NTNU – Trondheim
Norwegian University of
Science and Technology

Integration of a Fractal Generator with Mali GPU

Per Kristian Kjøll

Master of Science in Electronics
Submission date: June 2012
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

Candidate name: Per Kristian Kjøll

Assignment title: Integration of a Fractal Generator with Mali-GPU

Assignment Text

This proposal describes a possible subject for a project thesis for students with background in microelectronics and computer graphics. The description given here is meant for an autumn project only. A possible extension of the content into a master thesis in the spring may be discussed with the supervisors as the work proceeds.

Abstract

In a recent master thesis Per Christian Corneliussen successfully developed a fractal generator. The purpose of this project thesis is to take this fractal generator, design AXI and APB interfaces to the fractal generator and integrate it into a system with a Mali-400 GPU and finally run a OpenGL demo that uses data structures generated by the fractal generator. A demo has already been implemented by Per Christian, but the student is encouraged to extend it in order to give it a personal touch. The fractal generator will be realized in an FPGA. When the fractal generator is successfully integrated in the system, 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,"[4] a property called self-similarity. A mathematical fractal is based on an equation that undergoes iteration, a form of feedback based on recursion[9].

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 [7]. The point c belongs to the Mandelbrot set if and only if

$|Zn| = 2, foralln = 0, whereZn + 1 = Zn^2 + c \; and \; Z_0 = 0$

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

The master thesis by Per Christian Corneliussen describes a fractal generator for use with a Mali GPU. Moreover it describes a demo program that creates an animated 3D landscape where the height and the colour of the landscape at any given point is determined by the z-coordinate of vertices generated by the fractal generator.

In the first part of this thesis the student should present the Mandelbrot set and give an overview of the fractal generator.

The core of the fractal generator is complete, but it is necessary to also implement a AXI and a APB interface in order to integrate it into a system with Mali-400. This will be the main task in this project. The RTL code must be written in Verilog.

In order to run the demo it will also be necessary to do some minor changes to the software driver in order to configure the fractal generator prior to each frame. The demo itself could also be reworked in order to give the student the possibility of adding a personal touch to the demo.

Finally the system must be synthesized for FPGA, set the fractal generator up to feed the OpenGLES application with datastructures and participate in a demo competition held at ARM Norway.

Co-supervisor:Øystein Gjerdmundnes, ARM Norway AS

Supervisor: Per Gunnar Kjeldsberg, NTNU

Abstract

The Mandelbrot set is a well-known fractal with mathematical properties that can be exploited to create 3D-landscapes. The operations required to calculate a heightmap using the Mandelbrot set are highly parallelizable and is thus suitable for a hardware implementation. Generation of 3D-landscapes,on-the-fly, using the Mandelbrot set is desirable since the Mandelbrot set is infinitely complex[4] and deterministic. This makes possible the creation of many different landscapes with complex patters in, for example, computer games.

A previous master thesis[4] presents a vertex array generator(VAG) that generates the vertices of a 3D-landscape based on an area of the Mandelbrot set. This thesis explores different architectures that connect this vertex array generator with the Mali-400 graphics processing unit(GPU). The result is that the VAG in its current state is not suitable for integration, mostly since it does not support random access to vertices. Thus, a new fractal generator architecture is presented, reusing parts of the VAG.

The new fractal generator is implemented in Verilog and its functionality is verified using the Universal Verification Methodology(UVM). Then, the fractal generator is integrated with the Mali-400 GPU in an FPGA framework and synthesized on FPGA. Tests are also performed at each step of integration.

An OpenGL for Embedded Systems 2.0 demo is written to showcase the functionality of the fractal generator. Changes have been made to the Mali-400 drivers to automatically configure and set-up the fractal generator while the demo is running.

The fractal generator is shown to be working as intended with a scalable performance based on a number of internal cores. Using 64 cores the fractal generator has a worst-case frame time of 51.1 ms at 400Mhz which equals a frame rate of 450 frames pr second, vastly outperforming a software implementation.

The fractal generator is currently limited to creating landscapes of $128 \cdot 128$ points, the intention was to use the demo and driver to increase the resolution but this has not been solved.

Increasing the resolution and optimizing the cache size of the fractal generator has been left for future work.

Sammendrag

Mandelbrot-settet er en velkjent fraktal med matematiske egenskaper som kan brukes for å tegne 3D-landskaper. De matematiske utregningene man trenger for å regne ut høydene til et landskap er svært paralleliserbare og egner seg for implementasjon i hardware. Generering av landskap basert på Mandelbrot-settet er ønskelig siden settet er uendelig komplekst og deterministisk, slik at mange forskjellige landskaper kan bli laget fra settet.

En tidligere masteroppgave beskrev en fraktalgenerator(VAG) som genererte punktene til et 3D-landskap basert på et område av Mandelbrot settet. Denne masteroppgaven utforsker forskjellige hardware-arkitekturen som kan koble VAG til Mali-400 GPU. VAG viser seg å være uegnet for integrasjon med Mali og det blir bestemt at en ny fraktalgenerator skal lages som kan gjenbruke deler av VAG.

Den nye fraktalgeneratoren er implementert i Verilog og dens funksjonalitet er testet med Universal Verification Methodology(UVM). Deretter blir fraktalgeneratoren integrert med Mali-400 og syntetisert på FPGA.

En OpenGL for Embedded Systems 2.0 demo har blitt skrevet for å vise funksjonaliteten til fraktalgeneratoren. Endringer har blitt utførst på Mali-400 driveren for å automatisk konfigurere og sette-opp fraktalgeneratoren mens demoen kjører.

Fraktalgeneratoren fungerer som planlagt med skalerbar ytelse basert på et antall indre kjerner. Ved bruk av 64 kjerner har fraktalgeneratoren i verste fall en frame tid på 51.1 ms ved 400Mhz, noe som utgjør en hastighet på 450 bilder i sekundet. Fraktalgeneratoren viser seg å være mye raskere enn en software implementasjon av fraktalgeneratoren.

Fraktalgeneratoren er begrenset til å lage landskaper med en oppløsning på $128 \cdot 128$ punkter. Intensjonen var å bruke demoen og driveren for å øke oppløsningen men dette fungerte ikke som planlagt.

Å øke oppløsningen og å optimalisere cache-størrelsen på fraktalgeneratoren har blitt satt til framtidig arbeid.

Preface

The problem description for this thesis was originally given for a term project and asked for two interfaces to an existing fractal generator. The existing fractal generator was designed in a previous master thesis given by ARM. During the term project it was decided that the existing fractal generator was not suited for integration with Mali-400 and a new fractal generator should be designed.

The term project began on the work presented in this thesis, specifically it explored different architectures and begun on the fractal generator design. Some of this work is recapped in this thesis to gather the whole fractal generator design process in one place.

The Vertex Array Generator presented in Section 3 was designed by Per Christian Corneliussen in a previous master thesis. The architecture exploration and discussion in Chapter 4 was done in the term project.

The rest of the chapters present the work done for this thesis.

To keep the thesis brief, the chapters does not cover low level implementation details. Some exceptions are made when the details are important to understand the discussions and choices made. If more details are needed, see the appendices for source code.

In addition to this, descriptions of the Mali-400 and its drivers has been limited in some cases due to NDA. The driver source code has been left out for this reason as well.

Thanks to Per Gunnar Kjeldsberg(NTNU) and Øystein Gjerdmundnes(ARM Norway AS) for help throughout the semester.

Contents

List of Figures

List of Tables

1 Introduction

Today, more than half of the world population has a mobile phone. In 2011, 17% of mobile phones were smartphones and the percentage is increasing rapidly[8].

One factor leading to the increased adoption of smartphones has been the improvement in display and graphics technologies[3]. The phone has gone from being just a phone to an entertainment platform with music, video, games and the internet.

Since, among other tasks, 3D-rendering for games can always be done more efficiently on special-purpose hardware than on a general-purpose CPU[3], the increased demand for powerful 3D-rendering in smartphones require new phones to have dedicated graphics processing units(GPU). An example of such a GPU is ARM's Mali-400(Mali), which is for example available on the smartphone Samsung S3.

In computer games and other graphical visualizations it is common to provide a 3D-landscape or terrain for the user to navigate. The height coordinates of a 3D-landscape can be calculated by exploiting the mathematical properties of the Mandelbrot set, a well-known geometric fractal. The computations involved are computationally expensive but highly parallelizable, each point can be processed independently of the others, so the calculations are suited for hardware implementation.

In this thesis, a fractal generator is a component that calculates the height coordinates as explained above. The main motivation for designing a fractal generator in hardware is 3D-landscape generation. A fractal generator can generate landscapes on-the-fly, requiring little or no memory between frames. Since the Mandelbrot set is infinitely complex[4] and deterministic, the generator can be used to draw a large amount of different landscapes. Another usage of a fractal generator is to draw aesthetically pleasing 2D images.

1.1 Thesis Objectives

The main objective in this thesis is to design a fractal generator and integrate it with Mali on an FPGA. The fractal generator shall be able to accelerate the generation of a 3D-landscape, where the 3D-landscape is based on the Mandelbrot set and animated using OpenGL ES. The fractal generator must be implemented in Verilog and shall communicate with Mali using the ARM

APB and AXI bus communication protocols. Figure 1 illustrates how a fractal generator could be connected to a Mali system.

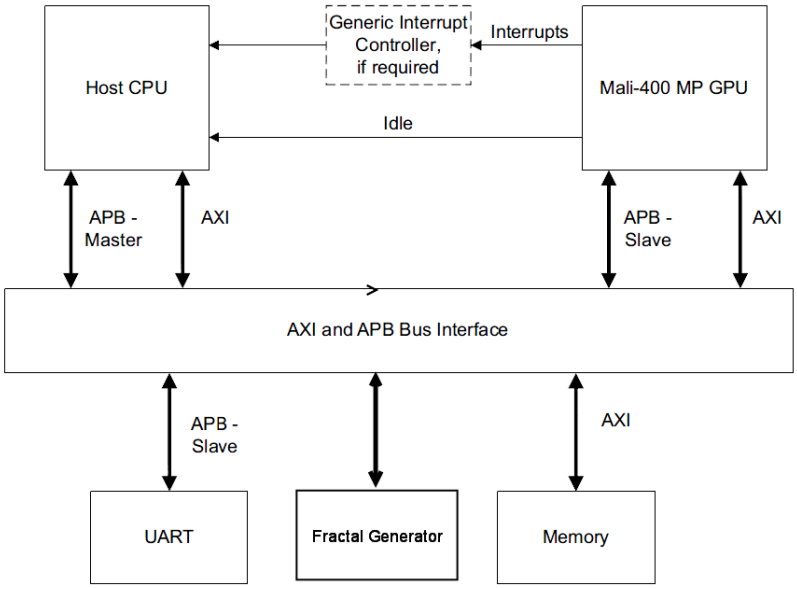

Figure 1: An example system with Mali and the fractal generator. Produced from [13].

The secondary objective in the project is to alter the Mali drivers, such that control of the fractal generator is performed automatically by the driver, based on input from the OpenGL ES demo.

1.2 Thesis Outline

The thesis starts with background theory and previous work.
Chapter 2 will explain the concepts, terms and components needed to understand the later chapters of the thesis.

Chapter 3 describes the task of the fractal generator in detail and presents a previous fractal generator implementation.

Chapter 4 explores different hardware architectures that integrate a fractal generator with Mali, discusses advantages and disadvantages, and ultimately selects an architecture to implement in Verilog.

Chapter 5 presents the chosen fractal generator design and all its modules in detail. Alternative choices are also discussed briefly for each module.

Section 5.7 verifies the behavior of the fractal generator hardware using the Universal Verification Methodology(UVM).

To avoid difficult debugging, it is necessary to ensure working hardware on FPGA before starting on the driver.
Chapter 6 describes the methodology used to achieve this. It connects the fractal generator to Mali, integrates the whole system with an FPGA framework and synthesizes it. Tests are performed at each step.

Chapter 7 presents the OpenGL ES demo, written to showcase the functionality of the fractal generator, and how it communicates with the driver to control and set-up the fractal generator prior to each frame. Furthermore it presents the Mali driver architecture and the changes made to enable software control of the fractal generator.

Chapter 8 is the profiling chapter, it examines the performance of the fractal generator and compares the hardware accelerated version with other solutions.

Chapter 9 concludes the thesis, it discusses if using the fractal generator increased performance, and if the fractal generator is useful for any practical purposes.

2 Background Theory

This chapter explains the terms and theory needed to understand the concepts, descriptions and discussions used in the rest of the thesis.

2.1 The Mandelbrot Set

The Mandelbrot set is a fractal set of complex numbers c. The set is formally defined by the iterative equation

$$Z_{n+1} = Z_n^2 + c, \qquad c, Z \in C \tag{1}$$

where the number c is part of the Mandelbrot set if the equation remains bounded when $n- > \infty$. It can be shown that if $(|Z_{re}| > 2)$ || $(|Z_{im}| > 2)$, the equation will diverge [2, p. 81]. Thus, to examine if a point is in the Mandelbrot set or not, one has to iterate the equation until either the real or imaginary part of Z exceeds two.

The Mandelbrot set is named after Benoit B.Mandelbrot and is infinitely complex[5, p. 197] and connected[10]. If the Mandelbrot set is plotted in a two-dimensional coordinate system as in Figure 2,using the real and imaginary part of c as its x- and y-coordinates, it shows a boundary with a distinctive and easily recognizable two-dimensional fractal shape.

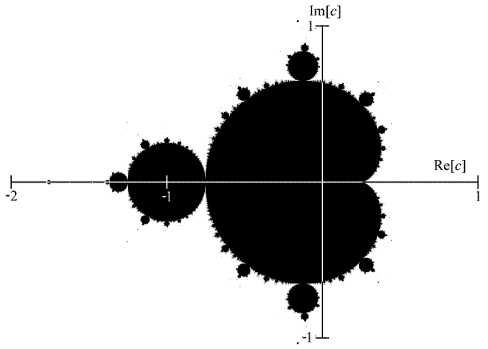

Figure 2: The Mandelbrot set[4].

By coloring points in the aforementioned 2D coordinate system with a gradient, based on the number of iterations required to determine if the point is in the Mandelbrot set or not, the Mandelbrot set reveals a complex and aesthetically pleasing structure. Zooming in on specific areas of the set shows self-similarity and high detail as in shown in Figure 3.

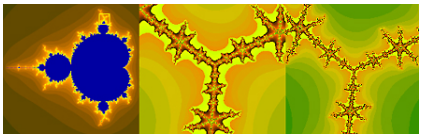

Figure 3: The Mandelbrot Set with color and zoom[9].

Since many points require an infinite number of the above iterations, it is necessary to set an iteration limit when drawing the Mandelbrot set in practice. This limit will affect the detail of the resulting image(Figure 4).

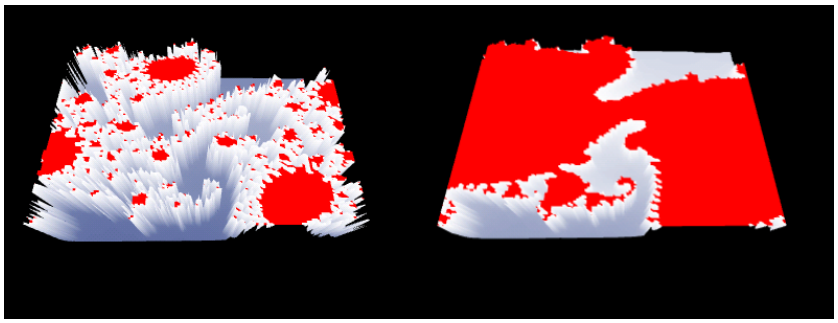

Figure 4: Two images of the mandelbrot set with different iteration limits.Left = 160 iterations, right = 80 iterations. Points that hit the limit are colored red.

The Mandelbrot set can be used to create 3D landscapes: Instead of, or in addition to, using the number of iterations to color each point, as explained above, the number is used as a third coordinate z. The z-coordinate represents the height of the landscape at that point. For example, the point C_1 in Listing1 would get a z-coordinate of 80(the iteration limit). Since zooming in on the Mandelbrot constantly reveals new patterns and more complexity, the set can be used to generate a large number of unique landscapes.

```
1 MAX_ITERATIONS = 80
  C_1 = (0,0) = 0 + i0
3 Z_0 = 0
  Z_1 = Z_0^2 + c = 0
5 ...
  Z_80 = 0
7 Z is still bounded when the algorithm reaches the iteration
    limit, so (0,0) is part of the Mandelbrot Set.
```

Listing 1: Example of iteration

2.2 OpenGL ES 2.0

OpenGL ES is an Application Programming Interface(API) for 3D graphics in embedded systems(ES). It is based on the widespread desktop-API OpenGL, and aims to be smaller and optimized for constrained devices such as mobile phones[6]. In short, the API is a portable and fast software interface to graphics hardware[7]. This thesis will use OpenGL ES 2.0 when coding a demo(Section 7.1) for use with the fractal generator.

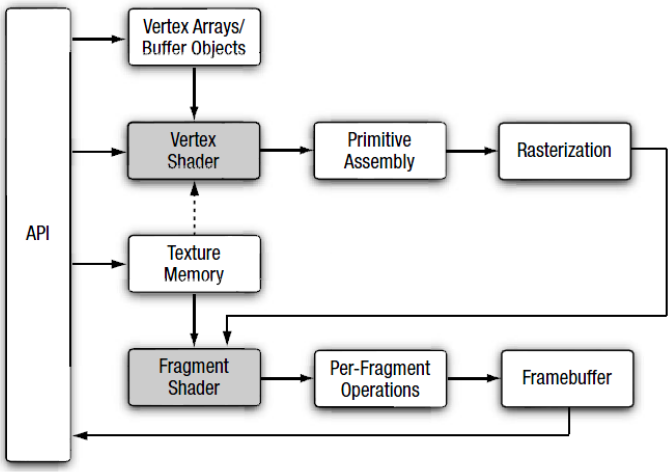

Figure 5: The OpenGL ES 2.0 Graphics Pipeline [6].

OpenGL ES 2.0 implements the graphics pipeline in Figure 5. The basic function of the API, or pipeline, is to project vertices represented in three-dimensional virtual space onto a two-dimensional screen. In addition to this, the API facilitates vertex transformation prior to the projection. For example, the **vertex shader** can transform the vertex positions, compute the lighting at each point and more. Since many of these transformations are done using expensive matrix operations, special hardware(GPUs) is used to accelerate the API calls.

When using OpenGL ES functions in an application, a call is made to the OpenGL ES driver. The driver passes information, data structures and hardware control register settings, to the GPU(if there is one) [19] [16]. Section 2.3 explains how the Mali GPU uses this information to draw frames.

A brief explanation of some OpenGL ES concepts:

Primitives:
When OpenGL ES draws an object it constructs it by combining groups of vertices into primitives[7]. The most common primitive is the triangle, and there are several drawing modes that uses the triangle as a primitive[4]. If an array contains three vertices,i.e. nine coordinates((x,y,z)*3), and it is drawn with the GL_TRIANGLES mode, it will be drawn as a single triangle. Adding three more vertices will add another triangle.

Triangle Strip:
When drawing a second triangle primitive; instead of drawing a new triangle by adding three vertices as above, a new triangle can be drawn by only adding one more vertex and draw lines to it from the first triangle. The GL_TRIANGLE_STRIP mode exploits this fact and avoids drawing all the vertices shared between triangles multiple times[6][7]. However, to keep the concept of primitives, some must still be drawn twice.

Vertex buffer object(VBO):
Vertex buffer objects allow OpenGL ES applications to allocate and cache vertex data in high performance graphics memory[6]. There are two types of buffer objects; *array buffer objects* contains vertex data(the coordinates), and *element buffer objects* describes the order in which to connect the vertices to create an object. Element buffer objects are also called index arrays, which are used throughout this thesis. All OpenGL ES vertices in this thesis are stored in VBOs.

2.3 The Mali-400 GPU

The Mali-400 MP GPU(Mali) is a hardware accelerator for 2D and 3D graphics systems. The GPU implements a graphics pipeline supporting the OpenGL ES and OpenVG APIs[16]. The main processing units in the GPU are the geometry processor(GP) and pixel processors(PP). The pipeline is divided into two main jobs, a GP job and a PP job. Each job performs specific parts of the pipeline, with the GP job doing the vertex shading and the PP job doing the fragment shading.

Below is a brief explanation how the Mali-400 GPU reads and process vertex data to draw geometry.

As described in the above section the OpenGL ES driver create data structures in memory for Mali and configures the hardware prior to each scene(frame). Following this step, the Mali geometry processor transforms each vertex with the instructions in a vertex shader program[16]. This vertex shader pro-

gram is written by the user and loaded in the OpenGL ES application. The OpenGL ES driver compiles the program into a command list for the Mali vertex shader.

The Mali vertex shader can have several input streams of vertex data where each stream is read from memory by the vertex loader component.Vertices from each stream can be transformed, moved or combined in the 3D space. The shader can also add lightning or change the perspective of the vertices.

After the vertex shader there are several more steps in the graphics pipeline. However, they are not relevant for this thesis and will not be explained here. See [16] for more information about each stage. See[6] or [7] for a more thorough explanation of the shader language.

2.4 The AMBA AXI Protocol

The Advanced Extensible Interface(AXI) protocol is a part of the Advanced Micro controller Bus Architecture(AMBA) family. The AXI protocol is a communication protocol suitable for high-performance, high-frequency system designs[17].

The AXI protocol uses separate address/control and data phases. The protocol supports unaligned transfers, burst transactions, multiple outstanding addresses and out-of-order transactions[17]. Figure 1 shows the bus structure of an example system using the AXI and APB protocols.

The vertex loader component in Mali uses the AXI protocol. It reads vertex data in transactions of 32bytes. The external AXI bus width from Mali is 128 bits. This means that the AXI requests from the vertex loader will result in incremental bursts with two 16-byte(128bit) transfers in each burst. The vertex loader is used to read data from the fractal generator.

2.5 The AMBA APB Protocol

The APB protocol is a communication protocol optimized for low power consumption and reduced interface complexity. The APB protocol can either be used with low-bandwidth peripherals that require less performance than the AXI protocol, or, it can be used to program control registers of peripheral devices[14].

In Mali, the APB protocol is used to configure internal control registers. Note

that there is no common external APB bus on the GPU, each APB interface is instead given a range of addresses on the AXI bus. Reads or writes to this range are converted to 32-bit APB signals ahead of the APB interfaces[1].

2.6 The PL301

When connecting several AXI slaves and one AXI master to the same AXI bus there needs to be a way to determine the intended target of an AXI request. This can be done by giving each slave a range of AXI addresses. ARM has a component called PL301 that can be inserted onto the bus and perform this type of address management. The address mapping in the PL301 can be configured using the ARM software AMBA-designer. The software outputs the complete Verilog code of the configured component.

2.7 Hardware Verification

This section will explain the terms and techniques that will be used to verify the functionality of the fractal generator in Section 5.7.

Simulation-Based Verification is the most commonly used verification approach [12]. Simulation-based verification uses a test bench to apply input stimuli to, and record output from, a design. The output from the design is then compared to a reference output. Thus, simulation-based verification is a form of verification by redundancy [12].

Directed testing: Using the directed test approach, a list of tests that each concentrate on a set of specific features are obtained from the hardware specification. The list of tests is then used as a verification plan[22]. Test bench stimuli vectors are written manually to exercise this test, and examine the specific features, in the device-under-test(DUT). Once the test works, it is marked as successful in the verification plan, and the procedure is repeated for the next test[22].

Constrained random testing: In a complex design, the directed testing approach consumes a lot of time and resources. When the complexity doubles, it takes twice as long, or twice as many people to complete[22]. A faster methodology is needed in order to obtain a high amount of coverage.

[1]For example, AXI requests in address range 0x0000-0x1000 could be routed to the APB interface on the geometry processor.

Constrained random testing(CRT) can help eliminate much of the manual nature of directed testing[1]. The CRT approach adds a component to the test bench which automatically generates random stimuli and applies it to the DUT. The stimuli can be constrained to emphasize certain aspects of the DUT, and to avoid illegal inputs to the design[12].

Coverage: When a test bench using CRT is is randomly testing the design states of a DUT, two important questions are: What have been tested by this stimuli? Have the test bench verified enough? [12][22] When using either CRT or directed testing, the answer to these questions, the verification progress, can be gauged using coverage.

Functional coverage is a measure of which design features have been exercised by the tests [22], it measures the implementational completeness and correctness of the functions obtained from the design specification[12]. It is closely tied to the design intent and is sometimes called specification coverage.

Code coverage provides insight into how thoroughly the code of a design is simulated by a test bench[12], it is the easiest way to measure verification progress [22] and can be measured automatically by simulator tools.

The Universal Verification Methodology The Universal Verification Methodology(UVM) is a complete methodology that codifies the best practices for efficient and exhaustive verification[21]. The UVM is implemented as a System Verilog class library, and consists of several object-oriented, reusable, UVM verification components(UVC).

The UVM and libraries provide:

- Infrastructure to partition the verification environment into specific, hierarchical and reusable components. The infrastructure also streamlines the creation of test bench environments.
- Built-in functions to perform common activities like printing, comparing and packing items.
- Module- and system level stimulus generation, where data can be randomized and configured according to the system state.
- Incorporating functional coverage and data checks using best-known practices.

3 The Vertex Array Generator

The background theory states that objects drawn in OpenGL ES are passed to the vertex shader as a stream of vertices. The vertex shader can then perform transformations on the stream. Thus, in order for Mali to do transformations on the landscape created by the fractal generator, the landscape vertices has to be provided to the vertex shader. The actual job of the fractal generator(in hardware) can be now be more precisely formulated. It has to do two specific tasks:

1. Calculate the number of iterations of each point in a given landscape with the Mandelbrot set equation.
2. Provide the landscape to the Mali GP as a stream of vertices.

This chapter presents a hardware fractal generator from a previous master thesis [4] and explains how it generates a stream of vertices. To avoid confusion with the fractal generator design in later chapters, the fractal generator in this chapter is from now on referred to as the vertex array generator(VAG). The discussions in Chapter 4 will use the VAG as a reference point [2].

The Vertex Array Generator The vertex array generator(VAG) implements the following algorithm to calculate the Mandelbrot set.

1. Take as input an area of the Mandelbrot set.
2. Calculate the height(using equation 1) of all vertices in a single row of the area, starting with the bottom row.
3. Store the row as a triangle strip of vertices(as explained in Chapter 2).
4. Triangle strip is ready, signal Mali for retrieval.
5. Transmit strip and go back to step two, calculating all the rows from bottom to top until the entire area is calculated and transmitted as triangle strips.

The VAG is implemented in hardware by several components as shown in Figure 6, each component and their roles in the algorithm is described below.

[2] Chapter 4 explores different architectures that integrates a fractal generator with Mali.

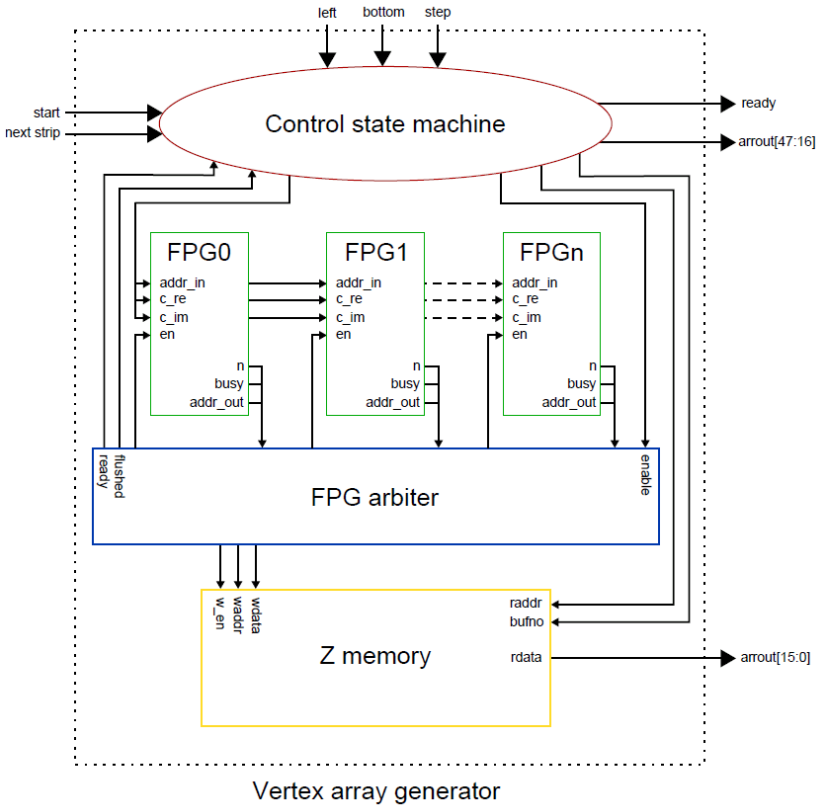

Figure 6: The Architecture of the VAG. Taken from [4]

The control state machine(CSM) controls the execution of the algorithm. It takes as input three values, left, bottom and step size. These values, together with $NUMPOINTS$ described below, define which area of the Mandelbrot set to draw. Left and bottom is the coordinates to the bottom left point in the area, step size is the distance between points. After receiving the area, the CSM calculates the xy- coordinates of each vertex point in the current strip of the area, these coordinates are then fed to the fractal point generators(FPGs).

The fractal point generator(FPG) does the calculations for each vertex at step two in the algorithm. It takes as input the x- and y- coordinates of a pixel, use the coordinates as the real and imaginary values of the point c in equation 1, and returns the z-coordinate; i.e., how many iterations it takes to determine if the pixel is in the Mandelbrot set or not. The VAG can contain several FPGs to calculate z-coordinates in parallel.

The FPG arbiter decides which of the FPGs that get to do calculations on a xy-coordinate from the CSM, it keeps track of which FPGs are busy and free. The FPG arbiter is also responsible for storing results from the FPGs into the Z-memory.

All the vertices of the current triangle strip is stored in the Z memory. The Z memory consists of two buffers with space to contain $NUMPOINTS$ vertex coordinates(x,y,z). $NUMPOINTS$ is a constant parameter set ahead of runtime, where the entire fractal has a resolution of $NUMPOINTS \cdot NUMPOINTS$. The CSM keeps track of which buffer to store coordinates in and avoids recalculating z-coordinates that was calculated the previous triangle strip.

All coordinates in the VAG are represented by 16bit GLfloat values. This datatype was chosen instead of the 32bit equvialent to save bandwith when transferring the vertices to Mali.

4 Architecture Exploration

This chapter will examine different architectures that integrate the fractal generator and Mali. The chapter will discuss the advantages and disadvantages of each architecture, and ultimately use the discussion to choose a fractal generator design for Verilog implementation.

4.1 The Vertex Array Generator as an AXI Slave

As described in the previous chapter, the vertex array generator(VAG) outputs its vertex data using triangle strips. Drawing a frame is done strip by strip from bottom to top. One possible way to integrate the vertex array generator with Mali is to connect the VAG as a slave to the AXI bus from the Mali L2 cache(Figure 7). Mali can then request strips one at the time from the VAG with, slightly modified, AXI requests. The area of the Mandelbrot set to draw from could be specified by extending the VAG with an APB slave. This architecture is a very memory efficient design since the all

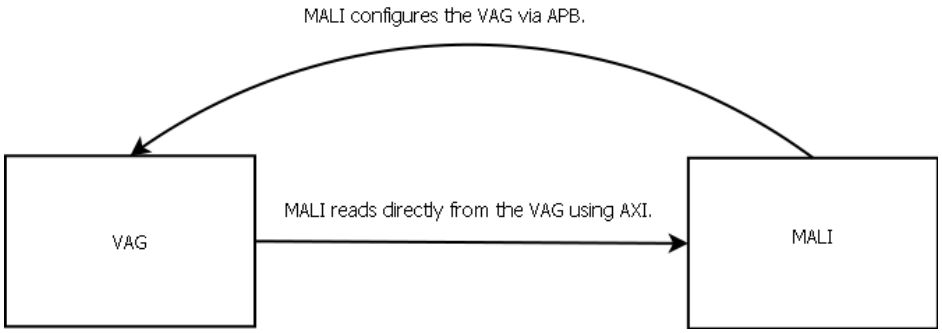

Figure 7: The Vertex Array Generator as an AXI slave.

the coordinates(x,y and z) are transferred directly, i.e, not being stored in memory ahead of Mali(Mali will store data later though). There are however many disadvantages to this integration.

The main disadvantage with this architecture is that Mali has no say in which order the vertices are incoming, the AXI address does nothing, Mali has to read strip by strip. This is unacceptable, since the AXI bus from the L2 cache does not always request data in the same order. Mali would have to buffer all the vertexes internally and wait for the correct vertex before continuing. Thus, the design needs to support random access to the z-coordinates.

Another disadvantage is that Mali would have to check for a new strip very often, and be ready to receive data constantly, otherwise the VAG would stop when the Z memory got full. Polling like this would consume time from the L2, and the VAG would be a bottleneck for the overall performance of the system. An improvement would be to transfer data from the VAG to a buffer connected to the bus, like a cache, and then transfer data in bursts. Unfortunately, the vertex shader could still end up waiting really long for a coordinate, if it was not included in the previous burst.

4.2 The Vertex Array Generator with DMA

Another way to integrate by using the vertex array generator is to make it a master on the AXI bus and transfer data to the system RAM instead of Mali(Figure 8). With this solution Mali can ask for data from the RAM when it chooses, and random access would be supported, when the VAG is finished with a frame.

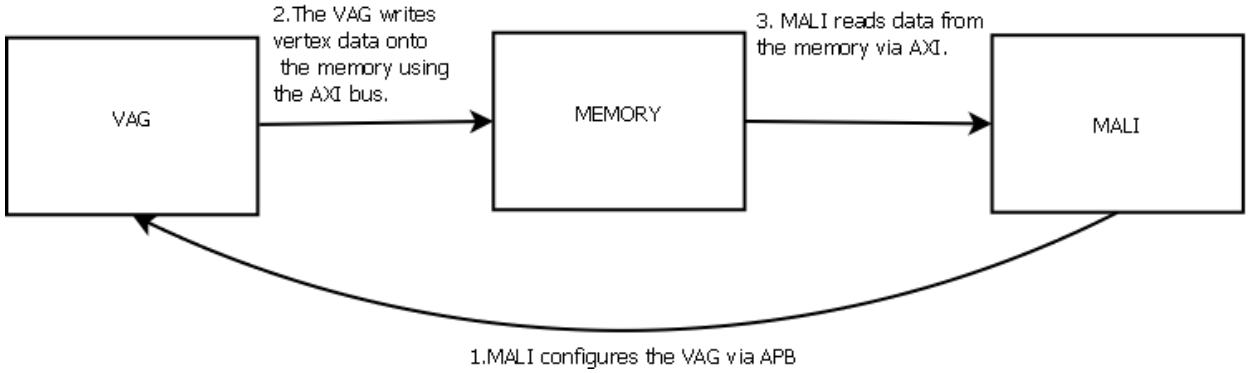

Figure 8: The Vertex Array Generator as an AXI master(DMA).

However, the arrays would still come in order and, if a Mali requests a vertex not yet in the RAM, it could take a long time before the vertex is ready. In addition this would require double traffic on the RAM bus, since data would be transferred to the RAM from the VAG, and then to Mali from the RAM. Bus traffic is already a bottleneck in the Mali system(source) so this is a large deterrent.

Another big disadvantage to using the VAG, in both the previous methods, is that the VAG transfers all the coordinates of a vertex(x,y and z) via the triangle strips. This is unnecessary since the xy-coordinates of the Mandelbrot set are not needed to draw the landscape, they are only needed to calculate

the height at each point. Instead, a flat landscape with static xy-coordinates, stored in memory, can be used as a frame for all the fractal landscapes. When drawing the landscape, the z-coordinates are read from the fractal generator and the xy-coordinates are read from the static landscape. As long as the total number of points are the same, and the z-coordinates are placed in the correct order, the landscape will be correct.

Using static xy-coordinates is a big advantage since it greatly reduces the traffic between the fractal generator and Mali. Thus, the xy-coordinates that are transferred to Mali from the fractal generator should rather be stored by using vertex buffer objects in OpenGL ES. In theory, this would enable Mali to store the entire static landscape in cache. In addition to this, the saved bandwidth from the static xy-coordinates can be used to change the representation of the coordinates from 16 to 32bits. This will allow the fractal generator to zoom in further on the Mandelbrot set before losing detail from rounding of the decimal numbers.

One last disadvantage for the VAG is that the VAG forces Mali to use the GL_TRIANGLE_STRIP method when drawing the landscape with OpenGL ES. This is mostly a theoretical disadvantage, since triangle strips are good for landscapes and one would rarely wish to use another drawing method.

4.3 Fractal Generator With Cache

As discussed in the above sections, there are several disadvantages when connecting the vertex array generator with Mali. This section will present a new fractal generator that does not use the VAG. The new fractal generator will use the discussions above as a basis. It will reuse the good elements of the VAG and redesign the bad.

The new design will use the fractal point generators from the VAG to calculate the z-coordinates. No disadvantages have been connected to this component. It will also use the same method as the VAG to describe an area of the Mandelbrot set, except that it will use the xy-coordinates of the top-left corner and step size between points. This is instead of the bottom-left corner and seems a more intuitive way to define an area, mostly personal preference. The initial xy-coordinates will be referred to as x0 and y0 throughout the thesis. The rest of the fractal generator design is new and is made to avoid the disadvantages discussed above.

The new fractal generator is connected to the Mali AXI bus as an AXI slave.

The fractal generator consists of four main components described below. The components are connected, and draw a frame, as described in Figure 9.

- The APB interface: This interface allows an APB master to configure hardware registers inside the fractal generator. The registers decide which area of the Mandelbrot set the z-coordinates will be calculated from.
- The coordinate cache: A cache where the z-coordinates are stored as they are calculated, the cache is built up by several RAMs pasted together.
- The AXI interface: This component is responsible for answering AXI requests. Based on the address of a request, the AXI interface tries to read the corresponding z-coordinates from the coordinate cache. If the coordinates are not there, the AXI interface signals the arbiter to calculate them. After reading the coordinates they are transmitted on the AXI bus.
- The arbiter: This module uses the values of the APB registers to calculate the x- and y-coordinates of all the vertices in the chosen area of the Mandelbrot set. These coordinates are then *arbitrated* to a parameterized number of fractal point generators[4]. The FPGs calculate the number of iterations for each point and the resulting z-coordinates are stored in the coordinate cache.

The landscapes that the fractal generator generate consist of a static number of vertices. The number is set by a parameter, $NUMPOINTS$, and is equal to $NUMPOINTS^2$. Another parameterized value in the design is the number of FPG units inside the arbiter, this value decides how many z-coordinates that can be calculated in parallel. These parameters are the same as used in the VAG [4] and are presented further in Section 5.1.

This architecture has several advantages when compared to the ones using the VAG. Most importantly it supports random access of vertices by AXI addressing. Furthermore, continually calculating coordinates and using a cache means that Mali can request coordinates when it wants to, without affecting the performance of the fractal generator. No polling is required. In addition to this, the architecture saves bandwidth since it only transfers z-coordinates, and since it doesn't use hardware triangle strips it is not limited to using GL_TRIANGLE_STRIP in OpenGL ES.

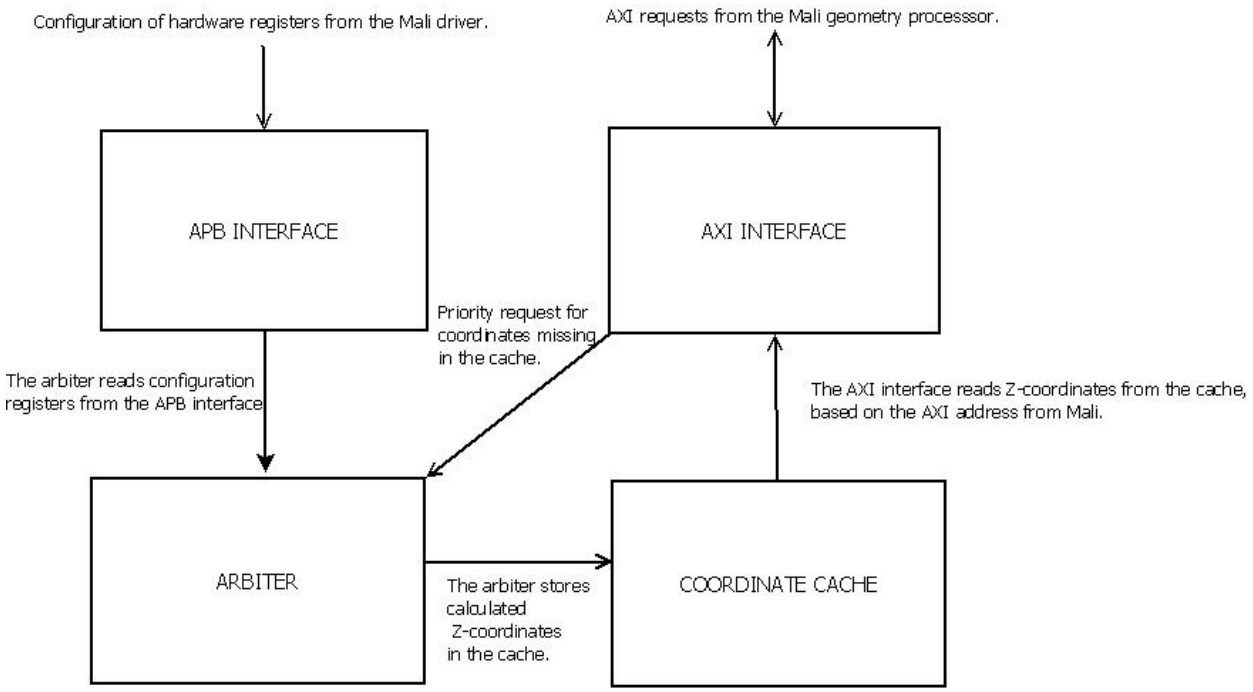

Figure 9: Structure of the Fractal Generator and how it draws a frame.

This architecture combats all of the disadvantages seen when using the VAG. It has been selected for implementation in Verilog. A detailed description of the design is presented in the next chapter.

5 Fractal Generator Design

The previous chapter chose a hardware architecture for the fractal generator and explained its general functionality(Section 4.3). This chapter describes the functionality of the fractal generator in detail by presenting the fractal generator's submodules. In addition to this, the chapter discusses some alternative implementations of each module and explains why the chosen implementation has been selected.

Section 5.1 presents the configuration parameters of the fractal generator and the data types used to represent coordinates used by the fractal generator.

Section 4.3 states that the AXI interface tries to read z-coordinates from the cache based on the received AXI address. This requires a predefined relationship between AXI addresses and each vertex(x,y,z) in the fractal landscape. A predefined relationship enables Mali to request the z-coordinates of specific xy-pairs by setting the current AXI address to the address that is mapped to the wanted coordinate.
Section 5.2 explains how the fractal generator maps the vertices internally and how to translate from an AXI address to a given vertex, the actual translation is done by the AXI interface.
The following sections explain the functionality and Verilog implementation of each submodule of the fractal generator(Figure 9).

5.1 Configuration Parameters and Data Types

This section describes the three configurable parameters in the fractal generator. It also describes the data types used by the fractal generator.

NUMPOINTS This parameter sets the amount of total points in the fractal made by the fractal generator. The fractal generator creates a square fractal of $NUMPOINTS \cdot NUMPOINTS$ z-coordinates. This parameter together with the APB configuration registers decide which area of the Mandelbrot set to cover with the fractal. The $NUMPOINTS$ parameter is limited to be a power of two, this is done to reduce calculations with the parameter to bit-shift operations.

The current OpenGL ES demo uses one index array to describe a fractal. Since the index array is limited by the data type unsigned short, the maximum value of $NUMPOINTS$ with this implementation is

128.[3] Unless drawing very small landscapes, 128 is recommended as the standard value for this parameter. The resolution of the final landscape is further discussed in Section 7.3.1.

NUMUNITS This is the number of FPGs in the system. The parameter configures the amount of z-coordinates that can be calculated in parallel. Increasing this parameter will increase the performance of the fractal generator at the cost of area. Chapter 8 examines the performance of the fractal generator with varying $NUMUNITS$.

MAX_ITERATIONS This is the iteration limit the FPGs use when determining if a point is in the Mandelbrot set or not. If the iterations reach this limit, the point is defined as part of the Mandelbrot set.

Lowering this number will increase the performance of the fractal generator, since each point will be finished faster by the FPGs. However, it will also reduce the detail of the produced image, since the maximum height is limited. The points that would've gotten higher iterations will all have the same height and color. Chapter 8 will show an example of this.

Data Types As mentioned in Section 4.2, the amount of bits used to represent coordinates in the Mandelbrot set affect the maximum zoom level before losing detail. This loss of detail is caused by rounding decimal numbers when they reach the limits of their data types. Increasing the size of the data types, e.g. from 16bit to 32bit floating point(fp32) values, increases the details of the landscape at the cost of bandwidth and storage[4]. Since many interesting aspects of the Mandelbrot set appear at large levels of zoom, it was decided to use fp32 values when calculating on the Mandelbrot set internally.

This means that the inputs to the APB interface are 32 bits for the initial coordinate(x0,y0) and step size. In addition to this, the internal calculations on the fractal landscape are done with fp32 operations. Since fp32 calculations are done in one cycle, the internal performance is not affected by choosing this data type, but storing the temporary coordinates take more space. See Section 5.5 for more information about the calculations.

The output of the fractal generator are the z-coordinates of the landscape.

[3]Since the vertex array with $NUMPOINTS = 256$ would have indexes above the range of unsigned short

[4]The thesis uses the IEEE754 binary32 standard to represent numbers in fp32.

Since the z-coordinates are the number of iterations in the Mandelbrot set equation, they can be represented as integers without loss of detail. Thus, the z-coordinates are stored as integers internally.

However, the programming language used to write the vertex-shader and fragment-shader(Section 7.1), does not allow a stream of integers as input [11, p. 31]. It only allows floating point numbers. Thus, the z-coordinates are converted to fp32 on the output of the cache. It would be better to convert the coordinates to fp16, but since the limitation by the shader language was discovered late in the design, and ARM had finished unsigned integer to fp32 converters, fp32 it was chosen instead. This causes a bit of performance loss since reading data requires twice as many bursts. Since it does not affect the internal performance, and the internal performance of the fractal generator will be the bottleneck in the system[5], converting to fp16 is left as future work.

5.2 Relationship Between AXI address and Coordinates of a Fractal

As explained above there needs to be a predetermined method of translation between an AXI address and a given fractal vertex, this section explains how the translation is performed.

Internally, the fractal generator maps the z-coordinates to the address range 0 to $NUMPOINTS^2 - 1$(Figure 10). The first row of the fractal is given addresses 0 to $NUMPOINTS - 1$, with increasing address from left to right. There is a total of $NUMPOINTS$ rows where each row is divided into addresses like the first. Thus, the address range of row n is given by the below equation.

$$row\ n\ address\ range = [n * NUMPOINTS\,(n * NUMPOINTS) + NUMPOINTS - 1)$$
$$where\ 0 <= n <= NUMPOINTS - 1 \tag{2}$$

This address setup enables easy translation from an address to a specific fractal vertex: The row of the vertex is found by $address/NUMPOINTS$ and the column is found by $address\ mod\ NUMPOINTS$. Using the parameters set via the APB interface, the column and row can be used to calculate the exact x- and y-coordinates(Section 5.5), which then can be used to calculate the z-coordinates with the Mandelbrot equation(1. To ensure fast address translation in hardware, $NUMPOINTS$ is limited to be a power of

[5]Mali only has to read data while the fractal generator is calculating.

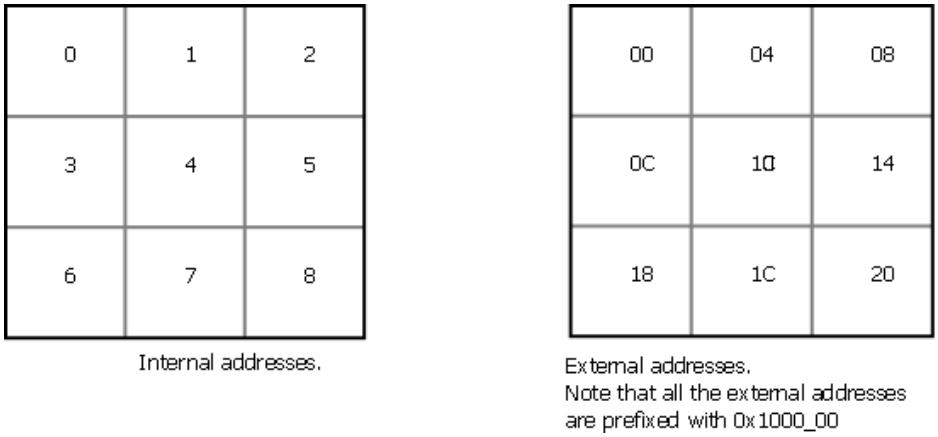

Figure 10: Internal and external AXI address mapping of the vertices in a 3x3 fractal.

two, which reduces the calculations of x and y to shift operations. See the arbiter source code for implementation, and see Section 5.1 for setting the $NUMPOINTS$ parameter.

Externally, the AXI bus uses byte addressing and the fractal generator is mapped to a specific address range on the bus. Otherwise the mapping is the same as the internal; coordinates are address mapped in row major order(Figure 10) [6].

The AXI Interface is the only module that sees the external requests from Mali, and it has to convert the addresses to the internal format before communicating with the other modules. The conversion is done by simply ignoring the left most bits of the AXI address, and dividing the resulting number by four(by shift). The leftmost bits can be removed since they are used to select components on the AXI bus, and this selection is performed ahead of the AXI Interface. Since each coordinate is four bytes, the internal address can then be found by dividing by four.

5.2.1 Discussion

Why not use the external mapping internally and remove the conversion? Using the external mapping as the internal mapping and removing the conversion would simplify the AXI interface. However, this would complicate

[6]The mappings are not equal by coincidence, the external mapping is set by the index array in the OpenGL ES demo.

several of the internal calculations in the fractal generator. For instance, the arbiter uses the address to calculate the row and column of the xy-coordinate in the Mandelbrot set(Equation 1 on page 4). To calculate this with the external mapping, one would have to both remove the leftmost bits and divide by four(to find the column). This operation is identical to the address translation so nothing is saved.

Another example is in the coordinate cache, the cache use the address to determine which rams to read and write from. To properly determine the ram column, the address would have to be converted in this component as well. Thus, it was chosen to separate the internal and external mappings.

No other internal mappings have been considered since no drawbacks have been connected to the current one. One could switch to column major addressing in both hardware and the demo, but the performance would be the same.

5.3 The APB Interface

The APB interface enables an APB master, e.g. the Mali driver, to configure hardware registers in the fractal generator. The registers control which area of the Mandelbrot set that is used by the fractal generator when drawing a frame. Changing the contents of these registers thus change the landscape drawn by the fractal generator.

5.3.1 Verilog Implementation

The APB interface is implemented in Verilog as a finite state machine. When the interface detects an incoming APB write, it stores the incoming data in one of the following registers. The destination register is determined by the APB address.

- The x0-coordinate register, which determines the x coordinate, in the Mandelbrot set, of the top left point of the fractal.
- The y0-coordinate register, which determines the y-coordinate, in the Mandelbrot set, of the top-left point of the fractal.
- The step size register, which determines the distance between each point in the chosen area of the Mandelbrot set. The step size value determines the zoom-level of the area to be drawn from.

After storing data in one of the registers, the interface signals to the APB master that data has been received and that it is ready to receive new data.

In addition to the above registers, the APB interface also provides read access to a couple of debug registers. These registers provide information from the arbiter and the AXI interface and can be read via APB. These registers are not in use during normal operation, but are useful for debugging purposes. For example, the Mali driver can read from these registers to confirm that the fractal generator has been configured properly and that it has started its calculations.

5.3.2 Discussion

The APB interface is a very light-weight interface; it only uses a small subset of the APB protocol to communicate. Since it so small, does no computation, and no disadvantages have been observed, no alternative APB interfaces have been considered.

5.4 The AXI Interface

The AXI interface module is defined as an AXI slave device; it cannot initiate AXI transfers on the bus[17]. Its main responsibility is to wait for an AXI request from Mali and respond with the z-coordinates that correspond to the AXI address of the request. The coordinates are read from the coordinate cache described in Section 5.6 and transmitted on the bus in two bursts with four coordinates(128 bit) each burst. Note that an AXI request only provides the address of the first coordinate in the transfer; the AXI interface automatically has to read and transmit subsequent coordinates based on the initial AXI address, and the burst and transfer length signals.[7]

5.4.1 Verilog Implementation

The AXI interface is implemented as a finite state machine. When there is an AXI request, the AXI interface checks if the first four coordinates of the transfer are in the coordinate cache. If they are ready the AXI interface checks the next four. On the other hand, if they are not, the AXI interface

[7]The AXI interface only supports transfers with one or two bursts, which equals four or eight coordinates. The Mali vertex loader will never ask for more than two bursts, so no more is needed.

interrupts the arbiter with the address of the first missing coordinate. The arbiter will then prioritize the request from the AXI interface and calculate the missing coordinates. The AXI interface proceeds to wait for the coordinates to be ready. After all eight coordinates are ready the AXI interface examines if Mali is ready to receive the burst. If Mali is ready, the coordinates are sent in two cycles, four coordinates each cycle. After transmission, the AXI interface goes back to the waiting state and waits for a new request.

5.4.2 Discussion

The AXI part of the AXI interface is optimized for communication with the vertex loader in Mali, as the APB interface it uses the bare minimum of signals to communicate. No other implementations have been considered on this part. On the other hand, several implementations have been considered on the reading of coordinates from the coordinate cache and how to check if a coordinate is valid(i.e. calculated by the arbiter and is not belonging to an old frame). Currently, all the coordinates in a burst is read four at the time, and if one of them are not ready, the address of the first one is given to the Arbiter. It would also be possible to check the coordinates one by one, and only give the address of the missing one to the arbiter, but this would require more cycles to check. Also, since the arbiter calculates coordinates sequentially, it is likely that one missing coordinate in a burst means that the other coordinates are also missing. In this case, checking one and one coordinate would waste a lot of time compared to the chosen solution, since the arbiter would have to be interrupted several times in a row. See the discussion in the coordinate cache section for details on the validity check.

5.5 The Arbiter

The primary function of the arbiter [8] is to calculate all the z-coordinate values of a fractal and store them in the coordinate cache. As displayed in Figure 11 the arbiter consists of several sub-components; a feeder, the fractal point generators(FPGs) and a coordinate storer. The number of FPGs is parameterized and determines the number of z-coordinates that can be calculated in parallel. The functionality of each component is explained below.

[8] This is not a descriptive name, the module does more than arbitration, but good names are hard to find.

The job of the feeder is to feed x- and y-coordinates to the FPGs; the FPGs will then calculate the z-coordinate of the given vertex. The Feeder is given the initial coordinates and step size of the fractal landscape from the APB interface; once the values have been received the Feeder starts calculating the xy-coordinates of the vertices. When an xy-coordinate is ready it is fed to the first available FPG.

The feeder assumes that Mali will request the coordinates in incremental order, based on AXI addresses as specified in Section 5.2 , so it calculates coordinates in row major order until the fractal is complete or there is an interrupt from the AXI interface. If there is an interrupt, the AXI interface just tried to read missing coordinates from the coordinate cache. These coordinates must be given priority since Mali is waiting for them. The feeder, after finishing the current coordinate, handles the interrupt by calculating the coordinates of the data burst that belongs to the address from the AXI [9]. Once the interrupt has been handled, the feeder continues to calculate from the position it was interrupted.

The FPG is taken from the work in [4] and is explained in Chapter 3. It uses Equation 1 to calculate the z-coordinate of a vertex, using the x- and y-coordinate of the vertex as input.

The coordinate storer reads finished z-coordinates from the FPGs and stores them in the coordinate cache.

5.5.1 Verilog Implementation

The feeder is implemented as a state machine: At the initial state it checks for an interrupt from the AXI Interface. As explained above, the status of the interrupt decides which pair of xy-coordinates to calculate next. In the case of an interrupt, the feeder keeps track of its current position with an internal address register.

Calculation of coordinates is done according to the steps below; note that the feeder uses two memories to store finished xy-coordinates in, this is done to save calculations as the coordinates are used $NUMPOINTS$ times in the calculations of a frame.

1. The coordinates of the top left vertex(x0,y0) and the step size between

[9] The address given to the arbiter from the AXI interface is in the internal format described in Section 5.2. The arbiter converts this address into the corresponding xy-coordinate.

vertices of the fractal are read from the APB interface.

2. The row and column of the x,y,z vertex is found by translating the vertex address as described in Section 5.2.

3. The feeder checks if the xy-coordinates are in memory. If they are, they have already been calculated and is fed straight to an available FPG, skipping the next steps.

4. If the xy-coordinates are not in memory they are calculated:
   $y = y0 + (stepsize * rownumber)$
   $x = x0 + (stepsize * columnnumber)$
   This step requires one 32bit floating-point adder and one 32bit floating-point multiplier. In addition, it requires a 32bit fixed- to floating-point converter to convert the row and column numbers. All these components are provided by ARM libraries.

5. Finally, the coordinates are fed to an FPG and stored in memory for reuse at a later time.

The FPGs are taken straight from [4] so the implementation details are not presented here. See Chapter 3.

The coordinate storer is implemented as a state machine. It continuously checks if any of the FPGs are finished with a coordinate. When an FPG finishes, its z-coordinate is written to the z-coordinate's address in the coordinate cache.

5.5.2 Discussion

The arbiter has many design options, especially regarding the calculation and storing of xy-coordinates. Currently it calculates the xy-coordinates and then stores them in a memory. Another option is to not have memories; this would save area at the cost of performance. Since performance is the most important criteria for the fractal generator, memories are used.

Currently the xy-coordinates are calculated when they are used for the first time. An option that has been considered is to calculate the xy-coordinates in parallel with the regular operation of the Feeder. This would improve performance since the xy-coordinates would be more likely to be in the memories when needed. However, implementing this would make the design more complex, and it would only have an performance impact once for each vertex(the

first time an x- or y-coordinate is seen). Thus, it has not been chosen for implementation.

5.6 The Coordinate Cache

The coordinate cache stores the computed z-coordinates of the fractal surface. It has two address inputs; one from the arbiter and one from the AXI interface. The address from the arbiter is used to write a single coordinate to memory, and the address from the AXI interface is used to read a burst of four coordinates at once, starting from the coordinate at the AXI interface address. When reading a burst, the coordinate cache indicates if all the coordinates in a burst exist in memory. This information is used by the AXI interface to interrupt the arbiter when coordinates are missing.

Upon reset, all the data in the coordinate cache is cleared to zero by the AXI interface. The data is cleared in bursts of four coordinates each cycle. Since the Mandelbrot equation can never result in zero iterations, a zero in memory indicates that the coordinate at that address is missing. In addition to the global reset, the fractal generator's reset is connected to the reset of the Mali geometry processor(GP). The GP is reset after each completed GP job; since the fractal generator only provides data to the GP, this can be used to clear the coordinate cache as well. In fact, since the clear takes much less time than the PP job following the GP job, this clear is completely without performance loss.

5.6.1 Verilog Implementation

The coordinate cache consists of four separate memory blocks with common input and output signals(Figure 12). The separate memory blocks are written in Verilog to infer as block RAMs when synthesizing on a Xilinx FPGA. Each memory has space for one z-coordinate in width, and $NUMPOINTS * NUMPOINTS/4$ coordinates in depth. Since the coordinate cache is organized in rows of 4 coordinates it needs to do some math to translate between address and z-coordinate position. All RAMs are connected to the same enable and address(depth) bus. When reading and writing from the RAMs the buses are set according to the following equation.

$$\begin{aligned} Write: \quad & enable[ARBITER\_ADDRESS\%4] <= 1; \\ & address <= (ARBITER\_ADDRESS >> 2) \\ Readburst: \quad & enable[3:0] <= 4'hf; \end{aligned}$$

$$address <= (AXI\_ADDRESS >> 2) \tag{3}$$

The $NUMPOINTS$ parameter must be restricted to be a multiple of 4 if the addresses are shifted as above.

5.6.2 Discussion

The main advantage of organizing the coordinate cache with 4 separate RAMS is that both reading a burst of coordinates and checking if the coordinates are valid can be done in two clock cycles. This enables the AXI interface to respond very quickly to Mali, leading to a higher frame rate if the coordinates needed are in the cache. The main disadvantage of this method is the translation needed from address to coordinate when reading and writing. These operations could potentially reduce the maximum frequency of the system.

An easier implementation would be to have one big memory block with a depth of $NUMPOINTS^2$ coordinates. Reading a burst from this RAM would take 5 cycles, and if the last coordinate was missing, the cycles would've been wasted since the arbiter would get interrupted later. Since performance is the main requirement of the fractal generator, the first option was chosen.

Determining if data is valid in the cache can be done in many ways. Initially the cache used a valid bit for each coordinate; when a coordinate was read or written the bit got set to 0 or 1 respectively. As with the chosen solution, this requires an initial reset to get the cache in a known state. Additionally it requires an extra storage bit and a little extra logic in the memory blocks compared to the chosen solution.

Since clearing the whole cache between frames doesn't affect performance, except at the first frame, clearing the cache was chosen. Performance might be lost at the first frame, since Mali might request data while the clear is ongoing. For all other frames the cache is cleared during a PP job. [10]

Cache Size Currently the cache stores the entire $NUMPOINTS^2$ fractal. Since Mali requests vertices in a fairly predictive manner according to the OpenGL ES index array, the memory size can be reduced without much

[10] This consideration did not include power consumption. Since the clearing method requires more switching of bits, it probably uses more power overall.

performance loss. For example, a cache with ten rows and $NUMPOINTS$ columns could first calculate the first 10 rows of the fractal. As Mali began reading from the rows, the data would be continuously replaced with the fractal data from subsequent rows. However, this would require a more advanced addressing scheme. The fractal generator would need to keep track of the rows currently in the cache, and know which external addresses they belong to.

Reducing and optimizing the cache size has been left for future work. Using the recommended parameters, the cache stores $128 \cdot 128$ coordinates which equals 16kB of RAM.

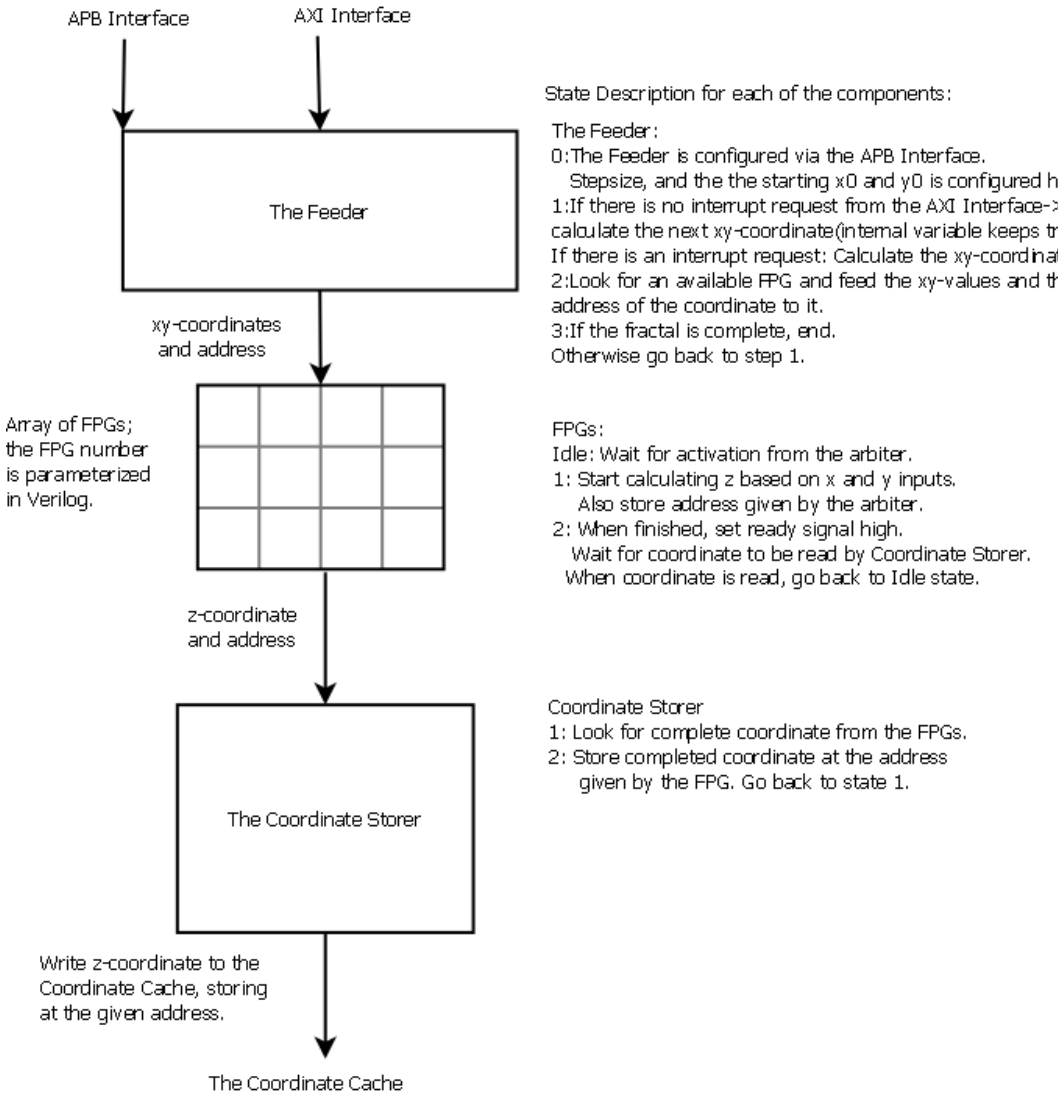

Figure 11: The internal components of the Arbiter and their state descriptions.

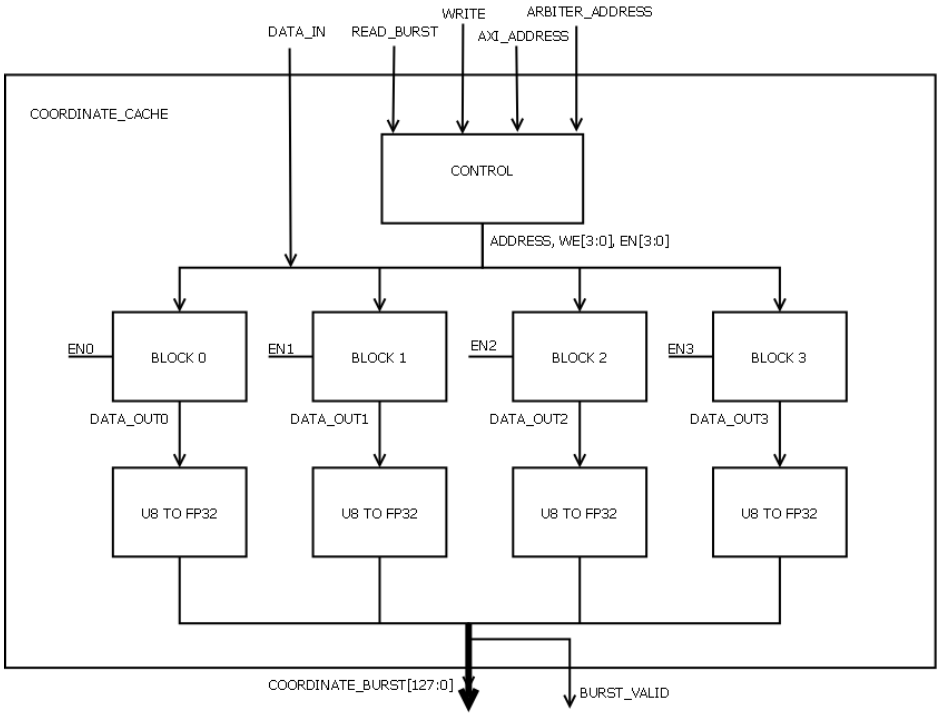

Figure 12: The structure of the Coordinate Cache.

5.7 Fractal Generator Verification

This section will use the Universal Verification Methodology to create a verification environment and perform tests to ensure that the fractal generator is working as intended. This is done to limit the sources of error when integrating the fractal generator with Mali in later chapters.

The Universal Verification Methodology is an emerging standard of verification. Its biggest strength arguably lies in a large amount of code reuse and flexibility. However, setting up the first framework for tests require quite a bit of code, so it is not optimal when doing a single verification like in this thesis. Ultimately, the methodology was chosen mostly for educational purposes.

A single verification environment will be used to verify the fractal generator. I.e., the framework will only connect to the top-level module of the generator. In an industry situation, each subcomponent would have their own tests and framework. The sub-tests would get inherited by the top-level framework, ensuring correctness at all levels.

The framework uses a combination of directed and constrained-random testing.

5.7.1 The Verification Framework

The verification framework is shown in Figure 13. The list below briefly explains the functionality of each of the components. The full source code can be found in the appendix.

APB Agent The APB agent is connected to the APB interface of the fractal generator. It consists of a sequence, sequencer, driver and monitor. The sequencer provides APB items from the sequence to the driver. The driver simulates the APB protocol and stimulates the fractal generator with the contents in the given APB items. The monitor logs the APB traffic and transmits the traffic information to the scoreboard.

An APB item contains the address,data and direction of an APB transfer. The sequencer sends three items to configure the fractal generator with x0,y0 and the step size.

AXI Agent The AXI agent has the same subcomponents as the APB agent. It does the same as the APB agent except it is connected to the AXI

interface of the fractal generator and uses the AXI protocol to read data from the fractal generator.

Virtual Sequencer The virtual sequencer controls the sequences in the agents and thus controls the timing of the entire system. The sequencer makes sure that the fractal generator is configured through the APB interface before the AXI agent starts reading vertices.

Scoreboard The scoreboard is responsible for checking if the response from the fractal generator is correct, it uses the data from both agents monitors to do its own calculation of the fractal coordinates. The calculation is done with the following steps.

1. The scoreboard registers when an APB transfer configures the fractal generator and stores the configuration values.
2. When an AXI read is performed by the virtual sequencer the scoreboard checks the address of the read request. This address, and the configuration values from step 1, is used to calculate the coordinate in the Mandelbrot set that maps to this address(5.2).
3. The coordinate from step 2 is used to calculate the rest of the burst, the coordinates are input into a function calculate_iterations that runs the Mandelbrot set equation on the coordinates. Calculate_iterations is an external c function that has been imported into System Verilog, it uses the same algorithm as the demo to calculate the iterations for each coordinate.
4. The iteration values from the scoreboard calculations are compared with the actual data read from the fractal generator.

5.7.2 The Verification Plan

The following functional coverpoints have been chosen to verify the fractal generator.

- Read the version register from the APB interface and confirm the value. This confirms that the APB interface is properly connected.
- Read the debug register and check that the fractal generator has started. This confirms proper connection between the APB interface and the other components. It also confirms proper reset behavior.

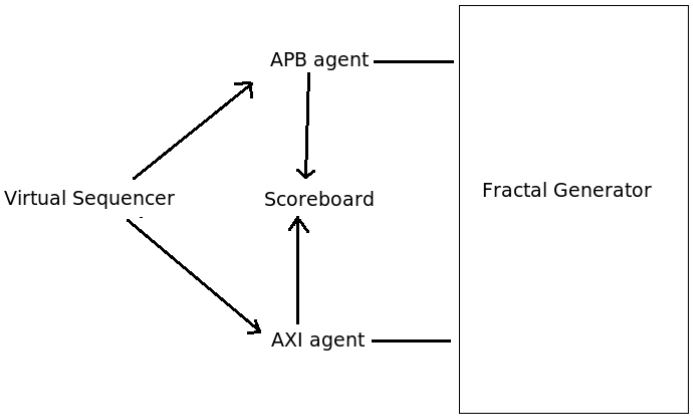

Figure 13: The verification framework.

- Read one address from the top, middle and bottom row of a given fractal configuration. This will confirms the general functionality of the fractal generator. It also checks all math and address translation between the components

If the system were to be completely verified one should perform these checks for a large range of random fractal configurations, i.e. many areas of the Mandelbrot set. However, this verification will only use one area of the Mandelbrot set(x0=-1.5,y0=0.8, step size=0.01) with $NUMPOINTS = 128$

The functional coverpoints will be verified with assert-statements in the scoreboard, manually changing the addresses of the AXI requests between runs. In a more thorough test case one would use UVM to generate these addresses randomly and test all the possible addresses of a fractal.

In addition to functional coverage, line coverage, state coverage and conditional coverage will be measured automatically by the Synopsys VCS simulator.

5.7.3 Results and Discussion

All the coverpoints in the verification plan were successfully tested, the functional coverage is 100%. The coverage measurements done by VCS are shown in Figure 14.

The functional verification showed that the fractal generator is working as

Total Module Definition Coverage Summary

SCORE	LINE	COND	FSM
72.55	93.48	64.61	59.57

Total modules in report: 17

SCORE	LINE	COND	FSM	NAME
56.53	89.19	33.33	47.06	axi_interface
67.50	94.37	40.62		vithar_lib_f32_addsub
75.81	100.00	51.61		vithar_lib_f32_mul
78.14	88.00	75.00	71.43	fpg
80.04	97.60	84.62	57.89	arbiter
88.24	76.47		100.00	apb_interface
95.00	100.00	90.00		vithar_lib_fp32_adder_main
96.15	100.00	92.31		vithar_lib_fp32_partial_cancel_add
100.00	100.00	100.00		coordinate_cache
100.00	100.00			vithar_lib_clz32
100.00		100.00		vithar_lib_u32_to_f32
100.00	100.00			fractal_tb_top
100.00	100.00	100.00		block_ram
100.00	100.00			vithar_lib_uu_int_mul
100.00	100.00			vithar_lib_fp_clz32
				fractal_generator_main
				vithar_lib_fp32_ctz26

Figure 14: Coverage metrics logged by VCS while running the verification framework.

intended, the calculation of coordinates and address translation is correct. The values in Figure 14 are a measure of how thoroughly the test bench simulates the fractal generator. Some of the numbers seem too low, but when looking deeper they are all acceptable.

For example, the conditional coverage of the AXI interface is only 33%, but there are only three conditional blocks and two of them contain debug checks for signal values that should never happen. Thus, the conditions will never be fulfilled unless there is an error.

Two other examples are the state coverages of the AXI interface and the arbiter. Both modules contain a state for flushing of memories. Almost all the missing state transitions involve moving to the flushing state from the normal operation states, this should never happen, so the state coverage will be low.

In conclusion, the verification was successful and the test confirms that the

general functionality of the fractal generator is implemented correctly. If more time was to be put in verification, the test should be run with more APB configurations to examine the fractal generator in detail. These tests should also use constrained random testing and measure the functional coverage automatically using the UVM capabilities.

6 Integration of the Fractal Generator

The two previous chapters presented the chosen design for the fractal generator and verified its functionality. This chapter will describe and verify the integration of the fractal generator with Mali. In addition to this, the chapter will connect the fractal generator and Mali to an FPGA framework and synthesize the complete system onto an FPGA.

Ultimately, the fractal generator is to be controlled by the software driver. Implementing and testing the driver without testing the communication between Mali and the fractal generator could lead to a lot of painful debugging; a problem could be located in either hardware, software or both. It was therefore decided early in the design process that the hardware should be tested independently. The following steps were performed to confirm working hardware on FPGA.

1. Design and verify the fractal generator(done in previous chapters).
2. Connect, as described in Chapter 4, the fractal generator with Mali.
3. Test the connection between Mali and the fractal generator.
4. Integrate Mali and the fractal generator with the FPGA framework.
5. Test the complete FPGA framework.
6. Synthesize the framework onto an FPGA and test it again.
7. Start implementing the driver.

ARM has a test system(Figure 16) that allows configuration of Mali by parsing commands from text files. The files are parsed and simulated using the Synopsys VCS simulator. The commands can perform writes using the APB protocol and change memory contents. Each test in the list above uses the same(almost, see sections below) set of text files to run specific GP and PP(see Section 2.3) jobs. The test system also provides access to the framebuffer output from the jobs. The framebuffer data can be displayed as an image of the frame drawn by the jobs. This will be used to compare the output from the tests with the original input.

For a test to examine the integration of Mali and the fractal generator, the input text files needs to actually use the fractal generator for something. This means that the test cannot draw a random image, it has to be made

specifically for the fractal generator. Making these text files manually is not feasible, so a different method was required.

The Mali drivers can be compiled and configured to activate dumping of frames. When dumping a frame, you can choose to also dump all the information about the memory, and the jobs, used to draw the frame. The dump information is stored in hexadecimal format precisely like the input text files. Thus, the following steps were taken to obtain the proper test stimuli:

1. Write the OpenGL ES demo(Section 7.1) that draws an image from the Mandelbrot set using the fractal generator, but use a software implementation of the fractal generator to draw it.

2. Run the demo on a platform containing the Mali 400-GPU, and dump a single frame of the demo. This dump can now, with some slight changes, be used as input to all the tests listed above [11]. The platform used for this was the Odroid-A tablet and the output image can be seen in Figure 15.

Figure 15: Left:Fractal landscape using a blue to white gradient. Right: The Odroid-A tablet.

The test files obtained from the Odroid were used as inputs to all the tests listed above. The below sections describe how the hardware was connected in the different scenarios, and the results of running the test.

[11] Changes needed: Change the address of the fractal generator vertex stream ,add configuration of the fractal generator and delete all the input in the PP job that are supposed to be from the GP.

6.1 Connecting Mali and the fractal generator

A Mali-400 test bench has been provided by ARM; the fractal generator is to be connected with Mali using this test bench. The text files obtained above can be used to configure Mali and the fractal generator through this test bench. As shown in Figure 16, the ARM test bench usually connects Mali directly to an external memory(the RAM) via the AXI bus.

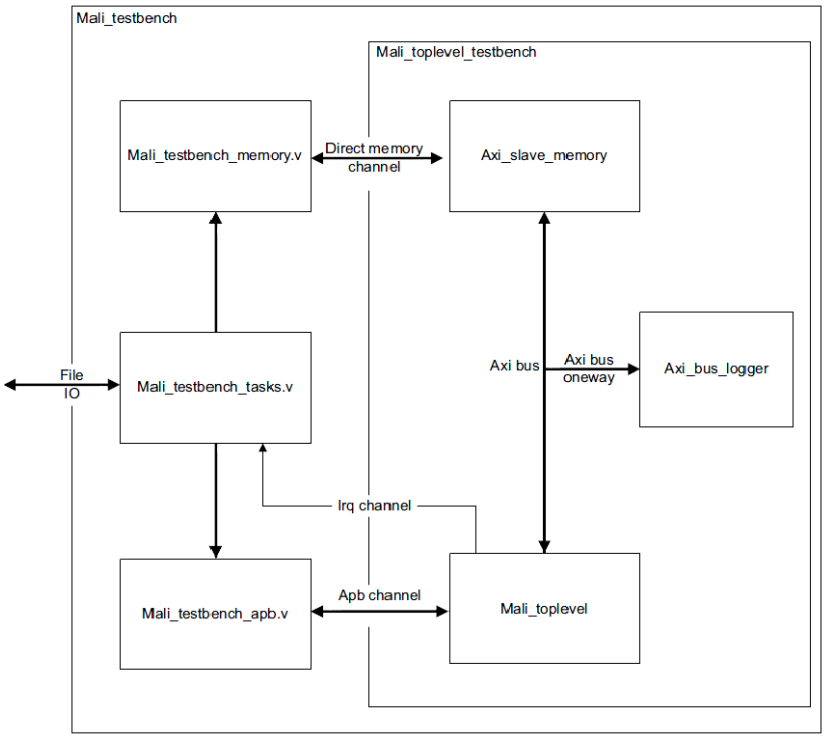

Figure 16: ARM Test Bench for Mali [15].

To integrate the fractal generator with this test bench, it needs to be connected to the AXI bus. This is done by inserting the PL301 component, as described in Chapter 2, between Mali and the slave memory in Figure 16. The memory and the fractal generator is then both connected to the PL301 as AXI slaves, and the PL301 is connected to Mali. The PL301 will now arbitrate the AXI communication from Mali to the correct AXI slave. In this case the PL301 is configured such that AXI address 0x1000_0000 to 0x1fff_ffff selects the fractal generator, while all other addresses enables the memory. In addition to this, the original test setup has a APB bus connected directly to Mali. In the final test bench the APB bus is reconfigured by adding a mux so that the fractal generator has its own APB address range. The new

top level module consisting of Mali, the fractal generator and PL301 can be seen in Figure 17. It replaces the *Mali_ toplevel* module in Figure 16.

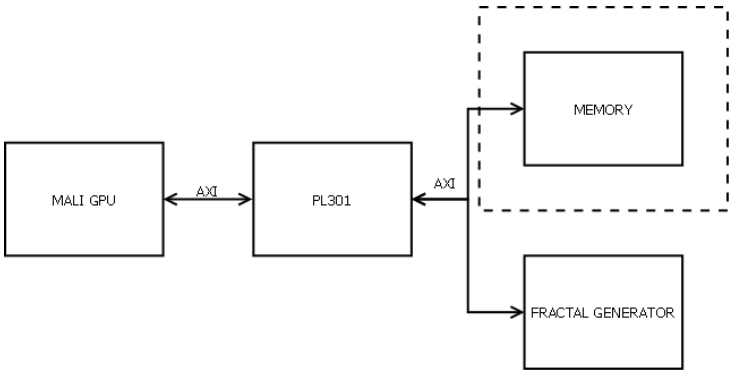

Figure 17: Connecting Mali and the fractal generator.

6.1.1 Test Results

All the data written to memory by Mali during the test is dumped to a hex-file. The frame buffer, and thus also the frame produced by the test can be extracted from this hex-file using ARM software. The image produced was identical to the image produced by the Odroid in Figure 15.

6.1.2 Integration Discussion and Conclusion

The integration test was successful and shows that the communication between Mali and the fractal generator works as intended. It also shows that the Open GL ES demo and vertex shader are configured correctly, only the vertex stream of z-coordinates are retrieved from the fractal generator; the other streams are read from memory. This test says nothing about performance or achieved frame rate though, it only confirms that the communication between Mali and the fractal generator works.

The image retrieved from the frame buffer after running the test is identical to the image produced by the demo without the fractal generator. The fractal generator is producing the correct z-values for this frame and the test was successful.

6.2 FPGA Integration and Test

The previous section confirmed that Mali and the fractal generator(the system) are communicating properly. ARM has provided a platform and FPGA framework for the system. This section will describe the framework and how the system is connected with the framework. In addition to this, the section will test the complete framework, synthesize it and load it onto an FPGA. When the system is on the FPGA it will be tested once more, all the tests in this section are as described in Section 6.

6.2.1 Platform and Framework Description

The platform used in this thesis is the Versatile Express Motherboard Express μATX with the daugtherboard Coretile Express A9 and two Virtex 6 FPGAs(Figure 18. A complete verilog framework for use with Mali-450 was given by the ARM FPGA group. To integrate the system used in Chapter 6.1 with this framework, the Mali-450 was removed from the framework and replaced with the new top-level module described in Figure 17. In addition to this, the features unique to Mali-450 were removed to accomodate the Mali-400, which is used with the fractal generator. Makefiles to compile, build, and synthesize the framework were also reconfigured to include the fractal generator and to use Mali-400 instead of Mali-450. The framework was compiled and simulated in VCS and synthesized using Synopsys Certify, Certify also partioned the framework across the two FPGAs.

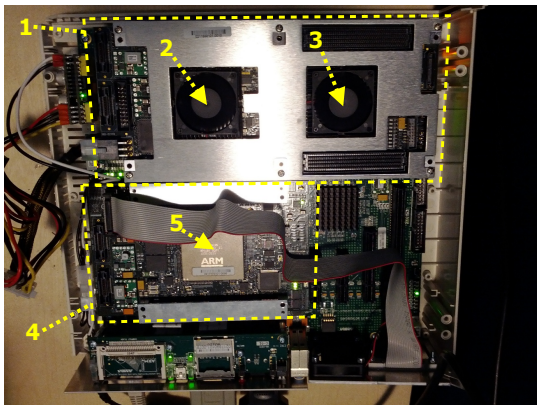

Figure 18: 1. Motherboard Express μATX. 2-3. Virtex6 FPGAs. 4.Coretile-Express. 5.The A9 CPU

6.2.2 Testing the FPGA Framwork

Included with the framework was a test bench similar to the one in Chapter 6.1. This allows the simulation of the test system directly on the framework with VCS. The original plan was to use the Odroid dumps directly on this test bench, just as in Section 6.1,to confirm that the new top level module was connected correctly in the framework.

However, since the memory mapping was different from the Odroid to the FPGA, the addresses in the dump had to be remapped [12]. To do this without corrupting the information contained in the dumps, specifically the connection between GP and PP jobs, the memory dumps(hex files containing the data structures of the jobs) had to be left alone. Instead the memory management units(MMUs) inside Mali were used to remap the data. A MMU page table was constructed to remap the addresses from Odroid mapping to FPGA mapping, and then manually edited to map the fractal generator address as well. After constructing the page table, the dumps were edited slightly to enable the MMUs and read the height coordinates from the fractal generator. The MMU mapping was successful, and the dumps could now be used to test single frames on both the FPGA test bench and on the FPGA itself.

6.2.3 Test Results

After remapping the dumps the framework produced the same image as the software fractal generator(Figure 15).

6.2.4 Synthesis Results and Test on FPGA

After synthesizing the framework onto the FPGA the framework was again tested. Since the dumps were already modified to fit the FPGA address mapping, the same dumps could be used again. Simulation on the FPGA could not be done using VCS, since the FPGA is actual hardware and not a simulation. Instead, an ARM program that parses the same text files to configure the hardware on the platform was used.

The image produced by the FPGA was identical to the image produced by the software fractal generator(Figure 15).

[12] This was not the case with the Mali integration.

Module	Max clock frequency(Mhz)	Number of LUTS
Fractal Generator(8 FPGs)	48.1	15826
Arbiter(8 FPGs)	48.6	15398
One FPG	91.3	1750
AXI interface	446	211
APB interface	555	13
Coordinate Cache	204	501

Table 1: Synthesis results for the fractal generator with 8 FPGs on a Xilinx Virtex-6x FPGA.

6.2.5 FPGA Test Discussion and Conclusion

The tests were successful; the proper image was produced both in simulation of the framework and when testing it on the FPGA. Mali and the fractal generator is properly connected with the FPGA framework.

The address map collision between Odroid and the FPGA cost a lot of time to debug and should have been avoided. The Odroid-A address map can be adjusted in the driver, and ideally this mapping should have been set equal to the FPGAs address range before starting the tests.

Ultimately, the series of tests performed in this chapter revealed a lot of bugs that would have been harder to solve with the driver in place. The test methodology worked as intended, and the system is now ready to be controlled by the driver.

7 The OpenGL ES 2.0 Demo and Driver

The problem description states that an OpenGL ES demo(the demo) must be written to showcase the functionality of the fractal generator. In order to run a demo with the fractal generator, changes to the Mali software driver are needed to set up the fractal generator prior to each frame. This chapter starts with a brief overview of the written demo and the driver, how they communicate and what they use the fractal generator for. Furthermore, the chapter presents and discusses the details of the chosen implementations for both the demo and driver.

The demo is an application that draws 3D graphics using the OpenGL ES library function calls. The task of the driver is to configure the Mali GPU and the fractal generator to perform these function calls in hardware, improving their performance when compared to a pure software implementation.

The demo constructs a 3D landscape by combining two separate vertex streams. The first stream provides the x- and y-coordinates of a static and flat landscape. This landscape can be viewed as a frame to put height coordinates on; its xy-coordinates does not change throughout the demo[13]. The second stream is the important one in this thesis. This stream is generated by the fractal generator based on the Mandelbrot set and provides the z-coordinates of the landscape. Combining the heights from the fractal generator with the static landscape makes a three-dimensional landscape. Changing the stream from the fractal generator, by changing the chosen area of the Mandelbrot set, thus changes the landscape.

For the fractal generator to generate a stream it needs to be configured. The configuration must be done via the APB interface and it must set the initial xy-coordinate and step size to calculate the wanted area of the Mandelbrot set. To configure the fractal generator, the demo has to let the driver know when a function call uses the generator, and pass along the settings required to configured it. The driver reads the settings, configures the fractal generator, and then configures Mali to perform the function call. The function call tells Mali to read data from the streams and combine them into a landscape. The combined landscape vertices then proceeds through the rest of the OpenGL ES pipeline.

The above explains the general relationship between the driver, demo and fractal generator. The following sections present the implementations in more detail.

[13]This stream has nothing to do with the Mandelbrot set.

7.1 The Demo

The demo is divided into two major phases, one initialization phase and one rendering phase, described below. The demo also implements a software version of the fractal generator. The software implementation was used to obtain test stimuli in Chapter 6, and will be used as a reference point when testing the hardware performance in Chapter 8.

This section will only present the OpenGL ES information relevant to the fractal generator; there is much mandatory OpenGL ES coding that has been left out. The demo source code can be found in the appendix. The section will also present the shaders written for the demo. Discussions around the implementations in this chapter will be done in Section 7.3 since the choices made in the demo are strongly connected to the driver implementations.

7.1.1 The Initialization Phase

The initialization phase performs platform specific EGL and OpenGL ES initialization and loads the shaders. Afterwards it constructs the flat landscape mentioned in the above section.

The 2D landscape is drawn using the GL_TRIANGLE_STRIP mode. The vertex data and index array(Section 2.2) of the landscape is stored as one VBO each and bound to a variable in the vertex shader. This enables the vertex shader to read data from the VBO as a stream of vertices.

In addition to this, the phase creates and binds a third VBO that will contain the height vertices of the 2D landscape. An initial point and step size in the Mandelbrot set is used to calculate the height values. This VBO uses the same index array as the 2D landscape but is bound to a different variable in the vertex shader, this makes the z-coordinates a separate stream input to the shader.

If the hardware fractal generator is used the third VBO will be left empty in the demo; the vertices will be given from the fractal generator. Otherwise, the software implementation of the fractal generator is used to calculate and store all the height vertices of the landscape in this VBO.

7.1.2 The Rendering Phase

The rendering phase is written as a looping state machine, where each loop focuses on animations around an interesting point in the Mandelbrot set. The list below explains what is done in each state.

State 1: The demo contains a list of interesting points in the Mandelbrot set. Here, interesting means that zooming in on the point reveals some good looking geometrical structures. The first state in the demo positions the camera into a starting position and randomly selects a point from the list.

State 2: In the second state the camera is moved towards the current interesting points, positioning the camera to start zooming down into the fractal.

State 3: In the third state the camera zooms in on the interesting point by keeping the point in the center while lowering the step size between points in the Mandelbrot set.

Each of the above states can make several calls to glDrawElements(i.e., a state can last for several frames) and has to either calculate the height values with the software fractal generator or configure the hardware one for each call. The rendering phase does not do anything with the 2D-landscape created by the initialization phase, it only moves the camera and changes the fractal generator configuration.

7.1.3 Configuring the Fractal Generator

The demo uses an array of three floats called *fractal_ configuration* for each of the interesting points. The array contains the three APB settings(x0,y0 and step size)that selects an area of the Mandelbrot set. To configure the fractal generator this array has to be passed to the driver.

When a draw call is made in OpenGL ES the driver copies the OpenGL ES state and data structures into a big C struct. Thus, inside the driver, one has access to all the relevant OpenGL ES information for the draw call.

glVertexAttribPointer is a function that describes the data inside a VBO. When the VBO is drawn by a draw call, the information stored by this function is available in the driver. The function can set a parameter *pointer* which is an offset into the VBO where the data described by glVertexAttribPointer is stored. This can be used to store different sections of data in the same VBO. However, this functionality is not crucial, since one can always make another VBO without any offset instead. Thus, the *pointer* field is used to

pass the fractal configuration to the driver. Instead of using the field as an offset, the pointer points to the *fractal_ configuration* array for the current Mandelbrot set area to be drawn. If the data in a VBO is not provided by the fractal generator, the *pointer* field must be set to the null pointer. See Listing 2.

```
glVertexAttribPointer(afheightLoc, 1, GL_FLOAT, GL_FALSE, 0,
                      landscapes[i]->fractal_configuration)
                      ;
```

Listing 2: Calling glVertexAttribPointer with a fractal configuration.

7.1.4 The Vertex Shader

As explained in the above sections, two vertex streams are input to the vertex shader. One stream is the coordinates in the 2D-landscape, and one is the height vertices. The vertex shader combines the two streams into a single 3D-landscape and then perform matrix operations on it(moving the camera). The merging is done by simply setting the y-coordinate of the 2D-landscape to the current value in the height stream [14]. Since both streams contain equal amount of points and uses the same index array this makes the height coordinates appear at the correct point in the array, drawing the Mandelbrot set. The vertex shader also divides the height values by 80, this is done to normalize the height values, since the standard OpenGL ES range is between -1 and 1.

7.1.5 The Fragment Shader

The fragment shader reads the height value for each coordinate from the vertex shader. It examines the value of the height coordinate and adds color to it. The color of each point is decided using the built in function mix and a gradient, but if the point has the maximum iteration value the point gets a color of red. The maximum iteration points are represented by a separate color to provide a clear visual boundary, and to easily see differences with varying numbers of the MAX_ITERATIONS parameter.

[14] This is a bit confusing since it is the z-coordinate,not y, that has been referred to as the height throughout the thesis. However, in the shader the height is put in the y-coordinate position.

7.1.6 Software Fractal Generator

The demo is used in Chapter 6 to obtain test stimuli for the fractal generator. In order to do this, the demo must simulate the hardware fractal generator with a software implementation. The software fractal generator is presented below.

The software fractal generator uses the same three values(x0,y0 and step size) for configuration as the hardware version. These parameters and a pointer to an array is input to a function which calculates the z-coordinates of the Mandelbrot set and stores them in the given array. The main function uses an internal function, getFractPoint, made by Per Christian Corneliussen in [4] to calculate the iterations for each point. See Listing 3.

```
void getFractalHeights(float x, float y, float step size, int
    numpoints, GLfloat *iterations)
// Calculates the height/iterations of all the vertices in the
    fractal specified
// by x,y,step size and numpoints. The fractal is assumed to be
    square = numpoints^2.
// Assumes an array with space for numpoints^2 integers.
{
  int i,j;
  for(i=0; i<numpoints;i++) //y-axis = imaginary
  {
    for(j=0;j<numpoints;j++) //x-axis
    {
      iterations[j+(i*numpoints)] = getFractPoint(x+(j*step size)
          ,y-(i*step size));
    }
  }
}
```

Listing 3: The software fractal generator.

7.2 The OpenGL ES 2.0 Driver

This section explains the Mali driver architecture and the changes made to the driver to properly configure the fractal generator for each call to glDrawElements made by the demo.

7.2.1 The Mali GPU Open GL ES Driver Architecture

The main task of the Mali driver is to configure, and provide input to, the Mali GPU based on OpenGL ES calls from a graphics application. The driver uses the Mali GPU to accelerate the OpenGL ES operations, which improves performance and power consumption compared to a software-only solution [19].

The Mali driver is modular and divided into separate layers(Figure 19). This provides several levels of abstraction and enables different drivers to share common software components. Below is a brief presentation of each layer that is relevant to the fractal generator.

Graphics application The graphics application that are making the OpenGL ES calls. In this thesis the application is the OpenGL ES demo.

OpenGL ES driver The OpenGL ES driver translates the current OpenGL ES state and draw operations into Mali GPU data structures and jobs. The data structures enable hardware graphics features based on the current state. This driver also performs set-up and initialization of the different hardware jobs.

Base driver The base driver provides the OpenGL ES driver with an abstraction from the operating system and hardware platform. It is responsible for memory management, job handling and interfacing with the OS. The base driver provides communication between user-mode and kernel-mode OS operation, and hides the low-level details from the OpenGL ES API.

Mali device driver The Mali device driver runs in the OS kernel and is the interface between the base driver and the hardware. It reads and writes hardware control registers, dispatches jobs to the Mali hardware, performs low level memory management and handles multiple concurrent users.

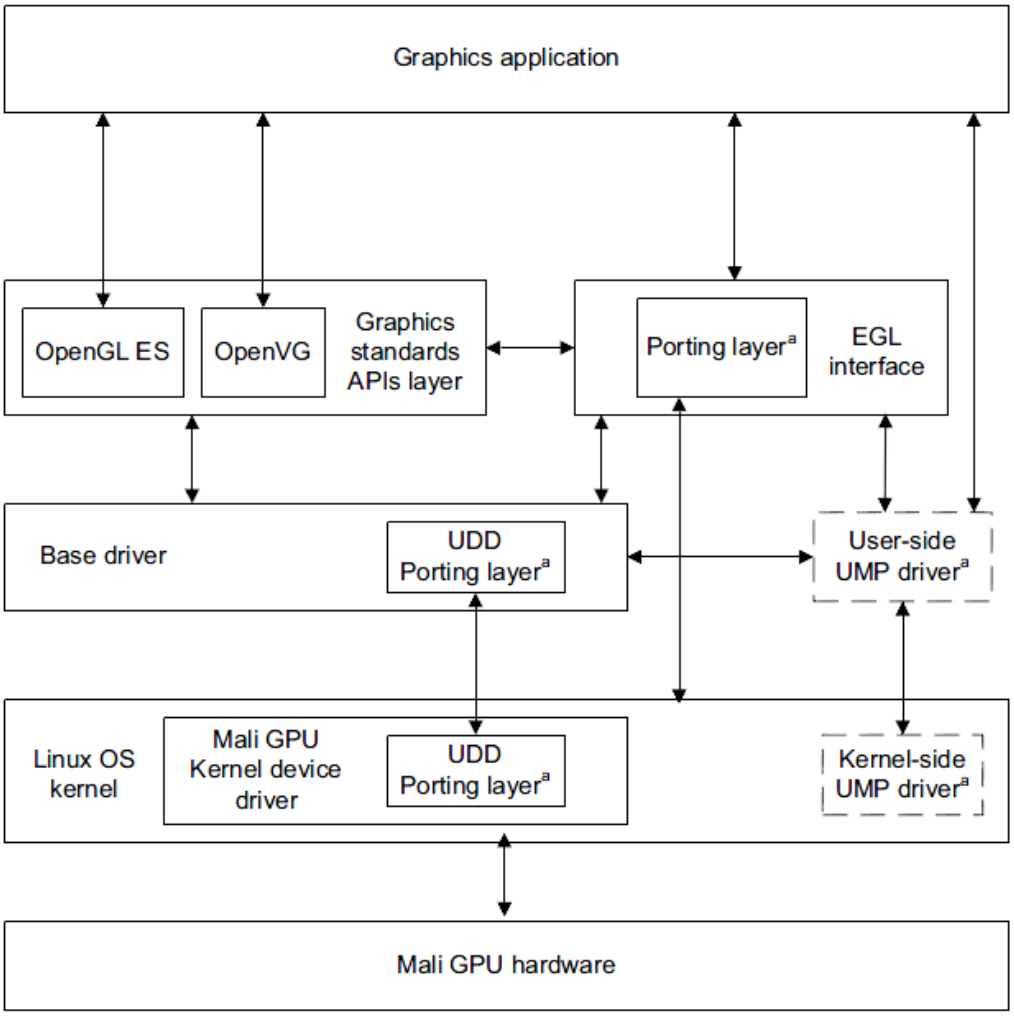

Figure 19: Hardware and software components of the Mali graphics system architecture(Linux). [18, p. 1.4]

7.2.2 Controlling the fractal generator using the Mali driver

For the Mali GPU to read any vertices provided by the demo from the fractal generator, the fractal generator needs to be configured(and thus started) via the APB interface. The driver has to perform these APB writes at the correct moment, based on the information provided by the OpenGL ES demo as explained in Section 7.1. Specifically, the driver has to monitor the draw calls made by the demo, determine if the draw call contains a fractal steam(a stream of vertices that are to be provided by the fractal generator), determine the fractal configuration of the stream and transmit this information to the fractal generator through its APB interface. In addition to this, the fractal generator must be configured at the right time, to avoid reconfiguration in the middle of the previous draw call.

The Mali GPU also has to be informed when a stream of vertices is to be read from the fractal generator instead of the main memory, the driver has to do this by setting the stream address to the physical address of the fractal generator, which is hard-coded in the RTL to access the fractal generator's AXI interface [15].

The following changes have been made to the driver code to enable the above features.

Changes to the base driver The base driver uses a struct to describe a Mali GP job, the struct is visible to the OpenGL ES driver and contains an array of 6 address registers that points to important data structures for the current job. This address array is, before a job is started, copied to the Mali device driver. Thus, this array can be used to provide the Mali device driver with the fractal configuration described in 7.1.3. The array is therefore extended with room for the three *fractal_ configuration* floats.

Changes to the OpenGL ES driver When a call is made to glDrawElements in the demo, the OpenGL ES context is stored in a data structure in the driver, and a whole series of function calls are made to configure and initialize the hardware. One of these calls are called _gles_gb_setup_input_streams and is responsible for setting up the vertex stream information that is provided to the vertex shader(and a given GP job) in the Mali GPU. One of the streams to the vertex shader are to be provided by the fractal generator, and thus changes has to be made to this function. The function uses the OpenGL

[15] This is the address 0x1000_0000, which has been mentioned earlier in the thesis.

ES context to examine the type of each of the incoming streams, e.g. checks if they are vertex buffer objects or not, and then configures, among other things, the memory address of the stream such that the Mali GPU knows where to find the vertices to work with. A stream from the fractal generator needs to have a specific physical address. Since the current function is responsible for setting addresses, this can be implemented in this function. The implementation is done in the following manner:

When the function checks if a stream is a vertex buffer object, a second check is made to see if a pointer was passed with the VBO. This pointer is usually used as an offset into the VBO, but in the OpenGL ES demo, this pointer points to the fractal configuration array. Only the VBO to be provided by the fractal generator will ever have a non null pointer here, and this check will thus determine if a stream is to come from the fractal generator.(assuming all fractal streams are VBOs in the demo).

If a stream is a fractal stream, the Mali memory address to find the stream is set to 0x1000_0000, and the fractal configuration array is copied into the registers that have been added to the Mali GP job structure described above.

```
if(attrib_array->pointer != 0)
        {
          fractal_configuration = attrib_array->pointer;
          unsigned int * lol = (unsigned int *)
              fractal_configuration;
          gp_job->registers[6] = *lol;
          gp_job->registers[7] = *(lol+1);
          gp_job->registers[8] = *(lol+2);
          mem->mali_memory->mali_address=0x10000000;
          mem_addr=0x10000000;
          printf("Driver: Mali address of block= %x\n", mem->
              mali_memory->mali_address);
          printf("Driver: Cpu-seen address of block=%x\n",mem->
              mali_memory->cpu_address);
          attrib_array->pointer = 0;
          printf("Driver: mem_addr2 = %x\n", mem_addr);
        }
```

Listing 4: Then OpenGL ES driver finding a fractal stream.

Changes to the Mali device driver

Allocating memory to the fractal generator The Mali GP and PP have their own MMUs, this means that when the GP tries to read vertices

from address 0x1000_0000, as set by the OpenGL ES driver, this address has to be in the MMU page table and it has to be mapped to a valid allocated address. Since the fractal generator is at physical address 0x1000_0000, the virtual address should be mapped to the same physical address. Since the Mali device driver is responsible for low level memory management on the Mali GPU, this mapping has to be done here.

The page table has to be mapped and the memory allocated, fortunately the driver has built in functions that do this. Thus, in the function that(among other things) allocates the standard page table, a second page table is added that maps virtual address 0x1000_0000 to physical address 0x1000_0000, and allocates enough space for a $NUMPOINTS * NUMPOINTS$ array of height coordinates.

Adding the fractal generator to the APB memory map The fractal generator must be configured through its APB interface, for this to be possible, the fractal generator's APB addresses must be allocated in the device driver.

The Mali GP is represented in the device driver by a struct called mali_gp_core. The struct contains a hardware core(this is the GP in hardware) that also have to get allocated memory. Functions that create and map the hardware core already exist, so to add the fractal generator a new hardware core called "fractal core" has been added to the GP struct. With some small changes, the fractal core can now be created and mapped in the same function as the GP core.

Configuring and starting the fractal generator After allocating APB and memory the fractal generator is ready to be configured via APB. The device driver provides the functions needed to read and write from hardware registers(APB) so the question is where and when to configure. Since the fractal core was added to the GP as explained in the above section, the register functions can be used on the fractal generator in any function that has access to the GP core.

The previous sections state that the fractal generator configuration is contained within every GP job struct, thus the fractal generator should be configured ahead of every GP job. The device driver is responsible for configuring the hardware and initialize jobs, and it uses a function called mali_gp_job_start to start each job. The function is called with the struct of the job, and the struct of the GP core, as arguments. These structs provide

the fractal configuration and access to the fractal generators APB interface. This is everything needed to configure the fractal generator and this function is a chosen as the location to start the fractal generator in.

At every call to this function, a check examines the contents of the array of address registers mentioned in section7.2.2. This check examines if the starting job requires the fractal generator. If the fractal generator is needed, the fractal configuration is read from the address registers and written to the fractal generator using the provided device driver functions. Mali_gp_job_start then proceeds to start the job as normal.

After starting a GP job, the device driver is continually monitoring for interrupts. One of the possible interrupts happen when the GP requires more heap memory to start. When this happens, the job is stopped and then restarted using a different function(mali_gp_resume_with_new_heap). The interrupt causes the fractal generator to reset, and it must therefore be reconfigured as above in this function as well.

```
/* starting fractal generator */
  /* reg6 = x, reg7=y, reg8=step size*/
  if(job->frame_registers[8] != 0)
  {
    MALI_DEBUG_PRINT(1,("Devicedrv : This is a fractal job.
        Configuring registers\n"));
    mali_hw_core_register_write(&core->fractal_core, 0x0000,job->
        frame_registers[6]);
    mali_hw_core_register_write(&core->fractal_core, 0x0004,job->
        frame_registers[7]);
    mali_hw_core_register_write(&core->fractal_core, 0x0008,job->
        frame_registers[8]);
    MALI_DEBUG_PRINT(1,("Devicedrv : Configured registers\n"));
    MALI_DEBUG_PRINT(1,("Devicedrv : X = 0x%08X\n",
        mali_hw_core_register_read(&core->fractal_core, 0x0000)))
        ;
    MALI_DEBUG_PRINT(1,("Devicedrv : Y = 0x%08x\n",
        mali_hw_core_register_read(&core->fractal_core, 0x0004)))
        ;
    MALI_DEBUG_PRINT(1,("Devicedrv : STEPSIZE = 0x%x\n",
        mali_hw_core_register_read(&core->fractal_core, 0x0008)))
        ;
    MALI_DEBUG_PRINT(1,("Devicedrv : Fractal registernr = 0x%08X\
        n", mali_hw_core_register_read(&core->fractal_core, 0
        x0014)));
  }
```

Listing 5: Finding a fractal job and configuring the fractal generator in the device driver.

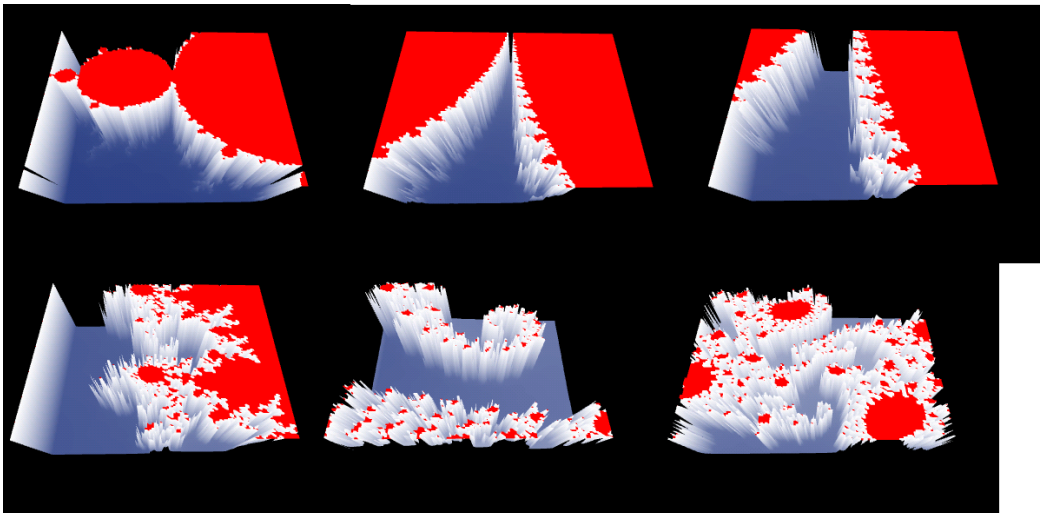

Figure 20: The fractal demo zooming in on a point.

7.3 Results and Discussion

The driver and demo implementations were successful. Figure 20 shows a photo of the demo running with the hardware fractal generator. The performance of the demo is examined in Chapter 8.

Demo Discussion An important aspect of the implementation was the communication between the demo and driver. The fractal generator configuration has to be obtainable from the OpenGL ES state for the driver to access it. In addition to this, the configuration has to be connected to a VBO, since a single draw call can draw several VBOs and not all VBO data will be provided by the fractal generator.

glVertexAttribPointer satisfies all the requirements, it is called to configure each VBO and the parameters set by the function are accessible in the driver. In addition to this, the function is called regardless of the fractal generator, so there is no performance loss from using it. The only cost from using it are the three floats that configure the fractal generator. A disadvantage from using the *pointer* field in the function is that the field can not be used for its normal purpose. However, this is a very minor inconvenience. Thus, the glVertexAttribPointer solution was chosen. Since the chosen solution has no practical cost, no other solutions have been explored. In theory, any function that configures a VBO could be used. One could also modify existing functions or add extra functions to avoid drawbacks like losing the

offset pointer. However, this would be much work for minimal, if any, gain.

Driver Discussion Checking if a stream is a fractal stream is done in the driver by checking the *pointer* field of all VBOs. The driver is already checking if input streams are VBOs or not, so the only needed change is an extra check of the pointer field. When a fractal stream is discovered, the *fractal_ configuration* array is stored in the base driver and the address is set to the fractal generator. Thus, the cost of the chosen solution is one if statement and three assignments for each fractal stream. This is a very efficient solution, so no other solutions have been evaluated. In fact, no other solutions have been identified; the function(_gles_gb_setup_input_streams) that has been modified is the only place were one can easily change the address of input streams.

Allocating memory and adding the fractal generator to the memory map has to be done. The chosen implementation mirrors the memory management of existing components(like the geometry processor) in the driver. It uses optimized built in functions for all memory management and the functions are only called once during the lifetime of the demo. There is little to no room for improvement here so no other solutions have been explored.

The actual configuration of the fractal generator is done by the device driver. The current solution mirrors the initialization and startup of the geometry processor to start the fractal generator. Since the fractal generator provides data to the GP this ensures that they stay in sync. The implementation consists of a single check of the *fractal_ configuration* registers and a couple of register writes to the fractal generator. The solution is very efficient and is implemented in functions that have to be called any way. There is very little to gain from changing this solution so no others have been considered. If the fractal generator were to be configured at a different time, one would have to make sure that the configurations stay synchronized with the GP. No method to do this have been identified.

7.3.1 The Fractal Resolution

As mentioned in Section 5.1 the resolution of the landscape created by the fractal generator is equal to $NUMPOINTS \cdot NUMPOINTS$. It also states that when using a single index array for a landscape, the maximum value of $NUMPOINTS$ is 128.

The demo was originally written to draw several 128x128 landscapes and

connect them together into a big high resolution fractal. The plan was to draw each 128x128 fractal with separate draw calls, configuring the fractal generator for each call. The plan assumed that one draw call was made into one GP job on Mali, which would make it possible to configure the fractal generator correctly for all the landscapes. However, when the driver sees that draw calls are combined to draw a single frame, it also combines the draw calls into one big GP job. No method to separate the calls into separate GP-jobs was identified. Thus, one can not use the intended method to increase the resolution since one doesn't know when to configure the fractal generator. There is no identified way for the driver to extract which draw call the GP is currently working on.

The resolution of the demo in this thesis has been limited to $128 \cdot 128$ and improving the resolution has been left for future work. Below is a list of suggestions to increase the resolution.

1. Change the driver to split the draw calls into separate jobs. This would likely require many changes to the driver, but all hardware could be left as-is. Separate jobs leads to more set-up overhead, so the performance would be worse than with a single job.

2. Make the driver monitor the combined GP job and understand when it is working on a part of the landscape. Then configure the fractal generator to create that landscape. This method requires intimate knowledge of the GP to determine what it is currently doing. In addition there is no easy way to monitor the hardware from the driver. It might require big changes to the Mali GP or constant monitoring of memory, so the method is not seen as feasible.

3. Currently the landscape calculated by the fractal generator is limited by the index array describing the landscape. Instead of having one VBO(and one index array) for the vertices from the fractal generator, one could split a landscape into several VBOs and give each VBO a separate index array. This would allow the fractal generator to create a big landscape where each VBO is assigned a part of the landscape and an address range on the fractal generator.

   This would require changes to the hardware of the fractal generator, at least the addressing has to be changed, but it is definitely a feasible method.

8 Profiling

To determine if the hardware fractal generator is useful its performance has to be measured. This chapter will measure the average frame time when the fractal generator is running the demo and compare the results with the software implementation of the fractal generator [16].

The first performance measure is done inside the demo. However, measuring performance in the demo is not straightforward since the landscape is not drawn on screen at any exact time in the program; the calls and data is pipelined in hardware. A good practical solution is to measure the time between consecutive returns of the swapbuffer command[20]

[17]. Even if this does not give the exact time taken by the hardware to draw a frame, it is good enough to use as a comparison tool between solutions.

The performance will be measured for different amounts of FPGs in the fractal generator, and also varyings of the max iterations parameter.

It is important to note that the software fractal generator is running on the A9 CPU on the platform. This processor is not on the FPGA and is capable of running at 400Mhz [?], a much larger frequency than the circuit on FPGA. The frequency of the hardware version will also vary by small amounts, this is because of the complicated routing done by the synthesis tool. Certify uses random seeds when it routes so the results may vary slightly.

The table below presents the achieved results. SW/HW is the software and hardware versions of the fractal generator. MAX_ITER is the number of iterations at each point. NU is the number of FPGs, i.e. the number of points that are calculated in parallel. Time is the average frame time over 500 frames of the demo. FRQ is the frequency of the fractal generator used in that test.

To get more precise measure of how the fractal generator is doing, another performance measurement is done using the instrumented mode of the Mali drivers. This mode dumps a lot of performance data to a text file, which allows a precise measurement of the hardware jobs. This will show where the time is consumed in hardware while avoiding the time consumption of the demo.

[16] Frame time is the time taken to draw a single frame and is the inverse of frames pr second(FPS). Frame time is considered to be a more accurate performance measure than FPS since frame time is a linear value.

[17] The swapbuffer command posts the current color buffer to the native screen.

HW/SW	MAX_ITER	NU	TIME(s)	FRQ
SW	80	N/A	0.757	400 Mhz
SW	160	N/A	1.167	400 Mhz
HW	80	64	0.316	20 Mhz
HW	160	32	0.427	20 Mhz
HW	80	32	0.429	20 Mhz
HW	80	16	0.431	20 Mhz
HW	80	4	0.574	20 Mhz

Table 2: Average frame time performance over 500 frames of the demo.

Since the instrumented drivers are much slower when performing the measurements, running 500 frames of the demo is not feasible. Instead, this measurement will test the computing capabilities of the fractal generator in the worst-case scenario. The worst-case scenario is a landscape where each point reaches the $MAX\_ITERATIONS$ limit, and the measure is performed over a duration of 30 frames. In addition to this, the performance dump allows frequency scaling, so the frame time at higher frequencies can be estimated. This will be used to estimate the actual frame time of the fractal generator at 400 Mhz, to compare properly with the software implementation.

HW	MAX_ITER	NU	FT(ms)	VST(ms)	PLBUT(ms)	FPS
HW	80	64	2.592	0.637	0.622	450
HW	80	32	3.907	0.637	2.713	261

Table 3: Average geometry processor frame time, vertex shader time, and Polygon List Builder Unit time during 30 frames of the worst-case scenario. All results estimated at 400 Mhz.

The detailed measurements show that the vertex shader time is the same for both fractal configurations, this is as expected since the delay should be ahead of the vertex shader. Increasing the performance of the fractal generator instead improves the PLBU time.

9 Conclusion

The fractal generator designed in this thesis is working as intended. It is able to calculate the heights of a landscape based on the Mandelbrot set and provide the Mali-400 GPU with the results. The fractal generator is designed with scaleable performance at the cost of area.

The area consumption of the design is equal to 1826 LUTs plus 1750 LUTs pr FPG. For a system with 32 FPGs the area consumption is about 58k LUTs. The achieved clock frequency of the fractal generator is 48.1Mhz on a Virtex-6x FPGA. This frequency is higher than Mali's maximum frequency so the frequency of the system is limited by Mali.

The fractal generator outperformed a software implementation of itself by a large margin. With 64 cores running at 20Mhz, the demo running with the fractal generator achieved a total frame time of 0.3 seconds.

Without the demo, the fractal generator with 64 cores achieved a frame time of 2.592ms at 400Mhz which equals 51.8 ms at 20Mhz. The performance of the fractal generator is fast enough to generate real time 3D-landscapes. Thus, the thesis has successfully implemented a useful fractal generator for landscape generation.

9.1 Future Work

Scaling the resolution of the fractal landscape did not work as intended(Section 7.3.1). Increasing the resolution has been left for future work.

The fractal generator is currently storing all the z-coordinates of the landscape in cache, this is not necessary for operation and optimizing the cache size has been left for future work(Section 5.6.2).

The fractal generator is currently transferring its z-coordinates to Mali in 32 bit floating point format, reducing the format to 16 bits floating point is possible without loss of detail and has been left for future work.

References

[1] Hamilton B.Carter and Shankar G. Hemmady. *Metric Driven Design Verification.* 2007.

[2] Bodil Branner. The mandelbrot set. *Proceedings of Symposia in Applied Mathematics*, 39:75–105, 1989.

[3] Tolga Capin, Kari Pulli, and Tomas Akenine-Moller. The state of the art in mobile graphics research. 2008.

[4] Per Christian Corneliussen. Design of a fractal generator for on-the-fly generation of textures for mali gpu. Master's thesis, Norwegian University of Science and Technology, 2011.

[5] David Darling. *The Universal Book of Mathematics: From Abracadabra to Zeno's Paradoxes.* 2004.

[6] Dan Ginsburg Dave Shreiner, Aaftab Munshi. *OpenGL ES 2.0 Programming Guide.* 2008.

[7] Richard S. Wright et al. *OpenGL SuperBible.* 2010.

[8] Clayton Shepard et.al. Livelab: Measuring wireless networks and smartphone users in the field. 2011.

[9] Inc. Wikimedia Foundation. The mandelbrot set. http://en.wikipedia.org/wiki/Mandelbrot_set.

[10] Adrien Douady John H. Hubbard. *Étude dynamique des polynômes complexes.* 1984.

[11] Robert J.Simpson and John Kessenich. *The OpenGL ES Shading Language*, 2009.

[12] William K. Lam. *Hardware Design Verification.* 2005.

[13] ARM Limited. *Mali-400 MP GPU Integration Manual*, 2009.

[14] ARM Limited. *AMBA APB Protocol Specification*, 2010.

[15] ARM Limited. *Mali-400 MP GPU Configuration and Sign-off Guide*, 2010.

[16] ARM Limited. *Mali-400 MP GPU Technical Overview*, 2010.

[17] ARM Limited. *AMBA AXI and ACE Protocol Specification*, 2011.

[18] ARM Limited. *Mali-200 and Mali-400 MP GPU Linux DDK Integration Guide*, 2011.

[19] ARM Limited. *Mali-200 and Mali-400 MP GPU OpenGL ES Technical Overview*, 2011.

[20] OpenGL.org. Performance. http://www.opengl.org/wiki/Performance.

[21] Sharon Rosenberg and Kathleen A Meade. *A Practical Guide to Adopting the Universal Verification Methodology*. 2010.

[22] Chris Spear. *System Verilog for Verification*. 2006.

10 Appendices

A Source Code for the Fractal Generator

```
module apb_interface(
    input CLK,
    input RESET_N,
    // APB signals for receiving start coordinates from Mali
        driver.
    input PSEL,  //x ?
    input PWRITE,
    input [31:0] PWDATA,
    input [31:0] PWADDR,
    input PENABLE,// Currently unused cause I assume no wait
        states.TODO: OK?
  output reg PREADY,
  output reg [31:0] PRDATA,
    // Signals to and from the fractal generator core(FGC)
    output reg [31:0] x_fgc, // x coordinate of bottom left pixel
    output reg [31:0] y_fgc,
    output reg [31:0] stepsize_fgc, //stepsize between x and y
        coords.
  input [31:0] AXI_DEBUG,
  input [31:0] ARBITER_DEBUG
    );

// CSM for the fractal_generator
reg [1:0] state;
parameter idle = 0, receive = 1, send=2;
// Don't know how the APB addressing works entirely so this
    parameter can be
// set to change the address space of the apb_interface module.
// Currently x = start_address, y=x+1,stepsize = y+1;
parameter start_address = 32'h0000C000; //

always @ (posedge CLK or negedge RESET_N)
begin
  if (!RESET_N) begin
  state <= idle;
  PREADY <= 0;
  x_fgc <= 0;
  y_fgc <= 0;
  stepsize_fgc <= 0;
  PRDATA<= 0;
 end
 else begin
  case (state)
```

```
40      idle: begin // wait for APB activity, store data.
         if ((PSEL && PWRITE) == 1) begin
42        // no wait states needed.so doesn't have to check enable
          if (PWADDR[15:0] == 16'hC000) begin
44          state <= receive;
           PREADY <= 1;
46          x_fgc <= PWDATA;
          end
48        else if (PWADDR[15:0] == 16'hC004) begin
           state <= receive;
50         PREADY <= 1;
           y_fgc <= PWDATA;
52        end
          else if (PWADDR[15:0] == 16'hC008) begin
54         state <= receive;
           PREADY <= 1;
56         stepsize_fgc <= PWDATA;
          end
58       end
         else if ((PSEL && ~PWRITE) == 1) begin
60        // APB read interface for debugging purposes.
          //TODO: Rewrite into case
62        if (PWADDR[15:0] == 16'hC000) begin
           state <= send;
64         PREADY <= 1;
           PRDATA <= x_fgc;
66        end
          else if (PWADDR[15:0] == 16'hC004) begin
68         state <= send;
           PREADY <= 1;
70         PRDATA <= y_fgc;
          end
72        else if (PWADDR[15:0] == 16'hC008) begin
           state <= send;
74         PREADY <= 1;
           PRDATA <= stepsize_fgc;
76        end
          else if (PWADDR[15:0] == 16'hc00c) begin
78         state <= send;
           PREADY <= 1;
80         PRDATA <= ARBITER_DEBUG;
          end
82        else if (PWADDR[15:0] == 16'hc010) begin
           state <= send;
84         PREADY <= 1;
           PRDATA <= AXI_DEBUG;
86        end
          else if (PWADDR[15:0] == 16'hc014) begin
88         state <= send;
```

```
      PREADY <= 1;
      PRDATA <= 32'h1337feed;
     end
    end // else if
   end
   receive: begin
    PREADY <= 0;
    state <= idle;
   end
   send: begin
    PREADY <= 0;
    state <= idle;
    PRDATA <= 0; // TODO: This is not needed I suppose.
   end
  endcase
 end // else if reset is off
end // CSM APB
endmodule
```

../source/generator/apb_interface.v

```
// The job of the arbiter is to calculate fractal coordinates
    based on addresses
// from the axi_interface. If no priority addresses are given the
     arbiter tries
// to predict coordinates and put them into cache.
//


// Does log2(n) :)
`define clogb2(n) ((n) <= (1<<0) ? 0 : (n) <= (1<<1) ? 1 :\
(n) <= (1<<2) ? 2 : (n) <= (1<<3) ? 3 :\
(n) <= (1<<4) ? 4 : (n) <= (1<<5) ? 5 :\
(n) <= (1<<6) ? 6 : (n) <= (1<<7) ? 7 :\
(n) <= (1<<8) ? 8 : (n) <= (1<<9) ? 9 :\
(n) <= (1<<10) ? 10 : (n) <= (1<<11) ? 11 :\
(n) <= (1<<12) ? 12 : (n) <= (1<<13) ? 13 :\
(n) <= (1<<14) ? 14 : (n) <= (1<<15) ? 15 :\
(n) <= (1<<16) ? 16 : (n) <= (1<<17) ? 17 :\
(n) <= (1<<18) ? 18 : (n) <= (1<<19) ? 19 :\
(n) <= (1<<20) ? 20 : (n) <= (1<<21) ? 21 :\
(n) <= (1<<22) ? 22 : (n) <= (1<<23) ? 23 :\
(n) <= (1<<24) ? 24 : (n) <= (1<<25) ? 25 :\
(n) <= (1<<26) ? 26 : (n) <= (1<<27) ? 27 :\
(n) <= (1<<28) ? 28 : (n) <= (1<<29) ? 29 :\
(n) <= (1<<30) ? 30 : (n) <= (1<<31) ? 31 : 32)


module arbiter(
```

```
 input CLK,
 input RESET_N,
 input [31:0] ADDRESS, // this is a priority address from the
     axi_interface.
 input PRIORITY, // shouldn't be needed but here for now.
  input [31:0] FRAME1_X, FRAME1_Y, FRAME1_STEPSIZE,
 output reg [31:0] CACHE_ADDRESS,

 // CACHE WRITER SIGNALS
 output reg CACHE_WRITE, // tells the cache to write
 output reg [7:0] CACHE_DATA,
 /* ARBITER_DEBUG is used to debug via tha APB-interface.
  * It currently keeps track of some of the arbiter states.
  * This is mainly done to confirm that the arbiter has started
     via the driver.
 */
 output reg [31:0] ARBITER_DEBUG
);

//CONFIGURABLE PARAMETERS:
parameter NUMPOINTS = 32'd400;//Number of points in each
    direction.
parameter NUMUNITS = 16;  //Number of fractal point generators(
    fpg)
parameter LOG2_NUMPOINTS =`clogb2(NUMPOINTS);


// ARBITER - FPG COMMUNICATION TODO: Comment what these regs are
    for.
reg [31:0] feed_x, feed_y;
// fpg_address is the input address to the fpgs, while
    address_reg is a temp
// register used for calculating feed_x and feed_y. Might not
    need both TBH.
wire[31:0] ADDRESS_reg, fpg_address; // TODO: Need two
    assignments? Try to smarten it up.
reg [0:NUMUNITS-1]fpg_en;
wire [1:0]fpg_busy[0:NUMUNITS-1];
reg [0:NUMUNITS-1] fpg_read ;
wire [7:0]fpg_n[0:NUMUNITS-1];
wire [31:0]fpg_addr_out[0:NUMUNITS-1];

generate
 //GenerateFPGs.
 genvar gi;
 for(gi=0; gi<NUMUNITS; gi=gi+1)begin:UNITS
  fpg fpgi(
   .clk(CLK), .reset_n(RESET_N),
   .c_re(feed_x), .c_im(feed_y), .enable(fpg_en[gi]), .addr_in(
```

```
        fpg_address),
68      .n(fpg_n[gi]), .busy(fpg_busy[gi]), .addr_out(fpg_addr_out[gi
        ]),
        .iterations_read(fpg_read[gi])
70    );
  end
72 endgenerate

74 // MEMORY FOR X AND Y COORDINATES
   // So that each coordinate only is calculated once.
76 reg [31:0] xmem_addr, ymem_addr;
   reg [32:0]  xmem_in, ymem_in; // 33 bits, 1 bit is a valid bit.
78 wire [32:0] xmem_out, ymem_out;
   reg xmem_en,xmem_we, ymem_en, ymem_we;
80
   block_ram #(NUMPOINTS, 33) xmem(
82  .CLK(CLK), .ADDRESS(xmem_addr),
    .DATA_IN(xmem_in), .DATA_OUT(xmem_out),
84  .WE(xmem_we), .EN(xmem_en)
    );
86
   block_ram #(NUMPOINTS, 33) ymem(
88  .CLK(CLK), .ADDRESS(ymem_addr),
    .DATA_IN(ymem_in), .DATA_OUT(ymem_out),
90  .WE(ymem_we), .EN(ymem_en)
    );
92
   // ARITMETHIC COMPONENTS
94 // Used for calculating FP32 numbers.
   reg [31:0] converter_in;
96
   wire[31:0] mul_in;
98 reg mul_enable;
   reg [31:0] add_in;
100 reg add_enable;
    wire [31:0] mul_out, add_out;
102 wire mul_valid, add_valid;

104 // 32bit integer to 32bit fp converter.
    vithar_lib_u32_to_f32 converter(.outp(mul_in), .inp(converter_in)
        );
106
    // fp32 multiplier
108 vithar_lib_f32_mul mul(
    .clk(CLK), .reset_n(RESET_N), .enable(mul_enable),
110 .a(FRAME1_STEPSIZE), .b(mul_in), .dout(mul_out),
    .valid(mul_valid)
112 );
```

```
// fp32 adder
vithar_lib_f32_addsub add(
 .clk(CLK), .reset_n(RESET_N), .enable(add_enable),
 .a(mul_out), .b(add_in), .dout(add_out),
 .valid(add_valid),
 .dout_guard() ,.dout_round() ,.dout_sticky() // TODO: Needed?
);


// ARBITER CSM START
reg [31:0] pred_address;

integer i; // loop counter.
reg [3:0] state;
// pred_status is 1 if the current coordinates are predicted, i.e
    not priority.
reg pred_status;
reg found; //Used in feed_coordinates to find an available fpg
reg [3:0] burst_offset;

// Burst offset is used to get all the coordinates in an AXI
    burst from just
// the initial address.
assign fpg_address = (pred_status) ? pred_address : (ADDRESS+
    burst_offset);
assign ADDRESS_reg =(PRIORITY) ? (ADDRESS+burst_offset) :
    pred_address;

parameter idle = 0, check_xy = 1, calculate_x=2, calculate_x2=3,
    calculate_y=4,
      calculate_y2 = 5,feed_coordinates = 6,flush_mem = 7,
      wait_cache_flush = 8;
always @ (posedge CLK or negedge RESET_N)
begin
 if (!RESET_N) begin
  // Flush memories upon reset
  ARBITER_DEBUG <= 0;
  xmem_addr <= (NUMPOINTS-1);
  xmem_en <= 1'b1;
  xmem_we <= 1'b1;
  xmem_in <= 33'b0;
  ymem_en <= 1'b1;
  ymem_addr <= (NUMPOINTS-1);
  ymem_we <= 1'b1;
  ymem_in <= 33'b0;
  state <= flush_mem;
  // Initial signal values
  feed_x <= 0;
  feed_y <= 0;
```

```
158     fpg_en <= 0;
        pred_address <= 0;
160     pred_status <= 0;
        burst_offset <= 0;
162     converter_in <= 0;
        mul_enable <= 0;
164     add_in <= 0;
        add_enable <= 0;
166   end
      else begin
168   case(state)
       wait_cache_flush: begin
170     ARBITER_DEBUG <= 1;
        // Wait for the axi_interface to finish flushing the cache.
172     // Needed to avoid writing to the cache during flush.
        if(ADDRESS == 32'hffffffff) begin
174       state <= wait_cache_flush;
        end
176     else begin
         state <= idle;
178     end
       end
180    idle: begin
        fpg_en <= 0; // fpg_enable is set in feed_coordinates, only
            need high for 1 cycle.
182     if (pred_status) begin
         // previous coordinate was a predicted one.
184      pred_status <= 0;
         pred_address <= pred_address + 1;
186     end
        if(pred_address >= (NUMPOINTS*NUMPOINTS)) begin
188      // Predictor has predicted all needed coordinates here.
         // wait for reset
190      state <= idle;
        end
192     else if (FRAME1_STEPSIZE != 0) begin
         // Stepsize is the last to get updated in my TB. TODO: MAli
             might
194      // start with stepsize...make generaL? :)
         // ADDRESS_reg mod NUMPOINTS, assumes that NUMPOINTS = 2^n
196      xmem_addr <= ADDRESS_reg & (NUMPOINTS-1);
         // ADDRESS_reg / NUMPOINTS, same assumption.
198      ymem_addr <= ADDRESS_reg >> LOG2_NUMPOINTS;
         xmem_en <= 1;
200      ymem_en<= 1;
         state <= check_xy;
202      pred_status <= !PRIORITY;
        end
204   end //idle
```

```
206   check_xy: begin
       ARBITER_DEBUG <= 2;
208    if ((xmem_en && xmem_we) || (ymem_en && ymem_we)) begin
        // a write has been done during this cycle, write has
            priority so
210     // data has now been written regardless of read. Need to set
            write low
        // to get data out.
212     xmem_we <= 0;
        ymem_we <= 0;
214     state <= check_xy;
       end
216    else if ((ymem_out[32] && xmem_out[32]) == 1)begin
        // The values exist in the rams and are valid.
218     feed_y <= ymem_out;
        feed_x <= xmem_out;
220     state <= feed_coordinates;
        xmem_en <= 0;
222     ymem_en <= 0;
       end
224    else if (xmem_out[32] == 1) begin
        // If xmem == 1, ymem != 1, could be x so need to check this
            way.
226     converter_in <= ymem_addr;
        mul_enable <= 1;
228     state <= calculate_y;
       end
230    else begin
        // Either ymem == 1, or (xmem && ymem) != 1. Both could be 0
            or x.
232     // Calculate xmem.
        converter_in <= xmem_addr;
234     mul_enable <= 1;
        state <= calculate_x;
236    end
      end // check_xy
238
      calculate_y: begin
240    // Calculate the Y coordinate corresponding to the address
          given.
       // Y =  (FRAME1_Y - (FRAME1_STEPSIZE*(ADDRESS/NUMPOINTS)
242    // Y =  FRAME1_Y - FRAME1_STEPSIZE*ymem_addr
        /* The sign of the y coordinate is inverted to enable
            subtraction using the same adder.
244      The sign of the result is also inverted to obtain the
            correct result.
        (-y+x)*-1 = y-x
246    */
```

```
    if(mul_valid) begin
248     //fp32 addition.
        add_in[30:0] <= FRAME1_Y[30:0];
250     add_in[31] <= ~FRAME1_Y[31];
        add_enable <= 1;
252     state <= calculate_y2;
      end
254   else begin
        state <= calculate_y;
256   end
    end
258 calculate_y2: begin
      if(add_valid) begin
260     ymem_in[32] <= 1'b1;
        ymem_in[30:0] <= add_out;
262     ymem_in[31] <= ~add_out[31];
        ymem_we <= 1;
264     add_enable <= 0;
        mul_enable <= 0;
266     state <= check_xy;
      end
268   else begin
        state <= calculate_y2;
270   end
    end
272
    calculate_x: begin
274   // Calculate the X coordinate corresponding to the address
          given.
      // X = FRAME1_X + FRAME1_STEPSIZE*(ADDRESS % NUMPOINTS);
276   // X = FRAME1_X + FRAME1_STEPSIZE*xmem_addr;
      if(mul_valid) begin
278     //fp32 addition
        add_in <= FRAME1_X;
280     add_enable <= 1;
        state <= calculate_x2;
282   end
      else begin
284     state <= calculate_x;
      end
286 end

288 calculate_x2: begin
      if(add_valid) begin
290     xmem_in[32] <= 1'b1;
        xmem_in[31:0] <= add_out;
292     xmem_we <= 1;
        add_enable <= 0;
294     mul_enable <= 0;
```

```
      state <= check_xy;
296    end
       else begin
298      state <= calculate_x2;
       end
300  end

302  feed_coordinates: begin
       // Finds a free fpg. Enable is reset to 0 in idle.
304    found = 0;
       for (i=0;i<NUMUNITS;i=i+1) begin: find_free_fpg
306      if(!found) begin
           if(fpg_busy[i]==2'b00) begin
308          fpg_en[i] <= 1;
             found = 1;
310          state <= idle;
             // Adjust burst_offset to get the next coordinate in the
                burst.
312          // Needs to be inside loop to only add when state changes
                :)
             if(!pred_status) begin
314            if(burst_offset == 15)
                 burst_offset <= 0;
316            else
                 burst_offset <= burst_offset + 1;
318          end
           end
320      end
       end
322  end

324  flush_mem: begin
       // Reset all values in xmem and ymem to zero.
326    if(!ymem_addr) begin
         // Flush complete
328      state <= wait_cache_flush;
         xmem_en <= 1'b0;
330      xmem_we <= 1'b0;
         ymem_en <= 1'b0;
332      ymem_we <= 1'b0;
       end
334    else begin
         // Iterate through memories.
336      // Inputs are zero, write and enable == 1 here.
         xmem_addr <= xmem_addr - 1'b1;
338      ymem_addr <= ymem_addr - 1'b1;
         state <= flush_mem;
340    end
     end // flush_mem
```

```
342  endcase
     end // else RESET_N
344 end

346

348 // CACHE_WRITER - writes finished coordinates to cache.
    reg writer_state;
350 reg z_found; // Used to find finished fpg
    integer j; // writer fpg counter
352 parameter  write = 1;

354 always @ (posedge CLK or negedge RESET_N)
    begin
356  if (!RESET_N) begin
      // reset stuff
358   CACHE_WRITE <= 0;
      CACHE_DATA <= 0;
360   CACHE_ADDRESS <= 0;
      fpg_read <= 0;
362   writer_state <= idle;
     end
364  else begin
     case(writer_state)
366   idle: begin
       z_found = 0;
368    for(j=0;j<NUMUNITS;j=j+1)begin: find_finished_fpg
        if (!z_found) begin
370      if (fpg_busy[j]==2'b10) begin
          // The fpg has calculated the number of iterations for the
             Z coordinate @ fpg_addr_out.
372       CACHE_ADDRESS <= fpg_addr_out[j];
          CACHE_DATA[7:0] <= fpg_n[j];
374       CACHE_WRITE <= 1;
          fpg_read[j] <= 1;
376       writer_state <= write;
          z_found = 1;
378      end
        end
380    end
      end
382   write: begin
       fpg_read<= 0; // Not reading from anyone here. They can keep
           working if free.
384    CACHE_WRITE <= 0;
       writer_state <= idle;
386   end
     endcase
388  end // else if RESET_N
```

```
end
endmodule
```

../source/generator/arbiter.v

```
// This module is an axi_slave responsible for transferring the
// Z-coordinates stored in the coordinate_cache through the AXI bus to MALI.
// The module reads the address given by Mali and the 31 subsequent addresses
// from the coordinate_cache. This will give a total of 32 Z-coordinates to
// transfer,16 each burst. Each coordinate is 8 bits.
// If any of the Z-coordinates are missing from the cache, the interface
// has to ask the FPG-arbiter to calculate them and then transfer.

// Written by Per Kristian Kjøll.

// Need to define parameter NUMPOINTS.

module axi_interface(
 // AXI SIGNALS - have only included the ones in use.
 input ACLK,
 input ARESET_N,
 input [31:0]ARADDR,
 input [3:0] ARLEN, // Number of bursts. Usually = 2
 input [4:0] ARID,
 // ARSIZE = max number of bytes in a burst, see table A3-2, Usually = 0b100 = 16
 // bytes
 input [2:0] ARSIZE,
 input [1:0] ARBURST, // Burst type. Usual = Incremental = 0b01
 input ARVALID,
 input RREADY, // Master is ready to accept the read data and response information
 output reg ARREADY, // Ready to accept address and control signals.
 output reg [4:0] RID,
 output reg RVALID,
 output reg RLAST,
 output reg [127:0] RDATA,
 output reg DEBUG, // signal to indicate unexpected inputs...for testing only.
 /* AXI_DEBUG is used to debug via APB-interface.
  * It currently counts the number of initiated AXI-reads.
 */
 output reg [31:0]AXI_DEBUG,
```

```
36  // COORDINATE_CACHE SIGNALS
    output reg [31:0] CACHE_ADDRESS, // cache has space for one
        frame currently.
38  output reg CACHE_READ_BURST,
    output reg CACHE_CLEAR_BURST,
40
    input [127:0] CACHE_BURST,
42  input [1:0]  CACHE_BURST_VALID,
    input ARBITER_WRITE, // The arbiter is currently writing to the
        cache.
44  // ARBITER SIGNALS
    // Used to calculate coordinates of the signals that are invalid
         at time of request.
46  output reg [31:0] ARBITER_ADDR, // TODO: Combine this with
        CACHE_ADDRESS?
    output reg ARBITER_PRIORITY
48   );

50 // AXI registers
   reg [1:0] burst_counter;
52 // Keeps track of how many of transfercoordinates have yet to be
       checked
   reg [127:0] burst1;
54 reg [127:0] burst2;

56 reg [2:0] state;
   parameter waiting_for_address = 0, check_coordinates_a=1,
       check_coordinates_b=2,
58       check_coordinates = 3, send_coordinates = 4,
             send_coordinates2 = 5,
         end_transfer = 6, flush_cache=7;
60
   parameter start_address = 32'h1000_0000; // This is the start of
       the AXI address region of the fractal generator.
62 // State machine
   always @ (posedge ACLK or negedge ARESET_N)
64 begin
    if (!ARESET_N) begin
66   state <= flush_cache;
     CACHE_ADDRESS <= 32'h00003ffc; // TODO: Check, 3fea = 128x128
         last address.
68   CACHE_CLEAR_BURST <= 1;
     ARBITER_ADDR <= 32'hffffffff;
70   burst1 <= 0;
     burst2 <= 0;
72   burst_counter <= 0;
     ARREADY <= 0;
74   RID <= 0;
     RVALID <= 0;
```

```
76    RDATA <= 0;
      RLAST <= 0;
78    DEBUG <= 0;
      CACHE_READ_BURST <= 0;
80    ARBITER_PRIORITY <= 0;
      AXI_DEBUG <= 0;
82  end
    else begin
84  case(state)
    waiting_for_address: begin
86    ARREADY <= 1;
      if(ARVALID == 1) begin
88      state <= check_coordinates_a;
        AXI_DEBUG <= AXI_DEBUG + 1'b1;
90      RID <= ARID;
        // This works since address is 0x1000_0000
92      // ARADDR is divided by 4. Each coordinate is 4 bytes and in
            the
        // cache each coordinate has its own address while externally
            it is
94      // byte addressed.
        CACHE_ADDRESS[31:28] <= 4'b0000;
96      CACHE_ADDRESS[27:0] <= (ARADDR[27:0]>>2); // General : ARADDR
            [31:0]-start_address; // TODO: Faster general way?
        CACHE_READ_BURST <= 1;
98      // Number of bursts = ARLEN + 1
        if(ARLEN == 1) begin
100       burst_counter <= 2;
        end
102     else if(ARLEN == 0) begin
          // burst_counter = 0 supports bursts of length 1.
104       DEBUG <= 1'b1;
          burst_counter <= 0;
106     end
        else begin
108       // Interface only supports ARLEN==1 and ARLEN==2
          DEBUG <= 1'b1;
110     end
        if( (ARBURST != 2'b01) || (ARSIZE != 3'b100) ) begin
112       DEBUG <= 1; // Not incremental burst, find out why.
        end
114   end
    end
116
    check_coordinates_a: begin
118   // Read burst arrives to axi on the second cycle. Wait a cycle
          here.
      state <= check_coordinates_b;
120 end
```

```
check_coordinates_b: begin
 state <= check_coordinates;
end

check_coordinates: begin
 // Checks if all coordinates in burst are valid, if not calculates the
 // ones that aren't by using the arbiter.
 // TODO: This uses 1 cycle each coordinate, ihope. Could modify coordinate_cache
 // to check all coordinates in a cycle..but then would not know which, if
 // any fails. Or I could, with a big check vector..hmm.
 // Find out how often all coords are valid..if they are most of
 // the time, make a faster check. Potential speedup here, if it is too
 // slow.
 if(CACHE_BURST_VALID[0] == 1'b1) begin
  // The entire burst is valid.
  ARBITER_ADDR <= 0;
  ARBITER_PRIORITY <= 0;
  burst_counter <= burst_counter - 1;
  if(burst_counter == 1) begin
  // Store the second burst.
   burst2 <= CACHE_BURST;
   CACHE_READ_BURST <= 0;
   state <= send_coordinates;
  end
  else begin
   // Store the first burst
   burst1 <= CACHE_BURST;
   CACHE_ADDRESS <= CACHE_ADDRESS + 4; // TODO: Check this.
   state <= check_coordinates_a;
  end
 end
 else if(CACHE_BURST_VALID[1] == 1'b1) begin
  // There was a write during the last read. This should rarely happen.
  // Burst might be valid here but not updated yet. Wait.
  state <= check_coordinates;
 end
 else begin
  // The burst is not valid, send the intital address of the burst to
  // arbiter. Some of the coordinates in the burst might be ready, but no matter.
  ARBITER_ADDR <= CACHE_ADDRESS; // stay in same state until valid.
  ARBITER_PRIORITY <= 1;
```

```
162     state <= check_coordinates;
        end
164   end

166   send_coordinates: begin
      // Sends two bursts in two cycles.
168   // TODO: Check timings here more carefully.
       if(RREADY <= 1) begin
170     RDATA <= burst1;
        RVALID <= 1; // timing?
172     if (burst_counter == 2'b00) begin
         state <= send_coordinates2;
174     end
        else begin
176       // TODO: As of now, this will never happen?
          // The burst_counter started at 0 (ARLEN == 0, burst length =
              1)
178       RLAST <= 1;
          state <= end_transfer; // This is added to support ARLEN == 0
180     end
       end
182    else
        state <= send_coordinates;
184   end

186   send_coordinates2: begin
       RLAST <= 1;
188    RDATA <= burst2;
       state <= end_transfer;
190   end

192   end_transfer: begin
       RVALID <= 0;// Here or the previous cycle?
194    RLAST <= 0;
       ARREADY <= 0; // TODO: Don't think this is necessary
196    state <= waiting_for_address;
      end
198
      flush_cache: begin
200    // Deletes all data in the cache. This shall be done at reset
           between each frame.
       if(!CACHE_ADDRESS) begin
202     // Flush complete
        state <= waiting_for_address;
204     CACHE_CLEAR_BURST <= 1'b0;
        ARBITER_ADDR <= 32'h00000000;
206    end
       else begin
208     CACHE_ADDRESS <= CACHE_ADDRESS - 4;
```

```
    state <= flush_cache;
  end
 end // flush_cache

 default: begin
  //shouldn't happen ^^,
  DEBUG <= 1;
 end
 endcase
 end // reset_n else
end
endmodule
```

../source/generator/axi_interface.v

```
// block ram for use with the coordinate cache module.
// Simple synchronous ram to make use of the Xilinx block ram on
    the FPGA.
//


module block_ram(CLK, ADDRESS, WE, EN, DATA_IN, DATA_OUT);

parameter DEPTH = 1024;
parameter WIDTH = 8; // bit pr read burst.

input CLK,WE,EN;
input [31:0] ADDRESS;
input [WIDTH-1:0] DATA_IN;

output reg [WIDTH-1:0] DATA_OUT;


// Note that the last width bit is used as a valid bit.
reg [WIDTH-1:0] mem [0:(DEPTH-1)];

always @ (posedge CLK)
begin
 if (EN) begin
  if(WE==1)
   // Displays new value ASAP.
   DATA_OUT <= DATA_IN;
  else
   DATA_OUT <= mem[ADDRESS];
 end
 else
  DATA_OUT <= 0;
end
```

```
34 always @ (posedge CLK)
   begin
36  if (EN && WE)
    begin
38    mem[ADDRESS] <= DATA_IN;
    end
40 end
   endmodule
```

../source/generator/block_ram.v

```
1  // Explain functionality here :)

3
   // Set syn_preserve synplify directive to deactivate equivalent
       optimization.
5  // Otherwise it optimizes away several of the RAM-blocks.

7
   module coordinate_cache(
9   input CLK,
    input [31:0]AXI_ADDRESS,
11  input [31:0]ARBITER_ADDRESS,
    input READ_COORDINATE,
13  input WRITE,
    input READ_BURST, // special function to speed up the usual
        request behavior.
15  input [7:0] DATA_IN, // data in should be also contain the two
        status bits :)
    input CLEAR_BURST,
17
    output [1:0] BURST_VALID,
19  output reg [7:0] COORDINATE, //data out from single read
    output [127:0]COORDINATE_BURST //data out from burst read
21    );

23
   //reg [33:0] mem_bank [0:(NUMPOINTS^2)];
25 // mem_bank has two extra status bits in addition to the 8 for Z-
       iterations;
   // read and valid. Read is used by arbiter and valid is used by
       axi.
27 parameter xpoints = 256;
   parameter ypoints = 256;
29 parameter total_points = (xpoints*ypoints);
   parameter BLOCKSIZE = total_points/4;
31

33 parameter NUMBER_OF_RAMS = 4;
```

```
35
reg [31:0] blockAddress;
37
reg [7:0] ram_in;
39 reg we;
reg [NUMBER_OF_RAMS-1:0]en;
41 wire [7:0]blockOut [NUMBER_OF_RAMS-1:0];
wire byte_valid;
43 reg write_during_read;

45 assign BURST_VALID[0] = byte_valid;
assign BURST_VALID[1] = write_during_read;
47


49
// Generate memories. These should be inferred to Xilinx block
   RAM.
51
generate
53  genvar k;
 for(k=0;k<NUMBER_OF_RAMS;k=k+1)begin:BLOCKS
55   /* synthesis syn_preserve = 1 */
   block_ram #(BLOCKSIZE) ram(
57    .CLK(CLK), .EN(en[k]),
    .ADDRESS(blockAddress), .WE(we),
59    .DATA_IN(ram_in), .DATA_OUT(blockOut[k])
   );
61  end
endgenerate
63
// Convert burst output from u8 to f32. This is done since
65 // float is requiered(?) when using attributes in the shader.
wire [31:0] u32_in [3:0];
67 wire [31:0] f32_out [3:0];

69 generate
genvar k;
71  for (k=0;k<4;k=k+1) begin
   vithar_lib_u32_to_f32 u_vithar_lib_u32_to_f32
73   (
    .inp(u32_in[k]), .outp(f32_out[k])
75   );
  end
77 endgenerate


79
assign u32_in[0] = blockOut[0];
81 assign u32_in[1] = blockOut[1];
```

```
assign u32_in[2] = blockOut[2];
assign u32_in[3] = blockOut[3];

assign COORDINATE_BURST[31:0] = f32_out[0];
assign COORDINATE_BURST[63:32] = f32_out[1];
assign COORDINATE_BURST[95:64] = f32_out[2];
assign COORDINATE_BURST[127:96] = f32_out[3];

// If either of the bursts are all zeroes, the burst is not valid.
assign byte_valid=((COORDINATE_BURST[31:0] & COORDINATE_BURST[63:32] &
        COORDINATE_BURST[95:64] & COORDINATE_BURST [127:96])
        == 32'b0) ? 1'b0 : 1'b1;
// Block RAM control
// TODO: This does not have to be clocked? :)
always @ ( posedge CLK) begin
 // Write has priority.
 // Arbiter never has to read, only write.
 ram_in <= DATA_IN;
 if (WRITE) begin
  we <= 1;
  en <= 0;
  en[ARBITER_ADDRESS % NUMBER_OF_RAMS] <= 1;
  COORDINATE <= 0;
  blockAddress <= (ARBITER_ADDRESS >> 2);
  if(READ_BURST) begin
   write_during_read <= 1;
  end
 end
 else if(CLEAR_BURST == 1) begin
  // Clears one row in the RAM.
  ram_in <= 8'h00;
  we <= 1;
  en[3:0] <= 4'hf;
  blockAddress <= (AXI_ADDRESS >> 2);
 end
 else if (READ_BURST) begin
   we <= 0;
   blockAddress <= (AXI_ADDRESS >> 2);
   en[3:0] <= 4'hf;
 end
 else begin
  we <= 0;
  en <= 0;
  blockAddress <= 0;
  write_during_read <= 0;
  COORDINATE <= 0;
 end
```

```
129 end
    endmodule
```

../source/generator/coordinate_cache.v

```
    // This is top-level of the fractal generator, it will connect
       all the sub-components.
 2
    module fractal_generator_main(
 4   // General
     input CLK,
 6   input RESET_N,
     // APB signals for receiving start coordinates from Mali driver.
 8   input PSEL,  //x ?
     input PWRITE,
10   input [31:0] PWDATA,
     input [31:0] PWADDR,
12
     input PENABLE,
14   output PREADY,
     output [31:0] PRDATA,
16   // AXI I/O
     input [4:0] ARID,
18   input [31:0]ARADDR, // How big should this be?
     input [3:0] ARLEN, // Only need 4 bits, AXI 3.
20   input [2:0] ARSIZE,// Max number of bytes in a burst, see table
       A3-2, Usually =
     input [1:0] ARBURST, // Burst type
22   input ARVALID,
     input RREADY, // Master is ready to accept the read data and
       response information.
24   // lots of extra inputs that the module doesn't care about.
     output ARREADY, // Ready to accept address and control signals.
26   output DEBUG,
     // data is 128 bits each burst, usually two bursts => 16*2 = 32
       = 8 coordinates
28   output [4:0] RID,
     output [127:0] RDATA,
30   output RVALID, RLAST
    );
32  wire [31:0] FRAME1_X, FRAME1_Y,FRAME1_STEPSIZE; // APB -> ARBITER
    wire [127:0] CACHE_BURST_OUT; // CACHE -> AXI
34  wire [31:0] CACHE_ADDRESS_AXI, CACHE_ADDRESS_ARBITER,
      ARBITER_ADDR;
    wire [7:0] CACHE_DATA;
36  wire [7:0] CACHE_READ_OUT; // TODO: Bad name IMO. Not currently
      used actually.
    wire [1:0] CACHE_BURST_VALID;
38  wire  CACHE_READ_BURST, CACHE_CLEAR_BURST,
```

```
    ARBITER_PRIORITY, CACHE_WRITE, CACHE_READ;
wire [31:0] ARBITER_DEBUG, AXI_DEBUG;
// resolution of fractal
//
parameter numpoints = 128; // Points in each direction, total res
    = np^2.
parameter numunits = 8; // The number of fpgs in the arbiter.

apb_interface apb(
 .CLK(CLK), .RESET_N(RESET_N),
  .PSEL(PSEL), .PWRITE(PWRITE), .PWDATA(PWDATA), .PWADDR(PWADDR),
 .PRDATA(PRDATA), .PENABLE(PENABLE), .PREADY(PREADY),
 .x_fgc(FRAME1_X), .y_fgc(FRAME1_Y), .stepsize_fgc(
    FRAME1_STEPSIZE),
 .AXI_DEBUG(AXI_DEBUG), .ARBITER_DEBUG(ARBITER_DEBUG)
 );

axi_interface axi(
 .ACLK(CLK), .ARESET_N(RESET_N),
 .ARID(ARID), .ARADDR(ARADDR), .ARLEN(ARLEN), .ARSIZE(ARSIZE),
 .ARBURST(ARBURST),.ARVALID(ARVALID), .RREADY(RREADY),
 .ARREADY(ARREADY), .DEBUG(DEBUG),
 .RID(RID), .RDATA(RDATA), .RVALID(RVALID), .RLAST(RLAST),
 .CACHE_ADDRESS(CACHE_ADDRESS_AXI), .CACHE_BURST(CACHE_BURST_OUT)
    ,
 .CACHE_READ_BURST(CACHE_READ_BURST), .CACHE_CLEAR_BURST(
    CACHE_CLEAR_BURST),
 .CACHE_BURST_VALID(CACHE_BURST_VALID), .ARBITER_WRITE(
    CACHE_WRITE), // Arbiter is writing to cache
 .ARBITER_ADDR(ARBITER_ADDR),.ARBITER_PRIORITY(ARBITER_PRIORITY),
 .AXI_DEBUG(AXI_DEBUG)
 );
  //need to add single to axi?

arbiter #(numpoints,numunits) arbiter(
 .CLK(CLK), .RESET_N(RESET_N), .ADDRESS(ARBITER_ADDR), .PRIORITY(
    ARBITER_PRIORITY),
 .FRAME1_X(FRAME1_X), .FRAME1_Y(FRAME1_Y), .FRAME1_STEPSIZE(
    FRAME1_STEPSIZE),
 .CACHE_ADDRESS(CACHE_ADDRESS_ARBITER),
 .CACHE_WRITE(CACHE_WRITE),.CACHE_DATA(CACHE_DATA),
 .ARBITER_DEBUG(ARBITER_DEBUG)
 );

 // Currently no single read operation done with the cache. Might
     need to
 // later, if not all transfers are in burstsizes.
coordinate_cache #(numpoints, numpoints) cache(
 .CLK(CLK),
```

```
80    .AXI_ADDRESS(CACHE_ADDRESS_AXI), .ARBITER_ADDRESS(
          CACHE_ADDRESS_ARBITER),
      .READ_COORDINATE(CACHE_READ), .WRITE(CACHE_WRITE),
82    .READ_BURST(CACHE_READ_BURST), .DATA_IN(CACHE_DATA),
      .COORDINATE(CACHE_READ_OUT), .COORDINATE_BURST(CACHE_BURST_OUT),
84    .CLEAR_BURST(CACHE_CLEAR_BURST), .BURST_VALID(CACHE_BURST_VALID)
      );
86
  endmodule // fractal_generator
```

../source/generator/fractal_generator_main.v

B Source Code for UVM Verification Framework

```
// Apb interface test classes
`ifndef APB_COMP_LIB_SVH
`define APB_COMP_LIB_SVH

// APB INTERFACE
interface apb_if; // APB protocol signals
 logic clk, resetn;
 logic psel, pwrite, pready, penable;
 logic [31:0] pwdata, pwaddr,prdata;
 logic [31:0] x_fgc, y_fgc, stepsize_fgc;
endinterface : apb_if

// APB DATA ITEM
class apb_item extends uvm_sequence_item;
 rand int unsigned pwaddr;
 rand int unsigned pwdata;
 int unsigned prdata;
 rand logic pwrite;
 //constraint c1 { pwaddr == 32'h000C000; } //Random value atm...


 // UVM automation macros
 `uvm_object_utils_begin(apb_item)
  `uvm_field_int(pwaddr, UVM_DEFAULT)
  `uvm_field_int(pwdata, UVM_DEFAULT)
 `uvm_object_utils_end

 //  Constructor
 function new (string name = "apb_item");
  super.new(name);
 endfunction : new
endclass : apb_item

// APB SEQUENCE AND SEQUENCER
class apb_sequencer extends uvm_sequencer #(apb_item);
 `uvm_component_utils(apb_sequencer)
 function new(string name, uvm_component parent);
  super.new(name, parent);
  $display("Apb sequencer");
 endfunction : new
endclass : apb_sequencer

class apb_sequence extends uvm_sequence #(apb_item);
 function new(string name="apb_sequence");
  super.new(name);
```

```
46    $display("Apb sequence");
  endfunction
48
  `uvm_object_utils(apb_sequence)
50
  virtual task body();
52   uvm_test_done.raise_objection(this, "APB sequence");
   `uvm_info("apb_sequence", "Starting sequence", UVM_MEDIUM)
54   `uvm_create(req)
   // Configure the fractal generator
56   // X0 = -1.5
   start_item(req);
58   req.randomize() with { pwaddr==32'h0000c000; pwdata==32'
       hbfc00000; pwrite ==1'b1;};
   finish_item(req);
60   // Y0 = 0.8
   start_item(req);
62   req.randomize() with { pwaddr==32'h0000c004; pwdata==32'
       h3f4ccccd; pwrite ==1'b1;};
   finish_item(req);
64   // STEPSIZE = 0.01
   start_item(req);
66   req.randomize() with { pwaddr==32'h0000c008; pwdata==32'
       h3c23d70a; pwrite ==1'b1;};
   finish_item(req);
68   // Check the version register to ensure correct setup.
   start_item(req);
70   req.randomize() with { pwaddr==32'h0000c014; pwrite ==1'b0;};
   finish_item(req);
72   if(req.prdata==32'h1337feed) begin
    $display("LEETFEED");
74   end
   else begin
76    $display("Ikke leet :(");
    $display("prdata = %h", req.prdata);
78   end
   // Wait for the clearing of caches and memories to complete.
80   // Use the DEBUG registers to check this
   // Check AXI_DEBUG
82
   forever begin
84    start_item(req);
    req.randomize() with { pwaddr==32'h0000c00c; pwrite ==1'b0;};
86    finish_item(req);
    if(req.prdata==32'h00000002) begin
88     $display("ARBITER STARTED. CACHE IS FLUSHED HERE. Ready to
         read via AXI.");
     uvm_test_done.drop_objection(this, "APB sequence");
90     break;
```

```
    end
92    else begin
      // Send delay item
94    $display("AXI not started. Prdata = %h", req.prdata);
      start_item(req);
96    req.randomize() with { pwaddr==32'h0000_0000; pwrite ==1'b0
          ;};
      finish_item(req);
98    end
    end
100 endtask
endclass : apb_sequence
102

104 //APB DRIVER
class apb_driver extends uvm_driver #(apb_item);
106  virtual apb_if apb_vif;
  // UVM automation macros
108  `uvm_component_utils(apb_driver)

110  // Constructor
  function new (string name = "apb_driver", uvm_component parent=
      null);
112   super.new(name,parent);
  endfunction : new
114
  // Build phase registers the vif resource
116  function void build_phase(uvm_phase phase);
   string inst_name;
118   super.build_phase(phase);
   if(!uvm_config_db#(virtual apb_if)::get(this,"","apb_vif",
       apb_vif))
120   begin
    `uvm_fatal("NOVIF",{"virtual interface must be set for: ",
        get_full_name(),".apb_vif"});
122   end
  endfunction : build_phase
124
  // run_phase retrieves data_items and drives them
126  task run_phase(uvm_phase phase);
   apb_item a_item;
128   super.run_phase(phase);
   @(posedge apb_vif.resetn);
130   forever begin
    seq_item_port.get_next_item(a_item);
132    drive_item(a_item);
    seq_item_port.item_done();
134   end
  endtask : run_phase
```

```
136
  // drive_item is the logic required to push the item onto the
      apb
138  // In this case it is the APB protocol signals and timing.
  task drive_item (input apb_item item);
140   if(item.pwaddr == 0) begin
     repeat(1000) @ (posedge apb_vif.clk);
142   end
    else begin
144    @ (posedge apb_vif.clk)
     begin
146     apb_vif.psel <= 1'b1;
      apb_vif.pwrite <= item.pwrite;
148     apb_vif.pwaddr <= item.pwaddr;
      if(item.pwrite == 1'b1) begin
150      apb_vif.pwdata <= item.pwdata;
       //$display("Her skal man sende et item");
152     end
      // read
154     else begin
       @(posedge apb_vif.pready);
156      //$display("Her skal man motta et item");
       item.prdata<= apb_vif.prdata;
158     end
     end
160    @ (posedge apb_vif.clk)
      apb_vif.psel <= 1'b0;
162     apb_vif.pwrite <= 1'b0;
   end //else
164  endtask : drive_item
endclass : apb_driver
166
// APB MONITOR
168 class apb_monitor extends uvm_monitor;
    virtual apb_if apb_vif;
170   bit checks_enable = 1;
    bit coverage_enable = 1;
172
  uvm_analysis_port#(apb_item) item_collected_port;
174  event apb_written_event; // Events needed to trigger covergroups
  protected apb_item apb_write;
176  `uvm_component_utils_begin(apb_monitor)
   `uvm_field_int(checks_enable, UVM_ALL_ON)
178   `uvm_field_int(coverage_enable, UVM_ALL_ON)
  `uvm_component_utils_end
180  covergroup cov_apb_write @apb_written_event;
   option.per_instance =1;
182   pwaddr : coverpoint apb_write.pwaddr{option.auto_bin_max=8;}
   pwdata : coverpoint apb_write.pwdata{option.auto_bin_max=8;}
```

```
184 endgroup : cov_apb_write

186 // Constructor
function new(string name, uvm_component parent);
188  super.new(name, parent);
 cov_apb_write = new();
190  cov_apb_write.set_inst_name({get_full_name(), ".cov_apb_write"
      });
 apb_write = new();
192  item_collected_port = new("item_collected_port", this);
endfunction : new
194
function void build_phase(uvm_phase phase);
196  super.build_phase(phase);
 if(!uvm_config_db#(virtual apb_if)::get(this, "", "apb_vif",
     apb_vif))
198   `uvm_fatal("NOVIF",{"virtual interface must be set for: ",
   get_full_name(),".apb_vif"});
200 endfunction : build_phase

202 virtual task run_phase(uvm_phase phase);
 fork
204   collect_transactions(); // Spawn collector task
 join
206 endtask : run_phase

208 virtual protected task collect_transactions(); // TODO: Why
    protected?
 @(posedge apb_vif.resetn)
210  forever begin
  @(posedge apb_vif.clk iff((apb_vif.pwrite==1'b1) && (apb_vif.
      psel==1'b1)));
212    begin
    // Collect data from the bus into apb_write.
214    apb_write.pwaddr = apb_vif.pwaddr;
    apb_write.pwdata = apb_vif.pwdata;
216    `uvm_info(get_type_name(), $sformatf("Transfer collected:\\n%
       s",
    apb_write.sprint()), UVM_FULL)
218    if (checks_enable)
     perform_transfer_checks();
220    if (coverage_enable)
     perform_transfer_coverage();
222    item_collected_port.write(apb_write);
    end
224  end
endtask : collect_transactions
226
virtual protected function void perform_transfer_coverage();
```

```
228     // Signal coverage event, this samples the coverpoints.
    -> apb_written_event;
230   endfunction : perform_transfer_coverage;

232   virtual protected function perform_transfer_checks();
      // Perform data checks on trans_collected here...
234   endfunction : perform_transfer_checks

236   virtual function void report_phase(uvm_phase phase);
      `uvm_info(get_full_name(),$sformatf("Covergroup 'cov_apb_write'
          coverage:%2f",
238   cov_apb_write.get_inst_coverage()), UVM_LOW)
    endfunction : report_phase
240 endclass : apb_monitor

242 // APB AGENT
    class apb_agent extends uvm_agent;
244   // Agent components
    uvm_active_passive_enum is_active = UVM_ACTIVE;
246   apb_sequencer sequencer;
    apb_driver driver;
248   apb_monitor monitor;
    virtual apb_if apb_vif;
250
    // Constructor and UVM macros here
252   `uvm_component_utils_begin(apb_agent)
      `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_DEFAULT
          )
254   `uvm_field_object(sequencer, UVM_ALL_ON)
      `uvm_field_object(driver, UVM_ALL_ON)
256   `uvm_field_object(monitor, UVM_ALL_ON)
    `uvm_component_utils_end
258   function new(string name, uvm_component parent);
      super.new(name,parent);
260   endfunction : new

262   // Create subcomponents using factory
    virtual function void build_phase(uvm_phase phase);
264   super.build_phase(phase);
      monitor = apb_monitor::type_id::create("monitor",this);
266   if (is_active == UVM_ACTIVE) begin
        // Build the sequencer and driver.
268     sequencer = apb_sequencer::type_id::create("sequencer",this);
      driver = apb_driver::type_id::create("driver", this);
270   end
      if(!uvm_config_db#(virtual apb_if)::get(this, "", "apb_vif",
          apb_vif)) begin
272     `uvm_fatal("APB/AGT/NOVIF", "No virtual interface specified
          for agent")
```

```
    end
274  endfunction : build_phase

276  // Connect driver and sequencer
     virtual function void connect_phase(uvm_phase phase);
278   if(is_active == UVM_ACTIVE) begin
       driver.seq_item_port.connect(sequencer.seq_item_export);
280   end
     endfunction : connect_phase
282 endclass : apb_agent

284 `endif
```

../source/tbench/apb_comp_lib.sv

```
     // Apb interface test classes
  2  `ifndef AXI_COMP_LIB_SVH
     `define AXI_COMP_LIB_SVH
  4
     // AXI INTERFACE
  6  interface axi_if; // APB protocol signals
      logic clk, resetn;
  8   logic [4:0] arid, rid;
      logic [3:0] arlen;
 10   logic [2:0] arsize;
      logic [1:0] arburst;
 12   logic arvalid, rready, arready, rvalid, rlast;
      logic [31:0] araddr;
 14   logic [127:0] rdata;
     endinterface : axi_if
 16
     // APB DATA ITEM
 18  class axi_item extends uvm_sequence_item;
      rand logic [31:0] araddr;
 20   rand logic [4:0] arid;
       // Both bursts are stored in rdata.
 22   //255-128 is the first burst
      logic [255:0] rdata;
 24
      logic [4:0]   rid;
 26
      // UVM automation macros
 28   `uvm_object_utils_begin(axi_item)
       `uvm_field_int(araddr, UVM_DEFAULT)
 30    `uvm_field_int(arid, UVM_DEFAULT)
       `uvm_field_int(rdata, UVM_DEFAULT)
 32    `uvm_field_int(rid, UVM_DEFAULT)
      `uvm_object_utils_end
 34
```

```
  //   Constructor
36  function new (string name = "axi_item");
     super.new(name);
38  endfunction : new
  endclass : axi_item
40
  // APB SEQUENCE AND SEQUENCER
42 class axi_sequencer extends uvm_sequencer #(axi_item);
   `uvm_component_utils(axi_sequencer)
44  function new(string name, uvm_component parent);
     super.new(name, parent);
46   $display("Axi sequencer");
  endfunction : new
48 endclass : axi_sequencer

50 class axi_sequence extends uvm_sequence #(axi_item);
   function new(string name="axi_sequence");
52   super.new(name);
     $display("AXI sequence");
54  endfunction

56  `uvm_object_utils(axi_sequence)

58  virtual task body();
     uvm_test_done.raise_objection(this, "AXI sequence");
60   `uvm_info("axi_sequence", "Starting sequence", UVM_MEDIUM)
     `uvm_create(req)
62   // Perform an AXI read to address 0000_0000.
     start_item(req);
64   $display("axi_item started");
     req.randomize() with { araddr==32'h0000_4000;};
66   finish_item(req);
     uvm_test_done.drop_objection(this, "AXI sequence");
68  endtask
  endclass : axi_sequence
70

72 //APB DRIVER
  class axi_driver extends uvm_driver #(axi_item);
74  virtual axi_if axi_vif;
   // UVM automation macros
76  `uvm_component_utils(axi_driver)

78  // Constructor
   function new (string name = "axi_driver", uvm_component parent=
       null);
80   super.new(name,parent);
   endfunction : new
82
```

```
  // Build phase registers the vif resource
84 function void build_phase(uvm_phase phase);
   string inst_name;
86 super.build_phase(phase);
   if(!uvm_config_db#(virtual axi_if)::get(this,"","axi_vif",
       axi_vif))
88 begin
    `uvm_fatal("NOVIF",{"virtual interface must be set for: ",
        get_full_name(),".axi_vif"});
90 end
  endfunction : build_phase
92
  // run_phase retrieves data_items and drives them
94 task run_phase(uvm_phase phase);
   axi_item a_item;
96 super.run_phase(phase);
   @(posedge axi_vif.resetn);
98 forever begin
    seq_item_port.get_next_item(a_item);
100  drive_item(a_item);
    seq_item_port.item_done();
102 end
  endtask : run_phase
104
  // drive_item is the logic required to push the item onto the
      axi
106 // In this case it is the APB protocol signals and timing.
  task drive_item (input axi_item item);
108   // Wait for the fractal generator to be ready.
    // Then perform an AXI read.
110   $display("Driving AXI item");
    forever begin
112   @(posedge axi_vif.clk iff axi_vif.arready==1);
     axi_vif.arvalid <= 1'b1;
114    axi_vif.arlen <= 4'b0001;
     axi_vif.arburst <= 2'b01;
116    axi_vif.arsize <= 3'b001;

118    axi_vif.araddr <= item.araddr;
     axi_vif.arid <= item.arid;
120    $display("AXI read requested");
     break;
122   end
    @ (posedge axi_vif.clk);
124    @ (posedge axi_vif.clk)
     axi_vif.arvalid <= 1'b0;
126    axi_vif.rready <= 1'b1; // Ready to accept data.

128   // Wait for transmission from the fractal generator
```

```
    @(posedge axi_vif.rvalid);
130     $display("Receiving AXI-data.");
        item.rdata[127:0]<= axi_vif.rdata;
132     item.rid <= axi_vif.rid;
    @(posedge axi_vif.clk);
134     item.rdata[255:128] <= axi_vif.rdata;
  endtask : drive_item
136 endclass : axi_driver

138 // APB MONITOR
    class axi_monitor extends uvm_monitor;
140   virtual axi_if axi_vif;
    bit checks_enable = 1;
142   bit coverage_enable = 1;

144  uvm_analysis_port#(axi_item) item_collected_port;
    event axi_read_event; // Events needed to trigger covergroups
146  protected axi_item axi_read;
    `uvm_component_utils_begin(axi_monitor)
148   `uvm_field_int(checks_enable, UVM_ALL_ON)
    `uvm_field_int(coverage_enable, UVM_ALL_ON)
150  `uvm_component_utils_end
    covergroup cov_axi_read @axi_read_event;
152   option.per_instance =1;
    rdata : coverpoint axi_read.rdata{option.auto_bin_max=8;}
154   rid : coverpoint axi_read.rid{option.auto_bin_max=8;}
    endgroup : cov_axi_read
156
    // Constructor
158  function new(string name, uvm_component parent);
    super.new(name, parent);
160   cov_axi_read = new();
    cov_axi_read.set_inst_name({get_full_name(), ".cov_axi_read"});
162   axi_read = new();
    item_collected_port = new("item_collected_port", this);
164  endfunction : new

166  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
168   if(!uvm_config_db#(virtual axi_if)::get(this, "", "axi_vif",
        axi_vif))
     `uvm_fatal("NOVIF",{"virtual interface must be set for: ",
170    get_full_name(),".axi_vif"});
  endfunction : build_phase
172
    virtual task run_phase(uvm_phase phase);
174   phase.raise_objection(this);
    fork
176    collect_transactions(); // Spawn collector task
```

```
    join
178   phase.drop_objection(this);
    endtask : run_phase
180
    virtual protected task collect_transactions(); // TODO: Why
        protected?
182   @(posedge axi_vif.clk iff(axi_vif.rvalid==1'b1));
      // Collect data from the bus into axi_read.
184   $display("Collecting axi data");
      axi_read.araddr =axi_vif.araddr;
186   axi_read.rdata[127:0] = axi_vif.rdata;
      axi_read.rid = axi_vif.rid;
188   @(posedge axi_vif.clk);
      axi_read.rdata[255:128] = axi_vif.rdata;
190    $display("Collecting axi data2");
      `uvm_info(get_type_name(), $sformatf("Transfer collected:\\n%s
          ",
192   axi_read.sprint()), UVM_FULL)
      if (checks_enable) begin
194    perform_transfer_checks();
      end
196   if (coverage_enable) begin
       perform_transfer_coverage();
198    $display("writing axi data");
       item_collected_port.write(axi_read);
200   end
    endtask : collect_transactions
202
    virtual protected function void perform_transfer_coverage();
204   // Signal coverage event, this samples the coverpoints.
     -> axi_read_event;
206 endfunction : perform_transfer_coverage;

208 virtual protected function perform_transfer_checks();
     // Perform data checks on trans_collected here...
210 endfunction : perform_transfer_checks

212 virtual function void report_phase(uvm_phase phase);
     `uvm_info(get_full_name(),$sformatf("Covergroup 'cov_axi_read'
         coverage:%2f",
214  cov_axi_read.get_inst_coverage()), UVM_LOW)
    endfunction : report_phase
216 endclass : axi_monitor

218 // APB AGENT
    class axi_agent extends uvm_agent;
220  // Agent components
     uvm_active_passive_enum is_active = UVM_ACTIVE;
222  axi_sequencer sequencer;
```

```
  axi_driver driver;
224  axi_monitor monitor;
  virtual axi_if axi_vif;
226
  // Constructor and UVM macros here
228  `uvm_component_utils_begin(axi_agent)
   `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_DEFAULT
       )
230   `uvm_field_object(sequencer, UVM_ALL_ON)
   `uvm_field_object(driver, UVM_ALL_ON)
232   `uvm_field_object(monitor, UVM_ALL_ON)
  `uvm_component_utils_end
234  function new(string name, uvm_component parent);
   super.new(name,parent);
236  endfunction : new

238  // Create subcomponents using factory
  virtual function void build_phase(uvm_phase phase);
240   super.build_phase(phase);
   monitor = axi_monitor::type_id::create("monitor",this);
242   if (is_active == UVM_ACTIVE) begin
    // Build the sequencer and driver.
244    sequencer = axi_sequencer::type_id::create("sequencer",this);
    driver = axi_driver::type_id::create("driver", this);
246   end
   if(!uvm_config_db#(virtual axi_if)::get(this, "", "axi_vif",
       axi_vif)) begin
248    `uvm_fatal("APB/AGT/NOVIF", "No virtual interface specified
       for agent")
   end
250  endfunction : build_phase

252  // Connect driver and sequencer
  virtual function void connect_phase(uvm_phase phase);
254   if(is_active == UVM_ACTIVE) begin
    driver.seq_item_port.connect(sequencer.seq_item_export);
256   end
  endfunction : connect_phase
258 endclass : axi_agent


260
 `endif /* AXI_COMP_LIB_SVH */
```

../source/tbench/axi_comp_lib.sv

```
  // Fractal verification top level module
2
 `include "uvm_pkg.sv"
4 import uvm_pkg::*;
```

```
`include "apb_comp_lib.sv"
`include "axi_comp_lib.sv"
`include "fractal_testlib.sv"
`include "fractal_generator_main.v"

module fractal_tb_top;

 apb_if apb_vif(); // Interface to the component
 axi_if axi_vif();

 fractal_generator_main fractal_generator(
  .CLK(apb_vif.clk), .RESET_N(apb_vif.resetn),
  // APB
  .PSEL(apb_vif.psel), .PWRITE(apb_vif.pwrite),
  .PREADY(apb_vif.pready), .PENABLE(apb_vif.penable),
  .PWDATA(apb_vif.pwdata), .PWADDR(apb_vif.pwaddr),
  .PRDATA(apb_vif.prdata),
  // AXI
  .ARID(axi_vif.arid), .ARADDR(axi_vif.araddr), .ARLEN(axi_vif.
      arlen),
  .ARSIZE(axi_vif.arsize), .ARBURST(axi_vif.arburst),
  .ARVALID(axi_vif.arvalid), .ARREADY(axi_vif.arready),
  .RREADY(axi_vif.rready),
  .RID(axi_vif.rid), .RDATA(axi_vif.rdata),
  .RVALID(axi_vif.rvalid), .RLAST(axi_vif.rlast),
  .DEBUG()
 );
 /*
 apb_interface apb(
  .CLK(apb_vif.clk), .RESET_N(apb_vif.resetn),
  .PSEL(apb_vif.psel), .PWRITE(apb_vif.pwrite), .PREADY(apb_vif.
      pready), .PENABLE(apb_vif
  .penable),
  .PWDATA(apb_vif.pwdata), .PWADDR(apb_vif.pwaddr),
  .x_fgc(apb_vif.x_fgc), .y_fgc(apb_vif.y_fgc), .stepsize_fgc(
      apb_vif.stepsize_fgc)
 );
  */

  assign axi_if.clk = apb_if.clk;
  assign axi_if.resetn = apb_if.resetn;

 initial begin
  uvm_config_db#(virtual apb_if)::set(uvm_root::get(), "*", "
      apb_vif", apb_vif);
  uvm_config_db#(virtual axi_if)::set(uvm_root::get(), "*", "
      axi_vif", axi_vif);
  run_test();
 end
```

```
50  initial begin
      $vcdpluson;
52    $vcdplusmemon;
      apb_vif.resetn <= 1'b0;
54    apb_vif.clk <= 1'b1;
      #51 apb_vif.resetn <= 1'b1;
56  end

58  always begin
      #5 apb_vif.clk <= ~apb_vif.clk;
60  end

62 endmodule : fractal_tb_top
```

../source/tbench/fractal_tb_top.sv

```
    //Fractal_testlib constructs the fractal verification env and
       tests
2


4  // VIRTUAL SEQUENCE
   import "DPI" function int unsigned calculate_iterations(input
       shortreal re, input shortreal im);
6
   class fractal_virtual_sequence extends uvm_sequence;
8   `uvm_object_utils(fractal_virtual_sequence)

10  function new(string name="fractal_virtual_sequence");
      super.new(name);
12    $display("Fractal sequence");
    endfunction : new
14


16  apb_sequencer apb_seqr;
    axi_sequencer axi_seqr;
18
    apb_sequence apb_seq;
20  axi_sequence axi_seq;

22  virtual task body();
      apb_seq = apb_sequence::type_id::create("apb_seq");
24    axi_seq = axi_sequence::type_id::create("axi_seq");
      apb_seq.start(apb_seqr, this);
26    axi_seq.start(axi_seqr, this);
    endtask :body
28 endclass : fractal_virtual_sequence


30
```

```
class fractal_scoreboard extends uvm_scoreboard;
`uvm_component_utils(fractal_scoreboard)
int sbd_error = 0;
`uvm_analysis_imp_decl(_axi)
`uvm_analysis_imp_decl(_apb)
uvm_analysis_imp_apb #(apb_item, fractal_scoreboard) apb_export;
uvm_analysis_imp_axi #(axi_item, fractal_scoreboard) axi_export;
shortreal x0 = null;
shortreal y0 = null;
shortreal stepsize = null;

protected bit disable_scoreboard = 0;

function new(string name, uvm_component parent);
 super.new(name, parent);
endfunction : new

function void build_phase(uvm_phase phase);
 apb_export = new("apb_export", this);
 axi_export = new("axi_export", this);
endfunction : build_phase

virtual function void write_apb(apb_item apb);
 if(!disable_scoreboard)
 begin
  $display("apb_write");
  if (apb.pwaddr == 32'h000_C000) begin
   this.x0=$bitstoshortreal(apb.pwdata);
  end
  else if (apb.pwaddr == 32'h000_C004) begin
   this.y0=$bitstoshortreal(apb.pwdata);
  end
  else if (apb.pwaddr == 32'h000_C008) begin
   this.stepsize=$bitstoshortreal(apb.pwdata);
  end
 end
endfunction :write_apb

virtual function void write_axi(axi_item axi);
int row = 0;
int column = 0;
int unsigned lol =0;
int unsigned iteration = 0;
int unsigned temp[8];
int start,stop;
shortreal x,y;
 if(!disable_scoreboard);
  $display("axi_write with araddr = %h",axi.araddr[31:0]);
  //rdata1 is the 4 first coordinates
```

```
80      row = (axi.araddr/4)/128;
        column = (axi.araddr/4)%128;
82      x=this.x0+(this.stepsize*column);
        y=this.y0-(this.stepsize*row);
84      $display("x,y=%f,%f",x,y);
        // Assumes requests doesnt cross rows.
86      for(int i=0;i<8;i++) begin
         iteration = calculate_iterations(x+(i*this.stepsize),y);
88       // Assert each of the coordinates in the transfer
         temp[0]=fp32toint(axi.rdata[31+(0*32):0*32]);
90       temp[1]=fp32toint(axi.rdata[31+(1*32):1*32]);
         temp[2]=fp32toint(axi.rdata[31+(2*32):2*32]);
92       temp[3]=fp32toint(axi.rdata[31+(3*32):3*32]);
         temp[4]=fp32toint(axi.rdata[31+(4*32):4*32]);
94       temp[5]=fp32toint(axi.rdata[31+(5*32):5*32]);
         temp[6]=fp32toint(axi.rdata[31+(6*32):6*32]);
96       temp[7]=fp32toint(axi.rdata[31+(7*32):7*32]);
         // There are some rounding issues, so I give a bit of
98       // leeway.
         assert(temp[i]-1 <= iteration <= temp[i]+1) begin
100       $display("ASSERT PASSED");
          $display("iteration=%d, temp=%d",iteration,temp[i]);
102       $display("rdata = %h", axi.rdata);
         end
104      else begin
          $error("iteration=%d, temp=%d",iteration,temp[i]);
106       $display("rdata = %h", axi.rdata);
         end
108     end
    endfunction :write_axi
110
    // Used to convert fp32 iterations to integers.
112 //Fp32 iterations are always integers and need no rounding.
    function int fp32toint(int lol);
114  int temp=0;
     if ((lol[31] != 1'b0) || (lol[21:0] != 0)) begin
116   $display("Decimal value in iteration lol=%h",lol);
     end
118
     // Raw exponent
120  temp[7:0]=lol[30:23];
     temp -= 127;
122  temp = 2**temp;
     if(lol[22]==1)
124   temp+=1;
     return temp;
126 endfunction: fp32toint

128 endclass : fractal_scoreboard
```

```
130
// APB TESTBENCH
132 class fractal_tb extends uvm_env;
 `uvm_component_utils(fractal_tb)
134  virtual apb_if apb_vif;
  virtual axi_if axi_vif;
136
 apb_agent apb0;
138  axi_agent axi0;

140  fractal_scoreboard fractal_scoreboard0;
  // Constructor
142  function new(string name, uvm_component parent);
   super.new(name, parent);
144  endfunction : new

146  virtual function void build_phase(uvm_phase phase);
   super.build_phase(phase);
148   uvm_config_db#(int)::set(this,"*.apb0.sequencer", "count", 0);
   uvm_config_db#(int)::set(this,"axi0.sequencer", "count", 0);
150   apb0 = apb_agent::type_id::create("apb0", this);
   if(!uvm_config_db#(virtual apb_if)::get(this,"","apb_vif",
       apb_vif))
152   begin
    `uvm_fatal("NOVIF",{"virtual interface must be set for: ",
154              get_full_name(),".apb_vif"});
   end
156
   axi0 = axi_agent::type_id::create("axi0", this);
158   if(!uvm_config_db#(virtual axi_if)::get(this,"","axi_vif",
       axi_vif))
   begin
160    `uvm_fatal("NOVIF",{"virtual interface must be set for: ",
              get_full_name(),".axi_vif"});
162   end


164


166   fractal_scoreboard0 = fractal_scoreboard::type_id::create("
       fractal_scoreboard0", this);
  endfunction : build_phase
168
  virtual function void connect_phase(uvm_phase phase);
170   // Connect monitors to scoreboard.
   apb0.monitor.item_collected_port.connect(fractal_scoreboard0.
       apb_export);
172   axi0.monitor.item_collected_port.connect(fractal_scoreboard0.
       axi_export);
```

```
  endfunction : connect_phase
endclass : fractal_tb

class fractal_base_test extends uvm_test;
  `uvm_component_utils(fractal_base_test)
  fractal_tb fractal_tb0;
  uvm_table_printer printer;
  bit test_pass = 1;

 // Constructor
 function new(string name ="fractal_base_test", uvm_component
     parent = null);
  super.new(name,parent);
 endfunction : new

 // Build phase
 virtual function void build_phase(uvm_phase phase);
  // Configure default sequence in the sequencer
  super.build_phase(phase);

  fractal_tb0 = fractal_tb::type_id::create("fractal_tb0", this);

  // Enable transaction recording for everything

    uvm_config_db#(int)::set(this, "*", "recording_detail",
       UVM_FULL);

  // Create test bench and printer
  printer = new();
  printer.knobs.depth = 3;
 endfunction : build_phase

 task run_phase(uvm_phase phase);
  // Build virtual sequence
  fractal_virtual_sequence virtual_sequence;
  virtual_sequence = fractal_virtual_sequence::type_id::create("
      virtual_sequence");
  phase.raise_objection(this);


  // Connect sequence to sequencers
  // Connect virtual sequencer to sequencers
  virtual_sequence.apb_seqr = fractal_tb0.apb0.sequencer;
  virtual_sequence.axi_seqr = fractal_tb0.axi0.sequencer;
  virtual_sequence.grab(fractal_tb0.apb0.sequencer);

  // Start sequence
  virtual_sequence.start(null);

```

```
    phase.drop_objection(this);
220  endtask : run_phase

222

  function void extract_phase(uvm_phase phase);
224     if(fractal_tb0.fractal_scoreboard0.sbd_error)
          test_pass = 1'b0;
226   endfunction : extract_phase

228  function void report_phase(uvm_phase phase);
        if(test_pass) begin
230       `uvm_info(get_type_name(), "** UVM TEST PASSED **",
              UVM_NONE)
        end
232     else begin
          `uvm_error(get_type_name(), "** UVM TEST FAIL **")
234     end
    endfunction : report_phase
236
endclass : fractal_base_test
```

../source/tbench/fractal_testlib.sv

C Source Code for the Fractal Demo

```
// Generation of fractal figures here

#include "../include/fractals.h"


//#define SKYPOINTS 64
//#define DEBUG


// Arrays for 2D fractal(sky)
/*GLfloat skyVertices[(SKYPOINTS*SKYPOINTS*4)];
GLfloat skyIterations[(SKYPOINTS*SKYPOINTS)];
GLushort skyIndices[((SKYPOINTS-1)*(2*SKYPOINTS))+ (2*(SKYPOINTS-2))];
*/

#ifndef FRACTAL_GENERATOR
void zoomAtPoint(struct fractal_landscape * landscape)
{
 landscape->fractal_configuration[0] = landscape->fractal_configuration[0] - ((NUMPOINTS/2)*landscape->fractal_configuration[2]);
 landscape->fractal_configuration[1] = landscape->fractal_configuration[1] + ((NUMPOINTS/2)*landscape->fractal_configuration[2]);
 updateIterations(landscape);
}
#endif /* ! FRACTAL_GENERATOR */

// getFractPoint is copied from Corneliussens master thesis.
int getFractPoint(float re, float im)
// Returns the number of iterations before |z| exceeds 2.
// If the number is 80 the point is not in the Mandelbrot set.
{
 int n;
 float z_re = 0.0f, z_im = 0.0f;
 for(n=1;n<80;n++)
 {
  float z_re_old = z_re;
  z_re = z_re*z_re - z_im*z_im + re;
  z_im = 2.0f * z_re_old * z_im + im;

  // Simplified boundary check
  if (( z_re >= 2.0f) || (z_re <= -2.0f) || (z_im >= 2.0f) || (z_im <= -2.0f) )
   break;
```

```
    // Precise booundary check.

    //if (_hypotf(z_re, z_im) >= 2.0f)
      //break;
  }
  return n;
}


#ifndef FRACTAL_GENERATOR
void getFractalHeights(float x, float y, float stepsize, int
    numpoints, GLfloat *iterations)
// Calculates the height/iterations of all the vertices in the
    fractal specified
// by x,y,stepsize and numpoints. The fractal is assumed to be
    square = numpoints^2.
// Assumes an array with space for numpoints^2 integers.
{
  int i,j;
  for(i=0; i<numpoints;i++) //y-axis = imaginary
  {
    for(j=0;j<numpoints;j++) //x-axis
    {
      iterations[j+(i*numpoints)] = getFractPoint(x+(j*stepsize),y-(
          i*stepsize));
    }
  }
}
#endif /* ! FRACTAL_GENERATOR */

void printFractalHeights(GLfloat *iterations, int numpoints)
{
  int i,j;
  for(i=0; i<numpoints;i++) //y-axis = imaginary
  {
    for(j=0;j<numpoints;j++) //x-axis
    {
      printf("%2.0f ",iterations[j+(i*numpoints)]);
    }
    printf("\n");
  }
}
void printFractalVertices(float* vertices, int numpoints)
{
  int i;
  for(i=0; i<(numpoints*numpoints)*4;i++) //y-axis = imaginary
  {
    if( (i % (numpoints*numpoints) == 0) && (i != 0) )
      printf("\n");
```

```
86    else if( (i % numpoints == 0) && (i != 0) )
       printf("  ");
88    printf("%1.1f,",vertices[i]);
    }
90  printf("\n");
    printf("\n");
92
}
94  void getFractalVertices(GLfloat x, GLfloat y, GLfloat stepsize,
       GLfloat* vertices)
    /* Creates a fractal square with numpoints*numpoints points.
96   * Reaching from x -> x+stepsize*numpoints, same with y.
     * Note that both x and y can be negative.
98   * All y-coordinates are == 0.0f
    */
100 {
     int i = 0; // Counts the index in vertices.
102  int k; // Loop counters.
     int j;
104  // Starts with the top left coordinate.
     for (j=0;j<NUMPOINTS;j++)
106  {
      for (k=0;k<NUMPOINTS;k++)
108   {
       vertices[i] = x + (k*stepsize); // x
110    vertices[i+1]= 0.0f; // y
       vertices[i+2]= y - (j*stepsize); // z
112    vertices[i+3]= 1.0f; // w
       i = i+4;
114   }
     }
116 }


118
    void getFractalIndices(GLushort* indices, int numpoints)
120 // Takes in a pointer to a square of vertices and generates an
       array of indices to draw
    // the square.
122 {
     int i;
124  int j = 0;
     int x,y;
126  // For each row of the vertices
     i = 0;
128  y = 0;
     while(y < (numpoints*numpoints) )
130  {
      x = i*numpoints;
132   y = x+numpoints;
```

```
    while(x < ((i*numpoints)+(numpoints-1)))
    {
     // Draws two triangles to create a little square for each
        column of x.
     indices[j] = x;
     indices[j+1] = x+1;
     indices[j+2] = y;
     indices[j+3] = x+1;
     indices[j+4] = y;
     indices[j+5] = y+1;
     y++;
     x++;
     j=j+6;
    }
    i++;
  }
}

void getFractalIndicesTriangleStrip(struct fractal_landscape *
    landscape)
{
  // Uses degenerate triangles to create one big triangle strip.
  int i = 0;
  int j = 0;
  GLushort x,y;
  // For each row of the vertices
  y=NUMPOINTS, x = 0;
  while(y < (NUMPOINTS*NUMPOINTS) )
  {
    while ( (x<((i*NUMPOINTS)+(NUMPOINTS))) )
    {
     landscape->indices[j] = x;
     landscape->indices[j+1]=y;
     j=j+2;
     x++;
     y++;
    }
    // Add degenerate triangles
    landscape->indices[j] = y-1;
    landscape->indices[j+1] = x;
    j=j+2;
    i++;
  }
}

#ifndef FRACTAL_GENERATOR
void updateIterations(struct fractal_landscape* landscape)
{ // Used for zooming in on the landscape by updating the
    iterations.
```

```
 getFractalHeights(landscape->fractal_configuration[0],landscape
    ->fractal_configuration[1],
180         landscape->fractal_configuration[2],NUMPOINTS,
                landscape->iterations);
 glBindBuffer(GL_ARRAY_BUFFER,landscape->vboIds[1]);
182 GL_CHECK(glBufferData(GL_ARRAY_BUFFER, sizeof(landscape->
    iterations),
    landscape->iterations, GL_STATIC_DRAW));
184 }
#endif /* ! FRACTAL_GENERATOR */
186
void createFractalLandscape(struct fractal_landscape* landscape)
188 {
#ifdef DEBUG
190  int debug;
#endif // DEBUG
192  // Get attribute locations of non-fixed attributes like colour
    and texture coordinates from the shader

194  positionLoc = GL_CHECK(glGetAttribLocation(landscape->shader, "
    av4position"));
  afheightLoc = GL_CHECK(glGetAttribLocation(landscape->shader, "
    afheight"));
196  // Set number of dimensions to three.
  // This is done to use the same shader when drawing the 3D
    landscape and the sky.
198  // TODO: Remove this and add extra shader?
  dimensionLoc = GL_CHECK(glGetUniformLocation(landscape->shader,
    "dimension"));
200  glUniform1f(dimensionLoc,3); // Set number of dimensions to
    three.
#ifndef FRACTAL_GENERATOR
202  getFractalHeights(landscape->fractal_configuration[0], landscape
    ->fractal_configuration[1],
         landscape->fractal_configuration[2] ,NUMPOINTS,
             landscape->iterations);
204 #endif /* ! FRACTAL_GENERATOR */
  // VBO initialization for fractal landscape here
206  landscape->vboIds = calloc(3,sizeof(GLuint));
 GL_CHECK(glGenBuffers(3, landscape->vboIds)); // One coordinate
    vertices, one indices, one heights.
208  // Vertex 2-D coordinates
 GL_CHECK(glBindBuffer(GL_ARRAY_BUFFER, landscape->vboIds[0]));
    // Vertex data in vboIds[0]
210 GL_CHECK(glBufferData(GL_ARRAY_BUFFER, sizeof(landscape->
    vertices),
    landscape->vertices, GL_STATIC_DRAW));
212
```

```
    // Vertex Height coordinates, Y coordinate. Merged with 2D in
        shader.
214 GL_CHECK(glBindBuffer(GL_ARRAY_BUFFER, landscape->vboIds[1]));
        //
    GL_CHECK(glBufferData(GL_ARRAY_BUFFER, sizeof(landscape->
        iterations),
216     landscape->iterations, GL_STATIC_DRAW));

218 GL_CHECK(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,landscape->vboIds
        [2]));
    GL_CHECK(glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(landscape
        ->indices),
220 landscape->indices, GL_STATIC_DRAW));

222 #ifdef DEBUG
    // for(debug=0;debug<(((NUMPOINTS-1)*(2*NUMPOINTS))+ (2*(
        NUMPOINTS-2)));debug++)
224 //   fprintf(stderr, "landscapeIndices[%i] = %i\n",debug,
        landscape->indices[debug]);
    for(debug=0;debug<(4*NUMPOINTS*NUMPOINTS);debug=debug+1)
226   fprintf(stderr, "landscapeVertices[%i] = %f\n",debug, landscape
        ->vertices[debug]);

228 //for(debug=0;debug<NUMPOINTS*NUMPOINTS;debug++)
    // fprintf(stderr, "iterations[%i] = %i\n",debug, landscape->
        iterations[debug]);
230
    printf("IndexVBOid = %i\n",landscape->vboIds[2]);
232 fprintf(stderr, "sizeof landscapeIterations = %i\n",sizeof(
        landscape->iterations));
    fprintf(stderr, "sizeof landscapeIndices = %i\n",sizeof(
        landscape->indices));
234 fprintf(stderr, "sizeof landscapeVertices = %i\n",sizeof(
        landscape->vertices));

236 fprintf(stderr, "sizeof GLfloat = %i\n",sizeof(GLfloat));
    #endif /* DEBUG */
238 }
```

../source/demo/fractals.c

```
1
    #include "../include/main.h"
3   #include "../include/shader.h"
    #include "../include/matrix.h"
5   #include "../include/fractals.h"
    #include "../include/platform.h"
7
    #include "GLES2/gl2.h"
```

```
9  #include "EGL/egl.h"

11 #ifdef PROFILING
   #include "sys/time.h"
13 #endif /* PROFILING */

15 /* Global variables */
   #ifdef _WIN32
17 HWND hWindow;
   HDC  hDisplay;
19 #endif /* _WIN32 */
   GLint iLocPosition = 0;
21 GLint iLocColour, iLocMVP;
   GLuint uiProgram, uiFragShader, uiVertShader;
23
   struct fractal_landscape *landscapes[(TOTAL_POINTS / NUMPOINTS)*(
       TOTAL_POINTS / NUMPOINTS)]; // Array of pointers to
       landscapes
25 EGLDisplay sEGLDisplay;
   EGLContext sEGLContext;
27 EGLSurface sEGLSurface;
   const unsigned int uiWidth  = 1;
29 const unsigned int uiHeight = 1;
   GLushort num_of_landscapes;
31 int RENDER_STATE = 5;
   // Camera movement etc
33 float Identity[16] = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 1 };
   float yRotateAngle = 0.0f;
35 float xRotateAngle = 65.f;
   // Set initial camera position
37 float zTranslation = -3.5f; // Move camera a little back
       initially.
   float yTranslation = 0.0f;
39 float xTranslation = 0.0f;

41 // Create landscapes combines several NUMPOINTS^2  landscapes
       into one large TOTALPOINTS^2 landscape
   void create_landscape(struct fractal_landscape * landscapes[],
       GLushort * num_of_landscapes,
43         GLfloat x, GLfloat y, GLfloat stepsize)
   {
45  short i,j,temp;
    // Stepsize between the static vertex coordinates.
47  float vertex_stepsize = (2.0f/ (TOTAL_POINTS -1.0f));
    *num_of_landscapes = (TOTAL_POINTS / NUMPOINTS);
49  // Landscapes is an array of pointers to landscape structs
    //landscapes = calloc(*num_of_landscapes*(*num_of_landscapes),
        sizeof(struct fractal_landscape *));
```

```
51  for(i=0;i<(*num_of_landscapes);i++)
    {
53   for(j=0;j<(*num_of_landscapes);j++)
     {
55    temp = (i*(*num_of_landscapes)) + j;
      landscapes[temp]= malloc(sizeof(struct fractal_landscape));
57    landscapes[temp]->fractal_configuration[0] = x + (j*NUMPOINTS*
          stepsize);
      printf("landscapes[temp]->fractal_configuration[0]= %f\n",
          landscapes[temp]->fractal_configuration[0]);
59    landscapes[temp]->shader = uiProgram;
      landscapes[temp]->fractal_configuration[2]= stepsize;
61    landscapes[temp]->fractal_configuration[1] = y - (i*NUMPOINTS*
          stepsize);
      getFractalIndicesTriangleStrip(landscapes[temp]);
63    getFractalVertices( (-1.0f+(j*(NUMPOINTS-1)*vertex_stepsize)),
                (1.0f-(i*(NUMPOINTS-1)*vertex_stepsize)),
65              vertex_stepsize, landscapes[temp]->vertices
              );
67    printf("loop lol\n");
     }
69  }
}
71

73// Fractal point functions
/*
75 * initial_rotation rotates around the y - axis in positive or
     negative direction until angle
*/
77
void initial_rotation(struct fractal_point* fp, float*
    rotation_matrix)
79{
 yRotateAngle += 0.1f*fp->r_speed;
81 rotate_matrix(yRotateAngle,0.0f,1.0f,0.0f,rotation_matrix);
 if(yRotateAngle >= fp->initial_angle) // Rotation is finished
83  RENDER_STATE = 1;
}
85void navigate_to_point(struct fractal_point* fp, float* matrix)
{
87 matrix[14] += 0.1f*fp->n_speed;
 if(matrix[14] <= fp->distance)
89  RENDER_STATE = 2;
}
91/* zoom_point
 * This is currently hardcoded to use landscapes[0], to enable
     zoom with a
93 * higher resolution than 128x128, this needs to be made general.
```

```
*/
void zoom_point(struct fractal_point* fp)
{
 landscapes[0]->fractal_configuration[0] = fp->x - (NUMPOINTS/2)*
     landscapes[0]->fractal_configuration[2];
 landscapes[0]->fractal_configuration[1] = fp->y + (NUMPOINTS/2)*
     landscapes[0]->fractal_configuration[2];
 landscapes[0]->fractal_configuration[2] = fp->z_speed*0.95f*
     landscapes[0]->fractal_configuration[2];
#ifndef FRACTAL_GENERATOR
 updateIterations(landscapes[0]);
#endif /* ! FRACTAL_GENERATOR */
 if (landscapes[0]->fractal_configuration[2] <= fp->
     final_stepsize)
  RENDER_STATE = 3;
}

void printMatrix(float *matrix)
{
 int i;
 for(i=0;i<4;i++)
 {
  printf("%1.1f %1.1f %1.1f %1.1f\n",matrix[i],matrix[i+4],matrix
      [i+8],matrix[i+12]);
 }
}

#ifdef _WIN32
void initializeEGL(){
    /* EGL Configuration */
    EGLint aEGLAttributes[] = {
        EGL_RED_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_BLUE_SIZE, 8,
        EGL_DEPTH_SIZE, 16,
        EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
        EGL_NONE
    };

    EGLint aEGLContextAttributes[] = {
        EGL_CONTEXT_CLIENT_VERSION, 2,
        EGL_NONE
    };

    EGLConfig aEGLConfigs[1];
    EGLint  cEGLConfigs;

  /* EGL Init */
    hDisplay = EGL_DEFAULT_DISPLAY;
```

```
    sEGLDisplay = EGL_CHECK(eglGetDisplay(hDisplay));
    EGL_CHECK(eglInitialize(sEGLDisplay, NULL, NULL));
    EGL_CHECK(eglChooseConfig(sEGLDisplay, aEGLAttributes,
        aEGLConfigs, 1, &cEGLConfigs));
    if (cEGLConfigs == 0) {
        printf("No EGL configurations were returned.\n");
        exit(-1);
    }
    hWindow = create_window(uiWidth, uiHeight);
    sEGLSurface = EGL_CHECK(eglCreateWindowSurface(sEGLDisplay,
        aEGLConfigs[0], (EGLNativeWindowType)hWindow, NULL));
    if (sEGLSurface == EGL_NO_SURFACE) {
        printf("Failed to create EGL surface.\n");
        exit(-1);
    }
    sEGLContext = EGL_CHECK(eglCreateContext(sEGLDisplay,
        aEGLConfigs[0], EGL_NO_CONTEXT, aEGLContextAttributes));
    if (sEGLContext == EGL_NO_CONTEXT) {
        printf("Failed to create EGL context.\n");
        exit(-1);
    }
    EGL_CHECK(eglMakeCurrent(sEGLDisplay, sEGLSurface,
        sEGLSurface, sEGLContext));
}
#endif /* _WIN32 */

void initializeShaders()
{

 /* Shader Initialisation */
    process_shader(&uiVertShader, "shader.vert", GL_VERTEX_SHADER
        );
  printf("vertx shader completed.\n");
 process_shader(&uiFragShader, "shader.frag", GL_FRAGMENT_SHADER)
    ;

  printf("fragment shader completed.\n");
    /* Create uiProgram (ready to attach shaders) */
    uiProgram = GL_CHECK(glCreateProgram());

    /* Attach shaders and link uiProgram */
    GL_CHECK(glAttachShader(uiProgram, uiVertShader));
    GL_CHECK(glAttachShader(uiProgram, uiFragShader));
    GL_CHECK(glLinkProgram(uiProgram));

 /* Get uniform locations */
    iLocMVP = GL_CHECK(glGetUniformLocation(uiProgram, "mvp"));

 GL_CHECK(glUseProgram(uiProgram));
```

```
181 }
#ifndef _WIN32
183 void render(platform_info_t * platform_info)
{
185 #else
void render()
187 {
 MSG sMessage;
189 #endif /* ! _WIN32 */
 int bDone = 0;
191  int i;
#ifdef WORST_CASE
193  int num_frames = 30;
#else
195  int num_frames = 500;
#endif /* WORST_CASE */
197
#ifdef PROFILING
199  struct timeval tim;
 double t1 = 0;
201  double t2 = 0;
 double time_passed = 0;
203 #endif /* PROFILING */
 struct fractal_point fp[4];
205  struct fractal_point initial_fp;
 struct fractal_point* current_fp;
207  // Matrices
 float rotate[16] = {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
      1 };
209  float landscapeScaling[16] = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1,
     0, 0, 0, 0, 1 };
 float landscapeMvp[16];
211  float aPerspective[16] = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
     0, 0, 0, 1 };
 float landscapeModelView[16] = { 1, 0, 0, 0, 0, 1, 0, 0, 0, 0,
     1, 0, 0, 0, 0 , 1 };
213  /* Interesting points
   -0.1011f, 0.9563f
215   -0.743643, 0.131825
 */
217  /* Initial fp */

219 #ifndef WORST_CASE
 initial_fp.x = -1.5;
221  initial_fp.y = 0.8;
 initial_fp.initial_stepsize = 0.01;
223  initial_fp.final_stepsize = 0.01;
 initial_fp.z_speed = 1;
225 #else
```

```
  // Worst case
227  initial_fp.x = 0.0;
  initial_fp.y = 0.0;
229  initial_fp.initial_stepsize = 0.00001;
  initial_fp.final_stepsize =  0.00001;
231  initial_fp.z_speed = 1;
#endif /* !WORST_CASE */
233
  fp[0] = initial_fp;
235
  fp[1].x = -0.743643f;
237  fp[1].y = 0.131825f;
  fp[1].initial_angle = 0.5f;
239  fp[1].distance = -0.5;
  fp[1].r_speed = 1;
241  fp[1].n_speed = 1;
  fp[1].z_speed = 1;
243  fp[1].final_stepsize = 0.0000001f;

245  fp[2].x = -0.1011f;
  fp[2].y = 0.9563f;
247  fp[2].initial_angle = 0.5f;
  fp[2].distance = -0.5;
249  fp[2].r_speed = 1;
  fp[2].n_speed = 1;
251  fp[2].z_speed = 1;
  fp[2].final_stepsize = 0.0000001f;
253
  //Worst case point 0.0
255  fp[3].x = 0.0f;
  fp[3].y = 0.0f;
257  fp[3].initial_angle = 0.5f;
  fp[3].distance = -0.5;
259  fp[3].r_speed = 1;
  fp[3].n_speed = 1;
261  fp[3].z_speed = 1;
  fp[3].final_stepsize = 0.0000001f;
263


265

  glEnable(GL_DEPTH_TEST);
267  //glCullFace(GL_BACK);
  //glDisable(GL_CULL_FACE);
269  /* Enter event loop */
#ifdef PROFILING
271  gettimeofday(&tim,NULL);
  t2 = tim.tv_sec + (tim.tv_usec/1000000.0);
273#endif /* PROFILING */
    while (bDone<num_frames)
```

```
275  {
#ifdef _WIN32
277          if(PeekMessage(&sMessage, NULL, 0, 0, PM_REMOVE)) {
                 if(sMessage.message == WM_QUIT) {
279                  bDone = 1;
                 } else {
281                  TranslateMessage(&sMessage);
                     DispatchMessage(&sMessage);
283              }
             }
285 #endif /* _WIN32 */

287    perspective_matrix(1.05, (double)uiWidth/(double)uiHeight,
           0.01, 100.0, aPerspective);
       multiply_matrix(aPerspective, landscapeModelView, landscapeMvp)
           ;
289
       // Draw landscape VBO
291    glUniformMatrix4fv(iLocMVP, 1, GL_FALSE, landscapeMvp);
         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
             GL_STENCIL_BUFFER_BIT);
293
       for(i=0;i<(num_of_landscapes*num_of_landscapes);i++)
295    {
        // Bind attributes
297     //printf("Landscapes.xy = %f,%f\n", landscapes[i]->x,
            landscapes[i]->y);
        //printf("Landscapes.vxy = %f,%f\n", landscapes[i]->vertices
            [0], landscapes[i]->vertices[2]);
299     glBindBuffer(GL_ARRAY_BUFFER, landscapes[i]->vboIds[0]);
        glEnableVertexAttribArray(positionLoc);
301     glVertexAttribPointer(positionLoc, 4, GL_FLOAT, GL_FALSE, 0,
            0);

303     glBindBuffer(GL_ARRAY_BUFFER, landscapes[i]->vboIds[1]);
        glEnableVertexAttribArray(afheightLoc);
305 #ifdef FRACTAL_GENERATOR
        glVertexAttribPointer(afheightLoc, 1, GL_FLOAT, GL_FALSE, 0,
307                 landscapes[i]->fractal_configuration);
    #else
309     glVertexAttribPointer(afheightLoc, 1, GL_FLOAT, GL_FALSE, 0,
            0);
    #endif /* FRACTAL_GENERATOR */
311     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,landscapes[i]->vboIds[2])
            ;
       GL_CHECK(glDrawElements(GL_TRIANGLE_STRIP, sizeof(landscapes[i
            ]->indices)/2,
313                  GL_UNSIGNED_SHORT, NULL));
       }
```

```
315     //printf("\n\n");
        // Draw Sky
317     //GL_CHECK(glDrawElements(GL_TRIANGLE_STRIP, indexInfo[1].size,
            GL_UNSIGNED_SHORT, NULL));
    #ifdef _WIN32
319     if (!eglSwapBuffers(sEGLDisplay, sEGLSurface)) {
                printf("Failed to swap buffers.\n");
321         }
            Sleep(20);
323 #else
        platform_swapbuffers( platform_info );
325     bDone++;
    #ifdef PROFILING
327     gettimeofday(&tim, NULL);
        t1 = tim.tv_sec + (tim.tv_usec/1000000.0);
329     time_passed += (t1-t2);
         t2 = t1;
331     printf("Time passed = %.6lf seconds\n", time_passed);
    #endif /* PROFILING */
333 #endif /* _WIN32 */

335     switch(RENDER_STATE)
        {
337     case(0):
         printf("Demo: Performing initial rotation.\n");
339      initial_rotation(current_fp, rotate);
         multiply_matrix(rotate, landscapeModelView, landscapeModelView
             );
341      break;
        case(1):
343      printf("Demo: Navigating to point.\n");
         navigate_to_point(current_fp, landscapeModelView);
345      multiply_matrix(rotate, landscapeModelView, landscapeModelView
             );
         break;
347     case(2):
         printf("Demo: Zooming towards point\n");
349      zoom_point(current_fp);
         break;
351     default:
         // Move camera to starting position
353      printf("Demo: Moving camera to starting position.\n");
         landscapes[0]->fractal_configuration[0] = initial_fp.x;
355      landscapes[0]->fractal_configuration[1] = initial_fp.y;
         landscapes[0]->fractal_configuration[2] = initial_fp.
             initial_stepsize;
357      rotate_matrix(xRotateAngle,1.0f,0.0f,0.0f,rotate);
         multiply_matrix(rotate, Identity, landscapeModelView);
359      rotate_matrix(yRotateAngle,0.0f,1.0f,0.0f,rotate);
```

```
    multiply_matrix(rotate, landscapeModelView, landscapeModelView
        );
361    landscapeModelView[14] = zTranslation;
    landscapeModelView[13] = yTranslation;
363    // Select a random point from the list of interesting points.
    // There is no call to srand so always same sequence, but that
        is okay.
365  #ifndef WORST_CASE
    //current_fp=&fp[(rand()%2)+1];
367    current_fp=&fp[1];
  #else
369    current_fp=&fp[3];
  #endif /* !WORST_CASE */
371    RENDER_STATE = 0;
    break;
373   }
  } /* while */
375  printf("The average frame time was = %.6lf\n", time_passed/
      num_frames);
}
377
void cleanup(){
379  /* Cleanup shaders */
     GL_CHECK(glUseProgram(0));
381     GL_CHECK(glDeleteShader(uiVertShader));
     GL_CHECK(glDeleteShader(uiFragShader));
383     GL_CHECK(glDeleteProgram(uiProgram));
#ifdef _WIN32
385     /* EGL clean up */
     EGL_CHECK(eglMakeCurrent(sEGLDisplay, EGL_NO_SURFACE,
         EGL_NO_SURFACE, EGL_NO_CONTEXT));
387     EGL_CHECK(eglDestroySurface(sEGLDisplay, sEGLSurface));
     EGL_CHECK(eglDestroyContext(sEGLDisplay, sEGLContext));
389     EGL_CHECK(eglTerminate(sEGLDisplay));
#endif /* _WIN32 */
391 }

393
int main(int argc, char **argv)
395 {
  // Declarations on top
397  struct fractal_landscape fractal;
  int i;
399 #ifndef _WIN32
  platform_info_t platform_info;
401  platform_info.num_bits_rgba[0]=5;
  platform_info.num_bits_rgba[1]=6;
403  platform_info.num_bits_rgba[2]=5;
  platform_info.num_bits_rgba[3]=0;
```

```
405  platform_info.num_samples = 4;
     platform_info.width = 400;
407  platform_info.height = 300;
     platform_info.api = PLATFORM_EGL_API_GLES2;
409  if ( FALSE == platform_init(&platform_info) )
     {
411   printf("Failed to initialize windowing system.\n");
      return -1;
413  }
    #endif /* ! _WIN32 */
415  // Pre rendering
     printf("Initializing\n");
417 #ifdef _WIN32
     initializeEGL();
419  printf("Initialized EGL\n");
    #endif /* _WIN32 */
421  initializeShaders();
     printf("Initalized shaders\n");
423  // Initialize and create fractals. TODO: Should be functions to
        make large landscape.
     fractal.shader=uiProgram;
425  fractal.fractal_configuration[0]= -1.5f;
     fractal.fractal_configuration[1] = 0.8f;
427  fractal.fractal_configuration[2] = 0.01f;
     printf("Creating landscape\n");
429  create_landscape(landscapes,&num_of_landscapes,
             fractal.fractal_configuration[0],
431          fractal.fractal_configuration[1],
             fractal.fractal_configuration[2]);
433  printf("Landscape initialized\n");
     printf("num_of_l = %i\n", num_of_landscapes);
435  for(i=0;i<(num_of_landscapes*num_of_landscapes);i++)
      createFractalLandscape(landscapes[i]);
437
     printf("Landscape created\n");
439  // Render and cleanup
    #ifdef _WIN32
441  render();
    #else
443  render(&platform_info);
    #endif /* _WIN32 */
445  printf("Finished render\n");
     cleanup();
447  return 0;
    }
449
    #ifdef _WIN32
451 HWND create_window(int uiWidth, int uiHeight) {
     WNDCLASS wc;
```

```
453  RECT wRect;
     HWND sWindow;
455  HINSTANCE hInstance;

457  wRect.left = 0L;
     wRect.right = (long)uiWidth;
459  wRect.top = 0L;
     wRect.bottom = (long)uiHeight;
461
     hInstance = GetModuleHandle(NULL);
463
     wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
465  wc.lpfnWndProc = (WNDPROC)process_window;
     wc.cbClsExtra = 0;
467  wc.cbWndExtra = 0;
     wc.hInstance = hInstance;
469  wc.hIcon = LoadIcon(NULL, IDI_WINLOGO);
     wc.hCursor = LoadCursor(NULL, IDC_ARROW);
471  wc.hbrBackground = NULL;
     wc.lpszMenuName = NULL;
473  wc.lpszClassName = "OGLES";

475  RegisterClass(&wc);

477  AdjustWindowRectEx(&wRect, WS_OVERLAPPEDWINDOW, FALSE,
         WS_EX_APPWINDOW | WS_EX_WINDOWEDGE);

479  sWindow = CreateWindowEx(WS_EX_APPWINDOW | WS_EX_WINDOWEDGE, "
         OGLES", "main", WS_OVERLAPPEDWINDOW | WS_CLIPSIBLINGS |
         WS_CLIPCHILDREN, 0, 0, uiWidth, uiHeight, NULL, NULL,
         hInstance, NULL);

481  ShowWindow(sWindow, SW_SHOW);
     SetForegroundWindow(sWindow);
483  SetFocus(sWindow);

485  return sWindow;
     }
487  /*
      * process_window(): This function handles Windows callbacks.
489   */
     LRESULT CALLBACK process_window(HWND hWnd, UINT uiMsg, WPARAM
         wParam, LPARAM lParam) {
491  switch(uiMsg)
     {
493    case WM_CLOSE:
         PostQuitMessage(0);
495      return 0;
       case WM_CHAR:
```

```
497     switch(wParam)
        {
499       case 0x1B: //Escape
          PostQuitMessage(0);
501       return 0;
        }
503     case WM_ACTIVATE:
        case WM_KEYDOWN:
505     switch(wParam)
        {
507       case VK_LEFT:
          // Rotates camera to the left
509       yRotateAngle +=0.5f;
          return 0;
511       case VK_RIGHT:
          // Rotates camera to the right.
513       yRotateAngle -=0.5f;
          return 0;
515       case VK_DOWN:
          // Moves camera backward
517       zTranslation -= 0.02f; // Move camera a little back.
          return 0;
519       case VK_UP:
          // Moves camera forward
521       zTranslation += 0.02f;
          return 0;
523       case VK_PRIOR: // Page up.
          // Rotates camera up
525       xRotateAngle += 0.5f;
          return 0;
527       case VK_NEXT: // Page down.
          // Rotates camera down.
529       xRotateAngle -= 0.5f;
          return 0;
531       case VK_HOME:
          return 0;
533       case VK_INSERT:
          return 0;
535       case VK_DELETE:
          return 0;
537       case VK_END:
          return 0;
539     }
        case WM_KEYUP:
541     case WM_SIZE:
          return 0;
543   }
      return DefWindowProc(hWnd, uiMsg, wParam, lParam);
545 }
```

```
#endif /* _WIN32 */
```

../source/demo/main.c

```
/*
 * This proprietary software may be used only as
 * authorised by a licensing agreement from ARM Limited
 * (C) COPYRIGHT 2009 - 2011 ARM Limited
 * ALL RIGHTS RESERVED
 * The entire notice above must be reproduced on all authorised
 * copies and copies may only be made to the extent permitted
 * by a licensing agreement from ARM Limited.
 */

#define MAX_ITERATIONS 80.0
attribute vec4 av4position;
attribute float afheight;

uniform mat4 mvp;

varying float vfcolor;

void main()
{
  // afheight2 is used to avoid two divisions by 80,
  // once here and once in the fragment shader.
  float afheight2 = afheight/MAX_ITERATIONS;
  vfcolor = afheight2;
   vec4 pos = vec4( av4position.x, afheight2, av4position.zw );
  gl_Position = mvp * pos;
}
```

../source/demo/shader.vert

```
/*
 * This proprietary software may be used only as
 * authorised by a licensing agreement from ARM Limited
 * (C) COPYRIGHT 2009 - 2011 ARM Limited
 * ALL RIGHTS RESERVED
 * The entire notice above must be reproduced on all authorised
 * copies and copies may only be made to the extent permitted
 * by a licensing agreement from ARM Limited.
 */

precision lowp float;

varying float vfcolor;

void main()
```

```
16 {
18  vec3 v_color;
20  const vec3 c1 = vec3(0.165, 0.244, 0.518); // Light blue
    const vec3 c2 = vec3(1.0, 1.0, 1.0); // White
22
    if (vfcolor == 1.0) // MAX_ITERATIONS REACHED
24   v_color = vec3(1.0, 0.0, 0.0); // Red
    else
26   v_color = mix(c1, c2, (vfcolor) );
28  gl_FragColor = vec4(v_color, 1.0);
    }
```

../source/demo/shader.frag