NTNU

Norwegian University of
Science and Technology

Design of a fractal generator for on-the-fly generation of textures for Mali GPU

Per Christian Corneliussen

Master of Science in Electronics
Submission date: June 2011
Supervisor: Per Gunnar Kjeldsberg, IET
Co-supervisor: Øystein Gjermundnes, ARM Norway AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Abstract
This proposal describes a possible subject for a master thesis for students with background in microelectronics and computer graphics. The main part of the work related to this thesis will go into the design of a fractal generator for on-the-fly generation of textures for a Mali GPU. The fractal generator will be realized in an FPGA. When the fractal generator is finished, the student may participate in one of the regular demo competitions at ARM Norway.

Introduction
A fractal is "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole,"(1) a property called self-similarity. A mathematical fractal is based on an equation that undergoes iteration, a form of feedback based on recursion(2).

One such mathematical fractal is the fractal defined by the Mandelbrot set. The Mandelbrot set is a mathematical set of points in the complex plane, the boundary of which forms a fractal (3). The point c belongs to the Mandelbrot set if and only if $|Z_n| = 2$, for all $n = 0$, where $Z_{n+1} = Z_n^2 + c$ and $Z_0 = 0$ (Eq. 1)

An image can be created from the Mandelbrot set by mapping the (x,y) coordinates of the pixels in the image to the real and imaginary parts of a complex number. For each pixel it is computed how many iterations that is necessary of Eq. 1 before the absolute value of the complex number Zn exceeds 2. The number of iterations is then used as an index into a colour palette which finally determines the colour of the pixel.

A globe is made by wrapping a map around a sphere. This process is known as texture mapping in the field of computer graphics, and is used for drawing or wrapping an image on to a 3D object. Textures are in many cases generated in advance to running a computer game, but they could also be generated on the fly.

Thesis statement
Give an overview of different fractals with the focus on fractals generated by the Mandelbrot set. Also explain briefly relevant computer graphics terms such as texturing.

Create a OpenGL ES 2.0 program that creates a animated 3D landscape where the height and the colour of the landscape at any given point is determined by the colour of a Texel in a texture.

Design a fractal generator for later implementation in an FPGA. Do computations to find out what kind of frame rate you would expect as well as some coarse size estimates that can tell you early on whether or not the architecture will fit into the provided FPGA.

The fractal generator must have an AXI read interface. The read address should identify the (x,y) coordinates of a pixel in a Mandelbrot fractal and the read data returned should be the colour of that pixel. Library functions for multiplication, addition, division and square root should be used and can be provided by ARM.

Finally implement the fractal generator using Verilog, integrate the fractal

generator into a system together with a Mali-GPU, synthesize the fractal generator for FPGA, set the fractal generator up to feed the OpenGLES application with textures and participate in a demo competition held at ARM Norway.

References

(1) Mandelbrot, B.B. (1982). The Fractal Geometry of Nature. W.H. Freeman and Company.. ISBN 0-7167-1186-9.
(2) Briggs, John (1992). Fractals:The Patterns of Chaos. London : Thames and Hudson, 1992.. p. 148. ISBN 0500276935, 0500276935.
(3) Mandelbrot set http://en.wikipedia.org/wiki/Mandelbrot_set

Supervisors:

Per Gunnar Kjeldsberg, NTNU
Øystein Gjermundnes, ARM Norway AS

Abstract

The *Mandelbrot set*, shown on the front page of this report, is perhaps the most well-known example of a *fractal*. *Fractals* is a certain family of shapes with a very distinctive, interesting shape. The term was coined by Benoit B. Mandelbrot, for whom the Mandelbrot set is named after. The Mandelbrot set and other fractals are traditionally used for aesthetic purposes, such as in art, clothing, computer games, etc. However, there are also several practical applications for fractals, such as image compression[3].

The Mandelbrot set is infinitely complex[9], making it desirable to generate images of arbitrary sections of the set. Several software programs that generate such images exists, but due to the computationally expensive nature of this task, these implementations are typically very slow, even on modern computers. However, the problem can be shown to be highly parallelizable, suggesting that a hardware implementation of such as generator should be able to generate smooth real-time zoom animations, unlike existing software implementations.

A hardware fractal generator for the Mandelbrot set has been designed and implemented in Verilog-2001. The design is very scalable, having a parameter specifying the number of *fractal point generators* (cores) the synthesis tool should implement. Furthermore, it is designed so that the floating point units in the cores are utilized nearly 100% of the time under normal operation. The design was tested on a Xilinx Virtex-6 FPGA with up to 16 cores, and it was shown that the design was faster than a reference software solution running on a desktop computer when the number of cores was set to 2 or more.

Additionally, a simplified Mandelbrot set algorithm is proposed and studied experimentally. In the simplified algorithm, the break condition in the algorithm loop is $(|z_{re}| > 2)||(|z_{im}| > 2)$ as opposed to the standard $|z| > 2$. The images produced using the simplified algorithm was judged to be nearly indistinguishable from those produced with the standard algorithm, and therefore preferred as it is easier to implement.

Finally some future work is proposed. The integration of the fractal generator with the Mali-400 GPU originally planned as part of this thesis is left as future work. It is also suggested to consider designing a custom fixed-point format for use internally in the fractal generator, as the standard *binary32* floating-point format (FP32) is shown to be badly suited for this application.

Preface

The original goal for the work that was carried out as part of this thesis was to integrate the designed hardware fractal generator with the Mali-400 graphics processor from ARM and assemble a working demo on an embedded platform. This demo was to use the designed fractal demo software (presented in Chapter 6) to demonstrate the system.

However, it was decided that this was a complex task, requiring insight in the inner workings of parts of the Mali-400 core, and that time was insufficient. The designed hardware fractal generator is still synthesized for the FPGA planned for this aforementioned demo, and its performance is evaluated and compared with a software implementation. The final integration with the Mali-400 is left for future work.

Per Gunnar Kjeldsberg (Department of Electronics and Telecommunications, NTNU, Trondheim) and Øystein Gjermundnes (ARM Norway AS, Trondheim) supervised the project. Their feedback throughout the semester has been greatly appreciated.

Contents

1 Introduction

The procedures involved in the generation of *fractal* images such as the one on the front page of this report are typically complex and computationally expensive. Several platforms for doing this exists, such as the Windows software *Ultra Fractal* by Frederik Slijkerman, which can be used to create fractal zoom animations. However, it is slow and far from able to generate real-time animations. To ease this limitation, this particular software has a feature for distributing the fractal calculations over several computers to accelerate rendering of animations and stills.[20]

Evidently, it would be desirable to speed up the process by doing the fractal calculations in hardware as opposed to in software. It so happens that the pixels in renderings of the *Mandelbrot set* are independent and processed separately[23], implying that the process of generating these renderings is easily parallelizable and therefore a strong candidate for hardware implementation[25].

Although the traditional use of fractal images are non-practical (i.e., used for aesthetic purposes like art and fashion), recent advances have uncovered several useful applications for fractals. For example, there is a lossy image compression method known as *fractal compression* which is based on fractals.[3] Another example is the generation of patterns for camouflage clothing, such as the MARPAT pattern used by the United States Marine Corps, which was produced using fractal equations[24].

This first part of this thesis is background. Chapter 2 introduces fractals and the *Mandelbrot set*, Chapter 3 gives an introduction of 3D computer graphics and the *OpenGL ES* API, then finally Chapter 4 talks about fixed and floating-point representation of real numbers.

The remaining part of the thesis is own work. Chapter 5 introduces the fractal generator. It gives and overview of how it should work, how it is configured, and some high-level decisions and ideas that have been considered and applied. It will build on the background from Chapter 2 and 3.

Chapter 6 will present the OpenGL ES 2.0 demo and software fractal generator that was developed to gain some insight in design decisions for the hardware implementation and eventually demonstrate the function of the hardware fractal generator. A performance analysis of the software implementation is presented.

Then, Chapter 7 will discuss how to represent coordinates in the Mandelbrot set, the implications of using a floating-point format and some propositions for future work. This chapter builds on the number representation theory from Chapter 4.

Chapter 8 suggests an optimization of the Mandelbrot algorithm presented in Chapter 2, and investigates the implications of this technique. The fractal demo from Chapter 6 is used as part of this investigation.

Finally Chapter 9 will present the implementation of the hardware fractal generator and the design process leading to it. Synthesis results and performance analysis is presented and compared with the software implementation.

The final chapter will present a conclusion of the whole project and suggest some future work on the topic.

The main contribution from this thesis is the design of the hardware fractal generator, which is an elegant and highly scalable design shown to be superior to a reference software model when implemented on an Xilinx Virtex-6 FPGA.

Additionally, a simplified algorithm for generating Mandelbrot set renderings is proposed and shown to produce results nearly indistinguishable from those of the original algorithm. Ideas for future work is also proposed and discussed.

2 Fractals

A *fractal* is a family of shapes that have an "irregular" and "fractured" appearance. The term was first coined by Benoit B. Mandelbrot (1924-2010) in 1975.[11] There is no precise definition of the word. In his book *The Fractal Geometry of Nature* [17], Mandelbrot writes:

> I coined *fractal* from the Latin adjective *fractus*. The corresponding Latin verb *frangere* means "to break:" to create irregular fragments. It is therefore sensible—and how appropriate for our needs!—that, in addition to "fragmented" (as in *fraction* or *refraction*), *fractus* should also mean "irregular," both meanings being preserved in *fragment*.

Many sources state that fractals are structures with a property known as *self-similarity* [23, 12] (A self-similar object is an object where one or more parts of the object has the same shape as the object as a whole, as illustrated in Figure 1.) However, this definition is inprecise at best. Although some fractals do have this property, there are fractals that are not self-similar.[11, p.113] Even the *Mandelbrot set* is not strictly self-similar.[4]. Furthermore, there are self-similar objects that are not considered fractals, such as a line segment.

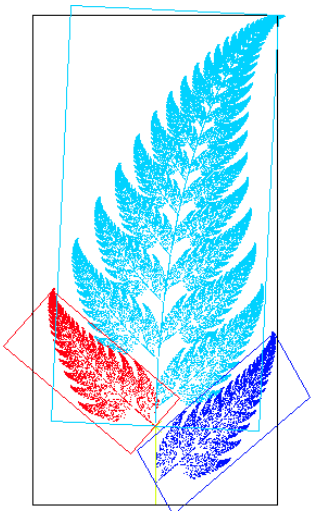


Figure 1: An illustration of the concept of *self-similarity*.

Although there exists more technical definitions that are somewhat more precise [11, p.ix], we will leave this discussion and focus on established shapes (*fractal sets*) that are considered part of the fractal-family, and do have precise, mathematical definitions. Specifically, the *Mandelbrot set*.

2.1 The Mandelbrot set

The Mandelbrot set, M, is a fractal set named after Benoit B. Mandelbrot by A. Douady and J. H. Hubbard in 1984.[2] It is formally defined as the set of complex numbers $c \in \mathbb{C}$ for which the iterative function $z_0 = 0$, $z_{n+1} = z_n^2 + c$ remains bounded as

$n \to \infty$.[4] The set is *connected* [2] (there are no "islands"), its boundary is generally considered a fractal. The Mandelbrot set is plotted in Figure 2, where the black points are in the set and the white are not.

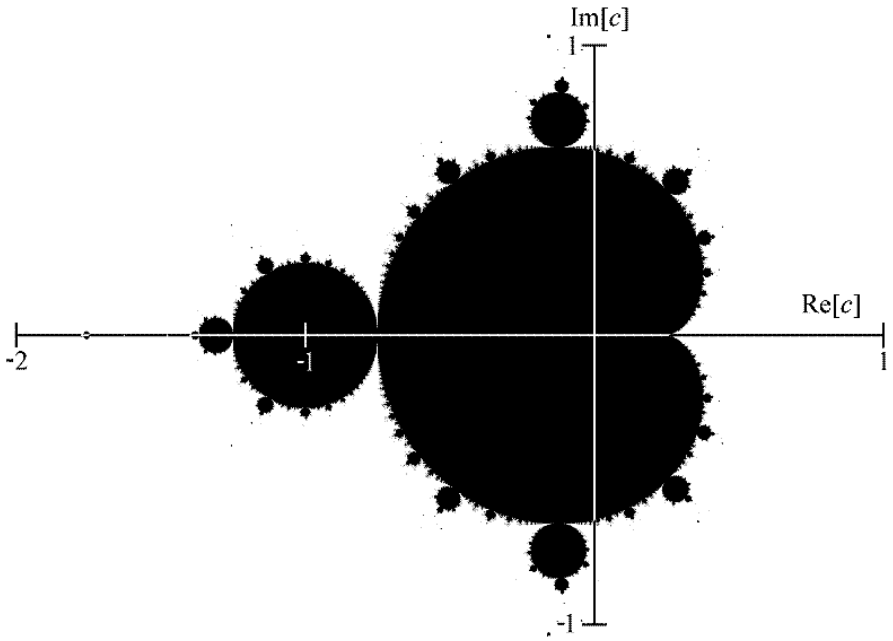


Figure 2: The Mandelbrot set (dark area), plotted in the complex plane.

Despite its simple definition, the Mandelbrot set has a very complex appearance. Zooming in on the edges of the black portion of Figure 2 will reveal some highly unusual graphical structures, two of which are depicted in Figure 3 and 4. The nature of the different colors will be explained shortly.

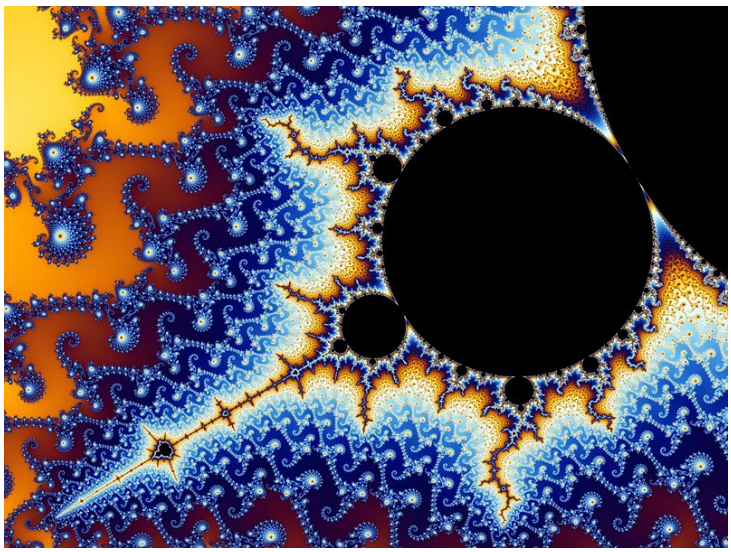


Figure 3: Colored section of the Mandelbrot set, centered at (0.743644786, 0.1318252536) with a diameter of 0.0000029336 . Copyright ©Wolfgang Beyer.

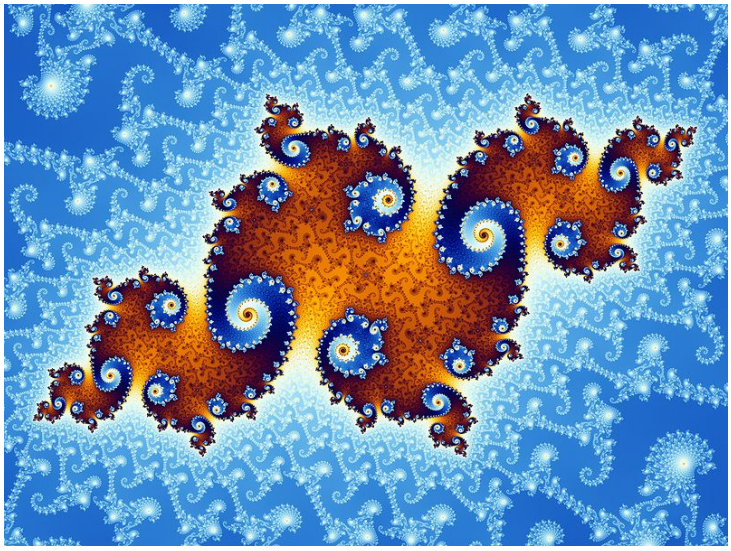


Figure 4: Colored section of the Mandelbrot set, centered at (0.743643887037151, 0.131825904205330) with a diameter of 0.000000000051299. Copyright ©Wolfgang Beyer.

2.1.1 Computing the Mandelbrot set

The task of generating the Mandelbrot set comes down to creating an algorithm that determines whether a complex number, c, is in the Mandelbrot set or not. To generate an image like shown in Figure 2, boundaries of the coordinate system are chosen (in the example, (-2, -1) and (1, 1)) and a resolution (i.e., number of pixels in the image.) The algorithm will then be executed for every pixel in the image.

The definition of the Mandelbrot set suggests the algorithm depicted in Listing 1. If the procedure returns N_ITERATIONS, c is in the Mandelbrot set. Note that this is pseudocode and assumes a programming language with a data type capable of representing complex numbers.

```
1  int mbrot()
2  {
3    z = 0;
4    for (n = 0; n < N_ITERATIONS; n++) {
5      z = z^2 + c;
6      if (abs(z) > BREAK_VAL)
7        break; // c is not in the Mandelbrot set.
8    }
9    return n;
10 }
```

Listing 1: Pseudocode for determining whether a complex number, c, is in the Mandelbrot set

According to the mathematical definition, both N_ITERATIONS and BREAK_VAL will have to be set to infinity for this to work. The problem with doing this, of course, is that then the algorithm would never finish. Some points can be determined

analytically (it can, for example, easily be shown that $c = 1$ will cause z to tend to infinity), but no general method exists. Thus, numerical approximations must be employed; more specifically the algorithm in Listing 1 must be used with the value of N_ITERATIONS and BREAK_VAL restricted.

Fortunately, it can be shown that BREAK_VAL=2 is sufficient. If z at any point exceeds 2, the series is guaranteed to diverge.[4, p. 81] Thus, this is not even an approximation, and so the definition itself can be simplified. It follows that the Mandelbrot set is contained entirely in the disk with radius 2 around the origin of the complex coordinate system.

Unfortunately, there is no corresponding hard limit on the number of iterations of the algorithm. (N_ITERATIONS in Listing 1.) Fractal generators typically zoom in on specific points of interest (e.g., sections with aesthetic shapes), and these areas are typically located somewhere on the border between the black and white area of the graph visible in Figure 2. It can be said generally, from experiments done on the software implementation presented later in this thesis, that the closer one gets to the edge of the set, the more iterations are needed. While the set visible in Figure 2 could be generated with just ≈ 10 iterations for each pixel, some images require thousands of iterations.

2.1.2 Low-level implementation

Typical low-level programming languages (such as C) do not have data types capable of representing complex numbers, although there are extensions/libraries that add support for it. Either way, the hardware module will have to be designed in a more low-level fashion, and so it is beneficial to design the software implementation without utilizing any complex data type.

The heart of the Mandelbrot algorithm presented in Listing 1 on the preceding page is the following equation:

$$z_{n+1} = z_n^2 + c \tag{1a}$$

$$= (z_{n,re} + iz_{n,im})^2 + c_{re} + ic_{im} \tag{1b}$$

To separate the real and imaginary parts, the exponentiation is expanded:

$$z_{n+1} = z_{n,re}^2 + i2 \cdot z_{n,re} \cdot z_{n,im} - z_{n,im}^2 + c_{re} + ic_{im} \tag{2}$$

and thus

$$z_{n+1,re} = z_{n,re}^2 - z_{n,im}^2 + c_{re} \tag{3a}$$

$$z_{n+1,im} = 2 \cdot z_{n,re} \cdot z_{n,im} + c_{im} \tag{3b}$$

Using this, a modified implementation is designed in Section 6.3. These calculations are relatively trivial and can be found elsewhere such as on Wikipedia[23].

2.1.3 Adding color

Benoit B. Mandelbrot's original renderings of the Mandelbrot set were in black and white like in Figure 2.[4, p. 82] Like described so far in this chapter, he colored points

that belonged in the Mandelbrot set (the interior) black, and others (the exterior) white.

More recent renderings are more colorful. This is achieved by coloring points in the exterior based on the number of iterations needed for z to exceed some fixed value ≥ 2 (Usually, 2). Typically, some arbitrary palette that maps the iteration count (n) to a color is chosen. To make renderings more aesthetically pleasing, this color palette should be a *gradient*. This concept is illustrated in Figure 5. Note that the points in the interior are still colored black (typically.)

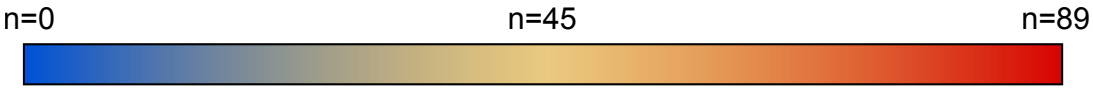


Figure 5: An example of a color palette, where each color is assigned to an iteration count (n). The maximum iteration count is here set to 90.

Why it makes sense to do this goes beyond the scope of this thesis. However, based on experiments, this method works well aesthetically — it produces smooth height curves in the fractal demo presented later. For the mathematics behind, see [4].

3 Landscape generation in OpenGL ES 2.0

OpenGL ES is a subset of the well-known *OpenGL* graphics library, designed for embedded systems. This chapter will give a general introduction to computer graphics rendering with OpenGL, and more specifically, landscape generation.

3.1 Introduction to 3D computer graphics

The basic idea of traditional 3D computer graphics is to take a number of points in a virtual three-dimensional space and project them onto a two-dimensional space, for viewing on two-dimensional surfaces such as a computer screen. This process is called *orthographic projection*. (See Figure 6.) The points in space are in computer graphics terms called *vertices* (singular *vertex*).[19, chap. 2]

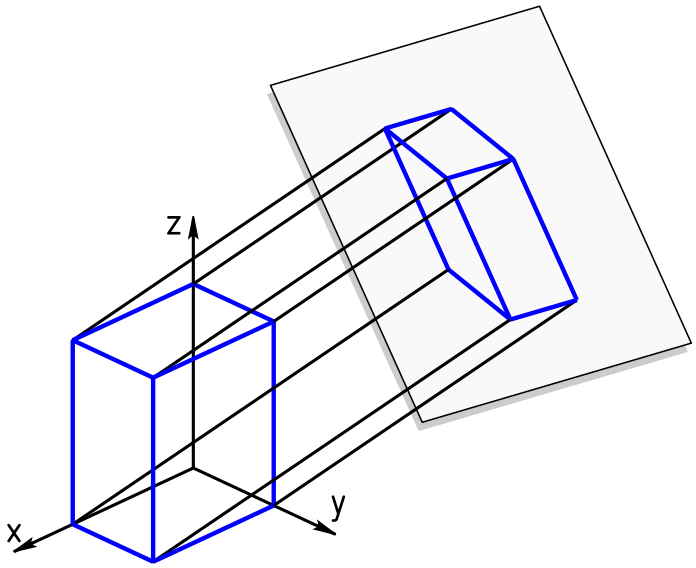


Figure 6: An orthographic projection of a cube onto a two-dimensional surface.

OpenGL handles this task using simple API-calls[1]. Furthermore, it allows for a number of *transformations* to be performed on the vertices prior to projection. For example, the OpenGL *glRotate()* API-call will rotate all the vertices that have been specified with prior API-calls around some specified point in space. (In the abstract three-dimensional space.) This particular transformation involves multiplying each vertex with a *rotation matrix* [19, chap. 3], which can be a very computational expensive task. Therefore, most modern implementations of OpenGL are accelerated by hardware designed specifically for performing these matrix multiplications; a *graphics processing unit* (GPU).

The vertices can be considered the "corners" of the geometric figures one desires to model. By drawing lines between the vertices after projecting them on the two-

[1] An *API-call* (application programming interface) is the calling of a function in a software library that performs some specific task.

dimensional screen (a task OpenGL can perform as well), so-called *wireframe* models appear. Figure 7 shows an example of a wireframe model. Further processing such as coloring, texturing[2] and lightning is possible, consider the *OpenGL Programming Guide* by Dave Shreider [19] for more information.

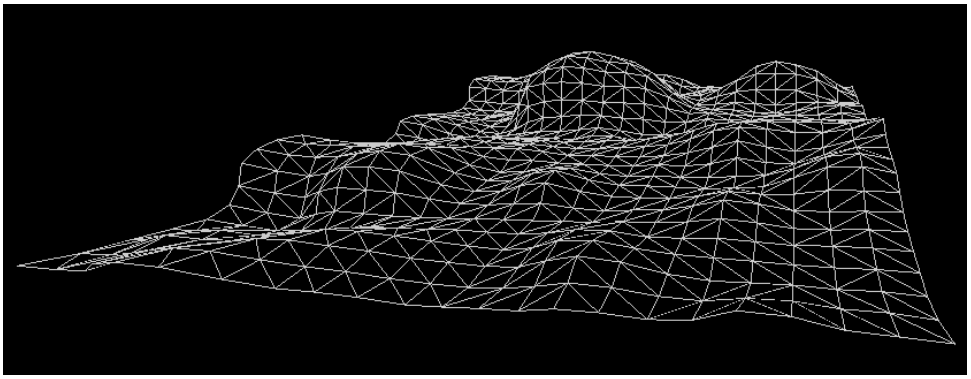


Figure 7: A wireframe model of a landscape.

3.2 Specifying vertices

In OpenGL ES, vertices must be specified (drawn) with the *glDrawArrays()* API-call. (Standard OpenGL has additional ways of drawing vertices.) As the name suggests, the coordinates of all the vertices must be packed into a C array, which are then drawn with glDrawArrays(). Figure 8 illustrates an example of such an array.

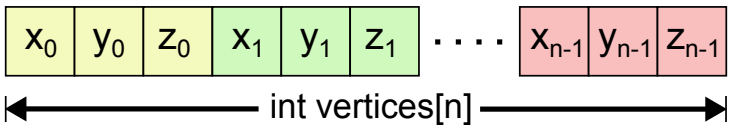


Figure 8: Illustration of a vertex array.

glDrawArrays() accepts a *mode* parameter that determines how the vertices in the array will be rendered. It can take the following [19, chap. 2] values:

- GL_POINTS
- GL_LINE_STRIP
- GL_LINE_LOOP
- GL_LINES
- GL_TRIANGLE_STRIP
- GL_TRIANGLE_FAN

[2]Applying images to the surface of geometric objects.

- GL_TRIANGLES

For what simple wireframe models are concerned, this parameter will determine how lines are drawn between the vertices after any tranformations and the 2D projection are done. For example, GL_TRIANGLES will cause every set of three vertices to be connected together, while GL_LINE_LOOP will cause every vertex to be connected to the previous one. The final vertex will be connected to the first so that the vertices form a loop, hence the name. This is illustrated with an example in Figure 9.

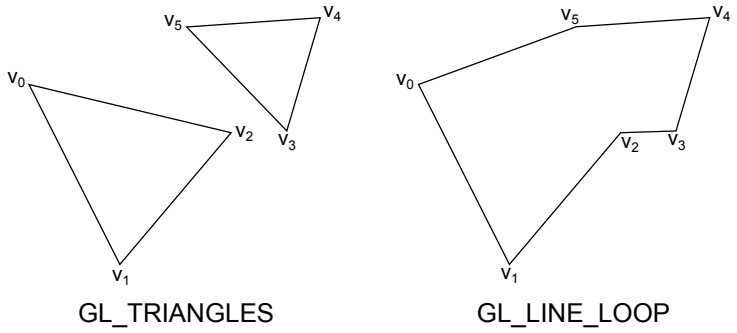


Figure 9: Example rendering of six vertices using two different modes.

Going beyond wireframe models, the mode-parameter also has to do with how models are colored or textured. GL_TRIANGLES will for example define every set of three vertices as a triangle *primitive*, whose interior can be colored, etc.

3.3 Generating landscapes

For various reasons, most objects are in OpenGL modelled with triangles, including landscapes.[18, sec. 3.1] The problem with generating a landscape with triangle primitives is that all the triangles will share some of their vertices with other triangles. In fact, most will share each vertex with as many as five other triangles. This is evident from Figure 7, which was modelled with triangles.

To avoid increasing the amount of vertex data sent to the GPU fivefold, which would happen if GL_TRIANGLES were used, OpenGL/OpenGL ES provides the GL_TRIANGLE_STRIP render mode. This mode is well suited for drawing triangles where each new triangle shares one of the three edges of the previous triangle. To reduce the amount of vertex data, it allows for drawing only one vertex per triangle, except the first one which needs all three.

The concept of triangle strips is illustrated in Figure 10. Here, two triangle strips are drawn; consider it a very coarse-grained landscape. (The height-dimension is not visible.) The orange numbers in circles show the vertex draw order for the lower strip, while the green numbers indicate the order of the upper strip. Note how the middle row of vertices still has to be drawn twice. However, this is a good as it gets without losing the concept of primitives and thus losing the ability to add color to the landscape later. Note that each triangle strip is drawn with a separate call to glDrawArrays().

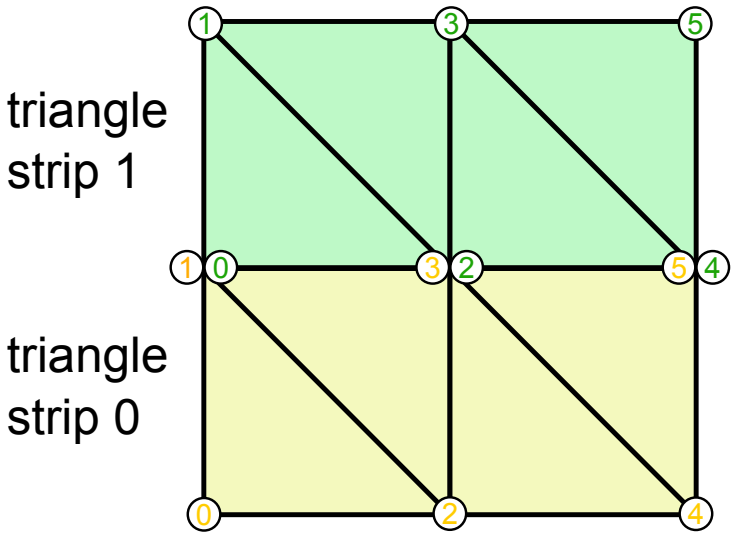


Figure 10: The order in which vertices must be drawn when using GL_TRIANGLE_STRIP. Viewed from above.

4 Representing numbers

In computing, there are generally two main techniques for representing numbers, *fixed point* and *floating point* representation. Both techniques are widely used and have different applications. This chapter will quickly introduce and compare these.

4.1 Fixed-point representation

Fixed point formats are typically used to represent *integers*. However, by defining some fixed decimal point, they can be used to represent decimals[3] as well.

Consider the example of a fixed point data type in Figure 11. In this 8-bit type, four bits are used for representing an integral part, and the other four represent a fractional part. For example, the bitstring 01110011 (as shown in the figure) represents the decimal number $2^2 + 2^1 + 2^0 + 2^{-3} + 2^{-4} = 7.1875$.

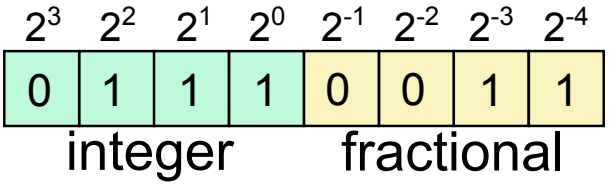


Figure 11: An example of a fixed-point data type representing the number 7.1875

Fixed-point formats can represent negative numbers using a *sign bit*, or more commonly, using *two's complement*.

4.2 Floating-point representation

The other main technique, floating-point representation, is the one more typically used for representing decimals. In the C programming language, the data types *float* and *double* are both used to store floating point numbers.[6]

Unlike fixed-point formats, floating point formats do not have a fixed decimal point; instead some bits are reserved for specifying its location. Hence, floating-point representation can be said to be analogous to *normalized scientific notation*. Figure 12 shows an example of a floating point data type and Equation 4 loosely describes how to interpret the format. Note the implicit 1 to the left of the decimal point. The *IEEE 754* standard presented in Section 4.4 uses this convention. The implicit 1 is only used when the exponent is non-zero, however.

$$(-1)^{\text{signbit}} \times 1.\text{fractionbits} \times 2^{\text{exponentbits}-3} \tag{4}$$

Using this, the bitstring 01011101 shown in the figure represents the decimal number $(-1)^0 \times 2^{5-3} \times (1 + 2^{-1} + 2^{-2} + 2^{-4}) = 7.25$. The reason for the exponent offset (here -3) is that it would otherwise be impossible to represent numbers smaller than 1.

[3] A *decimal* is here defined as a real number composed of both an integral and a fractional part.

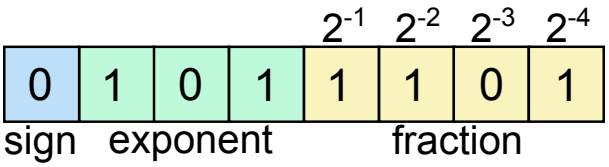


Figure 12: An example of a floating-point data type representing the number 7.25

4.3 Comparing fixed-point and floating-point formats

Generally, fixed-point formats have better precision than floating-point formats, but lower range. This is certainly true for the formats introduced in the last two sections. The bitstring in Figure 12 was chosen to give the best possible approximation of the number represented by the fixed-point format example of Figure 11, and as seen this approximation is not especially good. However, the floating point format can represent a wider range of numbers, as illustrated in Figure 13. The difference would be more radical if the number of exponent bits in the floating-point format were increased, like in more commonly used formats.

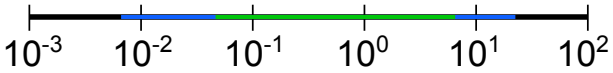


Figure 13: Positive range of the example fixed-point (green) and floating-point (blue) formats

4.3.1 Linear precision of floating-point formats

On a linear scale, the precision of floating-point formats will increase at small magnitudes. Figure 14 illustrates the precision of the simple floating point format presented in this chapter. The bars are the distinct values the format can represent, on a linear scale. Notice the large gap between 0.130 and 0.250; this happens because the implicit 1 is not used anymore when the exponent is at its minimum.

Figure 14: Lower range of the example floating-point format

This is usually a good thing, because the high precision desired at low magnitudes usually is unnecessary at high magnitudes. However, it can cause problems when there is some sort of offset involved — like when doing high precision operations around some large number. (I.e., adding small numbers to larger numbers, which causes rounding errors corresponding to the precision at the magnitude of the largest number.) This is a problem that is encountered in this project, as discussed in detail in Section 7.

4.4 IEEE 754

The *IEEE Standard for Floating-Point Arithmetic* (IEEE 754) [13] is a widely used standard for floating-point formats. For example, the Intel Corporation follows the IEEE 754 standard in all their x86 series of microprocessors.[7]

IEEE 754 defines a number of floating-point formats that occupy different number of bytes, most notably the *binary32* (32-bits, "single precision") and *binary64* (64-bits, "double precision") formats. These normally correspond to the *float* and *double* data types in the C programming language.[6]

Table 1 summarizes the number of bits used for the sign, exponent and fractional parts in binary32 and binary64, respectively.

	binary32	binary64
Sign	1	1
Exponent	8	11
Fractional	23	52
Total	32	64

Table 1: Summary of the most frequently used floating-point formats of IEEE 754.

5 Fractal generator

The main task in this thesis is the development of a Mandelbrot fractal generator in hardware. A typical approach when designing a hardware module like this is to design a software model first, to be regarded as the specification for the hardware module.[10]

This chapter will present an overview of the fractal generator that was designed, including high-level decisions leading to the final software and hardware designs. The software and hardware implementation will then be presented in Chapter 6 and 9, respectively. Both implementations both perform the same task, although the hardware implementation is designed to be much faster than the software model.

There were some iterations in the design process of the system. First a very basic software implementation of the fractal generator was designed with the fractal demo (presented in Chapter 6), then later during the hardware design process the software model was modified gradually to investigate techniques desired for the hardware module.

5.1 Overview

It was decided early in the process that the fractal generator should generate the vertex arrays for landscape generation directly. An alternative would be to just generate some sort of *heightmap* (a texture) and assemble the vertex arrays in software or in the vertex shader in the GPU. However, even the latter would yield higher bandwidth usage between the GPU and graphics memory than when generating the whole arrays directly in the fractal generator. The precise reasons for this goes beyond the scope of this thesis; the decision was taken in collaboration with ARM-employees with technical insight in the Mali GPU.

It is a natural approach to divide the design into a module (or C function) that runs the actual Mandelbrot algorithm (i.e., calculates iteration counts from fractal coordinates) and a surrounding module that generates the vertex array. This approach is similar to other (software) implementations.[5] The latter will then interface the aforementioned Mandelbrot module (or call the C function.) From here on, these two modules will be referred to as the *fractal point generator* (FPG) and the *vertex array generator* (VAG).

The basic function of the fractal generator is illustrated in Figure 15. As discussed in Chapter 6 the landscape is drawn with a number of triangle strips, each in the form of a vertex array. Note that the hardware implementation will be able to interface several FPGs, while the software implementation can only implement one, provided that the software is written for a single-core CPU.

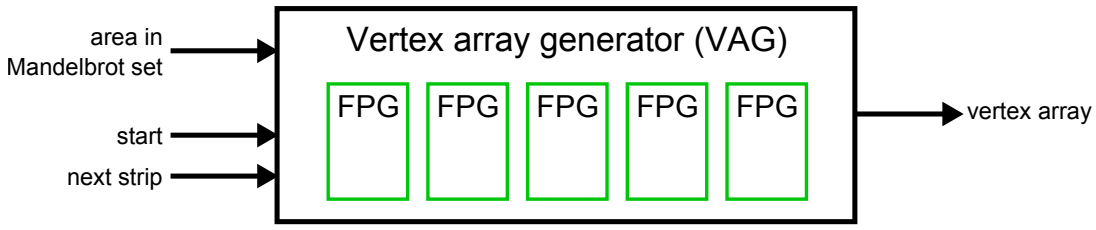


Figure 15: Basic function of the fractal generator. An area in the Mandelbrot set is specified using some parameters, then each triangle strip vertex array is retrieved in order.

5.2 Configuration parameters

The idea is that the fractal generator is configured once for each frame, to render a different section of the Mandelbrot set. The configuration parameters must uniquely specify this section. For example, one might want to render the section within the green square in Figure 16.

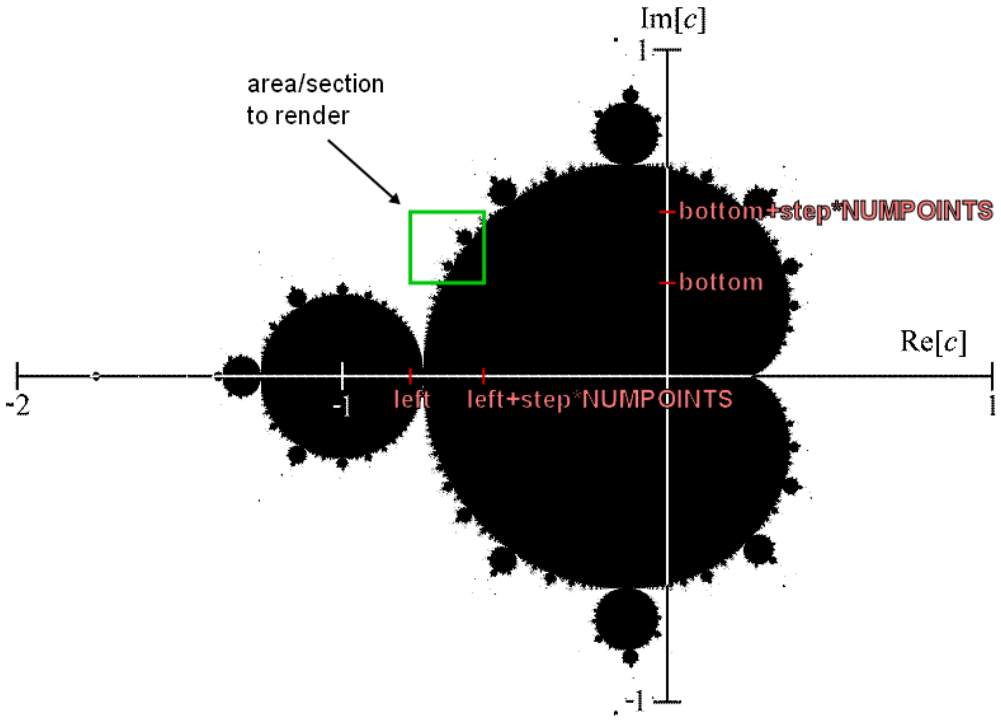


Figure 16: Parameters (in red) for rendering the section of the Mandelbrot set within the green square

The chosen setup is to use three configuration parameters. They are the two fractal coordinates of the bottom left corner of the section, as well as the distance (in fractal coordinates) between vertices (heightpoints) in the landscape — the step size. Thus the parameters determine the location in the Mandelbrot set and the zoom-level of the fractal to be rendered.

The parameters were chosen to simplify the hardware implementation. More specifically, simplify the calculation of the sequence of fractal points sent to the fractal point generator by the vertex array generator. An alternative would be to send the center coordinates and width of the coordinate system, but this would be less efficient. In fact, these are the parameters used by the fractal demo, which converts these to the bottom left coordinates and step size when configuring the fractal generator.

A final, static parameter is the number of vertices in the landscape. This parameter cannot be modified at run-time; it is implemented as the C macro definition NUMPOINTS in the fractal demo. The number of vertices in the landscape will always be NUMPOINTS^2, forming a static grid of (x, y) vertex coordinates. The z coordinates are the only coordinates that vary between frames, each representing the number of Mandelbrot iterations for the corresponding fractal point. See Section 6.3.2.

The interpretation of the parameters are indicated in red on Figure 16.

5.3 Vertex array format, vertex and fractal coordinates

The vertex arrays use fixed-point integers only. All coordinates are represented with the *GLshort* data type, which is a 16-bit signed integer format.[19, chap. 1] This decision was made to reduce the size of the vertex arrays; the alternative would be to use the 32-bit floating-point format *GLfloat*, spending twice the bandwidth between the CPU and GPU.

To make this work, the fractal generator must translate x and y vertex coordinates (which are integers) into floating-point fractal coordinates, based on how the fractal generator is configured. For example, the (x, y) coordinates (1, 2) are translated into (left+step_size, bottom+2step_size)[4] and the iteration count for this fractal point is then used as the z-coordinate of the vertex. Thus, if this iteration count is 10, the vertex coordinates are (1, 2, 10).

From here on, the terms *vertex coordinates* and *fractal coordinates* will be used to refer to the integer coordinates of the vertex arrays and the floating-point coordinates in the Mandelbrot set, respectively.

5.4 Storing z-coordinates

An important issue in the design of the fractal generator is how the z vertex coordinates are stored between triangle strips. As discussed in Section 3.3, each triangle strip in the landscape share half their vertices with the adjacent strips. It would be very expensive to calculate all the z-coordinates (Mandelbrot iteration counts) of the vertices twice, which is what would happen in a naively designed implementation.

To avoid generating the same fractal points twice, the z-coordinates of the lower vertex points can be reused from the previous strip. Due to the nature of the triangle strip method of drawing landscapes, one method of achieving this is by storing all the z coordinates each strip, and then shift all the values to the left before the next strip,

[4]See Section 5.2 for an explanation of what *left*, *bottom* and *step_size* represent.

as illustrated in Figure 17. Every second value, namely those corresponding to the upper vertices of the strip, are then updated with new Mandelbrot iteration counts. See Figure 10 on page 12 for insight in why this works. For the first (bottommost) triangle strip, all the values must be generated.

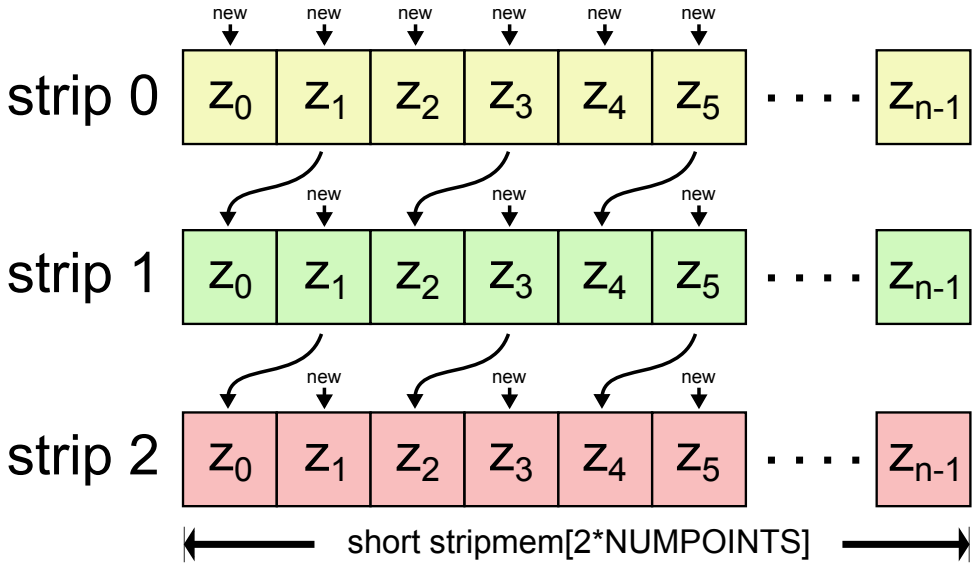


Figure 17: Updating strip memory between triangle strips

Another method that accomplishes the same thing is to have two buffers, each with the capacity to hold one row of z-coordinates (half a strip.) For the first strip both buffers are filled with the lower and upper coordinates, respectively. For the second strip only the first buffer is filled with the upper coordinates of that strip, for the third strip only the second buffer is filled with the upper coordinates, and so on. This is illustrated in Figure 18. Green indicates the buffer that holds the lower row for the corresponding strip, yellow the upper row.

The second method is probably the better one for the hardware implementation, and the method of choice for this project. When a strip is calculated and ready to be sent out of the module serially, it is easy to make a design that alternates between the two buffers. The first method is a strong candidate as well, because it is probably easy to implement the aforementioned shifting operation in hardware. However, it might be desirable to use a type of memory where this is not the case (such as SDRAM.)

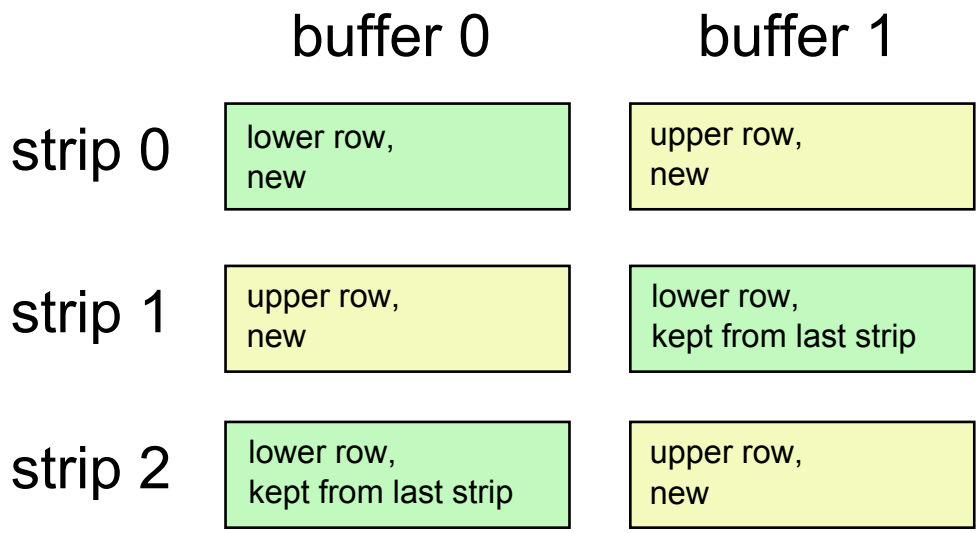


Figure 18: Second method of updating strip memory between triangle strips

6 Fractal demo and software implementation

A software implementation of the fractal generator and an OpenGL ES 2.0 demo application was designed first to gain insight in design matters such as what kind of data structure the hardware module should produce. The end purpose of the demo application was to demonstrate the hardware implementation of the fractal generator, but this will have to be reserved for future work. The designed demo will use the software fractal generator only. This chapter will introduce and describe the fractal demo and the software implementation of the fractal generator.

6.1 Overview

The thesis description states to *create an OpenGL ES 2.0 program that creates a animated 3D landscape where the height and the color of the landscape at any given point is determined by the color of a texel in a texture.* The idea is that this texture is a fractal image that was produced beforehand, by a fractal generator.

The fractal generator is described in Chapter 5, a quick recap is provided here. The generator will not produce a texture, but rather a set of vertex arrays that represent the landscape as triangle strips. See Chapter 3. The x and y coordinates of all the vertices are constant between frames. The z coordinates are the iteration counts of the Mandelbrot algorithm. The generator is configurable to specify what section/area of the Mandelbrot set to render, as well as the number of vertices (triangles) in the landscape. The number of vertices is not configurable at run-time.

Animated 3D landscape is interpreted as drawing the Mandelbrot set as a 3D-landscape, then create an animation that zooms into the fractal.

6.2 Fractal demo design

The fractal demo was written in the C programming language with the *OpenGL ES 2.0 Emulator* from ARM.[14] This is essentially a library that translates OpenGL ES 2.0 API calls to desktop OpenGL API calls, allowing applications destined for Mali-accelerated platforms to be written and run on a normal desktop computer running Microsoft Windows or Linux. The emulator also provides an *EGL* library, which is an interface between the OpenGL ES 2.0 API and the underlying platform window system. That is, it provides an easy way to open a window in the operating system and have the result of OpenGL ES draw calls appear in it.

Some of the code in the demo was taken from the *cube*-example that comes with the OpenGL ES 2.0 Emulator. In particular, those portions that are concerned with opening the window and setting up the drawing surface. The full source code of the fractal demo can be found in Appendix A, where the portions of code that was borrowed from the *cube* example are clearly marked. The screenshot in Figure 19 shows the demo in action.

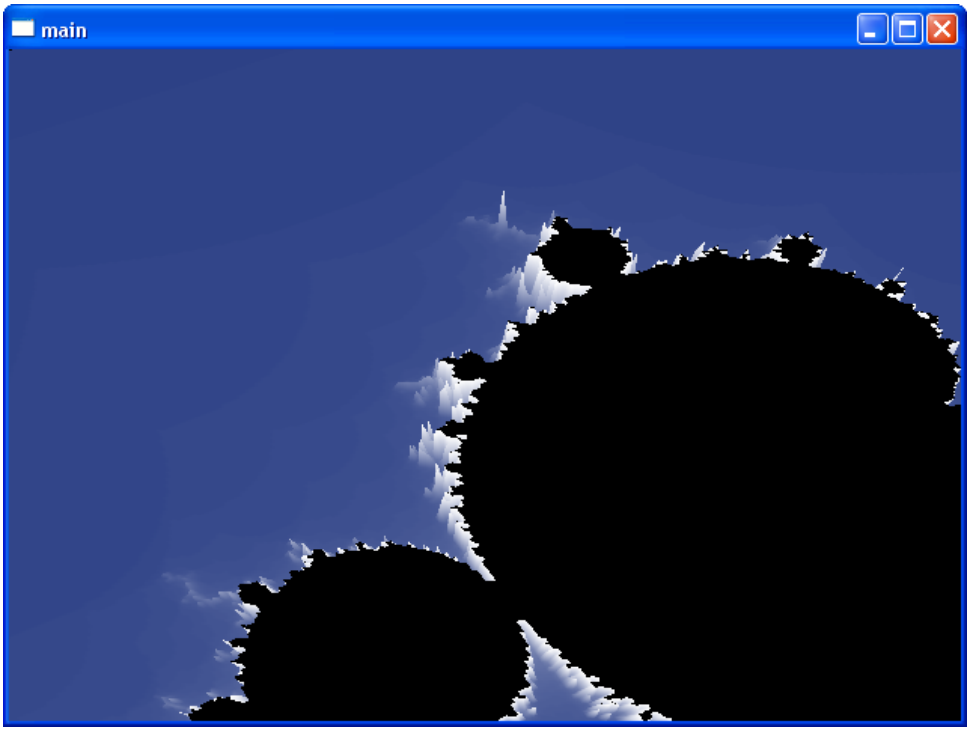


Figure 19: A screenshot of the fractal demo application in action

The demo draws a landscape using the vertex arrays produced by the fractal generator directly, using one draw call per vertex array/triangle strip. The vertices are rotated so that the landscape is viewed from an angle rather than from a bird's-eye perspective. The vertices are then scaled so that the edges of the landscape fall outside the edges of the drawing surface/window. Figure 20 shows a screenshot of the demo without this final scaling.

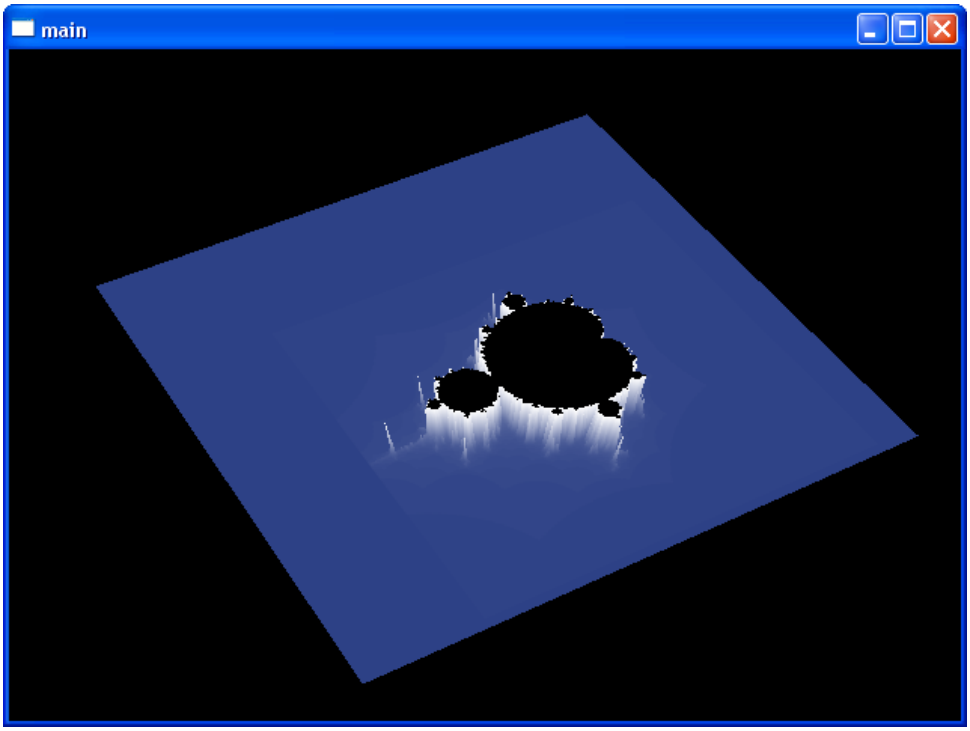


Figure 20: Fractal demo application without final scaling

The demo as it is in the source code in Appendix A is not interactive, although it could easily be modified to be. Instead it will simply zoom in on a predefined point in the Mandelbrot set (the *fractal center*), and then at some point return to the original zoom level and repeat.

In the shown screenshots the landscape is drawn using 400 by 400 fractal points/vertices. This corresponds to $399 \cdot 2 \cdot 399 = 318,402$ triangles. A new set of vertex arrays are generated for each frame, where the step size is reduced and the bottom left coordinates are adjusted to keep the desired fractal center in the center of the rendering. This creates a zoom-animation.

A video of the demo in action is attached with this report. The video was created by recording the demo application real-time using the VLC media player from VideoLAN.[21] The video shows the application operating somewhat slower than normally, because the recording process uses resources that would otherwise be available to the fractal demo process.

6.2.1 Coloring

As discussed in Section 2.1.3, coloring of the landscape is done with a color gradient, where the iteration count decides what color (i.e., what position in the gradient) that fractal point will be given. In OpenGL, coloring of pixels is done by a *fragment shader*, which is a small program that is executed for each pixel in the finished rendering. The fragment shader for the fractal demo is reproduced in Listing 2.

```
1  precision lowp float;
2  varying float v_iterations;
3
4  void main()
5  {
6    vec3 v_color;
7
8    const vec3 c1 = vec3(0.165, 0.244, 0.518); // Light blue.
9    const vec3 c2 = vec3(1.0, 1.0, 1.0); // White.
10
11   // Assuming max iteration count of 80.
12   if (v_iterations > 79.5)
13     v_color = vec3(0.0, 0.0, 0.0);
14   else
15     v_color = mix(c1, c2, v_iterations / 80.0);
16
17   gl_FragColor = vec4(v_color, 1.0);
18 }
```

Listing 2: Fragment shader for fractal demo

A fragment shader takes one or more type of parameters called *varyings*, which are interpolated versions of vertex parameters. (Weighted averages of the vertices nearest to the current pixel.) In this case, the only varying is *v_iterations*, which is the number of iterations in the Mandelbrot algorithm for each vertex. For example, a pixel halfway between two vertices with iteration counts 50 and 60 will have the value of the v_iterations varying set to 55. This is discussed extensively in the *OpenGL ES 2.0 Programming Guide* [1].

In this case, pixels near vertices corresponding to fractal coordinates with iteration counts of 80 are colored black. Other pixels are colored using a simple blue to white gradient, realized using the `mix()` function in the shader language.

6.3 Software fractal generator

The software implementation of the fractal generator was written to function as similar as possible to the hardware implementation; each C function corresponds to the functionality of a hardware module. As stated earlier there were some iterations between the design of the software and hardware module. The final software model is contained entirely in *fractal.c*, which can be found in Listing 7 in Appendix B.

6.3.1 Fractal point generator (FPG)

The fractal point generator is a low-level implementation based on Equation 3 on page 6. This function is reproduced below, in Listing 3. The boundary condition check used here (line 11-12) is the simplified method, which will be introduced in Chapter 8.

```
1  int getFractPoint(float re, float im)
2  // Get the number of iterations (n) before |z| exceeds 2.
3  {
4    int n;
5    float z_re = 0.0f, z_im = 0.0f;
6    for (n = 1; n < 80; n++) {
7      float z_re_old = z_re;
8      z_re = z_re*z_re - z_im*z_im + re;
9      z_im = 2.0f * z_re_old * z_im + im;
10
11     if ((z_re >= 2.0f) || (z_re <= -2.0f) || (z_im >= 2.0f) || (z_im
           <= -2.0f))
12       break;
13   }
14   return n;
15 }
```

Listing 3: C model of fractal generator

6.3.2 Vertex array generator (VAG)

Although the fractal point generator is the actual workhorse in the fractal generator, the fractal points must be assembled into an array structure that can be sent directly to the graphics processor for drawing. This is what the /textttgetNextLandscapeStrip() function does, corresponding to a vertex array generator hardware module.

The vertex array generator produces a vertex array containing a single triangle strip of the landscape. After `initLandscape()` is called with configuration parameters, `getNextLandscapeStrip()` will return the bottommost triangle strip of the landscape. Subsequent calls will return the whole landscape strip by strip, upwards from the bottommost strip.

The `getNextLandscapeStrip()`-function consists of two loops. The first loop fills an array with iteration counts by calling `getFractPoint()` repeatedly. Since there are only two different y-coordinates per strip and as a result only two different imaginary

parts of the fractal coordinates, these two are calculated before the loop is started. The first real part (*re*) is always the *left* configuration parameter because all triangle strips start on the left side of the grid. Subsequent real parts are calculated by adding *step* to *re* once for every loop iteration. This way a single floating point adder should be sufficient when the hardware module is designed.

The second loop constructs the actual vertex array using values from the array generated by the first loop. The purpose of splitting up these tasks is that it more closely resembles how it will be done in hardware.

6.3.3 Performance

To be able to compare the software solution to the hardware solution presented later, some performance investigations were performed. The execution time required to generate one fractal landscape varies greatly, depending on the section of the Mandelbrot set. However, the worst-case scenario is when all the fractal points in the section are in the Mandelbrot set. (i.e., have an iteration count of 80.) Thus this is the situation that will be considered.

The measurements were carried out using the `clock()` function of *time.h* on a desktop computer with a Pentium 4 3.4 GHz CPU. The used method is shown below in Listing 4, as described in the Microsoft MSDN function reference.[8].

```
1  #include <time.h>
2
3  void display()
4  {
5    clock_t start;
6    double duration;
7    static int framec = 0;
8
9    // ...
10
11   start = clock();
12   // ... (Generate fractal)
13   duration = (double)(clock() - start) / CLOCKS_PER_SEC;
14   printf("Frame %d took %2.3f seconds to generate.\n", framec++,
       duration);
15
16   // ...
17 }
```

Listing 4: Performance measurement method

Experimental results show that the execution time for a worst-case scenario fractal is between **0.297** and **0.313** seconds on the used computer. This corresponds to a frames-per-second (FPS) rate of around 3.2 - 3.3.

6.4 Discussion

There are a few issues with the fractal demo, related to shortcomings of the fractal generator. The main issue is that the zoom animation appears very "noisy" between frames. For example, some of the spikes in the landscape (See examples in Figure 20) will suddenly disappear in one frame and reappear in the next. This is not very aesthetically pleasing. The reason why it is happening is that there are lots of very narrow black areas in the Mandelbrot set, and it will be somewhat random whether the vertex points will hit these spots, forming a spike in the landscape, or not.

The noise issue can be fixed using certain blending and anti-aliasing techniques.[5] However, these are computationally expensive methods and hard to implement in hardware. Due to limited time it was therefore decided not to do this.

Another issue is that the frame rate of the animation is very varying throughout the animation, and overall not good enough. Typically, the frame rate will be high at first, when most of the fractal points in the landscape have a low iteration count. (I.e., more blue and less white and black.) Later on most of the fractal points will have a high iteration count (when zooming in on the boundary between points that are in the Mandelbrot set and not.) This simply happens because calculation the iteration counts is a computationally expensive task; this problem should go away as soon as the task of generating the fractals is moved to hardware.

7 Representation of fractal coordinates

Choosing the data formats in which fractal coordinates are represented is a very significant decision for this project. Several issues will arise when rendering smaller areas[5] of the Mandelbrot set if the range of the format is too low.

The range and precision of the format is therefore the limiting factor of how far into the Mandelbrot set the system can zoom. This section will present and discuss the background for this and give an overview of the available options.

7.1 Rounding errors

The way in which the fractal generator is implemented is not optimal with regards to utilization of the available range in the floating point format. What will eventually happen is that the step size parameter (*step*) will become zero (or sufficiently smaller than the coordinates of the fractal center, see next two paragraphs), resulting in that the whole landscape will have the same color and height, namely that of the bottom left point.

Before this breaks the system completely, however, another effect will compromise the aesthetic appearance of the zoom animation. When the step size becomes very low, the animation will appear "shaking" — the center of the landscape (the *fractal center*, the point being zoomed in on) will shift between subsequent frames.

The reason for this effect and other related irregularities is that the coordinates of the bottom left corner generally will be several magnitudes larger than the step size at some point. (The only exception would be if the fractal center was near the origin, but that would not make much sense, as all the points around the origin are part of the Mandelbrot set.) Adding a low-magnitude number to a high-magnitude number will lead to severe rounding errors with floating point numbers. This is exactly what happens when fractal points are calculated in the software fractal generator. At some point the step size will become so small that adding it to either of the bottom left coordinates makes no difference, and thus the whole landscape will have the same height and color. For insight in why this is happens, see Section 4.3.1.

There is a way to reduce the rounding problem, but it is not a perfect solution. Instead of adding *step* to *bottom* for each strip and to *left* for each column of each strip, the fractal coordinates can be calculated each time using the following equations:

$$\text{im} = \text{bottom} + (y \cdot \text{step}) \tag{5a}$$

$$\text{re} = \text{left} + (x \cdot \text{step}) \tag{5b}$$

where (re, im) is the fractal coordinate corresponding to vertex coordinates (x, y). This will help because the rounding error from adding $(y \cdot \text{step})$ to *im* is not as bad as adding *step* directly. This will fix the shaking problem, but it will lead to another undesired effect, apparent in Figure 21. It is also a more computationally expensive solution.

[5] I.e., fractals with a small *step size*. See Section 5.2.

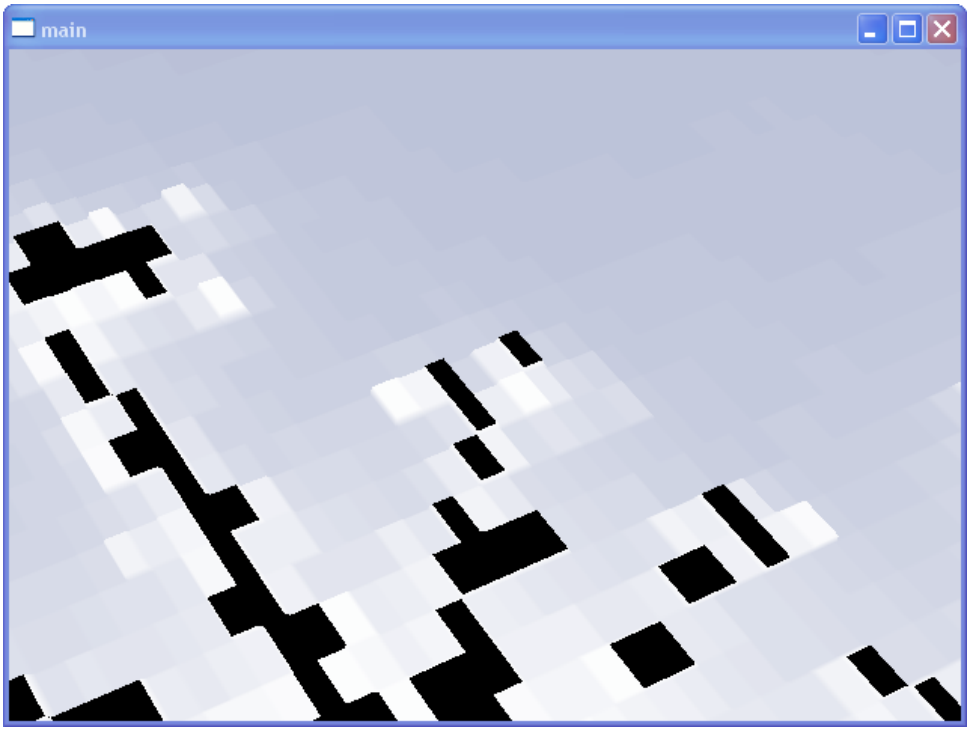


Figure 21: The effect at low step sizes with the alternative fractal coordinate calculation method

This interesting effect occurs because many of the calculations of Equation 5 will yield the same answer due to rounding errors at this step size. What you are looking at is really the resolution/precision of the used floating point format at the magnitude of the fractal center. (The bottom left point, to be precise.) However, this effect becomes appearent quite a bit later than the shaking. In fact, it happens at about the same time as the animation starts to shake due to rounding errors when calculating the bottom left coordinates from the fractal center.

Interestingly, the step size value is nowhere near the smallest value that the floating point format can represent when these rounding errors break the animation. The demo uses the *float* C datatype for fractal coordinates, which is the *binary32* format of IEEE 754 on the used computer and most other systems that implement the C programming language. In Figure 21, the step size is around 4E-9, or close to 2^{-28}. The smallest value that the binary32 format can represent is 1.4E-45, less than 2^{-127}.

7.2 Discussion

It may be desirable to design a custom fixed-point format for representing fractal coordinates for this project. When designing software that needs to do calculations on real numbers (decimals), one typically has to choose between the *float* and *double* data types, both floating-point units, corresponding to *binary32* and *binary64* in IEEE 754. This is related to how computers are designed. However, as suggested in the previous section, floating-point formats may be a bad choice for this project, because the precision of the neccessary calculations will be limited by the precision

of the format around the fractal center anyway. Furthermore, there is no need to represent large numbers in this project; in fact it is sufficient to be able to represent numbers up to 2.0, because the whole Mandelbrot set is located inside a disc of radius 2.0 in the complex coordinate system. Operations on fixed-point formats (such as multiplication and addition) are also simpler to realize in hardware.

However, this will have to be reserved for future work; the *binary32* format will be used throughout this project. This is because it is much more convenient to design with floating-point units in software, and thus verify the hardware implementation. Also, units for performing addition and multiplication on the binary32 format are already available from ARM; these are introduced later in Section 9.1.

8 Boundary checking in the Mandelbrot algorithm

As presented in Section 2.1.1, 2 is a sufficient size of the *BREAK_VAL*-parameter in the Mandelbrot algorithm (Listing 1); when the absolute value of z exceeds 2 it is guaranteed that c is not part of the Mandelbrot set.[4, p. 81] However, any value of BREAK_VAL larger than 2 will work.

This opens for a simplication of the Mandelbrot algorithm that will be presented here. As an alternative to calculating $\sqrt{z_{re}^2 + z_{im}^2} > 2$, or equivalently $z_{re}^2 + z_{im}^2 > 4$, one can simply check whether either $|z_{im}|$ or $|z_{re}|$ exceeds 2. This will certainly be much faster — which is important as this check is performed for each iteration of every pixel in the finished image.

The disadvantage with this method is that it might require more iterations of the algorithm to determine that a point is not in the set. If z_n falls within the blue area in the complex coordinate system in Figure 22, the precise method would catch that the point is not in the set and finish, while the simplifed method would need to continue iterating until z escapes the outer rectangle in Figure 22. In the example in the figure, the precise method will break at $n = 2$, while the simplified method will break at $n = 3$.

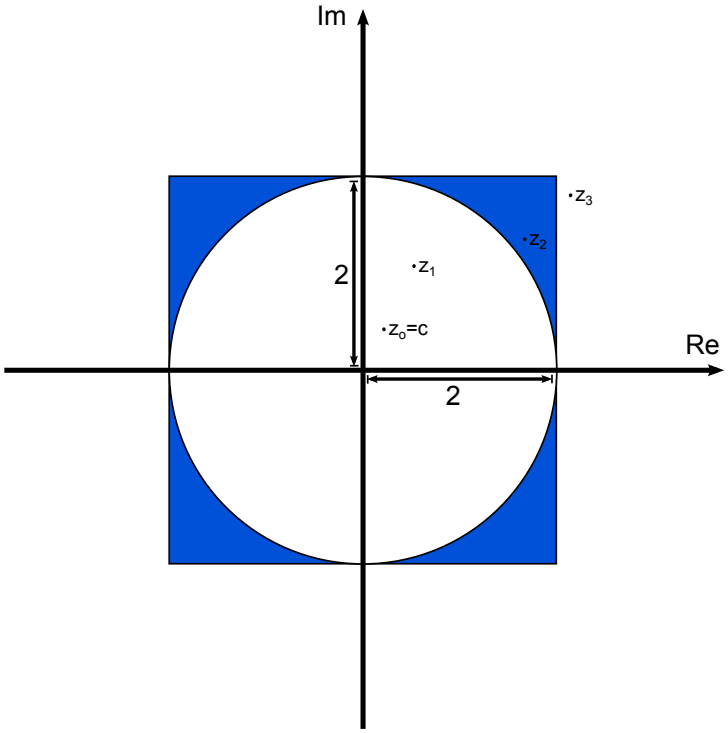


Figure 22: Illustration of the overhead (blue area) from using the simplified boundary check.

Another problem with the simplified boundary checking method is that is could cause problems with the coloring scheme presented in Section 2.1.3. Even though it will work fine for detecting whether a point is in the interior or not, the number of iterations for points in the exterior will sometimes be altered slightly. The effect of

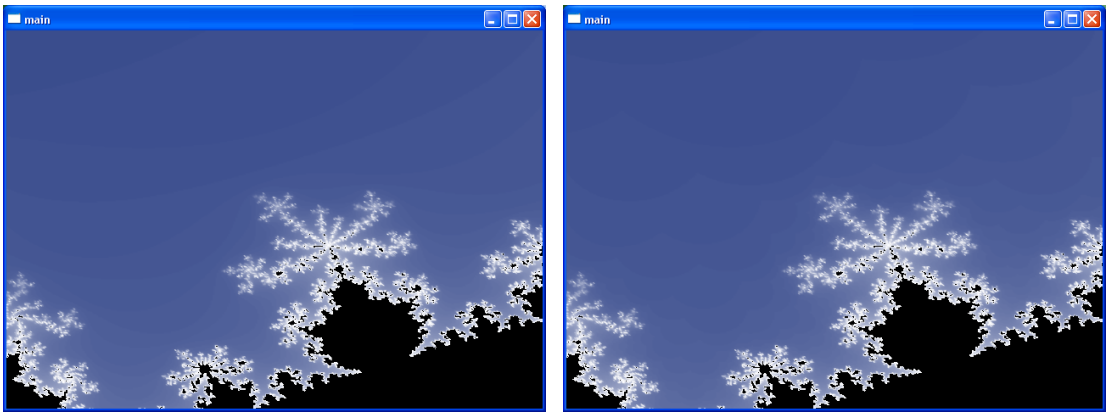
this is investigated in the next section.

8.1 Comparison of boundary check methods

The simplified boundary check method proposed earlier might cause the colored exterior of the set to appear different than with the precise boundary check. This is because the simplified method may report a slightly higher iteration count than the precise method for a given fractal point.

This was investigated with a modified version of the fractal demo. The demo was modified to display the fractal as a flat surface viewed directly from above. A version of the fractal point generator that uses the precise method was then implemented with the hypotf() C library function. A specific fractal point and zoom level were then chosen and a screenshot was taken for each boundary check method.

Figure 23 shows the screenshots of the modified demo using the precise and simplified boundary check method, respectively.



(a) Precise method (b) Simplified method

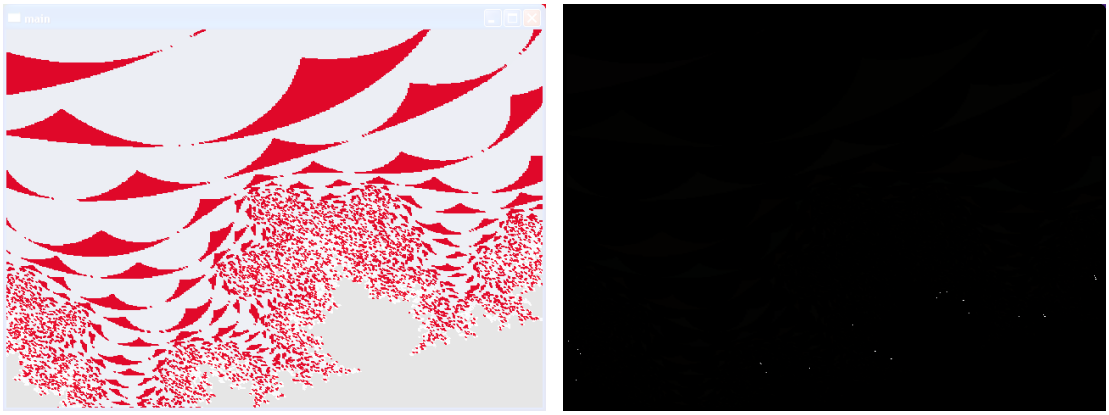
Figure 23: The modified demo application with different boundary check methods

The two screenshots were finally compared pixel for pixel. They were stored in a lossless format and compared using the *ImageMagick* suite of raster image tools.[15]. Figure 24a shows the output of the 'compare'-tool, where red indicates that the corresponding pixel differs between the two images. Figure 24b shows the output of 'composite -compose difference', which is the first image substracted from the second.

8.2 Discussion

It is relatively difficult to see the difference between the two screenshots in Figure 23. Looking closely, the height-curves in the upper half of Figure 23a are more smooth than those of Figure 23b, however. Furthermore, it can be seen from Figure 24b that some of the pixels colored black (it. count 80) when the simplified algorithm is used, should have been white (it. count 79.)

The simplified method is probably the better choice. It is hard to settle on whether these differences are critical or not, but judging by this simple quantitative



(a) Differing pixels (b) Relative difference

Figure 24: Outputs from ImageMagick-tools

investigation, the difference is hardly noticable to the naked eye. Furthermore, the precise method is much more complex, requiring two additional multiplications and one additional addition operation per iteration of the Mandelbrot algorithm. This would likely prove to be a huge performance penalty when the system is implemented in hardware. The simplified method is therefore arguably best and is the method of choice for the fractal generator that was developed as part of this thesis.

9 Hardware implementation

The hardware implementation of the Mandelbrot set generator consists of two parts that were designed separately:

- A fractal point generator unit (*FPG*)
- A vertex array generator that interfaces the fractal point generator(s) (*VAG*)

9.1 Floating point arithmetic

Library modules from ARM are used for floating point addition and multiplication, respectively. These are confidential and thus not reproduced in this thesis. However, the following information is provided:

- Both units are purely combinatorial
- The units are balanced to have about the same critical path length
- The adder is more complex than the multiplicator, and should thus be used sparingly

The modules use the single-precision format (*binary32* of IEEE 754).

9.2 Fractal point generator (FPG)

The core data flow of the fractal point generator (*FPG*) is depicted in Figure 25, arising from Equation 3 on page 6. The *x2*-operator is multiplication with 2, which can be easily implemented as an 11-bit incrementor for FP32. Negating Z_{im}^2 is also a simple operation.

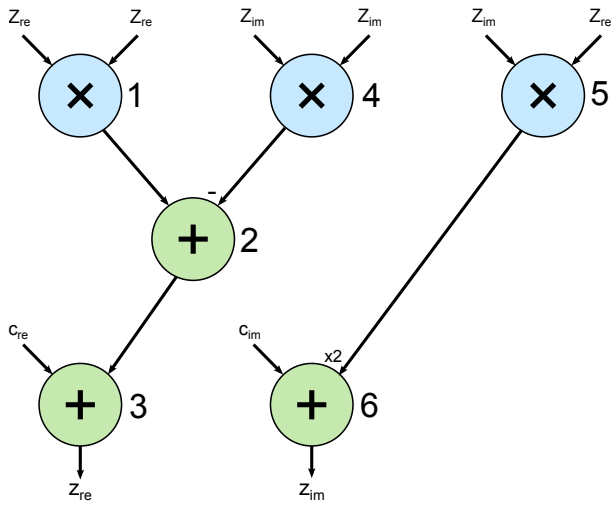


Figure 25: The dataflow graph of one iteration of the Mandelbrot set algorithm.

Even though this design is not especially area-critical, it has to be designed with area in mind. Generating the whole heightmap requires several thousands of

iterations of the data flow graph in Figure 25, and implementing thousands of FP32-adders and multipliers is certainly not viable. It must be assured that all adders and multipliers are utilized as much as possible.

9.2.1 Design exploration

For each vertex point in the height map, the data flow graph in Figure 25 will have to be iterated a number of times. That is, the outputs at the bottom must be fed back to the inputs at the top until one of them exceeds a certain number (such as 2,) or until a counter reaches the iteration limit (such as 80.) For every new vertex point, zeros must be injected at the top instead, and the counter must be reset. The simplest possible design would consist of a number of elements that implement this graph directly. However, this would yield inefficient utilization of the processing elements, regardless of whether the design is pipelined or not.

To ensure that all the functional units are being utilized constantly, consider the option of pipelining the graph in Figure 25 and replicate it downwards as many times as the iteration limit, connecting the outputs of the first circuit to the inputs on the next and so on. This system would be able to accept a new vertex point every clock cycle, and thus utilize all the functional units all the time. This solution would also be very simple to implement. However, most of the vertex points in a single frame will normally not be in the Mandelbrot set, meaning that it would be unnecessary to iterate most points all the way to the iteration limit. Put differently, this design would perform a lot of unnecessary calculations under normal operating conditions. Furthermore, it would not be especially flexible and scalable. For example, what if the iteration limit was 80 and the then minimum of 240 multipliers required more area than what was available in the target?

Many compromises between these two architectures could have been proposed, such as having three vertices share pipelined processing units. However, one additional strategy should be investigated first. The fact that there are exactly three multiplication operations and three addition operations in this particular data flow graph suggests that a processing unit consisting of just two constantly utilized functional units (one adder and one multiplicator) might be a possibility. This would be advantageous; not only is this solution very easily scalable as each unit is very small — it is also relatively simple to design, contrary to the aforementioned system where several vertices share the same pipeline at every given time.

Experiments with the data flow graph reveals that a processing unit with just two functional units can in fact perform the algorithm while utilizing the functional units constantly. Figure 26 illustrates the experimentally obtained high-level synthesis schedule of a single iteration of the algorithm. Note how operation 2 and 3 are switched (relative to Figure 25) to allow this to work out. The iteration is started with $n = 1$, where $Z_{n,re} = c_{re}$ and $Z_{n-1,re} \cdot Z_{n-1}, im = 0$. A counter must be implemented to keep track of the value of n, as well as comparators that halts the iteration process whenever Z_{re}, Z_{im} or n exceeds their thresholds.

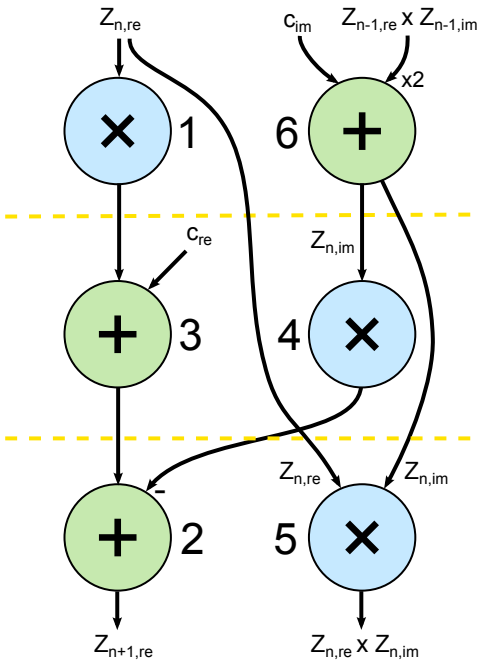


Figure 26: The experimentally obtained maximal utilization schedule of a single iteration of the Mandelbrot set algorithm.

9.2.2 Implementation and verification

The fractal point generator was implemented in Verilog-2001. It was tested with a simple, dynamic test bench written in SystemVerilog-2005. The design was simulated with Synopsys VCS/DVE.

The design is basically a four-state state machine. Its state diagram is shown in Figure 27. The three S_IT-states correspond to the three stages of the data flow graph in Figure 26.

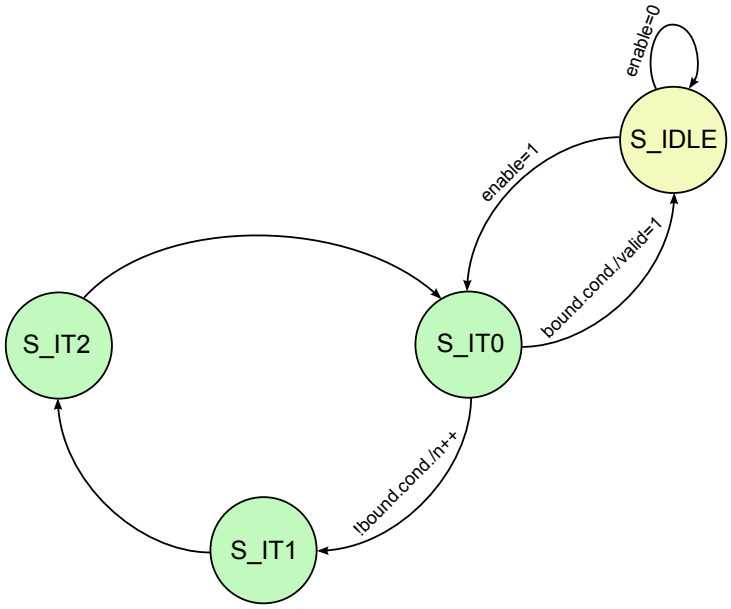


Figure 27: State diagram for the fractal point generator.

The two functional units have their input data selected with multiplexers controlled by the state register. This is illustrated in Figure 28. Additionally a counter keeps track of the iteration count, and logic is added for detecting whether the counter has exceeded the iteration limit or whether $|z_{re}|$ or $|z_{im}|$ has exceeded 2. The full source code of the fractal point generator is enclosed in Listing 8 in Appendix C.

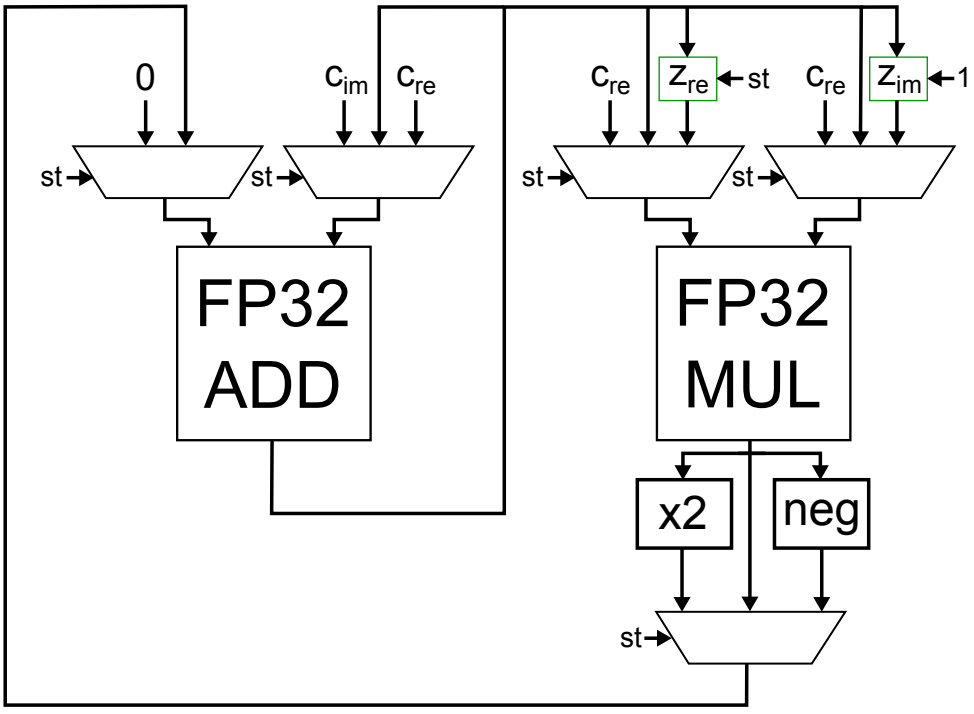


Figure 28: The core data flow of the fractal point generator (FPG). The green boxes are registers, where the arrows entering on the right edges are enable-signals.

The test bench defines an array of test vectors that were generated using the software implementation of the design. The DUT is stimulated with each vector, and the n-output (number of iterations) is compared with the correct answer. The test bench makes use of the *clocking block* functionality in SystemVerilog, as well as (dynamic) assertions and the struct data type. The full source code of the test bench for the fractal point generator is enclosed in Listing 9 in Appendix C.

9.3 Vertex array generator (VAG)

A circuit for generating the vertex array representing the fractal landscape was developed as part of the fractal generator design. This design is mostly concerned with fixed-point arithmetic, but it also has to generate floating point coordinates for the fractal point generators. It interfaces one or more fractal point generators (FPGs).

The vertex array generator (*VAG*) should have the following two important features:

- It should be area-efficient. This means that the number of floating-point units should be minimized and any storage memory should be as small as possible.
- It should be scalable, i.e., the number of interfaced FPGs should be easily configurable, as well as the number of vertices in the landscape.

The generator should be scalable because factors like the number of vertices in the heightmap and the desired frame rate may require faster fractal generation. There is thus a tradeoff between vertex count/frame rate and area. This tradeoff will be investigated later.

9.3.1 Implementation

The vertex array generator was like the fractal point generator designed in Verilog-2001. The design uses the double-buffer method for storing z coordinates as opposed to the shifting method, as discussed in Section 5.4. Apart from this feature no alternative design solutions have been evaluated. However, the chosen implementation is very compact and simple, easily scalable and have no identified weaknesses.

The vertex array generator implements one floating-point adder only, and no multipliers. (Not counting adders and multipliers in the fractal point generators.) The source code of the vertex array generator can be found in Listing 10 in Appendix D.

This being the top-level module for this project, it was not verified separately. The system-level verifications are presented in Section 9.5.

The system was split into three parts, a *control state machine*, an *FPG arbiter* and finally a *Z memory*. Figure 29 shows the structure of the whole design. The rest of this section will present these three parts.

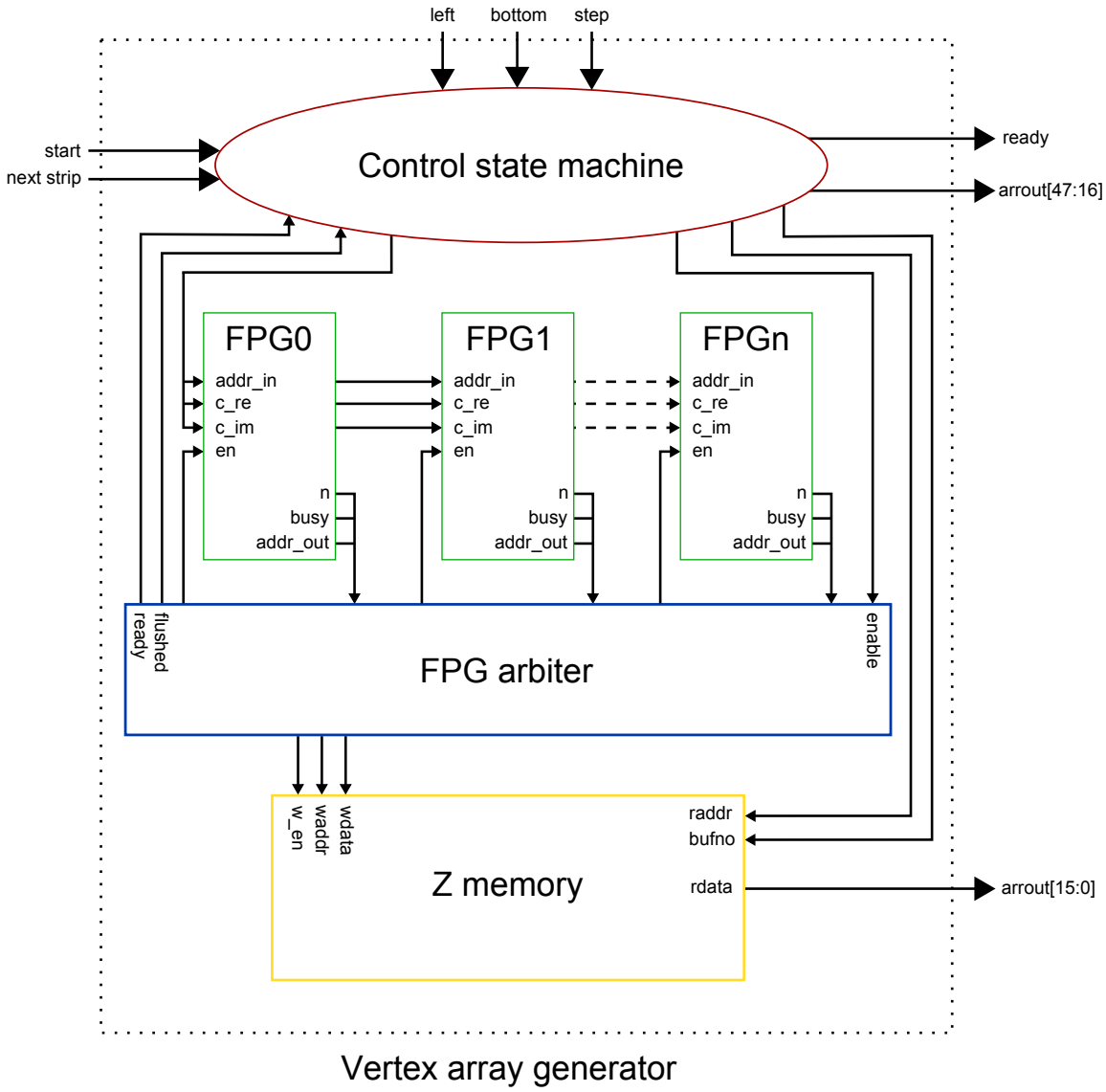


Figure 29: The structure of the vertex array generator hardware design

Control state machine The control state machine (CSM) is a state machine that calculates fractal coordinates and feeds them to the FPGs. The state diagram of this state machine is shown in Figure 30. The term *fractal point* refers to one set of fractal coordinates.

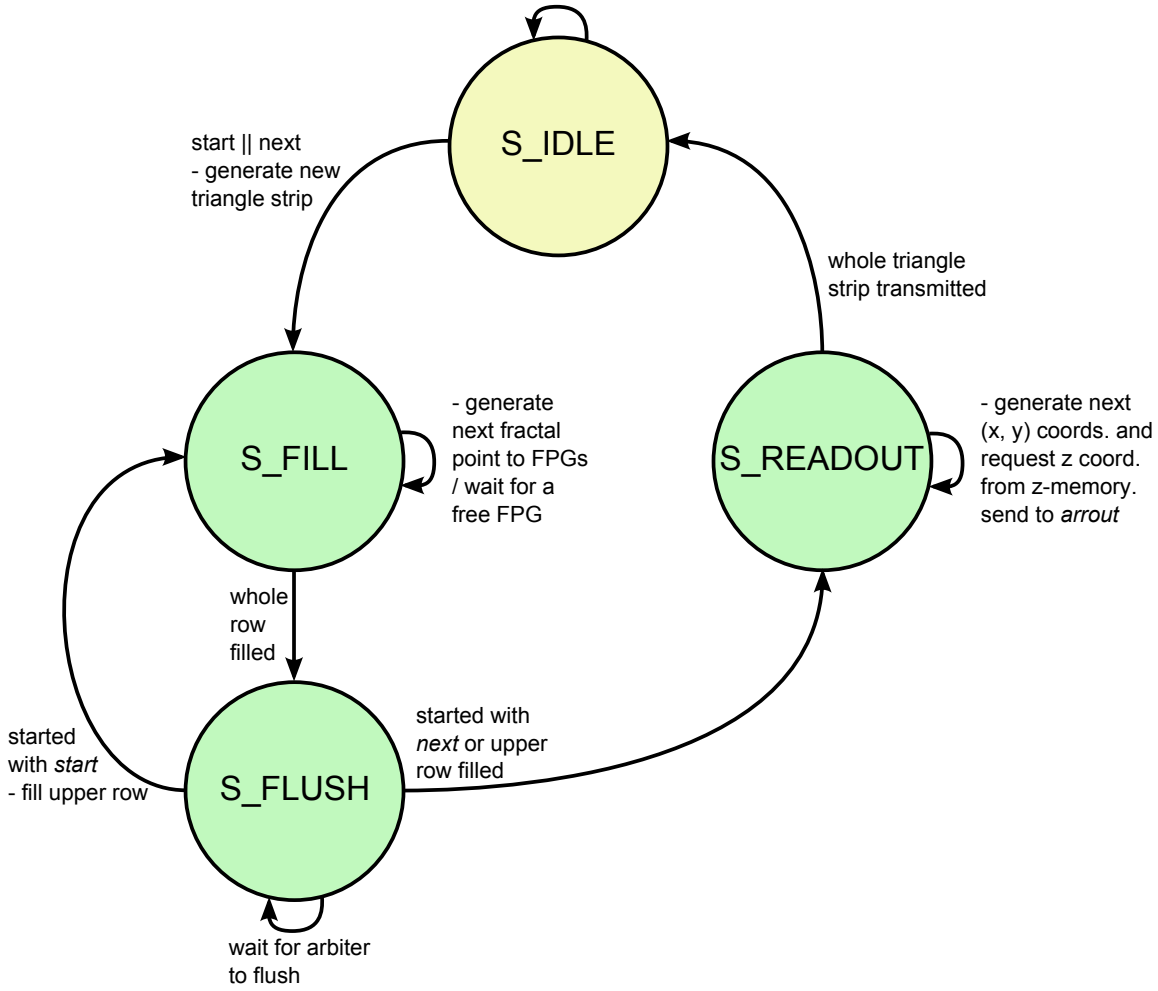


Figure 30: State diagram of the control state machine of the vertex array generator

The CSM has a single floating point (FP32) adder. The fractal coordinates are calculated by adding *step* to *bottom* or *left* and fed to all the FPGs in the system. It is the FPG arbiters task to decide which FPG that is to start calculating the iteration count for the applied coordinates and tell the CSM when to send a new fractal point to the FPGs.

When all the iteration counts in the current triangle strip has been calculated and stored in the Z memory, the CSM will start sending the assembled vertex array out of the module. The X and Y coordinates will be generated by the CSM using two incrementors, while the Z coordinates come from the Z memory. The CSM will keep track of the memory addresses, however.

FPG arbiter The FPG arbiter is responsible for keeping track of which FPGs are busy and which are free. When one or more FPGs are done with a fractal point,

the arbiter will transmit the result from one of them (the iteration count, *n*) to the Z memory. The address to write the result to is generated by the CSM and stored in the FPG while working. (Through *addr_in* and *addr_out*.) Then, any pending fractal point generated by the CSM is accepted by asserting *en* on the free FPG. Finally *ready* is asserted to let the CSM know that the current fractal point has been accepted and consumed.

When all the fractal points in the current vertex row have been accepted, the CSM will wait for *flushed*, indicating that all the FPGs are free (i.e., that all the results are in the Z memory.)

Z memory As discussed in Section 5.4, the Z memory is made up of two identical buffers each large enough to store *NUMPOINTS* z vertex coordinates (half a triangle strip.) The CSM will keep track of what buffer to use. When the vertex array generator is started with *start*, the bottommost vertex row is calculated and stored in the first buffer. Then, *bufno* is toggled and the second row is calculated and stored in the other buffer. For each strip, *bufno* is toggled so that the buffer containing the upper row of the previous strip is preserved. When transmitting the vertex array out of the module, the buffer number is toggled for each vertex so that the correct order of vertices is achieved.

9.4 Synthesis

The design was synthesized using Xilinx ISE 12.4 for a Xilinx Virtex 6 FPGA. The purpose of the synthesis was to get some numbers of the resource usage and speed of the design; the design was not tested in an actual FPGA. Specifically, the design was synthesized for the *xc6vcx75t-1ff484* device. This particular device has 46,560 *look-up tables* (LUTs) available.[26] The number of LUTs utilized by a design corresponds to the complexity (area) of the design; for details the the Virtex-6 FPGA User Guide [27]. The synthesis tool was ran without configuring timing constraints for the design, and with NUMPOINTS set to 400, corresponding to a full 400 by 400 point landscape as used in the fractal demo.

The number of LUTs and estimated clock frequency as a function of number of FPGs are shown in Table 2. The frequencies are calculated as the inverse of the clock period, which is reported by the synthesis tool.

FPGs	LUTs	Clock period	Clock frequency
1	3,637	13.350 ns	75 MHz
2	5,350	14.502 ns	69 MHz
4	9,042	13.883 ns	72 MHz
8	15,763	16.259 ns	62 MHz
16	29,577	15.618 ns	64 MHz

Table 2: Synthesis results

It can be observed that the number of LUTs approximately follows the formula

$$2000 + 1700 \cdot \text{NUMUNITS} \tag{6}$$

where NUMUNITS is the number of FPGs in the design. This implies that the FPGs spend about 1700 LUTs each, while the VAG spends about 2000 LUTs. Note that the FPG arbiter (see Figure 29) is the only component of the VAG that changes when the number of FPGs are increases or decreased. Evidently these changes to the FPG arbiter does not affect the area of the VAG to any significant degree.

The estimated clock frequencies appear to be somewhat uncorrelated to the number of FPGs in the design. There is no clear explanation for this effect — it is assumed that the synthesis tool applies certain strategies on a sporadic basis when routing the design on the FPGA. However, there is a slight tendency for that the frequency decreases when the number of FPGs is increased. This would be expected as more FPGs means larger, slower multiplexers in the FPG arbiter and higher fan-out between the state machine and FPG inputs. (i.e., the state machine outputs have to drive more gates.)

9.5 Verification and performance analysis

The hardware fractal generator was verified and analyzed using two dynamic test benches written in SystemVerilog-2005, as when verifying the fractal point generator (FPG) earlier. Simulations were done with Synopsys VCS/DVE.

The purpose of the first test bench is to verify the functionality of the complete system, consisting of the vertex array generator and a number of fractal point generators. The purpose of the second test bench is to assess the performance of the system, and not be concerned about correct functionality.

To ease simulation, the first test bench tests the system with a single 20 by 20 point fractal landscape test case. This differs from the 400 by 400 point fractals used in the fractal demo (see Chapter 6). The test case was chosen to have a wide selection of different iteration counts (heights) across the landscape; it is shown in Figure 31. The software demo was modified to write the output of the fractal generator to a file that was then used to compare the output of the hardware fractal generator in the test bench. The source code of the first test bench can be found in Listing 11 in Appendix D.

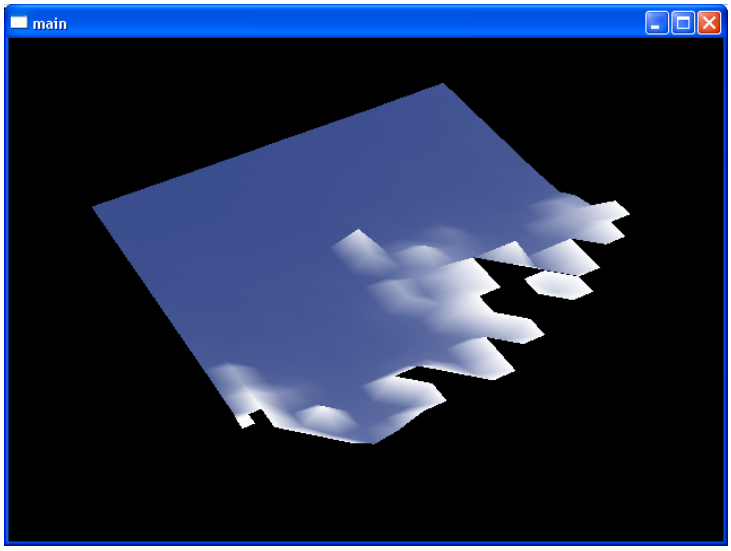


Figure 31: The 20 by 20 point fractal test case

The test bench will apply configuration parameters to the device under test (DUT), then request the first triangle strip array and wait for the ready-signal. When the ready-signal arrives, the data-output of the DUT is compared to the "golden" solution from the software implementation, stored in the text file `testcase.txt`. This is done using SystemVerilog assertions [16]. A short excerpt from a typical output of the first test bench is shown in Listing 5 below. Each PASS correspond to one vertex in the generated vertex array from the DUT.

```
1  Chronologic VCS simulator copyright 1991-2011
2  Contains Synopsys proprietary information.
3  Compiler version E-2011.03; Runtime version E-2011.03;  Jun 16 14:47
       2011
4  Requesting strip           0.
5  Waiting for ready-signal...
6  PASS            0,          0
7  PASS            0,          1
8  PASS            0,          2
9  PASS            0,          3
10 PASS            0,          4
11 PASS            0,          5
12 (...)
```

Listing 5: Sample output from simulation

It is believed that this test bench proves correct behavior of the fractal generator to a satisfying degree. It revealed some design mistakes that were fixed and now runs without any failing assertions. The test case was chosen to provide a wide range of test vectors with different outcomes. It has additionally been verified that the test case runs successfully with several different numbers of FPGs in the system. See next paragraph.

Effect from increasing the number of FPGs The number of clock cycles spent during generation of the fractal arrays were recorded; the results are in Table 3, as a function of the number of fractal point generators the vertex array generator implements. (The NUMUNITS parameter.)

FPGs	CCs
1	31,251
2	16,541
4	9,297
8	5,883
16	4,240

Table 3: Number of clock cycles spent generating the fractal arrays as a function of number of fractal point generators. 20 by 20 point test case.

Note how the number of clock cycles per FPG decreases as more FPGs are added. This happens because there are only 20 new fractal points per triangle strip, and so at the end of each triangle strip most of the FPGs will typically be unused. For example, if the number of FPGs are 16 and all the fractal points in a strip have the same iteration count, the first 16 fractal points will complete at the same time, leaving 12 FPGs unused when the final 4 fractal points are being calculated.

9.5.1 Comparable performance

To be able to compare the performance of the hardware fractal generator with the software fractal generator, the same (all black) landscape as in Section 6.3.3 is requested by the second test bench. Unlike the first test bench, no solution text file is imported; it is simply checked whether the z coordinates in the generated vertex arrays are set to the iteration limit (80). Thus this test bench hardly checks for correct behavior, its purpose is to investigate the performance of the design by recording the number of clock cycles required to generate this worst-case scenario landscape.

Combined with the synthesis results from Section 9.4, generation times in time units can be obtained and compared with the software implementation. The results can be found in Table 4.

FPGs	CCs	Est. exec. time	Est. frame rate
1	38,560,397	0.515 s	2 FPS
2	19,440,797	0.282 s	4 FPS
4	9,881,597	0.137 s	7 FPS
8	5,103,197	0.083 s	12 FPS
16	2,716,397	0.042 s	24 FPS

Table 4: Number of clock cycles spent generating the fractal arrays as a function of number of fractal point generators. 400 by 400 point worst-case scenario.

The estimated execution times were calculated by multiplying the number of clock cycles with the clock period from the synthesis. The frame rates are calculated by

rounding the inverse of the execution times.

9.6 Discussion

Comparing the obtainable frame rates from the software and hardware implementation of the fractal generator, the hardware implementation is faster as soon as the number of fractal point generators (FPGs) is increased to 2 or more. However, the estimated frame rate of 4 FPS with 2 FPGs is hardly impressive — 4 FPS will provide a very "choppy" animation. To obtain a good frame rate of 24 FPS, which is a well established standard in the TV and movie-making business[22], 16 FPGs are needed.

Even with 16 FPGs, the resource usage (area) of the fractal generator is relatively modest. With 16 FPGs it spent 29,577 LUTs during synthesis, which is 64% of the smallest available FPGA in the Virtex 6 family of FPGAs from Xilinx (LX75T) and only 6% of the largest available FPGA in the same family (LX760).[26, tab.1]

The worst-case scenario test case arguably does not represent a typical use of the fractal generator. The fractal demo will mostly have a frame rate of more than 3 FPS (the worst-case scenario frame rate), because most of the points in the fractal landscape typically have iteration counts of less than the maximum (80). However, when zooming in on the edge of the Mandelbrot set, the average iteration count will increase with the zoom level, slowing the demo animation. In fact, the maximum zoom level is limited by the iteration count limit; further zooming would require a higher limit, making the animation even slower than the studied worst-case scenario test case.

As a final comment, the computer used to obtain the estimated frame rate for the software implementation of the fractal generator is much more powerful than the CPU in embedded systems targeted by the Mali family of graphic processors. Thus is would arguably be fairer to compare the performance of the software implementation to an ASIC implementation rather than an FPGA implementation of the hardware fractal generator. Alternatively, the software implementation should have been tested on an embedded platform rather than on a desktop computer.

10 Conclusion and future work

Although the designed fractal generator was not tested on a hardware platform, the work done in this thesis yielded several results. Future work should focus on integrating the fractal generator with the Mali-400 graphics processor on an FPGA and produce a working fractal demo that utilizes this hardware.

The most important contribution from this thesis is the design of an elegant, scalable fractal generator which with few resources was shown to beat the corresponding software implementation running on a desktop computer. With 16 FPGs (fractal point generator cores), the implementation was almost 10 times faster than in software and spent 63 % of the resources in the smallest available Xilinx Virtex-6 FPGA. A OpenGL ES 2.0 demo was written to demonstrate the software fractal generator, planned to eventually be used for demonstrating the hardware implementation as well.

A simplification of the Mandelbrot set algorithm was proposed in Chapter 8, and judged to be superior to the original algorithm. The two algorithms were tested and compared using the fractal demo (presented in Chapter 6). The simplified algorithm is the one implemented in the hardware fractal point generator.

Another area that should be considered for future work is the development of a custom fixed-point format for representing fractal coordinates. Chapter 7 identifies problems with using floating-point formats for this task that in the end will limit how far into the fractal the system can zoom. However, the iteration count limit (which is related to the achievable frame rate, see the discussion in Section 9.6) will also limit the maximum zoom level. This means that to gain any significant benefit from changing to a fixed-point format for fractal coordinates, the number of FPGs in the system must be increased as well, increasing the size of the design.

A Source code, fractal demo

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include <time.h>
5
6  #include "GLES2/gl2.h"
7  #include "EGL/egl.h"
8
9  #include "fractal.h"
10 #include "window.h"
11 #include "shader.h"
12 #include "esUtil.h"
13
14 static int xrot = 25, zrot = -30;
15 static float fractcenter[2] = {-1.786863383,-2.369371475e-06};
16 //static float fractcenter[2] = {-0.3912647562f, 0.6785442331f};
17 //static float fractcenter[2] = {0.0f, 0.0f};
18
19 static float minwidth = 4.775061513e-09;
20 static float fractwidth = 50.0;
21
22 static GLfloat ls_scale = 2.5;
23
24 static const unsigned int uiWidth  = 640;
25 static const unsigned int uiHeight = 480;
26
27 #ifdef _WIN32
28 static HWND hWindow;
29 static HDC  hDisplay;
30 #else
31 static Window hWindow;
32 static Display *hDisplay;
33 #endif
34
35 static EGLDisplay sEGLDisplay;
36 static EGLContext sEGLContext;
37 static EGLSurface sEGLSurface;
38 static GLuint uiProgram, uiFragShader, uiVertShader;
39 static ESMatrix perspective, modelview, mvp;
40
41 static GLshort *landscape;
42
43 void init()
44 {
45   // EGL configuration. Adopted from 'cube' application.
46   EGLint aEGLAttributes[] = {
47     EGL_RED_SIZE, 8,
48     EGL_GREEN_SIZE, 8,
49     EGL_BLUE_SIZE, 8,
50     EGL_DEPTH_SIZE, 16,
51     EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
52     EGL_NONE
53   };
54
55   EGLint aEGLContextAttributes[] = {
56     EGL_CONTEXT_CLIENT_VERSION, 2,
57     EGL_NONE
58   };
59
60   EGLConfig aEGLConfigs[1];
61   EGLint    cEGLConfigs;
62
63   GLint iLocPosition = 0;
64
65   GLint iLocColour, iLocTexCoord, iLocNormal, iLocMVP;
66   GLint iLocXangle, iLocYangle, iLocZangle;
67   GLint iLocAspect, iLocLightPos, iLocSampler, iLocSampler2;
```

```
68
69    GLenum myTex, myTex2;
70
71    int i, linked;
72
73    // EGL initialization. Adopted from 'cube' application.
74  #ifdef _WIN32
75    hWindow = create_window(uiWidth, uiHeight);
76  #else
77    hWindow = create_window("OpenGL␣ES␣2.0␣Example␣on␣a␣Linux␣Desktop", uiWidth,
78      uiHeight, hDisplay, sEGLDisplay, aEGLConfigs[0], &colormap, &pVisual);
79  #endif
80
81  #ifdef _WIN32
82    hDisplay = GetDC(hWindow);
83  #else
84    hDisplay = XOpenDisplay(NULL);
85  #endif
86
87    if (!hDisplay) {
88      printf("Could␣not␣open␣display\n");
89      exit(-1);
90    }
91
92    sEGLDisplay = eglGetDisplay(hDisplay);
93
94    (eglInitialize(sEGLDisplay, NULL, NULL));
95    eglChooseConfig(sEGLDisplay, aEGLAttributes, aEGLConfigs, 1, &cEGLConfigs);
96
97    if (cEGLConfigs == 0) {
98          printf("No␣EGL␣configurations␣were␣returned.\n");
99      exit(-1);
100   }
101
102   sEGLSurface = eglCreateWindowSurface(sEGLDisplay, aEGLConfigs[0], (EGLNativeWindowType)
        hWindow, NULL);
103
104     if (sEGLSurface == EGL_NO_SURFACE) {
105         printf("Failed␣to␣create␣EGL␣surface.\n");
106     exit(-1);
107     }
108
109   sEGLContext = eglCreateContext(sEGLDisplay, aEGLConfigs[0], EGL_NO_CONTEXT,
        aEGLContextAttributes);
110
111     if (sEGLContext == EGL_NO_CONTEXT) {
112         printf("Failed␣to␣create␣EGL␣context.\n");
113     exit(-1);
114     }
115
116   eglMakeCurrent(sEGLDisplay, sEGLSurface, sEGLSurface, sEGLContext);
117
118   // Shader initialization.
119   process_shader(&uiVertShader, "shaders/shader.vert", GL_VERTEX_SHADER);
120   process_shader(&uiFragShader, "shaders/shader.frag", GL_FRAGMENT_SHADER);
121
122   uiProgram = glCreateProgram();
123
124   // Attach shaders and link uiProgram.
125   glAttachShader(uiProgram, uiVertShader);
126   glAttachShader(uiProgram, uiFragShader);
127   glLinkProgram(uiProgram);
128
129   // Bind position to attribute 0.
130   glBindAttribLocation (uiProgram, 0, "a_position");
131
132   // Link the program.
133   glUseProgram(uiProgram);
134
135   // Check the link status.
```

```
136   glGetProgramiv(uiProgram, GL_LINK_STATUS, &linked);
137   if (!linked) {
138     printf("Program not linked.\n");
139     exit(-1);
140   }
141
142   // Enable depth testing.
143   glEnable(GL_DEPTH_TEST);
144 }
145
146 void reshape(GLsizei w, GLsizei h)
147 {
148   // Update perspective matrix.
149   const float edge = NUMPOINTS / 2;
150   esMatrixLoadIdentity(&perspective);
151   esFrustum(&perspective, -edge, edge, -edge, edge, NUMPOINTS * 1.25, 3 * NUMPOINTS);
152   glViewport(0, 0, w, h);
153 }
154
155 void display()
156 {
157   int s;
158   GLint mvpLoc;
159   float left, bottom, step;
160   unsigned int *leftp, *bottomp, *stepp;
161   static int framec = 0;
162   clock_t start;
163   double duration;
164
165   // Update modelview matrix.
166   esMatrixLoadIdentity(&modelview);
167   esTranslate(&modelview, 0, 0, -2.0 * NUMPOINTS);
168   esRotate(&modelview, xrot, 1, 0, 0);
169   esRotate(&modelview, zrot, 0, 0, 1);
170   esScale(&modelview, ls_scale, ls_scale, 1);
171   esTranslate(&modelview, -(NUMPOINTS-1) / 2, -(NUMPOINTS-1) / 2, 0);
172
173   // Update MVP matrix by multiplying the modelview matrix with the perspective matrix.
174   esMatrixMultiply(&mvp, &modelview, &perspective);
175
176   // Clear buffers.
177   glClearColor(0.0, 0.0, 0.0, 0.0);
178   glClearDepthf(200);
179   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
180
181   // Load the MVP matrix:
182   mvpLoc = glGetUniformLocation(uiProgram, "u_mvpMatrix");
183   glUniformMatrix4fv(mvpLoc, 1, GL_FALSE, (GLfloat*) &mvp.m[0][0]);
184
185   glEnableVertexAttribArray(0);
186
187   // Draw the landscape.
188   left = fractcenter[0] - fractwidth / 2;
189   bottom = fractcenter[1] - fractwidth / 2;
190   step = fractwidth / (NUMPOINTS-1);
191   landscape = initLandscape(left, bottom, step);
192   //landscape = initLandscape(left, bottom, fractwidth);
193
194   leftp = (unsigned int*) &left;
195   bottomp = (unsigned int*) &bottom;
196   stepp = (unsigned int*) &step;
197   printf("left:%x, bottom:%x, step:%x\n", *leftp, *bottomp, *stepp);
198
199   start = clock();
200   for (s = 0; s < (NUMPOINTS-1); s++) {
201     // For each triangle strip in the landscape.
202     getNextLandscapeStrip(); // Generate the strip.
203     glVertexAttribPointer(0, 3, GL_SHORT, GL_FALSE, 0, landscape);
204     glDrawArrays(GL_TRIANGLE_STRIP, 0, NUMPOINTS*2);
205   }
```

```
206   duration = (double)(clock() - start) / CLOCKS_PER_SEC;
207
208   printf("Frame %d took %2.3f seconds to generate.\n", framec++, duration);
209
210   glFlush();
211   if (!eglSwapBuffers(sEGLDisplay, sEGLSurface)) {
212     printf("Failed to swap buffers.\n");
213   }
214 }
215
216 int main(int argc_in, char **argv) {
217   int bDone = 0;
218   MSG sMessage;
219
220   init();
221   reshape(uiWidth, uiHeight);
222   display();
223
224   // Enter event loop.
225   while (!bDone) {
226 #ifdef _WIN32
227     if(PeekMessage(&sMessage, NULL, 0, 0, PM_REMOVE)) {
228       if(sMessage.message == WM_QUIT) {
229         bDone = 1;
230       } else {
231         TranslateMessage(&sMessage);
232         DispatchMessage(&sMessage);
233       }
234     }
235 #else
236     while (XPending(hDisplay) > 0) {
237       XNextEvent(hDisplay, &e);
238
239       if (e.type == ButtonPress) {
240         bDone = 1;
241       }
242     }
243 #endif
244
245     /*
246     if ((ls_scale *= 1.02) >= 5.0) {
247       ls_scale = 2.5;
248       if ((fractwidth *= 0.5) < minwidth) {
249         fractwidth = 50.0;
250       }
251     }
252     */
253
254     /*
255     if ((fractwidth *= 0.98) < minwidth) {
256       fractwidth = 50.0;
257     }
258     display();
259     */
260
261     fractwidth *= 0.98;
262     display();
263
264     //Sleep(1000);
265   }
266   return 0;
267 }
```

Listing 6: C source code for the fractal demo application. Other source files and shaders are generally taken from the *cube* demo application (discussed earlier) and delivered with the thesis.

B Source code, fractal generator, software model

```
1  //
2  // fractal.c, by Per Christian Corneliussen, March 2011
3  //
4  // This is the software version of the fractal generator. It is layed out so
5  // that each C function corresponds to a hardware module. The algorithms should
6  // also be as similar as possible to how it is planned done in hardware.
7  //
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <math.h> // For hypotf()
12 #include "GLES2/gl2.h" // Emulator header, for GLshort.
13
14 #include "fractal.h" // Defines NUMPOINTS.
15
16 static GLshort *stripmem = NULL;
17 static GLshort *vertexmem = NULL;
18 static float left, bottom, step;
19 static int k; // Current strip number.
20
21 int getFractPoint(float re, float im)
22 // Get the number of iterations (n) before |z| exceeds 2. This function
23 // corresponds to the fractal point generator hardware module, mbrot.v.
24 {
25   int n;
26   float z_re = 0.0f, z_im = 0.0f;
27   for (n = 1; n < 80; n++) {
28     float z_re_old = z_re;
29     z_re = z_re*z_re - z_im*z_im + re;
30     z_im = 2.0f * z_re_old * z_im + im;
31
32     // Simplified boundary check.
33     if ((z_re >= 2.0f) || (z_re <= -2.0f) || (z_im >= 2.0f) || (z_im <= -2.0f))
34       break;
35
36     // Precice boundary check. This is the one used for the "comparison of
37     // boundary check methods" chapter of the report.
38     /* if (_hypotf(z_re, z_im) >= 2.0f)
39       break; */
40   }
41   return n;
42 }
43
44 GLshort *initLandscape(float _left, float _bottom, float _step)
45 // Allocate memory for landscape generation.
46 {
47   // Allocate memory for a single triangle strip. If initLandscape() has been
48   // called previously, the old chunk of allocated memory is reused.
49   if (vertexmem == NULL) {
50     vertexmem = malloc(NUMPOINTS * 2 * 3 * sizeof(GLshort));
51     stripmem = malloc(NUMPOINTS * 2 * sizeof(GLshort));
52     if (stripmem == NULL) {
53       fprintf(stderr, "Out of memory.\n");
54       exit(-1);
55     }
56   }
57
58   // Store configuration:
59   left = _left;
60   bottom = _bottom;
61   step = _step;
62
63   // Set current strip number to 0.
64   k = 0;
65
66   return vertexmem;
67 }
```

```
68
69  void getNextLandscapeStrip()
70  // Update memory area returned by initLandscape() with next triangle strip.
71  // initLandscape() must be called first for each frame. This module corresponds
72  // to the vertarray.v hardware module.
73  {
74    int i = 0, j = 0;
75    int x = 0, y = k;
76    float re = left;
77    float im_upper = bottom + step;// * (y+1);
78    float im_lower = bottom;// + step * y;
79
80    //printf("Strip no. %d:\n", k);
81
82    // Generate fractal points for this strip. If on the first strip (k == 0), all
83    // the points are generated. Otherwise, only the upper row is generated, and
84    // the lower row is reused from the previous (lower) strip.
85    while (i < (NUMPOINTS * 2)) {
86      //re = left + step * (i/2);
87      // Lower row:
88      if (k == 0) stripmem[i++] = getFractPoint(re, im_lower);
89      else        stripmem[i++] = stripmem[i+1];
90      // Upper row:
91      stripmem[i++] = getFractPoint(re, im_upper);
92      re += step;
93    }
94
95    // Finally, the vertex array is generated. (This should be easily implemented
96    // in hardware, as everything is constant except stripmem[].)
97
98    for (i = 0; i < (NUMPOINTS*2); i++) {
99      // For each vertex in the triangle strip.
100     vertexmem[j++] = (y == k) ? x : x++;
101     vertexmem[j++] = (y == k) ? y++ : y--;
102     vertexmem[j++] = stripmem[i];
103     //printf("%04x%04x%04x\n", vertexmem[j-3], vertexmem[j-2], vertexmem[j-1]);
104   }
105
106   // Finally, increment 'bottom' for next strip. Note how this whole thing is
107   // made so that strips must be retreived in order.
108   bottom += step;
109   k++;
110 }
```

Listing 7: C source code for the fractal generator software model

C Source code, fractal point generator, hardware

```
1  //
2  // fpg.v, by Per Christian Corneliussen, March 2011.
3  //
4  // Fractal point generator. This module will calculate the height of a single
5  // point in the Mandelbrot set, specified with the (c_re, c_im) inputs. It will
6  // spend three clock cycles per iteration of the algorithm, and so the execution
7  // time is not constant. The 'valid' output will go high for one clock cycle
8  // when the algorithm is finished. A new operation may then be started
9  // immediately.
10 //
11 // There is no support for subnormal numbers. c_re is registered, but c_im is
12 // not. This is done to assist the vertex array generator (VAG.)
13 //
14
15 module fpg(
16   input             clk,      // Clock.
17   input             reset_n,  // Reset.
18   input      [31:0] c_re,     // Real value of point to be calculated. FP32.
```

```
19    input      [31:0] c_im,      // Corresponding imaginary value. FP32.
20    input             enable,    // Synchronous start.
21    input      [15:0] addr_in,   // Will be moved to addr_out whenever started.
22    output reg [ 7:0] n,         // Number of iterations.
23    output reg busy,             // Except after reset, busy='0' means n is valid.
24                                 // Note that after reset, n will be 0.
25    output reg [15:0] addr_out
26    );
27
28    // Configurable parameters:
29    parameter MAX_ITERATIONS = 80;
30
31    // State encoding:
32    parameter S_IDLE = 0, S_IT0 = 1, S_IT1 = 2, S_IT2 = 3;
33    parameter M_OFF = 0, M_X2 = 1, M_NEGATE = 2;
34
35    reg  [ 2:0] state;                      // State register.
36    reg  [ 1:0] mulsel;                     // Specifies multiplicator post-op.
37    reg  [31:0] mul_a, mul_b, add_a, add_b; // Functional unit inputs.
38    wire [31:0] mul_out_t, add_out;         // Functional unit outputs.
39    reg  [31:0] mul_out;                    // Multiplicator post-op output.
40    wire [32:0] add_out_t;                  // Adder temporary output.
41    reg  [31:0] z_re, z_im, c_re_r;         // Temporary storage.
42
43    // Functional units. The FP32 adders uses a derived 33-bit format internally
44    // that needs to be converted to and from FP32. Even though the adder
45    // supports subnormal numbers, the multiplicator (and this module) does not,
46    // thus the FP32<->FP33 conversion is done in a simplified way.
47    vithar_lib_fp32_adder_main adder(
48      .a({add_a[31:23], 1'b1, add_a[22:0]}),
49      .b({add_b[31:23], 1'b1, add_b[22:0]}),
50      .res(add_out_t)
51    );
52    assign add_out = {add_out_t[32:24], add_out_t[22:0]};
53
54    vithar_lib_f32_mul multiplier(
55      .clk(clk),          // Note that clk and reset_n are unused unless specific
56      .reset_n(reset_n),  // parameters are set. Otherwise the unit is fully
57      .enable(1'b1),      // combinatorial and valid is connected to enable.
58      .a(mul_a),
59      .b(mul_b),
60      .dout(mul_out_t),
61      .valid()
62    );
63
64    always @ (mulsel, mul_out_t)
65    begin
66      // To support the data flow graph of the Mandelbrot algorithm, there is
67      // logic for post-processing the output of the multiplication unit. Note
68      // that the M_X2 (multiplication by 2) operation is made in the simplest
69      // possible way - it does not support NaN or subnormal numbers.
70      case (mulsel)
71        M_X2: mul_out = {mul_out_t[31], mul_out_t[30:23] + 8'b1, mul_out_t[22:0]};
72        M_NEGATE: mul_out = {~mul_out_t[31], mul_out_t[30:0]};
73        default: mul_out = mul_out_t;
74      endcase
75    end
76
77    always @ (posedge clk or negedge reset_n)
78    begin
79      if (!reset_n) begin
80        // Reset everything.
81        mul_a <= 0;
82        mul_b <= 0;
83        add_a <= 0;
84        add_b <= 0;
85        z_re <= 0;
86        z_im <= 0;
87        c_re_r <= 0;
88        addr_out <= 0;
```

```
89          n <= 0;
90          busy <= 0;
91          state <= S_IDLE;
92          mulsel <= M_OFF;
93
94        end else begin
95          // The Mandelbrot set is designed as an FSM with three repeating states
96          // for each iteration of the algorithm.
97
98          z_im <= add_out;  // z_im needs to be stored between S_IT1 and S_IT2;
99                            // this can be implemented as add_out delayed.
100
101         case (state)
102           S_IDLE: begin
103             // Unit idle. Send start values into the functional units.
104             mul_a <= c_re;
105             mul_b <= c_re;
106             add_a <= 0;
107             add_b <= c_im;
108             z_re <= c_re;
109             c_re_r <= c_re;
110             mulsel <= M_OFF;
111
112             if (enable) begin
113               // If there is a pending request, initiate the algorithm by
114               // jumping to the next state.
115               state <= S_IT0;
116               busy <= 1'b1;
117               n <= 0;
118               addr_out <= addr_in;
119             end else begin
120               state <= S_IDLE;
121               n <= n;
122               busy <= 0;
123             end
124           end
125
126           S_IT0: begin
127             // Iteration cycle 0.
128             mul_a <= add_out;
129             mul_b <= add_out;
130             add_a <= mul_out;
131             add_b <= c_re_r;
132             mulsel <= M_NEGATE;
133             n <= n + 8'b1;
134
135             // The boundary check is done here. This is done by checking whether
136             // |z_re| or |z_im| exceeds 2, or whether n exceeds max. it. count.
137             //$display("Boundary check at n=%d: z=(%x, %x)", n, z_re, add_out);
138             if ((n == (MAX_ITERATIONS-1)) || (add_out[30:23] > 127) || (z_re[30:23] > 127))
139             begin
140               // Iteration count or z exceeds boundaries. Assert valid-output
141               // and return to idle state.
142               busy <= 0;
143               state <= S_IDLE;
144             end else begin
145               busy <= 1'b1;
146               state <= S_IT1;
147             end
148           end
149
150           S_IT1: begin
151             // Iteration cycle 1.
152             mul_a <= z_re;
153             mul_b <= z_im;
154             add_a <= mul_out;
155             add_b <= add_out;
156             z_re <= z_re;
157             mulsel <= M_X2;
158             n <= n;
```

```
159             busy <= 1'b1;
160             state <= S_IT2;
161           end
162
163           S_IT2: begin
164             // Iteration cycle 2.
165             mul_a <= add_out;
166             mul_b <= add_out;
167             add_a <= mul_out;
168             add_b <= c_im;
169             z_re <= add_out;
170             mulsel <= M_OFF;
171             n <= n;
172             busy <= 1'b1;
173             state <= S_IT0;
174           end
175
176           default: begin
177             // Undefined state.
178             mul_a <= 32'bx;
179             mul_b <= 32'bx;
180             add_a <= 32'bx;
181             add_b <= 32'bx;
182             z_re <= 32'bx;
183             mulsel <= M_OFF;
184             n <= 0;
185             busy <= 0;
186             state <= S_IDLE;
187           end
188         endcase
189       end
190     end
191   endmodule
```

Listing 8: Verilog source code for the fractal point generator

```
1  //
2  // fpg_tb.sv, by Per Christian Corneliussen, March 2011.
3  //
4  // SystemVerilog test bench for fpg.v. This is an informal test bench, it
5  // simply tests a number of Mandelbrot points that was calculated beforehand
6  // with the software implementation.
7  //
8
9  module fpg_tb();
10
11   // DUT ports:
12   logic clk, reset_n, enable, busy;
13   logic [31:0] c_re, c_im;
14   logic [7:0] n;
15   logic [15:0] addr_in, addr_out;
16
17   // Connect DUT ports:
18   fpg dut(.*);
19
20   typedef struct packed {
21     bit [31:0] re;
22     bit [31:0] im;
23     bit [7:0] n;
24   } test_vector_t;
25
26   // Test vectors:
27   test_vector_t test_vector [] = '{
28     {32'h40000000, 32'h40000000, 8'd1},
29     {32'h3c23d70a, 32'h3f851eb0, 8'd6},
30     {32'h3ba3d70a, 32'h3f80a3d7, 8'd8},
31     {32'h3dcccccd, 32'h3f266666, 8'd14},
32     {32'h3dcccccd, 32'h3f23d70a, 8'd48},
33     {32'h3dcccccd, 32'h3f1aca58, 8'd65},
34     {32'h3e0ea4a9, 32'h3f1aca9b, 8'd78},
35     {32'h0, 32'h0, 8'd80}
36   };
37
38   assign addr_in = 16'b0;
39
40   // Clock driver:
41   always #5 clk = !clk;
42
43   clocking cc @(posedge clk);
44     default input #1 output negedge;
45     output reset_n;
46     output enable, c_re, c_im;
47     input busy, n;
48   endclocking
49
50   initial begin
51     // Enable dumping:
52     $vcdpluson(0, dut);
53
54     // Initiate clock and reset DUT:
55     clk = 0;
56     reset_n <= 0;
57     enable <= 0;
58
59     ##1 cc.reset_n <= 1; // Will be asserted on the negative edge.
60
61     foreach(test_vector[i]) begin
62       // Send test vector to DUT:
63       cc.c_re <= test_vector[i].re;
64       cc.c_im <= test_vector[i].im;
65       cc.enable <= 1'b1;
66       ##1 cc.enable <= 1'b0;
67       // Wait for valid-signal:
68       @(negedge cc.busy);
69       // Assert answer:
```

```
70        assert (cc.n == test_vector[i].n) $display("PASS %d", i);
71        else $error("FAIL %d (Expected %d, got %d)", i, test_vector[i].n, cc.n);
72      end
73      $finish;
74    end
75  endmodule
```

Listing 9: SystemVerilog source code for the test bench of the fractal point generator

D Source code, vertex array generator, hardware

```
1   //
2   // vag.v, by Per Christian Corneliussen, May 2011.
3   //
4   // Vertex array generator. This module will generate a vertex array
5   // representing one triangle strip of the fractal landscape.
6   // The generated array itself contains fixed-point 16-bit (GLshort) values
7   // only. However, FP32 coordinates specifying a location in the Mandelbrot
8   // set (bottom left coordinates) and a zoom level (given as a step size) must
9   // be provided. See port list.
10  //
11
12  module vag(
13    input           clk,      // Clock.
14    input           reset_n,  // Reset.
15    input    [31:0] left,     // Real value of the bottom left fractal coord.
16    input    [31:0] bottom,   // Corresponding imaginary value.
17    input    [31:0] step,     // Distance between fractal points.
18    input           start,    // Initiate generation of new landscape.
19    input           next,     // Initiate generation of next triangle strip.
20    output reg      ready,    // Output data ready. After 'ready' toggles, a new
21    output   [47:0] arrout    // set of vertex coordinates will be available each
22                              // clock cycle on 'arrout'.
23    );
24
25    // Configurable parameters:
26    parameter NUMPOINTS = 400;  // Number of points in each direction. The number of
27                                // vertices per triangle strip will be 2*NUMPOINTS.
28    parameter NUMUNITS = 2;     // Number of fractal point generators.
29
30    // State encoding:
31    parameter S_IDLE = 0, S_FILL = 1, S_FLUSH = 2, S_READOUT = 3;
32
33    //------------
34    // Variables:
35    //------------
36
37    // Adders:
38    reg  [31:0] add_a;
39    wire [31:0] add_b;
40    wire [31:0] add_out;
41    wire [32:0] add_out_t;
42    wire  [15:0] x_inc, y_inc;
43
44    // Control state machine (CSM):
45    reg [31:0] re, im;          // The current fractal coordinates. Connected to
46                                // c_re and c_im inputs on all FPGs.
47    reg [31:0] re_reg, im_reg;// Registrered fractal coordinates.
48    reg [15:0] x, x_reg;        // The current column. (x vertex coordinate)
49    reg [15:0] y, y_reg;        // The current row. (y vertex coordinate)
50    reg firststrip;             // Whether this is the first strip or not.
51    reg upper;                  // Whether sending the upper or lower vertex.
52    reg [1:0] state;            // Current state.
53
54    // FPGs:
55    reg [0:NUMUNITS-1] fpg_en;
```

```
56  wire [0:NUMUNITS-1] fpg_busy;
57  wire [7:0] fpg_n [0:NUMUNITS-1];
58  wire [15:0] fpg_addr_out [0:NUMUNITS-1];
59
60  // FPG arbiter:
61  reg arbt_enable;  // Whether to process the fractal point or not.
62  reg arbt_ready;   // Will go high for one cc when the fractal point specified
63                    // by the above variables is received and being processed.
64  wire arbt_flushed;// Will stay high when no fractal points are being processed.
65  reg [0:NUMUNITS-1] arbt_fpg_wb, arbt_fpg_wb_next; // Whether an FPG had its
66                                                    // result stored yet.
67  integer i;        // Loop counter.
68
69  // Z memory:
70  reg [7:0] zmem0 [0:NUMPOINTS-1]; // Ram for one row of z coordinates.
71  reg [7:0] zmem1 [0:NUMPOINTS-1]; // Ram for one row of z coordinates.
72  reg zmem_w_en;
73  reg zmem_bufno;
74  reg [15:0] zmem_waddr;
75  reg [7:0] zmem_wdata;
76  wire [15:0] zmem_raddr;
77  wire [7:0] zmem_rdata;
78
79  //----------
80  // Adders:
81  //----------
82
83  // Floating point (FP32) adder. The FP32<->FP33 conversion is done in a
84  // simplified way as in the FPGs. (I.e., dropped support for subnormal numbers.)
85  vithar_lib_fp32_adder_main adder(
86    .a({add_a[31:23], 1'b1, add_a[22:0]}),
87    .b({add_b[31:23], 1'b1, add_b[22:0]}),
88    .res(add_out_t)
89  );
90  assign add_out = {add_out_t[32:24], add_out_t[22:0]};
91
92  // 16-bit integer incrementors.
93  assign x_inc = x_reg + 16'b1;
94  assign y_inc = y_reg + 16'b1;
95
96  //------------------------------
97  // Control state machine: (CSM)
98  //------------------------------
99
100 assign add_b = step;      // The second add operand is always 'step'.
101 assign arrout[47:32] = x_reg; // x coordinate.
102 assign arrout[31:16] = (upper ? y_inc : y_reg); // y coordinate.
103 assign arrout[15: 0] = zmem_rdata; // z coordinate.
104 assign zmem_raddr = x_reg;
105
106 always @ (*)
107 // Combinatorial block for controlling feeding of fractal points to FPGs.
108 begin
109   // Default values:
110   x <= x_reg;
111   y <= y_reg;
112   add_a <= 32'bx;
113   re <= 32'bx;
114   im <= im_reg;
115   arbt_enable <= 0;
116
117   case (state)
118     S_IDLE: begin
119       x <= 0;
120       if (start) begin
121         // If starting on a new frame, send bottom left coordinates to FPGs.
122         y <= 0;
123         re <= left;
124         im <= bottom;
125         arbt_enable <= 1'b1;
```

```
126         end else if (next) begin
127           // If starting on a new strip, increment im with 'step'.
128           y <= y_inc;
129           add_a <= im_reg;
130           re <= left;
131           im <= add_out; // im += step
132           arbt_enable <= 1'b1;
133         end
134       end
135       S_FILL: begin
136         x <= x_inc; // x++
137         add_a <= re_reg;
138         im <= im_reg; // Within a row, the imaginary value (im) will remain constant.
139         arbt_enable <= 1'b1;
140
141         if (arbt_ready) begin
142           // The previous fractal point sent to the FPGs was consumed. Calculate
143           // next fractal point.
144           x <= x_inc; // x++
145           re <= add_out; // re += step
146         end else begin
147           x <= x_reg;
148           re <= re_reg; // Keep old value, waiting for a free FPG.
149         end
150       end
151       S_FLUSH: begin
152         x <= 0;
153         if (arbt_flushed && firststrip) begin
154           // If the FPGs have been flushed and this is the first triangle strip,
155           // start calculating the upper row.
156           add_a <= im_reg;
157           re <= left;
158           im <= add_out;
159           arbt_enable <= 1'b1;
160         end
161       end
162     endcase
163   end
164
165   always @ (posedge clk or negedge reset_n)
166   // State control.
167   begin
168     if (!reset_n) begin
169       state <= S_IDLE;
170       re_reg <= 0;
171       im_reg <= 0;
172       x_reg <= 0;
173       y_reg <= 0;
174       firststrip <= 0;
175       ready <= 0;
176       upper <= 0;
177       zmem_bufno <= 0;
178     end else begin
179       re_reg <= re;
180       im_reg <= im;
181       x_reg <= x;
182       y_reg <= y;
183
184       // Default values:
185       ready <= 0;
186       upper <= 0;
187
188       case (state)
189         S_IDLE: begin
190           // Wait for either 'start' or 'next'.
191           if (start) begin
192             state <= S_FILL;
193             firststrip <= 1'b1;
194             zmem_bufno <= 0; // Always start with the first buffer.
195           end else if (next) begin
```

```
196             state <= S_FILL;
197             firststrip <= 0;
198             zmem_bufno <= ~zmem_bufno; // Change to the other buffer.
199           end else begin
200             state <= S_IDLE;
201           end
202         end
203         S_FILL: begin
204           // Calculate the z coordinates for the current vertex row.
205           if (x == (NUMPOINTS-1)) begin
206             // If at the right end of the row, jump to flush state.
207             state <= S_FLUSH;
208           end else begin
209             state <= S_FILL;
210           end
211         end
212         S_FLUSH: begin
213           // Wait for final fractal points to be calculated.
214           if (arbt_flushed) begin
215             // Done. If started with 'next', jump to the readout state.
216             // Otherwise if started with 'start', change the memory buffer
217             // and continue with the upper row of the triangle strip.
218             if (firststrip) begin
219               state <= S_FILL;
220               firststrip <= 0;
221               zmem_bufno <= ~zmem_bufno;
222             end else begin
223               state <= S_READOUT;
224               ready <= 1'b1;
225             end
226           end
227         end
228         S_READOUT: begin
229           // Vertex array generated. Transmit values out of module.
230           x_reg <= (upper ? x_inc : x);
231           upper <= ~upper;
232
233           if (!upper && (x == (NUMPOINTS-1))) begin
234             // If the vertex following this clock cycle is the final one, jump
235             // back to the idle state.
236             state <= S_IDLE;
237           end
238         end
239       endcase
240     end
241   end
242
243   //------
244   // FPGs:
245   //------
246   // The fractal point generators (FPGs) calculate the iteration counts of
247   // incoming fractal points. (I.e., the z coordinates of the vertices.)
248
249   generate
250     // Generate FPGs.
251     genvar gi;
252     for (gi=0; gi < NUMUNITS; gi=gi+1) begin : UNITS
253       fpg fpgi (
254         .clk(clk), .reset_n(reset_n),
255         .c_re(re), .c_im(im), .enable(fpg_en[gi]), .addr_in(x),
256         .n(fpg_n[gi]), .busy(fpg_busy[gi]), .addr_out(fpg_addr_out[gi])
257       );
258     end
259   endgenerate
260
261   //--------------
262   // FPG arbiter:
263   //--------------
264   // The purpose of the FPG arbiter is to keep track of free FPGs and store
265   // results in the z memory. The variables below are used to interface the
```

```
266   // arbiter by the control state machine (CSM.)
267
268   reg arbt_break1, arbt_break2; // Temporary variables.
269
270   assign arbt_flushed = & arbt_fpg_wb;
271
272   always @ (*)
273   begin
274     // If any of the FPGs are done, write the result to the z memory. If several
275     // FPGs are done, only the first one will have its result written this cc.
276     zmem_w_en <= 0;
277     zmem_waddr <= 0;
278     zmem_wdata <= 0;
279     arbt_ready <= 0;
280     arbt_fpg_wb_next <= arbt_fpg_wb;
281     fpg_en <= 0;
282     arbt_break1 = 0;
283     arbt_break2 = 0;
284     for (i=0; i < NUMUNITS; i=i+1) begin
285       // For each FPG.
286       if (!fpg_busy[i]) begin
287         // This FPG is free.
288         if (!arbt_fpg_wb[i] && !arbt_break1) begin
289           // The result of the previous operation is not yet written to the
290           // z memory. Do it.
291           zmem_w_en <= 1'b1;
292           zmem_waddr <= fpg_addr_out[i];
293           zmem_wdata <= fpg_n[i];
294           arbt_fpg_wb_next[i] <= 1'b1;
295           arbt_break1 = 1'b1;
296         end
297         // Also assign any pending request from the CSM to this FPG.
298         if (arbt_enable && !arbt_break2) begin
299           arbt_ready <= 1'b1;
300           fpg_en <= 0;
301           fpg_en[i] <= 1'b1;
302           arbt_fpg_wb_next[i] <= 0;
303           arbt_break2 = 1'b1;
304         end
305       end
306     end
307   end
308
309   always @ (posedge clk or negedge reset_n)
310   begin
311     if (!reset_n) begin
312       arbt_fpg_wb <= {NUMUNITS{1'b1}};
313     end else begin
314       if (arbt_fpg_wb != arbt_fpg_wb_next)
315       arbt_fpg_wb <= arbt_fpg_wb_next;
316     end
317   end
318
319   //------------
320   // Z memory:
321   //------------
322
323   // Readout from the memories. 'zmem_bufno' is the buffer used for the upper row
324   // of vertices. 'upper' is whether to read a lower or upper row vertex.
325   assign zmem_rdata = upper ?
326                           (zmem_bufno ? zmem1[zmem_raddr] : zmem0[zmem_raddr]) :
327                           (zmem_bufno ? zmem0[zmem_raddr] : zmem1[zmem_raddr]);
328
329   always @ (posedge clk)
330   begin
331     if (zmem_w_en) begin
332       if (zmem_bufno)
333         zmem1[zmem_waddr] <= zmem_wdata;
334       else
335         zmem0[zmem_waddr] <= zmem_wdata;
```

```
336       end
337     end
338   endmodule
```

Listing 10: Verilog source code for the vertex array generator

```
1  //
2  // vag_tb.v, by Per Christian Corneliussen, May 2011
3  //
4  // SystemVerilog test bench for vag.v. As fpg_tb.sv, this is an informal test
5  // bench that simply requests a single landscape from the vertex array generator
6  // and checks the result with a solution calculated earlier with the software
7  // implementation.
8  //
9
10 module vag_tb();
11
12   // DUT ports:
13   logic clk, reset_n, start, next, ready;
14   logic [31:0] left, bottom, step;
15   logic [47:0] arrout;
16
17   // Connect DUT ports:
18   vag dut(.*);
19
20   // Test data:
21   logic [47:0] vertices [0:759]; // 19 strips * 40 vertices = 760
22   assign left = 32'hbefb870e;
23   assign bottom = 32'h3f141b79;
24   assign step = 32'h3c2c7692;
25   assign addr_in = 16'b0;
26
27   // Loop counters:
28   integer i, j;
29
30   // Clock cycle counter:
31   integer ccs;
32   always @ (posedge clk or negedge reset_n)
33     if (!reset_n) ccs <= 0;
34     else ccs <= ccs + 1;
35
36   // Clock driver:
37   always #5 clk = !clk;
38
39   default clocking cc @(posedge clk);
40     default input #1 output negedge;
41     output reset_n;
42     output start, next;
43     input ready, arrout;
44   endclocking
45
46   initial begin
47     // Enable dumping:
48     $vcdpluson(0, dut);
49
50     // Load golden solution from testcase.txt:
51     $readmemh("testcase.txt", vertices);
52
53     // Initiate clock and reset DUT:
54     clk = 0;
55     reset_n <= 0;
56     start <= 0;
57     next <= 0;
58
59     ##1 cc.reset_n <= 1;
60
61     for (i=0; i < 19; i=i+1) begin
62       // For each strip.
63       $display("Requesting␣strip␣%d.", i);
64
65       if (i == 0) begin
66         cc.start <= 1;
67         ##1 cc.start <= 0;
68       end else begin
69         cc.next <= 1;
```

```
70          ##1 cc.next <= 0;
71        end
72        // Wait for ready-signal.
73        $display("Waiting for ready-signal...");
74        @(posedge cc.ready);
75        // Assert answer:
76        for (j=0; j < 40; j++) begin
77          assert (cc.arrout == vertices[i * 40 + j]) $display("PASS %d,%d", i, j);
78          else $error("FAIL %d,%d (Expected %h, got %h)", i, j,
79            vertices[i * 40 + j], cc.arrout);
80          ##1; // Wait one clock cycle.
81        end
82      end
83      $display("Test bench completed after %d clock cycles.", ccs);
84      $finish;
85    end
86  endmodule
```

Listing 11: SystemVerilog source code for the first test bench of the vertex array generator

```
1  //
2  // vag_tb2.v, by Per Christian Corneliussen, June 2011
3  //
4  // SystemVerilog test bench for vag.v. This test bench will test a 400 by 400
5  // fractal landscape consisting of black points only. The point is to measure
6  // performance.
7  //
8
9  module vag_tb();
10
11   // DUT ports:
12   logic clk, reset_n, start, next, ready;
13   logic [31:0] left, bottom, step;
14   logic [47:0] arrout;
15
16   // Connect DUT ports:
17   vag dut(.*);
18
19   // Test data:
20   assign left = 32'hbe67295c;
21   assign bottom = 32'hbe67295c;
22   assign step = 32'h3a94506e;
23   assign addr_in = 16'b0;
24
25   // Loop counters:
26   integer i, j;
27
28   // Clock cycle counter:
29   integer ccs;
30   always @ (posedge clk or negedge reset_n)
31     if (!reset_n) ccs <= 0;
32     else ccs <= ccs + 1;
33
34   // Clock driver:
35   always #5 clk = !clk;
36
37   default clocking cc @(posedge clk);
38     default input #1 output negedge;
39     output reset_n;
40     output start, next;
41     input ready, arrout;
42   endclocking
43
44   initial begin
45     // Initiate clock and reset DUT:
46     clk = 0;
47     reset_n <= 0;
48     start <= 0;
49     next <= 0;
50
51     ##1 cc.reset_n <= 1;
52
53     for (i=0; i < 399; i=i+1) begin
54       // For each strip.
55       $display("Requesting␣strip␣%d.", i);
56
57       if (i == 0) begin
58         cc.start <= 1;
59         ##1 cc.start <= 0;
60       end else begin
61         cc.next <= 1;
62         ##1 cc.next <= 0;
63       end
64       // Wait for ready-signal.
65       $display("Waiting␣for␣ready-signal...");
66       @(posedge cc.ready);
67       // Assert answer:
68       for (j=0; j < 800; j++) begin
69         assert (cc.arrout[15:0] == 80) $display("PASS␣%d,%d", i, j);
```

```
70          else $error("FAIL␣%d,%d␣(Expected␣80,␣got␣%h)", i, j,
71            cc.arrout[15:0]);
72          ##1; // Wait one clock cycle.
73        end
74      end
75      $display("Test␣bench␣completed␣after␣%d␣clock␣cycles.", ccs);
76      $finish;
77    end
78  endmodule
```

Listing 12: SystemVerilog source code for the second test bench of the vertex array generator

References

[1] Dave Shreiner Aaftab Munshi, Dan Ginsburg. *OpenGL® ES 2.0 Programming Guide*. Addison-Wesley Professional; 1 edition, 2008.

[2] John H. Hubbard Adrien Douady. Étude dynamique des polynômes complexes. *Prépublications mathémathiques d'Orsay 2/4*, 1984.

[3] M.F. Barnsley. Fractal image compression. *Notices of the American Mathematical Society*, pages 657–662, June 1996.

[4] Bodil Branner. The mandelbrot set. In *Proceedings of Symposia in Applied Mathematics*, volume 39, pages 75–105, 1989.

[5] Sean Brennan. Fractal zoom. http://www.zettix.com/Graphics/fractal/, 2008.

[6] Dennis M. Ritchie Brian W. Kernighan. *The C Programming Language (2nd Edition)*. Prentice Hall, 1988.

[7] Intel Corp. Intel and floating point. http://www.intel.com/standards/floatingpoint.pdf.

[8] Microsoft Corporation. Visual studio 2005 function reference: clock(). http://msdn.microsoft.com/en-us/library/4e2ess30(v=vs.80).aspx.

[9] David Dewey. Introduction to the mandelbrot set. http://home.olympus.net/ dewey/mandelbrot.html.

[10] Daniel D. Gajski et al. Specification and design of embedded hardware-software systems. *Design & Test of Computers, IEEE*, Spring 1995.

[11] K. J. Falconer. *The geometry of fractal sets*. Cambridge University Press, 1985.

[12] Christiane Fellbaum. *WordNet: An Electronic Lexical Database*. Bradford Books, 1998.

[13] IEEE. Standard for floating-point arithmetic (754-2008), 2008.

[14] ARM Limited. Opengl es 2.0 emulator. http://www.malideveloper.com/developer-resources/tools/opengl-es-20-emulator.php.

[15] ImageMagick Studio LLC. http://www.imagemagick.org/.

[16] Doulos Ltd. Systemverilog assertions tutorial. http://www.doulos.com/knowhow/sysverilog/tutorial/assertions/.

[17] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Co., 1982.

[18] Tom McReynolds and David Blythe. Advanced graphics programming techniques using opengl. *SIGGRAPH '99 Course*, 1999.

[19] Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Professional; 7 edition, 2009.

[20] Frederik Slijkerman. Ultra fractal 5 features. http://www.ultrafractal.com/features.html.

[21] VideoLAN. Vlc media player. http://www.videolan.org/vlc/.

[22] Inc. Wikimedia Foundation. Frame rate - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Frame_rate.

[23] Inc. Wikimedia Foundation. Mandelbrot set - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mandelbrot_set.

[24] Inc. Wikimedia Foundation. Marpat - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/MARPAT.

[25] Inc. Wikimedia Foundation. Parallel computing - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Parallel_computing.

[26] Inc. Xilinx. Ds150: Virtex-6 family overview. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, March 2011.

[27] Inc. Xilinx. Ug364: Virtex-6 fpga configurable logic block user guide. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf, September 2009.