ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
ADDIS ABABA INSTITUTE OF TECHNOLOGY
ELECTRICAL & COMPUTER ENGINEERING DEPARTMENT

Design of 3D Graphics Accelerator Core for FPGA

By

Kibret Abebe

Advisor

Dr. Getachew Alemu

A thesis submitted to the school of Graduate studies of Addis Ababa University in partial fulfillment of the requirements for the degree of Masters of Science in Microelectronics Engineering

October 2011
Addis Ababa, Ethiopia

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
ADDIS ABABA INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Design of 3D Graphics Accelerator Core for FPGA

By

Kibret Abebe

Advisor

Dr. Getachew Alemu

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

Design of 3D Graphics Accelerator Core for FPGA

By

Kibret Abebe Messalea

ADDIS ABABA INSTITUTE OF TECHNOLOGY

APPROVAL BY BOARD OF EXAMINERS

_____	_____
Chairman, Dept. of Graduate Committee	**Signature**
Dr. Getachew Alemu	_____
Advisor	**Signature**
_____	_____
Internal Examiner	**Signature**
_____	_____
External Examiner	**Signature**

Declaration

I, the undersigned, declare that this thesis work is my original work, has not been presented for a degree in this or any other universities, and all sources of materials used for the thesis work have been fully acknowledged.

Kibret Abebe Messalea _____
Name signature

Place: Addis Ababa

_____
Date of submission

This thesis has been submitted for examination with my approval as a university advisor.

Dr. Getachew Alemu _____
Advisor's name signature

Dedication

For my mother Bayoush Tsegaye and father Abebe Messalea.

ACKNOWLEDGMENTS

First I would like to thank to my advisor Dr. Getachew Alemu for his invaluable support and encouragement. In addition, I thank the department of Electrical and Engineering and Addis Ababa University School of Graduates for their invaluable support. My gratitude also goes to Xilinx University Program (XUP) for their free of charge equipment donation. Last but not least, I thank Michael Tetemke for his friendly support; a friend in need is a friend in deed.

Table of Contents

Content **Page**

Declaration i
Dedication ii
ACKNOWLEDGMENTS iii
LIST OF TABLES vi
LIST OF FIGURES vii
LIST OF ACRONYMS viii
ABSTRACT x
CHAPTER ONE: INTRODUCTION 1
1.1 BACKGROUND 1
1.2 MOTIVATION 2
1.3 STATEMENT OF THE PROBLEM 3
1.4 OBJECTIVES 4
1.5 SCOPE OF THE THESIS 4
1.6 METHODOLGY 4
1.7 CONTRIBUTION OF THE THESIS 5
1.8 THESIS ORGANIZTION 5
CHAPTER TWO: LITRATURE REVIEW 6
2.1 3D GRAPHIC PIPELINE 6
2.2 FPGA ARCHITECTUR 14
2.3 RELATED WORKS 19
2.4 CHAPTER SUMMARY 25
CHAPTER THREE:3D GRAPHICS PIPELINE AND DESIGN 26
3.1 GRAPHICS AND 3D TRANSFORMATIONS 26
3.1.1 TRANSFORMATION MODULE DESIGN 28
3.2 CLIPPING UNIT 31
3.2.1 INTERSECTION TESTS 32
3.2.2 THREE DIMENTIONAL CLIPPING 33
3.3 RASTERIZATION UNIT 38
3.3.1 LINE RASTERIZER 39
3.4 CHAPTER SUMMARY 42
CHAPTER FOUR 43
4.1 GRAPHICS CORE SYNTHESIS AND TIMING SIMULATION 43
4.2 TRANSFORATION UNIT SIMULATION 44
4.2.1 MATRIX MULTIPIER 44
4.2.2 MATRIX TRANSFORMATION 45
4.3 CLIPPING UNIT SIMULATION 47
4.3.1 CODE GENERATIOR SUBUNIT 47
4.3.2 EDGE INTERSECITON SUBUNIT 48
4.3.3 TOP MODULE OF CLIPPING UNIT 49
4.4 RASTERIZER UNIT SIMULATION 50
4.5 3D GRAPHICS ACCELERATOR TOP MODULE 52

4.6 CHAPTER SUMMARY 57
CHAPTER FIVE: RESUSLT AND DISCUSSION 58
5.1 MATRIX MULTIPLIER UNIT 58
5.2 TRANSFORMATION UNIT 59
5.3 CLIPPING UNIT 60
5.4 RASTRIZATION UNIT 61
5.5 GRAHPICS ACCELERATIOR TOP MODULE 62
5.6 CHAPTER SUMMARY 63
CHAPTER SIX 64
CONCLUSION AND FUTURE WORK 64
6.1 CONCLUSION 64
6.2 FUTURE WORK 65
RERERENCES 66
APPENDIX 70

LIST OF TABLES

Table **Page**
Table 2.1 Performance and Feature Summary of Low Power Rasterization Unit...……22
Table 3.1 3D Parallel Projection OutCode Assignment……34
Table 3.2 3D Perspective Projection Out Code Assignment……34
Table 3.3 3D Parallel Projection Clipping Intersection Equations……..…35
Table 3.4 3D Perspective Projection Clipping Intersection Equations……36
Table 4.1 Input & Output Signal Discription of Line Rasterizer Module……50
Table 4.2 Inputs Descriptions of 3D Graphics Accelerator Core…….……...53
Table 4.3 Outputs Descriptions of 3D Graphics Accelerator Core ……...54
Table 5.1 Device Utilization of Matrix Multiplier ……58
Table 5.2 Device Utilization of Matrix Transformer……59
Table 5.3 Device Utilization of Clipping Unit……...60
Table 5.4 Device Utilization of Rasterizer ……61
Table 5.5 Device Uitilization of 3D Graphics Accelerator……62

LIST OF FIGURES

Figure **Page**

Figure 2.1 3D Graphics Pipeline Stages 6
Figure 2.2 Graphics Pipeline 7
Figure 2.3 Space Coordinate Systems in 3D Graphics 8
Figure 2.4 Modeling Transformation Example 9
Figure 2.5 Viewing Frustum 10
Figure 2.6 Perspective Transformation and Its Matrix 11
Figure 2.7 Normalized Device Coordinate Space 12
Figure 2.8 Device Coordinate Space 13
Figure 2.9 SRAM Based FPGA Configuration 14
Figure 2.10 Generic Island Stayle Routing Architecture 15
Figure 2.11 Vertex-II Pro Slice 16
Figure 2.12 Architecture of 3D Graphics Accelerator Platform 19
Figure 2.13 3D graphics Full Pipeline 23
Figure 2.14 Triangle Setup Engine Block Diagram 24
Figure 3.1 Sample 3D Object Definition 26
Figure 3.2 Matrix 4X4 Multiplier Blocks 29
Figure 3.3 Matrix Transformation Unit Block Diagram 30
Figure 3.4 Region Codes For Nine Regions 31
Figure 3.5 Outcod Generator for Clipping Unit 32
Figure 3.6 Coordinates of Corners of the Clipping Window 33
Figure 3.7 Parallel Projection 33
Figure 3.8 Perspective Projection 34
Figure 3.9 Edge Intersection Calculation 37
Figure 3.10 Rasterization in Block Diagram 38
Figure 3.11 Pixel Grid for Bresenham's Midpoint Based Line Generator 39
Figure 4.1 Top Level Blocks that Define 3D Graphics Accelerator 43
Figure 4.2 Top Module of Matrix Multiplier 44
Figure 4.3 Simulation of Matrix Multiplier 44
Figure 4.4 Top RTL of Matrix Transformaiton Unit 45
Figure 4.5 Simulation of Matrix Transformation 46
Figure 4.6 RTL of Code Generator for Clipping Decision 47
Figure 4.7 Code Generator Simulation 47
Figure 4.8 RTL Edge Intersaction Calculation 48
Figure 4.9 Simulation of Edge Intersection Calculation 48
Figure 4.10 RTL of Clipping Unit 49
Figure 4.11 Simulation of Clipping Unit 49
Figure 4.12 RTL of Top Level Rasterizer 51
Figure 4.13 Rasterizer Unit Simulation 52
Figure 4.14 3D Graphics Accelerator Top Module 53
Figure 4.15 RTL view of Top Core Components 56

LIST OF ACRONYMS

Altera	FPGA Manufacturer Company
ARM	Advanced RISC Machines Ltd.
ASIC	Application Specific Integrated Circuits
API	Application Interface
CLB	Configurable Logic Block
CPU	Central Processing Unit
CPLD	Complex Programmable Logic Devices
DCM	Digital Clock Manager
DSP	Digital Signal Processing
DVI	Digital Visual Interface
EDA	Electronic Design Automation
FIFO	First In , First Out
FPGA	Field Programmable Gate Array.
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
I/O	Input Out put
IP-Core, Core	Intellectual Property-Core
JTAG	Joint Test Action Group
LCD	Liquid Crystal Display
LE	Logic Elements
LUT	Look-Up Table
LVDS	Low Voltage Differential Signal
Microblaze	A soft-core 32 bit RISC microprocessor designed specifically for Xilnix FPGAs
MIPMAP	3D computer graphics texture filtering technique
NVIDIA	Graphics processor Company
OPEN GL	Open Graphics Library
OPENGL ES	Open Graphics Library for Embedded Systems
PAL	Programmable Array Logic
PCI	Peripheral Component Interconnect
Picoblaze	A fully embedded 8-bit microcontroller macro for virtex series of FPGAs.
Pipeline	A sequence of functional units which performs task in several steps.
Pixel	Contraction of Picture element
Polygon	A plane figure having many angles , and consequently many sides
PROM	Programmable Read Only Memory
QVGA	Quarter Video Graphics Array
Raster graphics	Computer graphics in which an image is composed of an array of pixels arranged rows and columns
RAM	Random Access Memory
RGB	Red-Green-Blue
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory

RS-232	Serial line standard
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on Chip
UART	Universal Asynchronous Receiver and Transmitter
Verilog	A Hardware Description Language for electronics de-sign and gate level Simulation
Vertex	The point of intersection of lines
Virtex 5	Xilinx FPGA
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
VGA	Video Graphics Array
Xilinx	Company invented the FPGA
XUP	Xilinx University Program
XPS	Xilinix Platform Studio
ZBT	Zero Bus Turnaround
3D	Three Dimensional

Abstract

In this thesis we designed and synthesized 3D graphics accelerator Intellectual Property (IP) using VHDL. The main parts which were designed in this work include: the geometry unit and the rasterizer unit of 3D graphics pipeline. The geometry unit design contains two main parts , the matrix transformation unit and clipping unit. The rasterzer unit based on bresenham line drawing algorithm. By doing this state-of-the-art design methodology divided in two sections, the first section is used to describe each block (i.e the matrix multiplier, matrix transformer , clipping and rasterizer) of the pipeline using VHDL and simulate each unit behaviorally. Second part synthesizes the system using Xilinx ISE 13.1 tool on XUP5VLX110T-1F1136. After doing the synthesis the designed 3D graphics accelerator performance was found to be 100M pixel fill rate with maximum clock frequency of 169 MHz.

CHAPTER ONE
INTRODUCTION

1.1 BACKGROUND

Three-dimensional graphics started with the display of data on hardcopy plotters and CRT screens soon after the introduction of computers themselves. It has grown to include the creation, storage, and manipulation of models and images of objects. These models come from a diverse and expanding set of fields, and include physical, mathematical, engineering, architectural, and even conceptual structures, natural phenomena, and so on[1].

Until the early 1980s, 3D graphics was used in specialized fields because the hardware was expensive and there were few graphics-based application programs that were easy to use and cost-effective. Since personal computers have become popular, 3D graphics is widely used for various applications, such as user interfaces and games. Today, almost all interactive programs, even those for manipulating text (e.g., word processors) and numerical data (e.g., spreadsheet programs), use graphics extensively in the user interface and for visualizing and manipulating the application-specific objects. So 3D graphics is no longer a rarity and is indispensable for visualizing objects in areas as diverse as education, science, engineering, medicine, commerce, military, advertising, and entertainment [2].

Fundamentally, 3D graphics simulates the physical phenomena that occur in the real world especially dynamic mechanical and lighting effects – on 2D display devices. Thus the role of the 3D graphics pipeline is to project 3D objects on to a 2D screen with appropriate lighting effects. The 3D graphics pipeline is composed of application, geometry, and rendering stages. The application stage computes the dynamic behavior description of 3D objects; the objects are transformed and vertex information is computed in the geometry stage; and the information for each pixel is computed in the rendering stage. Recently, programmability has been introduced into the 3D graphics pipeline to support various graphics effects, including non-photorealistic effects. This approach supports programmability in the geometry and rendering stages [5].

1.1. MOTIVATION

Accompanied with the improvement of silicon process the single chip has a capacity for some computation exhaustive of jobs such as Three Dimensional graphics calculations. Hardware accelerated (3D) graphics processing is no longer only proper to the desktop PCs or workstations but also to the embedded system. Although 3D graphics is not a new field for computers, almost every desktop computer has the ability to process 3D graphics, but for the low cost consumer electronics the CPU may not have enough computing power and it becomes a new challenge for an embedded system to accomplish such jobs. It needs a specialized accelerator that provides enough performance addition to minimized cost. To build such kind of SoC (system-on-a-chip) there are two well known methods. The first one is to continue using the desktop technology but eliminating the unnecessary functions to reduce cost. It is the fastest way to build the SoC and most designers use this way to design a commercial 3D accelerating chip because of time-to-market. Although it is fast, but it is usually difficult to eliminate the whole redundant hardware clearly and it will not be the lowest cost design. Secondly, it is to design a totally new SoC according to the specific applications. The advantage will be getting a better system we expected but it is a time consuming work.

Since real time graphics processing requires extreme high performance, hardware solutions using Application Specific Integrated Circuits (ASICs) are the standard within the industry. While ASICs are a more than adequate solution for implementing high performance custom hardware, the design, implementation and testing of ASIC based designs are becoming cost prohibitive due to the massive up front verification effort needed as well as the cost of fixing design defects. Field Programmable Gate Arrays (FPGAs) provide an alternative to the ASIC design flow. More importantly, in recent years FPGA technology have begun to improve in performance to the point where ASIC and FPGA performance has become comparable. In addition, FPGAs address many of the issues of the ASIC design flow. The ability to reconfigure FPGAs reduces the upfront verification effort and allows design defects to be fixed easily.

1.2. STATEMENT OF THE PROBLEM

In modern graphics systems there are generally two architectures used: the classic fixed pipeline and the multiprocessor approach. Fixed Pipeline is the most ideal for implementation on FPGA. In addition, applications like graphics could take advantage of FPGA based processing implementations as opposed to using ASICs for development so that, we try to see the feasibility of designing 3D graphics system on Virtex 5 FPGA.

The FPGA based 3D graphics accelerator core requirements include:

- Graphics accelerator capable of rendering in both 2D and 3D
- Processing of wireframe objects.
- Support standard Microblaze Interface, and
- Support standard display interfaces

The objects defined can be rotated, scaled or moved within virtual world and displayed on to a 2D display.

The thesis presents the work performed to realize an FPGA based graphics accelerator IP Core. The document first must define requirements which are subset of graphics processing functions to be implemented. Once identified and described, the functions must be designed to run efficiently within an FPGA device. Given the design requirements an FPGA must selected that has the logic resources and performance to implement these proposed graphics functions. Following design, the FPGA based graphics processor must then be fully verified and tested for functional correctness as well as its area and speed performance using Xilinx ISE 13.1.

1.2. OBJECTIVES

i. General Objective

The general objective of this thesis is to design 3d graphics accelerator core using VHDL.

ii. Specific Objectives

The specific objectives of the thesis are:

- To implement graphics based on pipelining which can process wireframe objects.
- To test the feasibility of implementing complex graphics processing functions on Xilinx XUP5VLX110T-1F1136 FPGA device.
- To study Vertex 5 FPGA architecture.
- To do performance analysis for the designed 3D graphics accelerator core.

1.3 SCOPE OF THE THESIS

This work designed 3D graphics accelerator core that can only process wired frame objects. The pipeline components that are designed and synthesized in this work are the Matrix transformation unit , Clipping unit and rasterizer unit. The clipping unit is based on Cohen- Sutherland clipping algorithm which can be used for 2D and 3D objects clipping. The rasterizer unit which converts the lines to pixel designed based on Bresenham algorithm. In addition the area and speed performance measures are drawn based on balanced synthesis on Xilinx 13.1 synthesizer by selecting XUP5VLX110T-1F1136 FPGA.

1.4 METHODOLOGY

To fulfill these objectives the hardware description, analysis, synthesis and Test bench simulation were used on Xilinx EDA. The design and synthesis methods include:

- Study about 3D graphic rendering algorithms and mathematical background in the area of 3D graphics.
- Design 4X4 matrix multiplier Using VHDL then, the matrix transformation unit designed using matrix multiplier and FIFO.
- Design clipping unit using VHDL by using Cohen-Sutherland 3D clipping algorithm.

- ❖ Design Rasterizer unit using VHDL by using Bresenham line rasterizer.
- ❖ Then VHDL test benches were designed to verify the graphics engine components. This is followed by synthesizing the design on Xilinx ISE Synthesizer and performance measures were drawn.

1.5 CONTRIBUTION OF THE THESIS

The designed 3D accelerator core can be applied to Xilinx FPGA based SoCs. In addition, the design implements some of graphic algorithms which are parts of the graphic pipeline.
Contribution of this thesis includes:

- Conversion graphics algorithms such as clipping and rasterizer to hardware netlist by using VHDL.
- Design of 3D graphic accelerator which can be used as open source hardware on FPGAs.
- Synthesized 3D graphics accelerator IP core which can have 100Million fill rate with a maximum frequency of 169 MHz.

1.6 THESIS ORGANIZATION

Chapter One Covers the overview of the work to be done and presents the background and the purpose of the thesis. An introduction to 3d graphics techniques and algorithms related to 3d graphics and review literatures related to this research were covered in the next chapter. Chapter three discusses all about design of 3D rendering pipeline for FPGA implementation .Moreover, the fourth chapter focuses on Top module and its components simulation , synthesis RTL views and their input output descriptions the result and discussion part were dedicated in the fifth chapter based on speed performance and area utilization. The next Chapter contains conclusion and future work.

CHAPTER TWO
LITERATURE REVIEW

2.1 THE 3D GRAPHICS PIPELINE

A 3D graphics rendering system is usually organized in the way of pipeline as presented. in the figure 2.1 and figure 2.2.We can see that a rendering system consists of three stages: application process, geometry processing and rendering. State of application layer is mainly achieved by software and used for human-computer interaction and data entry. Collision detection, texture animation, geometric transformation can also be done[2][4].

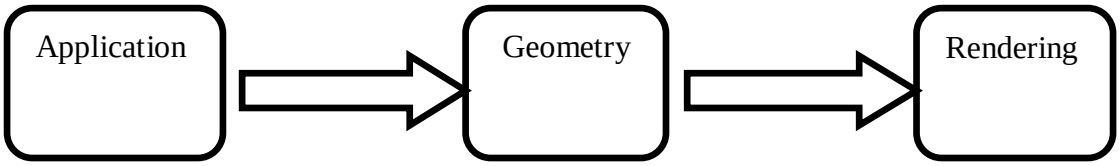


Figure 2.1: 3D graphics pipeline stages [5]

The rendering primitive is ultimately passed into the geometric stage in the rendering pipeline. Geometric phase is a computation intensive phase and completed for most of the pixel with the polygon model and points of view transformation, illumination, projection, clipping, screen, etc. Usually, it is approximate 100 times precision floating-point operations around for each vertex under one light source cases .This part of work can be realized in the form of hardware or software. The vertexes, color, texture coordinates and other data are sent to the raster state. The raster stage, mainly complete the pixel shader and realize anti-aliasing, texture mapping, fogging and other operations to provide the graphics rendering authenticity. Ultimately, these processed pixels are sent into the frame buffer, and output to the display device under the control of frame buffer controller, thus completing the mapping of the entire rendering pipeline[5].

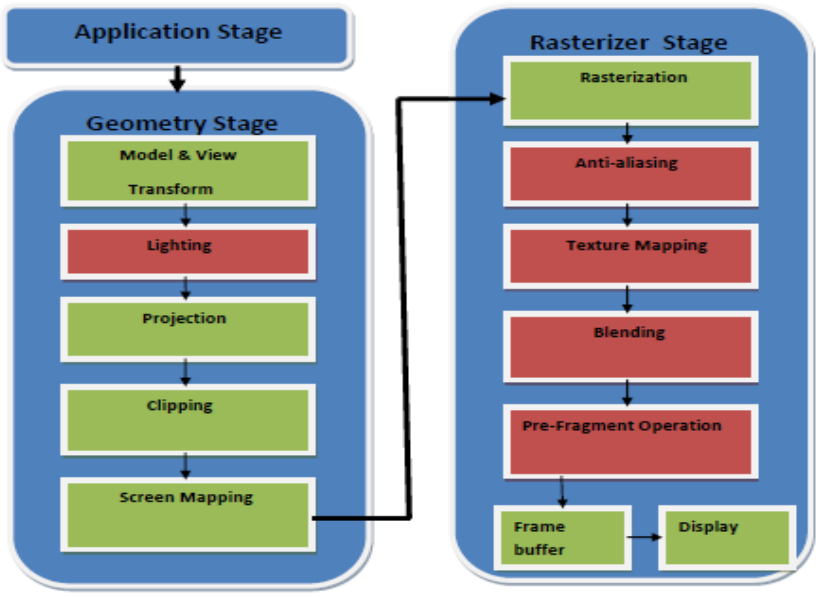


Figure 2.2 Graphic Pipelines [4]

2.1.1 THE APPLICATION STAGE

The application stage starts and drives the 3D graphics pipeline by feeding 3D models to be rendered according to the information determined in the application stage. Thus it should be understood in the context of the 3D graphics pipeline although the application stage is not an actual part of the 3D graphics subsystem. The application stage generates the movements of 3D objects based on the information gathered from the environment. The environmental information includes the user interaction from keyboard or mouse, and internally generated information in the real world. Thus the application stage also processes the artificial intelligence (AI), collision detection, and physics simulations to generate this information. Based on these, the objects' movements produce the 3D animation by moving the objects from frame to frame. The dynamics of the 3D objects and the camera position defined in the application stage also affects the animation, in which the 3D objects are moved by frames taken at certain viewpoints. In the application stage, the 3D objects are represented as sets of polygons or triangles and their movements are specified by geometry transformation matrices. These matrices are sent to the geometry stage to be used for transformation of vertex positions[5][6].

2.1.2 THE GEOMETRY STAGE

The geometry stage operates on the polygons or vertices. The major operations in this stage are, first, geometric transformation of the vertices according to the matrices determined in the application stage and, second, the lighting which determines the color intensity of each vertex according to the relationship between the properties of the vertex and the light source. The geometric transformation goes through several coordinate transformations as shown in Figure The objects defined in local model coordinate space are transformed into world coordinate and camera coordinate spaces, and finally into the device coordinate space. Each coordinate space for the geometric transformation is explained in detail in this section[5][6].

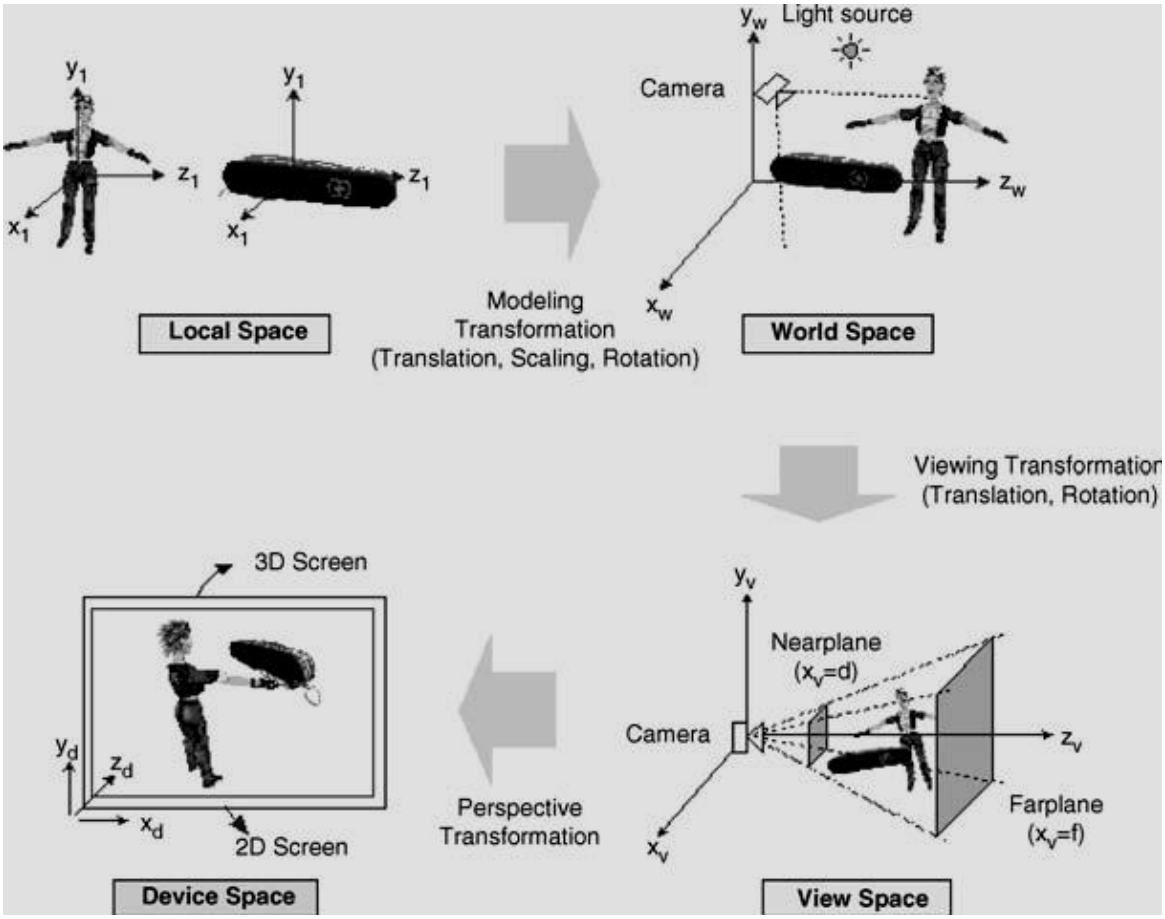


Figure 2.3 Space and Coordinate systems in 3D Graphics [5]

2.1.3 LOCAL COORDINATE SYSTEM

The local coordinate space is the space where 3D objects are developed. For modeling convenience, the 3D objects are modeled in their local coordinate spaces and the origin is located at the center or corner of each model. These models are gathered into the world space by transforming the center of each local coordinate space to the point where the object is located in the world space. This is called modeling transformation and it involves shifting, rotating and scaling operations on the original 3D object. The vertex normal is also transformed into the world space for the lighting operation. Each operation is specified by matrix coefficients, and the matrices are combined into a single modeling transformation matrix by multiplying the matrices. The following figure shows the modeling transformation operations and examples of corresponding matrices[5][7].

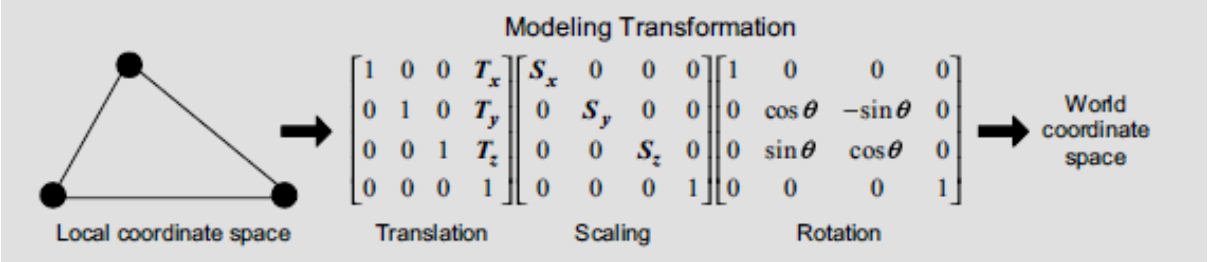


Figure 2.4: Modeling Transformation Example [5]

2.1.4 VIEWING COORDINATE SPACE

After viewing the transformation, all the objects are spaced with respect to the camera position at the origin of the view space. In this view space, culling and clipping operations are carried out in preparation for later rendering stage operations. When only the front-facing polygons of a 3D object are visible to the camera, a culling operation, also called "back-face culling," can remove polygons that will be invisible on the 2D screen. Thus the number of polygons to be processed in the later stages is reduced. This is done by rejecting back-facing polygons when seen from the camera position[5][11].

Therefore, a large amount of processing in later stages can be avoided if the visibility of a polygon is determined and culled out at this stage. In the view space, the view frustum is defined to determine the objects to be considered for a scene. Figure 2.5 shows a view frustum defined with six clipping planes, including the near and far clip planes. The objects are transformed into

the clipping coordinate space by perspective transformation shown in Figure 2.6, which is defined in terms of the view frustum definition[4][5].

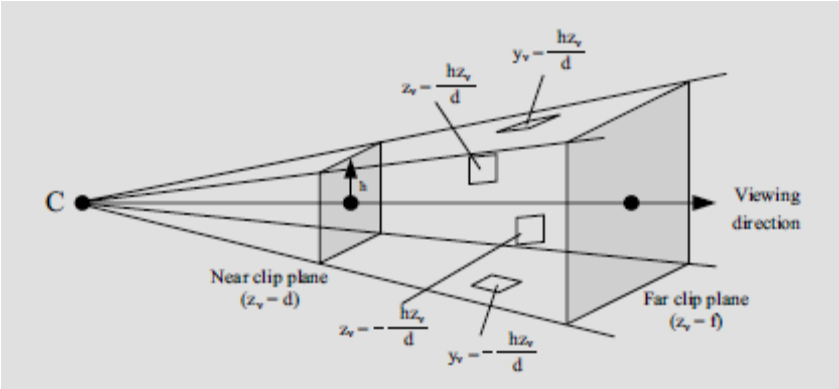
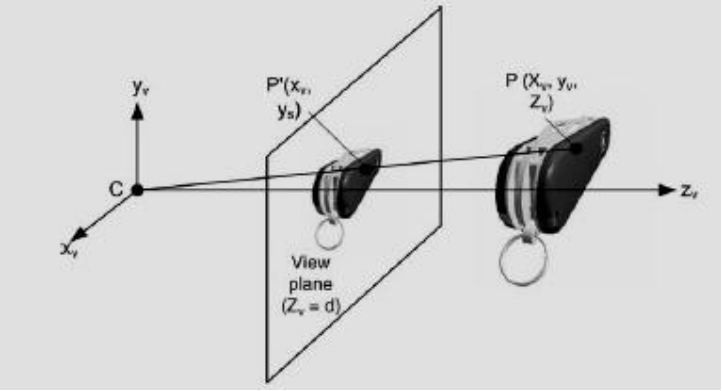


Figure 2.5: Viewing Frustum [5]



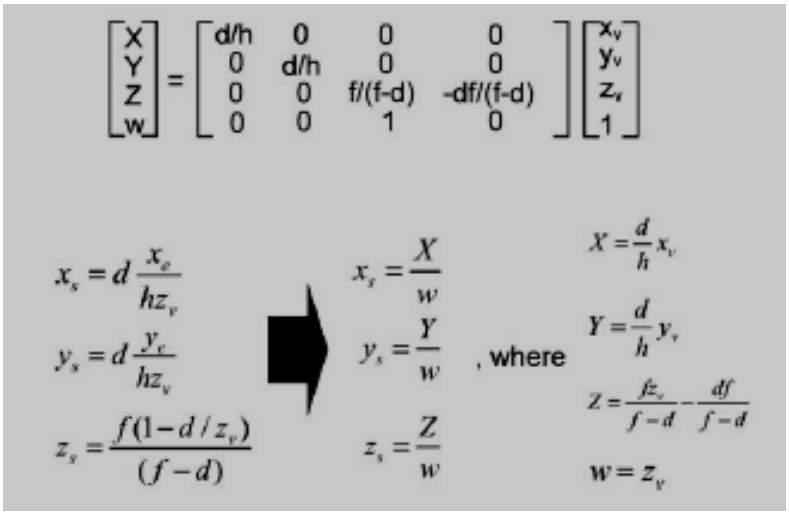


Figure 2.6: Perspective Transformation and Its Matrix[5]

2.1.5 CLIPPING COORDINATE SPACE

Although polygon clipping can be done in the view coordinate space against the view frustum with six planes, the clipping occurs in this clipping space to avoid solving plane equations. Polygon clipping against a square volume in this space is easier than in the view space, since simple limit comparisons with w component value as follows are sufficient for the clip tests:

$$-w \le x \le w$$
$$-w \le y \le w$$
$$-w \le z \le w \qquad (2.2)$$

The polygons are tested according to (above equation 2.2) and the results fall into one of three categories: completely outside, completely inside, or straddling. The “completely outside” polygons are simply rejected. The “completely inside” polygons are processed as normal. The “straddling” polygons are clipped against the six clipping planes, and those inside are processed as normal. After clipping, the polygons in the clipping space are divided by their w component, which converts the homogeneous coordinate system into a normalized device coordinate (NDC) space[2][8].

2.1.6 NORMALIZED DEVICE COORDINATE SPACE

The range of polygon coordinates in the normalized device coordinate (NDC) space is [-1,1] as shown in Figure 2.7. The polygons in this space are transformed into the device coordinate space using the viewport transformation, which determines how a scene is mapped on to the device screen. The viewport defines the size and shape of the device screen area on to which the scene is mapped. Therefore, in this transformation the polygons in NDC are enlarged shrunk or distorted according to the aspect ratio of the viewport[13][19].

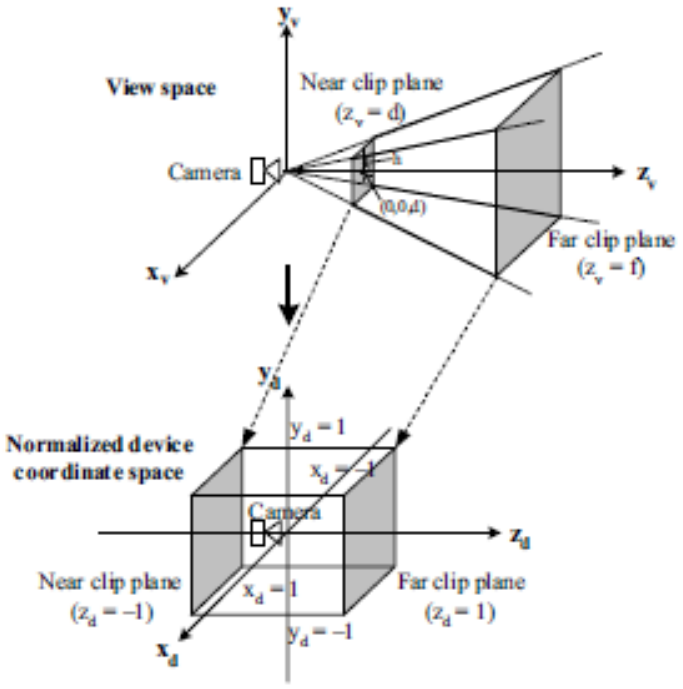


Figure 2.7 Normalized Device Coordinate Space[5].

2.1.7 DEVICE COORDINATE SPACE

After viewport transformation, the polygons are in the device coordinate space as shown in Figure 2.8. In this space, all the pixel-level operations, such as shading, Z testing, texture mapping, and blending are performed. Up to the viewport transformation is called the geometry stage, and the later stages are called the rendering stage, where each pixel value is evaluated to fill the polygons[5].

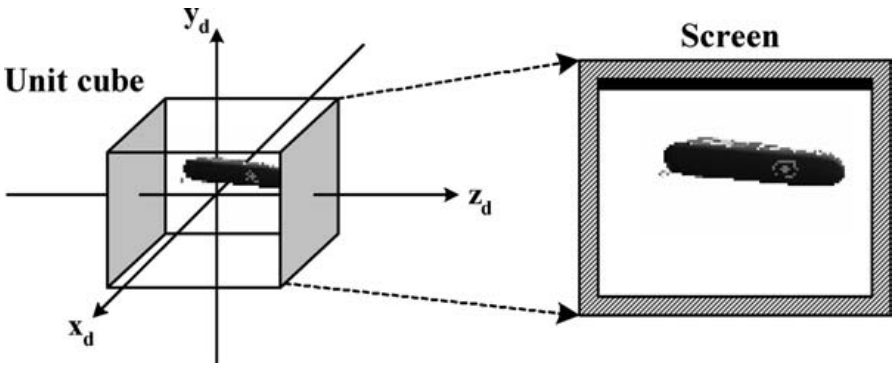


Figure 2.8: Device Coordinate Space [5]

2.1.8 THE RENDERING STAGE

Pixel-level operations take place in the device coordinate space in the rendering stage. Various pixel-level operations are performed, such as pixel rendering by Gouraud or Phong shading, depth testing, texture mapping, and several extra effects such as alpha blending and anti-aliasing[5][7].

2.2 FPGA ARCHITECTURE

The majority of FPGAs is SRAM-based and can therefore be programmed as easily as standard SRAM. The SRAM bits are coupled to configuration points in the FPGA (Figure 2.9 left) and controls whether or not a connection is made. This is normally accomplished by a pass gate structure (Figure 2.9 right) that turns the connection on or off depending on the logic value (True or False) supplied by the SRAM. Because they are SRAM based, FPGAs are volatile. As such, they must be programmed each time power is applied. This is normally accomplished with another part of the circuit that reloads the configuration bit stream, such as a PROM[11].

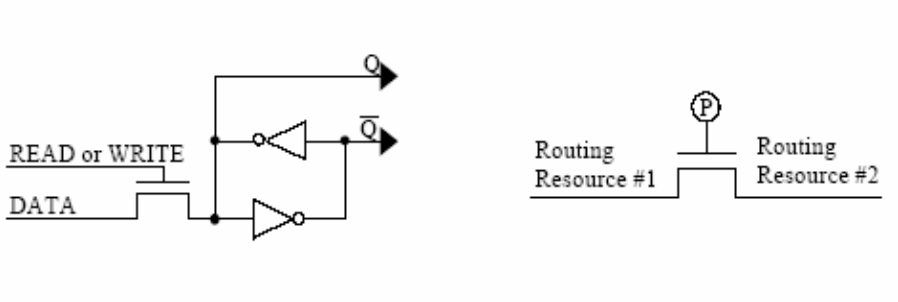


Figure 2.9: SRAM Based FPGA Configuration.[11]

The configuration bit stream stored in the SRAM controls the connections made and also the data to be stored in the Look-up tables (LUTs). The LUTs are essentially small memories that can compute arbitrary logic functions. Each manufacturer has a distinct name for their basic block, but the fundamental unit is the LUT. Altera call theirs a Logic Element (LE) while Xilinx's FPGAs have configurable logic blocks (CLBs) organized in an array. The configurable logic blocks of an FPGA are generally placed in an island style arrangement (Figure 2.10). Each logic block in the array is connected to routing resources controlled by a interconnect switch matrix.

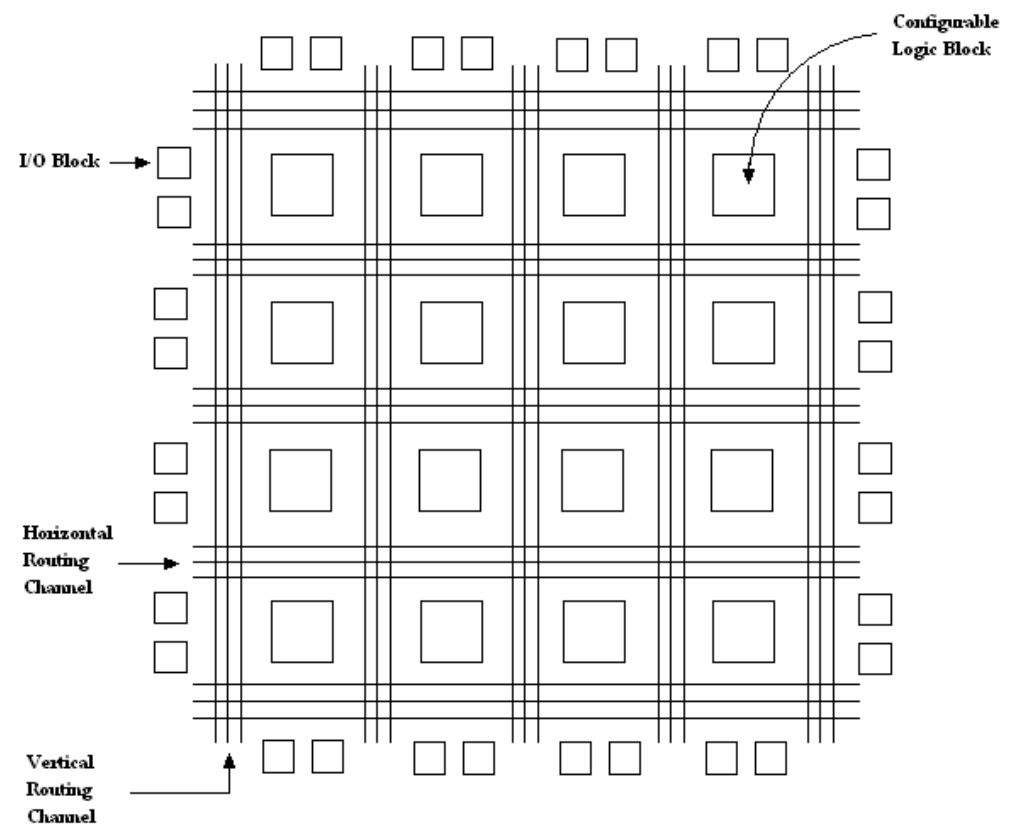


Figure 2.10: Generic Island Style Routing Architecture[11]

With this layout, a very large range of connections can be made between resources. A downside to this flexible routing structure is that unlike the CPLD, signal paths are not fixed beforehand, which can lead to unpredictable timing. However, the tradeoff is the FPGA's increased logic complexity and flexibility. Each CLB in a Xilinx FPGA encompasses four logic slices, which in turn contain two 4 input function generators, carry logic, arithmetic logic gates, wide function multiplexers and two storage elements the top half of a slice is shown in figure 2.11.

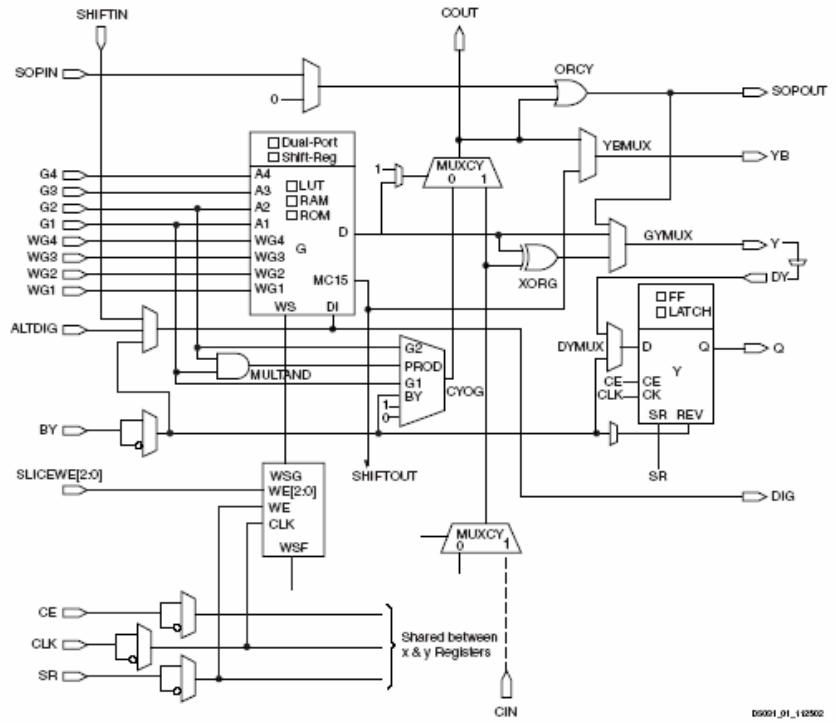


Figure 2.11: Virtex-II Pro Slice[11][7].

The LUT is capable of implementing any arbitrary defined Boolean function of four inputs and the propagation delay is therefore constant regardless of the function. Each slice also contains flip-flops and a fast carry chain. The dedicated fast carry logic allows the FPGA to realize very fast arithmetic circuits.

2.2.1 DEVICE CONFIGURATION

Manually defining the routing connections in a programmable device may have been feasible with the early PALs but is nearly impossible considering the density of modern FPGAs. Configuring these programmable devices can be achieved in several ways, such as schematic design entry, the use of hardware description languages (HDLs), and the use of high-level language compilers. These methods are listed in increasing levels of abstraction, with schematic design entry being the lowest level[11].

2.2.2 SCHEMATIC DESIGN ENTRY

Schematic design practices entails selecting standard logic gates from a library to create a graphic description of the circuit to be realized, and manually wiring them together. The schematic design library typically includes standard Boolean logic gates, multiplexers, I/O buffers, and macros for device specific functions, such as clock dividers. Custom components can be constructed from the smaller blocks to create user macros for use in large designs[7][11].

2.3.3 HARDWARE DESCRIPTION LANGUAGES

The most popular hardware description languages are Verilog and VHDL. Both are text-based depictions of the behavior of the digital circuit, and their syntax contains explicit notations for expressing time and concurrency. Gateway Design Automation Inc. started the Verilog language around 1984 as a proprietary hardware modeling language .The language went public in 1990 and has since been very popular in the semiconductor industry for ASIC and FPGA design. VHDL is a hardware description language that grew out of the VHSIC program sponsored by the Department of Defense and was first released in 1985[7][8].

2.2.4 HIGH LEVEL LANGUAGES

There is increasing interest in using high-level programming languages for FPGA design. Some, such as Celoxica's DK Design Suite, generate HDL from a C-like language. The Confluence language, based on Python, also takes this approach. The custom language is compiled to generate a VHDL or Verilog circuit description. The AccelFPGA tool from AccelChip similarly produces a register transfer level (RTL) circuit description from a Matlab m-file. An alternate approach is to generate the device netlist directly from the high-level description. This is what the Lava language, still under research by Xilinx and others, does. Lava is based on the lazy programming language Haskell, but is not yet available for system design. A shortcoming of the high-level design languages is their inability to instantiate vendor specific functions, such as block RAMs and DSP blocks. With this move toward incorporating further highly specific blocks, such as microprocessors, this shortcoming will need be overcome before any of these languages takes hold[11].

2.2.5 CURRENT TRENDS

The current trend in FPGA architectures is a move toward complete embedded systems. FPGA densities have increased to the point that entire RISC microprocessor soft cores can fit comfortably with additional logic on a single chip. Recognizing this trend, FPGA manufacturers are also including embedded block RAM and hard microprocessor cores in several of their new FPGAs. Altera's Excalibur device contains an ARM922T™ processor core whereas Xilinx's Virtex-II Pro contains up to four IBM Power PC microprocessors. This gives engineers the flexibility to mix hardware and software in embedded applications to achieve the maximum performance. The idea of integrating all the components of a computer system on a single chip is known a System-on-Chip (SoC). This includes the microprocessor, embedded RAM, and output interfaces such as UART or Ethernet MAC. FPGAs are highly attractive for this because the less common components can always be included as a soft core. Standard FPGAs will most likely be produced for a long time, with the dominating trend moving toward those including hard IP cores[11].

2.3 RELATED WORKS

3D graphics applications are steadily increasing; so that many researches have been done in this specific area; one of the works which has been done includes “3D Graphics Accelerator Platform for Mobile Devices” by Jung-Woo Kim et al [10]. They designed 3D graphics accelerator system which can do basic graphics processing activities such as shading , texture mapping , back and front culling Z-buffering and alpha blending and it is tested on FPGA board with standard 3D API and generally can be applicable for consumer electronics with 3d applications. The implementation of 3D accelerator system consists of ARM CPU, I/O systems and FPGA chips. It has a bus controller, SDRAM controller, and a 3D graphics accelerator with two FPGA chips. Figure 2.12 shows the designed architecture of 3d accelerator system

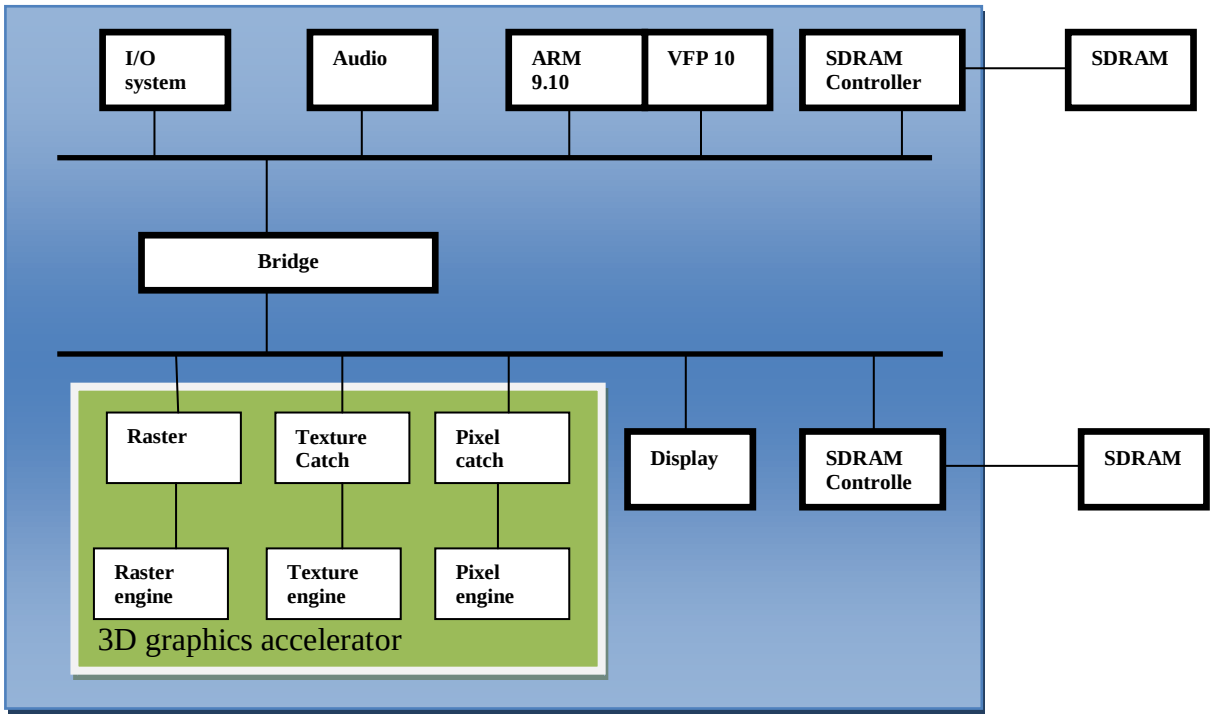


Figure 2.12: Architecture of 3D graphics accelerator platform[10]

It is implemented with two separate buses and SDRAM with each bus with its own arbitrators and SDRAM controller . In addition, it is possible to interchange a data between SDRAM s through the bridge. Bus separation helps to decrease bus delay time of 3D data because a heavy

bus traffic caused by 3D graphics accelerator is dispersed from other bus traffic caused system. The 3d graphics accelerator implemented on FPGA using Mesa 3d software library which was reduced to OPENGL ES specifications. The whole system was proposed and implemented on FPGA test board; It can accelerate the 3D rasterization and could reduce die size by using pixel cache and texture cache instead of embedded memory. Also the trade-off points such as the architecture of geometry and processing unit, internal memory or cache size and bus traffic distribution would be easily identified and evaluated with this platform. It could shorten the design and verification cycles of 3D graphics accelerator for mobile devices.

The system on FPGA operates at about 30 MHz, display board is consists of VGA size LCD screen and LCD controller. The demo application worded at 14-15 fps on the test board. The other work which is implemented similarly on FPGA with title "Design for Scalability in 3D Computer Graphics Architecture" by Hans Holten-Lund[14]. He presented the background for parallel 3D computer graphics architectures with a special focus on scalability. State of the art in current scalable commercial rendering architectures was discussed. From the available research it seems that a combination of parallel rendering techniques is a good method for achieving scalability. It used the Hybris graphic architecture around a primarily sort-middle architecture based on image-parallel subdivision of the screen into many small square tiles mapped to virtual local frame buffers. For each tile, bucket sorting and buffering work is used to load balanced the jobs across virtual processors. Each optimized for rendering one small square tile. In addition a partial sort-last architecture using object-parallel subdivision of the 3D model input data looks promising. The input data is split into many small sub-objects to distribute work over several geometry processors while maintaining data coherence. Finally sort-last is used to assemble image from tiles. Image composition of overlapping tiles might be useful in order to allow the architecture to scale even further, if correct handling of transparency is not an issue[14]. The rendering and VGA mapped on to a xilinx Virtex XCV1000 FPGA and operates reliably at a 25MHz clok frequency without pipelining of the datapaths. The system performance on FPGA is 1,087,716 triangles/s.

The most important and vast research work has been done in Korian Institute of Technology and published after ten years of work by the title " 3D Mobile Graphics from algorithm to Chip"

which discuss new architecture for 3d graphics for consumer electronics and implementation core designs and controllers using Verilog .[5]

Rendering operations such as rasterization and texture mapping dominate the 3D graphics pipeline, and require high memory bandwidth. Solving the bandwidth bottleneck with traditional approaches such as high-speed crossbars and off-chip DDR-SDRAMs can result in increased power consumption. However, the limited screen resolutions in mobile terminals (e.g., QVGA) imply that a reasonable amount of integrated memory, from a few tens of kilobytes to a few megabytes, is sufficient for graphics memories, depth buffer, frame buffer, and texture memory. In addition, by embedding all the required memory with the logic on a single die, external memory accesses are dramatically reduced, so we can develop more efficient architectures in terms of performance and power consumption.[5]

This was the first graphics processor to implement texture mapping in mobile devices. It focuses more on realtime 3D gaming applications, drawing bilinear MIPMAP texture-mapped pixels with special rendering effects such as fogging and motion blur at 66 Mpixels/s and 264 Mtexels/s, as well as supporting the shading operations. The performance and features of the rasterizer. It is designed for low-end mobile devices, so it targets 20 Mpixels/s fill rate at 10MHz operating frequency. It supports basic rendering functions: Gouraud shading, perspective correct bilinear texture filtering, and alpha blending.

Table 2.1: Performance and features summary of low-power rasterization unit

Screen Resolution	320x240
Color Depth	16-bit(Red: Green:Blue=5b:6b:5b)
Rendering Performances	20Mpixels/s pixel fill rate @10MHz
	80 Mtexels/s texel fill rate
	Two Pixel Processors
Shading feature	Gouraud Shading
	Pixel Alpha Blending
	Texture Blending (decal/modulate)
Z-buffer	16-bit embedded Z-buffer
Texture mapping	Perspective correct texture address generation
	Texture sampling : point sampling Bilinear filtering
	Maximum texture size:256x256 pixels
	Maximum number of texture =4
	Software controllable texture address calculation
	Efficient texture fetch through cache alignment
	Logic texture filtering
Operation frequency	10MHz
External SRAM capacity	1MB(256x32 bits)

Kyungsu Kim et al. at Electronics and Telecommunications Research Institute Daejeon, Korea have done research 3D graphics hardware with title “Implementation of 3D Graphics Accelerator Using Full Pipeline Scheme on FPGA” [22]. The research based on the graphics pipline Figure 2.13 which contains Geometry stage and Rasterization stage. Geometry stage consists of vertex shader, clipping engine and viewport mapping. Rasterization stage is composed of triangle setup engine, rasterizer, pixel shader and raster operators.

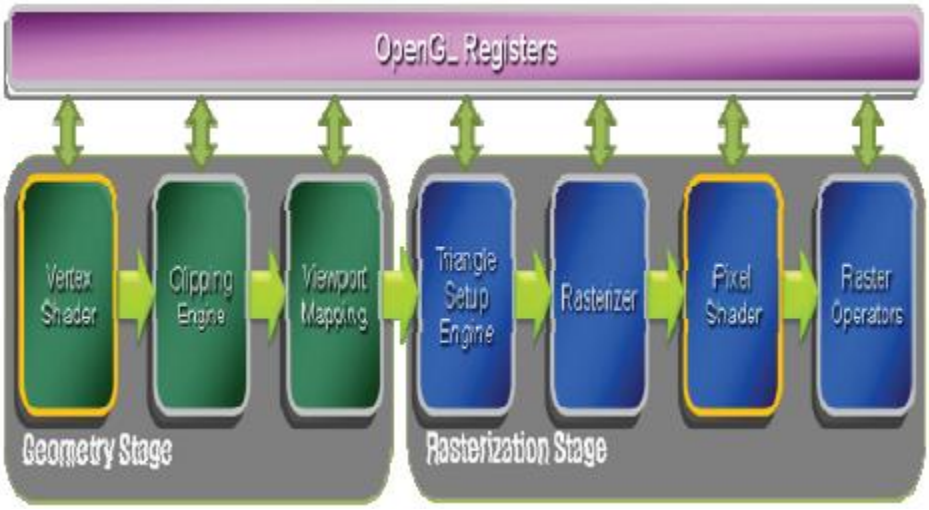


Figure 2.13: 3D graphics full pipeline[22]

The vertex shader is changeably increased the issue number of instruction according to the situation of the Data-Path so that the Vertex processing capability and optimization with satisfying the standard of the shader model 3.0. It is designed to the SIMD(single Instruction Multiple Data) Structure composed of the vector of the operator of 32 bit floating-point of 128 bit. Moreover, it is the design with 4 threaded structures in order to remove the latency of Data-Path.

The Triangle setup engine receives 3 vertexes and organizes a triangle. Figure 2.14 shows the block diagram of Triangle setup engine. The Triangle setup engine is positioned between the viewport Mapping and Rasterizer. The Triangle setup engine comprise the following parts

- Trivial X/ Y Clipping: when it is the triangle completely deviating from a screen or not determines.
- Face Culling: when it determines whether it is the back face whether it is the front face based on the normal vector of a triangle.
- An arrangement the vertex order based on Y Sorter: Y value.
- Start Point Decision: initial point crystal.
- Setup Parameter Calculation: setup parameter calculation.

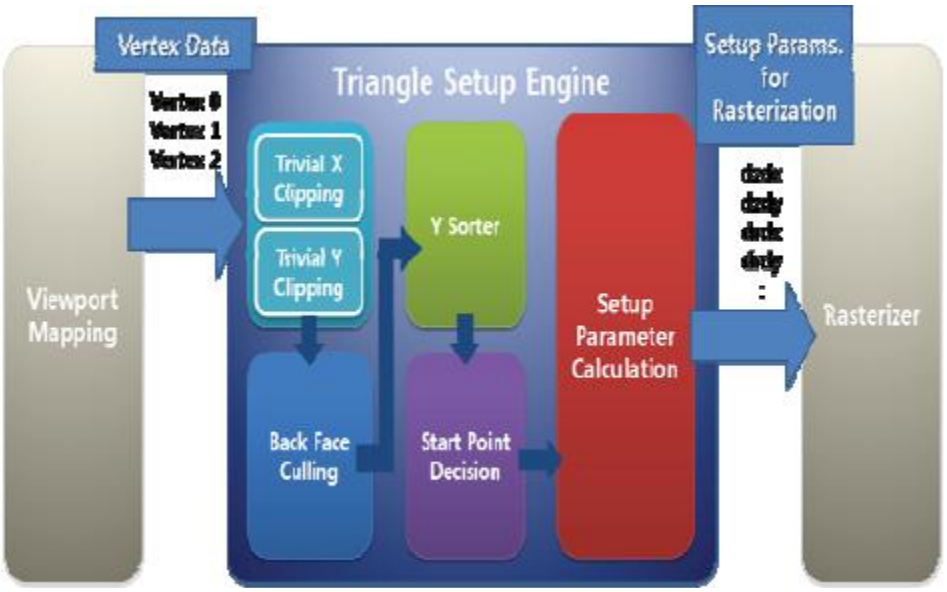


Figure 2.14: Triangle Setup Engine Block Diagram[22]

The Rasterizer carry out two operations in order to look for data filling the inside of a triangle of each pixel. The other one is the task calculating the color, a depth, and the coordinate about the pixels which is inside the triangle.

The research which is synthesized and implemented on Virtex 5 FPGA can operate with 35MHz clock frequency and able to render 70,000 triangles of Stanford Bunny at 30 frame/Sec. From the above related works even though much has been done in this particular field in based on S/W and H/W, FPGA based SoCs are not widely applicable in the area of 3D graphics.

Since the processing capability of those stated FPGA based research results still have some limitations in terms of processing performance based on vertex per second . Now a days, with the advance of FPGAs in speed and Area there is still a room to contribute in this field of area(i.e Realizing Systems on reconfigurable devices).

2.5 CHAPTER SUMMARY

In this chapter background concepts involved in the design of 3D graphics system such as view coordinate systems, world coordinate system, and projections were presented. Some of the basics

architectures to implement rendering on FPGA were seen in the background section. The architecture of FPGA systems also discussed in order to give some in site to understand some of applications done with reconfigurable systems. Further, Literatures related to the thesis were discussed and some of the results and the implementation architectures were put in the figures and tables.

In the chapter coming forth, the main components 3D graphics core will be designed based on the algorihthm chosen and the block diagram of each rendering pipeline component such as 3D transformation, Clipping,Projection and Rasterization will be discussed.

CHAPTER THREE

3D GRAPHIC PIPELINE SYSTEM AND DESIGN

3.1 GRAPHICS AND 3D TRANSFORMATIONS

Several software suites, graphics libraries and application programming interfaces (API) have been developed for graphic design and graphic animation purposes. Two of the well known graphics packages are open graphics library (OpenGL) and DirectX. These packages include several functions for creating computer graphics. It is possible to access functions provided by these graphics packages through most programming languages such as C/C++, C#, Java and Visual Basic. The first step of creating animation using these packages is to form 2D or 3D mathematical models of animation objects using vertices, edges and surfaces. Modeling even a simple animation object requires to define hundreds even thousands of vertices, edges, and surfaces. Figure 3.1 shows a sample animation model of famous Utah Teapot. Geometric transformations are an unavoidable part of the graphics packages. While generating animations, several 2D or 3D geometric transformations are perform on the mathematical models of the animation objects. There basic transformations are translation, rotation and scaling. While in some cases only one transformation is required, in most cases combination two or more transformation is applied to the object to create animation effects [16].

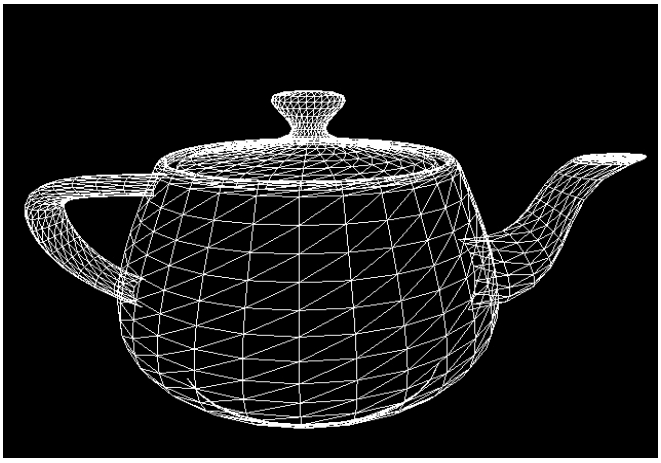


Figure 3.1: Sample 3D object definition [16][29].

When using three dimensional Cartesian coordinate system, the animation objects and scene are defined with three coordinate values (*x, y, z*). In cartesian coordinate system, 3D rotation or scaling operations of a single vertex requires multiplication of a 3x3 matrix and a 3x1 matrix while translation requires addition of a 3x3 matrix and a 3x1 matrix. Most of the time more than one transformations have to be applied to objects to obtain desired results. In such a case, combining all transformations in to one transformation matrix and then applying it to the objects is the desired solution. On the other hand, translation operation is not a linear operation and cannot be calculated through matrix multiplication. Moreover, it cannot be combined with other transformations. Homogenous coordinate representation of the objects is used to standardize all geometric transformations. In this representation, all transformations, applied to a single vertex, require multiplication of a 4x4 matrix and a 4x1 matrix. Homogenous representation also helps to combine more than one transformation in to one transformation matrix. While converting vertices defined in 3D Cartesian coordinate system (*x, y, z*) to homogeneous coordinate system, a fourth coordinate value, *w*, is added to the vertex and the vertex is defined as (*x, y, z, w*), ($w \neq 0$ should be satisfied). Usually $w = 1$ is selected and different *w* values cause scaling of the object while converting to homogeneous coordinate system (Figure 3.1). Below translation, rotation and scaling operations are given in parametric and matrix multiplication forms in Equations (3.1, 3.2, 3.3). In translation operation, *tx, ty,* and *tz* parameters define the amount of move of the object in each dimension, in rotation operation, q parameter defines the rotation angle, and in scaling operation *sx, sy,* and *sz* parameters define the scaling factors in each directions. *P* represents the original coordinate of the vertex and *P'* is the new coordinate of the vertex[29].

$$P' = T(t_x, t_y, t_z).P \qquad (3.1)$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R(\theta).P \qquad (3.2)$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = S(s_x, s_y, s_z).P \qquad (3.3)$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

As it can be seen from the above equations, all transformations require multiplication of 4x4 and 4x1 matrices. Usually these transformations are not applied to object uniquely. First, a combination of these transformations is formed as a new transformation matrix, and then, this new matrix is applied to the animation objects to reduce computational complexity. When these transformations are combined into a new matrix, the size of the matrix is again 4x4. As a result, combined transformations also require multiplication of a 4x4 and a 4x1 matrices. Graphics packages create animation effects, by applying above mentioned transformations on to mathematically defined objects. To create a simple camera move action, new coordinates of all objects in the scene have to be calculated and these calculations are done through matrix multiplication.

3.1.1 TRANSFORMATION MODULE DESIGN

In this research work, for 3D homogeneous transformation, a hardware module was designed to be used with FPGA based custom computing machines. The module is designed to multiply a constant 4x4 matrix with a series of 4x1 matrices and to produce a new series of 4x1 matrixes. The module is designed to comply with IEEE 754-1985 standard and to process 18-bit floating point data. It is clear that 18 bit floating width is not IEE 754 standard but it is chosen in order to use ZBT SRAM which has 36 bit wide data bus so that we can use it for color and primitive data values. The module design is coded in VHDL and mapped to Xilinx's *Virtex5* chip using Xilinx's ISE 13.1 ISE electronic design automation (EDA) tool. Here, details of the module design are presented. Since the module is designed as a data processor of ertexes, it has a block memory to hold each vertex for the coming processing. It reads data from one memory unit, processes the

data, and make ready to the next stage for after a cycle or more according to the design. Using 18-bit data boxes, the module is able to address 4 small block memory address space and process 18-bit floating-point data. For each memory unit, to synchronize read/write operations, the module produces separate memory control signals, which are used to control underflow overflow, and read/write. Reset, Start and Done signals are used for handshaking with the host. The module was designed in two parts which are the control unit and the data processing unit. The purpose of the control unit is to generate required control signal for both handshaking with the controlling unit and processing data. For handshaking purpose, the controller listens reset and start signals and it generates an interrupt signal. For data processing, the controller is responsible for generating control signals that go to both memory units and controller signals that coordinate data flow in the data processing unit. Details of the controller are given in the following section. The Data Processing Unit consists of block memory, adders, multipliers, and multiplexers, and can perform 4x4 and 4x1 matrix multiplication through parallel working multipliers and adders. This unit is also responsible for tracking source memory addresses. Block diagram of the data processing unit is shown in Figure 3.2. The Data Processing Unit is designed in two parts which are data access counters and core unit.

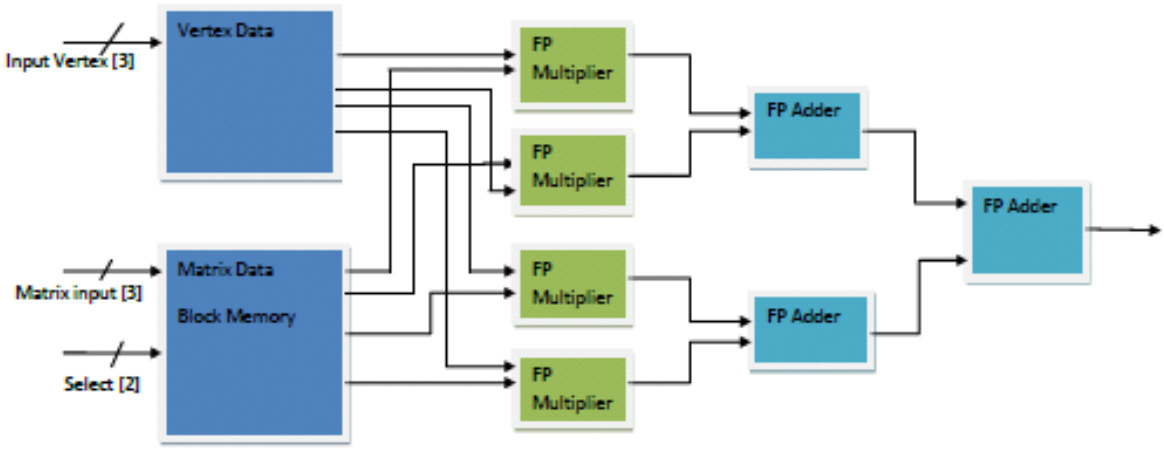


Figure 3.2: Matrix 4x4 Multiplier Blocks

Figure 3.2. Shows block diagram of the 4x4 multiplier core unit. The core unit consists of a transformation matrix block memory, a vertex block memory for floating-point multiplication units, and three floating-point addition units. As shown in Figure 3.2., each row of block memory contains four 18-bit loadable memories. Block memory is used to hold a row of values to be

fetched by the multiplier stage. Each row holds one column of the matrix. During a multiplication operation, through parallel working multiplexers in each register, rows of the transmission matrix are selected one by one and send to multipliers. Similarly the virtex data is taken from the FIFO which is part of the matrix transformer. A matrix transformer does have core matrix multiplier, vertex and matrix FIFO, and floating point division cores. Figure 3.2 shows the whole set up of a matrix transformer using block representation.

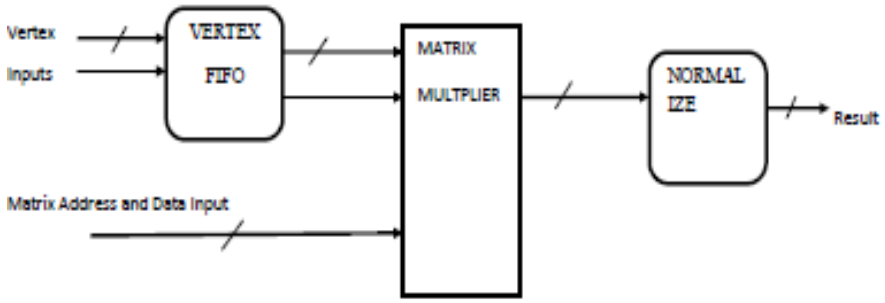


Figure 3.3: Matrix Transformation Unit Block Diagram

3.2 CLIPPING UNIT DESIGN

In the line clipping procedures if the line is not completely inside the clipping window then we have no option but to divide the line into segments at the intersections of the line and clipping window edges and then identify which segment is inside and which segment is outside the clipping window. But if the line is completely outside the clipping window, the line is eligible for total clipping or trivial rejection [1].

In the 2D clipping every line end-point is assigned a four-bit binary code called region outcode that identifies the location of the point relative to the boundaries of the clipping window rectangle.

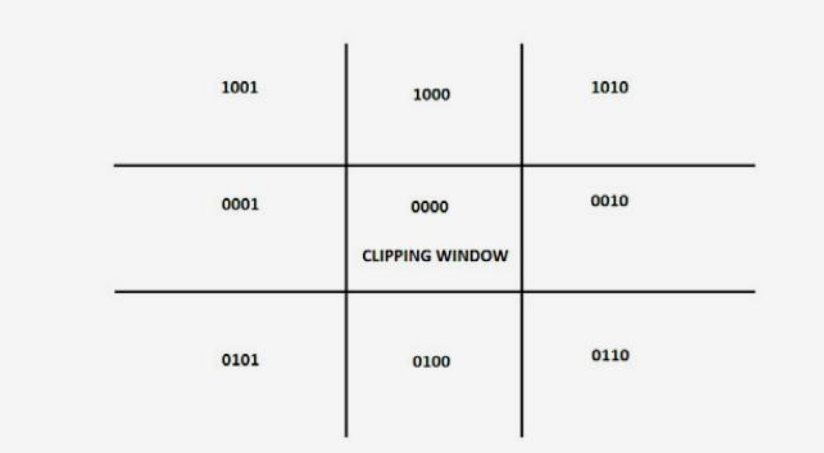


Figure 3.4 : Outcodes For Nine Regions[1][2]

Regions are set up in reference to the boundaries as shown in figure 3.4. Each bit position the region code is used to indicate one of the four relative co-ordinate positions of the point with respect to the clip window: to the left, right, top, or bottom. By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions can be correlated with the bit positions as bit 1 for left, bit 2 for right, bit 3 for below and bit 4 for above. A value of 1 in any bit position indicates that the point is in that relative position: otherwise, the bit position is set to 0. If a point is within the clipping rectangle the region code is 0000. Figure 3.5 shows the block diagrams for implementation of outcode generator for 3D vertexes, here floating point compare cores are used.

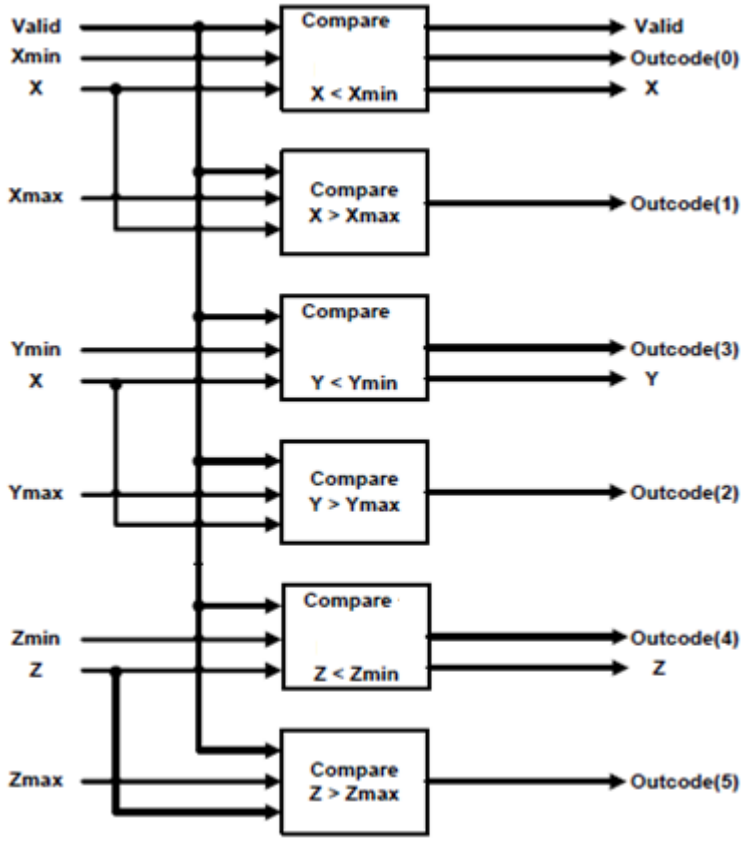


Figure3.5: outcode generator for clipping unit

3.2.1 INTERSECTION TESTS

Line intersection in 2D plane can occur at left, right, bottom and top clipping window edges as shown by figure 3.4. The equation of the line with end points (x1,y1) and (x2,y2) and slope m=(y2 –y1)/(x2-x1) is given by:

$$y-y1=m(x-x1) \qquad (3.4)$$

A rectangular clip window is taken with coordinates of the lower left, lower right, upper left and upper right corners as (Xwmin , Ywmin), (Xwmax,Ywmin),(Xwmin,Ywmax) and (Xwmax,Ywmax) respectively shown in the figure 3.5. After calculating the outcode for a single line end points if the line found out of clipping window its values reassigned again. For instance if one point(X1,Y1) of a line is at bottom of clipping window then new value assigned using one of the edge y=Ymin and x= (Ymin-Y1)/m +X1.

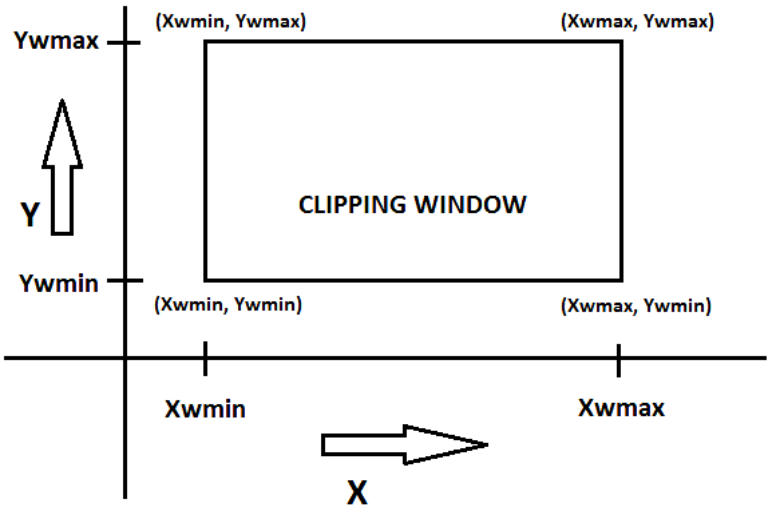


Figure 3.6: Coordinates Of Corners Of The Clipping Window[2]

3.2.2 THREE DIMENSIONAL CLIPPING

Clipping can be easily extended to 3D. Instead of a clipping window, in 3D a clipping volume is used. 3D clipping is different based on the projection type. In the case of parallel projection, the clipping volume is a 3D cubic.

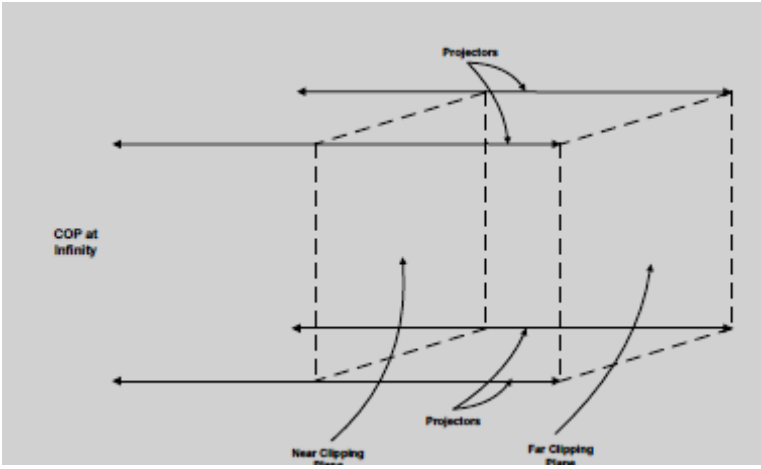


Figure 3.7: Parallel Projection[2]

In the case of perspective projection the clipping volume will be

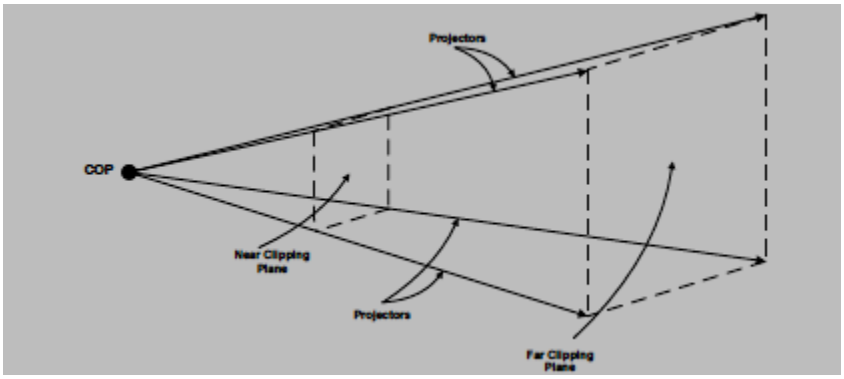


Figure 3.8 Perspective Projection[2]

Parallel projection clipping uses a six bit out code with a unit clipping cube. The parameters are shown below in Table 3.1

Table 3.1: 3D Parallel Projection Out code Assignment

Bit Number	**Location of End Point**	**condition**
First Bit	Above the Clipping Volume	If y> 1 then set bit to 1 else 0
Second Bit	Below the Clipping Volume	If y<-1 then set bit to 1 else 0
Third Bit	Right of Clipping Volume	If x> 1 then set bit to 1 else 0
Fourth Bit	Left of Clipping Volume	If x< -1 then set bit to 1 else 0
Fifth Bit	Behind the Clipping Volume	If z<- 1 then set bit to 1 else 0
Sixth Bit	In Front of the Clipping Volume	If z> 0 then set bit to 1 else 0

Perspective projection has a six bit out code which varies with the depth within the conical view volume, hence the z values within the conditionals.

Table 3.2: 3D Perspective Projection Outcode Assignment

Bit Number	**Location of End Point**	**condition**
First Bit	Above the Clipping Volume	If y> -z then set bit to 1 else 0
Second Bit	Below the Clipping Volume	If y< z then set bit to 1 else 0
Third Bit	Right of Clipping Volume	If x> -z then set bit to 1 else 0
Fourth Bit	Left of Clipping Volume	If x< z then set bit to 1 else 0
Fifth Bit	Behind the Clipping Volume	If z<-1 then set bit to 1 else 0
Sixth Bit	In Front of the Clipping Volume	If z> z_{min} then set bit to 1 else 0

A 2D line is trivially accepted if both endpoints have an outcode of all zeros and trivially rejected if the bit by bit logical AND of both points does not yield zero. 3D is no different except the clipping volume is defined by six planes as opposed to four edges and 27 unique sections exist as

opposed to nine. The intersection calculations for each of the six sides of the viewing volume can be found once again using parametric equations. Assuming a line from P0(x0,y0,z0) to P1(x1,y1,z1) the parametric equations is described as such:

$x = x_0 + t(x_1 - x_0)$

$y = y_0 + t(y_1 - y_0)$

$z = z_0 + t(z_1 - z_0)$ (3.5)

Solving for t on the extremes of the clipping volume yields six planar intersection equations. Below is a table which shows how to calculate the new x, y and z coordinates for a clipped line based on the plane intersected in both parallel and perspective projections[26].

Table 3.3: 3D Parallel Projection Clipping Intersection Equations.

Clip Edge	Solve for t	Planar Intersection Equations
y=1	$t = \frac{(1 - y_0)}{(y_1 - y_0)}$	$x = x_0 + \frac{(x_1 - x_0)(1 - y_0)}{(y_1 - y_0)}$ $z = z_0 + \frac{(z_1 - z_0)(1 - y_0)}{(z_1 - z_0)}$
y=-1	$t = \frac{(-1 - y_0)}{(y_1 - y_0)}$	$x = x_0 + \frac{(x_1 - x_0)(-1 - y_0)}{(y_1 - y_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-1 - y_0)}{(z_1 - z_0)}$
x=1	$t = \frac{(1 - x_0)}{(x_1 - x_0)}$	$y = y_0 + \frac{(y_1 - y_0)(1 - x_0)}{(x_1 - x_0)}$ $z = z_0 + \frac{(z_1 - z_0)(1 - x_0)}{(x_1 - x_0)}$
x=-1	$t = \frac{(-1 - x_0)}{(x_1 - x_0)}$	$y = y_0 + \frac{(y_1 - y_0)(-1 - x_0)}{(x_1 - x_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-1 - x_0)}{(x_1 - x_0)}$
z=1	$t = \frac{(-1 - z_0)}{(z_1 - z_0)}$	$x = x_0 + \frac{(x_1 - x_0)(-1 - z_0)}{(z_1 - z_0)}$ $y = y_0 + \frac{(y_1 - y_0)(-1 - z_0)}{(z_1 - z_0)}$
z=0	$t = \frac{-z_0}{(z_1 - z_0)}$	$y = y_0 + \frac{(y_1 - y_0)(-z_0)}{(z_1 - z_0)}$ $x = x_0 + \frac{(x_1 - x_0)(-z_0)}{(z_1 - z_0)}$

Table 3.4: 3D Perspective Projection Clipping Intersection Equations

Clip Edge	Solve for t	Planar Intersection Equations
y=-z	$t = \frac{(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$	$x = x_0 + \frac{(x_1 - x_0)(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-z_0 - y_0)}{(y_1 - y_0) + (z_1 - z_0)}$
y=z	$t = \frac{(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$	$x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}$
x=-z	$t = \frac{(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$	$y = y_0 + \frac{(y_1 - y_0)(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(-z_0 - x_0)}{(x_1 - x_0) + (z_1 - z_0)}$
x=z	$t = \frac{(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$	$y = y_0 + \frac{(y_1 - y_0)(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$ $z = z_0 + \frac{(z_1 - z_0)(z_0 - x_0)}{(x_1 - x_0) - (z_1 - z_0)}$
z=-1	$t = \frac{(-1 - z_0)}{(z_1 - z_0)}$	$x = x_0 + \frac{(x_1 - x_0)(-1 - z_0)}{(z_1 - z_0)}$ $y = y_0 + \frac{(y_1 - y_0)(-1 - z_0)}{(z_1 - z_0)}$
$z=z_{min}$	$t = \frac{(z_{min} - z_0)}{(z_1 - z_0)}$	$x = x_0 + \frac{(x_1 - x_0)(z_{min} - z_0)}{(z_1 - z_0)}$ $y = y_0 + \frac{(y_1 - y_0)(z_{min} - z_0)}{(z_1 - z_0)}$

In addition to all intersection equations being evaluated in parallel, the intersection equations above must be completely pipelined as to not impose bottleneck on the rest of the system. The intersection equations are composed of multiplications, addition and division floating point operations. Table 3.4 intersection equations can be calculated using the following system shown in the Figure 3.9.

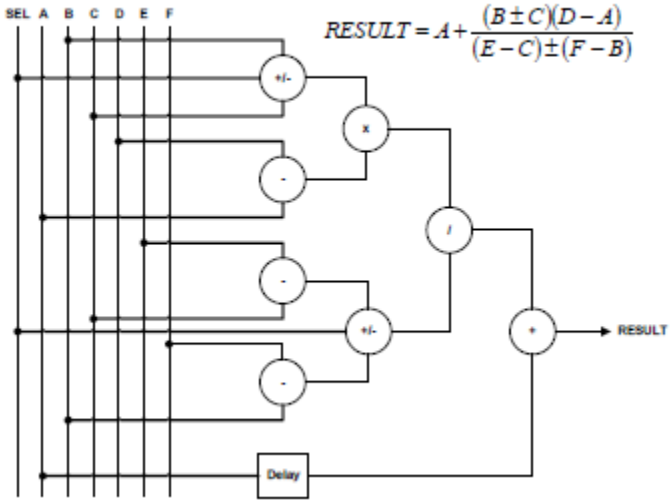


Figure 3.9: Edge Intersection Calculation

As an example the bottom edge equation intersection $x= x_0 + (x_1-x_0)(y_{min} – y_0)/(y_1-y_0)$ can be mapped to edge intersection calculator, the following substitutions $A=x_0$, $B=F=y_{min}$, $C=y_0$, $D=x_1$ and $E=y_1$ as well as selecting subtraction for the +/- blocks can be used to calculate the bottom edge intersection of a 2D viewing window . Two edge intersection calculators are needed for the 3D engine while only one is needed for 2D engine.

3.3 RASTERIZATION UNIT

Rasterization is the process by which a primitive is converted to a two-dimensional image. Each point of this image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which squares of an integer grid in window coordinates are occupied by the primitive. The second is assigning a color and a depth value to each such square. The results of this process are passed on to the next stage of the pipeline, which uses the information to update the appropriate locations in the frame buffer. Figure 3.10 diagrams the rasterization process. A grid square along with its parameters of assigned color, z (depth), and texture coordinates is called a fragment*;* the parameters are collectively dubbed the fragment's associated data. A fragment is located by its lower-left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's center, which is offset by (1/2 ,1/2) from its lower-left corner (and so lies on half-integer coordinates).[8]

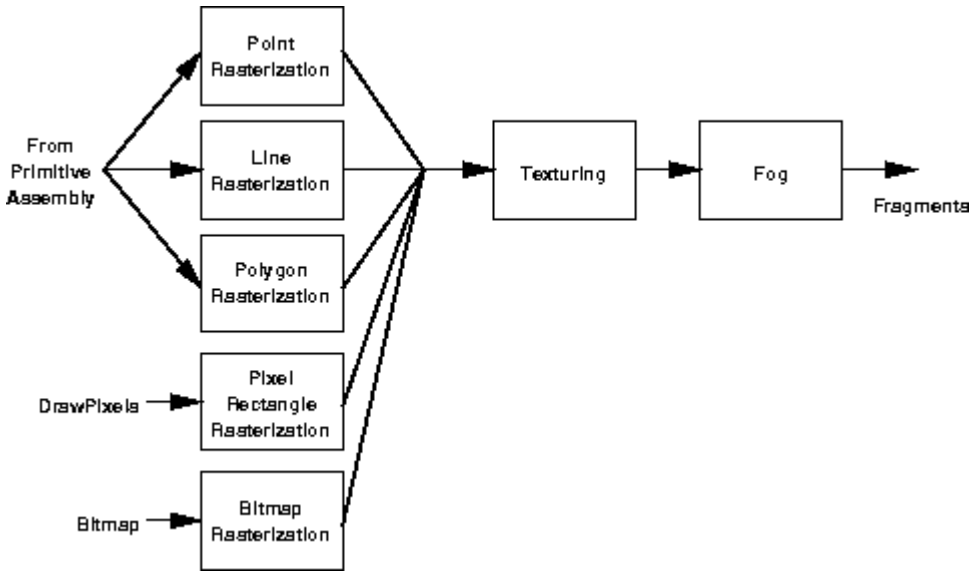


Figure 3.10: Rasterization in block Diagram [6]

From figure 3.10 it can be seen that rasterization has got different variants since we are working only wired frame objects line rasterization is the only option. The entire rasterization is written using VHDL.

3.3.1 LINE RASTERIZER ALGORITHM

Bresenham developed a classic algorithm, which uses only integer arithmetic. The choice of pixels is made by testing the sign of a Discriminator based on the Midpoint principle The Discriminator obeys a simple recursive strategy where the chosen pixel will be the closest to the true line. We assume that the slope of the line is between 0 and 1, where (X_1, Y_1) represents the lower-left endpoint and (X_2,Y_2) represents the upper-right endpoint[3].

Consider the line in Figure 3.11 where the previously selected pixel appears as black circle and the two pixels from which to choose at the next stage are shown as unfilled circles. Assume that we have just selected the pixel P at (X_p ,Y_p) and now must choose between the pixel one increment to right (called the east pixel, E) or the pixel one increment to right and one increment up (called the north-east pixel, NE). Let Q be the intersection point of the line being scan-converted with the grid line $X= X_p + 1$. In Bresenham's formulation, the difference between the vertical distances from E and NE Q is computed, and the sign of the difference is used to select the pixel whose distance from Q is smaller as the best approximation to the line. In the Midpoint formulation, we observe on which side of the line the Midpoint M lies. If M lies above the line, pixel E is closer to line, and if M lies below the line, pixel NE is closer to the line. The line may pass exactly between E and NE, or both pixels may lie on one side of the line. Also the error which is the vertical distance between the chosen pixel and the actual line is always less than a half [3].

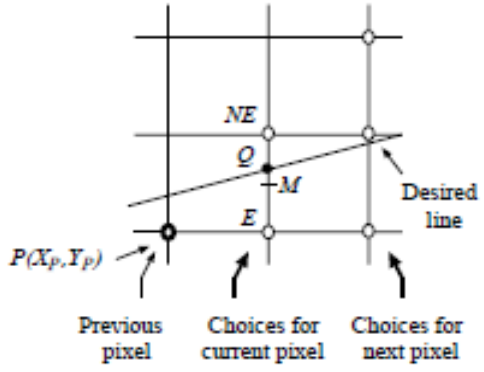


Figure 3.11: Pixel Grid for Bresenham's Midpoint Based Line Generator

Now all we need is a way to calculate on which side of the line M lays. Let us represent the line by an implicit function with coefficients a, b, and c :

$F(X,Y) = a.X + b.Y + c = 0.$ (3.5)

If $\Delta Y = Y_2 - Y_1$, and $\Delta X = X_2 - X_1$, the slope – intercept from can be written as

$$Y = \frac{\Delta Y}{\Delta X} . X + B$$

Therefore:

$F(X,Y) = \Delta Y.X - \Delta X.Y + B. \Delta X = 0,$ (3.6)

Here $a = \Delta Y$, $b = -\Delta X$ and $c = B. \Delta X$

It can easily be verified that F(X,Y) is zero on the line, positive for points below the line, and negative for points above the line. To apply the Midpoint criterion, we need only to compute $F(M) = F(X_p+1 , Y_p + 0.5)$, and to test its sign. Because our decision is based on the value of the function at $(X_p + 1, Y_p + 0.5)$, we define a decision variable $dv = F(X_p+1 , Y_p + 0.5)$.

By definition $dv = a.(X_p+1)+ b.(Y_p + 0.5) + c$,

If $dv \leq 0$, then pixel E is selected, M is incremented by one in X direction , and the next position we need to consider is $(X_p+2 , Y_p + 0.5)$. Here we have:

$Dv(E) = F(X_p+2 , Y_p + 0.5) = a.(X_p +1) + b.(Y_p + 0.5) + c + a = dv + a,$ (3.7)

Where we call the increment to add $\Delta E^- = a = \Delta Y$.

If $dv > 0$, then pixel NE is selected, M is incremented by one step in both X and Y coordinates, and the next position we need to consider is $(X_p +2 , Y_p + 1.5)$. Here we have:

$dv(NE)= F(X_p+2 , Y_p + 1.5) = a.(X_p + 1) + b.(Y_p +0.5) + c + a + b = dv +a +b$ (3.8)

where we call the increment to add $\Delta E^- = a + b = \Delta Y - \Delta X$.

Since (X_1,Y_1) is on the line, $F(X_1,Y_1) = 0$, so we can directly calculate the initial value of dv for choosing between E and NE. The fires midpoint $(X_p+1 , Y_p + 0.5)$ and :

$$F(X_p+1 , Y_p + 0.5) = F(X_1,Y_1) + a + b/2 = F(X_1,Y_1) +\Delta Y - \Delta X/2 = F(X_1,Y_1) + dv_{start} \quad (3.9)$$

Using $dv_{start,}$ we choose the second pixel, and so on. To eliminate the fraction in dv_{start}, we multiply the original function F(X,Y) (Equation 3.2) by 2:

$$F(X,Y) = 2.(a.X + b.Y + c) \quad (3.10)$$

This also multiplies the constants ΔE^- and ΔE^+ and the decision variable dv_{start} , without affecting its sign .

Bresenaham summarized the above formulation in to the following algorithm (note that the decision variable dv renamed to E):

BresenahamLineGenerator(X_1, Y_1, X_2, Y_2, I)

$\Delta X = X_2 - X_1$; $\Delta Y = Y_2 - Y_1$;

$E = -\Delta X$; $\Delta E^- = 2.\Delta Y$; $\Delta E^+ = 2(\Delta Y - \Delta X)$;

$Y = Y_1$;

For $X = X_1$ **to** X_2

If $E \leq 0$ **then** $E^+ = \Delta E^-$

Else $E^+ = \Delta E^+$; Y^{++};

Endif

Add Frame Buffer(X,Y,I)

Endfor

end

3.4 CHAPTER SUMMARY

In this chapter we have discussed 3d graphics components such as transformation , clipping reasterization with special focus to line based algorithms since the final core is supposed to process wired frame objects. Based graphic pipelining components the architecture which is implemented using VHDL has been covered. An introduction to general concepts in 3D graphics mathematical background has been given. Components such as transformation unit expressed designed block diagram and others such as rasterization and clipping stated based on chosen line algorithms (i.e Bresenahm line rasterizaion and cohen-sutherland line clipping respectively).

For clipping the components used to clip a line such as edge intersection and outcode generator are designed using block diagrams.

In the next chapter the conceptual , block diagrams and algorithms discussed in chapter three are behaviorally simulated and synthesized after writing VHDL codes and incorporating IP from Xilinx. In addition , the inputs and outputs of each top level units are stated .Furthermore , to synthesize the top level 3d graphic accelerator core pipeline components such as view translator , world translator , clipping ,projection , screen translator and rasterizer units are connected . Also the pipeline components connected with DVI controller for standard display, ZBT SRAM as a frame buffer and Microblaze Soft processor for future works that are going to interface it.

CHAPTER FOUR

4.1 GRAPHICS CORE SYNTHESIS AND TIMING SIMULATION

Figure 4.1 shows the graphic pipeline implemented on FPGA each block is described using VHDL with the help of Xilinx IPs and tested using ISE simulator. After simulating the blocks the system incorporated as custom IP on EDK and tested with fragments written using C language which is integral part of the soft core Xilinx microblaze. In this chapter the timing simulation of each block is described. Each part described by view transformation, world transformation, projection, screen transformation are represented by similar block matrix transformer their difference is the value of the matrix which is supposed to defined by the microprocessor.

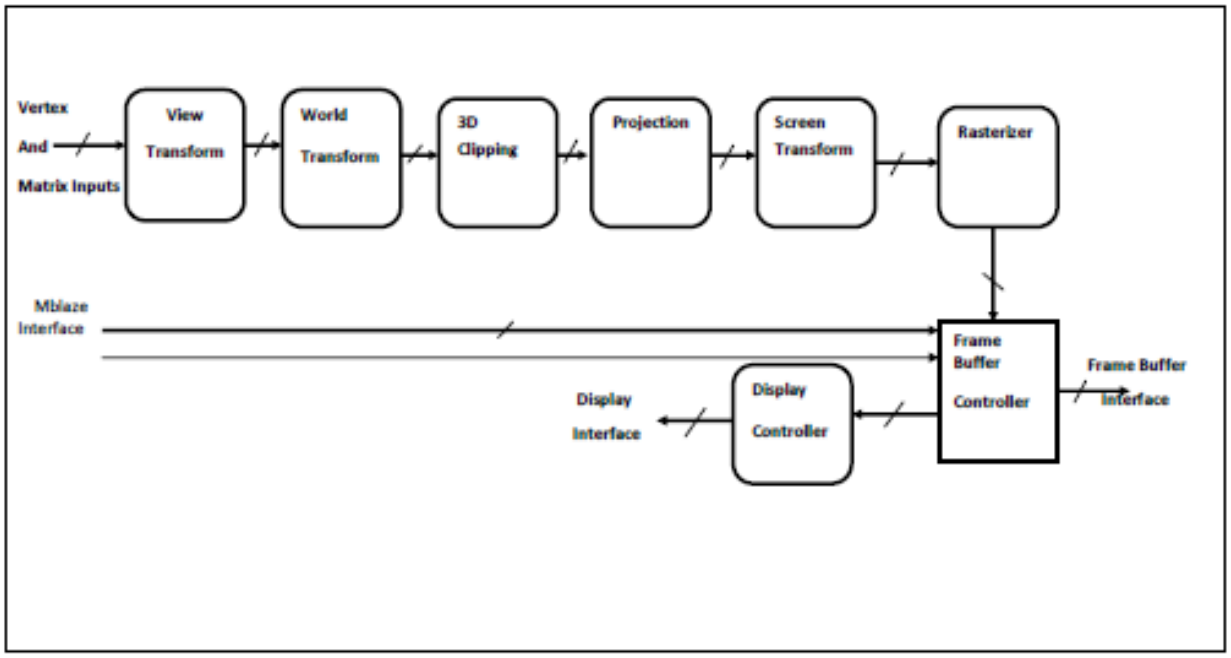


Figure 4.1: Top level blocks that define the 3D graphics accelerator

4.2 TRANSFORMATION UNIT SIMULATION

4.2.1 MATRIX MULTIPLIER

This unit is the main part of the graphics core accelerator since it is known that matrix multiplication requite intensive computational power and require much area inside fpga. This module accepts row of a matrix and vertexes by using different cycles it will multiply 4x4 matrix and it has a block ram to store vertices and matrix vectors.

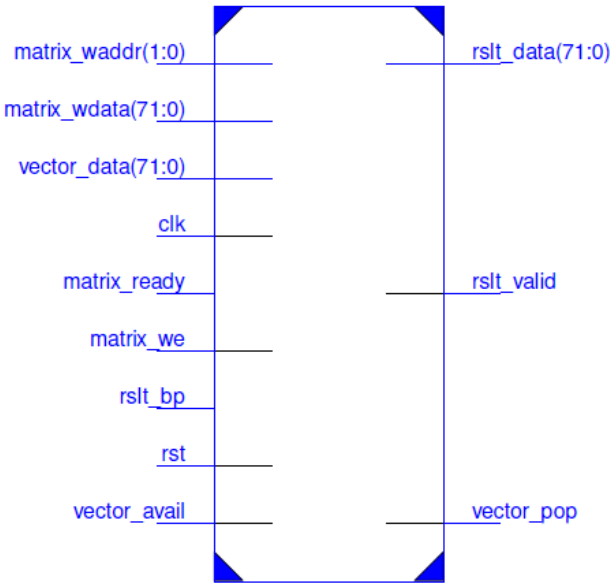


Figure 4.2 : Top module of Matrix multiplier

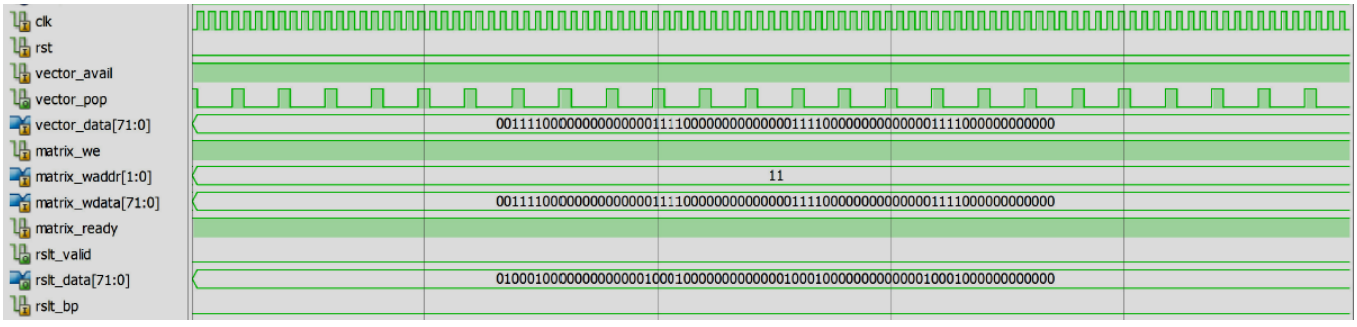


Figure 4.3 : Simulation of matrix multiplier

Figure 4.2 shows the Synthesized Matrix Multiplier with its input and output wires. Wires on the left side are inputs and the right side are outputs where ***matrix_wdata[71:0]*** is the row of a matrix (18x4 =72) , ***the vector_data[71:0]*** is the column data of four vertices (x,y,z ,w) , matrix_waddr is the addresses of four rows of the matrix . The other signals in the input side are control signals. From the output side ***rslt_data[71:0]*** shows the transformed versions of (x,y,z,w).

Figure 4.3 shows sample to show the timing diagram behaviorally each value of the row of a matrix set to ***001111000000000000 ,*** 18 bit representation of each matrix row element and the vertices as well set ***001111000000000000*** and the result shows the multiplication for each row and column . The simulation diagram shows fourth row.

4.2.2 MATRIX TRANSFORMATION

Matrix transformation unit contains matrix multiplier and fifo for pipe lining purpose and division module in order to normalize the homogenous 4X4 matrix. This unit is used for view transformation, screen transformation, projection transformation by setting the parameters of the matrix intended for specific job. Top level diagram of the module is shown below.

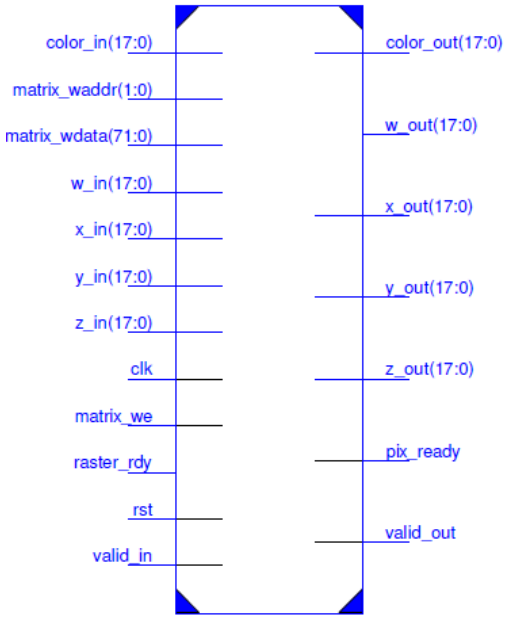


Figure 4.4 : Top RTL of Matrix transformation unit.

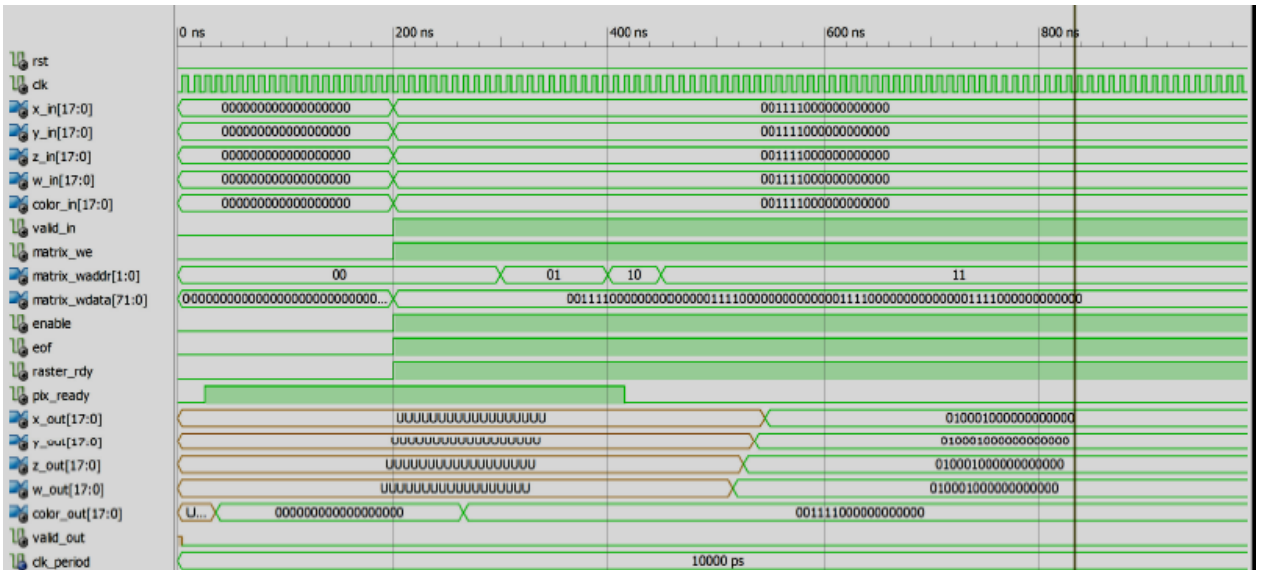


Figure 4.5 : Simulation of Matrix Transformation

Figure 4.4 shows the implementation of figure 3.3 using VHDL and the inputs are the matrix elements accesed by it row by row as it is shown form its RTL ***matrix_waddr [1:0]*** used to collect each of four rows , the vertices are given to the matix transformer by (***x[17:0],y[17:0] ,z[17:0] and w[17:0]***) additionally color of vertexes are also given with depth of 18 bits.

The behavioral simulation shows single vertex data and multiplying it unit matrix each value of the elements of matrix is ***001111000000000000*** and the result is multiplication of the vertexes. More over the output vertes(***x_out, y_out, z_out,w_out***) this is due to the processing time needed for multipliers and adders in the matix multiplier.

4.3 CLIPPING UNIT SIMULATION

4.3.1 CODE GENERATOR SUB UNIT

As stated in the previous chapter this sub unit purpose is to give the outcode for the position of a line in 3D space. Figure 4.6 shows the synthesized block of the outcode sub unit and figure 47 shows timing diagram for single end point line test.

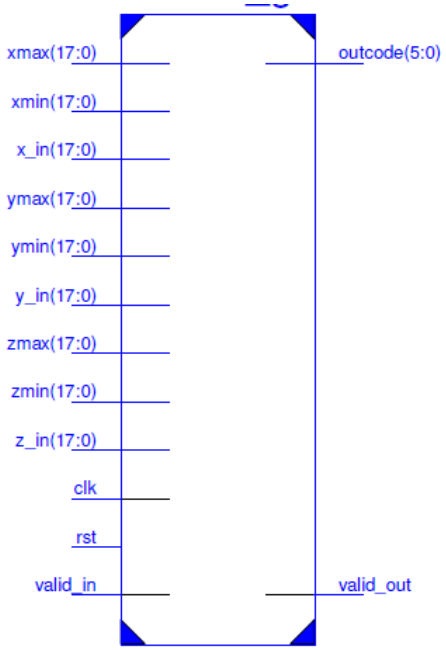


Figure 4.6 : RTL of Code generator for clipping decision

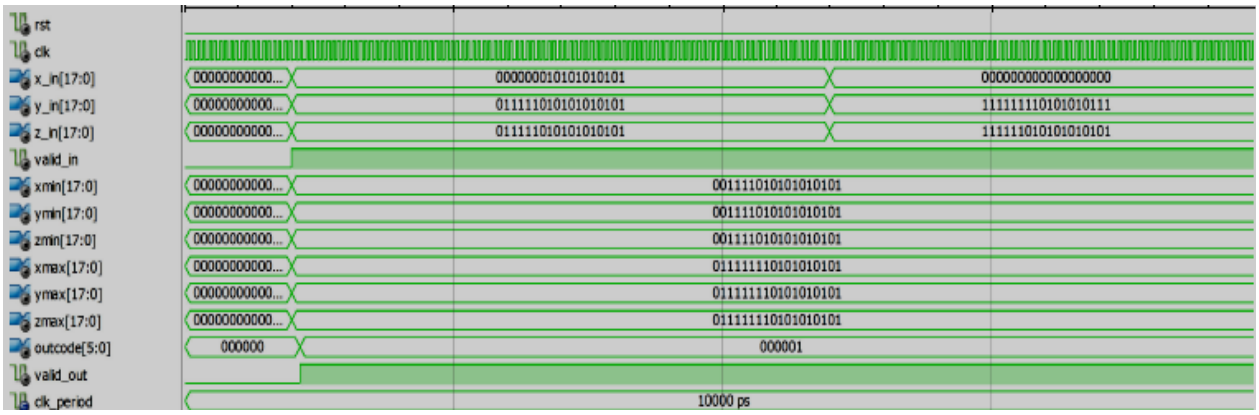


Figure 4.7: Code generator simulation

The figure 4.5 shows the code generator RTL , x_{min} ,x_{max} ,y_{min}, y_{max} ,z_{min} ,z_{max} define the clipping volume the values of ***x_in , y_in, z_in*** clipped checked against this volume and we get the result ***outcode[5:0].***

Figure 4.7 shows the simulation showing the vertex against the clipping volume which is set as shown from the values of the simulation and the result shows ***outcode[5:0]*** = 000001,the value of x_in is below x_{min} .

4.3.2 EDGE INTERSECTION SUB UNIT

This unit used to clip the lines with viewing volume edges which is calculated based on line edges . In addition segmentation of line is take place at this stage.

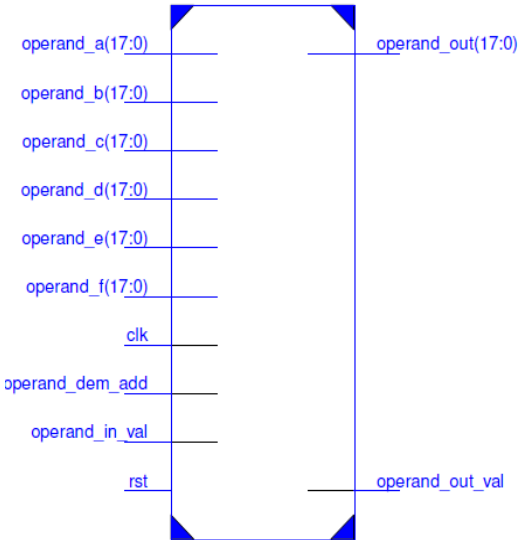


Figure 4.8 : edge intersection calculation

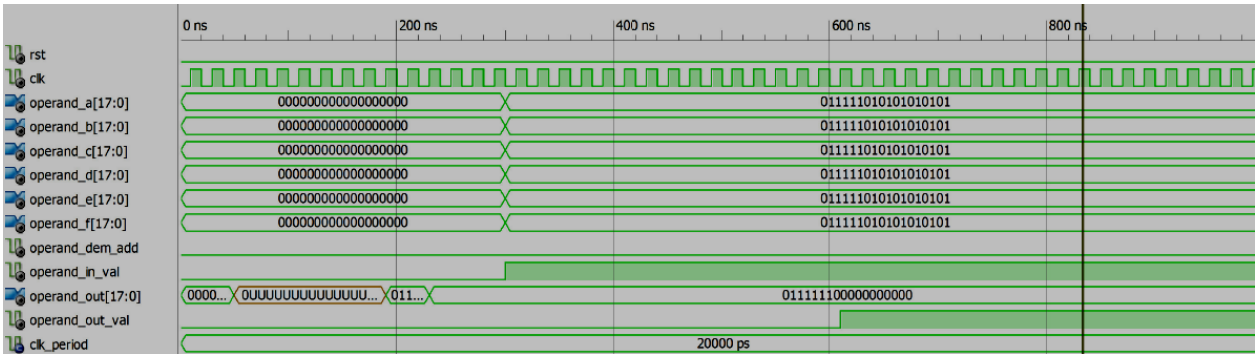


Figure 4.9: simulation of edge intersection calculation

Figure 4.8 shows the edge intersection calculator RTL based on figure 3.9 each point is used if a line is not in a clipping volume. To Check its functionality it is separately simulated as shown by figure 4.9 and the simulation shows the fife inputs used as discussed in table 3.4 planner intersections equation and finally the result is ***operand_out[17:0].***

4.3.3 THE TOP MODULE OF CLIPPING UNIT

The clipping unit ultimate function is to calculate the lines inside the viewing volume to be processed and displayed on after rasterization which is used to decrease processing time.

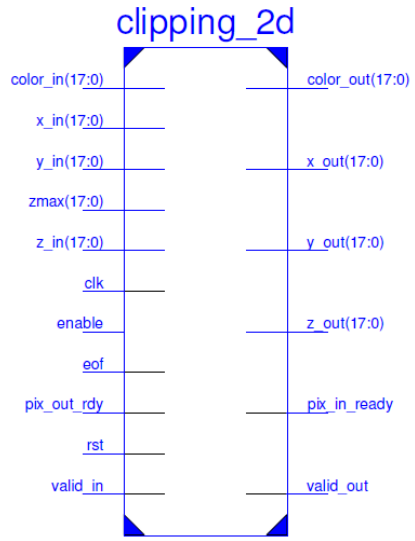


Figure 4.10 : RTL of Clipping Unit

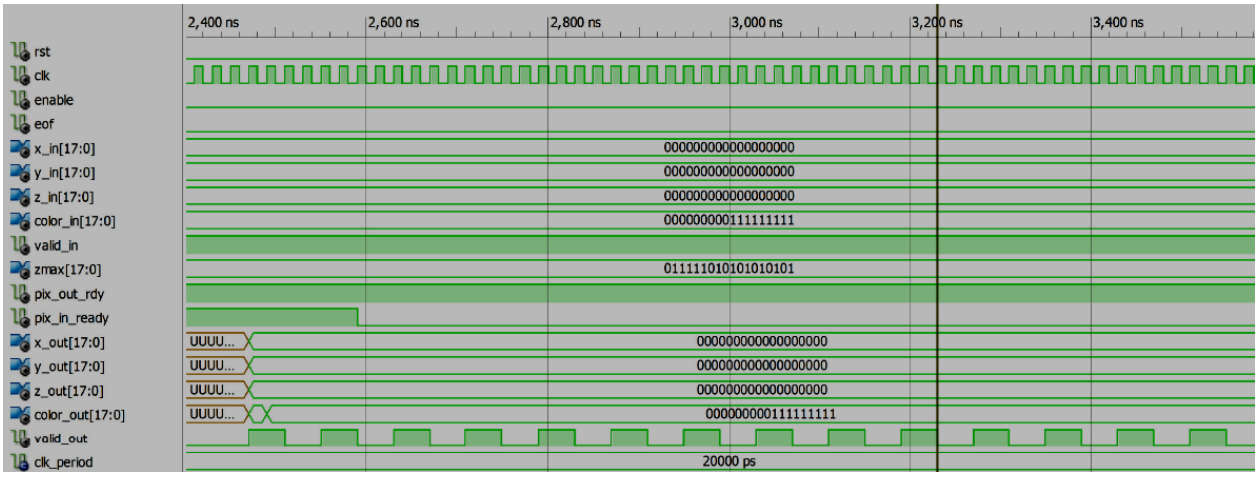


Figure 4.11: Simulation of Clipping Unit

Figure 4.11 shows the (***z_in,y_in,z_in,color_in***) inputs and **zmax** is set for clipping purpose then the output shows (***x_out,y_out,z_out , color_out***) for the next stage usage projection , screen translation and rasterization.

4.4 RASTERIZATION UNIT SIMULATION

The RTL level shown by figure 4.12 is the line rasterizer module it accept two integer points and by using line algorithm it finally map virtual lines in to pixels on 2D screen. The description of each input and output wires describe at by the table 4.1. And the timing simulation , figure 4.13 shows the data , input signals , status signals when a single frame data is processed for 320X240 resolution display , which contains 76800 pixels to be processed for single frame .

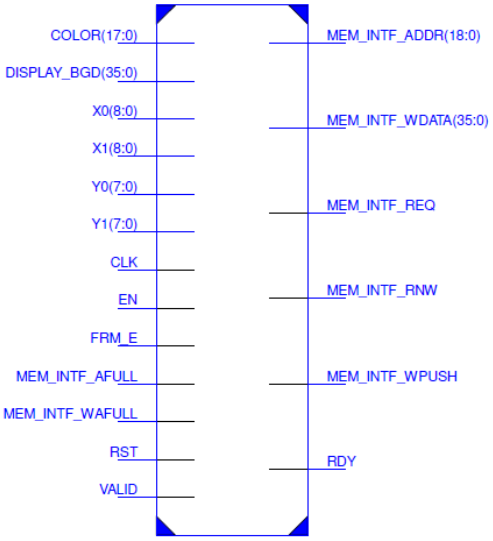


Figure 4.12: RTL of top level rasterizer unit

Table 4.1 Input and output signal description of Line Rasterizer Module

Name of signal	Direction	Width	Description
clk	Input	1	Clk of the system >100M
rst	Input	1	Reset rasterizer to initial positon and clear screen
color	Input	18	Clor of each pixel defined enclosed by end points
x0	Input	9	X values to represent a virtual line
x1	Input	9	X values to represent a virtual line
y0	Input	8	Y values to represent a virtual line
y1	Input	8	Y values to represent a virtual line
display_bgd	Input	36	Background color data since the background color and pixel color should be different
en	Input	1	Control signal to enable writing on 2d screen
frm_e	Input	1	Frame status signal indicate end of it
mem_intf_afull	Input	1	Fram buffer memory status signal almost full
mem_intf_wafull	Input	1	Frame buffer memory status almost data written

			almost full
valid	Input	1	Status signal pixel is valid or not
mem_intf_addr	Output	19	Wires connected to frame buffer memory
mem_intf_wdata	Output	36	Data to frame buffer memory
mem_intf_req	Output	1	Control signal to control graphics per frame basis
mem_intf_rnw	Output	1	Control signal to drow frame to 2d screen
mem_intf_wpush	Output	1	Control siganl to push data to frame buffer
rdy	Output	1	Frame processed and ready to be written to frame buffer

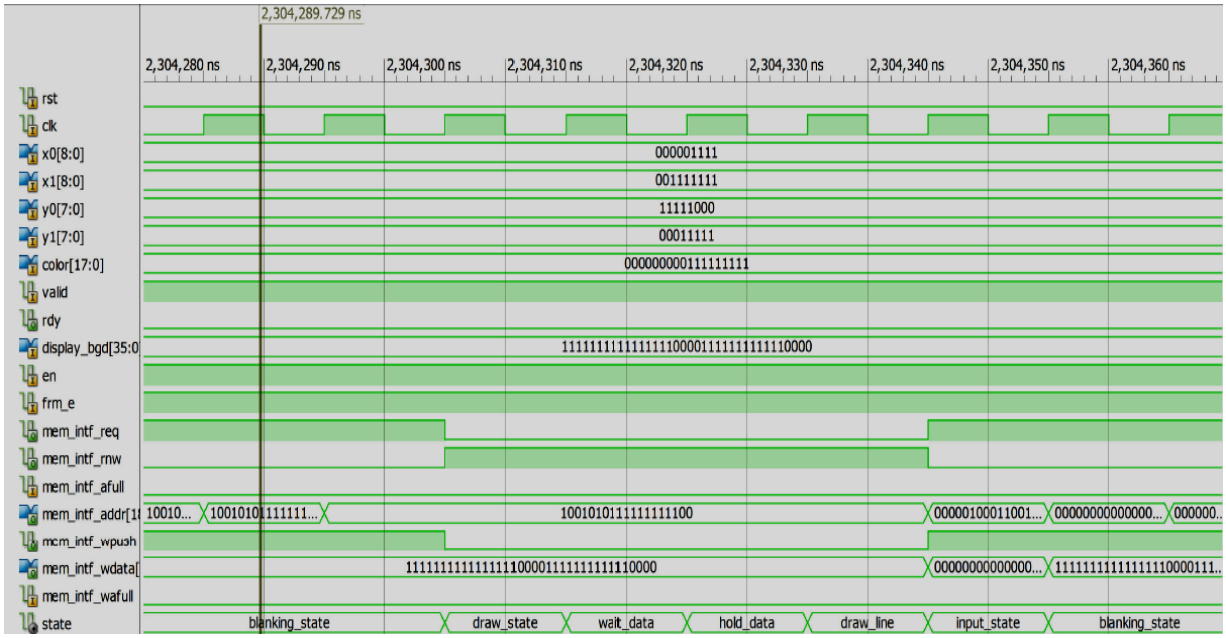


Figure 4.13: Rasterizer Unit Simulation.

Figure 4.12 shows the RTL of rasterizer module and table 4.1 describes its I/O . Figure 4.13 shows the simulation of a single line by pushing line end points (x_0,y_0) and (x_1,y_1) in to the rasterizer and rasterize points to pixels that are saved in the frame buffer . The states on the timing diagram shows rasterizer uses FIFO after it gets full and the state machine start to draw pixel per frame basis.

4.5 3D GRAPHIC ACCELERATOR TOP MODULE

As shown form the RTL of the 3D Graphic Accelerator Top Module , the core has interface with display , ZBT SRAM Interface with Microblaze Soft Processor Cores and control units. The memory interface uses ZBT-SRAM as a frame buffer since the ZBTs have been developed to be used in applications where fast memory writing and reading are required. The input side additionally shows the primitive values and fragment parameters such as color of vertexes and background.

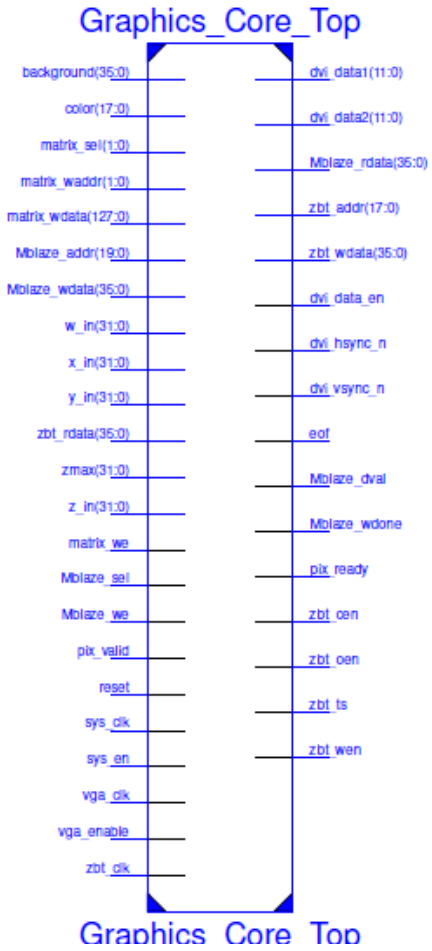


Figure 4.14: 3D Graphic Accelerator Top Module

Table 4.2: Inputs Descriptions of 3D Graphics Accelerator Core

Name of signal	Width	Description
background	36	Display back ground color data interfaced with the rasterizer
color	18	Primitive color to be processed with each point
mblaze_addr	20	Address of mircroblaze interface with zbt_sram
mblaze_wdata	36	Data interface of microblaze with zbt_sram
matrix_sel	2	Input used for selection projection, view , screen, world matrix
matrix_waddr	2	Data selection of each a row matrix.
matrix_wdata	128	A row matrix data .
w_in	32	Value for matrix homoginity
x_in	32	Primitive vertex x value
y_in	32	Primitive vertex y value
zbt_rdata	36	Read data of zbt_sram
zmax	32	Maximum value of z used for clipping.
z_in	32	Primitive vertex z value
mbaze_sel	1	Select
mbalze_we	1	Write enable for microblaze processor
sys_en	1	Enable signal for the top system
matrix_we	1	Enable singnal for matrix write enable
pix_valid	1	Used for control valid inputs .
reset	1	Set value of signals to initial value.
sys_clk	1	Clock for the top system
vga_clk	1	Separate clock for vga interface
vga_enable	1	Enable singal for vga
zbt_clk	1	Separate clock for frame reading and writing (zbt_sram)

Table 4.3: Outputs Descripitons of 3D Graphics Accelerator Core

Signal	Width	Description
mblaze_rdata	36	Read data Interface with processor
mblaze_dval	1	Microblaze data control
mblaze_done	1	Status processing is done
dvi_data1	12	DVI data port
dvi_data2	12	DVI data port
dvi_data_en	1	DVI data enable
dvi_hsync_n	1	DVI horizontal synchronizer
dvi_vsync_n	1	DVI vertical synchronizer
eof	1	End of frame status
pix_ready	1	Pixel ready status
zbt_addr	18	ZBT address port
zbt_wdata	36	ZBT data port
zbt_cen	1	ZBT chip enable
zbt_oen	1	ZBT output enable
zbt_ts	1	Sleep Mode enable
zbt_wen	1	ZBT write enable

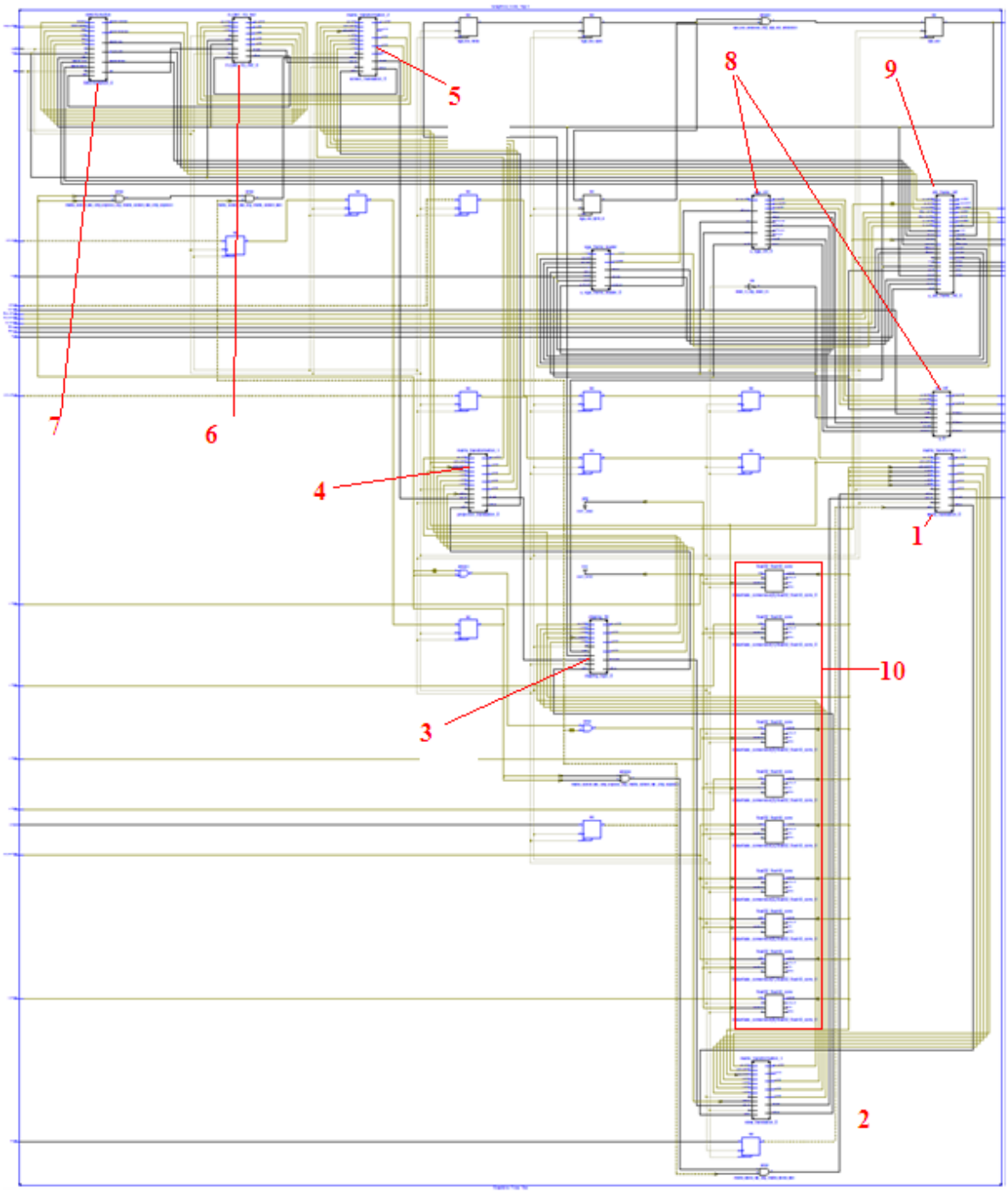


Figure 4.15: RTL view of Top Core Components

As can be seen from figure 4.1 the top core is made up of from components that are stated in this chapter. Additionally, figure 4.15 shows the RTL view of the interconnected components that make up 3D graphics accelerator core .

The blocks described as follows

1. World Translation
2. View Translation
3. Clipping
4. Projection
5. Screen Translation
6. Float to Integer Converter
7. Rasterizer
8. Display Controllers for DVI Interface
9. ZBT SRAM Controller
10. 32-Bit float to 18 bit converters

4.7 CHAPTER SUMMARY

In this Chapter the basic building blocks of the 3D graphics accelerating system are synthesized after simulating each component behaviorally and the VHDL codes used to realize these components can be referred in the appendix. The timing simulations are based on stimulus from test benches. The test benches are prepared for each main component and written in VHDL. In addition, the simulator used to run test bench coder is Xilinx Integrated simulator (ISIM).

After writing a VHDL codes each block the top level initiate each of them as a component.

Finally, the top level synthesized RTL diagram and its components are discussed in this chapter.

CHAPTER FIVE

RESULTS AND DISCUSSION

5.1 MATRIX MULTIPLIER UNIT

In this module there are 16 multiplication stages and 12 addition stages, since there is only one row and column multiplier, 16 cycles are required for the 18 bit floating point operations each multiplier core has built in delays which can be set at IP core parameters with my choice to be 4 cycles. Similarly each addition stages has got 4 cycles delay as a consequence the matrix multiplier needs 23 clock cycles in order to get result of operation. From synthesis of matrix multiplier the maximum frequency of the module found to be 309 Mhz with this speed the matrix multiplier can process 309/23 = 13.43 Million vertex transformation per second. Assuming VGA interface which has a standard 60Hz frame rate, 224,000 vertexes can be processed for each frame. Since the matrix multiplier is the basic unit for all transformations its performance is optimum for this application as the above result.

Table 5.1 : Device Utilization of Matrix Multiplier Unit

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2350	69120	3%
Number of Slice LUTs	1706	69120	2%
Number of fully used LUT-FF pairs	1469	2587	56%
Number of bonded IOBs	226	640	35%
Number of Block RAM/FIFO	0	148	0%
Number of BUFG/BUFGCTRLs	1	32	3%

5.2TRANSFORMATION UNIT

The main unit for the transformation units of view translation, world translation, screen translation, and projection is matrix multiplier which its result shown at 5.2 the additional component for this module is the input FIFO which hold primitives for pipelining purpose. This unit can work up to a maximum frequency of 309MHz but the synthesis shows more delays than the matrix multiplier 23 for matrix multiplier and additional 3 cycles a total of 26 , therefore , the unit can process 309/26 =11.88 Million vertex translation for world translation per second .

Table 5.2: Device Utilization each World, View, Screen Translations and Projection Module

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	2616	69120	3%
Number of Slice LUTs	1836	69120	2%
Number of fully used LUT-FF pairs	1518	2934	51%
Number of bonded IOBs	261	640	40%
Number of Block RAM/FIFO	0	148	0%
Number of BUFG/BUFGCTRLs	1	32	3%

5.3 CLIPPING UNIT

This unit has a maximum operating clock frequency of 176 MHz and from synthesis and behavioral simulation it has a delay of 30 cycles so that , its processing performance is 176/30 = 5.67 Million vertexes per second which is its ultimate maximum performance and its device utilization is shown in the table 5.3

Table 5.3: Device Utilization of Clipping Module

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	6051	69120	8%
Number of Slice LUTs	6305	69120	9%
Number of fully used LUT-FF pairs	3753	2587	43%
Number of bonded IOBs	169	640	26%
Number of Block RAM/FIFO	0	148	0%
Number of BUFG/BUFGCTRLs	1	32	3%

5.4RASTERIZER UNIT

This unit finally transfers vertexes to 2D display and based on balance automatic synthesis on Xilinx XC5VLX110T-1ff1136 has a maximum operating frequency 170 MHz and its takes 10 cycles after the taking data from full FIFO. Therefore, it has a capacity to fill convert 17 Million vertexes per second to pixels on to the display and the device utilization on the FPGA is given in Table 5.4

Table 5.4: Device Utilization Rasterizer Module

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	844	69120	3%
Number of Slice LUTs	678	69120	<1%
Number of fully used LUT-FF pairs	212	1310	16%
Number of bonded IOBs	154	640	24%
Number of Block RAM/FIFO	0	148	0%
Number of BUFG/BUFGCTRLs	1	32	3%

5.5GRAPHIC ACCELERATOR TOP MODULE PERFORMANCE

This project top file Synthesized without errors on Target device XC5VLX110T-1F1136 Viretex5 with balanced design goal and automated synthesis used; therefore, speed and area optimization for FPGA is balanced. From timing summary of the top module with the FPGA device parameter set to its maximum speed grade which -1. It is found that the minimum period of the system clock is 5.897ns (maximum clock frequency: 169.578MHz) and minimum input arrival time before clock is 3.838ns without no path with maximum combinational path delay. After Behavioral Simulation initial verification was done using a VHDL test bench on ISIM 13.1 to check that the Top Module was functionally correct. After doing this the design was synthesized on XC5VLX110T-1F1136 FPGA. Its performance was found to be able to process geometry primitives up to 10million polygons per second. And the design area and device utilization is summarized below.

Table 5.5: Device Utilization of the 3D Graphic Accelerator module

Logic Utilization	Used	Available	Utilization
Number of Slice Registers	20598	69120	29%
Number of Slice LUTs	16707	69120	24%
Number of fully used LUT-FF pairs	11238	26067	43%
Number of bonded IOBs	573	640	89%
Number of Block RAM/FIFO	50	148	33%
Number of BUFG/BUFGCTRLs	4	32	12%

5.6 CHAPTER SUMMARY

In this chapter the result of basic components such as transformation unit, clipping unit, and rasterization unit were discussed in terms of vertex per second processing capability. In addition, the areas used by each component on Xilinx XUP Virtex5 XC5VLX-1F1136 were put in tables. Moreover, after synthesizing the system and analyzing summary report from Xilinx ISE built in synthesizer the number of clock cycles needed to perform transformation, clipping and rasterization presented. And the maximum frequency for components as well as the top core is stated in MHz and the top core performance was also stated in pixel fill rate and area on Xilinx XUP Virtex5 FPGA device.

. CHAPTER SIX

CONCLUSION AND FUTURE WORK

6.1. CONCLUSION

As it was stated in the objective above, the thesis has tried to implement 3D graphics pipeline in order to assist CPU and decrease the overhead associated with it by processing graphics primitives separately and accelerate the process by separate hardwire. The designed graphics core can process 3d wired frame objects in virtual world on FPGA which has necessary interface with display units and ZBT-SRAM that acts as frame buffer .

To fulfill the objectives main parts of the graphics pipeline implemented using VHDL on Xilinx 13.1 such as geometric transforms, line rasterizer algorithms, clipping algorithm. Each algorithm was selected based on the importance regarding line processing since the implementation did not incorporated shading and texture mapping.

After designing and incorporating necessary Xilinx free IP Cores such as floating point multipliers, adders, dividers, compare and FIFO cores each block was simulated behaviorally. After simulation of necessary pipeline parts, the system was synthesized on FPGA. Some of the performance parameters such as vertex/sec were drawn after obtaining the synthesis report of each block from Xilinx synthesizer.

The Top module can work at a maximum frequency of 169.578Mhz and process more than 100M pixels per second. In addition, the design take less than 50% of the area on XC5VLX110T-1f1136 FPGA so that there is huge area to implement other important 3D graphics algorithms such as hidden surface removal, shading, texture mapping, alpha blending etc.

6.2.FUTURE WORK

From the above conclusion XC5VLX110T-F1136 has got enough area to continue working on other 3d graphics pipeline components so that full 3d graphics processor can further be implemented on FPGA. The other important features that can make 3d graphics accelerator system more advanced includes

- Incorporating soft embedded processors to interface with the 3d graphics processor such as micro blaze which is 32-bit RISC processor based on Xilinx BSB tools. Additionally, the H/W and S/W parts can be implemented in EDK and SDK respectively by adding the designed core as a component to the system.
- Increasing the clock speed and performance of clipping unit by using parallel implementation of the unit on hardware.
- Continuing evaluation of open source IP-cores from Open Cores, Xilinx and from any other sources is always feasible and can contribute to the development of FPGA based systems like 3D graphics accelerators.

REFERENCES:

[1] Jhon Vince , Mathemathics for Computer graphics ,2nd Edition , Springer - Verlag London Limited ,2006

[2] Foley, van Dam, Feiner, Hughes , Computer Graphics Principles and Practice , 2nd Edition , Addison Wesley Publishing Company,1990

[3] Ali Mohamed Ali Abbas, "Transformation of Rendering Algorithms For Hardware Implementations", Phd. Thesis , Faculty of Electrical Engineering and Informatics Budapest University of Technology and Economics

[4] Niklas Knutsson , FPGA -based 3D Graphics System, Master's thesis in Electronic Systems , Linkoping 2006

[5] Jeong-Ho Woo, Ju-Ho Sohn , Byeong-Gyu Nam , Hoi-Jun Yoo , Mobile 3D Graphics From Algorithm to Chip , Korea Advanced Institute of Science and Technology ,2010

[6] http://www. icculus.org/manticore/Open Source 3D Graphics Accelerator,2001

[7] Hartmut F.W Sadrozinski, Jinyuan Wu, Applications of FPGAs in Scientific Research, Taylor and Francis Group, 2011

[8] Iosif Antiochi, Suitability of Tile-based Rendering for Low-Power 3D Graphics Accelerators, Technical university of Delft , 2007

[9] Kenneth Wiliam Taylor, The Design and Implementation of a 3D Graphics Pipeline for the Raw Reconfigurable Architecture, Massachusetts Institute of Technology, 2004

[10] Jung_Woo Kim et al, 3D Graphics Accelerator Platform for Mobile Devices, IEEE conference on Field-Programable Technology, Pages: 387-390 , 2004

[11] Jean-Pierre Deschamps, Gery Jean ,Gustavo D.Sutter ,Synthesis of Arithimetic Circuits ,FPGA ,ASIC, and Embedded Systems ,John Wiley & Sons Publication ,2006

[12] Claudio Brunelli , Design of Hardware Accelerators for Embedded Multimedia Applications , Tampre University of Technology , Phd Thesis , 2008

[13] Ruei-Ting Gu et al , A Low Cost Tile-Based 3D Graphics Full Pipeline with Real-time Performance Monitoring Support for OpenGL ES in Consumer Electronics , IEEE International Symposium On Consumer Electronics ,Pages :1-6 ,2007

[14] Hans Holten-Lund , Design and Scalability in 3D Computer Graphics Architectures , Phd Thesis , Computer Science and Technology Informatics and Mathematical Modelling Technical University of Denmark , 2001

[15] F.Bensaali,A.Amira and A.Bouridane, Accelerating Matrix Product on Reconfigurable Hardware for Image Processing Applications, IEEE Proceedings, Vol.152 , Issue:03,Pages:236 - 246, 2005

[16] F.Bensaali,A.Amira, An FPGA Based Coprocessor for 3D Affine Transformations, Electronics , Circuits and Systems IEEE International Conference, Vol.02 ,Pages: 715-718, 2003

[17] Tulika Mitra, An FPGA Implementation of Triangle Mesh Decompression, Field-Programmable Custom Computing Machines 10th Annual IEEE Symposium, Pages:22-31,2002

[18] Zemcik P.,Herout A. Particle Rendering Engine in DSP and FPGA, Engineering of Computer-Based Systems 11th IEEE International Conference and Workshop , pages:361-368,2004

[19] Mateusz Majer, et al. Co-Design Architecture and Implementation for Point-Based Rendering on FPGA, Rapid System Prototyping 19th IEEE International Symposium, Pages:142-148,2008

[20] Chanho Lee, Eunmin Kim, Design of a Geometry Engine for Mobile 3D graphics, SoC Design Conference , Vol.01, Pages: 222-225,2008

[21] "An Effective Pixel Rasterization Pipeline Architecture for 3D Rendering Processors,IEEE Transactions on Computers, Vol. 52, No. II,pp. 1501-1508, Nov. 2003

[22] Kyungsu Kim, Hoosung-Lee Seonghyum Cho, Seongmo Park , Implementation of 3D Graphics Accelerator Using Full Pipeline Scheme on FPGA , SoC Design Conference , Vol. 02 ,Pages:97-100, 2008

[23] Fabio Garzia, Claudio Brunelli, Implementation of a Floating-Point Matrix-Vector Multiplication on Reconfigurable Architecture, Parallel and Distributed Processing IEEE International Symposium ,Pages:1-6,2008

[24] Zdenka Safarzik et al, Implementation of Division-Free Perspective-Correct Rendering Optimized for FPGA Devices, Proceedings of 33rd International Convention,Pages:177-182,2010

[25] Michael Steffen, Phillip Jones, and Joseph Zambreno, Teaching Graphics Processing and Architecture Using a Hardware Prototyping Approach, Microelectronics Systems Education IEEE International Conference ,Pages:13-16 , 2011

[26] James Ryan Warner , Real Time 3-D Graphics Processing Hardware Design Using FPGA , MSc Thesis , Swanson School of Engineering , 2008

[27] J.Fender , Design and Implementation of a Hardware Accelerated Raytracer Using TM3a FPGA Prototyping System ,MSc Thesis , Faculty of Applied Science and Engineering University of Toronto , 2002

[28] Daniel Ramiro Humbrto , Arthimetic Soft Cores , Institute of Technology Di Costarica , 2007

[29] Ibrahim Sahin , A 32-bit floating-point module design for 3D graphic transformations ,Scientific Research and Essays Vol.5(20) , pp.3070-3061, 2010

[30] David H.Eberly ,3D Game Engine Design , A Practical Approach to Real-Time Computer Graphics ,Morgan Kaufmann publishers ,2004

[31] Benjamin Thomas Cope, Video Processing Acceleration Using Reconfigurable Logic and Graphics , Processors , Phd thesis , Imperial College London , 2008

[32] Sven Woop , A Ray Tracing Hardware Architecture for Dynamic Scenes , MSc Thesis , University of Saarlands , 2004

[33] Jag Mohan Singh, Real Time Rendering of Implicit Surfaces on The GPU, MSc Thesis , International Institute of Information Technology Hyderabad ,2008

[34] Anthony Edward Nelson , Implementation of Image Processing Algorithms on FPGA Hardware , MSc Thesis , Graduate School of Vanderbilt University , 2000

[35] Embeded Processor Block in Virtex-5 FPGAs, Reference Guide, 2010

[36] Dnald Hearn , M. Paulin Baker , Computer Graphics , 2nd Edition , 2002

[37] Jung- Woo Kim, Jae-One Oh, Cheol-Ho Jeong and Jue-Hyun Kim, 3D Graphics Accelerator Platform for Mobile Devices, Digital Media R&D Centel; Samsung Electronics, 2003

[38]Ruei-Ting Gu, Tse-Chen Yeh, Wei-Sheng Hunag, A Low Cost Tile-based 3D Graphics Full Pipelinewith Real-time Performance Monitoring Support for OpenGL ES in Consumer Electronics

[39] Yanru Ma, Xuzhi Wang, et al, Rasterization of Geometric Primitive in Graphics Based on FPGA, Audio Language and Image Processing International Conference, Pages:1211-1216,2010

[40] Peter Szanto, Bela Feher, Scalable Rasterizer Unit, Budapest University of Technology Department of Measurement and Information Systems,2006

[41] Pavel Zemcik, Hardware Acceleration of Graphics and Imaging Algorithms Using FPGAs, Department of Computer Graphics and Multimedia, Faculty of Information Technology, Brno University of Technology, 2006

[42] Design and implementation of an FPGA-based Parallel Graphics Renderer for Displaying CGS Surfaces and Volumes, Computers & Electrical Engineering ,Vol.30,Issue 2, Pages: 97-117, March 2004

[43] Martin Whit, et al. The TAYRA 3D Graphics Raster Processor, Computer & Graphics , Vol.21, Issue 2, Pages:129-142, April 1997

[44] Rafael J.Segura, Francisco R.Feito An Algorithm for Determining Intersection Segment-Polygon in 3D, Computers & Graphics, Vol.22,Issue 5, Pages:587-592 , October 1998

[45] Xilinx, Inc. Vertex-5 FPGA Datasheets. Visited Online 2011, http://www.xilinx.com

[46] Xilinx, Inc. ML510 Resources, Visited Online 2011, http://www.xilinx.com

[47] Xilinx Inc. Coregen IP. Generator Documentation. Visisted Online 2011, http://www.xilinx.com

[48] Xilinx Inc. , Floating-Point Operator v8.1. Visited Online 2011, http://www.xilinx.com

[49] Xilinx Inc. Microblaze Processor Reference Guide. Visited Online 2011, http://www.xilinx.com

[50] http://www.opencores.org/projects/2D Graphic Accelerator , 2011

APPENDIX A

Sample VHDL codes for Matrix Transformer .

```
1 -- Matrix Transformation Module
2 -- Used for view Translation , World Translation , Screen Translation
3
4 library ieee;
5 use ieee.std_logic_1164.all;
6 use ieee.std_logic_arith.all;
7 use ieee.std_logic_unsigned.all;
8
9 entity matrix_transformation is generic
10 ( NORMALIZE : integer := 0;
11 MATRIX_MULT_LATENCY : integer := 4;
12 LATENCY : integer := 16;
13 );
14 port (
15 -- Reset/Clock
16 rst : in std_logic;
17 clk : in std_logic;
18 -- Incomint points (18 bit floating point).
19 x_in : in std_logic_vector(17 downto 0);
20 y_in : in std_logic_vector(17 downto 0);
21 z_in : in std_logic_vector(17 downto 0);
22 w_in : in std_logic_vector(17 downto 0);
23 color_in : in std_logic_vector(17 downto 0);
24 valid_in : in std_logic;
25 pix_ready : out std_logic;
26 -- Matrix Access (18 bit floating point).
27 matrix_we : in std_logic;
28 matrix_waddr : in std_logic_vector(1 downto 0);
29 matrix_wdata : in std_logic_vector(71 downto 0);
30
31
32 -- Output Integers, a line or edge.
33 x_out : out std_logic_vector(17 downto 0);
34 y_out : out std_logic_vector(17 downto 0);
35 z_out : out std_logic_vector(17 downto 0);
36 w_out : out std_logic_vector(17 downto 0);
37 color_out : out std_logic_vector(17 downto 0);
38 valid_out : out std_logic;
39 raster_rdy: in std_logic );
40 end matrix_transformation;
41
42 architecture hdl of matrix_transformation is
43 component matrix_multiplier port(
44 -- Clock, rst and enable signals
45 clk : in std_logic;
46 rst : in std_logic;
47 -- Vector Inputs.
48 vector_avail : in std_logic;
49 vector_pop : out std_logic;
50 vector_data : in std_logic_vector(18*4-1 downto 0);
```

```
51 -- Matrix Inputs.
52 matrix_we : in std_logic;
53 matrix_waddr : in std_logic_vector(1 downto 0);
54 matrix_wdata : in std_logic_vector(18*4-1 downto 0);
55 matrix_ready : in std_logic;
56 -- Matrix Outputs.
57 rslt_valid : out std_logic;
58 rslt_data : out std_logic_vector(18*4-1 downto 0);
59 rslt_bp : in std_logic ); end component;
60 -- Floating point Division.
61 component float18_div is port (
62 a : in std_logic_vector(17 downto 0);
63 b : in std_logic_vector(17 downto 0);
64 operation_nd : in std_logic;
65 operation_rfd : out std_logic;
66 clk : in std_logic;
67 result : out std_logic_vector(17 downto 0);
68 underflow : out std_logic;
69 overflow : out std_logic;
70 invalid_op : out std_logic;
71 divide_by_zero : out std_logic;
72 rdy : out std_logic );
73 end component;
74 -- Single fifo clk.
75 component fifo_1clk_trans is
76
77 port (
78 -- Clock and rst
79 rst : in std_logic;
80 clk : in std_logic;
81 -- Control signals
82 wr_en : in std_logic;
83 rd_en : in std_logic;
84 -- Read write data
85 din : in std_logic_vector(89 downto 0);
86 dout : out std_logic_vector(89 downto 0);
87 -- Status flags.
88 almost_full : out std_logic;
89 almost_empty : out std_logic;
90 empty : out std_logic;
91 full : out std_logic
92 );
93 end component;
94 signal vector_pop : std_logic;
95 signal vector_data : std_logic_vector(18*4-1 downto 0);
96 signal vector_avail : std_logic;
97 signal f_push : std_logic;
98 signal f_pop : std_logic;
99 signal f_wdata : std_logic_vector(18*5-1 downto 0);
100 signal f_rdata : std_logic_vector(18*5-1 downto 0);
101 signal f_afull : std_logic;
102 signal f_aempty : std_logic;
103 signal f_empty : std_logic;
104 signal f_full : std_logic;
105 type color_pipe_dly_t is array (0 to MATRIX_MULT_LATENCY-1) of
std_logic_vector(17downto 0);
```

```
106 signal color_pipe_dly : color_pipe_dly_t;
107 signal divx_rslt : std_logic_vector(17 downto 0);
108 signal divx_underflow : std_logic;
109 signal divx_overflow : std_logic;
110 signal divx_invalid_op : std_logic;
111 signal divx_divide_by_zero: std_logic;
112 signal divx_rdy : std_logic;
113 signal divy_rslt : std_logic_vector(17 downto 0);
114 signal divy_underflow : std_logic;
115 signal divy_overflow : std_logic;
116 signal divy_invalid_op : std_logic;
117 signal divy_divide_by_zero: std_logic;
118 signal divy_rdy : std_logic;
119 signal divz_rslt : std_logic_vector(17 downto 0);
120 signal divz_underflow : std_logic;
121 signal divz_overflow : std_logic;
122 signal divz_invalid_op : std_logic;
123 signal divz_divide_by_zero: std_logic;
124 signal divz_rdy : std_logic;
125 signal divw_rslt : std_logic_vector(17 downto 0);
126 signal divc_rslt : std_logic_vector(17 downto 0);
127 type divc_pipe_dly_t is array (0 to LATENCY-1) of std_logic_vector(17
downto 0);
128 signal divc_pipe_dly : divc_pipe_dly_t;
129 signal rslt_valid : std_logic;
130 signal rslt_data : std_logic_vector(18*4-1 downto 0);
131 signal rslt_bp : std_logic;
132 begin
133 -- Handle input fifo.
134 pix_ready <= (not f_afull);
135 f_push <= valid_in;
136 f_wdata <= color_in & x_in & y_in & z_in & w_in;
137 f_pop <= vector_pop;
138 vector_data <= f_rdata(18*4-1 downto 0);
139 vector_avail <= not f_empty;
140 -- Input fifo that buffers incomming vertices.
141 f_0 : fifo_1clk_trans
142
143 port map
144 ( -- Clock and rst
145 rst => rst,
146 clk => clk,
147 -- Control signals
148 wr_en => f_push,
149 rd_en => f_pop,
150 -- Read write data
151 din => f_wdata,
152 dout => f_rdata,
153 -- Status flags.
154 almost_full => f_afull,
155 almost_empty => f_aempty,
156 empty => f_empty,
157 full => f_full );
158 -- Pipeline delay register for color.
159 color_pipe_dly_prc : process (clk,rst)
160 begin
```

```
161 if (rst = '1') then
162 color_pipe_dly <= (others => (others => '0'));
163 elsif (clk = '1' and clk'event) then color_pipe_dly(0) <= f_rdata(18*5-1
downto 18*4);
164 for i in 1 to MATRIX_MULT_LATENCY-1 loop
165 color_pipe_dly(i) <= color_pipe_dly(i-1);
166 end loop;
167 end if;
168 end process;
169 translation_matrix : matrix_multiplier port map (
170 -- Clock, rst
171 clk => clk,
172 rst => rst,
173 -- Vector Inputs.
174 vector_avail => vector_avail,
175 vector_pop => vector_pop,
176 vector_data => vector_data,
177 -- Matrix Inputs.
178 matrix_we => matrix_we,
179 matrix_waddr => matrix_waddr,
180 matrix_wdata => matrix_wdata,
181 matrix_ready => '1',
182 -- Matrix Outputs.
183 rslt_valid => rslt_valid,
184 rslt_data => rslt_data,
185 rslt_bp => rslt_bp );
186 pro_a: process(clk)
187 begin
188 rslt_bp <= not raster_rdy;
189 end process;
190 disable_normalization : if (NORMALIZE = 0) generate
191 -- X result.
192 divx_rslt <= rslt_data(18*4-1 downto 18*3);
193 divx_underflow <= '0';
194 divx_overflow <= '0';
195 divx_invalid_op <= '0';
196 divx_divide_by_zero <= '0';
197 divx_rdy <= rslt_valid;
198 -- Y result.
199 divy_rslt <= rslt_data(18*3-1 downto 18*2);
200 divy_underflow <= '0';
201 divy_overflow <= '0';
202 divy_invalid_op <= '0';
203 divy_divide_by_zero <= '0';
204 divy_rdy <= rslt_valid;
205 -- Z result.
206 divz_rslt <= rslt_data(18*2-1 downto 18*1);
207 divz_underflow <= '0';
208 divz_overflow <= '0';
209 divz_invalid_op <= '0';
210 divz_divide_by_zero <= '0';
211 divz_rdy <= rslt_valid;
212 -- W result.
213 divw_rslt <= rslt_data(18*1-1 downto 18*0);
214 -- C result.
215 divc_rslt <= color_pipe_dly(MATRIX_MULT_LATENCY-1);
```

```
216 divc_pipe_dly <= (others => (others => '0'));
217 end generate;
218 enable_normalization : if (NORMALIZE = 1) generate
219 -- Division, x/w.
220 normalize_x : float18_div port map ( a => rslt_data(18*4-1 downto 18*3),
221 -- x
222 b => rslt_data(18*1-1 downto 18*0), -- w
223 operation_nd => rslt_valid,
224 operation_rfd => open,
225 clk => clk,
226 result => divx_rslt,
227 underflow => divx_underflow,
228 overflow => divx_overflow,
229 invalid_op => divx_invalid_op,
230 divide_by_zero => divx_divide_by_zero,
231 rdy => divx_rdy );
232 -- Division, y/w.
233 normalize_y : float18_div port map ( a => rslt_data(18*3-1 downto 18*2),
234 -- y
235 b => rslt_data(18*1-1 downto 18*0),
236 -- w
237 operation_nd => rslt_valid,
238 operation_rfd => open,
239 clk => clk,
240 result => divy_rslt,
241 underflow => divy_underflow,
242 overflow => divy_overflow,
243 invalid_op => divy_invalid_op,
244 divide_by_zero => divy_divide_by_zero,
245 rdy => divy_rdy );
246 -- Floating point Division, z/w.
247 normalize_z : float18_div port map ( a => rslt_data(18*2-1 downto 18*1),
248 -- z
249 b => rslt_data(18*1-1 downto 18*0),
250 -- w
251 operation_nd => rslt_valid,
252 operation_rfd => open,
253 clk => clk,
254 result => divz_rslt,
255 underflow => divz_underflow,
256 overflow => divz_overflow,
257 invalid_op => divz_invalid_op,
258 divide_by_zero => divz_divide_by_zero,
259 rdy => divz_rdy );
260 -- w/w .
261 divw_rslt <= "00" & x"F000";
262 -- Delay the color to match up with the normalized result.
263 divc_pipe_dly_prc : process (clk,rst)
264 begin
265 if (rst = '1') then
266 divc_pipe_dly <= (others => (others => '0'));
267 elsif (clk = '1' and clk'event) then
268 divc_pipe_dly(0) <= color_pipe_dly(MATRIX_MULT_LATENCY-1);
269
270 for i in 1 to LATENCY-1 loop
271 divc_pipe_dly(i) <= divc_pipe_dly(i-1);
```

```
272 end loop;
273 end if;
274 end process;
275 divc_rslt <= divc_pipe_dly(LATENCY-1);
276 end generate;
277 -- Wire up outputs.
278 x_out <= divx_rslt;
279 y_out <= divy_rslt;
280 z_out <= divz_rslt;
281 w_out <= divw_rslt;
282 color_out <= divc_rslt;
283 valid_out <= divx_rdy;
284 end hdl;
```