Deep zoom Mandelbulb on FPGAs

High-Precision Ray Marching on FPGAs using CORDIC

1st Kevin Klein
Institute of Computer Engineering
Heidelberg University
Heidelberg, Germany
Kevin.Klein@stud.uni-heidelberg.de

2nd Dirk Koch
Institute of Computer Engineering
Heidelberg University
Heidelberg, Germany
dirk.koch@ziti.uni-heidelberg.de

***Abstract*—This project presents the first implementation of the Mandelbulb 3D fractal on Field-Programmable Gate Arrays (FPGAs), highlighting the importance of high precision in ray marching for accurate fractal rendering. Ray marching, a technique used to render complex 3D structures like the Mandelbulb, requires high precision to achieve detailed and accurate visualizations. Traditional implementations on GPUs often face precision limitations, especially at deep zoom levels required for the detailed exploration of the Mandelbulb. In this project we introduce a novel high-precision ray marching concept using a 3D Coordinate Rotation Digital Computer (CORDIC) core, primarily designed for coordinate transformations that are crucial for calculating the Mandelbulb. The 3D CORDIC core enhances precision and computational efficiency on FPGAs, enabling higher precision calculations compared to conventional GPU implementations. This allows for deeper zoom levels and more detailed visualizations of the Mandelbulb fractal.**

***Index Terms*—mandelbulb, ray marching, high precision**

I. Introduction

In the domain of graphics processing, GPUs are the predominant choice, excelling in ray tracing, real-time rendering in video games, machine learning, deep learning, and more. Manufacturers provide various GPUs tailored for different needs. For instance, consumer GPUs like NVIDIA's GeForce RTX series are optimized for graphical rendering in video games, while specialized GPUs such as the NVIDIA Tesla series are geared towards scientific computing. Consumer GPUs excel in single-precision floating-point operations, allowing for rapid calculation of simple primitive polygons, though they also support double precision at a reduced performance rate.

Despite their versatility, GPUs face challenges with certain visualization algorithms requiring arithmetic accuracy beyond single or even double precision. Ray marching [1] is one such algorithm and almost always used for visualizing high-precision images like the Mandelbulb and complex surfaces [2] due to the difficulty or impossibility of finding ray tracing intersection formulas for these cases.

Ray marching is also advantageous in direct volume rendering [3], [4], where it samples points sequentially along a ray cast through a 3D dataset. This technique accurately represents complex volumetric phenomena like smoke, fog, or biological tissues [5], [6]. High precision ensures subtle variations in density, color, or intensity within the volume are captured and rendered, which is crucial for applications like medical imaging, aiding in diagnosis and treatment planning.

In this project, we implemented a hardware ray marcher capable of calculating the Mandelbulb [7], [8] at very deep zoom levels, see section V. This implementation allows for deeper zoom into the Mandelbulb compared to a standard consumer GPU. We compared the FPGA implementation on the Nexys Video Artix-7 XC7A200T with an RTX 3080. Our design is modular and scalable, meaning it can be relatively easily transferred to FPGAs of different sizes, with higher precision achievable on FPGAs with more resources.

A. Problem Statement

Introduced by Daniel White and Paul Nylander in 2009, the Mandelbulb was created through a modification of the original Mandelbrot formula, incorporating power functions and quaternion mathematics, see subsection IV-B. Rendering 3D fractals is a complex task that often involves techniques like ray marching to create detailed images of fractal structures. The heart of ray marching is the Signed Distance Function (SDF, see section II), which calculates the shortest distance to the 3D fractal from any point in space. Particularly the Mandelbulb SDF is very complex from a hardware perspective, as it involves: powers, roots, logarithms, trigonometric functions, division, and hyperbolic functions.

Ray marching is frequently utilized for rendering complex objects such as implicit scalar fields or (hyper)-complex functions in order to realistically depict phenomena such as clouds, three-dimensional fractals (Mandelbulb), and more. Often, this involves the use of 3D coordinate transformations to convert Cartesian coordinates into spherical coordinates and vice versa. Our novel approach involves utilizing a 3D CORDIC-Core for this purpose as shown in subsection IV-C. In general the CORDIC algorithm has been utilized for computing trigonometric functions and more on FPGAs for many years [9] and constitutes an essential component of the libraries of various manufacturerss [10].

Furthermore, to achieve a scalable design that can be easily transferred to various FPGA sizes, we propose the approximate multiplier for adjustable bit precision, see subsection IV-D. This allows to specify how many DSP slices and LUTs should be used, which is sensible for FPGAs with fewer resources.

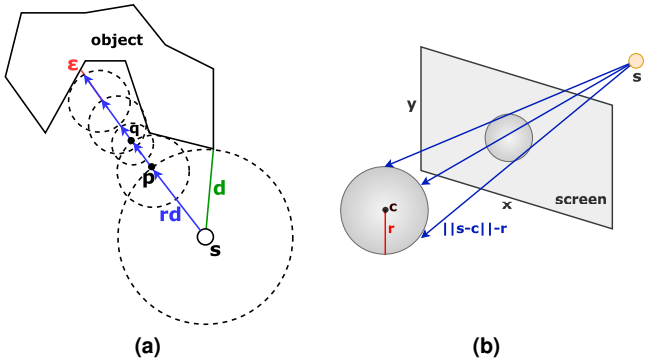


Fig. 1. (a): Ray marching an object's surface. (b): SDF of a sphere projected on screen.

```
def ray_march(SDF, s, rd, max_steps,
    epsilon, max_dist):
  tn = 0
  tl = 0
  for i in range(max_steps):
    dn = SDF(s + tn*rd)
    tn = dn + tn
    tl = tn
    if (tn > max_dist or dn <= epsilon):
      break
  return tl
```

Listing 1. General ray marching in Python

We primarily employ the CORDIC algorithm due to its reputation for computing with arbitrary precision and its suitability for pipelining, which is crucial for real-time rendering (our implementation achieves 5 fps). Additionally, pipelining is always advantageous as it conserves power and eases the timing constraints for the place and route tool.

As mentioned, in ray marching, objects are represented by SDFs and for very complex objects like the Mandelbulb, the distance can only be approximated using a distance estimator, see subsection IV-E, which necessitates the use of logarithmic computations. For this purpose the CORDIC method can be used again. Specifically, we used the BKM algorithm [11] as it is somewhat simpler to implement.

II. A Brief Introduction to Ray Marching

In ray marching, a ray is projected from the viewer's perspective and moves outward into the scene. At each step along the ray, the distance from the ray to the nearest surface in the scene is calculated. Ray marching is performed on "signed distance functions" (SDFs), which provide the shortest distance between a point in space and the surface of an object. If the distance d between the ray and the object is $d \leq \varepsilon$, it indicates a collision. The characteristics of the object at that point determine the color and brightness of the pixel. For example, a pixel can be colored based on the number of ray marching steps or the distance between the viewer and the object.

The technique of ray marching relies on a step size that is not constant but rather calculated by the SDF as shown in Figure 1(a). From a starting point s, the SDF always calculates the distance $\mathrm{SDF}(s) = d$ to the nearest surface of an object. One can move this distance along a view ray rd to a new point p without intersecting with the object. This process is then repeated for the new starting point p. The process terminates if a specified number of iterations k is exceeded, if a collision occurs $d \leq \varepsilon$, or if one exceeds a maximum defined distance from the object $d \geq$ `max_dist`. This allows us to approach complex surfaces surfaces very quickly.

The rays are generated in three dimensions by using the (x,y) screen coordinates as a two-dimensional projection plane in space and calculating the direction vectors rd to s, which then point to the object described by the SDF, see Figure 1(b).

Formally, ray marching in $\mathbb{R}^3$ can be defined as follows, let $s \in \mathbb{R}^3$ be the starting point i.e. the viewer, $rd \in \mathbb{R}^3$ the ray direction with $\|rd\| = 1$ and $(d_n)_{n=0\ldots k}$ the distance to the surface recalculated at each step, and $k \geq 1$ the maximum number of steps. Then the total length of the ray $t_l \in \mathbb{R}$ with $(t_n)_{n=0\ldots k}$ is calculated as follows:

$$\begin{aligned} t_0, d_0 &= 0 \\ d_n &= \mathrm{SDF}(s + t_n \cdot rd) \\ t_{n+1} &= d_n + t_n \end{aligned} \tag{1}$$

If any of these conditions $\{d_n \leq \varepsilon,\ t_n >$ `max_dist`$,\ n = k\}$ is true for n for the first time, than we break the loop and set $t_l = t_n$. In this way, t_l is the sum of all ray marching steps that we have taken until the condition is met.

Listing 1 shows an implementation in code of ray marching. This serves to better illustrate the algorithmic principle. Any arbitrary SDF can be taken as input because, for ray marching, it is only important to obtain a numerical value representing the shortest distance between an object and the camera.

III. Related Work

In the field of 3D real-time rendering on FPGAs, there have been several noteworthy developments and research areas. We draw inspiration from some interesting approaches and projects in that field. Although there is limited research on high-precision ray marching in relation to FPGAs, there are still some areas where ray marching in combination with FPGAs has been successfully applied, like an advanced system for enhancing real-time localization in miniature autonomous vehicles, leveraging FPGAs. Key to this system is the use of a Particle Filter algorithm, where the ray marching technique plays a critical role. Ray marching is often used to process LiDAR data for localization, creating rays from each particle to represent their positions and orientations. By accelerating the ray marching step on FPGAs, the system achieves greater efficiency and speed in determining the vehicle's location, which is essential for autonomous navigation [12].

To enhance Simultaneous Localization and Mapping (SLAM) in autonomous robotics, a method for LiDAR-based

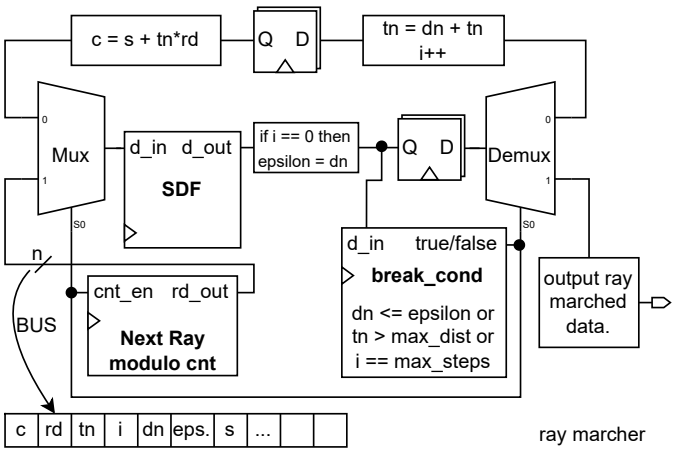


Fig. 2. A generic hardware ray marcher for marching any pipelined SDF.

SLAM in robotics was accelerated on an FPGA-for energy-efficiency [13]. This method employs a Truncated Signed Distance Function (TSDF) for detailed 3D mapping, allowing the robot to efficiently map its environment and track its location. The TSDF accurately represents complex shapes by storing the distance to the nearest surface in each voxel of a grid.

In addition to the mentioned methods, our work focuses on high precision in ray marching to explore complex surfaces in detail even when zooming in real-time.

IV. Implementation

To implement the Mandelbulb on an FPGA, two main steps are necessary:

1) A hardware ray marcher must be created that can perform SDF ray marching.
2) The Mandelbulb SDF must be developed, which will then be used by the hardware ray marcher.

Most GPU visualizations use the Mandelbulb SDF as described by Inigo Quilez [8]. Our project is based on this version of the SDF.

A. Ray Marching on Hardware

First, a general ray marcher must be designed, which will then receive values from a SDF block every clock cycle.

The SDF block in Figure 2 is a placeholder for any pipelined SDF that can be implemented on an FPGA. Later we will use the Mandelbub SDF for the SDF block.

The hardware ray marcher enables load balancing by running multiple instances on different pixel regions in parallel. This is achieved by limiting the Next Ray module's modulo counter to specific pixel areas per instance. Complex geometries require frequent ray marching per view ray to capture small details caused by grooves, as shown in Figure 1(a). Simple geometries, such as spheres, require less ray marching iterations in total due to their smoothness, see Figure 1(b). To create a detailed visualization of, for example, the Mandelbulb's complex surface, the SDF block in Figure 2 had to be reused 200 times in our experiments, that means that data must be circulate at most 200 times around the pipeline. Because `break_cond` is a large combinatorial block, we have clocked it so that the signal arrives at the demultiplexer with a one-clock-cycle delay.

The most relevant part of the hardware ray marcher is the synchronous circuit of the Multiplexer and Demultiplexer. If `break_cond` is false, the SDF core is reused, and the data circulate along the null input of the Multiplexer and the null output of the Demultiplexer, performing ray marching as in the code within the loop. If `break_cond` is true, the Multiplexer and Demultiplexer switch to one, and the circulating data packet is output, while at the same time, the next ray is generated for the next pixel and introduced into the SDF loop. The data to be processed is stored on a large data bus because the data must stay together since the loop is unrolled. The advantage of this ray marching concept is that no gaps occur in the data pipeline, and data is output immediately when it is no longer needed.

To ensure that the distance from the camera to the object never becomes greater than `epsilon` when zooming, one sets `epsilon = dn >> b`. It is important that `a` for `epsilon = a` is *not* a constant number like 0.0001. Because if we zoom in, we eventually get closer to the object than 0.0001 and zooming is not possible anymore. If `a` is excessively small, such as $a = 10^{-12}$, then aliasing becomes a significant issue, as we have encountered in our experiments. This is because for a small `a` too much details are visible at once. To address the issues, we have set `epsilon` in relation to the current camera position (daynamic epsilon) with `epsilon = dn >> b`. Although `a` becomes very small in the dynamic epsilon approach, this does not pose a problem because aliasing only occurs when `a` is very small and the camera is positioned far from the object. For the Mandelbulb example, we set `b=12`, which was determined experimentally.

B. Mandelbub SDF on FPGAs

The Mandelbulb originates from a modification of the Mandelbrot formula $z_{n+1} = z_n^d + c,\ z_n, c \in \mathbb{C}$. The key idea to get the Mandelbulb is to represent the complex numbers in polar form. Thus, one obtains the Mandelbrot formula in polar form:

$$z_n = r_n e^{i\varphi_n} \Rightarrow z_n^d = r_n^d e^{i(d\varphi_n)} = r_n^d(\cos(d\varphi) + i\sin(d\varphi))$$

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = r_n^d \begin{pmatrix} \cos(d\varphi_n) \\ \sin(d\varphi_n) \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \end{pmatrix} \tag{2}$$
$$r_n = \sqrt{x_n^2 + y_n^2} \quad \varphi_n = \arctan2(y_n, x_n)$$

It is now possible to easily transform Equation 2 into higher dimensions. For the Mandelbulb, Equation 2 is expanded to three dimensions by simply using spherical coordinates:

$$
\begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} = 0, \quad \|r_n\| = \sqrt{x_n^2 + y_n^2 + z_n^2} \leq 1.5
$$

$$
\begin{pmatrix} x_{n+1} \\ y_{n+1} \\ z_{n+1} \end{pmatrix} = r_n^d \begin{pmatrix} \sin(d\theta_n)\cos(d\phi_n) \\ \sin(d\theta_n)\sin(d\phi_n) \\ \cos(d\theta_n) \end{pmatrix} + \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} \tag{3}
$$

$$
r_n = \sqrt{x_n^2 + y_n^2 + z_n^2}, \quad \phi_n = \arctan 2(y_n, x_n),
$$

$$
\theta_n = \arccos\left(\frac{z_n}{r_n}\right)
$$

The advantages of using polar or spherical coordinates lie in the fact that for large powers of d, no large polynomials are created. Only the angles need to be scaled with d, and the radius is raised to the power of d.

The main advantage revealed by Equation 3 is that it only computes coordinate transformations rather than high order polynomials. A point is transformed into spherical coordinates, then scaled and rotated, and then transformed back into Cartesian coordinates. This means trigonometric functions and magnitudes can be calculated with CORDIC [9], [14].

CORDIC, is well suited for achieving high bit precision. Thus, it is also possible to calculate the Mandelbulb on small FPGAs with lower precision, making the implementation scalable. To obtain the Mandelbulb SDF, usually a distance estimator [8], [15] must be added to Equation 3, since only an approximation of the distance from the camera to an object can be achieved, which is sufficient for Mandelbulb visualization:

$$
dr_{n+1} = d \cdot r_n^d \cdot dr_n + 1 \tag{4}
$$

$$
\mathrm{dst} = 0.5 \cdot \ln(r_k) \cdot \frac{r_k}{dr_k} \tag{5}
$$

The shortest distance from the camera to Mandelbulb is given by dst (distance estimator), see Equation 5. Here, k is the index of the last iteration, exactly as with Mandelbrot when the escape radius is exceeded or the maximum number of iterations has been reached. The pixels of the Mandelbulb are colored based on the steps k.

For the distance estimator, we must also calculate the derivative (see Equation 4). This can happen during the SDF iterations [16]–[20].

In Equation 5, we will use the BKM algorithm [11] for the logarithm in "L-Mode", and again CORDIC for division [21]. BKM is similar to CORDIC and well suited for high precision.

The concept of coordinate transformations and the distance estimator is not limited to the Mandelbulb. Many SDFs use these methods, especially (hyper)-complex SDFs, to avoid large polynomials or SDFs that do not have an explicit representation [18], [20]. The Mandelbulb SDF is just one representative of a large class of implicit scalar fields.

We will deeply pipeline the Mandelbulb SDF and than deploy it inside the hardware ray marcher. For this, the loop of the Mandelbulb SDF must be unrolled as shown in Figure 3.

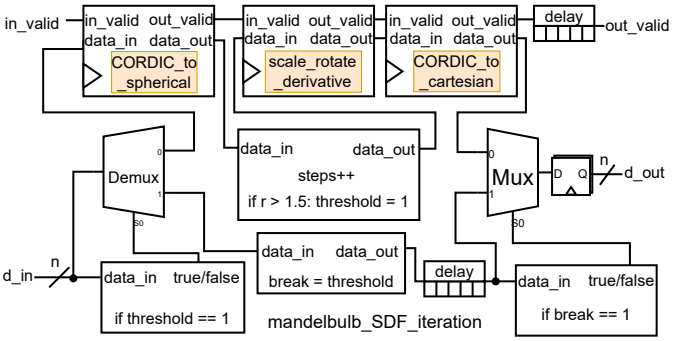


Fig. 3. An unrolled Mandelbulb SDF loop. It is designed to support a pipelined version of the Mandelbub SDF.

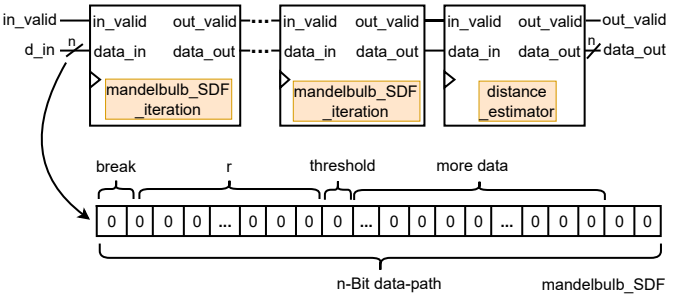


Fig. 4. Pipelined Mandelbub SDF implementation.

The most important blocks of the Mandelbulb SDF iteration are the CORDIC blocks in which the coordinate transformations take place. It must be checked in each iteration whether the escape radius of 1.5 is exceeded, then the flag `threshold` is set so that this data is no longer processed further.

The Mandelbulb SDF iteration core is designed to be scalable, a valid signal at the input and output gives the next Mandelbulb SDF iteration core the information whether the data at the end of the pipeline are already valid for further calculations.

In this way, the Mandelbulb SDF can be easily constructed by connecting several Mandelbulb SDF iteration cores in sequence, as shown in Figure 4. The individual Mandelbulb SDF iterations work on the same data bus as the hardware ray marcher. Each intermediate result of a Mandelbulb SDF iteration is passed on to the next Mandelbulb SDF iteration via the data bus.

The number of Mandelbulb SDF iteration cores corresponds to the number of iterations k. The distance estimator core is the implementation of Equation 5 and is located outside of the loop and writes the shortest distance from the Mandelbulb to the camera onto a part of the data bus. During ray marching, we set $d_n = \mathrm{dst}$ because we need to use the SDF for a pixel multiple times.

C. 3D CORDIC Core

The Mandelbulb SDF is described by Equation 3, Equation 4, and Equation 5, and implemented on the FPGA as illustrated in Figure 3 and Figure 4. In each iteration,

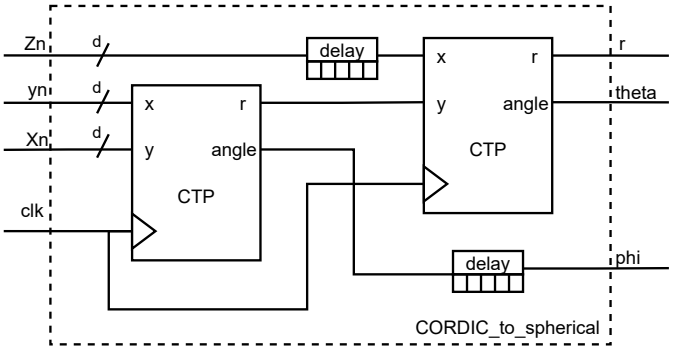


Fig. 5. 3D cartesian-to-spherical coordinates transform using CORDIC (as implemented in the Mandelbulb SDF iteration).

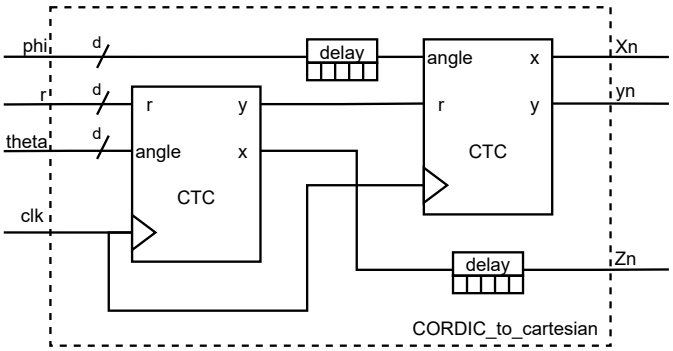


Fig. 6. Spherical-to-3D cartesian coordinates transform using CORDIC (as implemented in the Mandelbulb SDF iteration).

coordinate transformations must be performed. Our innovative idea is to use CORDIC for these coordinate transformations, specifically in three-dimensional space.

CORDIC is primarily designed for two dimensional coordinates, and 2D coordinate transformations can be implemented on an FPGA as shown in [9].

Since our ray marching design focuses on 3D space, the CORDIC algorithms also need to work in three dimensions. Therefore, we use the proposed "3D-CORDIC processor" from [22] for converting 3D Cartesian coordinates to spherical coordinates and vice versa.

However, the proposed "3D-CORDIC processor" only converts Cartesian coordinates to spherical coordinates. We extended its functionality to enable bidirectional conversion, which is essential for Mandelbulb calculations.

The architecture of the conversion functions are shown in Figure 5 and Figure 6, respectively. Here, we use two individual CORDIC stages to form 3D-CORDIC Cores. In the figures, CTP stands for the conversion from 2D Cartesian coordinates to polar coordinates, and CTC stands for the conversion from polar to 2D Cartesian coordinates as described in [9]. Given that no available CORDIC core provided the magnitude directly, we developed our own CORDIC implementation.

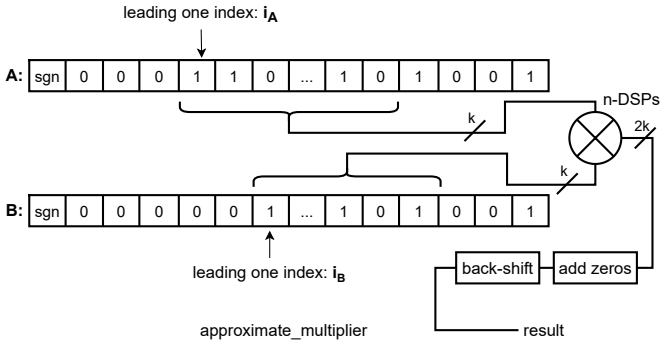


Fig. 7. Concept of an approximate multiplier with adjustable bits precision.

D. Multiplication

To enable ray marching on various FPGA sizes, we introduce a approximate multiplier concept, as shown in Figure 7. Our approach reduces the number of DSP slices by allowing adjustable precision in calculations. It identifies the leading one in inputs A and B, multiplies the subsequent k bits, and shifts the result back into the correct Q-format using the leading one indices i_A and i_B. With this, we maximize the effective number of significant bits processed by the DSP block.

In smaller FPGAs, for example, k can be set so that only one DSP slice is used.

E. Distance Estimator

The distance estimator is described by Equation 5, and implemented on the FPGA as illustrated in Figure 4. Typically, the SDF calculates the exact shortest distance from an object to the camera. However, due to the complexity of the Mandelbulb, this distance can only be approximated using a distance estimator. For the distance estimator, it is crucial to compute the directional derivative to the isosurface, as demonstrated in Equation 4.

In the distance estimator, the logarithm needs to be calculated, for which we use the BKM [11] algorithm in "L-Mode" (BKM has also modes like CORDIC). The BKM algorithm works similarly to CORDIC, except that a table with values $\log(1+2^{-n})$ needs to be created. It's important to note that the algorithm only converges for input values between 1 and 4.768. The inputs must consequently be normalized beforehand and denormalized later.

In the distance estimator, division must also be performed. For this, we use CORDIC in linear mode [21]. It is important that the result of the positive division must be less than or equal to two. Therefore, we normalize the numbers to be divided before the CORDIC-Division and denormalize the result afterwards.

F. Number Representations

For the Mandelbulb SDF, we will use fixed-point numbers since the intermediate results from Equation 3 lie within a small interval, due to the boundedness of the trigonometric

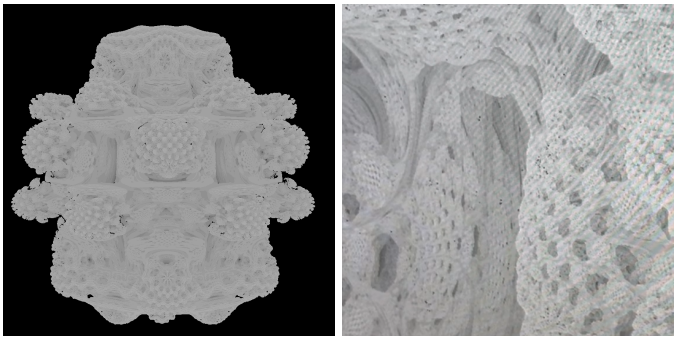


Fig. 8. (a): The Mandelbulb rendered on the FPGA using our hardware ray marcher. (b):The Mandelbulb on the FPGA zoomed in a bit.

functions. Most intermediate results are smaller than 128. Only the derivative from Equation 4 becomes very large, up to 2^{22}, so, for this case, a different fixed-point representation must be chosen.

We use 65-bit numbers in two's complement form. For most calculations, the Q8.57 format is used since only a few integer bits are needed. For the derivative, the Q33.32 format is used because the derivative becomes very large and many integer bits are required. The goal of this number representation is also to be able to calculate more accurately than double precision, but this primarily depends on the resources, especially LUTs.

V. Results and Comparison with a GPU

We implemented the Mandelbulb together with the hardware ray marcher on the Artix-7 XC7A200T FPGA, the board used is the Digilent Nexys Video. To test the scalability of our design, we also implemented it on the Artix-7 XC7A100T platform, with the Nexys 4 DDR board.

On the Nexys Video board, we were able to achieve significantly deeper zoom levels compared to a standard consumer GPU (RTX 380).

By zoom level, the number of decimal/binary places that do not change when zooming is meant, we will refer to this with z_l. The maximum zoom level $z_m = |\log_2(d)|$ for a given zoom position is a measure of how small the distance d on the surface of an object can be between two different rays r_n, r_m so that the GPU or FPGA can still differentiate them. It is clear that d must always become smaller when zooming because the camera is approaching the object.

At deep zoom levels, the units in last place (ULP) problem occurs: $d = |r_n - r_m| \leq \text{ULP}(a)$. Here, $a \in [0.8, 1.2]$ is set because a large part of the Mandelbulb surface has a distance interval of a from the origin. As a consequence, images become pixelated when z_m is approached beacuse the rays are too close for further precision.

In the following sections, we will discuss the details of our results.

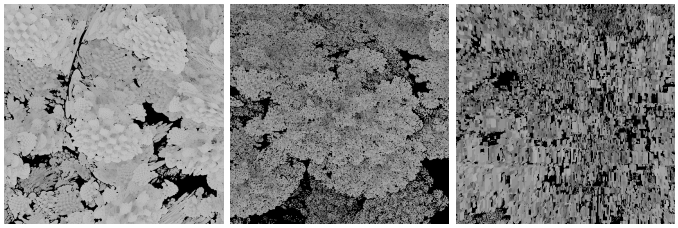


Fig. 9. (a): Less zoomed Mandelbulb on the GPU. (b): Medium zoomed Mandelbulb on the GPU. (c): Deeply zoomed Mandelbulb on the GPU.

Fig. 10. (a): Less zoomed Mandelbulb on the FPGA. (b): Medium zoomed Mandelbulb on the FPGA. (c): Deeply zoomed Mandelbulb on the FPGA.

A. GPU Analysis

We conducted our GPU tests on Shadertoy [23]. One can write their own shader on the website, which is rendered by Web-GL on their own GPU. Our GPU was an RTX 3080. The shader language is modeled after GLSL. With these tests, we often compared our FPGA implementation. Moreover, and this is our goal, we wanted to find out how our FPGA implementation can be better than a GPU implementation in terms of zoom level an render speed. The shaders use the data type `float` for the representation of rational numbers. We do not need to derive floating point in detail, but a few facts are important to understand how our FPGA implementation could be better than the GPU. The `float` data type is in IEEE-754 single precision which means, the number has a bit-width of 32 consisting of a sign bit s, 8 bits for the exponent e, and 23 bits for the mantissa m. The exponent is in the biased form and takes values from -126 to 127 in decimal.

The ULP(1) for single precision is $\text{ULP}(1) = 2^{-23}$. This means this is the smallest number that added to one does not yield one.

For the three images in Figure 9 we have the following zoom levels in bits: $(a) : z_l = 12$, $(b) : z_l = 21$, $(c) : z_l = 22$. For this particular zoom position the maximum zoom level of a GPU is $z_m = 23$ with $d = \text{ULP}(1)$.

B. FPGA Demo on the Nexys Video

On this board the ray marcher writes the data into the BRAM, which is used as a frame-buffer. The image output is in HD via the HDMI port. The ray marching data-path, including the SDF implementation, is deeply pipelined with

Component	Available	Used
LUTs	134600	117861 (87.6%)
Flip-Flops	269200	105386 (39.1%)
Logic Slices	33650	32725 (SLICEL 21244; SLICEM: 11841)
Block RAM	13 MBits	5.46 MBits
Clock	100MHz – 450MHz	100MHz
DSPs	740	312
Switches	8	(2)
Buttons	5	5
Device	**FPS at PP**	**Peak Power (PP)**
FPGA	0.96	4.6W
GPU	4.85	331.611W

TABLE I. Resource breakdown and peak power on the Artix-7 XC7A200T FPGA.

Component	Available	Used
LUTs	63400	58798 (92.74%)
Flip-Flops	126800	53198 (41.95%)
Logic Slices	15850	15530 (SLICEL 10781; SLICEM: 4749)
Block RAM	4 MBits	4 MBits
Clock	100MHz – 450MHz	100MHz
DSPs	240	161
Switches	8	0
Buttons	5	5
Device	**FPS at PP**	**Peak Power (PP)**
FPGA	1.8	2.52W
GPU	4.85	331.611W

TABLE II. Resource breakdown and peak power on the Artix-7 XC7A100T FPGA.

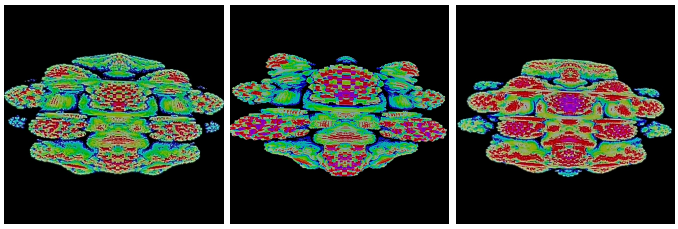
in total 1096 pipeline stages. This means that we have the same number of parallel executed rays in the pipeline in order to keep the hardware fully utilized. Due to the limited number of LUTs (134,600), only one ray marcher instance is used. The Mandelbulb is rendered between 5 and 1 fps, depending on the zoom depth. The accuracy is maintained at 32 bits, despite the use of 65-bit numbers for the implementation. This is attributed to the fact that the CORDIC algorithms on this board are limited to 32 iterations due to constrained resources, otherwise, calculations more precise than those achieved with double precision could be performed. Table I provides a more detailed overview of the resources used on this FPGA.

Table I also displays the peak power consumption for both the GPU and the FPGA, as well as the corresponding frame rates during these peak periods. While the FPGA consistently maintains a power consumption of 4.6W across various zoom levels, the GPU exhibits a significant increase in power consumption as the zoom level intensifies. This rise is attributed to the fact that higher zoom levels necessitate a greater number of ray marching iterations up to the maximum iteration limit (`max_iter`). In contrast, the FPGA's power consumption remains constant due to its data pipelines being perpetually filled. Consequently, it is logical that the frame rate drops significantly during peak power periods, particularly for the GPU, due to the high number of iterations.

Figure 8 shows a side view of the Mandelbulb. The Mandelbulb SDF was used in the hardware ray marcher with power $d = 8$.

The precision difference between FPGA and GPU when zooming can be seen in Figure 10, various stages of zooming deeper into the Mandelbulb are performed on the FPGA. We attempted to reach the same zoom locations and zoom levels as in Figure 9 for a good comparison between GPU and FPGA. Our FPGA implementation has a maximum zoom level of $z_m = 29$. For the three images in Figure 10, we have the following zoom levels in bits: $(a) : z_l = 12$, $(b) : z_l = 21$, $(c) : z_l = 26$.

Even with $z_l = 26$, the FPGA still generates clear images, see Figure 10(c), which would not have been possible with the GPU due to precision reasons because it only has a maximum zoom level of $z_m = 23$. The extended precision allows the FPGA to display $2 \times 2^{(29-23)} = 4096$ times more rays per area than the GPU.



(a) **(b)** **(c)**

Fig. 11. (a): Start position to view the Mandelbub. (b): The view of a zoomed Mandelbub. (c): It is posible to change the ray length in realtime to get different surface colors.

C. FPGA Demo on the Nexys 4 DDR

This is a demo of the Mandelbulb on the Artix-7 XC7A100T, see Figure 11(a). The Mandelbub is rendered at 10 FPS, and the user can zoom in by pressing a button, see Figure 11(b), changing the ray length, which leads to a change in color on the surface as shown in Figure 11(c). It will quickly become apparent that the Mandelbub looks pixelated at closer zoom levels, because this FPGA implementation can effectively only calculate with 8-9 bits due to the few CORDIC iterations because of the limited resources. The quality of the Mandelbulb should exponentially improve if there were more CORDIC iterations, because one gets one bit more precision per iteration. The FPGA has an energy consumption of 2.52W, and the image is outputted via VGA. Table II provides a more detailed overview of the resources used on this FPGA. The clock frequency that we used to clock the flip-flops was at 100MHz and 25MHz is the VGA output frequency.

We have here an accuracy of 2^{-9}, which implies that the first three decimal places are correct. The quality is almost exclusively dependent on the CORDIC iterations that can be implemented sequentially. In these Mandelbulb images, the aspect ratio has not been considered while normalizing the rays, hence the Mandelbulb appears wider. Another factor reducing quality is the low resolution of the VGA output of 640×480. As a result, the rays during normalization are accurate only to two decimal places. However, the low resolution of these VGA ports becomes particularly noticeable

on screens with higher resolution where the image is upscaled. The FPGA possesses 4860 KBits of BRAM, which we utilize as a framebuffer. Within this space, the calculated pixels are stored and subsequently read independently by another controller, each operating with its own clock. The VGA port supports 12-Bit RGB colors, implying that the entire BRAM is allocated for the framebuffer's use. Most flip-flops were utilized for delaying data, and the majority of DSP Slices were employed for the multiplication of the eighth and seventh powers of the Mandelbulb SDF.

In this study, we have utilized the Mandelbub implementation from the Nexys 4 DDR and successfully ported it to the Nexys Video. This transition was accomplished with relative ease, requiring no modifications to the algorithms. A new framebuffer was necessitated due to the implementation's operation at HD resolution 1280×720. This framebuffer possesses a size of 921600, and in this iteration, the color IDs are stored utilizing 5 bits which then address a 25×24 ROM color table. The output for this version (Nexys Video) is facilitated through HDMI.

VI. Difficulties

We implemented each algorithm individually on the FPGA (CORDIC, BKM, Multipliers) and checked how many resources were consumed, whether the timings were correct, and whether they functioned correct. We noticed that if an algorithm consumes a certain number of resources and we implement it twice, then the total resources are usually less than the sum of the resources consumed by the individual algorithms. Here, by resources, we mean LUTs and flip-flops.

The simulation of the Mandelbub SDF turned out to be challenging, since its implementation is deeply piplined (depending on CORDIC iterations we started at 900 stages), it cannot just return the distance, all intermediate results that appear in an iteration of the Mandelbulb SDF must be returned because the loop was unrolled. This means we had to simulate several runs and compare the intermediate results with the Python implementation, as it is correct.

The next significant challenge of this project was the high resource consumption. This is due to our deep pipeline stages, which prevent the reuse of larger logic blocks because we aim to produce output from the Mandelbulb SDF with every clock cycle. However, pipelining is crucial for achieving optimal frame rates. For example, a single iteration of the Mandelbulb SDF consumes 20K LUTs, and we require at least 25 iterations for satisfactory results. We addressed this by pipelining 5 iterations of the Mandelbulb SDF and reusing these 5 iterations five times, although this approach reduces the frame rate by a factor of five. Therefore, better results can always be expected on larger FPGAs.

Dealing with the power limits of the FPGA boards was also challenging. Due to the high resource consumption on both the Nexys 4 DDR and the Nexys Video, our power consumption is consistently at the maximum limit, causing the chips to become very hot. Nevertheless, it was important for us to run the Mandelbulb on these boards because they are frequently used in university education, making it easier for others to implement the Mandelbulb.

VII. Conclusion

In this project, we implemented a scalable design for a general ray marcher on FPGAs, aiming at enhancing the rendering of complex 3D scenes with a high level of detail and accuracy, exemplified through the visualization of the Mandelbulb fractal. Our approach leverages the inherent flexibility and computational capabilities of FPGAs, demonstrating the potential for executing ray marching algorithms more efficiently than traditional GPU-based methods, especially for handling high-precision computations.

A pivotal aspect of our project lies in the innovative utilization of the CORDIC algorithm extended to three dimensions for performing coordinate transformations, alongside the integration of an approximate multiplier. This methodological approach enables the precise and adaptable handling of high-precision SDFs. Such a technique is indispensable for the accurate rendering of complex geometrical structures, notably the Mandelbulb, allowing for the meticulous depiction of its elaborate details.

Moreover, our design's scalability ensures that the methods developed for the Mandelbulb SDF can be generalized and applied to other SDFs, opening avenues for rendering a wide range of detailed 3D objects across different FPGA platforms.

References

[1] M. Walczyk, "Ray marching," 2020. [Online]. Available: https://michaelwalczyk.com/blog-ray-marching.html

[2] K. Watters and F. Ramallo, "Raymarching toolkit for unity: a highly interactive unity toolkit for constructing signed distance fields visually," in *ACM SIGGRAPH 2018 Studio*, ser. SIGGRAPH '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3214822.3214828

[3] S. Green, "Volume rendering for games," in *Game Developers Conference*, vol. 2005. NVIDIA, 2005. [Online]. Available: https://download.nvidia.com/developer/presentations/2005/GDC/Sponsored_Day/GDC_2005_VolumeRenderingForGames.pdf

[4] R. A. Drebin, L. Carpenter, and P. Hanrahan, "Volume rendering," ser. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 65–74. [Online]. Available: https://doi.org/10.1145/54852.378484

[5] K. Bittner, "The current state of the art in real-time cloud rendering with raymarching," 01 2020. [Online]. Available: https://www.researchgate.net/publication/343404421_The_Current_State_of_the_Art_in_Real-Time_Cloud_Rendering_With_Raymarching

[6] K. Zhou, Z. Ren, S. Lin, H. Bao, B. Guo, and H.-Y. Shum, "Real-time smoke rendering using compensated ray marching," in *ACM SIGGRAPH 2008 Papers*, ser. SIGGRAPH '08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1399504.1360635

[7] D. White, "Real 3d mandelbulb," 2009. [Online]. Available: https://www.skytopia.com/project/fractal/mandelbulb.html

[8] I. Quilez, "Mandelbulb," 2009, https://iquilezles.org/articles/mandelbulb.

[9] R. Andraka, "A survey of cordic algorithms for fpga based computers," in *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, ser. FPGA '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 191–200. [Online]. Available: https://doi.org/10.1145/275107.275139

[10] AMD, "Cordic." [Online]. Available: https://www.xilinx.com/products/intellectual-property/cordic.html#overview

[11] J.-C. Bajard, S. Kla, and J.-M. Muller, "Bkm: a new hardware algorithm for complex elementary functions," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 955–963, 1994.
[12] A. Bernardi, G. Brilli, A. Capotondi, A. Marongiu, and P. Burgio, "An fpga overlay for efficient real-time localization in 1/10th scale autonomous vehicles," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022, pp. 915–920.
[13] M. Flottmann, M. Eisoldt, J. Gaal, M. Rothmann, M. Tassemeier, T. Wiemann, and M. Porrmann, "Energy-efficient fpga-accelerated lidar-based slam for embedded robotics," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 1–6.
[14] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959.
[15] Syntopia, "Distance estimated 3d fractals (v): The mandelbulb and different de approximations," 2011. [Online]. Available: http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/
[16] J. C. Hart, D. J. Sandin, and L. H. Kauffman, "Ray tracing deterministic 3-d fractals," *SIGGRAPH Comput. Graph.*, vol. 23, no. 3, p. 289–296, Jul. 1989. [Online]. Available: https://doi.org/10.1145/74334.74363
[17] I. Quilez, "3d julia sets," 2001. [Online]. Available: https://iquilezles.org/articles/juliasets3d
[18] ——, "distance estimation," 2011. [Online]. Available: https://iquilezles.org/articles/distance
[19] ——, "distance to fractals," 2004. [Online]. Available: https://iquilezles.org/articles/distancefractals
[20] Y. Dang, L. H. Kauffman, and D. Sandin, *Hypercomplex Iterations*. WORLD SCIENTIFIC, 2002. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/3625
[21] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, ser. AFIPS '71 (Spring). New York, NY, USA: Association for Computing Machinery, 1971, p. 379–385. [Online]. Available: https://doi.org/10.1145/1478786.1478840
[22] M. Al-Homosy, M. Abass, A. Al-Kholy, and R. A., "A novel design and implementation of fpga based 3d-cordic processor," *The International Conference on Electrical Engineering*, vol. 6, no. 6th International Conference on Electrical Engineering ICEENG 2008, pp. 1–18, 2008. [Online]. Available: https://iceeng.journals.ekb.eg/article_34337.html
[23] Shadertoy, 2014. [Online]. Available: https://www.shadertoy.com