

國立臺灣師範大學

資訊工程研究所碩士論文

指導教授： 張鈞法 博士

光線追蹤應用程式介面整合OpenGL程式設計



Ray Tracing API Integration for
OpenGL Applications

研究生： 賴威豪 撰

中華民國 一〇五年七月

摘要

光線追蹤(ray tracing)是電腦圖學中一種重要的 rendering 技術，藉由模擬運算光的折射、反射...等光學現象來產生高品質畫面，但需要大量的運算與時間來滿足，因此在以前的硬體設備上想要達成即時(real-time)的效果幾乎不可能辦到。現今的 GPGPU 硬體設計架構使得 GPU 可被用來做自定義的平行化計算，因此許多學者提出運用 CUDA 或 OpenCL 實作光線追蹤或路徑追蹤(path tracing)...等 physically based rendering 的方法，期望藉由 GPU 高效率的平行計算使光線追蹤的效能大幅度提升。然而以光線追蹤方法仍難以受到主流遊戲開發業者採用，因為市場上的產品大部分都以光柵化(rasterization) rendering 的設計為主，不容易在短期內發生改變，因此本篇論文想介紹一套類似於 OpenGL API 的整合設計，而底層 rendering 置換為光線追蹤或路徑追蹤...等 physically based rendering 的方法，幫助開發者使用現有的 OpenGL 應用程式原始碼並透過少許的修改就能獲得光線追蹤相關演算法 rendering 的畫面以及其帶來的好處。

關鍵字

三維圖學、光線追蹤、遊戲引擎、OpenCL、physically based rendering

Abstract

Ray tracing is one of the most important rendering technique in computer graphics. By means of simulating reflection and refraction of light, it generates an image with more photorealism than scanline rendering techniques. But the high computational cost of ray tracing is one of the algorithm main disadvantages. In recently years, the computing power of GPU has increased gradually, and the GPGPU applications model is more popular now. There are many scholars who introduce some physically based rendering methods with CUDA or OpenCL, in order to improve the image quality and increase rendering speeds. Nevertheless, there's no way ray tracing can be accepted by game development industry in the near future, because the design of the most products is based on rasterization rendering. In this research we introduce a ray tracing API integration for OpenGL applications, and we replace OpenGL rendering technique with ray tracing or other physically based rendering methods. So we can have advantages of ray tracing rendering technique by modifying few source codes.

Keywords

3D rendering, ray tracing, game engine, OpenCL, physically based rendering

誌謝

感謝我的指導教授張鈞法以及一齊在圖學實驗室度過了兩年時光的同學們，同期畢業的夥伴黃浩庭和唐昌宇，感謝他們在我撰寫論文時提出寶貴意見，彼此互相照應完成碩士學業，還有其他的夥伴蘇柏元、鄒毓偉、陳佑欣以及張祐瑄，感謝你們在實驗室的相處時光，無論是實驗室 meeting 或是討論課業，又或者午間閒聊和晚上一齊出去找晚餐地點，大家的關係不僅僅只是待在同一間實驗室的人而是能相互體諒相互較勁彼此最好的夥伴，在最後也非常感謝指導教授對我學業上帶來最好的教誨以及指引學術之路。

感謝父母親的栽培使我能順利完成碩士學業，還有我的弟弟和陪我成長已經 15 歲的老狗，沒有家人的精神支持做後盾我無法走到這一步，儘管父母親身體微恙，但身為家中的長子就算再怎麼樣的辛苦，對比於父母親將我拉拔長大，我覺得已經是微不足道了。而現在對我來說是值得紀念的里程碑，儘管在往後人生中可能是件不足為奇的事，希望今後我能不怕風浪繼續向前走，共勉之。

目錄

摘要.....	I
ABSTRACT	II
誌謝.....	III
目錄.....	IV
圖目錄.....	VI
表目錄.....	VI
第一章 緒論.....	1
第一節 研究背景.....	1
第二節 研究目的.....	1
第三節 論文架構.....	2
第二章 文獻探討.....	3
第一節 POWERVR OPENRL™.....	3
第二節 NVIDIA OPTIX™.....	5
第三節 MESA 3D.....	6
第四節 PBRT.....	7
第五節 OPENCL™.....	7
<i>Kernels and the OpenCL Execution Model.....</i>	<i>8</i>
<i>Memory Model.....</i>	<i>9</i>
第三章 系統實作.....	10
第一節 整合系統概述.....	10
支援何種版本 OpenGL 以及功能.....	11
系統流程架構.....	12

第二節	API 整合和擷取.....	14
	<i>The matrix manipulation functions</i>	15
	<i>Client-side vertex arrays</i>	17
	<i>Fixed-function lighting</i>	18
	OpenGL 固定功能支援以及前置準備.....	19
第四章	光線追蹤.....	20
第一節	演算法概述	20
第二節	實作細節	20
	計算交點測試.....	21
	光照模型.....	22
	場景加速結構.....	22
第五章	實驗與結果分析.....	23
	實驗環境與場景資訊.....	23
	驗證擷取功能.....	24
	Cornell Box 場景.....	24
	額外擴充功能—反射.....	25
	場景特性與效能測試.....	26
第六章	結論與未來方向.....	29
附錄 A	30	
圖片引用來源	31	
參考文獻	32	

圖目錄

圖 1: OpenR 架構	4
圖 2: OpenCL Memory model.....	8
圖 3: 整合系統流程	12
圖 4: 傳統光柵化管線步驟.....	14
圖 5: Projection 矩陣影響視景體.....	16
圖 6: 交點測試比較圖	24
圖 7: Cornell Box 比較圖	25
圖 8: 反射額外擴充效果	26
圖 9: 折射額外效果.....	26
圖 10: Sponza 場景(262,267 triangles).....	28



表目錄

表 1: 整合系統支援呼叫列表.....	18
表 2: 實驗環境與場景設定.....	23
表 3: 各場景效能列表	28

第一章 緒論

第一節 研究背景

受益於 GPGPU (general-purpose computing on graphics processing units) 硬體設計架構，光線追蹤可以充分利用 GPU 平行化的特性，過去認為無法達到即時效果的光線追蹤在現今終於可以實現。由於光線追蹤普遍被應用在電影工業以及 3D 動畫產業，這些商業應用的作品注重的是畫面的真實感與高品質的光影效果，往往透過 offline rendering 方式完成畫面或動畫，對於即時繪製的速度比較沒有特別要求，但近年來 3D 遊戲產業的蓬勃發展，消費者對遊戲的要求不但要有即時的互動性也希望能有高品質又逼真的畫面，因此近年來學者們紛紛提出光線追蹤的相關研究，希望在保持高品質畫面基礎下也兼顧繪製的速度。

第二節 研究目的

由於目前遊戲市場的主流以光柵化(rasterization)的開發為主，像是 OpenGL、OpenGL ES 以及 Direct3D... 等非常知名的 SDK，因此以光線追蹤為主的設計不易在短期內被開發者採用，其中的原因可能是需要重新熟悉新的 API 呼叫風格或者由於現有的硬體設備而受到限制，這些成本考量使得開發廠商偏向選擇主流的解決方案，因此本研究希望建立一套類似於 OpenGL API 的函式庫，並將原本

OpenGL 底層 rendering 的設計由光柵化置換為光線追蹤或路徑追蹤...等 physically based rendering 的演算法，幫助開發者以最小限度地修改現有的程式原始碼就能獲得光線追蹤或相關技術 rendering 出來的高品質畫面。

第三節 論文架構

本論文共有五個章節，第一章是緒論，概述研究動機、研究目標和接下來會提到的整合設計；第二章是文獻探討，稍微介紹以光線追蹤為主的 API 和相關應用；第三章為系統實作，詳述本論文提出的整合方法，關於整合的對象、使用的技術以及整體架構的內容；第四章介紹本論文實作光線追蹤的方式；第五章則是實驗結果分析並比較使用整合系統後的差異；第六章為本篇論文的結論，並探討未來能改進的方向。

第二章 文獻探討

本論文提出的整合設計期望擁有相容性、通用性以及可擴展性，對 OpenGL 應用程式的整合盡量讓差異減至最低，並保留 OpenGL 應用程式可跨平台的特點和嘗試維持即時運算的速度，另外也保留可以擴充新功能的彈性。接下來是分析一些現在市面上知名的圖形 SDK，以及本論文提出的設計會利用的資源和技術。

第一節 PowerVR OpenRL™

PowerVR OpenRL™[1]提供可編程的應用程式介面來加速光線追蹤，能用於開發圖學相關領域或是非圖學領域(像是物理模擬)的應用程式，有類似於 OpenGL shader 程式碼來控制光線追蹤的邏輯，對光線追蹤的流程有完整的定義並且能透過 OpenRL shader 來自定光線追蹤的行為，而 OpenRL 總共有三種類型的 shader 程式，分別為 Vertex shader，負責場景中所有頂點資料的運算；Frame shader，主要是設定畫面中的光線與採樣方式；Ray shader，定義光線如何攜帶顏色資訊與能量強度。OpenRL shader 程式雖然效仿 OpenGL shader 的呼叫風格，但部份的流程仍有差異性，像是必須定義光線採樣來建立畫面累加方法，這種作法在 OpenGL 中並沒有相似的流程，因此兩者實際相比，OpenGL 提供了以光柵化 rendering 的 API，而 OpenRL 則提供光線追蹤的操作，所以根據底層的不同也會影響程式邏輯的設計。

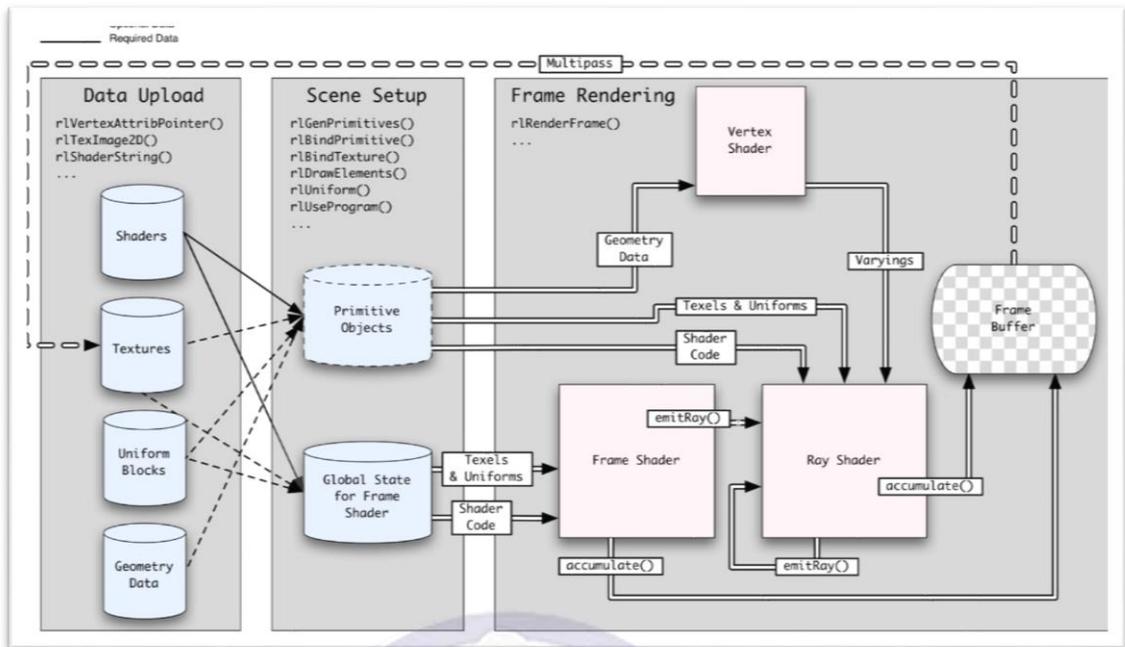


圖 1: OpenR 架構

Data Upload 階段收集整個場景的資料，Scene Setup 階段設定傳送給 shader 的參數，最後 Frame Rendering 階段則是三種 shader 執行時傳遞的順序。圖片引用(1)

OpenRL SDK 整體架構共分為 Data Upload、Scene Setup 和 Frame Rendering 三個階段參考圖 1 內容，所有的場景資料必須在 Data Upload 設定完成，由於光線追蹤在產生畫面時會和場景內物件做交點測試，因此 OpenRL 自然也會有這樣的規定，而接下來 Scene Setup 階段是設定要傳給 shader 的參數，包括被綁定的場景資料、貼圖和 UV 座標、取樣器或是唯獨區塊...等，最後 Frame Rendering 會執行上段落提到的三種 shader 並產生出繪製的畫面。

總體來說 OpenRL 讓開發自由地設計光線追蹤的邏輯控制，但必須學習它特有的 shader 邏輯，因此以 OpenGL 開發者的角度來說必須花時間來熟悉新的 shader 程式邏輯無疑是一種額外的成本，而下一章將會介紹本論文提出的整合設計，其優勢在不需要提供額外的 shader，程式碼僅需要少許變更或修改就能套用運行。

第二節 NVIDIA OptiX™

OptiX[2]是 NVIDIA 開發的光線追蹤引擎 SDK，採用 NVIDIA CUDA GPU 運算架構，是一套很成熟的商業化解決方案，常見的專業繪圖軟體譬如 Autodesk、Maya 和 Lightworks...等均搭載 OptiX 的 plug-in，另外電影公司 Pixar 在 SigGraph 2013 也展示搭載 OptiX 的即時燈光工具(realtime lighting tool)和旗下電影「怪獸大學」使用的動畫引擎也使用 OptiX 來開發，這些應用都藉由 NVIDIA GPU 的繪圖加速優勢實現即時畫面繪製並且模擬真實物理反應特性。

雖然 OptiX 有強大的性能與廣泛的應用，但是必須搭配特定廠商也就是自家的硬體才能運作，本篇論文提出的整合設計期望能讓 OpenGL 應用程式獲得光線追蹤的體驗，因此背後整合的工具也必須保留跨平台的特性，這也是整合設計上會與 OptiX 相異之處。

第三節 Mesa 3D

Mesa 3D[3]最初是由 Brian Paul 設計的專案，而現今已成為 OpenGL 規範下開源的實現，可以被用在不同平台裝置及開發環境進行軟體模擬或是實際硬體加速，而本論文提出的設計理念和 Mesa 3D 類似，希望能讓 OpenGL 開發者在熟悉的程式呼叫風格下獲得光線追蹤產生的畫面。

由於 Mesa 3D 為實現多平台上皆能使用類似 OpenGL 呼叫風格的應用程式介面，因此需要提供各種平台相對應的實作，可以想像在維護方面肯定非常繁瑣，一旦 OpenGL 版本更新，Mesa 3D 也勢必需要配合相對應的版本更新。相較於本篇論文的做法，與 Mesa 3D 最不同的地方是僅提供 OpenGL API 中繪製圖形的最少指令需求，並不像 Mesa 3D 針對所有的 OpenGL API 進行支援，而其中最大的原因在於 OpenGL 是光柵化為基底，而本論文的基底則使用光線追蹤，因此對 OpenGL 所有的 API 必定無法完整地對應。還有其他不同處是某些 OpenGL 功能可以直接被繼承使用，譬如說 Matrix 系統 OpenGL 已經有 fixed functionality 維持，ModelView Matrix 及 Projection Matrix 可以直接從中進行擷取而不必重新實作功能，但遇到關鍵繪製影像的函式就一定要提供整合方案，而最常見的兩種分別是 `glDrawArrays` 和 `glDrawElements`，OpenGL 應用程式使用這些呼叫來通知設備進行繪圖指令的佇列，所以針對這類的函式要有相對應的整合方案。

第四節 PBRT

Pharr and Humphreys [4] 著作的 *Physically Based Rendering: From Theory to Implementation* (以下稱 PBRT) 是一本把現今計算機圖學領域中幾乎所有的高階技術從理論到實作完全具體呈現出來，書籍內容完整又有系統地闡述圖形 rendering 流程，由它的實作程式所生成的圖片非常具有真實感絲毫不遜色其他同性質的商業軟體，另外開源軟體 LuxRender 的系統也參考 PBRT 一書附帶的原始程式碼，並且也成為市面上一些知名建模軟體的 plug-in。

PBRT 一書非常值得研讀與參考，附帶的程式碼也可以幫助讀者更容易了解理論是如何被實作，但由於 PBRT 專案是給 CPU 程式碼並且是非即時的设计，而本論文的光線追蹤則是希望設計成多平台架構皆能運行的形式，即使應用層面上不同，PBRT 的光線追蹤架構仍非常值得學習與參考。

第五節 OpenCL™

OpenCL[5](Open Computing Language) 是一套跨平台的程式撰寫框架，目前由 Khronos Group 負責制定架構，此異構平台可以由 CPU、GPU、DSP 或 FPGA... 等硬體加速器組成，在現今已是許多家廠商共同支持的多核心計算標準。本論文的光線追蹤設計會採用 OpenCL 撰寫，原因在於光線追蹤的計算量非常龐大，為了達到即时的速度勢必需要 GPU 的平行計算能力，而且不受開發平台的限制對

整合系統來說也是一個非常有利的條件，而下面稍微敘述由 Gaster, et al. [6]所撰寫的 *Heterogeneous Computing with OpenCL* 書籍，內文介紹到 OpenCL 兩個重要的系統架構，分別為 Execution Model 以及 Memory Model，在第四章會說明整合設計如何配合其制定的規範完成光線追蹤演算法。

Kernels and the OpenCL Execution Model

OpenCL 程式可分為 host 端和 device 端，前者 host 端指的是一般應用程式且負責掌握所有 OpenCL 運作機制，後者 device 端則是實際執行 kernel 程式的裝置，host 端可以啟動多個 kernel 程式進行不同的任務，並且也能同時指派不同的裝置運行，OpenCL 平行計算以及多裝置設計是現今最被看好的優勢之一。

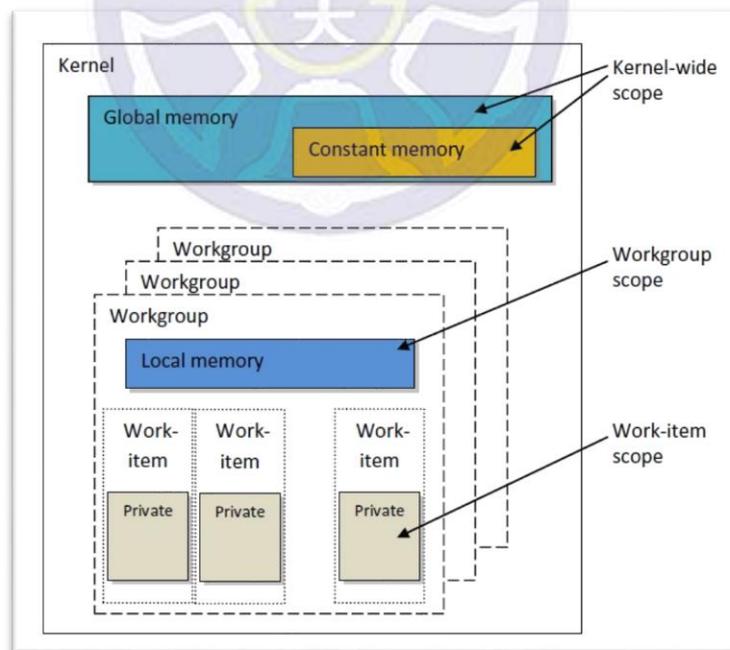


圖 2: OpenCL Memory model

OpenCL 記憶體階層示意，由上至下分為四層 Global、Constant、Local 和 Private，程式內變數的可視範圍也依照記憶體階層不同會有所限制。圖片引用(2)

OpenCL 的 kernel 程式邏輯是以 C99 為基底所構成，所以和 C 語言幾乎沒有太大差別，但注意的是不支援遞迴呼叫以及每個變數要遵守特殊的記憶體階層架構，另外 kernel 程式在 device 端是以 SIMT(Single Instruction Multiple Thread)模式運行，裝置會同時執行複數的 kernel 程式而其中每個單位被稱為 work item，並且多個 work item 會被集結成群組稱為 work group，這些工作量以及分組的設定可以根據不同裝置的運算特性做調整進而影響平行化的效率。

Memory Model

為了統一不同裝置上的差異，OpenCL 規範了記憶體階層架構如圖 2 所示，由上到下分別是全域記憶體(global memory)、唯獨記憶體(constant memory)、局部共享記憶體(local memory)和私有記憶體(private memory)四個階層。

全域記憶體是所有的 work item 都能存取的記憶體範圍，在階層中容量最大但是存取速度則較慢，host 端的資料傳送到 device 端做計算通常都會利用此記憶體階層。唯獨記憶體是全域記憶體的一部分且存取速度稍微快一些，但是此記憶體階層的資料無法做修改。

另外局部共享記憶體的可視範圍以 work group 為單位，根據不同硬體的規格此記憶體存在的位置也不同，但通常會被對應到 on-chip 的記憶體上，此階層的存取速度會比全域記憶體還快。私有記憶體存取範圍則是以每個 work item 為單位，相當於硬體暫存器的位置，容量最小並且是記憶體階層中存取速度最快的。

第三章 系統實作

此章節將闡述本論文提出的光線追蹤應用程式介面整合 OpenGL 程式設計，再一次強調本論文的目標在建立一套類似 OpenGL API 的函式庫，並讓底層 rendering 置換為光線追蹤或路徑追蹤...等 physically based rendering 的繪圖方法，因此函式庫必須盡可能的配合 OpenGL 參數傳遞方式，讓使用者以最小限度的修改程式碼後就能獲得底層置換為光線追蹤所繪製的畫面。

第一節 整合系統概述

本系統設計可分為兩大部分，第一部分是函式呼叫整合系統，負責將 OpenGL API 轉接到由自己維護的函式庫內，並且蒐集使用者呼叫 OpenGL 函式後的參數加以轉換成系統內部格式，接下來等待同步指令下達後再啟動光線追蹤的計算。第二部分是採用了 OpenCL 開發的光線追蹤繪圖程式，就如同第二章介紹 OpenCL 的段落一樣，有負責掌握運作機制的 host 端以及實際執行平行化運算的 device 端。

OpenGL 應用程式使用本論文的整合系統函式庫其實非常的簡單，只需在原始程式碼內引用一個標頭檔就可以自動將原本的程式邏輯導向到整合系統內，接下來應用程式就會改以光線追蹤來繪製，此時應用程式的底層已不再是 OpenGL 的光柵化，而是由整合系統的光線追蹤繪製所產生的畫面。

支援何種版本 OpenGL 以及功能

OpenGL 3.0 版本[7]制定的規格，不再支援一指令組繪製一次圖形的用法，此方式又稱為圖形立即繪製模式(immediate mode primitive rendering)，取而代之的是鼓勵採用 VAO (vertex array objects)和 VBO(vertex buffer objects)...等將資料放入緩衝陣列後再一起傳遞給 OpenGL driver 藉此減少與繪圖裝置溝通的次數，可以降低 IO 延遲並提升繪圖效率，而這樣的改變也對整合系統抽換底層從光柵化管線(rasterization pipeline)改為光線追蹤的方法十分有利。

Legacy OpenGL 泛指的是 3.2 版本之前，程式風格以固定功能(fixed-function)的呼叫模式為主，其中有 lighting、materials、color materials、shadow mapping 和 bump mapping，優點在於過去廣泛使用的繪圖技術皆被內建在 legacy OpenGL，但是現今 GPU 硬體架構突飛猛進，制式化的固定功能已經沒辦法滿足消費者的口味，因此出現了 shader 語言讓光柵化管線更自由地實現自定義的光影效果，另一個原因是所有的固定功能呼叫模式所產生的效果都可以由撰寫 shader 來模擬實現，因此官方制定的規格在 OpenGL 3.2 版本以上也決定移除這些固定功能並且要求 OpenGL 程式必須提供 shader 程式才能運行。

即使如此，現今仍有一些 OpenGL 應用程式採用這樣的寫法，特別是基礎計算機圖學教學課程會介紹 OpenGL 設計的演進以及技術的介紹，因此本篇論文以整合這些固定功能呼叫模式為目標，並帶來光線追蹤為底層的繪製。

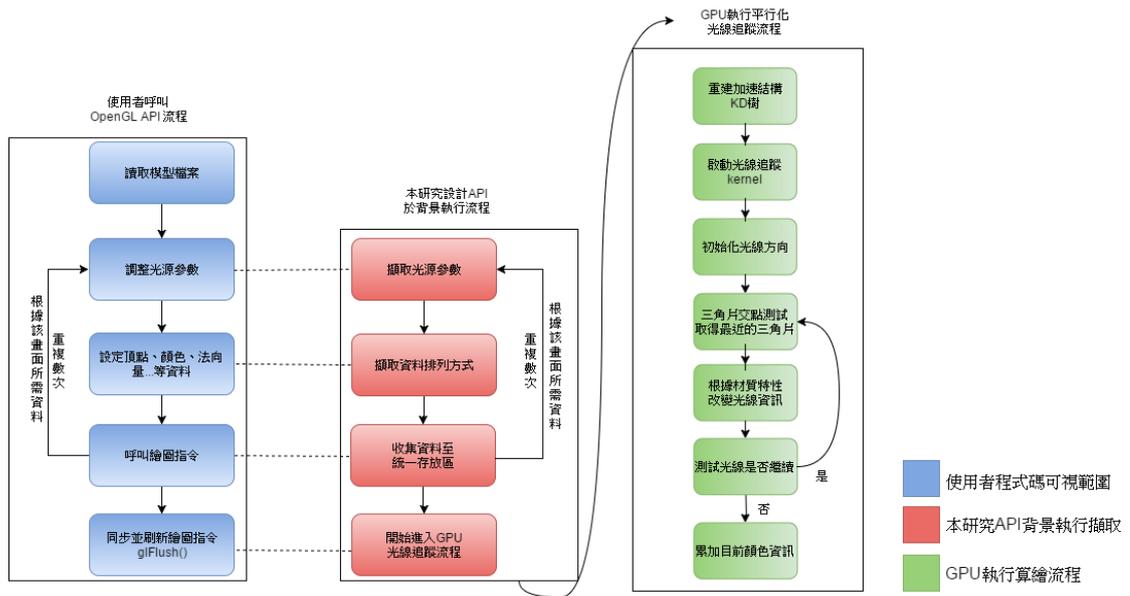


圖 3:整合系統流程

本論文提出的光線追蹤應用程式介面整合 OpenGL 程式設計流程一覽，左邊區塊代表常見的 OpenGL 應用程式產生畫面流程，右邊區塊是光線追蹤運作的流程，原本的光柵化會在最後同步繪圖指令的階段改以啟動光線追蹤，並將畫面結果返還給 OpenGL 顯示。

系統流程架構

本段落以圖 3 整合系統流程圖介紹 OpenGL 應用程式繪圖實際範例，圖中左邊使用者程式碼可視範圍區塊代表開發者撰寫的 OpenGL 程式碼，通常會包括讀取模型檔案、訂定光源細節、設定緩衝數據資料(頂點、顏色和法向量...等)和呼叫繪圖指令。

同時這些 OpenGL API 會被整合系統導向到內部呼叫，並且擷取其中的參數資料作為光線追蹤前置準備，而這些動作使用者完全不用額外的操作，就像是在執行原來程式碼一樣，此部分是圖 3 中間 API 背景執行擷取區塊對應的工作。

因為前兩個區塊相互的對應是處於同時進行的關係，但是仍有最關鍵的問題是在流程最下方繪圖指令下達後何時該執行光線追縱，由於光柵化的特性每個繪圖指令被呼叫時，繪圖裝置可能會立即執行也有可能在佇列狀態，換句話說場景資料會被分段進行 rendering 再根據深度測試覆蓋原本被擋到的圖形，但是光線追縱直到確定場景資料已經準備完成，才能開始對每個物件做交點測試計算畫面。

整合系統的設計必須考慮兩者差異性然後找出共通點來啟動光線追縱，因此決定以 OpenGL 用來刷新繪圖指令的 `glFlush` 函式作為整合系統啟動光線追縱地同步信號，雖然上段已提到繪圖指令執行可能有不一致行為，但現今各家廠商的驅動設計會保證繪圖指令一定在有限時間內被排入裝置，這樣同步指令對於原本光柵化的程式並沒有太大的影響，但對本篇論文提出的整合系統來說是最佳的同步時機。

當 OpenGL 應用程式執行了同步繪圖指令後才會真正啟動光線追縱流程，也就是進入如圖 3 右邊的 GPU 執行平行化任務區塊，其中光線追縱又有更細部的演算法流程，這部份會在第四章完整的敘述。另一方面，更新畫面緩衝這個步驟則有比較特殊的處理，由於 OpenCL 提供了一種互通性(interoperability)機制讓其 image 型態與 OpenGL texture 型態結合再一起，代表畫面緩衝資料不用從 OpenCL 環境傳回 host 端再傳回 OpenGL 環境來顯示，而是讓處於 GPU 記憶體內的緩衝資料直接讓 OpenGL 來使用，因此可節省傳遞延遲並提升畫面更新速率。

第二節 API 整合和擷取

本節將會介紹如何把 OpenGL 3.0 版本固定功能以及矩陣功能整合至 OpenCL

語言撰寫的光線追蹤，看下圖 4:傳統光柵化管線步驟。

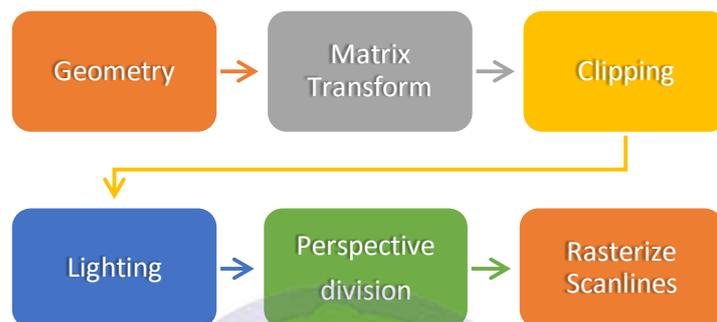


圖 4:傳統光柵化管線步驟

首先在 Matrix Transform 時處於 3D 空間座標的物件會轉換到螢幕坐標系上，這個轉換階段用到的矩陣是投影(projection)矩陣、視覺(view)矩陣和模型(model)矩陣的乘積(有固定的運算順序)，再來 Lighting 階段會計算 Diffuse、Ambient 和 Specular...等標準光照公式，最後才開始執行光柵化相關作業。

大致敘述了傳統光柵化管線後，可以發現上述點名的步驟共通點都是應用使用者設定的數值來計算，既然這些數值是由人所決定而不是限定光柵化才有用，那麼就可以直接擷取數據並由整合系統內的光線追蹤來利用。

以下將會分段來介紹如何完成 API 擷取，而這些工作與光線追蹤 host 端前置息息相關，並為了轉換成 kernel 端設計的格式做準備。

The matrix manipulation functions

Legacy OpenGL 提供了矩陣系統用來設定光柵化階段所需要的座標轉換，有 Projection 和 ModelView 兩種型態，後者因為視覺矩陣和模型矩陣可以反向操作來達成相同目的，所以 OpenGL 將兩者視為相同型態。其他還有針對 3D 空間的操作呼叫，像是位移(Translate)、旋轉(Rotate)和縮放(Scale)...等，也都是矩陣系統提供的功能，每次的操作都是對現在指定的矩陣型態才有效果。

矩陣系統的設計是用來描述處於 3D 空間物件的位置，不僅在光柵化流程中使用，光線追蹤對物件的定義也能套用此系統，最重要的就是上段落所說的兩種型態的矩陣，而整合系統也必須獲得這些矩陣資訊。

幸運的是 Legacy OpenGL 可以透過呼叫 glGet 函式來取得內部狀態，因此前面段落提到的矩陣操作不需要再重新實作一遍，換言之不必將這些矩陣操作運算重新導向到整合系統內，因此唯一需要注意的是何時才要擷取矩陣，這個答案也呼之欲出，回到圖 3 的整合系統流程圖可以發現繪圖指令對應的是整理資料並放入統一存放區，並不是執行光線追蹤流程，整合系統這樣的行為是因為一定要確定該畫面在繪製以前場景內容要全部設定完畢，因此擷取矩陣的真正時機就是每一次的繪製指令呼叫，並且矩陣將不會被用在光柵化管線而是被整合系統的光線追蹤作為其中的一筆繪圖數據。

除了上頁敘述的矩陣功能，OpenGL Utility Library (GLU)也廣泛地被使用在 OpenGL 應用程式中，譬如說 gluPerspective 以及 gluLookAt，前者的功能是指定觀察視景體(frustum)並定義出 3D 世界坐標系，而後者是指定觀察位置和方向，雖然兩者沒有透過矩陣功能來改變數值，但實際上 Projection 和 ModelView 矩陣皆會因為呼叫兩者函式而受到改變，由於 GLU 提供更方便的呼叫模式，通常被使用的次數不會少於原本矩陣功能，所以需要整合常用的 GLU 函式。

比較特別的地方是 Projection 矩陣對於光線追蹤來說比較沒有用處，但指的是矩陣本身而不是功能性，其實真正有用的是影響該矩陣的參數，如圖 5 所示，左邊代表原本未套用 Projection 矩陣示意圖，藍色是物體而紅色是視景體，可以想像的紅色的範圍即是使用者看到的畫面。接下來再看看右邊套用了 Projection 矩陣的示意圖，物體會受到視景體遠近平面的影響而產生形變，而紅色視景體的部分則壓縮成螢幕的矩形，也就是說畫面更像真實世界的表現了。

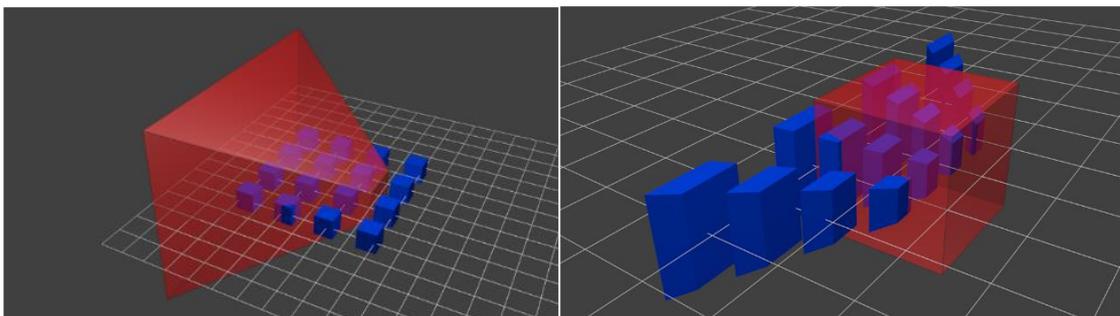


圖 5: Projection 矩陣影響視景體

左邊是原本 3D 空間物件分布，右邊是套用了矩陣後的變化，由於針對視景體做修正，藍色物件會因距離切平面的遠近而產生形變，有如現實世界中觀察物體的現象。 *圖片引用(3,4)*

拉回到剛剛提到的矩陣問題，光線追蹤需要的是如何在 3D 場景中定義每個像素取樣的方式也就是光線出發的方向以及取樣間格的設定，所以 Projection 矩陣的參數像是視野角度(field of view)、長寬比例(aspect ratio)...等才是擷取的重點。

Client-side vertex arrays

VBO 是 OpenGL 為了更有效率地繪圖而將物件緩衝再送入裝置去運算，藉此減少來回呼叫延遲，而相對於現今 modern OpenGL(3.3 版本以上)將 VBO 應用在 shader，以前 legacy OpenGL 的 VBO 是以 client-side 的概念用在光柵化固定管線。

在 client-side 的概念中必須先呼叫 `glEnableClientState` 來啟動某類別的服務，舉個例子來說，若現在 VBO 包含了頂點資料以及顏色資料，那麼 client-side 必須要啟動 `GL_VERTEX_ARRAY` 和 `GL_COLOR_ARRAY` 兩個列舉型態，這樣在繪圖管線進行時此 VBO 才會抓取頂點和顏色作為輸入。

上述提到 client-side 掌控哪些服務狀態要被啟用，而對於整合系統來說也要獲得這些狀態的資訊，因此採取的作法是重新實作相同功能，唯一改變的僅是從光柵化管線變成給光線追蹤來做使用。

除了 client-side 系統之外，VBO 也是整合系統要獲取的對象，無論是頂點、顏色或法向量在固定功能呼叫系統中都有專門處理的函式，換個方式來說，每當 VBO 建立後，OpenGL 不會知道使用者會把它拿來做什麼用途，而且資料設定的方式也以宣告緩衝區的大小以及起始指標來複製指定記憶體區塊，最後直到呼叫

了剛剛提到的專門處理函式像是處理頂點的 `glVertexPointer`，這樣 VBO 才會被用來做指定的頂點分析，根據位元的位移長度來抓取每筆頂點資料在記憶體區塊的位置，另外顏色和法向量設定也是相同的道理。

本整合系統對於 VBO 的處理方式和 client-side 一樣，都是將原本提供給光柵化管線的功能，轉移到系統內的光線追蹤使用，但並不會立即啟動繪製，而是將資料放在統一存放區。

Fixed-function lighting

Legacy OpenGL 提供了光照的固定功能，透過 `glLightfv` 來指定光源的屬性，最常見的設定是顏色資訊和位置，而根據 `GL_LIGHT0` 至 `GL_LIGHT7` 的參數可以對七個光源做個別設定，原則上系統對光源整合也很直觀，就如前面段落的做法類似，每當呼叫了光源的設定就將函式導向整合系統做數值紀錄，並作為光線追蹤前置作業的一環。

Buffer	Rendering	Capability	Lighting	GLU	Extension
<code>glGenBuffers</code>	<code>glVertexPointer</code>	<code>glEnableClientState</code>	<code>glLightfv</code>	<code>gluPerspective</code>	<code>rtMaterialEXT</code>
<code>glBindBuffer</code>	<code>glColorPointer</code>	<code>glDisableClientState</code>		<code>gluLookAt</code>	<code>rtBuildKdTreeEXT</code>
<code>glBufferData</code>	<code>glNormalPointer</code>	<code>glEnable</code>			
	<code>glDrawArrays</code>	<code>glDisable</code>			
	<code>glFlush</code>				

表 1: 整合系統支援呼叫列表

Extension 部分並不是 OpenGL 內建的函式，但由於光線追蹤可依照物體的材質改變光線的行為，實現反射和折射的效果，因此新增這一類的擴充函式。

OpenGL 固定功能支援以及前置準備

上頁表 1 是整合系統目前支援的所有固定功能函式列表，從左至右所列出的項目在先前的段落皆有詳細介紹，這裡再次強調本篇論文的目標，即是希望能提供一個光線追蹤整合函式庫使得設計 OpenGL 應用程式時可以方便的使用光線追蹤作為繪製畫面的方法，所以到目前為止的論文內容都是討論如何整合 OpenGL 應用程式中常見的函式，總而言之，整合系統就是為了讓這些函式所提供的參數以及設定數據都成為光線追蹤 kernel 端的輸入，而得到的輸出結果就是顯示在螢幕上的畫面。另外，上頁表 1 中 Extension 的部分列出的是本整合系統提供的擴充函式，因兩者繪圖方式的差異勢必會有原本光柵化方法沒有的功能，而這裡提到的函式後面會再提到更進一步的介紹。

第四章 光線追蹤

本論文第三章介紹整合系統的功能以及處理函式的方法，而第四章將會介紹整合系統另一個重要的部分，也就是光線追蹤的 kernel 端程式，接下來本論文會分成數個小節討論所本系統使用的光線追蹤演算法細節。

第一節 演算法概述

光線追蹤是一種模擬光線在三維空間中根據物體的表面特性所發生的互動行為，Whitted [8]提出了 Whitted Ray Tracer 模型來定義光線追蹤的流程，其演算法的順序如下，首先以觀察者位置與成像平面的各個像素點形成方向向量，而這些向量可以產生等同於對畫面取樣的光線。接下來光線透過與三維空間的物體作交點測試而得到碰撞點，再利用碰撞點檢查與光源之間是否有物體遮蔽，這個階段產生的是陰影光線並且利用光照公式計算出成像的顏色資訊。最後根據物體的表面特性，從交點出發並應用物理的反射或折射公式再產生新的光線繼續進行新的一輪追蹤，而達到終止條件時會將光源攜帶的資訊回傳成為像素點的顏色。

第二節 實作細節

回到圖 3 整合系統流程圖，在圖右邊是平行化光線追蹤流程，第一步的動作，會針對場景資料製作加速結構，進行光線追蹤時略過不必要的交點測試，可以讓

運算效率大幅提升，由於光線追蹤昂貴的運算成本，加入場景加速結構已經是不可或缺的一環。至於流程後面的部分，本整合系統的設計參考 Whitted Ray Tracer 模型，唯一不同之處是 OpenCL 語言不能使用遞迴函式呼叫，所以必須以迴圈作為代替，滿足原本模型中遞迴的光線產生。

本論文的光線追蹤實際程式以 OpenCL 語言撰寫，並讓整合系統導向和擷取原本 OpenGL 函式的參數，這些已經在第三章充分敘述過，擷取出的資訊會被用來作為光線追蹤的前置準備，而目前實作的方式是將場景資料以 OpenCL 全域記憶體的分層送入裝置，另外只有光源數據是放入私有記憶體，因為對應 OpenGL 固定功能的定義，光源一定會有 `GL_LIGHT0` 至 `GL_LIGHT7` 共 8 個數量，因為數量不多所以光源的數據直接讓所有的工作單位各自持有。

計算交點測試

雖然 legacy OpenGL 固定功能呼叫模式支援點、線、多邊形...等其他物體型態，但後來在 modern OpenGL 後幾乎全部被移除並且只留下三角形為基礎物件型態，所以本論文的光線追蹤以三角形作為主要支援場景型態。

交點測試對於光線追蹤來說佔了很大比例的運算資源，所以交點測試的演算法好壞會影響運算效率，因此利用 Möller and Trumbore [9] 提出的演算法(詳細的程式碼見附錄 A)，可以快速的算出光線與三角形之間的交點，而且不需求出三角形在三維空間所形成的平面方程式，然後才讓光線向量與平面方程式求解。

除了三角形之外，光源採用光線對球面的交點測試，因為 OpenGL 固定功能的光源是點光源，因此可以將光源設計成極小的球面來模擬點光源的效果。

光照模型

當交點測試後確定了光線擊中的物體，並且實現成像的顏色，而這個光照模型最被人所熟知的是 Phong [10]提出的 Phong lighting 模型，此模型定義了粗糙表面的散射光(diffuse)、光滑表面的鏡面反射光(specular)以及場景中固定量值的環境光(ambient)，其中散射光以物體到光源的向量和觀察點到物體的向量做內積並乘上物體表面的顏色所形成，本整合系統也參考此模型的散射光作為光照的作法。

場景加速結構

雖然光線追蹤已經使用 OpenCL 操作 GPU 做平行化加速，但其運算量仍相當龐大，因此光線追蹤必須減少對場景資料的交點測試次數，而這種加速結構中最常見的是 KD 樹與 BVH 樹，本系統參考 PBRT[4]書中的 KD 樹加速結構，根據場景資料中三角形物件的分布對三維空間分割出樹狀節點並記錄其中擁有的三角形。而在光線追蹤時先對樹狀結構進行探訪，根據樹狀節點每次進行二分的路徑選擇，最後只對光線會經過的葉節點其擁有的三角形進行交點測試，這樣其他光線到達不了的空間所包含的三角形就完全不會做交點測試，因此可以大大的減少不必要的運算。

第五章 實驗與結果分析

實驗環境與場景資訊

本章實驗將會以相同的 OpenGL 應用程式原始碼，分別執行原本光柵化方法以及使用整合系統後光線追蹤方法，由於本論文系統設計的便利性，在原始碼文件中引用整合系統函式庫的標頭檔即可達成底層 rendering 置換。以下表 2 是實驗環境的詳細資訊：

開發環境	
作業系統	Windows 10
編譯工具	Visual Studio 2015
OpenCL 版本和分析工具	Intel OpenCL 1.2 SDK
GPU 1: Intel® Iris™ Graphics 6100	
核心數量	48
頻率	300 MHz (up to 1100 GHz)
運算能力	844.8 GFLOPS
GPU 2: NVIDIA® GeForce GTX 970	
核心數量	1664
頻率	1050 MHz (up to 1178 MHz)
運算能力	3494 GFLOPS
場景資訊	
解析度	800 × 600
特性	靜態場景、可移動光源和可移動視角

表 2: 實驗環境與場景設定

驗證擷取功能

在整合系統開發初期，為了驗證 OpenGL 應用程式內 VBO 相關函式可以正確地被整合系統所擷取，以下設計了只有簡單的兩個三角形的場景，並且讓場景內的三角形為白色而背景是黑色，如圖 6 所示，實驗結果無論是由 OpenGL 的光柵化產生(左圖)或是由整合系統的光線追蹤產生(右圖)，兩者的畫面幾乎沒有任何差異，而此結果印證本論文的整合系統無論是原本用於光柵化的場景資料、視角或是投影方式，皆可以轉化套用至光線追蹤。



圖 6: 交點測試比較圖

左圖以原本 OpenGL 的光柵化產生，右圖則是整合系統的光線追蹤所產生，兩者的畫面幾乎沒有任何差異，這也表示整合系統擷取這些參數後，光線追蹤可以盡量保持場景視角和相對位置。

Cornell Box 場景

下頁圖 7 的 Cornell Box 場景比較，左圖為原本 OpenGL 的光柵化，右圖是本系統的光線追蹤畫面，成功擷取了顏色和光源資訊並且場景物件出現在相對正確的位置，這裡可以觀察到兩者繪圖方式呈現畫面的差異在陰影部分，由於光線

追蹤產生硬陰影(hard shadow)的效果非常容易，因此在相同參數下使用本系統的光線追蹤後可以很自然的獲得陰影效果，但是以原本 OpenGL 應用程式所呼叫的固定功能函式是無法呈現陰影的，只能透過更改程式碼並加入硬陰影 shader 程式才能實現。在此實驗的設定中，點光源在場景的正中間位置，如右圖以白色球體代表點光源位置，並放大 10 倍使之容易被觀察。



圖 7: Cornell Box 比較圖

左圖是光柵化，右圖是光線追蹤，實驗的程式碼中加入光源功能以及顏色設定。右圖可以很自然地根據點光源*產生相對應的硬陰影，而原本光柵化的方法需要透過其他額外的做法才能達成。

*註：白色球狀物體代表點光源，放大 10 倍使之容易觀察

額外擴充功能—反射

上段落提到使用本系統光線追蹤可以獲得內建的硬陰影效果，另外有些額外的效果是原本 OpenGL 固定功能所沒有的，所以本系統也設計一個物件材質屬性的擴充功能函式 `rtMaterialEXT`，只要設定 VBO 時在其程式碼前加上此函式，就能在光線追蹤時將此 VBO 視為特定材質並且使之呈現該材質的效果，下頁圖 8 可以看到左邊紅色牆面設定為鏡面材質並反射 dragon 模型。

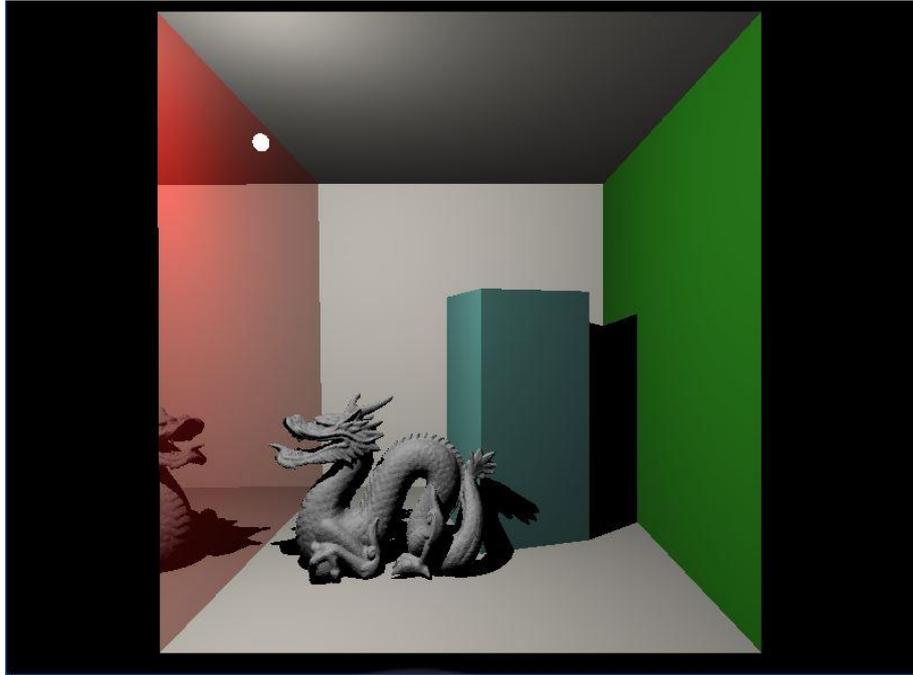


圖 8: 反射額外擴充效果

Cornell Box 模型中紅色左牆面設定為鏡面材質，並反射三角片數量為 10,000 的 dragon 模型，而畫面更新率為 31.12 fps 達到即時的速度。



圖 9: 折射額外效果

畫面中間偏前方的正方體設定為玻璃材質，可以看到立方體內部 dragon 模型經過折射而稍微位移，並擁有 30.15 fps 的即時繪製速度。

額外擴充功能—折射

上頁圖 9 所示，畫面中間偏前方的正方體設定為玻璃材質，可以看到立方體內部 dragon 模型經過折射而稍微位移並產生出折射的效果，其中光線追蹤設定光線的最大深度為 4 層，光線會根據物體折射率而改變行進方向，並增加其深度再重新對場景進行交點測試，相比於沒有特殊材質的場景會花更多時間來繪圖。

效能測試

除了上述擷取數據、整合功能測試以及額外擴充效果之外，系統內的光線追蹤效能也是很重要的議題，因為 OpenGL 應用程式擁有即時繪製的速度，為了讓整合後的應用程式不要有太大的落差感，譬如說需要數秒或數分鐘才能完成一張畫面的這種差異，因此本系統的光線追蹤勢必要盡量保留即時速度的特性。

上頁圖 8 和圖 9 的 Cornell Box 加上 Chinese Dragon 場景，總共有 10,034 個三角片，以及下頁圖 10 的 Sponza 場景也有 262,267 個三角片，在此數量級下本系統的光線追蹤對場景部份建立了 K-D 樹加速結構使之達到即時互動的速度。

具體的數據如下頁表 3，比較 OpenGL 的光柵化以及本系統的光線追蹤兩者效能列表，在預料之中的光線追蹤雖然已透過 GPU 加速但仍然沒辦法和光柵化方法相提並論，但值得一提的是在有一定數量規模的場景中光線追蹤也能有每秒數十張左右的速度，這代表的是現在的硬體能力漸漸能負擔這樣大量的計算量，未來光線追蹤也有很大的進步空間。



圖 10: Sponza 場景(262,267 triangles)

整合系統效能實驗以 262,267 個三角片數量的 Sponza 場景做測試，使用了 KD 樹加速結構使得交點測試的次數得以減少，本場景的畫面更新率為 11.05 fps。

	Triangle Sizes	OpenGL (GPU 1)	OpenGL (GPU 2)	Our System (GPU 1)	Our System (GPU 2)
Cornell Box	34	1332.40 fps	5730.58 fps	98.41 fps	264.19 fps
B & D	100,034	371.50 fps	5149.60 fps	23.86 fps	39.12 fps
B & D w/ Reflec	100,034	—————	—————	22.83 fps	33.93 fps
B & D w/ Refrac	100,034	—————	—————	16.41 fps	31.50 fps
Sponza	262,267	148.96 fps	2612.49 fps	5.34 fps	11.01 fps

表 3: 各場景效能列表

GPU 1: Intel® Iris™ Graphics 6100

GPU 2: NVIDIA® GeForce GTX 970

Reflec: Reflection, 額外反射效果

Refrac: Refraction, 額外折射效果

B & D: Cornell Box & dragon

fps: 每秒畫面張數(frame per second)

第六章 結論與未來方向

本篇論文主要的貢獻是提出光線追蹤應用程式介面整合 OpenGL 程式設計，讓開發者快速地獲得光線追蹤帶來的好處，並且印證光線追蹤應用程式已經可以接近即時的速度。但此系統仍有相當大進步空間，像是讓光線追蹤改成路徑追蹤並增加全域照明技術，使得畫面可以更逼真以及光影效果更接近現實世界，或是支援其他的 OpenGL 函式，讓此系統的整合功能趨近完整。

最後效能實驗的結果畫面解析度 800×600 和 Sponza 場景，呈現出光線追蹤方法約 11 fps 左右和光柵化方法 2612 fps 左右兩者計算速度上仍有極大的差距，雖然已透過 GPU 加速但光線追蹤卻僅僅是達到互動性門檻而已，其中本系統未使用到 OpenCL 內記憶體架構中的局部共享記憶體，或許改善這方面可能可以帶來些許的效能提升。

隨著 GPU 運算能力越來越強大，即時速度的光線追蹤一定能獲得更好的效能提升，像是 Wu, et al. [11]曾嘗試將加速結構讓 GPU 來計算希望突破靜態場景的限制，或是 Keller, et al. [12]在 ACM SIGGRAPH 2012 課程中提到更進階的蒙地卡羅路徑追蹤相關技術，這些增進畫面品質的作法在現在幾乎沒辦法實作到即時效果的光線追蹤或路徑追蹤演算法裡，但在未來或許能因為 GPU 硬體的提升而讓即時速度的光線追蹤能獲得這些技術所帶來的益處。

附錄 A

以下是 Möller and Trumbore [9]提出的光線與三角形交點測試程式碼，並且為了應用在本論文的光線追蹤改寫成 OpenCL 語言實作：

```
// Triangle Intersection
bool TriINTXN(Record* rec, const Ray* ray, const Triangle* tri, uint ID)
{
    if(rec->primID == ID) return false;

    float4 e1, e2, P, Q, T;
    float det, tt, uu, vv, inv_det;
    e1 = tri->v1 - tri->v0;
    e2 = tri->v2 - tri->v0;
    P = cross(ray->dir, e2);
    det = dot(e1, P);
    if (det > -FLT_EPSILON && det < FLT_EPSILON) return false;
    inv_det = native_recip(det);

    T = ray->ori - tri->v0;
    uu = dot(T, P) * inv_det;
    if (uu < 0.0f || uu > 1.0f) return false;

    Q = cross(T, e1);
    vv = dot(ray->dir, Q) * inv_det;
    if (vv < 0.0f || (uu + vv) > 1.0f) return false;
    tt = dot(e2, Q) * inv_det;

    if (tt > 0.0f && tt < rec->t) {
        rec->primID = ID;
        rec->type = TRI;
        rec->uv = (float2)(uu, vv);
        rec->t = tt;
        return true;
    }
    return false;
}
```

圖片引用來源

(1) 圖 1: OpenR 架構

來自以下網站:

Imagination Technologies, OpenRL SDK - Imagination Community.

Available: <https://community.imgtec.com/developers/powervr/openrl-sdk/>

(2) 圖 2: OpenCL Memory model

來自以下書籍篇章:

Chapter 2, Page 29 Figure 2.3B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1*: Newnes, 2012.

(3) 圖 5: Projection 矩陣影響視景體(左圖)

來自以下網站:

Free tutorials for modern OpenGL (3.3 and later) in C/C++ *Tutorial 3 : Matrices.*

Available: <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>

(4) 圖 5: Projection 矩陣影響視景體(右圖)

來自以下網站:

Free tutorials for modern OpenGL (3.3 and later) in C/C++ *Tutorial 3 : Matrices.*

Available: <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>

參考文獻

- [1] *Imagination Technologies, OpenRL SDK - Imagination Community*. Available: <https://community.imgtec.com/developers/powervr/openrl-sdk/>
- [2] *NVIDIA® OptiX™ Ray Tracing Engine | NVIDIA Developer*. Available: <https://developer.nvidia.com/optix>
- [3] B. Paul. *The Mesa 3D Graphics Library*. Available: <http://www.mesa3d.org/>
- [4] M. Pharr and G. Humphreys, *Physically based rendering: From theory to implementation*: Morgan Kaufmann, 2004.
- [5] *Khronos Group, OpenCL - The open standard for parallel programming of heterogeneous systems*. Available: <https://www.khronos.org/opencv/>
- [6] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL: Revised OpenCL 1*: Newnes, 2012.
- [7] *Khronos Group, OpenGL - The Industry Standard for High Performance Graphics*. Available: <https://www.opengl.org/registry/>
- [8] T. Whitted, "An improved illumination model for shaded display," presented at the Proceedings of the 6th annual conference on Computer graphics and interactive techniques, Chicago, Illinois, USA, 1979.
- [9] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," presented at the ACM SIGGRAPH 2005 Courses, Los Angeles, California, 2005.
- [10] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, pp. 311-317, 1975.
- [11] Z. Wu, F. Zhao, and X. Liu, "SAH KD-tree construction on GPU," presented at the Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, Vancouver, British Columbia, Canada, 2011.
- [12] A. Keller, S. Premoze, and M. Raab, "Advanced (quasi) Monte Carlo methods for image synthesis," in *ACM SIGGRAPH 2012 Courses*, 2012, p. 21.