Markus Prettner

Nonlinear Rasterization in a Software Graphics Pipeline

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Computer Science

submitted to

Graz University of Technology

Supervisor

Prof. Dr. Markus Steinberger
Institute for Computer Graphics and Vision

Graz, Austria, Oct. 2020

Abstract

Traditional graphics hardware can only do regular, linear rasterization, meaning that the distance between two neighboring pixels sample locations is uniform over the entire image. This works well for displaying images on rectangular screens but not so well for surfaces that need to display a non-rectangular image, which most prominently is the case in virtual reality headsets and various shadow mapping techniques. Existing solutions to this problem lack in either quality, performance, or require advanced hardware features. We propose a rasterizer design that can do nonlinear rasterization without storing an intermediate image while delivering good quality and competitive performance.

Kurzfassung

Herkömmliche Grafikhardware kann nur für lineare Rasterisierung verwendet werden, das heißt, dass der Abstand zwischen den Abtastpunkten zweier benachbarter Pixel überall auf dem Bild gleich sein muss. Das funktioniert sehr gut, um Bilder auf rechteckigen Bildschirmen anzuzeigen, jedoch eher schlecht, wenn kein rechteckiges Bild dargestellt werden soll, was bei Virtual Reality Headsets meistens und bei Shadowmapping häufig vorkommt. Bei existierenden Lösungen zu diesem Problem mangelt es entweder an Qualität, Performance oder es werden sehr moderne Hardwarefunktionen benötigt. Wir beschreiben eine Rasterisierungspipeline mit der man nichtlineare Verzerrungen darstellen kann, ohne dabei auf ein zwischengespeichertes, normal rasterisiertes Bild zurückzugreifen. Außerdem liefert unsere Pipeline hohe visuelle Qualität mit vergleichsweise guter Performance.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

______________________ ____________________________

Date Signature

Acknowledgments

First and foremost, I would like to thank Prof. Steinberger for introducing me to this topic and taking the role as my supervisor very serious, and always being helpful and supporting. Even after one and a half years of constant ups and downs, including prolonged periods of little progress, he managed to keep me motivated which allowed me to finish in a timely fashion. Since he also supervised my bachelor thesis, I was well aware that doing the bare minimum of work will not suffice, and I am glad to say that this choice was not a mistake this time either.

A special thanks goes to my parents, who supported my ambitious goal of studying Computer Science in a timely manner from the beginning, and while the planned timeframe for finishing university got prolonged by about a year, they never stopped believing in me, although the questions about when I will finish my thesis were asked less and less often the later it became.

I also would like to thank my brother, Florian, for being such a great role model throughout my entire school and university education. Having someone so ambitious and determined as a close relative really did a lot towards setting my goals to more than just average. Even though I was not able to finish my degree as fast as he did, I firmly believe that his contributions to me finishing within a reasonable timeframe were significant.

Finally, I am very grateful to my good friends Benjamin, Christian, Wolfgang, Anneliese and Kirsten. All of you have been great companions since I started studying in Graz, and without anyone of you, the experience of living and studying in Graz would not have been the same. You all are awesome.

Contents

1 Introduction **1**
1.1 CUDA . . . 2
1.2 OpenGL pipeline . . . 3
1.3 Rendering techniques . . . 5

2 Related work **7**

3 Pipeline design **9**
3.1 Rasterizer design . . . 10
3.2 Experimentation Pipeline . . . 11

4 Nonlinear rendering **15**
4.1 Rectilinear distortion . . . 16
4.2 Bijective distortion . . . 17
4.3 Forward and inverse distortions . . . 19
4.4 Nonlinear pipeline modifications . . . 20

5 Performance evaluation **25**
5.1 Test setup . . . 25
5.2 Real-world scenes . . . 25
5.3 Artificial scenes . . . 26
5.4 Distortion tiers . . . 28
5.5 Remarks . . . 31

6 Conclusion **33**

Bibliography **35**

List of Figures

1.1 CUDA architecture . . . 2
1.2 CUDA Grids, Blocks and Threads . . . 3
1.3 OpenGL graphics pipeline . . . 4
1.4 Simple depiction of rasterization . . . 5

3.1 High-level overview of a sort-middle architecture . . . 10
3.2 Clipping and guard band . . . 11
3.3 Scanline rasterization . . . 13

4.1 Forward and inverse rectilinear distortion . . . 16
4.2 Cube with rectilinear distortion being applied . . . 17
4.3 Forward and inverse radial distortion . . . 18
4.4 Cube with radial distortion being applied . . . 18
4.5 Pixel displacement error with radial distortion . . . 19
4.6 Forward and inverse distortion polynomials . . . 20
4.7 Maximum error of forward - inverse distortion . . . 21
4.8 Scanline rasterization for radial distortion . . . 24

5.1 Rendering time comparison between implementations for real-world scenes, 1920×1080, 1080Ti . . . 26
5.2 Rendering time comparison between implementations for real-world scenes, 3840×2160, 1080Ti . . . 27
5.3 Rendering time comparison between implementations for real-world scenes, 1920×1080, 2080Ti . . . 27
5.4 Rendering time comparison between implementations for real-world scenes, 3840×2160, 2080Ti . . . 28

5.5 Rendering time comparison between implementations for artificial test scenes, 1920×1080, 1080Ti 29
5.6 Rendering time comparison between implementations for artificial test scenes, 3840×2160, 1080Ti 29
5.7 Rendering time comparison between implementations for artificial test scenes, 1920×1080, 2080Ti 30
5.8 Rendering time comparison between implementations for artificial test scenes, 3840×2160, 2080Ti 30
5.9 Distortion tiers performance comparison 31

1 Introduction

In the early years of computer graphics, every part of the graphics pipeline was processed on the Central Processing Unit (CPU). This meant that modifications to the pipeline could be easily made by changing code, but the performance left much to be desired. Therefore, specialized hardware was invented to reduce CPU bottlenecks when dealing with graphics data and thus increase performance. This specialized hardware was further developed over time, until it became what we call Graphics Processing Unit (GPU) today. On the other hand, this reduced flexibility, since changes to the hardware parts of the pipeline were no longer easily possible. The graphics pipelines, which are predominantly in use today, use hardware acceleration for many of their compute-heavy tasks. There are, however, parts of the pipeline that are programmable. These are called shaders, and can be useful for a great variety of graphical effects and optimizations, but their use cases are limited by constraints set by the pipeline. Technical advancements in this field are limited by the speed of the hardware design processes, which may not be optimal in a ever-changing domain. Modifications to the pipeline are also subject to the same limitations, meaning that use cases that do not fit the predefined pipeline can simply not be implemented at all.

One way to achieve greater flexibility would be to implement the pipeline entirely in software, as it was before GPUs. Every step of the pipeline would be freely programmable, modifications would be as easy as changing some code. This can be used to test potential features for future hardware, as changing code is much easier than designing hardware for the same task. Although modern GPUs can be used in compute mode, which means that they act similar to a regular CPU, which can then be used to implement software pipelines, there is still a large performance penalty in comparison to a hybrid pipeline. There have been many implementations of such a software pipeline, such as cuRE by Kenzel et al. [9], FreePipe by Liu et al. [12] and CudaRaster by Laine et al. [11]. In our work, we implemented such a software rasterization pipeline with a focus on a customizable rasterization stage.

1.1 CUDA

To better understand the following chapters, it is advantageous to know the fundamentals of CUDA programming in terms of modern Nvidia GPUs. A more detailed description of the following, shortened explanations was written by Cook [5], with the official documentation by Nvidia [15] focusing heavily on the programming side. Fundamentally, a GPU is a coprocessor with a focus on massively parallel computation employing a throughput-oriented design.

From the top down, the graphics processor is made up of several multiprocessors and memory which can be accessed from within all multiprocessors, the so-called global memory. Each multiprocessor has its own exclusive memory, which is called shared memory, and a number of single-instruction, multiple-data (SIMD) processors. A visual representation of these hierarchical relationships is depicted in Figure 1.1.

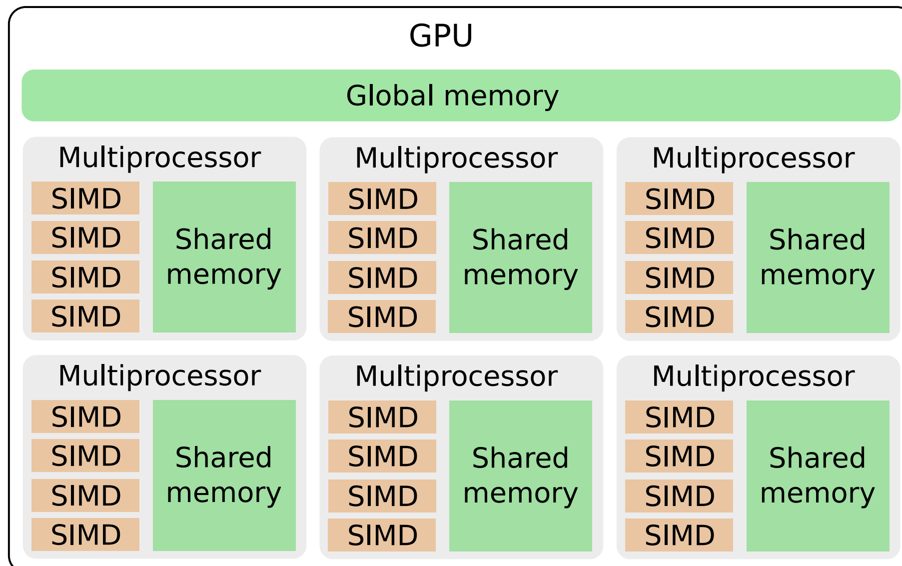


Figure 1.1: High-level overview of the Nvidia GPU architecture. A multiprocessor is made up of SIMD processors and shared memory, the GPU of global memory and multiple multiprocessors.

The developer now needs to divide the compute workload into meaningful chunks that fit the hardware architecture. The abstractions for this are called grid, blocks and threads, which are depicted in Figure 1.2. A grid is made up of multiple blocks, and each block is further subdivided into threads, whereas each block has the same number of threads. Design considerations here are that efficient communication is only possible within a block, but using only a single block would severely reduce overall performance as each block is executed on only one multiprocessor. The threads of a block are partitioned in groups of 32, called warps, and executed on the multiprocessor. All active threads within a warp have to simultaneously execute the same instruction, meaning that if one thread needs to wait for data to become available, e.g. due to a cache miss, all other threads have

to wait as well. If code paths of groups of one or more threads within a warp diverge, these groups have to be executed in series, with all other threads being deactivated in the meantime, until all code paths merge again. This behavior is called thread divergence, and it is unwanted because it reduces the amount of calculations per clock cycle, making the implementation less efficient.

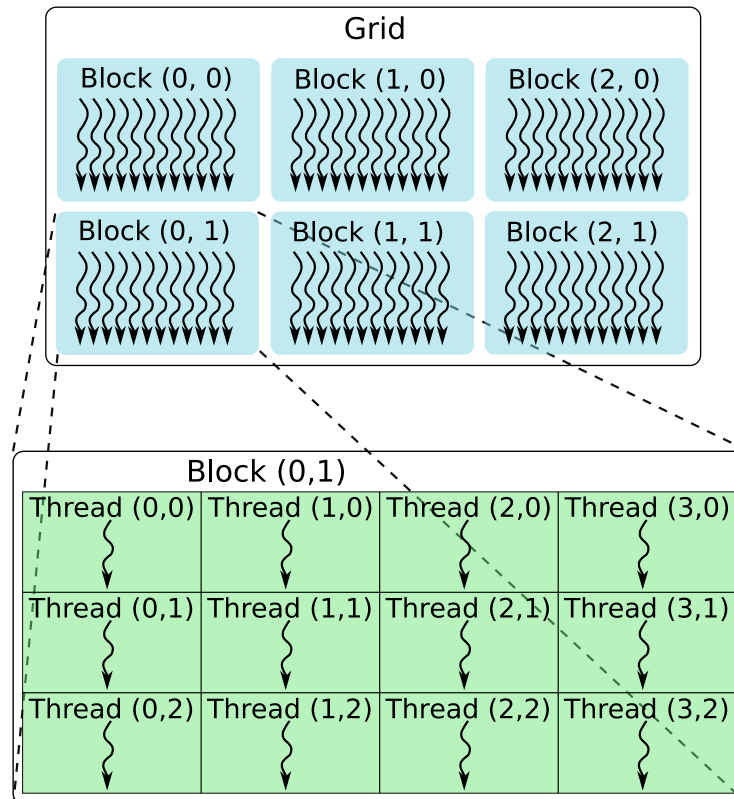


Figure 1.2: High-level overview of the CUDA programming model. A grid is composed of multiple blocks, and each block contains an equal number of threads. When executing the grid, each block is run on a single multiprocessor. Groups of 32 threads, which are called warps, are run in sequence on the multiprocessor.

1.2 OpenGL pipeline

In a similar way, it is also beneficial to have a rough understanding of how the graphics pipeline works and what stages it is comprised of. A detailed description of the entire OpenGL graphics pipeline is given in the OpenGL specification [22]. For better comprehension, the following descriptions are illustrated in Figure 1.3.

The host application starts the pipeline by feeding primitives, such as triangles or quads, into the pipeline. At first, the vertices of the primitives are processed in the vertex shading stage, which uses the programmable vertex shader to allow for fine control

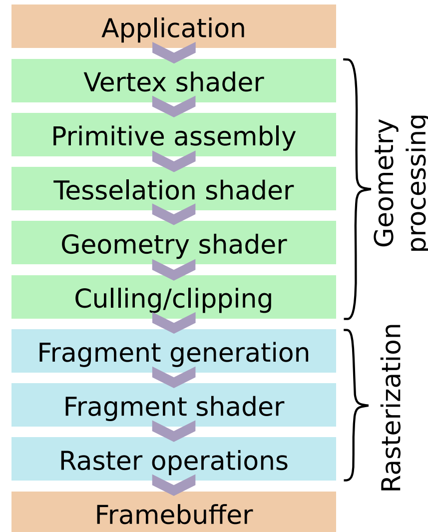


Figure 1.3: High-level overview of the stages of the OpenGL graphics pipeline. Starting from the host application, vertex data is propagated through various geometry and rasterization stages until it reaches the framebuffer.

over how the vertex positions and attributes are calculated. The following, fixed-function primitive assembly stage assembles the primitives in accordance with the specified format of the input data. The two following, optional, stages work on a per-primitive basis and can be used to generate additional geometry. Tesselation, which is comprised of a programmable control shader, a programmable evaluation shader, and the fixed-function primitive generation in between, is the first optional stage. The important distinction to the following stage is the presence of adjacency information, which is commonly used for variable subdivision, which, in turn, can be used to dynamically adjust the level of detail of geometry or to perform displacement mapping. The geometry shading stage can also be used to generate additional geometry, but since the programmable geometry shader does not have access to adjacency information, its use cases are more limited

To finish geometry processing, culling and clipping is performed. Culling means that primitives that are not visible at all are discarded, either because they are completely outside of the visible area, or optionally, are facing away from the camera. Clipping deals with primitives that are partially visible, for example if they pass through the near or far plane. Depending on the implementation, it might additionally be necessary that primitives do not extend too far from the visible area due to floating point or fixed-point precision. The fixed-function rasterizer then creates fragments for the transformed, culled and clipped primitives, and interpolates vertex attributes to be used in the fragment shading stage, which in turn calls the programmable fragment shader on each fragment to determine its color and additional attributes of interest. At the very end, raster operations are performed, which consist mainly of fragments being written to the framebuffer, with various pipeline settings determining the result of two or more fragments claiming the

same location.

1.3 Rendering techniques

Rendering of computer-generated imagery can be classified into two large categories: Ray tracing and rasterization. Traditionally, rasterization was used for real-time rendering and ray tracing for offline high quality rendering. Ray tracing used for rendering, which was first described by Appel [1], is a highly flexible algorithm that can be used to determine arbitrary point-to-point visibility. By extending the algorithm to cast additional, recursive rays from the hit surface, Whitted [25] gave rise to simple and elegant high-quality rendering techniques, such as path tracing, which is a widely-used variant in use today. To make ray tracing efficient, spacial acceleration structures, such a bounding volume hierarchies, are required.

Rasterization, on the other hand, is mostly used for determining the primary visibility of geometry, although tricks also allow for secondary effects, such as shadow maps and the like. In comparison to image order ray tracing, a rasterization pipeline employs an object order method, which means that every primitive is touched only once, projected to screen, determines the covered pixels (the actual rasterization), determines the color for each pixel and resolves the image using depth buffering. While the typical real-time rendering pipeline has evolved over the years, the basic concept of rasterization, as described by Shirley and Marschner [23], and depicted in Figure 1.4, is still the same.

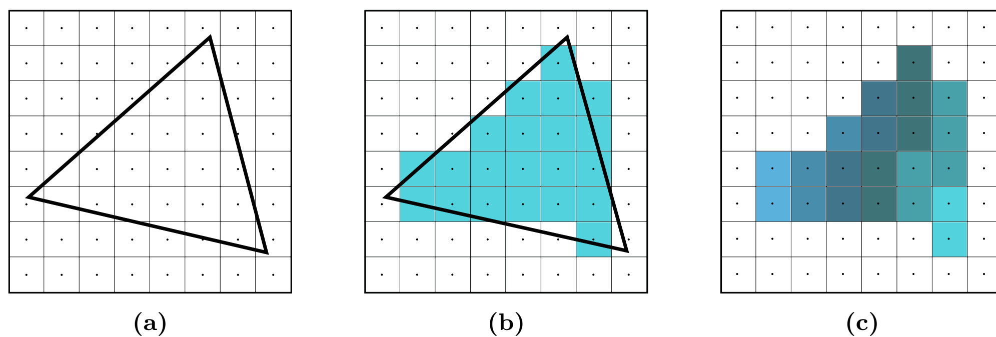


Figure 1.4: Simplified depiction of the rasterization algorithm. Starting with a triangle in screen-space coordinates (1.4a), the fragments whose centers are covered by the triangle are filled (1.4b) and shaded (1.4c).

The popularity of rasterization pipelines comes from the fact that they are efficient: Memory is accessed efficiently, every primitive is loaded only once from memory. Shading execution is well suited for SIMD execution as all fragments of a primitive are generated together. Preprocessing of the data is not needed, it can be passed to the rasterization pipeline unmodified, there is no need for an acceleration data structure, thus all types of dynamic scenes can be handled. The pipeline design is efficient, temporary data can be stored in on-chip buffers, thus data amplification can be dealt with in a simple and efficient

manner. The hardware for basic math operations is efficient, which is used extensively in the rasterizer for e.g., pixel coverage tests.

Ray tracing has different strengths: There is no overdraw, although determining the first hit may be more costly than determining any hit. Also, rays can be shot in arbitrary directions, which are not bound to any fixed pattern, such as a raster.

Aiming for higher and higher quality images also in real-time rendering, ray tracing is by now not only suppoerted in hardware, but also used in hybrid rendering pipelines for effects such as reflections, shadows, and global illumination approximations. In the future, we may see ray/path tracing take an even bigger hold in real-time rendering, especially if we reach the compute resources to apply full path tracing. While ray tracing seems inevitable to determine secondary light paths, ray-tracing is also being increasingly used for primary rays [8].

Ray tracing is attractive for primary rays when rasterization reaches its limits, for example in head-mounted displays (HMDs) where lens distortion leads to non-linear distortion and, thus, cannot be rasterized directly [24], or when targeting foveated rendering [18] where many pixels are needed for the current focal point and only few in the periphery. Paradoxically, rendering on mobile devices, such as HMDs, or when performing foveated rendering, one's goal is to render as efficiently as possible and, thus, rasterization seems to be a more natural choice, if it was not for the limitations of rasterization.

In this paper, we analyze the possibilities of non-linear rasterization and propose an architecture that is applicable to different distortion functions, we make the following contributions:

- A software rasterization pipeline with programmable rasterizer, which is able to achieve interactive framerates
- A rectilinear and bijective rasterizer implementation with efficient hierarchical rasterization, with the bijective rasterizer being able to render images with lens distortion and foveation.

2 Related work

Previous work on dual-paraboloid shadow mapping as described originally by Brabec et al. [3] is based on the idea that if geometry is tesselated finer and finer, with a nonlinear transformation being applied to the tesselated vertices, the rendered image looks more and more correct. If the tesselation is fine enough, this approximation makes the visual artifacts vanish completely, making the image indistinguishable from a mathematically correct rendering. Advancements in graphics hardware and further investigations lead to practical implementations of this technique using hardware tessellation, as described by Osman et al. [17]. The tessellation requirement, however, still leads to a performance overhead and makes this technique unavailable to platforms that do not support hardware tessellation. In his work on making shadow mapping more efficient while maintaining graphical fidelity, Rosen [21] also relies on tessellation to approximate more expensive calculations for scenes with a high number of triangles. Depending on the tested scene, the rendering time can easily double compared to not using tessellation, when using the recommended number of triangle subdivisions for a good tradeoff between performance and quality. The ability to rasterize nonlinear projections without generating additional triangles could lead to performance improvements for this type of application.

To the best of our knowledge, there exists no similar pipeline modification for any of the GPU-based software graphics pipelines that we considered for comparison, namely FreePipe by Liu et al. [12], CUDARaster by Laine et al. [11] and Piko by Patney et al. [19]. Therefore, results are later, among others, compared to a hardware pipeline where distortion happens in a post-processing stage.

Perceptual rasterization by Friston et al. [6] also aims at improving the experience of virtual reality (VR) applications, and does so by using nonlinear rendering and foveation. Foveated rendering means that the image is rasterized with different resolutions depending on the on-screen distance to the point where the eye is currently looking at. This significantly reduces the number of pixels to be drawn, but also requires eye tracking, which is not a common feature in contemporary consumer HMDs. The nonlinear rendering part of this work can be used irregardless of the presence of eye tracking.

Modifications to the hardware graphics pipeline have previously, among others, been proposed by Lloyd et al. [13]. In their work they propose to adapt the hardware rasterizer to allow for logarithmic rasterization, which they demonstrated can be used for less bandwidth-heavy shadow map calculations while maintaining the same shadow quality. The pipeline modifications that we describe in section 4.1 can also be used to implement logarithmic rasterization as described in their work, since it is a version of rectilinear distortion where one dimension is not distorted while the other is logarithmically distorted.

Raytracing, which predates programmable graphics hardware by many years [20], can also be used to achieve the same effects as described in this work. Until recently, raytracing had the significant disadvantage of not having the possibility to resort to hardware acceleration, which did not allow for competitive rendering times. Since the introduction of hardware raytracing in newer Nvidia GPUs [7] this is now also a feasable method for realtime applications. The downside of needing efficient acceleration structures, which is necessary to achieve interactive frame rates, still persists. The time required to recreate acceleration structures in non-static scenes can add a significant portion of the total rendering time. Rasterization, on the other hand, naturally handles arbitrarily dynamic scenes, since the raw mesh data is processed by the pipeline.

3 Pipeline design

First, we discuss the basic design of rasterization pipelines and the underlying ideas of going from a linear rasterizer to a non-linear rasterizer.

Previous approaches to non-linear rasterization either distort the geometry or perform some form of raytracing and thus build a hybrid rasterizer. Distorting geometry essentially comes down to applying distortion functions to the triangles. Typically a distortion function is a simple function in the form

$$[x', y']^T = f(x, y), \tag{3.1}$$

with x and y being the original screen-space coordinates, and x' and y' the distorted coordinates. The function f uniquely maps each two-dimensional coordinate to another two-dimensional coordinate. This distortion function not only distorts the vertices, but may also turns straight edges into curves. As a result, traditional techniques cannot be applied. If triangles are small enough, the distortions may become small enough for the nonlinearities introduced on top of the edges to be tolerable. However, this puts unnecessary high loads on temporary memory bandwidth, rasterization loads and complicated shading, such as quad shading because of generating large numbers of triangles. Using per-triangle raytracing for rasterization, as Friston et al. [6] did, requires the rasterizer to run on a bounding shape around the distorted triangle and perform pixel tests using raytracing in the fragment shader, leading to excessive number of fragments being generated an discarded.

Our approach, on the other hand, achieves the same result by moving the sample location of each fragment according to the underlying distortion function before evaluating the fragment. By doing this on a per-fragment basis, we do not need to change the geometry in any way, we only need to account for nonlinearities in some of the pipeline stages. To verify that this approach makes sense from a performance point of view, we evaluated the performance in chapter 5.

3.1 Rasterizer design

In accordance with the models described by Molnar et al. [14], our pipeline is classified as a sort-middle pipeline. This means that the sort from object space to screen space happens in the middle of the pipeline, between the geometry and rasterizer stage. We chose this model because it is the most common for parallel rendering systems, and fits nicely between the disadvantages of the susceptibility of sort-first to load imbalance because of adverse primitive distribution and the potentially very high pixel traffic of the sort-last approach. A high-level overview of our pipeline can be seen in Figure 3.1.

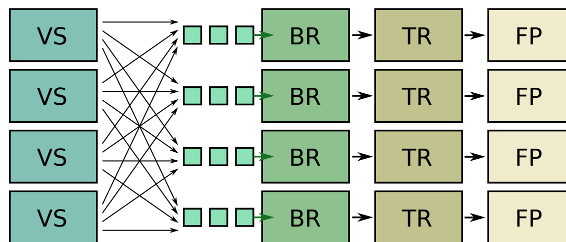


Figure 3.1: High-level overview of our sort-middle pipeline. Global work distribution only takes place right between the geometry stage, which contains the vertex shader (VS), and the rasterization stages. During the following bin rasterizer (BR), tile rasterizer (TR) and fragment processing (FP) there is only local work distribution.

Geometry Processing Geometry processing operates on input primitives, performing many different types of geometric transformations, including projection, and outputs vertices in clip space and indices. This part of the pipeline can contain steps like tessellation, geometry shading, or even mesh shaders. The only requirement is on the output. Thus, various transformations are typically happening during geometry processing, such as skinning or displacement maps. This part of the pipeline is not affected by the type of projection.

Right at the end of geometry processing, operations related to the rasterizer are performed on a per-primitive and per vertex basis:

Culling removes triangles that are outside of the view frustum, modern pipelines may do so in a mesh shader already.

Clipping cuts triangles if they cannot be handled as a whole, for example if they reach behind the near/camera plane. Furthermore clipping may also be necessary if fixed-point conversion is otherwise not possible. Fixed-point arithmetic only works within a pre-defined range – the range supported by the chosen format. This supported range is typically called guard band, see Figure 3.2. If a triangle reaches out of the guard band it is clipped and only the inside part is forwarded to the next stage. Clipping may produce one or two triangles.

Fixed-point conversion takes the projected vertex coordinates and snaps them to the fixed-point raster. Typically one works with 256, i.e., 8 bits, between pixel locations.

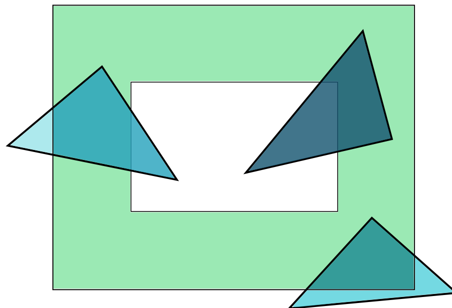


Figure 3.2: Visualization of clipping and guard band. The viewport is the white space in the center, the light green area around it is the guard band. Everything outside of the guard band, visualized as the slightly brighter areas of the triangles, is removed, i.e., clipped.

Finally, to cater for parallelism and data locality, primitives are assigned to screen space bins or tiles, each being assigned to a specific rasterizer. Note that this is both true for desktop devices with fine-granular rasterizer patterns [10] and mobile devices which implement a tiled rasterizer [4].

Fragment Processing The last step of rasterization is fragment processing, which generates all fragments covered by a primitive. Note that additional helper fragments may be generated to support derivative computations, which results in so-called quad shading. For each fragment, a fragment shader is called which produces the output color of the fragment. Finally the fragment is subject to depth testing and potentially blending with the already present color in the framebuffer to adjust the color in the frame buffer. To reduce the number of fragment shader invocations which do not contribute to the output color, as they are discarded by depth testing, fragments, or even entire groups of fragments, may be discarded before fragment shading or during rasterization in so called early-depth testing. To this end, a depth value closer to the camera than the newly generated fragment(s) must have been generated beforehand.

Our non-linear rasterization approaches do not affect any parts of the above mentioned processing after rasterization. We only change parts of the pipeline at the end of geometry processing and in the rasterizer itself.

3.2 Experimentation Pipeline

To allow testing of our approach, we implemented our approach as a parallel software rasterizer, running on the programmable cores of current GPUs. Our approach follows the previous software rasterization designs [9, 11, 19]. Our rasterizer follows a tiled hierarchical rasterization approach in four stages:

1. Geometry processing
2. Load-balancing

3. Bin-to-tile rasterization
4. Tile-to-fragment rasterization and fragment processing

Geometry processing follows the previously discussed general geometry processing processing strategies, with rasterization preparation steps at the end. In our simple setup geometry processing takes and index and vertex buffer as input, starts one thread per triangle, fetches the referenced vertices, performs a matrix multiplication to go from object to clip space for each triangle, and hands the triangle with its attributes over to the system.

After culling and clipping we store each surviving or generated triangle into a single global triangle queue. Additionally, we assign the triangle to a raster of fixed size bins. For our linear baseline, we use bins of 64×64 pixels as bins which are organized regularly with the screen center being located at $[0, 0]$ and touches four bins. For efficient bin assignment, we take the bounding box of every triangle and insert the triangle into the queue associated with each bin the bounding box touches. The queues only store indices to the global triangle queue.

Load-balancing ensure that the following rasterization stages can operate in limited memory. We use tiles of 8×8 pixels. Using another set of queues for each tile on screen may use significant amounts of memory, one of the issues of CUDARaster [11] and Piko [19] according to Kenzel et al [9]. To avoid this issue, we, for each bin, compute the worst case number of tile entries that may be produced, i.e., the number of bin queue entries multiplied by 64. We then use a modified binary reduction to merge bins that can run concurrently while staying within a given maximum amount of memory. Essentially, we just combine neighboring bins until their sum of bin queue entries exceeds a threshold. Combining this binary reduction with a prefix sum, we can directly compute the offset of each bin within the the combined set of bins. The result of this step is a list of grouped bins that can run concurrently. For small scenes, all bins may be able to run concurrently, for extreme cases every bin may need to run separately.

The bin-to-tile rasterizer takes all triangles associated with the bin as input and assigns each triangle to all tiles it covers. For efficient processing, we use one thread per triangle and create a bitmask of covered tiles, before we insert the triangle ids into the respective tile queues. For efficient rasterization, we follow the approach outlined by Kenzel et al. [9], stepping along each edge in y-direction and masking entire rows of pixels in the bit mask. The process is outlined in Figure 3.3. While Kenzel et al. [9] operate in floating point, we perform these computations in fixed-point.

The tile rasterizer takes each triangle assigned to the tile as input and computes the triangle coverage and shading in parallel. We use a warp per triangle, with a tile size of 8×8 each thread is responsible for exactly two fragments. Each thread fetches the fixed-point vertex positions, computes the edge equations and inserts its pixel values into the edge equations. The results of the edge equation tests can directly be transformed into barycentric coordinates, which in turn can easily be corrected for perspective interpolation

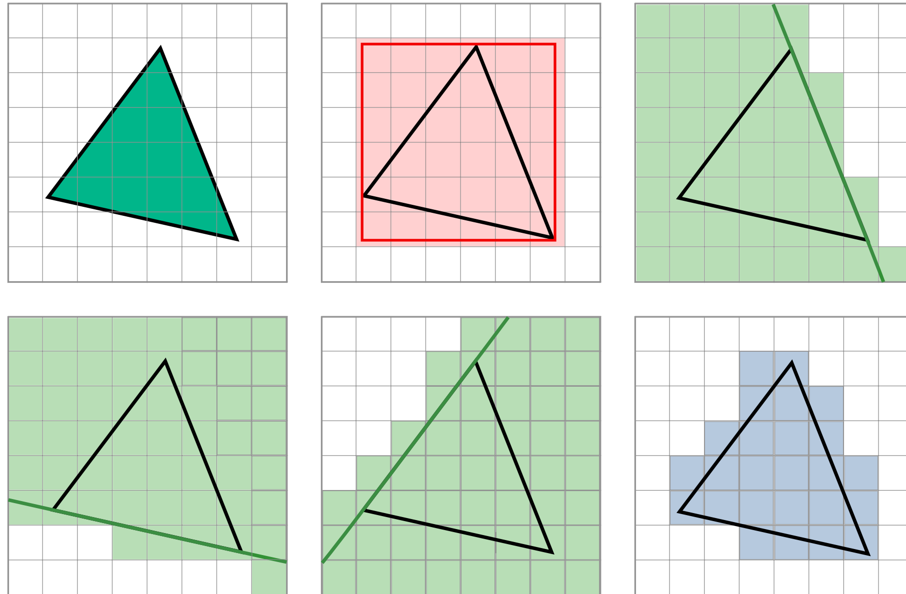


Figure 3.3: Visualization of scanline rasterization in the bin rasterizer. The relevant tiles are the ones that are covered by the bounding rectangle and all three edge half-planes.

and are used for fragment shading in our case.

For depth testing, we currently use global memory atomic operations on 64-bit words, where we use 32-bit for depth and 32-bit as color payload. Thus, we do not need to synchronize between fragments that may write to the same buffer location in this prototype renderer. To support more complex blending and output to multiple buffers, we could either ensure that only a single thread accesses a pixel, similar to Kenzel et al. [9], or even keep the buffers in shared memory and coordinate writes within the same warps [11].

4 Nonlinear rendering

Hardware acceleration plays an important role in the high performance of contemporary GPUs, but the upsides of higher throughput and lower energy consumption come at the cost of less flexibility. Programmable pipeline stages take some flexibility back, but can, of course, not cover every possible use-case. The constraint that is discussed in this work is that hardware-accelerated graphics pipelines can only efficiently depict linear mappings, since it contains non-programmable parts that only work in this setting. Nonlinear mappings can not be efficiently rendered.

Based on complexity of the implementation, we identify three tiers of nonlinearity. First, we want a mapping that distorts the output image in x and y direction independently. We call this rectilinear distortion, which is explained in more detail in section 4.1, and could mainly be used for various shadow mapping techniques. Second, a mapping that distorts the output image based on a function of both x and y, which could be used to create for example barrel distortions, which, in turn, can be used to render VR applications without having to apply distortions in a post-processing step. We call this bijective distortion, which is extensively explained in section 4.2. A possible third tier would be to have arbitrary sample locations in the output image. This approach could, however, lead to a non continuous, non invertible distortion function and would therefore not be usable by our software pipeline implementation. There are also no obvious use cases apparent to us, therefore, the third tier was not pursued. Since the first tier of nonlinearity is also implicitly included in the second tier, it is important to note that the distinction still makes sense from a performance point of view, as the less complex implementation for rectilinear distortion leads to better performance over bijective distortion. This correlation is later shown in section 5.4.

In this work, the term distortion is described as a mapping $x, y \mapsto f(x, y)$, whereas both x and y are screen-space coordinates in the range $[-1; 1]$. We call this forward distortion. In order for this mapping to be useable by the pipeline, it needs to be continuous and bijective. For bijectivitiy, we introduce the appropriate inverse mapping $x', y' \mapsto f^{-1}(x', y')$, which we call inverse distortion. It has the property of $f(f^{-1}(x, y)) = f^{-1}(f(x, y)) = x, y$,

meaning successively applying forward and inverse distortion yields the original coordinates.

4.1 Rectilinear distortion

In rectilinear distortion the sample locations of the fragments are distorted independently in x and y direction. This means that all sample locations who share either x or y coordinates before applying the distortion, do so also after the distortion was applied. Mathematically, this means the general distortion function is simplified to the mapping $x, y \mapsto f(x), g(y)$. Consequentially, the inverse distortion therefore is $x', y' \mapsto f^{-1}(x'), g^{-1}(y')$. An example of such a rectilinear mapping can be seen in Figure 4.1, which shows a cylindrical projection. The inverse distortion can in this case also be analytically calculated, leading to no rounding errors when doing a forward calculation followed by an inverse calculation.

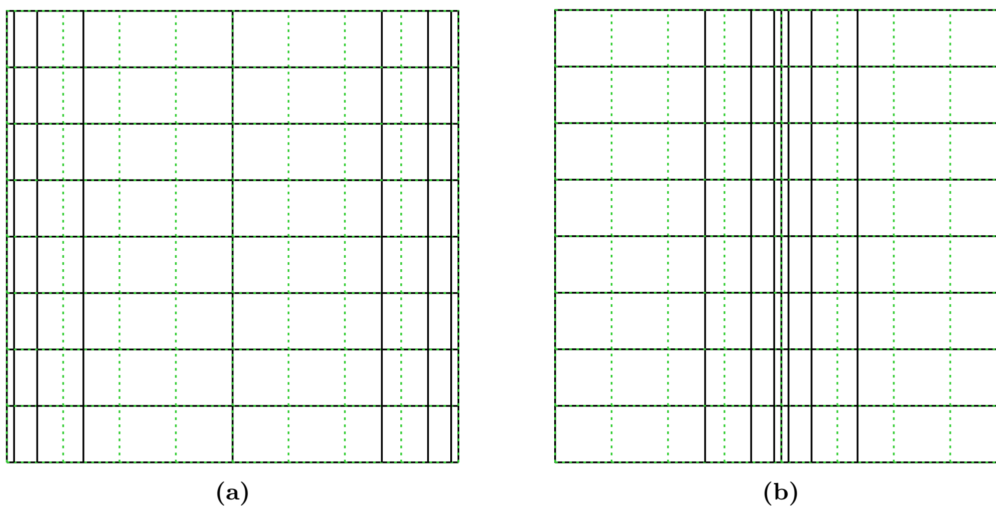


Figure 4.1: Visualization of forward distortion (4.1a) and inverse distortion (4.1b) as described in section 4.1 being applied to a regular grid, with the regular grids being shown as dotted, green lines and the distorted grids as solid, black lines. Since a rectilinear distortion is used, which applies to x and y direction independently, lines are still lines after distortion being applied.

In order to adapt the pipeline for this type of rendering, only two small changes are necessary. First, when finding out which bin a given fragment belongs to, the forward distortion has to be applied to its sample location, and second, when rasterizing the fragment, the inverse distortion has to be applied to its sample location. All the other parts of the pipeline stay the same, and can be inherited without changes. An image of a cube rendered with this distortion can be seen in Figure 4.2.

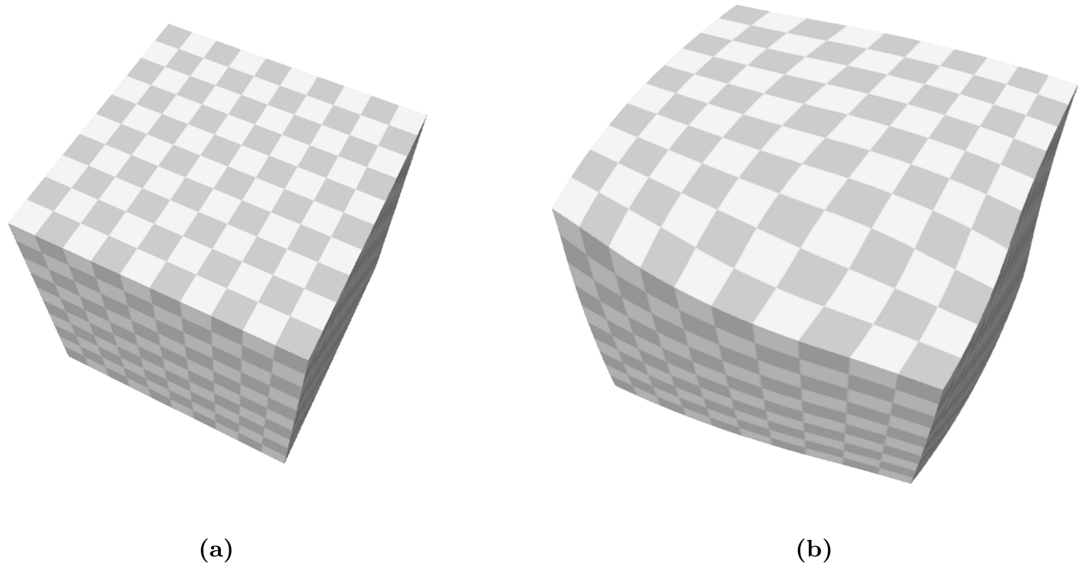


Figure 4.2: Visulization of a cube without distortion (4.2a) and rectilinear distortion being applied during rendering (4.2b). It is the same distortion that is used in Figure 4.1a. The parts in the center become horizontally stretched, while the parts at the left and right edge are compressed, leading to a s-shaped appearance of some of the edges.

4.2 Bijective distortion

Here, x and y are distorted by a two-dimensional function and cannot be distorted independently. Sample locations that share x or y coordinates before applying the distortion do not necessarily still form a line after the distortion is applied. A visualization of such a forward and inverse distortion being applied to a regular grid with the parameters that are later used in the implementation can be seen in Figure 4.3. An image of a cube rendered with the same distortion can be seen in Figure 4.4. Contemporary VR headsets use lenses to simulate large perceptible distances for eyes to focus on, but introduce distortion in the process. For most popular contemporary headsets, this is pincushion distortion, which needs to be countered by applying its inverse, a barrel distortion, which is a form of bijective distortions, somewhere in the graphics pipeline.

The forward distortion function that is used in our experimental implementation is a cubic polynomial, the reasons for that are explained in section 4.3. Since evaluating the exact inverse of a third-order polynomial is computationally intensive, an approximate inverse is calculated instead. For convenience and runtime performance reasons, this is done for a univariate polynomial, which maps normalized x, y coordinates to their corresponding inversely distorted points by using it in a radial distortion function. The coefficients of the polynomial for this mapping are estimated by minimizing the maximum error after a sequence of distortion and inverse distortion for each point on a regular point grid on the normalized x, y-plane. The order of the polynomial of this inverse distortion

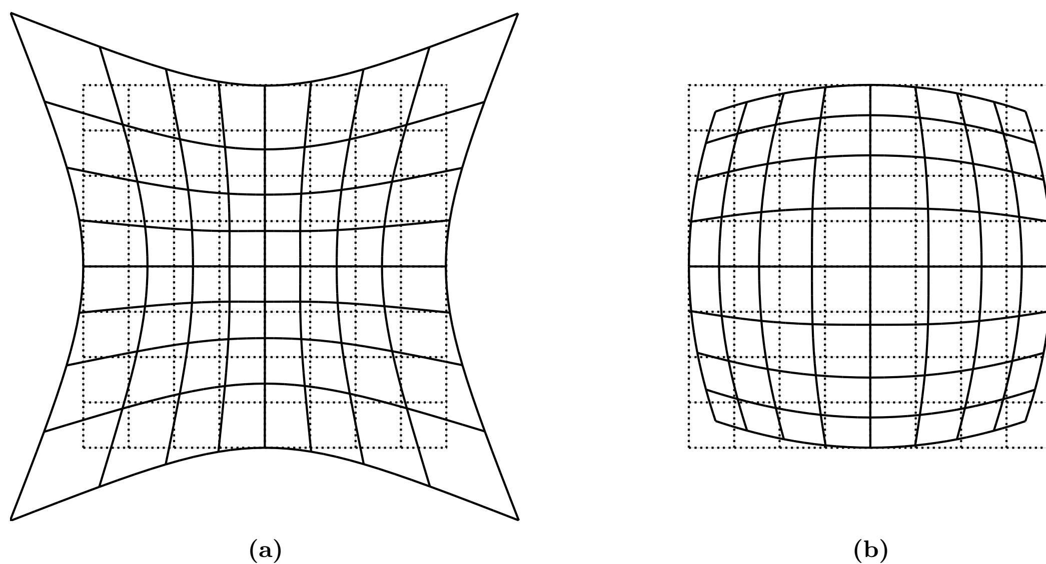


Figure 4.3: Visualization of forward radial distortion (4.3a) and inverse radial distortion (4.3b) as described in section 4.2 being applied to a regular grid, with the regular grids being shown as dotted lines and the distorted grids as solid lines.

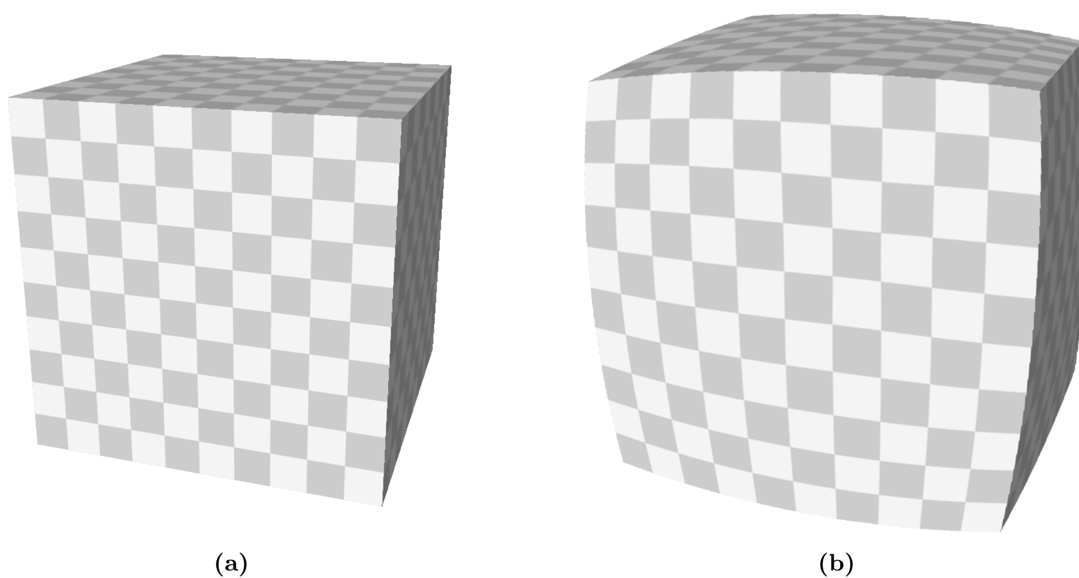


Figure 4.4: Visulization of a cube without distortion (4.4a) and barrel distortion, a form of radial distortion, being applied during rendering (4.4b). It is the same distortion that is used in Figure 4.3b. The curvature of the edges, which bend away from the center, is clearly visible.

can be arbitrary chosen, with higher-order polynomials leading to smaller errors but being computationally more expensive. When, for example, choosing an ninth-order polynomial, we empirically determined the residual error on a Full-HD screen to be less than one pixel, which is good enough for our use case. Nonetheless, since this polynomial is of finite order, the minimized maximum error does not reach 0, which is accounted for by adding safety margins proportional to the maximum error around the distortion-aware triangle boundaries. A visualization of the residual error of the chosen inverse distortion can be seen in Figure 4.5.

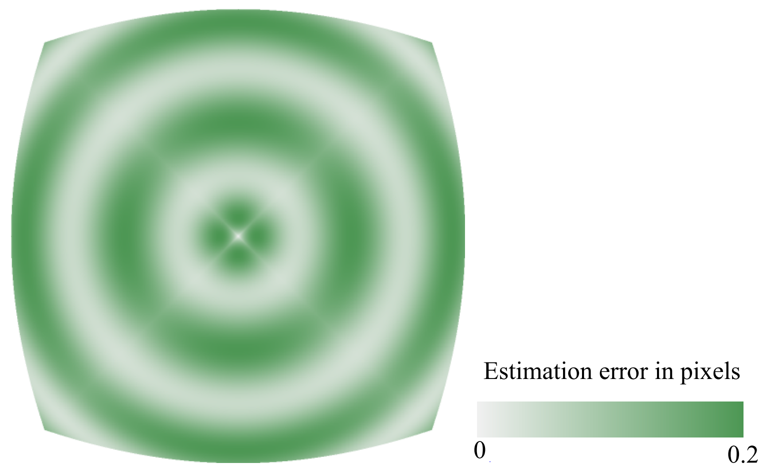


Figure 4.5: Visualization of pixel displacement error compared to original positions after doing a forward distortion followed by an inverse distortion. Gray colors indicates a small error, green indicates a large error. The maximum error in this picture in normalized coordinates is about 0.0004, which corresponds to about 0.2 pixels at a size of 512x512 pixels.

4.3 Forward and inverse distortions

The implementation of bijective distortion in our experimental pipeline is loosely based on the moderately popular OpenHMD library [16], which uses an universal distortion shader to unify rendering for many different virtual reality headsets under a common API. The library also contains distortion parameters for many common HMDs, which were calculated by taking images of the HMD from the eye perspective with a camera and using computer vision algorithms to deduce the approximate parameters. These parameters are to be interpreted as coefficients of a third-order polynomial, which is used to distort normalized coordinates inversely to the respective lens distortion, such that they cancel each other out. Some of these are used for testing in this work.

In addition to these parameters, the inverse of this distortion is required to do correct bin and tile assignments when calculating the distortion-aware boundary of each triangle. Both forward and inverse distortion are implemented as radial distortions. This means

that screen-space positions are displaced radially based on the formula

$$l = \left\| \begin{pmatrix} x \\ y \end{pmatrix} \right\|$$
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} (c_n \cdot l^n + c_{n-1} \cdot l^{n-1} + \ldots + c_1 \cdot l + c_0), \tag{4.1}$$

whereas $c_0 \ldots c_n$ are the polynomial coefficients, x and y are the undistorted coordinates and x' and y' are the distorted coordinates. The forward and inverse polynomial which is used in the implementation can be seen in Figure 4.6.

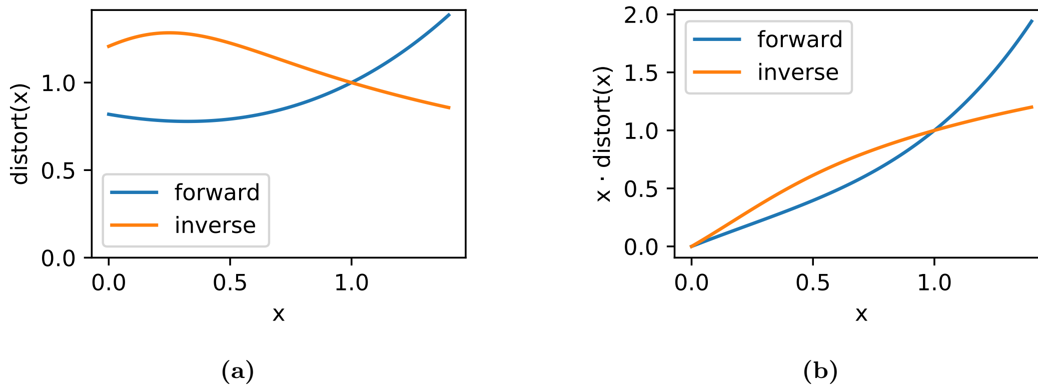


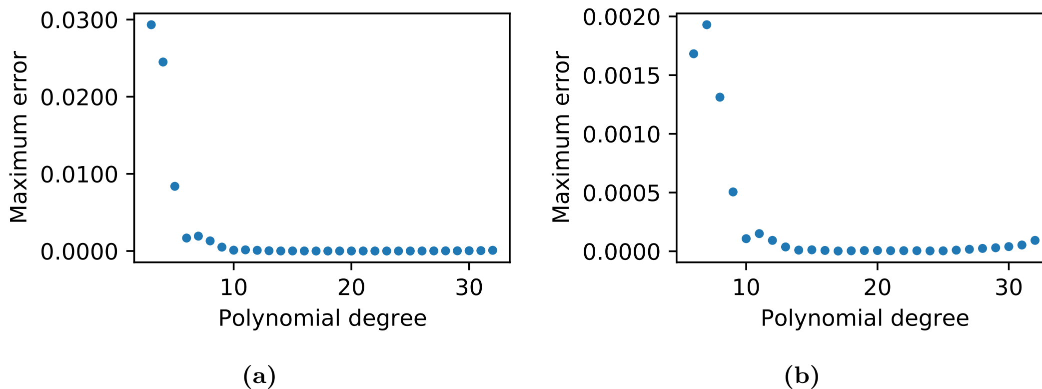
Figure 4.6: Visualization of forward and inverse distortion polynomials (4.6a) and the one-dimensional distortion functions applied to the variable range $0 \ldots \sqrt{2}$ (4.6b). The polynomials themselves are not monotone, but the resulting distorted values are monotone and need to be monotone in order for the inverse mapping to work.

The effect of different polynomial orders for the inverse polynomial on the maximum error can be seen in Figure 4.7. Choosing the correct polynomial degree is a tradeoff between computation cost of evaluating the polynomial and computation cost of having to deal with additional fragments due to the pixel displacement error as described in Figure 4.5.

4.4 Nonlinear pipeline modifications

At the end of geometry processing, out of screen culling, near plane clipping, guard-band clipping, fixed-point conversion and backface culling are happening as usual. Small triangle culling, bin assignment, triangle collection and tile rasterization need to be adapted for nonlinear rendering.

Small triangle culling Small triangle culling, also called between sample culling, is important for scenes with very fine geometry and can significantly increase performance



(a) (b)

Figure 4.7: Visualization of the maximum error when doing a forward distortion followed by a inverse distortion while varying the degree of the inverse polynomial of the one-dimensional distortion functions in the variable range $0 \ldots \sqrt{2}$. Figure 4.7a shows the range 3 to 32, while Figure 4.7b shows the range 6 to 32 to enhance the detail in the higher orders. Starting at about degree 26 overfitting can be observed.

as triangles that will not produce any samples can be removed early in the pipeline. With non-linear rasterization, we face the problem that the sample locations used in the end are transformed individually and, thus, simple culling, as is traditionally used, is not possible. Typically, one would simply compute the bounding box of a triangle and, if the bounding box falls between two adjacent sample rows or columns, directly cull the triangle.

For non-linear rasterization we need to consider the following: First, we could take the bounding box, use the inverse distortion to compute the distorted bounding box and see whether the bounding box falls between non-distorted sample rows or columns. However, as the inverse distortion is not an analytic inverse of the forward distortion this would result in errors. Also note that the inverse distortion applied to the bounding box also needs to consider the case where transforming the corners is not sufficient to get the right result. Second, we could compute the standard bounding box and search for the pixels that after forward distortion intersect with the bounding box. However, finding those pixels without additional information is non-trivial in the general case.

However, given that we have the inverse distortion function—albeit it is not being exact—we can use it to guide the search for pixels that are closest to the triangle. Thus, our algorithm for rejecting triangles that fall between sample locations looks as follows:

```
FixedPointBBox bbox(v1_e, v2_e, v3_e);

FixedPointVector blower = undistort(bbox.min);
FixedPointVector bupper = undistort(bbox.max);
int2 lower_pixel = {blower.x - halfPixel - 1,
                    blower.y - halfPixel - 1};
int2 upper_pixel = {bupper.x + halfPixel + 1,
                    bupper.y + halfPixel + 1};
```

```
if (upper_pixel.x - lower_pixel.x > 1 &&
  upper_pixel.y - lower_pixel.y > 1)
{
  culled = false;
}
else
{
  int2 loxloy = distort({lower_pixel.x, lower_pixel.y});
  int2 loxhiy = distort({lower_pixel.x, upper_pixel.y});
  int2 hixloy = distort({upper_pixel.x, lower_pixel.y});
  int2 hixhiy = distort({upper_pixel.x, upper_pixel.y});

  if (upper_pixel.x - lower_pixel.x == 1 &&
      ((lower_pixel.y < 0) ^ (upper_pixel.y < 0)) == 0 &&
      (loxloy.x < bbox.min.x && hixloy.x > bbox.max.x &&
        loxhiy.x < bbox.min.x && hixhiy.x > bbox.max.x))
  {
    culled = true;
  }
  else if (upper_pixel.y - lower_pixel.y == 1 &&
    ((lower_pixel.x < 0) ^ (upper_pixel.x < 0)) == 0 &&
    (loxloy.y < bbox.min.y && loxhiy.y > bbox.max.y &&
      hixloy.y < bbox.min.y && hixhiy.y > bbox.max.y))
  {
    culled = true;
  }
}
```

First we compute the bounding box from the fixed-point vertex locations in undistorted space. We then perform inverse distortion to get an estimate of the distorted bounding box. Next, we round the surrounding undistored pixel locations, giving us the closest pixel indices outside the bounding box. If those are further than one pixel apart in both x and y direction, the triangle reaches out both a pixel row and a pixel column and the triangle is not culled. This path is highly efficient and equal to the previously described option one.

However, if a triangle could still be culled, we now perform bit stable test. We apply the inverse distortion the so found four pixel locations. Finally we test whether the original triangle bounding box falls between either the distorted pixel row or column. Depending on the distortion function slightly more or less tests are necessary. For rectilinear distortions, we only have two x and two y coordinates, min and max, to consider and we do not need to have any special case handling. For bijective distortions, we need to consider all four corners as those may transform differently. Furthermore, if the bounding box crosses the respective coordinate axis, the transformation is not monotonic and the "bending" may

allow crossing multiple pixel rows/columns while the extrema do not capture this fact. Thus, we exclude those cases from culling and accept a few additional triangles in favor of performing more complicated tests. Note that in this way, the undistorted bounding box computed from the already quantised fixed-point vertices is tested against that fixed-point distorted, i.e., moved pixel locations which are used during tile rasterization. Thus the results are fully stable.

Bin assignment For bin assignment, we compute the conservative inverse distorted bounding box and add the triangle to all bins that are touched by the bounding box. Depending on the distortion function, the bounding box computation is slightly different: For rectilinear distortion, we just perform the inverse transform on the bounding box x and y coordinates. For bijective distortion, we perform the inverse transform on all three vertices, which is more efficient than transforming the 4 bounding box corners. Then we compute min and max over the transformed vertices. If the triangle is crossing an axis, we use the original bounding box intersection with the axis and again inverse transform on the intersection and update min and max with the new values.

Finally we add an additional margin on top of the bounding box to ensure we accommodate for inaccuracies between the forward and inverse transform. This epsilon is determined before setting up the transform, computing the maximum difference in between the distortion functions, as described in Figure 4.7. Note that we could even use a higher order error bound, however a constant is a better tradeoff in terms of computation cost and accuracy.

Triangle collection The triangle collection steps—the bin-to-tile and tile rasterizer—also need to be modified to account for distortion. The steps described in section 3.2 need to be adapted because an unmodified scanline rasterization would not work unless an intersection between the triangles edges and a distorted bin grid could be calculated with little effort. Since this does not seem to be possible, and calculating the tile bitmask by checking every tile separately would also not be very efficient, we use a binary search over the tile row instead, which halves the performance impact over checking each tile individually on average. We did not find a more efficient way of finding the intersections. A graphical representation of this step can be seen in Figure 4.8.

Tile rasterizer The tile rasterizer also needs to be altered, with a forward distortion being applied to the fragment positions before they are checked against the triangles edge equations. Since in fragment processing one warp is responsible for one tile, and therefore one thread for only two fragments, we did not use binary search in this stage, as it would not improve performance.

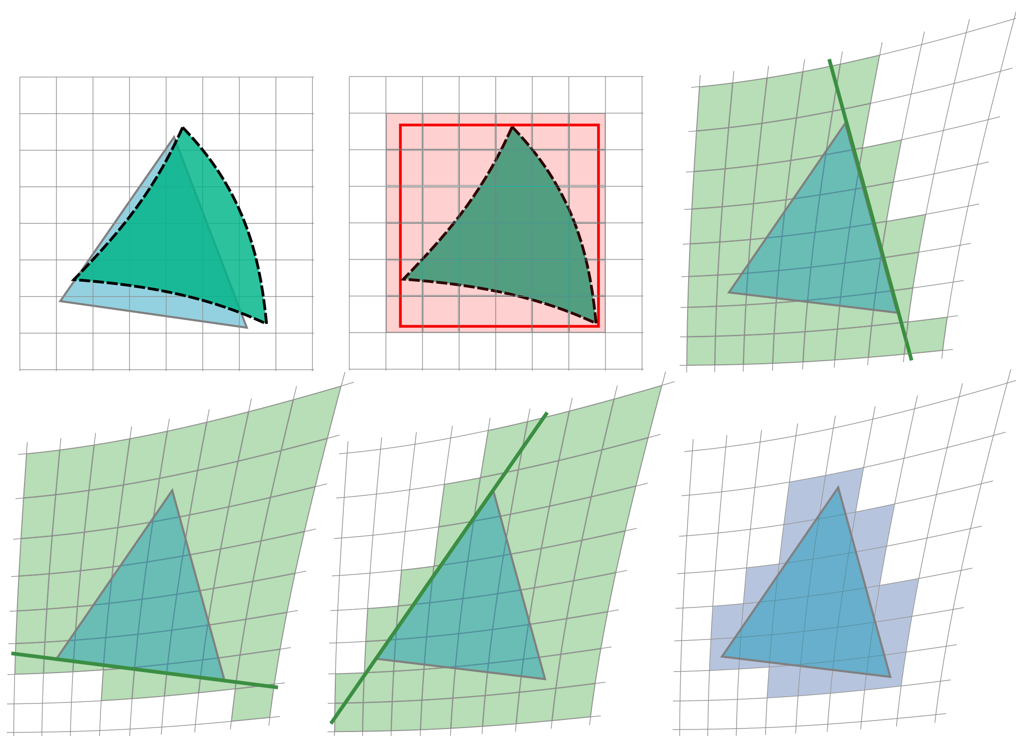


Figure 4.8: Visualization of scanline rasterization in the bin rasterizer for a radial distortion. Each bin covered by the bounding rectangle is subdivided into tiles. After the forward distortion is applied to the grid, the three edge equations are used to find tiles that are completely outside of the triangle, which are then discarded together with tiles outside of the bounding box. The remaining tiles are processed further.

5 Performance evaluation

For evaluating the performance we ran various test scenes, both with single-mesh objects and scenes captured from video games. Since we used theses captured scenes we were able to get a good understanding of how our pipeline works with real-world data. For comparison of the performance of rectilinear and bijective distortions we compared our implementation to a standard OpenGL pipeline with distortions being applied in a post-processing stage, the lens distortion part of the work by Friston et al. [6], and a raytracing implementation based on Nvidia Falcor [2]. For testing the pipeline without distortion we only compared it to a standard OpenGL pipeline and raytracing.

5.1 Test setup

We ran our test scenes with an Intel Core i5-4690 CPU and 16 GiB or RAM, on a PC running Windows 10. All tests were performed with a Nvidia Geforce GTX 1080 Ti and RTX 2080 Ti. The in comparison relatively old CPU does not influence the results in a negative way since we only tested static scenes, which makes running the pipelines require very little CPU cycles. As resolutions, we chose 1920x1080 and 3840x2160 pixels for all test cases. The findings presented here represent a summary of all measurements, the test results as a whole are not included.

5.2 Real-world scenes

At 1080p resolution, and with the 1080 Ti, comparing our implementation with real-world scenes yields that an OpenGL implementation with distortion being applied in postprocessing is currently by far the fastest way of achieving distorted rendering. Our pipeline performs on average slightly better than Perceptual Rasterization for rectilinear and radial distortion, and slightly worse for foveation distortion and no distortion, which is illustrated in Figure 5.1.

At 2160p (4K) resolution, our implementation is almost universally significantly better-performing than Perceptual Rasterization, with the only exception being the very simple scenes of Age of Mythology. Also the difference to OpenGL shrank, which we attribute to the large constant overhead of doing a screen-space rendering pass at a higher resolution, as shown in Figure 5.2

With the 2080 Ti on the other hand our implementation is better performing in every game except, again, Age of Mythology. While the frametimes are slightly better on the lower resolution, at the higher resolution the difference is significant. Also the difference between our implementation and OpenGL is smallest at the higher resolution, with the difference being less than one millisecond most of the time. These figures can be seen in Figure 5.3 and Figure 5.4.

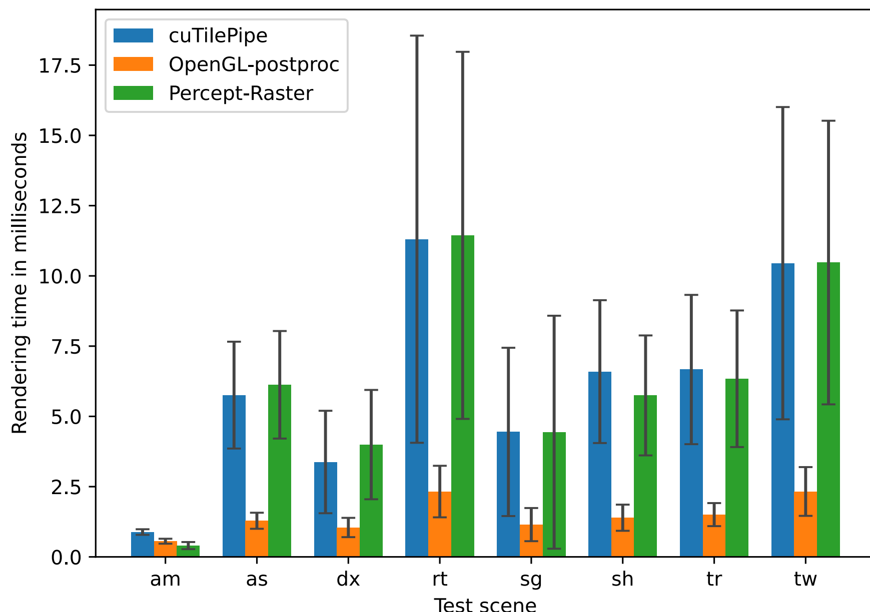


Figure 5.1: Comparison of radial distortion rendering times between implementations for real-world scenes (1920×1080, 1080Ti).

5.3 Artificial scenes

Drawing conclusions from the artificial test scenes is more difficult, since we only tested ten of them. Stating with the 1080 Ti, we observed that the different resolutions did not make that large of a difference as with the real-world scenes. What is still universally true is that OpenGL is the fastest, no matter the resolution or distortion type. Our implementation, on the other hand, seems to be faster than Perceptual Rasterization at 1080p more often than not, which is depicted in Figure 5.5. Surprisingly, this is not also

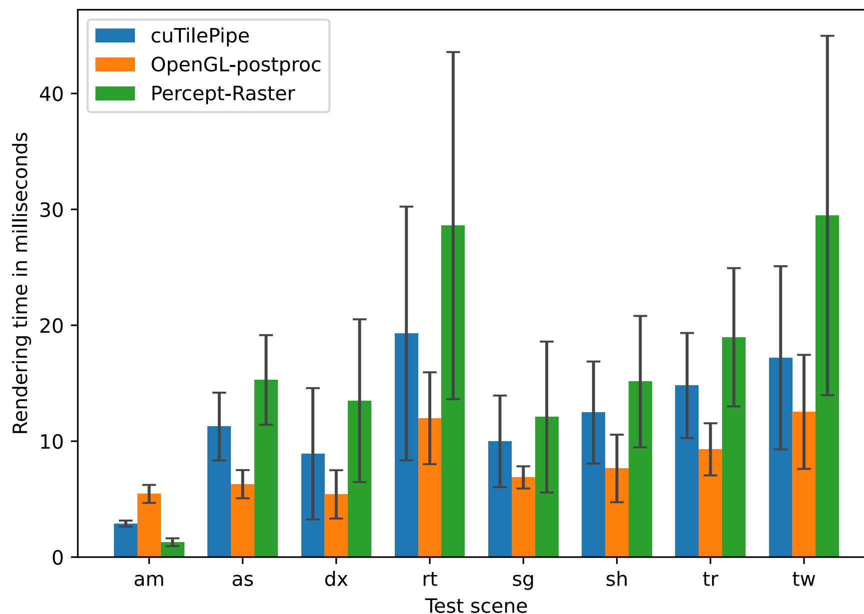


Figure 5.2: Comparison of radial distortion rendering times between implementations for real-world scenes (3840×2160, 1080Ti).

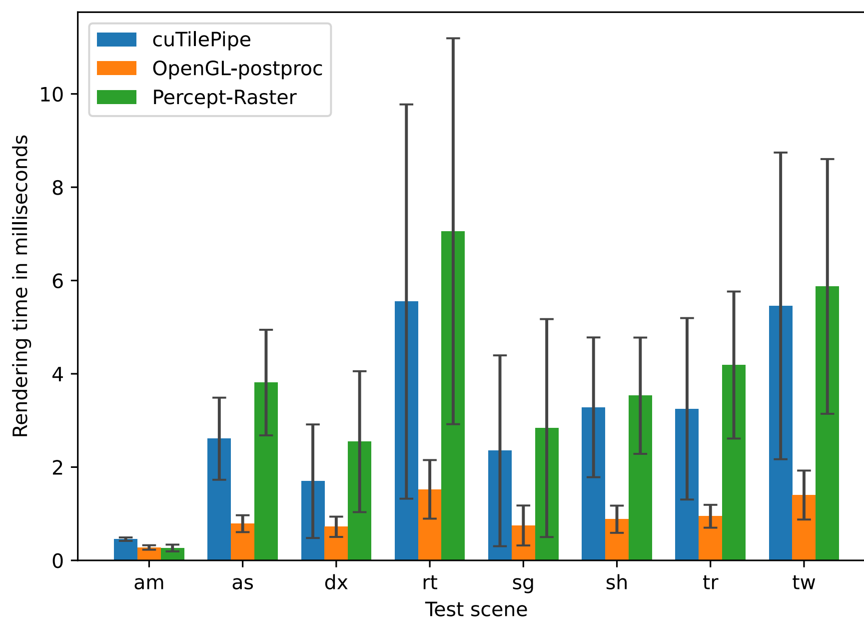


Figure 5.3: Comparison of radial distortion rendering times between implementations for real-world scenes (1920×1080, 2080Ti).

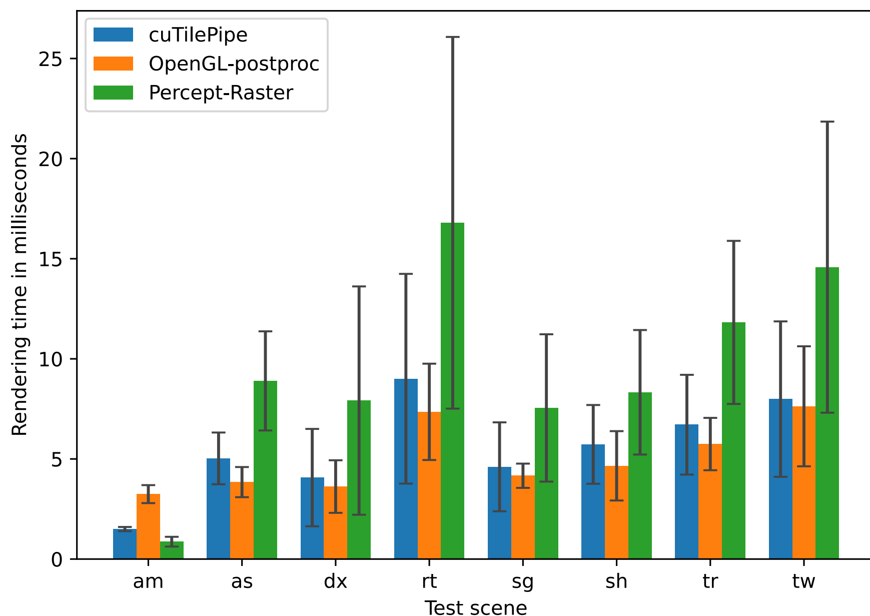


Figure 5.4: Comparison of radial distortion rendering times between implementations for real-world scenes (3840×2160, 2080Ti).

the case at 4k, where this circumstance is reversed. Ray tracing, no matter the resolution, is mostly slower than our pipeline, mostly because of the missing hardware support on this device. This can be seen in Figure 5.6.

Changing to the 2080 Ti, which has hardware support for ray tracing, draws a completely different picture: Ray tracing is now universally faster than even OpenGL, even more so at higher resolutions. When accounting for acceleration structure rebuild times, which would play a role if an animated scene was rendered, it is no longer faster than OpenGL, but still faster than the other two implementations. Our implementation tends to be faster than Perceptual Rasterization at 1080p, while they are more or less evenly matched at 4k resolution. Figure 5.7 and Figure 5.8 show this situation.

5.4 Distortion tiers

By comparing the rendering times of our implementation using no distortion, rectilinear distortion and radial distortion, we get a good understanding of how our nonlinear pipeline modifications affect performance. A summary of this data for the real-world scenes for the configuration 1080p/2080Ti is depicted in Figure 5.9, the other configurations look very similar. The linear rasterizer is the fastest in all scenes, while the bijective rasterizer is the slowest, with the rectilinear rasterizer being in the middle. This confirms our assumption from chapter 4 that making rectilinear distortion a separate tier make sense from a performance point of view.

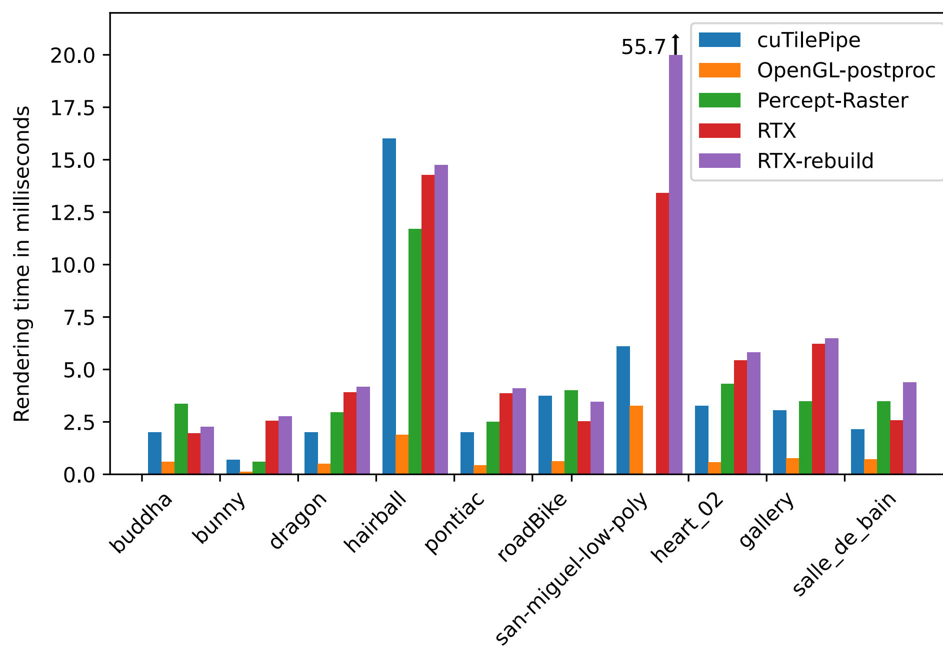


Figure 5.5: Comparison of radial distortion rendering times between implementations for artificial test scenes (1920×1080, 1080Ti).

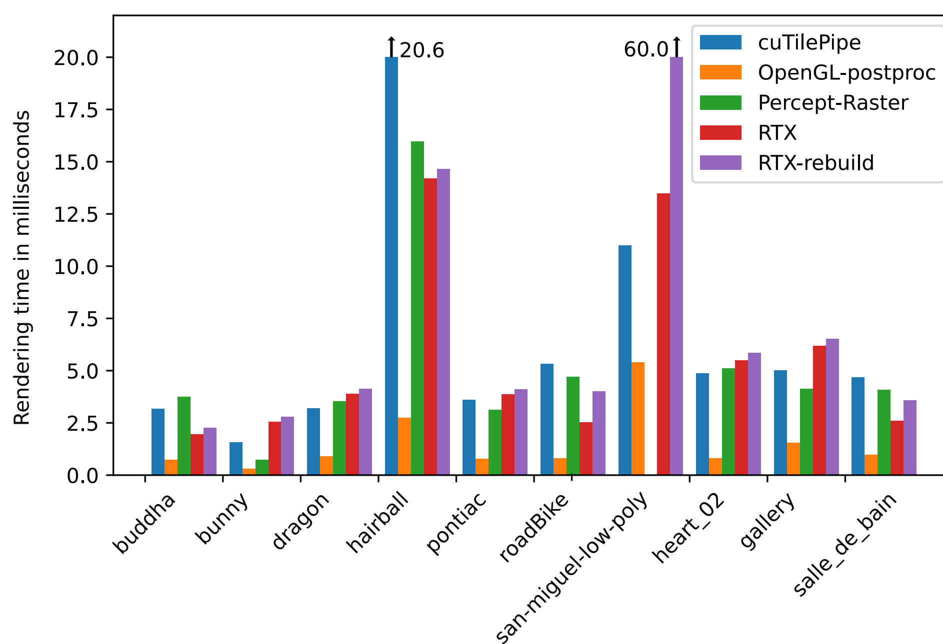


Figure 5.6: Comparison of radial distortion rendering times between implementations for artificial test scenes (3840×2160, 1080Ti).

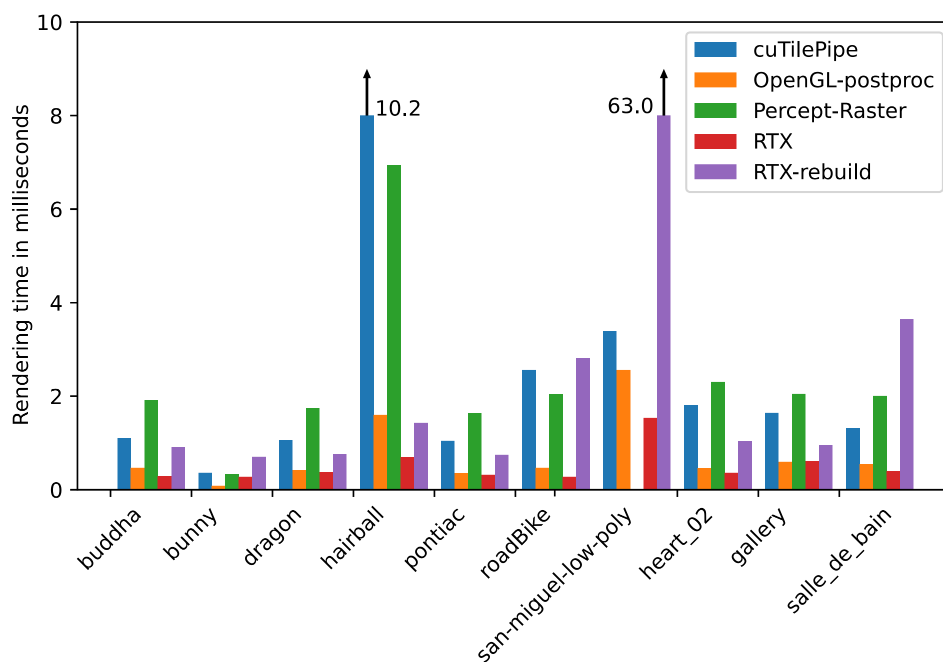


Figure 5.7: Comparison of radial distortion rendering times between implementations for artificial test scenes (1920×1080, 2080Ti).

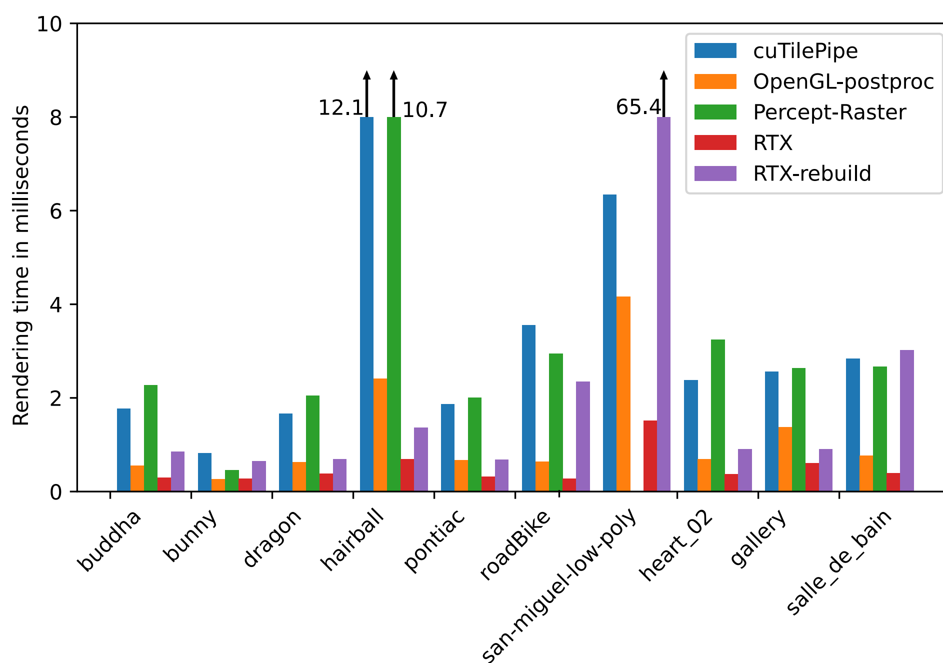


Figure 5.8: Comparison of radial distortion rendering times between implementations for artificial test scenes (3840×2160, 2080Ti).

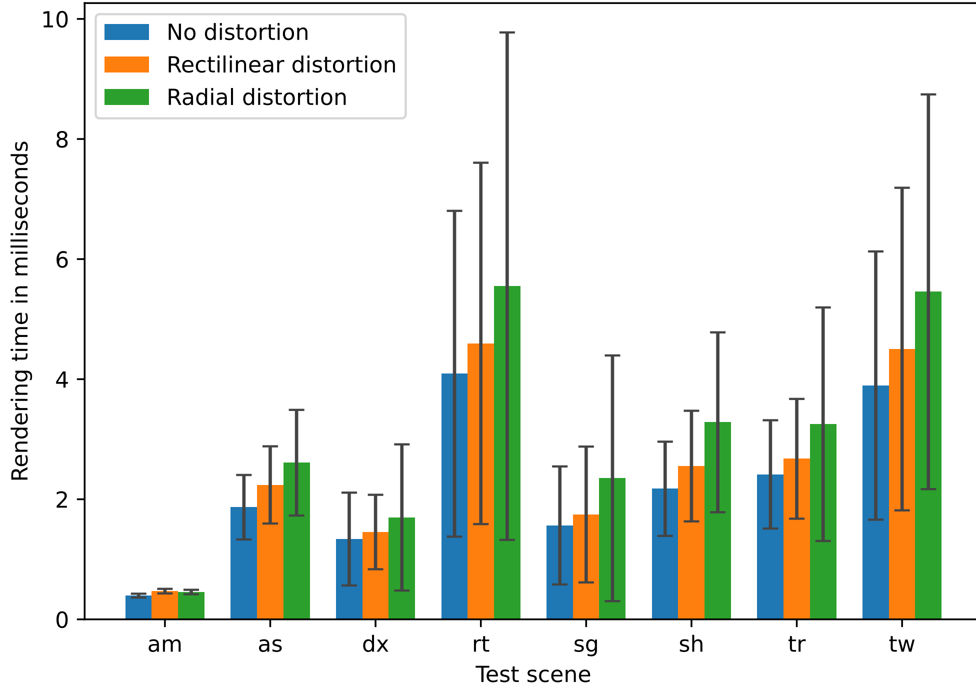


Figure 5.9: Comparison of rendering times of the real-world scenes in our implementation with varying distortion type. It is apparent that rendering without distortion is fastest while radial distortion is slowest (1920×1080, 2080Ti).

5.5 Remarks

While comparing raw performance numbers does make sense if the visual output is the same, it is not so easy if this is not the case. The OpenGL implementation, for example, leads to slightly worse image quality around the center of the image because of resampling at a varying pixel density. Simultaneously, the quality at the edges of the image is higher than with the other implementations because of the same reason. If we were to increase the intermediate image size of the OpenGL implementation, meaning the size of the image of the first rendering pass, until there is no difference in image quality at the center, the results may look different, but we did not check if that would make a significant difference.

6 Conclusion

We demonstrated how to implement a software rasterization pipeline and adapt it for nonlinear rendering. Furthermore, we showed that it can achieve competitive performance when rendering nonlinear distortions. While our implementation is almost always performing worse than a naïve OpenGL implementation with postprocessing, the exception being high resolutions with very simple scenes, the tradeoff between quality and performance of these two implementations is at least debatable. The performance of Perceptual Rasterization, which yields the exact same quality as our implementation, is more often than not worse than ours, with this circumstance being more pronounced at higher resolutions. This applies to both artificial test scenes and real-world scenes. The performance of ray tracing is, as to be expected, largely dependent on hardware support. On the 2080Ti, which has hardware support for ray tracing, our implementation came nowhere near close to that of ray tracing, at least on the tested static scenes. It also has to be noted that there was no significant fragment shader load in our experimental implementations, with all of them being implemented as deferred shading and only counting the geometry pass towards the rendering time. We did not investigate how a more bandwidth- and compute-heavy fragment shader would influence the results, but given that we use deferred shading anyway, using a different fragment shader would result in similar absolute differences in overall rendering time.

Since our pipeline is implemented entirely in software, significant performance gains are to be expected if the nonlinear rasterizers were implemented in hardware. As rasterization will most likely still be relevant in the near future, especially on mobile devices where ray tracing might not be a good fit because of the inherent non-uniform memory access and memory bandwidth playing a more important role than on desktop systems, extending the graphics hardware would make sense. On desktop systems, on the other hand, the meaningfulness of such a hardware change is not as clear, as ray tracing performance is constantly improving, with the current generation of GPUs doubling that of the 2080Ti from two years ago.

Bibliography

[1] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45, 1968. (page 5)

[2] N. Benty, K.-H. Yao, P. Clarberg, L. Chen, S. Kallweit, T. Foley, M. Oakes, C. Lavelle, and C. Wyman. The Falcor rendering framework, 03 2020, `https://github.com/NVIDIAGameWorks/Falcor`. (page 25)

[3] S. Brabec, T. Annen, and H.-P. Seidel. Shadow mapping for hemispherical and omnidirectional light sources. *Advances in Modelling, Animation and Rendering (Proceedings Computer Graphics International 2002), Springer, 397-408 (2002)*, 06 2002. (page 7)

[4] T. Capin, K. Pulli, and T. Akenine-Möller. The state of the art in mobile graphics research. *IEEE Computer Graphics and Applications*, 28(4):74–84, 2008. (page 11)

[5] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012. (page 2)

[6] S. Friston, T. Ritschel, and A. Steed. Perceptual rasterization for head-mounted display image synthesis. *ACM Trans. Graph.*, 38(4):97:1–97:14, July 2019, `http://doi.acm.org/10.1145/3306346.3323033`. (page 7, 9, 25)

[7] E. Haines and T. Akenine-Moller. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Apress, USA, 2019. (page 8)

[8] W. Hunt, M. Mara, and A. Nankervis. Hierarchical visibility for virtual reality. *Proc. ACM Comput. Graph. Interact. Tech.*, 1(1), July 2018, `https://doi.org/10.1145/3203191`. (page 6)

[9] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger. A high-performance software graphics pipeline architecture for the gpu. *ACM Trans. Graph.*, 37(4), November 2018. (page 1, 11, 12, 13)

[10] B. Kerbl, M. Kenzel, D. Schmalstieg, and M. Steinberger. Effective static bin patterns for sort-middle rendering. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery, `https://doi.org/10.1145/3105762.3105777`. (page 11)

[11] S. Laine and T. Karras. High-performance software rasterization on gpus. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 79–88, New York, NY, USA, 2011. ACM, `http://doi.acm.org/10.1145/2018323.2018337`. (page 1, 7, 11, 12, 13)

[12] F. Liu, M.-C. Huang, X.-H. Liu, and E.-H. Wu. Freepipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 75–82, New York, NY, USA, 2010. ACM, `http://doi.acm.org/10.1145/1730804.1730817`. (page 1, 7)

[13] D. B. Lloyd, N. K. Govindaraju, S. E. Molnar, and D. Manocha. Practical logarithmic rasterization for low-error shadow maps. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '07, page 17–24, Goslar, DEU, 2007. Eurographics Association. (page 8)

[14] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE computer graphics and applications*, 14(4):23–32, 1994. (page 10)

[15] Nvidia. Cuda c++ programming guide, version 11.1. *NVIDIA Corp*, 2020. (page 2)

[16] OpenHMD. Openhmd - foss hmd drivers for the people, 2019, `http://www.openhmd.net/`. (page 19)

[17] B. Osman, M. Bukowski, and C. McEvoy. Practical implementation of dual paraboloid shadow maps. In *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, pages 103–106, New York, NY, USA, 2006. ACM, `http://doi.acm.org/10.1145/1183316.1183331`. (page 7)

[18] A. Patney, M. Salvi, J. Kim, A. Kaplanyan, C. Wyman, N. Benty, D. Luebke, and A. Lefohn. Towards foveated rendering for gaze-tracked virtual reality. *ACM Transactions on Graphics (TOG)*, 35(6):179, 2016. (page 6)

[19] A. Patney, S. Tzeng, K. A. Seitz, Jr., and J. D. Owens. Piko: A framework for authoring programmable graphics pipelines. *ACM Trans. Graph.*, 34(4):147:1–147:13, July 2015, `http://doi.acm.org/10.1145/2766973`. (page 7, 11, 12)

[20] M. Potmesil and I. Chakravarty. Synthetic image generation with a lens and aperture camera model. *ACM Trans. Graph.*, 1(2):85–108, April 1982, `https://doi.org/10.1145/357299.357300`. (page 8)

[21] P. Rosen. Rectilinear texture warping for fast adaptive shadow mapping. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '12, pages 151–158, New York, NY, USA, 2012. ACM, `http://doi.acm.org/10.1145/2159616.2159641`. (page 7)

[22] M. Segal and K. Akeley. The opengl graphics system: A specification, 2019, `https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf`. (page 3)

[23] P. Shirley and S. Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2009. (page 5)

[24] B. A. Watson and L. 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, 1995. (page 6)

[25] T. Whitted. An improved illumination model for shaded display. In *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, page 14, 1979. (page 5)