terms of quality and performance, we multiply the overall average ꟻLIP error with the average frame time to describe the tradeoff between the two metrics. This tradeoff value is minimal when both metrics are sufficiently small. Therefore, the minimum value corresponds to the desired properties when both metrics are considered equally important. Figure 5.16 depicts the described tradeoff for each tested resolution. This figure shows that the resolution of 128×36 cells provides a minimal error for a reasonable frame time. Lower grid resolutions suffer from a high error that is not compensated by faster rendering. Higher resolutions suffer from the increase in frame time, while the gain in accuracy is limited.

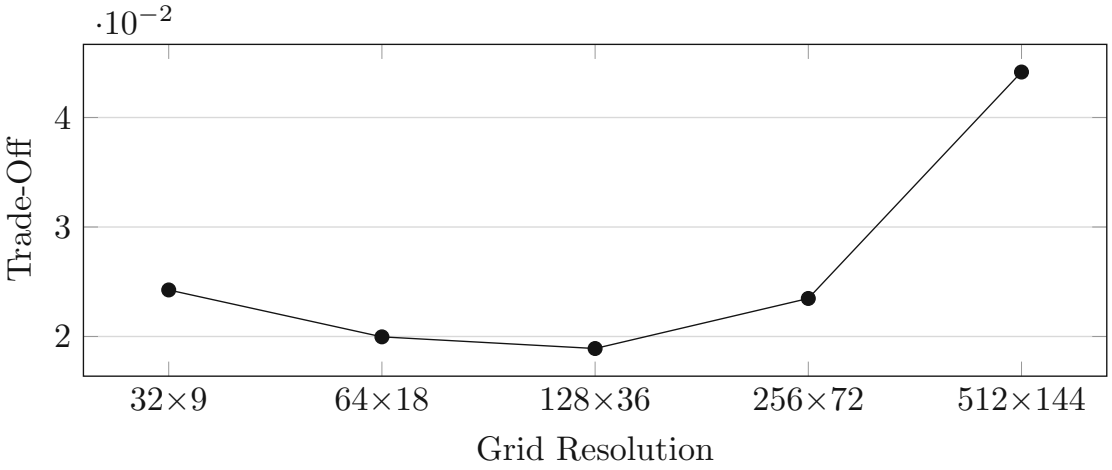


Figure 5.16: The graph depicts the tradeoff for the tested grid resolutions. This tradeoff is calculated by multiplying the ꟻLIP error and frame time and is minimal when both metrics are sufficiently small. The figure shows that the grid resolution of 128×36 cell provides the best ratio of performance to error.

5.4.2 Number of Output Invocations

As discussed in Section 4.4, our implementation using the graphics mesh pipeline only utilizes a limited number of invocations of every mesh shader workgroup to subdivide triangles and pass them to the rasterizer. The size of the output buffers constrains the number of invocations involved, since all potentially generated triangles have to be written to it. Our output buffer is sized to provide space for the triangles generated by 13 invocations. Nevertheless, it is interesting to test the influence of the number of output invocations to gain insight into potential improvements for our implementation.

In Figure 5.17, we show the rendering performance when varying the number of output invocations while the size of the output buffer remains fixed. Generally, the frame time is lower with more output invocations, but it does not scale particularly well. The average frame time with 10 output invocations is about 7.25ms, and the frame time with 30 invocations is only 36% faster with 4.64ms. Furthermore, the diagram indicates that increasing the number of invocations results in a decreasing gain in frame time and that

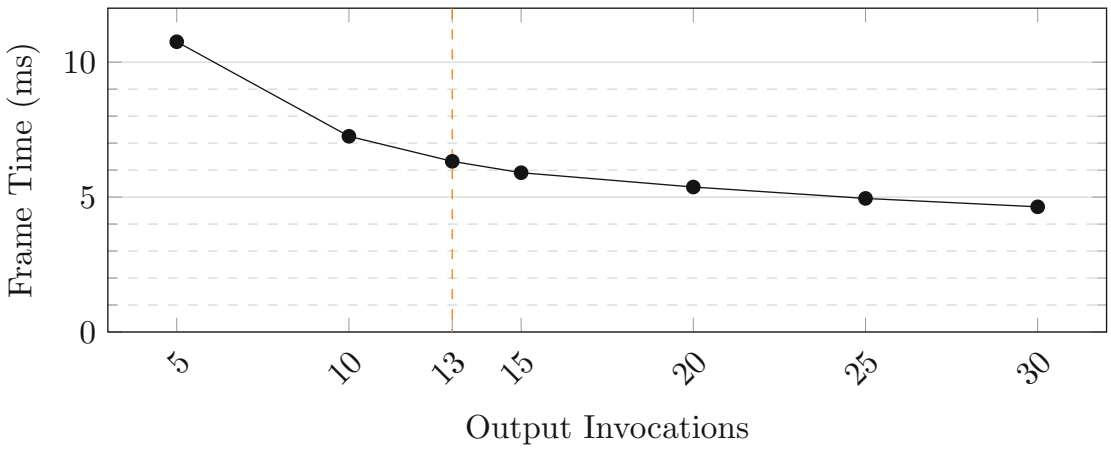


Figure 5.17: The chart shows the average frame time based on the number of invocations in the bistro benchmark for a viewing distance of 100cm. Note that the output buffers have a constant size, and artifact-free rendering is not guaranteed when more than 13 invocations are used.

a limit is approached. Hence, while the number of invocations is a contributing factor, the inefficient usage of the available invocations is not the sole reason for the slower performance compared to the geometry-based implementation using tessellation.

Note that for more than 13 output invocations, the rendering is not guaranteed to be free from artifacts. Missing triangles that did not fit into the output buffer can be the result. While not drawing all triangles may reduce the frame time further and skew the measurement, we already know, from comparing the two tessellation heuristics, that rasterizing considerably more triangles has only a small impact on performance. Therefore, the measured frame time is reasonable even when not all triangles are rendered.

It is noteworthy that we did not notice missing triangles up to 20 output invocations. This observation reflects that our estimation for the number of generated triangles is not optimal and that there is potential for finding a better approximation and, thereby, utilizing the allocated output buffers more efficiently. However, we continue using our trivial estimation to guarantee artifact-free rendering and use 13 invocations for all benchmarks.

5.5 Comparison of the Methods

In this section, we directly compare the quality and performance of the different rendering methods across three scenes using the previously determined configurations. For the image-based method, we increase the render resolution by 1.25× and use cubic texture filtering with the correction in the fragment shader. The hardware tessellation method utilizes our extended tessellation level heuristic, and the grid-based subdivision in the mesh shaders is configured with a grid resolution of 128×36. Finally, our implementation

using ray tracing has no special parameters and is used as is. The three scenes are *UE4 Sun Temple* (0.6M triangles) [Gam17], *Amazon Lumberyard Bistro* (2.8M triangles) [Lum17], and *NVIDIA Emerald Square* (10M triangles) [NHB17] from the ORCA [NVI22]. The variety of geometry in these scenes provides a broader picture of the behavior of the methods under different circumstances. Also, the geometric complexity in terms of the number of triangles in the scenes represents different workloads, with *UE4 Sun Temple* containing the least triangles and *NVIDIA Emerald Square* the most.

Not only the triangle count is important, but also the nature of the geometry is a consideration. In the *UE4 Sun Temple* scene, the geometry consists mostly of planar walls constructed from large triangles, which usually meet at right angles. The *NVIDIA Emerald Square* scene, on the other hand, has a large number of triangles in foliage that is spread over a large volume in a central park. Finally, the *Amazon Lumberyard Bistro* scene provides a mix of dense geometry on complex objects, foliage, and large buildings and thereby represents a more generic workload.

The overall frame time achieved by the different methods across the three scenes is presented in Figure 5.18. The results for two viewing distances of 100cm and 25cm from the monitor are shown. Generally, the image-based variant achieves the best performance, followed by the geometry-based implementations using hardware tessellation. In our benchmarks, the image-based implementation is faster by about a factor of 1.3–2.6 compared to the hardware tessellation method. In most cases, our mesh shader implementation is still faster than ray tracing for the *UE4 Sun Temple* and *Amazon Lumberyard Bistro* scenes. However, compared to hardware tessellation, our grid-based implementation is slower by a factor of 2.1–2.3 across the scenes. In the geometrically demanding *NVIDIA Emerald Square* scene, the geometry-based methods lose ground compared to the other methods. In this scene, the performance of the hardware tessellation method is comparable to that of ray tracing, and the mesh shader implementation is considerably slower and dips below 30 FPS. The figure further shows that the geometry-based and image-based methods scale approximately with the number of triangles. This observation is comprehensible since every triangle has to be processed individually to generate the final rendering. The implementation using hardware-accelerated ray tracing seems to gain performance in the largest scene compared to the other methods. The designated acceleration structures used for ray tracing presumably reduce the overhead of rendering the complex scene.

There are notable considerations regarding the measured performance of the methods. First, our scenes are entirely static. With moving objects, the TLAS—one of the hierarchical acceleration structures used for ray tracing—must be rebuilt every frame. This additional workload may reduce the overall performance of an implementation using ray tracing. Another aspect to consider is that rasterization-based methods may utilize standard hardware-supported anti-aliasing strategies, whereas any anti-aliasing for ray tracing may increase the frame time noticeably. Furthermore, we are confident that optimizations to our grid-based subdivision scheme are possible. One argument for this claim is the significant amount of custom shader code that can be tuned and improved.

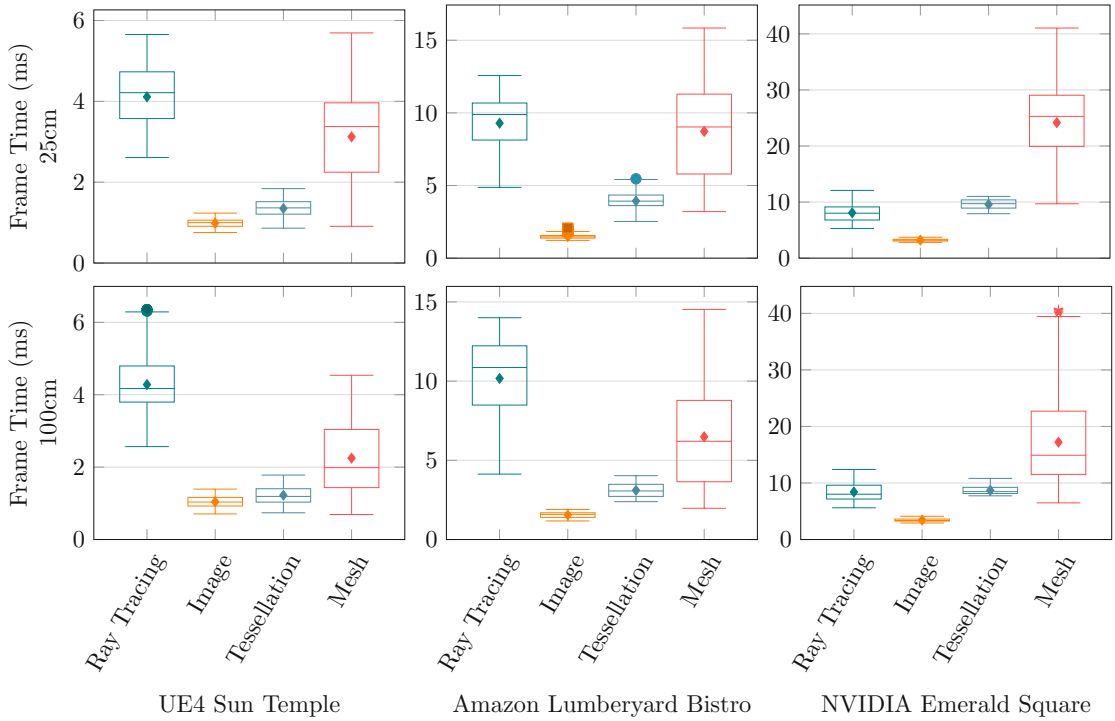


Figure 5.18: The charts depict the frame time in milliseconds for the two viewing distances of 100cm and 25cm in the three scenes.

Another argument is that we observed an overhead when using the mesh shader pipeline for generating a linear projection compared to the traditional pipeline. In this scenario, the mesh shader pipeline should perform similarly to the traditional pipeline. However, we notice an overhead of 14% in the *UE4 Sun Temple* scene and 75% in the *NVIDIA Emerald Square* scene. Although we use a library for generating optimized meshlets[1], one potential reason for the difference might be a suboptimal assignment of geometry to meshlets in our scenes. Another option is a generally unfavorable utilization of the meshlet pipeline. The actual cause for the difference is subject to further investigation.

The number of rasterized triangles is presented for the two viewing distances and the three scenes in Figure 5.19. Ray tracing is excluded in this case since there is no rasterization involved. The image-based variant represents approximately the number of triangles visible on the screen after clipping. Therefore, this number is close to the minimum amount of triangles that must be rasterized in any case. Note that it is only an approximation of the minimum since some triangles may not be visible on screen after warping, and two triangles from rendering the screen-filling quad for the warping pass are added. The figure shows that hardware tessellation generates the most triangles in the first two scenes and that the difference is more pronounced for the close viewing

[1] `https://github.com/zeux/meshoptimizer#mesh-shading` (last accessed on July 17, 2022)

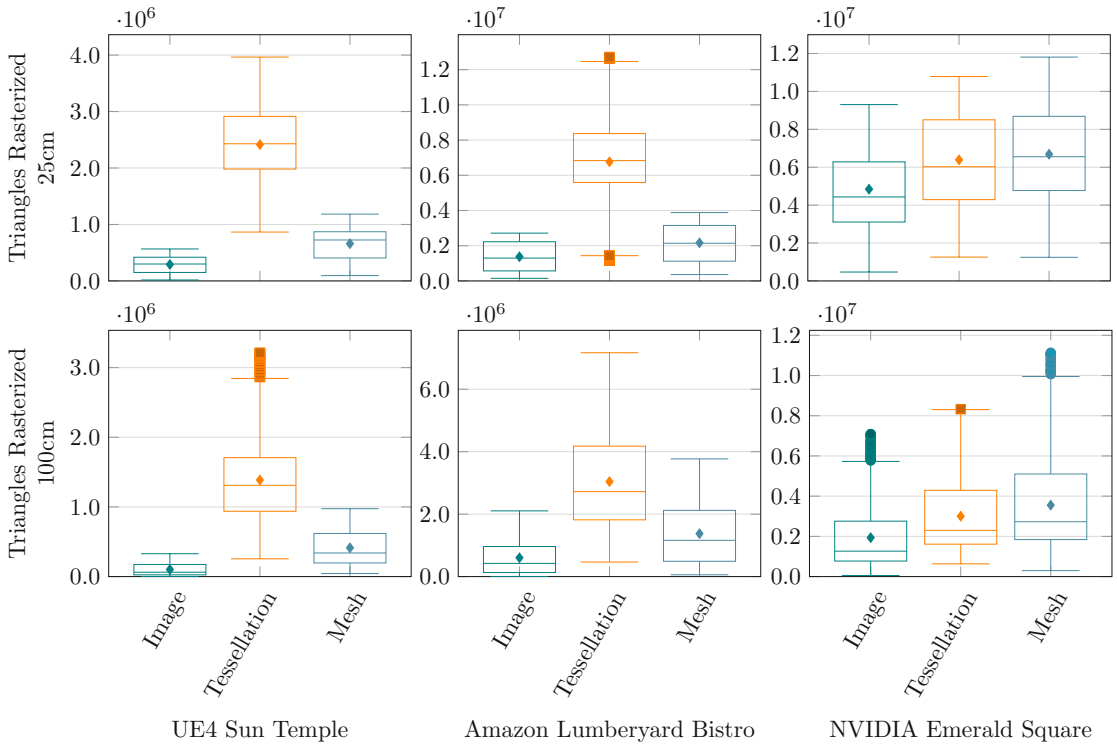
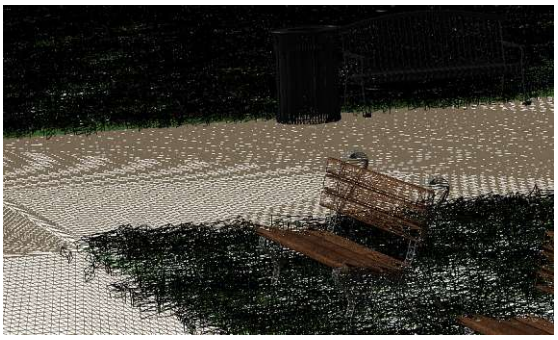


Figure 5.19: The charts show the number of rasterized triangles for the two viewing distances of 100cm and 25cm in the three scenes.

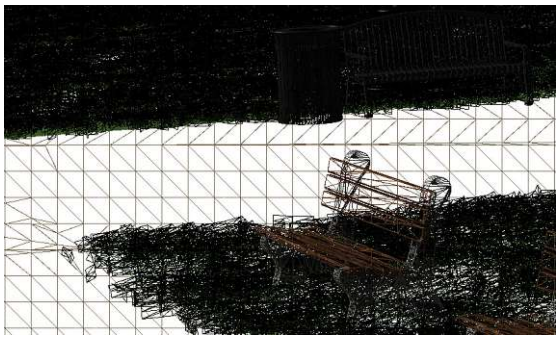
distance. Our grid-based subdivision generates about 62%–71% fewer triangles than the hardware tessellation in these scenes. Interestingly, this is not the case for the *NVIDIA Emerald Square* scene, where the grid-based subdivision generates 12% more triangles on average than the hardware tessellation method. This scene contains many small triangles for rendering leaves and grass blades. A visual example of the amount of geometry is visible in Figure 5.20a, where the individual meshlets are highlighted with different colors. Furthermore, the figure depicts a small section of the view as wireframe rendering for comparison of the tessellation shader implementation in Figure 5.20b and the grid-based subdivision in Figure 5.20c. While the grid-based subdivision generates considerably fewer triangles for the floor and bench, the tessellation heuristic results in fewer subdivisions for the already sufficiently small triangles of the vegetation. For the mesh shader implementation, these small triangles will naturally intersect with the grid at some locations and, therefore, will be subdivided. For this view, the tessellation heuristic generates about $7.3 \cdot 10^6$ triangles after clipping, while the grid-based subdivision generates nearly $9.9 \cdot 10^6$ triangles in total. This result suggests that the grid-based subdivision is unsuitable for small and dense geometry, which presumably is the reason for its bad performance in the *NVIDIA Emerald Square* scene.

A related issue to the high triangle count of our mesh shader implementation in the

(a) Highlighted Meshlets



(b) Tessellation Wireframe



(c) Grid-Based Wireframe

Figure 5.20: The *NVIDIA Emerald Square* scene contains many small triangles for the vegetation. This kind of geometry is unfavorable for the grid-based subdivision (c). While the implementation with hardware tessellation (b) is less efficient for subdividing the floor, it does not further subdivide small triangles of the vegetation.

NVIDIA Emerald Square scene is recognizable in the frame time during the *UE4 Sun Temple* benchmark. For each sample point during the benchmark, the frame time achieved by the four methods is depicted in Figure 5.21. Halfway into the benchmark, a sudden increase in frame time for the mesh shader implementation is observed. The other methods do not reflect an increase to the same extent. At that point, the camera is pointed at the end of a hallway where only a few objects remain in the view volume. Then the camera is turned around 180° sharply, and a large portion of the scene is contained in the view volume. The grid-based subdivision is not limited by distance, and all triangles in the view volume have to be considered for subdivision. Similar to the *NVIDIA Emerald Square* scene, the dependence on the number of triangles in the view volume is the main downside of our implementation. Furthermore, this downside is likely the reason for the large spread of minimum and maximum frame time during the benchmarks. Nevertheless, right before the increase, it is notable that our mesh shader implementation achieves a comparable frame time to the hardware tessellation implementation and, in fact, performs slightly better. Therefore, our implementation presents a viable alternative when only a limited amount of geometry is rendered.

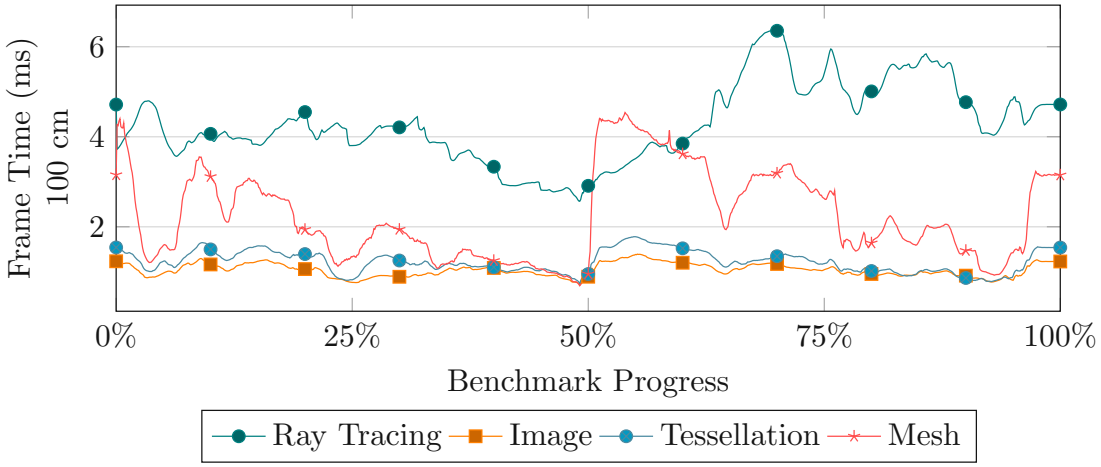


Figure 5.21: The chart depicts the time series of the average frame time achieved by the four methods throughout the *UE4 Sun Temple* benchmark with a viewing distance of 100cm.

Analogous to the frame time and the number of rasterized triangles, the mean ꟻLIP error using the reference images generated with the grid-based subdivision is shown in Figure 5.22 for the three scenes. As explained in Section 3.3, we use SSAA without additional texture filtering to generate anti-aliased images for a fair comparison. However, it is important to mention that the results presented must be interpreted with care since the error of the different methods originates from different sources. For ray tracing, the differences come from the sampling pattern that is different from that used by the rasterizer for generating the reference images with hardware-supported SSAA. Differences associated with the image-based method originate from resampling the linearly projected images. For the geometry-based methods, the difference essentially measures geometric artifacts only. Furthermore, the image differences are generally small, and from our subjective assessment, all methods generate acceptable image quality when configured appropriately. The relative differences between the methods presented in the figure may exaggerate an actual real-world experience.

The results show that the sampling error of the ray-traced images is comparable to that of the image-based method for the 100cm viewing distance. For the closer viewing distance of 25cm, ray tracing achieves better results than the image-based method since ray tracing does not introduce any additional blur. The render resolution and filter used for the image-based method should be adapted dynamically to optimize the results across different viewing distances.

The geometry-based methods achieve the lowest perceptual difference to the reference across all scenes. This result is expected since geometric errors only appear for small regions in the image, whereas the sampling artifacts of the other methods appear across the entire image. Any region in the image without geometric errors is essentially equivalent to the reference image. The average ꟻLIP error is similar for both depicted geometry-

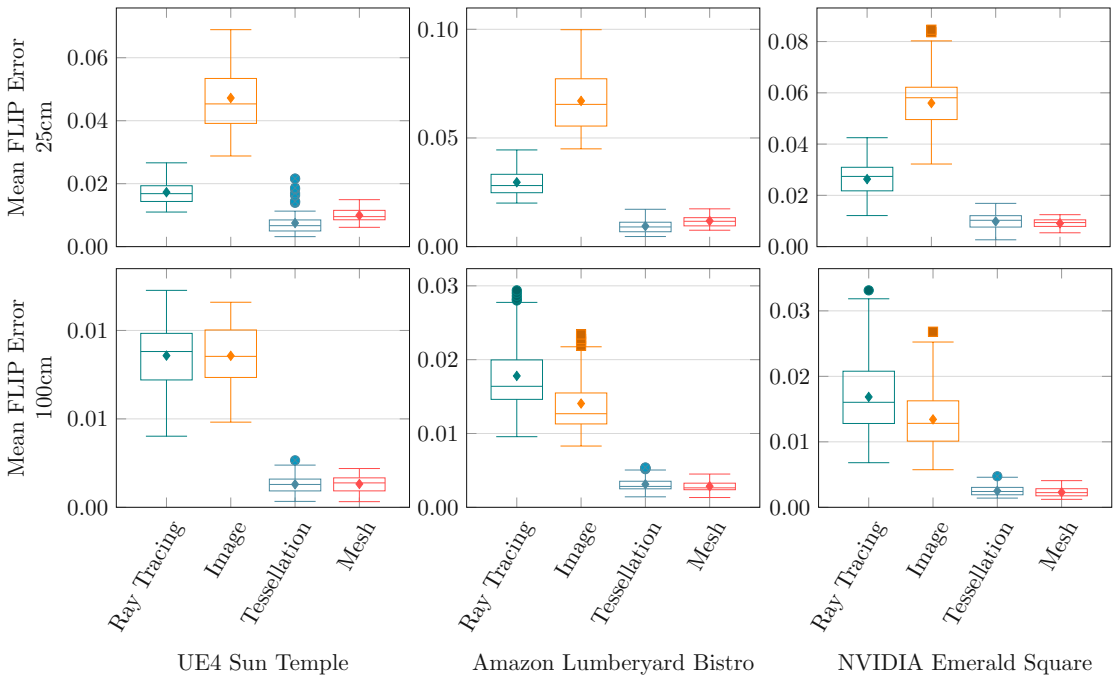


Figure 5.22: The mean ꟻLIP error for the two viewing distances of 100cm and 25cm in the three scenes.

based methods, and we consider their image quality equivalent for the 100cm viewing distance. Both methods gain in terms of ꟻLIP error for the 25cm viewing distance, but the grid-based subdivision appears slightly worse since a constant grid resolution is used. In other words, the same grid resolution is used to correct a more substantial distortion, while the subdivision is adjusted dynamically for the method using tessellation shaders. Another interesting observation regarding hardware tessellation is that there are noticeably more potential outliers for the 25cm viewing distance in the *UE4 Sun Temple* scene. We attribute these potential outliers to samples where the hardware limit of tessellation is reached, and walls near the camera are not subdivided sufficiently.

In summary, the image-based method generally provides the best performance, but further improvements regarding render resolution and filtering should be considered to reduce resampling artifacts. Geometry-based methods and ray tracing produce sharp images out of the box since no resampling is involved, but they struggle to compete with the image-based method in terms of performance. Both geometry-based methods generate results close to the reference images, and geometric artifacts are not impeding the user experience in most cases. The hardware tessellation method generates more triangles for coarse geometry, suggesting that our grid-based subdivision scheme is more efficient in generating the corrected geometry for coarse geometry. Therefore, the grid-based subdivision may be viable for large structures of coarse geometry, such as buildings and infrastructure. The presented implementation of the grid-based subdivision is not appropriate for dense

geometry due to its performance overhead. Nevertheless, observing cases where our subdivision scheme achieves comparable performance to hardware tessellation when a limited amount of geometry is rendered is a promising result.

5.6 Visual Differences & Geometric Artifacts

In this section, we discuss artifacts of the methods and compare their visual difference under certain conditions. In general, the average differences in frame time and ꟻLIP error provide an overview of the tradeoffs of the implementations. This data provides sufficient insight for the image-based solution since the resampling artifacts are constant for a specific resolution and viewing distance. However, for the geometry-based methods, some artifacts may not be captured with our benchmarks as they are present only under certain circumstances. We, therefore, present cases where we find the geometry-based methods to generate inaccurate results.

An essential difference between the discussed implementations is their capability to present a wide FOV. For example, there is no limit to the FOV for ray tracing. Our implementations of the image-based and geometry-based methods using tessellation shaders are limited to 180°. In both cases, adaptations to support larger FOVs exist. For the image-based method, multiple linear projections can be stitched together [TNAM16, BS18]. The geometry-based implementation using tessellation shaders can be extended with an additional clipping stage to ensure that triangles are rendered correctly [ALMM14]. In the traditional pipeline, this clipping process must be performed in the geometry shader after the tessellation for all generated triangles. Our mesh-shader implementation has the positive side effect that this clipping stage is included implicitly, and the method supports a FOV up to 360°, as shown in Figure 5.23. Triangles spanning the discontinuity of the projection at $\pm\pi$ are split when they are clipped against the grid cells.

One problem regarding hardware tessellation is the hardware limit for how often triangles can be subdivided. The limit of current graphics hardware is typically 64 subdivisions and can be found in the Vulkan hardware database for a wide range of GPUs under the property name *maxTessellationGenerationLevel* [Wil22]. This limit can result in unsatisfactory correction for large triangles when viewed at a close distance since the subdivision may not be sufficient. This problem is apparent in Figure 5.24, where a large wall, rendered with the hardware tessellation method and a viewing distance of 25cm, is insufficiently subdivided. The corresponding ꟻLIP difference map shows the difference to the accurate reference generated with our grid-based subdivision. The issue is even more noticeable when the geometry intersects with the camera, as the edges at the intersection will be inaccurate. A wall intersecting with the display cylinder is shown in Figure 5.25. For the hardware tessellation method, the cut through the wall is jagged since the hardware limit during the subdivision of the wall is reached. With the grid-based subdivision, the cut resembles the expected parabola closely since the subdivision does not depend on the size of the triangles and only subdivides visible triangle regions.

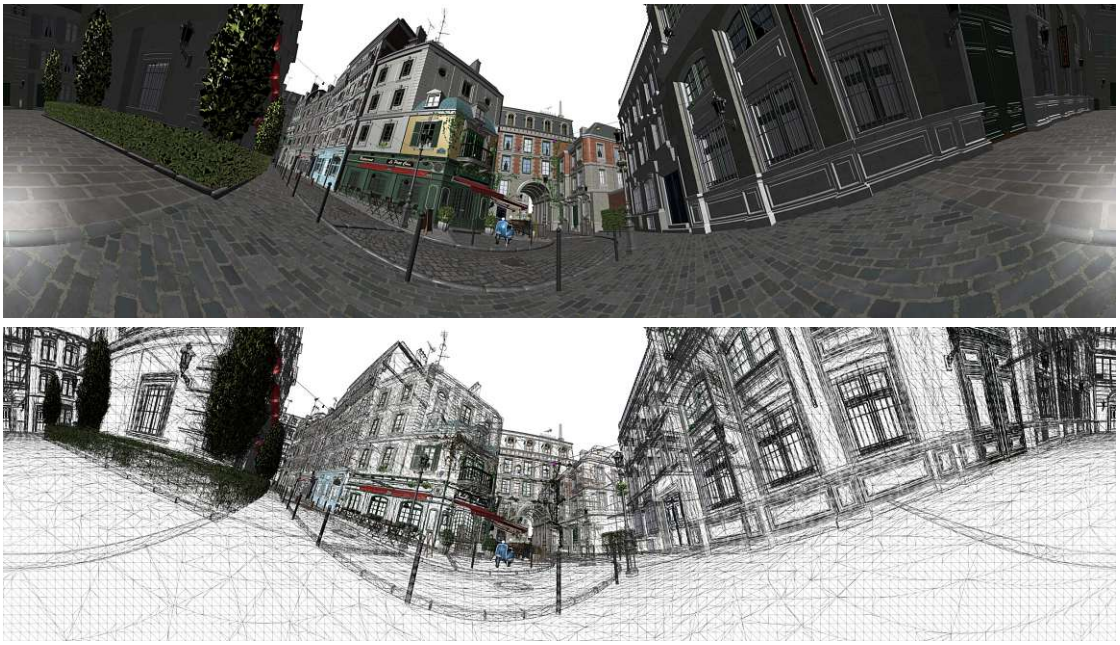


Figure 5.23: The images shows an artifact-free 360° rendering and the corresponding wireframe generated with our grid-based subdivision.

Another issue of the geometry-based methods arises from parallel geometry that is very close to each other. Hardware tessellation may not subdivide the triangles at the same locations and at the same rate. This inconsistent subdivision can lead to occluded geometry protruding the occluding geometry. In other words, vertices of the occludee may be projected in front of the interpolated fragments of the occluder. One example of the resulting artifacts for a viewing distance of 100cm from the monitor is provided in Figure 5.26, where the restaurant sign is protruded by the wall behind. The figure also shows that our grid-based implementation seems to suffer less from this issue as, generally, both geometries are subdivided at the same location on the grid. Therefore, the error due to the linear interpolation is similar for the inner fragments of both geometries.

The grid-based implementation is not entirely immune to this kind of error when the occluded geometry is denser than the occluding geometry in the first place. In this scenario, protruding vertices may already exist between the cell boundaries. Figure 5.27 shows a section of a rendering with a viewing distance of 25cm where this problem is apparent. In the rendering, the grid-based subdivision suffers from protruding geometry on the ground. The same error does not occur for the hardware tessellation due to higher tessellation. Additionally, the same figure shows another inaccuracy of the grid subdivision when an intersection of geometry is viewed from a shallow angle along the intersection. The intersection of the wall and the ground is imprecise and does not appear as a straight line. These artifacts are directly related to the viewing distance and are less visible for a viewing distance of 100cm.

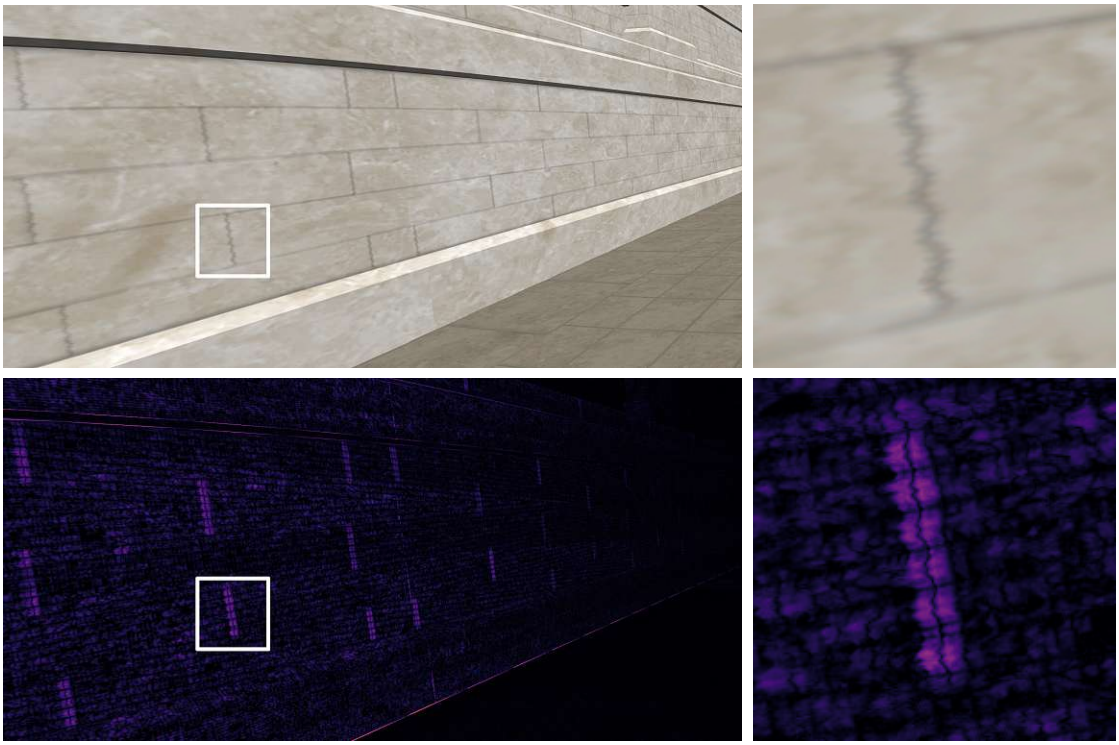


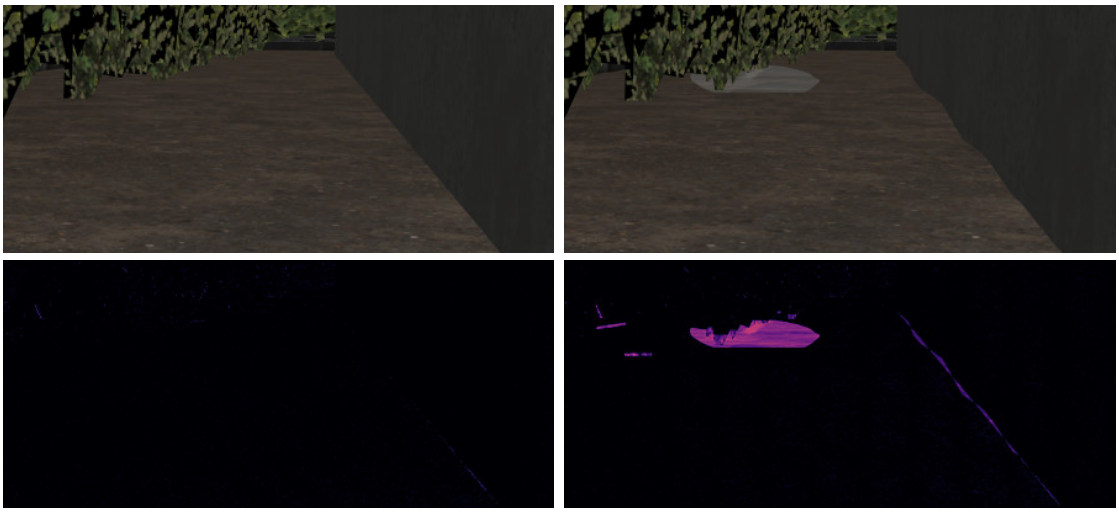
Figure 5.24: The images show a wall rendered using the hardware tessellation method with a viewing distance of 25cm and the corresponding ꟻLIP difference map. The maximum number of subdivisions is reached, and errors are visible near the camera.

(a) Hardware Tessellation (b) Grid-Based Subdivision

Figure 5.25: The images show the differences between the geometry-based methods when geometry intersects with the camera. Hardware tessellation produces inaccurate cutaways when geometry intersects with the camera and the subdivision limit of the hardware is reached. This problem is not present with our grid-based subdivision scheme since the subdivision is tied to the screen space.

(a) Hardware Tessellation (b) Grid-Based Subdivision

Figure 5.26: The images show geometric artifacts of a wall protruding a restaurant sign in front for the hardware tessellation method (a). In this case, the mesh shader implementation (b) appears more resilient since the grid-based subdivision splits the geometry of the wall and the sign at the same positions. The images were rendered with a viewing distance of 100cm from the monitor.



(a) Hardware Tessellation (b) Grid-Based Subdivision

Figure 5.27: The images show artifacts for our grid-based subdivision when the geometry is viewed at a shallow angle. The hardware tessellation method (a) generates better results in this case due to stronger tessellation. The mesh shader implementation (b) suffers from incorrect depth rendering on the ground, and the wall appears warped near the ground. The images were generated with a viewing distance of 25cm from the display.

The discussed depth artifacts of the geometry-based methods result from the linearly interpolated vertex attributes. It is possible to improve these visibility artifacts by calculating and writing the correct depth values to the z-buffer in the fragment shader. While this approach does not enhance the rasterized geometry, it does guarantee a correct depth ordering of objects. However, this solution can reduce the performance since the fragment shader has to be executed in every instance to identify the real depth value of the fragment. As a result, the early fragment test [KBR19] can not be utilized to discard fragments of occluded geometry.

5.7 Head Tracking

In this section, we provide a short assessment of the FTVR experience using the two implemented head-tracking methods. Note that achieving a believable FTVR experience generally requires low system latency and a good calibration of the viewing parameters. Otherwise, various artifacts like "swimming" of objects and diminished depth perception can be the result [PGRS13, FSD+19]. We do not expect a perfect experience since we concentrate on the rendering methods in this work. Nevertheless, the quality that is achievable right away when using current tracking hardware is a relevant aspect.

We found that webcam-based tracking is too slow for a good experience out of the box. In our implementation, the latency between head movement and the reflected perspective in the experiment is considerable. Also, we found it challenging to configure the parameters correctly so that the tracking is sufficiently accurate. An automated camera calibration feature for estimating the intrinsic camera parameters of a webcam is missing in the *opentrack* software at the time of writing.

The implementation using the *Tobii Eye-Tracker 5* was considerably faster, more accurate, and easier to set up. However, we still recognize latency between head movement and the response in the application. Especially when moving fast, this issue becomes apparent. Another problem with the tracker, although it may be easy to fix, is a slight jitter in the tracking data. Even slight variances in the tracking data are apparent in the rendering since the edges of the geometry can appear to flicker due to the small variations in perspective. Therefore, the tracking data must be as stable as possible to provide a pleasant experience. Additional filtering or thresholding of the input data may be enough to alleviate the issue.

In conclusion, consumer-grade tracking hardware like the *Tobii Eye-Tracker 5* provides a reasonable estimate of the viewpoint position for rendering a correct perspective. Nevertheless, additional improvements regarding tracking latency and accuracy must be implemented to create a good FTVR experience. Also, since the volume in which the user is tracked is narrow, combining multiple tracking devices may be necessary for a monitor with a comparable size to the one used in this work.

CHAPTER 6

Conclusion & Future Work

In this last chapter, we present our conclusion based on the conducted experiments and our measurements. Furthermore, we describe the limitations of our work and possible directions for future work.

6.1 Conclusion

In this work, we compare various real-time distortion correction methods for the perspective-correct rendering of 3D scenes on curved monitors using current consumer-grade hardware. Generally, real-time simulations and games should strive to support correct rendering for curved displays to create an accurate impression of the geometry in a scene. Rendering scenes with a corrected projection for ultra-wide curved monitors presents a significant improvement compared to using linear perspective on the same displays. It conserves the impression of straight lines and produces a correct perspective in the rendered scene. Based on our experiments, we are confident that consumer-grade head-tracking hardware is sufficiently precise to provide the user's viewpoint that is necessary for an exact correction. However, additional latency mitigation and calibration strategies should be implemented in the application when the primary goal is to provide a convincing FTVR experience.

Our measurements confirm that image-based methods overall achieve the best performance. Depending on the scene, our image-based implementation is faster by a factor of about 1.3–2.6 on average compared to the geometry-based implementation using hardware tessellation on our hardware. Hence, for performance-critical applications, image-based methods are the first choice for correcting the curvature of monitors. The main downside of image-based methods is the loss of sharpness from resampling the linear projection. Using multiple linear projections should be considered to decrease the required resolution for achieving acceptable quality or for covering a FOV greater than 180° [TNAM16, BS18].

Alternatively, geometry-based methods and hardware-accelerated ray tracing offer reasonable performance and do not suffer from image resampling artifacts. Generally, the performance of all rasterization-based methods is proportional to the geometric complexity of the scene. For the *UE4 Sun Temple* (0.6M triangles) [Gam17] and *Amazon Lumberyard Bistro* (2.8M triangles) scenes, the geometry-based methods create accurate results with a smaller overhead than ray tracing. Hardware-accelerated ray tracing, in contrast, scales well for complex static scenes consisting of many triangles by default due to the required acceleration structures. In a scene such as *NVIDIA Emerald Square* (10M triangles) [NHB17], ray tracing with 1 SPP can compete with the geometry-based methods performance-wise when no additional optimizations are implemented. Furthermore, the geometry-based methods are not entirely free from artifacts, and hardware limits regarding the tessellation level of tessellation shaders must be considered. In the case of large triangles, for example, used for walls and streets, these hardware limits can prevent a fine enough tessellation necessary for accurate results.

Our presented subdivision scheme based on a grid in screen space can be implemented in a single render pass using the graphics mesh pipeline to avoid the hardware limits of hardware tessellation while achieving similar quality. One advantage of this approach is that the grid is attached to the camera and does not depend on the input geometry. Triangles are subdivided along this grid exclusively, and the subdivision is not restricted by the hardware limits of the tessellation stage. Unfortunately, geometry far from the camera and small triangles that do not need further refinement are subdivided when they intersect with the grid. Hence, we suggest using this method for objects close to the camera and objects with coarse geometry, like buildings and infrastructure, only. Another positive property of the grid-based subdivision is that it reflects the geometry of the monitor better than the fixed tessellation patterns of hardware tessellation. As a result, fewer triangles are generated for coarse geometry. In the *UE4 Sun Temple* and *Amazon Lumberyard Bistro* scenes, the number of rasterized triangles is decreased by about 62%–71% with our grid-based subdivision compared to the hardware tessellation method. For the *NVIDIA Emerald Square* scene, we observe an increase in triangles of about 12% since most triangles are already sufficiently small. The reduction in triangles does not translate to performance, and our implementation is about 2.1–2.3 times slower across the scenes compared to hardware tessellation. Still, the presented single-pass software implementation using the graphics mesh pipeline achieves real-time performance and can significantly reduce the number of triangles generated for coarse geometry. We also recognize the potential for further optimizations by, for example, utilizing the mesh shader workgroups more efficiently. The experiments, therefore, demonstrate the potential for improved geometry-based methods on current hardware.

6.2 Future Work

Our implementation of the grid-based subdivision scheme is fast enough for usage in real-time applications but lags behind the implementation using hardware tessellation. Further performance optimizations to reduce the difference in frame time compared to

hardware tessellation are considered future work. We are convinced that the subdivision scheme can be optimized to get close to the performance of the presented hardware tessellation implementation. Low-level optimizations for a more efficient implementation of the triangle clipping in the shader can be performed [McG11]. Similar optimization can be applied throughout the shader code to tune the overall performance.

Distributing the work performed in the mesh shader across the invocations more efficiently is another goal for future work. One way to achieve this goal is to reduce the duplicated vertices on the boundaries of neighboring cells in the grid. Generating fewer vertices allows more invocations to process grid cells and generate triangles for the rasterizer. Sharing and synchronizing the vertex indices between the invocations to define the triangles may pose a challenge. Another option is to improve our conservative estimate of the number of triangles generated during the subdivision. Using more invocations than allowed by our upper bound still produces images without missing triangles and shows that our bounds are not optimal. Finding an upper bound closer to the actual number of generated triangles is an interesting problem and would allow utilizing the workgroups more efficiently.

There may also be potential for optimizing the grid used in our subdivision scheme. For example, sizing the cells based on a constant angle from the viewpoint instead of targeting a constant size in screen space may decrease the required grid resolution and reduce geometric artifacts. Note that such adaptations also require changes to the calculation of the bounding box of a triangle and might reduce performance.

A further compelling direction for future work is to combine multiple subdivision schemes using the graphics mesh pipeline. Implementing a classical tessellation and our grid-based subdivision in software allows using different methods depending on the distance and geometry of the object during the same render pass. Furthermore, a shader implementation of a classical tessellation algorithm may use non-uniform fractional tessellation where the tessellation pattern is spread non-uniformly over the triangle depending on the distance from the camera [MHAM08].

A notable limitation regarding our implementation of the image-based method is that we do not use multiple linear projections to approximate the curved screen. Doing so would reduce the required render resolution to achieve sharp images [TNAM16]. Nevertheless, the measurements in our experiment suggest that increasing the resolution alone is not the only way to reduce resampling artifacts. Developing more sophisticated filters with dynamically sized filter kernels that accurately reflect the distortion at every sample location might reduce the resampling artifacts further. Another interesting opportunity for further investigation is to improve the quality of image-based implementations using amortized supersampling [YNS+09], where the results of previous frames are reprojected and incorporated into the current frame to improve image quality.

Finally, while using the ꟻLIP error as an error metric works well for optimizing the parameters of the different methods in isolation, we think it is less valuable when comparing the differences across the methods. Slight differences from texture resampling

during the image-based method or from a different sampling pattern used during ray tracing result in a higher mean FLIP error compared to the geometry-based methods. At the same time, the image quality and sharpness are subjectively acceptable. This observation suggests that the FLIP error alone may not be a definite metric for reasoning about the quality of the different methods. Future work may focus on further comparisons with additional metrics or user studies.

List of Figures

List of Tables

List of Listings

Acronyms

API application programming interface. 10, 11

BLAS bottom-level acceleration structure. 10, 23

CPU central processing unit. 18

FOV field of view. 1–3, 5, 8, 9, 13, 19, 24, 26, 27, 31, 32, 36–38, 51, 67, 78, 83, 87

FPS frames per second. 8, 68, 72

FTVR Fish Tank Virtual Reality. 3, 9, 13, 53, 82, 83

GLSL OpenGL Shading Language. 29, 31, 93

GPU graphics processing unit. 9–12, 14, 18, 19, 21, 23, 29, 43, 78

HMD head-mounted display. 1, 3, 6, 7, 9

ORCA Open Research Content Archive. 18, 72

PPP Piecewise Perspective Projections. 8, 47, 49

SPP samples per pixel. 17, 55, 84

SSAA Supersampling Anti-Aliasing. 17, 18, 23, 55, 76

TLAS top-level acceleration structure. 10, 23, 72

UBO Uniform Buffer Object. 21, 54

UDP User Datagram Protocol. 54

Bibliography

[AG93] Pietro Acquisto and Eduard Gröller. A distortion camera for ray tracing. *WIT Transactions on Information and Communication Technologies*, 5, 1993.

[ALMM14] Jérôme Ardouin, Anatole Lécuyer, Maud Marchal, and Eric Marchand. Stereoscopic rendering of virtual environments with wide field-of-views up to 360. In *2014 IEEE Virtual Reality (VR)*, pages 3–8. IEEE, 2014.

[AM01] Tomas Akenine-Möllser. Fast 3d triangle-box overlap testing. *Journal of Graphics Tools*, 6(1):29–33, 2001.

[ANAM+20] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D Fairchild. Flip: A difference evaluator for alternating images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 3(2), August 2020.

[Bay95] Salvador Bayarri. Computing non-planar perspectives in real time. *Computers & Graphics*, 19(3):431–440, 1995.

[Bou04] Paul Bourke. Offaxis fisheye projection. `http://paulbourke.net/dome/fisheye`, 2004. Last accessed on May 19, 2022.

[Bou09] Paul Bourke. idome: Immersive gaming with the unity3d game engine. *Computer Games and Allied Technology*, 9, 2009.

[Bou15] Jean-Yves Bouguet. Camera calibration toolbox for matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/index.html`, 2015. Last accessed on May 19, 2022.

[BS18] Swaroop Bhonde and Mahalakshmi Shanmugam. Turing multi-view rendering in vrworks. `https://developer.nvidia.com/blog/turing-multi-view-rendering-vrworks/`, 2018. Last accessed on May 20, 2022.

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. *SIGGRAPH Comput. Graph.*, 18(3):207–212, January 1984.

[Der99] Ton Derksen. Discovery of linear perspective and its limitations. *Philosophica*, 63, January 1999.

[DH11] Rubén Díaz Hernández. Rendering stereoscopic 3d images in cylindrical spaces. Master's thesis, 2011.

[Fle95] Margaret M Fleck. Perspective projection: the wrong imaging model. *Department of Computer Science, University of Iowa*, pages 1–27, 1995.

[FSD+19] Dylan Fafard, Ian Stavness, Martin Dechant, Regan Mandryk, Qian Zhou, and Sidney Fels. Ftvr in vr: Evaluation of 3d perception with a simulated volumetric fish-tank virtual reality display. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[Gam17] Epic Games. Unreal engine sun temple, open research content archive (orca). `http://developer.nvidia.com/orca/epic-games-sun-temple`, October 2017. Last accessed on May 19, 2022.

[GG99] Georg Glaeser and Eduard Gröller. Fast generation of curved perspectives for ultra-wide-angle lenses in vr applications. *The Visual Computer*, 15(7):365–376, 1999.

[GHFP08] Jean-Dominique Gascuel, Nicolas Holzschuch, Gabriel Fournier, and Bernard Peroche. Fast non-linear projections using graphics hardware. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 107–114, 2008.

[Gib60] James J Gibson. Pictures, perspective, and perception. *Daedalus*, 89(1):216–227, 1960.

[HB16] Rolf R Hainich and Oliver Bimber. *Displays: Fundamentals & Applications*. AK Peters/CRC Press, 2016.

[Hen18] Neil Henning. Vulkan subgroup tutorial. `https://www.khronos.org/blog/vulkan-subgroup-tutorial`, 2018. Last accessed on May 23, 2022.

[HM91] Paul S. Heckbert and Henry P. Moreton. Interpolation for polygon texture mapping and shading. *State of the art in Computer graphics: Visualization and Modeling*, pages 101–111, 1991.

[HTB+] Stanislaw Halik, Chris Thompson, Donovan Baarda, Xavier Hallade, Michael Welter, Attila Csipa, Wei Shuai, and Stéphane Lenclud. opentrack. `https://github.com/opentrack/opentrack`. Last accessed on July 2, 2022.

[Ige99] Homan Igehy. Tracing ray differentials. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 179–186, 1999.

[JMN+08] Kensei Jo, Kouta Minamizawa, Hideaki Nii, Naoki Kawakami, and Susumu Tachi. A gpu-based real-time rendering method for immersive stereoscopic displays. In *SIGGRAPH Posters*, page 111, 2008.

[KBR19] John Kessenich, Dave Baldwin, and Randi Rost. The opengl® shading language, version 4.60.7. `https://registry.khronos.org/OpenGL/specs/gl/`, 2019.

[Koo09] Robert Kooima. Generalized perspective projection. 2009.

[Kub18] Christoph Kubisch. Introduction to turing mesh shaders. `https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/`, 2018. Last accessed on May 19, 2022.

[Kub20] Christoph Kubisch. Using mesh shaders for professional graphics. `https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/`, 2020. Last accessed on June 29, 2022.

[LD09] Haik Lorenz and Jürgen Döllner. Real-time piecewise perspective projections. In *GRAPP*, pages 147–155, 2009.

[LDP+02] J.J.-W. Lin, H.B.L. Duh, D.E. Parker, H. Abi-Rached, and T.A. Furness. Effects of field of view on presence, enjoyment, memory, and simulator sickness in a virtual environment. In *Proceedings IEEE Virtual Reality 2002*, pages 164–171, 2002.

[LGB22] Martin-Karl Lefrançois, Pascal Gautron, and David Bickford, Neil amd Akeley. Nvidia vulkan ray tracing tutorial. `https://nvpro-samples.github.io/vk_raytracing_tutorial_KHR/`, 2022. Last accessed on May 23, 2022.

[Lum17] Amazon Lumberyard. Amazon lumberyard bistro, open research content archive (orca). `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`, July 2017. Last accessed on May 19, 2022.

[McG11] Morgan McGuire. Efficient triangle and quadrilateral clipping within shaders. *Journal of Graphics, GPU, and Game Tools*, 15(4):216–224, 2011.

[MGN18] Grzegorz Muszyński, Krzysztof Guzek, and Piotr Napieralski. Wide field of view projection using rasterization. In *Multimedia and Network Information*

Systems (MISSI 2018), pages 586–595. Springer International Publishing, August 2018.

[MHAM08] Jacob Munkberg, Jon Hasselgren, and Tomas Akenine-Möller. Non-uniform fractional tessellation. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 41–45, 2008.

[MMSB19] Jonathan Martschinke, Jana Martschinke, Marc Stamminger, and Frank Bauer. Gaze-dependent distortion correction for thick lenses in hmds. In *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pages 1848–1851. IEEE, 2019.

[Mor01] Henry Moreton. Watertight tessellation using forward differencing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 25–32, 2001.

[NHB17] Kate Anderson Nicholas Hull and Nir Benty. Nvidia emerald square, open research content archive (orca). `http://developer.nvidia.com/orca/nvidia-emerald-square`, July 2017. Last accessed on May 19, 2022.

[NVI18] NVIDIA Corporation. Nvidia turing gpu architecture: Graphics reinvented. `https://www.nvidia.com/en-us/design-visualization/technologies/turing-architecture/`, 2018. Last accessed on July 13, 2022.

[NVI22] NVIDIA Corporation. Orca: Open research content archive. `https://developer.nvidia.com/orca`, 2022. Last accessed on June 12, 2022.

[PGRS13] Kevin Ponto, Michael Gleicher, Robert G Radwin, and Hyun Joon Shin. Perceptual calibration for immersive display environments. *IEEE transactions on visualization and computer graphics*, 19(4):691–700, 2013.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[PJB13] Daniel Pohl, Gregory S Johnson, and Timo Bolkart. Improved pre-warping for wide angle, head mounted displays. In *Proceedings of the 19th ACM symposium on Virtual Reality Software and Technology*, pages 259–262, 2013.

[PRO19] Mariano Pérez, Silvia Rueda, and Juan M Orduña. Geometry-based methods for general non-planar perspective projections on curved displays. *The Journal of Supercomputing*, 75(3):1241–1255, 2019.

[Reh03] Klaus Rehkämper. What you see is what you get: The problems of linear perspective. *Looking into pictures*, pages 179–190, 2003.

[RG79] Brian Rogers and Maureen Graham. Motion parallax as an independent cue for depth perception. *Perception*, 8(2):125–134, 1979.

[RVSS10] Carlos Ricolfe-Viala and Antonio-José Sánchez-Salmerón. Correcting non-linear lens distortion in cameras without using a model. *Optics & Laser Technology*, 42(4):628–639, 2010.

[Sal06] David Salomon. *Transformations and Projections in Computer Graphics*, volume 233. Springer, 2006.

[SBGS06] Martin Spindler, Marco Bubke, Tobias Germer, and Thomas Strothotte. Camera textures. In *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia 2006*, pages 295–302. Association for Computing Machinery, 2006.

[SH74] Ivan E. Sutherland and Gary W. Hodgman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.

[Smi87] Ray Smith. *The artist's handbook*. Knopf, 1987.

[TD08] Matthias Trapp and Jürgen Döllner. A generalization approach for 3d viewing deformations of single-center projections. *GRAPP*, (3):162–170, 2008.

[The22] The Khronos® Vulkan Working Group. Vulkan® 1.3.212 - a specification (with all registered vulkan extensions). `https://registry.khronos.org/vulkan/specs/1.3-extensions/`, 2022.

[TNAM16] Robert Toth, Jim Nilsson, and Tomas Akenine-Möller. Comparison of projection methods for rendering virtual reality. In *Proceedings of High Performance Graphics*, pages 163–171. 2016.

[Tob] Tobii AB. Tobii eye tracker 5. `https://gaming.tobii.com/product/eye-tracker-5/`. Last accessed on July 2, 2022.

[WAB93] Colin Ware, Kevin Arthur, and Kellogg S Booth. Fish tank virtual reality. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 37–42, 1993.

[WH95] Benjamin Allen Watson and Larry F. Hodges. Using texture maps to correct for optical distortion in head-mounted displays. In *Proceedings Virtual Reality Annual International Symposium'95*, pages 172–178. IEEE, 1995.

[Wil22] Sascha Willems. Vulkan hardware database. `https://vulkan.gpuinfo.org/`, 2022. Last accessed on June 14, 2022.

[XHH+21] Jianghao Xiong, En-Lin Hsiang, Ziqian He, Tao Zhan, and Shin-Tson Wu. Augmented reality and virtual reality displays: emerging technologies and future perspectives. *Light: Science & Applications*, 10(1):1–30, 2021.

[YNS+09] Lei Yang, Diego Nehab, Pedro V. Sander, Pitchaya Sitthi-amorn, Jason Lawrence, and Hugues Hoppe. Amortized supersampling. *ACM Transactions on Graphics*, 28(5):1–12, December 2009.

[ZYX+20] Tao Zhan, Kun Yin, Jianghao Xiong, Ziqian He, and Shin-Tson Wu. Augmented reality and virtual reality displays: Perspectives and challenges. *iScience*, 23(8), 2020.