PROPOSED OPENGL GPU ARCHITECTURE AND IMPLEMENTATION OF LINE RASTERIZATION ALGORITHM

By

Ahmed Ibrahim Samir Khalil

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

PROPOSED OPENGL GPU ARCHITECTURE AND IMPLEMENTATION OF LINE RASTERIZATION ALGORITHM

By

Ahmed Ibrahim Samir Khalil

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Prof. Serag E. D. Habib
Professor
Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

Prof. Hossam A. H. Fahmy
Professor
Electronics and Communications Engineering Department
Faculty of Engineering, Cairo University

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

PROPOSED OPENGL GPU ARCHITECTURE AND IMPLEMENTATION OF LINE RASTERIZATION ALGORITHM

By

Ahmed Ibrahim Samir Khalil

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Approved by the Examining Committee:

Prof. Serag E. D. Habib, Thesis Main Advisor

Prof. Hossam A. H. Fahmy, Thesis Advisor

Assoc. Prof. Amr G. Wassal, Internal Examiner

Prof. Hussein Esmail Shaheen, External Examiner
(Faculty of Engineering, Ain-Shams University)

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2015

Engineer's Name:	Ahmed Ibrahim Samir Khalil
Date of Birth:	18/07/1989
Nationality:	Egyptian
E-mail:	a.ibrahim.samir@cu.edu.eg
Phone:	01003988145
Address:	Electronics and Communications Engineering Department, Cairo University, Giza 12613, Egypt
Registration Date:	01/10/2011
Awarding Date:	..../..../....
Degree:	Master of Science
Department:	Electronics and Communications Engineering

Supervisors:

Prof. Serag E. D. Habib
Prof. Hossam A. H. Fahmy

Examiners:

Prof. Serag E. D. Habib	(Thesis Main Advisor)
Prof. Hossam A. H. Fahmy	(Thesis Advisor)
Assoc. Prof. Amr G. Wassal	(Internal Examiner)
Prof. Hussein Esmail Shaheen	(External Examiner) (Faculty of Engineering, Ain-Shams University)

Title of Thesis:

Proposed OpenGL GPU Architecture and Implementation of Line Rasterization Algorithm

Key Words:
GPU; OpenGL; Line Rasterization Algorithm; Line Drawing; Incremental Linear Interpolation

Summary:
The GPU has become an essential block for the embedded system devices. This thesis introduces a CUGPU, the Cairo University GPU, architecture based on the OpenGL ES 1.1 CL profile. CUGPU supports the fixed-function 3D graphics pipeline. Also, two designs of the line rasterization algorithm were implemented using VHDL code and synthesized at the TSMC 65 nm low power technology node. The first design scores a typical clock frequency of 270 MHz and an area of 0.088 mm^2. The second design scores a typical clock frequency of 200 MHz and an area of 0.052 mm^2.

Acknowledgments

In the name of Allah the most merciful the most gracious; all thanks to Allah the Lord of the Heavens and Earth and peace be upon Mohamed and his companions.

First and foremost I would like to express my indebtedness and gratefulness to my academic advisors: Prof. Serag E. D. Habib and Assoc. Prof. Hossam A. H. Fahmy. It has been a pleasure to work with them and learn from such extraordinary advisors. They have always made themselves available for help and advice with their boundless enthusiasm and positive thinking.

Special thanks to my colleagues at Cairo University; especially Mohamed Wagih, Ahmed Adel, Amr Mahmoud, Khalid Yehia, Mamdouh Hassan, Mahmoud Essam, Khaled Elmasry, Safaa Ahmed, and Ahmed Reda for their continuous support and the great times we had together throughout my work as a teaching assistant at the Department of Electronics and Electrical Communications Engineering at Cairo University. They made the graduate school a very enjoyable experience.

Last but not least, I want to thank my family, especially my parents, for their invaluable support during my whole life; After Allah, without their help and support I would not have accomplished anything in my life.

Ahmed I. S. Khalil

June, 2015.

To my family and all my friends

Table of Contents

List of Tables

List of Figures

List of Symbols and Abbreviations

Symbols	Description
(α,β,γ)	Baracentric coordinates of the produced fragment relative to triangle vertices.
(R,G,B,A)	The components of the RGBA color.
(S,T,R,Q)	3D homogenous texture coordinates.
(x,y,z,w)	3D homogenous vertex coordinates in the object space .
(x_w,y_w)	Vertex coordinates in the window space.
f	Associated data of the primitive.
L	Luminance color component.
M	The model view matrix.
m	The slope of the line.
P	The projection matrix.
t	The interpolation value.
V	The viewport matrix.
z	The depth value.

Abbreviations	Description
2D	Two Dimensional.
3D	Three Dimensional.
API	Application Programming Interface.
APU	Accelerate Processing Unit.
CL	Common Lite.

CM	Common Mode.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
CUGPU	Cairo University GPU.
CUSPARC	Cairo University SPARC.
DSP	Digital Signal Processing.
FPGA	Field Programmable Gate Array.
GFLOPS	Giga FLoating-point Operations Per Second.
GP	General Purpose.
GPU	Graphics Processing Unit.
IEEE	Institute of Electrical and Electronics Engineers.
IP	Intellectual Property.
LoD	Level of Details.
LP	Low Power.
OpenCL	Open Computing Language .
OpenGL	Open Graphics Library.
OpenGL ES	Open Graphics Library for Embedded Systems.
OpenVG	Open Vector Graphics.
PGA	Professional Graphics Adapter.
RTL	Register Transfer Language.
SM	Streaming Microprocessor.
SoC	System On Chip.
SPARC	Scalable Processor Architecture.
TSMC	Taiwan Semiconductor Manufacturing Company.
VGA	Video Graphics Array.
VHDL	VHSIC Hardware Description Language.

Abstract

The Graphic Processing Unit (GPU) has become an essential block for the embedded system devices. The GPU's main purpose is to accelerate the rendering of images, animations and videos. This process is essential for many devices such as smart phones, tablets, and gaming devices. These devices are quickly becoming our most valuable personal computers. Nowadays, we use the mobile devices to read email, browse the Web, take photos, or play games. With this rising demand of graphics applications such as games, the rapid development of embedded systems, and the enhancement of the functionality of the integrated circuits for the smaller technology nodes, GPUs are used for the general-purpose computations in addition to the graphics computations. However, their main target is the reasonable performance with low power since the embedded systems' applications are simpler and resolutions are smaller.

Our motivation is to design and implement the Cairo University GPU (CUGPU), the first embedded GPU in Egypt. We propose the CUGPU architecture based on Common-Lite (CL) profile of the Open Graphics Library for Embedded System 1.1 (OpenGL ES 1.1). CUGPU provides high-performance support of the fixed-function 3D graphics pipeline. Moreover, CUGPU can be integrated with the Cairo University SPARC (CUSPARC) processor into a complete embedded system. Such a complete embedded system may be used in educational or commercial kits and devices.

As a starting point to implement the CUGPU and to have a clear vision about the hardware cost and performance, we implemented line rasterization algorithm that satisfied the diamond-exit rule. In order to satisfy the diamond-exit rule conditions for line drawing, we modified the Bresenham algorithm by adjusting the initial distance. We also simplified the OpenGL interpolation equations and reformulated them to be performed incrementally and hence they are adapted to Bresenham algorithm.

Because we do not know definitely the bottleneck unit of the CUGPU, we implemented two designs of the line rasterization. The first design targeted the maximum

frequency while the second design minimized the area for a reasonable frequency. We then synthesized it at TSMC 65nm low power technology node. The first design scores a typical clock frequency of 270 MHz and an area of 0.088 mm^2. The second design scores a typical clock frequency of 200 MHz and an area of 0.052 mm^2.

Chapter 1

Introduction

1.1 Goal and Motivation

With the rising demand of graphics applications such as games and the rapid development of embedded systems, GPU has become an essential block for the embedded system devices for different purposes. Our team had developed the CUSPARC processor [1–3], the first Egyptian embedded processor. So, our motivation is to design and implement the Cairo University GPU (CUGPU), the first embedded GPU in Egypt based on OpenGL ES 1.1 CL profile. Moreover, CUGPU can be integrated with the CUSPARC processor into a complete embedded system. Such a complete embedded system may be used in educational or commercial kits and devices.

1.2 Results

We proposed the CUGPU architecture based on the OpenGL ES 1.1 CL profile. It satisfied the mandatory specifications only to minimize the power and area without deteriorating its performance. We implemented two design of the line rasterization algorithm. The implementation and verification of the line rasterization algorithm is carried out at TSMC 65nm low power technology node. The first design scores a typical clock frequency of 270 MHz and an area of 0.088 mm^2. The second design scores a typical clock frequency of 200 MHz and an area of 0.052 mm^2.

1.3 Organization of the Thesis

The thesis is organized as follows: Chapter 2 discusses the graphics pipeline and gives an overview of the GPU-related APIs and their usages. Chapter 3 presents the evolution of GPUs, their architectures, and their products. Chapter 4 discusses our CUGPU architecture and specifications. It also presents the register file, architecture, and algorithms of each block. In chapter 5, we present our line rasterization algorithm. Also, we discuss our hardware implementation of this algorithm. In chapter 6, the line rasterization algorithm is verified using line samples of popular scenes. In addition, we present the synthesis result of our design. Finally, we conclude the thesis in chapter 7 with presenting some future work ideas.

Chapter 2

Graphics Pipeline

The Graphics Processing Unit (GPU) is a programmable logic device that accelerates the rendering of images and animations for display devices. Rendering [4–7] is the process of drawing objects by calculating the pixels that are covered by this object and sending them to the screen. This rendering process can be classified into two categories:

1. The ray tracing, pixel-order rendering, is the process of looping over pixels, one by one, and finding the objects that influence it. The image quality is higher than rasterization since it calculates the relation between all objects that affect the pixel, but it needs more computational resources and consumes more time because it iterates over the number of pixels.

2. The rasterization, object-order rendering, is the process of drawing objects, one by one, by calculating the pixels that are occupied by the object primitives. Rasterization is more efficient than ray-tracing, especially for large primitives, because it iterates over the primitives. So, it is more suitable for the interactive applications such as games. However, it produces lower quality images.

The graphics pipeline is the sequence of operations that are required to render the images objects, beginning by receiving vertices and ending by sending pixels to the screen. It mainly includes the operation of transformation, lighting, clipping, rasterization, texture mapping, culling, and fragments testing. Availability, order, and the corresponding implementation of these operations are based on the corresponding API. The mandatory units are the transformations and rasterization blocks. The blocks can be re-ordered based on the desired cost, performance, and image quality. Each block can be implemented by fixed hardware, programmable hardware, or software. API is a high level set of functions,

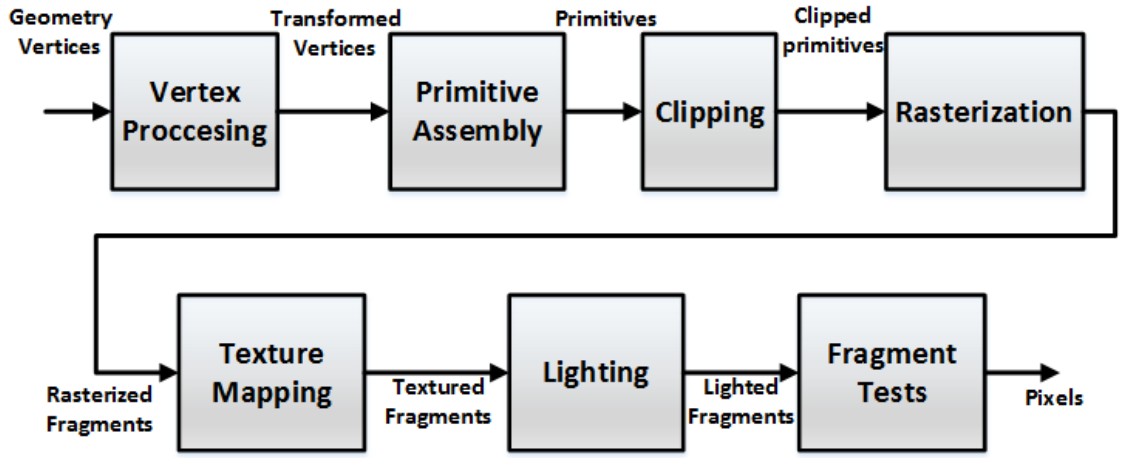


Figure 2.1: The graphics pipeline.

variables, and data structures that are provided by libraries to support writing and modifying the graphics and computational applications in a simple and flexible way. It defines the functions, data types, and operations that are independent of their corresponding implementation. So, implementer is free to design their GPU architectures that satisfy their desired API. In this chapter, we discuss, in details, the common 3-stages graphics pipeline in section 2.1. Then, we present an overview of the GPU-related APIs and their usages in section 2.2. Finally, we discuss briefly the OpenGL ES 1.1 API in section 2.3.

2.1 Common Graphics Pipeline

Figure 2.1 shows the basic stages of the graphics pipeline [6]. It receives vertex coordinates, colors, normal, and texture coordinates of the object from the application and generates the pixels' colors for the display. It consists mainly of vertex transformation, primitive assembly, clipping, rasterization, texture mapping, lighting, and fragment tests. First, the vertex processing unit transforms the vertex coordinates from the object space to the screen space and makes the required transformations on the texture coordinates. Then, the primitive assembly assembles the primitives from their coordinates based on the primitive's type (point, line, and triangle) and vertices' indices. Then, the assembled primitive is checked against the view volume in the clipping unit. The clipping unit may pass, discard, or generate new primitives to the rasterization unit. The rasterization unit determines the fragments that are covered by the primitives and calculates the fragments' colors, texture coordinates, and normal by interpolating the colors, texture coordinates, and normal of the primitives' vertices, respectively. If the texture mapping is enabled, the texture unit replaces incoming fragment's color by the texture color. This color is fetched

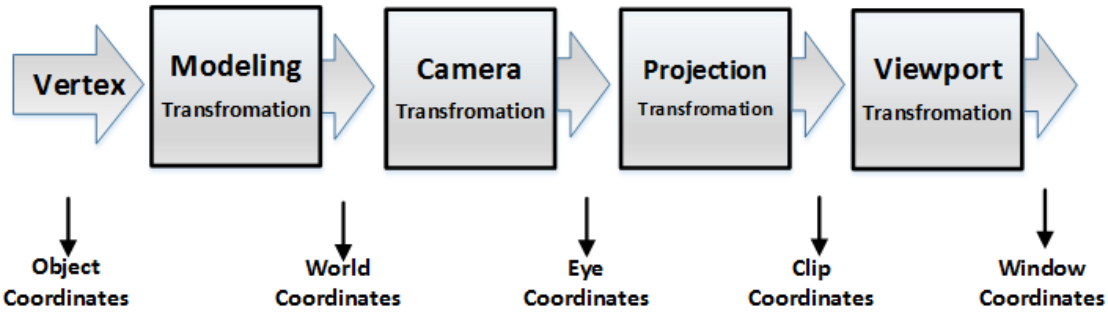


Figure 2.2: The sequence of vertex transformations

from the texture image by the texture coordinates. Then, the lighting unit calculates the final color of the fragment by performing shading computations on each fragment. This depends on the light sources and environment effects. Finally, some tests are performed on the fragment to determine if the corresponding pixel, in the frame buffer, is updated or not.

2.1.1 Vertex Processing

The vertex processing unit [6] maps the incoming vertices' locations, represented as (x, y, z) coordinates in the object space, to the coordinates in the window space, represented as (x_w, y_w). This process is divided into consecutive transformations as shown in Figure 2.2. First, the incoming vertices' locations are transformed from the object space to the world space. This transformation depends on the location of the object in the world space. Then, the vertices are transformed to the eye space. In this space, the camera is placed at the origin. It depends on the camera position and direction. Then, the resulting object is projected from the 3D camera space to the 2D clip space. All visible points fall in the range -1 to 1. There are two types of projection:

1. The perspective projection is the process of representing the 3D object on a 2D surface to be seen as the original image in the 3D space. In this space, rays converge at the camera location. The object size varies inversely with distance to look realistic; the far objects appear smaller than the near objects. The distance and angles between objects are not reserved.

2. The parallel projection is the process of representing the 3D object on a 2D surface to be seen as if you looked from the front view. In this space, rays converge at infinity. Parallel lines remain parallel and relative proportions of objects are deserved. The images have less realistic look; the far away objects do not get smaller.

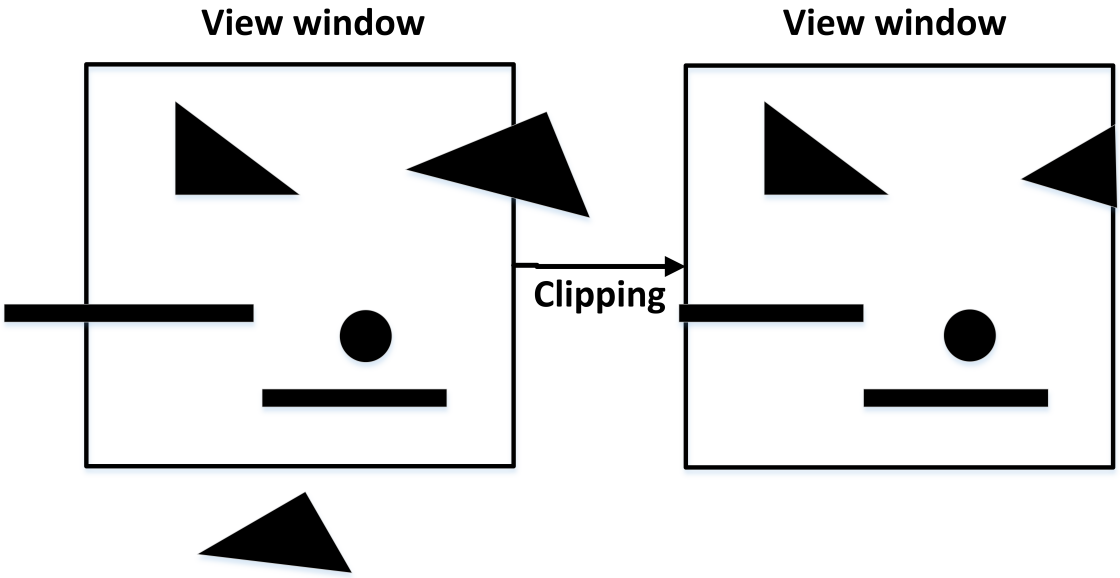


Figure 2.3: illustration of clipping process

Finally, the 2D clip space is transformed to the screen space. It is just a scaling to a window with the width and height of the space.

2.1.2 The Primitive Assembly

The primitive assembly unit [6] constructs the primitive from the consecutive incoming vertices based on the primitive type. For a point primitive type, each incoming vertex belongs to a new primitive. For a primitive with a line type, the line is constructed from two vertices based on the line type: line segment or line strip. For a triangle type, the triangle is assembled from three consecutive vertices based on the triangle type. They may be separated triangles, triangle strips, or triangle fans.

2.1.3 The Clipping Unit

The clipping unit [7] removes the portions of objects that are outside the view volume or behind the eye because it may lead to incorrect results during rasterization of these primitives and consumes more delay and power. For a point type, the clipping unit discards the incoming point if it is located outside the view volume. Otherwise, it passes the point to the rasterization unit. For a line type, it is discarded if the two vertices are located outside the view window and the line does not intersect the view window. If the line intersects the view window, a new line is produced based on the intersection points.

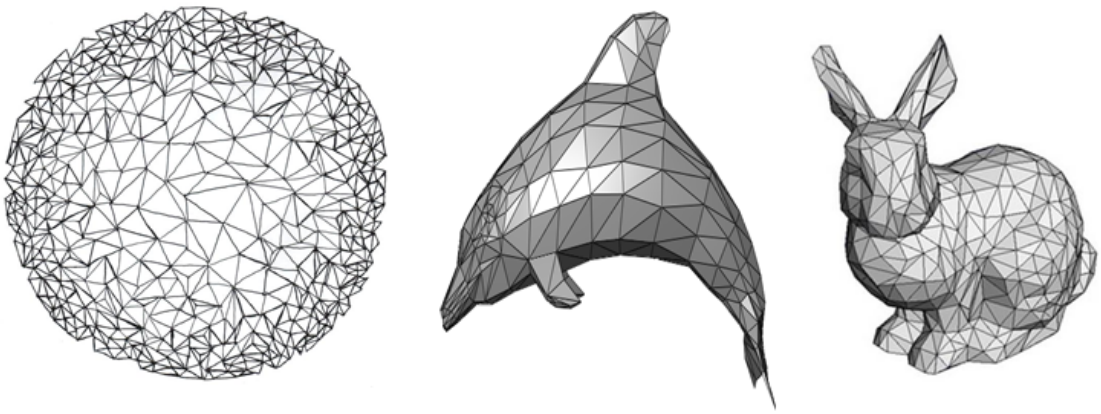


Figure 2.4: Representation of different shapes using triangle primitives.

Otherwise, the line is passed to the rasterization unit. For a triangle type, it is discarded if its vertices are located outside the view window and the triangle does not intersect the view window. If the triangle intersects the view window, new triangles are produced based on the intersection points. Otherwise, the triangle is passed to the rasterization unit. Figure 2.3 illustrate the clipping function by demonstrating different cases of point, line, and triangle primitives.

2.1.4 The Rasterization Unit

The rasterization [6]is the main operation in the rendering process. The post-clipped primitives are fed to the rasterization unit. For each primitive's vertex, there are window coordinates, associated color, associated texture coordinates, normal, and depth. First, the rasterization determines the fragments that are covered by the incoming primitive. Then, it calculates the associated color, texture coordinates, normal, and depth for each produced fragment by interpolating the primitive associated colors, texture coordinates, normal, and depth, respectively. The basic primitive's types are the lines and triangles since any complex shapes can be represented by them, as shown in Figure 2.4, and their geometry simplifies the rasterization operation. However, it may support circle, rectangle, and ellipse primitives.

For a line primitive, as shown in figure 2.5a, there are many drawing algorithms based on the line equation and how to use the relation between the consecutive fragments. The most popular algorithms are the incremental Direct Drawing Algorithm (DDA), the mid-point algorithm, the Bresenham algorithm, and the double-step mid-point algorithm. We discuss the line drawing, in particular, in chapter 5. For a triangle primitive, as shown in

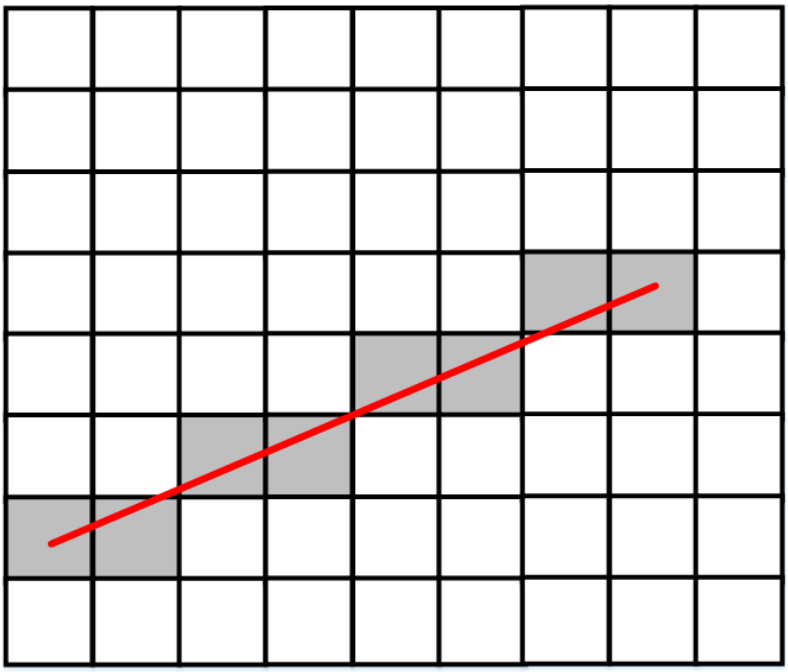
figure 2.5b, it can be implemented based on the line drawing method. The most popular method is the scan-conversion method. It is based on drawing the triangles' lines first. Then, it draws the fragments located in the area between these lines. The colors, texture coordinates, normal, and depth of the produced fragments are calculated by interpolating the colors, texture coordinates, and normal of the primitive vertices by the equation $f = f_0 * \alpha + f_1 * \beta + f_2 * \gamma$ where (α, β, γ) are the barycentric coordinates of the produced fragment relatively to the primitive vertices. And (f_0, f_1, f_2) are the associated data for the three vertices of the primitive.

2.1.5 The Texture Mapping Unit

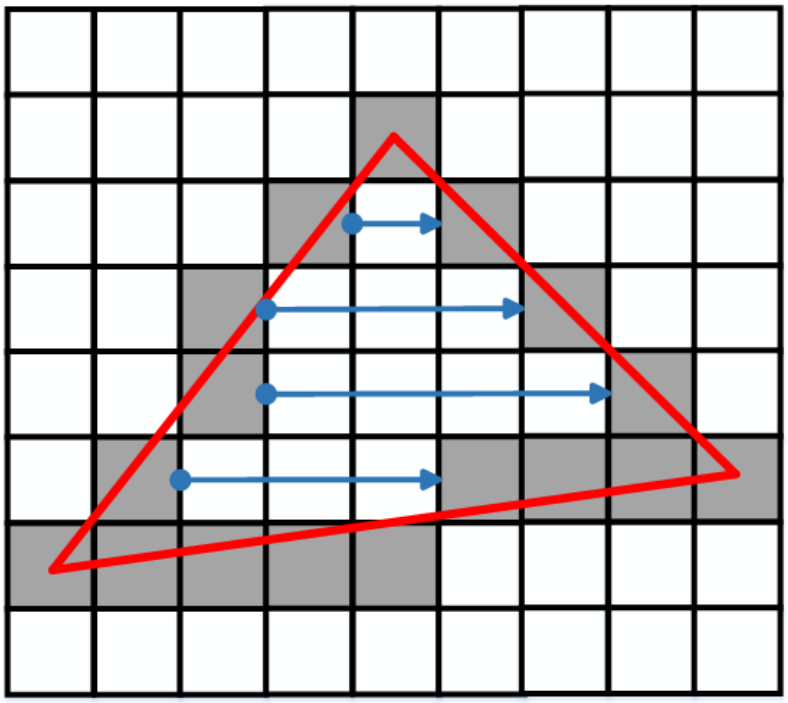
Rasterization produces fragments with its initial color. However, if the texture mapping [7] is enabled, the fragment's initial color is replaced by or combined with a texture color that is fetched from the active texture. The texture is an image that is used to add extra details to the fragment to look realistic, especially for the complex surfaces. The texture mapping unit uses the associated texture coordinate (s, t, r), with each fragment, to address the texture image and calculate the fragment color. Figure 2.6 shows an example of mapping part from the texture image to the triangle area. The texture mapping process depends on

1. The dimensions of the texture images. The texture may be 3D or 2D images. For a 3D image, it is addressed using the texture coordinates (s, t, r); otherwise, the texture coordinates (s, t) only are used.
2. The texture is a complete procedural or a table look-up.
3. The Filtering mode. For a table look-up, the texture unit depends on the magnification or minification filtering method. It may be linear, nearest, or average. It specifies how to calculate the texture for a general address from the look-up table.
4. The texture coordinates' mode.

The standard coordinates (u, v) of the images are located in the interval [0, 1]. So, the texture coordinates have to be transformed to this bounded interval. The coordinates' modes are clamping or repeated. Each of these modes is applied per component of a texture. Figure 2.7 illustrates the texture coordinates' modes by describing the relation between the input texture coordinate (s) and the used texture coordinates $f(s)$.



(a) Line drawing.



(b) Triangle drawing.

Figure 2.5: Primitive drawing example.

2.1.6 The Lighting Unit

The lighting unit [6, 7]calculates the final color of the fragment by applying the shading model. Shading is the process of finding the color of each pixel. In general, there are three standard shading models:

1. Flat shading which uses the same color for all fragments that are produced from the same primitive.

2. Gouraud shading performs the shading computations on the vertices based on the light sources' locations and colors before the rasterization. Then, the fragment color is calculated by interpolating the lighted vertices' colors.

3. Phong shading which calculates per-fragment normal by interpolating the associated normal with the vertices in the rasterization stage. Then, it calculates the shading computations on each fragment separately according to its location, its normal,

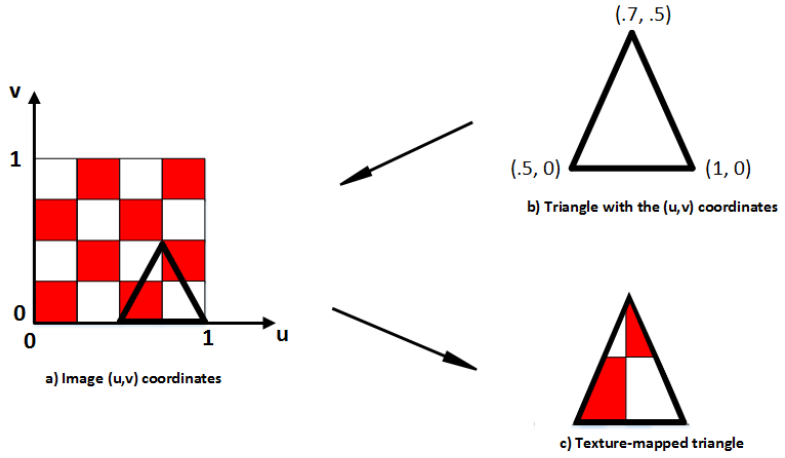


Figure 2.6: Triangle texture mapping example.

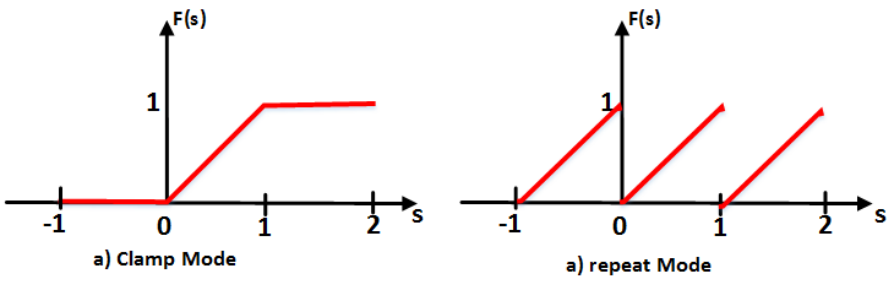


Figure 2.7: The functions of the coordinate's modes.

the light sources' location, and the light sources' color. For a fixed-function graphics pipeline, it supports the flat and Gouraud shading only. But a programmable graphics pipeline may also support the Phong shading before it requires more computations resources.

Also, the lighting effects are determined, based on type of the light source and the material of the primitive. These materials give the object surface characteristics to represent that the object may emit light beside the reflection. The standard types of light sources are

1. Directional lights that are located at infinity and the directions of the light rays are parallel such as the sun.
2. Point lights that are located at finite locations and their light rays are in all direction.
3. Spotlights that are located at finite locations, but emit light only within a cone.

2.1.7 The Fragment Tests Unit

Finally, some tests are performed on the fragment before updating the fragment buffer. The essential tests are scissor test, stencil test, depth buffer test, and blending test. Each test checks conditions on the incoming fragment to decide whether to discard or pass the fragment as shown in figure 2.8. First, the scissor test determines if the incoming fragment lies within the scissor rectangle defined by four input values from the user. If the fragment is located inside this window, the fragment is passed to the stencil test; otherwise, the fragment is discarded. Then, the stencil test conditionally discards a fragment based on the outcome of a comparison between the value in the stencil buffer at the fragment location and a reference value defined by the user. Then, the depth test discards the incoming fragment if a depth comparison fails. Generally, it checks if the incoming

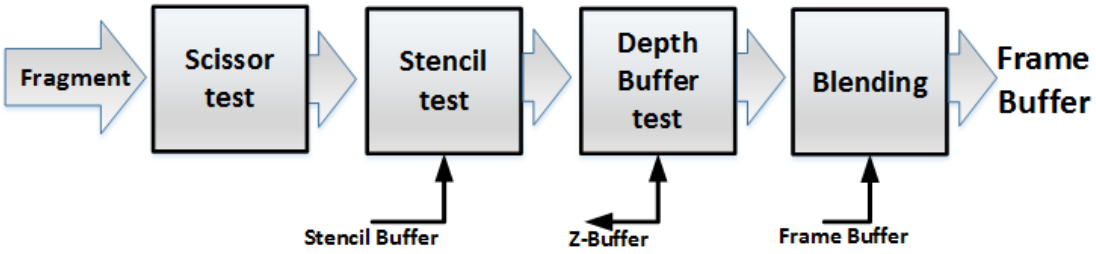


Figure 2.8: the fragment tests block diagram.

fragment is closer than the stored fragment to the camera or not. If the incoming fragment has less depth than the stored depth value, the stored fragment is replaced by the incoming fragment; otherwise, the fragment is discarded. Finally, the blending combines the incoming fragment color components (Red, Green, Blue) with the stored pixel color according to the transparency value (Alpha) to generate the final color of the pixel and update the frame buffer.

2.2 Application Programming Interface (API)

The Application Programming Interface (API) [8–10] is a set of functions, variables, objects, and data structures for building the software application. It defines the tasks and functions of a system, such as hardware driver or video card; regardless of its implementation in order to make the building of applications simpler and more flexible. It is often provided by libraries that include the specification for routine, classes, and variables.

In the graphics case, the API defines the functions of the graphics pipelines regardless of the implementation of the graphics hardware. It may be designed as a dedicated hardware such as GPU. Also, it may be implemented as a software library running on the CPU. Or, it may be implemented using the CPU and GPU both. It defines the graphics pipeline units such as vertex processing, rasterization, and lighting. For each unit, it defines its function and its specifications. Also, it defines commands that are used to control its function if applicable. There are many APIs that are defined for the desktop-based and embedded systems' graphics pipeline. For the desktop-based, the most popular APIs are DirectX, CUDA, OpenGL, and OpenCL. For the embedded based, the corresponding APIs are Open GL ES, OpenVG, and DirectMobile. Each API may have different versions; some for the fixed-function graphics pipeline and the other for the programmable graphics pipeline. We discuss briefly the different APIs for the desktop and embedded graphics pipelines.

2.2.1 Desktop-Based APIs

The Desktop-Based GPU have many APIs. Some APIs are related to the graphics processing on the GPU and the others concern about the general computations on the GPU. Most APIs are hardware independent. However, some APIs target a specific hardware such as CUDA. The performance is the main target for these APIs.

2.2.1.1 DirectX

DirectX [11] is the most popular API group for supporting the multimedia applications on the desktop PCs. It is a collection of APIs for handling graphics and sounds tasks, on Microsoft platform, such as Direct3D, Direct2D, DirectCompute, DirectDraw, DirectMusic, DirectPlay, DirectSound. It is included with window 95 and all subsequent releases. The DirectX 7 is the last API support fixed-function graphics pipeline. The main APIs that are related to the GPU are

1. Direct3D. It concerns about the rendering process of the three-dimensional graphics. Its applications mainly target the high performance such as 3D-games. It discloses the graphics capabilities of the 3D graphics hardware such as clipping, culling, and texture mapping. Its C++ library is D3DX.
2. Direct2D. It concerns about the rendering of the 2D and vector graphics on the Microsoft platforms. It takes the advantage of existence of the hardware acceleration units.
3. DirectCompute. It is an API that supports the execution of the general-purpose computations on the graphics processing units. It is supported on DirectX 11 GPUs.

2.2.1.2 OpenGL

OpenGL [9, 12] is an open standard API for rendering the 2D and 3D graphics. OpenGL is a cross-platform API, not a specific platform. So, GPUs often have an OpenGL implementation. The graphics applications can be used on different types of graphics hardware. OpenGL is supported on Windows 95 and the subsequent versions, Linux, and Mac OSX. There are different generations for the OpenGL

1. OpenGL 1.x generation is an API for the fixed-function graphics pipeline.
2. OpenGL 2.x and 3.x generations supported both fixed and programmable graphics pipeline functionality.
3. OpenGL 4.x generation supports the programmable function graphics pipeline only.

2.2.1.3 OpenCL

OpenCL [13] is open-standard, cross-platform, API to support the general-computation execution on the programmable devices. It specifies APIs to execute the

application across the heterogeneous platforms consisting of CPUs, GPUs, DSPs, and FPGAs. The OpenCL 1.1 was released in 2009.

2.2.1.4 CUDA

CUDA [14] is not just API. But, it is a parallel computing platform that is implemented on the NVIDIA's GPUs. NVIDIA intended to accelerate the execution of the general-purpose computations on their GPUs. It provides two APIs

1. CUDA driver API, a low-level API
2. CUDA runtime API, a higher level API that is implemented over the CUDA driver API.

2.2.2 Embedded Systems APIs

The APIs for Embedded Systems are driven from the Desktop-based APIs. But, the main target is the reasonable performance with low power since the embedded systems' applications are simpler and resolutions' are smaller. The popular APIs are OpenGL ES, OpenVG, and Direct Mobile.

2.2.2.1 Direct Mobile

Direct3D Mobile is the embedded systems' API that supports for the 3-D graphics application running on Windows CE-based platforms. It belongs to the DirectX family.

2.2.2.2 OpenGL ES

OpenGL ES [8, 9] is the embedded systems generation of the OpenGL. It is an open standard API for rendering the 2D and 3D graphics. OpenGL is a cross-platform API, not a specific platform. There are different versions for the OpenGL ES

1. OpenGL ES 1.1 is an ES API for the fixed-function graphics pipeline, derived from the Open GL 1.5.

2. OpenGL ES 2.x and 3.x versions supports programmable graphics pipeline functionality.

2.2.2.3 OpenVG

OpenVG [10]is an embedded system API designed for hardware-accelerated 2D vector graphics.

2.3 OpenGL ES 1.1

OpenGL ES 1.1 [8, 15] is embedded fixed-function hardware OpenGL API. It is based on the OpenGL 1.5 graphics system, but is designed for graphics hardware running on embedded devices. Its features are more adapted for the embedded devices and their computational abilities. It is fully compatible with Open GL ES 1.0. It enhanced functionality to improve the performance and save the power. OpenGL ES 1.1 is implemented in more than 180 commercial products. It is supported by different operating systems such as Android and iOS by enormous vendors as NVIDIA, Apple, ARM, QUALCOMM, and Intel. Table 2.1 shows a subset of these products.

There are two profiles for OpenGL ES 1.1: Common mode (CM) and Common-Lite (CL). The common-lite profile targets the systems of simpler graphics and it supports the fixed-point operations only whereas the common profile targets a wider range of applications and supports the high performance floating-point formats. The Common-Lite profile supports only commands taking fixed-point arguments, while the Common profile also includes many equivalent commands taking floating-point arguments.

OpenGL ES 1.1 supports different features for the graphics pipeline

1. Draw arrays. It supports drawing the primitive from vertices, colors, and texture coordinates that are stored in the system memory arrays. It can draw them as points, line strips, line loops, separate lines, triangle strip, triangle fans, and separate triangles.

Table 2.1: Subset of OpenGL ES 1.1 Products

Vendor	Products	Operating System
ARM	Mali T720	Linux 3.4
Intel	Processor Z3000 series	Android 4.2.2
Apple	Apple iPhone 4	iOS 7
QUALCOMM	MSM 8610	Android 4.3
NVIDIA	Tegra 3	Android 4.1

2. Buffer objects. User may choose to create buffer object and store the vertices and their associated data in the fast graphics memory. Then, they can be rendered from graphics memory directly. This improves the performance.

3. Point Sprites. It supports different sizes, fetch from size array, for the points instead of the single size of the array's points. Also, the texture coordinates are interpolated across the point

4. User-defined clip planes. It allows the user to define clip-planes that may increase performance and save power.

5. Texture processing. It enhances the texture mapping by supporting two texture units at minimum. Also, it uses the texture combiner functionality for improving 3D effects.

6. Direct Control. It provides a direct control over the 2D and 3D fundamental operations such as the lighting, antialiasing, and texture mapping.

Chapter 3

GPU Literature Survey

Over the last 35 years, the graphics hardware has evolved dramatically from a simple frame buffer to an enormous heterogeneous system of hundreds of cores. Initially, only a frame buffer was used to store the color values of pixels that can be displayed on the screen. Then, the CPU was used to perform some operations to accelerate the rendering process such as transforms and lighting. Then, a dedicated hardware, the Graphics Processing Unit (GPU), was involved to perform the graphics operations. Currently, a heterogeneous system that consists of CPUs, GPUs, and FPGAs programmable devices is used to perform the graphics processing. Additionally, the graphics hardware is used to carry out massive general-purpose computations.

To satisfy the requirements of graphics and general-purpose applications, GPU's architecture has been developed dramatically from fixed-function architecture to the unified-shader programmable architecture. The NVIDIA GeForce 256, the first commercial GPU, is a fixed-function architecture. Today, most GPUs are stream processors based on the unified-shader architecture. GPUs evolved based on the rising demand of the graphics applications and the enhancement of speed, computational power, and functionality of the integrated circuits for the smaller technology nodes. Nowadays, GPUs may be located in a wide range of systems such as desktops, laptops, and mobile phones. Their hardware units are used for the graphics and general-purpose computations.

In this chapter, we discuss the evolution of GPUs, their architectures, and their products. We start by a survey of the GPUs that are used in the Desktop-based systems in section 3.1. Then, we discuss briefly the embedded GPUs in section 3.2. Finally, we focus on ARM Mali-200 GPU as an example of GPUs that support the OpenGL ES 1.1 in section 3.3.

3.1 Desktop-Based GPUs

GPUs originally targeted the desktop-based systems. They were implemented to execute the graphics pipeline. The main corporations are NVIDIA and AMD. Before 1999, the graphics operations were executed on the GPU and CPU. In 1999, the first standalone GPU was introduced by NVIDIA. It was a fixed-function architecture. From this time, all graphics operations could be performed on the GPU. In 2001, NVIDIA released the GeForce 3 that was the first programmable GPU. In 2006, the GeForce 8800 is the first unified-shader architecture. In 2010, the Fermi Architecture, released by NVIDIA, was the first GPU architecture that allowed the execution of the general-purpose computation on the GPU. In 2011, AMD released Llano, the first Accelerated Processing Unit (APU). APU is a single chip that contains the CPU and GPU on the same chip. In 2010, Fatahalian [16–18] introduced a real-time graphics pipeline for micro-polygon rendering as the next era graphics pipeline. His architecture targets the high-performance but its cost is high.

3.1.1 Graphics Accelerator

The first GPU was introduced in 1999. Before 1999, the graphics hardware was just supported hardware to accelerate the rendering process. Till 1980, the graphics hardware [19] was just integrated frame buffer that is used to store the pixels' colors. In 1984, IBM released the Professional Graphics Adapter (PGA), the first video card for the PC. It was responsible for all tasks that were related for drawing and coloring the polygons. It was an important step in the evolution of the graphics hardware because this was the first time to think about having a separated hardware for the graphics processing. But, it did not achieve market success due to its high cost and lack of compatibility with non-IBM applications.

In 1987, more features had to be supported with the graphics pipeline such as vertex lighting, rasterization, and color blending. These operations were performed on the CPU due to the computations limitations of graphics hardware at this time. In mid-1990s, 3DFX, NVIDIA, and AMD released their Voodoo, TNT, and Rage 3D graphics boards, respectively. However, they still could not handle more than one triangle simultaneously. In 1996, 3DFX released a new Voodoo version that was considered the first actual 3D game card. It only offered 3D acceleration. The CPU still did vertex transformations, lighting, and clipping while the graphics card performed texture mapping and rasterization.

With the increase of the computational power of the graphics hardware, it was responsible for more tasks such as lighting and clipping as shown in figure 3.1. In 1999, NVIDIA GeForce 256, the first GPU took the full responsibility of the graphics tasks including vertex transformations, lighting, clipping, rasterization, and blending.

3.1.2 Fixed Function GPU

In 1999, NVIDIA released GeForce 256 GPU [20], the first graphics hardware that did the all graphics pipeline operations. It was a single chip GPU that performs vertex transformation, lighting, clipping, primitive-assembly, rasterization, texture mapping, culling, and blending. It was known as fixed-function pipeline because after the graphics data is sent to the GPU's pipeline, this could not be changed.

Each unit is controlled by some parameters to operate on the graphics data. For example, the vertex transformation is loaded by the model-view and projection matrices to control the transformation of the object's vertices. The clipping unit is controlled by the view volume that defines the drawing region. The lighting unit needed the locations and directions of light sources, the shading model and so on. GeForce 256 series supports DirectX 7.0 and OpenGL 1.2. Also, ATI releases Radeon R100 [21], the first ATI's GPU with the fixed-function pipeline. It was fully compliant with DirectX 7 and OpenGL 1.2.

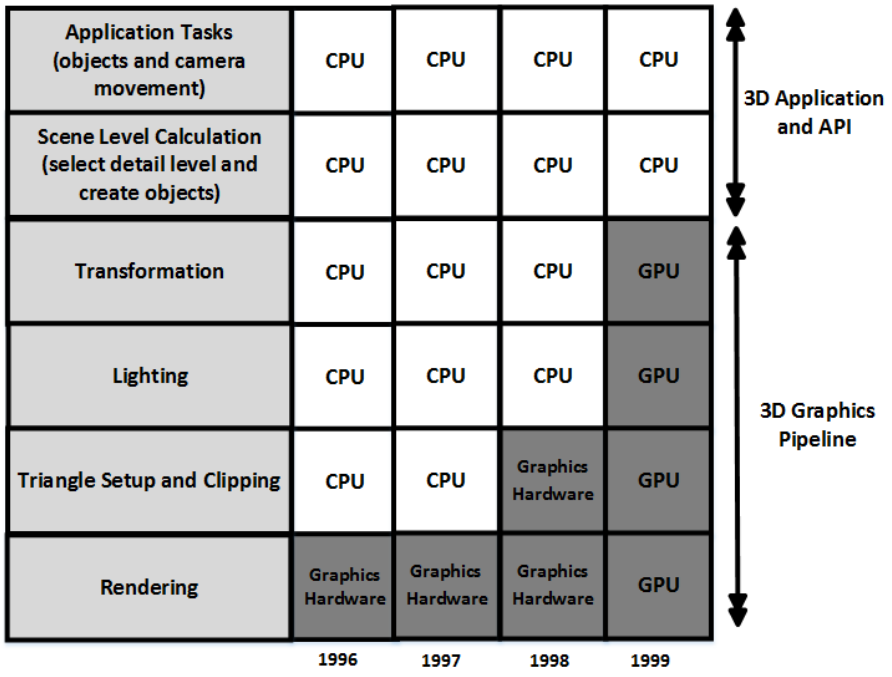


Figure 3.1: Evolution of graphics hardware before 2000

The main drawback of the fixed function pipeline was the inflexibility of the GPU feature since the OpenGL and DirectX APIs were implemented in the hardware. That's why NVIDIA and ATI produced their programmable GPUs.

3.1.3 Programmable GPU with Fixed Shader

To overcome the inflexibility problems, NVIDIA and ATI modified their architectures gradually [21]. They replaced parts of the fixed-function pipeline's units by the programmable ones. It allows the programmers to program parts of the graphics pipeline. They wrote shaders that operate on the objects' data. These shaders are programs that are written in assembly-like shader languages to set each programmable unit. In 2001, NVIDIA released the GeForce 3 series, the first GPUs that had some programmable parts. The vertex processing unit is replaced by a vertex shader unit. They support the DirectX 8.0 and OpenGL 1.3. Then, ATI released a similar GPU, Radeon R200. It supports OpenGL 1.4 and DirectX 8.1.

In 2002, the first fully programmable GPUs were introduced. NVIDIA and ATI released the GeForce 5 and Radeon R300 series, respectively. These GPUs had separate vertex and fragment programmable processors as shown in figure 3.2. So, the programmer can write two shaders: vertex shader and fragment shader. Vertex shader was responsible for transforming the vertices' coordinates from the object space to the screen space. Also, it produced the texture coordinates and lighted the vertices, while fragment shader was responsible for calculating the final pixel color and performing the texture mapping. GeForce FX and Radeon R300 both supported the DirectX 9.0 and OpenGL 2.0.

In 2004, NVIDIA and ATI released the GeForce 6 series [22]and Radeon R420 series, respectively. They were updated versions that enhanced the buses, the speed, and the shader languages. They supported the DirectX 9.0 and OpenGL2.1. The graphics pipeline, as shown in figure 3.2, contains programmable vertex processors, a fixed clipping unit, a fixed rasterization unit, a programmable fragment processor, and fixed-blending units. The main disadvantage is that the performance may be degraded based on the fragment and vertex payloads ratio of the application. For some applications, the vertex processors may be idle whereas the fragment processors are fully occupied and vice versa.

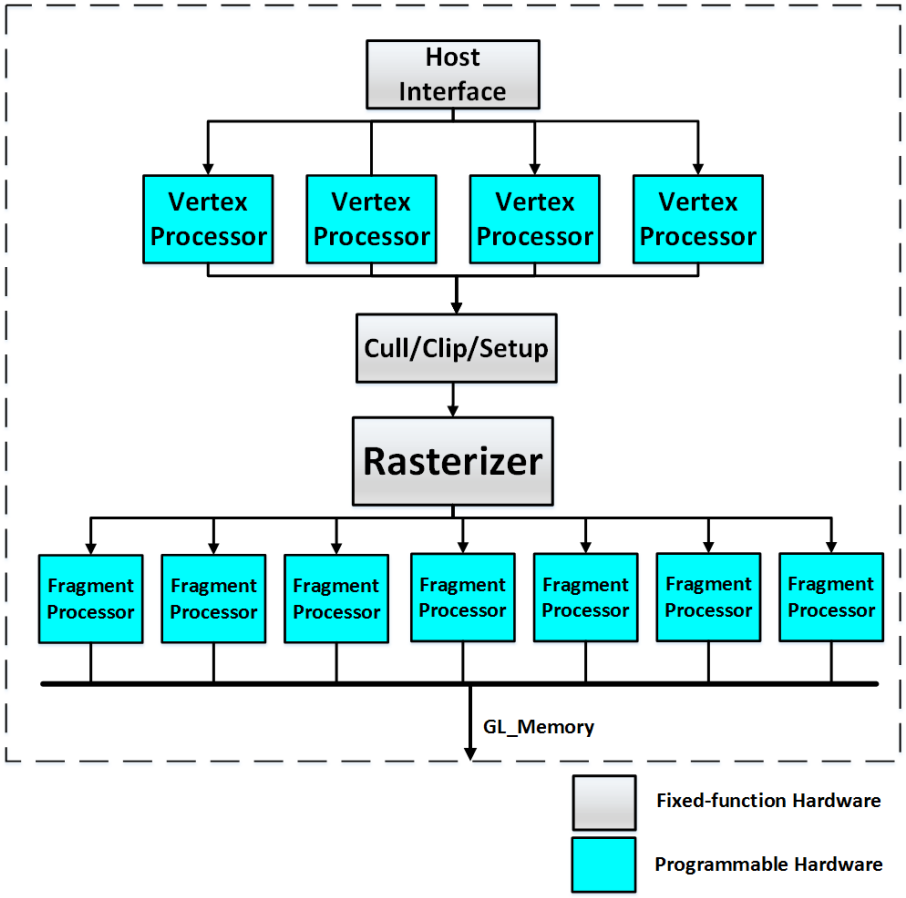


Figure 3.2: Programmable GPU with fixed shader architecture

3.1.4 Programmable GPU with Unified Shader

In 2006, the GPUs were exposed as streaming processor [21]. NVIDIA released the GeForce 8 GPUs series. GeForce 8 series was the first programmable GPUs with unified processors. The unified processor, called the Streaming Microprocessor (SM), is a simple core that can handle vertex, geometry, and fragment computations as shown in figure 3.3. So, the vertex, geometry, and fragment shaders could be considered as threads. These threads could be run on any SM and graphics pipeline become a software abstraction. ATI released the TeraScale 1 family, its first unified programmable GPU family. Both GeForce 8 and TeraScale 1 were fully-compliant with DirectX 10 and OpenGL 3.3.

Figure 3.3 demonstrates an example of unified GPU's architecture; the GeForce 8800 [23] architecture. It is considered as a stream of programmable processors that can run the vertex, geometry and fragment programs based on the application. So, the same processor may execute the vertex thread on the input vertices, the geometry thread on the input primitives, or the fragment thread on the input fragments.

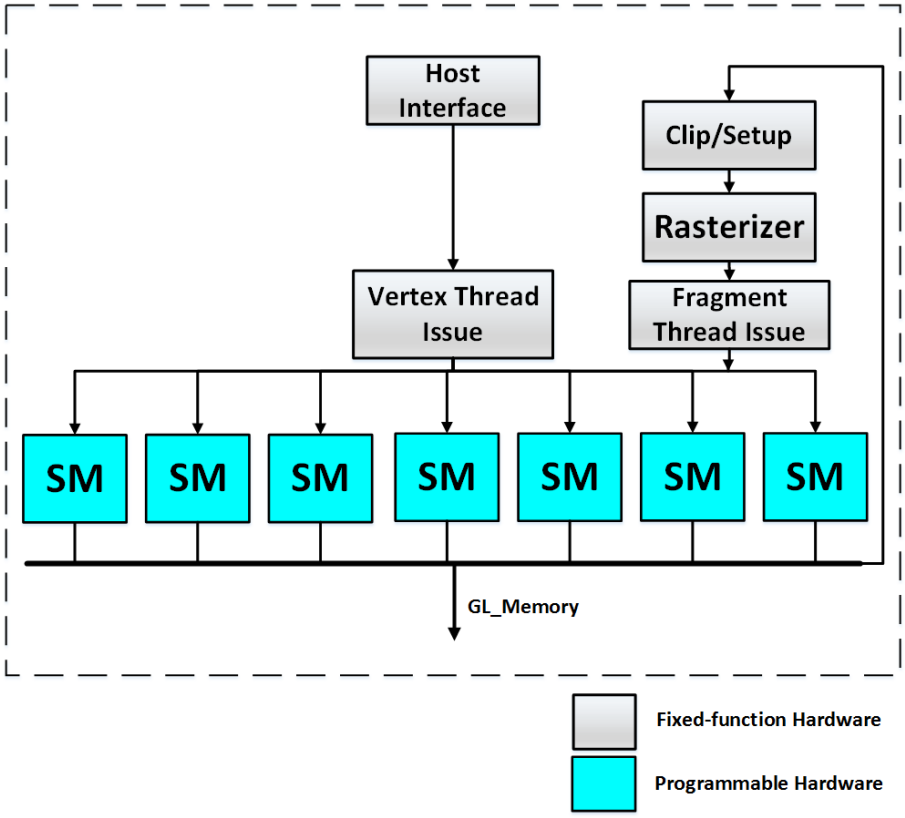


Figure 3.3: Programmable GPU with unified shader architecture

3.1.5 General-Purpose Computational GPU (GPGPU)

With the rapid evolution of GPUs from a fixed graphics processor to a programmable parallel processor, the GPUs become a many-core multiprocessor that carries out both graphics and general-purpose computations applications [24]. Nowadays, GPUs use hundreds of parallel processor cores to rapidly perform large amounts of parallel computations.

After the introduction of the fully programmable GPU in 2002, programmers looked forward for performing the general purpose computations on this programmable hardware. In 2003, a new era of GPU usage was started by introducing the DirectX 9. DirectX 9 allowed the programmers to carry out their non-graphics programs on the GPU by taking advantage of the programmable hardware. In 2006, the unified-shader GPUs improved the usage of GPU resources to carry out the non-graphics, general purpose computations. The GeForce 8800 was the first unified GPU that supports graphics and general-purpose computations. It could be programmed in C with CUDA for computing. From 2007 to 2009, NVIDIA released GeForce 8, 9, 100, 200, 300 series that were based on the Tesla architecture [25]. For example, GeForce GTX 280 is a unified graphics and computing

GPU that featured IEEE single precision floating point arithmetic. It supports CUDA, OpenGL, and DirectCompute.

From late 2009 to 2012, NVIDIA released the GeForce 300, 400, 500, and 600 series, the first GPU designed for the GPGPU computations. It is based on the Fermi architecture [26, 27]. Fermi's capacity for general-purpose computations has delivered great improvements in games and multimedia applications. However, Fermi architecture has delivered all features that were required for the high performance applications. It is compliant with IEEE 754-2008, the IEEE standard for floating point arithmetic that was released in 1985 and revised in 2008. Its addressing model is linear with caching at all levels. It supported OpenCL, CUDA, and DirectCompute. Also, it could be programmed using C++, Java, and MATLAB.

In 2013, NVIDIA released the GeForce 700 series based on the Kepler architecture. Kepler architecture mainly focused on improving the energy efficiency. For example, two Kepler cores consume 90% of one Fermi core power. NVIDIA presented Kepler GK110 as the most efficient high performance architecture ever built. Kepler GK110 achieved great improvement in power efficiency with respect to performance. It delivered up to 3x the performance per watt of Fermi. It supported DirectX 11, OpenGL 4.5 and CUDA 3.5. It helps to solve difficult computing problems for different areas such as the signal processing, biochemistry simulations, computer aided engineering and data analysis. In 2014, NVIDIA released the GeForce 900 series that were based on the Maxwell architecture. Maxwell architecture [28] could be considered as an improved version of the Fermi architecture. Finally, NVIDIA announced that Pascal GPU is the graphics architecture to come after the Maxwell architecture and is due to appear in 2016.

3.1.6 Accelerated Processing Unit (APU)

Accelerated Processing Unit (APU) is a processing unit that includes additional computational resources to accelerate the computations outside the CPU. Generally, it can include any programmable unit such as GPU or FPGA. It is usually implemented on the same die with the CPU. So, it improves the performance by speeding up the data transfer rates. It significantly reduces the power consumption. It is mainly used in AMD APUs [29], Intel HD Graphics, and Project Denver.

AMD released its first APU, named Brazos, in 2011. Brazos integrated the Bobcat CPU cores, TeraScale 2 GPU, and unified video decoder. Then, it released Liano, Trinity,

Kaveri, and Carrizo, subsequently. These APUs are mainly used for Sony PlayStation 4 and Microsoft Xbox. In 2013, Intel released its Iris Graphics APUs that were integrated with CPUs in the same die. In 2014, NVIDIA released Tegra K1 SoC pairing NVIDIA GPUs and ARM CPUs.

3.1.7 Future Micro-polygon Rendering GPU

Future graphics systems are mainly targeted rendering complex geometries and film quality images in real time. So, the GPUs have to deal with high-resolution surfaces represented by sub-pixel area micro-polygons. The current GPUs have two main limitations to deal with micro-polygons. First, GPUs require additional computational resources. This problem is overcome by the development of GPUs and the scaling of technology nodes. Second, the graphics pipeline units execute inefficiently with micro-polygons. In 2010, Fatahalian [16–18] introduced a real-time graphics pipeline to increase the efficiency of rasterization, tessellation, and shading of micro-polygon objects. He claimed this pipeline could satisfy the micro-polygon rendering in real time.

3.2 Embedded System GPUs

Mobile devices are quickly becoming our most valuable personal computers. Nowadays, we use the mobile devices to read email, browsing the Web, take photos, or play games. With this raising demand of the graphics applications such as games, the rapid development of embedded systems, and the enhancement of the functionality of the integrated circuits for the smaller technology nodes, the GPU become the essential part in each embedded system. However, the main target is the reasonable performance with low power since the embedded systems' applications are simpler and resolutions are smaller. For the embedded GPUs, the main corporations are QUALCOMM, NVIDIA, and ARM.

3.2.1 QUALCOMM GPUs

Qualcomm has over 42% of the total market for personal mobile devices [30]. Snapdragon [31] is its main family of mobile System-on-chip (SoC). It contains GPU, CPU, wireless hardware, and communication modem on the same die. The main advantages of the Snapdragon, over other embedded SoC, is the integration of the modem for cellular communications, Wi-Fi, and Bluetooth on the same die. Most Snapdragon SoCs are based on the Krait or Scropion CPU and Adreno GPU. Scropion and Krait are QUALCOMM's

own design, but they are based on the ARMv7 instruction set and their features are similar to ARM Cortex-A8 and ARM Cortex-A15 respectively. Adreno GPU is QUALCOMM's own IP.

In 2008, QUALCOMM released QSD8650, its first Snapdragon chip. It was based on the Adreno 200 GPU and the Scropion CPU. It was supported OpenGL ES 2, OpenVG 1.1, and Direct3D 9.0c. In 2012, QUALCOMM released MSM8227. It was based on the Adreno 305 GPU and Krait CPU. It supported OpenGL ES 1.1, OpenVG 1.1, and DirectX 11.1. In 2013, QUALCOMM released Snapdragon 200, 400, 600 and 800 series. They were based on the Cortex A5 and A7 CPU. And they were based on Adreno 305 and 405 GPUs.

3.2.2 ARM GPUs

As the ARM Cortex CPU, ARM produces the Mali GPUs [32] as IP core to be used in ASIC design by ARM partners. So, it provides an optimized heterogeneous platform that uses the Cortex CPU and Mali GPU. Mali was used by various vendors such as MediaTek, Samsung, and STMicroelectronics. Mali has two architectures: the Utgard and Midgard. Utgard is a non-unified GPU. It has discrete fragment and vertex shaders while Midgard is unified GPU.

Mali-55 is the first and smallest Mali GPU. It supports the OpenGL ES 1.1. In 2007, ARM released the Mali-200 which was based on the Utgard architecture and it is fabricated using the 65nm technology. It supports the OpenGL ES 1.1, OpenGL 2.0, and OpenVG 1. Then, ARM released the Mali-300, Mali-400, and Mali-450 which were based on the Utgard architecture and supports the OpenGL ES 1.1 and OpenGL ES 2.0. Then, ARM released the Mali-604, the first Midgard architecture GPU. It supports the OpenGL ES 3.0 and OpenCL 1.1. In 2012, ARM released the Mali-624 as the second generation Midgard GPU. It was an embedded graphics and GPU Compute accelerator that enables the development of gaming capabilities and GPU Compute applications. Then, ARM released the Mali-T720 and Mali-T820 as third and fourth Midgard architecture GPUs, respectively.

3.2.3 NVIDIA GPUs

In 2008, NVIDIA started to release its Tegra series SoC for mobile devices. NVIDIA integrated ARM Cortex A9 and Cortex A15 with its GPUs. In 2008, it released the Tegra APX 2500, the first SoC of the Tegra family. In 2010, NVIDIA released the Tegra 2 series. It integrated the ARM Cortex A9 CPU with ultra-low power GeForce GPU. It supports the OpenGL ES 2.0 and OpenVG 1.1. In 2013, NVIDIA released Tegra 4 series. It integrated the Cortex A15 CPU with the Tegra 4 processor. Tegra 4 [33] processor has 72 programmable cores and supports the main features of the OpenGL ES 3.0.

In 2014, NVIDIA released Tegra K1 [34] series; its first series designed for the general-purpose computing on the embedded SoC. NVIDIA integrated its Denver CPU with Kelper GPU which has 204 cores. It supports the OpenGL ES 3.1, DirectX 11, and CUDA 6.5. Early 2015, NVIDIA released its Tegra X1 [35] series. It integrated the Cortex A57 processor and Maxell-based GPU that has 256 cores. It supports OpenGL 4.5, DirectX 12, and CUDA 6.0. It performs more than 500 GFLOPS for the 32-bit workloads.

3.3 ARM Mali-200

ARM presented Mali-200 [36] as the popular GPU that supports the OpenGL ES 2.0 API. It is a fully programmable architecture that supports the 3D graphics using OpenGL ES 2.0 and OpenGL ES 1.1 APIs. Also, it supports the 2D graphics using the OpenVG 1.1. Mali-200 is used in many application areas such as mobile internet devices and navigation.

Mali-200 supports 4x and 16x multi-sampling. It uses the AMBA AXI which is compatible with many buses and peripheral IPs. It was implemented at the TSMC 65 nm technology node. It scores post-layout area of 4.1 mm^2including all memories. The maximum frequency is 230 MHz for the low power (LP) option and 380 MHz for the general purpose (GP) option. Mali-200 throughput is 16 million triangles/second and 275 million pixel/second at the operating frequency 275 MHz.

We consider Mali-200 results as a general relaxed guide for us because it was implemented at 65 nm technology node. However, it supports the OpenGL ES 2.0, OpenGL ES 1.1, and OpenVG whereas our CUGPU supports the Open GL ES 1.1 CL profile only because our main targets are the minimum cost and simplicity of the design.

Chapter 4

CUGPU Architecture

We developed our CUGPU architecture that is based on OpenGL ES 1.1 [8, 15] Common-Lite profile. We implemented the mandatory commands and specifications only to minimize the power and area without deteriorating its performance. Our CUGPU specifications are:

1. It provides high-performance support of the fixed-function 3D graphics pipeline. All graphics operations are performed in the graphics hardware.
2. It supports the Common-Lite (CL) profile only. So, all computations are performed in fixed-point format.
3. It supports two texture units only. Model-view stack size is sixteen matrices while projection and texture units stack sizes are two matrices each.
4. It supports 4x multisampling and eight light sources at maximum.

As we considered Mali-200 as a general relaxed guide for our design, CUGPU targets a post-layout area of $2\,mm^2$ and a maximum frequency of 200 MHz at the TSMC 65nm LP technology. It supports the VGA resolution,$640\,x\,480$ with throughput of 12 M tri/sec and 200 M Pixel/sec at 200 MHz.

Figure 4.1 displays the overall system-on-chip architecture. The CPU communicates with the CUGPU directly through the Wishbone bus. It sends commands to control the graphics state and to supply the primitives' data to draw. All graphics operations, in our system, are performed on the GPU. It has access to the system memory to fetch the primitives' data and to copy the texture (images) from the system memory to the graphics memory. Also, it has a dedicated fast memory (graphics memory) to supply the CUGPU

units with their data, to store the texture data and frame buffer arrays. In addition, it has direct access to the display through the memory arbitration.

Figure 4.2 shows the architecture of CUGPU block diagram. Our methodology Focuses on:

1. CUGPU performs as much as possible operations out of the graphics pipeline critical path. The matrix construction, texture handling, and data fetch units carry out some operations outside the critical path. The matrix construction constructs the matrices for the vertex processing, clipping, and lighting offline. The texture handling creates the mipmapping copies while the objects are processed. Also, the data fetch unit create the buffer objects and copy their data from the vertex arrays.

2. CUGPU performs graphics operations as soon as possible. For example, the lighting unit applies the shading model on the vertices' colors regardless of the clipping outcome. Then, the post-clipping unit calculates the final color based on the clipping outcome.

3. Each CUGPU unit carries out operations that have the same nature and it has its distributed register file. So, we can optimize this hardware.

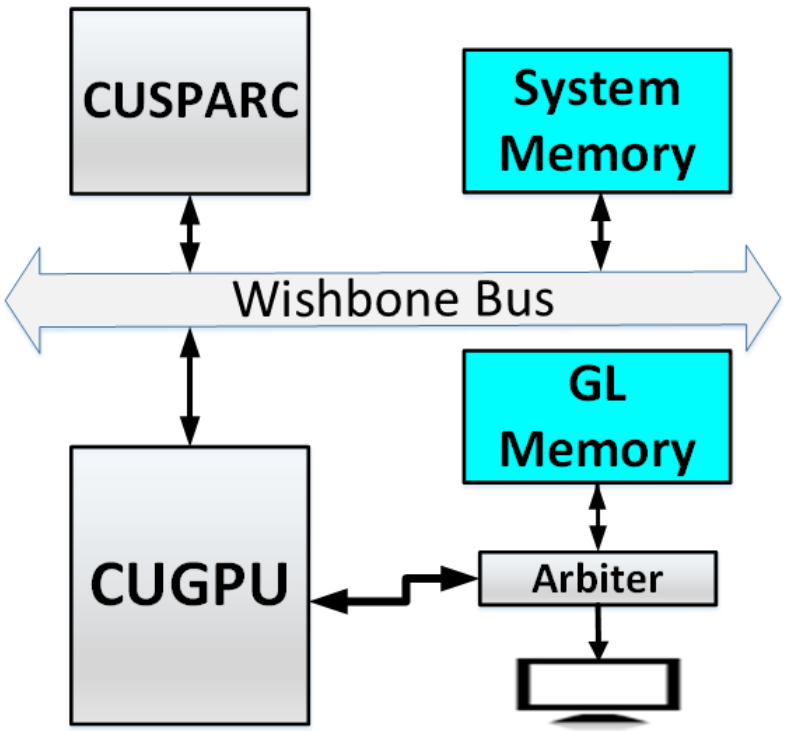


Figure 4.1: the overall SoC architecture

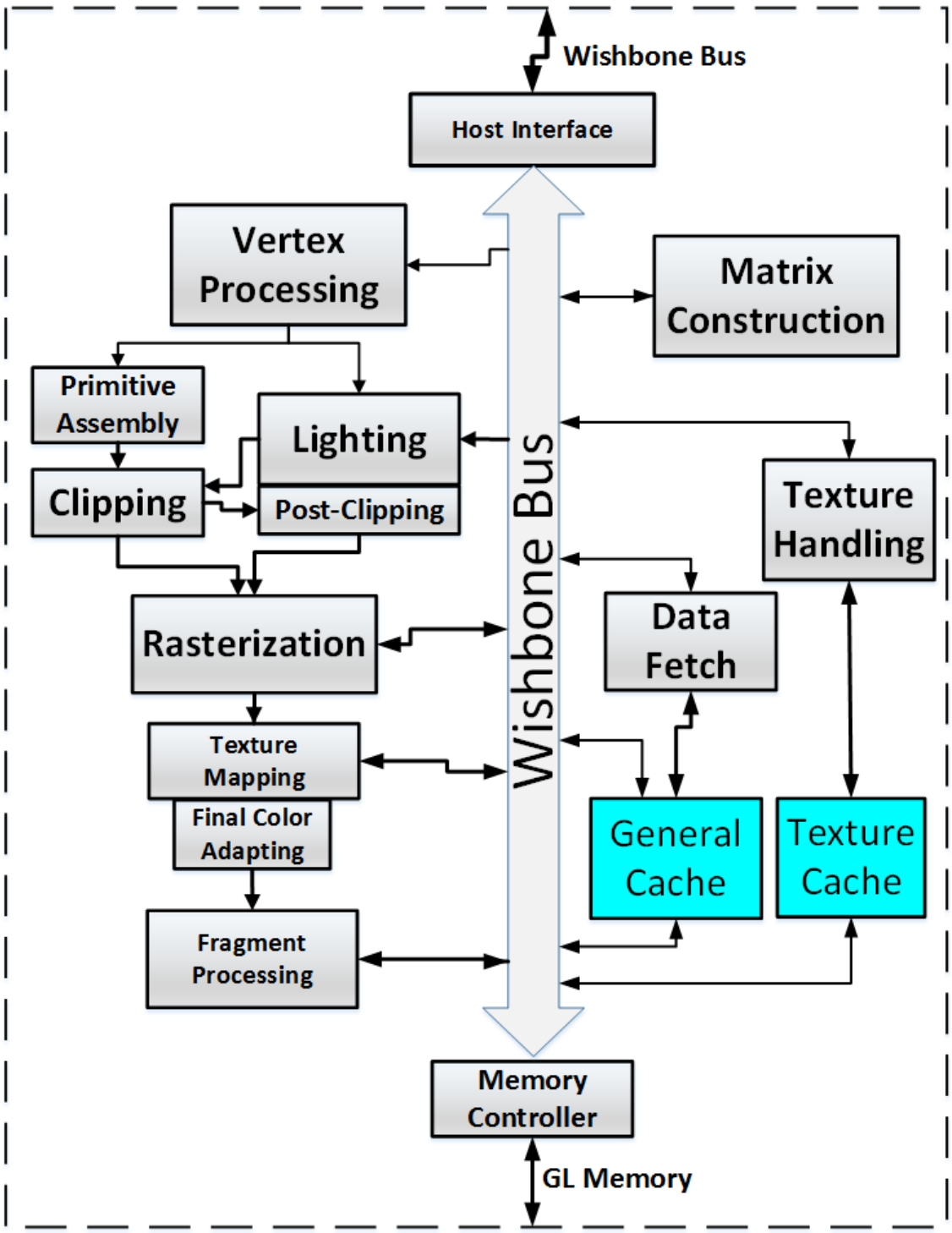


Figure 4.2: CUGPU block diagram

Instructions that are received by the graphics hardware are processed by the host interface and are then placed on internal Wishbone bus to the targeted sub-component. Also, there is a direct connection between pipeline stages and host interface for receiving configuration instructions. One sub-component on the internal Wishbone bus is a data fetch component which is responsible for fetching data from the system memory and the graphics memory, creating the buffer objects, and controlling the vertex arrays and buffer objects. The texture handling unit is used for creating texture copies and controlling them within the graphics memory space.

The data fetch unit sends primitive data to the vertex processing unit which performs vertex model-view and projection transformation, normal transformation, and texture coordinates transformation on the fly. The matrix construction is used to construct the transformation matrices offline. The output from the vertex processing unit is fed to the primitive assembly and lighting units. The lighting calculates the vertex color based on the given light sources when the primitive assembly constructs the primitives from the supplied vertices based on the drawing mode (point, line …etc.). Then, the primitive vertices are fed to the clipping unit which checks them against the user-defined planes. If a new vertex is introduced, the post-clip operations unit calculates its associated data (color, depth, texture coordinates) according to its location.

Then, the vertices and their associated data are fed to the rasterization unit which is the main block in the graphics pipeline. It determines the fragments that intersect with the drawing primitive and interpolates their associated data. Then, if the texture mapping is enabled, the texture mapping unit calculates the final color of the fragment based on the incoming color and the corresponding texture color. Then, final color adapting unit performs the fog, anti-aliasing, and multisampling operations to obtain the final fragment colors. Finally, the fragment processor performs blending, stencil test, depth test, and other tests on the incoming fragments and updates the frame buffer. In the following subsections, each unit is explained in particular. We describe its function, its register file, and the detailed proposed architecture and algorithms. Appendix A summarizes the supported OpenGL ES 1.1 commands for each unit. These commands are divided into groups based on the corresponding units.

4.1 Data Fetch Unit

The data fetch unit is responsible for fetching primitive vertices and their associated data from the vertex arrays in the client memory (system memory) and from the buffer objects in the server memory (graphics memory). Also, it creates the buffer objects to store the vertex arrays and element indices data and bind them to the corresponding arrays. In addition, it is used to store the current color, normal, and the texture coordinates values to be used when the corresponding array is disabled. There are 17 commands that affect this unit operation. These commands are divided into three categories: vertex arrays commands, drawing commands, and buffer objects creation and binding commands. This unit handles the required functions set by OpenGL ES 1.1 [8], sections 2.7, 2.8, and 2.9.

4.1.1 Register File

1. Array buffer table: it contains buffer's name, size, location at graphics memory, and usage. It also contains the name of the bounded array to array buffer.

2. Element array buffer table: it contains element buffer's name, size, location at graphics memory, and usage. It also contains the name of the bounded array to element array buffer.

3. Color table: it contains color array enable bit and the RGBA current color. Also, it contains the color array's type, size, stride, pointer, and color array bounded buffer name. The color array enable is initially false and the initial current color is $(1,1,1,1)$. The initial array type is fixed-point and the initial size is 4. The initial stride is zero and the initial pointer is Null. The initial bounded buffer is buffer 0.

4. Normal table: it contains normal array enable bit and the current normal. Also, it contains the normal array's type, stride, pointer, and normal array bounded buffer name. The normal array enable is initially false and the initial current normal (n_x, n_y, n_z) is $(0,0,1)$. The initial array type is fixed-point and the initial stride is zero. The initial pointer is Null. The initial bounded buffer is buffer 0.

5. Point size table: it contains point array enable bit and the current point size. Also, it contains the point size array's type, stride, pointer, and point array buffer binding name. The point array enable is initially false and the initial point size is 1.0. The initial array type is fixed-point and the initial stride is zero. The initial pointer is Null and the initial bounded buffer is buffer 0.

6. Texture array table: it contains texture array enable bits, 1-bit for each texture unit, and two-valued integer represents the active texture. It includes the current texture coordinates (s,t,r,q). Also, it contains, for each texture unit, the texture array's type, stride, size, pointer, and texture array bounded buffer name. The texture array enable is initially false for all texture units, the active texture value is initially 0, and the initial current texture coordinates is $(0,0,0,1)$. The initial array type is fixed-point, the initial stride is zero, and the initial size is 4. The initial pointer is Null and the initial bounded buffer is buffer 0.

7. Vertex array table: it contains the vertex array's type, stride, size, pointer, and vertex array bounded buffer name. The initial array type is fixed-point, the initial stride is zero, and the initial size is 0. The initial pointer is Null and the initial bounded buffer is buffer 0.

8. Drawing parameters: it contains 8-valued integer represents the drawing mode, the first integer refers to the location of the first element to be drawn, the count represents the object size, indices pointer refers to the locations of drawn sub-array elements, and indices type.

4.1.2 Algorithm

Data fetch unit handles the input commands according the algorithm 4.1.

4.2 Matrix Construction Unit

The matrix construction unit is responsible to construct the model view matrix M, projection P and view-port matrix V for transforming the vertices coordinates. Also, it constructs the normal matrix M_u^{-1}, the inverse of the upper left $(3x3)$ part of the M matrix for transforming the normal. Also, it constructs $M^{-1}$and P^{-1} for transforming the clipping-planes coefficients. In addition, it handles Load, Mult, Rotate, Scale, Frustum, Ortho, Push, and Pop commands. There are 14 GL commands that affect this unit operation. This unit handles the required functions set by OpenGL ES 1.1 [8], section 2.10.

4.2.1 Register File

1. Active texture unit selector: 2-valued integer, initially texture0 is selected.

2. Current matrix mode selector: 4-valued integer, initially model view matrix is selected.

3. Current model view matrix: one $4x4$ fixed-point matrix and Model-view stack pointer. The model view stack size is 16 matrices and the current model view matrix is initially the identity matrix.

4. Current projection matrix: one $4x4$ fixed-point matrix and projection stack pointer. The projection stack size is 2 matrices and the projection matrix is initially the identity matrix.

5. Current texture matrix: for each texture unit, there is one $4x4$ fixed-point matrix for and texture stack pointer. The texture stack size is 2 matrices for each texture unit and the texture matrix is initially the identity matrix.

6. Current viewport matrix: four-integer values represent the origin (O_x, O_y) and the viewport's width and height (P_x, P_y) and two clamped fixed-point values represent the factor and offset (n, f) applied to the depth value. Initially $(O_x = w/2, O_y =$

Algorithm 4.1 Data fetch algorithm

```
If (command ∈ [Color4, Normal3, MultiTexCoord4, PointSizeX]) Then
        →Update the corresponding current value.
ElseIF (command ∈ [ClientaActiveTexture, EnableClientState, DisableClientState]) Then
        →Update the corresponding control bit.
ElseIF (command = BindBuffer) Then
        If (buffer name = 0) Then
                →Unbind the bounded buffer
        ElseIF (buffer name exists) Then
                →Bind this buffer.
        Else
                →Create a new buffer entry and bind it.
ElseIF (command ∈ [BufferData, BufferSubData]) Then
        If (the bounded buffer name ~= 0)
                →Update the bounded buffer by transferring data from
                  the system memory to the graphics (GL) memory.
ElseIF (command ∈ [VertexPointer, NormalPointer, ColorPointer, PointSizePointerOES, TexCoord-
Pointer]) Then
        →Update the corresponding array table parameters
        If (the bounded buffer name ~= 0)
                →Bind the bounded buffer object to its relative array.
ElseIF (command ∈ [DrawArrays, DrawElements]) Then
        →Fetch the vertices coordinates, colors, texture coordinates,
          point size, normal from their corresponding arrays and their
          bounded buffer objects if these arrays are enabled.
        →Supply the pipeline by these data, element by element, if these
           arrays are enabled; or by the current values otherwise.
```

$h/2, P_x = w, P_y = h$) where w and h are the width and height of the screen respectively.

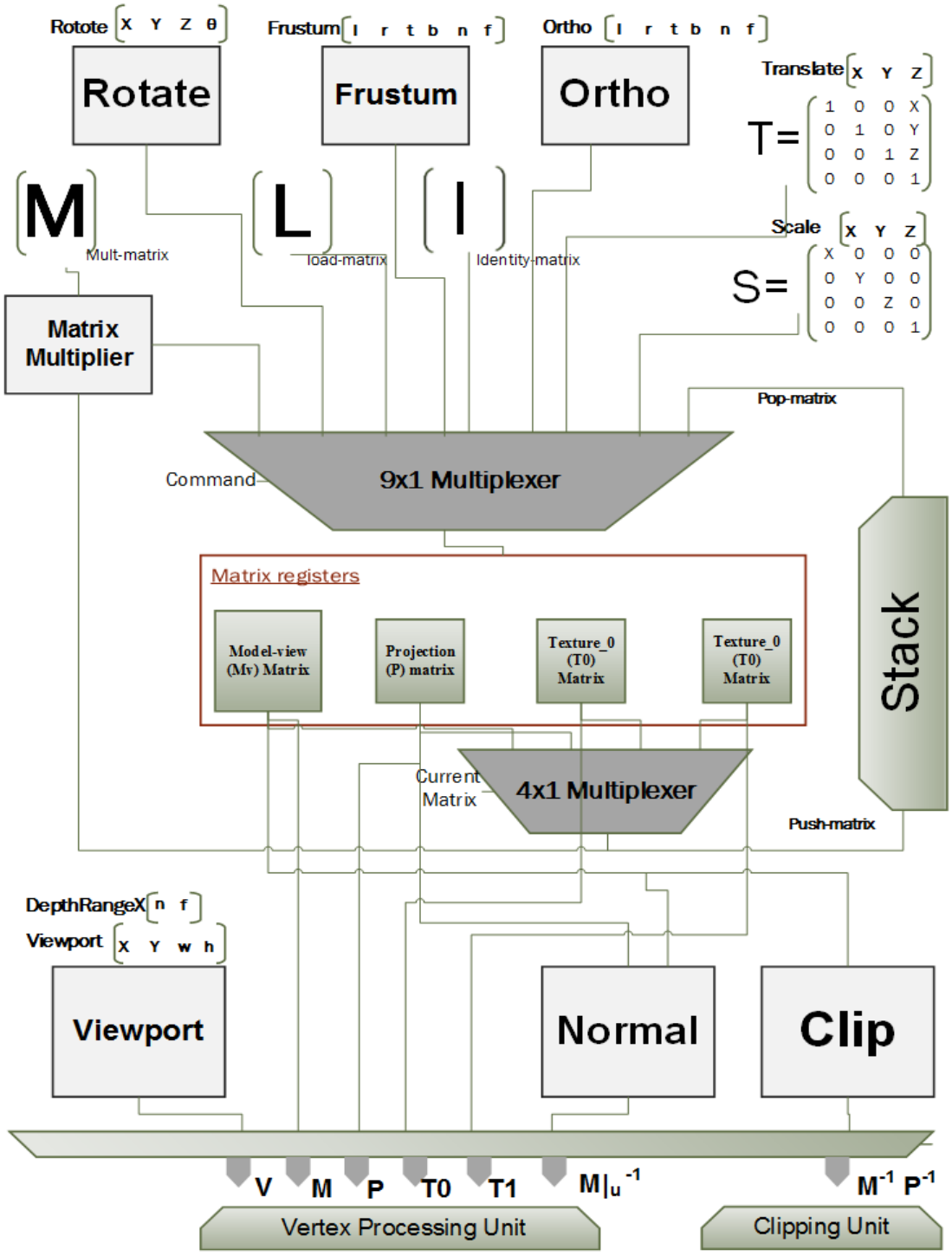


Figure 4.3: The matrix construction block diagram:

4.2.2 Architecture

Figure 4.3 shows the whole matrix construction unit architecture. This is responsible for constructing the matrices $(V, M, P, T_0, T_1, M_u^{-1})$ for the vertex processing and lighting units; and the matrix (M^{-1}, P^{-1}) for the clipping unit. These matrices are constructed by 9 different ways based on the input command. The current matrix is loaded directly by the identity matrix I, load matrix L, translate matrix T, and scale matrix S if the command is LoadIdentity, LoadMatrixx, Translatex, and Scalex respectively. If the command is Pop, the current matrix is loaded by the second matrix in this current matrix stack. Also, if the command is MultMatrixx, the current matrix is multiplied by the matrix M. If the command is Frustumx, the current matrix is loaded by the matrix f from the frustum unit:

$$f = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2f.n}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4.1}$$

If the command is Orthox, the current matrix is loaded by the matrix o from the ortho unit:

$$O = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

where r, l, t, b, n, *and* f are the right, left, top, bottom, near, and far boundaries of the view volume, respectively. If the command is Rotatex by $(\ominus, x, y, z)$, the current matrix is loaded by the matrix R_4 from the rotate unit:

$$R_4 = \begin{bmatrix} & & & 0 \\ & R_3 & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.3}$$

$$R_3 = U * U^T + Cos(\ominus) * (I - U * U^T) + Sin(\ominus) * S \tag{4.4}$$

$$S = \begin{bmatrix} 0 & -z` & y` \\ z` & 0 & -x` \\ -y` & x` & 0 \end{bmatrix} \tag{4.5}$$

$$U = v/\|v\| = \begin{Bmatrix} x` \\ y` \\ z` \end{Bmatrix} \tag{4.6}$$

$$v = \begin{Bmatrix} x \\ y \\ z \end{Bmatrix} \tag{4.7}$$

The normal unit constructs the M_u^{-1} matrix to transform the normal to the eye space. Also, the clip unit constructs the (M^{-1}, P^{-1}) for transforming the clip-plane coefficient to the clip-space. The viewport constructs the viewport matrix V

$$V = \begin{bmatrix} \frac{p_x}{2} & 0 & 0 & o_x \\ 0 & \frac{p_y}{2} & 0 & o_y \\ 0 & 0 & \frac{f-n}{2} & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.8}$$

where n and f are the factor and offset applied to the depth (Z), respectively.

4.3 Vertex Processing Unit

The vertex processing unit is responsible for transforming the vertices coordinates (x, y, z), the texture coordinates (s, t, r, q), and the normal (n_x, n_y, n_z). The incoming vertices coordinates are transformed from the world space to the eye space to be used in lighting unit. Also, they are transformed to the clip space to feed the clipping unit. The normal is transformed to eye space to be used in lighting unit. For each texture unit, the texture coordinates are multiplied by the corresponding texture matrix. This unit handles the required functions set by OpenGL ES 1.1 [8], section 2.10.

4.3.1 Register File

1. Normal mode: 2-bits to select between OFF, RESCALE_NORMAL, NORMALIZE, or RESCALE&Normalize.

2. Current normal: 3 fixed-point values (n_x, n_y, n_z) that stored the current normal.

3. Current texture coordinates: 4 fixed-point values (s, t, r, q) for each texture unit.

4. Current values enable: 3-bits; one for the current normal and one for each current texture coordinates.

4.3.2 Architecture

Figure 4.4 shows the whole vertex processing unit architecture. The vertex coordinates v, normal n, and texture coordinates (t_1, t_2) are supplied from the data fetch unit. The matrix construction unit feed the vertex processing unit with the matrices $(M, P, T_0, T_1, M_u^{-1})$. The vertex coordinates (v) are transformed to the eye space by multiplying by the model view matrix M. Then, they are transformed to the clip space by multiplying by the projection matrix P. If the current texture is disabled, the texture coordinates (t_0, t_1) are multiplied by the texture matrices $T_0$and T_1, respectively. Otherwise, the current texture coordinates are used instead. If the current normal is disabled, the normal (n_x, n_y, n_z) is transformed to the eye-space by multiplying by the inverse of the model view matrix M_u^{-1}. If the current normal is enabled, the current normal is used instead. Then, the resulting normal (n_x', n_y', n_z') is modified according the normal algorithm 4.2.

4.4 Primitive Assembly Unit

The primitive assembly unit constructs primitives from the incoming vertices according to the mode argument which is set by the DrawArray or DrawElements commands. This unit handles the required functions set by OpenGL ES 1.1 [8], section 2.6.1.

Algorithm 4.2 Normal algorithm

If (Rescale =1)

(nx'',ny'',nz'')=(nx',ny',nz')*$\frac{1}{\sqrt{m_{31}^2+m_{32}^2+m_{33}^2}}$

Else

(nx'',ny'',nz'')=(nx',ny',nz')

If (Normalize =1)

(nx''',ny''',nz''')=(nx'',ny'',nz'')*$\frac{1}{\sqrt{n_{31}^2+n_{32}^2+n_{33}^2}}$

Else

(nx''',ny''',nz''')=(nx'',ny'',nz'')

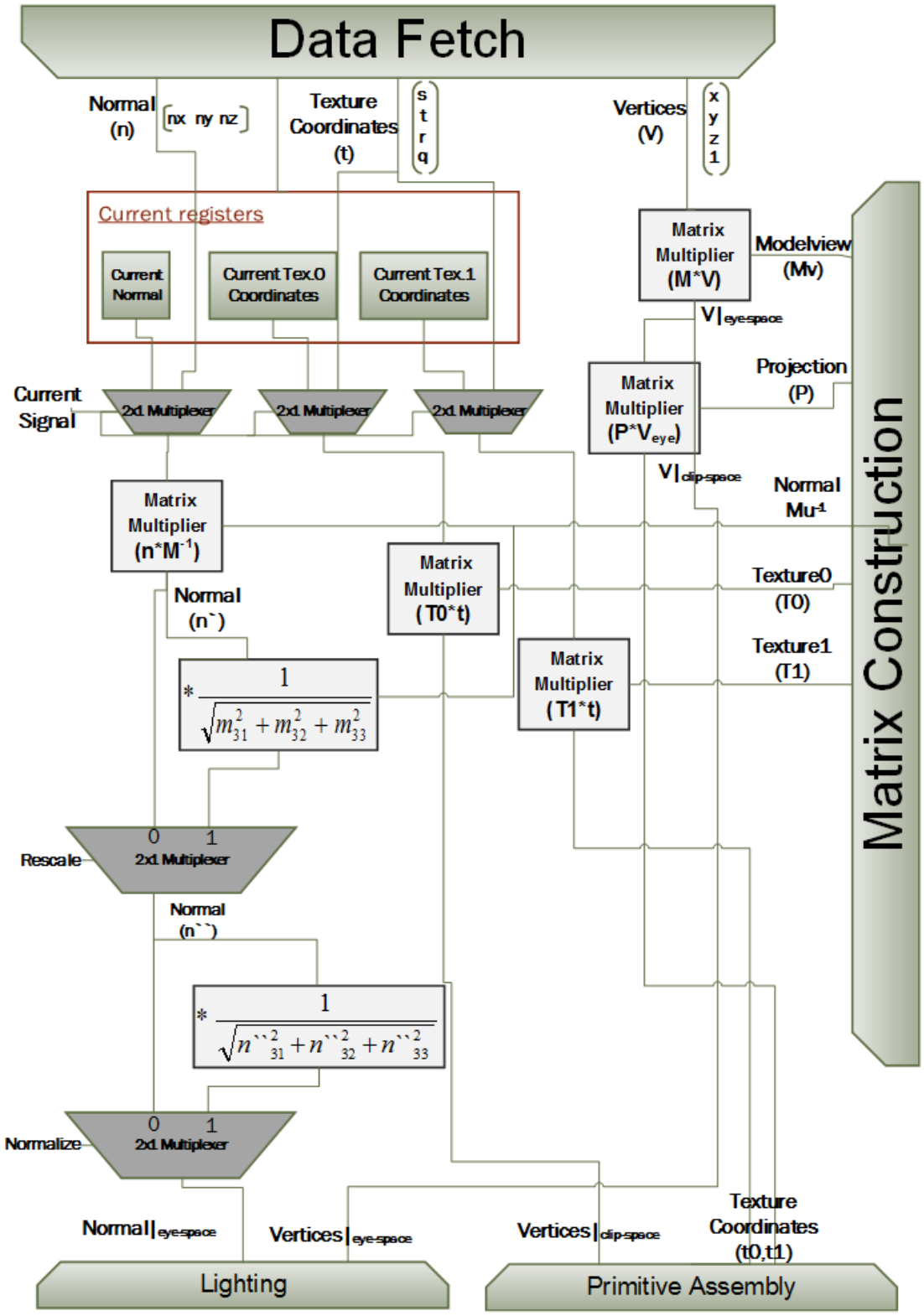


Figure 4.4: Vertex processing block diagram

4.4.1 Register File

1. Draw-mode: 7-valued integer that specifies the primitive type: points, line strips, line loops, lines, triangle strips, triangle fans, and separate triangle.
2. First vertex flag: 1-bit that indicates if the incoming vertex is the first vertex in the object or not. Initially, the first-vertex-flag is 1.
3. Vertex index: 1-bit that indicates if the incoming vertex order is odd or even vertex. Initially, this flag is odd.
4. First Triangle flag: 1-bit that indicates if the first triangle is completed or not.
5. Triangle vertices counter: 2-bits that determine the triangle vertices order. This counter initial value is 1.
6. Registers A, B: two temporary registers to store the incomplete triangle vertices.
7. Replacement pointer: 1-bit that refers to the register to be replaced by the incoming vertex if the draw-mode is triangle strips. Initially, this refers to register A.

4.4.2 Algorithm

Data fetch unit handles the input commands according to algorithm 4.3.

4.5 Lighting Unit

The lighting unit is responsible for calculating the pre-clipping front and back colors for all vertices. All lighting operations should be performed in the eye-space. So, the light sources location and direction are transformed to the eye-space first. All lighting computations are performed in the fixed-point format. So, all vertices colors, normal, and coordinates are transformed to the fixed point format. Our design supports 8 light sources. This unit handles the required functions set by OpenGL ES 1.1 [8], section 2.12.

4.5.1 Register File

1. Lighting enable: 1-bit to enable/disable the lighting. Initially, it is disabled.
2. Color tracking: 1-bit to enable/disable the material color tracking of the incoming vertex color. Initially, the color tracking is disabled.

Algorithm 4.3 Primitive assembly algorithm

```
If (Draw_mode = points)
        → The incoming vertex (v) belongs to new primitive.
ElseIF (Draw mode = Line Strips)
        → If (first vertex flag =1)
                - First vertex flag = 0
                - Register A = incoming vertex.
          Else
                - Assemble line from the incoming vertex and register A vertex.
                - Register A = incoming vertex.
ElseIF (Draw mode = Line Loops)
          If (first vertex flag =1)
                - First vertex flag = 0
                - Register A = incoming vertex.
          Else - Assemble line from register A vertex and the incoming vertex.
ElseIF (Draw mode = Lines)
          If (vertex index = odd)
                - Register A = incoming vertex.
          Else
                - Assemble line from register A vertex and the incoming vertex.
ElseIF (Draw mode =Triangle strips)
          If (first triangle = 1)
                If (Triangle vertices counter =1)
                        - Register A = incoming vertex.
                ElseIF (Triangle vertices counter =2)
                        - Register B = incoming vertex.
                        - first triangle = 0.
          Else
                - Assemble triangle from register A vertex, registerB
                 vertex and the incoming vertex.
                - Store the incoming vertex in the register which the
                 pointer is referred to.
                - Toggle the pointer.
ElseIF (Draw mode =Triangle fans)
          If (first triangle = 1)
                If (Triangle vertices counter =1)
                        - Register A = incoming vertex.
                ElseIF (Triangle vertices counter =2)
                        - Register B = incoming vertex.
                        - first triangle = 0.
          Else
                - Assemble triangle from register A
                 vertex, register B vertex and the incoming vertex.
                - Store the incoming vertex in the register B.
ElseIF (Draw mode = Separate Triangle)
          If (first triangle = 1)
                If (Triangle vertices counter =1)
                        - Register A = incoming vertex.
                ElseIF (Triangle vertices counter =2)
                        - Register B = incoming vertex.
          Else
                - Assemble triangle from register A vertex, register B vertex and the
                 incoming vertex.
```

3. Current-Color: RGBA value used as a vertex color if the current color is enabled.

4. Material & Light Model parameters: table 4.2 summarizes the material parameters.

5. Light sources parameters: table 4.1 summarizes the light sources parameters and their initial values for each light source i where $i \in [0,7]$.

6. Current-Color Enable: 1-bit to enable the current-color.

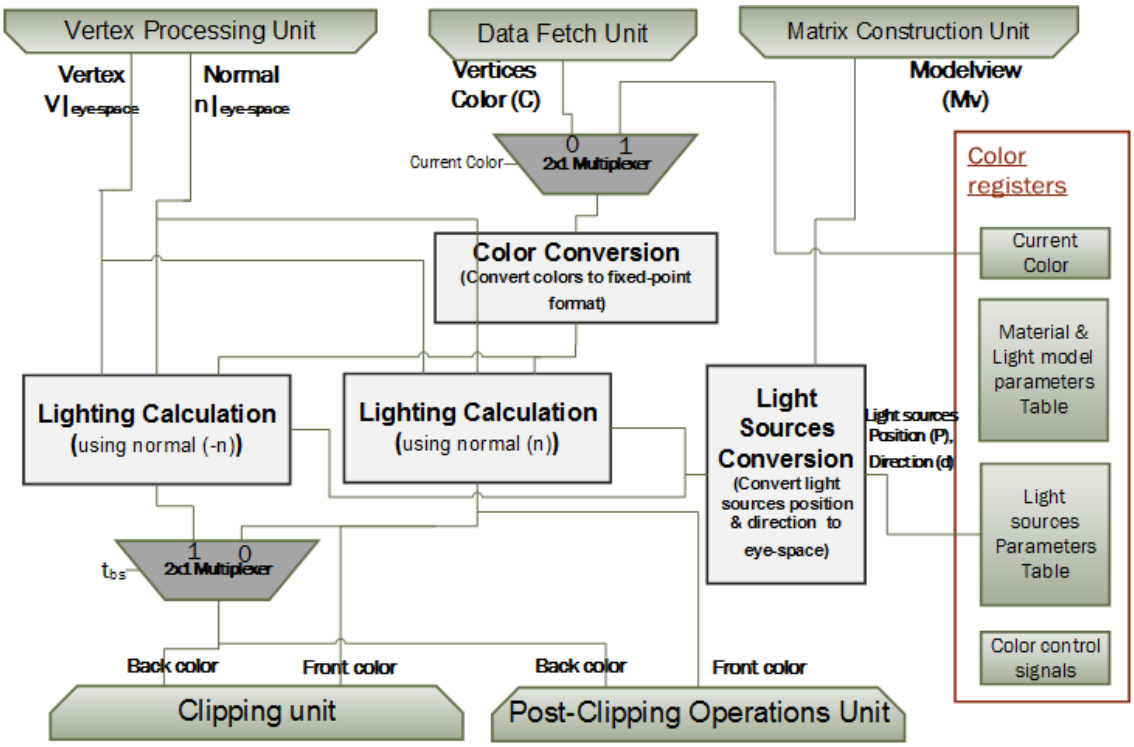


Figure 4.5: The lighting unit architecture

Parameter	Type	Initial value	Description
A_{cm}	color	$(0.2, 0.2, 0.2, 1)$	ambient color of material
D_{cm}	color	$(0.8, 0.8, 0.8, 1)$	diffuse color of material
S_{cm}	color	$(0, 0, 0, 1)$	specular color of material
E_{cm}	color	$(0, 0, 0, 1)$	emissive color of material
S_{rm}	real	0	specular exponent
A_{cs}	color	$(0.2, 0.2, 0.2, 1)$	ambient color of scene
T_{bs}	Boolean	FALSE	two-sided lighting mode enable

Table 4.2: Materials and light models parameters

Parameter	Type	Initial value	Description
A_{cli}	color	$(0,0,0,1)$	ambient intensity of light i
$D_{cli}(i=0)$	color	$(1,1,1,1)$	diffuse intensity of light 0
$D_{cli}(i>0)$	color	$(0,0,0,1)$	diffuse intensity of light i
$S_{cli}(i=0)$	color	$(1,1,1,1)$	specular intensity of light 0
$S_{cli}(i>0)$	color	$(0,0,0,1)$	specular intensity of light i
P_{pli}	position	$(0,0,1,0)$	position of light$i$
S_{dli}	direction	$(0,0,-1)$	direction of spotlight for light$i$
S_{rli}	real	0	spotlight exponent for light$i$
C_{rli}	real	180	spotlight cutoff angle for light$i$
K_{0i}, K_{1i}, K_{2i}	real	$(1,0,0)$	attenuation factors for light$i$

Table 4.1: Light source's parameters

4.5.2 Architecture

Figure 4-5 shows the whole lighting unit architecture. The vertex coordinates v and normal n, in the eye-space, are supplied from the vertex processing unit. The matrix construction unit feed the lighting unit with the model view matrix M that is used to transform the light sources positions and directions to the eye-space. The current color signal selects the associated color to the incoming vertex. If the current color is enabled, the current color is assigned to the incoming vertices. Otherwise, the color supplied form the data fetch unit is used instead. All lighting operations should be performed in the fixed-point format. So, the color conversion unit converts the color to the fixed point format if it is encoded in the unsigned binary format. Also, the lighting operations are performed in the eye-space. So, the light source conversion block transforms the light sources position p_i to the eye space by multiplying by the model view matrix M. In addition, it transforms the light sources directions by multiplying by the upper left $3x3$ matrix of the model view matrix. Then, the front color is calculated by the lighting calculation block using the normal n. The lighting produces a color c by the following equation

$$\begin{aligned} C &= E_{cm} + A_{cm} * A_{cs} \\ &\quad + \sum_{i=0}^{7} (att_i) * (spot_i) * [A_{cm} * A_{di} \\ &\quad + (n \odot \overrightarrow{VP_{pli}}) * D_{cm} * D_{cli} + f_i * (n \odot \hat{h}_i)^{S_{rm}} * S_{cm} * S_{cli}] \end{aligned} \tag{4.9}$$

$$att_i = \begin{cases} \frac{1}{K_{0i}+K_{1i}\|VP_{pli}\|+K_{1i}\|VP_{pli}\|^2} & if\ P'_{pli}sw \neq 0 \\ 1.0 & otherwise \end{cases} \tag{4.10}$$

$$spot_i = \begin{cases} (\overrightarrow{P_{pli}V} \odot \hat{S}_{dli})^{S_{rli}} & C_{rli} \neq 180, \overrightarrow{P_{pli}V} \odot \hat{S}_{dli} \geq Cos(C_{rli}) \\ 0 & C_{rli} \neq 180, \overrightarrow{P_{pli}V} \odot \hat{S}_{dli} < Cos(C_{rli}) \\ 1 & C_{rli} = 180 \end{cases} \tag{4.11}$$

$$f_i = \begin{cases} 1 & if\ n \odot \overrightarrow{VP_{pli}} \neq 0 \\ 0 & otherwise \end{cases} \tag{4.12}$$

$$h_i = \overrightarrow{VP_{pli}} + (\ 0 \quad 0 \quad 1\ )^T \tag{4.13}$$

Another color (temp_color) is calculated using the normal $-n$. The two-sided lighting mode T_{bs} determines if the back color is calculated using the normal n or the normal $-n$. If T_{bs} is0, the back color is the same as the front color. Otherwise, the back color is the temp_color. Finally, the front and back colors are sent to the clipping and the post-clipping operations units.

4.6 Clipping Unit

The clipping unit is responsible for checking the incoming primitives against the view volume and against the user-defined clip planes. Our design is implemented based on the Cohen-Sutherland [37] or Kodituwakku [38] line clipping algorithms.

It determines whatever this primitive is discarded, this primitive is passed to the rasterization block without any modification, or new primitives are produced based on the intersection between the incoming primitive and the view volume and the user-defined clip planes. If a new primitive is produced, an interpolation value t is calculated for each new vertex. This is used to compute the associated data, color and texture coordinates, for this new vertex from the incoming vertices associated data. This unit handles the required functions set by OpenGL ES 1.1 [8], sections 2.11 and 2.12.7.

4.6.1 Register File

1. Clip-planes table: for each clip plane, it stores its name, its coefficient (P1, P2, P3, P4), and its enable bit. Initially, this table is empty.

2. Draw mode: - 2-bits to specify the incoming primitive type (triangle, line, point).

4.6.2 Architecture

Figure 2.3 presents the clipping unit architecture. First, the incoming primitive's coordinates are fed to the view volume clipping unit. It applies the Cohen view-volume algorithm 4.5 to check this primitive coordinates (X_c, Y_c, Z_c) against the view volume that is defined by:

$$\begin{aligned} -W_c \leq X_c \leq W_c \\ -W_c \leq Y_c \leq W_c \\ -W_c \leq Z_c \leq W_c \end{aligned} \tag{4.14}$$

Then, the view volume clipping unit sends the resulting primitives to the user-defined clipping unit. The user-defined clipping unit checks the primitives against each enabled clip plane by the user clipping algorithm 4.7. This clip planes coefficients $(P_1 P_2 P_3 P_4)$

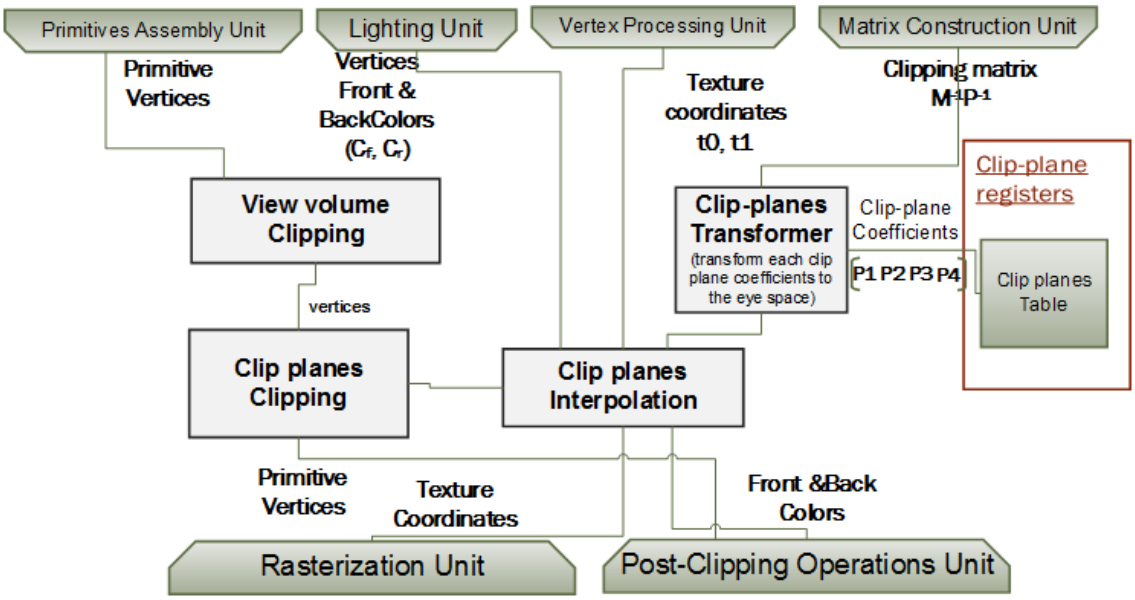


Figure 4.6: The clipping unit architecture

stored in the world space. So, the coefficients, of the enabled clip planes, should be transformed to the clip space by multiplying by the clipping matrix $M^{-1}P^{-1}$ that is received from the matrix construction unit

$$(P_1^{`}P_2^{`}P_3^{`}P_4^{`}) = (P_1P_2P_3P_4) * M^{-1}P^{-1} \tag{4.15}$$

Finally, the associated data unit calculates the colors and texture coordinates of the introduced vertices based on the locations of these vertices. For a new point P_r which lies on the line with end-points P_0 and P_1, the associated data f is calculated from the associated data $f_0$and $f_1$of the two end-points:

$$\begin{aligned} f &= (1-t) * f_0 + (t) * f_1 \\ t &= \frac{(P_r - P_0) \bullet (P_1 - P_0)}{\| P_1 - P_0 \|^2} \end{aligned} \tag{4.16}$$

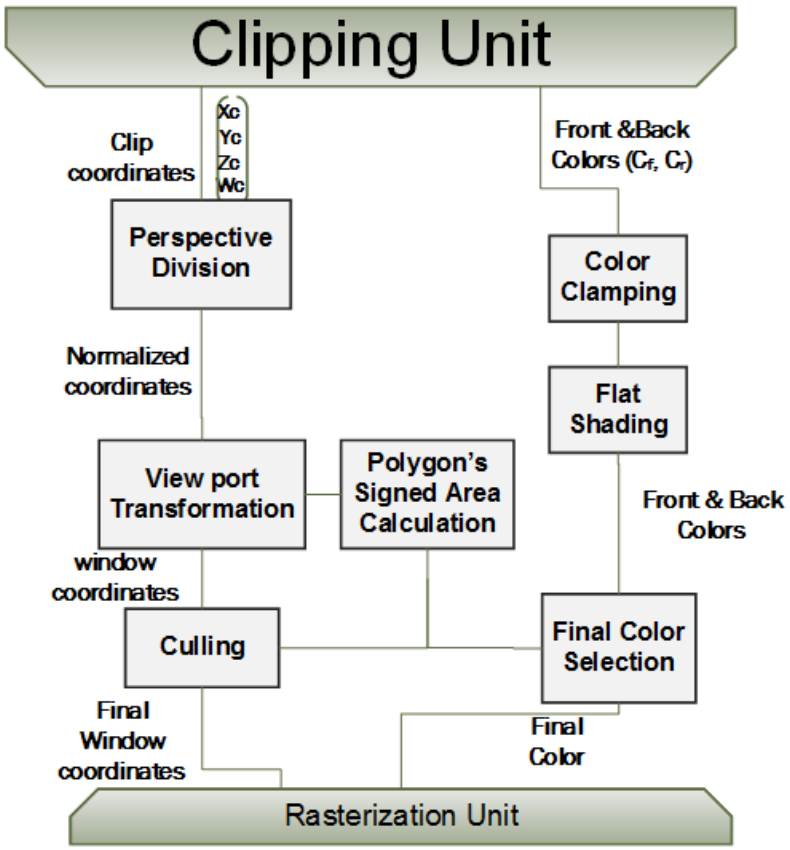


Figure 4.7: Post-clipping unit block diagram

Algorithm 4.4 Cohen-Sutherland Point-Check algorithm

```
Point-Check (X0, Y0, Z0, Wc)
    IF (Zc > Wc)
        Front = 1
    Else
        Front = 0
    IF (Zc < -Wc)
        Back = 1
    Else
        Back = 0
    IF (Yc > Wc)
        Top = 1
    Else
        Top = 0
    IF (Yc < -Wc)
        Bottom = 1
    Else
        Bottom = 0
    IF (Xc > Wc)
        Right = 1
    Else
        Right = 0
    IF (Xc < -Wc)
        Left = 1
    Else
        Left = 0
     return (front, back, top, bottom, right, left)
```

Algorithm 4.5 View volume clipping algorithm

```
If (Draw mode = points)
        →If (Check-Point (X0, Y0, Z0, Wc) = 000000)
                Pass this point to the clip plane clipping unit
          Else
                Discard it and terminate.
ElseIF (Draw mode = Line)
        →DrawLine (P0, P1)
ElseIF (Draw mode = Triangle)
        →DrawLine (P0, P1)
        →DrawLine (P1, P2)
        →DrawLine (P2, P10)
        →Construct Triangles from the resulted line
```

Algorithm 4.6 Cohen-Sutherland line-drawing algorithm

Draw-Line (P0, P1)

→ OUT0 = Point-Check (X0, Y0, Z0, Wc)
→ OUT1 = Point-Check (X1, Y1, Z1, Wc)
Repeat
→If ((Out0 = 000000) AND (Out1 = 000000)) then
Pass this line to the clip plane clipping unit
ElseIF ((Out0 AND Out1) != 000000)
Discard it and terminate.
Else
If (Front) then
→Znew = Wc
→$t = \frac{Z_{new}-Z_1}{Z_0-Z_1}$
→ Xnew = t X0 + (1-t) X1
→ Ynew = t Y0 + (1-t) Y1
ElseIF (Back)
→ Znew = -Wc
→$t = \frac{Z_{new}-Z_1}{Z_0-Z_1}$
→ Xnew = t X0 + (1-t) X1
→ Ynew = t Y0 + (1-t) Y1
ElseIF (Top)
→ Ynew = Wc
→$t = \frac{Y_{new}-Y_1}{Y_0-Y_1}$
→ Xnew = t X0 + (1-t) X1
→ Znew = t Z0 + (1-t) Z1
ElseIF (Bottom)
→ Ynew = -Wc
→$t = \frac{Y_{new}-Y_1}{Y_0-Y_1}$
→ Xnew = t X0 + (1-t) X1
→ Znew = t Z0 + (1-t) Z1
ElseIF (Right)
→ Xnew = Wc
→$t = \frac{X_{new}-X_1}{X_0-X_1}$
→ Ynew = t Y0 + (1-t) Y1
→ Znew = t Z0 + (1-t) Z1
ElseIF (Left)
→ Xnew = -Wc
→$t = \frac{X_{new}-X_1}{X_0-X_1}$
→ Ynew = t Y0 + (1-t) Y1
→ Znew = t Z0 + (1-t) Z1
End
→If (P0 is the outer point) then
P0 = Pnew,
Point-Check (X0, Y0, Z0, Wc)
Else
P1 = Pnew,
Point-Check (X1, Y1, Z1, Wc)
Until done

Algorithm 4.7 User clipping algorithm

For each clip plane CLIP_PLANEi with coefficients
$(P1'\ P2'\ P3'\ P4') = (P1\ P2\ P3\ P4) * M^{-1}\ P^{-1}$
IF (CLIP_PLANEi enable = 1)
If (Draw_mode = points)
→If$((P1'\ P2'\ P3'\ P4') * \ (X_C \quad Y_C \quad Z_C \quad W_C)^T \ \geqslant 0)$
If (CLIP_PLANEi = last clip plane)
Pass this point to the rasterization block.
Else
Discard it and terminate.
ElseIF (Draw mode = Line)
→If$((P1'\ P2'\ P3'\ P4') * \ (X_C \quad Y_C \quad Z_C \quad W_C)^T \ \geqslant 0$ for the two vertices)
If (CLIP_PLANEi = last clip plane)
Pass this line to the rasterization block.
Else$((P1'\ P2'\ P3'\ P4') * \ (X_C \quad Y_C \quad Z_C \quad W_C)^T \ < 0\ )$
Discard it and terminate.
Else
- Calculate the intersection ratio (t) by solving the equations
1) P = (P1-P2)*t + P2 where P1 & P2 are the line end points
2) (P1‘ P2‘ P3‘ P4‘) *P = 0.
- generate two lines. The first one has the end points (P1, P) and the second one has the end points (P1, P2).
ElseIF (Draw mode = triangle)
→If $((P1'\ P2'\ P3'\ P4') * \ (X_C \quad Y_C \quad Z_C \quad W_C)^T \ \geq 0$ for all vertices)
If (CLIP_PLANEi = last clip plane)
Pass this line to the rasterization block.
ElseIF$((P1'\ P2'\ P3'\ P4') * \ (X_C \quad Y_C \quad Z_C \quad W_C)^T \ < 0$ for all vertices)
Discard it and terminate.
Else
- Calculate the intersection ratios (t1, t2) for the two intersected lines by solving the equations
1) P = (P1-P2)*t + P2 where P1 & P2 are the line end points.
2) (P1‘ P2‘ P3‘ P4‘) *P = 0.
- calculate the intersection points Px and Py.
- If (only one point (P0) lies in the half-space defined by the plane)
Generate one triangle with vertices (P0, Px, Py).
ElseIF (two point (P0, P1) lies in the half-space defined by the plane)
Generate two triangles with vertices (P0, P1, Px) and (P0, P1, Py).

4.7 Post-Clipping Unit

The post-clipping unit performs the final processing on the vertices coordinates and colors before sending them to the rasterization process. It transforms the coordinates from the clip space to the window space. Also, it performs the culling check and calculates the final vertices colors based on the shade model. This unit handles the required functions set by OpenGL ES 1.1 [8], sections 2.10.1, 2.12.5, 2.12.6 and 2.12.8.

4.7.1 Register File

1. Shade model: 1-bit to specify the shade model; initially, the shade model is SMOOTH.
2. Cull Face mode: 2-bits to specify the cull face; initially, the cull face is BACK.
3. Cull Face enable: 1-bit to enable/disable the cull face; initially, it is disabled.
4. Front Face direction: 1-bit to specify the front face direction; initially, it is CCW.

4.7.2 Architecture

Figure 4.7 shows the post clipping unit architecture. The clipping unit sends the primitive, in the clipping space (X_c, Y_c, Z_c, W_c), vertices and their corresponding colors. First, the perspective division unit normalizes the coordinates. The normalized vertices(X_c, Y_c, Z_c) are calculated by:

$$\begin{pmatrix} X_{nc} \\ Y_{nc} \\ Z_{nc} \end{pmatrix} = \begin{pmatrix} \frac{X_c}{W_c} \\ \frac{Y_c}{W_c} \\ \frac{Z_c}{W_c} \end{pmatrix} \tag{4.17}$$

Then, they are transformed to the window space by multiplying by the view port matrix V. Then, the signed area a is calculated from the window coordinates (X_w, Y_w):

$$a = \sum_{i=0}^{n-1} X_w^i * Y_w^{(i+1) mod\, n} - X_w^{(i+1) mod\, n} * Y_w^i \tag{4.18}$$

Where x_w^i and y_w^i are the window coordinates of the i^{th} vertex of the n-vertex polygon; vertices are numbered starting from zero . Finally, the culling block decides either to cull or pass the primitive according the culling algorithm 4.8:

In parallel, the front and back colors are clamped to [0, 1] range. Then, if the shading model is smooth, the shading block passes these colors without any modification; otherwise, the primitive colors are assigned to same color based on the drawing mode. If the primitive is a line, the flat color is the color of the second vertex. If it is a triangle, the flat color is the color of the last vertex color. Then final color selection unit selects, for each vertex, the front color or the back color according to the final color algorithm 4.9.

Algorithm 4.8 Culling algorithm

```
If (front face polygon a > 0)
        If (culling mode = FRONT or FRONT&BACK)
                Polygon is discarded
        Else
                Polygon is rasterized
If (back face polygon a < 0)
        If (culling mode = BACK or FRONT&BACK)
                Polygon is discarded
        Else
                Polygon is rasterized
```

Algorithm 4.9 Final color algorithm

```
If (Primitive is a line or a point)
        Final color = front color
ElseIF (Primitive is a triangle)
        If (signed area a > 0)
                If (Front face direction is CCW)
                        Final color = front color
                Else
                        Final color = back color
        Else
                If (Front face direction is CCW)
                        Final color = back color
                Else
                        Final color = front color
```

4.8 Rasterization Unit

The rasterization unit is the main block in the graphics pipeline. Our architecture supports three primitive: triangle, line, and point. For the triangle, it produces the fragments that are intersected with the primitive and calculates its data by interpolating the associated data of the three vertices. For the line, it supports unity-width rasterization, wide line rasterization, and line smoothing rasterization. For the point, it supports point rasterization, point smoothing rasterization, and point sprite rasterization. It also support the 4x multisampling for all primitives. It receives the coordinates, color component, depth value, and texture coordinates of each vertex. Also, it may receive the point width if the point array is enabled. It produces the screen coordinates (X_w, Y_w) and the associated data (color, texture coordinates, depth) of each fragment that is intersected with the primitive. Also, it calculates the coverage samples if the multisampling is enabled. This unit handles the required functions set by OpenGL ES 1.1 [8], sections 3.3, 3.4, 3.5, and 3.6.

4.8.1 Register File

1. Control bits: 6-bits to enable or disable the rasterization options. These options are the multisampling, point smoothing, point sprite, line smooth, polygon offset fill, and point coordinates replace. Initially, all these options are disabled.

2. Draw mode: 2-bits to specify the incoming primitive type (triangle, line, point).

3. Point data. They are 7 fixed-point variables that control the point rasterization: one variable for the current point width which is used if the point size array is disabled, one variable for the point-fade threshold size, two variables for the point maximum and minimum sizes, and three variables for the point attenuation factors (a, b, c). Initially, the point width is one.

4. Line width: one fixed-point value to specify the line width. Initially, the line width is one.

5. Polygon variable: two fixed-point variables to store the factor and units.

6. Sample buffers: integer value that controls the multisampling operation.

7. Point fade threshold size: fixed-point value is used instead of the derived point width if it go below this threshold.

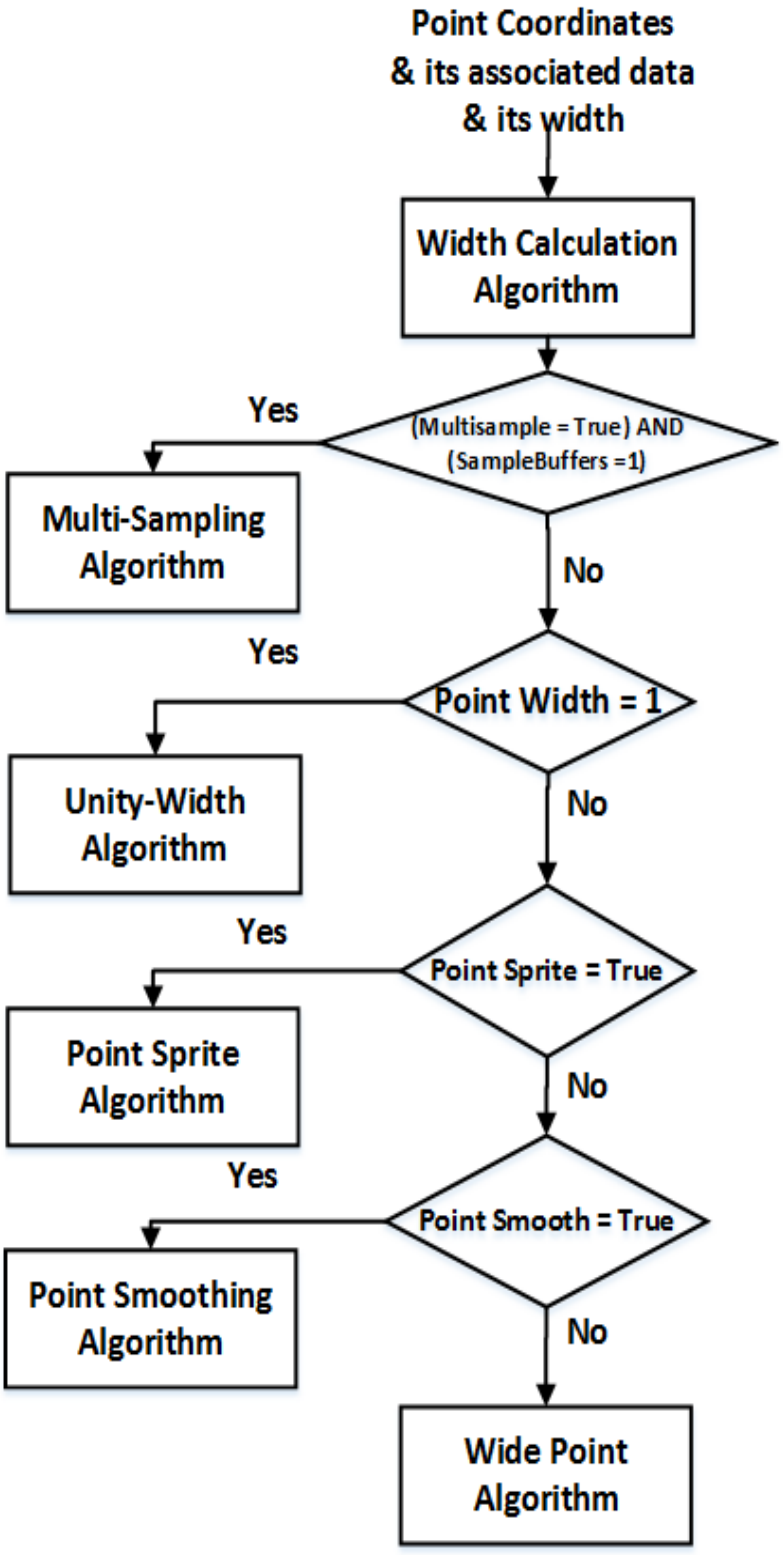


Figure 4.8: Point rasterization flow chart

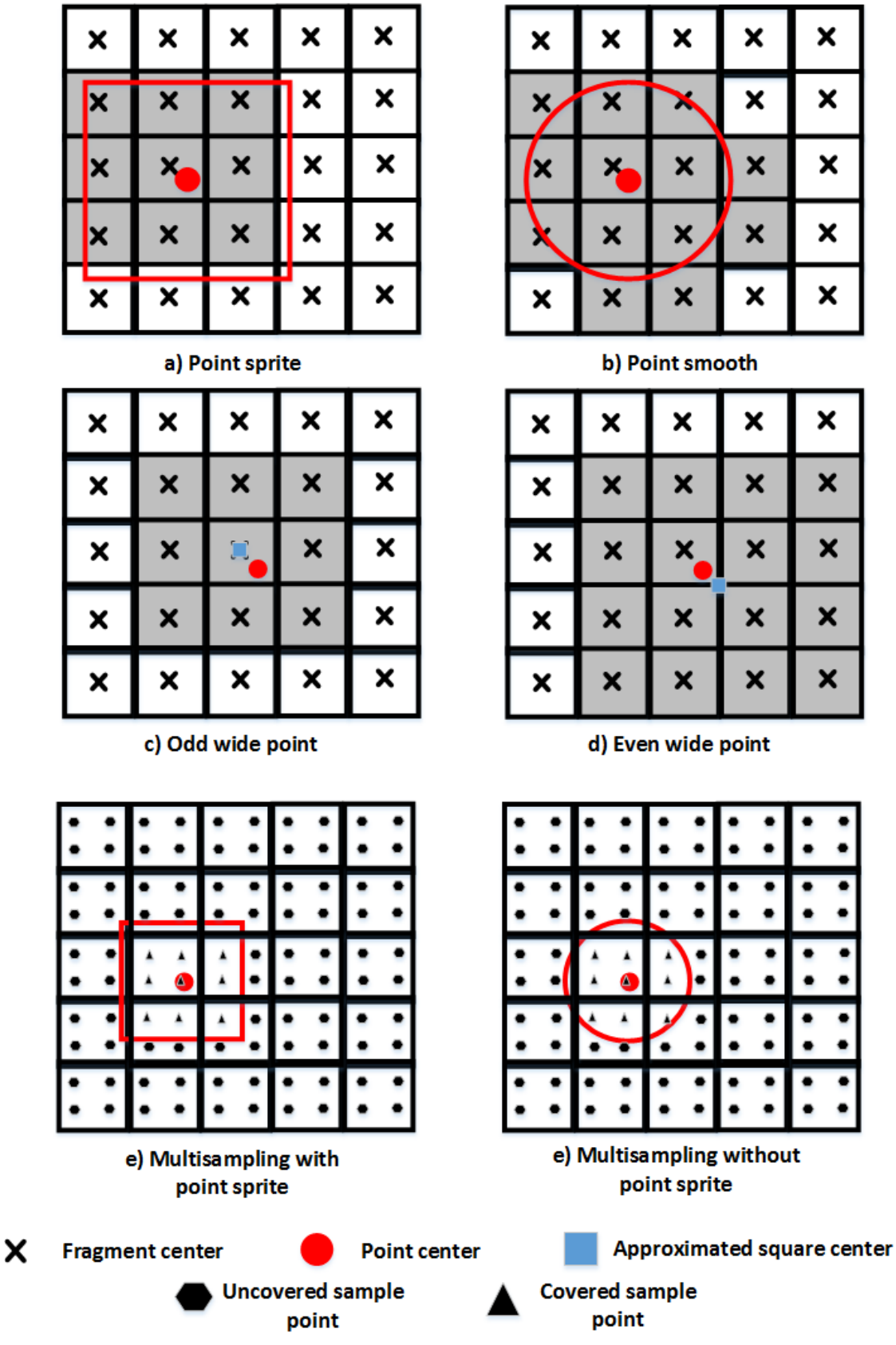


Figure 4.9: Point area for different rasterization algorithms

Algorithm 4.10 Width algorithm

→Derived-Size = Implementation_Clamp ($\frac{point-size}{\sqrt{a+b*d+c*d^2}}$) where d is the eye coordinate distance from the eye location (0,0,0,1).
→If (multisampling is enabled)
If (derived-size ≥ point-fade-threshold-size)
width = derived-size
Else
width = point-fade-threshold-size
Else
width = derived-size
→Clamp width to [point-size-min, point-size-max] range.

Algorithm 4.11 Point multisampling algorithm

If (Point Sprite = false)
If (sample points lies in a circle with a center (X_w, Y_w) and a diameter of the point width)
→ Sample coverage bit = 1.
→ Sample point's color, depth, and texture coordinates are the associated data of the point.
Else
If (sample points lies in a square with a center (X_w, Y_w) and a side length of the point width)
→ Sample coverage bit = 1.
→ Sample point's color and depth are the associated data of the point.
→ For each texture unit, if (point coordinates replace = True)
Texture coordinates is calculated for each sample point:
$S = 0.5 + \frac{X_f + .05 - X_w}{Point\,Width}$, $T = 0.5 + \frac{Y_f + .05 - Y_w}{Point\,Width}$
$R = 0$, $Q = 1$
where (X_f, Y_f) is the sample point coordinates in the window space.

Algorithm 4.12 Point sprite algorithm

If (fragment center lies in a square with a center (X_w, Y_w) and a side length of the point width)
→ fragment is drawn.
→ fragment's color and depth are the associated data of the point.
→ For each texture unit, if (point coordinates replace = True)
Texture coordinates is calculated for each fragment:
$S = 0.5 + \frac{X_f + .05 - X_w}{Point\,Width}$, $T = 0.5 + \frac{Y_f + .05 - Y_w}{Point\,Width}$
$R = 0$, $Q = 1$
where (X_f, Y_f) is the fragment coordinates in the window space.

Algorithm 4.13 Point smooth algorithm

If (fragment's square area intersects with a circle with a center (X_w, Y_w) and a diameter of the point width)
→ Fragment is drawn.
→ Fragment's color, depth, and texture coordinates are the associated data of the point.
→ Coverage area is calculated and is used to adapt the final alpha value of the fragment.
→ The coverage area is the area of the intersection of the point's circular region
with the fragment's square area.

Algorithm 4.14 Wide point algorithm

→ square width = NRE (point width)
→ If (square width = 0)
square width = 1.
→ If (square width is an odd number)
→ square center $(x, y) = (\,floor(X_w) + 0.5,\, floor(Y_w) + 0.5)$
→ draw a square grid with this odd width.
Else
→ square center $(x, y) = (\,floor(X_w + 0.5),\, floor(Y_w + 0.5))$
→ draw a square grid with this even width.

4.8.2 Architecture

The rasterization unit is divided into three units: point rasterization, line rasterization, and triangle rasterization. We discuss the algorithms of each unit in the following sections.

4.8.2.1 Point Rasterization

As shown in figure 4.8, multiple algorithms deal with the point according to the control bits and the sample buffers variable. Initially, the derived width is calculated by the width calculation algorithm. Then, the multisampling, unity-width, point sprite, point smoothing, or wide line algorithm is applied. For unity-width, point sprite, point smoothing, or wide point algorithms, if the multisampling is enabled, the coverage bits for the covered fragment's samples are 1's and their data is the calculated fragment data.

The Width Calculation Algorithm

The width calculation algorithm 4.10 compute the final point width based on the input point width and the attenuation factors.

The Multisampling Algorithm

Our CUGPU supports the 4x multisampling. So, the multisampling algorithm 4.11 produces a fragment for each pixel with one or more sample points that intersect a region with a center (X_w, Y_w), the point coordinates in the window space. It calculates the color, depth, and texture coordinates for each sample point. If the point sprite is enabled, the point area is an square with line length equals the point width as shown in figure 4.9(e); otherwise, the point area is a circle with a diameter equals the point width as shown in figure 4.9(f).

The Unity Width Algorithm

For a point with unity width, the fragment with coordinates $(int(X_w) + .5, int(Y_w) + 0.5)$ is drawn. The fragment color, depth, and texture coordinates are the associated data with the point vertex.

The Point Sprite Algorithm

The point sprite algorithm 4.12 produces fragments which their centers lay in a square with a center (X_w, Y_w) and a side length of the point width as shown in figure 4.9(a).

The Point Smooth Algorithm

The point smooth algorithm 4.13 produces fragments which their square area intersects with a circle with a center (X_w, Y_w) and a diameter of the point width as shown in figure 4.9(b).

The Wide Point Algorithm

The wide point algorithm 4.14 produces an odd or even square of fragments based on the point width. For a point with an odd width, figure 4.9(c), the square center is laying

on the half-grid. For a point with an even width, figure 4.9(d), the square center is laying on the integer grid.

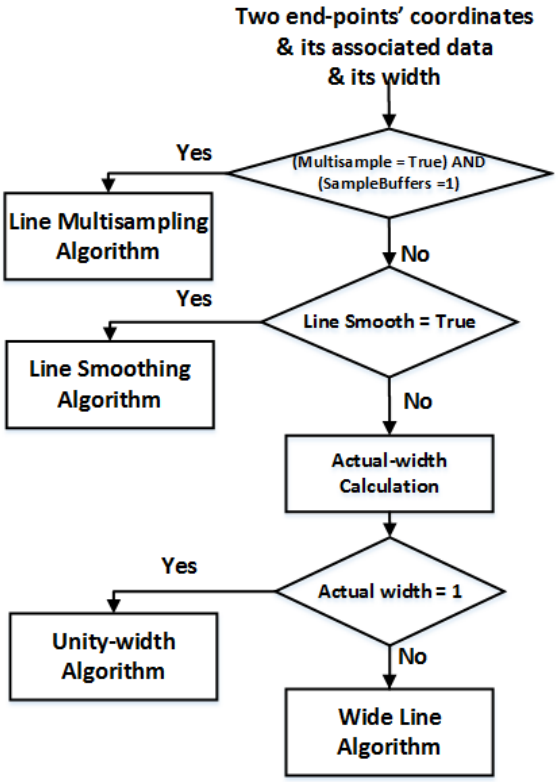


Figure 4.10: Line rasterization flow chart

Algorithm 4.15 Wide line algorithm

If (x-major line)

→ Calculate the coordinates of parallel line end points (X_1, Y_1) and (X_2, Y_2).
$(X_1, Y_1) = (X_a, Y_a - \frac{w-1}{2})$ and $(X_2, Y_2) = (X_b, Y_b - \frac{w-1}{2})$.

→ Draw the unity-width line with end points (X_1, Y_1) and (X_2, Y_2).

→ Produce a column of fragments of height w at each x location with the same associated data.

Else

→ Calculate the coordinates of parallel line end points (X_1, Y_1) and (X_2, Y_2):
$(X_1, Y_1) = (X_a - \frac{w-1}{2}, Y_a)$ and $(X_2, Y_2) = (X_b - \frac{w-1}{2}, Y_b)$.

→ Draw the unity-width line with end points (X_1, Y_1) and (X_2, Y_2).

→ Produce a row of fragments of width w at each y location with the same associated data.

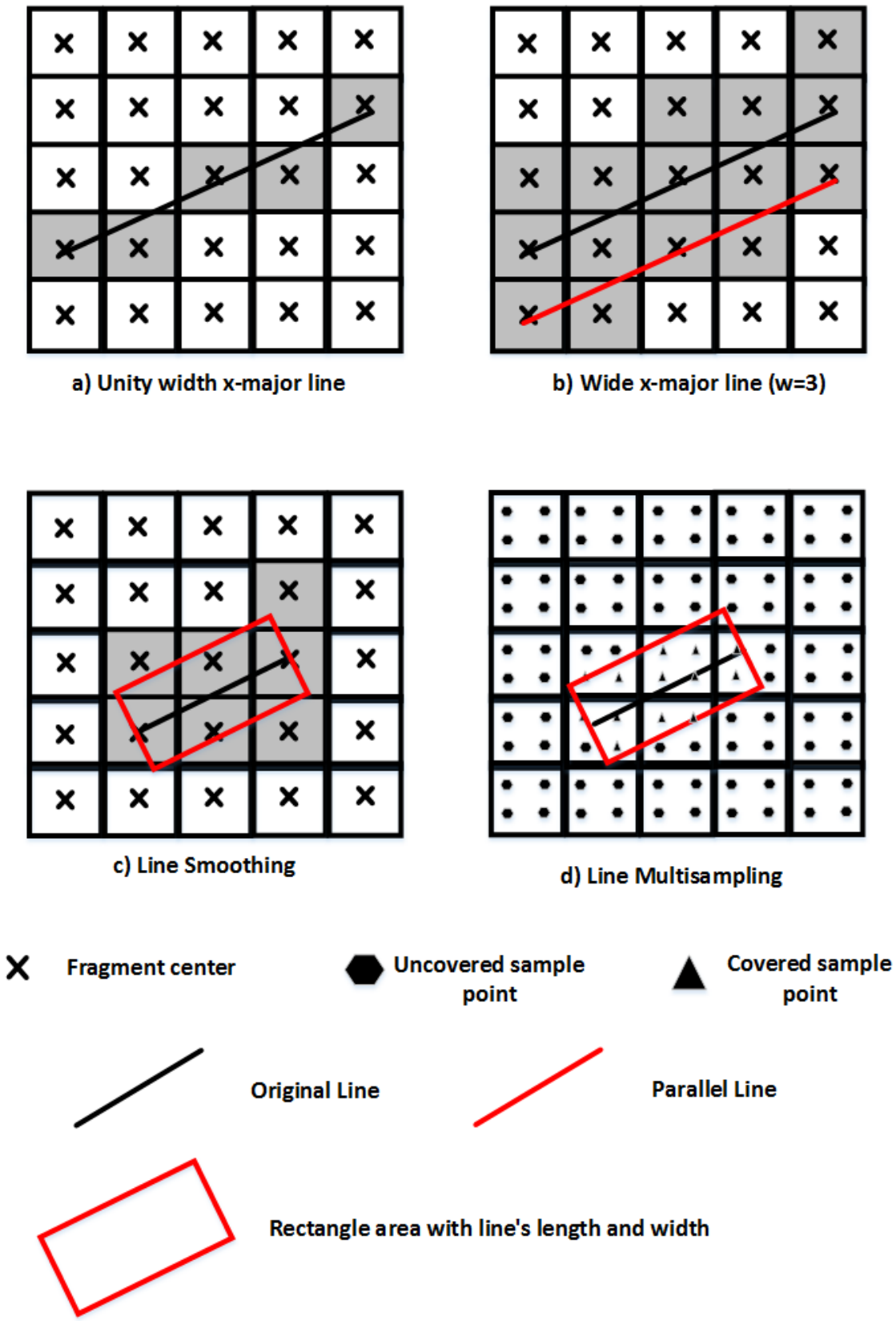


Figure 4.11: Line area for different rasterization algorithms

4.8.2.2 Line Rasterization

As shown in figure 4.10, multiple algorithms are applied on line according to the control bits and the sample buffers variable: line multisampling, line smoothing, unity-width, or wide line algorithm. The incoming line is defined by the two end points $P_a(X_a, Y_a)$ and $P_b(X_b, Y_b)$ and their associated color, depth, and texture coordinates. For line smoothing, unity-width, or wide line algorithms, if the multisampling is enabled, the coverage bits for the covered fragment's samples are 1's and their data is the calculated fragment data.

Line Multisampling Algorithm

The multisampling algorithm produces a fragment with one or more sample points that intersect a rectangle region with line's length and width as shown in figure 4.11 (d). The texture coordinates, color, and depth of the produced sample points are computed by interpolating the two end-points texture coordinates, color, and depth, respectively. The interpolation is performed according the OpenGL equations 5.3,5.4, and 5.2.

Line Smoothing Algorithm

smoothing algorithm produces a fragment that intersects a rectangle region with line's length and width as shown in figure 4.11(c). For each pixel, a coverage value is computed and used for modifying the alpha value. This coverage value equals the intersection area between the fragment and the rectangle region. The texture coordinates, color, and depth of the produced sample points are computed by interpolating the two end-points texture coordinates, color, and depth, respectively. The interpolation is performed according the OpenGL equations 5.3,5.4, and 5.2.

Actual Line Calculation

The line width is rounded to the nearest integer above. If this rounded width is 0, the actual width will be 1.

Unity Width Algorithm

The unity width algorithm produces a fragment that intersects a line with a width of 1 fragment in the minor direction as shown in figure 4.11(a). For x-major lines, no

two fragments may be produced that lay in same column. For y-major line, no two fragments may be produced that lay in same row. The texture coordinates, color, and depth of the produced sample points are computed by interpolating the two end-points texture coordinates, color, and depth, respectively. The interpolation is performed according the OpenGL equations 5.3,5.4, and 5.2. In Chapter 5, we discussed briefly our implementation of the unity width algorithm.

Wide Line Algorithm

The wide line algorithm 4.15 is as same as the unity width algorithm except it produces width fragments in the minor direction as shown in figure 4.11(b). For an input line with end points$P_a(X_a, Y_a)$ and $P_b(X_b, Y_b)$ and a width w, we apply the unity width algorithm for a parallel line to the input line. Then, a column of fragments of height w is produced at each x location for x-major lines. Otherwise, a row of fragments of length w is produced at each y for y-major lines.

4.8.2.3 Triangle Rasterization

As shown in figure 4.12, the triangle rasterization has two modes: the basic rasterization and the multisampling rasterization modes. In the basic rasterization mode, it produces the fragments that its center lies within the input triangle using the triangle scan conversion method. It also interpolates color, depth, and texture coordinates of the three vertices to compute the color, depth, and texture coordinates of the produced fragment, respectively.

The multisampling rasterization is as same as the basic rasterization except it calculates the color, depth, and texture coordinates for each sample point that lay inside the input triangle. If the multisampling is enabled and the sample buffer is not one, the basic algorithm is applied, the coverage bits for the covered fragment's samples are 1's, and their data is the calculated fragment data. Finally, the depth values are modified, according to algorithm 4.16, by adding an offset value if the polygon offset fill is enabled.

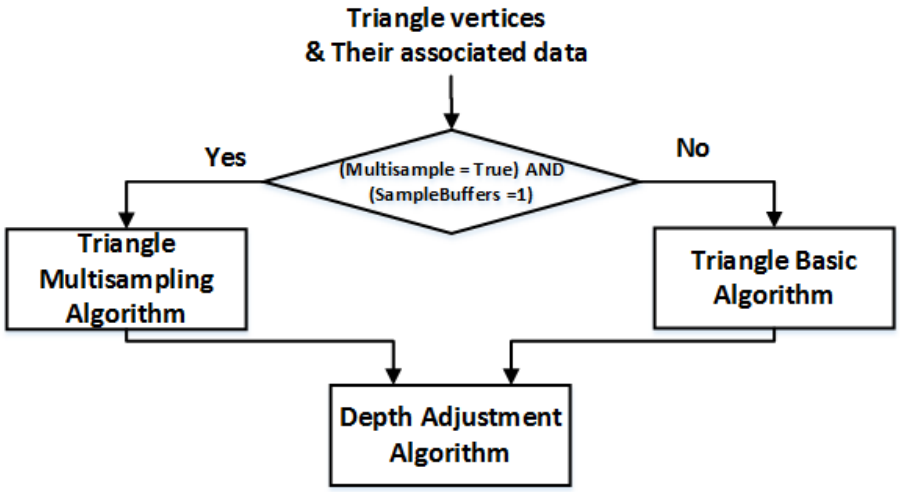


Figure 4.12: Triangle flow chart

Algorithm 4.16 Depth adjustment algorithm

Offset $(O) = m * factor + r * units$
where r is implementation-dependent constant represents
minimum resolvable difference and m is the maximum depth slope:
$m = max\{|\frac{dZ_w}{dX_w}|, |\frac{dZ_w}{dY_w}|\}$

If (Polygon offset fill is enabled)

$$Depth(Z) = Depth(Z) + O$$

Clamp the depth value to $[0, 1]$

4.9 Texture Handling Unit

The texture handling unit creates texture objects in the graphics memory and copies their data from images stored in the system memory or from the frame buffer. Also. it converts the color from the type and format of images in system memory to the internal format of texture images in graphics memory. In addition, it performs auto mipmapping of the texture image's arrays if auto mipmapping is enabled. Moreover, it specifies the bounded texture object that is used in texture mapping operation for the active texture unit. Initially, the bounded texture buffer is Texture0.

The texture handling unit creates or modifies the bounded texture object by four commands:

1. TexImage2D: it specifies an image, in the system memory, to be copied to the bounded texture object, in the graphics memory.

2. CopyTexImage2D: it specifies part of the frame buffer, in the graphics memory, to be copied to the bounded texture object, in the graphics memory.

3. TexSubImage2D: it specifies an image, in the system memory, to be copied to a rectangular sub-region of the bounded texture object, in the graphics memory.

4. CopyTexSubImage2D: it updates a rectangle sub-region of the bounded texture object from a rectangle part of the frame buffer, in the graphics memory.

This unit handles the required functions set by OpenGL ES 1.1 [8], sections 3.6, 3.7.1, 3.7.2, 3.7.9, 3.7.10, and 3.7.11.

4.9.1 Register File

1. Texture objects data: each texture object has an integer number represents the object name, an integer represents the number of mipmap sets for these objects, and a mipmap table that stored the details of each mipmap array. Table 4.3 demonstrates the structure of the mipmap table. Initially, only texture0 object is defined.

2. Pixel store alignment: 4-valued integer that specifies the alignment, in byte unit, for the start4.13 of each pixel row in the system memory.

3. Texture units table: it specifies state of each texture unit and defines the bounded texture object for each texture unit. Initially, all texture units are disabled.

4. Current bounded texture object: an integer number that defines the bounded texture object. All texture handling operations are performed on this bounded texture object.

Array number (LoD)	**Width**	**Height**	**Internal format**	**Resolution (R,G,B,A,Luminance,Intensity)**	**Address**
0	**initially(0)**	**initially(0)**	**initially(1)**	**resolution of the color component**	**graphics memory**
1					

Table 4.3: Materials and light models parameters

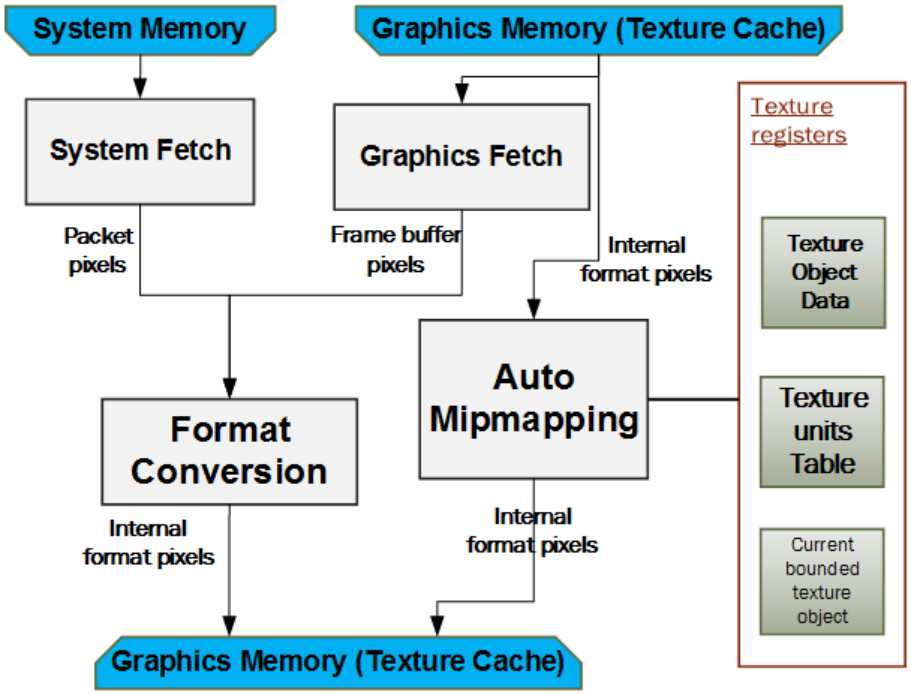


Figure 4.13: Texture handling unit block diagram

4.9.2 Architecture

Figure 4.13 presents the block diagram of the texture handling unit. The system fetch unit fetches pixels data from the system memory if command is TexImage2D or TexSubImage2D. Whereas the graphics fetch unit fetches the pixels data from the frame buffer (graphics memory) if command is CopyTexImage2D or CopySubTexImage2D. Then, the format conversion unit converts the pixel from the storage format, in the system memory, or the frame buffer format to texture internal format and updates the texture images. In parallel, the auto mipmapping unit generates different LoD texture image arrays from the zero-level texture array.

4.9.2.1 System Fetch Unit

The system fetch unit reads a rectangle of pixels from the system memory if command is TexImage2D or TexSubImage2D. The *data* pointer is the location of the first pixel in the system memory. Number of pixels per column equals the *width* parameter whereas number of rows is the *height* parameter. Table 4.4demonstrates number of bytes per pixel for all valid pixel *format* and *type* combinations. It feds the format conversion unit by streams of pixels.

Format	Type	Pixel Size (in bytes)
RGBA	ubyte	4
RGB	ubyte	3
RGBA	short_4_4_4_4	2
RGBA	ushort_5_5_5_1	2
RGB	ushort_5_6_5	2
Luminance-Alpha	ubyte	2
Luminance	ubyte	1
Alpha	ubyte	1

Table 4.4: Pixel size for all valid format and type combinations

Color buffer format	Texture internal format				
	A	L	LA	RGB	RGBA
Alpha (A)	yes	no	no	no	no
Luminance (L)	no	yes	no	no	no
LA	yes	yes	yes	no	no
RGB	no	yes	no	yes	no
RGBA	yes	yes	yes	yes	yes

Table 4.5: Valid color buffer and texture internal formats combinations

Internal texture format	RGBA pixel components
Alpha (A)	A
Luminance (L)	R
Luminance_Alpha (LA)	R, A
RGB	R, G, B
RGBA	R, G, B, A

Table 4.6: Conversion from RGBA pixel components to internal texture components

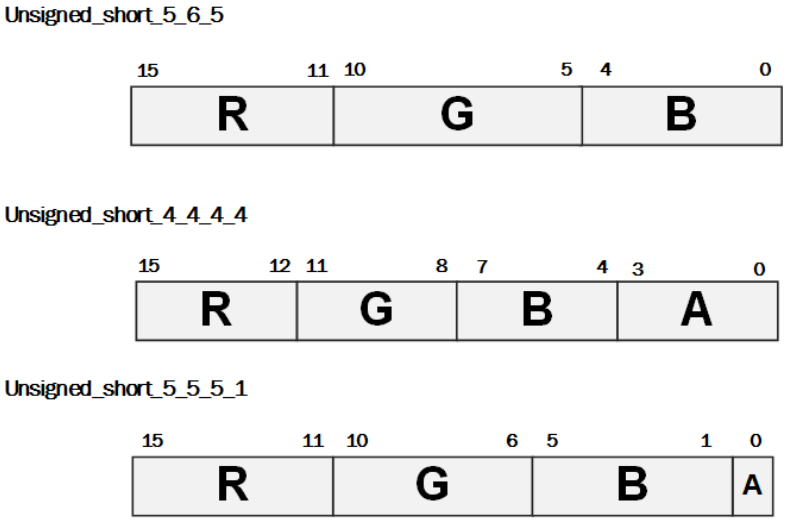


Figure 4.14: Unsigned short formats

4.9.2.2 Graphics Fetch Unit

The graphics fetch unit reads a rectangle of pixels from the frame buffer in the graphics memory if command is CopyTexImage2D or CopyTexSubImage2D. The (x, y) coordinates define the left lower corner of the frame buffer region to be copied. Number of pixels per column equals the *width* parameter whereas number of rows is the *height* parameter. It feds the format conversion unit by streams of pixels in color buffer format.

4.9.2.3 Format Conversion Unit

The format conversion unit converts the format of the incoming pixel to the texture's internal format to be stored according to algorithm 4.17. It receives pixels, stored in system image's format, from the system fetch unit or pixels, stored in the frame buffer's format, from the graphics memory.

For pixels that are received from the system fetch unit, it unpacks pixels if they are encoded in ushort_4_4_4_4, ushort_5_5_5_1, or ushort_5_6_5 type as shown in figure 4.14. Then, it converts the pixel to RGBA format. Then, it clamps each color components to $[0, 1]$ range and selects the color components to be stored in the texture object array based on the internal format. Table 4.6summarizes the conversions from the RGBA formats to the texture internal formats.

For pixels that are received from the graphics fetch unit, they are received in color buffer format. Then, the format conversion unit clamps color components to$[0, 1]$ range

and converts them to the internal format if this is an applicable transformation. Table 4.5summarizes the available transformations from the color buffer formats to the texture internal formats. Finally, it updates the zero-LoD texture array.

4.9.2.4 Auto Mipmapping Unit

The auto mipmapping unit creates higher ordered set of arrays that represents the same texture image. This set of arrays are derived from the Zero-LoD texture array. These contents are computed by using 2x2 box filter reduction method: each 2x2 box filter pixels are averaged to compute a pixel of a higher LoD texture array.

Algorithm 4.17 Format conversion algorithm

→ If (Command ∈TexImage2D or TexSubImage2D)
- → Receive pixel colors from the system fetch unit.
- → Unpack the pixel color if pixel type is unsigned short, figure 4.14.
- → If (pixel format = luminance (L) or luminance alpha (LA))
  Red (R), Green (G), and Blue (B) color components = L.
- → If (R,G, and B do not exist)
  Red (R), Green (G), and Blue (B) color components = 0.0.

ElseIF (Alpha (A) does not exist)
  Alpha (A) color component = 1.0.
- → Clamp the color components to $[0,1]$range.
- → Select the color component from the RGBA format based on the texture internal format, table 4.6.

ElseIF (Command ∈CopyTexImage2D or CopyTexSubImage2D)
- → Clamp the color components to $[0,1]$range.
- → Converts the color components to the internal format if this is an applicable transformation, table 4.5.

→ If (Command ∈TexImage2D or CopyTexImage2D)
- → Update the whole bounded texture zero-LoD array, started from the lower left corner $(0,0)$.

ElseIF (Command ∈TexSubImage2D or CopyTexSubImage2D)
- → Update a rectangle sub-region of the bounded texture zero-LoD array. This rectangle sub-region is defined by its lower left corner $(xoffset, yoffset)$, width, and height.

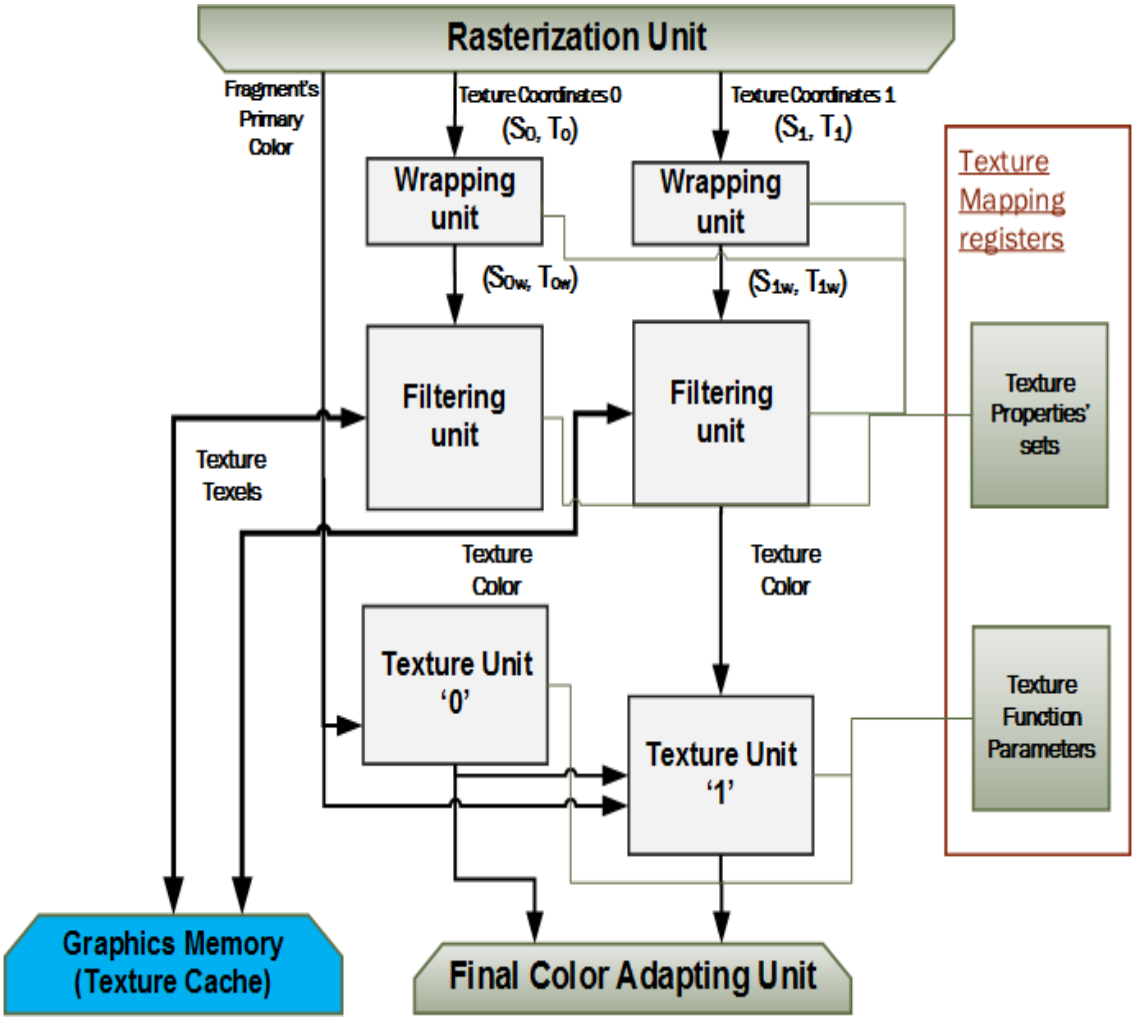


Figure 4.15: Texture Mapping Architecture

Property	Type	Options	Initial value
Texture_Minification_Filter	6-valued integer	Nearest, Linear, Nearest_Mipmap_Nearest Linear_Mipmap_Nearest Nearest_Mipmap_Linear Linear_Mipmap_Linear	Nearest_ Mipmap_ Linear
Texture_Magnification_Filter	2-valued integer	Nearest, Linear	Linear
Texture_wrap_S	2-valued integer	Repeat, Clamp_to_Edge	Repeat
Texture_wrap_T	2-valued integer	Repeat, Clamp_to_Edge	Repeat

Table 4.7: Texture properties set

Parameter	Type	Options	Initial value
Texture mode	6-valued integer	Replace, Modulate, Decal, Blend, Add, Combine	Modulate
Combine_RGB	8-valued integer	Replace, Modulate, Add, Add_signed, Interpolate, Subtract, Dot3_RGB, Dot3_RGBA	Modulate
Combine_Alpha	6-valued integer	Replace, Modulate, Add, Add_signed, Interpolate, Subtract	Modulate
SRC0_RGB, SRC0_Alpha	4-valued integer	Texture, Constant, primary_color, previous	Texture
SRC1_RGB, SRC1_Alpha	4-valued integer	Texture, Constant, primary_color, previous	Previous
SRC2_RGB, SRC2_Alpha	4-valued integer	Texture, Constant, primary_color, previous	Constant
Operand0_RGB, Operand1_RGB	4-valued integer	SRC_Color, One_minus_SRC_Color, SRC_Alpha, One_minus_SRC_Alpha	SRC_Color
Operand2_RGB	4-valued integer	SRC_Color, One_minus_SRC_Color, SRC_Alpha, One_minus_SRC_Alpha	SRC_Alpha
Operand0_Alpha, Operand1_Alpha, Operand2_Alpha,	2-valued integer	SRC_Alpha, One_minus_SRC_Alpha	SRC_Alpha
RGB_Scale, Alpha_Scale	3-valued integer	(1.0, 2.0, 4.0)	1.0
Texture Environment Color	4 fixed-point values	RGB (C_c) and Alpha (A_c)	(0,0,0,0)
Texture0unit_enable Texture1unit_enable	Boolean	(True, False)	False

Table 4.8: Texture function parameters

4.10 Texture Mapping Unit

The texture mapping unit receives fragment's texture coordinates (S,T) and their primary colors (C_f, A_f) that are resulted from rasterization unit. It uses fragment's texture coordinates for addressing the texture object array and computing the fragment's texture source colors (C_s, A_s). Then, it produces the final color of the fragment by applying the texture function such as modulate and replace on the fragment's primary color, fragment's texture source colors, and texture environment color (C_c, A_c). Texture mapping process is divided into three steps:

1. compute the texture address by wrapping and filtering the fragment's texture coordinates.
2. fetch the fragment's texel from the bounded texture object array.
3. compute the final fragment's color by applying the texture functions on the fragment's primary color, fragment's texture source colors, and texture environment color.

This unit handles the required functions set by OpenGL ES 1.1 [8], sections 3.7.5, 3.7.6, 3.7.7, 3.7.10, 3.7.12, and 3.7.13.

4.10.1 Register file

1. Texture properties set: texture properties control the filtering and wrapping functions that are applied on the texture coordinates. Our architecture has two sets: one for each texture coordinates. Table 4.7 summarizes the texture properties and their initial values.
2. Texture function parameters: for each texture unit, the texture function parameters control the execution of the texture function on the fragment's primary color, fragment's texture source colors, and texture environment color. Table 4.8 summarizes the texture function parameters and their initial values.

4.10.2 Architecture

As shown in figure 4.15, the texture mapping unit receives fragment's texture coordinates (S_0, T_0) for texture unit 0, texture coordinates (S_1, T_1) for texture unit 1, and their

primary colors. First, the wrapping blocks modify the texture coordinates (S_0, T_0) and (S_1, T_1) based on the texture wrap mode: repeat or clamp_to_edge. Then, the filtering unit calculates texel's address and fetches the texel's color components from the graphics memory (texture cache). Also, it computes the final texel color and converts it from the texture internal format to the RGBA format.

Finally, if texture unit 1 is disabled, texture unit 0 produces the final color of the fragment by applying the texture function such as modulate and replace on the fragment's primary color (C_f, A_f), fragment's texture source colors (C_{s0}, A_{s0}), and texture environment color (C_c, A_c); otherwise, texture unit 1 calculates the final color by applying the texture function on the fragment's primary color (C_f, A_f), fragment's texture source colors (C_{s1}, A_{s1}), texture environment color (C_c, A_c) and the resulted color from texture unit 0 (C_p, A_p) as a primary color component.

4.10.2.1 Wrapping unit

The wrapping units modify the texture coordinates (S_0, T_0) and (S_1, T_1) based on the texture wrap mode: repeat or clamp_to_edge according to algorithm

4.10.2.2 Filtering unit

First, the filtering unit determines the filtering mode (magnification or minification) by calculating the level of details (LoD) parameter $\lambda(x, y)$ for the given primitive according to algorithm 4.19. Then, it calculates texel's addresses, fetches them, and computes the filtered texture components from the graphics memory based on the filtering mode according to algorithm 4.20. Then, it maps the color components from the internal format

Algorithm 4.18 Format conversion algorithm

If (Texture_wrap_S =repeat)
$S' = S - floor(S)$.
Else
Clamp S to $[\frac{1}{2*TextureWidth}, 1 - \frac{1}{2*TextureWidth}]$.

If (Texture_wrap_T =repeat)
$T' = T - floor(T)$.
Else
Clamp T to $[\frac{1}{2*TextureHeight}, 1 - \frac{1}{2*TextureHeight}]$.

of the filtered texture components to the texture source components (C_s, A_s) as shown in table 4.9.

Algorithm 4.19 Filtering mode algorithm

$u(x,y) = 2^{log_2(texture\,width)} * S(x,y)$

$v(x,y) = 2^{log_2(texture\,height)} * T(x,y)$

If (primitive =triangle)

$\rho = max\{\sqrt{(\frac{\delta u}{\delta x})^2 + (\frac{\delta v}{\delta x})^2}, \sqrt{(\frac{\delta u}{\delta y})^2 + (\frac{\delta v}{\delta y})^2}\}.$

ElseIF(primitive =line)

$\rho = \frac{\sqrt{(\frac{\delta u}{\delta x} * \triangle x + \frac{\delta u}{\delta y} * \triangle y)^2 + (\frac{\delta v}{\delta x} * \triangle x + \frac{\delta v}{\delta y} * \triangle y)^2}}{\sqrt{\triangle x^2 + \triangle y^2}}.$

else

$\rho = 1.$

$\lambda(x,y) = log_2[\rho(x,y)]$

If (Texture_Magnification_Filter =linear) AND (Texture_Minification_Filter =Nearest_Mipmap_Nearest or Nearest_Mipmap_Linear)

$C = 0.5.$

else

$C = 0.$

If$\lambda(x,y) \leq C$

filtering mode = magnification.

else

filtering mode = minification.

Texture Base Internal Format	Texture source color	
	C_s (R,G,B)	A_s (A)
Alpha (A)	(0,0,0)	A_t
Luminance (L)	(L_t, L_t, L_t)	1
Luminance_Alpha(LA)	(L_t, L_t, L_t)	A_t
RGB	(R_t, G_t, B_t)	1
RGBA	(R_t, G_t, B_t)	A_t

Table 4.9: Valid color buffer and texture internal formats combinations

Algorithm 4.20 Filtering algorithm

Filtering (u, v, filtering mode, texture_mag_filter, texture_min_filter, texture_wrap_S, texture_wrap_T, LoD (λ))

If (filtering mode = magnification)

If (texture_mag_filter = nearest)

$texel(\tau)$ =Single_fetch (u, v, 0)

ElseIF (texture_mag_filter = linear)

$texel(\tau)$ =Box_fetch (u, v, 0)

ElseIF (filtering mode = minification)

If (texture_min_filter = nearest)

$texel(\tau)$ =Single_fetch (u, v, 0)

ElseIF (texture_min_filter = linear)

$texel(\tau)$ =Box_fetch (u, v, 0)

ElseIF (texture_min_filter = nearest_mipmap_nearest or linear_mipmap_nearest)

$$d = \left\{ \begin{array}{cc} 0 & \lambda \leq 0.5 \\ \lceil \lambda + 0.5 \rceil - 1 & 0.5 < \lambda \leq q + 0.5 \\ & \lambda > q \end{array} \right\}$$

where $q = max\{n = log_2(image\,width), m = log_2(image\,width)\}$

if (texture_min_filter = nearest_mipmap_nearest)

$texel(\tau)$ =Single_fetch (u, v, d)

ElseIF (texture_min_filter = linear_mipmap_nearest)

$texel(\tau)$ =Box_fetch (u, v, d)

ElseIF (texture_min_filter = nearest_mipmap_linear or linear_mipmap_linear)

$$d1 = \left\{ \begin{array}{cc} q & q \leq \lambda \\ \lfloor \lambda \rfloor & otherwise \end{array} \right\}$$

$$d2 = \left\{ \begin{array}{cc} q & q \leq \lambda \\ d1 + 1 & otherwise \end{array} \right\}$$

If (texture_min_filter = nearest_mipmap_linear)

$texel(\tau 1)$ =Single_fetch (u, v, d1)

$texel(\tau 2)$ =Single_fetch (u, v, d2)

ElseIF (texture_min_filter = linear_mipmap_linear)

$texel(\tau 1)$ =Box_fetch (u, v, d1)

$texel(\tau 2)$ =Box_fetch (u, v, d2)

$\tau = [1 - frac(\lambda)] * \tau 1 + frac(\lambda)] * \tau 2$

return τ

Single_fetch (u, v, mipmap array(d))

$$i = \left\{ \begin{array}{cc} \lfloor u \rfloor & S < 1 \\ 2^n - 1 & S = 1 \end{array} \right\}$$

$$j = \left\{ \begin{array}{cc} \lfloor v \rfloor & T < 1 \\ 2^m - 1 & T = 1 \end{array} \right\}$$

return $\tau = \tau(i, j)$ of mipmap array (d)

Box_fetch (u, v, mipmap array(d))

$$i_0 = \left\{ \begin{array}{cc} \lfloor u - 0.5 \rfloor mod 2^n & Texture\,wrap\,S\,is\,repeated \\ \lfloor u - 0.5 \rfloor & otherwise \end{array} \right\}$$

$$j_0 = \left\{ \begin{array}{cc} \lfloor v - 0.5 \rfloor mod 2^m & Texture\,wrap\,T\,is\,repeated \\ \lfloor v - 0.5 \rfloor & otherwise \end{array} \right\}$$

$$i_1 = \left\{ \begin{array}{cc} (i_0 + 1) mod 2^n & Texture\,wrap\,S\,is\,repeated \\ (i_0 + 1) & otherwise \end{array} \right\}$$

$$j_1 = \left\{ \begin{array}{cc} (j_0 + 1) mod 2^m & Texture\,wrap\,T\,is\,repeated \\ (j_0 + 1) & otherwise \end{array} \right\}$$

$\alpha = frac(u - 0.5), \beta = frac(v - 0.5)$

return $\tau = (1 - \alpha)(1 - \beta) * \tau(i_0, j_0) + (\alpha)(1 - \beta) * \tau(i_1, j_0)$

$+(1 - \alpha)(\beta) * \tau(i_0, j_1) + (\alpha)(\beta) * \tau(i_1, j_1)$

4.10.2.3 Texture unit

The texture unit applies the texture function on its input colors to produce the fragment's final color. Texture function is determined based on the texture base internal format and the texture function parameters, table 4.8. Each texture unit receives four colors components:

1. Fragment's primary color (C_f, A_f) that is resulted from the rasterization unit.
2. Texture source color (C_s, A_s) that is fetched by the texture coordinates from the texture image.
3. Texture environment color (C_c, A_c) that is specified by the TexEnv command.
4. Primary color component (C_p, A_p) that is resulted from the previous texture unit. For texture unit 0, primary color component (C_p, A_p) equals fragment's primary color (C_f, A_f).

Then, it computes the primary color components(C_v, A_v) by applying the texture function as specified in the OpenGL ES 1.1 [8], tables 3.15, 3.16, and 3.17. Finally, it scales and clamps the primary color components (C_v, A_v) to calculate the output color component (C_r, A_r):

$$\begin{aligned} C_r &= Clamp(C_v * RGB - Scale)\, to\, [0,1] \\ A_r &= Clamp(A_v * Alpha - Scale)\, to\, [0,1] \end{aligned} \tag{4.19}$$

4.11 Final Color Adapting unit

The final color adapting unit computes the final pixel color by blending the fragment's post-texturing color (C_r, A_r) with the fog color (C_f, A_f) with a user-defined blending factor f. Then. it adjusts the fragment's alpha value using the coverage value. This unit handles the required functions set by OpenGL ES 1.1 [8], section 3.8.

4.11.1 Register File

1. Fog mode: 3-valued integer to choose the fog mode (exp, exp2, linear). Initially, the fog mode is exp.

2. Fog parameters: 3 fixed-point values to specify the fog density (d), fog start (s), and fog end (e).Initially, fog density $d = 1.0$, fog start $s = 1.0$, and fog end $s = 0.0$.

3. Fog color (C_f, A_f): 4 fixed-point values that stores the RGBA values of the fog color. Initially, the fog color is (0,0,0,0).

4. Control flags: 4-bits to enable or disable the fog, the multisampling, line smoothing, and point smoothing. Initially, fog, line smoothing, and point smoothing are disabled whereas the multisampling is enabled.

5. Coverage value: a fixed-point value that is computed in the rasterization unit to modify the alpha value.

4.11.2 Architecture

The post-textured color (C_r, A_r) produces the final color (C, A) according to algorithm 4.21.

4.12 Fragment Processing Unit

Our fragment processing unit is exactly as same as the OpenGL ES 1.1 fragment unit: same pipeline order and operations. It performs blending, stencil test, depth test, and other tests on the incoming fragments and updates the frame buffer. This unit handles the required functions set by OpenGL ES 1.1 [8], chapter 4.

Algorithm 4.21 Final color adapting algorithm

→ If (Fog is enabled)
→ If (Fog mode = exp)
$f = exp(-d * c)$
ElseIF (Fog mode = exp2)
$f = exp(-(d * c)^2)$
ElseIF (Fog mode = linear)
$f = \frac{e-c}{e-s}$ where c is the eye-coordinate distance from the eye to the fragment center.
→ Final RGB color$C = C_r * f + C_f * (1 - f)$
Else
→ Final RGB color$C = C_r$
→ If (Primitive is point AND point smoothing is enabled) OR (Primitive is line AND line smoothing is enabled)
→ Final Alpha value$A = A_r * Coverage\,value$

4.13 Conclusion

Our CUGPU architecture supports more than 90% of the OpenGL ES CL profile commands and their corresponding operations. It supports the graphics operations such as vertex processing, lighting, clipping, rasterization, and texture mapping. However, it does not support the get commands, error handling, and compressed texture images. Among all CUGPU units, some units such as rasterization, texture mapping, and clipping units need more research on their architectures and algorithms whereas other units can be implemented directly such as vertex processing and fragment processing units. For example, the rasterization unit need more investigation and optimization of point smoothing, point sprite, and line smoothing algorithms to be designed incrementally. Also, the triangle rasterization and line rasterization hardware may be combined to save hardware cost. The clipping algorithm may be modified to check the view volume and the user-defined clipping planes in parallel. The texture mapping and data fetch units need more research about management their memory requests to avoid memory congestion. In addition, we need more analysis to determine the required bandwidth between the CUGPU and CUSPARC and between the CUGPU and graphics memory. Also, we need more investigation about the performance and limitations of the Wishbone bus if it is used as off-chip data bus.

Chapter 5

Diamond-Exit Rule Line Rasterization

Line rasterization algorithms are very important in the design of computer graphics hardware, where many images are mostly composed of line segments. Moreover, the scan-line conversion is used for triangle rasterization. This identifies the set of those fragments that approximate the appearance of lines. OpenGL ES 1.1 ideally uses the diamond-exit rule to determine the fragments that are produced by drawing a line segment. As shown in figure 5.1, for each fragment, OpenGL define a diamond-shaped region:

$$R_f = \left\{(x,y) \mid \left|x - x_f\right| + \left|y - y_f\right| < \frac{1}{2}\right\} \tag{5.1}$$

wherex_f and y_f are the fragment f coordinates in the window-space. A line segment, starting at P_a and ending at P_b, produces a fragment if the line intersects the diamond-shaped region R_f of the fragment except if the end point P_b is located in R_f. Figure 5.1 shows the diamond area for the x-major and y-major lines. For x-major lines ($-1 \leq slope\,(m) \leq 1$) the diamond includes the higher left edge, higher right edge, and top corner of the diamond; it excludes the remaining corners and edges. For y-major lines ($-\infty < slope < -1\, or\, 1 < slope < \infty$), the diamond includes the higher left edge, higher right edge, top corner, and right corner; it excludes the remaining corners and edges.

For a line segment, starting at P_a and ending at P_b, the fragments may be divided into three parts. The starting-point fragment, fragment contains the P_a; it is produced if P_a is located on the diamond area and the line intersects the diamond boundaries. The end-point fragment, one contains the P_b; it is produced if the line intersects and exits the diamond boundaries. The remaining fragments; each is drawn if the line intersects its diamond boundary.

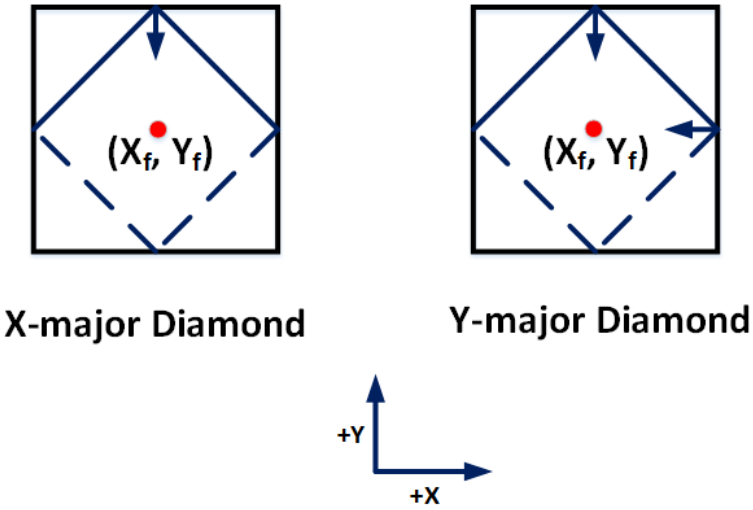


Figure 5.1: Diamond area.

The initial and final conditions of the diamond-exit rule are difficult to implement. So, the GL implementer is free to choose the line segment rasterization algorithm, subject to the following rules:

1. The resulting fragment coordinates do not deviate more than one fragment from the corresponding fragment produced by the diamond-exit rule algorithm.
2. The error, in the number of fragments, is less or equal to one fragment.
3. For an x-major line, no two fragments may be produced that lie in the same column (for a y-major line, no two fragments may appear in the same row).

Then, the color, depth, and texture coordinates are determined for the produced fragment by interpolating the end-points, P_a and P_b, associated data. A fragment with center $P_r = (X_r, Y_r)$ is at a distance t

$$t = \frac{(P_r - P_a) \bullet (P_b - P_a)}{\| P_b - P_a \|^2} \tag{5.2}$$

on the line from point P_a. This distance t is used as the interpolation value for any associated data f for the fragment, which may be the color components (R,G,B,A), the depth value(Z), or the texture coordinates (S,T), based on the value of data at the starting point P_a and the ending point P_b. If the associated data f is the color components (R,G,B,A) or the depth value(Z), it is calculated as:

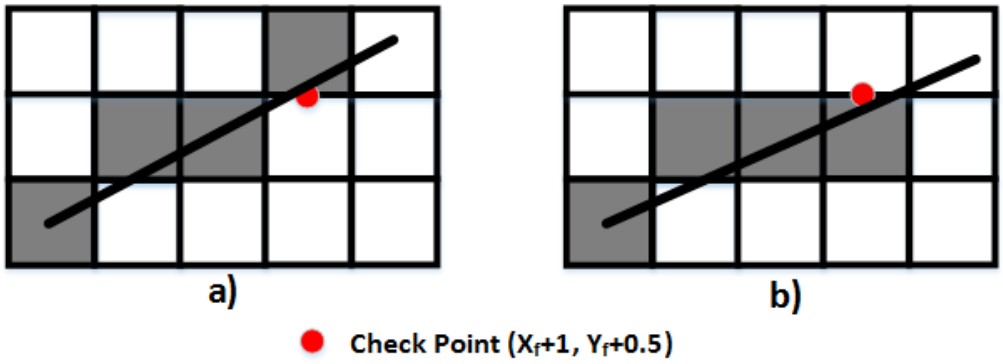


Figure 5.2: The mid-point algorithm Illustration

$$f = (1-t)*f_a + (t)*f_b \tag{5.3}$$

If the associated data f is the texture coordinates (S,T), it is calculated as:

$$f = \frac{(1-t)*\frac{f_a}{w_a} + (t)*\frac{f_b}{w_b}}{(1-t)*\frac{Q_a}{w_a} + (t)*\frac{Q_b}{w_b}} \tag{5.4}$$

where W_a and W_b are the clip w coordinates of the starting and ending endpoints of the segments, respectively; Q_a and Q_b are the texture Q coordinates of the starting and ending endpoints of the segments, respectively.

We divided the line rasterization that satisfied the diamond-exit rules into three main problems. First, we implemented the line drawing without considering the initial and final conditions of the first and last fragments. We modified Bresenham algorithm [39] to satisfy the diamond-exit rule. This might generate the first and last fragment correctly. Second, we checked the first and last fragments using Moreton [40] rasterization technique. It depends on calculating the Manhattan distances for the starting and ending vertices from the diamond edges. Finally, we adapted the line interpolation algorithm to use the incremental linear interpolation method [41, 42].

5.1 Line Rasterization Algorithms

The line rasterization is based on the line equation and how to use the relation between the consecutive fragments. We discuss the line drawing in the first octant($0 \leq Slope(m) \leq 1$) and the lines, in the others octants, are switched to the first octant before drawing. For x-major line, the line is moving faster in x than in y. So, x is incremented in each step and y

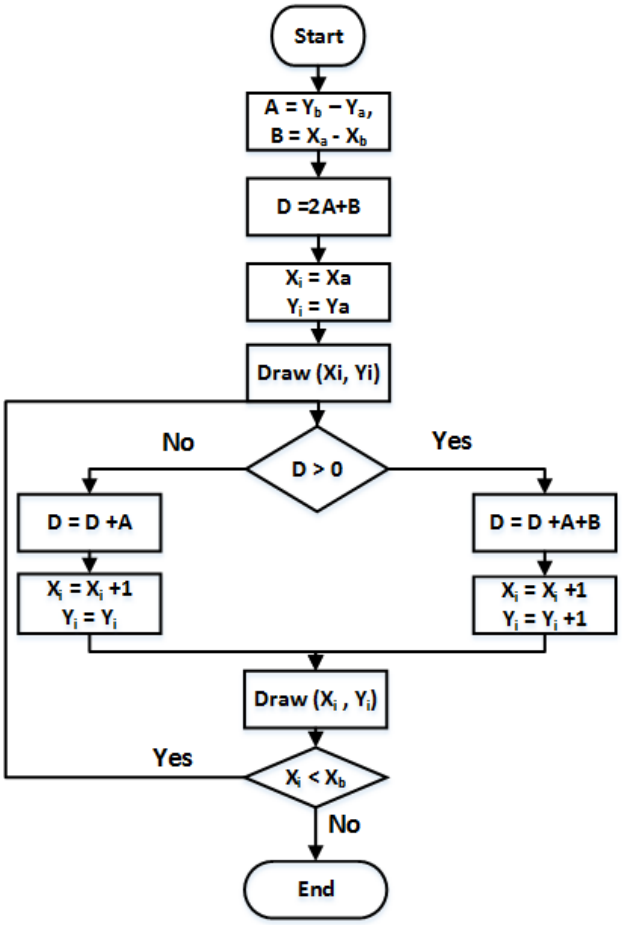


Figure 5.3: Bresenham algorithm flow chart

may be incremented or not based on the line slope. The basic line equation $y = m * x + c$ can be implemented directly by moving from the starting-point x coordinate (x_a) to the end-point (x_b) and substituting in this equation. But, this algorithm performs a multiplication for every step in x.

The direct implementation of the line equation is expensive. So, some algorithms [41, 43, 44] were developed to overcome this algorithm such as mid-point algorithm and Bresenham algorithm. These algorithms have the same visualization of the line, but area and delay of Bresenham algorithm is less than the area and delay of mid-point algorithm, respectively. Also, Bresenham algorithm can be modified to satisfy the OpenGL rules easily.

The basic mid-point algorithm depends on the relation between the consecutive fragments for the different slopes. For a line with the two end-points$P_a(x_a, y_a)$ and $P_b(x_b, y_b)$, the line equation can be written as

$$
\begin{aligned}
f(x,y) &= (y_b - y_a) * x + (x_a - x_b) * y + ((y_a - y_b) * x_a + (x_b - x_a) * y_a) \\
&= A * x + B * y + C
\end{aligned} \tag{5.5}
$$

Where $A = (y_b - y_a)$, $B = (x_a - x_b)$, and $C = ((y_a - y_b) * x_a + (x_b - x_a) * y_a)$. For x-major lines, if we draw a fragment f where (x_f, y_f) is the pixel center, the next eligible pixel would be $(x_f + 1, y_f)$ or $(x_f + 1, y_f + 1)$. So, they used the midpoint $(x_f + 1, y_f + 0.5)$to determine the next drawn fragment. Figure 5.2 illustrates the two cases of the mid-point algorithm. In the left, $f(x_f + 1, y_f + 0.5) = -ve$ value since the line passes above the mid-point; in the right, $f(x_f + 1, y_f + 0.5) = +ve$ value because the line passes below the mid-point. It will be very difficult to substitute in it for each point, so they used the incremental method based on the incremental relations between the subsequent mid-points.

Bresenham modified this algorithm for the special case when the starting point $P_a(x_a, y_a)$ located exactly at the fragment center. In this case, the initial distance d is given by

$$
\begin{aligned}
d = f(x_a + 1, y_a + 0.5) &= f(x_a, y_a) + A + \frac{B}{2} \\
&= A + \frac{B}{2}
\end{aligned} \tag{5.6}
$$

Because this algorithm concerned with the sign of the distance only, this equation is multiplied by 2 to avoid the division

$$
d = f(x_a + 1, y_a + 0.5) = 2A + B \tag{5.7}
$$

Figure 5.3 shows the flow chart of the Bresenham algorithm. Also, the candidates' fragments have some patterns, based on the line slope m. So, this method was extended for the double step, triple, and quadratic line drawing [45–47].

5.2 Our Modified Bresenham Algorithm

The Bresenham algorithm assumes that the end-points' coordinates are located at the fragments centers. But the OpenGL coordinates could be any 32 bits fixed point value: OpenGL uses 16 bits for the integer part and 16 bits for the fraction part. So, we adapted this algorithm to satisfy the diamond-exit rule line. These modifications concerned about

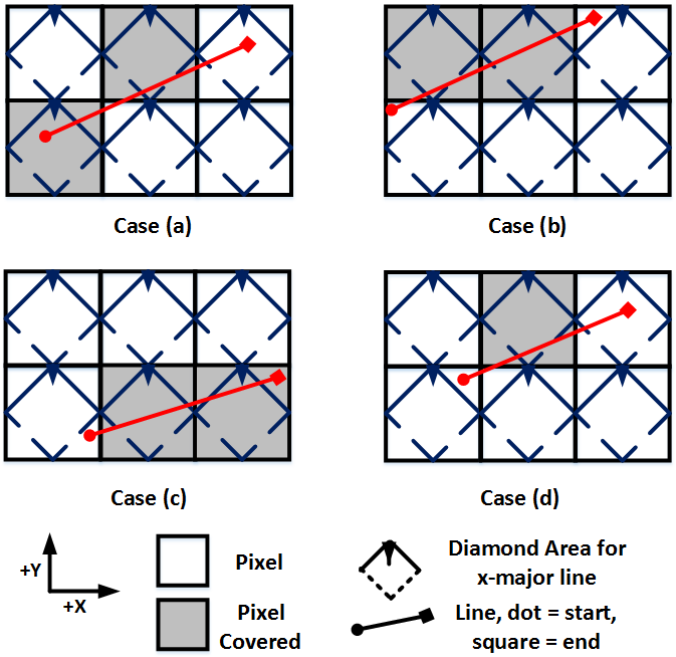


Figure 5.4: Line drawing examples

how to specify the first candidate fragment for the Bresenham algorithm and the calculation of the initial distance d, the substitution of the first check point in the line equation $f(x,y)$. Figure 5.4 demonstrates different lines' starting points and their resulting first drawn fragment. This shows that, in cases (b,c,d), the first drawn fragment is not the fragment of the starting point P_a. For a x-major line with the starting vertex (x_a, y_a) that is located inside the fragment with center (x_1, y_1), the center of the first drawn fragment could be (x_1, y_1), (x_1, y_1+1), (x_1+1, y_1), or (x_1+1, y_1+1) as shown in the figure 5.4.

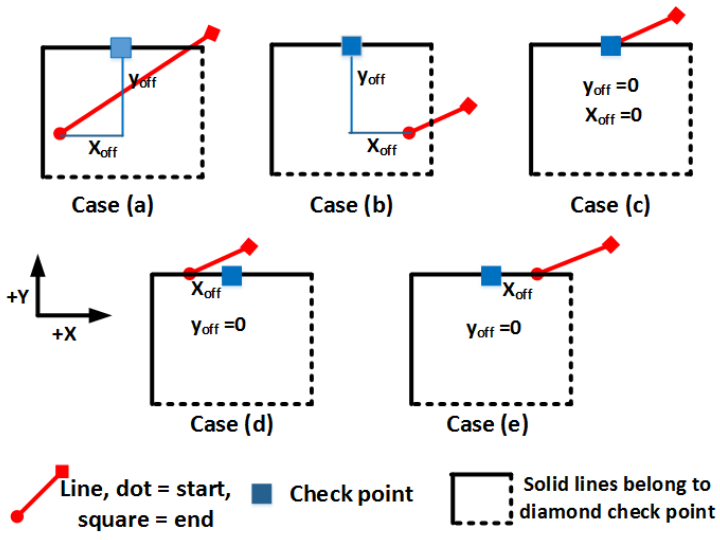


Figure 5.5: The check point distance

To apply the Bresenham algorithm on the given line, we need to choose the start fragment center. We chose the center x coordinate to be x_1 because this does not affect the correctness of Bresenham algorithm. However, this might draw one extra fragment, the starting-point fragment as shown in cases (c) and (d) in figure 5.4. This error is handled by the Moreton [40] algorithm. So, the two candidate fragments' centers are (x_1, y_1) or $(x_1, y_1 + 1)$. To distinguish between them, we check the initial mid-point $(x_c, y_c) = (x_1, y_1 + 0.5)$ versus the line equation. If the line passes above it, then the first drawn fragment's center is $(x_1, y_1 + 1)$; otherwise, the first drawn fragment's center is (x_1, y_1). So, the initial distance is $d = f(x_c, y_c)$

$$\begin{aligned} d = f(x_c, y_c) &= f(x_1, y_1 + 0.5) \\ &= A * x_c + B * y_c + C \\ &= A * (x_a + x_{off}) + B * (y_a + y_{off}) + C \\ &= [A * (x_a) + B * (y_a) + C] + [A * (x_{off}) + B * (y_{off})] \\ &= A * (x_{off}) + B * (y_{off}) \end{aligned} \tag{5.8}$$

Where x_{off} and y_{off} are the distance from the starting point (x_a, y_a) of the line to the check point (x_c, y_c) as shown in figure 5.5

$$\begin{aligned} x_{off} &= x_c - x_a \\ &= 0.5 - frac(x_a) \in]-0.5, 0.5] \end{aligned} \tag{5.9}$$

$$\begin{aligned} y_{off} &= y_c - y_a \\ &= 1 - frac(y_a) \in [0, 1[\end{aligned} \tag{5.10}$$

If distance (d) is less than or equal 0, the output fragment's center is (x_1, y_1), the next check point is $(x_c + 1, y_c)$, and the incremental distance step ($\triangle d_1$) is

$$\begin{aligned} \triangle d_1 &= f(x_c + 1, y_c) - f(x_c, y_c) \\ &= A \end{aligned} \tag{5.11}$$

Otherwise, the output fragment's center is $(x_1, y_1 + 1)$, the next check point is $(x_c + 1, y_c + 1)$, and the incremental distance step ($\triangle d_2$) is

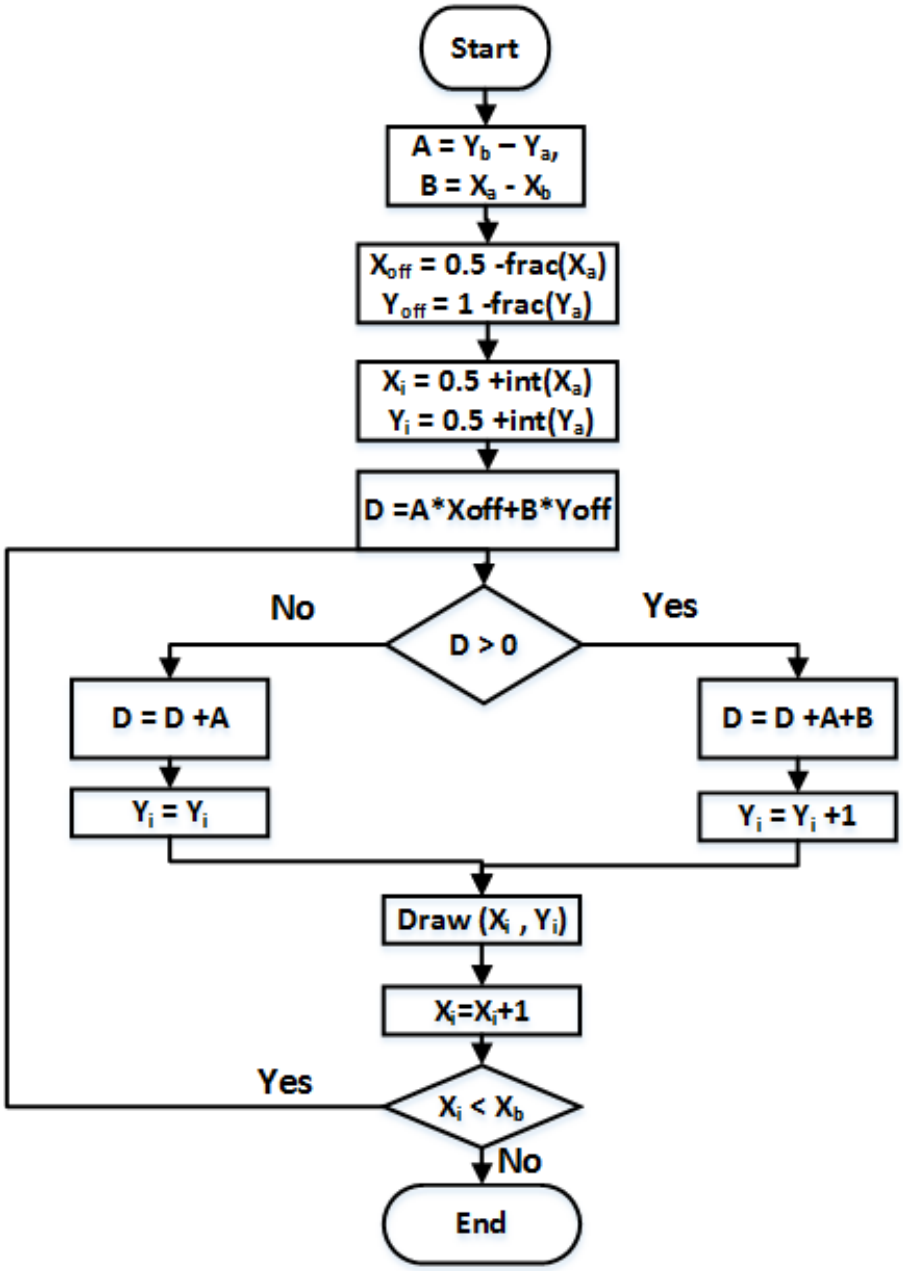


Figure 5.6: Our line drawing algorithm flow chart

$$\begin{aligned} \triangle d_2 &= f(x_c+1, y_c+1) - f(x_c, y_c) \\ &= A + B \end{aligned} \tag{5.12}$$

This is recursively repeated until x_c reaches x_b. Figure 5.6 shows our algorithm for a given x-major line with the end-points $P_a(x_a, y_a)$ and $P_b(x_b, y_b)$.

5.3 Initial and final Conditions Handling

The initial and final conditions of the diamond-exit are difficult to implement. So, we trade them separately. The initial and final conditions specify the rules for drawing the fragments which include the starting point and ending point, respectively. These fragments are drawn if the line intersects the fragment diamond area and the end point of this line does not fall inside their diamond's areas. Taylor [48] and Brown [49] algorithms' ideas, for drawing the anti-aliasing lines, could be modified to check the initial the final fragments. This depends on checking the starting and ending points against the equations of their diamonds' edges. Also, Moreton [40] developed an algorithm to check the initial and final conditions and to cull the short lines.

Moreton algorithm checks the initial the final conditions by calculating the distances from these points to the fragments' centers. We implemented the Moreton algorithm, in parallel to the Bresenham algorithm, to check the initial and final conditions because our modified Bresenham algorithm may produce the first and last fragments incorrectly if the starting point locates outside the diamond area of the fragment, as shown in figure 5.4(c,d). So, the first and last fragments and their associated data that are produced from the Bresenham algorithm are stored in a buffer and do not send to the texture mapping unit until the Moreton algorithm solves the initial and final conditions. However, we could implement the initial condition only since OpenGL allows generating one incorrect fragment.

5.4 Incremental Linear Interpolation

The associated color, texture coordinates, and depth with the produced fragment is calculated by interpolating the associated data with the starting and final fragments. Since the Bresenham algorithm generates the fragments incrementally, it is useful to adapt the interpolation equations to interpolate the associated data incrementally. Bresenham [41] and Field [42] interpolated the associated data incrementally. For example, the equation $f = f_a + t * (f_b - f_a)$, where t is the interpolation ratio and f is the associated data, can be implemented by using f_a as initial value and $(\Delta t) * (fb - fa)$ as an incremental step. Moreover, Wu [47] and Bao [45] proposed the double step and quadratic line generation. And Graham [46] implemented the double and triple step line interpolation. We implement the single-step incremental linear interpolation because the double, triple, and quadratic

steps incremental interpolation require more hardware which may be unsuitable for the embedded systems' GPUs.

We adapt the OpenGL interpolation equations to use the incremental linear interpolation. For a line with the two end-points $P_a(x_a, y_a)$ and $P_b(x_b, y_b)$, the interpolation valuet_1, for a produced fragment center $P_r(x_r, y_r)$, is calculated by

$$\begin{aligned} t_1 &= \frac{(P_r - P_a) \bullet (P_b - P_a)}{\| P_b - P_a \|^2} \\ &= \frac{[(x_r - x_a), (y_r - y_a)].[(x_b - x_a), (y_b - y_a)]}{[(x_b - x_a) + (y_b - y_a)]^2} \\ &= \frac{(x_b - x_a)}{[(x_b - x_a) + (y_b - y_a)]^2} x_r + \frac{(y_b - y_a)}{[(x_b - x_a) + (y_b - y_a)]^2} y_r \\ &\quad + \frac{-x_a(x_b - x_a) - y_a(y_b - y_a)}{[(x_b - x_a) + (y_b - y_a)]^2} \\ &= A_1 * x_r + B_1 * y_r + C_1 \end{aligned} \tag{5.13}$$

Where $A_1 = \frac{(x_b - x_a)}{[(x_b - x_a) + (y_b - y_a)]^2}$, $B_1 = \frac{(y_b - y_a)}{[(x_b - x_a) + (y_b - y_a)]^2}$, and $C_1 = \frac{-x_a(x_b - x_a) - y_a(y_b - y_a)}{[(x_b - x_a) + (y_b - y_a)]^2}$. However, we can interpolate some associated data with respect to the x-coordinates only and neglect the y-coordinates effect because we deal with first-octant lines only. These lines' angle Θ is in the range from $0°$ to $45°$. For a line with slope near $0°$, there is a small change in y-coordinates over the line produced fragments' coordinates. Also, for a line with slope near $45°$, the line moves with the same speed in the x-direction and y-direction. So, if we interpolate versus the x-coordinates, the resulted value t_2(equation 5.14) is approximately equal the exact t_1 (equation 5.13). Moreover, if the line angle is near $23°$, the error is still accepted for the color components because their range is from 0 to 1. The interpolation value t_2 is calculated by

$$\begin{aligned} t_2 &= \frac{(x_r - x_a)}{(x_b - x_a)} \\ &= \frac{1}{(x_b - x_a)} x_r + \frac{-x_a}{(x_b - x_a)} \\ &= A_2 * x_r + C_2 \end{aligned} \tag{5.14}$$

Where $A_2 = \frac{1}{(x_b - x_a)}$ and $C_2 = \frac{-x_a}{(x_b - x_a)}$. We used the interpolation value t_1(equation 5.13) for interpolating the texture coordinates (S, T, Q) and depth (Z). However, we used the interpolation value t_2(equation 5.14) for interpolating the color components (R, G, B, A) since its error, as we discuss in chapter 6, is accepted for the color range $[0, 1]$.

5.4.1 Color Interpolation

The color components (R, G, B, A) are interpolated using the interpolation equation

$$\begin{aligned} f &= f_a * (1-t) + f_b * t \\ &= f_a + (t) * (f_b - f_a) \end{aligned} \tag{5.15}$$

Since t equals 0 at the starting point, the initial color value, which is assigned to the first drawn fragment, $f_{ini} = f_a$. By substituting by the interpolation value $t_2$5.14into the interpolation equation

$$\begin{aligned} f &= f_a + (t) * (f_b - f_a) \\ &= f_a + (A_2 * x_r + C_2) * (f_b - f_a) \\ &= [A_2 * (f_b - f_a)] * x_r + [f_a + C_2 * (f_b - f_a)] \end{aligned} \tag{5.16}$$

If the drawn fragment is(x_i, y_i), the next drawn fragment is $(x_i + 1, y_i)$ or $(x_i + 1, y_i + 1)$. So, the difference of x (Δx) for two consecutive fragments is 1 and the difference of the color components are$\Delta f = A_2 * (f_b - f_a)$.

5.4.2 Depth Interpolation

The depth (Z) is interpolated is interpolated using the interpolation equation 5.15 with $f_{ini} = f_a$. But, it used the interpolation value $t_1$5.13

$$\begin{aligned} f &= f_a + (t) * (f_b - f_a) \\ &= f_a + (A_1 * x_r + B_1 * y_r + C_1) * (f_b - f_a) \\ &= [A_1 * (f_b - f_a)] * x_r + [B_1 * (f_b - f_a)] * y_r \\ &\quad + [f_a + C_1 * (f_b - f_a)] \end{aligned} \tag{5.17}$$

For a drawn fragment (x_i, y_i); if the next drawn fragment is $(x_i + 1, y_i)$, than$\Delta x = 1$ and $\Delta y = 0$. So, the difference of the depth component $\triangle f_1 = A_1 * (f_b - f_a)$. Otherwise,$\Delta x = 1$, $\Delta y = 1$, and the difference of the depth component is $\triangle f_2 = (A_1 + B_1) * (f_b - f_a)$.

5.4.3 Texture Coordinates Interpolation

The rasterization unit receives the texture coordinates in the format (S,T,R,Q) which represent the 3D-texture address $(\frac{S}{Q},\frac{T}{Q},\frac{R}{Q})$. However, the OpenGL ES 1.1 supports the 2D-texture images only. So, we calculated $(\frac{S}{Q},\frac{T}{Q})$ only. For generating these coordinates, we had to apply the OpenGL interpolation equation

$$f = \frac{(1-t)*\frac{f_a}{w_a}+(t)*\frac{f_b}{w_b}}{(1-t)*\frac{Q_a}{w_a}+(t)*\frac{Q_b}{w_b}} \tag{5.18}$$

Where f_a and f_b are the texture coordinates (S,T) associated with the starting and ending endpoints of the segment, respectively; W_a and W_b are the clip w coordinates of the starting and ending points of the segments, respectively; Q_a and Q_b are the texture Q coordinates of the starting and ending endpoints of the segments, respectively. It is hard to implement this equation per produced fragment in the hardware. So, we tried to simplify this equation by two methods.

The first method is that we interpolate for each coordinate $(S`,T`,Q`)$ independently where $(S`,T`,Q`)$ is a modified texture coordinates that is used to replace the division operations by multiplications. Then, texture mapping unit determined $(\frac{S}{Q},\frac{T}{Q}) = (\frac{S`}{Q`},\frac{T`}{Q`})$ using look-up tables.

$$\begin{aligned}
\frac{f`}{Q`} &= \frac{(1-t)*\frac{f_a}{w_a}+(t)*\frac{f_b}{w_b}}{(1-t)*\frac{Q_a}{w_a}+(t)*\frac{Q_b}{w_b}} \\
&= \frac{\frac{f_a}{w_a}+(t)*(\frac{f_b}{w_b}-\frac{f_a}{w_a})}{\frac{Q_a}{w_a}+(t)*(\frac{Q_b}{w_b}-\frac{Q_a}{w_a})} * \frac{W_a*W_b}{W_a*W_b} \\
&= \frac{(f_a*W_b)+(t)*(f_b*W_a-f_a*W_b)}{(Q_a*W_b)+(t)*(Q_b*W_a-Q_a*W_b)} \\
&= \frac{K+(t)*L}{M+(t)*N}
\end{aligned} \tag{5.19}$$

Where $K=(f_a*W_b), L=(f_b*W_a-f_a*W_b), M=(Q_a*W_b), and\, N=(Q_b*W_a-Q_a*W_b)$. Then, we calculate the $f``$ and $Q`$ separately by the same incremental methodology that is used for the depth interpolation

$$\begin{aligned}
f_{int}` &= K=(f_a*W_b) \\
\triangle f_1 &= A_1*L=A_1*(f_b*W_a-f_a*W_b)
\end{aligned}$$

$$\triangle f_2 = (A_1 + B_1) * L = (A_1 + B_1) * (f_b * W_a - f_a * W_b) \tag{5.20}$$

$$\begin{aligned} Q_{int}\text{`} &= M = (Q_a * W_b) \\ \triangle Q_1 &= A_1 * N = A_1 * (Q_b * W_a - Q_a * W_b) \\ \triangle Q_2 &= (A_1 + B_1) * N = (A_1 + B_1) * (Q_b * W_a - Q_a * W_b) \end{aligned} \tag{5.21}$$

The second method implementes the incremental linear interpolation procedure directly on the texture interpolation equation with some assumptions. First, we use the interpolation value t_2(equation 5.14). Also, we substitute with $x_r = \frac{(x_a + x_b)}{2}$ in the final expression of Δf.

$$\begin{aligned} f = \frac{f\text{`}}{Q\text{`}} &= \frac{K + (t) * L}{M + (t) * N} \\ &= \frac{K + (A_2 * x_r + C_2) * L}{M + (A_2 * x_r + C_2) * N} \\ &= \frac{(A_2 * L) * x_r + (K + C_2 * L)}{(A_2 * N) * x_r + (M + C_2 * N)} \\ &= \frac{G * x_r + E}{J * x_r + I} \end{aligned} \tag{5.22}$$

Where $G = (A_2 * L), E = (K + C_2 * L), J = (A_2 * N), and N = (M + C_2 * N)$. The initial texture coordinates f_{ini} and the incremental step Δf are calculated by

$$f_{ini} = \frac{K}{M} = \frac{f_a}{Q_a} \tag{5.23}$$

$$\begin{aligned} \triangle f &= f_2 - f_1 \\ &= \frac{(G * I - E * J)}{(I + J * x_{avg}) * (I + (J + 1) * x_{avg})} \end{aligned} \tag{5.24}$$

Where f_1 and f_2 are the associated produced data of two consecutive fragments with x-coordinates x_1 and $x_1 + 1$, respectively. This equation produces errors due to our two approximations; especially for the long lines. So, we interpolate the texture coordinates with the first method (equation 5.19).

5.5 RTL Implementation

Figure 5.7 shows the complete line rasterization algorithm. We implement it using the VHDL language to have a clear vision about its cost and speed. Figure 5.8shows the line rasterization block diagram. It receives the starting vertices $P_a(x_a, y_a)$ and the

ending vertices $P_b(x_b, y_b)$ in the window space. Also, it receives a color (R, G, B, A), depth (Z), and texture coordinates (S, T, Q) of each vertex. It produces the drawn fragment coordinates with their associated data.

The architecture can be divided into two main blocks: line drawing and line interpolation. For the line drawing, the step calculation block determines the Bresenham steps $(A, B, A+B)$ and the octant number. Then, the octant switch blocks transforms the line to the first octant. The offset calculation unit calculates the x_{off},and y_{off}. The initial distance block calculates the initial distance $D = A * x_{off} + B * y_{off}$. The fragment generation unit produces the coordinates of the drawn fragments sequentially. For the line interpolation, the incremental interpolation unit calculates the interpolation steps (A_1, B_1, A_2) using look-up tables. The texture and color units calculates the initial color components

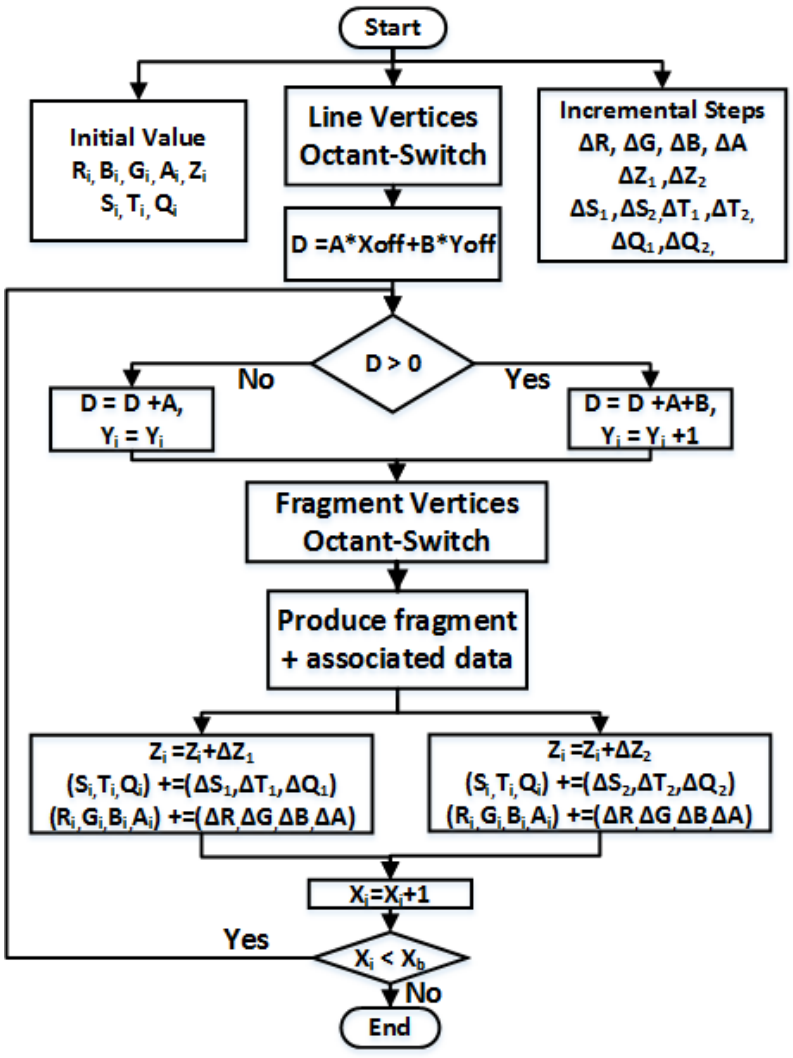


Figure 5.7: Line rasterization algorithm flow chart:

Figure 5.8: Line rasterization architecture

Octant	$Z_2 = sign(\Delta y)$	$Z_1 = sign(slope)$	$Z_0 = (slope \geq 1)$
0	0	0	0
1	0	0	1
2	0	1	1
3	0	1	0
4	1	0	0
5	1	0	1
6	1	1	1
7	1	1	0

Table 5.1: The line octant encoding

$(R_{ini}, G_{ini}, B_{ini}, A_{ini})$, texture coordinates $(S_{ini}, T_{ini}, Q_{ini})$, and depth values (Z_{ini}). Also, it calculates some parts of the incremental steps. Then, incremental interpolation steps calculates the incremental steps for the color $(\triangle R, \triangle G, \triangle B, \triangle A)$, the texture coordinates $(\triangle S, \triangle T, \triangle Q)$, and the depth $(\triangle Z)$. Then the data generation block produces the associated data of the generated fragments sequentially.

5.5.1 Step Calculation Unit

The Bresenham algorithm deals with lines in the first octant only; slope $m \in [0, 1]$ and $(x_b > x_a)$. So, the given line must be transformed to the first octant. Then, the Bresenham algorithm is applied on the transformed line. Finally, the produced fragments are transformed back to its initial octant. Step calculation unit encodes the eight octants by the octant number (Z_2, Z_1, Z_0) as a function of Δy sign, slope m sign, and $(\Delta y - \Delta x)$ sign, where $\Delta y = y_b - y_a$ and $\Delta x = x_b - x_a$, as shown in table 5.1.

Also, it calculates the Bresenham incremental steps $A = (y_b - y_a)$,$B = (x_a - x_b)$, and $A_{plus}B = (y_b - y_a) + (x_a - x_b)$. It feeds the initial distance unit with A and B to determine the initial distance $d_{ini} = A * x_{off} + B * y_{off}$. Also, it feeds the fragment generation unit with A and $A_{plus}B$ that are used as Bresenham incremental steps.

5.5.2 Octant Switch Unit

The line starting and ending points are transformed according the octant number. Table 5.2 demonstrates the produced vertex coordinates(x_o, y_o) for a given vertex (x_i, y_i). Figure 5.9 presented the block diagram of the octant switch unit.

Octant	$Produced\,Vertex$ (x_o, y_o)
0	(x_i, y_i)
1	(y_i, x_i)
2	$(y_i, -x_i)$
3	$(-x_i, y_i)$
4	$(-x_i, -y_i)$
5	$(-y_i, -x_i)$
6	$(-y_i, x_i)$
7	$(x_i, -y_i)$

Table 5.2: The octant vertices transformation

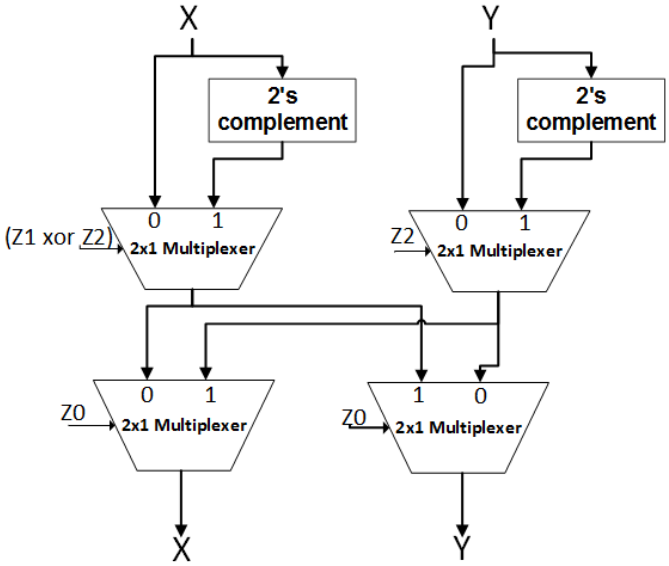


Figure 5.9: The octant switch block diagram

5.5.3 Offset Calculation Unit

It calculates the offset from the starting vertex of a line to the mid-point as shown in figure 5.5. For a line with a starting vertex (x_a, y_a) that is located in a fragment with center (x_i, y_i), the mid-point coordinates are $(x_i, y_i + 0.5)$. So, the initial offset (x_{off}, y_{off})

$$\begin{aligned} x_{off} &= 0.5 - frac(x_a) \\ y_{off} &= 1 - frac(y_a) \end{aligned} \tag{5.25}$$

For a fixed-point input (x_a, y_a), the x_{off} and y_{off} are a 17-bit 2's complement number.x_{off} range is from-0.5 to 0.5 and y_{off} range is from 0 to1. Figure 5.10 demonstrates the block diagram of the offset calculation unit. It are implemented by two 16-bit 2's complement units.

5.5.4 Initial Distance Unit

To determine the first drawn fragment coordinates, (x_i, y_i) or $(x_i, y_i + 1)$, we have to substitute in the line equation by the mid-point $(x_i, y_i + .5)$ in the line equation Initial. So, the initial distance $d_{ini}$is calculated by

$$d_{ini} = A * (x_{off}) + B * (y_{off}) \tag{5.26}$$

If the initial distance is equal or less than 0, the drawn fragment coordinates are (x_i, y_i). Otherwise, the drawn fragment coordinates are $(x_i, y_i + 1)$. We need only to check if the distance is greater than 0. So, we produce the distance in a redundant form, two vectors, to save the final stage adder delay. The A and B are 32-bit 2's complement number. And, x_{off}andy_{off} are 17 bits 2's complement number. We produce the output as two 33-bit vectors by ignoring the least 16-bit. First, 17-partial products are produced from $A * x_{off}$. Another 17-partial products are generated from $B * y_{off}$. Then, the 34-partial products are reduced to two vectors using the Carry-Save adders.

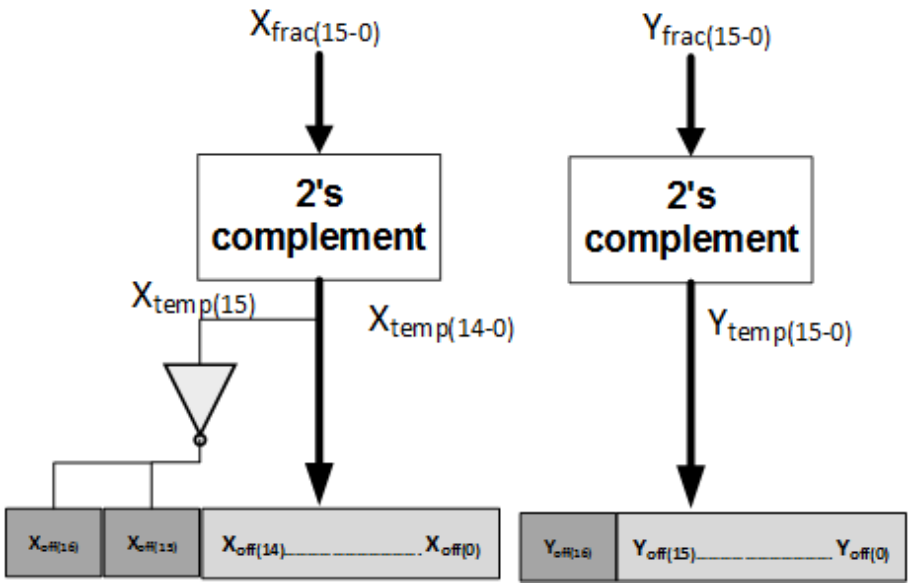


Figure 5.10: The offset calculation block diagram

Figure 5.11: The reduction tree of the initial distance

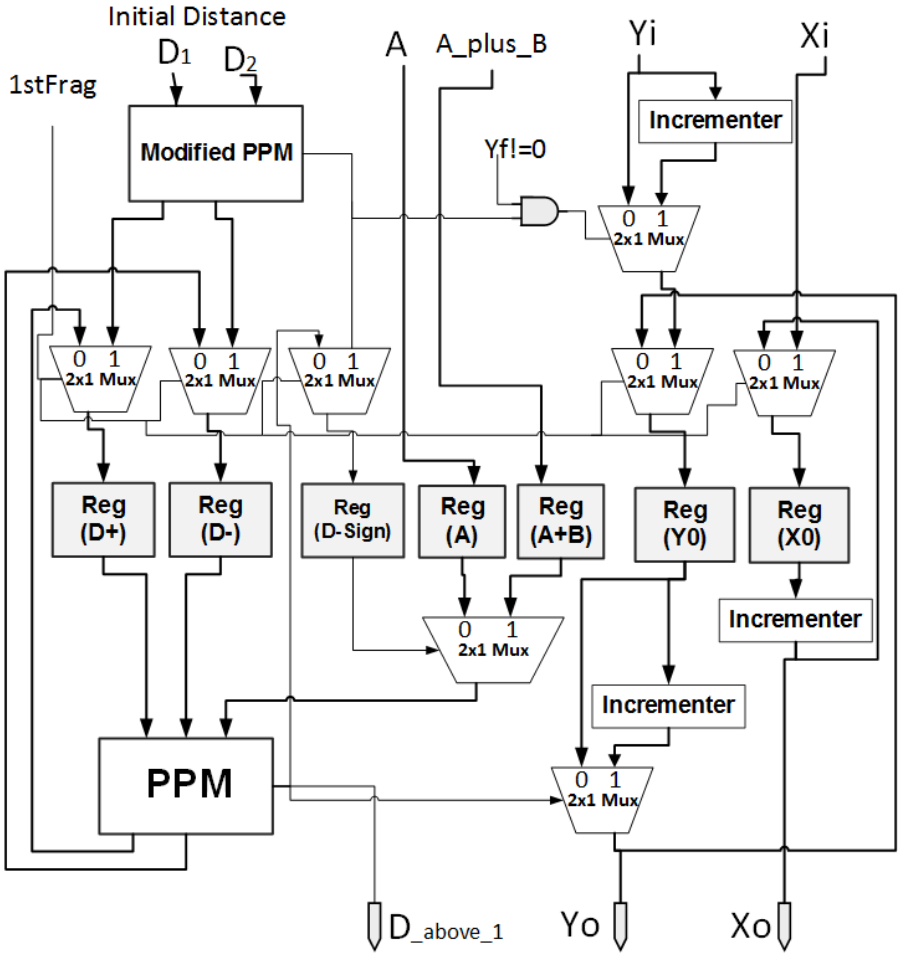


Figure 5.12: The fragment generation block diagram

Figure 5.11 shows the partial product array before the reduction. We ignore the partial products bits that have a weight less than 2^{-16}. So, the reduction tree is 34 partial products, in the black lines' box, with elements have a weight between 2^{-16} and 2^{15}. We reduce them to 2 redundant vectors by eight levels Wallace tree [50].

5.5.5 Fragments' Coordinates Generation Unit

Figure 5.12 shows the block diagram of the fragment generation unit. It produces the drawn fragments sequentially from the first fragment to the last one. It mainly applies the modified Bresenham algorithm. It receives the initial fragment's coordinates (x_i, y_i) and the final fragment's coordinates (x_f, y_f). Also, it receives the initial distance d_{ini} and the incremental steps$A$ and $A_{plus}B$. As we mentioned before, we need the Distance sign only. So, we encode the distance in the binary redundant format [51] using a modified Plus Plus Minus (PPM) adder [52]. Then, we check the distance sign in each step. If this sign is

Digit Value	x^+	x^-
0	0	0
-1	0	1
1	1	0
0	1	1

Table 5.3: Redundant binary representation

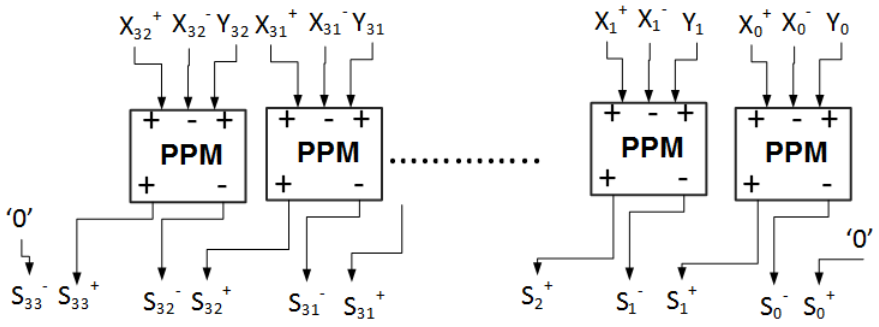


Figure 5.13: PPM adder block diagram

greater than zero, the step $A_{plus}B$ is added to the old distance using the PPM adder and the generated fragment center is (x_i+1, y_i+1); Otherwise, the new distance equals the old distance plus the step A and the generated fragment center is (x_i, y_i+1).

5.5.5.1 Redundant Binary Representation

We used the redundant binary representation with digits $d_i \in \{-1, 0, 1\}$ to represent the distance. Each digit d_i is represented by two bits x^+ and x^- as shown in table 5-3. This representation simplifies the addition and subtraction. The addition can be performed directly with a 2's complement signed number using the Plus Plus Minus (PPM) hybrid adder [52]. Also, the subtraction with a 2's complement signed number is calculated directly using the Plus Minus Minus (PMM) hybrid adder [52].

5.5.5.2 Hybrid PPM Adder

The hybrid PPM adder is used to add a redundant number (x^+, x^-) with a 2's complement number y. The addition is carried out in two steps. First, for each bit location i, an intermediate sum $p_i \in [-1, 2]$ is computed

y_i	x_i^+	x_i^-	t_i	u_i
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	1	0
1	1	1	1	1

Table 5.4: PPM cell truth table

$$\begin{aligned} p_i &= x_i^+ - x_i^- + y_i \\ &= 2t_i - u_i \end{aligned} \tag{5.27}$$

Then, the sum digits (s_i^+, s_i^-), for each bit i, are formed as

$$\begin{aligned} s_i^+ &= t_{i-1} \\ s_i^- &= u_i \end{aligned} \tag{5.28}$$

Figure 5.13 demonstrates the block diagram of the hybrid PPM adder. Each PPM cell computes the expression p_i according to the table 5.4. So, t_i and u_i bits are computed by

$$\begin{aligned} t_i &= x_i^+(y_i + \overline{x_i^-}) + y_i\overline{x_i^-} \\ u_i &= x_i^+ xor\, x_i^+ xor\, y_i \end{aligned} \tag{5.29}$$

For encoding the distance two vectors $(d1, d2)$ by the redundant binary form, we could use the PPM adder with $d1, 0, and\, d2$ as $x+, x-, and\, y$ inputs respectively. So, we optimized the PPM adder given these inputs. The t_i and u_i bits expressions of the modified PPM adder are

$$\begin{aligned} t_i &= x_i^+ + y_i \\ u_i &= x_i^+ xor\, y_i \end{aligned} \tag{5.30}$$

Associated Data	$Initial Value$ f_{ini}	$Initial Value$ $\triangle f_1$	$Initial Value$ $\triangle f_2$
Color (R)	R_a	$A_2 * (R_b - R_a)$	
Color (B)	B_a	$A_2 * (B_b - B_a)$	
Color (G)	G_a	$A_2 * (G_b - G_a)$	
Color (A)	A_a	$A_2 * (A_b - A_a)$	
Depth (Z)	Z_a	$A_1 * (Z_b - Z_a)$	$(A_1 + B_1) * (Z_b - Z_a)$
Texture Coor. (S‘)	$S_a * W_b$	$A_1 * (S_b * W_a - S_a * W_b)$	$(A_1 + B_1) * (S_b * W_a - S_a * W_b)$
Texture Coor. (T‘)	$T_a * W_b$	$A_1 * (T_b * W_a - T_a * W_b)$	$(A_1 + B_1) * (T_b * W_a - T_a * W_b)$
Texture Coor. (Q‘)	$Q_a * W_b$	$A_1 * (Q_b * W_a - Q_a * W_b)$	$(A_1 + B_1) * (Q_b * W_a - Q_a * W_b)$

Table 5.5: The initial and incremental steps of the fragments' associated data

5.5.5.3 Distance Sign Detection

We could not detect if the distance is greater than 0 or not directly since it was encoded in the redundant binary representation. So, we use the carry look ahead tree [53] to check the sign instead. For each digit, we define the propagate signal $P_i = 1$ if the digit equal zero and the generate signal $G_i = 1$ if the digit is greater than one.

$$\begin{aligned} P_i &= x_i^+ \, xnor \, x_i^- \\ G_i &= x_i^+ \, and \, \overline{x_i^-} \end{aligned} \tag{5.31}$$

Then, we implement a three-level tree by grouping each four consecutive propagate and generate into a higher level propagate and generate.

5.5.6 Data Interpolation Unit

In order to interpolate the associated data incrementally, we calculate the initial values f_{ini} and the incremental steps $\triangle f$ for the colors, depth, and texture coordinates before calculating the associated data with the produced fragments. For the color components (R, G, B, A), the initial value and the incremental steps are

$$\begin{aligned} f_{ini} &= f_a \\ \Delta f &= A_2 * (f_b - f_a) \end{aligned} \tag{5.32}$$

For the depth (z), the initial value and the incremental steps are

$$
\begin{aligned}
f_{ini} &= f_a \\
\Delta f_1 &= (A_1) * (f_b - f_a) \\
\triangle f_2 &= (A_1 + B_1) * (f_b - f_a)
\end{aligned} \tag{5.33}
$$

For the texture coordinates (S‘, T‘, Q‘), the initial value and the incremental steps are

$$
\begin{aligned}
f_{int}\text{‘} &= (f_a * W_b) \\
\triangle f_1 &= A_1 * (f_b * W_a - f_a * W_b) \\
\triangle f_2 &= (A_1 + B_1) * (f_b * W_a - f_a * W_b)
\end{aligned} \tag{5.34}
$$

Where $A_1 = \frac{(x_b - x_a)}{[(x_b - x_a) + (y_b - y_a)]^2}$, $B_1 = \frac{(y_b - y_a)}{[(x_b - x_a) + (y_b - y_a)]^2}$, and$A_2 = \frac{1}{(x_b - x_a)}$.$\triangle f_1$and $\triangle f_2$are the incremental steps for the two cases of the distance: distance ($d \leq 0$) and ($d > 0$) respectively. Table 5.5summarizes the initial and incremental steps for each component.

5.5.6.1 Division Look-up Tables

The $A_1, A_2, and A_{1plus} B_1$ variables are shared for all associated data steps and required dividers that consume more area and introduce long delay. However, they depend only on the differences Δx and Δy coordinates. For a screen resolution 640*480, the maximum Δx coordinate is 640 and the maximum Δy coordinate is 480. Also, after we made statics about the line's lengths, we found that, for the simple objects, the average line length is 21 pixels and more than 90 % of lines have lengths less than 60 pixels. Also, for the complex scenes, the average line length is 5 pixels and more than 95 % of lines have lengths less than 20 pixels. So, we implemented them using three look-up tables by representing Δx and Δy by 6-bit each.

5.5.6.2 Color and Texture Preparation

These blocks prepare the step coefficients for the color, depth, and textures components. For the color, they calculate (R_b–R_a), (G_b–G_a), (B_b–B_a), and (A_b–A_a). For the depth, they calculate (Z_b–Z_a). For the texture coordinates, they calculate ($S_b * W_a$–$S_a * W_b$), ($T_b * W_a$–$T_a * W_b$), and ($Q_b * W_a$–$Q_a * W_b$).

5.5.6.3 Incremental Interpolation Steps

This block calculates the final steps Δf_1 and Δf_2 for the color, texture, and depth components as mentioned in table 5.5. It multiplies the step coefficients by the interpolation coefficients $A_1, A_2, and A_{1plus} B_1$.

5.5.6.4 Associated Data Generation

The initial associated data are assigned to the first drawn fragment. Then, the consecutive fragment data are calculated according the distance sign as shown in figure .

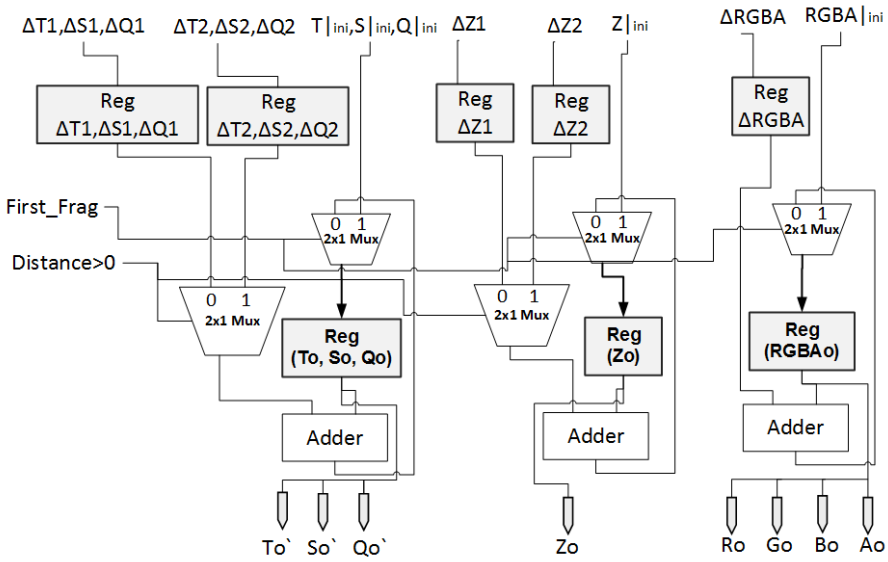


Figure 5.14: The associated data generation block diagram

Chapter 6

Results

We verified our line drawing algorithm using line samples from different scenes:

1. McGuire Graphics Data[54].
2. Physically Based Rendering Scene [55].
3. ORSoC Graphics Accelerator Model [56].

Also, we analyzed these data sets and made statistics about line lengths using different simple objects and complex scenes for a screen resolution $640 * 480$. For the simple objects, the average line length is 21 pixels, as shown in table 6.1a, and more than 90 % of lines have lengths less than 60 pixels as shown in figure 6.1a. Also, for the complex scenes, the average line length is 5 pixels, as shown in table 6.1b, and more than 95 % of lines have lengths less than 20 pixels as shown in figure 6.1b. So, we implemented our look-up tables by representing $\triangle x$ and $\triangle y$ addresses by 6-bits each.

6.1 Color Interpolation Approximation

For the color component, we compare between the resulted fragment's colors of our approximated interpolation equation 5.14 versus the resulted fragment's colors of the exact interpolation equation 5.13. We test lines for three different slopes $m \in [0, 0.5, 1]$ and check the maximum and average errors. The maximum error is the maximum difference between the color of the approximated interpolation equation and the color of the exact interpolation equation for a single fragment. And, the average error is the summation of the difference between resulted colors of all fragments, divided by the number of fragments per line. From results, we conclud these error are negligible for all slopes.

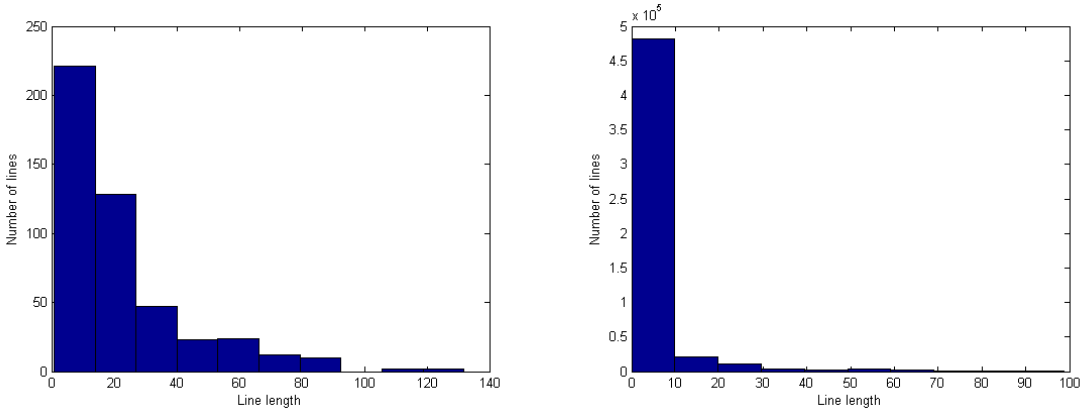
Object	Number of lines	Average ($\triangle x$, $\triangle y$)	Maximum ($\triangle x$, $\triangle y$)	Average length
Bird	275	(8.9, 6.1)	(25.58, 21.4)	10.8
Dude	94	(32.72, 16.8)	(129.88, 69.35)	36.72
Funny	100	(28.78, 17.3)	(86, 60.6)	33.57

(a) Lines of simple objects

Object	Number of lines	Average ($\triangle x$, $\triangle y$)	Maximum ($\triangle x$, $\triangle y$)	Average length
Conference Room	300000	(1.1, 2.67)	(640, 480)	2.8746
Sibenik Cathedral	225117	(2.9, 3.66)	(125.29, 242.3)	4.7
Dabrovic Sponza	151632	(3.25, 7.1)	(301, 133)	8

(b) Lines of complex objects

Table 6.1: Analysis of line lengths for different objects and scenes



(a) Lines of simple objects **(b) Lines of complex scenes**

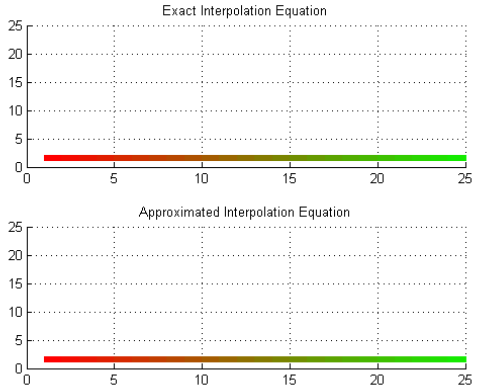
Figure 6.1: Distribution of line's lengths

6.1.1 Test Case 1

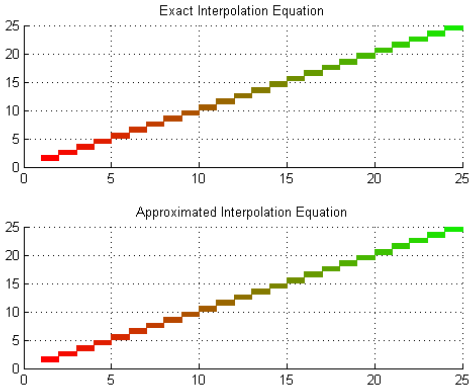
We tested lines with $\triangle x = 25$ for different slopes $m \in [0, 0.5, 1]$. Table 6.2 summarizes the maximum and average errors for each line. Also, figure 6.2 shows the output fragments that are resulted by drawing these lines using the two interpolation equations.

slope (m)	average error	maximum error
$m = 0$	0	0
$m = 0.5$	0.0037	0.0075
$m = 1$	0	0

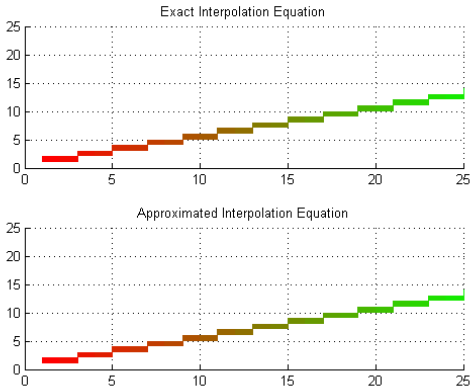
Table 6.2: Color errors for line with $\triangle x = 25$



(a) slope $m = 0$



(b) slope $m = 1$



(c) slope $m = 0.5$

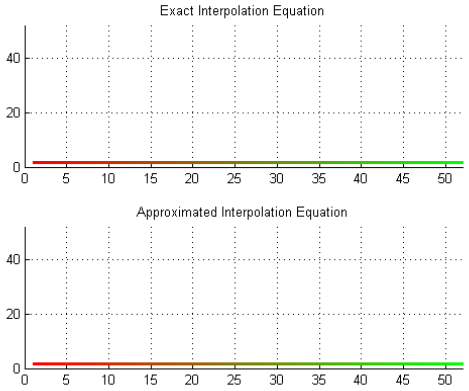
Figure 6.2: Line rasterization example $\triangle x = 25$

6.1.2 Test Case 2

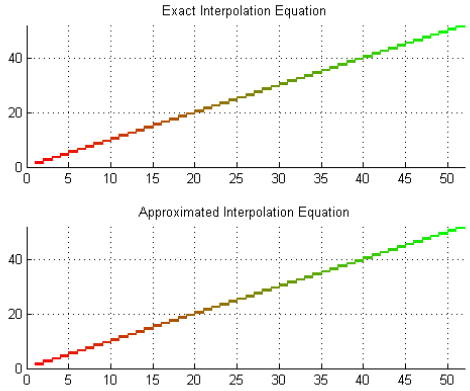
We tested lines with $\triangle x = 50$ for different slopes $m \in [0, 0.5, 1]$. Table 6.3 summarizes the the maximum error and the average error for each line. Also, figure 6.3 shows the output fragments that are resulted by drawing these lines using the two interpolation equations.

slope (m)	average error	maximum error
$m = 0$	0	0
$m = 0.5$	0.002	0.004
$m = 1$	0	0

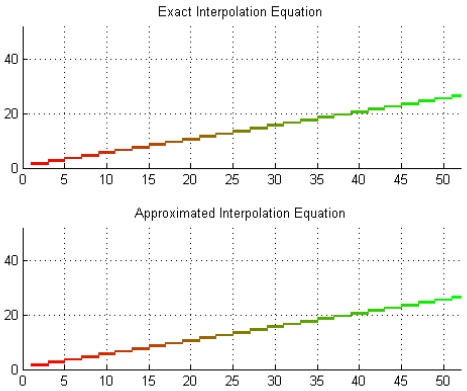
Table 6.3: Color errors for line with $\triangle x = 50$



(a) slope $m = 0$



(b) slope $m = 1$



(c) slope $m = 0.5$

Figure 6.3: Line rasterization example $\triangle x = 50$

6.2 Synthesis Results

we implement three designs with different area and delay constraints and target throughput because we did not have a solid information about the area and delay of individual CUGPU units and we have two implementation parameters that affected the throughput of line rasterization unit:

1. number of clock cycles required for the initial calculations.
2. number of produced fragments per iteration (clock cycle).

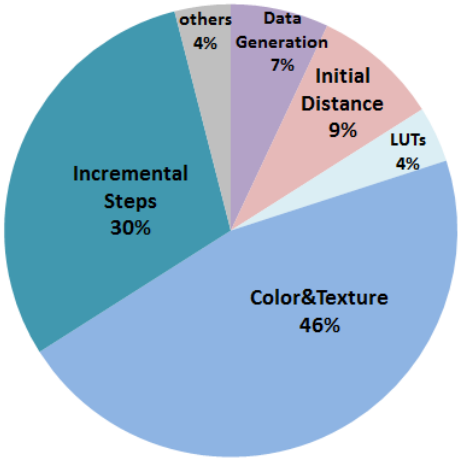
Our designs are

1. Design *a* computes the initial calculation on one clock cycle. Also, it produces one fragment per clock cycle to maximize the frequency. This design has the largest area because it has multiple adders and multipliers to calculate the initial and incremental step values of distance (d), color components (R, G, B, A), texture coordinates (S, T, Q), and depth (Z) in one clock. These preparation units consumed 76% of design *a* area as shown in figure 6.4. So, we shared adders and multipliers resources in designs *b* and *c* to overcome this problem.
2. Design *b* computes the initial and incremental steps on three clock cycle to share the hardware resources and minimize the total area of our design. So, the area of these preparation units is reduced to $30-33\,\%$. However, it still produces one fragment per clock cycle to maximize the frequency. This design saves 46% of design *a* area.
3. Design *c* is the same as design *b*, but it balances between the area and the maximum frequency. It is optimized for a maximum frequency of 200 MHz. It saves 40% of design *b* area. Also, it can be modified to produce two fragment per clock cycle since the iteration period equals half of the minimum clock period.

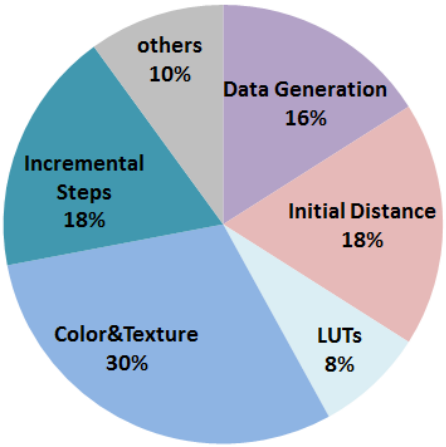
Table 6.4 summarizes the synthesis results of the three designs. Design *b* scores a maximum frequency of 270 MHz with a suitable area of $0.088\,mm^2$, but it can produces only one fragment per clock. Whereas, Design *b* scores a maximum frequency of 200 MHz with a minimum area of $0.052\,mm^2$, but it can be modified to produce two fragment per clock. We can not select the suitable design, design *b* or design *c*, for the CUGPU until we implement the other units and know the bottleneck of our CUGPU.

	Design *a*	Design *b*	Design *c*
minimum clock period	3.7 ns	3.7 ns	5 ns
iteration period	2.5 ns	2.5 ns	2.5 ns
maximum frequency	270 MHz	270 MHz	200 MHz
area	.162 mm^2	.088 mm^2	.052 mm^2

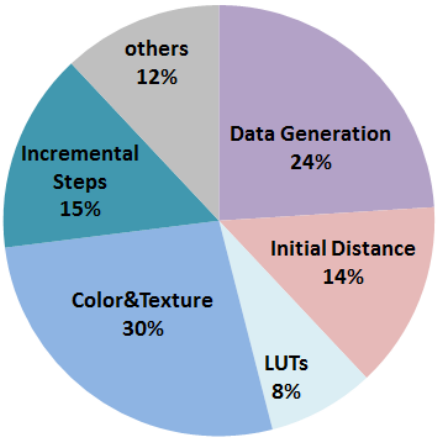
Table 6.4: The synthesis results of line rasterization algorithm



(a) design *a*



(b) design *b*



(c) design *c*

Figure 6.4: Area distribution of line rasterization algorithm

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, our main goal was to introduce the CUGPU, the first embedded GPU in Egypt and to make a clear vision about the cost of implementation. Our contribution is as follows:

1. CUGPU architecture based on the OpenGL ES 1.1 CL profile was proposed. CUGPU provides high-performance support of the fixed-function 3D graphics pipeline. It satisfied the mandatory specifications only to minimize the power and area without deteriorating its performance.

2. A MATLAB model implementing the line rasterization algorithm, that satisfied the OpenGL ES 1.1 rules, is introduced. This model is verified using line samples from popular scenes.

3. Two designs of the line rasterization algorithm were implemented using VHDL code and synthesized on the TSMC 65 nm low power technology. The first design scores a typical clock frequency of 270 MHz and an area of 0.088 mm^2. The second design scores a typical clock frequency of 200 MHz and an area of 0.052 mm^2.

7.2 Future Work

There are several future work to be done for our CUGPU:

1. Implement the smooth and multisampling line drawing algorithms.

2. Implement the triangle rasterization algorithm and share the hardware resources with the line rasterization resources.

3. Implement the other blocks of the CUGPU such as vertex processing, lighting, and texture mapping.

4. Design the hardware driver and our own instruction set.

5. Integrate the whole CUGPU units and verify the complete GPU functionality.

References

[1] E. Hussein, S. Shams, M. Ali, A. Suleiman, K. ElWazeer, E. Sobhy, A. Ibrahim, A. Ibrahim, M. Khairy, M. Fouda, *et al.*, "CUSPARC IP processor: design, characterization and applications," in *Microelectronics (ICM), 2010 International Conference on*, IEEE, 2010, pp. 435–438.

[2] A. A. Suleiman, A. F. Khedr, and S. Habib, "ASIC implementation of Cairo University SPARC "CUSPARC" embedded processor," in *Microelectronics (ICM), 2010 International Conference on*, IEEE, 2010, pp. 439–442.

[3] M. R. Soliman, H. Fahmy, S. Habib, *et al.*, "NoC-based many-core processor using CUSPARC architecture," in *Microelectronics (ICM), 2014 26th International Conference on*, IEEE, 2014, pp. 84–87.

[4] J. Blinn, *Jim Blinn's corner: a trip down the graphics pipeline*. Morgan Kaufmann, 1996.

[5] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. CRC Press, 2008.

[6] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of computer graphics*. CRC Press, 2009.

[7] D. H. Eberly, *3D game engine design*. San Francisco: Morgan Kaufmann Publishers, Inc, 2001.

[8] D. Blythe, A. Munshi, and J. Leech, "OpenGL ES common and common-lite profile specification version 1.1. 12 (full specification)," *The Khronos Group Inc*, 2008.

[9] K. Group. (2015). OpenGL, [Online]. Available: https://www.opengl.org/ (visited on 05/25/2015).

[10] K. Group. (2015). OpenVG, [Online]. Available: https : / / www . khronos . org / openvg/ (visited on 06/01/2015).

[11] Microsoft. (2015). DirectX graphics and gaming, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx (visited on 06/01/2015).

[12] K. Group. (2015). OpenGL, [Online]. Available: https://www.khronos.org/opengl/ (visited on 06/01/2015).

[13] ——, (2015). OpenCL, [Online]. Available: https://www.khronos.org/opencl/ (visited on 06/01/2015).

[14] NVidia. (2015). CUDA, [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html (visited on 06/01/2015).

[15] D. Blythe and A. Munshi, "OpenGL ES common/common-lite profile specification," *The Khronos Group Inc*, 2008.

[16] K. Fatahalian, E. Luong, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan, "Data-parallel rasterization of micropolygons with defocus and motion blur," in *Proceedings of the Conference on High Performance Graphics 2009*, ACM, 2009, pp. 59–68.

[17] M. Fisher, K. Fatahalian, S. Boulos, K. Akeley, W. R. Mark, and P. Hanrahan, "Diagsplit: parallel, crack-free, adaptive tessellation for micropolygon rendering," in *ACM Transactions on Graphics (TOG)*, ACM, vol. 28, 2009, p. 150.

[18] K. Fatahalian, "Evolving the real-time graphics pipeline for micropolygon rendering," PhD thesis, Stanford University., 2011.

[19] T. S. Crow, "Evolution of the graphical processing unit," PhD thesis, University of Nevada Reno, 2004.

[20] NVidia. (1999). GeForce 256, [Online]. Available: http://www.nvidia.com/page/geforce256.html (visited on 06/01/2015).

[21] C. McClanahan, "History and evolution of gpu architecture," *A Survey Paper*, 2010.

[22] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[23] NVidia., "8800 GPU architecture overview," 2006.

[24] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE micro*, vol. 30, no. 2, pp. 56–69, 2010.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia Tesla: a unified graphics and computing architecture," *Ieee Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[26] P. N. Glaskowsky, “Nvidia’s Fermi: the first complete GPU computing architecture. white paper.,” 2009.

[27] NVidia., “Nvidia’s next generation cuda compute architecture,” 2009.

[28] ——, (2014). Whitepaper: NVIDIA GeForce GTX 980., [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF (visited on 06/01/2015).

[29] AMD. (2015). Amd accelerated processing units (apus), [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/apu (visited on 06/01/2015).

[30] J. Peddie. (2014). Mobile devices and the GPUs inside, [Online]. Available: http://jonpeddie.com/publications/mobile-devices-and-the-gpus-inside/ (visited on 06/01/2015).

[31] Snapdragon. (2011). Snapdragon S4 processors: system on chip solutions for a new mobile age, [Online]. Available: https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age (visited on 06/01/2015).

[32] ARM. (2015). ARM graphics, [Online]. Available: http://www.arm.com/products/multimedia/mali-graphics-hardware/ (visited on 06/01/2015).

[33] NVidia. (2013). Whitepaper: NVIDIA Tegra 4 Family GPU architecture, [Online]. Available: http://www.nvidia.com/docs/IO/116757/Tegra_4_GPU_Whitepaper_FINALv2.pdf (visited on 06/01/2015).

[34] ——, (2014). Whitepaper: NVIDIA Tegra K1: a new era in mobile computing., [Online]. Available: http://www.nvidia.com/content/pdf/tegra_white_papers/tegra_k1_whitepaper_v1.0.pdf (visited on 06/01/2015).

[35] ——, (2015). Whitepaper: NVIDIA Tegra X1: NVIDIA’s new mobile superchip., [Online]. Available: http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf (visited on 06/01/2015).

[36] ARM. (2007). ARM Mali-200 GPU, [Online]. Available: http://www.arm.com/markets/mali-200.php (visited on 06/01/2015).

[37] D. Heare and M. P. Baker, *Computer Graphics (C Version)*. New Jersey: Prentice Hall International Inc, 1998.

[38] R. Kodituwakku, K. Wijeweera, and M. Chamikara, “An efficient line clipping algorithm for 3d space,” *International Journal*, vol. 2, no. 5, 2012.

[39] J. E. Bresenham, "Algorithm for computer control of a digital plotter," *IBM Systems journal*, vol. 4, no. 1, pp. 25–30, 1965.

[40] H. P. Moreton and F. C. Crow, *Line rasterization techniques*, US Patent 8,482,567, Jul. 2013.

[41] J. E. Bresenham, "Incremental line compaction," *The Computer Journal*, vol. 25, no. 1, pp. 116–120, 1982.

[42] D. Field, "Incremental linear interpolation," *ACM Transactions on Graphics (TOG)*, vol. 4, no. 1, pp. 1–11, 1985.

[43] M. L. V. Pitteway and A. Green, "Bresenham's algorithm with run line coding shortcut," *The Computer Journal*, vol. 25, no. 1, pp. 114–115, 1982.

[44] J. Van Aken and M. Novak, "Curve-drawing algorithms for raster displays," *ACM Transactions on Graphics (TOG)*, vol. 4, no. 2, pp. 147–169, 1985.

[45] P. G. Bao and J. G. Rokne, "Quadruple-step line generation," *Computers And Graphics*, vol. 13, no. 4, pp. 461–469, 1989.

[46] P. Graham and S. S. Iyengar, "Double-and triple-step incremental linear interpolation," in *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, ACM, 1993, pp. 368–372.

[47] X. Wu and J. G. Rokne, "Double-step incremental generation of lines and circles," *Computer Vision, Graphics, and Image Processing*, vol. 37, no. 3, pp. 331–344, 1987.

[48] R. C. Taylor, D. B. Clifton, D. Gotwalt, M. A. Mang, T. A. Piazza, and J. D. Potter, *Methods and systems for rendering line and point features for display*, US Patent 6,433,790, Aug. 2002.

[49] P. R. Brown, *Arrangements for antialiasing coverage computation*, US Patent 6,985,159, Jan. 2006.

[50] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, no. 1, pp. 14–17, 1964.

[51] A. Avizienis, "Signed-digit number representations for fast parallel arithmetic," *Electronic Computers, IRE Transactions on*, no. 3, pp. 389–400, 1961.

[52] H. Srinivas and K. K. Parhi, "A fast VLSI adder architecture," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 5, pp. 761–767, 1992.

[53] A. Weinberger and J. Smith, "A one-microsecond adder using one-megacycle circuitry," *Electronic Computers, IRE Transactions on*, no. 2, pp. 65–73, 1956.

[54] M. McGuire. (2015). Mcguire graphics data, [Online]. Available: http://graphics.cs.williams.edu/data/meshes.xml (visited on 06/01/2015).

[55] M. P. G. Humphreys. (2015). Physically based rendering scene, [Online]. Available: http://www.pbrt.org/scenes.php (visited on 06/01/2015).

[56] OpenCores. (2012). Open core ORSoC graphics accelerator model, [Online]. Available: http : / / opencores . org / project , orsoc _ graphics _ accelerator (visited on 06/01/2015).

Appendix A

OpenGL ES 1.1 Commands

A.1 Data Fetch Unit Commands

1. **Void Color4{x, ub}(T red, T green, T blue, T alpha)**. It specifies the current color components (R, G, B, A) which is used as vertex associated color when the color array is disabled. The argument types, T, are fixed point x or unsigned byte ub.

2. **Void Normal3x** $(T\,n_x, T\,n_y, T\,n_z)$**.** It specifies the current normal (n_x, n_y, n_z) which is used as vertex associated normal when the normal array is disabled. The argument type, T, is fixed point only.

3. **Void MultiTexCoord4x (enum texture,** $T\,s, T\,t, T\,r, T\,q$). It specifies the current texture coordinates for the texture units. These coordinates are used as vertex associated texture coordinates for the corresponding texture unit when this texture array is disabled. The argument type, T, is fixed point only. The texture argument is a symbolic constant of $Texture0$ for texture unit 0 or $Texture1$ for texture unit 1.

4. **Void PointSizex (fixed size)**. It specifies the current point sprite *size* that is used if the point size array is disabled. The default value is 1.0.

5. **Void ClientActiveTexture (enum texture)**. It is used to select the vertex array client state parameters to be modified by the $TexCoordPointer$ command and the array affected by $EnableClientState$ and $DisableClientState$ with parameter TEXTURE_COORD_ARRAY. The texture argument is a symbolic constant of $Texture0$ for texture unit 0 or $Texture1$ for texture unit 1.

6. **Void EnableClientState/DisableClientState (enum array).** It is used to enable or disable the corresponding array as vertices or colors. The array argument is a

symbolic constant of Point_size_array_OES, Texture_Coord_array, Vertex_Array, Normal_Array, or Color_Array. If the corresponding array is disabled, the current value is used instead.

7. **Void VertexPointer (int size, enum type, sizei stride, void *pointer)**. It describes the vertex array location and organization. The argument Pointer is the address of the first element in the client memory if Vertex_Array_Buffer_Binding is 0 or offset of the first element in the graphics memory otherwise. Stride specifies the distance between the two consequences elements in unsigned bytes unit. Type is byte, short, or fixed point. The vertex coordinates are (X, Y, Z, W) and Size is 2, 3, or 4. If the size equals 2, the (Z, W) coordinates are (0, 1) implicitly and if the size equals 3, the W is 1 implicitly.

8. **Void TexCoordPointer (int size, enum type, sizei stride, void *pointer)**. It describes the texture coordinates array's location and organization. Type is byte, short, or fixed point. The texture coordinates are (u, v, r, q) and size is 2, 3, or 4. If the size equals 2, the (r, q) coordinates are (0, 1). If the size equals 3, the q is 1 implicitly.

9. **Void NormalPointer (enum type, sizei stride, void *pointer)**. It describes the normal array location and organization. Size is 3 by default.

10. **Void PointSizePointerOES (enum type, sizei stride, void *pointer)**. It describes the point size array location and organization. Size is 1 by default.

11. **Void ColorPointer (int size, enum type, sizei stride, void *pointer)**. It describes the color array's location and organization for the corresponding vertices. Argument Size is 3 only. Type is unsigned byte or fixed point only.

12. **Void BindBuffer (enum target, uint buffer)**. It Creates and binds buffer object in the graphics memory by binding buffer unused name to target Array_Buffer or Element_Array_Buffer. Also, it is used to bind existing buffer object by binding buffer used name to target Array_Buffer or Element_Array_Buffer. The argument target is Array_Buffer or Element_Array_Buffer and the argument buffer is an integer name. In the initial state the reserved name zero is bound to ARRAY BUFFER and Element_Array_Buffer. There is no buffer object corresponding to the name zero. This used to de-bind the bounded buffers.

13. **Void BufferData (enum target, sizeiptr size, const void *data, enum usage)**. It transfers the array data from the system memory to the bounded target Array_Buffer or Element_Array_Buffer in the graphics memory.

14. **Void BufferSubData (enum target, intptr offset, sizeiptr size, const void *data)**. It modifies part of the bounded target Array_Buffer or Element_Array_Buffer in the graphics memory by transferring the array data from the system memory.

15. **Void DeleteBuffers (sizei n, const uint * buffers).** It deletes n names of buffer objects.

16. **Void DrawArrays (enum mode, int first, sizei count)**. It draws primitives which are formed from the elements first through first+count-1 of each enabled array. Mode specifies the primitive type: points, line strips, line loops, lines, triangles strips, triangle fans, and separate triangle.

17. **Void DrawElements(enum mode, sizei count, enum type, void * indices)**. It draws primitives which are formed from the elements defined by indices of each enabled array. Mode specifies the primitive type: points, line strips, line loops, lines, triangles strips, triangle fans, and separate triangle.

A.2 Matrix Construction Unit Commands

1. **Void MatrixMode (enum mode).** It specifies the current matrix that will be affected by the subsequent Load, Mult, Push, and Pop commands. Mode parameter is Texture, Model-view, or Projection.

2. **Void LoadMatrixx (**$Tm[16]$**) / LoadIdentity (void).** They load the current matrix by matrix at memory location specified by 4x4 pointer m or by the identity matrix.

3. **Void MultMatrixx (**$Tm[16]$**)**. It multiplies the current matrix by matrix at memory location specified by 4x4 pointer m.

4. **Void Rotatex (**$T\Theta, Tx, Ty, Tz$**) / Translatex(**Tx, Ty, Tz**) / Scalex (**Tx, Ty, Tz**).** They load the current matrix by the rotation of v = (x, y, z)T by an angle Θ, the translation matrix (T), or the scale matrix (S).

5. **Void Frustumx (**Tl, Tr, Tb, Tt, Tn, Tf**) / Orthox (**Tl, Tr, Tb, Tt, Tn, Tf**)**. It loads the current matrix by the perspective or parallel projection matrix.

6. **Void ActiveTexture (enum texture)**. It specifies the active texture unit that will be affected by the subsequent matrix commands.

7. **Void PushMatrix (void)**. It duplicates the current matrix, the matrix at the top of the stack, in the location below it.

8. **Void PopMatrix (void)**. It replaces the current matrix, the matrix at the top of the stack, with the matrix below it.

9. **Void Viewport (int** *x***, int** *y***, sizei** *w***, sizei** *h***)**. It specifies the viewport space: it's left-lower point location (x, y), height (h), and width (w).

10. **Void DepthRangeX (clampx** *n***, clampx** *f*). It specifies the factor and offset applied to the depth (Zd) in the final viewport transformation.

A.3 Vertex Processing Unit Commands

1. **Void Enable/Disable (enum Target).** It specifies the scaling mode for the normal (nx, ny, nz). The normal target are either RESCLAE_NORMAL or NORMALIZE. 2)

2. **Void EnableClientState/DisableClientState (enum array)**. It is used to enable or disable the corresponding array as vertices or colors. If the normal, color, or the texture arrays are disabled, the corresponding current value is used. The array argument is a symbolic constant of Texture_Coord_array, or Normal_Array.

A.4 Lighting Unit Commands

1. **Void Enable/Disable (enum Target)**; Target is lighting. It enables/disables the lighting operation when the target is lighting. If the lighting is disabled, the incoming vertex color is passed to the post-clipping unit without any modification.

2. **Void Materialx (***v***) (enum face, enum pname,** *T* **param (***s***))**. It specifies the material properties for front and back face. In the OpenGL ES, the same material must be set for the front and back faces. So, the face value must be FRONT_AND_BACK. Table 4.2 summarizes the material parameters and their initial values.

3. **Void LightModelx (***v***) (enum pname,** *T* **param (***s***)).** It specifies the light model color and the normal direction that is used in the front and back face calculations. The parameter is Light_Model_Ambient (a_{LS}) or Light_Model_two_side. The front color calculation is always performed using the normal (n). If the light model

two side is false, the back color calculation is performed using the normal (-n). Otherwise, the normal (n) is used for the back color calculation.

4. **Void Lightx (v) (enum Light, enum pname, *T* param (*s*))**. It specifies the light sources parameters. The light argument is lighti for the light source $i \in [0,7]$. Table 4.1 summarizes the light sources parameters and their initial values.

5. **Void Enable/Disable (enum Target)**; Target = Color_Material. It enables/disables the color tracking. If the color tracking is enabled, the ambient color (acm) and diffuse color (dcm) properties of the front and back material are set to the incoming vertex color.

A.5 Clipping Unit Commands

1. **Void Clipplanex (enum *P*, const *T* eqn[4]).** It defines a new clip plane in the world space. The argument P is CLIP_PLANEi where i is the number of clip plane, i ϵ [0, n-1]. The eqn parameters are the clip plane equation coefficients (P1, P2, P3, P4).

2. **Void Enable/Disable (enum target);** target = CLIP_PLANEi. It enables/disables the clip plane i.

A.6 Post-Clipping Unit Commands

1. **Void ShadeModel (enum Mode)**. It specifies the shade model. The mode is either SMOOTH or FLAT. If the mode is SMOOTH, the vertices colors are passed without any modification; otherwise, colors of the primitive vertices are assigned to same color based on the drawing mode. If the primitive is a line, the flat color is the color of the second vertex. If it is a triangle, the flat color is the color of the last vertex color. Initially, the shade model is SMOOTH.

2. **Void Enable/Disable (enum target)**; target = CULL_FACE. It enables/disables the culling operation. Initially, the culling is disabled.

3. **Void CullFace (enum mode)**. It specifies the cull face. It may be Front, Back, or Front&Back. Initially, the cull face is back.

4. **Void FrontFace (enum dir)**. It specifies the front face direction. It may be CW or CCW. Initially, this direction is CCW.

A.7 Rasterization Unit Commands

1. **Void Enable/Disable (enum target)**; target = MultiSample. It enables/disables the multisampling operation for points, lines, and triangles. Initially, the multisampling is disabled.

2. **Void Enable/Disable (enum target)**; target = PointSmooth. It enables/disables the point smooth operation. Initially, point smooth is disabled.

3. **Void Enable/Disable (enum target)**; target = Point_Sprite_OES. It enables/disables the point sprite operation. Initially, the point sprite is disabled.

4. **Void PointSize (fixed size)**. It specifies the point size. Initially, the point size is 1.0

5. **Void PointParameterX (enum pname, T param(s))**; pname = {Point_Size_min, Point_Size_max, Point_Distance_Attenuation, Point_Fade_Threshold_Size}. It specifies the point parameters that are required to compute the actual width.

6. **Void TexEnvX (enum Target, enum Pname, T param)**; target = Point_Sprite_OES, pname = Coord_replace_OES, param = {false, true}. It enables/disables the texture coordinates replacement for the active texture unit. Initially, the texture coordinates replacement is disabled.

7. **Void Enable/Disable (enum target)**; target = LineSmooth. It enables/disables the line smooth operation. Initially, the line smooth is disabled.

8. **Void LineWidthX (fixed width)**. It specifies the line width. Initially, the line width is 1.0.

9. **Void Enable/Disable (enum target)**; target = Polygon_Offset_Fill. It enables/disables the polygon offset fill. Initially, polygon offset fill is disabled.

10. **Void PolygonOffsetX (fixed factor, fixed units)**. It specifies the factor and units for computing the polygon offset. Initially, the polygon factor and units are 1.0.

A.8 Texture Handling Unit

1. **Void PixelStorei (enum pname, T param)**; pname = Unpack_Alignment, param is$1, 2, 4, 8$. It specifies the alignment, in byte unit, for the start of each pixel row in the system memory. For example, if param is 1, the pixels are byte-alignment.

2. **Void TexImage2D (enum target, int level, enum internal format, sizei width, sizei height, int border, enum format, enum type, void *data)**; target = TEXTURE_2D, format and internal format = {RGB, RGBA, Alpha, Luminance, Luminance_Alpha}, type = {UNSIGNED_BYTE, UNSIGNED_SHORT_5_6_5, UNSIGNED SHORT_4_4_4_4, UNSIGNED_SHORT_5_5_5_1}. It specifies an image, in the system memory, to be copied to the bounded texture object, in the graphics memory. **Level** is the level of details of the image. **Internal format** is the transformed texture format in the graphics memory after conversion**. Width** and **height** are the image size. **Border** should be zero. **Format** and **type** specifies the image in the system memory. **Data** is an address pointer of the start of the image in the system memory.

3. **Void CopyTexImage2D (enum target, int level, enum internal format, int** x**, int** y**,sizei width, sizei height, int border)**; target = TEXTURE_2D. It specifies part of the frame buffer, in the graphics memory, to be copied to the bounded texture object, in the graphics memory. This part is defined by its lower left corner (x, y), width and height.

4. **Void TexSubImage2D (enum target, int level, int xoffset, int yoffset, sizei width, sizei height, enum format, enum type, void *data)**; target = TEXTURE_2D. It specifies an image, in the system memory, to be copied to a rectangular subregion of the bounded texture object, in the graphics memory. This rectangle subregion is defined by its lower left corner $(xoffset, yoffset)$, width, and height.

5. **Void CopyTexSubImage2D (enum target, int level, int xoffset, int yoffset, int** x**, int** y**, sizei width, sizei height)**; target = TEXTURE_2D. It updates a rectangle subregion of the bounded texture object from a rectangle part of the frame buffer, in the graphics memory. The rectangle subregion of the bounded texture object is defined by its lower left corner $(xoffset, yoffset)$, width, and height whereas the frame buffer rectangle is defined by its lower left corner (x, y), width, and height.

6. **Void BindTexture (enum target, uint texture)**; target = TEXTURE_2D, texture = unused name. It binds the input **texture** object to the **TEXTURE_2D.** This texture object is used to direct the texture mapping operation. Also, it creates a texture object if **texture** is unused name.

7. **Void DeleteTextures (sizei** n**, uint *textures).** It deletes n texture objects. ***Texture** is an address pointer to the names of texture objects to be deleted.

8. **Void ActiveTexture (enum texture)**; texture is Texture0 or Texture1. It specifies the active texture unit. Initially, the texture object 0 is binded to the two texture units.

9. **Void TexParameterxv (enum target, enum pname, *T* param)**; target = TEXTURE_2D, pname = Generate_mipmap, param = {true, false}. It enabled the automatic mipmap generation. So, if any change occurs on the texels of the zero level of a mipmap, a complete set of mipmap arrays will be computed again.

A.9 Texture Mapping Unit

1. **Void TexParameter*ix* (enum target, enum pname, T param)**; target = TEXTURE_2D, pname = {Texture_wrap_S, Texture_wrap_T, Texture_Min_Filter, Texture_Mag_Filter}. It specifies texture properties to control the filtering and wrapping functions that are applied on the texture coordinates. Table 4.7 summarizes the texture properities and their initial values.

2. **Void TexEnv*ix* (enum target, enum pname, T param)**; target = TEXTURE_ENV, pname = {Texture_env_mode, Texture_env_color, RGB_Scale, Alpha_Scale, Combine_RGB, Combine_Alpha}. It specifies the texture function parameters to control the execution of the texture function on the fragment's primary color, fragment's texture source colors, and texture environment color for the active texture units. Table 4.8 summarizes the texture function parameters and their initial values.

3. **Void ClientActiveTexture (enum Texture)**; texture = { texture0, texture1}. It specifies the active texture unit that is modified by the TexEnv command.

A.10 Final Color Adapting Unit

1. **Void Enable/Disable (enum target)**; target = Fog, MultiSample, LineSmooth, and PointSmooth. It enables/disables the fog, multisampling, line smoothing, and point smoothing. Initially, fog, line smoothing, and point smoothing are disabled whereas the multisampling is enabled.

2. **Void Fogx (enum pname, T param)**; pname = {Fog_Mode, Fog_Density, Fog_Start, Fog_End}. It specifies the fog parameters to control the blending operation. Initially, the fog mode is exp, fog density d = 1.0 , fog start s = 1.0, and fog end s = 0.0.

3. **Void Fogxv (enum pname, T params)**; pname = Fog_Color. It specifies the fog color. Initially, the fog color is (0,0,0,0).

ملخص الرسالة

لقد أصبحت وحدة معالجة الرسوميات وحدة أساسية فى كل أجهزة الأنظمة المدمجة لإستخدامها فى عملية إنشاء الصور والفيديوهات بشكل سريع. هذه العملية مهمة لأجهزة عديدة مثل الهواتف الذكية وأجهزة الألعاب اللوحية. حاليا, نحن نستخدم تلك الأجهزة لقراءة الرسائل الألكترونية وتصفح صفحات الأنترنت والتصوير وتشغيل الألعاب. مع تزايد الطلب على تطبيقات الألعاب والتطور المتسارع للأنظمة المدمجة, تستخدم حاليا وحدات معالجة الرسوميات لتنفيذ الحسابات متعددة الأغراض بالأضافة إلى حسابات الجرافيكس.

هدفنا تصميم وتنفيد وحدة معالجة رسوميات جامعة القاهرةCUGPU كأول وحدة معالجة رسوميات للأنظمة المدمجة فى مصر وفقا لمكتبة الرسوميات المفتوحة الخاصة بالأنظمة المدمجة ١.١. هذة الوحدة تدعم خط الرسوميات ثلاثية الأبعاد ذو الوظائف الثابتة. أيضا, يمكن دمج وحدة معالجة رسوميات جامعة القاهرة CUGPU مع وحدة المعالجة المركزية لجامعة القاهرة CUSPARC لتكوين نظام مدمج متكامل لأستخدامه فى الأغراض التعليمية.

فى هذة الرسالة نقوم بتقديم تصميم مقترح لبنية وحدة معالجة رسوميات جامعة القاهرة CUGPU. لقد قمنا إيضا بتصميم خوارزمية تنقيط الخط بإستخدام لغةVHDL وفقا لقوانين خروج الماسة عن طريق تعديل خوارزمية Bresenham لرسم الخط. كما قمنا أيضا بتبسيط قوانين مكتبة الرسوميات المفتوحة لأستكمال الألوان والأحداثيات لتصبح خطية تزايدية مما جعلهم متلائمين مع خوارزمية Bresenham.

إيضا, لقد قمنا بتنفيذ تصميمين لخوارزمية تنقيط الخط بإستخدام TSMC65nmLP. كانت النتائج كالتالي: حقق التصميم الأول تردد 270 MHZ ومساحة $0.088mm^2$ بينما حقق التصميم التانى تردد 200 MHZ ومساحة $0.052mm^2$.

مهندس: احمد ابراهيم سمير خليل
تاريخ الميلاد: ١٩٨٩/٠٧/١٨
الجنسية: مصري
تاريخ التسجيل: ٢٠١١/١٠/٠١
تاريخ المنح: ..../..../....
الدرجة: ماجستير العلوم
القسم: هندسة الإلكترونيات والإتصالات الكهربية

المشرفون:

أ.د. سراج الدين السيد حبيب
أ.د. حسام علي فهمي

الممتحنون:

أ.د. سراج الدين السيد حبيب	(المشرف الرئيسي)
أ.د. حسام علي فهمي	(مشرف)
أ.م.د. عمرو جلال الدين وصال	(الممتحن الداخلي)
أ.د. حسين إسماعيل شاهين	(الممتحن الخارجي) (كلية الهندسة – جامعة عين شمس)

عنوان الرسالة:

بنية مقترحة لوحدة معالجة الرسوميات وفقا لمكتبة الرسوميات المفتوحة وتنفيذ خوارزمية تنقيط الخط

الكلمات الدالة:
وحدة معالجة الرسوميات، مكتبة الرسوميات المفتوحة، خوارزمية تنقيط الخط، رسم الخط، الإستكمال الخطى التزايدى

ملخص الرسالة:
لقد أصبحت وحدة معالجة الرسوميات وحدة أساسية فى كل أجهزة الأنظمة المدمجة. في هذه الرسالة نقوم بتقديم بنية مقترحة لوحدة معالجة رسوميات جامعة القاهرة CUGPU وفقا لمكتبة الرسوميات المفتوحة للأنظمة المدمجة ١.١. نحن نقدم إيضا تصميمين لوحدة تنقيط الخط بإستخدام لغة VHDL. تم تنفيذ هذه الوحدة بإستخدام TSMC 65nm LP وكانت النتائج كالتالي: حقق التصميم الأول تردد $270MHz$ ومساحة $0.088mm^2$ بينما حقق التصميم الثانى تردد $200MHz$ ومساحة $0.052mm^2$

بنية مقترحة لوحدة معالجة الرسوميات وفقا لمكتبة الرسوميات المفتوحة وتنفيذ خوارزمية تنقيط الخط

إعداد

احمد ابراهيم سمير خليل

رسالة مقدمة إلي
كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول علي درجة
ماجستير العلوم
في
هندسة الإلكترونيات والإتصالات الكهربية

يعتمد من لجنة الممتحنين:

أ.د. سراج الدين السيد حبيب – المشرف الرئيسي

أ.د. حسام علي فهمي – مشرف

أ.م.د. عمرو جلال الدين وصال – الممتحن الداخلي

أ.د. حسين إسماعيل شاهين – الممتحن الخارجي
(كلية الهندسة – جامعة عين شمس)

كلية الهندسة – جامعة القاهرة
الجيزة – جمهورية مصر العربية
٢٠١٥

بنية مقترحة لوحدة معالجة الرسوميات وفقا لمكتبة الرسوميات المفتوحة وتنفيذ خوارزمية تنقيط الخط

إعداد

احمد ابراهيم سمير خليل

رسالة مقدمة إلي
كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول علي درجة
ماجستير العلوم
في
هندسة الإلكترونيات والإتصالات الكهربية

تحت إشراف

أ.د. سراج الدين السيد حبيب
أستاذ
قسم هندسة الإلكترونيات والإتصالات الكهربية
كلية الهندسة – جامعة القاهرة

أ.د. حسام علي فهمي
أستاذ
قسم هندسة الإلكترونيات والإتصالات الكهربية
كلية الهندسة – جامعة القاهرة

كلية الهندسة – جامعة القاهرة
الجيزة – جمهورية مصر العربية
٢٠١٥

بنية مقترحة لوحدة معالجة الرسوميات وفقا لمكتبة الرسوميات المفتوحة وتنفيذ خوارزمية تنقيط الخط

إعداد

احمد ابراهيم سمير خليل

رسالة مقدمة إلي
كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول علي درجة
ماجستير العلوم
في
هندسة الإلكترونيات والإتصالات الكهربية

كلية الهندسة – جامعة القاهرة
الجيزة – جمهورية مصر العربية
٢٠١٥