國 立 成 功 大 學

電腦與通信工程研究所

碩 士 論 文

基於砌塊式繪圖架構之三維繪圖著色引擎的設計、分析與實現

Design, Analysis, and Implementation of a Rasterization Engine based on Tile-Based Rendering Architecture in 3D Graphics

研 究 生: 劉哲宇　　Student: Jhe-Yu Liou

指導教授: 陳中和　　Advisor: Chung-Ho Chen

Institute of Computer and Communication Engineering

National Cheng Kung University

Thesis for Master of Science

July 2009

中 華 民 國 九 十 八 年 七 月

Design, Analysis, and Implementation of a Rasterization Engine based on Tile-Based Rendering Architecture in 3D Graphics

by Jhe-Yu Liou

A Thesis Submitted to the Graduate Division in Partial Fulfillment of the Requirement for the Degree of Master of Science

at

National Cheng Kung University
Tainan, Taiwan, Republic of China
July 2009

Approved by:

	Yeong-Kang Lai
Chin-Yu Yang	

Advisor(s):

C. W. Chen	

Director: Pau-Choo Chung

[redacted]

國 立 成 功 大 學
碩 士 論 文

基於砌塊式繪圖架構之三維繪圖著色引擎的設計、分析與實現

研究生：劉哲宇

本論文業經審查及口試合格特此證明

論文考試委員：

陳中和	
黃悅夫	賴永康
楊佳玲	蔣文欽

指導教授：陳中和

所　　長：詹寶珠

中 華 民 國 九 十 八 年 七 月

基於砌塊式繪圖架構之三維繪圖著色引擎的設計、分析與實現

學生：劉哲宇　　指導教授：陳中和

國立成功大學電腦與通信工程研究所碩士班

摘要

3D 繪圖系統已經在桌上型電腦平台發展一段時日，所顯示出來的 3D 效果也相當的驚人。但由於需要大面積矽晶片以放置大量的運算元件，且擁有高溫、耗電等特性，使其無法被順利的整合在攜帶型電子產品之中。然而，隨著嵌入式系統的快速發展、硬體製程的進步、和消費性及攜帶性產品上對於 3D 繪圖的應用需求的大量增加，如何在這樣的產品中設計一個低成本的 3D 繪圖加速硬體已成為一項重要的課題。

一般的 3D 繪圖硬體中，依處理階段的不同，可分為前半部的 Geometry engine 跟後半部的 Rasterization engine。本論文根據砌塊式繪圖架構為基礎，設計出一高效率之 Rasterization engine。此引擎擁有以下幾點特點：使用 tile-boundary skip traversal 進行三角形掃描、使用 barycentric coordinate 轉換三角形座標進行內插、使用 multi Z test 和 6D block texture cache 等功能。這些特點使得 RE 硬體可以在維持一定繪圖品質的前提下，同時減少硬體設計面積跟提高硬體效能。本論文最後所完成之合成後 RTL 模型，可在 QEMU 全系統驗證平台下，以時脈 200MHz 和 OpenGL ES 應用程式軟體、Linux 作業系統、geometry engine 進行軟硬體協同模擬驗證，成功完成一可適用於嵌入式系統之 3D 繪圖引擎。

關鍵字:電腦圖學(繪圖), Tile-Based 繪圖管線, Rasterization 引擎

Design, Analysis, and Implementation of a Rasterization Engine based on Tile-Based Rendering Architecture in 3D Graphics

Student: Jhe-Yu Liou Advisor: Chung-Ho Chen

Institute of Computer and Communication Engineering,
National Cheng Kung University,
Tainan, Taiwan, R.O.C.

Abstract

3D graphic rendering system for desktop has been developed for a long time and has the capability to show amazing 3D effect. However, this system is hard to be integrated with mobile electronic products because of its large-area requirement, high temperature, and high-power consumption. Due to the rapid development of embedded system and hardware technology, and increasing demand of 3D graphic applications for consumer electronic, how to design a low-cost 3D graphic accelerator has become an important issue.

A typical 3D graphic accelerator can be divided into a geometry engine and a rasterization engine by the different processing stages. In this thesis, a highly efficient rasterization engine based on tile-based architecture is proposed. This engine has incorporated the following architecture techniques: tile-boundary skip traversal, barycentric coordinate, multi Z test, and 6D block texture cache. With these features, we reduce the area cost and improve the performance of the rasterization engine we design. The rasterization engine when synthesized with TSMC 0.18um technology can run up to 200MHz. We also verify the entire 3D graphic accelerator net-list with OpenGL ES application in Linux OS under a full system simulation platform constructed with CoWare and QEMU.

Keywords：Computer graphics, Tile-based rendering pipeline, Rasterization engine

誌謝

首先我要改謝我的指導老師陳中和教授，感謝老師在這兩年來所做的指導，使我可以順利的完成研究所碩士班的學業。也要感謝奇景光電的王經理，在這兩年對我們所做的督促，是催生這篇論文的主要原因。

另外要感謝實驗室和我一起研究的蔡順帆同學和沈協增同學，和我一起完成整個系統，一起討論研究。要是沒有你們和我一起通力合作，就沒有這些研究成果。

我還要感謝實驗室的其他學長、同學和學弟，給予我一個和樂融融的研究環境。其中，我要特別感謝廖培鈞和郭良宇兩位同學和 Klio 學長，適時的在規律的研究生活中添加歡樂色彩。謝謝你們。

目錄

圖目錄

表目錄

第1章 Introduction

1.1 Motivation

在手機或攜帶型的電子產品上面提供炫麗的 3D 畫面，可以讓使用者更快速的了解系統所提供的資訊，甚至是當出門在外的閒暇時刻，可以以 3D 畫面進行一些娛樂，這些事情，隨著嵌入式系統(embedded system)的快速發展，都變得可能。甚至有廠商，在手機上面，提供裸眼立體顯示介面(stereoscopic 3D display)，搭配上 3D 電腦圖像，就可以讓人在手機上體驗真實的 3D 影像。以上這些功能皆需要繪製 3D 電腦圖像，如果是由軟體去完成，也就是交由 CPU 去實現，在速度上將會相當的緩慢而難以應用。因此，開發一個可以提供快速繪製 3D 影像的 3D 加速硬體，且必須滿足嵌入式系統中省電、輕巧的要求，就變得十分重要。

之前在我們實驗室中，就有學長去研究Stereoscopic 3D display的技術[1]。在研究的同時，也有用軟體去模擬了一套繪圖管線。不過由於學長研究的主要目標不是在繪圖管線上的技術研究，所以是使用一套稱為Vincent [2]的open source軟體演算法來進行 3D圖像的繪製，並且對其中的演算法也沒有多加探討。而我們研究的主題，就是去研究用於三維繪圖中各式的演算法，並提出我們自己的繪圖管線，最後將它們實現成硬體。

在繪圖管線中，基本上就可以分為前級跟後級，我主要是負責研究在繪圖管線後級的運算。所以跟像素生成等相關演算法還有硬體架構，就是本論文主要探討的領域。

1.2 Contribution

- 根據砌塊式架構(tile-based architecture)設計出一具有高效率之後端繪圖管線硬體，此硬體相容於 OpenGL ES，適合用於嵌入式系統做為繪圖硬體。
- 使用質心座標內插法(barycentric coordinate interpolation)進行內插，可避免進行低精確度運算時造成的內插溢位錯誤。

- 採用 multi Z test 與 6D block texture cache 架構，搭配砌塊式繪圖架構，可以省下約 95%的外部材質記憶體存取。
- 支援 FSAA 全景反鋸齒技術，可大幅增加圖像的細緻程度。
- 依此硬體架構下所設計合成後的 RTL 模型在全系統模擬驗證平台下成功的與應用程式、作業系統、和前端繪圖管線(Geometry engine)進行模擬，同時在時脈為 200MHz 時，最大輸出可達每秒 200M 像素。

1.3 Organization

本論文分為六個章節，以下為各章節內容

- 第一章：序論
- 第二章：基本繪圖管線的動作介紹，及目前用於嵌入式系統中繪圖晶片的相關研究
- 第三章：本論文中用於各子模組的演算法採用理由、遇到之困難，與解決方案之介紹
- 第四章：本論文中整體硬體架構上之探討與改進
- 第五章：RTL 之實現與內部硬體元件分部的細節
- 第六章：模擬驗證環境的介紹與硬體模擬之結果
- 第七章：結論與未來展望

第2章 Background and related work

在說明我們提出的架構之前，有必要先了解相關的背景知識及我們已做的相關研究。這一章基本上就分為兩大部分，前幾節將會介紹跟 3D 繪圖相關的背景知識，主要是說明電腦如何將三維空間中的場景畫成二維的圖像。之後則是說明我到目前為止已作的相關研究。

2.1 3D Computer Graphic

一般我們再說到 3D(3-Dimension)這個詞的時候，指的是我們用長、寬、高三維座標來描述空間中的所有物件的位址和大小。然而，由於顯示科技限制的關係，我們只能在平面顯示器上面看到 2D 的圖像，並無法直接顯示 3D 的影像。而 3D 電腦圖像，基本上就是將原本以三維座標儲存在電腦中的物件資料，經過一連串的處理，轉化成 2D 的圖像顯示在顯示器上面，並讓我們看起來有立體的感覺。

建立一張 3D 電腦圖像可以分成三個步驟：建模、移動擺放跟繪製。

- 建模(modeling)

  一張 3D電腦圖像最基本的就是需要有模型以供繪製。目前最常見的方法就是以多邊形建模(polygon modeling)。這種建模法的優點，是在電腦系統中，可以很快速的被建立並運算繪製，缺點則是無法很精確的模擬帶有彎曲形狀的物體，除非使用大量的多邊形去近似，如圖 2.1。在多邊形建模時，基本的多邊形單位是三角形，所以我們可以將多邊形建模想像成是以一大堆三角形拼裝而成。

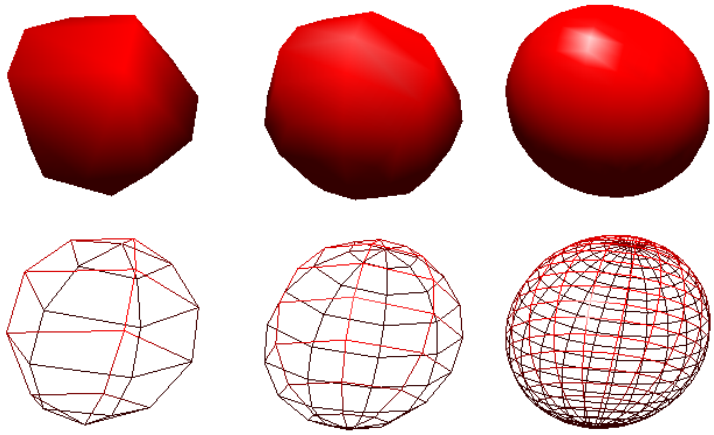


圖 2.1 細節層次不同的球模型

- 移動擺放(layout and animation)

  一旦我們有了 3D 模型之後，我們必須將它擺到我們想要的場景(scene)之中，並定義物件在空間中的相對關係。如果是動畫的話，還必須加上物件在空間中移動向量與時間的函數。

- 繪製(rendering)

  當以上兩個步驟完成之後，就要將定義好的場景輸出成 2D 圖像。這邊基本上有兩件事必需完成。第一個就是打光(lighting)，決定整個場景中各個物件跟光線互動後的顏色。第二個就是投影(projection)，將整個場景投影到 2D 的影像上，完成一張 3D 電腦圖像。通常，圖形設計者在這個階段會使用一些特定的圖像應用程式介面幫助他們繪圖，這樣設計者就只需要專注在場景或模型的建構上，而不用知道電腦繪圖的流程。這在下一節會提到。

2.2 Graphic application programming interface

目前有許多的軟體函式庫(software library)可以應用在開發 3D 圖形上面。通常一個函式庫擁有許多函式來表現幾何的特性並把這些特性給描繪出來。對於集合了這個函式庫裡面所有的函式而言，我們定義它們為圖形應用程式介面(graphic Application Programming Interface、API)。

OpenGL[3][4]和Direct3D [5]是目前兩套最熱門且用於即時繪製(real-time rendering)的電腦圖形API ，大部分的應用程式和遊戲都以這兩套 API 來進行開發。這樣大部分的 3D顯示卡就可以對它們進行硬體加速。利用純軟體來執行這些 API 也是可行的，但是，只有在有硬體加速的情況下才能對複雜的場景進行即時的繪製。總體而言，一個圖形 API 通常都是將一部分交給軟體來做，一部分交給硬體來做。從一開始應用程式將圖形基本元件產生出來，到最後把這些圖形基本元件以像素(pixel)顯示在顯示器上面，我們稱這一整個系統為一完整的繪圖管線(rendering pipeline)。

跟 Direct3D 比起來 OpenGL 是一個跨平台 3D 圖形函式庫，它提供一個跨平台且互動的環境來開發 2D 或 3D 圖形應用程式。OpenGL 是一個開放式且跨平台的標準圖形函式庫，對於研究而言，這樣開放的特性對我們是非常重要的。

2.2.1 OpenGL API

OpenGL 嚴格定義起來，是一套用來跟圖形硬體進行溝通的軟體介面。一開始，它是由 SGI(Silicon Graphics, Inc. 一間被公認在電腦圖形和動畫中具有領先地位的公司，最近已被其他公司併購)使用嚴謹的演算法則來發展並加以最佳化後，所得出來的產品。就像其他公司會不斷的發行更高效率的產品，並將產品原有的技術特性保留下來一樣，OpenGL 在經過長時間的演化過後，目前最新版本在 2009.3.24 發行，其版本號為 3.1。這是 OpenGL 自原始 1.0 版本發行以來，第 9 次的改版。OpenGL 的每一個新版本都可以向下相容，這意味著任何可以在 OpenGL 2.0、1.5、1.4、1.3、1.2、1.1 或 1.0 執行的程式，也可以在 OpenGL 3.1 下面執行而毋須做任何的改變。

OpenGL 並不是去描述物件本身有多複雜，而是提供一種機制去描述如何將一個複雜的物件畫出來。也就是說，OpenGL 屬於一種程序性的圖形 API，而不是描述性的。程式設計師按照一定的步驟寫程式並達成某種效果，而不是去描述一個場景是怎樣擺放，或者是一個物件是甚麼形狀。這些步驟跟 OpenGL 的指令息息相關，而這些指令是用來對三維空間中像是點、線或多邊形之類的基本元件進行

繪製。

2.2.2 OpenGL ES

OpenGL ES [6][7]是完整OpenGL的一個子集合，其中ES是Embedded System的縮寫，如同它字面上所示，OpenGL ES是一個專為嵌入式 3D圖形加速所使用的API，這包括了小型電腦、手機、小型裝置、車輛等。OpenGL ES 1.1 源自於桌上型OpenGL 1.5。它包含了桌上型OpenGL裡面常用、有效率、且必須的功能，另外把一些對於嵌入式系統來說，比較舊的、多餘的、累贅或不常使用的功能給移除掉。就像原本的OpenGL一樣，OpenGL ES也具有跟OpenGL ES 1.0 向下相容的能力。OpenGL ES包括常用的浮點數（floating point）格式系統和一個較為小型的定點數（fixed point）格式系統。

2.3 Conventional OpenGL pipeline

OpenGL 的規格定義了一個完整的繪圖管線，從頂點(vertex)資訊的輸入，直到將描繪好的圖像儲存到 frame buffer 中，對每一管線階段詳細定義了執行的指令，和相對應的狀態。了解一個 OpenGL 繪圖管線的動作流程，將有助於了解三維電腦繪圖如何將三度空間的物體顯現在二度空間平面的螢幕上。本節將簡單的介紹 OpenGL 繪圖管線的運算流程。

圖 2.2簡單的示意出傳統OpenGL繪圖管線 (rendering)的流程。需要注意的是，此圖的繪製流程只是說明一種概念，說明軟體API到繪製出 3D電腦圖像所需要經過的步驟。實際的硬體開發商並不一定需要使用這樣的流程順序開發硬體。

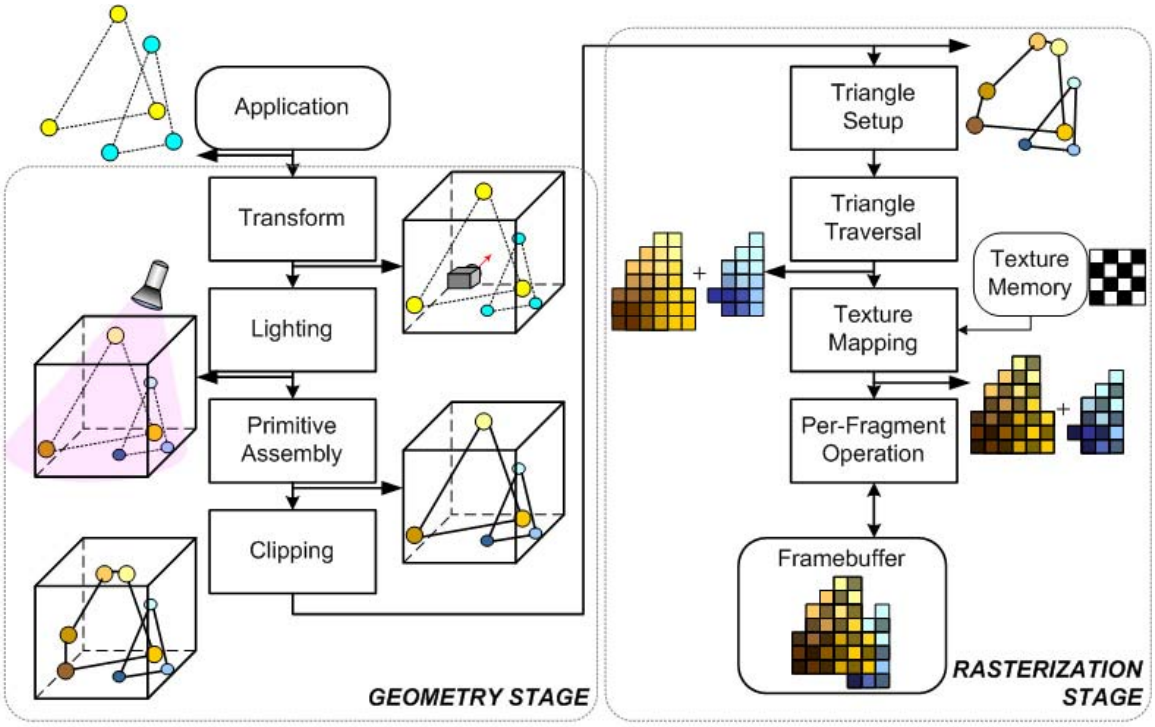


圖 2.2 傳統 OpenGL 繪圖管線的流程

在圖 2.2中，我們可以看到整個流程被切成了兩大階段，分別是Geometry Stage跟Rasterization Stage。Geometry Stage所處理的目標主要是三角形的頂點(vertex)，處理完三個頂點後再重新組合成一個三角形，接著送往下一個流程。而Rasterization Stage則是負責將Geometry Stage處理好了三角形拆成一個一個的fragment，並加以處理上色，這些處理完的fragment之後將會決定螢幕相對位置上像素(Pixel)的顏色。

由於本論文主要探討的部分為後端的繪圖硬體實作，也就是 Rasterization Stage 所負責的項目，所以這邊也會比較詳細的介紹此階段的流程。至於 Geometry Stage 將只會介紹比較重要的部分。以下將先介紹幾何 Geometry Stage 的流程。

2.3.1 Geometry Stage

再開始說明幾何處理階段之前，我們必須要先知道，一個三角形中，幾本上三個頂點之中，都各自帶有 X、Y、Z 三維空間中的座標值，同時可能還帶有顏色、貼圖座標等資料。這邊我們就假設頂點中都帶有這些資料，才能為接下來的運算

做說明。

Geometry stage 一般來講，又被稱為 T&L engine，也就是 transform and lighting engine。由此可知這兩件事對於 geometry stage 是最重要的。以下就簡單介紹這兩級所負責的動作流程。

■ Transform

所有 3D物件在場景中的平移、縮放、旋轉等動作，皆在此時完成。主要的原理，就是用線性代數中的矩陣轉換來達成。另外，除了上述動作外，此階段還有一個最重要的事情，就是做投影轉換(projection transformation)。由於人眼視野的形狀是類似於從一點往外擴散的弧狀展開，所以當我們在空間中擺放我們的物件之後，進行投影轉換時，必須要使用如圖 2.3所示的透視投影(perspective projection)，這樣才能欺騙人眼，使它在一個平面上看到的影像有立體的感覺。

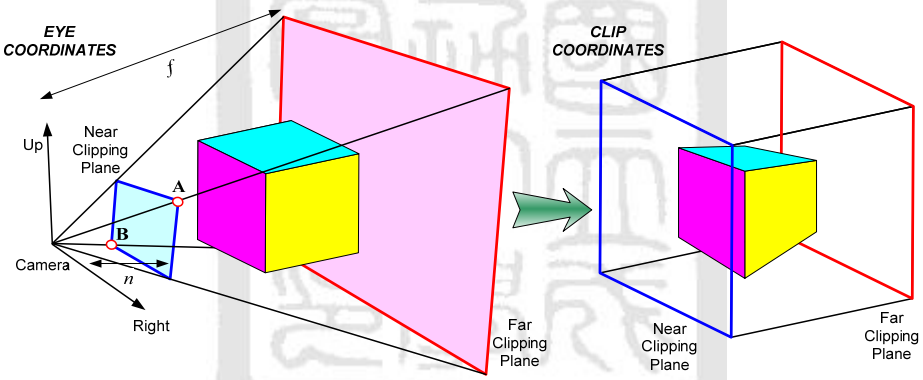


圖 2.3 Perspective projection 前後物件形狀的改變

■ Lighting

為了對 3D的物體有更真實的表現，在場景中通常會設置光源。而這些光源就會對物件的顏色產生影響。離光源越近理論上就要越亮，越遠就會越暗；面像光源的一面會比較亮，背像光源的面則會比較暗。另外，隨著物體表面對於光線的吸收或反射程度的不同，受光照所顯現的特性也會不同。這些改變物件表面顏色的計算就會在這個時候完成。如圖 2.4，多加入了一種稱為投射光的光線模型後，對場景中物件所產生的影響。

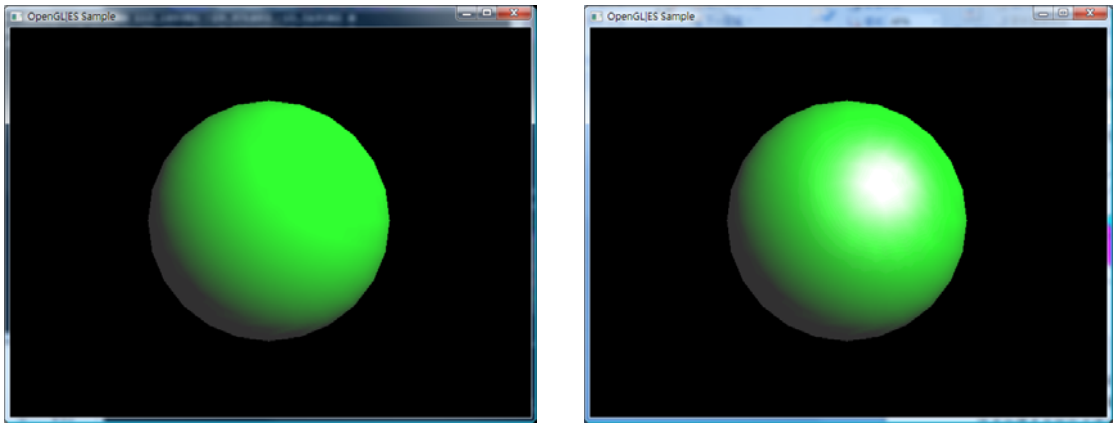


圖 2.4 使用投射光模型的差異

2.3.2 Rasterization Stage

在 Geometry Stage 處理完的三角形，由於已經做完一系列的空間轉換，所以我們可以將它們視為是在 2D 平面上的三角形。原本三角形三個頂點中各有 X、Y、Z 的座標值，經轉換後，我們可以只使用 X、Y 當作 2D 平面上的定位用座標。Z 值，也就是深度值，則成為一個只是用來判斷前後用的資料。

■ Triangle Setup and Traversal

從 Geometry Stage 處理完的三角形，接著就要把它們拆成一個一個的 fragment，以對應螢幕上相對位置的 pixel，如圖。這邊所說的 fragment，其實是跟螢幕上的 pixel 是相似的意義，都是指一個小點。不同的是 fragment 是指還在處理階段，不確定會顯示出來的點。而 pixel 就是指螢幕上的一個點，也可以指一定會顯示到螢幕上的一點。

簡單的講，此階段就是找出這個三角形將會在螢幕上占有多少個 pixel。一個最基本的找法，就是先找出一個能夠將三角形完全框住的最小方形區域，稱為邊界區域(Boundary Box)，然後逐一掃描將此邊界區域中的所有 fragment，測試它們是否有在三角形裡面。

測試完之後，我們就可以得到所有在三角形裡面的 fragment，這些 fragment 除了擁有自己的座標外並沒有任何資料。接著我們就要根據這些 fragment 在三角

形中的相對位置，並根據三角形三個頂點的資料，內插出像素的顏色等相關資料，以供接下來的運算使用。

■ Texture mapping

真實世界中，很多的物體都帶有大量的凹凸不平的表面細節，或者是一個平面上有著無法用數學函式加以描述、帶有豐富色彩的圖案。這些如果都要用多邊形建模來表示的話，將會花費極為可觀且大量的三角形。為了減少三角形的數量，我們可以用真實的圖片貼在一個物體的表面。也就是說，將預先取得的真實表面圖片貼在物體的表面上，這樣一來不僅不需要大量的多邊形計算，還可以有效地表現出真實圖片的效果。

一個物體可以有各式各樣的表面，例如顆粒、布面、磚面或是任何一種的型式皆可。這種將圖片貼到多邊形的方式稱為材質貼圖。而這張要貼上的影像我們稱為 texture，texture 中的每一個 pixel 我們稱之為 texel。圖 2.5右邊顯示了一個經過材質貼圖過後的立方體。

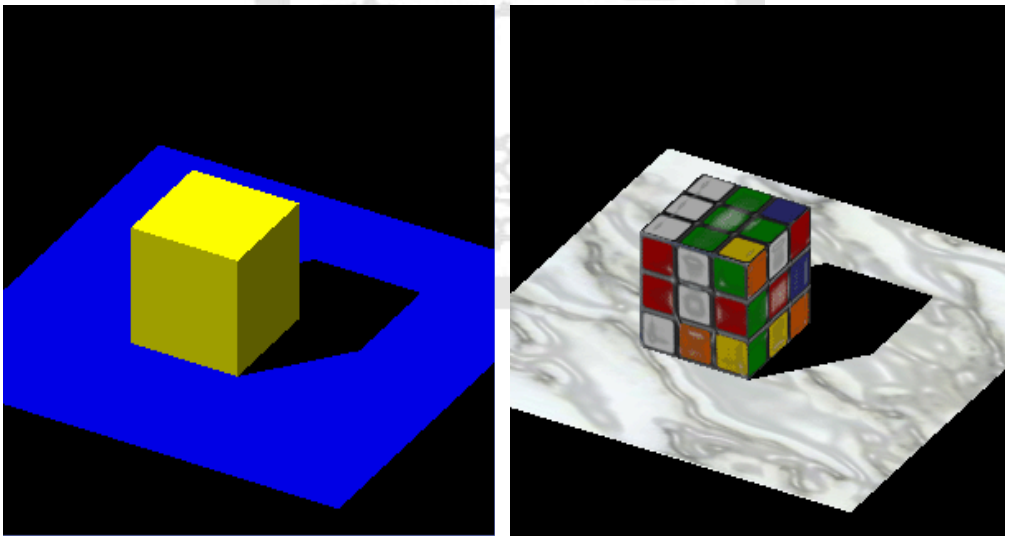


圖 2.5 比較有無材質貼圖的差異

一般來說，當一個 texture 需要被貼在物件表面上時，在 texture map 中的 texels 和螢幕上的 pixels 不太可能做一對一的對應，只有當幾何物件的大小完全等於 texture 原本的影像尺寸才會發生。當無法一對一的對應時，就要用 texture filter 來

決定這個 pixel 要跟哪一個 texel 作對應。這邊介紹兩種基本的 texture filter，nearest filter 和 linear filter。

Nearest filter是最簡單且快速的運算方法，就是在重新取樣的時候，直接取距離最接近的取樣點。這裡的取樣點也等於texel。把texel的顏色作為fragment所要的材質顏色。這個方法也稱為nearest sampling或point sampling。因為只抓一個取樣點，所以當影像被放大實，所呈現的影像品質通常都不太好，而且還會產生失真問題，我們可以看到圖 2.6中左邊的例子就是nearest filter。

Linear filter即為線性內插(linear interpolation)，Linear filter又有分為數種方法，這邊我們只介紹bilinear filter。Bilinear filter一般來說，是使用包含取樣點在內，相鄰的 2x2 的texels做混色，來取代只使用取樣點，也就是Nearest linear的做法。這種做法將可以使圖形變得更為平順。我們可以從圖 2.6的右邊看到使用linear filter的效果。

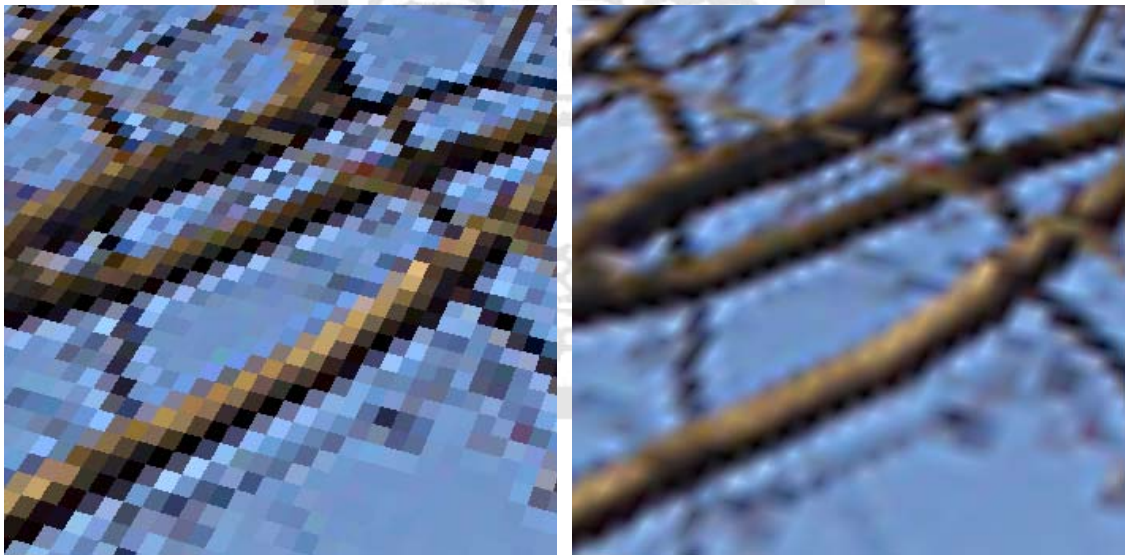


圖 2.6 比較 linear and nearest filter 的差異，左圖採用 linear filter，右圖則是 nearest filter

上面提到的貼圖方式都是直接將要貼上的圖像蓋在目標物件的表面，那如果是想要將貼圖跟物件原本的顏色作混色處理的話，就可以使用 texture blending 來達成。在這裡 OpenGL ES 有定義了一些操作，像是 replace、modulate 和 add 等。因此，程式設計人員可以依照所需要的效果來選擇其中某一個操作。

■ Frame buffer

由於等一下要介紹的 Per-fragment operation 中有很多動作，都要來回讀取寫入 frame buffer。所以這邊先說明何謂 frame buffer。

Frame buffer 是一塊存放所有即將顯示到螢幕上的像素資料的地方，其中分為 color buffer 跟 depth buffer。Color buffer 儲存所有像素的顏色，也就是我們在螢幕上所看的圖像就是根據 color buffer 顯示出來。在色彩三原色中，需要紅(R)、綠(G)、藍(B)來調和成各式各樣的顏色，故 color buffer 就會儲存這三個原色表現強弱的資訊。通常每一種原色會被賦予 8 個 bit 來進行表示。也就是單一原色的範圍是一從 0~255 的整數。另外我們在 color buffer 中還會儲存像素的透明度(Alpha)，也是用 8 個 bit 來表示。所以一個完整包含 RGBA 色彩資訊的像素，共需要 32 個 bit 來儲存其顏色，而 color buffer 就是用來存放這些資訊。

Depth buffer則是用來存放像素深度的資訊。通常一個像素的深度會使用 16 個 bit或是 32 個bit來儲存。將這些數字轉成單色顏色資訊的話，就會出現像圖 2.7右邊的depth image的樣子。比較亮的地方比較近，比較暗的地方則比較遠。

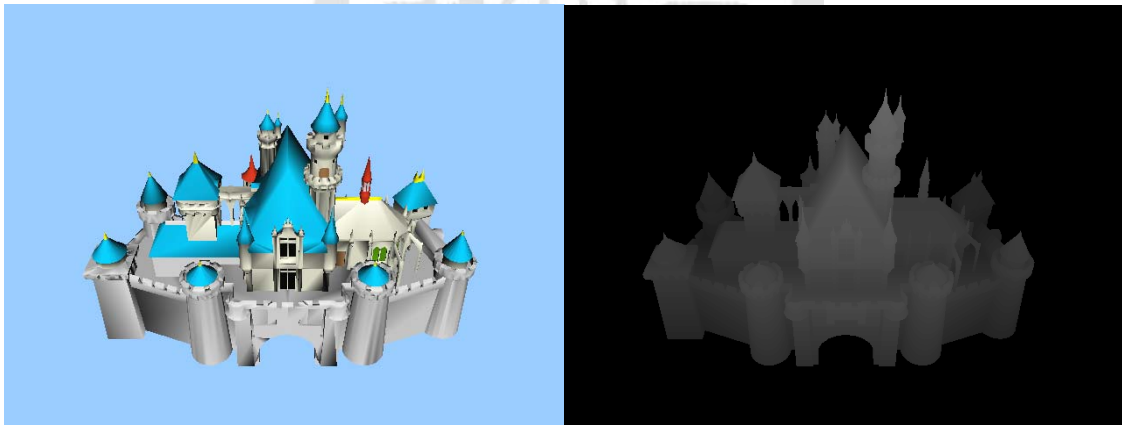


圖 2.7 Color image 與 Depth image

■ Per-fragment operation

Per-fragment 的操作在 OpenGL ES 繪畫管線中是最後層級的。在這階段裡，每一個輸入的 fragment 都須經過一連串的測試，依據這些測試出來的結果，那些存放在 frame buffer 裡的 pixel 有可能需要跟輸入的 fragment 進行特殊運算，甚至是

被完全取代。OpenGL 應用程式必須運用大量的參數和狀態去控制這些測試。Per-fragment operation 目前最主要的三個測試分別是透明度測試(alpha test)，深度測試(depth test 或 Z test) 和透明混色運算(alpha blending)。

透明度測試是指 OpenGL 會去參考一個由使用者輸入的對照 alpha 值，然後去比較輸入 fragment 的 alpha 值。簡單的說，就是透明度測試會拋棄那些比對照值還要透明的值。

深度測試是 OpenGL 用於比較 fragment 前後深淺順序的機制。理論上，當一個通過繪圖管線的 fragment 覆蓋到一已儲存在 frame buffer 裡的 pixel 時，前者的深度值必須小於後者的深度值，才會被保留。所以在 depth 測試的時候，比較靠近攝影機的 fragment 會更新它目前所在的 depth buffer 和 color buffer 的位置值，否則這個 fragment 根本不會去影響它的 color buffer 和 depth buffer 內的值。不過，這只是最常用的一種功能。OpenGL 有定義其他的 depth mode，如果使用者只想要畫最後面的物件的話，也有其對應的模式可以使用。

半透明混色運算指的是將進來的fragment顏色值和已存在frame buffer裡的顏色值作混合，而這個顏色值是依照輸入的fragment的透明值(alpha value)和存在於frame buffer裡對應pixel的透明值，去跟這兩者的顏色做特殊運算後的加總值。最常用在半透明效果之中，如圖 2.8。

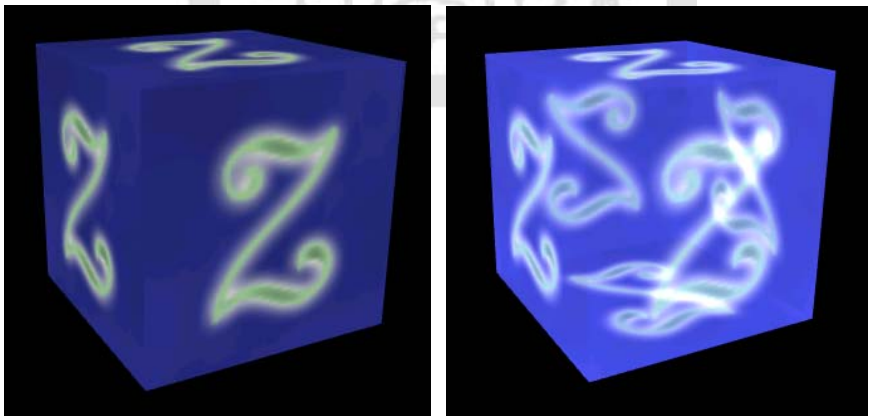


圖 2.8 使用 alpha blending 所達成的半透明混色效果。

2.4 Tile-Based rendering pipeline

一個完成的硬體繪圖管線如果依照圖 2.2來設計，就可以稱為傳統的繪圖管線，又稱為immediate mode rendering。在這之外，還有另外一個方法叫做tile-based rendering。Tile-Based繪圖管線會被提出來，完全是針對行動式平台有限的記憶體頻寬，同時可以達到省電的目的。

此架構中，螢幕上會被劃分為許多矩形的區塊（也就是tile），tile-based模組會檢視每個三角形的頂點座標，只要三角形有覆蓋到某一塊區塊，就會將此三角型分類到該區塊之中，所以有可能同一個三角形被分類到很多不同的區塊中。當所有的三角形都分類完畢之後，才啟動後端的rasterizer管線。從第一塊區塊開始，將分類至該區塊內的所有三角形繪製出來、寫回frame buffer，然後就按照同樣的步驟繼續進行下一個區塊的繪製，直到所有的區塊都繪製完畢。如圖 2.9所示。

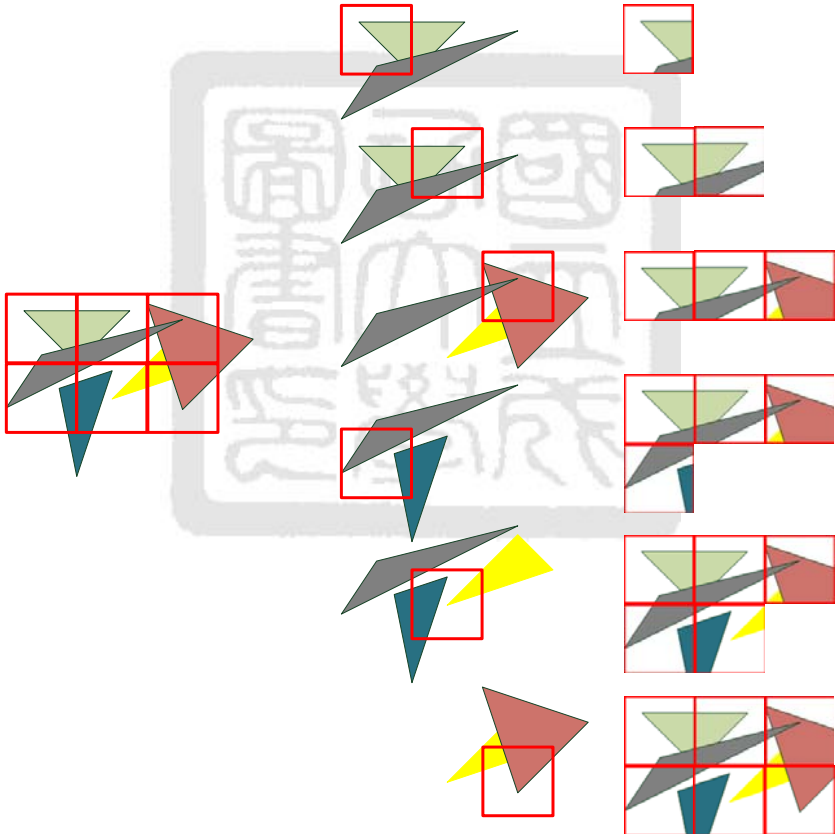


圖 2.9 Tile-based rendering 中六個區塊依序在螢幕上成像

這種方式的一個好處是，只要 tile 的尺寸分得足夠細小，所有的 Color buffer 和 Z-buffer 資料就可以完全置入到顯示晶片的內部晶片之中，免除了長期且連續對

外部 memory 中的 frame buffer 進行讀取、寫入的操作，只需在 tile 被處理完畢之後將 tile buffer 裡的資料一次寫回。

2.5 Related work

OpenGL 規格書是我們最早所參考的指標，此規格書中定義的所有 OpenGL 中的指令和這些指令要做甚麼事情。此外，它還提到了一個完成這些指令應該要有的基本流程。這對於我們來說是相當重要的，因為它提供了我們一個基本遵循的方向。前面所提到的背景介紹除了 2.5 節之外，大部分皆奠基於這份文件。

針對OpenGL這樣管線設計的硬體架構相當的多，幾乎所有出現在市場上的 3D 繪圖加速晶片接支援OpenGL，不過這些泰半都是用於桌上型平台的晶片。這些晶片設計的共通點大部分皆為使用大量的平行管線進行著色運算，而後端繪圖管線這邊就會出現第一個瓶頸點，就是記憶體存取量極為龐大。桌上型用的繪圖處理器由於比較沒有面積跟耗電的限制，故在此處就會使用大量的暫存單元來降低系統記憶體的利用率。也有在晶片的周圍自行配置專用記憶體來取代讀取外部記憶體的設計，甚至會有專用記憶體比主記憶體還大的情況出現。但這樣的設計在嵌入式系統中則比較少見。而在嵌入式系統中用來降低不必要記憶體存取量的方案，最具代表性的即為tile-based architecture [8][9]。

2.5.1 Tile-based architecture

由UNC所提出的Pixel-Plane 5 [10]，是第一個實做出的tile-based繪圖硬體。不過第一個真正出現在市場上使用Tile-based架構的則為Imagination Technologies的KYRO架構[11]，此架構應用在它們的KYRO II 3D顯示加速卡上面，於 2001 年上市。

此 3D加速引擎只有設計rasterization管線，幾何運算部分則交由CPU負責，架構如圖 2.10所示。

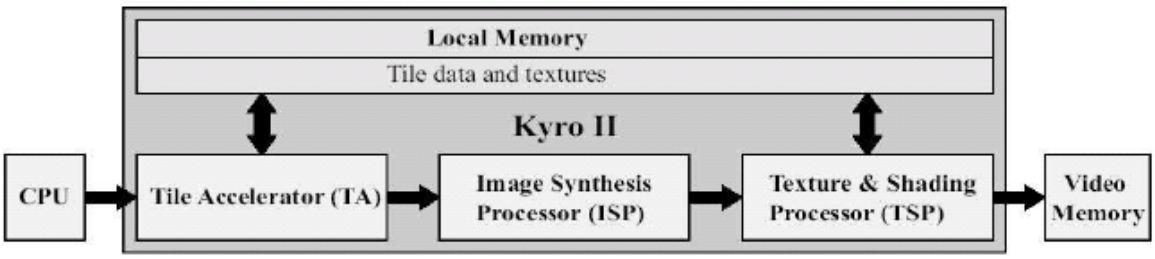


圖 2.10 採用tile-based架構的KYRO [12]

此架構由tile accelerator負責將三角形進行分類排序，分類完成之後，會將結果寫到此架構中自行設計的local memory之中。接著才依照tile的順序進行三角形的繪製，圖 2.10中的ISP跟TSP就是在進行rasterization部分的動作。繪製完的三角形也會回存至local memory，等到這個tile之中所有的三角形都繪製完畢之後，才將local memory裡的tile buffer寫回主記憶體中的frame buffer。

目前市場上仍採用tile-based架構所設計的作品有ARM的Mali [13]、Imagination Technologies的PowerVR系列[14]，和AMD&ATI的Xenos(for XBOX 360)[15]。前兩者主要用於嵌入式系統之中，後者則用於家用遊樂機。另外，Intel在 2008 年所發表的Larrabee繪圖系統[16]，計畫使用multi-core system來取代專用的繪圖處理器。其中為了能夠將三角形進行多工負載分配至不同的處理器進行處理，所採用的演算法，也是基於tile-based分類三角形的演算法去進行負載平衡。

2.5.2 Conventional immediate mode architecture

接著介紹在嵌入式系統中使用傳統immediate rendering mode架構所設計的繪圖處理晶片。我們在這邊引用An SoC with 1.3Gtexels/s 3D Graphics Full Pipeline Engine for Consumer Applications [17]這篇論文為例。這是由韓國高等科技學院(KAIST)所發表的完整繪圖加速晶片的設計，此設計的最大特點在於將Geometry engine跟rasterization engine裡的triangle setup設計成可程式化的硬體來執行更多樣化的指令。

在rasterization engine裡，除了triangle setup外，其特色還有使用特殊cache來進行早期深度測試、使用兩層Texture cache等。值得注意的地方就在於，由於不是採

用tile-based的架構，故需要有color cache跟depth cache負責減緩對外部記憶體存取所造成的影響，如圖 2.11中間偏右的C-line cache跟Z-line cache。

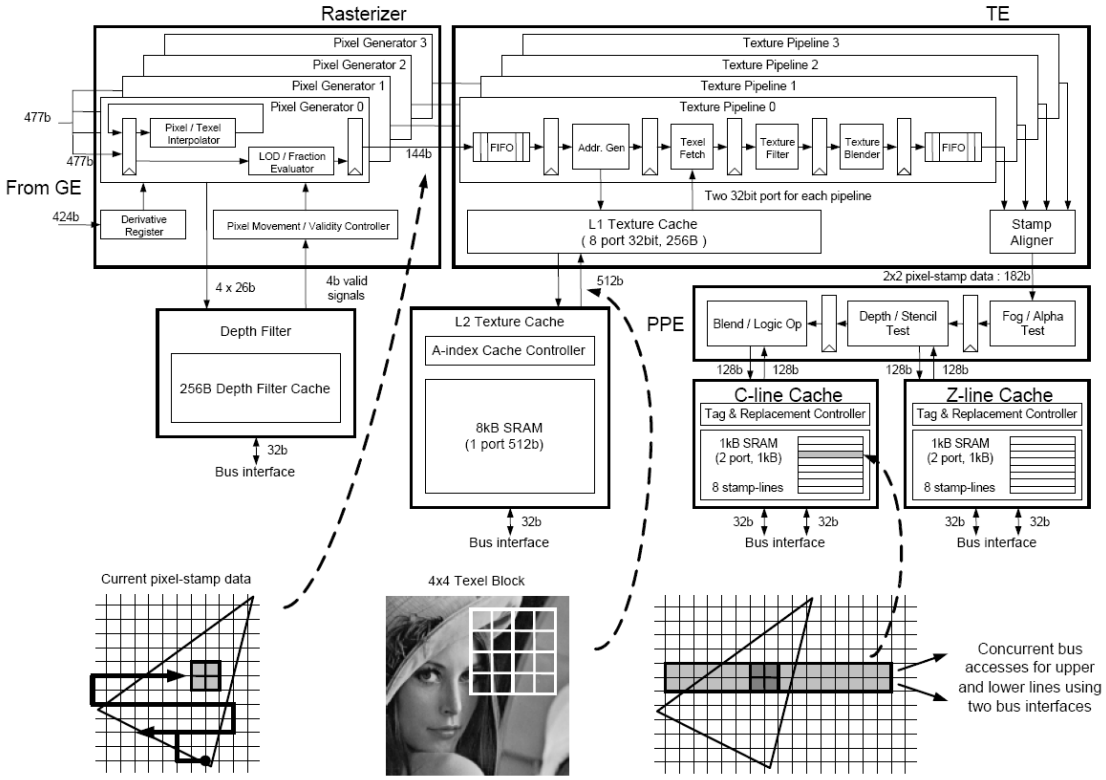


圖 2.11 An SoC with 1.3Gtexels/s 3D Graphics Full Pipeline Engine for Consumer Applications中rasterization的架構圖[17]

第3章 Rasterization algorithm exploration

本章旨在說明已研究過的各式用於 Rasterization 的演算法，詳列它們的優缺點，並決定我們的 Rasterization 將採用何種演算法，以及採用它的理由。

3.1 Edge function and Traversal

進入Rasterization後，第一步我們就要決定，那些fragment是位在三角形的內部。這邊我們使用Edge function來進行判斷[18]。

3.1.1 Edge function test

在平面上給定兩個點，將兩點連線可得一條無盡延伸的射線，並將此平面畫分為兩塊區域。這條射線可以簡單的用 $ax+by+c=0$ 此方程式表示，即為直線方程式。如果把平面上任意的一點(x,y)代入，可以得到一 e 值，我們就可以把方程式寫成 $e(x,y)=ax+by+c$ ，此即為 edge function。

e(x,y)依代入座標位置的不同，可以分成大於零、等於零、和小於零這三種情況。等於零的話，就表示這點在線上。大於零的話則表示此點在被線所畫開的其中一個平面之中，而小於零的點則在另外一半的那個平面上，至於是在哪一塊區域則要看你求直線方程式時所使用的向量方向而定，圖 3.1簡單的說明這三者之間與直線的關係。

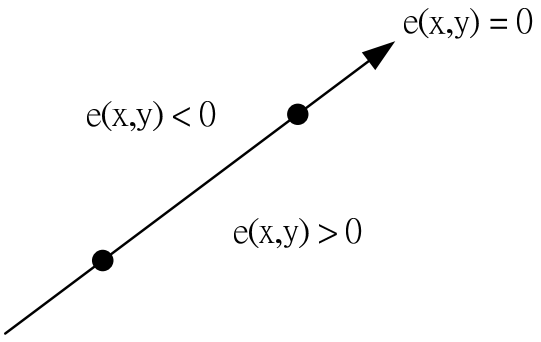


圖 3.1 Edge function 與平面不同區域的關係

有了edge function之後，我們就可以把它應用在尋找何處才是三角形內部區域中。首先，如圖 3.2所示，根據三角形的三個頂點(V_0、V_1、V_2)的X、Y座標值，我們可以找到三條直線(e_0、e_1、e_2)，跟它們的直線方程式。特別注意這邊在求直線方程式時，所應用的向量方向必須依順時針或逆時針繞行。

$$e_0(x,y) = -(y_2 - y_1)(x - x_1) + (x_2 - x_1)(y - y_1) = a_0 x + b_0 y + c_0$$
$$e_1(x,y) = -(y_0 - y_2)(x - x_2) + (x_0 - x_2)(y - y_2) = a_1 x + b_1 y + c_1$$
$$e_2(x,y) = -(y_1 - y_0)(x - x_0) + (x_1 - x_0)(y - y_0) = a_2 x + b_2 y + c_2$$

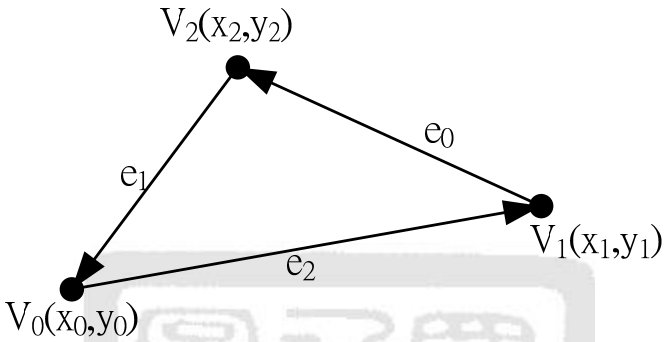


圖 3.2 三角形的頂點與邊界

接著開始將fragment的座標值逐一代入三條直線方程式，會得到三個值，如果三個值同號，也就是皆為正或皆為負時，就代表這個fragment在三角形的內部。相反的，只要三個值是異號，則這個fragment就位在三角形的外部。如圖 3.3所示

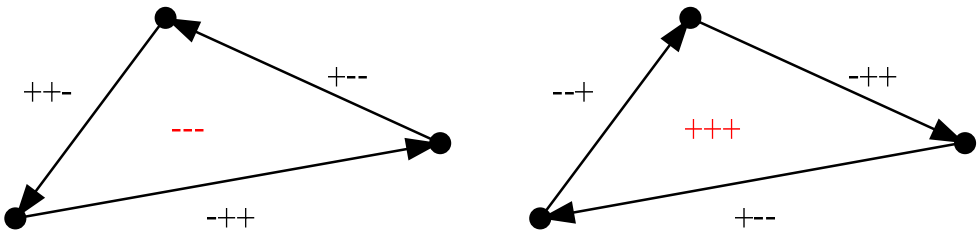


圖 3.3 edge function 的同號或異號決定了此點在三角型的內部或外部

3.1.2 Traversal algorithm

有了edge function test之後，我們就可以得知某fragment是否在三角形內部。可是光知道這還不夠，一個螢幕所含的pixel，依照螢幕的解析度，少則數萬，多則數千萬個pixel。對應於這些pixel的fragment難道都要一個一個丟進edge function測試嗎？這邊我們就需要建立一套掃描搜尋的機制。此即為traversal，也稱做scan conversion。以下先介紹我們找到的數種traversal的方法[18]。

A. Boundary Box traversal

一個最基本的traversal algorithm，稱為Boundary Box traversal。就是先找出一個能夠將三角形完全框住的最小方形區域，稱為邊界區域(Boundary Box)。然後從左而右，一行掃描完之後由下而上的繼續掃下一行，逐一將此邊界區域中的所有fragment皆掃過一遍，測試它們是否有在三角形裡面。如圖 3.4。

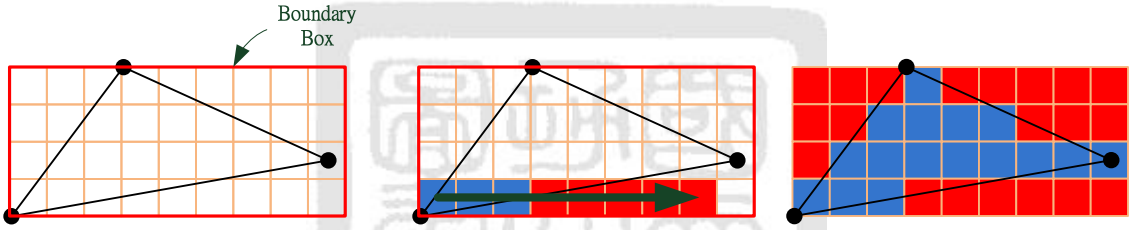


圖 3.4 Boundary Box traversal，藍色為有效 fragment，紅色為無效的 fragment

B. Skip traversal

接著介紹的是稱為Skip traversal的方法。此方法一樣是從左掃到右，而且也需要先找出boundary box，不同的是在一行之中，如果掃到了有效的fragment之後，接著掃到一個無效的fragment，那演算法就判定在此行之中尚未掃描的部分皆為無效，而直接跳下一行。整個演算法的過程如圖 3.5所示。我們可以明顯的看出，此演算法跳過了一些fragment而不用掃描。

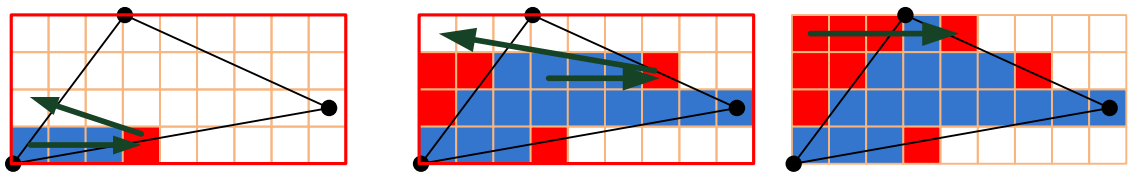


圖 3.5 Skip traversal，未著色的部分就是未掃描

C. Zigzag traversal

圖 3.6是zigzag traversal的流程示意圖。一開始先從三角形的一點開始，範例中是從左下角的那個頂點開始。接著的做法跟skip traversal相同，從有效fragment變成無效fragment之後就換下一行。接著就是這個演算法的重點，此演算法掃描的方向有兩種，從左到右或是從右到左，會交替轉變。換行的時候，直接從前一行最後掃到的無效fragment的上面那一個fragment開始掃，如圖 3.6的左圖。如果該fragment有效，則延續上一行的掃描方向去找當行該方向的第一個有效fragment，然後反方向掃描該行。如果該fragment無效，就直接跟上一行的掃描方向相反去掃描該行，如圖 3.6中間的示意圖。依照此步驟直到完成所有在Boundary box的scan line，即完成掃描。

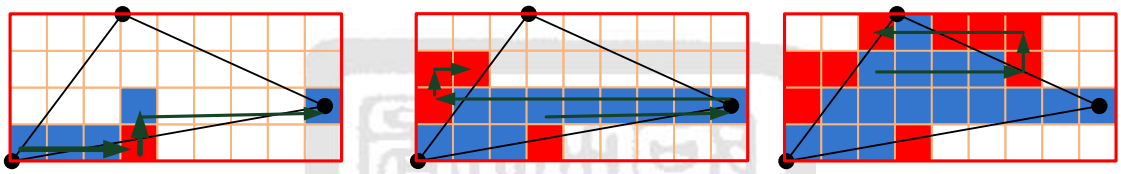


圖 3.6 Zigzag traversal

D. Bidirectional traversal

此方法較前幾種特殊。從三角形的一個頂點開始，如圖 3.7是從三角形下面那個頂點開始。然後以此為中心點，圖中淺綠色的點即為該行的中心點。從中心點往左右兩邊掃描。掃描完之後，先掃描中心點正上方的fragment是否為有效fragment。如果有效，則該fragment就成為該行的中心點並接著往左右掃描，如圖3.7左邊顯示的部分。如果無效，則一樣從該fragment開始往左右掃，把第一個掃到的有效fragment，當成該行的中心點。如圖 3.7中間跟右邊所顯示的一樣。依此步驟直到掃描完boundary box中所有的scan line。

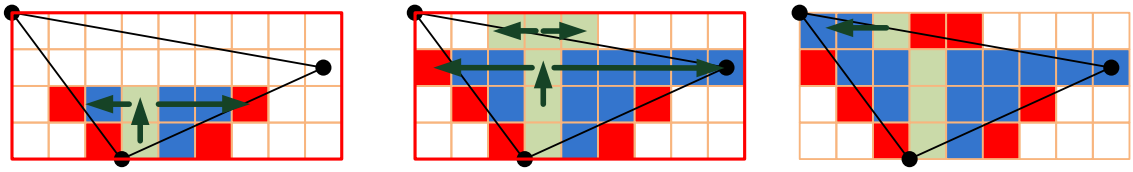


圖 3.7 Bidirectional traversal

3.1.3 Traversal method for tile-based architecture

介紹完以上數種 traversal 演算法之後，我們開始研究我們要採用何種演算法。在這邊必須先說明，我們整個繪圖管線的架構，是採用 tile-based architecture。這種架構，有可能會把三角形切割到不同的 tile 裡面去。所以，當我們正在處理某一塊 tile 時，我們必須避免處理到在此 tile 外的區域。也就是說，我們必須讓我們的 traversal 演算法在搜尋的時候不會跑到 tile 邊界以外的地方。對 zigzag traversal 和 bidirectional traversal 必須要從三角形某一頂點出發來說，要是三角形頂點落在 tile 之外而演算法又必須從那一個頂點出發，那我們不就必須從 tile 外開始搜尋起？這樣是不合乎我們要求的，所以我們只好放棄這兩個演算法，即使它們有較高的有效搜尋比例。

於是我們採用Skip traversal，不過我們必須修改boundary box的機制。我們在尋找boundary box的同時必須考慮tile 的邊界，就像圖 3.8所示。我們稱此為tile bounding skip traversal。

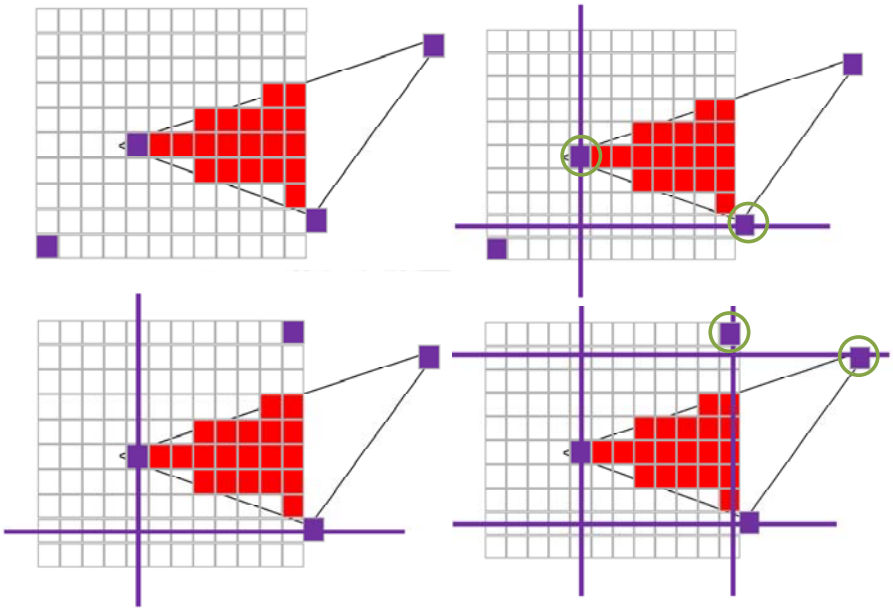


圖 3.8 在 tile 之中尋找三角形的 boundary box

在這個tile之中找出boundary box之後，再以左下角為起始點，進行skip

traversal。我們在這邊也有對skip traversal多加入了一條規則：如果前一行有掃到有效的fragment，而目前在掃描的這一整行都沒有掃到有效fragment的話，則當作已離開有效區域，掃描結束。這對於圖 3.9的情況相當有效。

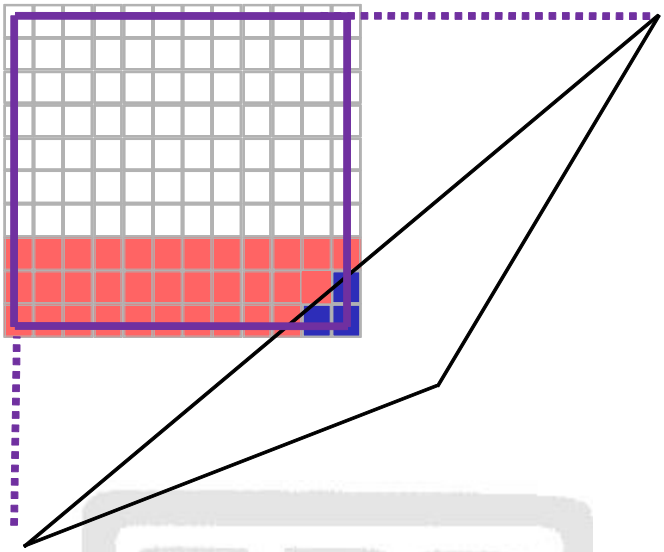


圖 3.9 Skip traversal with line skipping

3.1.4 The skip problem of tile-bounding skip traversal

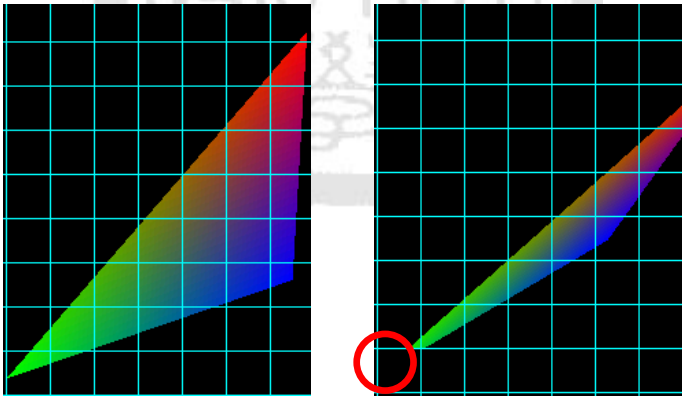


圖 3.10 執行 tile bounding skip traversal 時所發現的錯誤

如圖 3.10所示，我們試著讓這個三角形轉動，當三角形左下角的角度慢慢變小時，其中一個tile裡原本該畫出部分三角形的地方，竟然沒有畫出來。出現這個錯誤原因，就是因為我們幫skip traversal在前一小節中所加上的規則所產生的。

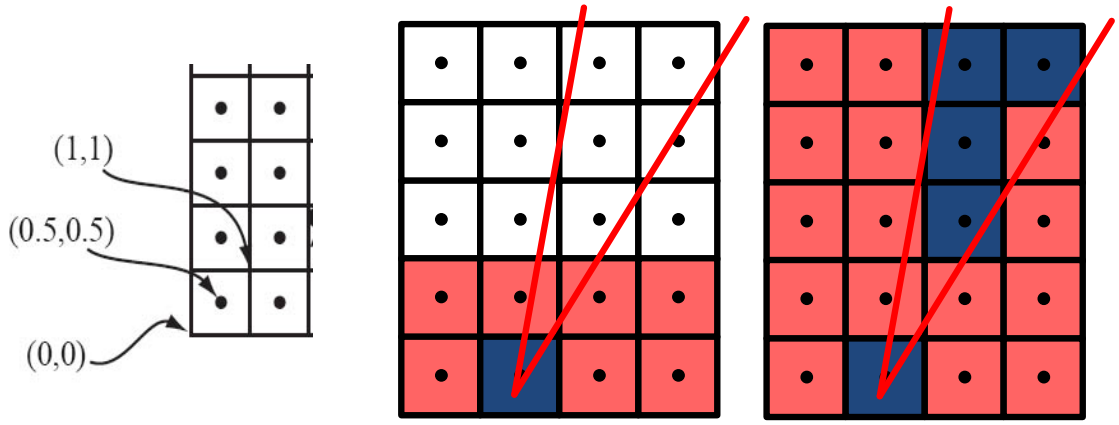


圖 3.11　造成 skip traversal 錯誤的原因

造成這個錯誤的原因，主要是因為相鄰兩個fragment的中心點座標值在數學上是不連續(discontinuous)的。如圖 3.11最左邊的圖，我們可以知道最下面的兩個fragment的中心點座標分別為(0.5 , 0.5)、(1.5 , 1.5)，我們再進行traversal的時候是只對fragment的中心點座標進行測試，在這兩點之間的區域就沒有測到。而我們後來加入skip traversal中的規則就明顯的將這個問題展現出來。如圖 3.11中間跟右邊的圖，掃描此三角形的第二行時，由於沒有蓋到任何一個fragment中心的取樣點，導致整行都沒有有效的fragment被掃出，於是演算法就以為三角形已經被掃描完畢而發生錯誤。

我們在這邊一開始所使用的解決方法，就是去對fragment的周圍四個點進行edge function測試，這樣就可以精確的判斷這個fragment是不是有被三角形覆蓋到。只要有一點點的覆蓋到，就算是在三角形裡面。則圖 3.11右邊的錯誤就會變成圖 3.12右邊的圖而被解決。

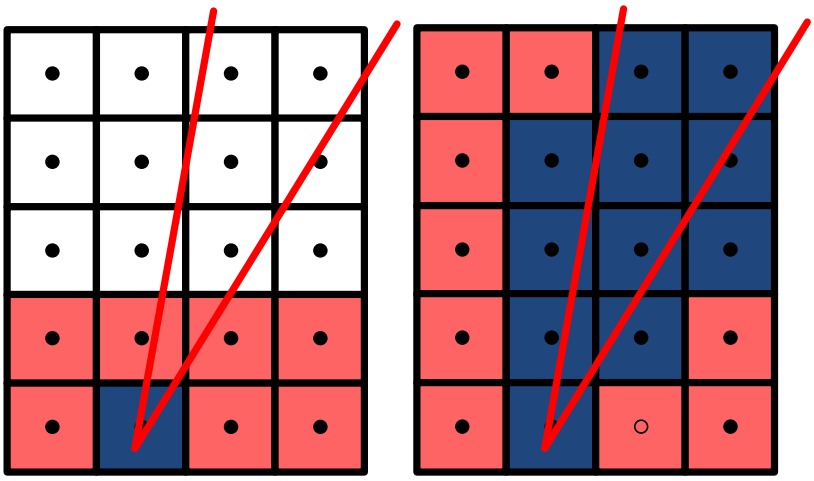


圖 3.12 改變edge function測試位置對圖 3.11所產生的改變

不過，這樣又產生了一個新的問題。由於我們現在使用測試fragment四個角落來判斷fragment是否在一個三角形的內部。而這個問題就是發生在，當同時間有多個三角形都覆蓋到同一個fragment時，就產生了嚴重的問題：到底這個fragment是屬於哪一個三角形的？而圖 3.13中的鋸齒邊緣即為此問題、三角形互相搶奪fragment的現象。

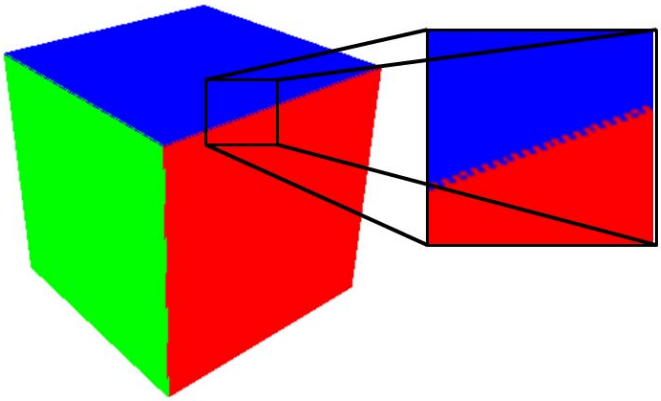


圖 3.13 三角形互相搶奪屬於自己 fragment 所發生現象

因此，我們最後決定，加上以三角形的中心點做為判斷三角形依歸的依據。也就是擁有fragment中心點的三角形才擁有此fragment。而fragment周圍四個點的判斷就成為只決定此行有沒有有效的fragment，用來給skip traversal參考用。如圖 3.14紅色的點為無效的fragment，而黃色跟藍色的點都為有效的fragment，可是黃色的

點由於中心點不屬於此三角形，所以不會著色；藍色的點才是真正屬於三角形，並且也會著色。

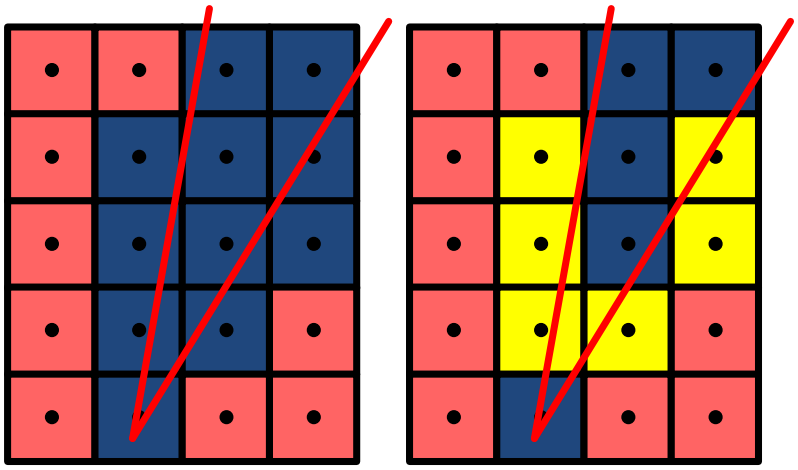


圖 3.14 對圖 3.12加上測試三角形中心點後所造成的改變

3.1.5 Performance of tile bounding skip traversal

接著我們開始評估這個演算法的有效掃描比例，由於 traversal 在進行掃描的時候，會走訪到有效跟無效的區域，此一比例即為有效掃描比例。演算法的有效掃描比例越高，理論上就越好，不過實際上演算法複雜度也是另一個需要考慮的地方。

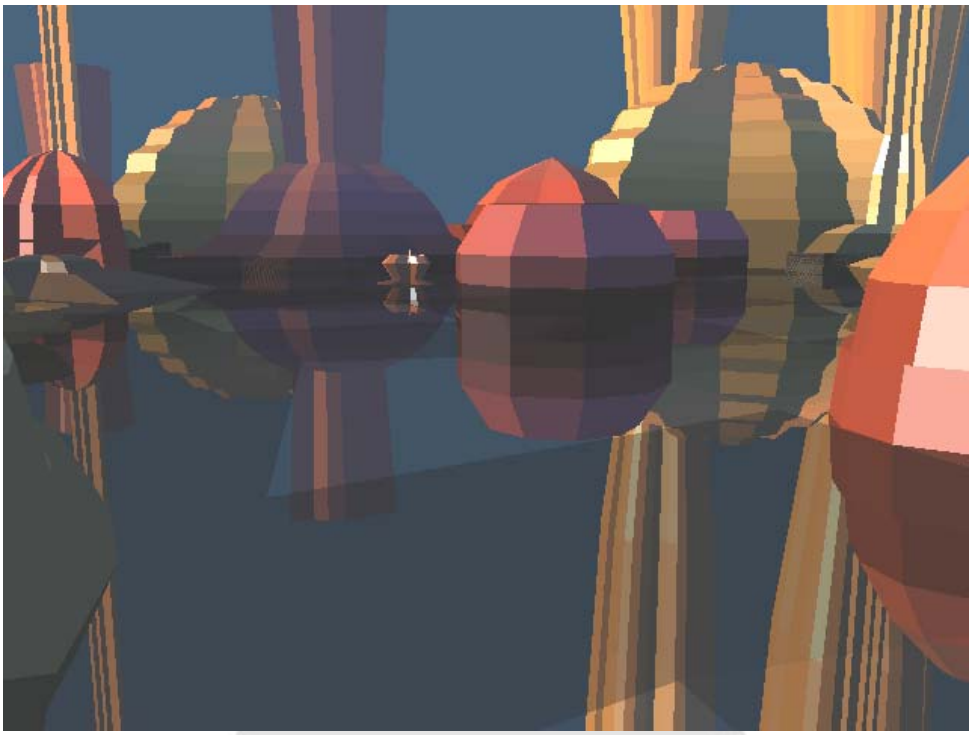


圖 3.15 San angels 的模擬畫面

我們在這邊使用一個叫San angels的benchmark，此benchmark跑起來的畫面如圖 3.15所示，我們讓此benchmark跑了 2000 個畫面(frame)，計算每個畫面的有效掃描比例，並將它們平均起來。這邊我們得到了我們tile bounding skip traversal的有效掃描比例平均約為 45%，最低為 25.3%，最高時則為 67.3%。

3.2 Interpolation

找出三角形裡面的 fragment 之後，就要開始內插該 fragment 的深度、貼圖座標、顏色、透視投影修正值等等資訊。最直觀的做法，就是用平面方程式來做內插，以下將介紹此做法。

3.2.1 Plane equation

假設今天有一個fragment座標(x, y)位在圖 3.2的三角形內，我們要內插出此點的深度值。首先，我們就要依照此三角形三個頂點(V_0、V_1、V_2)的三維座標來找出

此三角形的平面方程式 $ax+by+cz+d=0$，接著把該fragment的座標(x, y)代入此方程式之中，就可以得到z，也就是深度值。這就是使用平面方程式進行內插的基本想法。

依照平面方程式 $ax+by+cz+d=0$，我們可以知道，以方程式係數所組成的向量 (a,b,c)，就是此平面的法向量N。而要求出此平面的法向量，首先要任取此平面中兩互不平行的向量。我們可以用三角形的三個頂點中來取出兩向量 $\vec{u}(\overrightarrow{V_0V_2})$ 和 $\vec{v}(\overrightarrow{V_0V_1})$ 如圖 3.16，然後用 $N = \left(\begin{vmatrix} u_y & u_z \\ v_y & v_z \end{vmatrix}, \begin{vmatrix} u_z & u_x \\ v_z & v_x \end{vmatrix}, \begin{vmatrix} u_x & u_y \\ v_x & v_y \end{vmatrix}\right)$ 就可以求出法向量N。

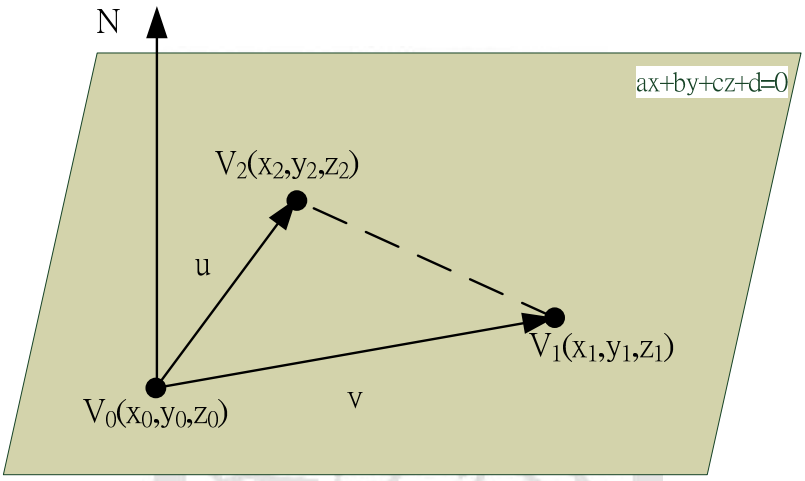


圖 3.16 平面方程式、三角形與法向量的關係

這樣平面方程式只差一個常數項 d 就完成了，不過好消息是，我們並不需要用 d 這個常數項。這邊，我們其實不完全是依照代入平面方程式來求得 z 值。我們可以先將平面方程式改寫為

$$ax + by + cz + d = 0$$
$$=> -cz = ax + by + d$$
$$=> z = \frac{a}{-c}x + \frac{b}{-c}y + \frac{d}{-c}$$
$$dz_y = \frac{b}{-c}, dz_x = \frac{a}{-c}$$
$$z = z_0 + dz_x \cdot (x - x_0) + dz_y \cdot (y - y_0)$$

這樣我們只要知道平面上任一點的座標的 z 值，就可以從那一點座標推導出我們指定座標的 z 值。剛好我們的三角形頂點都位在平面上面且都擁有 z 值，我們只要隨意取出其中一個頂點的 z 值，就可以完成整個方程式。到這邊深度內插就完成了。

可是還有其他值需要被內插。此時，我們只需要將原本整個系統中所有跟深度相關的資訊，換成指定類型的值，譬如顏色值。也可以這麼說，將三角形三個頂點的顏色值，偽裝並取代深度值，重新求一次平面方程式，就可以完成新的「顏色」方程式，就可以進行顏色的內插了。

3.2.2 Plane equation with fixed-point calculation

基本上，3.2.1 中所敘述的是一個正確無誤的方程式推導，我們將它寫進程式裡面模擬，也得到正確的結果。可是，在我們完成整個軟體部分的驗證、將程式重新改寫成定點數(fixed-point)版本、進行精確度測試的時候，出現了一個問題。圖 3.17左上半部就是此問題當初所展現的圖形錯誤。

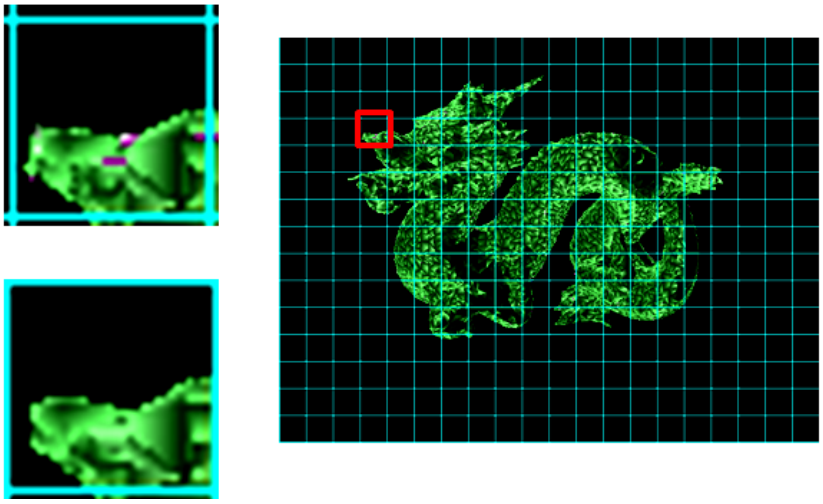


圖 3.17 因為定點數精確度不足，再進行 traversal 與 interpolation 時產生圖形錯誤。坐上角出現錯誤的是定點數軟體版本，而左下角沒有錯誤的是浮點數的軟體版本。

造成這個錯誤的原因，主要是edge function的精確度不足，使得一些本來在三角形外的fragment被當成在三角形內所導致的。而這些超出三角形範圍的fragment，就有可能在內插中發生溢位(overflow)的情形，如圖 3.18中發生顏色溢位範例。

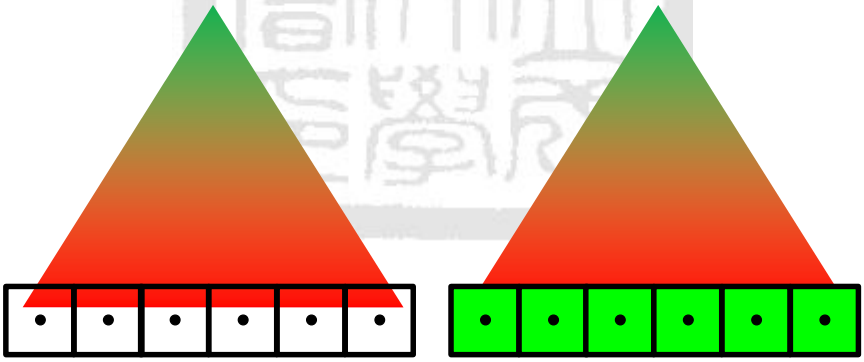


圖 3.18 因 edge function 精確度不足，所產生的溢位問題

這樣的溢位問題，看起來似乎可以簡單的用限制顏色值的範圍而達成。我們在軟體模擬之後，雖然看起來不會再溢位，可是在一些小面積三角形周圍外的fragment，代入低精確的 edge function 且發生誤判為有效時，顏色看起來會很不自然，即使它們沒有溢位。於是我們採用另一種內插演算法，稱為 barycentric coordinate interpolation。

3.2.3 Interpolation using barycentric coordinates

Barycentric coordinate為質心座標的意思，不過在這邊，並不是指質心的座標，而是指在任一點在三角形裡面的相對座標。此座標系統又被稱為areal coordinate。以下我們稱它為質心座標系。質心座標系有三個方向，分別根據三角形三個方向的高來作對應，如圖 3.19中的u,v,w各為一個方向上的座標。每一個方向上，由底邊當作 0 出發，到另一端的頂點當作 1 結束。依照圖 3.19中黑點的位子來看，我們可以知道它的質心座標就為(0.4, 0.3, 0.2)。

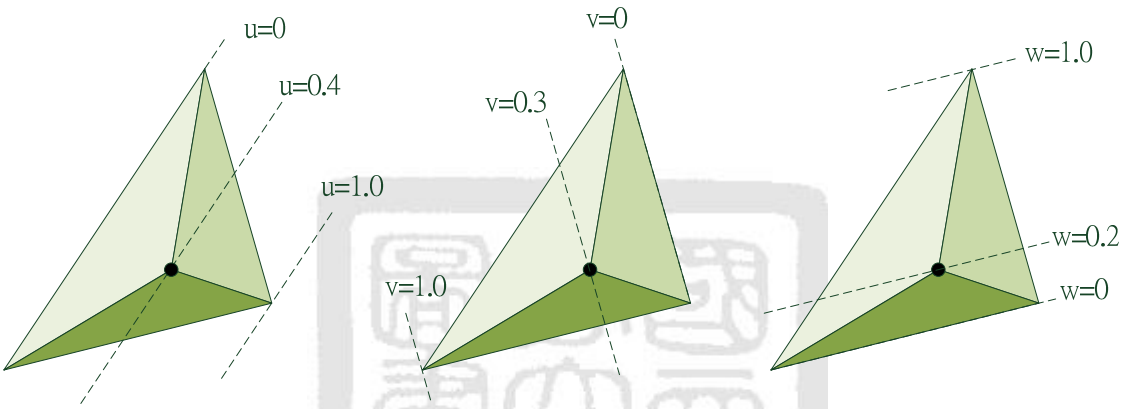


圖 3.19 質心座標系與它三個方向的座標

質心座標的求法，是依照該點與對應邊所組成的三角形面積占了全體三角形面的百分比來決定的。如圖 3.20中P點與V_1V_2 所組成的三角形A_1 就占了全部三角形A_Δ(A_{1+} A_{2+} A_0)的 40%左右，所以P點在u方向上的質心座標即為 0.4。這邊我們可以得到一個基本的公式：$P(u,v,w)=\frac{(A_1,A_2,A_0)}{A_\Delta}$，也就是P點的質心座標。另外 $A_1+A_2+A_0=A_\Delta \Rightarrow u+v+w=1$。

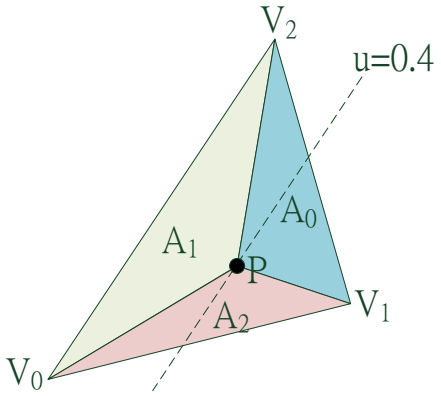


圖 3.20 P 點與三頂點所形成三角形與質心座標的關係

那麼我們要怎麼求三角形的面積呢？基本上是使用外積法(cross product)。以 A_1 為例：$A_1 = \frac{1}{2}\left[(P_X - V_{0X})(V_{2Y} - V_{0Y}) - (P_Y - V_{0Y})(V_{2X} - V_{0X})\right]$。這樣我們就可以用類似的方法把所有的三角形的面積都求出來。上面這個方程式其實跟圖 3.2上方的方程式一模一樣，只差一個 1/2。也就是我們可以把當初在代入edge function時所得到的結果直接留下來到這邊應用，而不用進行內部三個小三角形的面積運算，唯一需要計算的只有整個大三角形的面積。

現在我們有P點的質心座標(u,v,w)了，接著就要進行內插。以下我們用內插圖3.20中P點的深度值來做範例。

$$\begin{aligned} P_Z &= u*V_{1Z} + v*V_{2Z} + w*V_{0Z} \\ &= u*V_{1Z} + v*V_{2Z} + (1-u-v)*V_{0Z} \\ &= u*(V_{1Z} - V_{0Z}) + v*(V_{2Z} - V_{0Z}) + V_{0Z} \end{aligned}$$

此公式基本的概念，就是將三方向的質心座標，跟對應的頂點資料相乘之後，再把三個結果相加。

使用質心座標跟使用平面方程式比起來有兩個明顯的好處：第一，由於質量座標系統的關係，u,v,w都必需介於 0 到 1 之間。所以這邊可以輕易的避免掉overflow的問題。等於超出三角形的座標都會被移回三角形的邊界，再經過內插後所得到的值，就會是剛好位在邊界上的值，如圖 3.21所示。圖 3.22則顯示了質心座標法進行內插的結果。我們可以看到，原本左圖使用平面方程式進行內插時會有一些點發生顏色溢位的情形，換成質心座標法之後就改善了。

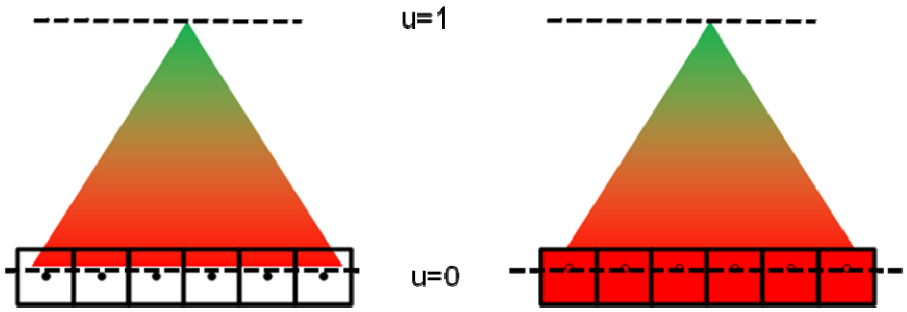


圖 3.21 超出三角形邊界的點經質心座標法內插之後的情況

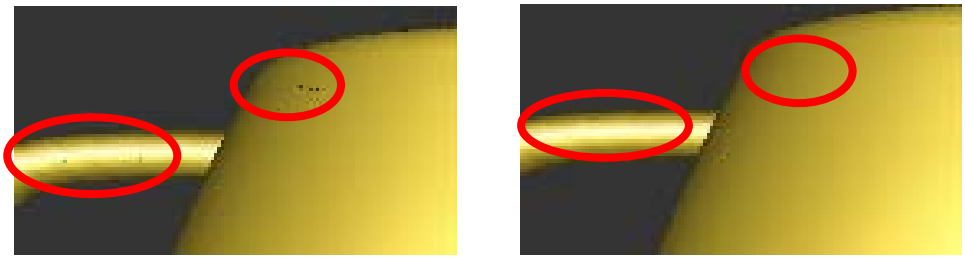


圖 3.22 使用平面方程式與質心座標法進行內插的差異

第二，得到質心座標之後，我們就可以根據質心座標跟三角形各頂點的值，直接內插出所有需要的資料。而使用平面方程式進行內插的話，還必須一個一個計算平面方程式，要內插多少種值出來，就必須要計算多少次平面方程式。比較起來，採用質心座標法進行內插可以明顯的節省計算複雜度。因此我們決定採用此內插演算法。

不過，此內插法不能改變一個事實。就是原本不屬於三角形的點仍然存在，物件的形狀可能會有輕微的變形。不過由於此超出三角形範位點的顏色跟邊界上點的顏色相同，所以肉眼難以發覺。也就是說此內插法是在彌補精確降低時所犯下的錯誤。

3.3 Anti-Aliasing(AA)

反鋸齒(Anti-Aliasing)是目前繪圖中用來增進畫質的重要技術，可以藉由增加在同一個fragment中採樣點的數目來讓圖型更平順，更真實，如圖 3.23。在fragment增加的採樣點數目，也必須通過traversal跟interpolation階段。也就是說，把多出來

的採樣點，當成真正存在的fragment來計算、來內插，最後還會將frame buffer擴大來儲存這些多出來的fragment。這種做法跟一種稱為sub-pixel的概念有關。實際上我們也可以這樣理解，把解析度擴大到我們指定的倍數並計算完畢後，才降低解析度並將多個fragment做混色處理成單一fragment。也因此，此方法的全名稱為全景反鋸齒(Full Scene Anti-Aliasing - FSAA)。在這邊，我們介紹 3 種基本的反鋸齒方法。

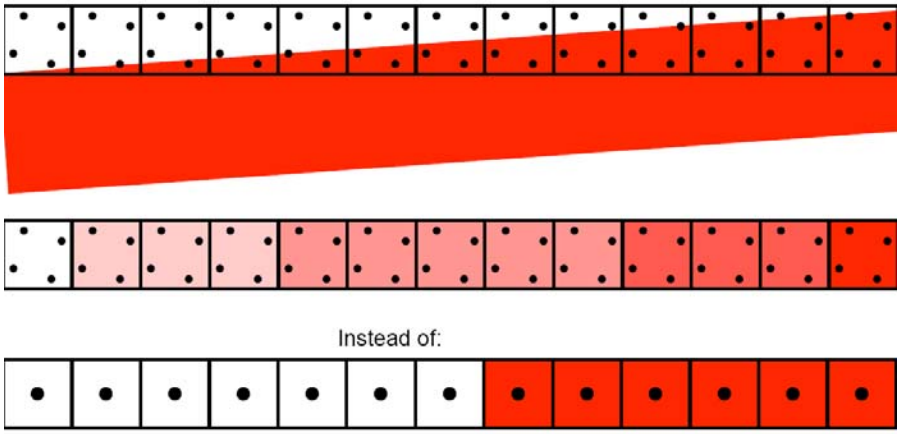


圖 3.23 增加取樣點來使 pixel 的顏色更接近平均值

在開始介紹以前，先放一張沒有開反鋸齒的圖以供比較。我們可以用圖 3.24中間那條接近垂直帶有一個鋸齒的軸線當作反鋸齒效果好壞的比較。

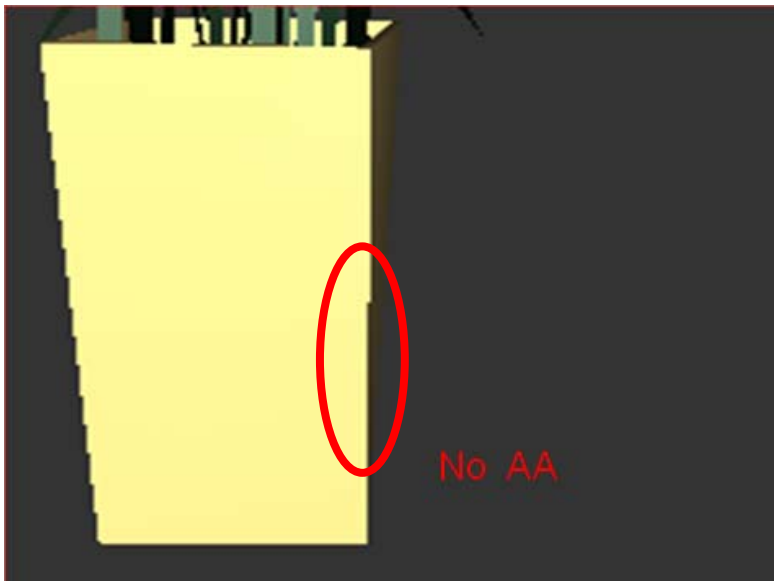


圖 3.24 沒有開反鋸齒的圖像

A. 2x anti-aliasing

每個 fragment 取樣四次，分別在四個角落，是標準兩倍反鋸齒所使用的方是。這種方法等同於解析度擴大 4 倍，所以效能理論上會降低 4 倍，且須要 4 倍的 frame buffer 空間。

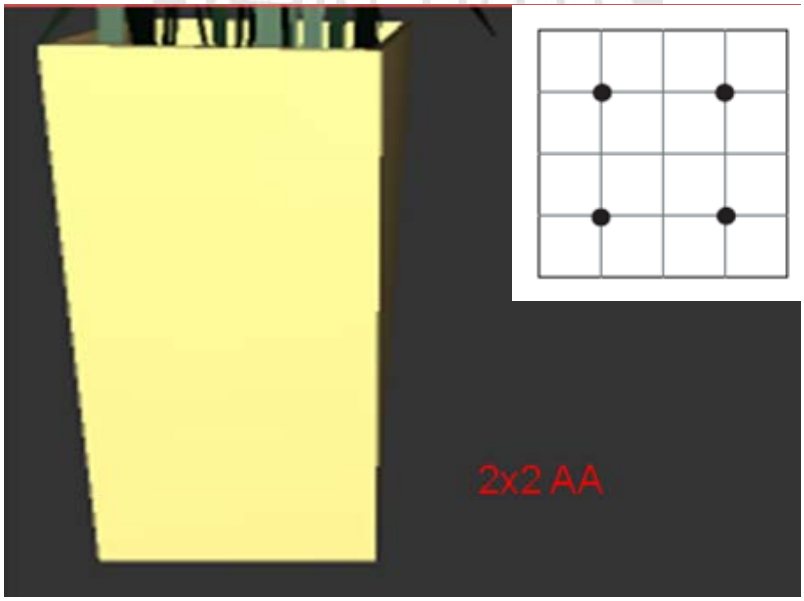


圖 3.25 2x super sampling 的樣式與範例圖片

B. Diagonal anti-aliasing

兩個取樣點，分別在 pixel 對角的位置，得到的品質接近 2x super sampling，但只需要降低一倍的效能與增加一倍的 frame buffer，這是這個取樣格式最大的優點。但是，其主要缺點為無法解決接近 45 度角所產生出來的鋸齒。

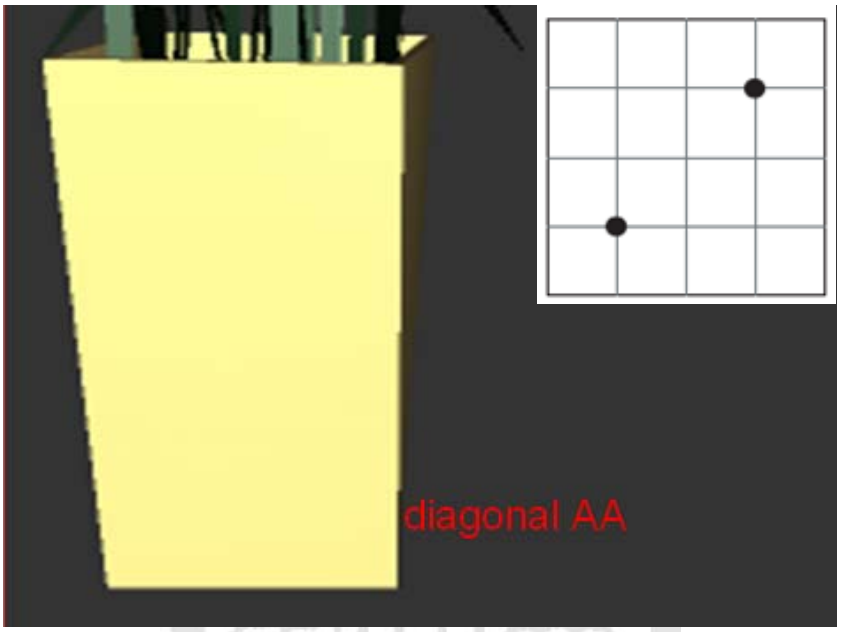


圖 3.26 Diagonal super sampling

C. Rotated grid super sampling(RGSS)

速度和需要的frame buffer大小與 2x super sampling相同，但取樣點根據n-rook algorithm [19]取在不同行列的 4 個位置上，效果較 2x super sampling好上許多。接近標準的 4x super sampling的畫質。

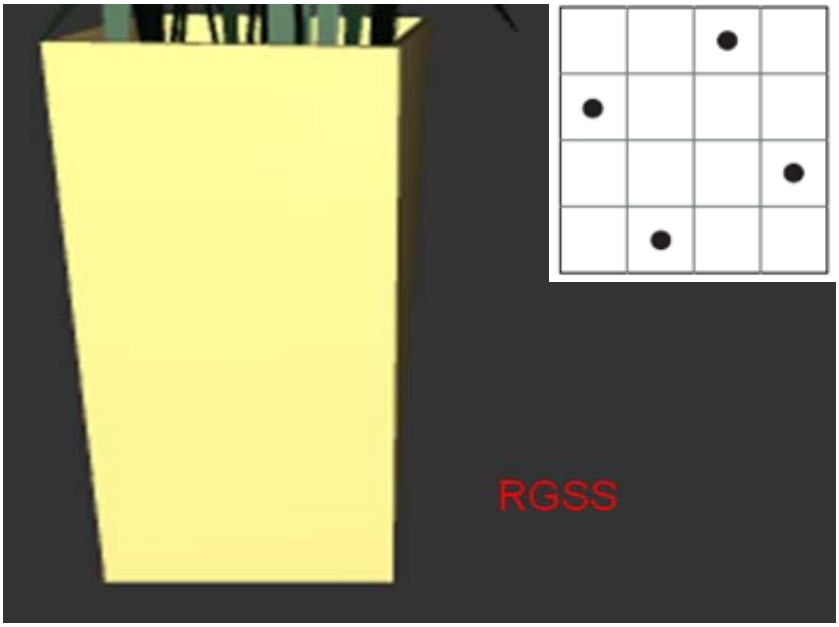


圖 3.27 Rotated grid super sampling

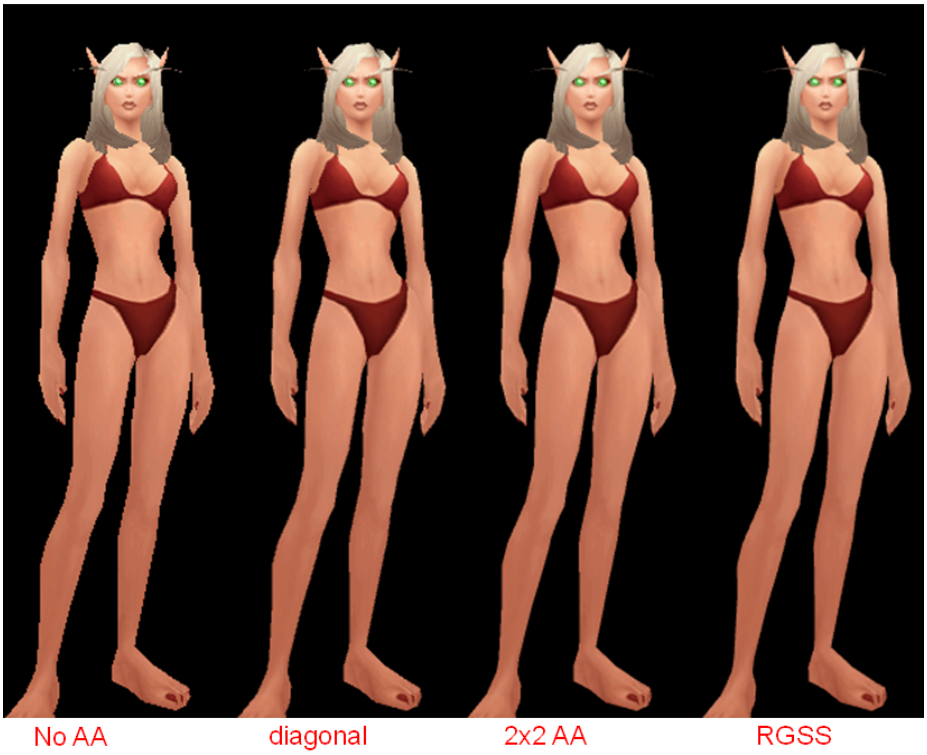


圖 3.28 無反鋸齒跟三種反鋸齒比較

增加採樣點雖可以獲得反鋸齒的效果，但需要較多的計算量，同時也需要較大的 frame buffer。由於採用 tile-based 架構，我們將有一塊 On-chip 的 tile buffer 來存放一塊 32*32 大小的像素資料。如果我們開啟 RGSS 反鋸齒的話，由於一個像素之中將會有四個取樣點，一個能夠存放 32*32 個像素大小所需的 tile buffer 將會是原來的四倍。這邊不太可能在硬體裡面做出一個不開反鋸齒時能夠存放 128*128 個像素的 tile buffer，因此我們選擇將隨著反鋸齒時取樣點的倍數來動態調整 tile size 的大小，來符合 tile buffer 的大小。例如在開啟 RGSS 反鋸齒時，將 tile size 縮小到 16*16；在開啟 diagonal AA 時，將 tile size 縮減成 32*16。

3.4 Texture mapping

將 2D圖像貼附在 3D物件的表面，稱為texture mapping，這件事情已經在 2.3 節中加以介紹過，我們的軟體模型有將它給做出來。在texture filter部分，我們也將nearest filter和linear filter實做出來。除此之外，OpenGL還有定義貼圖的環境模式，也就是外來的貼圖影像跟原本三角形的顏色要進行何種運算。我們這邊實做了四種，它們分別是replace、modulate、blend、和add。這四種模式的運算方式跟對圖形產生的結果再圖 3.29有詳細的展示。

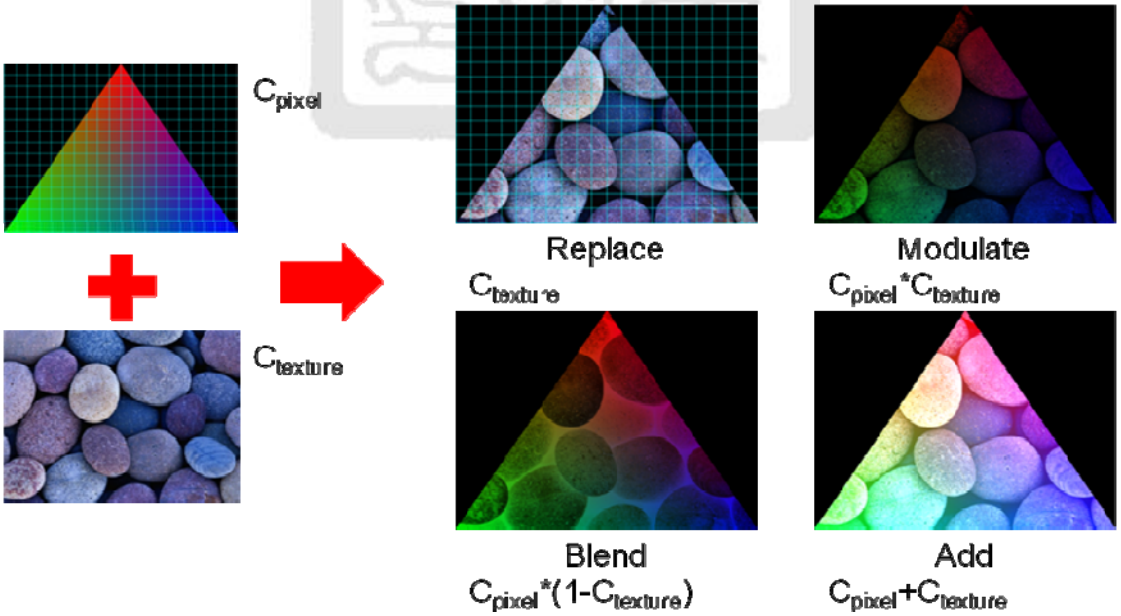


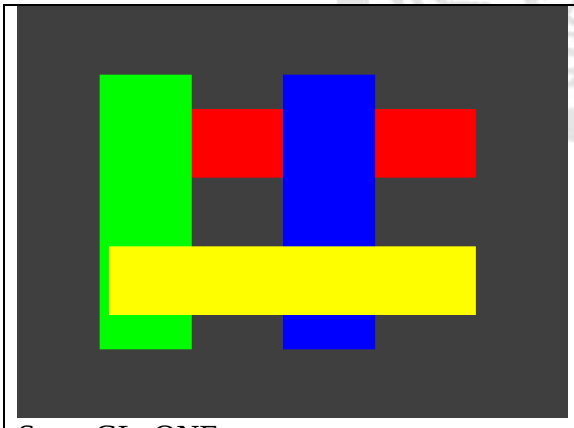
圖 3.29 各種不同的 texture environment mode

一般程式設計者在這邊最常使用的模式為 modulate，而不是 replace。因為這樣才能在貼上來的圖像中保留當初對三角形打光後所產生的顏色變化。如果用 replace 的話，貼上來的圖形就會完全覆蓋三角形的顏色而喪失及時光影的變化了。

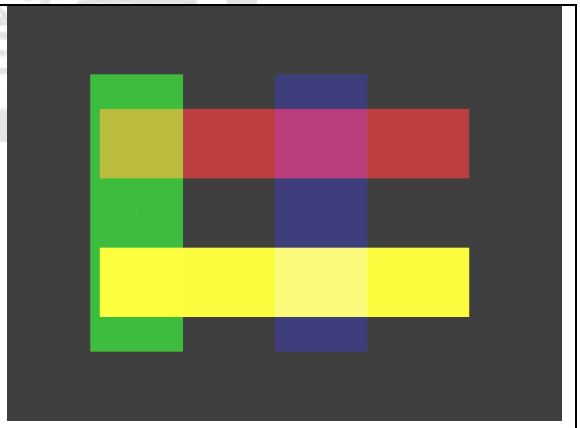
3.5 Per-fragment operation

透明度測試(Alpha test)跟深度測試(Depth test)在 OpenGL 規格書中都定義的相當的清楚，軟體實做起來也沒甚麼問題。比較複雜的則是半透明混色運算(Alpha blending)。

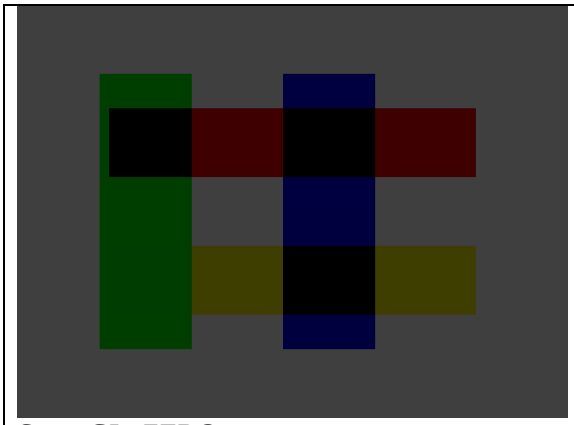
半透明混色運算在OpenGL中有定義來源使用模式(*src* mode)跟目的使用模式(*dst* mode)。而在進行透明度混色時，就要將來源、也就是通過透明度測試跟深度測試的fragment所擁有的顏色乘上來源使用模式，然後將目的、也就是從frame buffer過來的pixel所擁有的顏色乘上目的使用模式後相加。來源使用模式跟目的使用模式有可能是定值，如GL_ONE或GL_ZERO就代表著 1 跟 0。也有可能是通過管線fragment或來至frame buffer中pixel的透明值(alpha)。這兩種模式的變化搭配起來就將近 100 種。這些我們的都有做在軟體上面來進行模擬，下面的圖 3.30就簡單的列舉四種組合，並顯示它們組合後的影像結果。



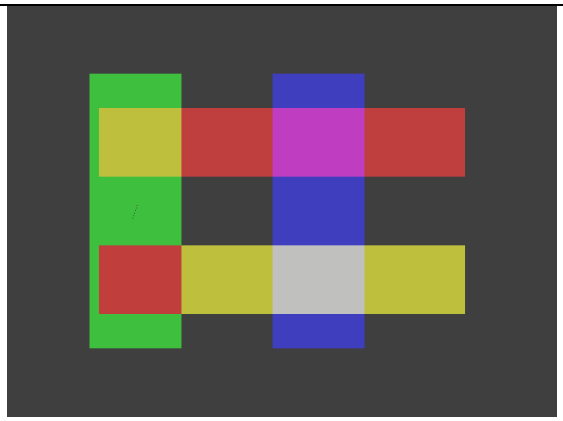
Src = GL_ONE
DST = GL_ZERO



Src = GL_SRC_ALPHA
DST = GL_ONE



Src = GL_ZERO
DST = GL_SRC_COLOR



Src = GL_ONE_MINUS_DST_COLOR
DST = GL_ONE_MINUS_SRC_COLOR

圖 3.30 不同 blending mode 的範例圖

第4章　Architecture development

圖 4.1顯示的是根據tile-based架構所擬定的整體RM架構圖。橘色的區塊代表的是記憶元件，包括一個 用來暫存從bus讀近來的資料的triangle input buffer跟一個用來當作tile buffer的on-chip buffer。中間的方塊則是計算元件，代表著第三章所介紹過的各個子模組。箭頭則是資料傳遞的方向，由於Texture mapping跟Per-fragment operation都需要從bus上讀取資料，所以這兩個模組除了從上一階的模組獲取資料外，也會有從bus來的資料傳遞箭頭。

接著，我們開始對整個 architecture 進行研究討論，找出可以提升效能的方式。第一個就是我們可以看到，從 bus 進來的資料大部分都會先通過 buffer，惟獨 Texture mapping 這邊的 buffer 還沒加以探討，所以我們將先加上 Texture buffer，也就是 texture cache。

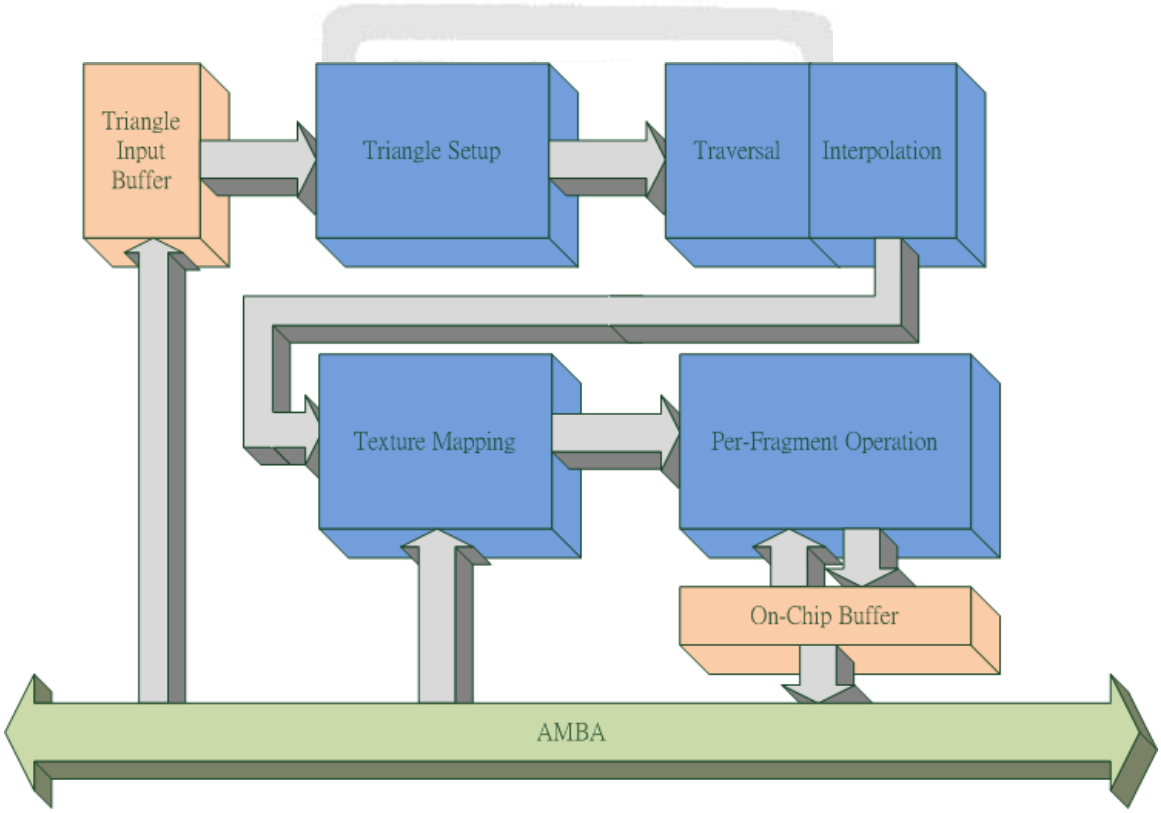


圖 4.1 依照 tile-based 架構所擬定的 RM 硬體

4.1 Texture cache

我們參考過已實做出來的GPU，幾乎都有加上Texture cache，以減少對記憶體的存取次數。於是我們一開始也試著模擬一塊基本的texture cache，一個擁有 256 bytes、line size 4 word的cache，可是模擬起來的效能相當的差，平均hit rate只有 50%左右。另外，我們也找到了數篇探討texture cache的論文[20][21][22]，其中一篇提到的 6D block cache的架構[23]，擁有相當高的hit rate。研究它的cache架構跟行為模式後，我們認為這跟texture mapping的動作相當的吻合，此架構在簡單的texture mapping動作中恐怕已經是最佳解。於是我們採用此架構做為我們的texture cache的基本模型。

4.1.1 6D block texture cache

一個fragment在存取材質影像(Texture image)時，需要一個二維空間上texture座標(s,t)來決定它要使用的texel在材質影像的位址。這邊一個texel就要 32 bits(RGBA各 8 個bit)，所以一個texel也就是一個word。而 6D block cache還會把s跟t這兩個座標再拆成三個階層：superblock、block、跟offset，如圖 4.2。這樣看起來就有 6 個座標，這也是 6D命名的由來。

圖 4.2 6D block cache 的位址排法

接著說明這六個座標的定義。Offset的意義跟line size的定義一樣，只不過這邊定義的cache是二維(s,t)的，而傳統的line size的想法是一維的。如果我定義S跟T的offset最大都是 2 的話，則一個基本的cache區塊的大小就是 4 個texel(word)。block的多寡則決定了整個cache的大小，依照傳統cache的想法，一個block就等於一個entry。剩下的部分則歸類到superblock這個址域，類似於傳統cache上的index。圖 4.3顯示了 6D block cache如何根據 6D的(s,t)位址抓取texel，這邊有一個地方要注意的

是，中間cache-sized大小的super block中各個block的排序，因為fragment不一定是循序存取texel，理論上不會像圖中的那麼整齊，只有在block-sized大小那邊的各個offset才會真的如圖中那麼整齊排列。

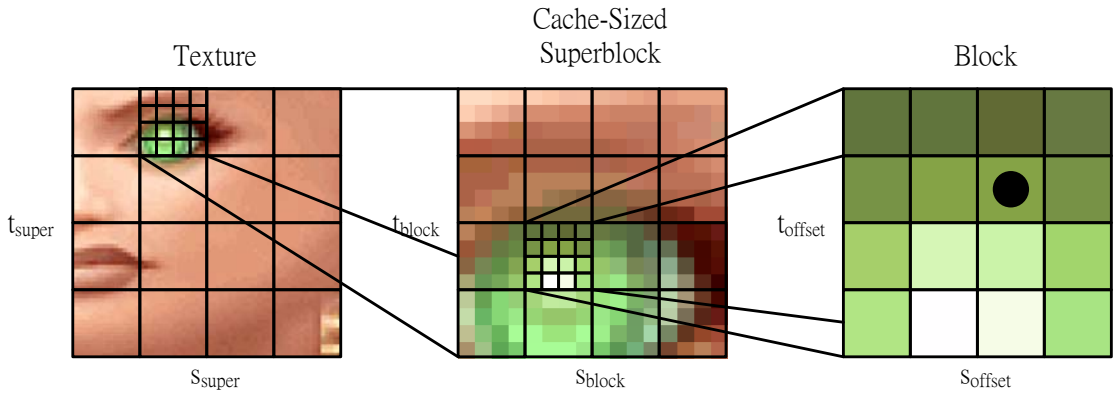


圖 4.3 材質影像跟 6D block texture cache 的組織示意圖

此 cache 的 policy 採用 direct-map 的方式，如果發生 cache miss 的話，就會將整個 block 裡面所有的 texel 通通拋棄。然後將 miss 掉的 texel 及 texel 所屬的整個 block 置換進對應的 block 裡。與傳統 cache 中最大的不同，就是在於整個 cache 定址的方式使用了二維平面的概念，而不是傳統 cache 中或是記憶體中的一維線性定址。由於 texture mapping 一次貼的是一個三角形，而三角形本來就是一個區塊的表現，此 cache 置換方式，是相當符合我們 texture mapping 所進行的動作。不過，這樣就會有一個小缺點，由於影像被作業系統讀進記憶體時，是以一行一行的方式來做為連續定址。所以我們可以想見，當此 cache 發生 miss 時要進行記憶體存取時，就不能使用 16-word burst read，而必須要進行 4 次的 4-word burst read。

4.1.2 Simulation of texture cache

接著就要對此texture cache進行模擬測試，我們在這邊使用如圖 4.4的物件跟貼圖，包括一個 32*32 跟兩個 256*256 材質貼圖，來進行我們的測試。除此之外，此物件貼圖的filter採用linear filter，這也會對cache hit rate產生一定的影響。

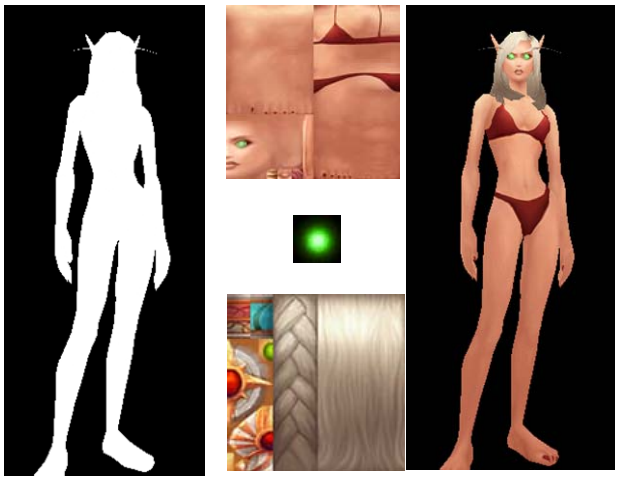


圖 4.4 包含 2 個 256*256 跟 1 個 32*32 材質貼圖的模型

另外，我們還可以決定我們一個block要放多少個texel(word)，在進行模擬實我們使用了 4-words、16-words、64-words這三種不同的大小做為我們的測試不同block size的模型。同時我們也改變整個cache的大小，這邊我們測試了 256 bytes、1K bytes、跟 4k bytes，來觀察不同大小對cache miss rate所造成的影響。藉由調整這兩參數，來建立不同的cache模型以進行測試。表 4.1列了在不同參數的調整下，cache miss rate的大小。

表 4.1 調整 block size 跟 cache size 時對 cache miss rate 的影響

	4 words	16 words	64 words
256 bytes	17.32%	9.06%	19.48%
1K bytes	8.8%	4.35%	3.35%
4k bytes	7.3%	2.56%	1.17%

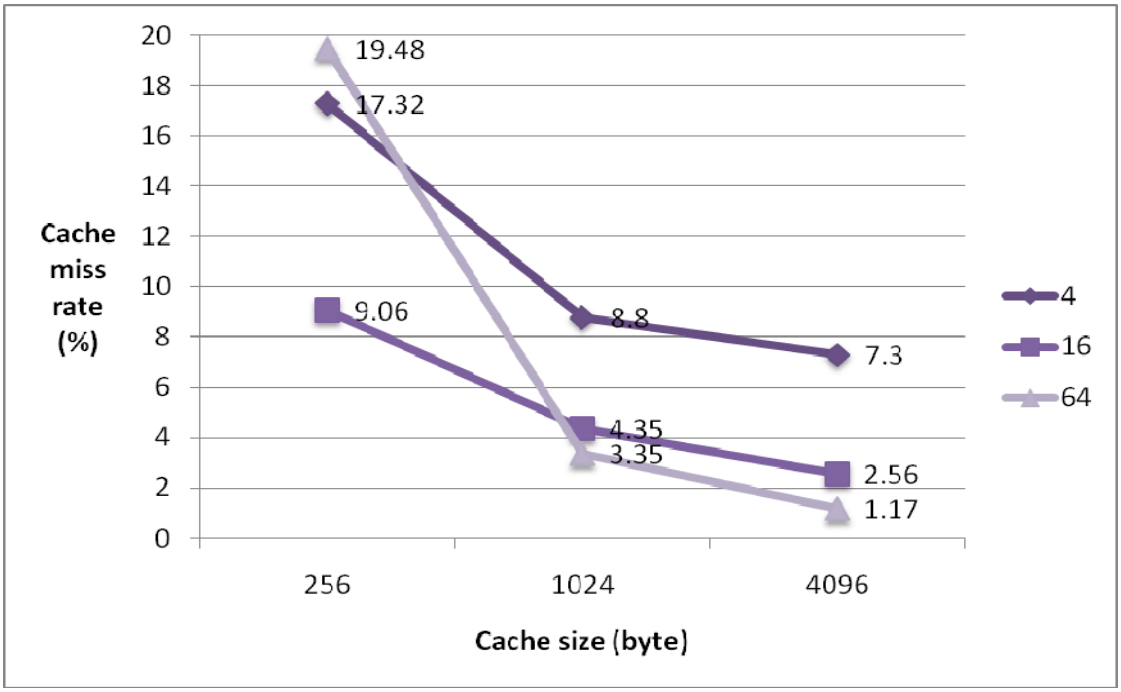


圖 4.5 將表 4.1繪製成折線圖

4.1.3 Summary of texture cache

由圖 4.5我們可以看到，在cache大小為 1K或者是 4K bytes時，block size為 64 個word都有最小的cache miss rate，然後緊追在後的則是block size為 16 個words。可是採用 64 個word的cache模型其miss penalty理論上至少是採用 16 個word的兩倍。因此，在大部分的情況下會比較建議採用 16 個word當成cache的block size。另外，在考慮嵌入式系統可能無法負擔太大硬體設計的情況下，我們決定使用大小為 1K bytes的cache做為我們的texture cache。表 4.2就是我們最後決定的texture cache模型。

表 4.2 最後定案的 texture cache 規格

Cache policy	Direct mapped
Cache size	1k bytes
Cache organization	16 block * 16 word * 4 bytes

4.2 Early depth test

就OpenGL規格書來看，深度測試(Depth test、Z-test)理論上必須要在per-fragment operation階段的透明度測試(Alpha test)之後才進行，如圖 4.6左圖，不過per-fragment operation已經是整個rasterization管線中的後期階段。就理論上來說，一個fragment最後會不會被frame buffer裡面的像素所遮蔽、蓋住，應該是在內插完深度之後就可以知道的事情。要是我們可以提早進行深度測試，在texture mapping前做完深度測試，這樣就可以避免讓一些本應被丟棄的fragment進入貼圖階段，減少texture fetch的次數，進而減少記憶體的存取量。而這就是早期深度測試(early depth test)的主要概念。而這樣的做法在許多的GPU中其實都有類似的概念[24][25][26]。

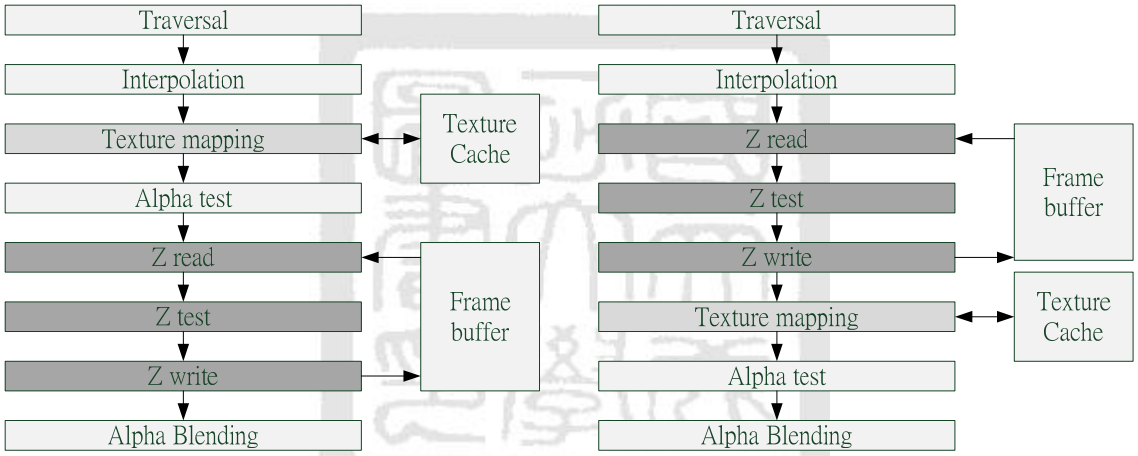


圖 4.6 傳統管線跟早期深度測試理論後的管線

要執行這樣的動作，最簡單的想法當然就是把depth test直接提早到texture mapping之前進行。如圖 4.6右圖。

4.2.1 Alpha test with early depth test

但實際上，OpenGL將深度測試放在透明度測試之後是理所當然的。由於深度測試完之後，成功通過深度測試的fragment其深度值將會被寫回depth buffer，也就是會被記錄下來，如圖 4.6中的Z-write就是把資料寫回depth buffer。所以任何決定

fragment存在與否的測試都必須在會被記錄下來之前完成，也就是必須在Z-write之前完成。其中又以alpha test最具代表性。以下將詳細解釋將深度測試置於透明度測試所引發的後果。

透明度測試在背景介紹裡只是負責將擁有比使用者指定 alpha 值還小的 fragment 給丟掉，實際上這個測試最常用在不規則貼圖，也就是帶有透明度的貼圖(transparency texture)之中，將我們需要的特殊形狀物件給「雕塑」出來。下面這個貼樹葉材質的例子將解釋如何利用 alpha test 做出不規則貼圖。

通常我們為了讓即時繪圖(real time rendering)成為可能，會盡量避面在即時繪圖系統上面使用高精細的模型，取而代之的是使用比較粗糙的模型，再貼上已經先在高精密系統算好的貼圖。以畫樹葉為例，大部分的即時繪圖系統(3D遊戲最為常見)不會一片一片的樹葉去畫。因為這樣每畫一片樹葉，就要去處理一個到數個三角形，畫一棵茂密的大樹恐怕就拖垮整個繪圖系統的速度。最簡單的想法就是直接畫一塊大的正方形，然後在這一塊大的正方形上貼上一張含有一堆樹葉的貼圖，如圖 4.7。

圖 4.7 在一塊正方形上貼上一張帶有大量樹葉的材質貼圖

這樣貼好的物件在空白的地方並不是透明的，實際上在非葉子地方的fragment是我們想要拋棄的。所以當我們在預先處理這張貼圖的時候，會先在空白地方讓alpha值等於 0，然後等到進行alpha test的時候，讓程式設計者指定alpha 值低於 0.5的fragment都必須被捨棄。這樣我們在貼圖完畢，進行完透明度測試的時候，這些fragment就會被捨棄，而不會把frame buffer裡面pixel給蓋掉。如圖 4.8。

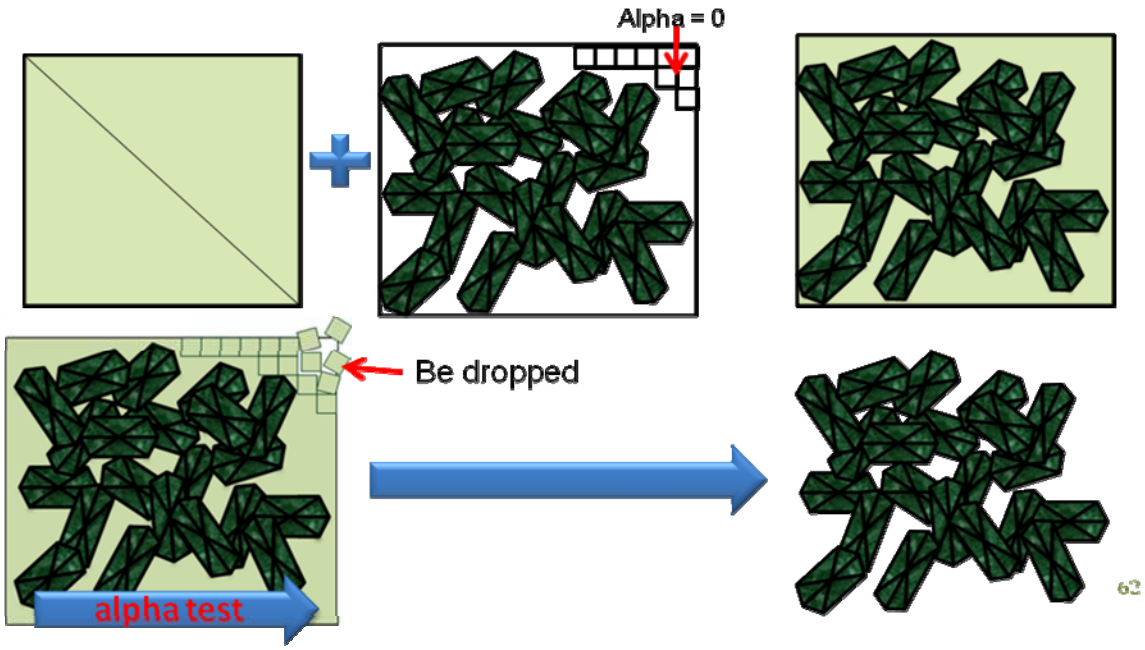


圖 4.8 使用 alpha test 將透明的地方捨棄

現在我們來看看使用early depth test對這樣帶有透明度的貼圖會發生甚麼事。由於depth test在alpha test前先做了，所以那些應該被alpha test踢掉的fragment在depth test的時候已經先更新了depth buffer，如果在這之後我們還要畫一個背景在這棵樹之後，在樹葉空白的地方由於先占據了depth buffer，背景的地方在進行depth test就會失敗而不會寫上去，最後造成的結果就會如圖 4.9左半部分一樣。而正常來講應該要像圖 4.9右邊能夠看到背景才是正確的。

圖 4.9 使用 early depth test 對 transparency texture 的影響

所以我們不能把深度測試放到透明度測試之前。實際上，精確的說法是，我們不能將Z-write放到alpha test之前，我們只要把Z-write跟Z-test分離並放到透明度測試之後，如圖 4.10的右圖。這樣的話，即使fragment通過了深度測試，也會在寫回depth buffer之前就被透明度測試給踢掉。這樣我們的帶有透明度的貼圖就不會出現問題，同時也可以提早進行深度測試。

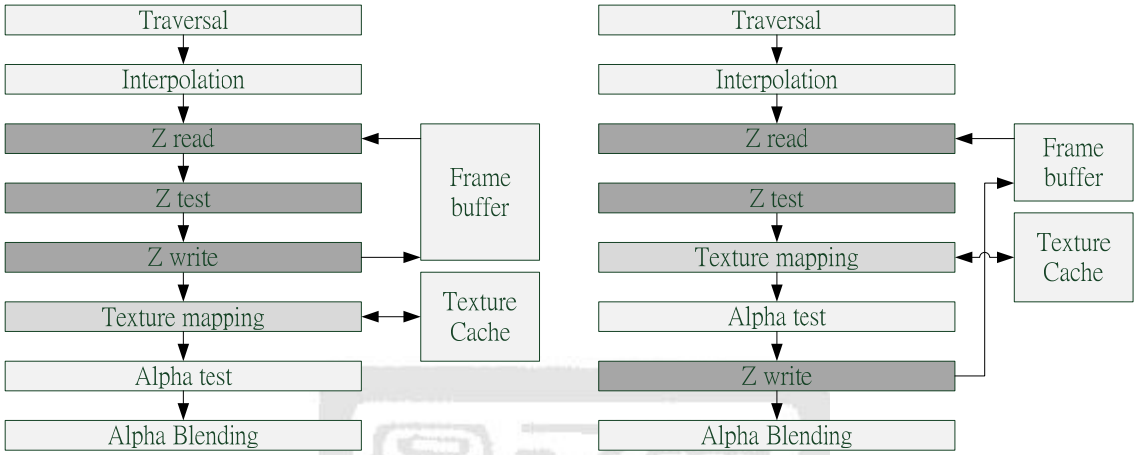


圖 4.10 之前所想像的早期深度測試和分離式早期深度測試

4.2.2 Data hazard in separated early depth test

不過採用分離式的深度測試，也就是像圖 4.10右圖中將深度項目測試和更新depth buffer這兩個動作分開，如果此硬體是採用pipeline架構的話，很明顯的就會產生一個問題：Data hazard。

讓我們已圖 4.11為例。再圖 4.11中，有三個fragments的平面座標皆相同，且紅色的深度值是 5、綠色是 10、而藍色為 15。在OpenGL的座標系統規範中，比較近的物件所擁有的深度值較低，而比較遠的物件則較高。所以應該是紅色在前面，綠色在中間，而藍色在最下面，如圖 4.11的左下角三個fragment相疊的樣子，最後畫出來的會成為該pixel顏色的理論上要是紅色。但這三個fragment進入管線的順序依序為藍、紅、綠，藍色這個fragment很早就進入了管線並且寫到frame buffer裡面。由於採用分離式的早期深度測試，當綠色開始進入管線中要做Z-read、也就是存取depth buffer時，紅色還沒將自己的深度值寫回depth buffer，導致depth buffer中讀起

來的是藍色的深度值，進而使綠色通過深度測試，最終將自己的深度值寫回depth buffer時覆寫掉紅色的深度值。這就是在pipeline中會出現的典型data hazard。

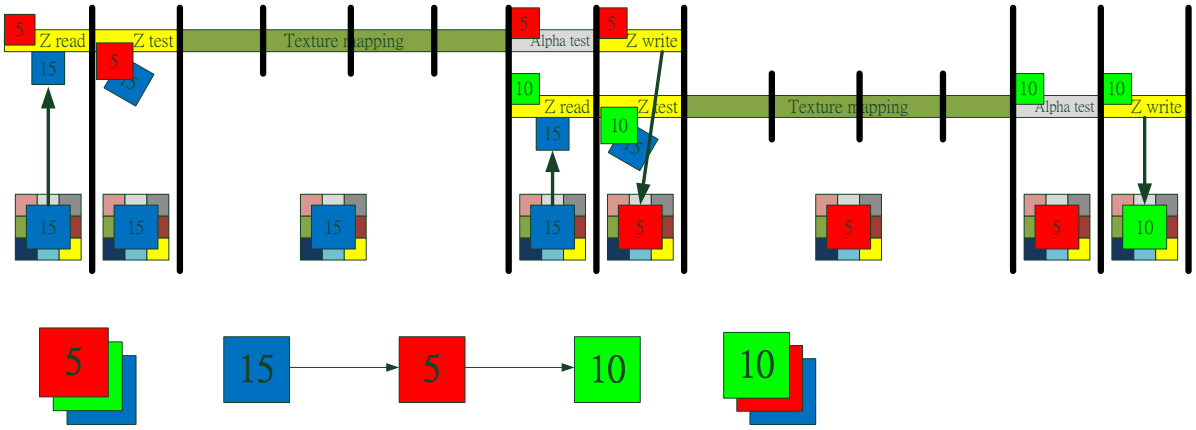


圖 4.11 分離式早期深度測試所引發的 data hazard

在CPU中，由於管線化的關係，也有data hazard的問題。而解決的方式就是使用hazard detection unit並加入data forwarding跟pipeline stall的機制，這部分在D.A.Patterson等人所著的Computer Organization and Design [27]此書中有詳細的介紹。通常CPU的管線深度不高，且大量需要由CPU處理的branch指令也可以由此機制來判斷，故加入hazard detection對CPU來說是相當合理且重要，所增加的硬體面積也不會太多。但GPU的管線相當的深，texture mapping的部分就至少 5 級，加上Alpha test還會更多，而且GPU之所以管線會這麼深的原因一部分就是因為很少有data dependency的問題。今天會出現data hazard還是為了減少外部記憶體存取量才產生的，而不是本來就有這樣的問題。因此，在這邊加入hazard detection這樣的解決方法雖然可行、但並不是那麼的符合常理。所以，下面將介紹另外一種解決方法，multi Z test。

4.2.3 Multi Z test

前面所提到將Z-test跟Z-write分離的早期深度測試會產生data hazard，其實主要引起此問題點的地方，就在於Z-write進行時已經有更新的fragment必須要被考慮進

來。因此我們就在Z-write進行的前面在進行一次Z test，使這一級成為完整的深度測試。如圖 4.12右邊的圖形所示。

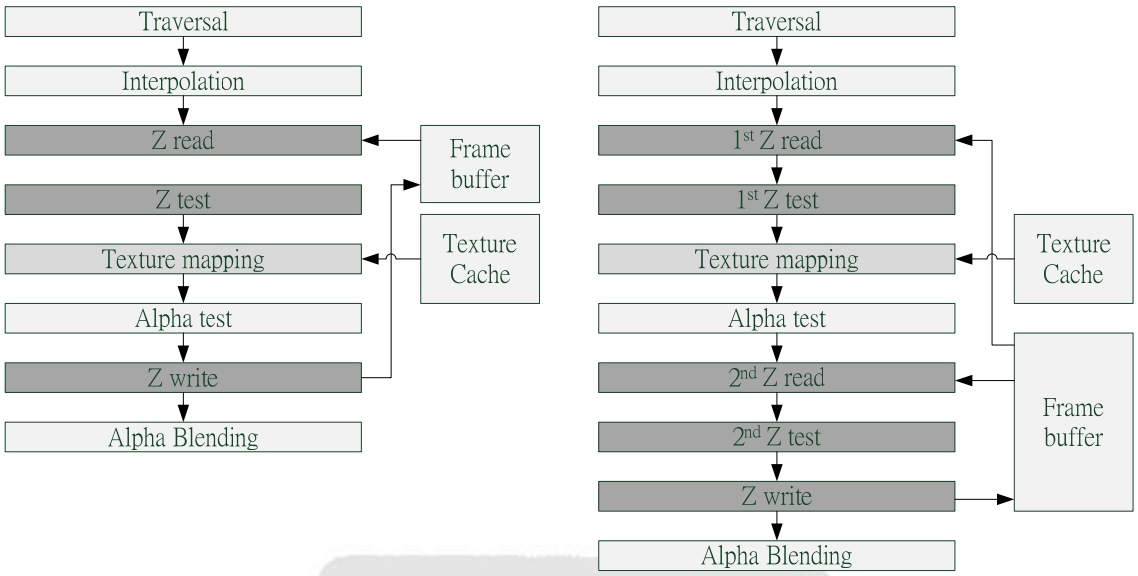


圖 4.12 分離式早期深度測試跟多重深度測試的示意圖

現在，我們必須進行兩次深度測試。第一次深度測試發生在 Texture mapping 之前，做完測試之後不會將結果寫回 frame buffer。而第二次深度測試則維持在 OpenGL 中原來定義的地方，且會將測試完的結果寫回 frame buffer，這樣就可以避免 data hazard 的發生，也可以達到 early depth test 的目的。

另外，增加一次 depth test 所付出的代價其實不大，在演算法層級只多了一個比較器。以硬體的眼光來看的話，則是多出了一到二級的 pipeline 暫存器與多一次 frame buffer 的 memory fetch。值得高興得是，在 tile based 的架構下，並不會進行外部記憶體的存取，而是存取硬體內的 on-chip tile buffer。唯一需要的改變的就是在此 tile buffer 多加上一個 read port，從原本一個 read port 一個 write port 變成兩個 read ports 一個 write port。

4.2.4 Simulation and summery of early depth test

接著我們必須對早期深度測試進行模擬以確定它的效能。由於早期深度測試主要是針對減少貼圖的存取，所以我們將以帶有貼圖的模型進行測試。我們在這邊所測試的模型依然是圖 4.4的貼圖模型。另外，我們在測試的時候，由於已經帶有一個 4.1 節中所提到的texture cache，所以主要模擬觀測目標為texture cache的hit數與miss數。

表 4.3 開啟早期深度測試對 Texture cache 所造成的影響

Early depth test	Texture cache miss	Texture cache hit
Off	21486	234862
On	15787	173673
Reduction ratio	26.5%	26.1%

基本上，影響最大的部分在於texture cache miss的時候，由於texture cache miss時要去外部記憶體讀取貼圖資料，直接減少texture cache miss就等於減少讀取記憶體的次數。在表 4.3我們可以看到打開早期深度測試可以減少約 26.5%的texture cache miss的次數，這對於texture mapping甚至是整個系統而言是一個相當大的影響。

早期深度測試對於一個 pipeline 架構來說，並不會因為早點踢掉不必要的 fragment 而加快 pipeline 的進行。因為 fragment 一旦進入管線，就算之後沒有通過任何一樣測試而被剔除，也會在管線中留下空的一級、稱為泡沫(bubble)。早期深度測試主要的好處是可以大量避免不必要的貼圖存取，就硬體上有另外一個好處，空的泡沫通過管線時不會進行運算，所以可以進一步節省一點電力消耗。最後，我們在這邊採用 multi Z test，因其在達成早期深度測試的同時，不用增加太多的硬體設計，同時跟分離式早期深度測試比起來，也不會造成 data hazard 的出現。

4.3 RM Architecture summery

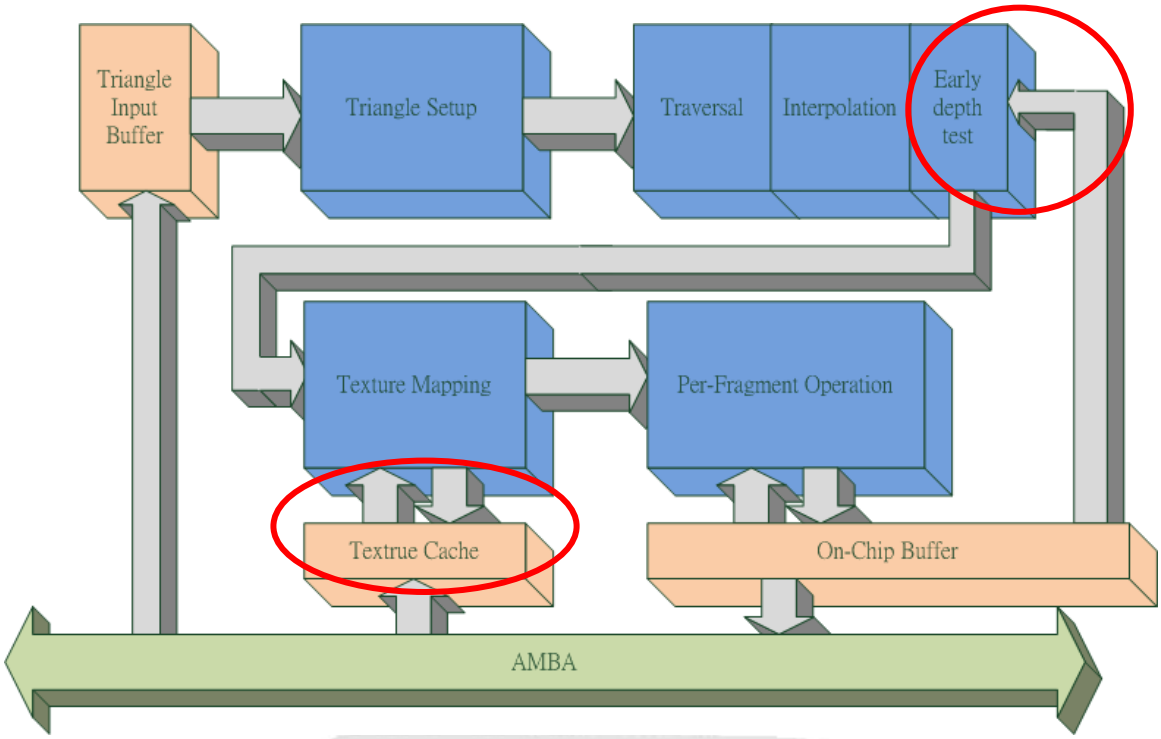


圖 4.13 加入早期深度測試及 texture cache 後架構圖的改變

在本章中，我們介紹了兩種影響整體效能的架構，分別是texture cache及早期深度測試。在texture cache裡，我們使用 6D block cache的架構，可使得texture cache的hit rate在大部分的情況達到 90%以上，依照數據測出為 95.65%。而在早期深度測試之中，我們使用multi Z test來進行，可以有效減少 20%以上外部記憶體存取量。加入這兩項特色後，整體RM架構的改變如圖 4.13。

第5章 RTL implementation

在RTL的部分，幾本上是完全依照定點數(fixed-point)軟體模型的部分來寫成，不過暫時沒有加進Texture unit。也就是實際上有Triangle setup、Rasterizer，和Per-fragment Operation三大部分，另外內含一塊On-chip buffer當作tile buffer，整體架構如圖 5.1所示。數學和邏輯運算單元全部使用Synopsys這間公司所出的DesignWare IP [28]來進行運算。對外則有一個Master wrapper和Slave wrapper負責和AMBA溝通。

另外，在合成gate level model上，我們使用TSMC 0.18um的cell library，並以Synopsys的Design compiler合成器[29]進行合成。合成結果將在各子模組之中跟本章的最後一節理說明。

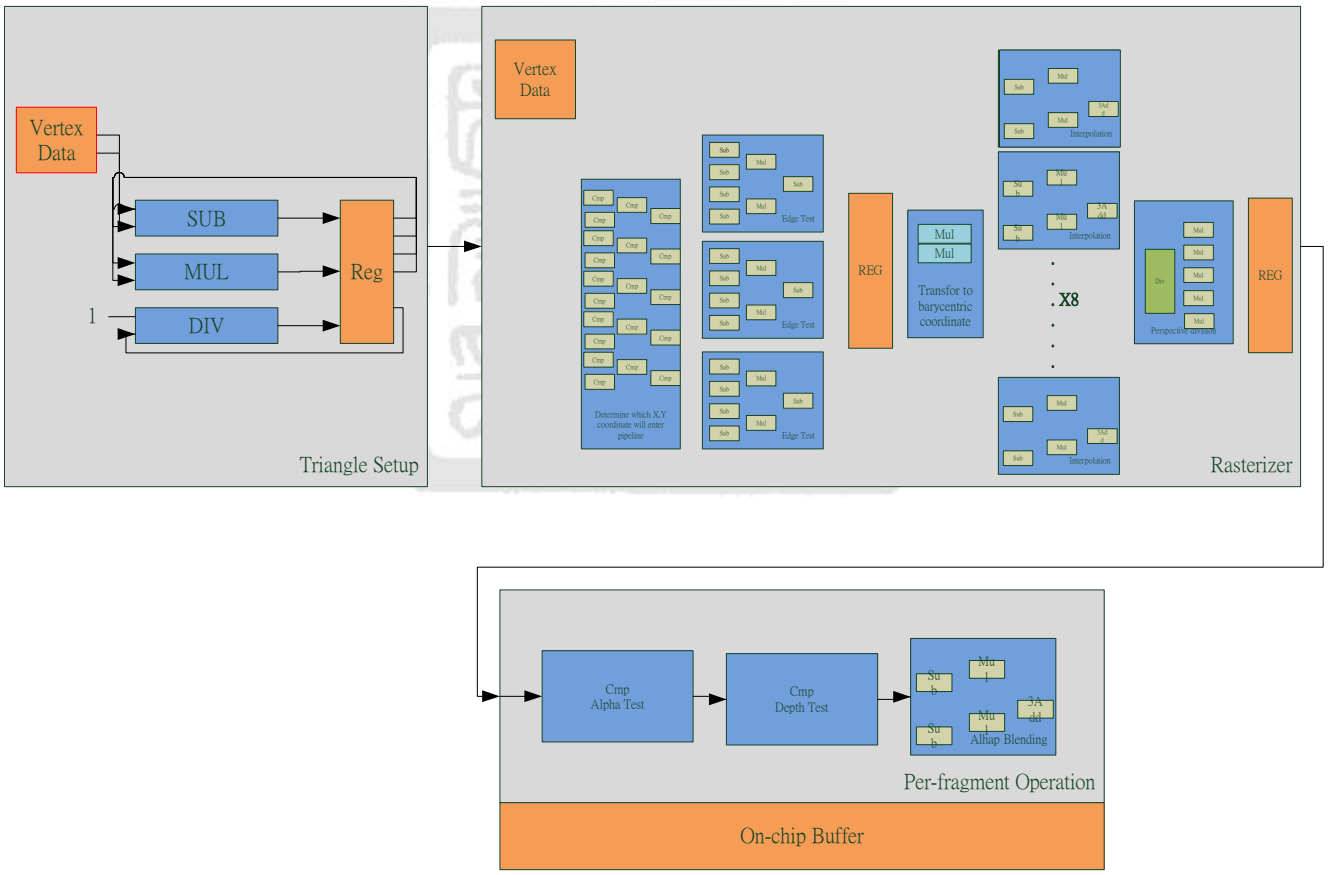


圖 5.1 整體 Rasterization 的元件方塊圖

5.1 Triangle setup

這一級所負責的事情在第三章裡並沒有詳述，只是把它歸類在traversal裡。其實Triangle setup是在進行三角形traversal之前，先把一個三角形中會重複使用到的資料算出來並預先儲存。其中包含了三角形三組直線方程式的係數跟一個兩倍三角形面積的倒數。如表 5.1。

表 5.1 Triangle Setup 所需要計算的參數

Notation	Description
$a_i, b_i, i \in [0,1,2]$	Edge function coefficient
$\frac{1}{2A_\Delta}$	Half reciprocal triangle area

Triangle setup 的硬體設計，由於這些參數在一個三角形中只需要計算一次，並沒有必要設計成管線化的架構，另外考量此階段的運算一定會比後面各級來的快上許多，所以採用 multi-cycle 的設計。此硬體含有一個加法器、一個除法器、和一個乘法器，經合成後花費 14633 gate count。區塊圖如下所示。

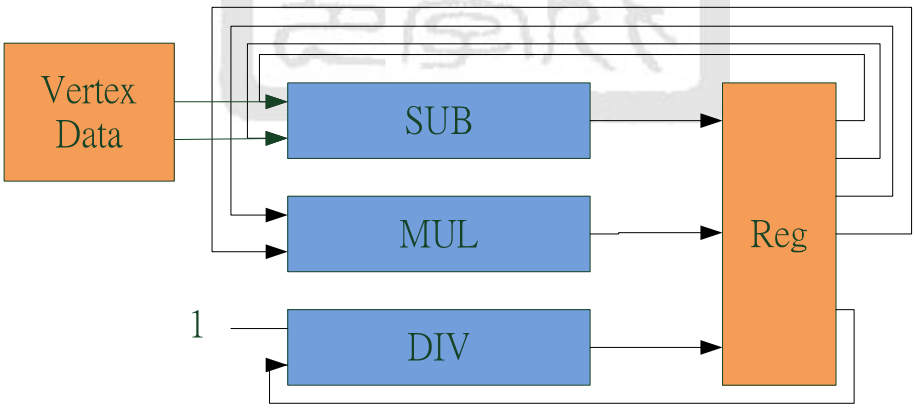


圖 5.2 Triangle Setup 的元件方塊圖

每一個三角形進入此硬體之後，將需要 12 個cycle才能計算完畢，圖 5.3在不同cycle數中執行運算的波形示意圖。

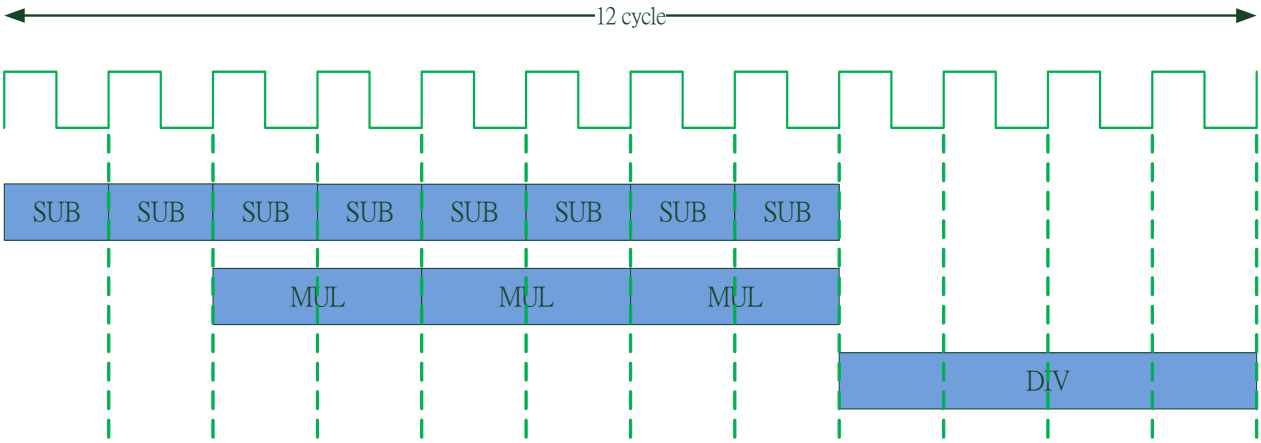


圖 5.3 Triangle Setup 的波型示意圖

Triangle Setup 硬體的效能，以每個 cycle 花費 5ns 來計算的話，其 throughput 將可達 16M triangle/sec。

5.2 Rasterizer

Rasterizer 和 Per-fragment operation 這兩塊模組都是以 pipeline 架構展現，由於暫時沒有加入 texture unit，所以這兩個模組的管線在銜接上面將是連續且沒有中斷的。

在Rasterizer方面，又可以細分成traversal和interpolation這兩個部分。Traversal 負責掃描三角形，而interpolation負責內插fragment的資料，這部分的演算法已於 3.1 跟 3.2 節之中探討說明。而整體包含Rasterizer和Per-fragment operation這兩塊模組的硬體元件圖如圖 5.4所示。

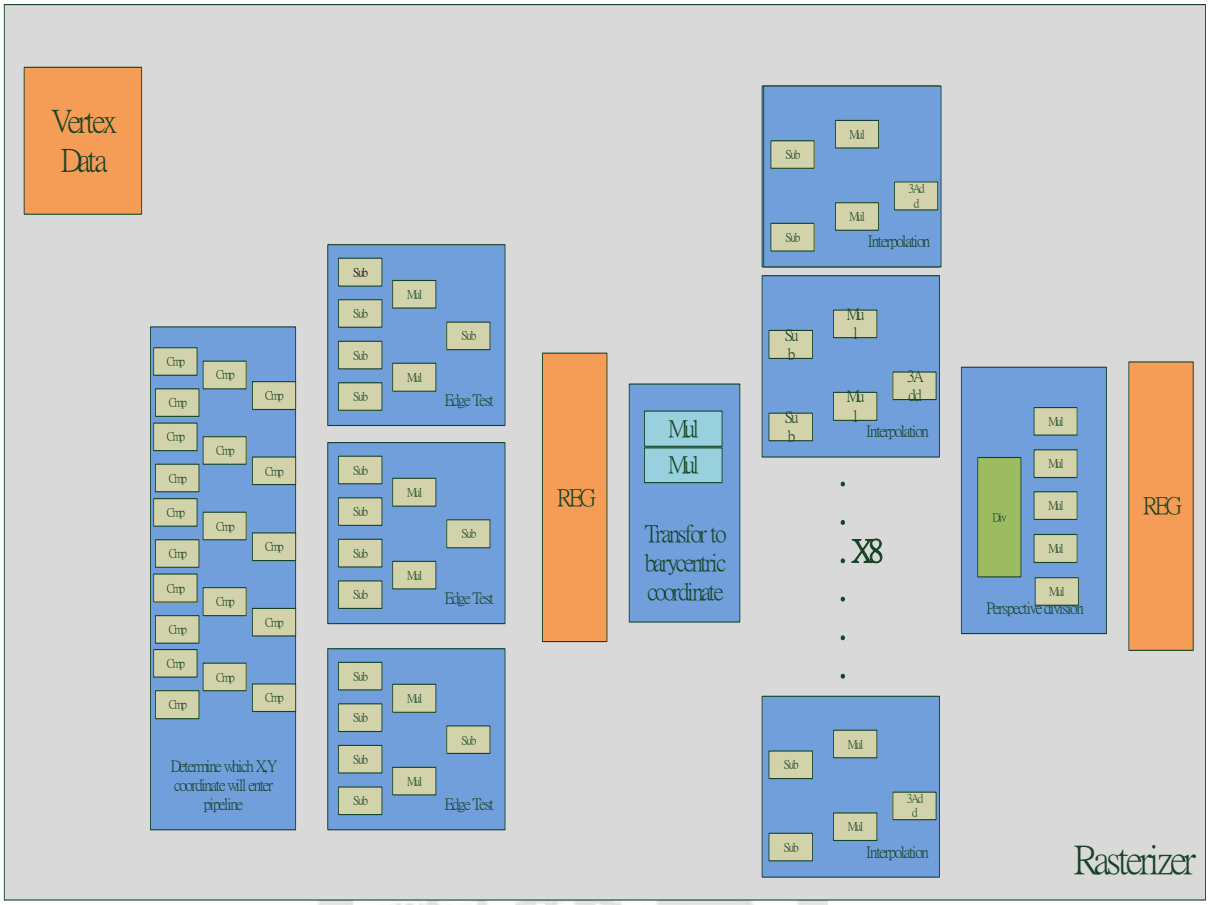


圖 5.4 Rasterizer 的元件方塊圖

圖 5.5顯示traversal的管線化元件架構圖，每一個像素一旦進入管線進行測試，無論如何就會占用掉一級的管線。如果像素有效，其資料就會進入管線暫存器並移到下一級的硬體繼續運算；如果無效，則管線暫存器就不會紀錄無效的像素資料，等於這一級不做任何運算，可是還是花費了一個cycle。所以，演算法的優劣在這邊將會有決定性的影響，演算法掃描出的有效像素比例較高的話，就可以減少花費cycle再無效的像素上。硬體設計上則沒有甚麼太大的技巧，這邊我們使用一些乘法器算出質心座標後，在用比較器來決定此像素是否為有效像素。經合成後花費 52260 gate count。

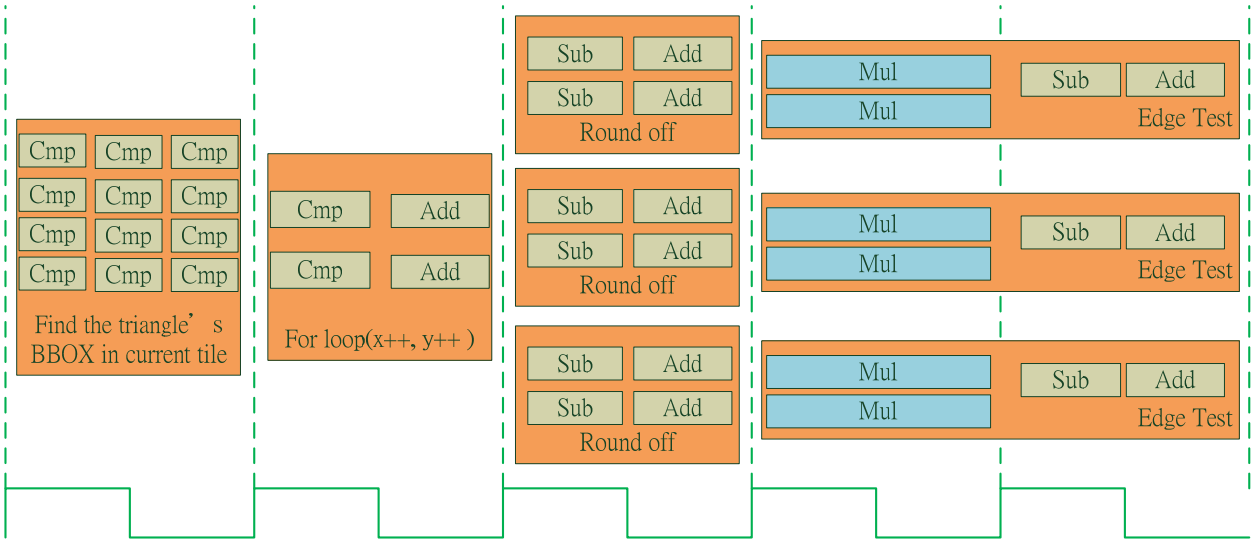


圖 5.5 Traversal 的管線化元件方塊圖

判斷為有效像素後，就要進行內差。由於每個像素需要計算的資料相當的多，需使用質心座標來內插深度、透式投影修正值倒數、包含 4 種 RGBA channel 的顏色，還有貼圖座標等資訊。然後再把上述剛內插出來的資料除以內插出來的透式投影修正值倒數(inverse W)進行透視投影修正。由於使用大量計算動作，而且一定要盡量加速這些動作的完成，在這邊我們無法使用 multi-cycle 的設計，一定要全部加以平行管線化處理，所以將會使用大量的運算元件在這些方面。目前一共使用了 29 個加減法器、24 個乘法器，還有 1 個除法器。經合成後共花費了 160660 gate count，是所有模組中花費最大的。

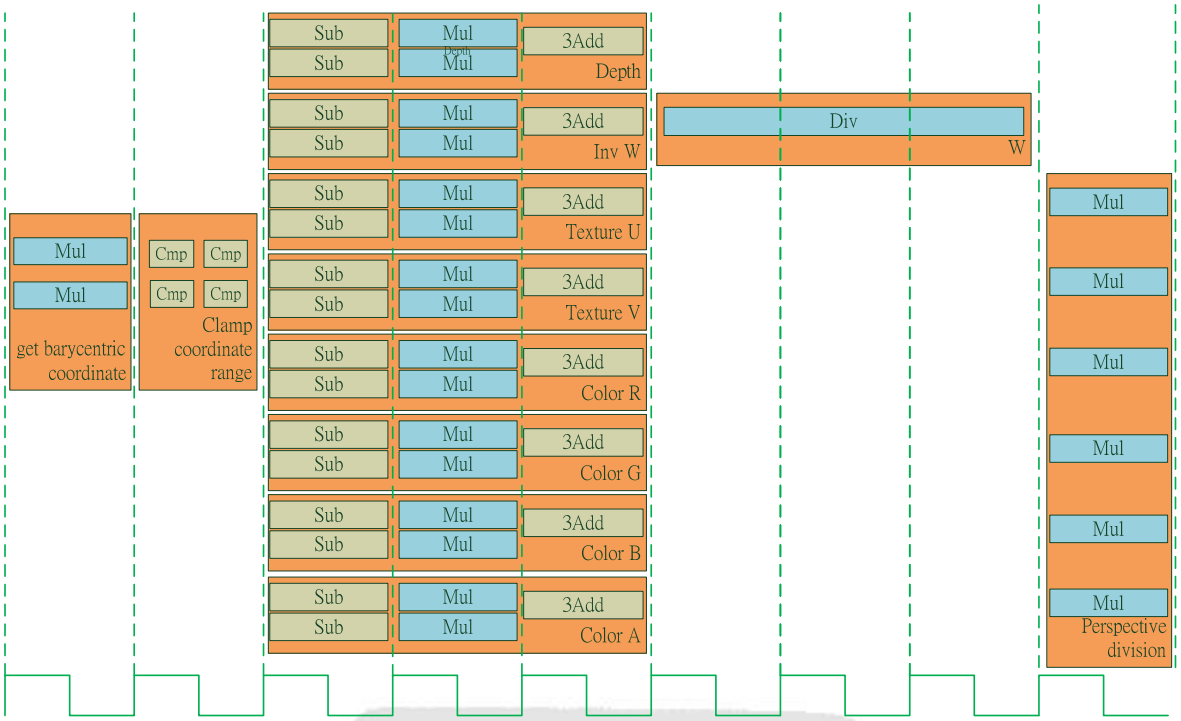


圖 5.6 Interpolation 的管線化元件方塊圖

5.3 Per-fragment operation

Per-fragment operation方面，在透明度測試(Alpha test)和深度測試(Depth test)中，只需使用到比較器來決定像素是否有通過測試，不過在半透明混色(Alpha blending)這個項目上就會需要乘法器和加法器來完成。其管線化元件方塊圖如圖5.7所示。經合成後花費 10318 gate count。

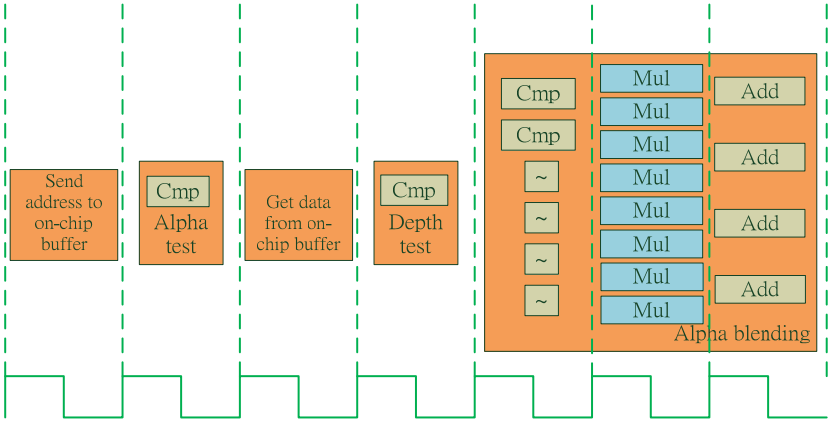


圖 5.7 Per-fragment operation 的管線化元件方塊圖

5.4 Summary of RTL implementation

表 5.2 整個 Rasterization 各子模組跟整體模組所花費的 gate count 數

Component	Gate count
Triangle Setup Engine	14633
Traversal	52260
Interpolation	160660
Per-fragment Operation	10318
Controller	15637
Entire RM	253508

我們將各子模組所花費的gate count列於表 5.2之中。其中我們可以看到，在整個Rasterization裡面，內插fragment的資料部分花費了最多的面積。這是由於它使用了大量的運算元件所造成的。最後整個Rasterization硬體模組約花費了 254K gate count。

Triangle setup 的 throughput 固定為 16M triangle/sec。而 Pixel generator & Per-fragment operation 的效能幾本上是取決於有效像素掃描的演算法。如果演算法可以大量避免無效的像素進入硬體管線中測試其有效與否，就可以增加管線的利用率。在這邊我們以前面演算法所做出的數據取其平均，可得知大概會有 45%的有效掃描，也就是每個像素平均要花 2.25 個 cycle 來進行運算，throughput 大約等於 90M pixel/sec。這個是由理論上推估而得到的數據，實際上還須看整個系統才能得知最終的效能。

我們在進行模擬的時候都有詳細的記下每一個 benchmark 所花費的 cycle 數，其結果跟分析將留至第六章再談。

第6章 Simulation and verification

本章將介紹我們模擬跟驗證的結果。所以一開始會先介紹我們實做出的數種 rasterization 模型，也會介紹模擬的平台與使用的工具。接著介紹驗證的方法。最後列出實驗結果並進行分析，探討結果與架構之間的關係。

6.1 Simulation platform

以下先介紹我們模擬的平台。我們在開發GPU時，一共使用了四種模擬用的模型，前兩種屬於軟體版本，後兩者則屬於硬體版本。在軟體部分的模擬系統，就是一個單純的C++ model simulation。GE跟RE都做成class來讓外層的API進行function call，如圖 6.1左邊software development那一區塊。而硬體版本的模擬則相對複雜得多，如圖 6.1右邊的全系統模擬平台。此模擬平台將在 6.1.2 在進行詳細的說明。

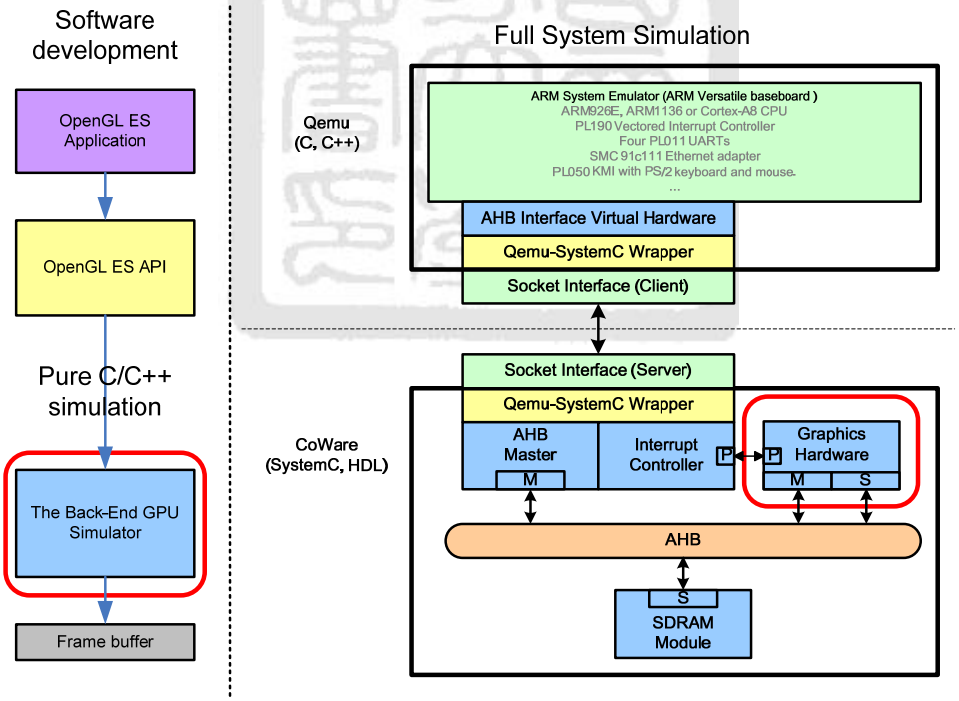


圖 6.1 System simulation framework

6.1.1 Simulation model

在設計 rasterization 的過程中，我們一共完成了三種不同層級的 rasterization 模型，分別是 floating-point model、fixed-point model、和 RTL model。在 RTL 模型中，經過合成處理之後，還可以產生一個 gate level model。底下將分別說明這四種模型之作用。

- Floating-point model

此版本為使用 C++所撰寫而成的浮點數版本，主要的用途是驗證概念上的 rasterization 繪圖流程是否可以運作，還有各個子模組、OpenGL ES 要求的功能等是否有正確設計，演算法層級的功能大部分都是在這個模型上進行模擬。此浮點數 C 模型還有另一個很重要的用途，用來學習瞭解 OpenGL 複雜的功能與流程。

- Fixed-point model

在硬體上來說，浮點數可以表現的數值範圍比定點數大的多，但在運算時需要的額外的硬體幫助計算。由於在 rasterization 階段時，基於 pixel 的概念，大部分的情況不需要很精確的小數，故可以定點數進行硬體設計，以節省硬體空間。也因此，開發一個定點數版本的模擬環境就顯得相對重要。

定點數模型依然為一使用 C++所撰寫而成的軟體版本。不同之處就在於，外部傳進來的值雖仍是使用浮點格式，但在軟體模型裡面則必須自行轉成定點數格式。像是一個 32 bits 的資料，前面 16 bits 為整數，後面 16 bits 為小數，這種固定小數點的格式。定點數模型除了可以決定運算使用的精確度以外，也可以幫我們找出一些原本在浮點數版本中可運作但換到定點數後就容易出錯的演算法，如質心座標法就是因為原本的平面方程式演算法在定點數中容易出現溢位而使用的。

基本上，這一個定點數版本很像是 behavior RTL model。而且，在 C++裡面模擬定點計算其實沒有在 RTL 中模擬來的方便。但是在 RTL 中開發，由於資料流型態的改變，必須要連外圍連結 GM 與 RM 的軟體 API 一起重寫。而且 RTL 模擬沒有 C 來的快速。所以當我們完成浮點數的軟體模型之後，為了快速開發，我們選擇先在 C++中開發定點數版本，而不是開發 behavior 的 RTL 版本。

- RTL model

由於已經開發了定點數的軟體版本，所以我們在開發 RTL 版本的時候，就是直接以開發帶有完整 pipeline 硬體架構且能夠合成的 RTL 模型為主。此模型可以讓我們得知整體硬體的效能。在開發此模組時，有兩個明顯困難。第一個就是 timing 資訊，這部分要不斷的透過合成 Gate level model 來取得 timing 的正確與否。第二個就是驗證問題，由於缺乏像 C++那樣，可以簡單的跟軟體 API 等進行溝通的方法，這邊需要建立全系統的驗證平台才能進行除錯與軟硬體共同驗證。這一部份在下一節講述軟硬體驗證的時候會有比較詳細的描述。

- Gate level model

Gate level model 為 RTL model 經過合成軟體合成之後，帶有 timing 資訊與硬體面積的模型。為最能反映真實硬體情況的模型。不過模擬起來曠日廢時，基本上除了拿來取得 timing 與 area 資訊外，只有驗證簡單的多邊形。

6.1.2 Full system simulation platform for RTL model

傳統的 RTL 模擬上，由於不能直接讓外部軟體呼叫，所以都要使用 test bench 模擬軟體的行為來送給 RTL simulator 進行硬體模擬。但由於我們所開發的系統相當龐大，且整體 GPU 被分成兩部分來獨立開發。我們勢必沒有辦法只靠 test bench 來進行完整的模擬，這邊就需要有全系統的模擬平台來幫助我們進行模擬。

■ CoWare Platform Architecture

我們必須先介紹一個由CoWare公司所開發的Platform Architecture軟體[30]。此軟體為一套整合型開發環境系統，可以讓工程師在這個平台上以SystmeC寫成的IP模組、或者是將已完成的RTL模組，跟整個系統做模擬驗證或是系統效能分析等動作。使用者可以購買他們所出的ARM開發套件，其中包括ARM CPU，AMBA bus等元件。可以讓使用者快速的建構出一完整的系統。之後使用者可以把自己開發的硬體掛上bus，並將自己用C寫成的外部軟體、driver、甚至是一個完整的作業系統，透過ARM C compiler編譯成ARM CPU可以執行的程式，送進系統中進行軟硬體協同模擬驗證。

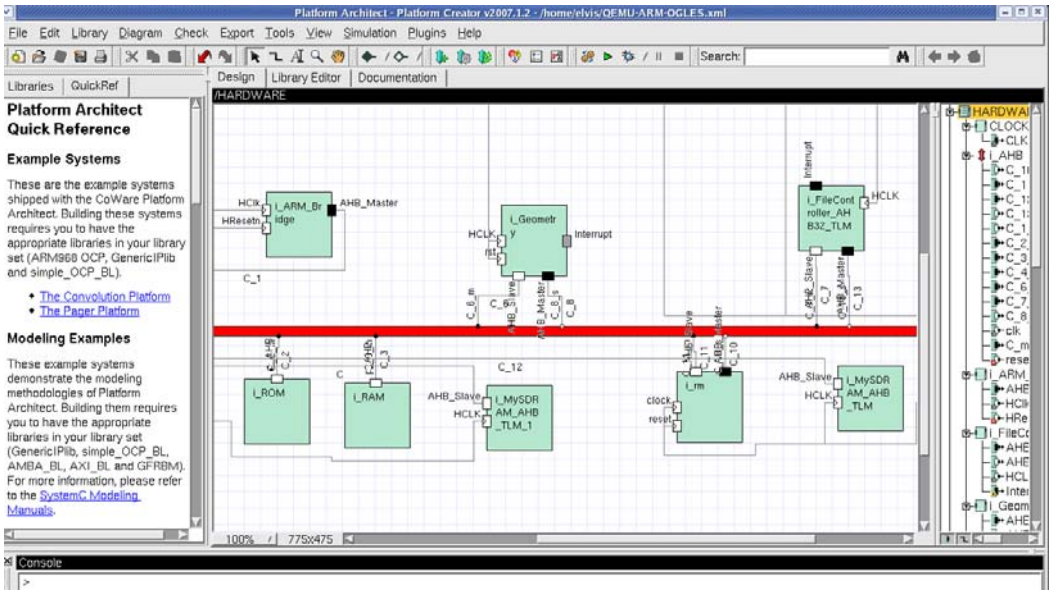


圖 6.2 CoWare Platform Architecture 的執行畫面

這對大型的開發計劃而言是一個相當實用的設計平台。軟體工程師或是韌體工程師可以在硬體只完成初階設計之後就開始執行他們軟體或者是驅動程式的設計，而不用等到硬體成品出來才開始設計。我們的 RTL 模型通通都是在此平台下跟上層的 API 進行模擬驗證。

■ QEMU

雖然 CoWare Platform Architecture 是如此的方便，擁有的 tool chain 是如此的完整。可是卻擁有一個大問題，就是在執行 ARM C 程式的時候，由於整個程式是真的要經過模擬上的系統交由模擬中的 CPU 執行，其執行的 cycle 跟所有的交換資訊都必須被精確的記錄下來，導致其執行上的效率頗為低落。光是執行 printf 這個在 C 語言中簡單的列印函式，都會感覺的有一點點的延遲，更不用說 boot 一個作業系統了。根據我們實驗室在此平台上 boot Debian Linux OS 所留下的紀錄，至少花費了 40 分鐘以上才完成。

解決的方法就是採用QEMU[31]。QEMU是一個可以模擬x86、x86_64、ARM926、SPARC、PowerPC等等處理器的開放原始碼軟體。對我們而言，就是一個可以模擬ARM CPU的虛擬機器(virtual machine)。在CoWare中，我們建立一個

ARM bridge去連結真實電腦中的某一個socket，來接收ARM的指令。而在QEMU之中，我們建立的一個網路裝置連接這個socket，並對這個網路裝置下達指令。整個平台簡單的示意圖如圖 6.3所示。簡單的講，我們用QEMU取代CoWare中的ARM CPU來對我們系統下達指令。

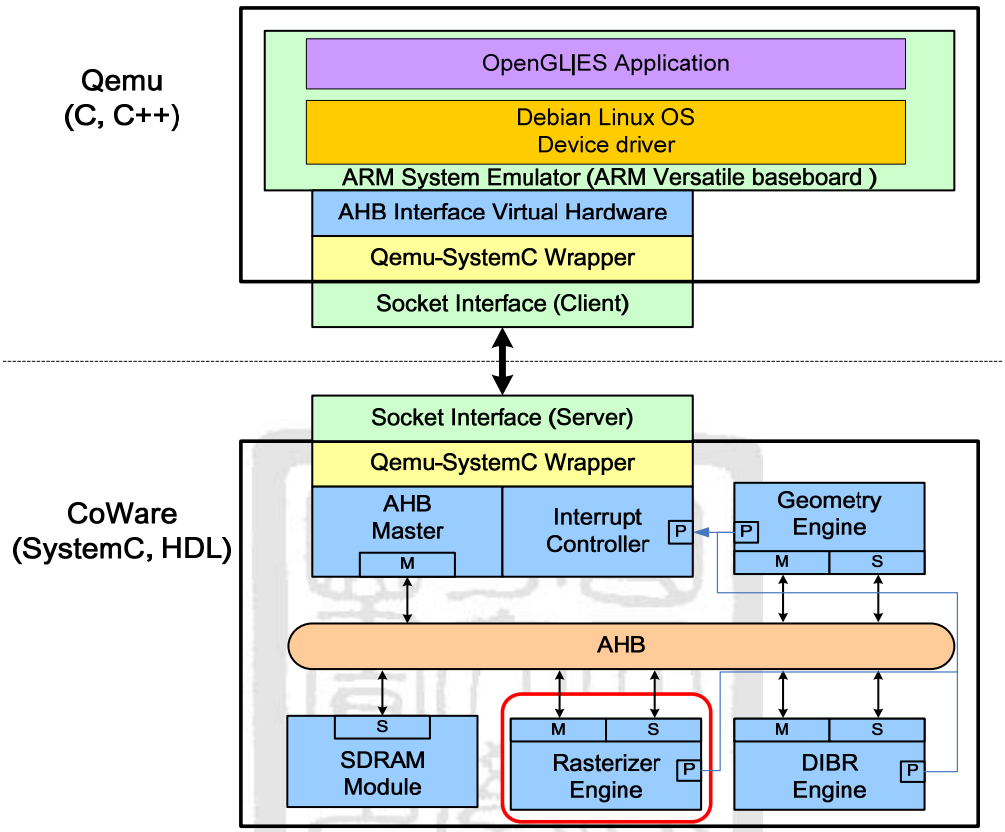


圖 6.3 QEMU 加 CoWare PA 平台的模擬示意圖，此平台同時也是我們最後的 full system simulation platform

通常一個虛擬機器的速度，依其核心程式碼跟 host OS 的不同，約有原機的 25%到 75%不等的效能。我們在 QEMU 上 boot 一個簡單的 Debian Linux OS 約只要 3 分鐘。跟原本在 CoWare 上有著天壤之別。而缺點就是喪失 boot 作業系統這一段的時序分析圖，不過我們是開發 GPU 不是 CPU，所以這一段對我們來說不是非常的重要。

我們的RTL模型就是在上面的這樣的全系統模擬驗證平台上進行模擬。此系統的各項規格我們列於表 6.1中。

表 6.1 我們 full system simulation platform 的規格

Host machine	Cent OS 4.7：Kernel 2.6.9
QEMU virtual machine (v0.9.1)	● ARM Versatile Platform Baseboard for ARM926EJS ● OS：Debian GNU/Linux 4.0 ● Kernel image 2.6.18-6 for versatile ● GCC 4.1.2
CoWare	● Platform Architect v2007.1.2 ● AHB master interface ● Interrupt controller ● Graphics hardware：GE & RE & DIBR
RTL simulator	● ModelSim 6.3a

6.2 Verification methodology

要進行驗證，首先就要有對照組。我們所製作的四種模型彼此都可以成為對方的對照組進行驗證。不過我們還需要一組他人的 implementation 來驗證我們自己開發的硬體是否有問題。一個最簡單的想法就是直接使用桌上型電腦的 3D 加速卡進行繪製，再將結果跟我們硬體畫出來的圖進行比較。不過由於桌上型電腦中，不論是作業系統還是各個 3D 加速卡開發廠商所開發的驅動程式，都是針對 OpenGL 來實做，而不是 OpenGL ES，所以沒辦法直接比較。這邊我們負責軟硬體協同驗證的同學，找到了一個在 PowerVR 產品中提供使用者開發嵌入式系統繪圖應用程式的 SDK(Standard Development Kit)。此 SDK 可以在 Windows 作業系統中執行 OpenGL ES 的應用程式，並且最終透過作業系統呼叫 3D 加速卡來繪製應用程式要求的項目。現在我們就可以跟桌上型的 3D 加速卡所繪製出來的結果進行比較。

現在，我們有了reference implementation、軟體演算法模擬平台、跟包含RTL

模型的全系統模擬驗證平台。我們就可以讓這些模型去繪製相同的OpenGL ES物件，並比較最後輸出圖像的差異。如圖 6.4在三個不同平台上面各自輸出圖像之後進行比較的例子。

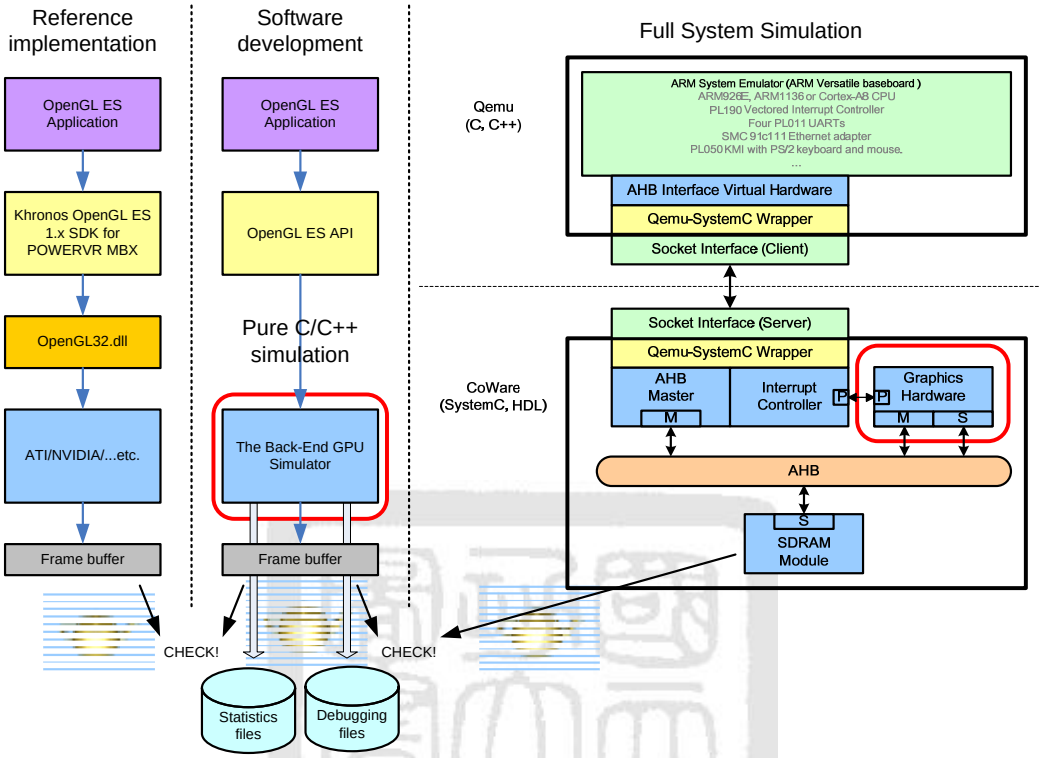


圖 6.4 Simulation and verification framework

在圖像比較上面，直接以肉眼比較可以快速的發現不一樣之處。可是當肉眼無法辨認出差異的時候，就需要軟體的輔助。在這邊我們取出各模型 frame buffer 輸出圖像中每一個像素的 RGB 數值，並互相比對差異，最後在將所有像素的差異值平均起來。

6.3 Simulation and verification result

6.3.1 Simulation result

各模組都擁有一個相同的模擬結果，那就是最後都能在我們自行實做的OpenGL ES API中繪製出肉眼看起來正確的圖形。除此之外，在RTL模型中的模擬

結果還可以進一步得到模擬過程中所花費的cycle數目，我把我們繪製不同物件所得到的cycle數結果列在表 6.2。而表 6.3則顯示出待測物件在各個階段之中所佔用的三角形數目。

表 6.2 待測物件在進入 RE 前各階段的三角形數量

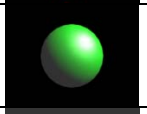
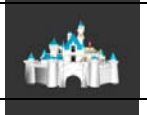
	Object	After GM	After tile divider	Image
Cube	12	6	133	
Sphere	1152	456	1550	
Teapot	6400	3007	5137	
Castle	13114	5534	11895	
Bunny	69451	28184	45288	

表 6.3 Rasterization繪製表 6.2物件所花費的cycle數(SRAM)

	Number of triangles (after tile divider)	Update frame buffer	Total cycles on RE rendering	Rendering cycles per triangle
Cube	133	922500	76861	578
Sphere	1550	922500	258455	167
Teapot	5137	922500	619579	121
Castle	11895	922500	1976330	166
bunny	45288	922500	4465867	99

首先讓我們看看表 6.3。一開始由API送進來的三角形個數列於第二欄之中。等到Geometry處理完之後，會將一部分用不到的三角形踢掉，故三角形數量會有所下降。但經過tile divider之後，由於當一個三角形被分類兩個、三個、或者更多個tile之中時，它就會被重複送進Rasterization管線中那麼多次，所以我們可以看到

在第一個物件，也就是立方體，在Geometry之後，原本只有 6 個三角形竟然被tile divider分出了 133 個三角形。原因就在於立方體一個三角形的面積相當大，橫跨了許多的tile，故有此結果。這就是tile-based architecture的缺點。我們在這邊並沒有要去探討這個問題，而是要說明我們在測各個物件花費的cycle數目時，三角形數量採用經過tile divider分類之後的結果。

在表 6.3中，列出了實際上在花費在繪製三角形的cycle數，跟從tile buffer寫回frame buffer所花費的cycle數。我們在圖 6.5將此表所列的資訊重新繪製成長條圖的樣子來呈現，可以讓我們清楚的比較在繪製不同的物件模型時，花費cycle數在此兩階段中的比例。

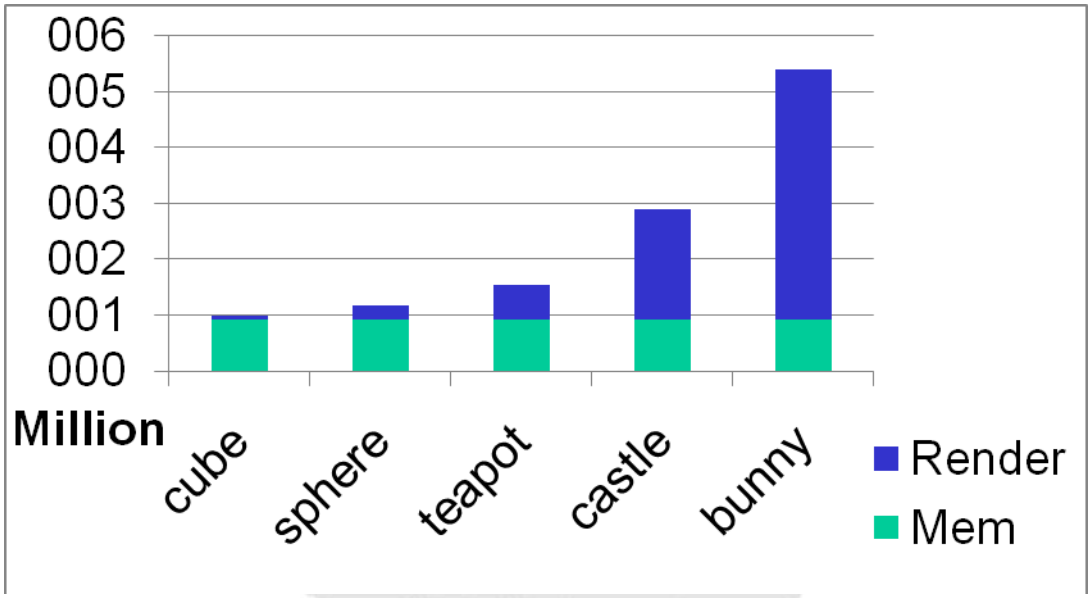


圖 6.5 將表 6.3的結果繪製成長條圖後的情況

從圖 6.5中可以看到，我們在寫回frame buffer中所花費的cycle數一直都是定值，這是由於理論上要跟frame buffer進行存取的動作都隱藏on-chip tile buffer之中，而真正要存取frame buffer的動作只剩下將on-chip tile buffer寫回frame buffer之中。這就是tile-based架構上做大的好處。

而當目標繪製物件的三角形數量較小的時候，花在寫回 frame buffer 的時間占了絕大多數，此時系統瓶頸將會是存取 frame buffer 所花費的時間，不過這不是我們目前想要探討的目標。因為畫這種簡單物件的時候，整體繪製速度都相當的快。

我們的 GPU 在全系統模擬驗證繪製像 cube 這樣的模型的時候 FPS 都約在每秒 6、70 個 frame。但是像畫 bunny 這樣精緻的物件時，FPS 約只剩下 7 到 8。而在這時，系統瓶頸變成是在處理三角形所花費的時間，如何縮短處理三角形的時間才是我們真正要去面對的問題。而在 5.4 有提到此處 RTL 的運算瓶頸為三角形 traversal 演算法的有效掃描比例，改善此掃描的演算法就是一個最直接的想法。

另外，在表 6.3還提供了一項資訊，就是平均每個三角形將花費幾個cycle在繪製上。基本上，一個三角形的面積越小，所花費的cycle數就越小；面積越大，所花費的cycle數就越多。但是像最後一個兔子物件：bunny，雖然每個三角形都很小，可是還是需要花費一定的cycle數在繪製上面。除了三角形traversal演算法的影響外，花費cycle在跑完pipeline的流程上面也對增加cycle數做出了不少貢獻。這邊找出一個控制三角形與三角形之間流程的schedule也是一個重要的議題。

6.3.2 Verification result

在 6.2 節中我們有提到驗證的方法，就是使用軟體去比較不同模型所畫出圖像的顏色差值。這部分經過模擬計算之後，得到的結果顯示在圖 6.7與圖 6.9之中。圖 6.7顯示的是reference implementation，也就是PowerVR，與我方浮點數版本GPU的平均像素差異。而圖 6.9顯示的則是我方浮點數版本與RTL版本GPU的像素差異比較。

圖 6.6 PowerVR 與我方浮點數版本 GPU 繪製出的茶壺模型

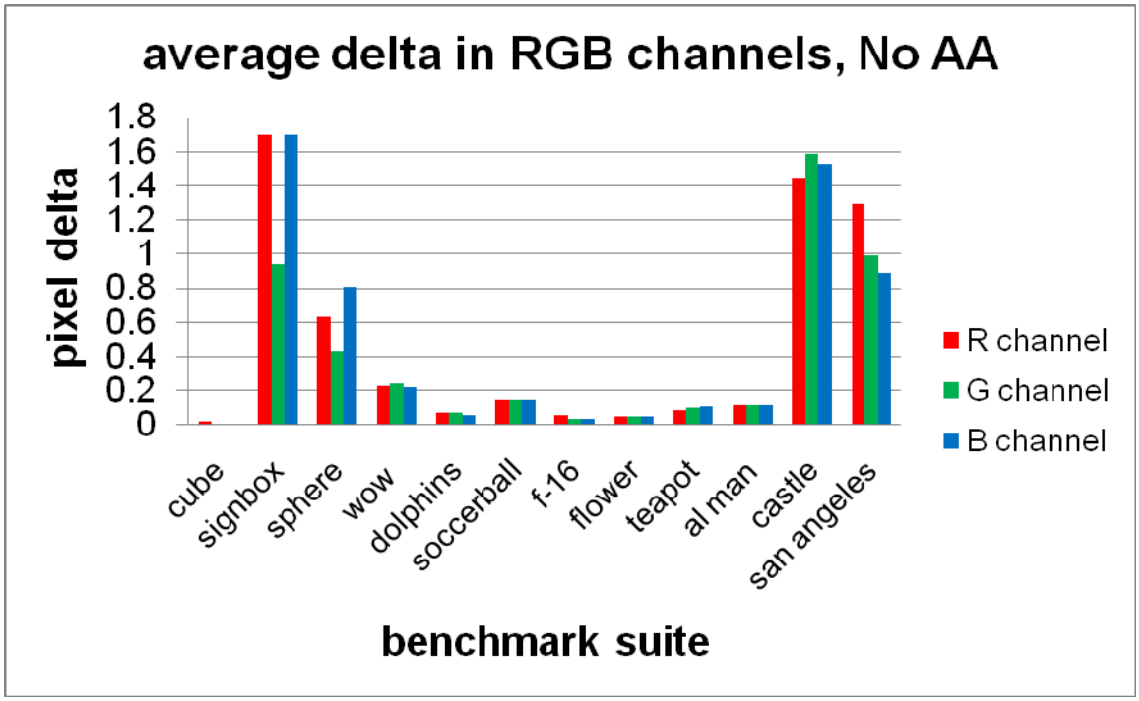


圖 6.7 在 PowerVR 與浮點數版本 GPU 的平均像素差異

在 PowerVR 與我方浮點數版本的比較中，平均每個像素差異皆小於一個單位。基本上，兩個像素的顏色差異若小於 1 至 2 個單位的話，則肉眼就難以分辨此二者的差別。

以此數據評估，我方浮點數版本與 PowerVR 約有 99.61%的近似程度。算是相當接近。而造成這部分依然有微小差異存在的原因，有可能是 Rasterization 內部內插或者是判斷 fragment 的演算法不同所導致，而 Geometry 那邊打光模組所採用的演算法影響到頂點顏色也是必須被考慮的。由於 PowerVR 是由呼叫 3D 顯示卡來進行圖形繪製，對於 3D 顯示卡內部的硬體架構我們無法完全掌握，所以這部分造成顏色差異的真正原因目前還難以確定。

另外，在 3D繪圖領域中，其實並沒有存在一真正客觀判斷顏色好壞的準則，故此處也沒辦法說哪一種設計是真正有問題的，除非以肉眼來看就可以發現明顯的問題。而在圖 6.6之中，以肉眼來看，PowerVR與我方浮點數版本GPU此二者所繪製的圖形幾乎一模一樣。所以，我們可以說我們在軟體的驗證上應是正確無誤的。

圖 6.8 我方浮點數版本與 RTL 版本所繪製圖形

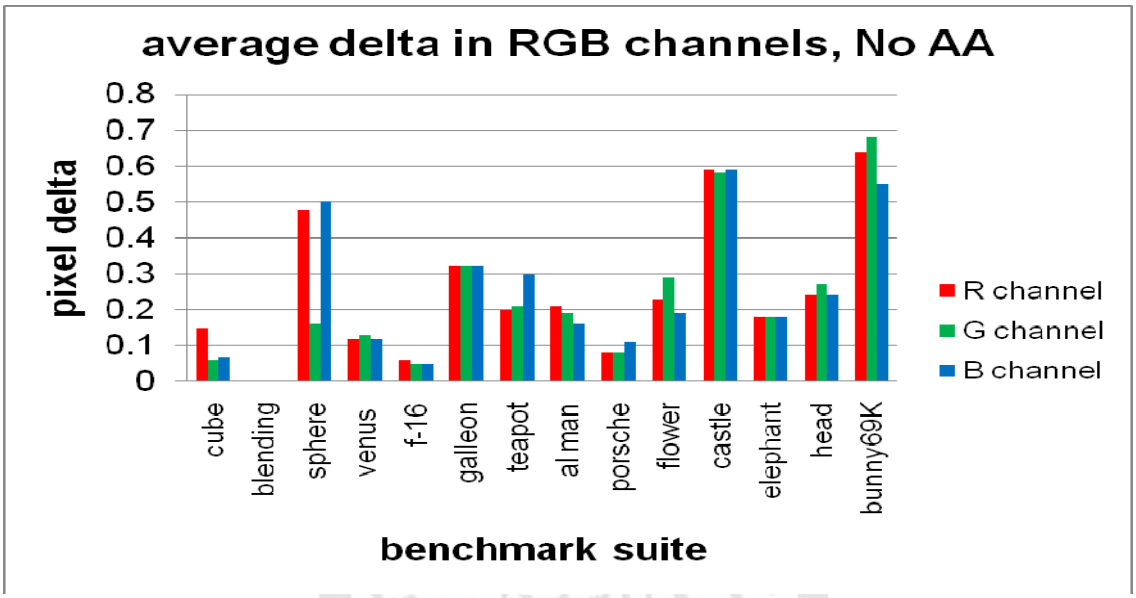


圖 6.9 在浮點數版本 GPU 跟 RTL 版本 GPU 中的平均像素差異

而在圖 6.9，我方浮點數版本與RTL版本的比較上面，大部分圖像的平均像素依然低於一個單位以下。造成指顏色差異的主要原因就在於硬體內定點數的運算與浮點數版本的運算不同，定點數會拋棄掉一些比較不重要的小數值，故顏色上會有細微的差異。不過，這也證明了我們RTL在設計與實現上應該是沒有問題的。

第7章 Conclusion and future work

7.1 Conclusion

我們在這裡完成了一個基於砌塊式架構的 rasterization 繪圖管線設計，除了能夠完成基本的 OpenGL ES 繪圖指令外，還擁有以下特色：

- 使用 Barycentric coordinate 進行內插運算，讓整個繪圖管線即使在精確度較低的定點數運算環境下，也不會造成顏色溢位的問題。
- 採用 Tile bounding skip traversal，可以在 tile-based 的架構下執行有效的三角形掃描。有像掃描比例約為 45%。
- 在 6D block texture cache 與 multi Z test 的演算法之下，可以減少約 95% 至外部記憶體抓取貼圖資訊的存取量。
- 採用了 FSAA 全景反鋸齒技術，如使用 RGSS 反鋸齒的話，畫質可達 4X 反鋸齒的畫質，大大增加了圖像的細緻程度。

Rasterization的RTL實現方面。在RTL的設計面積上，不包含texture unit及on-chip buffer這兩者，約使用了 254K gate count。而瞬間最高可輸出 200M pixel/sec。另外，此硬體在QEMU與CoWare Platform Architecture結合下在全系統驗證平台中完成快速的模擬驗證，整體GPU可以在 200MHz的頻率下正確的執行OpenGL ES繪圖指令。我們將Rasterization的硬體設計規格列在表 7.1中

表 7.1 Rasterization 的規格跟特性

Process Technology	**TSMC 0.18 um**
Gate Count	254K (exclude texture unit and on-chip tile buffer)
Working Frequency	200MHz
Maximum throughput	200M pixels/sec

Supported API	OpenGL ES 1.1 compliant
Resolution	Up to 640*480 (total: 300 tiles ; tile size : 32x32)
Communication Bus	AMBA 2.0 AHB

7.2 Future work

- 將 Texture mapping 實現成硬體：目前在 Rasterization 中，跟貼圖有關的相關演算法及模組只完成到定點數版本中的設計，在 RTL 之中尚未實現。未來如果實現在硬體上面之後，其讀取外部記憶體所造成的 pipeline stall 情形跟 texture cache 的 miss penalty 將是第一個要被考量且模擬的目標。
- 改善 traversal 使所用的掃描演算法：目前此三角型的掃描演算法有效掃描比例約只有 45%，雖然此演算法的複雜度相對簡單。但應該還有更為有效的掃描方法，且同時不會增加太多演算法複雜度。
- 支援多重貼圖：OpenGL ES 定義了 4 組的貼圖模組，而目前的 Rasterization 只完成一組貼圖模組設計。在這個部分上若重新設計的話，則相關的演算法架構，如 texture cache 與 early depth test 將需要重新審視，甚至有可能要重新設計。
- 硬體架構的改進：目前 Rasterization 中部分模組之間的 scheduling 並不是那麼的完善，此部分還需要再進一步的探討與改進。另外像在硬體元件的安排與減少硬體面積的花費上面，應還有改善的空間。

Reference

[1] Jen-Kai Cho, “Multi-view Image Generation Using Compensated Depth-Image-Based Rendering for 3D Displays,” Master’s Thesis, Institute of Computer and Communication Engineering, National Cheng Kung University, 2007.

[2] Vincent Mobile 3D Rendering Library. Available: http://sourceforge.net/projects/ogl-es/

[3] M. Segal and K. Akeley, “The OpenGL Graphics System: A Specification Version 2.1,” Silicon Graphics, Inc, http://www.opengl.org/registry/doc/glspec21.20061201.pdf

[4] R. S. Wright, Jr., and B. Lipchark, *OpenGL SuperBible*, 3rd edition. Sams Publishing, 2005.

[5] Microsoft Corp., DirectX Resource Center, http://msdn2.microsoft.com/zh-tw/xna/aa937781.aspx

[6] D. Blythe and A. Munshi, “OpenGL ES Common/Common-Lite Profile Specification Version 1.1.10,” Khronos Group, Inc. Available: http://www.khronos.org/registry/gles/specs/1.1/es_full_spec.1.1.10.pdf

[7] K. Pulli et al., Mobile 3D Graphic, MK publishing, 2008.

[8] G. Humphreys et al., “Chromium: A Stream Processing Framework for Interactive Rendering on Clusters,” *In Proc. 29th Annual Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH 2002)*, pp. 693–702, 2002.

[9] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, “A Sorting Classification of Parallel Rendering,” *In IEEE Comput. Graph. Appl.*, vol 14(4), pp. 23–32, 1994.

[10] H. Fuchs, et al., "Pixel-planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories," In *Proc. ACM SIGGRAPH*, pp. 79-88, 1989.

[11] Image Technologies Ltd, STMicroelectronics KYRO™ and KYRO II™ PowerVR 3D Graphics Accelerator, http://www.imgtec.com/corporate/newsdetail.asp?NewsID=252

[12] T. Akenine-Moller and E. Haines, *Real-time rendering*, 2nd edition. A. K. Peters, Ltd, Natick, MA, 2002.

[13] A. Stevens, White paper: ARM Mali 3D Graphics System Solution, December 2006, http://www.arm.com/miscPDFs/16514.pdf

[14] Imagination Technologies Ltd., PowerVR MBX 2D/3D Graphics, http://www.imgtec.com/Downloads/index.asp?download=true&f=PowerVRMBX.pdf&t=PowerVR%20PDP%20Documents

[15] AMD&ATI Ltd, Xenos: XBOX360 GPU, http://ati.amd.com/developer/eg05-xenos-doggett-final.pdf

[16] L. Seiler et al., "Larrabee: a many-core x86 architecture for visual computing." *ACM Trans. Graphics (Proc. SIGGRAPH '08)*, vol.27, no. 3, Aug. 2008.

[17] D. Kim et al., "An SoC with 1.3Gtexels/s 3D Graphics Full Pipeline Engine for Consumer Applications," In *IEEE J. Solid-State Circuits*, vol. 41, pp. 71, Jan. 2006.

[18] J. Pineda, "A parallel algorithm for polygon rasterization," *In Proc. 15th annual conference on Computer graphics and interactive techniques*, p.17-20, June

1988, http://portal.acm.org/citation.cfm?id=378457&dl=GUIDE&coll=GUIDE&CFID=43594580&CFTOKEN=14674529

[19] P. Shirley, “Physically Based Lighting Calculations for Computer Graphics,” PhD thesis, University of Illinois at Urbana Champaign, 1990.

[20] M. Cox, N. Bhandari, and M. Shantz, "Multi-level caching for 3D graphics hardware,” I*n Proc. 25th Annual International Symposium on Computer Architecture (ISCA'98)*, pp 86-97, 1998.

[21] Z. S. Hakura and A. Gupta. “The design and analysis of a cache architecture for texture mapping,” *In Proc. 24th annual international symposium on Computer architecture (ISCA '97)*, pp108-120, 1997.

[22] C. J. Choi et al., "Performance comparison of various cache systems for texture mapping," *High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on*, vol.1, no., pp.374-379 vol.1, 2000.

[23] H. Igehy et al, “Prefetching in a texture cache architecture,” *In Proc. ACM SIGGRAPH / EUROGRAPHICS conference on Graphics Hardware*, pp. 133-142, 1998.

[24] S. Morein, “ATI Radeon - HyperZ Technology,” *Hot3D Session 2000 Eurographics Workshop Computer Graphics Hardware*, 2000, http://www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf

[25] NVidia, “Technical Briefs: An In-Depth Look at Geforce3 Features,” http://www.nvidia.com/Products/GeForce3.nsf /technical.html

[26] W.-C. Park et al., "An effective pixel rasterization pipeline architecture for 3D rendering processors," *In Proc. IEEE Transactions on Computers*, vol.52, no.11, pp. 1501-1508, Nov. 2003

[27] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design*, 3rd edition, MK publishing, 2005.

[28] Synopsys, DesignWare IP Family Reference Guide, https://www.synopsys.com/dw/doc.php/doc/dwf/datasheets/dw_div_pipe.pdf

[29] Design Compiler, http://www.synopsys.com/home.aspx

[30] CoWare Platform Architecture, http://www.coware.com/products/platformarchitect.php

[31] QEMU Processor Emulator, official site : http://bellard.org/qemu/