Gabriel Stoll BSc

Hardware Implementation of a non-linear Rasterization Unit

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl.-Ing. Dr.techn. Markus Steinberger BSc

Co-Supervisor

Dipl.-Ing. Alexander Weinrauch BSc

Institute of Computer Graphics and Vision
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Dieter Schmalstieg

Graz, June 2021

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.

Date	Signature

FOV field of view
VR Virtual Reality
GPU Graphics Processing Unit
FPGA Field Programmable Gate Array
ICG Institute of Computer Graphics and Vision
LUT Look up Table
DSP Digital Signal Processing Unit
RAM Random Access Memory
BRAM Block RAM
CLB Controllable Logic Block
VHDL Very High Speed Integrated Circuit Hardware Description Language
GDDR Graphics Double Data Rate RAM
IDE Integrated Development Environment
MSB Most Significant Bit
GDDR6 Graphics Double Data Rate RAM 6
HBM High Bandwidth Memory
fps Frames per Second
GPC Graphics Processing Cluster
SM Streaming Multiprozessor
GE Gate Equivalents
CMOS Complementary Metal Oxide Semiconductor

Abstract

In modern graphics pipelines many steps, so-called shaders, are programmable and allow us to create a variety of things such as light effects, shadows, object manipulation, etc. These customizable parts run on stream-multiprocessors, which are available in hundreds on the latest Graphics Processing Units (GPUs). Other parts are optimized for the pixel layout of connected displays and are directly implemented in hardware to be even more efficient. To support distorted images, which are widely used in Virtual Reality (VR) headsets, an additional rendering step is needed since the rigid hardware rasterizer can not be adapted to that. In this thesis, we present an altered hardware design for a rasterization unit to support traditional as well as distorted rendering. This allows us to embed the otherwise costly distortion algorithm into efficient dedicated hardware.

Acknowledgments

I would first like to thank my supervisor Markus Steinberger for introducing and guiding me throughout my thesis. Together with my co-supervisor Alexander Weinrauch, we had many good discussions which kept me motivated and lead to the successful completion of my thesis.
I would also like to acknowledge my colleagues which supported me whenever needed and allowed me to solve many of the upcoming problems. Their feedback helped a lot in improving the clarity and readability of this thesis.
Finally, I would like to thank my whole family, especially my parents, for the opportunity to complete my studies and for supporting me in all the years.

Contents

List of Figures

List of Tables

1 Introduction

Rasterization is the process, executed on GPUs where the geometry description of a scene gets converted in pixels on a raster image. Because of their properties, triangles are used to describe the objects of the scene. They can be stored individually, using three vertices, or as a triangle-strips where each new vertex together with the previous two forms a new triangle. Before a triangle can be rasterized, its vertices have to be projected to the virtual screen. This is done using a projection matrix in the so-called vertex shader. Depending on the application an orthogonal or a perspective projection can be used. The rasterization of the projected triangles is then done using

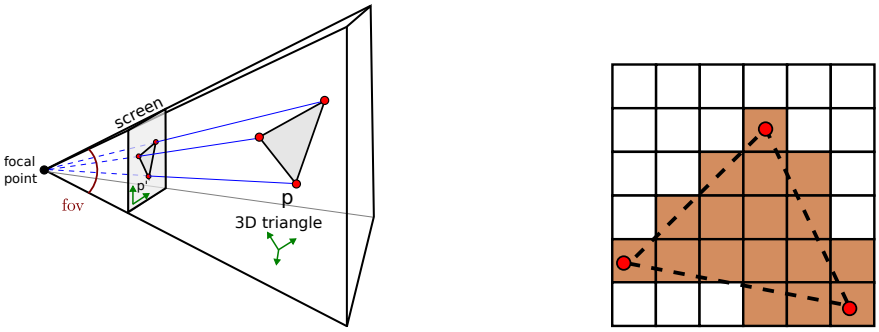


Figure 1.1: Illustration of a perspective triangle projection (left) to a virtual 2D screen which is used for rasterization (right). [1]

different algorithms. Some of the most popular ones are the scanline and centerline algorithms [2]. Both go through the pixels in a specific order and test if a pixel is inside or outside of the triangle. This evaluation can be done using the line equations of each edge of the triangle [2, p.20]. With increasing resolution, such algorithms were further improved and a tile-based approach was implemented [3] [4]. It first rasterizes the triangles to bigger tiles before going through every single pixel. Since this process is

the same for all applications, it is not running on the multiprocessors of modern GPUs, but on dedicated hardware. This allows to further optimize the algorithm and increase parallelism, with the main drawback that it can not be changed later on.

In VR applications the generated raster image has to be distorted at the end of the graphics pipeline to counteract the lenses in the used headsets. This distortion represents an additional step in the pipeline, decreases performance and increases latency. Since the rasterizer is a fixed hardware part, a non-linear software pipeline was implemented at the Institute of Computer Graphics and Vision (ICG) [5]. It skips the hardware rasterizer and does that step on the shader cores of the GPU. This allowed us, to make modifications to the rasterization algorithm and to introduce a distortion to the pixel location. In other words, the distortion which previously was introduced in an extra pipeline step can already be satisfied in the rasterization procedure.

After successfully testing the software pipeline a hardware design, using the same altered rasterization algorithm, was implemented. It was designed in such a way that it can be configured to enable and disable the distortion capability and switch between several distortion implementations. This allows us to compare the increased hardware requirements, performance, and memory throughput of different designs. Using these results together with the overall hierarchical structure, some considerations of using the implemented design in modern GPUs can be made, which probably would increase their performance significantly.

2 Lenses in VR headsets

The human eyes provide a field of view (FOV) of about $200 - 220°$, where each eye only sees $150°$ [7]. In a VR headset we try to cover most of this area using one screen per eye. Due to the limited size of the headset itself, these screens have to be placed as near as possible to the head. Therefore, using flat screens, it is only possible to cover a small fraction of the possible FOV of one eye and the experience would no longer be as immersive. One solution would be to use curved displays which surround each eye. But they are more expensive to produce and the number of pixels would become much bigger while keeping the same resolution in front of your eye. To still be able to use a cheap and relatively small displays, most VR headsets use lenses that break the light to increase the FOV drastically.

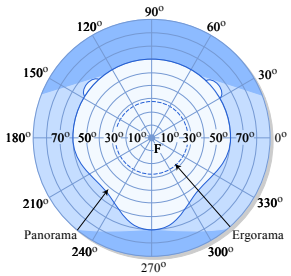


Figure 2.1: Human field of view. [6]

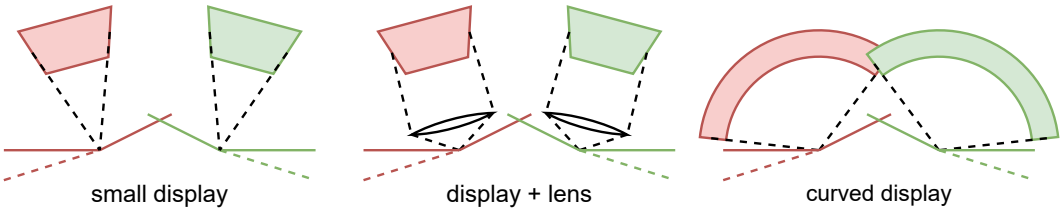


Figure 2.2: Comparison of achievable FOV using normal screens, screens with lenses, or curved displays.

The approach does also have an additional advantage; Since it uses a lens with a radial distortion function, the area in the middle of the screen stays mostly unaffected by the lens itself. The area on the outside instead has to be mapped to the border of the screen and therefore also the resolution of

the image will get worse. But since the focus of the displayed scene most often is in the middle of the screen this is not that bad and we can get an immersive experience without having to render the scene at a very high resolution.

Nevertheless, to counteract the lens distortion, the image displayed on the screen has to be distorted too. Such an inverse distortion, also known as barrel distortion, can be seen in figure 2.3.

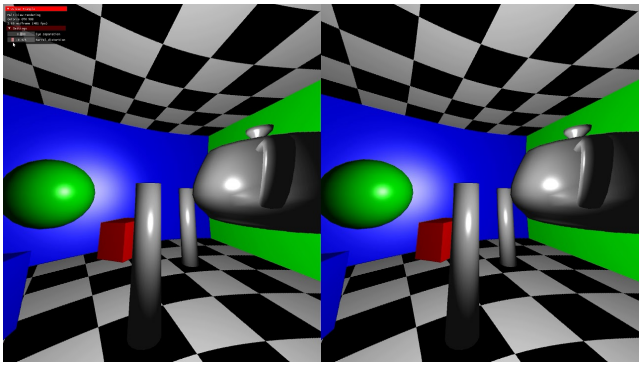


Figure 2.3: Example of a barrel distorted images suited for the left and right eye on a VR headset. [8]

2.1 Lens distortion models

Typically we distinguish three types of lens distortions:

- Barrel distortion
- Pincushion distortion
- Mustache distortion

The one produced by a typical lens of a VR headset is the pincushion distortion. The inverse distortion that has to be applied to the image is the barrel distortion. Several methods try to model this distortion and split it up into a distortion depending on the distance to a center point (sec. 2.1.1) and a distortion depending on the angle between the lens and the image plane (sec. 2.1.2). [10]

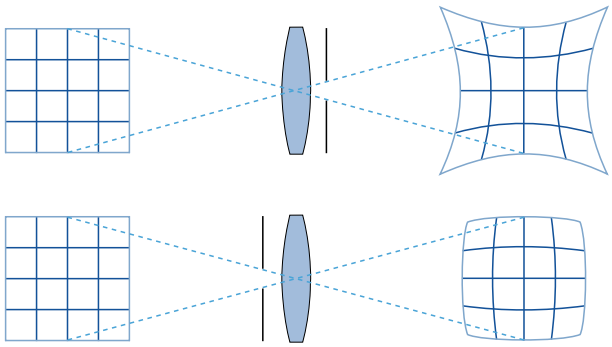


Figure 2.4: Pincushion (top) and barrel distortions (bottom). [9]

2.1.1 Radial distortion

The radial distortion only depends on the distance of a point on a plane to a selected center point (x_0, y_0) on the same plane. This distance is then used to change either the x and y coordinate.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \left(\begin{bmatrix} x_d \\ y_d \end{bmatrix} - \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} \right) \cdot f(r) \tag{2.1}$$

$$r = \sqrt{(x_d - x_0)^2 + (y_d - y_0)^2} \tag{2.2}$$

In term of simplicity we assume that the center of the image plane $(0, 0)$ corresponds to the chosen center point:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} x_d \\ y_d \end{bmatrix} \cdot f(r) \tag{2.3}$$

$$r = \sqrt{(x_d)^2 + (y_d)^2} \tag{2.4}$$

Polynomial model

As the name suggests the polynomial model uses a polynomial function that tries to approximate the distortion of the lens:

$$f(r) = 1 + k_1 \cdot r^1 + k_2 \cdot r^2 + ... + k_N \cdot r^N \tag{2.5}$$

Most models also introduce a further reduction of the polynomial and do only use the even- or odd-order coefficients.

$$f(r) = 1 + k_1 \cdot r^2 + k_2 \cdot r^4 + ... + k_N \cdot r^{2N} \tag{2.6}$$

$$f(r) = 1 + k_1 \cdot r^1 + k_2 \cdot r^3 + ... + k_N \cdot r^{2N-1} \tag{2.7}$$

Such polynomial models are well suited for relatively small distortions but require many parameters to fit a heavily distorted system. There is also no guarantee that an inverse function does exist, therefore we have to rely on numerical methods to find one. [11]

Division model

A similar approach is explained by Fitzgibbon in [12] and uses a polynomial function as a division coefficient:

$$f(r) = \frac{1}{1 + k_1 \cdot r^1 + k_2 \cdot r^2 + ... + k_N \cdot r^N} \tag{2.8}$$

$$f(r) = \frac{1}{1 + k_1 \cdot r^2 + k_2 \cdot r^4 + ... + k_N \cdot r^{2N}} \tag{2.9}$$

$$f(r) = \frac{1}{1 + k_1 \cdot r^1 + k_2 \cdot r^3 + ... + k_N \cdot r^{2N-1}} \tag{2.10}$$

Similar to the polynomial model it is mostly used with even coefficients only, but sometimes also only with the odd ones. All in all, it fits better to high distorted systems and uses fewer parameters. In many cases, a single parameter can be sufficient to be able to describe a lens accurately enough. Nevertheless also here the inverse computation is not analytically possible.

2.1.2 Tangential distortion

Apart from the radial distortion, the next important one is the tangential distortion (fig. 2.5), which occurs if the lens is not parallel to the image plane. [13]

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} (p_1 \cdot (r^2 + 2 \cdot (x_d)^2) + 2p_2 \cdot (x_d \cdot y_d)) \cdot (1 + p_3 \cdot r^2 + p_4 \cdot r^4 ...) \\ (p_2 \cdot (r^2 + 2 \cdot (y_d)^2) + 2p_1 \cdot (x_d \cdot y_d)) \cdot (1 + p_3 \cdot r^2 + p_4 \cdot r^4 ...) \end{bmatrix} \tag{2.11}$$

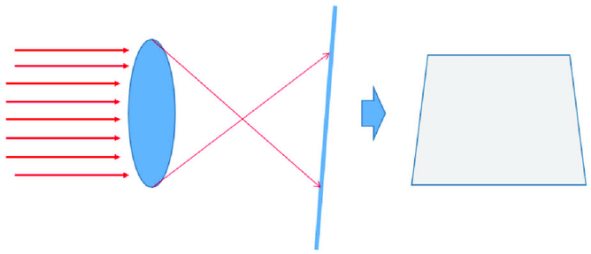


Figure 2.5: Tangential distortion, caused by non parallel positioning of the lens and the screen. [14]

2.1.3 Brown–Conrady distortion model

The Brown-Conrady model combines radial and tangential distortion. More precisely it uses an even-order polynomial-radial distortion as well as the above-mentioned tangential distortion. It is widely used and referred to by many VR platforms (ex. Google Cardboard SDK [15], Unreal Engine [16]). Nevertheless, the tangential part of the model can often be neglected by assuming perfect matching of the lens-plane and the headset's screen.

2.2 Used distortion models

In the software pipeline used in this thesis (sec. 3.5.1), the same distortion model was implemented, which is used by OpenHMD [17] and PanoTools [18]. Indeed it is a 3-parameter radial model using even and odd coefficients. Looking at the equation also a 4th parameter can be seen: k_0 which represents a scale factor but does not distort the image.

$$f(r) = k_0 + k_1 \cdot r^1 + k_2 \cdot r^2 + k_3 \cdot r^3 \tag{2.12}$$

Since the distortion runs on the GPU the radius has to be computed for every pixel. This requires two multiplications, one addition, and the square root. Since normalizing a vector is a very common operation on the GPU, Nvidia has implemented the reverse as well as the normal square root as individual instruction. This significantly improves performance, as otherwise, the calculation of the square root can take some time.

On an FPGA instead, we do not have such an acceleration unit and the square root has to be implemented with logic blocks. To save some area an approximation of the square root can be used (sec. 2.4.2) or as an alternative solution: a model, such as the Brown-Conrady distortion, which only uses even coefficients can be implemented without a square root.

To be able to compare different distortion models the parameters for the Oculus Rift [19] [20], used by OpenHMD, were used as a base point. In figure 2.6 we can see that there was some change in the lenses, starting with the Development Kit 1 (DK1), which had a relatively high distortion, to a lower distortion on the Development Kit 2 (DK2) and the Consumer Version 1 (CV1).

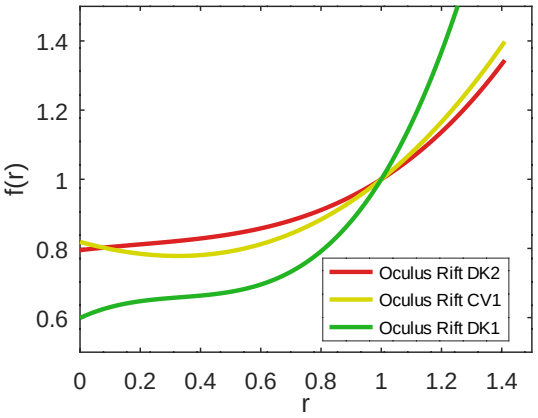
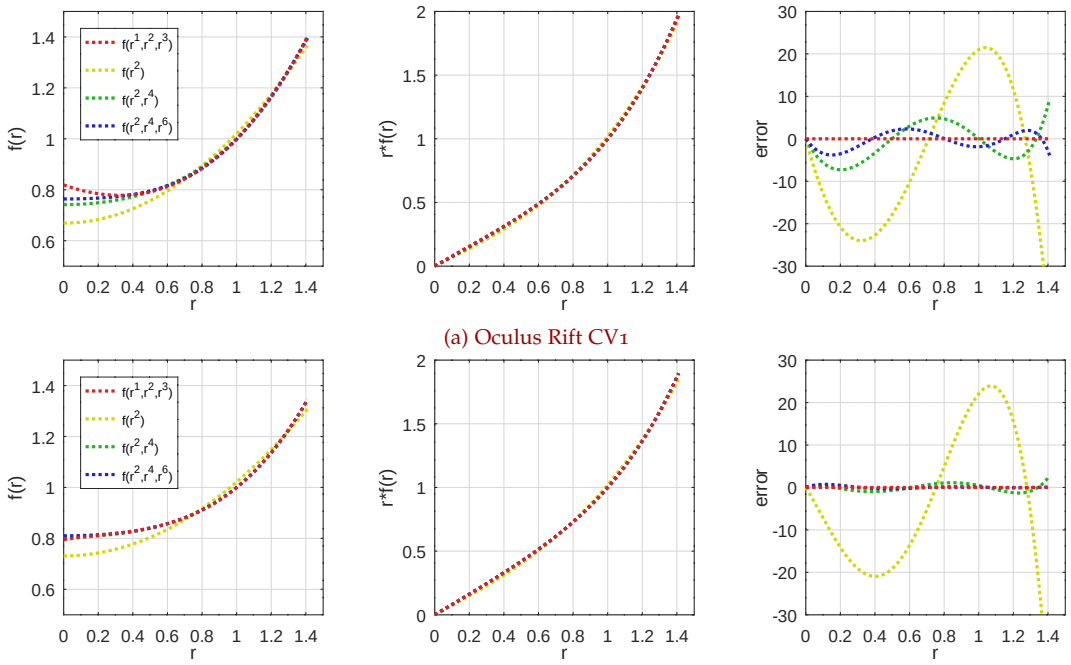


Figure 2.6: Distortion functions of the Oculus Rift DK1, DK2 and CV1 depending on the radius r.

Comparing the modeled functions of the DK2 and CV1 another change can be observed: While the DK2 is nearly monotone, the CV1's distortion does decrease at the beginning and increases after some time. This is also important while comparing different distortion models since some of them can easily model such a lens (polynomial with all coefficients) while others need many coefficients to get a similar behavior (polynomial with even coefficients). This can also be seen in figures 2.7b and 2.7a where even order polynomials are used to reproduce this distortion. Nevertheless, for many lenses, a model using only two even-order coefficients is sufficient.



(a) Oculus Rift CV1

(b) Oculus Rift DK2

Figure 2.7: Comparison of different polynomial lens distortion functions on the example of the Oculus Rift CV1 and DK2. The red lines represent the base-models which were approximated using a different number of even-order coefficients.

2.3 Inverse distortion

Another big problem, that has to do with the distortion, is the inverse distortion. Depending on the used pipeline you either have to distort the image/the vertices inversely or you have to distort the pixels according to the forward distortion. In the used graphics pipeline design both have to be done (sec. 3.5.1), since we have to inversely distort the triangles to compute their bounding box, while the pixel coordinates have to be forward distorted.

Sometimes the forward distortion function has not to be known (ex. Google Cardboard), since the scene is broken up into small triangles which get distorted individually (sec. 3.3). If such an approach is used, the inverse model parameters can directly be determined, while the forward distortion

is not of interest.

Nevertheless, often the forward distortion has to be known and we try to find a model which represents the lens as good as possible. In a second step, the inverse has to be calculated. For the function to be invertible, it has to be bijective, which is not correct for polynomial functions. Therefore an approximation has to be used. Another important characteristic for the forward-distorted function is, that it has to be monotone. Otherwise, several radii would overlap each other, which would result in artifacts. But we have to be careful: not the forward distortion function $f(r)$ itself has to be monotone, but the resulting coordinates x and y. In other words, the forward distorted radius $r \cdot f(r)$ has to be monotone.

In our case the parameters for the Oculus Rift DK2 were used, which result in the following forward distortion function:

$$f(r) = 0.795 + 0.103 \cdot r - 0.145 \cdot r^2 + 0.247 \cdot r^3 \tag{2.13}$$

To get rid of the square root, used to compute r, a polynomial function with even coefficients was calculated. This was done by an error-minimizing algorithm in Octave [21]. As can be seen in figure 2.7b a polynomial with r^2 and r^4 works already very well and differs less than 3 pixels from the original approximation. The original distortion was represented using OpenHMD's parameters for a screen resolution of $1024x1024px$.

$$f(r) = 0.805758802802 + 0.1165743428001 \cdot r^2 + 0.0781130808573 \cdot r^4 \tag{2.14}$$

A similar approach, using a minimizing algorithm, can be used to approximate the inverse of $r \cdot f(r)$. Since this function has only to be calculated on the GPU a polynomial with even and odd coefficients was used as an approximation. Since the maximum radius is $\sqrt{2}$ only the interval from 0 to 1.41 is considered.

Similar to the forward distortion function only a few coefficients are sufficient enough to get a very good approximation with an error (sum of squares) of less than 0.001. Also adding more coefficients does not always help and can cause overfitting [5, p.21]. In our case, a polynomial with 5 coefficient works well for the inverse of the monotone function $r \cdot f(r)$. The coefficient for r^0 was left out on purpose since the function starts at 0 and

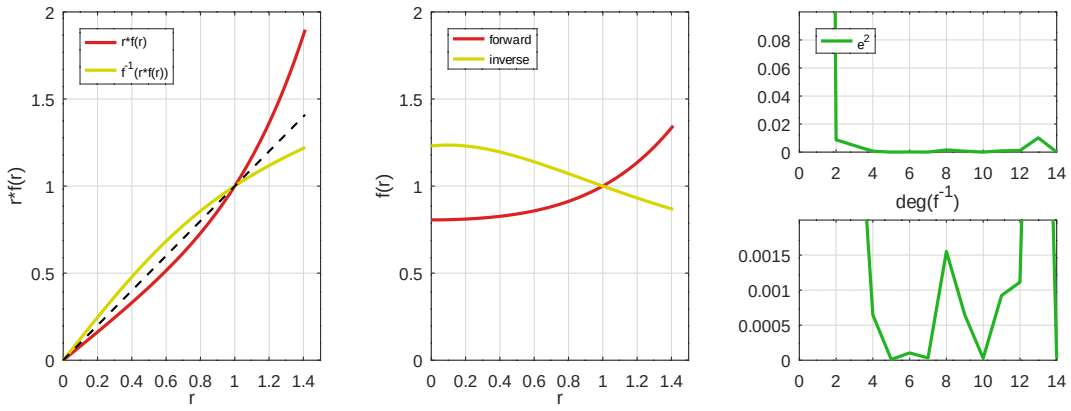


Figure 2.8: The inverse distortion function was approximated using polynomials of varying degrees. A polynomial with $deg(f^{-1}) = 5$ does already fit very well and has an error of less than 0.0001.

does not need an offset. Additionally, we have to divide the resulting inverse by r to get the inverse polynomial which would result in a coefficient for r^{-1} and is not desired.

$$f(r) = 0.8058...r^0 + 0.1166...r^2 + 0.0781...r^4 \tag{2.15}$$

$$r \cdot f(r) = 0.8058...r^1 + 0.1166...r^3 + 0.0781...r^5 \tag{2.16}$$

$$f^{-1}(r \cdot f(r)) = 1.2300...r^1 + 0.1131...r^2 - 0.6244...r^3 + 0.3427...r^4 - 0.0613...r^5 \tag{2.17}$$

$$f^{-1}(r) = \frac{f^{-1}(r \cdot f(r))}{r} = 1.2300...r^0 + 0.1131...r^1 - 0.6244...r^2 + 0.3427...r^3 - 0.0613...r^4 \tag{2.18}$$

2.4 Distortion properties

One of the most important properties of the distortion function is, that the resulting/distorted coordinates should still stay monotone. This means that if one input coordinate (x, y) has a greater radius than another, also the

resulting radius should be greater.

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_{d1} \\ y_{d1} \end{pmatrix} \cdot f\left(\left|\left| \begin{pmatrix} x_{d1} \\ y_{d1} \end{pmatrix} \right|\right|\right) \qquad \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_{d2} \\ y_{d2} \end{pmatrix} \cdot f\left(\left|\left| \begin{pmatrix} x_{d2} \\ y_{d2} \end{pmatrix} \right|\right|\right) \tag{2.19}$$

$$\left|\left| \begin{pmatrix} x_{d1} \\ y_{d1} \end{pmatrix} \right|\right| > \left|\left| \begin{pmatrix} x_{d2} \\ y_{d2} \end{pmatrix} \right|\right| \quad \Rightarrow \quad \left|\left| \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right|\right| > \left|\left| \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \right|\right| \tag{2.20}$$

The monotonicity of a function can simply be proven by looking at its first derivative (fig. 2.9). It should always be positive to be monotone rising. This is either true for the polynomial with even coefficients (eq. 2.14) as well as for the polynomial with all coefficients (eq. 2.13).

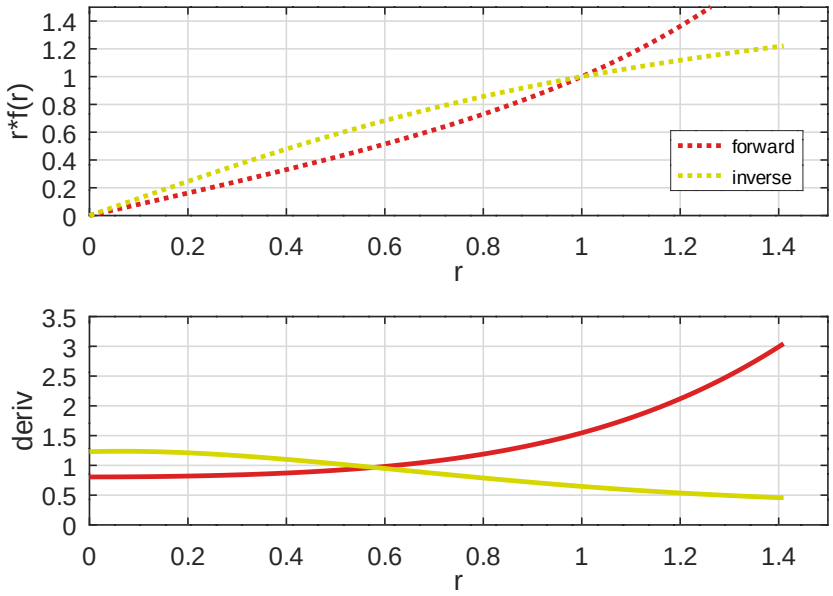


Figure 2.9: The monotonicity was proven for the forward and inverse distortion function by assuring that the derivation does not change its sign.

Until now, this consideration is only true while using floating-point numbers. To either increase repeatability and prevent artifacts, all coordinates of pixels, as well as all vertices, are represented in fixed-point. Therefore we have to check the monotonicity for that too.

2.4.1 Fixed-point distortion

All signed fixed-point coordinates are represented using a 24.8bit fixed-point number. This means that we have:

- 1 sign bit
- 23 pre-comma bits $\Rightarrow 2^{23} - 1 = 8388607$
- 8 post-comma bits $\Rightarrow 2^{-8} = 0.00390625$

Considering these limitations the size of the triangles is also limited, otherwise, we would get an overflow. To be able to apply the distortion to any screen size the coordinates get normalized, with $\pm\frac{screen\_size}{2} \Leftrightarrow \pm 1$. Since the fixed-point number can be interpreted as an integer, just changing the way how we represent this integer, can be used to perform a shift operation. In our case, if we have a screen size of $1024x1024px$ we would choose a 15.17bit representation.

$$\frac{128.0}{\frac{screen\_size}{2}} = \frac{0\ \ 000\ 0000\ 0000\ 0000\ 0100\ 0000\overset{\downarrow}{.}0000\ 0000_{24.8}}{512} = \tag{2.21}$$

$$0.25 = 0\ \ 00\ 0000\ 0000\ 0000\overset{\downarrow}{.}0010\ 0000\ 0000\ 0000\ 0_{15.17} \tag{2.22}$$

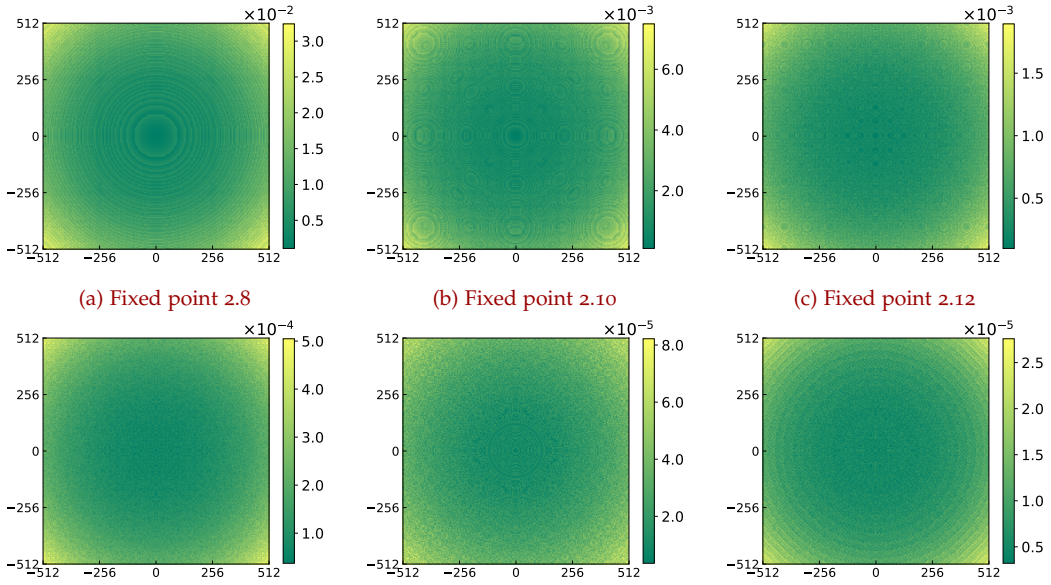
Using this approach no error occurs as long as the coordinate itself can be exactly represented by the fixed-point representation and no post-comma value less than 2^{-8} is present. But in the next step, we have to compute the radius, which includes a multiplication. The problem with multiplications is, that the number of post-comma bits sums up. If we then convert it back to our initial representation we would sacrifice precision.

$$0\ 0.00...001_{2.17} \cdot 0\ 0.00...001_{2.17} = 0\ 00.00...001_{3.34} = 0.0_{2.17} \tag{2.23}$$

Depending on the used distortion function several causes for errors are possible. As mentioned above, one would be the precision loss when multiplications are used. The second one would be the approximation of the square root (sec. 2.4.2) and the third one would be the approximation of the coefficients themselves since they also have to be converted to fixed-point numbers.

In the following window, this error can be seen depending on the number of post-comma bits. Since we normalize the x and y-coordinate to $[-1, 1]$ the number of pre-comma bits can be fixed to 2 (maximum $r^2 = 2$). As shown in figure 2.10 the previously used 8 post-comma bits are not enough for the calculation of the function $f(r)$. It would result in an error of approximately $4 \cdot 10^{-2} \cdot 512px \approx 20px$. Using more bits decreases the error continually by the factor 5 when adding two more bits.

For the screen size of $1024x1024px$ and a tolerated error of $0.25px$ we can have a maximum error of $\frac{0.25}{512} = 4.88 \cdot 10^{-4}$ in comparison to the ideal values. This can be achieved with at least 14 post-comma bits.



(a) Fixed point 2.8 (b) Fixed point 2.10 (c) Fixed point 2.12

(d) Fixed point 2.14 (e) Fixed point 2.16 (f) Fixed point 2.18

Figure 2.10: Error of the distortion function ($f(r) = k_0 + k_1 \cdot r^2 + k_2 \cdot r^4$) in comparison to the ideal distortion, using fixed-point numbers of different bit precision.

2.4.2 Square root approximation

Until now only the distortion function (eq. 2.14) without the need for the computation of a square root was used. The behavior changes drastically

if we need to approximate the square root to be able to use coefficients for r^1 and r^3. There are several algorithms that allow us to approximate the square root very efficiently and keep the error below 0.1%.

Fast inverse square root

One such method would be an algorithm first used in the source code of the first-person shooter Quake III Arena [22]. It does make use of the way how floating-point numbers are stored in combination with a newton iteration step. The floating-point representation is used since it stores the mantissa and the exponent separately. By casting it to `long` and subtract the shifted from a magic number a first guess for the inverse square root is made, which then can be used in the newton iteration. [23]

With this approach, an accuracy of 0.175% can be achieved which can be seen in figure 2.11. By adding a second newton iteration the relative error could be reduced to 0.0005%.

Listing 2.1: Fast inverse square root algorithm used in Quake III Arena.

```
01 float Q_rsqrt( float number ) {
02     long i;
03     float x2, y;
04     const float threehalfs = 1.5F;
05     x2 = number * 0.5F;
06     y  = number;
07     i  = * ( long * ) &y;                       // evil floating point bit level hacking
08     i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
09     y  = * ( float * ) &i;
10     y  = y * ( threehalfs - ( x2 * y * y ) );       // 1st iteration
11     // y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration
12     return y;
13 }
```

Linear approximation

The fast inverse square root algorithm described above works well and outperforms several other implementations. Nevertheless to use it for calculating the non-inverse square root a costly division has to be used. Inspired

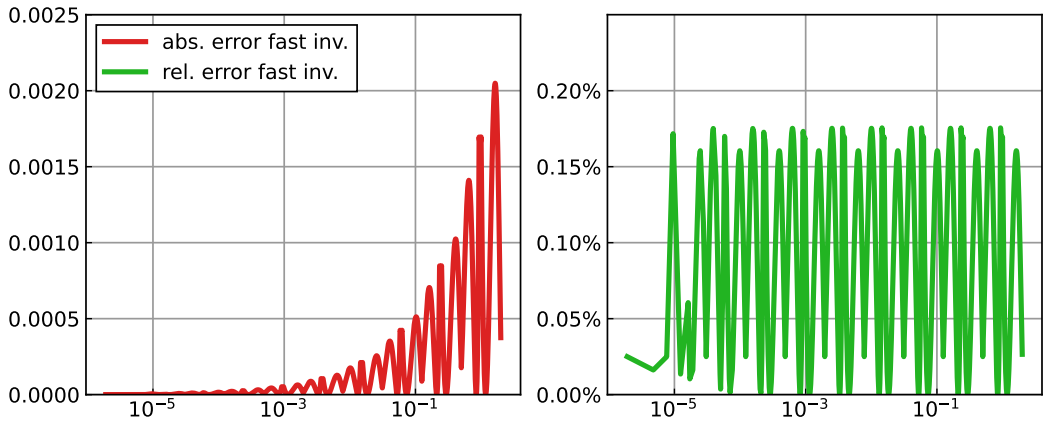


Figure 2.11: Absolute and relative error of the fast inverse square root algorithm using a single newton iteration.

by the algorithm a new approach was developed which tries to approximate the square root directly.

Since we still only have to calculate the square root for values between 0 and 2 we could use a polynomial to approximate it in this interval. One major problem is, that the square root arguments are not evenly distributed. If we consider a squared screen, 78% of the pixels already have a radius smaller than 1. This means that the approximation has to be very accurate for small numbers, but still accurate enough for values above 1.

To surpass this problem of non uniform distribution the floating-point representation was used as a starting point. In a float value, the mantissa gets always normalized to a value between 1.0 and 1.999... Therefore the problem could be split up into the square roots of mantissa and exponent. This reduces the approximation problem to the range of the mantissa. If the value is smaller than 1.0 the exponent will be increased and we end up in the range of 1 to 2 again.

$$x = x_m \cdot 2^{x_e} \qquad x \in [0.0, 2.0] \qquad x_m \in [1.0, 2.0[\qquad x_e \in [0, -1, -2, ...]$$
$$\sqrt{x} = \sqrt{x_m \cdot 2^{x_e}} = \sqrt{x_m} \cdot \sqrt{2^{x_e}} = \sqrt{x_m} \cdot 2^{\frac{x_e}{2}} \tag{2.24}$$

We would be able to approximate the mantissa, but still would end up with non-integer exponents $\left(\frac{x_e}{2}\right)$. This can be solved by lowering the limit of the

mantissa to 0.5.

$$x = x_m \cdot 2^{x_{e2}} \qquad x \in [0.0, 2.0] \qquad x_m \in [\mathbf{0.5}, 2.0[\qquad x_e \in [0, -2, -4, ...]$$
$$\sqrt{x} = \sqrt{x_m \cdot 2^{x_{e2}}} = \sqrt{x_m} \cdot \sqrt{2^{x_{e2}}} = \sqrt{x_m} \cdot 2^{\frac{x_{e2}}{2}} = \sqrt{x_m} \cdot 2^{x_{e2} \gg 1} \tag{2.25}$$

What is left, is to choose a suitable approximation for the square root in the range from 0.5 up to 2.0. A simple approach would be a polynomial again. But since this would need several multiplications an even simpler solution was used: a piece-wise linear interpolation consisting of 4 pieces/5 points.

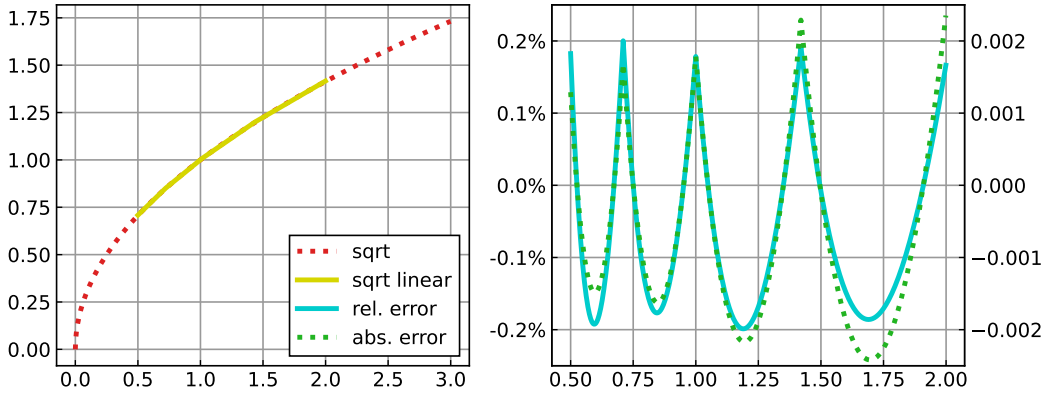


Figure 2.12: Piece-wise linear interpolated square root from 0.5 to 2.0 and the resulting relative and absolute error.

start	end	equation
0.50000000	0.70999146	$x \cdot 0.647171021 + 0.384811401$
0.70999146	1.00000000	$x \cdot 0.543045044 + 0.458740234$
1.00000000	1.41999817	$x \cdot 0.457458496 + 0.544326782$
1.41999817	2.00000000	$x \cdot 0.383911133 + 0.648773193$

Table 2.1: Coefficients for a piece-wise linear interpolation of the square root from 0.5 to 2.

The 5 points were not chosen with a similar distance to each other, since the relative error is more important in the end. In addition, if the ideal value would be chosen for these points the resulting function would be lower than

the ideal one for all other points. Therefore a small offset was added to each base-point to equally bring the error to $\pm 0.2\%$ (fig. 2.12).

Using this linear approximation similar accuracy as with the fast inverse square root can be achieved with an error of 0.175% (fig. 2.13). Using a resolution of $1024x1024px$ results in a maximum error of $\pm\frac{1024px}{2} \cdot 0.175\% = 1.8px$. As a nice side-effect only one addition, one multiplication, and one shift operation are needed.

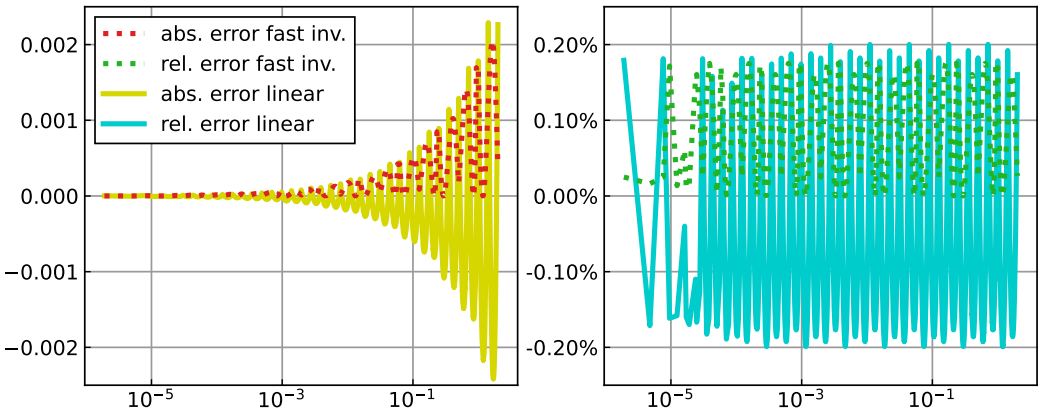
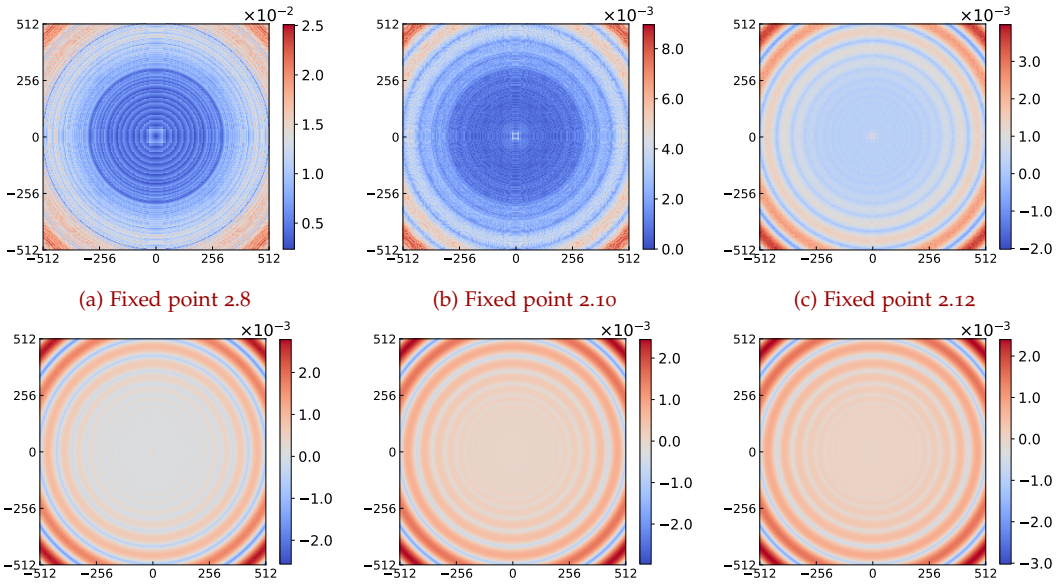


Figure 2.13: Error of the fast inverse square root algorithm in comparison with the piecewise linear approximation.

2.4.3 Fixed-point distortion with square root

Since the square root approximation does not get better using more bits, the error will not get lower than 0.2%. This equals to an absolute error of $\pm 512px \cdot 0.2\% \approx \pm 1px$. In figure 2.14 this can be observed with the error reaching this minimum ($2 \cdot 10^{-3}$) using at least 14 post-comma bits. To further decrease it, a different approximation would have to be chosen.



(a) Fixed point 2.8 (b) Fixed point 2.10 (c) Fixed point 2.12

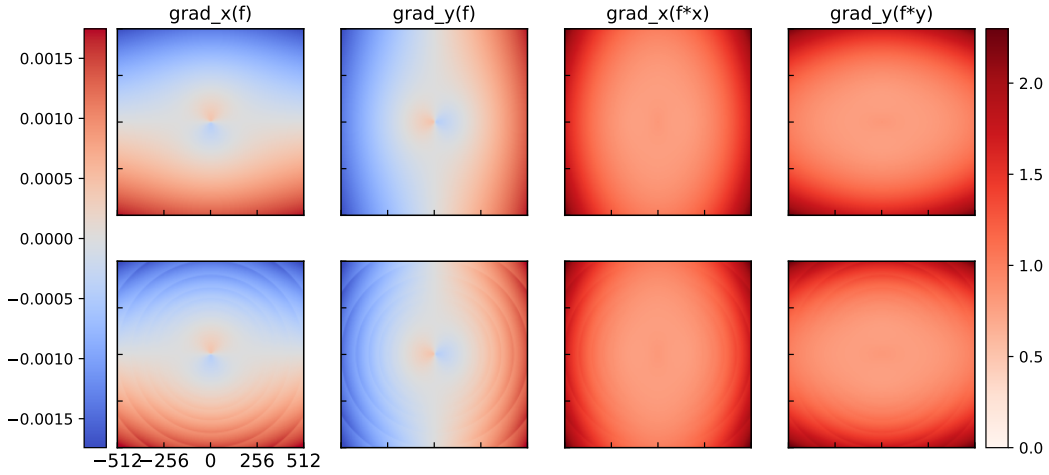
(d) Fixed point 2.14 (e) Fixed point 2.16 (f) Fixed point 2.18

Figure 2.14: Error of the distortion function ($f(r) = k_0 + k_1 \cdot r + k_2 \cdot r^2 + k_3 \cdot r^3$) in comparison to the ideal distortion, using fixed-point numbers of different bit precision.

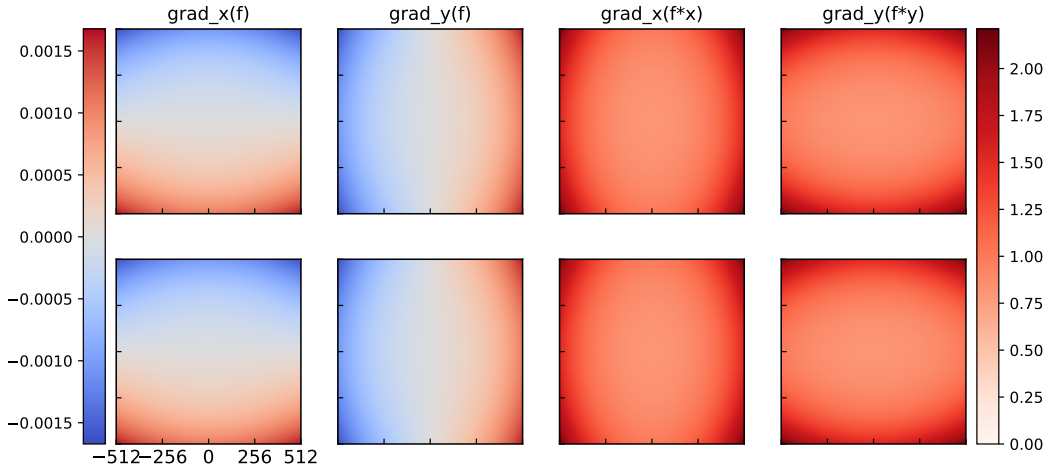
2.4.4 Monotonicity

As already stated previously the monotonicity of the distortion is very important to prevent artifacts, which occur if distorted pixels end up non-monotone regarding to their neighbors. To prove that the pixels' values are monotone the gradient in either x and y-direction can be calculated.

Depending on the distortion function this gradient should be monotone starting from the center. For more complex functions $f(r)$, which are non monotone, the gradient could also change its sign at a specific radius (fig. 2.15a - left). This does not mean that this function is unusable, but it is important that the resulting coordinates $x_d \cdot f(r)$ and $y_d \cdot f(r)$ are monotone (fig. 2.15a - right) and therefore always bigger than 0.



(a) Distortion function $f(r^1, r^2, r^3)$



(b) Distortion function $f(r^2, r^4)$

Figure 2.15: Gradient of the polynomial distortion functions and the resulting coordinates $x_d \cdot f(r)$ and $y_d \cdot f(r)$. The gradient of the resulting coordinates is bigger than 0 for the whole screen which proves monotonicity for the distorted coordinates.

3 GPU pipeline

To be able to discuss the non-linear pipeline created by M. Prettner, M. Steinberger, A. Weinrauch, etc. at the ICG, the normal GPU pipeline, used in modern software, has to be understood first.

3.1 Pixel shader

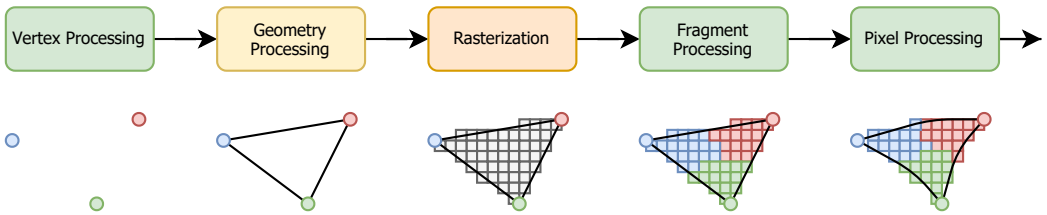


Figure 3.1: GPU pipeline using a pixel shader in a post-processing step for distorting the framebuffer according to the inverse distortion function of the lens.

One of the simplest approaches is a relatively naive one and can be done using any GPU. It does render the scene to a texture instead of the frame-buffer, at a higher resolution than the VR headset. This texture is then passed over to a pixel shader which uses the provided inverse-distortion function to render the texture to the frame-buffer.

A higher resolution is needed to not lose quality on the image at the center, which is the most important part of the screen regarding VR. This comes due to the fact that objects located in the middle of the screen increase in size. On the other hand, objects on the border of the screen will get smaller as can be seen in figure 3.2 and the increased resolution gets "thrown away".

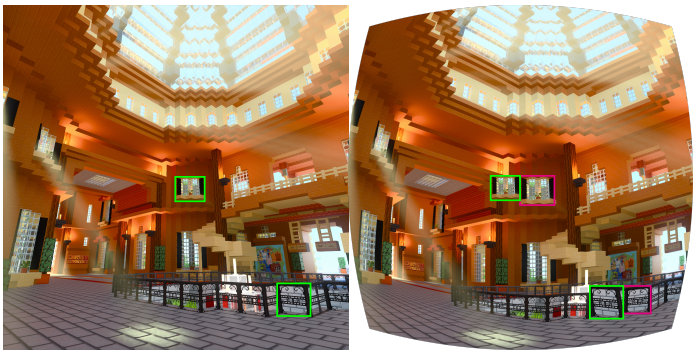


Figure 3.2: If a picture is distorted using the inverse lens-distortion function, objects in the middle of the screen grow while objects on the border shrink.

3.2 Foveated rendering

Until now we assumed that the eye of the user is looking at the center of the screen, but that is not completely correct, since the eye can also move. This, combined with the property, that the human eye does only have "full resolution" at its focused point the rendering resolution can be decreased in the non-focused areas. These areas are also known as peripheral view or fovea. The area in between is used for blending the two images with different resolutions. [24]

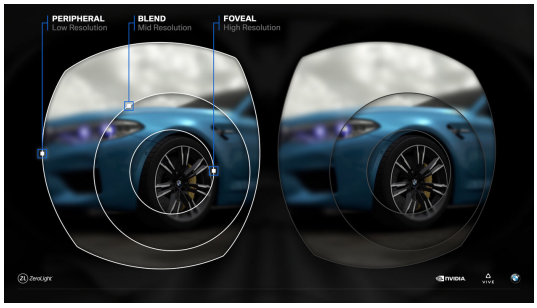


Figure 3.3: Example of a foveated renderes VR image. [25]

There are two approaches how such a foveation can be implemented. Either by tracking the eye using a sensor or by simply assuming that the eye will be focused at the center. It can improve frame times, compared to the previous approach, by a factor of 1.5 and either become more than two times faster if

other techniques, such as eye-dominance-guided foveated rendering (EFR), are used. [26]

3.3 Vertex displacement

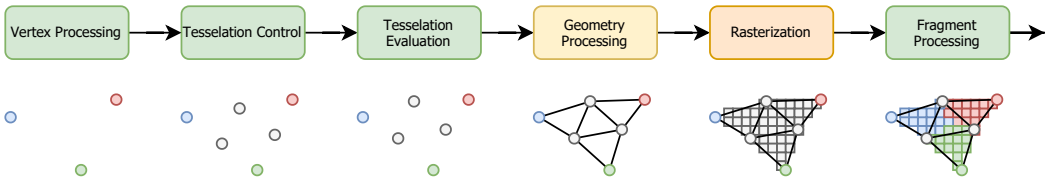


Figure 3.4: GPU pipeline using vertex distortion in the vertex or tesselation shader according to the lens-distortion function.

Previously discussed pipelines did not include changes to the geometry and vertex processing at all. They perform the distortion and blending in a separate pixel processing step. This additional step is time-consuming and the increased resolution, needed to fully utilize the possibility of the VR headsets screen, does sacrifice performance too.

To remove this step the inverse-distortion can be implemented in the vertex shader instead. This is a relatively lightweight operation since the distortion has only to be computed for every vertex and not for every pixel. The resulting image is not completely correct, since a straight line between two vertices will result in a straight line again, but will not be noticeable if the triangles are small enough. This can be done by increasing the triangle count in the tesselation shader. After that, the distortion can be applied to the old and new vertices in the tesselation evaluation shader. It is heavily used in mobile devices which can not handle an additional processing step [27].

3.4 Ray-tracing

With new hardware evolving over the years, especially with Nvidia RTX [28], ray-tracing came back to graphic pipelines. With this specialized hardware

and accelerated memory structures, it would be possible to do exactly that, what vertex displacement (sec. 3.3) tries to avoid: distorting every pixel. This is costly in a rasterization pipeline since every triangle is plotted to the virtual screen after the other. With ray-tracing, we have the advantage that we can send a ray for each of these pixels which are counter-checked with every triangle in the scene. This ray can easily be distorted only once, using the forward-lens-distortion function similar to the visual-ray from the user's eye would be distorted by the VR headsets lens. Therefore no inverse distortion of the triangle is needed and they remain "straight".

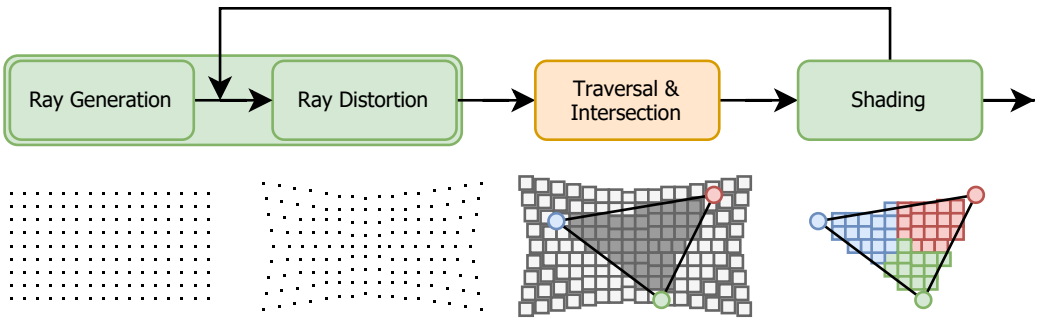


Figure 3.5: GPU raytraced-pipeline which distorts the rays using the forward lens-distortion function. [29]

Since the ray-tracing performance for high-end modern GPUs is at about 10 $GigaRays/s$, real-time ray-tracing is already possible [30]. To draw a scene without any special light effects 1 ray per pixel is needed. This would result in approximately 0.5 $GigaRays/s$ for a 4k resolution at 60fps. Therefore more rays can be sent out for one pixel and reflections and shadows become possible. Using this approach the ray-tracing bandwidth gets saturated really quick on such high-end cards too.

A much smoother experience, also for lower-end cards, can be achieved with a lower resolution for the ray-traced effects and a normal rasterization pipeline at a high resolution. Of course, both have to compute the distorted scene with the same distortion function.

3.5 Non-linear distortion pipeline

Inspired by the ray-tracing approach, which distorts the pixel center and therefore also the ray from the eye to the pixel-center, a non-linear rasterizer was designed. Due to the fact that the rasterizer is a fixed hardware component, it can not be changed and a software rasterizer, running on the shader-cores of the GPU, had to be implemented. It allows us to individually distort the x and y-coordinate of each pixel on the virtual screen used by the rasterization algorithm. The rest of the pipeline can be left similar to a normal graphics pipeline since the distortion does only happen in the rasterization step [5, p.9].

Compared to the other approaches discussed in this work the sample resolution has not to be increased and also no tesselation has to take place for big triangles. This does either reduce the number of triangles as well as the number of produced fragments that have to be processed using quad-shading. The second one is a big drawback regarding vertex displacement.

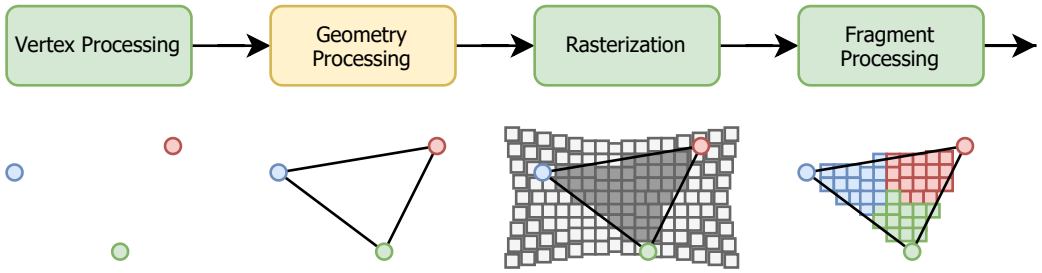


Figure 3.6: Non-linear pipeline distorting the pixel coordinates in the rasterization procedure to achieve lens-distortion.

3.5.1 Hierarchical non-linear distortion pipeline

Since the built-in rasterizer can not be used as a non-linear rasterizer, computing the distortion while rasterizing each triangle can be very costly. Also algorithms (scanline, centerline [2]) to improve performance can not simply be adapted to the non-linear behavior. This is due to the fact that neighboring pixels do not have a common coordinate and the difference

of distortion can not be easily computed. To still be able to get adequate performance a hierarchical, tile-based approach was implemented.

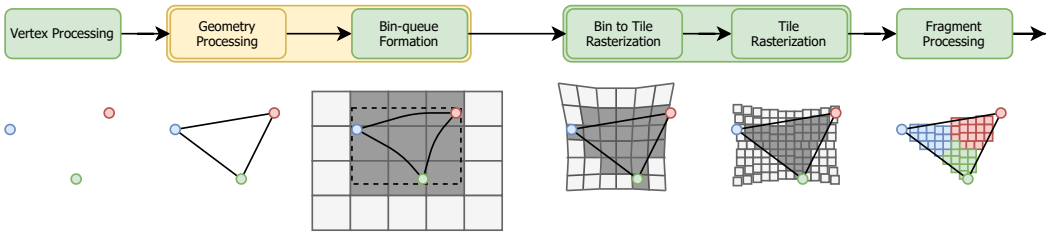


Figure 3.7: Hierarchical, non-linear pipeline distorting the pixel coordinates in the rasterization procedure to achieve lens-distortion.

Geometry processing and bin-queue formation

After the geometry processing, the vertex displacement method (sec. 3.3) is used to (inversely-)distort the triangle vertices. Using its bounding box the non-distorted triangle is queued in the overlapping bins. Each bin represents a similar-sized part of the screen (ex. $64x64px$). This would already be part of the rasterization step but was moved to the geometry processing instead because it can be cheaply done at the end of it together with clipping/culling and converting the triangles attributes to fixed-point representation [5, p.12]. To ensure that no bin is missed, because of inaccuracies of the inverse distortion, a small margin is added to the bounding box of the distorted triangle.

Bin-to-tile rasterizer

The newly created software-rasterizer was also split into two parts, where at first, bigger tiles get rasterized before going over pixel by pixel. This technique was first implemented by PowerVR and is very famous among mobile GPUs since their performance and energy usage are much lower compared to desktop GPUs. But due to the increasing resolution and performance requirements, from 2014 on also their desktop counterparts switched to such approaches [3] [4].

In contrast to the bin-queue formation, the tiles get distorted instead of the triangles. If no distortion would be applied it would be easy to check whether the tile lays completely inside, partly inside, or completely outside the triangle. This can be done using the edge equation algorithm [2, p.20], which allows us to determine if a point is inside a triangle, on each corner of the tile. The parameters for each edge of a triangle can be computed out of two vertices (a and b). Evaluating equation 3.3 with the coordinates of the point p a value e depending on the distance from the point to the edge can be determined. If this value is positive the point is on the left side of the edge, otherwise on the right.

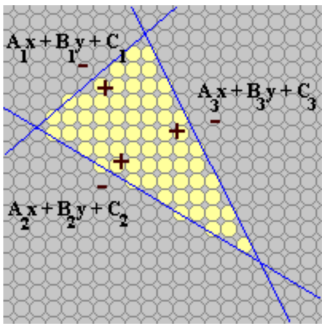


Figure 3.8: Triangle edge equations which can be used to determine if a pixel is in the triangle or not. [2]

$$A_1 = a_y - b_y \qquad B_1 = b_x - a_x \tag{3.1}$$

$$C_1 = \frac{A_1 \cdot (a_x + b_x) + B_1 \cdot (a_y + b_y)}{-2} \tag{3.2}$$

$$e_1 = A_1 \cdot p_x + B_1 \cdot p_y + C_1 \tag{3.3}$$

The evaluation, therefore consists of 3 edge equations that have to be calculated for all 4 corners. In total this would require 12 evaluations which have to be compared with zero. Afterwards we can distinguish three cases:

- The tile is **full out** if all corners are on the right side of **one** edge.
- The tile is **full in** if all corners are on the left side of **all** edges.
- Otherwise the tile is **partly in**

Since we do not need to distinguish tiles, that are partly inside from full-inside ones we can simplify the evaluation a little bit. This is possible since for each slope of an edge, depending on A and B, one corner has to be the nearest looking from the right side of the edge (fig. 3.9). If this corner is outside on that edge all others have also to be outside. So we do only have to evaluate the equation 3 times:

Listing 3.1: Edge-equation evaluation for a rectangle and its corners.

```
c[4]    ... 4 corners: x,y
eeq[3]  ... 3 edge equations: A,B,C

in[3]  = [true, true, true]
out[3] = [false, false, false]
for i in [0,2]
    for j in [0,3]
        e      = eval_eeq(eeq[i], c[j])
        left   = e >= 0.0
        in[i]  = in[i]  && left
        out[i] = out[i] || left

if all_true(in)
    return "full in"
if one_false(out)
    return "full out"
return "partly in"
```

```
for i in [0,2]
    p = nearest(c, eeq[i].A, eeq[i].B)
    e = eval_eeq(eeq[i], p)
    if e < 0.0
        return "full out"

    return "full/partly in"
```

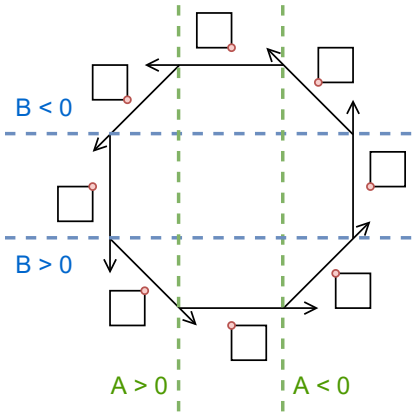
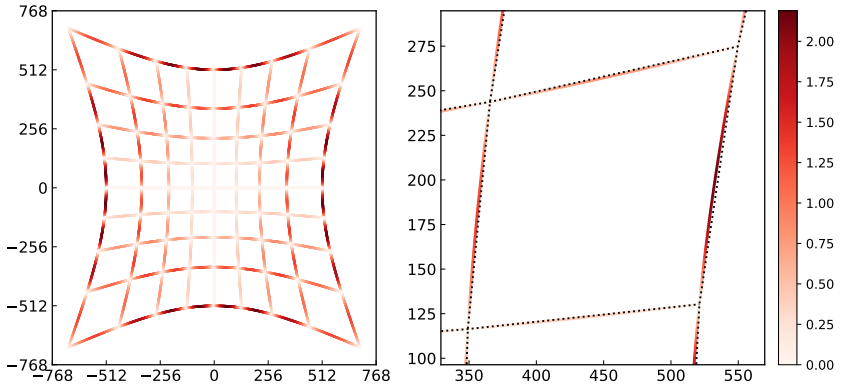


Figure 3.9: Nearest tile corner (red circles) depending on the edge-equation coefficients A and B.

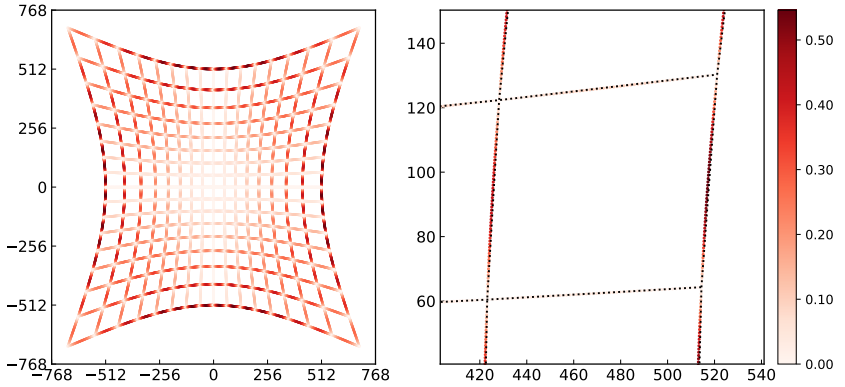
This simplification can only be made if the tiles bounding rectangle, as well as the edges of the triangle, are linear. This, of course, is not true for the distorted case. Therefore the approach with 12 evaluations has to be used, which still has some corner cases where it does not work since the tiles are no longer rectangular.

Going linearly from one tile-corner to the next one the error, between the distorted edge of the tile and the line between the two distorted corners, gets bigger. It reaches its maximum in the middle of the edge (fig. 3.10a).

Depending on the edge of the tile this is not a problem, since the linear tile does cover the distorted edge (right, top). But the other two edges are distorted in such a way, that up to 2 pixels are lost if only the corners are considered to determine if a tile intersects a triangle or not. Nevertheless, if we decrease the tile size to 64 pixels the error becomes approximately $0.5px$ (fig. 3.10b). Since the pixels get rasterized at their mid-point ($+0.5px$) this can be neglected.



(a) tile size = 128



(b) tile size = 64

Figure 3.10: Distance from the real/distorted tile edge to linear tile edge formed by its corners.

This is not a general limit but is true for the used distortion equation 2.14. In our case also the tile size is much smaller with $8x8px$. But since the bin-queues are formed conservatively using the triangles bounding box, the

corners of the bin can also be used to throw away bins using the forward distortion, as long as they are smaller than $64x64px$.

In the end, one small simplification can still be made, since we have to use the approach with 12 evaluations per tile. Neighboring tiles have two common corners which do not have to be computed twice. This results in $(tiles_x + 1) \cdot (tiles_y + 1)$ corners to be evaluated. In our case, these are 81 corners and 243 evaluations instead of 768.

$$\left(\frac{64px}{8px} + 1\right) \cdot \left(\frac{64px}{8px} + 1\right) \cdot 3 = (8+1) \cdot (8+1) \cdot 3 = 243 \tag{3.4}$$

Tile rasterizer

In the last hierarchical step, each pixel is evaluated for every tile which still is partly or fully inside the triangle. In that step, the edge-equations are again evaluated using the distorted pixel coordinate. If the result of all three equations is positive other parameters can be calculated, such as depth and barycentric coordinates. The second ones are used to interpolate vertex attributes across a triangle and can be efficiently computed out of the edge-equations results e_1, e_2, e_3.

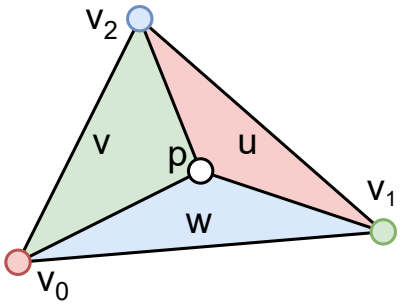


Figure 3.11: Barycentric coordinates u, v, w of a triangle which can be used to interpolate vertex-attributes depending on the location of the point p inside the triangle.

There is no real documentation of how modern hardware does compute such attributes, but most likely it is not done in the hardware rasterizer, since

to be able to get accurate results, floating-point calculations are necessary. A possible way to efficiently implement the calculations without adding lots of floating-point units to the rasterizer is to use the already existing ones in the shader cores. Some data would anyway have to be passed to the fragment shader running on these cores and it would make much more sense to pass the triangle id and/or the vertex attributes together with e_1, e_2, e_3 to a "pre-fragment shader" which then interpolates the attributes using the floating-point units. In our implementation, this can be done directly in the rasterizer, since it is a software pipeline running explicitly on the shader cores. [5, p.23]

Fragment shader

In the last step, each computed fragment is sent to the fragment shader. Depending on the implementation an early depth test can be performed before this shader. Such an early depth test would be fully applicable to the tiled structure, where each tile-processing or even bin-processing unit has its own small depth buffer. Otherwise, the depth test is done after shading the fragment.

4 Hardware implementation

The hierarchical non-linear distortion pipeline (fig. 3.7) was fully implemented as a software pipeline. This allows very high flexibility in each stage of the pipeline, which can be modified to whatever needs that will come up in the future. Nevertheless, this often comes with a relatively big performance penalty, since the dedicated acceleration unit, such as the hardware rasterizer, was optimized for decades [5]. Therefore the idea came up if it would be possible to change the pipeline in such a way, that it will work similarly to a "normal" pipeline. This implies that the rasterizer is a dedicated hardware component that makes use of the great parallelizability of such a stage.

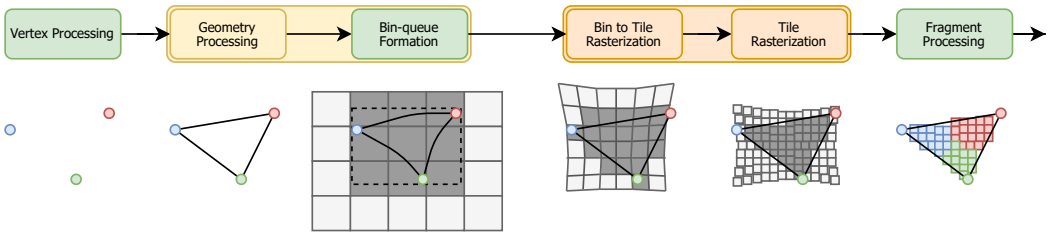


Figure 4.1: Hierarchical, non-linear pipeline distorting the pixel coordinates in the hardware rasterization procedure to achieve lens-distortion.

To be able to compare the needed modifications a design of a normal hardware rasterizer would be needed. Although there are some hardware implementations for GPUs [2] [31] [32], that are mostly very big projects or very small ones. Still, many hints were taken by the recursive, tile-based design by A.N. Torrentó design [2, p.30]. In the end, an individual design was created which consists of a base structure and can be configured as linear or distorted rasterizer.

4.1 Characteristics

To be able to show the difference in terms of needed area on the FPGA both, the linear and the distorted design were synthesized for the same hardware. The design was optimized for a design with one of the newest 16nm architecture by Xilinx: UltraScale+ [33]. Apart from many Controllable Logic Blocks (CLBs) or Look up Tables (LUTs) and Flip-Flops, there is also a bunch of specialized hardware which can be used to optimize the design.

4.1.1 BRAM

The Block RAM (BRAM) is a very fast Random Access Memory (RAM) embedded in between the logic cells of the FPGA. It can be read and written at the same clock as the flip-flops with no delay, which makes it perfect for very fast memory access. In some cases, it could also be used as a cache for an external RAM module. Normally the area is limited and therefore also the amount of BRAM, which in the best case consists of some MegaBytes.

It is used to store the bin-queues, coming from the software pipeline, as well as temporary results between the bin-to-tile and the tile-rasterizer. The BRAM can be accessed using a single-port, simple-dual-Port, or true-dual-port structure. Using the simple-dual-port structure the memory acts similar to a unidirectional cache where the first port can be used to write and the second one to read in each cycle.

4.1.2 DSP

Another important part of an FPGA are the Digital Signal Processing Units (DSPs), which allow us to accelerate a variety of arithmetic and logic operations. In the UltraScale+ architecture, so-called DSP48E2 instances are used, which stands for a 48bit DSP of version 2. It can be configured to perform several different operations using the input ports A, B, C, and D, as well as cascaded operation using the additional ports connected to other slices placed right above and below on the FPGA.

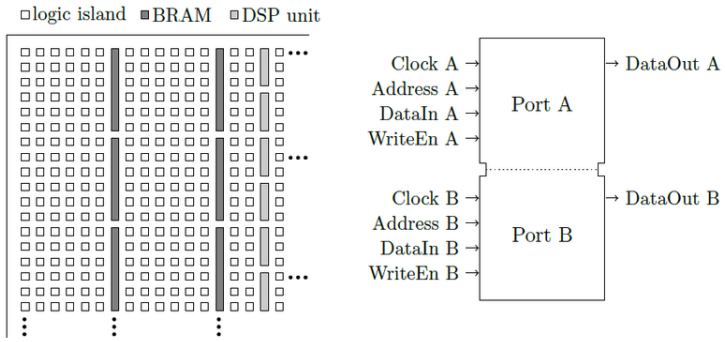


Figure 4.2: The Xilinx UltraScale+ architecture includes lines of BRAM and DSP cells. They are interconnected to allow cascading and have a specified port-schema. [34]

In the designed hardware several of such functions were used, with the most important one being the multiplication functionality. The use of a DSP can be identified by the synthesizing tool or also placed directly. This does only work efficiently if also the operands are sized properly. This means that the operand has to be resized to the corresponding signed-bit-vectors before using any arithmetic in the Very High Speed Integrated Circuit Hardware Description Language (VHDL) code.

- One DSP slice:
  - $27x18$bit multiplication: $P = A_{27} \cdot B_{18}$
  - 48bit addition: $P = A_{30} : B_{18} \pm C_{48}$
  - $27x18$+48bit multiply-accumulate: $P = A_{27} \cdot B_{18} + C_{48}$
- Two DSP slices:
  - $35x27$bit multiplication:
    $P_1 = A_{27} \cdot B_{34:17} = P_{CIN}$
    $P_2 = A_{27} \cdot B_{34,16:0} + (P_{CIN} \gg 17)$
  - $35x27$+48bit multiply-accumulate
  - $44x18$+48bit multiply-accumulate

There are still many other logic operations (xor, nor, and), shift operations (bus-shift, barrel-shift) as well as pattern-detection functionalities left which are described in the FPGAs documentation [35, p.45].

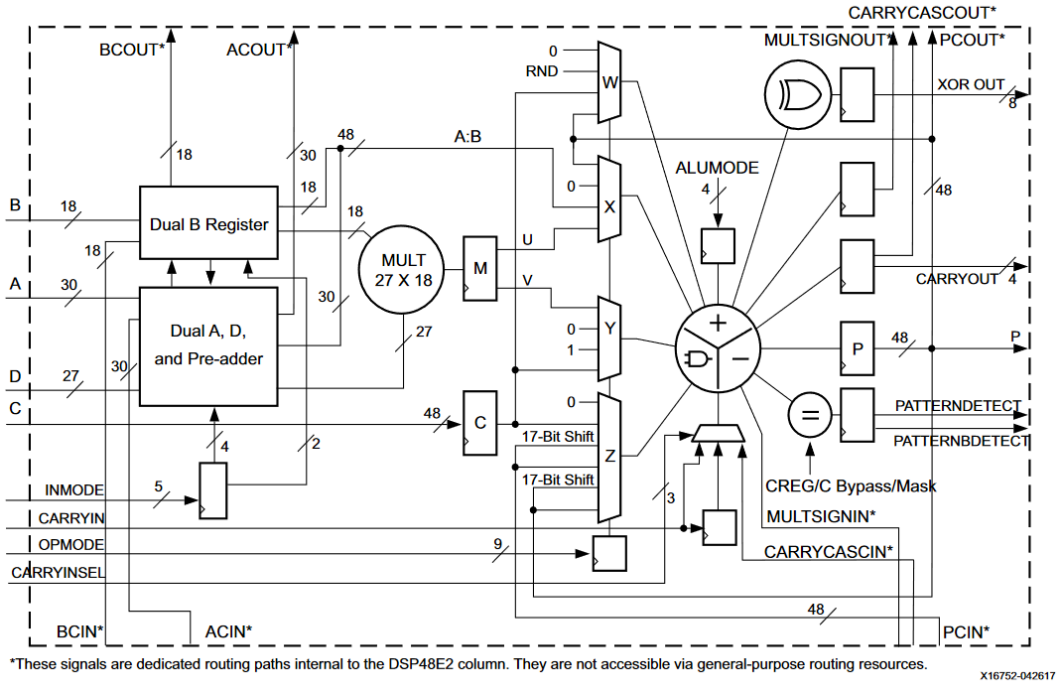


Figure 4.3: DSP48E2: 48bit DSP structure of version 2 which is used in UltraScale+ FPGAs. it can be configured to execute multiplication, addition and several logic/pattern-recognition functions. [35]

DSP48E1

Older FPGAs do also use an older version of DSPs. The major difference is that the multiplier does only allow $25x18$bit multiplications instead of $27x18$bit. To be able to run the code on both architectures my design was optimized for such DSPs too.

4.2 Structure

The design was created with hierarchical processing in mind (fig. 4.4). Beginning with a large global memory, which could be a large cache in the GPU, the bin-to-tile rasterizer modules begin to process one bin at a time. After retrieving the triangle, the edge equations are calculated as well as a bitmask, where one bit represents one tile. Both are stored in a smaller local memory which is then used by the tile rasterizers. They go through every pixel of a tile and check if it is inside or outside the triangle. If it is inside, the pixel coordinates as well as some additional data (if needed) are stored in an even smaller tile memory.

In the following sections, each module is described in more detail. To simplify it a little bit the screen-, bin- and tile-size were fixed to 1024, 64, and 8 pixels. All of them can be changed individually using VHDL-constants.

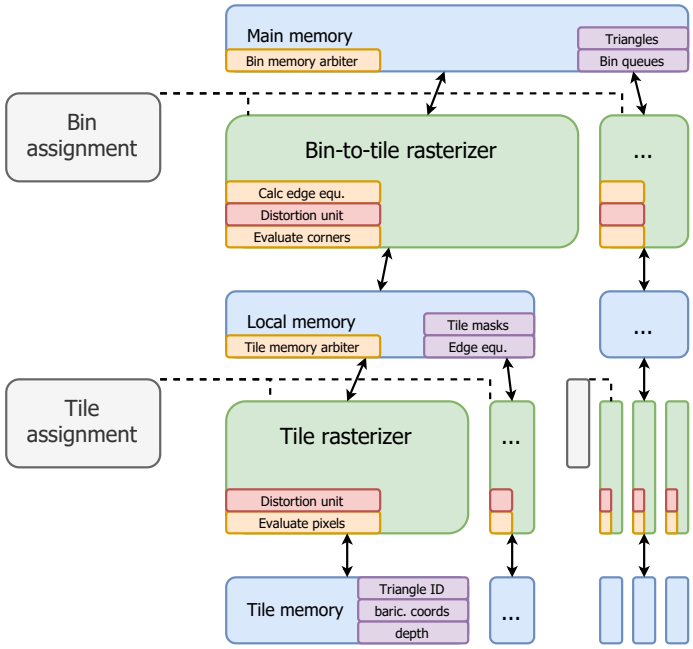


Figure 4.4: Hierarchical structure of the implemented hardware component, starting with the main memory as interface to the geometry shader.

4.2.1 Main memory

The main memory is the interface between the geometry processing, happening in the programmable stage, and the hardware rasterizer. As described in section 3.5.1 also the bin-queues are formed in this step. They represent a list of indices for each bin of the screen. The size of a list is not fixed, therefore also the offset and the length of each bin are stored separately. Using such an index allows us to access the data of one triangle from the main memory, which was already converted to a fixed-point.

At this stage the number of triangles and the maximum number of indices was limited, to be able to estimate the needed memory. This is necessary since BRAM is used, which is limited in size. In a more generic system, this memory would most probably be replaced by cached Graphics Double Data Rate RAM (GDDR).

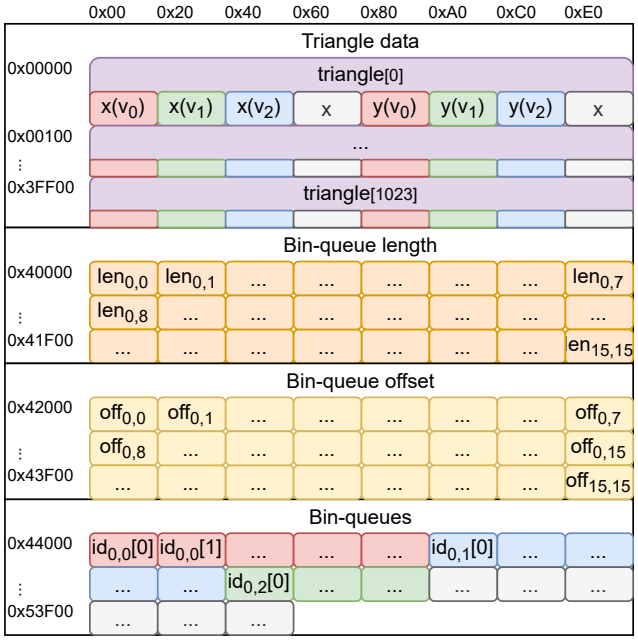


Figure 4.5: Main memory structure including triangle-data, bin lengths, offsets and indices.

The memory structure can be seen in figure 4.5 and is formed by many 32bit values. In the first memory section up to 1024 triangles can be stored,

which contain the fixed-point coordinates of each vertex. To be able to efficiently calculate addresses and safe memory space, eight 32bit numbers were chosen for one triangle and other vertex attributes (depth, color) were left apart. This makes a triangle 32Byte in size and the whole chunk 32kB.

Due to the fixed testbench (sec. 4.2), $16 \cdot 16 = 256$ bins have to be processed, which all have their individual bin-queue. The length and the offset are stored in the succeeding memory chunks and require $2 \cdot 256 \cdot 32\text{bit} = 2\text{kB}$. The bin-queues themselves and therefore many triangle indices are then stored with the corresponding offset in a big array which was also limited to twice the triangle count ($2 \cdot 1024 \cdot 32\text{bit} = 8\text{kB}$).

Main memory arbiter

Since not only one bin-to-tile rasterizer module has to access the main memory at a time, but multiple ones, some kind of arbitration has to take place. The BRAM modules on the FPGA do have a true-dual-port RAM module, which allows us to use two ports in parallel. But, since the initial memory would have to be written by the geometry stage, only one can be used.

A very simple arbitration algorithm was chosen since no real benefit can be observed if one bin-to-tile-rasterizer module would be faster than the others. The first module got the highest priority which decreases down to the last module with the lowest priority. To be able to detect if a module wants to access the memory, a request signal is set by each module. The arbiter then acknowledges the first one. In addition, it will also acknowledge all other modules which try to read/write to the same address.

A similar approach would be an algorithm using a counter which shifts the priorities of each module every cycle. Nevertheless, no real performance difference was observed and the previous approach was used which requires fewer logic blocks.

4.2.2 Bin assignment

The bin assignment module is the main controller of the rasterizer. If it gets enabled it will start distributing the work to the different bin-to-tile rasterizer units. If one of them is ready it will get the next bin.

4.2.3 Bin-to-tile rasterizer

The bin-to-tile rasterizer waits for the ready signal, which comes from the bin-assignment module. Together with that signal, it will also get a bin number. In our case two numbers, for x and y-direction, ranging from 0 to 15. First of all the amount of triangles ($\texttt{len}_{x,y}$) of the corresponding bin has to be retrieved from the main memory. If at least one triangle is present in the bin-queue, the offset ($\texttt{off}_{x,y}$) is read. Then each triangle is fetched and rasterized according to the algorithm described in section 3.5.1.

Calculate edge-equation

To be able to use this algorithm the edge-equations parameters A, B and C have to be calculated. The parameters of one edge depend on the x and y-coordinate of two vertices. Using a subtraction $A = a_y - b_y$ and $B = b_x - a_x$ can be calculated easily. To avoid a big subtraction logic, the 48bit adder functionality of the DSP slices can be used. The inputs have to be `signed(47 downto 0)` which perfectly suits the signed, 32bit, fixed-point representation used for the coordinates (24.8). The remaining 16 bits can be propagated using the first bit (sign bit).

Calculating $C = \frac{A \cdot (a_x + b_x) + B \cdot (a_y + b_y)}{-2}$ is a bit harder and requires three additional adders, two multiplications, and a shift operation at the end.

All of these operations can be done by DSPs to some extend. To be able to use the 35x27bit multiply functionality using two DSP slices, one of the inputs has to be 27bit only. This can be achieved by limiting the parameters A and B to a 19.8 representation, which limits them to a maximum of

$2^{18} - 1 = 262143$. Such a limitation could be solved by properly clipping the triangles which are that big (260 times bigger than the screen).

By tweaking the bit-lengths a little bit and using the multiply-accumulate functionality, a total of 8 DSPs and one shift-logic have to be used. Four of those are used to perform the addition and subtraction of the coordinates, two for multiplication and another two for the multiply-accumulate. There are still some issues with the used synthesizer, which does not always detect such multiply-accumulate instances. Therefore it was removed with the drawback of 9 instead of 8 used DSPs.

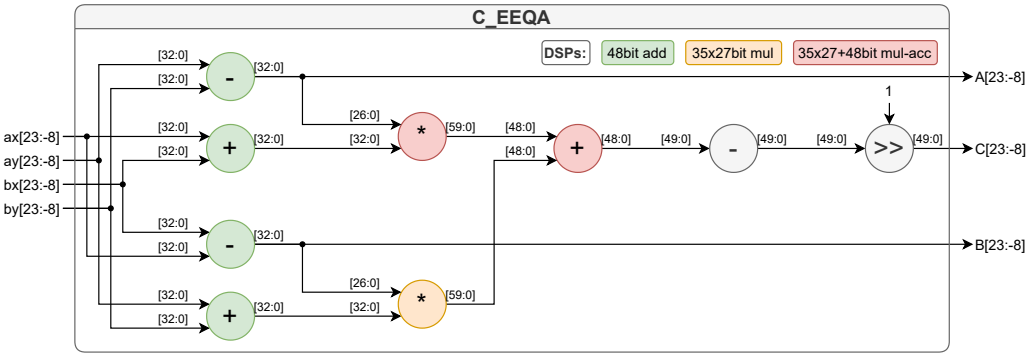


Figure 4.6: Schematic of the edge-equation parameter (A, B, C) calculation. It uses 4 DSPs in adder configuration, 2 cascaded DSPs as multiplier and another 2 as multiply-accumulate unit.

Evaluate bin corners

Normally the previously calculated edge-equations would be used to evaluate one corner, out of 81, after the other. But, to efficiently optimize the design, before evaluating the left bottom corner $c_{0,0}$, all other edge corners ($c_{8,0}, c_{8,8}, c_{0,8}$) are evaluated. They can then be used to detect bins that are completely inside/outside the triangle as described in section 3.5.1. If the bin intersects the triangle, all other 81 corners can be evaluated.

The formation of the corner coordinates represents the first step of the evaluation process. As a second step the coordinates have to be distorted (sec. 4.3). Depending on the used algorithm (no-distortion, $f(r^1, r^2, r^3)$, $f(r^2, r^4)$)

this can take zero to 4 cycles. Afterwards, the distorted coordinates can be used to evaluate the edge equations. This can be fully parallelized using multiple DSPs. In total $3 \cdot 6 = 18$ DSPs have to be used, since every equation needs two $35x27$bit multiplications as well as two 48bit additions. Again, to be able to use the multiplication functionality one of the two 32bit fixed-point numbers has to be shortened to 27bit. Since we know our maximum display resolution, the coordinates in the whole design are stored using less than 27bit, which solves this problem.

For each edge-equation, a bit vector is formed with 81bit containing the sign of the evaluation. The sign indicates if the corner is on the left (inside the triangle) or right (outside the triangle) side of the edge.

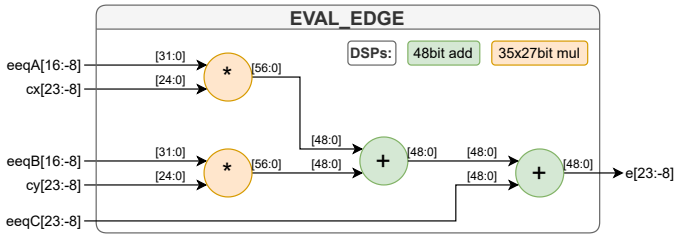


Figure 4.7: Schematic of the edge-equation evaluation $e = A \cdot c_x + B \cdot c_y + C$. Two cascaded multipliers ($2 \cdot 2$ DSPs) and two 48bit adders are used for each evaluation unit.

Generate tile mask

In the end, the three bit-vectors can be evaluated and a new 64bit tile-mask is created, which indicates if a tile is inside (1) or outside (0) the triangle. The way how to create this new vector is purely logical and does not require any additional arithmetic, which makes it fit nicely in hardware. The function of how it is formed can be found in listing 3.1.

This vector, together with the edge-equation parameters, is then stored in a local memory that is only accessible inside the bin-module (sec. 4.2.4).

Pipelining

To be able to operate the FPGA at a relatively high frequency the arithmetic part has to be split up using registers. This makes the evaluation very slow since every corner takes several cycles to be computed. This can be prevented by introducing a pipeline, which allows us to use the hardware more efficiently. In the following figure, this pipeline can be seen using a 3-cycle distortion unit.

In addition, the coordinate distortion can already start while loading the triangle vertices, since the resulting edge equations are not needed until the first evaluation. The edge-equation calculation unit is also reused instead of placing it three times because after loading the triangle the pipeline still needs 3 cycles until the edge-equation parameters are needed.

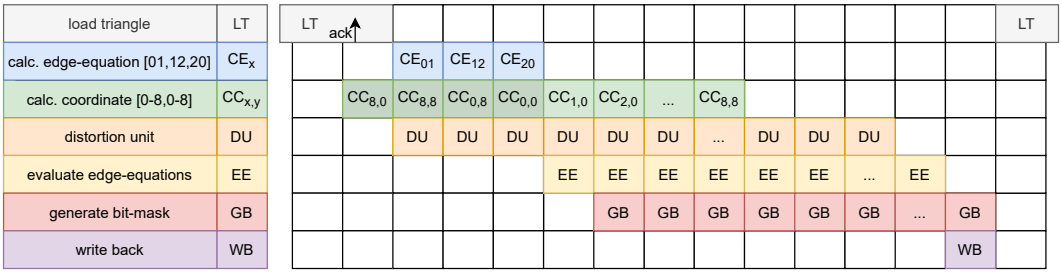


Figure 4.8: Used pipeline in the bin-to-tile rasterizer using a 3-cycle distortion unit.

4.2.4 Local memory

The local memory, which belongs to one bin-to-tile rasterizer, is used to store temporary results between the two rasterizer stages. So that we do not have to compute the edge-equation parameters more than one time, they are directly stored instead of the vertices of the triangle. In addition, also the tile-mask, indicating which tiles are covered by the triangle is added.

This data combined represents one entry in the memory and has a size of $3 \cdot 3 \cdot 32\text{bit} + 64\text{bit} = 352\text{bit}$. Of course, if the tile or bin size would be changed, also the entry size would change. To get better alignment in our case additional 32bit were added, which makes the entry 384bit wide. Since

the port of the used BRAM is not limited, each entry can be addressed at once and only one memory access is needed. There is also the option to remove unnecessary bits (ex. the added 32bit) when implementing the design for a specific FPGA.

On real hardware, the bus-width would be fixed and the entry would have to be split up into multiple memory accesses. Similar to the main memory, also this memory was implemented in BRAM and has to be limited in size, therefore a maximum of 512 triangles per bin was set, which makes one memory instance $384\text{bit} \cdot 512 = 25\text{kB}$ in size.

Due to the fact, that the memory is just a buffer between several modules and no external access is needed, the simple-dual port functionality can be used, which allows us to use one port for writing and the other for reading. Similar to the main memory arbiter (sec. 4.2.1) a local memory arbiter is needed which allows accesses from multiple tile-rasterizer units.

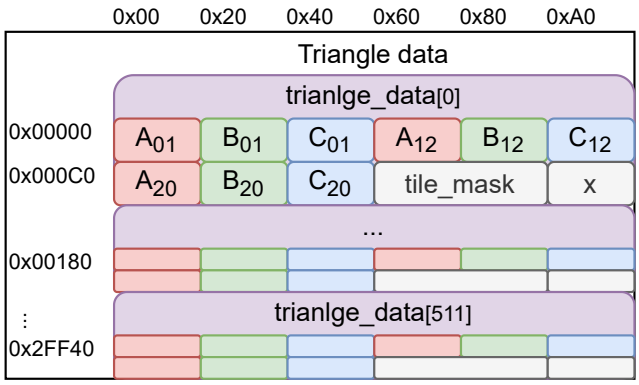


Figure 4.9: Local memory structure including edge-equation parameters of all triangles as well as the previously computed tile-mask.

4.2.5 Tile assignment

Similar to the bin assignment module, the tile assignment module is responsible to schedule the tile rasterizer modules. It will assign one tile after the other to a module that is currently free/ready. Implicitly it is also part of the bin-to-tile rasterizer module which waits for this module to finish. In a

different design, they could operate independently from each other, with the requirement of more than one local memory per bin-to-tile module. This could be achieved by using different memory spaces or different caches on the GPU.

4.2.6 Tile rasterizer

Where the bin-to-tile rasterizer still had to check each corner of a tile to determine the dependency of the tile and the triangle, the tile rasterizer does not require such a difficult approach. Each pixel is only depends on the location of its center-point to the triangle. This of course causes aliasing problems, which have to be handled by the overlying software using anti-aliasing methods. The tile rasterizer also gets all needed edge-equation parameters from the local memory entry. The only thing it has to check before rasterizing is, if the corresponding bit in the tile-mask is set or not. If it is not set, the rasterizer will immediately fetch the next triangle or go to the next tile.

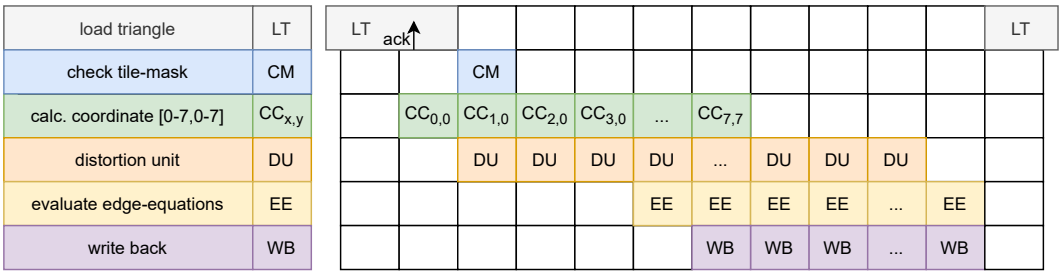


Figure 4.10: Used pipeline in the tile rasterizer using a 3-cycle distortion unit.

Pipelining

A slightly simpler pipeline can be used, which depends on the distortion unit again. To improve performance the coordinates are already calculated and distorted while still loading the triangle. Due to the fact that the loading can take an arbitrary time, we can only "pre-load" one coordinate. In the bin-to-tile rasterizer instead, we are able to fill the pipeline 4 cycles earlier.

4.2.7 Tile memory

As the last step, if the pixel lays inside the triangle, its attributes and other parameters have to be stored somewhere. Since we already have a hierarchical design the amount of memory needed is very small and can be stored locally to the tile. A well-known example would be a local depth-buffer which allows us to do the early-depth test in the tile buffer directly. Based on the depth test, other buffers (triangle-id-buffer, barycentric-coordinate-buffer, etc.) can be updated.

In our implementation, only the triangle-id is written and no depth or barycentric coordinate is calculated. This comes due to the fact, that floating-point arithmetic would be needed (sec. 3.5.1), which would need an enormous amount of logic on the FPGA. Additionally, no module was developed after this stage, since the following part would be the fragment-shader, which normally is programmable and therefore runs on the shader cores of the GPU.

To still get some information about the behavior of the rasterizer, a debug command was implemented which prints the most interesting information. Apart from the bin-, tile- and pixel-coordinates it prints the distorted coordinates (cx, cy) as well as the result of the edge-equation evaluation and triangle id.

Listing 4.1: Example debug output of one generated fragment.

```
Note: TX bx:4 by:0  tx:7 ty:3  px:2 py:2  cx:-2.002031e+02 cy:-4.921250e+02
        e01:6.207425e+04  e12:1.485980e+03  e20:4.959043e+03  in:'1' id:27 ...
Time: 1615 ns  Iteration: 1  Process: ...
```

4.3 Distortion unit

In the bin-to-tile and the tile rasterizer, an arbitrary distortion unit was described. This module takes the pixels (linear) coordinates as input and distorts them in several pipelined stages. The trivial distortion unit takes the input coordinates and outputs them right away. This takes no time and would result in traditional linear rasterization as we already know from

normal GPUs. Apart from the "linear" distortion unit, two other units can be configured to be used in the hardware design:

- Polynomial distortion using 4 coefficients:
  $f(r) = k_0 + k_1 \cdot r^1 + k_2 \cdot r^2 + k_3 \cdot r^3$ (eq. 2.5)
- Even-polynomial distortion using 3 coefficients:
  $f(r) = k_0 + k_2 \cdot r^2 + k_4 \cdot r^4$ (eq. 2.6)

Both models require some parameters, which for now were taken from the previously described Oculus Rift DK2 model (eq. 2.13, 2.14).

4.3.1 Polynomial distortion with square root

The first of those models needs the square root to calculate the distortion function $f(r)$. This can be efficiently achieved using the linear approximation described in section 2.4.2. We have already introduced a pipelined design in the previous chapter. This has also an effect on the distortion unit, which has to be designed to meet the needed timing requirements. As a consequence, the logic was split up into 4 stages, which implies 4 stages of the pipeline. Each stage contains part of the required arithmetic functions and terminates with a set of registers. They were chosen to be synchronous registers, which allows us to shift them to the corresponding output register of the DSP slices (fig. 4.3 - P).

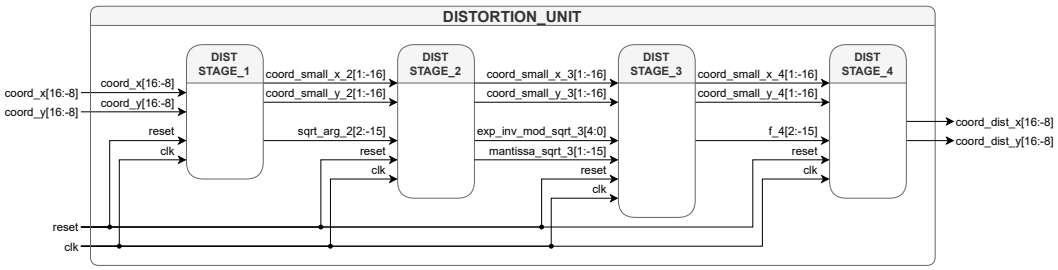
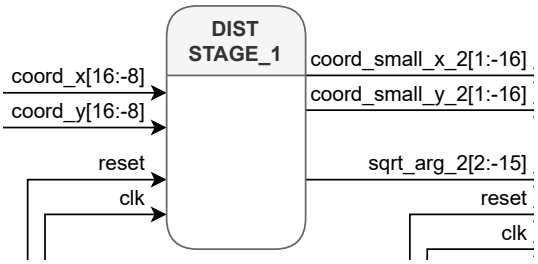


Figure 4.11: Schematic of the 4 stage polynomial distortion unit: $f(r) = 0.795 + 0.103 \cdot r - 0.145 \cdot r^2 + 0.247 \cdot r^3$.

First stage



The first stage directly uses the provided pixel coordinates and converts them into normalized coordinates. Normally a division would be needed to normalize the coordinate according to the screen size but can be prevented by setting the screen size to a power of 2. This works for our resolution of $1024x1024px$, but may not work for other resolutions. As a workaround, the next bigger power of 2 is used instead of the resolution. This implies that, if the screen size is not a power of 2, the distortion parameters are modified accordingly. By allowing this modification a shift operation, which is much smaller in hardware, can be used instead of the division.

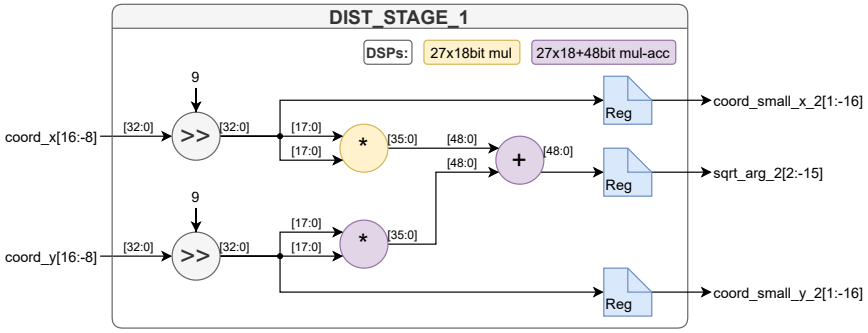


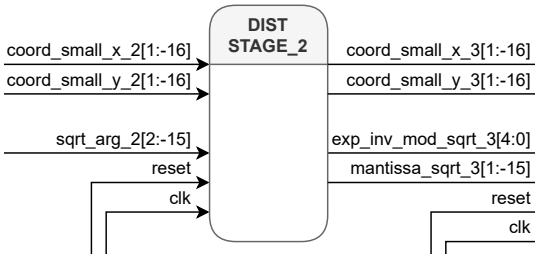
Figure 4.12: Schematic: First stage: Coordinate normalization and calculation of the square roots argument: $r^2 = x_n^2 + y_n^2$

In addition to the normalization also the argument of the square root (r^2) is calculated which is the sum of the square of both coordinates. This can be efficiently done using two DSPs. The first one will just calculate the square of the x-coordinate while the second one will use the multiply-accumulate functionality and adds the result of the first one to the square of the y-coordinate. In both units no cascade has to be used since the bit-length of the multiplication stays below 18bit and a $27x18$bit multiplier is already built-in. Since our coordinates normally consist of 25bit for the chosen resolution, a decreased accuracy would be a consequence of resizing it to 18bit. On the other hand, all coordinates that will ever be input into the

distortion unit will be generated by the overlying hardware, which will never produce coordinates with more than 1 post-comma bit ($\pm 0.5px$).

Second stage

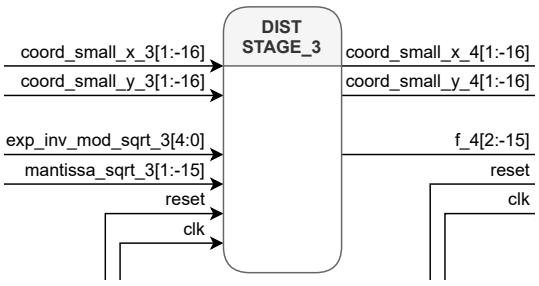
According to equation 2.25, the square root can be approximated by first determining the mantissa and exponent and calculate the square root separately. In hardware, the exponent of a fixed-point number can be determined by the Most Significant Bit (MSB). If we round the exponent to an even number, the mantissa is in the range between 0.5 and 2.0. This allows us to retrieve the square root of the exponent-part with a single right-shift operation which is very cheap in hardware. The square root of the mantissa is then approximated using the piece-wise linear coefficients of table 2.1. Apart from several multiplexers for the coefficients, one DSP is used for the multiplication and addition of the linear function.



$$\sqrt{x} = \sqrt{x_m \cdot 2^{x_{e2}}} = \sqrt{x_m} \cdot \sqrt{2^{x_{e2}}} = \sqrt{x_m} \cdot 2^{\frac{x_{e2}}{2}} = \sqrt{x_m} \cdot 2^{x_{e2} \gg 1}$$

Third stage

The third stage is responsible for calculating the distortion function (eq. 2.13) and consists of 6 multiplications and 3 additions. The radius r has first be reconstructed using the results of the previous stage ($\sqrt{x_m}$, $x_{e2} \gg 1$). By rearranging the order how the calculations are executed we can avoid computing r^2



and r^3. This reduces the complexity to only 3 multiplications and 3 additions. This again can be executed in 3 DSPs configured in multiply-accumulate mode.

$$f(r) = k_0 + k_1 \cdot r^1 + k_2 \cdot r^2 + k_3 \cdot r^3 \tag{4.1}$$

$$f(r) = k_0 + k_1 \cdot r^1 + r^2 \cdot \left(k_2 + r^1 \cdot k_3\right) \tag{4.2}$$

$$f(r) = k_0 + r^1 \cdot \left(k_1 + r^1 \cdot \left(k_2 + r^1 \cdot k_3\right)\right) \tag{4.3}$$

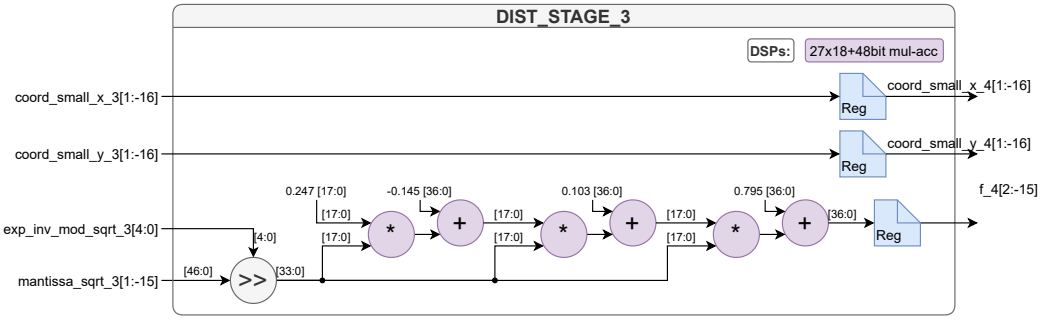
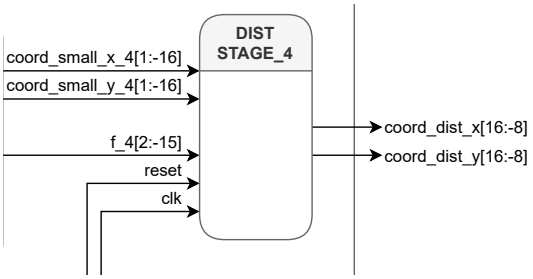


Figure 4.13: Schematic: Third stage: Calculation of the distortion coefficient f_4 calculated according to $f(r) = k_0 + k_1 \cdot r^1 + k_2 \cdot r^2 + k_3 \cdot r^3$

Fourth stage

In the last distortion stage, the distorted coordinate is finally being computed. This of course requires the result of the distortion function $f(r)$ and the non-distorted coordinates. They have to be passed through every stage since we have a pipelined design and the input-coordinate will already have a new coordinate assigned. To be more efficient the normalized coordinate is passed through the modules, which can be multiplied with $f(r)$. The result



can then be de-normalized to the screen size. Although the smaller coordinate has fewer bits, the output of the DSP multiplier unit will always be 48bit and is very accurate. The de-normalization is only a shift operation in our case, which can be performed by shifting the routing on the hardware.

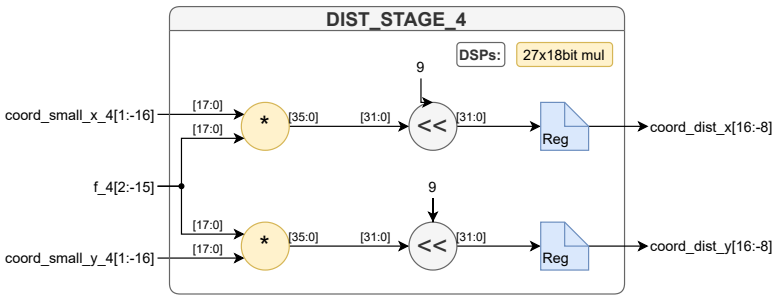


Figure 4.14: Schematic: Fourth stage: Distortion of the original coordinates according to $f(r)$: $\begin{pmatrix} x \\ y \end{pmatrix} \cdot f(r) \ll \log_2(1024/2)$

4.3.2 Polynomial distortion without square root

The second module uses a polynomial function with even-order coefficients only. This allows us to discard the square root calculation and use the squared radius r^2 directly. The resulting design is still very similar to the previously described one, without the need for the second stage:

- Stage 1: (prev. 1)
  - Normalize coordinates
  - Compute $t = x^2$
  - Compute $r^2 = y^2 + t$
- Stage 2: (prev. 3)
  - Compute the distortion function $f(r) = k_0 + k_2 \cdot r^2 + k_4 \cdot r^4$
- Stage 3: (prev. 4)
  - Compute distorted normalized coordinates: $\begin{pmatrix} x \\ y \end{pmatrix} \cdot f(r)$
  - De-normalize coordinates: $\begin{pmatrix} x \\ y \end{pmatrix} \cdot f(r) \ll \log_2(1024/2)$

The first and the last stage left unchanged and the second (previously third) had to be changed to support the new distortion function (eq. 2.14). Since only 3 coefficients are needed for this type of polynomial function the complexity reduces further and can be implemented using only two DSPs in multiply-accumulate mode.

$$f(r) = k_0 + k_2 \cdot r^2 + k_4 \cdot r^4 \tag{4.4}$$

$$f(r) = k_0 + r^2 \cdot \left(k_2 + r^2 \cdot k_3\right) \tag{4.5}$$

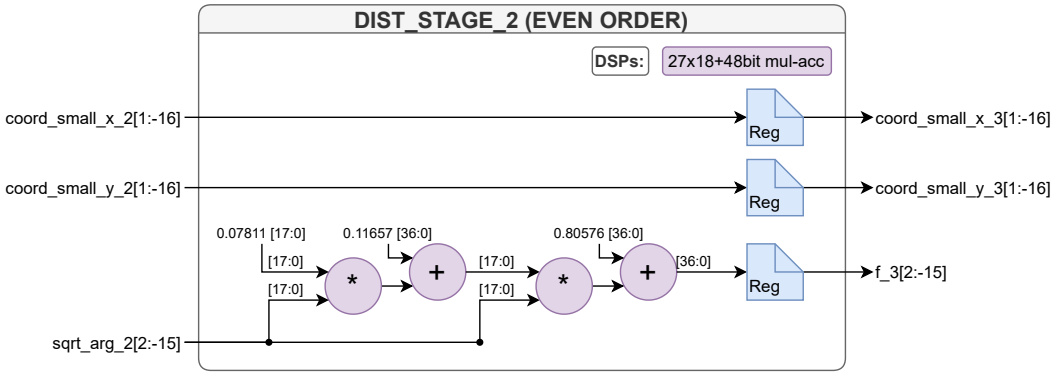


Figure 4.15: Schematic: Second stage: Calculation of the distortion coefficient f_3 calculated according to $f(r) = k_0 + k_2 \cdot r^2 + k_4 \cdot r^4$

4.4 Example

As an example of how the pipeline works a simple scene including 6 cubes was created. The scene can be seen in figure 4.16 and was displayed using a common 3d viewer program. The same file containing the corresponding 72 triangles was also used in the software rasterizer pipeline, which uses the software rasterizer. The resulting output image looks very similar to the previous one (fig. 4.17).

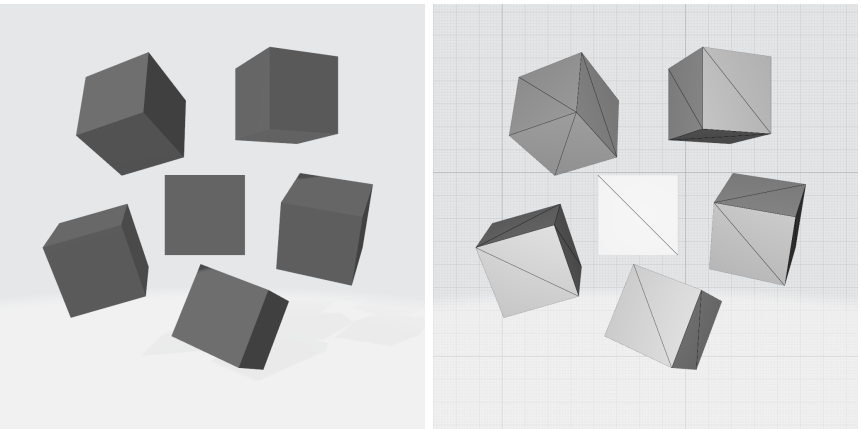


Figure 4.16: Scene used as an example, formed by 6 cubes of the same size which were rotated and moved in the scene.

4.4.1 Software Pipeline

In the same figure 4.17 on the right, the bins with the size of 64 pixels are drawn. In the geometry step, the corresponding bin-queues were formed which contain the intersecting triangles. As an example, the bin-queue of the highlighted bin ($b_x = 4$, $b_y = 4$) contains 3 triangles with the index 10, 32 and 33.

One of such triangles (32) is drawn again in figure 4.18 and covers only a part of the bin. This means, if we evaluate the edge-equation for every corner of the bin, we can see that it is partly inside whether than full inside/outside.

t_{id}	v_{0x}	v_{0y}	v_{1x}	v_{1y}	v_{2x}	v_{2y}
10	−189.59	−176.445	−258.715	−92.2695	−201.090	−251.074
32	−201.09	−251.074	−258.715	−92.2695	−436.641	−143.262
33	−201.09	−251.074	−436.641	−143.262	−371.461	−299.902

Table 4.1: Data of the three triangles which intersect the bin at $b_x = 4$ and $b_y = 4$.

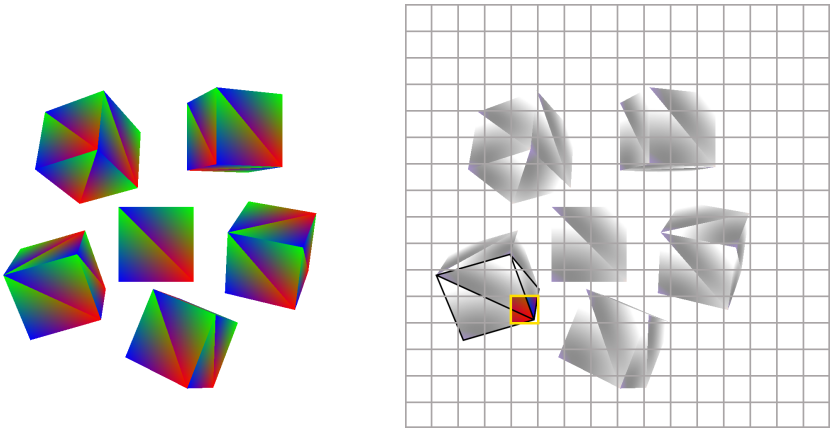


Figure 4.17: Cubes-scene output of the software pipeline (left) and the bin at $b_x = 4$, $b_y = 4$ covered by three triangles of the scene.

To do so, first, the edge-equation parameters have to be calculated according to equation 3.3. Afterwards the results for the four bin-corners (tab. 4.3) get compared with zero. Since on any edge, the result is negative for every corner the bin has to be fully or partly inside the triangle. Additionally, some results are negative, which means that some corners, in our case three, are not inside the triangle. This lets us assume that the triangle is not fully, but partly inside the triangle which matches the graphical representation.

A_{01}	B_{01}	C_{01}
−158.8045	−57.625	−46402.13616
A_{12}	B_{12}	C_{12}
50.9925	−177.926	−3224.61842
A_{20}	B_{20}	C_{20}
107.812	235.551	80820.64685

Table 4.2: Edge equation parameters for triangle 32.

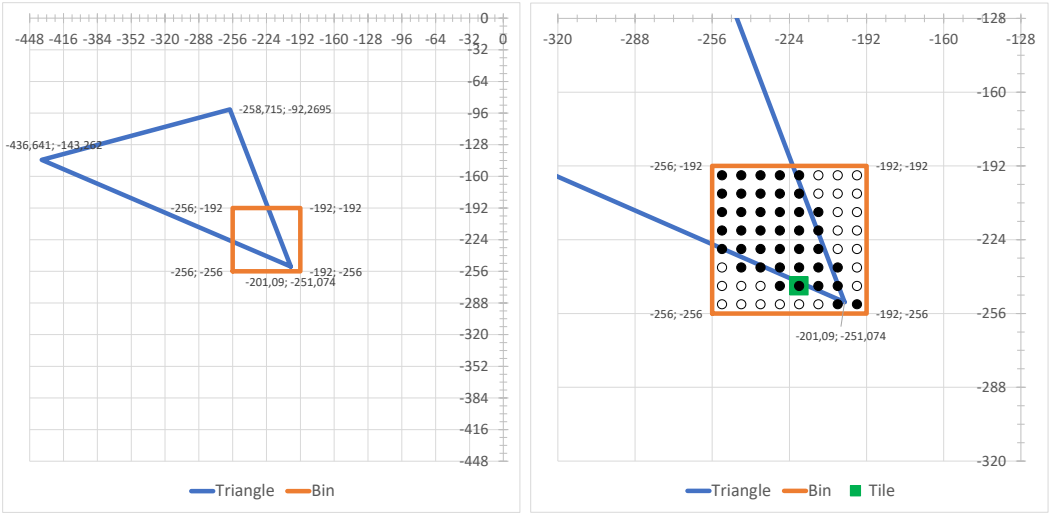


Figure 4.18: In a first step the 4 bin-corners are checked against the triangle which partly covers the bin (left). Afterwards the tile mask can be formed which describes if a tile is covered by the triangle or not (right).

$c_{x,y}$	e_{01}	e_{01}	e_{01}
$c_{0,0}$	9003.8158	29270.3576	−7080.2811
$c_{8,0}$	−1159.6722	32533.8776	−180.3131
$c_{0,8}$	5315.8158	17883.0936	7994.9829
$c_{8,8}$	−4847.6722	21146.6136	14894.9509

Table 4.3: Edge equation evaluation for the 4 bin corners and triangle 32.

Finally the same can be done for all 81 tile-corners and we would get the following tile-mask: $0x1F1F3F3F3F7E78C0$. A better representation can also be seen in the right image of figure 4.18, where a 1 in the bit-mask is represented as a filled circle. In the next step, we iterate over every covered tile and rasterize each pixel inside it. This requires fewer logic components, since the evaluation of a pixel does not rely on its corners, but on the center-point only. Regarding our example, the tile at $t_x = 4$ and $t_y = 1$ was chosen, which is marked in green (fig. 4.18). The per-pixel evaluation can be seen in figure 4.19

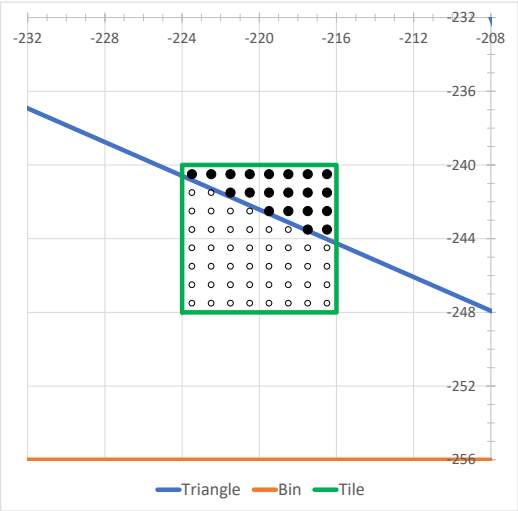
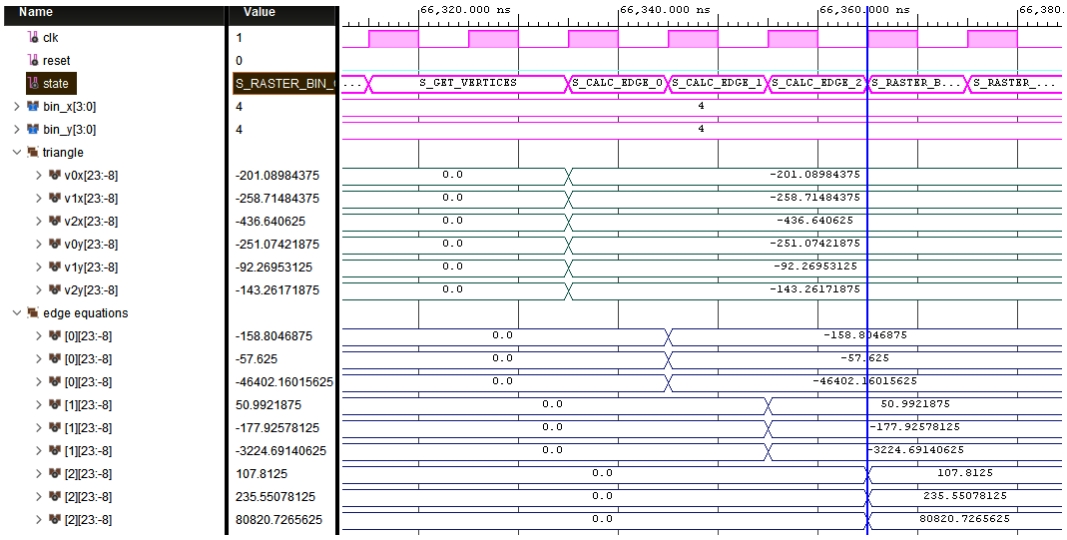


Figure 4.19: Tile rasterizer output of tile $t_x = 4$, $t_y = 1$ of bin $b_x = 4$, $b_y = 4$. The filled circles represent generated fragments.

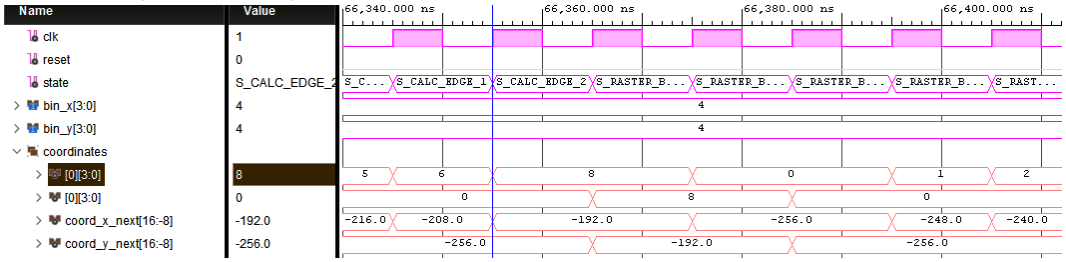
4.4.2 Hardware rasterizer

The previous results were created using the software pipeline described in [5]. Instead of using the bin-queues in the software rasterizer, they were stored in a so-called memory file together with the triangle vertices, bin lengths, and offsets. This memory represents the transfer medium between stream-multiprocessors and rasterization hardware. Using the build in VHDL simulator this file is loaded into the BRAM of the simulated FPGA and the rasterizer immediately begins to rasterizer one bin after the other.

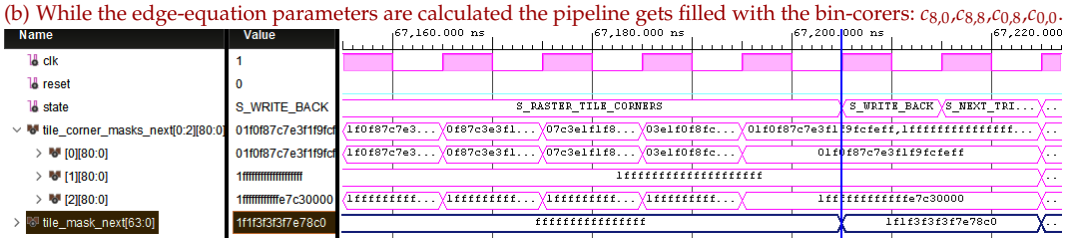
At some point, the previously discussed bin ($b_x = 4$, $b_y = 4$) gets rasterized with the corresponding 3 triangles. This happens at the simulation time of 65.325µs. After approximately 1µs the first triangles ($t_{id} = 10$) tile-mask was already written to the local memory and the second triangle ($t_{id} = 32$) gets handled. This corresponds to the triangle used in previous calculations. In figure 4.20a a snapshot of the loaded triangle can be observed. After loading the vertices of the triangle (`S_GET_VERTICES`) the three edge equations are calculated (`S_CALC_EDGE_x`). The resulting edge-equation parameters can be seen in the lower 9 wave-lines and correspond to the parameters in table 4.2 with very slight differences resulting from the fixed-point inaccuracies.



(a) In the first step (`S_GET_VERTICES`) the triangle data is loaded. Afterwards the edge-equation parameters are calculated (`S_CALC_EDGE_x`).



(b) While the edge-equation parameters are calculated the pipeline gets filled with the bin-corers: $c_{8,0}$,$c_{8,8}$,$c_{0,8}$,$c_{0,0}$.



(c) The corresponding tile mask is calculated after all corners were evaluated and is written to the local memory (`S_WRITE_BACK`).

Figure 4.20: Simulation waveforms of the hardware bin-to-tile-rasterizer.

While calculating, the pipeline is already filled with the coordinates of the 4 bin corners $c_{8,0}$, $c_{8,8}$, $c_{0,8}$, $c_{0,0}$ (fig. 4.20b). Since the bin is not fully inside nor outside the triangle, all 81 corners have to be evaluated. Three so-called corner masks are generated with 81bit each. These masks are then used to calculate the corresponding tile-mask (fig. 4.20c). The mask and the edge-equation parameters are stored in the local memory, which is then used by the tile rasterizer units to evaluate each individual pixel.

The generated simulation output contains a line, according to the format in listing 4.1, for each generated fragment. Additionally also the fragments failing the last evaluation in the tile rasterizer were printed for debug purposes. If we have a look back at the 5th pixel-line of figure 4.19, the right two pixels are inside the triangle while all others are outside. The corresponding debug output is shown in listing 4.2.

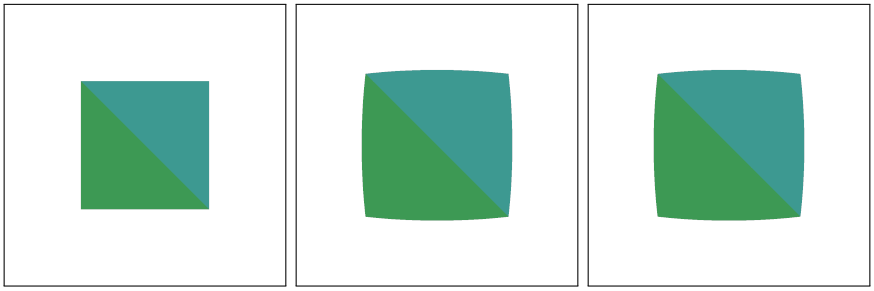
Listing 4.2: Debug output of the tile rasterizer (5th pixel line).

```
...px:0 cx:-2.235e+2 cy:-2.435e+2 e01:3.1e+3 e12:2.8e+4 e20:-6.3e+2  in:0 id:32
...px:1 cx:-2.225e+2 cy:-2.435e+2 e01:2.9e+3 e12:2.8e+4 e20:-5.2e+2  in:0 id:32
...px:2 cx:-2.215e+2 cy:-2.435e+2 e01:2.8e+3 e12:2.8e+4 e20:-4.1e+2  in:0 id:32
...px:3 cx:-2.205e+2 cy:-2.435e+2 e01:2.6e+3 e12:2.8e+4 e20:-3.0e+2  in:0 id:32
...px:4 cx:-2.195e+2 cy:-2.435e+2 e01:2.4e+3 e12:2.8e+4 e20:-2.0e+2  in:0 id:32
...px:5 cx:-2.185e+2 cy:-2.435e+2 e01:2.3e+3 e12:2.8e+4 e20:-9.2e+1  in:0 id:32
...px:6 cx:-2.175e+2 cy:-2.435e+2 e01:2.1e+3 e12:2.9e+4 e20:+1.4e+1  in:1 id:32
...px:7 cx:-2.165e+2 cy:-2.435e+2 e01:2.0e+3 e12:2.9e+4 e20:+1.2e+2  in:1 id:32
```

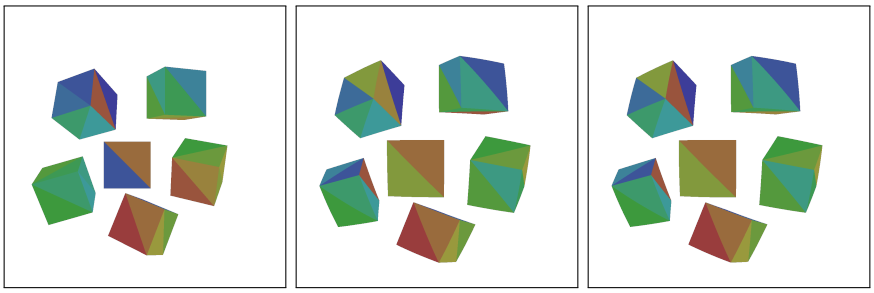
Finally, a small program was implemented, which reads each fragment out of the debug output and draws the corresponding pixel on a frame-buffer (fig. 4.21b). This allows us to compare the software and the hardware rasterizer. All in all, it took 444µs to generate a total of 290854 fragments. This is about 27% of the total pixel count $1024 \cdot 1024$. The same scene was used in combination with the two implemented distortion functions 2.13 and 2.14. Since the cubes are near the center, they get bigger, and also more fragments have to be produced. Therefore the non-linear rasterizer takes a little bit longer to get through all bins. The same is true for two other scenes with a different level of complexity (Cubes: 12 triangles, Suzanne: 968 triangles).

Scene	Cube		Cubes		Suzanne	
#triangles	2/12		34/84		650/968	
	time	$\#px$	time	$\#px$	time	$\#px$
$f(1)$	217µs	$244530px$	444µs	$290854px$	1110µs	$154602px$
$f(r^2, r^4)$	297µs	$325218px$	524µs	$350738px$	1238µs	$192841px$
$f(r^1, r^2, r^3)$	303µs	$325432px$	531µs	$351412px$	1249µs	$193078px$

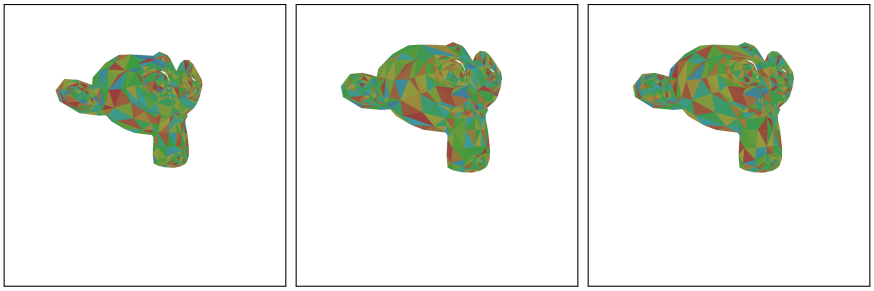
Table 4.4: Runtime and fragment-count of different scenes sorted by increasing complexity.



(a) Cube scene.



(b) Cubes scene



(c) Suzanne scene

Figure 4.21: Hardware rasterizer output of the three distortion options: linear/no-distortion (left), polynomial distortion (center), even order polynomial distortion (right).

5 Evaluation

The discussed hardware concept was designed with many configurable parameters, with some of them presented in listing 5.1. Depending on these parameters many others are derived and influence things like register-sizes for the coordinates and arithmetic functions. Also, the synthesis tool can use the restricted data types to further optimize the design and throw away unnecessary logic. In the next step, some performance measures are presented which depend on these parameters too. In the end, they can be used to fit the rasterizer unit(s) to individual requirements such as memory throughput, performance, area, etc. Some of them were evaluated separately in the following sections.

Listing 5.1: Hardware configuration parameters.

```
-- screen, bin and tile size in pixels
G_MAX_RES_X             := 1024;                -- used resolution
G_MAX_RES_Y             := 1024;
G_BIN_SIZE_X            := 64;                  -- bin size
G_BIN_SIZE_Y            := 64;
G_TILE_SIZE_X           := 8;                   -- tile size
G_TILE_SIZE_Y           := 8;                   --   max 64 tiles per bin

-- distortion configuration
G_DIST_SELECTED         := G_DIST_EVEN;         -- used distortion
G_MAIN_MEM_FILE         := "suzanne_even.mem"; -- used memory file

-- memory configuration
G_MAX_TRIANGLES         := 1024;                -- max #triangles
G_MAX_INDICES_PER_BIN   := 512;                 -- max #triangles per bin
G_MAX_INDICES           := 2048;                -- max indexed triangles

-- hardware rasterizer configuration
G_NUM_BIN_RASTER        := 2;                   -- #bin-to-tile rasterizer
G_NUM_TILE_RASTER       := 4;                   -- #tile rasterizer
```

5.1 Memory throughput

One of the major considerations regarding modern graphics cards is the possible memory bandwidth and the ability to use as much as possible. In our design, all memory instances (sec. 4.2.1, 4.2.4) are standalone components and their interfaces were tuned to fit the stored data. This results in the main memory using a 256bit interface and multiple local memories with 384bit interfaces. With the used FPGA system-clock of 100MHz, the bandwidth is limited to 3.2GB/s and 4.8GB/s for every instance.

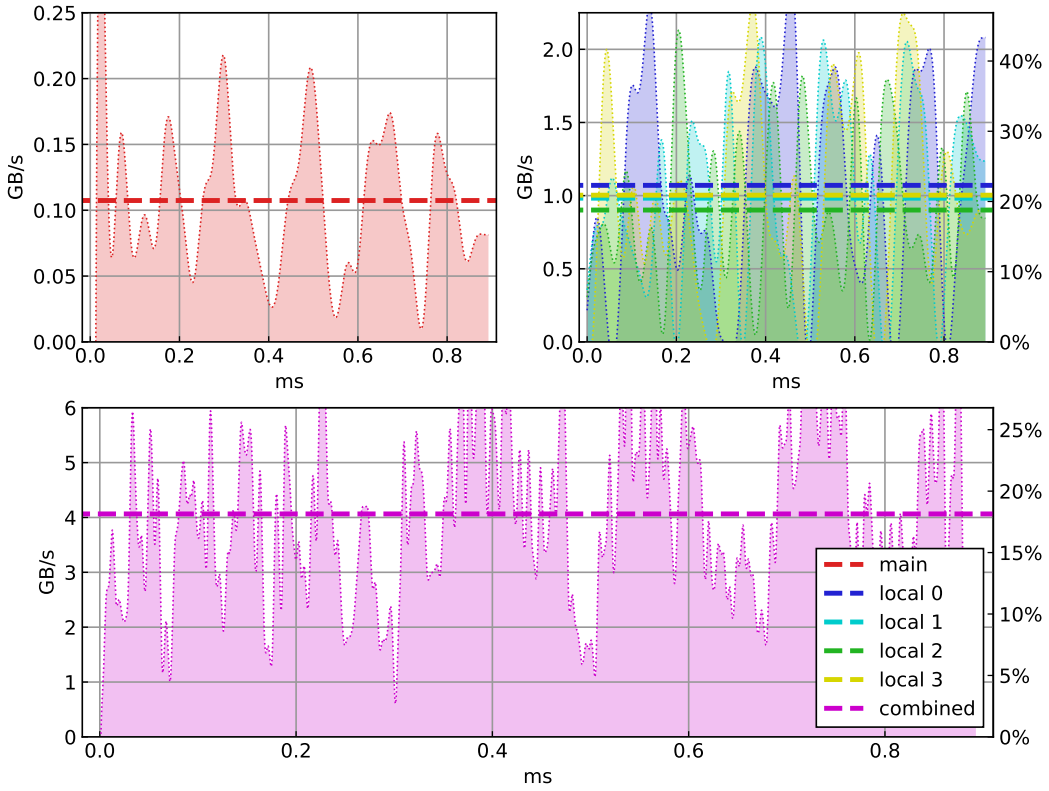


Figure 5.1: The combined memory throughput with four bin-to-tile and four tile rasterizer in GB/s (bottom). The main memory usage is shown on the top-left and the individual throughput of all 4 local memories on the top-right.

To get a general overview of how the memory behaves the suzanne-monkey scene (fig. 4.21c) was rasterized using different rasterizer configurations. The

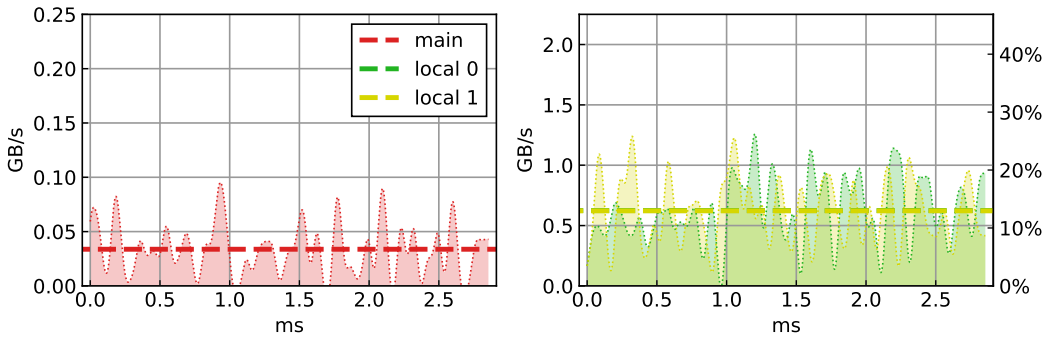
number of bin-to-tile rasterizers, as well as the number of tile rasterizers, was modified and the impact was observed. In table 5.1 the measured memory bandwidth can be seen. Looking at these numbers we can clearly see that the main memory usage is relatively low compared to the local memory bandwidth. This comes due to the fact that the bin-to-tile rasterizers, which are the only ones using the main memory, have to wait for the tile rasterizers to finish before they can proceed with the next bin. Therefore the memory usage is higher if many triangles do not intersects any tiles, which of course should not happen that often, since then the triangle should not be in the bin-queues.

#bin-to-tile	#tile	combined	main	local 0	local 1	
2	2	1.280GB/s	0.034GB/s	0.622GB/s	0.624GB/s	
2	4	2.056GB/s	0.055GB/s	1.048GB/s	0.953GB/s	
4	2	2.542GB/s	0.067GB/s	0.624GB/s	0.577GB/s	+2
4	4	4.064GB/s	0.107GB/s	1.003GB/s	0.901GB/s	+2
8	8	9.903GB/s	0.284GB/s	1.376GB/s	1.144GB/s	+6

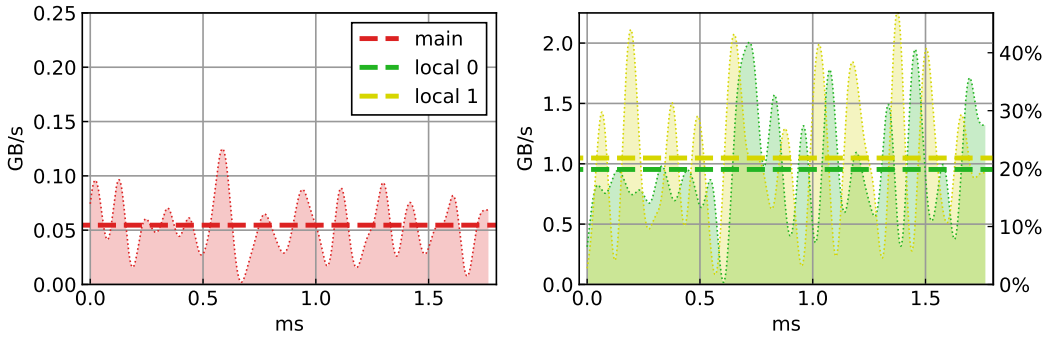
Table 5.1: Memory throughput of main and local BRAM units with different hardware configurations.

The local memory is heavily used instead and is mostly influenced by the number of tile rasterizers accessing the edge-equation parameters and tile masks. By doubling the number of tile rasterizers also the memory usage nearly doubles from ≈ 0.6GB/s to ≈ 1.0GB/s. It is a bit less than two times the needed bandwidth since also the bin-to-tile rasterizer has to write to the memory. By doubling the number of bin-to-tile rasterizers no change can be observed in the local memory, apart from the increased number of such memory blocks.

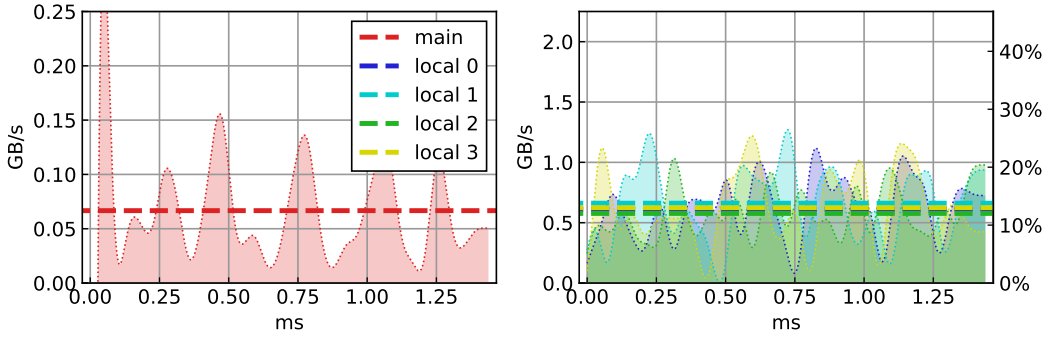
The previously measured values represent an average bandwidth requirement, with peaks reaching more than two times that value (fig. 5.2). Due to the fact that the scene and therefore also the resulting fragments were equal on every run, the amount of read/written memory has also to be the same. Therefore, if the runtime decreases the overall memory bandwidth increases by the same factor. Doubling the number of bin-to-tile rasterizers for example halves the runtime from 2.85ms to 1.44ms while doubling the overall bandwidth from 1.280GB/s to 2.542GB/s.



(a) Two bin-to-tile rasterizer with 2 tile rasterizer.



(b) Two bin-to-tile rasterizer with 4 tile rasterizer.



(c) Four bin-to-tile rasterizer with 2 tile rasterizer.

Figure 5.2: Memory throughput of main memory (left) and local memories (right) in GB/s.

Comparing the used memory structures with the existing ones on actual GPUs, the local memory could most probably be implemented in one of the already existing caches. The L2 cache would perfectly fit the required needs since its size is adequate high as well as the bandwidth. Another possibility would still be a small dedicated memory which is only accessible within the rasterizer. In the next step, the L1 cache could then be used as tile memory (sec. 4.2.7) to provide the fragment data to the fragment shader(s) running on the multiprocessors.

Looking at the maximum measured bandwidth of $\approx$ 10GB/s it seems relatively low due to the fact that some Graphics Double Data Rate RAM 6 (GDDR6) already achieves data-rates up to 20Gbit/s per pin. But such modules are also working at much higher frequencies while our design has a relatively low frequency of 100MHz. Assuming an increase of 20 in frequency ($\approx$ 2000MHz), the required bandwidth would also increase by the factor of 20 (200GB/s). Such data-rates can either be achieved by combining multiple pins of multiple GDDR6 modules, using High Bandwidth Memory (HBM), or by simply using a fast on-chip cache.

5.2 Performance

Apart from memory usage also the performance is a very important measure regarding GPUs. Normally the performance of a graphics pipeline is measured in Frames per Second (fps). This does also imply all stages from vertex to fragment shading and makes it hard to measure on our graphics pipeline. Instead we are able to measuring the software pipeline's (sec. 3.5) performance. For the Suzanne-monkey scene (fig. 4.21c) the performance numbers in table 5.2 were determined.

In the developed pipeline the first two stages were re-used and the last three will be implemented directly in hardware. Therefore a runtime of 0.6ms represents a base-line for the linear rasterizer unit. In this context, it is important to note, that the software pipeline runs on a mobile graphics card with a base frequency of 1GHz and the hardware implementation does only run on 100MHz.

	Quadro M2000M		RTX 2080Ti	
pipeline stage	linear	distorted	linear	distorted
init	0.010539ms	0.010507ms	0.003708ms	0.003827ms
geometry	0.018610ms	0.022755ms	0.008374ms	0.010431ms
schedule	0.108057ms	0.126666ms	0.017895ms	0.018251ms
bin-to-tile raster	0.081463ms	0.156639ms	0.046902ms	0.085999ms
tile raster	0.414587ms	0.569379ms	0.063161ms	0.069936ms
rasterization	0.633256ms	0.885947ms	0.140041ms	0.188443ms
	1579fps	1129fps	7141fps	5307fps

Table 5.2: Software pipeline performance on Nvidia Quadro M2000M (@1GHz) and RTX 2080Ti (@1.5GHz) rendering the suzanne-monkey scene.

Tile rasterizer units

Being able to increase the number of rasterizer units individually, the performance can not be expressed as a single number but has to be evaluated in more detail. First of all the number of tile rasterizers that work on the same bin in parallel was step-wise increased. These configurations were tested with the already known scenes (fig. 4.21), which represent a different level of complexity. The simple cube scene has only two triangles visible, which results in the fastest runtime of 3.98ms.

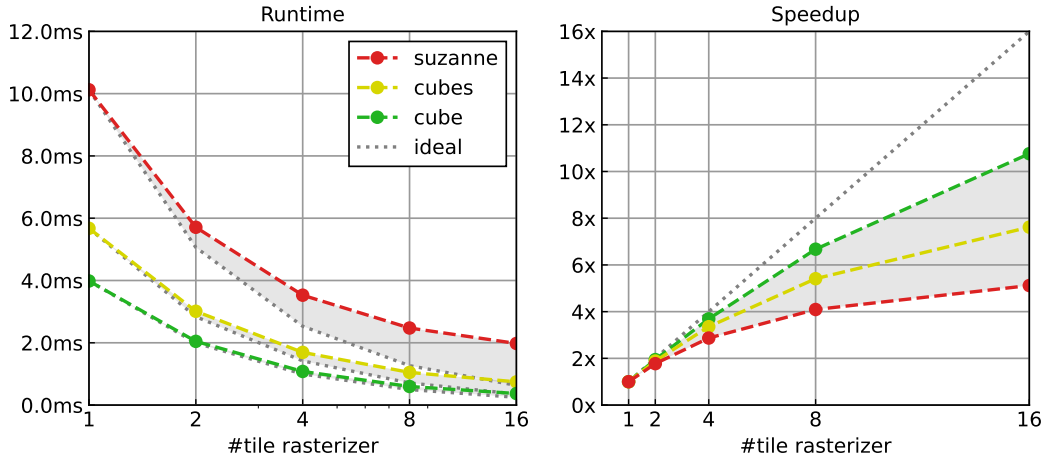


Figure 5.3: Runtime and speedup with varying number of tile rasterizer units.

With a moderate number of tile rasterizer units, the speedup is nearly ideal and scales proportionally to that number. Nevertheless, when using more than 4 units (speedup $3.69x$) the gained speedup becomes worse. This is mainly due to the fact that also the bin-to-tile rasterizer has to rasterize each triangle which creates a non-parallelizable part. If the tile rasterization becomes faster, this part becomes more and more dominant and decreases the possible speedup. This part becomes even bigger if more complex scenes are used and the speedup comes down to $2.86x$ using 4 tile rasterizer units. With the previous observation, the number of tile rasterizers was fixed to 4, since a speedup between $4.09x$ and $6.67x$ using 8 units would make the hardware unnecessarily big.

Bin-to-tile rasterizer units

Changing the number of bins that are handled in parallel can also be used to speed up the rasterization. In contrast to the tile rasterization, each bin is completely independent from the others, which makes them nearly 100% parallelizable. This was also verified with the three scenes, with a speedup between $7.37x$ and $7.88x$ using 8 parallel bin-to-tile rasterizer units.

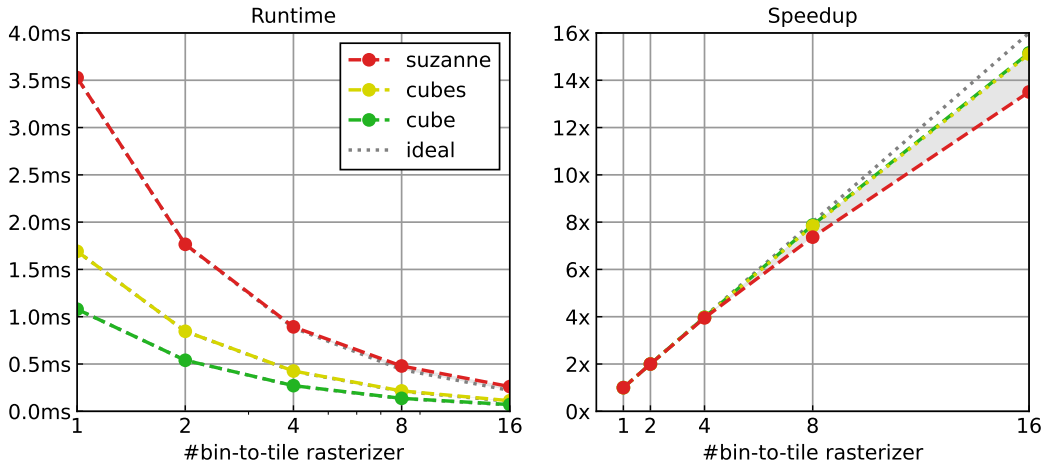


Figure 5.4: Runtime and speedup with varying number of bin-to-tile rasterizer units.

Considering the gained knowledge the bin-to-tile rasterizer unit could efficiently be integrated into modern GPU designs. As a base-line the newest Nvidia Ampere Architecture [36] was used. It consists of multiple Graphics Processing Clusters (GPCs), which can access the L2 cache as well as the main memory. Each GPC is formed out of a maximum of 12 Streaming Multiprocessors (SMs) containing $4x32$ processing cores, 4 tensor cores, and one ray-tracing core. Each GPC does also contain a so-called "Raster Engine", which can interact with all SMs. The amount of GPCs and SMs differs between the available chips and therefore also the rasterization performance. As we have already seen, the amount of bin-to-tile rasterizer units can be freely changed with a nearly $1:1$ speedup. This would allow high-end cards, such as the RTX 3090 (7 GPCs), to rasterize up to seven bins at a time, while lower-end cards (RTX 3070, 4 GPCs) can only rasterize 4 bins.

5.2.1 Comparison with software

Compared to the software pipeline running on a Quadro M2000M, which needs approximately 0.6ms executing the rasterization on 640 processing cores with up to 1098MHz, the hardware implementation can be tuned to match and surpass that performance significantly. With the lowest $1x1$ configuration, one bin-to-tile rasterizer, and one tile rasterizer unit, the suzanne-monkey scene still needs 10.1ms. Bringing the software pipeline to the same clock of 100MHz (0.6ms $\rightarrow$ 6ms) would be 4ms slower. But already with a $1x2$ (5.7ms) or $1x4$ (3.5ms) configuration, the hardware pipeline is faster.

Considering the previously suggested hardware configuration, each GPC of the used GPU can handle one bin at a time. Since an older, low-end card was used for testing, which only has one GPC, the single bin-to-tile rasterizer could be fitted nicely. Nowadays GPUs, such as the RTX 2080Ti, already have 6 GPCs running at higher clocks which would allow a $6x4$ configuration and a runtime well below 1ms.

Figure 5.5: Nvidia architecture design using multiple SMs per GPC. Each GPC additionally has its own raster engine, responsible for rasterization. [37]

5.2.2 Linear vs. non-linear design

Comparing the rasterizer with distortion and the rasterizer without distortion is not that easy either. In the software case, the reduced instruction count/complexity results in better performance (tab. 5.2). Using an advanced pipeline design in our hardware, this complexity can be hidden and the performance will stay the same. Nevertheless, due to the distortion, many objects change their size and more pixels/tiles have to be checked. This does indeed affect the performance but is hard to measure. The suzanne-monkey scene, for example, can be rasterized in 1.11ms instead of 1.25ms, but the pixel count increased from 154620px to 193078px.

Concerning the produced pixels per µs using the numbers from table 4.4, this similarity in performance becomes much clearer (tab. 5.3). In addition also the impact of lots of small triangles (suzanne) in comparison to few, big triangles (cube) can be seen, which significantly reduces the achievable throughput.

Scene	Cube	Cubes	Suzanne
#triangles	2/12	34/84	650/968
	$\#px/\mu s$	$\#px/\mu s$	$\#px/\mu s$
$f(1)$	1126.87	655.08	139.28
$f(r^2, r^4)$	1095.01	669.35	155.77
$f(r^1, r^2, r^3)$	1074.03	661.79	154.59

Table 5.3: Produced pixels per µs for different scenes and distortion algorithms.

5.3 Area

One thing that is very important concerning hardware design, is the needed area on a chip. There are several methods on how to measure such an area, with the trivial one in square millimeters. Since often the technology is not fixed and the "real" area depends a lot on that, Gate Equivalents (GE) as a manufacturing-technology-independent measure is often used. In modern Complementary Metal Oxide Semiconductor (CMOS) technology, a two-input NAND-gate with driving strength 1 equals one GE. All other gates (OR, AND, NOT, etc.) can then be mapped to a multiple of that measure.

On an FPGA the utilization of the different hardware components (BRAM, LUT, DSP, etc.) is most often used to compare different designs. This is also used to characterize our design because anyway, no reference area consumption of traditional rasterizer units is publicly available. The design was synthesized using different configurations and the output was compared.

The amount of BRAM used for the main memory is independent of the used distortion method and consists of 16 modules with 36kB. The local memory requires 10 of such modules and one smaller 18kB module. Each tile rasterizer also needs a small 18kB memory to store the triangle indices

for each pixel. This of course is not fixed and could be adapted to also store other attributes (depth, barycentric coordinates, etc.). All other hardware parts, such as registers, LUTs and DSPs, depend on the used distortion method and hardware configuration and are discussed in the following sections.

configuration		$f(1)$			$f(r^2, r^4)$		
#bin-to-tile	#tile	LUTs	REGs	DSPs	LUTs	REGs	DSPs
1	1	1408	982	45	1462	1127	57
1	2	1514	1029	63	1608	1274	81
1	4	1734	1123	99	1902	1567	129
1	8	1952	1311	171	2135	2536	225
1	4	1734	1123	99	1902	1567	129
2	4	3024	2226	198	3414	3500	258
4	4	5992	4432	396	6871	6980	516
8	4	11802	8844	792	13580	13860	1032
avg. / bin-to-tile		1028	915	27	1141	1115	33
avg. / tile		95	47	18	125	171	24

Table 5.4: Hardware requirements with different numbers of bin-to-tile and tile rasterizer units.

5.3.1 Tile rasterizer units

As expected the area consumption scales proportionally to the number of tile rasterizers. Especially for the amount of DSPs, we can clearly see, that with each new tile rasterizer unit 18/24/26 additional DSPs have to be used to be able to calculate the arithmetic functions. Also, the register count increases significantly due to the added pipeline stages which have to be terminated with a register. The amount of LUTs on the other hand does only grow using the polynomial distortion function with odd and even coefficients. By only using the even-coefficient polynomial most of the logic used to calculate the square root can be left out and the distortion can be implemented using DSPs and registers only.

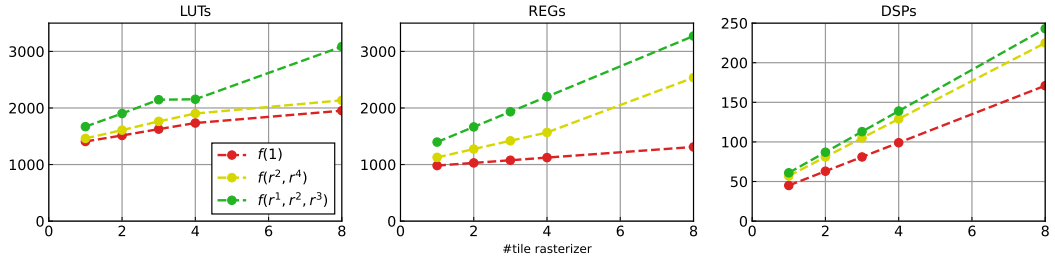


Figure 5.6: Hardware requirements depending on the amount of tile rasterizer units per bin-to-tile rasterizer.

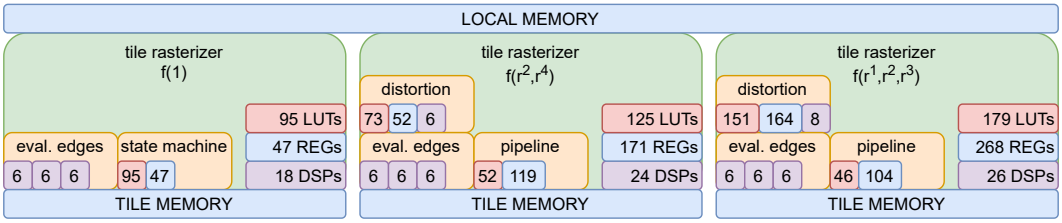


Figure 5.7: Summary of hardware requirements of the tile rasterizer units using no distortion (left), even-order polynomial distortion (center) and a full polynomial distortion (right).

5.3.2 Bin-to-tile rasterizer units

Also, the bin-to-tile rasterizer has to distort tile-corner coordinates and evaluate them using the edge equations in addition to the parameter calculation. This involves a bigger amount of DSPs, LUTs, and registers. Depending on the number of tile rasterizers this amount becomes more or less significant. Nevertheless by adding bin-to-tile rasterizers also at least one tile rasterizer has to be added and therefore also the slope of needed hardware becomes much steeper (fig. 5.8).

5.3.3 Comparison

Comparing the three designs against each other lets us assume that the required modifications to the hardware rasterizer are in an acceptable range. Considering a configuration with four efficiently working tile rasterizers

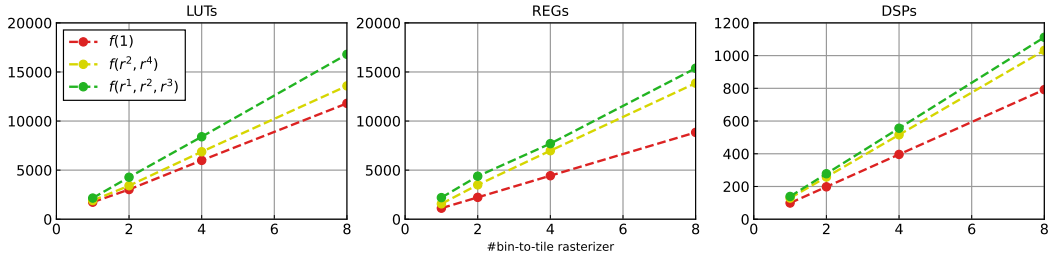


Figure 5.8: Hardware requirements depending on the amount of bin-to-tile rasterizer units.

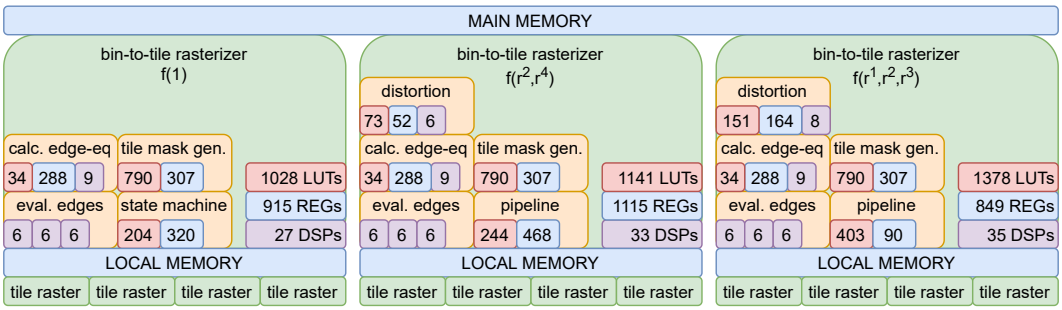


Figure 5.9: Summary of hardware requirements of the bin-to-tile rasterizer units using no distortion (left), even-order polynomial distortion (center) and a full polynomial distortion (right).

per bin-to tile rasterizer results in an increase of 30% in complexity. This can be said because approximately 30% more DSPs have to be used. While the increase of used LUTs is only 16% and can be left apart since in a normal design the DSPs would be implemented as logic too and would surpass the logic implemented in the additional LUTs. The register count on the other hand increased by 63%, due to the added pipeline stages.

These numbers were calculated using the even-order polynomial function. Using the full polynomial, additional logic has to be added to calculate the square root and handle the additional pipeline stage. While the register count only increased slightly (63% $\rightarrow$ 74%), since the pipeline increased only by one stage, the amount of LUTs grows drastically (16% $\rightarrow$ 49%). Also, 40% more DSPs have to be used compared to the non-distorted rasterizer. While the even-order distortion would increase the size of the hardware by approximately one-third the other would increase hardware costs by

at least 50% while being more complex and inaccurate due to the needed approximations.

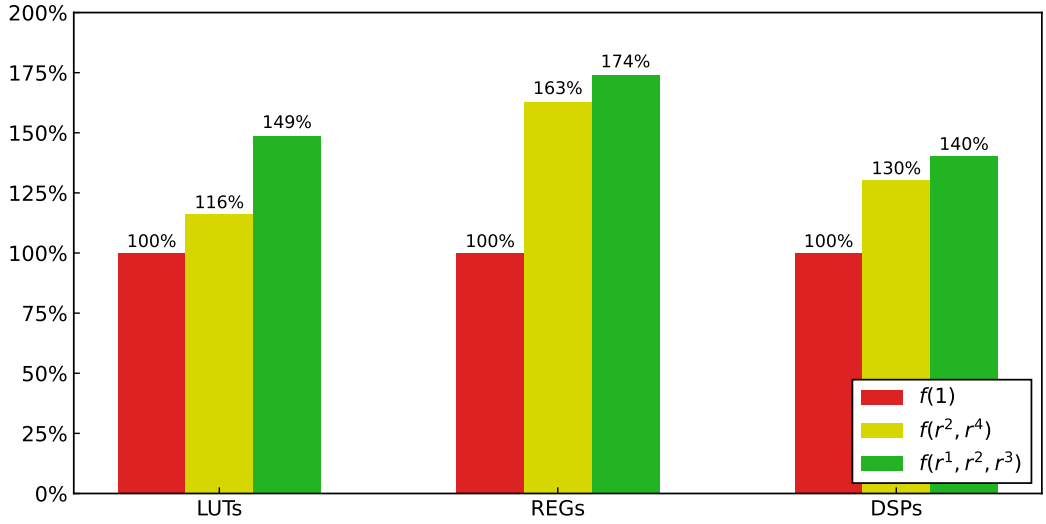


Figure 5.10: Increased hardware requirement in comparison to the traditional, linear rasterizer. The optimized, non-linear design (even-order polynomial) does use approximately 30% more hardware with 16% more LUTs, 63% more registers and 30% more DSPs.

6 Conclusion

In conclusion, the implemented non-linear hardware rasterizer was a great success. Using the highly flexible hierarchical structure, we were able to combine the benefits of the software pipeline implemented at the ICG [5] with the advantages of a dedicated hardware acceleration unit.

The non-linear algorithm allows us to remove an additional image-distortion step at the end of the graphics pipeline while being able to render at the native screen resolution. This increases performance while keeping the latency low, which both are very important aspects concerning VR applications. Compared to a traditional, linear rasterizer the needed chip-area increases by only 30%, which can be achieved by using a well-fitting, pipelined distortion function combined with heavy optimizations.

The implemented hierarchical structure would also fit into modern GPU designs and can be adapted to handle an arbitrary amount of display-tiles in parallel. The memory interface was developed in such a way, that it makes use of very fast local memories which are completely independent of each other. This, in combination with the tile-based rasterization, allows an additional performance boost in pixel independent operations such as early depth-testing.

Appendix

Bibliography

[1] D. Rohmer and J. Tierny. (2018). Computer graphics & scientific visualization., [Online]. Available: https://imagecomputing.net/damien.rohmer/teaching/2018_2019/semester_1/m2_mpri_cg_viz/class/01_surface_representation/content/016_projection_rasterization/index.html (visited on 07/28/2021) (cit. on p. 1).

[2] A. N. Torrentó, "Implementation of a gpu rasterization stage on a fpga," 2015 (cit. on pp. 1, 25, 27, 32).

[3] D. Kanter. (2016). Tile-based rasterization in nvidia gpus with david kanter of real world tech., [Online]. Available: https://www.youtube.com/watch?v=Nc6R1hwXhL8 (visited on 07/28/2021) (cit. on pp. 1, 26).

[4] Wikipedia. (2016). Tiled rendering., [Online]. Available: https://en.wikipedia.org/wiki/Tiled_rendering (visited on 07/28/2021) (cit. on pp. 1, 26).

[5] M. Prettner, "Nonlinear rasterization in a softwaregraphics pipeline," Institute for Computer Graphics and Vision, TU Graz, 2020 (cit. on pp. 2, 10, 25, 26, 31, 32, 55, 73).

[6] MIT-OCW. (Jan. 2013). Mit open courseware., [Online]. Available: http://dspace.mit.edu/bitstream/handle/1721.1/41881/4-492Fall-2004/NR/rdonlyres/Architecture/4-492Fall-2004/F11ED803-DC9A-4267-85A9-AB56368AFB07/0/4_492f04_class04.pdf (cit. on p. 3).

[7] A. Dey, "Perceptual characteristics of visualizations for occluded objects in handheld augmented reality," PhD thesis, Aug. 2013 (cit. on p. 3).

[8] S. Willems. (Jun. 2018). Vulkan api multiview and barrel distortion (vr)., [Online]. Available: https://www.youtube.com/watch?v=waKCmE_McLo (visited on 07/28/2021) (cit. on p. 4).

[9] Nicoguaro. (Oct. 2011). Lens distorsion., [Online]. Available: https://commons.wikimedia.org/w/index.php?curid=17027248 (visited on 07/28/2021) (cit. on p. 5).

[10] K. Lelowicz, "Camera model for lens with strong distortion in automotive application," in *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR)*, 2019, pp. 314–319. DOI: 10.1109/MMAR.2019.8864659 (cit. on p. 4).

[11] imatest. (2021). Distortion models., [Online]. Available: https://www.imatest.com/support/docs/pre-5-2/geometric-calibration-deprecated/distortion-models/ (visited on 07/28/2021) (cit. on p. 6).

[12] A. Fitzgibbon, "Simultaneous linear estimation of multiple view geometry and lens distortion," vol. 1, Feb. 2001, pp. I–125, ISBN: 0-7695-1272-0. DOI: 10.1109/CVPR.2001.990465 (cit. on p. 6).

[13] MathWorks. (2021). What is camera calibration? [Online]. Available: https://de.mathworks.com/help/vision/ug/camera-calibration.html (visited on 07/28/2021) (cit. on p. 6).

[14] H. Lee and O. Choi, "An efficient parameter update method of 360-degree vr image model," *International Journal of Engineering Business Management*, vol. 11, p. 184 797 901 983 599, Apr. 2019. DOI: 10.1177/1847979019835993 (cit. on p. 7).

[15] Google. (2021). Enter physical viewer parameters., [Online]. Available: https://support.google.com/cardboard/manufacturers/answer/6324808?hl=en#zippy=%2Cdistortion-coefficients%2Cadvanced-viewer-parameters (visited on 07/28/2021) (cit. on p. 7).

[16] U. Engine. (2021). Make lensdistortioncameramode., [Online]. Available: https://docs.unrealengine.com/en-US/BlueprintAPI/Utilities/Struct/MakeLensDistortionCameraModel/index.html (visited on 07/28/2021) (cit. on p. 7).

[17] openHMD. (2017). Universal distortion shader., [Online]. Available: https://github.com/OpenHMD/OpenHMD/wiki/Universal-Distortion-Shader (visited on 07/28/2021) (cit. on p. 7).

[18] PanoTools. (2021). Lens correction model., [Online]. Available: https://wiki.panotools.org/Lens_correction_model (visited on 07/28/2021) (cit. on p. 7).

[19] openHMD. (2021). Lens distortion parameters for oculus rift dk2., [Online]. Available: https://github.com/OpenHMD/OpenHMD/blob/4b2648f65c390588d7b8d41adf33857d4056b1c9/src/drv_oculus_rift/rift.c#L920 (visited on 07/28/2021) (cit. on p. 8).

[20] openHMD. (2021). Lens distortion parameters for oculus rift cv1., [Online]. Available: https://github.com/OpenHMD/OpenHMD/blob/4b2648f65c390588d7b8d41adf33857d4056b1c9/src/drv_oculus_rift/rift.c#L928 (visited on 07/28/2021) (cit. on p. 8).

[21] GNU. (2021). Gnu octave., [Online]. Available: https://www.gnu.org/software/octave/index (visited on 07/28/2021) (cit. on p. 10).

[22] Shaw. (2017). The legendary fast inverse square root., [Online]. Available: https://medium.com/hard-mode/the-legendary-fast-inverse-square-root-e51fee3b49d9 (visited on 07/28/2021) (cit. on p. 15).

[23] Wikipedia. (2021). Fast inverse square root., [Online]. Available: https://en.wikipedia.org/wiki/Fast_inverse_square_root (visited on 07/28/2021) (cit. on p. 15).

[24] Wikipedia. (2021). Foveated rendering., [Online]. Available: https://en.wikipedia.org/wiki/Foveated_rendering (visited on 07/28/2021) (cit. on p. 22).

[25] I. L. News. (2019). Vive pro eye ausprobiert: Foveated rendering für einen schöneren bmw-konfigurator., [Online]. Available: https://www.immersivelearning.news/2019/01/15/vive-pro-eye-ausprobiert-foveated-rendering-fuer-einen-schoeneren-bmw-konfigurator/ (visited on 07/28/2021) (cit. on p. 22).

[26] X. Meng, R. Du, and A. Varshney, "Eye-dominance-guided foveated rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 5, pp. 1972–1980, 2020. DOI: 10.1109/TVCG.2020.2973442 (cit. on p. 23).

[27] G. Developers. (2016). Vr distortion correction using vertex displacement for cardboard apps - google i/o 2016., [Online]. Available: https://www.youtube.com/watch?v=yJVkdsZc9YA (visited on 07/28/2021) (cit. on p. 23).

[28] Nvidia. (2021). Nvidia rtx 20-series., [Online]. Available: https://www.nvidia.com/en-us/geforce/20-series/ (visited on 07/28/2021) (cit. on p. 23).

[29] S. Nuno. (2018). Introduction to real-time ray tracing with vulkan., [Online]. Available: https://developer.nvidia.com/blog/vulkan-raytracing/ (visited on 07/28/2021) (cit. on p. 24).

[30] C. Base. (2021). Ampere gigarays / rtx-ops., [Online]. Available: https://www.computerbase.de/forum/threads/ampere-gigarays-rtx-ops.2008724/ (visited on 07/28/2021) (cit. on p. 24).

[31] E. Andersen. (2019). Fpga-gpu, [Online]. Available: https://github.com/charliehorse55/fpga-gpu (visited on 07/28/2021) (cit. on p. 32).

[32] J. Peddie. (2021). Rv64x: A free, open source gpu for risc-v., [Online]. Available: https://www.eetimes.com/rv64x-a-free-open-source-gpu-for-risc-v/ (visited on 07/28/2021) (cit. on p. 32).

[33] Xilinx. (2021). Delivering a generation ahead at 20nm and 16nm., [Online]. Available: https://www.xilinx.com/about/generation-ahead-16nm.html (visited on 07/28/2021) (cit. on p. 33).

[34] G. Krishna and S. Roy, "Fundamentals of fpga architecture," in. Jan. 2017 (cit. on p. 34).

[35] Xilinx. (Sep. 2020). Ultrascale architecture dsp slice., [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf (visited on 07/28/2021) (cit. on pp. 34, 35).

[36] Nvidia. (2021). Nvidia ampere-architecture., [Online]. Available: https://www.nvidia.com/en-us/data-center/ampere-architecture/ (visited on 07/28/2021) (cit. on p. 66).

[37] Nvidia. (2021). Nvidia ampere ga102 gpu architecture., [Online]. Available: https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf (visited on 07/28/2021) (cit. on p. 67).