國立臺灣師範大學
資訊工程研究所碩士論文

使用 OpenCL 實現蒙地卡羅路徑追蹤
於 Android 手機平台

OpenCL Monte Carlo Path Tracing On Android

研究生 ： 江冠儀 撰
指導教授： 張鈞法 博士

中華民國 一〇四 年 七 月

摘 要

隨著科技的發展手機的進步速度越趨明顯，手機遊戲也變成開發手機產品中一個不可或缺的角色。隨著手機晶片上的運算能力快速進步，讓手機遊戲可以從 2D 快速進入到 3D 的手機遊戲時代, 而相對於畫面品質和效能也是越來越苛求。為了使 3D 場景的繪製達到更逼真的畫質，至今已經發展了許多不同的演算法以模擬真實環境的光照情形，其中以光線追蹤方法擁有物理運算的正確性並能夠直覺性地計算光照效果，但因光線追蹤的運算速度相對於其他手機遊戲渲染效果是較慢的所以幾乎沒有人將其運用於手機遊戲當中。

本研究搭配 OpenCL 異質系統架構中平行且良好地運作之特性，對於光線追蹤龐大的計算下平行處理能讓畫面的渲染速度更有效率且畫面品質將有不錯的效果。本渲染對象以動態場景為主，會分別對於場景的複雜度和其渲染速度上做效能分析，將重點著重於可行性的探討與提升程式效能，針對所使用的平台進行實驗結果比較以及影響效能的相關探討，最後，我們將因應平台特性將演算法與之結合，試圖找出在多核心平行化架構下提升各平台程式效能的實驗依據並以其研究出適合於手機平台上的光線追蹤演算法架構。

關鍵詞: 圖形處理器、光線追蹤、異質性系統架構、平行運算、OpenCL、Android。

ABSTRACT

With development of mobile chip computing technique increasingly, it enhances the growth of a range of related to frame quality especially mobile game. It progresses the mobile game transform from two-dimensional space generation to three-dimensional (3D) space game. Graphics Processing Unit (GPU) of mobile phone offers powerful computing ability, it is applied to 3D scene compute. Majority of 3D game scene are not making user to experience realistic. In order to improve 3D sense realistic, that developed a number of different algorithms to compute ray in the scene. This research focus on ray tracing algorithm. Ray tracing uses mathematical formulas, it can calculate lighting realistically and correctly. However ray tracing need a lot of compute, it spends too much time to render. If ray tracing compare with others mobile's render technique, it will too slowly.

This research uses OpenCL to accelerate ray tracing in mobile phone. It will make scene not only realistically but also efficiently. We will use other platform to experiment our research, and find out good ray tracing algorithm for mobile phone.

Keywords: GPU, Ray Tracing, Android, OpenCL.

誌　謝

在師大求學過程的期間，我首先要謝謝我的指導教授張鈞法。讓我能了解許多關於計算機圖學的相關背景，還有知識。他總是會告訴我們一些科技的新知識並告訴我們面對困難時該如何解決。當不了解問題的答案時，他總是很有耐心的跟我們講解，並給予鼓勵。並訓練我在這個專業領域的思考能力，以及了解這個領域目前的發展趨勢。也因為實驗室有參與計畫，而這個計畫是屬於跨校合作，因此可以讓我在計畫中學習到不同領域的知識，和不同學校的同學合作，互相討論。也謝謝計畫中認識到的高至辰學長，給予我論文方面的意見，和程式方面的討論研究，感謝他總是很耐心的回答我的問題，並給予我鼓勵。

而在實驗室的兩年裡，我也要感謝我的實驗室夥伴和朋友們。首先謝謝實驗室的沁傑、郁凱、冠惟、庭瑋四位學長姐，在求學期間給我的鼓勵以及幫助，會不厭其煩的為我解答問題，並分享我工作上的經驗，也告訴我不要放棄要堅持下去。也謝謝別間實驗室的尹廷學長，雖然所學領域不同，但他還是熱心的教我如何使用 word 的一些編輯功能，讓我寫起論文時非常方便。也謝謝一起陪我奮鬥的隊友宜聰給我論文上的加油打氣，也謝謝毓偉、浩庭、威豪、柏元四位學弟們為實驗室帶來歡樂，和對論文上不同的見解，謝謝他們的幫助。也謝謝我的男朋友育祥不停的為我加油打氣，給我堅持永不放棄的信念，遇到困難不要逃避的勇氣，也謝謝他給予我論文方面的意見和幫助。最後要謝謝我的家人，給予我鼓勵和支柱我完成碩士學業，讓我能夠心無旁騖的面對研究論文時的挑戰。

學生 江冠儀

目 錄

附表目錄

附圖目錄

第一章　簡介

1.1 研究背景

近年來手機遊戲已經成為市場的主流，隨著手機晶片越來越厲害，功能越來越強大，手機 APP 已經成為大家生活不可或缺的一部分。隨著手機遊戲的快速進步，從 2D 進步到 3D 的手機遊戲，對於畫面品質和效能也是越來越苛求。光跡追蹤(ray tracing)在電腦圖學領域中扮演著重要的角色，它是一種用於渲染(rendering)的演算法，藉由數學式子和物理理論運算光的折射、反射與各種變化，藉以得到高品質的畫面且增加場景的真實性。然而光跡追蹤在渲染速度上是相對於其他渲染效果較慢，需要大量計算一些數學公式，所以使用了開放式計算語言(OpenCL)來提升效能，且在 GPU 上能讓設計程式功能便利許多，並且藉由多核心平行化的方式將能夠有效的處理計算機圖學龐大的計算量，已達到硬體加速的效果。讓手機也能在短時間內達到相當真實的畫面效果。

1.2 研究目的

光跡追蹤(ray tracing)一直是讓 3D 場景有很高擬真度的渲染方法，電腦圖學很早就開始在針對光跡追蹤去做研究，並將他們用在現今電腦遊戲中，以達到真實渲染效果。可是現今手機如此發達卻幾乎沒有將此技術運用在手機遊戲中。主要有開發上的限制，促使開發者不願去嘗試。因為光跡追蹤本身需要耗費大量的計算能力導致渲染時間過長，非常難以即時互動。所以本研究想在手機中嘗試使用光跡追蹤畫出高擬真度的畫面效果，並盡可能達到即時互動，讓手機上的遊戲或是渲染效果能更加逼真。本研究透過手機 GPU 的平行計算方式並配合開放式的計算語言(OpenCL)去加速光跡追蹤，讓手機上的畫面也可以在短時間內達到高品質的效果。本研究主要探討兩個方向，第一，光跡追蹤搭配 OpenCL 在手機上的運算效果和畫面真實度。第二，將實做出來的手機效能與電腦效能做比較，並歸納出一份分析報告，且找出光跡追蹤在手機性能上部分資源不足的情況。

1.3 論文架構

本篇論文一共分為五個章節：第一章為簡介，說明本篇論文的研究背景以及目的，和論文架構概述。第二章為文獻探討，主要針對所要移植的光線追蹤演算法做概論和分析，且針對 Android 的相關移植和跨平台技術，如何使用 Android NDK 將 OpenCL 移植到 Android 系統上等相關論文概述。以及針對 OpenCL 跨平台的彈性基礎上，配合 GPU 平行化架構特性(SIMT)，有效利用共享記憶體區塊，

加速最終彙集的步驟。第三章為系統實作，針對本研究所採用的技術，以及所實作的演算法和平台進行介紹。第四章為實驗數據分析，針對不同手機晶片加以測試，找出適合用於手機架構的 ray tracing 演算法。第五章為本研究的結論與未來展望的探討。

第二章　相關研究探討

2.1 Ray Tracing

傳統 ray tracing 是以相機為出發點，穿過畫面產生一條光線，藉由光線判斷是否與場景中的物體有相交，相交後在根據打到的物體產生第二條陰影或反射或折射光線，再去追蹤場景中的物體相交，不斷的遞迴相同動作到能量遞減為止，最終會根據回傳的物體顏色和光照能量來給予在像素上的顏色貢獻。

光線追蹤[**Kajiya 86**]透過物理公式的計算讓畫面品質有了極高的真實度，所以近年來也不斷提出新的演算法概念。為了讓畫面有更好的效果這些提倡的技術光照統稱為全域照明除了考慮到直接光源的照明，更包含了如間接照明、反射、折射、焦散等效果。此領域至今已經發展出許多方法能夠達到擬真的畫質，如輻射度(radiosity)、路徑追蹤(path tracing)、光子映射…等。本論文採用蒙地卡羅路徑追蹤演算法[**Keller 12**]，路徑追蹤演算法為光線追蹤演算法的延伸，是採用蒙地卡羅積分法來解成圖方程式。蒙地卡羅路徑追蹤法使要的概念是由相機出發一條主要光線通過畫面某一像素去判斷有無和 3D 場景中的物體相交。經過交點測試後，假如與物體交於 x1 點後在與光源連線看光源對此點有無貢獻此稱為直接照明，假如中間有物體擋住表示此點位於陰影內，反之表示光源對此點有直接照度的貢獻。針對此點算出照度貢獻後，還要對 x1 點以雙向分部函數來決定次要光線的方向及數量。次要光線則是用來計算其他物體對於 x1 點的間接照度貢獻，如此

反覆的計算直到能量衰弱後，將此照度最終匯集傳回給視點，這即是該像素應有的照度值。

2.2 Android NDK

Android一直是開發手機的作業系統，Google官方推薦使用基於Java程式語言的Android SDK來開發Android App，但Android SDK卻無法完整的開發出在Android執行上的所有功能，且Java語言在runtime的時候需要在直譯成機械碼，所以在效能上並不是這麼顯著。所以Android另外提供了NDK可以使用JNI (Java Native Interface)讓Android也可以使用底層的C/C++或其他語言來開發App。利用Android NDK **[Son 11]**，將使用NDK和沒使用的做法比較兩者速度上的差異。Son and Lee(2011)使用NyARToolKit當作範例來比較，NyARToolKit是以Java為主所的函式庫，主要是用來編寫擴增實境的應用程式。因為是以Java為主的語言，所以在runtime時速度較慢。因此將NyARToolKit內的其中一個class轉換成Native C，並透過JNI來修改。

Type	Target Method	Entire of NyARToolKit
Non using NDK	149.918 msec	2394.212 msec
Using NDK	2.353 msec	1280.824 msec

圖 1: 使用 NDK 前後差異比較(Son & Lee, 2011)

使用 JNI 將原生的 Java code 部分改成 Native C 得到了附表的結果。

speed up = 2394.212/1280.824=1.869，整個效能上明顯提升許多。

2.3 OpenCL Platform

OpenCL 全名為一個方放式跨平台計算語言(Open Computing Language)，其可在異質多核心系統架構上良好地運作，在平板電腦或是行動裝置都能有良好的搭載。可以使 GPUs 除了擴充圖形生成能力之外，更提供了任務為主與資料為主的平行化機制，針對異質性計算裝置(heterogeneous device)進行平行化運算，平行化運算是指許多指令得以同時進行的計算模式。在同時進行的前提下，可以將計算的過程分解成小部份，之後以並行方式來加以解決，使得複雜運算得以快速解決。在以往光線追蹤方法中，通常會使用大量迴圈來計算光線照度的結果，由於重複計算相同指令導致效能降低，若使用 GPGPU(General Purpose computing on Graphics Processing Unit)的平行計算能力，便可將複雜運算工作分給多個執行緒同時執行，以達到平行化的效果，也大幅減少了運算時間成本。

本論文主要是將 OpenCL ray tracing 移植入手機，而 OpenCL 在電腦上和行動裝置上的差別主要是在行動裝置目前只支援 OpenCL 1.1 版本，而電腦目前已開發到 OpenCL2.1 版本了。未來也許行動裝置會支援 OpenCL 1.1 以上的版本。

2.3.1 Execution Model

OpenCL 在執行上分為兩部分，為主機端(host)與裝置端(device)。主機端主要是執行 OpenCL 的 API 和執行一般的應用程式，程式透過 OpenCL API 取得 OpenCL 裝置(例如顯示晶片)的相關資料，並將要在裝置上執行的程式(Kernel Program)編譯成適當的格式，在裝置上執行。OpenCL 程式內核架構主要是以 Single Instruction Multiple Thread(SIMT)方式運行的，所以同一個工作群組(work group)中的內核執行緒(kernel thread)將同時執行同一行指令，因此使用者可以根據裝置內的運算單元數量來決定所以要分配的工作群組大小，以藉此提升程式平行化的效能。

2.3.2 Memory Model

在 OpenCL 裝置上記憶體可分為四個部份全域記憶體(Global Memory)、唯讀記憶體(Constant Memory)、共享記憶體(Local Memory)、私有記憶體(Private Memory)。裝置內的所有工作項目(Work Item)在全域記憶體中皆可讀寫，全域記憶體和其他三個比較再讀寫方面是較慢的。而唯讀記憶體是寫入的資料不能再做修改，且每個工作項目皆可使用它。共享記憶體則是在同一個工作群組中所有的工作項目一起共用，且速度和快取區域差不多，如果將資料移到這塊記憶體上使用，相信對節省記憶體頻寬會有很大的幫助。私有記憶體只供工作項目自己本身讀寫，相當於暫存器讀取速度也是全部最快的。

2.4 參考相關研究

關於手機上使用光跡追蹤演算法**[Wang 14]**，Wang(2014)是以硬體測試為主，不是真實在手機上比較實驗結果，是以Odroid XU 開發的晶片和Jetson TK1的開發晶片並燒入作業系統的方式去實驗的。且Wang(2014)實作以基本光線追蹤為主，加速結構上採用的是BVH (Bounding Volume Hierarchy)為主的加速結構，他將場景分成三個不同的模型依三角片的數量不同做區分，並將實驗程式分成五個kernels用GPU分析器去分析在這些kernel functions下GPU的實驗結果，這樣可以瞭解出在做光線追蹤時哪部分在GPU會花較多的時間。本研究和Wang(2014)目的同樣是想針對光線追蹤在手機上實作效果進行研究，但本研究與其實作方式不同，而是實際將光跡追蹤載入手機環境下執行，並分析研究實作結果。

第三章　演算法設計與討論

本章分為安裝使用和演算法解說兩部份，下圖為本實驗系統流程圖。

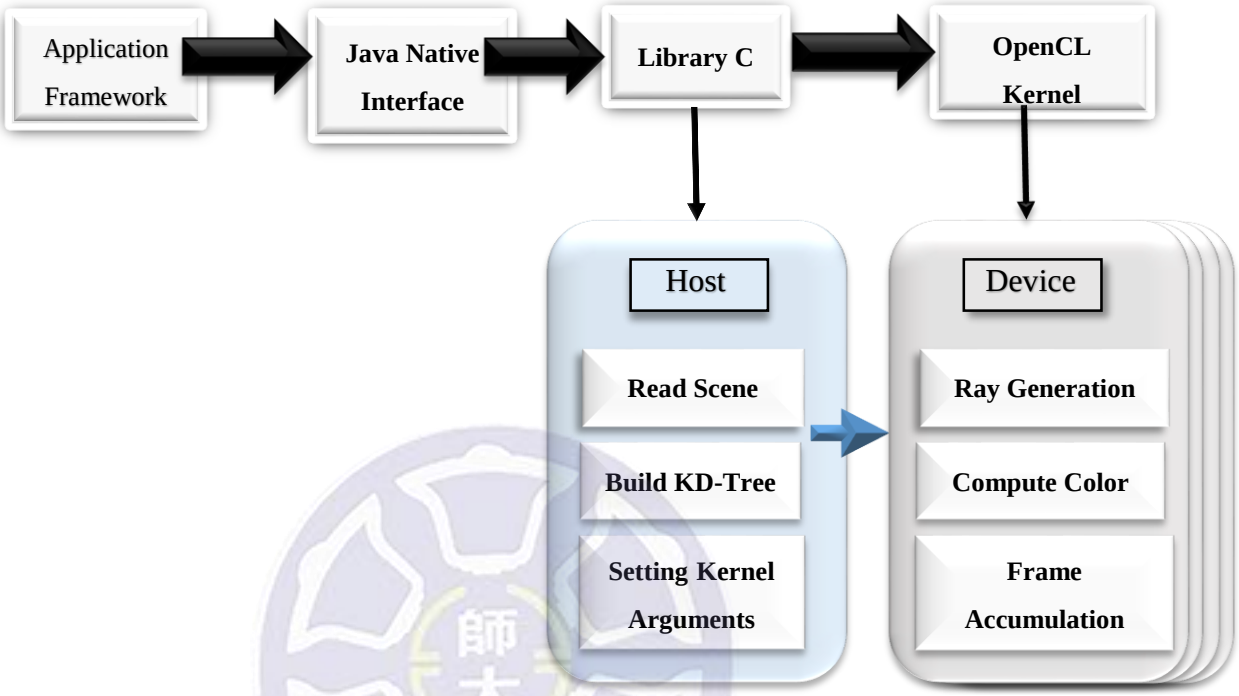

圖 2: 本研究系統流程圖

演算法結構分為 Java part 和 Native part 兩部份，Java 端是用來控使用者介面和載入 OpenCL 程式資料，而主要的畫面像素運算則是透過 Native 端並使用 OpenCL 來運算。兩者之間互相傳遞的方式則是透過 Java Native Interface(JNI)。

Native 端包含了所有 OpenCL 的 function call API，而 OpenCL 在本研究中是用來計算像素顏色。本研究的演算法為實現蒙地卡羅路徑追蹤演算法，且在複雜場景時使用 K-D Tree 的樹狀加速結構，增加程式的效能。當 Java 端透過 JNI 呼叫 Native 端的 openCLNR.cpp 檔(本研究的主程式)時，相當於先前提到的主機端，會先將場景資訊讀入，並建置加速結構，之後透過 OpenCL API 來設置傳入 OpenCL 的參數，之後主機端會將資料送入裝置端並儲存在全域記憶體中。裝置

端收到資料後會啟動內核程式來計算畫面的顏色，藉由 SIMT 的特性，就畫面中的每一個像素產生追蹤光線，並與場景做交點測試。當有交點產生時，再從此點產生新的光線繼續追蹤，直到光線衰弱並回饋顏色，如此不斷累積，最後將顏色匯集傳回到主機端，再由主機端透過 JNI 回傳到 Java 端並顯示每個像素所累積的顏色在手機螢幕上。

3.1 Monte Carlo Path Tracer

本研究所使用的蒙地卡羅路徑追蹤法來達到場景的全域照明效果，根據參考資料三的蒙地卡羅隨機亂數產生法來實場景中的光線，達到全域照明的效果。蒙地卡羅會因為隨機採樣的像素顏色差異大的原因而導致雜訊產生，所以本研究是以累積像素值的方式來減少雜訊。當光線與物體有交點時，我們會從該交點發射陰影光線，射向光源區域中隨機選擇一點採樣點，藉由隨機採樣光源區域，將累加採樣到像素緩衝區(frame buffer)，再由此交點隨機選擇一條光線去追蹤下次所生成的交點，隨著樣本數增加累積像素值，會慢慢減少場景中的雜訊，讓畫面效果更加真實。

3.2 K-D Tree

光線追蹤需跟場景做求交測試，假如場景都由三角片所組成，那畫面上每一個像素值都需跟場景中的所有三角片做求交測試，其結果在渲染一張畫面所需的時間要相當的久。所以演化出許多加速結構來減少場景中的求交測試。本

研究採用 K-D Tree 演算法來實作，雖然建置樹狀結構的時間會較久，但走訪時間是相對於其他加速結構中較快的。

3.2.1 K-D Tree Construction

本研究演算法首先在 OpenCL 主機端上建立樹狀結構，先計算每個三角片三個點的平均值作為空間描述的分類依據，並找出所有三角片的最大和最小值當作根結點走訪的第一個包圍盒(Bounding Box)。其後以座標軸對稱(Axis-Aligned Bounding Box)的格式切開場景，以由上往下的方式進行，將場景已 X.Y.Z 三軸輪流做切割如此遞迴下去。之後將切割好的樹狀結構以前序走訪(pre-order)存入一維陣列中，存入的三角片是以編號方式存入，此陣列作法是避免如果從主機端以指標結構的方式傳入裝置端會因為兩者所指向的記憶體區塊位置不同而造成資料傳送錯誤。最後把主機端陣列資訊傳入裝置端。下圖為場景切割建構圖。

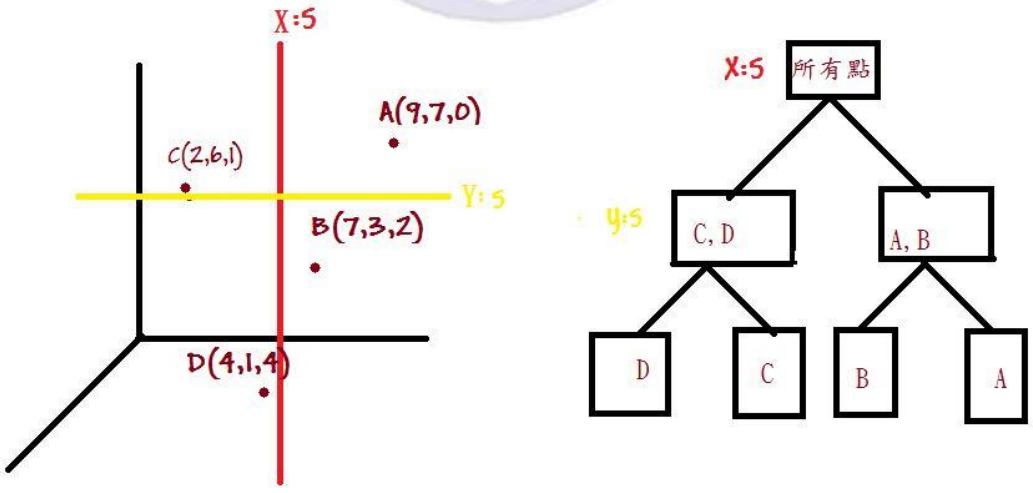

圖 3: K-D Tree 樹狀建構圖

3.2.2 K-D Tree Traversal

K-D Tree 的走訪是實作在裝置端上，光線會先從根節點(root)出發，先對根節點的包圍盒進行求交點測試，當無交點產生時即不需要走訪下去。當與根節點有交點時會根據此節點的邊界值(Bounding Value)來判定是該往右還是往左走，如此走訪下去直到走到葉節點(leaf)或是堆疊內的節點為空。當與葉節點的包圍盒有交點時，會再將光線與此節點內的所有三角片進行求交點測試，如此一來變可減少光線與場景中的所有三角片的求交點測試，並減少渲染時間，提高效能。在走訪的過程中我們採用固定大小的堆疊(stack)來記錄等待走訪的節點，在這我們以迴圈的方式走訪，因為 OpenCL 並不支援程式以遞迴的方式寫成。當光線與節點包圍盒產生交點時，會再根據此節點的邊界值來判定是左右子樹皆需走訪，還是只與一邊有交集。最後再根據光線位置來判定當大於邊界值則往右子樹走訪，小於邊界值則往左子樹走訪，當相等時會先把待走訪的節點放入堆疊中，等待一邊子樹走訪完後，再取出堆疊內的節點繼續走訪。堆疊方式紀錄等待走訪節點的方法預計在全域記憶體存取頻率較高時，會更加吃重裝置的記憶體頻寬(memory bandwidth)效能和區域記憶體(local memory)。附圖為傳入裝置端的一維陣列結構，當從主機端傳入裝置端的三角片會給予編號，objId 即為每個節點中所存的三角片編號。axis 為該節點是用 x, y, z 哪一軸做平分左右子樹。bound 為該節點切割標準值。objNum 為該節點內物體數量。leftChild 和 rightChild 則為該節點所記錄的左右子樹。num[3]為記憶體對齊所配置的額外空間，因 GPU 在做資料搬移時是一

次抓四個 byte 的資料，為了湊齊四的倍數，額外產生的配置空間。boundMax 和 boundMin 為此節點的最大最小值。

```
struct clTnode
{
	int objId[maxInnode];
	int axis;
	float bound;
	int objNum;
	int leftChild;
	int rightChild;
	int num[3];
	float4 boundMax;
	float4 boundMin;

};
```

圖 4: 自訂傳入裝置端的 K-D Tree 陣列

3.3 程式安裝說明

使用本研究程式碼須先安裝 Android SDK 和 Android NDK，之後點擊 Eclipse 即可開始實作。打開後須選擇本實驗資料夾所存放的位置之後點選 File->Import，以下步驟如圖。

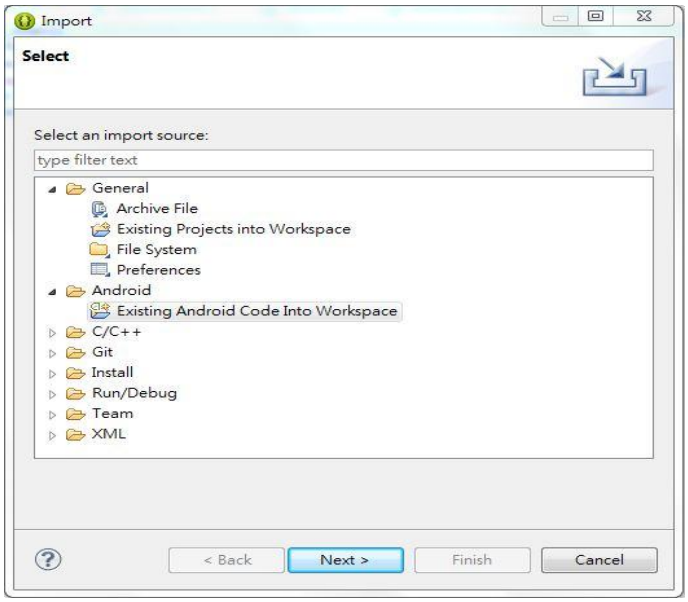

圖 5: eclipse 操作步驟圖

之後點 Browse 選取資料夾所在的路徑位置。

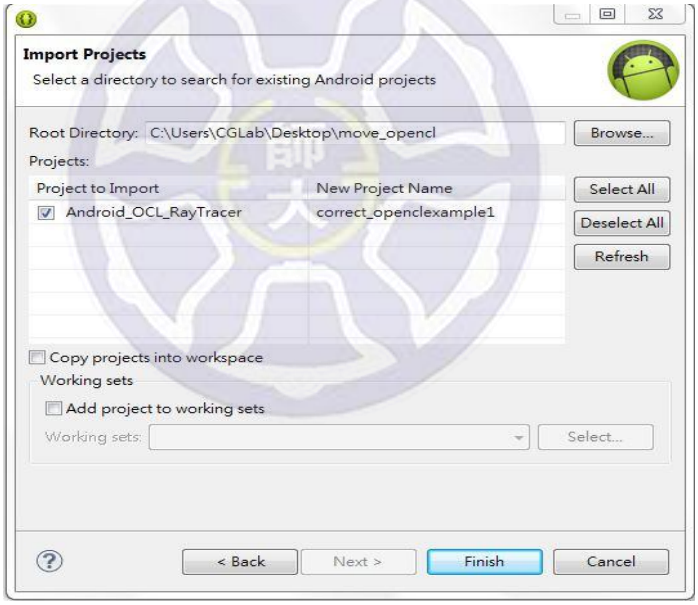

圖 6: eclipse 操作步驟圖

當檔案建置好後，還須調整 Android NDK 的路徑才能開始啟動本研究的程式。

在檔案上按右鍵點選 properties 之後如附圖。

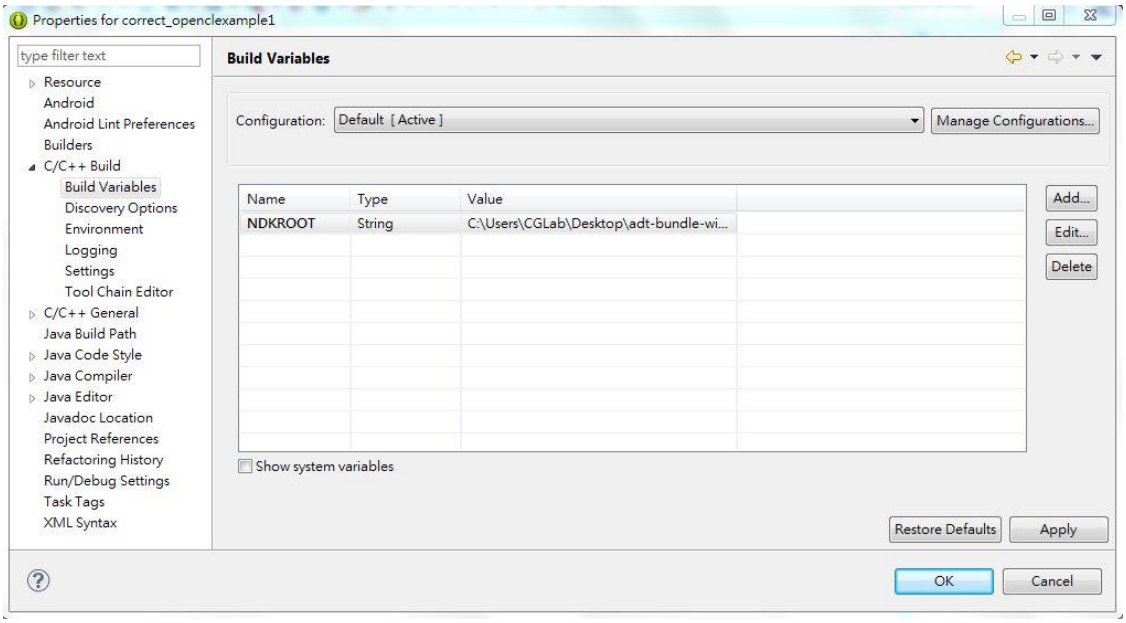

圖 7: eclipse 操作步驟圖

這時可以看到 NDKROOT 的原設定，把這個路徑改成所載的 Android NDK 所在的路徑位置即可。

3.4 Android JNI

主要就是使用 Android 中的 Java Native Interface(JNI)接口，將 Android 中沒有的功能當作函式庫加入並透過 JNI 傳遞資料，並結合 Cygwin 環境和 NDK 工具來實作。首先需將 OpenCL 函式庫編譯為可被 Android 調用的.so 檔，這邊可用 Cygwin 自由軟體生成，然後在 Eclipse 開發環境中選用 JNI 會自動生成兩個檔案，Android. mk 和 Application. mk，前者用來定義編譯文件的目標和文件編譯的順序，後者為程序平台版本和編譯器版本等內容。須更改 Android. mk 內容。本研究改變了 LOCAL_C_INCLUDES 將路徑設為要連結的 OpenCL 標頭檔位置，LOCAL_SRC_FILES 為所要編譯的程式碼檔案名稱，和 LOCAL_LDLIBS 加入剛剛編譯的. so 檔路徑位置，然後再將本研究所寫的程式檔拷貝到 JNI 資料夾底下。再來就是將 Java 端的資料傳入 Native 端。本研究的 sonyOpenCLexample1.cpp 檔為本實驗的資料傳遞的程式檔，先從 java 端傳 pixels 陣列(要計算像素顏色的暫

存空間)到此檔案中的 runOpenCL function，之後再此 function 內呼叫 native 端的主要 function 如此來回傳遞做資料上的溝通。傳過去的變數需加上'j'，例如 int 變數需轉變為'jint'，傳遞的 function 名稱也要改為 JNI 的語法。下表為 JNI 變數型態語法轉換。

Java Language Type	Native Type
boolean	jboolean
byte	jbyte
char	jchar
short	jshort
int	jint
float	jfloat
long	jlong

表 1: JNI 變數型態語法轉換對應表

第四章　實驗結果分析與設置

4.1 實驗平台分析

在這測試環境中，本研究使用手機和電腦作為研究平台。測試機一為 HTC desire 816 這台手機所用的晶片是 Qualcomm Snapdragon 400，另一台則是使用 ASUS PadFone S 使用晶片為 Qualcomm Snapdragon 801 兩者皆是屬於 System-On-Chip(SOC)的系統晶片，是將 CPU、GPU 和其他元件做在同一張晶片上。其晶片上所有的硬體元件都是共享一個雙通道的 LPDDR Memory cache，前者 CPU 為 1.6GHz Quad-core arm cortex-a7 其 GPU 為 Adreno 305，後者 CPU 為 2.36GHz Quad-core arm cortex-a7 其 GPU 為 Adreno 330，在電腦平台上則選用 GPU 為 GeForce GTX 750 Ti，CPU 為 Intel Core(TM) i5 4460。

	Type	Compute unit	GFLOPS
Adreno 305	GPU	24	21.6
Adreno 330	GPU	128	158.4
NVidia GeForce GTX750Ti	GPU	640	704

表 2: 使用平台 GPU 介紹圖

	Type	Compute unit	記憶體支援
Qualcomm Snapdragon 400	CPU	2	單通道 533 MHz LP-DDR2
Qualcomm Snapdragon 801	CPU	4	雙通道 800 MHz LP-DDR3 12.8 GB/sec
Intel Core(TM) i5 4460	CPU	4	雙通道 25.6 GB/sec

表 3: 使用平台 CPU 介紹圖

4.2 三角片場景測試結果

本章以三角片為實驗對象，針對不同三角片數量做分析與測試，附表為測試以蒙地卡羅路徑追蹤-直接照明為主，我們將用兩台手機做測試，並比較兩者效能上的差異。此章將會用來與 4.3 章有加速結構的實驗做比較分析。

	Size	Snapdragon400	Adreno305	Snapdragon801	Adreno330
Cube	12	0.277	0.707	0.159	0.191
Teapot	1056	10.373	9.963	5.199	2.139
bunny	4996	60.518	59.437	30.486	13.545
head	17684	x	x	120.069	53.66

表 4: 三角片在不同手機上的測試(單位時間為秒)

表中顯示'x'的符號為程式執行太久而造成無預警關閉。可以看出使用 Snapdragon801 晶片的手機效能較 Snapdragon400 要來的好，根據製程結果

Snapdragon801 在 GPU 製程結果運算能力 253.1 M Triangles/s 而 Snapdragon400 的 GPU 運算能力為 66.7 M Triangles/s，兩者的 GFLOPS 運算能力前者為 158.4 後者為 21.6，由兩者可以看出在做三角型運算時差了大概有四倍的運算能力，而實驗也可以看出 GPU 所花的時間差了有四倍多。

4.3 K-D Tree 實驗結果分析

由於場景大部分都是由三角片所組合，而在光線追蹤演算法架構下，會因為求交測試而浪費許多不必要的計算，導致計算時間過久造成手機程式無預警自動關閉。本研究採用 K-D Tree 來加速在 ray tracing 中的複雜場景三角片的求交測試，並探討在手機晶片下加速的情況和分析其效能。本研究因為手機裝置的 Local Memory 內存容量小，所以在 GPU 上執行時，當 K-D Tree 層數過多時會導致程式發生錯誤。而 CPU 容量相對於 GPU 是大上許多，所以並不會發生錯誤。

在實驗時發現如果在 GPU 上調用 OpenCL 的 API clBuildProgram 時在選項 options 上選 cl-fast-relaxed-math 可以讓 GPU 在計算速度上更加快速。下表為加速結果，這一章以 PadFone S 有無加速結構差別來比較。

	Size	Snapdragon801	Snapdragon801+tree	Adreno330	Adreno330+tree
Cube	12	0. 159	0. 178	0. 191	0. 105
Teapot	1056	5. 199	1. 012	2. 139	0. 635
Bunny	4996	30. 486	3. 053	13. 545	2. 322
Head	17684	120. 069	9. 831	53. 66	5. 316

表 5: PadFone S 手機加速前後差異圖(單位時間為秒)

可以發現當三角片數量較少時，有 K-D Tree 會因走訪而浪費較多時間但隨著三角片的數量越多時走訪成本一樣但求交次數減少，讓整個效能提升。而沒有 K-D Tree 的則會以線性方式成長，在做交點測試時所花費的時間要更多。下圖為在不同架構下的加速結構倍數差異，以圖表呈現。

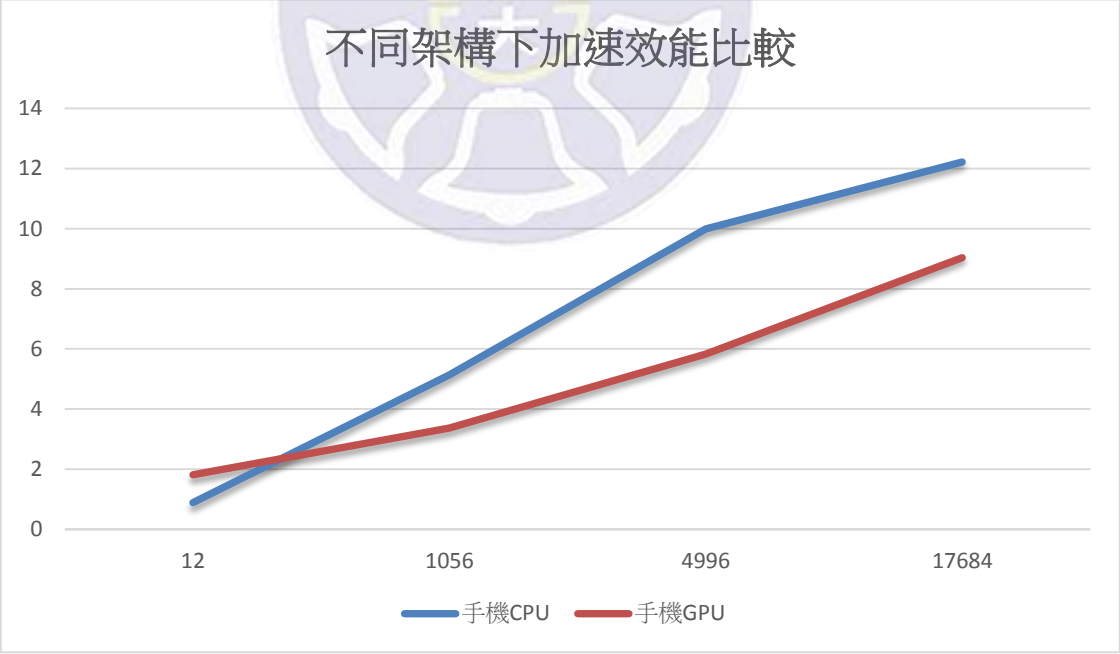

圖 8: 手機 CPU/GPU 加速效能比較圖

因為 GPU 在 Local Memory 的限制下，所以在建置 K-D Tree 時層數無法切太多，所以這邊只比較切三層時 GPU 和 CPU 兩者之間速度變化的差異。雖然

GPU 的運算速度相較於 CPU 是快很多的但因為平行運算下會造成 GPU 的分歧分支(thread divergence)問題，當資料量過多時，就會提高分歧分支的問題導致渲染時間延長。由實驗結果可看出 GPU 會因為分歧分支的問題導致資料量多的時候會讓加速結果效能下降。由於 GPU 內核執行緒平行化的特性，當程式發生分支(branch)問題時，每個執行緒必須等待所屬工作群組執行完畢，才能將結果回傳。下表為有無 K-D Tree 時光線的平均求交點次數差異圖。

	Size	No K-D Tree	K-D Tree
Teapot	1056	1056	92.64+1.98(box)
bunny	4996	4996	254.02+2.49(box)
head	17684	17684	950.29+2.45(box)

表 6: 有無加速結構的平均交點測試圖

無 K-D Tree 時每條光線和三角片求交點數量平均都是看三角片個數有多少就需求相交多少次，但很明顯看出有 K-D Tree 結構時每條光線和三角形的求相交數量明顯減少，只需先和 Bounding Box 求相交，當有相交時在與包圍盒裡面的三角片個數做交點測試，因為手機內存有限，所以層數上就沒分很多層，但假如再分割下去，在渲染畫面上效能是會再提升許多。

4.4 Cross Platform 實驗數據分析

此章節將比較手機和電腦平台的速度差異，並說明研究結果。表中為在電腦上使用加速結構前後的數據差異。使用 K-D Tree 層數為三層。

	Size	Intel i5 4460	Intel i5 4460+tree	GTX 750Ti	GTX 750Ti+tree
Cube	12	0.036	0.029	0.001	0.002
Teapot	1056	1.166	0.178	0.064	0.027
Bunny	4996	9.192	0.48	0.279	0.082
Head	17684	28.265	2.176	1.058	0.207

表 7: 電腦 CPU/GPU 加速前後比較表(單位時間為秒)

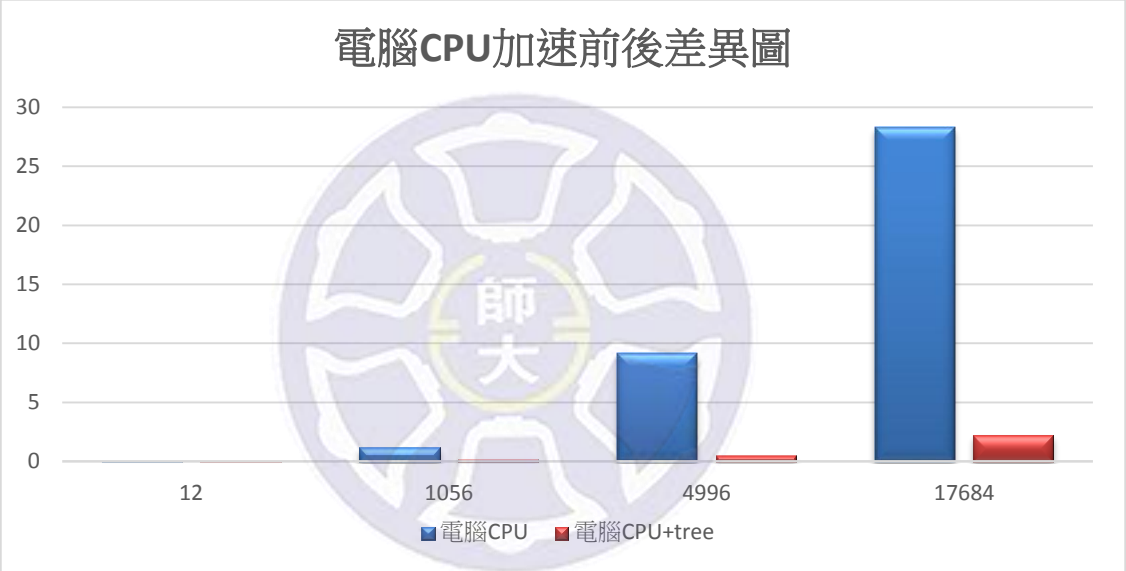

圖 9: 電腦 CPU 加速前後差異直方圖(x 軸為三角片個數 y 軸為時間秒數)

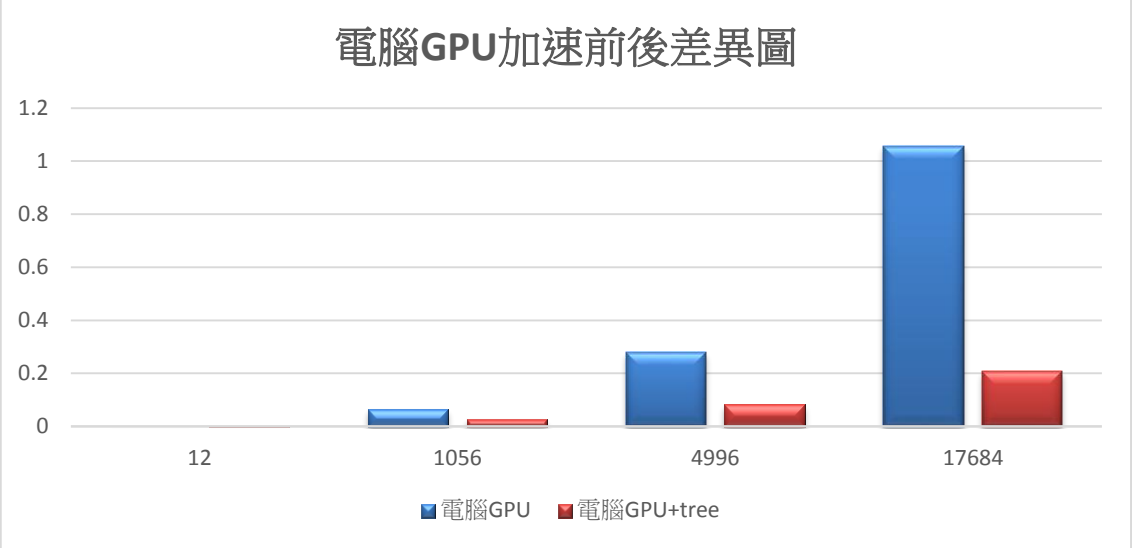

圖 10：電腦 GPU 加速前後差異直方圖(x 軸為三角片個數 y 軸為時間秒數)

配合手機因 Local Memory 大小問題，此實驗只將 K-D Tree 切三層實驗。可以看出加速後效能明顯變化很大。由表中可看出當場景複雜度提高，渲染畫面的時間也會跟著三角片數成指數提高，但有了加速結構後可以明顯看出，場景渲染時間降低許多。且可以發現 GPU 的運算能力對於 CPU 來說是強上許多的。

如圖 11 為手機和電腦在 CPU 未使用加速結構比較圖。而圖 12 則是手機和電腦在 GPU 未使用加速結構比較圖。

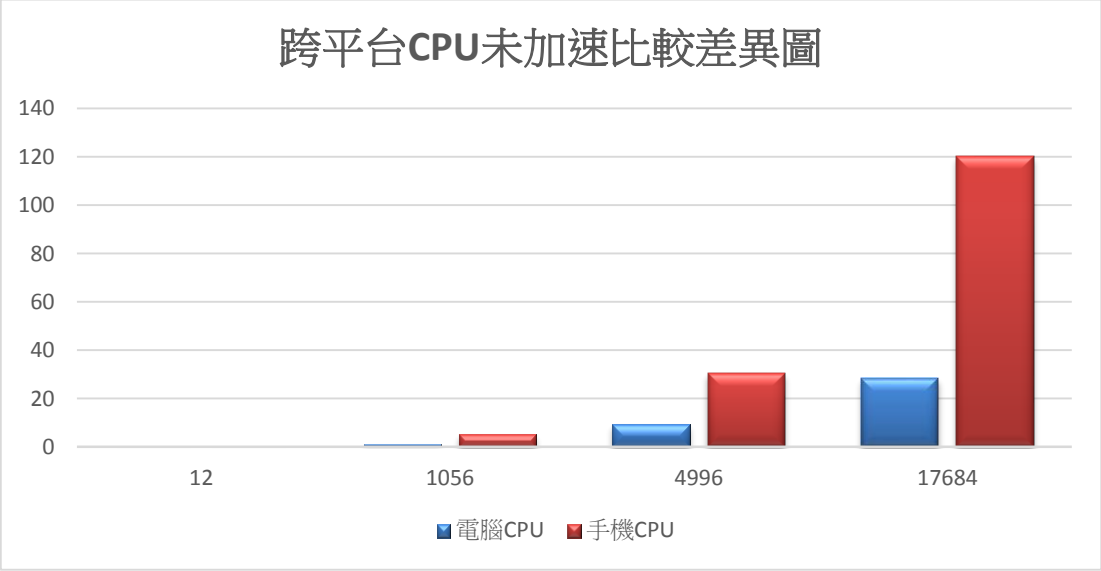

圖 11：跨平台 CPU 未加速前比較差異圖(x 軸為三角片個數 y 軸為時間秒數)

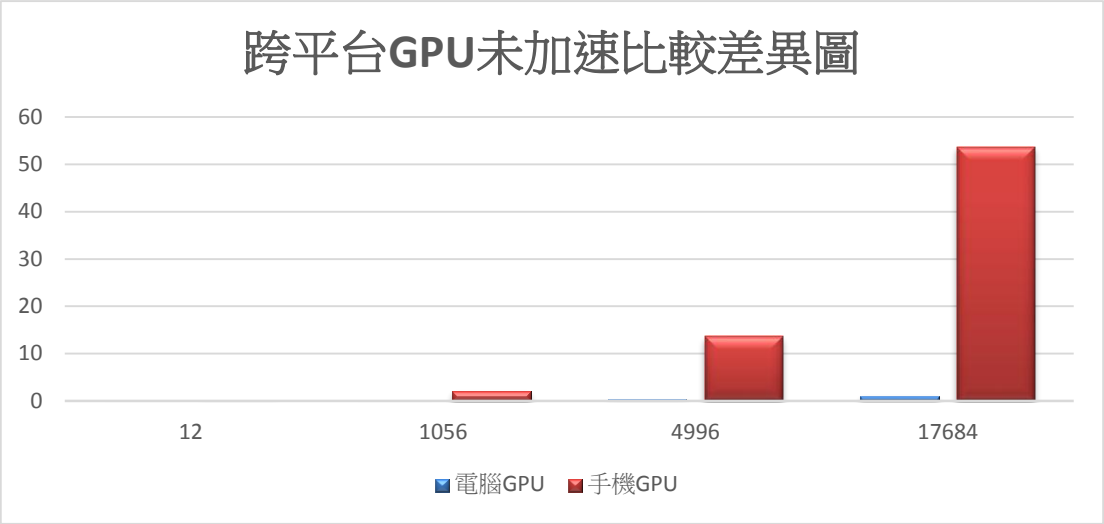

圖 12: 跨平台 GPU 未加速前比較差異圖(x 軸為三角片個數 y 軸為時間秒數)

可以看出雖然手機和電腦在場景複雜度低時沒有明顯變化，但在場景複雜度提高時手機速度會明顯慢許多，由 4.1 節的表中製程不同可以看出電腦運算速度較手機運算快速，且手機本身記憶體頻寬較低，所以在資料量過多時，因存取次數提高效能下降較快速，導致渲染時間變高。下圖為多層切割 K-D Tree 找出場景中最適合的層數加速效果，並比較手機和電腦在 CPU 上兩者之間加速倍率差異。

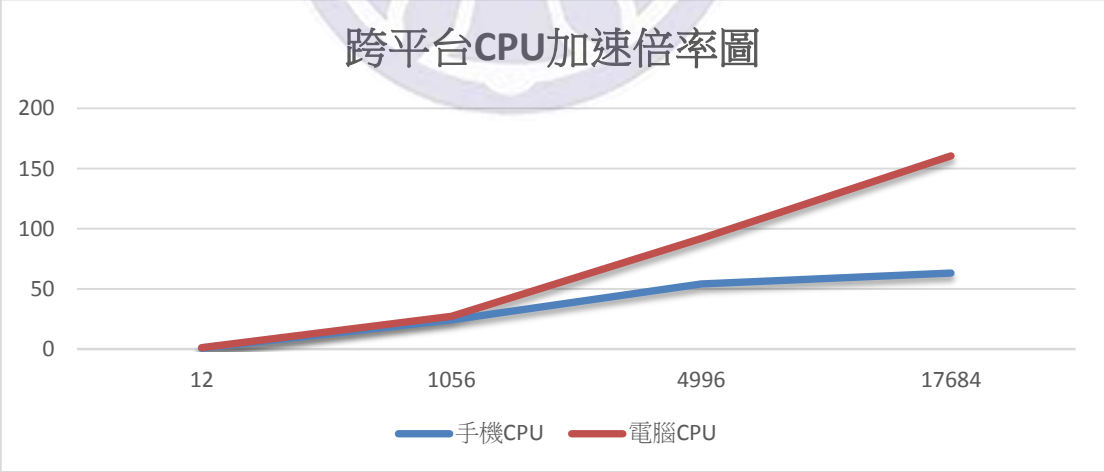

圖 13: 跨平台 CPU 加速倍率比較差異圖(x 軸為三角片個數 y 軸為時間秒數)

由圖中可以看出兩者在三角片較少時加速效果並沒有太大差異，但隨著場景複雜度提高可以看出電腦的加速效果會較手機要來的高。下圖為跨平台 GPU 加速圖，為配合手機 GPU Local Memory 不足只將 K-D Tree 層數訂為三層，來

比較兩者差異圖。

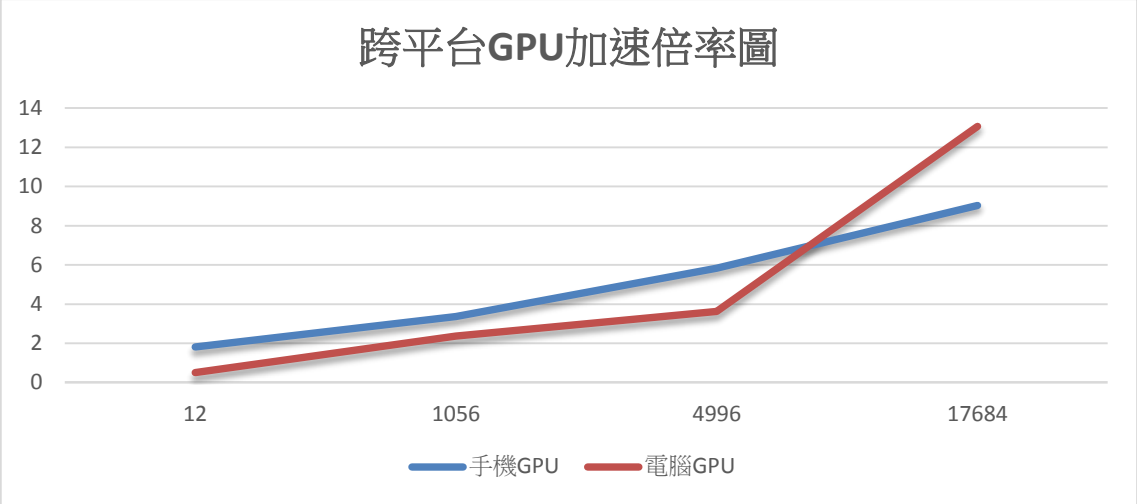

圖 14: 跨平台 GPU 加速倍率比較差異圖(x 軸為三角片個數 y 軸為時間秒數)

因為實驗用的手機在 GPU 的 Local Memory size 較小所以在建構 K-D Tree 時如果切的層數過高時，會因為走訪需要用到 stack 來紀錄走訪過的節點，而使用較多 Local Memory 導致數據錯誤。所以在 GPU 實驗時 K-D Tree 層數以三層為主，但 K-D Tree 層數提高效能是可以再加速的。當場景複雜度較低時，可以看出手機的加速倍率較電腦要來的高。因為手機的 GPU 和 CPU 共享一塊 LPDDR3 的記憶體快取，所以當 GPU 須從 CPU 搬資料時可以減少資料搬移時間，讓加速效果提高，但場景越複雜對於資料存取提高所以手機加速效果也跟著下降，而電腦因本身運算較快速所以加速效果就來的比手機好。且比較兩張圖表可以明顯看出電腦和手機的 GPU 加速倍率都較 CPU 來的慢，因為 GPU 會碰到分歧分支問題導致加速效果明顯下降。

如圖 15 至圖 20 所示，六張圖是以 Direct Lighting 方式呈現，其圖中的三角片個數不相同，可以看出光照在物體上所呈現的擬真效果。

圖 15：手機結果圖(三角片數 1056)

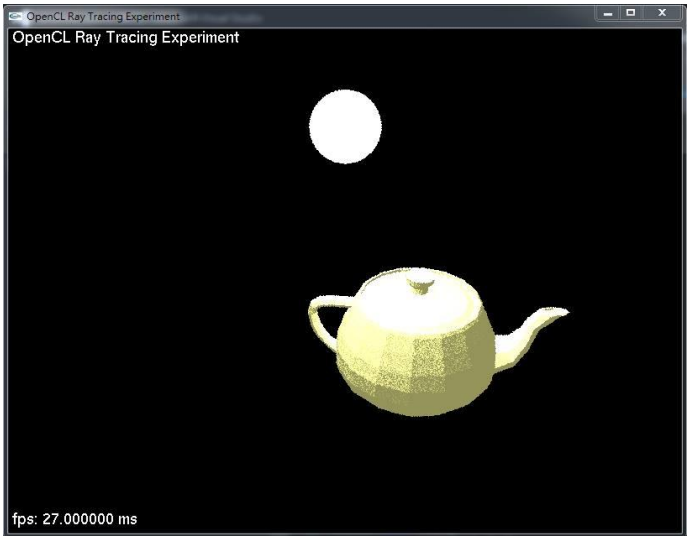

圖 16：電腦結果圖(三角片數 1056)

圖 17：手機結果圖(三角片數 4996)

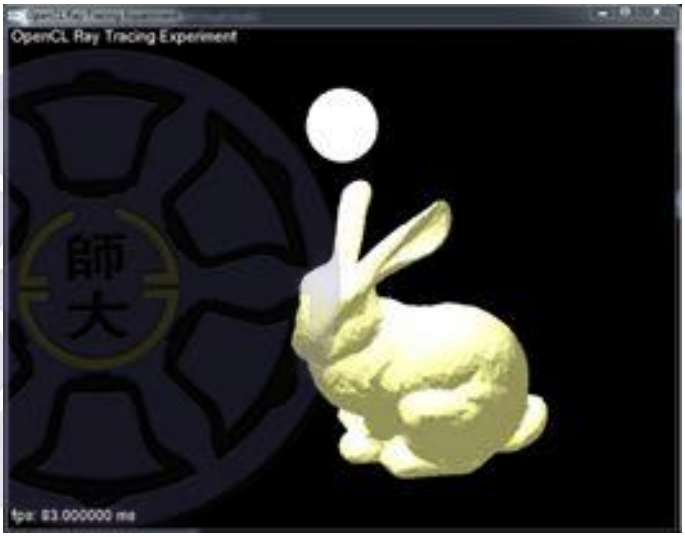

圖 18：電腦結果圖(三角片數 4996)

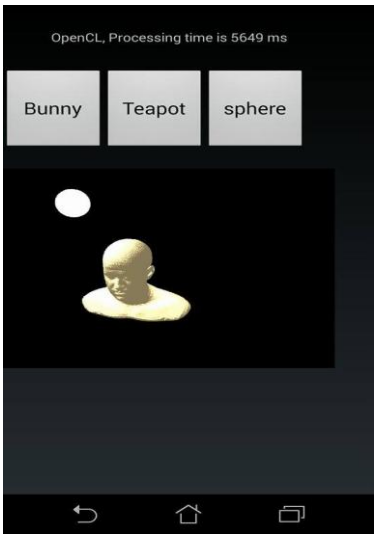

圖 19：手機結果圖(三角片數 17684)

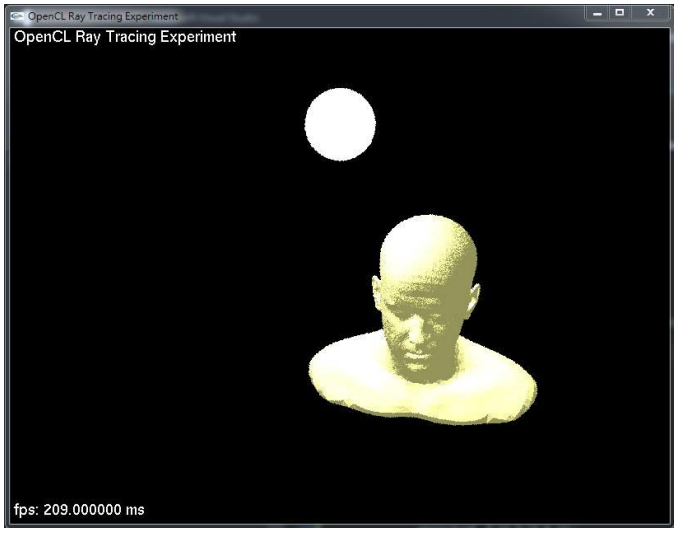

圖 20：電腦結果圖(三角片數 17684)

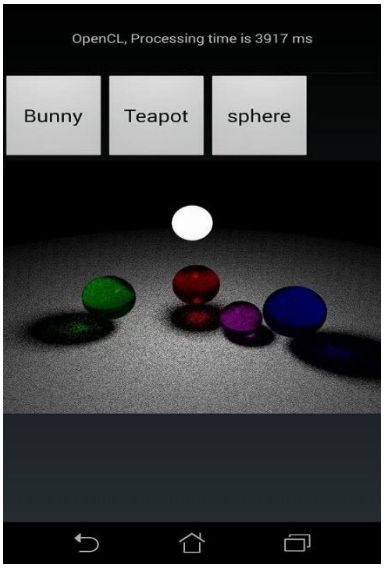

圖 21：手機上多次光線追蹤結果圖

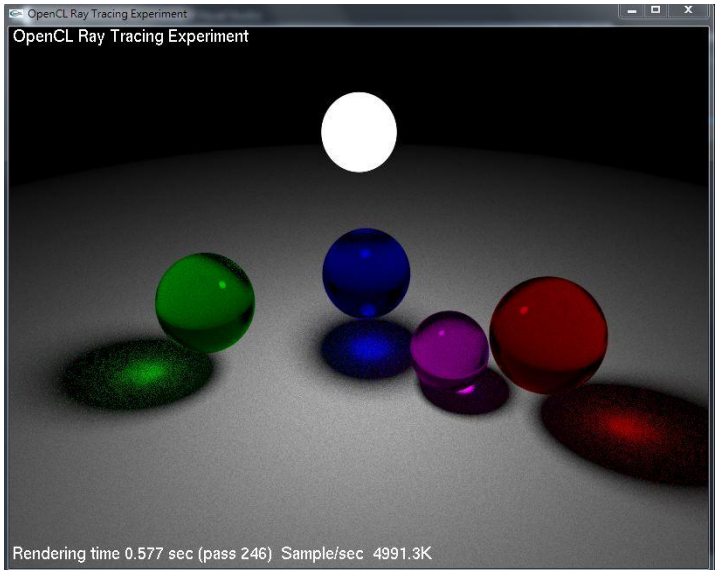

圖 22：電腦上多次光線追蹤結果圖

如圖 21、圖 22 所示，兩張圖是經過多次光線追蹤可以看出明顯的折射與反射現象，在玻璃球下的平面上可以明顯看出當產生反射光線時因與光源中間有遮蔽物，造成的陰影現象。而玻璃球本身的白點也是因反射光源所產生的亮點。而陰影上所產生的玻璃球顏色，則是光因折射穿過玻璃球所造成的焦散(caustics)現象。

下兩張表為最佳加速效果，本研究將 K-D Tree 切割層數提高，找出不同模型適合的層數，讓加速效果最快，下兩張表左邊為電腦加速效果右邊為手機加速效果。因手機 GPU 內存有限，這邊只顯示 CPU 的最佳化效果。

Size	CPU	build tree	GPU	build tree
1056	0.0425s	0.001s	0.026	0.001s
4996	0.1s	0.006s	0.050s	0.006s
17684	0.176s	0.029s	0.081s	0.029s

表 8: 電腦加速效率最佳化

Size	CPU	build tree
1056	0.213s	0.002s
4996	0.561s	0.006s
17684	1.901s	0.036s

表 9: 手機加速最佳化

第五章　結論與未來展望

5.1 結論

本研究的貢獻在於使用 OpenCL 加速蒙地卡羅光線追蹤於手機平台上，並比較和電腦上的差異，對於目前手機動畫或是遊戲並沒有人使用光線追蹤來實作，因為光線追蹤運算量過大需花費大量計算時間。但光線追蹤在計算機圖學上卻是具有高擬真度的渲染技術。本研究採用蒙地卡羅路徑追蹤來渲染 3D 場景，本研究也使用了 K-D Tree 做場景中的加速，可以明顯的看出使用加速結構後的明顯差異，對於較複雜的場景，其加速效果相對較好，也可以看出手機 GPU 上對於分歧分支的延遲。並找出因為手機上的 GPU Local Memory 不足的情況，導致無法再繼續加速。本研究還使用相同演算法在電腦上和手機上跑數據，並比較兩者之間的差別。在兩萬片的三角片計算上，手機 GPU 需花費 53.66 秒，CPU 則為 120.069 秒。當有了 K-D Tree 後速度明顯加快，GPU 只需 5.316 秒，CPU 需 9.831 秒。如果在沒有 Local Memory 的限制下 GPU 的速度會在相對提升。此研究使用蒙地卡羅路徑追蹤演算法讓實驗結果能在手機上產生良好的全域照明效果，實驗結果雖無法達到即時效率但也盡可能讓光線追蹤在手機上也能調適達成 Interactive Rate

更新畫面的效率，目前在網路上或其他論文中尚未看到有相關類似的研究，隨著手機的普及性和重要性來說相信對電腦圖學領域和手機是一個新的嘗試，對未來發展手機遊戲也能有良好的貢獻。

5.2 未來展望

本研究可以看出手機GPU的分歧分支問題導致加速倍率下降，也可發現因為K-D Tree採用堆疊的走訪方式讓手機上的Local Memory耗盡，也許可以嘗試其他加速結構來配合手機上的硬體規格，讓此技術能有更好的效能。本研究未來可以針對適合手機GPU特性來調整加速結構演算法，降低分歧分支所造成的效能損失，和減少Local Memory使用量讓研究可以達到即時的全域照明效果，來達到更真實的渲染。因本研究採用Asus PadFone S此台手機只支援OpenCL 1.1版本，但OpenCL 2.0版本將支援SVM的特性，也許未來如採用OpenCL 2.0特性會讓整體效能再更上層樓。

參考文獻

[Son 11] Son, K.C., Lee, J.Y., 2011. The method of android application speed up by using NDK. Awareness Science and Technology of 3rd International Conference. 382-385.

[Kajiya 86] Kajiya, J.T., 1986. The rendering equation. Computer Graphics. SIGGRAPH 1986, 143-150.

[Keller 12] Keller, A., Premoze, S., Raab, M., 2012. Advanced (quasi) monte carlo methods for image synthesis. SIGGRAPH 2012.

[Wang 14] Wang, Y., Liu,C., Deng,Y., 2014. A feasibility study of ray tracing on mobile GPUs. SIGGRAPH Asia 2014, Mobile Graphics and Interactive Applications.

[Zhou 08] Zhou, K., Hou, Q., Wang, R., and Guo, B., 2008. Real-time k-d tree construction on graphics hardware. SIGGRAPH Asia 2008, 126:1-126:11

[Foley 05] Foley, T., and Sugerman, J., 2005. K-d tree acceleration structures for a GPU ray tracer. In Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS Conference on Graphics Hardware, 15-22.

[Horn 07] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P., 2007. Interactive k-d tree GPU ray tracing. In Proceedings of Symposium on Interactive 3D graphics and Games, 167-174.

參考資料

[1] KHRONOS, 2011. OpenCL overview. https://www.khronos.org/opencl/
[2] JNI, 2011. http://zhuruenzhe2011-163-com.iteye.com/blog/1630531
[3] WIKI, Random number generation.
http://en.wikipedia.org/wiki/Random_number_generation

附錄

列出重要的 JNI 資料轉換程式碼:

```
extern "C" jint
Java_com_sony_openclexample1_OpenCLActivity_runOpenCL(JNIEnv* env, jclass clazz, jintArray info, jint weight,
                                                      jint height)
{

    unsigned int *  pixels;
    jint*i;
    i = env->GetIntArrayElements(info, 0);// info to i
    int w = weight;
    int h = height ;
    pixels = (unsigned int *)malloc(sizeof(unsigned int) *w*h+4);

    pixels[w*h+1] = i[w*h+1] ; // set turn 20
    pixels[w*h+2] = i[w*h+2] ;
    pixels[w*h+3] = i[w*h+3] ;
    if( pixels[w*h+2] == 0 ) {
      for( int k = 0 ; k < w*h ; k++ )
         pixels[k] = i[k];
    } //if
    openCLNR(pixels,w,h);
        //jint* i = env->GetIntArrayElements(info, NULL);


    for(int k = 0 ; k < w*h ; k++ ) {
        i[k] = pixels[k];// a 不透明
        /*i[k*3] = pixels[k] & 0x0000ff;
        i[k*3+1] = pixels[k] & 0x00ff00 >>8;
        i[k*3+2] = pixels[k]>>16;*/
    }

    i[w*h] = pixels[w*h];// time
    env->ReleaseIntArrayElements(info, i, 0);// i to info

    free(pixels);
    return 0;
}
```