



國立中山大學資訊工程學系

碩士論文

Department of Computer Science and Engineering National  
Sun Yat-sen University

Master Thesis

基於車道線辨識之前車偵測及加速

Lane-Based Front Vehicle Detection and Its Acceleration

研究生：陳傑琪 撰

Jie-Qi Chen

指導教授：蕭勝夫 教授

Shen-Fu Hsiao

中華民國 102 年 1 月

January 2013

國立中山大學研究生學位論文審定書

本校資訊工程學系碩士班

研究生陳傑琪（學號：M983040051）所提論文

基於車道線辨識之前車偵測及加速

Lane-Based Front Vehicle Detection and Its Acceleration

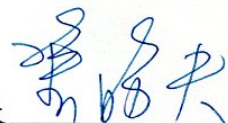
於中華民國 101 年 12 月 12 日經本委員會審查並舉行口試，  
符合碩士學位論文標準。

學位考試委員簽章：

召集人 李宗南



委員 蕭勝夫



委員 陳銘志



委員

\_\_\_\_\_

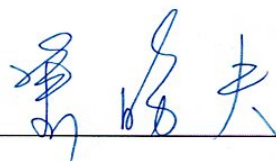
委員

\_\_\_\_\_

委員

\_\_\_\_\_

指導教授(蕭勝夫)



(簽名)

# 致謝

感謝指導教授 蕭勝夫教授一年來的指導，以及辛勞的幫忙論文的校正和定稿，使的本論文可以順利完成，同時也感謝口試委員的指導和建議，使的本論文更加的完整。

再次感謝 蕭勝夫教授提供非常完善的學習環境，研究資源非常的完善，使的我能全心的學習軟體和硬體，順利完成學業。

在這三年來，我換了三間的實驗室，在我什麼都不會的情況下，感謝士斌學長細心的幫我除錯，讓我在碩一課程中的作業能一一完成，並且學到不少撰寫軟體的技巧與能力，以及孟達、敏倫、裕信、育誠、庭偉學長的照顧，讓我有個充實的碩一。在碩三，也感謝冠甫學長對我的大力幫忙，讓我對 FPGA 這塊有了一定的認識，以及很多關於硬體和開發版的知識，以及其他學長姐的照顧，如家聖、張簡、信宏、文玲、維誠等學長姊，同屆的俊銘、普誠、柏翰、智宏在學業上和生活上的照顧，還有其他下一屆的學弟的支持，陪我度過最後一年研究所的時光，謝謝你們。

# 中文摘要

本論文是基於.NET Framework 4.0 為開發平台，並以 Visual C#為使用語言，對行車紀錄器的畫面做車道線偵測及前車偵測與追蹤，並使用不同種的方法對前處理時間進行加速，並對程式碼優化，減少執行時間。本文偵測部分分為兩種，一種是針對車道線偵測，另一種則是基於車道線範圍內的前車偵測，而追蹤是採用三步搜尋法進行畫面比對。前處理部分使用不同種方法運算，如單純 C#的指標運算、OPENCV 及 OPENCL 等方法加速軟體執行，並對程式碼進行加速及優化。在 i7-2600 3.4Ghz 下單張偵測最快可以達到每秒 30 張以上。除了 OPENCV 每秒 18 張外，其餘前處理方法的單張偵測皆可達到每秒 28 張以上，大多數加速方法皆能夠達到即時計算，並加入車輛輔助資訊，如前車車距、距離標線及車輛偏移警告。



**關鍵詞：** 道路線偵測 ，前車偵測，OpenCV，OpenCL，TPL，多執行緒

# Abstract

Based on .Net Framework4.0 development platform and Visual C# language, this thesis presents various methods of performing lane detection and preceding vehicle detection/tracking with code optimization and acceleration to reduce the execution time. The thesis consists of two major parts: vehicle detection and tracking. In the part of detection, driving lanes are identified first and then the preceding vehicles between the left lane and right lane are detected using the shadow information beneath vehicles. In vehicle tracking, three-pass search method is used to find the matched vehicles based on the detection results in the previous frames. According to our experiments, the preprocessing (including color-intensity conversion) takes a significant portion of total execution time. We propose different methods to optimize the code and speed up the software execution using pure C # pointers, OPENCV, and OPENCL etc. Experimental results show that the fastest detection/tracking speed can reach more than 30 frames per second (fps) using PC with i7-2600 3.4Ghz CPU. Except for OPENCV with execution rate of 18 fps, the rest of methods have up to 28 fps processing rate of almost the real-time speed. We also add the auxiliary vehicle information, such as preceding vehicle distance and vehicle offset warning.

**Keywords:** Lane Detection, Front Vehicle Detection, OpenCV, OpenCL, Task Parallel Library (TPL), Multi-threading.

## 目錄

<b>Abstract</b> .....	V
<b>第 1 章 緒論</b> .....	- 1 -
1.1 研究動機 .....	- 1 -
1.2 研究方法簡介 .....	- 2 -
1.3 論文章節說明 .....	- 3 -
<b>第 2 章 相關研究和軟體開發工具</b> .....	- 4 -
2.1 文獻回顧 .....	- 4 -
2.1.1 感測器 .....	- 4 -
2.1.2 道路線偵測 .....	- 4 -
2.1.3 前車偵測[6] .....	- 5 -
2.2 相關知識 .....	- 7 -
2.2.1 RGB 色彩模型[14] .....	- 7 -
2.2.2 HSL 和 HSV 色彩空間[14] .....	- 8 -
2.2.3 灰階化 .....	- 10 -
2.2.4 均值濾波 .....	- 10 -
2.2.5 Sobel 邊緣偵測[14] .....	- 11 -
2.3 軟體程式設計輔助工具 .....	- 11 -
2.3.1 C# Bitmap 類別 .....	- 11 -
2.3.2 C# Bitmap 指標存取 .....	- 12 -
2.3.3 OPENCV(Open Source Computer Vision Library) .....	- 13 -
2.3.4 OPENCL(Open Computing Language) .....	- 13 -
2.3.5 多執行緒 .....	- 18 -
2.3.6 TPL(Task Parallel Library) .....	- 18 -
2.3.7 Profile .....	- 19 -
<b>第三章 前處理與道路偵測</b> .....	- 20 -
3.1 前處理 .....	- 20 -
3.1.1 均值濾波 .....	- 21 -
3.1.2 邊緣偵測 .....	- 21 -
3.1.3 C#指標運算 .....	- 22 -

3.1.4 OpenCV .....	- 22 -
3.1.5 OpenCL.....	- 23 -
3.1.6 圖片旋轉.....	- 23 -
3.2 道路線偵測.....	- 24 -
3.2.1 道路線掃描.....	- 26 -
3.2.2 回歸直線.....	- 28 -
第四章 前車偵測 .....	- 31 -
4.1 前車偵測.....	- 31 -
4.1.1 近車偵測.....	- 32 -
4.1.2 自動陰影臨界值.....	- 33 -
4.1.3 陰影偵測.....	- 38 -
4.1.4 陰影資訊排序.....	- 40 -
4.1.5 車輛左右邊緣偵測.....	- 40 -
4.1.6 車輛頂部偵測.....	- 46 -
4.1.7 車輛驗證.....	- 47 -
4.2 連續圖片偵測.....	- 48 -
4.2.1 車輛追蹤.....	- 49 -
4.2.2 三種偵測模式.....	- 50 -
4.2.3 車輛偵測資訊.....	- 52 -
4.3 前車距離、距離標線及車輛偏移警告.....	- 53 -
4.3.1 回歸曲線.....	- 54 -
4.3.2 前車距離與距離標線.....	- 55 -
4.3.4 車輛偏移警告.....	- 56 -
第五章 執行加速與 GUI 介面設計 .....	- 57 -
5.1 執行加速.....	- 57 -
5.1.1 C# TPL 加速.....	- 57 -
5.1.2 OPENCL 優化.....	- 58 -
5.1.3 多執行緒及多執行緒載入圖片.....	- 60 -
5.1.4 程式碼優化.....	- 60 -
5.2 GUI 介面設計 .....	- 62 -
5.2.1 介面設計.....	- 63 -
5.2.2 功能介紹.....	- 64 -
5.2.3 程式碼參數化.....	- 66 -

5.2.4 影片參數設定.....	- 68 -
第六章 實驗數據及分析 .....	- 70 -
6.1 單張偵測時間.....	- 71 -
6.2 同影片不同偵測方法連續 1000 張運算結果.....	- 72 -
6.3 不同影片連續 1000 張運算結果.....	- 73 -
第七章 結論、未來研究方向 .....	- 76 -
7.1 結論.....	- 76 -
7.2 未來研究方向.....	- 76 -
參考文獻 (References).....	- 78 -
附錄一 (前處理未優化程式碼) .....	- 80 -
附錄二(OpenCL kernel code) .....	- 87 -



# 圖目錄 (List of Figures)

	頁
Fig.1 HG 和 HV 示意圖[6] .....	5 -
Fig.2 IPM 效果[6]，利用(a)(b)左右攝影機圖，可以經由公式轉換成高空投影圖，如圖(c)(d)，即可利用(c)(d)兩張圖推估前車距離 .....	6 -
Fig.3 三原色混和的表現 .....	8 -
Fig.4 HSV 色輪及圓錐表示 (a).HSV 色輪 (b).HSV 圓錐 .....	9 -
Fig.5 水平方向掃描使用第一張圖，垂直方向掃描使用第二張圖 .....	13 -
Fig.6 圖平台模型 – 一個主機加上一個或多個計算設備，每個設備具有一個或多個計算單元，每個計算單元具有一個或多個處理元件。 .....	15 -
Fig.7 OPENCIL 的記憶體模型 .....	17 -
Fig.8 欠缺像素處理，上邊緣，則以畫面最底部像素替補，左邊緣，則以畫面最右邊像素替補 .....	22 -
Fig.9 (a).鄉村道路, (b).國道前, (c).交流道下, (d).路口, (e).高速公路, (f).市區 .....	24 -
Fig.10 (a).路面標字干擾, (b).雙黃線 .....	25 -
Fig.11 (a).旋轉前畫面掃描方向 (b).旋轉後畫面掃描方向 .....	26 -
Fig.12 黃色為左道路線掃描點，藍色為右道路線掃描點，紅色為道路線掃描點 .....	28 -
Fig.13 (a).現實世界座標及斜率示意圖, (b).程式中座標及斜率示意圖 ....	29 -
Fig.14 (a).鄉村道路, (b).近距離車輛, (c).路面極黑的高架橋下, (d).卡車, (e).高速公路上, (f).有陸橋陰影干擾, (g).畫面些微反光, (h).鄉鎮道路 .....	32 -
Fig.15 道路線間的直方圖亮度由低向高取第 AutoThresholdcnt1 個有值的亮度-1 .....	34 -
Fig.16 道路線間不同行的直方圖亮度由低向高取第 AutoThresholdcnt2 個有值的亮度-1 .....	35 -
Fig.17 道路線間的加權直方圖亮度由低向高累積到 AutoThresholdcnt3 比例數量的亮度值 .....	37 -
Fig.18 由於處理時需由下而上，因此使用旋轉後的水平濾波灰度圖做處理，結果座標會使用座標轉換回正向的圖的座標，紅色為陰影輪廓掃描結果，結果顯示近車和遠車皆可陰影輪廓掃描，(a).為遠距離車輛 (b).為近距離車輛。 ...	39 -
Fig.19 陰影吃掉輪胎旁的垂直邊緣特徵，導致第一種左右邊緣偵測方法失效 .....	42 -

Fig.20 第一種左右邊緣偵測原理示意圖 .....	- 43 -
Fig.21 第二種左右邊緣偵測原理示意圖 .....	- 45 -
Fig.22 車輛背後的水平特徵 .....	- 48 -
Fig.23 三步搜尋法示意圖 .....	- 50 -
Fig. 24 追蹤為主模式流程圖.....	- 51 -
Fig. 25 偵測為主模式流程圖.....	- 51 -
Fig. 26 三種模式整合流程圖.....	- 52 -
Fig.27 文字檔中儲存每行資訊 .....	- 53 -
Fig.28 車輛偵測輔助資訊畫面 .....	- 54 -
Fig.29 此為某影片統計出的回歸曲線，其中 $m = 1.034118$ ， $b = 0.167287$ ，x 軸單位為距離底部的高度像素值，y 軸單位為(公分/像素).....	- 55 -
Fig.30 (a).車輛偏移角度示意圖, (b). 當車輛向左右偏移超過 <b>15°</b> ，顯示車輛 偏移警示.....	- 56 -
Fig.31 OpenCL 為優化前的時序分析圖 .....	- 58 -
Fig.32 調整順序後的 OpenCL 時序分析圖 .....	- 59 -
Fig.33 統一鎖定記憶體和對記憶體解鎖修改後的時序分析圖 .....	- 62 -
Fig.34 GUI 主要畫面最大化和最小化 .....	- 63 -
Fig.35 可活動式和可切換進階參數設定面板 .....	- 64 -
Fig.36 參數化程式碼內容 .....	- 66 -
Fig.37 參數設定檔位置 .....	- 69 -
Fig.38 單張測試時間，分別在 Intel Core Quad Q9500 2.8Ghz 和 i7-2600 3.4Ghz 測試時間，時間單位為 ms(毫秒).....	- 71 -
Fig.39 不同設定下的影片執行結果 .....	- 72 -
Fig.40 不同影片各段數據結果 .....	- 74 -

# 表目錄 (List of Tables)

	頁
表 1-1 章節說明圖 .....	- 3 -
表 3-1 圖片名稱與用途 .....	- 20 -
表 3-1 左右道路線決定的 pseudo code .....	- 30 -
表 4-1 近車偵測的 pseudo code .....	- 33 -
表 4-2 自動陰影臨界值方法一的 pseudo code .....	- 35 -
表 4-3 自動陰影臨界值方法二的 pseudo code .....	- 36 -
表 4-4 自動陰影臨界值方法三的 pseudo code .....	- 38 -
表 4-5 陰影輪廓偵測的 pseudo code .....	- 40 -
表 4-6 陰影大小校正的 pseudo code .....	- 41 -
表 4-7 左右邊緣偵測方法一的 pseudo code .....	- 44 -
表 4-8 左右邊緣偵測方法二的 pseudo code .....	- 46 -
表 4-9 車輛頂部偵測的 pseudo code .....	- 47 -
表 6-1 偵測率判斷方式表格 .....	- 70 -

# 方程式目錄 (List of Equations)

	頁
(1) HSL 公式.....	- 9 -
(2) HSV 公式.....	- 10 -
(3) 灰階公式.....	- 10 -
(4) 均值濾波遮罩(mask)[16].....	- 11 -
(5) 本文水平濾波遮罩(mask).....	- 11 -
(6) 垂直邊緣檢測遮罩(mask).....	- 11 -
(7) 水平邊緣檢測遮罩(mask).....	- 11 -
(6) 垂直邊緣檢測遮罩(mask).....	- 21 -
(7) 水平邊緣檢測遮罩(mask).....	- 21 -
(8) 回歸直線公式.....	- 28 -
(9) 紅色車燈判斷依據.....	- 32 -
(10) 陰影放大倍數.....	- 55 -
(11) 本文偵測率算法.....	- 70 -

# 第 1 章 緒論

## 1.1 研究動機

隨著時代的進步，高科技改變了人們生活方式，特別是在交通運輸上。目前，智慧型運輸系統( Intelligent Transportation System, ITS )與自動車輛( Autonomous Guided Vehicle, AGV )以十分快速的發展進行中，其中發展的一部份即是智慧型車輛，而智慧型車輛的發展又以安全駕駛為其最重要的一項，正所謂沒有安全，車子再先進也沒有用，而安全駕駛不外乎提供車輛與環境之間的資訊，如協助駕駛者能與四周車輛保持安全距離、注意交通警示燈和提供車輛駕駛輔助資訊等，可以避免車輛追撞或是提供更多的資訊輔助駕駛行車更安全。到現在，各大車廠已投入智慧型車輛的安全研究中，為的就是發展出一套行車安全的系統提供駕駛人更安全的行車系統。

而智慧型車輛中的發展，與行車安全相關最重要的就是車輛前方的資訊，其次才是車輛四周的資訊。因為車輛行駛是往前前進，如果車輛前方有其它的行人或是車輛，一個不小心就會發生意外，因此必須當前方確定安全後，才會考慮是否要切換車道或轉彎，所以，在智慧型車輛研究的議題，前方偵測的研究是非常重要的。在車輛行駛中，前方的偵測包括車道線及車輛偵測，利用這兩項資訊還可以提供不少的車輛輔助資訊，如前方車輛距離資訊及車輛偏移警示。有了這些資訊，駕駛人在行車的路途上會多了一層保障。

另外，由於行車必須是即時的，因此在發展成一套車輛偵測設備必須不能太過於緩慢，必須在處理速度上要能達到 25~30fps 才能達到實際的應用，所以在開發研究時除了要對偵測道路線及偵測車輛有不錯的方法，還需想辦法提升執行的速度，因此本文研究除了道路線偵測及前車偵測外，還研究了加速電腦處理速度的方法。在知道如何偵測以及加速方法後，在未來如果有機會做成

實際產品，勢必也會對何處修改可以提升執行效能有更多的了解，減少開發的難易度及增加開發的品質。

## 1.2 研究方法簡介

本文主要針對日間的行車紀錄器影片進行道路線偵測及前車偵測，並對執行速度進行優化，目的希望對影片執行的處理速度能達到每秒 25 fps (frames per second) 以上，並標出距離標線、前方車輛距離及車輛偏移警告。

本文在 PC 上把行車紀錄器影片切割成每秒 30 張的 jpg 圖檔，利用連續載入圖片的方法對連續圖片進行偵測與追蹤，在偵測方面利用圖片上的道路線及車輛特徵進行道路線偵測及前車偵測，並使用三步搜尋法進行車輛追蹤，並可在處理完後，儲存成連續圖片撥放的偵測與追蹤資訊。用各種不同方法對執行時間加速，比較不同方法的差異，並對程式碼優化進行研究，目標是達成處理速度，在 Intel i7-2600 3.4Ghz 能達到即時每秒 25fps 以上，而在 Intel Core Quad Q9500 2.8Ghz 上，由於 CPU 速度較慢，希望執行速度能在每張畫面執行處理時間達到 16fps 以上。

### 1.3 論文章節說明

本文可以分為 7 章，如表 1-1：

表 1-1 章節說明圖

章數	內容說明
第一章	說明研究動機與目的
第二章	相關文獻與知識和軟體工具介紹
第三章	前處理、道路線偵測
第四章	前車偵測、追蹤與車輛輔助資訊
第五章	執行加速、GUI 介面設計與參數設定
第六章	不同實驗結果比較，及效能評估
第七章	結論與未來展望

## 第 2 章 相關研究和軟體開發工具

### 2.1 文獻回顧

#### 2.1.1 感測器

感測器主要分為主動與被動：

主動感測器[1]像是雷達[2]、雷射(紅外線)[3][4]、超音波[5]，利用主動感測器發出和接收電磁輻射或聲波，來測量是否有障礙物或者是測出距離，它們的優點是可以直接做測量，而不需要一些強力的計算資源，但是相對的它們必須付出龐大的花費、空間解析度較低和掃描速度太慢[6]。

被動感測器[1]像是攝影機或是 CCD(Charge Coupled Device)，它們的優點是價格較主動感測器低廉、能夠 360 度視野且可以使用追蹤的方式偵測車輛。它能夠應用在道路偵測、交通訊號辨識或物件辨識，且不需要任何的道路設施，這個方法經由驗證可以使用在視覺為主的駕駛警示系統[6][7]。

#### 2.1.2 道路線偵測

道路線的偵測方式有分很多種。在一開始要先設定搜尋道路線的範圍，可以是全畫面或是鎖定局部畫面[8]。接著要對設定範圍內找出道路特徵，可以使用算子[9][10]對道路線特徵進行偵測，也可以使用掃描線[8][11][12][13]的方式，掃描道路特徵。最後是判斷出左右道路線。而描述道路線的方式又分為兩種，較簡單的方式是以直線描述道路線[8][9][10][11]，也可以用曲線描述道路線[12][13]。而找尋道路線的用意在於可以縮小 ROI(Region of Interest)區域，加速運算速度。或是利用找出的左右道路線資訊判斷車輛是否發生偏移，以發出偏移警告。在道路線搜尋過程中往往會因為一些干擾或者是特徵不明顯，例如有路面的雜訊(不乾淨的地面)、道路中央的字、斑馬線、道路線的斷裂或者有一些夜間用的道路



反光鏡，或是模糊不清的道路線，導致找出的道路線錯誤、歪斜或是搜尋不到道路線，因此好的道路線搜尋方式是很重要的。本論文中將提出運算簡單而又有效的道路線偵測方法。

### 2.1.3 前車偵測[6]

前車偵測分成了兩個步驟，HG (Hypothesis Generation)和 HV (Hypothesis Verification)。HG 是找出可能是車輛的目標，而 HV 是驗證目標的正確性，如下圖 Fig.1。

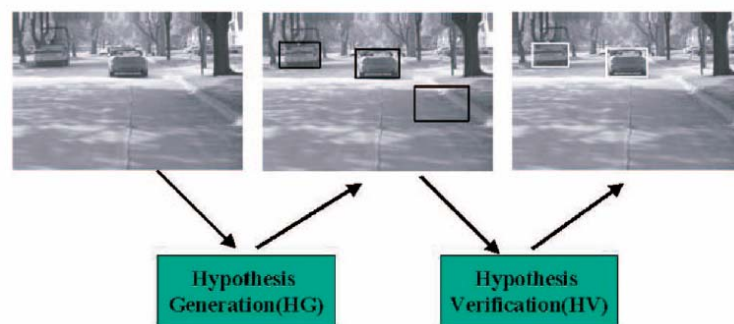


Fig.1 HG 和 HV 示意圖[6]

HG 的方法有很多種，大致可分成三大類：1. knowledge-based、2. stereo-based、3. motion-based。

1. Knowledge-based：可以使用對稱性、顏色、陰影、角點、垂直或水平邊緣、車尾燈來偵測車輛。一般而言，在日間最明顯的特徵就是陰影，只要是車子，在一定的光照下一定會產生陰影，所以最簡單也是最好用的車輛偵測方法就是使用陰影。其次是利用對稱性，因為車子是左右對稱，因此可使用對稱特徵對對稱軸投票，找出車輛中心軸及左右寬度。而在較古老的方法是使用顏色的方式搜尋車輛，但現代車輛顏色種類越來越多，顏色並不是所有車輛的共通點，也就是不是每台車輛都是相同顏色，而且當車輛顏色是灰色時，車輛就找不到了，因此此種做法在近年來很

少使用。而其他種特徵主要是混合陰影特徵一起做判斷，可以漂亮的框出整台車。

2. Stereo-based: 就是所謂的立體視覺，利用雙鏡頭可以重建出立體的畫面。而車輛在畫面中往往與背景具有明顯的不同深度，可以利用深度的方式判斷和找出車輛的位置，另外立體視覺還可以做 IPM(Inverse Perspective Mapping)，也就是可以垂直地面投影出車輛實際位置與距離，如圖 Fig.2。

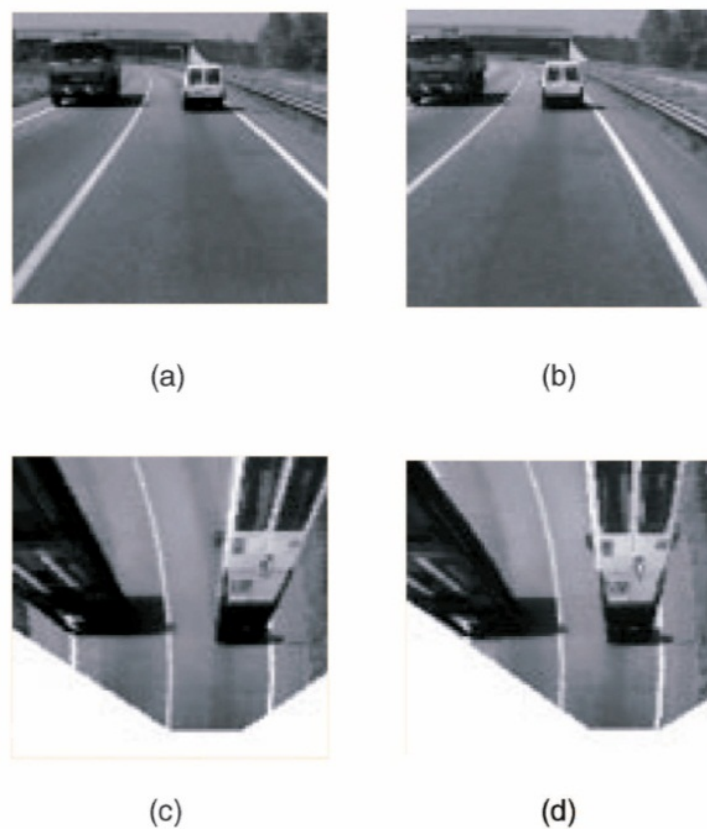


Fig.2 IPM 效果[6]，利用(a)(b)左右攝影機圖，可以經由公式轉換成高空投影圖，如圖(c)(d)，即可利用(c)(d)兩張圖推估前車距離

3. Motion-based: 也就是光流法。當目標在影像中某個畫面已經被框選，可以使用畫面搜尋演算法來作畫面比對，找出現在這個畫面中最像目標已經被框選畫面的框選目標範圍，並使用一個 motion vector 指向這個畫面比對結果的左上角，而這個畫面的框選目標即為此 motion vector 指向

的新位置。此種做法的優點是能高效率的追蹤車輛，但此做法有一個缺點，那就是當目標物靠近或遠離鏡頭時，目標物會放大縮小，導致比對時無法正確的比對，另外比對結果誤差可能會越來越大，導致追蹤失敗或錯誤。

HV 可以依照車輛特徵，來驗證框選範圍內是否為車輛，也可以將特徵擷取出來，參數化，代入 SVM(Support Vector Machine)中判斷是否為車輛。車輛特徵有很多種，如前面所說的，可以是陰影、對稱性、邊緣特徵...等等。而 SVM 的原理是利用資料庫裡的車輛資訊與非車輛資訊，轉化成特徵參數，特徵參數有很多種，可以是一個畫面的 Pixel 值，也可以是對畫面的特徵進行統計，轉換成參數。把特徵參數代入 SVM 訓練 Model，並利用訓練好的 Model 去分類找出來的框選範圍內是否為車輛。

## **2.2 相關知識**

### **2.2.1 RGB 色彩模型[14]**

RGB 色彩模型是一種加色模型，是用三種原色 – 紅色、綠色和藍色的色光以不同比例相加，以產生多種多樣的色光，如圖 Fig.3。



**Fig.3 三原色混和的表現**

目前在計算機硬體中採取每一像素用 24bit (位元) 表示的方法，所以三種原色光各分到 8bit，每一種原色的強度依照 8bit 的最高值  $2^8$  分為 256 個值。用這種方法可以組合 16777216 種顏色，但人眼實際只能分辨出 1000 萬種顏色。(不同的人分辨能力並不相同，這只是最大值)。

而在程式裡有時會多一項附加的額外訊息，就是透明度(Alpha)，也就是原來的 RGB 會變成 RGBA，當 A 為 0 時則意味完全透明，255 時為完全不透明，一共是 32bit(各 8bit)。

註：在 C#Bitmap 像素表示即使用此色彩空間表示，儲存的方式是 BGR(A)。

### **2.2.2 HSL 和 HSV 色彩空間[14]**

HSL 和 HSV 是對 RGB 色彩空間中點的兩種有關係的表示，它們常是描述比 RGB 更準確的感知顏色聯繫，並仍保持在計算上簡單。

H 指 **hue**(色相)、S 指 **saturation**(飽和度)、L 指 **lightness**(亮度)、V 指 **value**(色調)。

HSV 色彩空間還可以表示圓錐體的圓柱體，色相沿著圓柱體的外圓周變化，飽和度沿著從橫截面的圓心的距離變化，明度沿著橫截面到底面和頂面的距離而變化。這種表示可能被認為是 HSV 色彩空間的更精確的數學模型；但是在實際中可區分出的飽和度和色相的級別數目隨著明度接近黑色而減少。此外計算機典型的用有限精度範圍來存儲 RGB 值；這約束了精度，再加上人類顏色感知的限

制，使圓錐體表示在多數情況下更實用，如下圖 Fig.4。

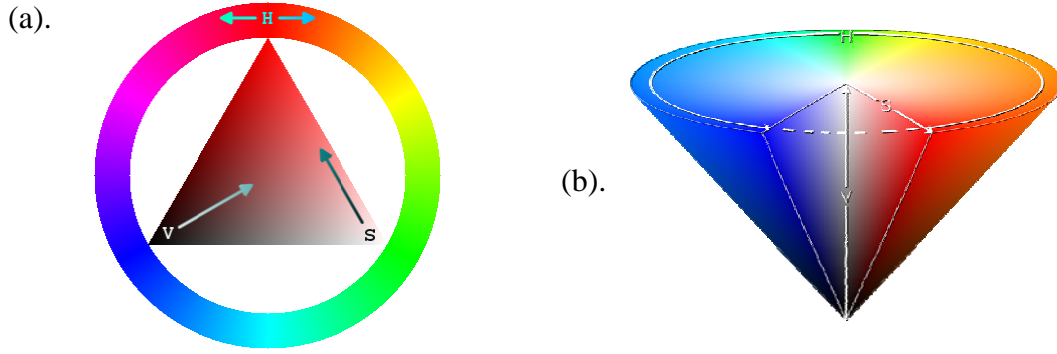


Fig.4 HSV 色輪及圓錐表示 (a).HSV 色輪 (b).HSV 圓錐

RGB 到 HSL 或 HSV 的轉換公式如下：

設  $(r, g, b)$  分別是一個顏色的紅、綠和藍坐標，它們的值是在 0 到 1 之間的實數。設  $\max$  等價於  $r, g$  和  $b$  中的最大者。設  $\min$  等於這些值中的最小者。要找到在 HSL 空間中的  $(h, s, l)$  值，這裡的  $h \in [0, 360)$  度是角度的色相角，而  $s, l \in [0, 1]$  是飽和度和亮度，計算公式為：

$$h = \begin{cases} 0^\circ, & \text{if } \max = \min \\ 60^\circ \times \frac{g - b}{\max - \min} + 0^\circ, & \text{if } \max = r \text{ and } g \geq b \\ 60^\circ \times \frac{g - b}{\max - \min} + 360^\circ, & \text{if } \max = r \text{ and } g < b \\ 60^\circ \times \frac{b - r}{\max - \min} + 120^\circ, & \text{if } \max = g \\ 60^\circ \times \frac{r - g}{\max - \min} + 240^\circ, & \text{if } \max = b \end{cases}$$

$$l = \frac{1}{2}(\max + \min)$$

$$s = \begin{cases} 0, & \text{if } l = 0 \text{ or } \max = \min \\ \frac{\max - \min}{\max + \min} = \frac{\max - \min}{2l}, & \text{if } 0 < l < \frac{1}{2} \\ \frac{\max - \min}{2 - (\max + \min)} = \frac{\max - \min}{2 - 2l}, & \text{if } l > \frac{1}{2} \end{cases} \quad (1)$$

h 的值通常規範化到位於 0 到 360°之間。而 h = 0 用於 max = min 的（就是灰色）時候而不是留下 h 未定義。HSL 和 HSV 有同樣的色相定義，但是其他分量不同。HSV 顏色的 s 和 v 的值定義如下：

$$s = \begin{cases} 0, & \text{if } \max = 0 \\ \frac{\max - \min}{\max} = 1 - \frac{\min}{\max}, & \text{otherwise} \end{cases} \quad (2)$$

$$v = \max$$

註：本論文在車輛偵測動作之前，會先利用 HSV 資訊偵測畫面中紅色的車尾燈做近車偵測。

### 2.2.3 灰階化

灰階的做法有很多種，最簡單的方法就是 RGB 的值相加除以三即可達到灰階化的效果，但較符合人眼視覺的算法則是下面的公式：

$$\text{Gray} = 0.299 \times \text{Red} + 0.587 \times \text{Green} + 0.114 \times \text{Blue} \quad (3)$$

註：本論文車輛偵測所需的車輛邊緣資訊，是從灰階化後之亮度資訊，經過 Sobel 邊緣運算而得。

### 2.2.4 均值濾波

均值濾波是低通濾波的一種，功用是可以對畫面進行降噪處理。做法是將 3\*3 的遮罩(mask)全部相加除以九即得到中間位置的像素值，而本文做法僅是將遮罩中央像素及中央左右兩個像素相加除以三，做均值濾波的原因是為了消除畫面中的雜訊，而本文的做法是為了避免將陰影邊緣模糊化，僅對水平方向進行畫面平滑（在 3.1 章會有詳細說明）。

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 9 \quad (4)$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} / 3 \quad (5)$$

### 2.2.5 Sobel 邊緣偵測[14]

索貝爾 (Sobel) 運算元是圖像處理中的運算元之一，主要用作邊緣檢測。在技術上，它是一離散性差分運算元，用來運算圖像亮度函數的梯度之近似值。在圖像的任何一點使用此運算元，將會產生對應的梯度向量或是其法向量。

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (6)$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (7)$$

## 2.3 軟體程式設計輔助工具

### 2.3.1 C# Bitmap 類別

C#是微軟推出的一種基於.NET 框架的、物件導向的高階語言。C#由 C 語言和 C++衍伸而來，繼承了其強大的效能，同時又以.NET 框架類別庫做為基礎，擁有類似 Visual Basic 的快速開發能力。而 C#中的 Bitmap 類封裝 GDI+ 的點陣圖，這個點陣圖是由圖形影像的像素資料及其屬性所組成。Bitmap 是用來處理像素資料所定義影像的物件[17]。儲存的內容除了圖片屬性外，還有圖片資訊，Bitmap 最簡單的存取方法是呼叫 GDI+ 函數 GetPixel 和 SetPixel，如下：

```
Bitmap bmp = new Bitmap(filename); //從檔案載入圖片  
Color pixel = bmp.GetPixel(x, y);    //取得座標(x, y)顏色  
//pixel.R 是紅色 pixel.G 是綠色 pixel.B 是藍色  
Bmp.SetPixel(x, y, pixel);           //設定座標(x, y)顏色
```

### 2.3.2 C# Bitmap 指標存取

由於使用 GetPixel 及 SetPixel 兩個 GDI+函數效率較差，尤其是圖片要存取上百萬次時的時間非常的久，因此在 C#中提供另一種較快速的做法，就是使用指標的方式存取。由於在 C#指標是違法的，因此必須先允許使用 unsafe{}，並把指標程式放在 unsafe 中。接著 C#的 Bitmap 類型提供 LockBits()函數可將影像的指定區域以 Byte Array 的形式存放在記憶體中，並回傳 BitmapData 類型的物件，並且可以使用 BitmapData.Scan0 獲得 Byte Array 的第一個 Byte 的首位指標，因此可以利用指標的方式存取圖片提升速度，使用完需 UnLockBits()釋放記憶體才可繼續使用圖片[18]，不然會回報錯誤。

Byte Array 的存放是以水平掃描畫面一行一行的存進 Byte Array，存放方式是 BGR(A)BGR(A)BGR(A)...[19]，所以水平方向的指標是連續的，並且尾端會在補齊 4 的倍數的 Byte 後接續下一行。而在記憶體存取裡，指標連續存取速度是最快的，而且可避免掉多次計算指標位址的乘法運算，所以如果需要垂直方向的掃描畫面，最好的做法不是利用乘法直接換算指標位址取值，這樣需做很多次的乘法運算，最好的方法是另外儲存一張相同的圖片，並將圖形旋轉 90 度，如此一來就可以對畫面垂直的方向作連續的掃描了，在後面道路線偵測及陰影偵測皆需要由底部往上掃描，如圖 Fig.5。



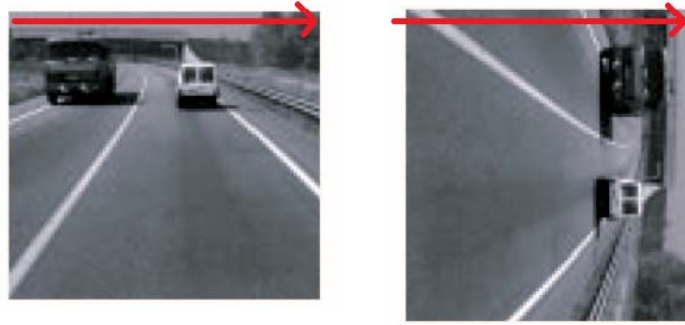


Fig.5 水平方向掃描使用第一張圖，垂直方向掃描使用第二張圖

### 2.3.3 OPENCV(Open Source Computer Vision Library)

OPENCV是一個跨平台的計算機視覺庫。OpenCV是由英特爾公司發起並參與開發，以BSD許可證授權發行，可以在商業和研究領域中免費使用。OpenCV可用於開發實時的圖像處理、計算機視覺以及模式識別程序[14]。

而在Visual C#中的OPENCV是EmguCV，他封裝了OpenCV image processing library，在C#使用僅需把需要的DLL檔加入參考，並在程式碼中加入下面四行，即可使用。

1. using Emgu.CV;
2. using Emgu.CV.Util;
3. using Emgu.CV.CvEnum;
4. using Emgu.CV.Structure;

### 2.3.4 OPENCL(Open Computing Language)

OpenCL 是由非盈利性技術組織 Khronos Group 掌管，是一個為異構平台編寫程序的框架，它可以由 CPU，GPU 或其他類型的處理器組成。OpenCL 的組成是由一門用於編寫 kernels 的語言（基於 C99）和一組用於定義並控制平台的 API 組成。[14]

OPENCL 在平行計算方面，有 data-parallel 和 task-parallel 兩種不同的計算模型。data-parallel 就是把一個問題分解為能夠同時執行的多個任務，而

task-parallel 則是同一個任務內，它的各個部分同時執行。[21]

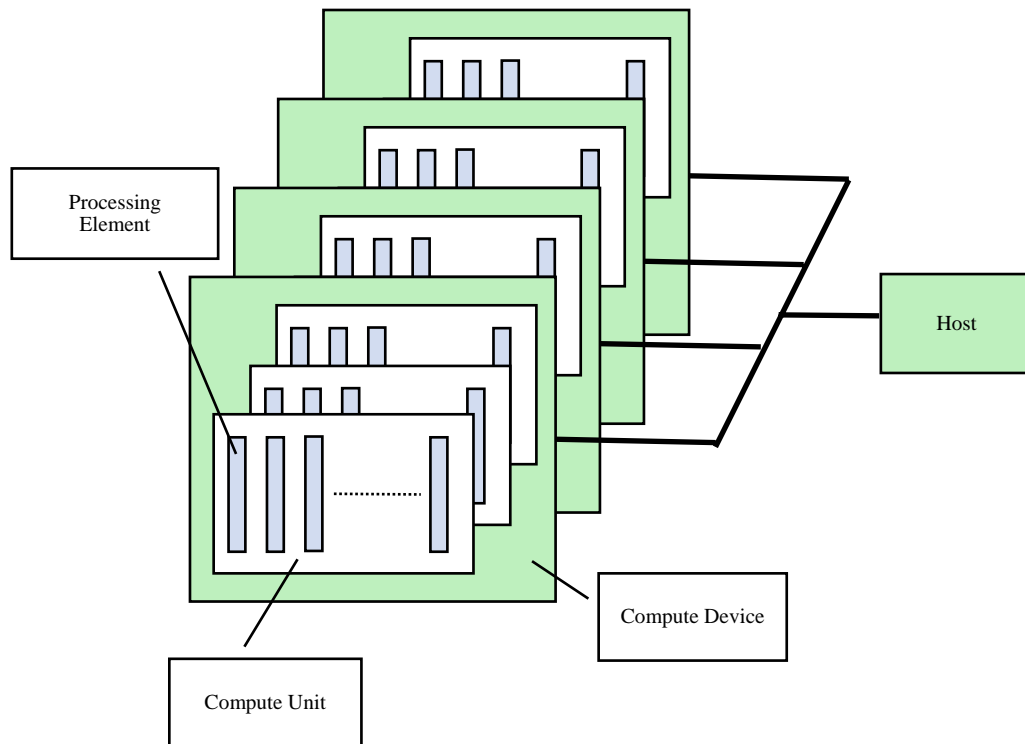
舉個例來說，一棵樹代表一個任務。在單一核心時像是一個工人要摘完所有樹上的蘋果，他必須一個人一棵樹一棵樹的把蘋果摘完。而 data-parallel 則是多個核心像是一群工人，它們會共同摘一棵樹的蘋果，因此可以很快地摘完。而 task-parallel，則是有很多個核心像是一群工人，他們會各自挑選一棵樹來摘蘋果，可以同時摘不同樹的蘋果，也就是 CPU 的 thread 的概念。[21]

為了描述 OPENCL 的核心思想，我們主要分成四個模型：[24]

#### 1. 平台模型

如 Fig.6 圖，一個 Host 可以連接到一個或多個 OPENCL 設備上，它可以是 CPU 或是 GPU 或者其他設備，而一個 OPENCL 設備被劃分成一個或多個計算單元(CU)，每個計算單元又被劃分成一個或多個處理元件(PE)，設備上的計算發生在處理元件中。

OPENCL 會按照宿主機平台的原生模型在這個宿主機上運行，從宿主機提交命令給設備上的 PE 來執行計算，而計算單元中的所有 PE 作為 SIMD 單元 SPMD 單元執行單個指令流。



**Fig.6 圖平台模型** – 一個主機加上一個或多個計算設備，每個設備具有一個或多個計算單元，每個計算單元具有一個或多個處理元件。

## 2. 執行模型

OPENCL 在一個或多個 OPENCL 設備上執行內核，或在宿主機上執行宿主機程。OPENCL 執行模型的核心是通過宿主機設定內核怎樣執行來定義，當宿主機提交內核來執行時，會定義一個 N 維的索引空間，內核會使用不同的 work-item 為其中的所有點而執行，並且各 work-item 有不同索引空間中的點的標示，這些工作項提供一個 global ID，所有工作項會執行相同的 kernel code，但是代碼執行的路徑和運算的數據可能會不同。

Work-item 可分配到不同的 work-group，每個 work-group 也有不同的 work-group ID，work-item 和 work-group ID 使用的索引空間具有相同的維度，每個 work-item 還有一個 local ID，此 ID 在其所在的 work-group 中是唯一的，所以單個工作項可以通過其 global ID 或通過其 local ID 加 work-group ID 作唯一標示。一個給定的 work-group 中的 work-item 會在單個 CU 中的多個 PE 上並發執行。

OPENCL 所支持的索引空間叫做一個 NDRange。NDRange 是一個 N 維的索

引空間，N 可以是 1、2 或 3，每個 work-item 的 global ID 和 local ID 都是 N 維的元素。並且 global ID 和 local ID 的取值範圍皆是各維度上各自的元素個數減 1。

宿主機為內核定義了一個 Context，它包含下列資源：

- a. 設備：宿主機可以使用的 OPENCL 設備集合。
- b. 內核：運行在 OPENCL 設備上的 OPENCL 函數。
- c. 程序對象：程序源碼和實行內核的執行體。
- d. 內存對象：對宿主機和 OPENCL 設備可見的一組內存對象，這些內存對象包含一些值，內核實例可以在這些值上進行運算。

宿主機使用 OPENCL API 中的函數來創建和操控 Context。宿主機創建一個叫做 Command Queue 的數據結構來協調設備上內核的執行。宿主機將命令放入 Command Queue，在設備上的 Context 中進行調度。包括：

- a. 內核執行命令：在設備的 PE 上執行內核。
- b. 內存命令：讀寫內存對象，或者自宿主機地址空間映射或解映射內存對象。
- c. 同步命令：限制命令的執行順序。

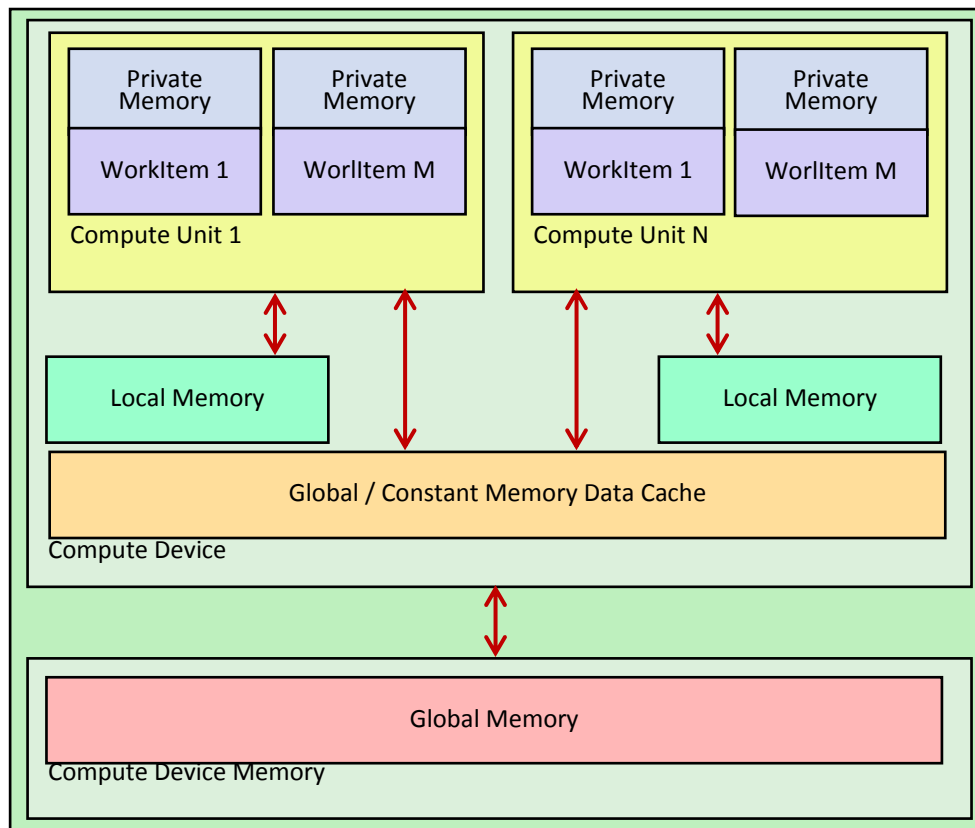
註：Command Queue 在設備上執行命令進行調度，其命令間可以是順序執行或是亂序執行。

### 3. 內存模型

主要分為四塊不同的內存區域：

- a. Global memory：存在 global address 空間，可以被所有的 work-item 存取。
- b. Constant memory：只可讀取的 memory
- c. Local memory：此 memory 可以被 work-group 分享
- d. Private memory：只可被自己的 work-item 存取

存取速度為：Private memory > Constant memory > Local memory > Global memory



**Fig.7 OPENCL 的記憶體模型**

#### 4. 編程模型

OpenCL 執行模型支持數據並行編程模型和任務並行編程模型，同時也支持這兩種模型的混和體。

OpenCL 可以在顯示卡上使用也可在多核 CPU 上執行，在多核 CPU 上執行需安裝 AMD-APP-SDK 裡附的 OpenCL 安裝檔。

在 C#裡使用 OpenCL 僅需將 OpenCLNET.dll 加入參考，加入下面兩行，即可使用。

1. using OpenCLNet;
2. using CL = OpenCLNet;

### 2.3.5 多執行緒

多執行緒是指從軟體或者硬體上實現多個執行緒並發執行的技術。具有多執行緒能力的計算機因有硬體支援而能夠在同一時間執行多於一個執行緒，進而提升整體處理效能。軟體多執行緒。即便處理器只能運行一個執行緒，作業系統也可以通過快速的在不同執行緒之間進行切換，由於時間間隔很小，來給用戶造成一種多個執行緒同時運行的假象。這樣的程序運行機制被稱為軟體多執行緒[14]。在本文裡的多執行緒是指軟體多執行緒，也就是利用多執行緒方式，減少 I/O 的延遲時間。

在 C# 中，執行緒的使用非常的簡單，有不傳參數的也有傳參數的，皆可以使用。而對於執行緒的使用，一個核心搭配一個執行緒可達到執行緒的最大利用。

在 C# 中僅需加入下列一行，即可使用。

```
using System.Threading;
```

### 2.3.6 TPL(Task Parallel Library)

近年來隨著多核電腦的普及，並行程式設計技術(多核程式設計技術)也逐漸為開發的主流。在 .NET Framework 4 中就引入了“並行程式設計”。如： Task Parallel Library, Parallel LINQ 等。[14]

而在傳統的程式設計模型中，程式設計師負責創建執行緒，為執行緒分配任務，管理執行緒。而在 .NET Framework 4 中的並行程式設計是依賴 TPL 實現的。簡單的說程式設計師只需指派 task，TPL 就會自動的管理執行緒，而傳統的多執行緒是以人工為導向的，所以當我們使用 TPL 中的並行技術的時候來執行多個 task 的時候，我們不用在關心底層創建執行緒，管理執行緒等。[14]

在 C# 中僅需加入下列兩行，即可使用。

1. using System.Threading;
2. using System.Threading.Tasks;

程式碼的寫法非常簡單，僅需將 for 迴圈形式改成 Parallel.For 的形式，即可完成。

例：

```
for (int i = 0; i <= Width; i++)  
{  
    Do something...  
};
```

改成

```
Parallel.For(0, Width, i =>  
{  
    Do something...  
});
```

### 2.3.7 Profile

Profile 的用途就是可以分析程式碼的執行時間，在本文裡使用到兩種 profile，一個是 Visual Studio 2010 本身的 profile，另一種是 NVIDIA 的 CUDA toolkit 裡的 Visual profile 可以對 NVIDIA 顯示卡的使用進行分析。

## 第三章 前處理與道路偵測

### 3.1 前處理

本文在研究過程中，一共使用了三種不同的方法對影像進行前處理，第一種是 C# 的指標運算方式，第二種是使用 OPENCV 函數庫，第三種是使用 OPENCL。使用上述三個方法對原始彩圖做灰階化、水平濾波、水平邊緣偵測，垂直邊緣偵測及邊緣偵測(包括水平和垂直邊緣偵測特徵)，由於後面偵測部分的程式需要由下往上對影像掃描，所以須將部分影像順時針旋轉 90 度，讓 C# 的指標可以以由左至右的方式對原來影像做由下往上的存取。前處理共產生一共有 8 張圖，如下表：

表 3-1 圖片名稱與用途

圖	程式變數名稱	用途
原圖	tempOrigin	近車偵測
灰階化	tempNinjaT	其他的影像處理來源
水平濾波	tempGray	道路線偵測、陰影偵測
邊緣檢測	tempSobel	顯示用
水平邊緣檢測	tempSobelH	車輛頂部偵測、顯示用
垂直邊緣檢測	tempSobelV	車輛左右邊緣偵測、顯示用
水平濾波(旋轉)	tempGrayRotate	道路線偵測、陰影偵測
垂直邊緣檢測(旋轉)	tempSobelVRotate	車輛左右邊緣偵測

上述的圖格式皆是 24bit，旋轉前的圖長寬都是 800\*450，旋轉後的最後兩張圖長寬都是 450\*800，其中最後兩張圖做旋轉的目的如上面所敘是為了使記憶體連續存取，加快記憶體存取。



### 3.1.1 均值濾波

如第二章所述，一般均值濾波算子如下：

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 9 \quad (\text{傳統均值濾波運算子}) \quad (4)$$

因為使用傳統的均值濾波會破壞原有畫面的特性，例如陰影的邊緣會變模糊，而本文程式偵測部分需由下往上搜尋陰影邊緣，如果使用傳統的均值濾波，一些較不明顯的陰影邊緣經由傳統均值濾波模糊化後，可能會找不到，因此本文只對水平方向做均值濾波，如公式 5，目的在於消除道路上的雜點，幫助道路線偵測和陰影偵測時能跳過道路，正確地找到道路線特徵及陰影特徵。

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} / 3 \quad (5)$$

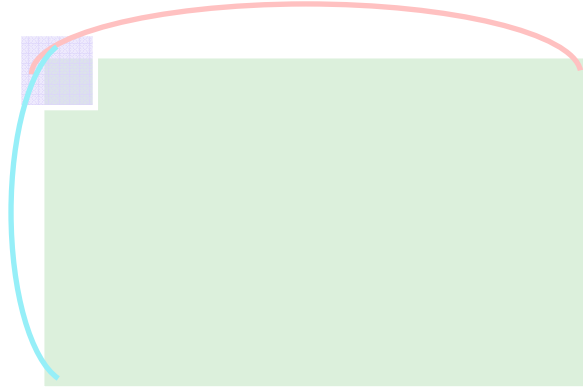
### 3.1.2 邊緣偵測

本論文所採用的 Sobel 邊緣運算子如下：

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad (\text{垂直邊緣偵測}) \quad (6)$$

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (\text{水平邊緣偵測}) \quad (7)$$

本文中的邊緣檢測處理皆有對畫面邊緣處理，處理方式為當在畫面邊緣時，算子欠缺的像素值會以畫面另一邊的像素替補，例如是上邊緣，則以畫面最底部像素替補，左邊緣，則以畫面最右邊像素替補，其他兩邊依此類推，如下圖 Fig.8。



**Fig.8** 欠缺像素處理，上邊緣，則以畫面最底部像素替補，左邊緣，則以畫面最右邊像素替補

### 3.1.3 C#指標運算

C#在使用指標前，需先允許使用 `unsafe{}` 並將使用指標的程式碼放置 `unsafe{}` 中，而在一開始需使用 `LockBits()` 將 `Bitmap` 鎖住，並把 `Bitmap` 的圖片內容以 `Byte Array` 形式存在記憶體中，接著利用指標讀取 `Byte Array` 相對應圖片內容所需的像素 `BGR` 值，並做運算，再將其利用指標方式存回空白的 `Bitmap` 中，並在結束時 `UnLockBits()` 釋放記憶體空間。

使用 C# 的指標運算，處理所有未旋轉的圖片內容，其中灰階化部分僅是公式轉換即可得到各像素的灰階值。接著將灰階化的圖當輸入產生水平濾波值、水平及垂直邊緣檢測值存在 `Bitmap` 中，再將此 `Bitmap` 資訊產生水平濾波圖及所有邊緣檢測圖，初始的原始程式碼附在附錄一。

### 3.1.4 OpenCV

C#中的 OpenCV 使用方式如下：

1. 先將 `Bitmap` 圖轉成 `Image<>` 格式
2. 使用 OpenCV 的函數運算
3. 將 `Image<>` 格式轉回 `Bitmap`

使用方式就是呼叫 OpenCV 的函數進行處理，沒有太特別的地方，唯一比較需要注意的地方在計算邊緣檢測值時，可能會產生負數，而在 C# 的 OpenCV 並無函數可以計算絕對值。替代的做法是使用 `cvAbsDiff` 函數，此函數的功用是將兩張圖相減取絕對值，所以把邊緣檢測圖和全像素為 0 的圖使用此函數相減取絕對值，如此就可以替代所需要的邊緣檢測圖取絕對值，再將這些數值圖利用 C# 指標處理的方式，轉成所需要的水平濾波圖及所有邊緣檢測圖。

### 3.1.5 OpenCL

C# 中的 OpenCL 名稱是 OpenCL.NET[22]，使用流程如下：

1. 選定使用的設備，可以是顯示卡或是多核心 CPU
2. 撰寫核心計算用的 kernel code 並編譯成 kernel
3. 將要計算的資料傳至設備
4. 使用 `EnqueueNDRangeKernel()` 執行核心
5. 使用 `EnqueueBarrier()` 等待執行完畢
6. 將計算好的資料傳回至主機

註：如果不清楚如何使用可參考 OpenCL.NET 的範例程式[22]

OpenCL 較難的地方在於撰寫 kernel code，其餘部分皆照範例程式一一呼叫即可，在未優化前由於初次接觸且力求簡單，因此使用了 OpenCL 的 `CL.Image` 作為使用的記憶體物件，這是 2 維圖像的格式，而 kernel code 撰寫如附錄二

### 3.1.6 圖片旋轉

C# 旋轉圖片十分簡單，使用 `Bitmap` 中的函數 `RotateFlip()` 即可旋轉圖片。本文中以順時針 90 度方向旋轉了水平濾波圖，垂直邊緣檢測圖兩張圖。

## 3.2 道路線偵測

有了各種前處理後的影像，接著就是對道路線進行偵測。在對道路線偵測前，首先需先觀察道路線的特徵，以下是不同行車紀錄器影片中擷取出的圖片(影片來源為 youtube)。

(a).



(b).



(c).



(d).



(e).



(f).



Fig.9 (a).鄉村道路, (b).國道前, (c).交流道下, (d).路口, (e).高速公路, (f).市區

使用觀察像素亮度的方法可以發現上述六張圖可以找出下述特徵：

重要特徵：

1. 左右道路線的延伸交點是地平線的消失點。
2. 柏油路上的路面相鄰像素的亮度差不會大於 4。

3. 不管道路線是否明顯，道路線的特徵會有著 暗(道路) $\leftrightarrow$ 明(道路線) $\leftrightarrow$ 暗(道路) 的特徵。
4. 由底向上掃描，當左邊和上面像素的亮度大於本身像素的亮度 4 時，此像素是左道路線的邊緣。
5. 由底向上掃描，當右邊和上面像素的亮度大於本身像素的亮度 4 時，此像素是右道路線的邊緣。

次重要特徵：

1. 由於行車紀錄器的架設，有些行車紀錄器的底部會有車前蓋，在掃描道路線時需跳過。
2. 道路線在無轉彎的情況下是接近直線的。
3. 地平線交點以上為天空或者是路邊的建築，可以忽略。
4. 道路線的寬度相同遠近會小於斑馬線的寬度。
5. 道路線上可能有字，是干擾的特徵。
6. 左右道路線可能是雙黃線

基於上述的特徵，以及本文研究目標僅限於前車偵測，因此想出了演算法，可以精確的標出最內側的兩條道路線，並且不受路面標字的影響，如下面兩張圖。

(a).



(b).

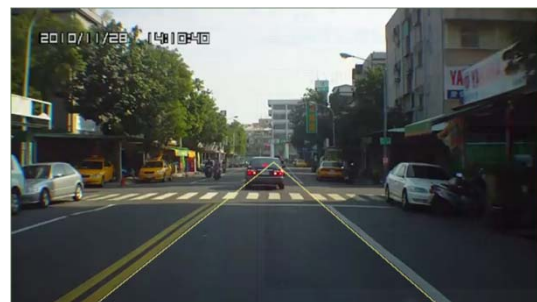


Fig.10 (a).路面標字干擾, (b).雙黃線

### 3.2.1 道路線掃描

本文裡的道路線特徵是採用畫面掃描的方式尋找，方法為由下而上，由左而右的掃描道路線特徵，由於目標是前車的左右道路線，因此採用此種方式掃描前車左右道路線的下方並不會受到其他道路線的遮蔽，也就是在同一行裡前車左右道路線的特徵會第一個找到，因此每行由下而上掃描只需找到第一個道路線特徵即可停止換下一行，可以節省很多的計算時間。另外由於前車左右道路線延伸會交於地平線消失點，在大量的觀察不同行車紀錄器的影片後，可以設定一個最高搜尋畫面的高度，當由下而上掃描到一定高度時(本文是設距離畫面底部 300pxiel)，代表超過地平線交點高度，找不到道路線，省略搜尋天空的時間，可節省很多計算的時間。

另外在 C#程式裡，水平方向掃描對記憶體才能連續存取，因此為了加快掃描速度，必須將要使用的圖向右旋轉 90 度，才可從畫面底部向上搜尋，如下圖。

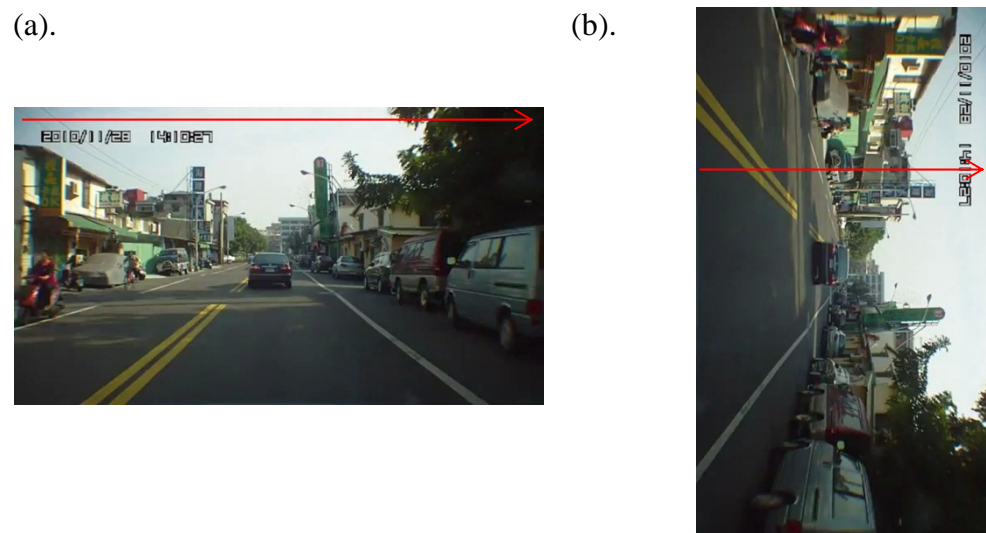


Fig.11 (a).旋轉前畫面掃描方向 (b).旋轉後畫面掃描方向

演算法如下：

Minluminance = 4

MaxLanewidth = 30

1. 由下而上，由左而右掃描

2. 由於道路線都會在道路的上方，所以：

(當目前的像素上方的第二個像素亮度-目前的像素亮度) $>$  Minluminance，是道路線掃描點，道路線掃描點需判斷是否為左道路線特徵或右道路線特徵。

左道路線特徵：

- a. (目前的像素左邊的第二個像素亮度-目前像素亮度) $>$  Minluminance
- b. 目前像素亮度和右邊第二個像素亮度相近
- c. 往左 MaxLanewidth 長度內會出現亮 $\rightarrow$ 暗

右道路線特徵：

- a. (目前右邊第二個像素亮度-目前像素亮度) $>$  Minluminance
- b. 目前像素亮度和左邊第二個像素亮度相近
- c. 往右 MaxLanewidth 長度內會出現亮 $\rightarrow$ 暗

3. 判斷左右道路線，失敗則繼續向上搜尋其他像素，成功則紀錄該點為掃描點並換行搜尋：

紀錄為 1：黃色為左道路線掃描點

紀錄為 2：藍色為右道路線掃描點

紀錄為 3：紅色為道路線掃描點

4. 直到結束

掃描結束會得到一個 800 大小的結構陣列，每個結構陣列儲存該點的 x 座標、y 座標及特徵類型，但轉換到圖上會出現圖 Fig.12。





Fig.12 黃色為左道路線掃描點，藍色為右道路線掃描點，紅色為道路線掃描點

### 3.2.2 回歸直線

回歸直線是一種統計的方法，他可以在座標圖上求出代表輸入的二維數據  $(x_i, y_i)$  的一條直線，其中  $i=1, 2, \dots, n$  也就是這條直線和這些數據點的距離和會最小，回歸直線法的使用方式很簡單，只需把同一群的二維數據代入公式，即可算出這群代表直線的斜率和截距，公式如下：

$$y = a + b \times x$$

其中

$$b = \frac{n \sum x_i y_i - \sum x_i \sum y_i}{n \sum x_i^2 - (\sum x_i)^2}$$

$$a = \frac{n \sum x_i^2 \sum y_i - \sum x_i \sum x_i y_i}{n \sum x_i^2 - (\sum x_i)^2} \quad (8)$$

$\sum x_i y_i$  為所有座標的  $x_i \times y_i$  值的和， $\sum x_i$  為所有座標  $x_i$  值的和， $\sum y_i$  為所有座標  $y$  值的何， $\sum x_i^2$  為所有  $x_i \times x_i$  值得和。

有了回歸直線法的使用方法後，接著將道路線掃描得到的 800 大小的結構陣列使用下面的 pseudo code 找出連續的點，並將不同段的連續點分別帶入回歸直線，算出數條連續直線，並從數條連續直線中找出前車偵測的左右道路線，左右道路線分別是斜率最大及最小的直線。



註：在程式中的座標位置和一般座標系統中的座標位置是不一樣的，一般座標系統中的座標原點位於畫面左下角，朝左上遞增，而程式中的座標位置的座標原點位於畫面的左上角，朝左下遞增，因此斜率在程式中表示也和現實世界中的表示也是不太一樣，是水平鏡像反過來的，如圖 Fig.13，下面 pseudo code 座標採用的是程式中座標。

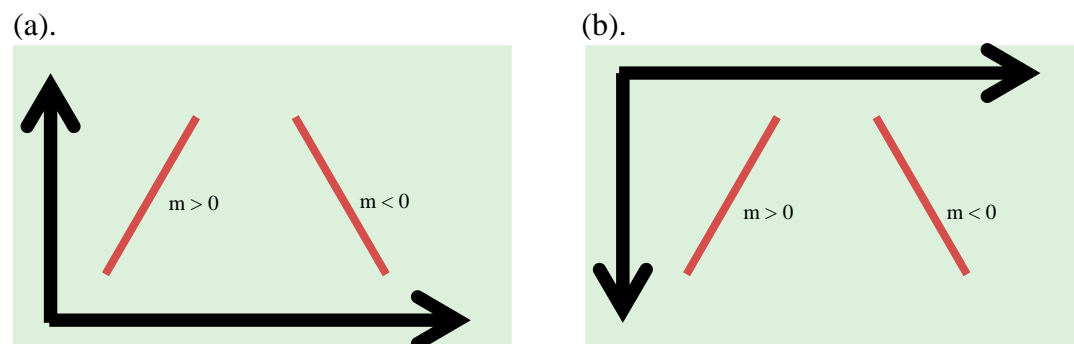


Fig.13 (a).現實世界座標及斜率示意圖, (b).程式中座標及斜率示意圖

pseudo code 如下：

```

輸入：800 大小的結構陣列 lanearray[800]
結構形式：
Struct lanerecord{
    int x;    //x 座標
    int y;    //y 座標
    int type; //type = 1 左道路線特徵點 type = 2 又道路線特徵點
              //type = 3 道路線特徵點}
輸出：左右道路線斜率及截距
參數：
    MinContiPoint = 20    //最小連續點數量
    MaxAvgError = 5      //平均誤差長度

MiddleLeftRightLane()
begin
    flag = 0;
    for( i = 0; i < 800; i++ )
    begin
        if(flag == 0)    //道路線起點
        begin
            if( lanearray[i] == 1 || lanearray[i] == 2 )
            begin
                初始化回歸直線計算變數
                紀錄目前道路線起點和終點陣列索引
                flag = 1; cnt = 0;
            end
        end
        continue;
    end
end

```



## 第四章 前車偵測

### 4.1 前車偵測

在有了前車兩旁的左右道路線後，便可以在前車左右道路線範圍內開始進行前車偵測，前車偵測的流程會先利用車尾燈的紅色特徵偵測近距離的車輛，在近距離車輛找不到的情況下，接著進入前車偵測的主軸。前車偵測的流程是先偵測畫面中亮度較暗的車底陰影，接著偵測陰影上方的車輛左右邊緣，最後偵測車輛的頂部，並進行簡單的車輛驗證，最後把判斷正確的车辆框出來，完成前車偵測的流程，結果如圖 Fig.14。

(a).



(b).



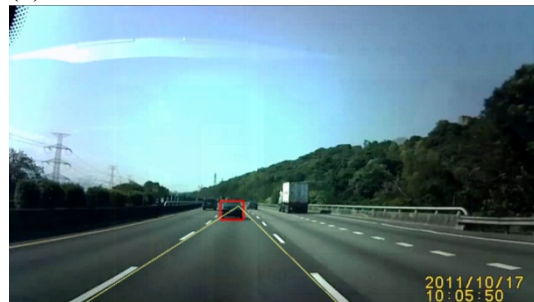
(c).



(d).



(e).



(f).



(g).



(h).



Fig.14 (a).鄉村道路, (b).近距離車輛, (c).路面極黑的高架橋下, (d).卡車, (e).高速公路上, (f).有陸橋陰影干擾, (g).畫面些微反光, (h).鄉鎮道路

### 4.1.1 近車偵測

在本文裡近距離的車輛，在找到的測試影片裡均保有紅色的車尾燈特徵，如圖 Fig.14(b)，因此本文使用基於車道線的方法，先對近距離的車輛進行偵測。

近車偵測判斷顏色的方法是依據 HSV 色彩空間設定的特徵值搜尋紅色，判斷紅色的特徵值如下：

1.  $H \leq 25$  or  $H \geq 335$
2.  $S \geq 0.8$
3.  $V \geq 80$  (9)

由於單憑紅色特徵還是有可能把道路上的雜點判斷為車尾燈的紅色特徵，因此本文裡的車尾燈還必須在水平方向有連續六個紅色的特徵點才算是車尾燈特徵，用此依據過濾掉路面上的雜點。

pseudo code 如下：

輸入：hahaha.TempOrigin(原圖)
輸出：車輛偵測框選資訊
參數：
SearchBottom = 5      //距離畫面底部的高度
VehicleLampFindHeight = 330      //車燈搜尋高度( 距離畫面底部的高度 )
CloseVehicleJudge()
begin
if( FindLamp() )      //可得到近車車尾燈最左和最右紅色特徵的座標位置
begin

```

    車輛框選的左下角和右下角為距離最左和最右紅色特徵下方寬三分之一的長度
    車輛框選的左上角和右上角為車輛框選的左下角和右下角上方約寬三分之一的 1.4 倍
    的高度
    return true;    //偵測成功
end
return false    //偵測失敗
end

FindLame()
begin
    for( j = 450 - 1 - SearchBottom; j >= VehicleLampFindHeight; j-- )
    begin
        for( i = y 軸為 450 的左道路線 x 位置; i < y 軸為 450j 的右道路線 x 位置; i++ )
        begin
            if( 像素[i, j]為紅色 )
            begin
                if( 像素[i+1, j] ~ 像素[i+6, j]皆為紅色 )
                begin
                    儲存 i 最小的[i, j]為最左邊車尾燈座標
                    儲存 i 最大的[i+6, j]為最右邊車尾燈座標
                end
            end
        end
        if( j==331 && 此高度有紅色特徵 )
            VehicleLampFindHeight = VehicleLampFindHeight - 5;
        end
        if( (最右邊車燈座標 y 值 - 最左邊車燈座標 y 值) <= 30 )
            存車尾燈最左和最右紅色特徵的座標位置
        if( (最右邊車燈座標 x 值 - 最左邊車燈座標 x 值) <= 100 )
            return false;    //沒找到近車
        if( 最左邊車尾燈和最右邊車尾燈找到 )
            return true;    //找到近車
        end
    end
end

```

表 4-1 近車偵測的 pseudo code

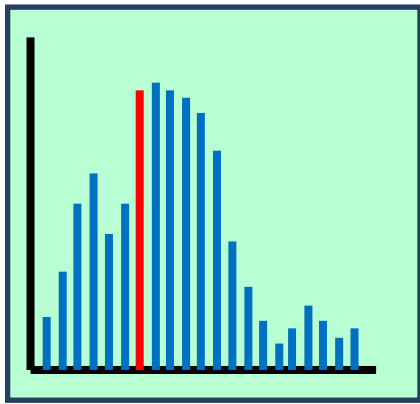
### 4.1.2 自動陰影臨界值

在近車偵測失敗後，即開始一般車輛偵測流程，本文中可以使用自動陰影臨界值選擇，也可使用預設的陰影臨界值來偵測陰影。使用自動陰影臨界值可以適應各種環境變化，找到合適的車輛陰影的臨界值，但是也有一個壞處是，當左右車道線範圍內沒有車輛時，自動陰影臨界值會找到錯誤的臨界值，導致判斷錯誤。本文裡一共想出了四種自動陰影偵測方法，但經過實際測試，發現第四種效果不太好，因此主要是選擇前三種自動陰影臨界值其中一種選擇需要的臨界值。前三種方法在於判斷陰影臨界值方法不太相同，實測結果發現效果是差不多的，因此

在本文裡將介紹前三種不同的陰影判斷方法。

### A. 自動陰影臨界值方法一

做法是由下而上，由左而右統計道路線內所有不同亮度的數量成直方圖，由直方圖亮度低往亮度高取有數量的亮度第 AutoThresholdcnt1(參數值：預設為 20)個亮度值-1 為陰影最大亮度，示意圖如 Fig.15，水平軸代表由小至大的亮度值，垂直軸代表車道線內該亮度值之總圖素個數。



**Fig.15 道路線間的直方圖亮度由低向高取第 AutoThresholdcnt1 個有值的亮度-1**

自動陰影臨界值方法一的 pseudo code 如下：

輸入：hahaha.TempRotate(水平濾波灰度圖(旋轉)) 輸出：ShadowMaxluminance //陰影最大亮度臨界值 參數： AutoThresholdcnt1 = 20
AutoVehicleThreshold1 () begin for( j = 0; j < 800; j++ ) begin for( i = 0; i < 底部距離左右車道線j位置的高度; i++ ) begin 直方圖graylevel[此像素的luminance] += 1; end end end for( i = 0; i < 256; i++ ) begin if( graylevel[i] != 0 ) begin cnt++; end end end

<pre> if( cnt == AutoThresholdcnt1 )     ShadowMaxluminance = i - 1; end end end end </pre>
表 4-2 自動陰影臨界值方法一的 pseudo code

## B. 自動陰影臨界值方法二

做法是由下而上，由左而右統計道路線內不同行所有不同亮度的數量成直方圖，一行統計一次直方圖，由直方圖亮度低往亮度高取有數量的亮度第 AutoThresholdcnt2(參數值：預設為 14)個亮度值-1，為每個直方圖中陰影最大亮度，選所有直方圖中陰影最大亮度中最小一個的當作這張圖的陰影最大亮度，示意圖如 Fig.15，水平軸代表由小至大的亮度值，垂直軸代表車道線內某行為該亮度值之圖素個數。這個做法分別統計每個垂直線（即每行）的直方圖的用意是，當車道線內有車子陰影，其有車子陰影的垂直線中找出的陰影最大亮度會是所有垂直線的陰影最大亮度中最小的，因此這也是一種找陰影最大亮度的方法。

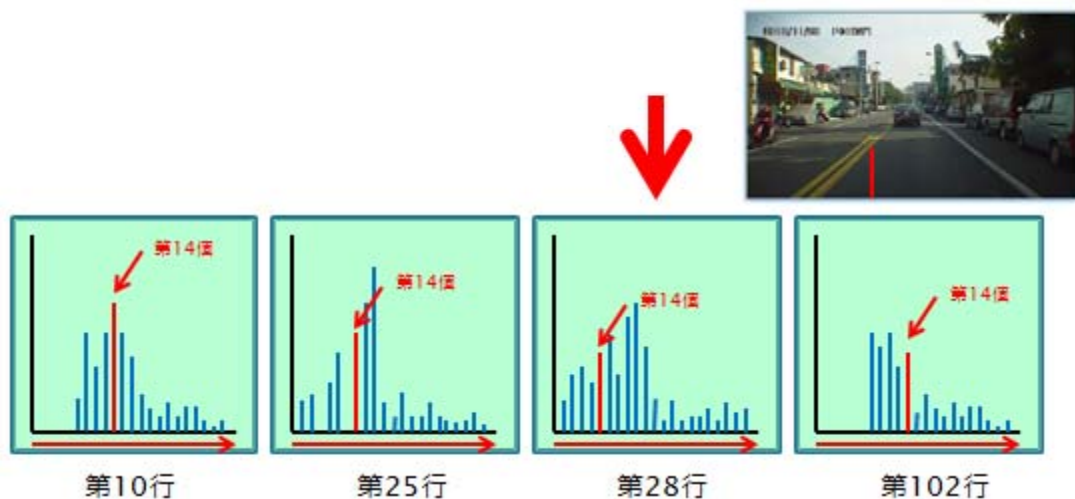


Fig.16 道路線間不同行的直方圖亮度由低向高取第 AutoThresholdcnt2 個有值的亮度-1

自動陰影臨界值方法二的 pseudo code 如下：

輸入：hahaha.TempRotate(水平濾波灰度圖(旋轉))
-----------------------------------

輸出：ShadowMaxluminance     //陰影最大亮度臨界值 參數： AutoThresholdcnt2 = 14
<pre> AutoVehicleThreshold2 () begin   for( j = 0; j &lt; 800; j++ )   begin     直方圖graylevel[]初始化     for( i = 0; i &lt; 底部距離左右車道線j位置的高度; i++ )     begin       直方圖graylevel[此像素的luminance值] += 1;       cnt = 0;     end     for( i = 0; i &lt; 256; i++ )     begin       if( graylevel[i] != 0 )       begin         cnt++;         if( cnt == AutoThresholdcnt2 )         begin           if( i &lt; min )           begin             min = i;             break;           end         end       end     end   end   end   ShadowMaxluminance = min - 1; end </pre>
表 4-3 自動陰影臨界值方法二的 pseudo code

### C. 自動陰影臨界值方法三

想法是車輛底下的陰影不管在遠還是近實際大小是固定的，但由於遠近關係，近的看起來比較大，遠的較小，所以須對遠的地方乘上權重，使的遠近陰影佔左右車道線內的比例更接近固定，又由於道路線是平行的直線，在圖上是呈三角形縮小，所以可以推測出水平方向權重的值是跟隨水平左右車道線間的距離以線性的比例放大，如圖 Fig.16。



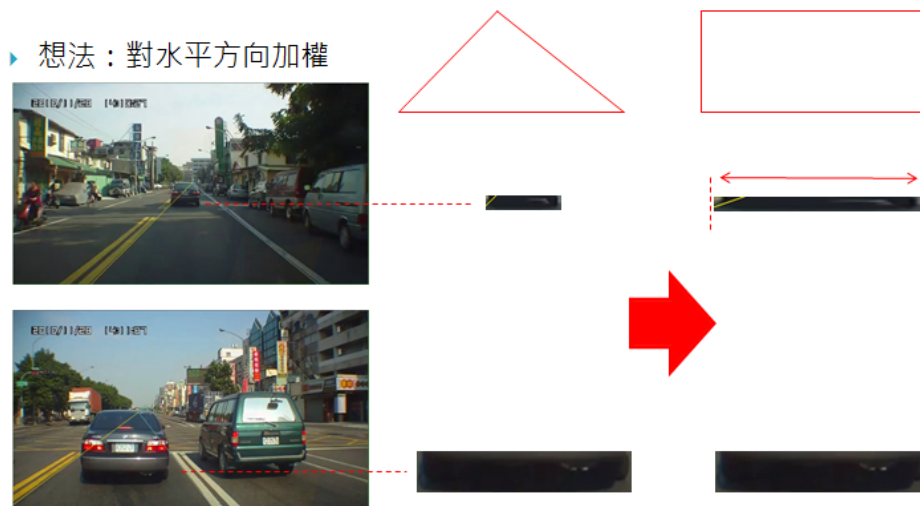


Fig.16 自動陰影臨界值方法三加權想法示意圖

對本文自動陰影臨界值方法三的權重算法舉個例，假設不同列左右道路線內寬度最大 100pixel，最小 2pixel，那最小寬度的每個點須各乘上權重 50，統計到直方圖裡此亮度值的數量，依此類推。

做法是由左而右，由上而下對不同列道路線內的不同亮度值的數量乘上權重，累積成直方圖，由直方圖亮度低往亮度高取佔全部加權後數量 AutoThresholdcnt3(參數值：預設為 0.04)比例的亮度值為陰影最大亮度，示意圖如 Fig.17。

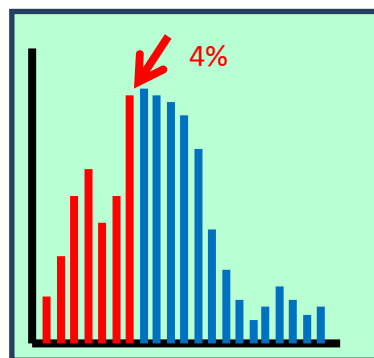


Fig.17 道路線間的加權直方圖亮度由低向高累積到 AutoThresholdcnt3 比例數量的亮度值

自動陰影臨界值方法三的 pseudo code 如下：

輸入：hahaha.TempRotate(水平濾波灰度圖(旋轉))  
 輸出：ShadowMaxluminance //陰影最大亮度臨界值  
 參數：

AutoThresholdcnt3 = 20  
 SearchBottom = 5

```

AutoVehicleThreshold3 ()
begin
  for(j = 左右道路線交點y值+1; j < 450 - SearchBottom; j++)
  begin
    加權值為道路線內最大寬度/本列寬度
    for(i = 左道路線y值為j的x座標; i < 右道路線y值為j的x座標; i++)
    begin
      graylevel[此像素的luminance值] += 加權值;
    end
  end
  for(i = 0; i < 256; i++)
  begin
    cnt += graylevel[i];
    if( cnt == 加權後的總數量*AutoThresholdcnt3 )
      ShadowMaxluminance = i;
  end
end
  
```

表 4-4 自動陰影臨界值方法三的 pseudo code

### 4.1.3 陰影偵測

在找到最大陰影亮度臨界值後，接著是對陰影進行偵測。在偵測之前，先觀察前面 Fig.9 的那幾張不同影片的畫面，找到這幾個畫面陰影的共同特徵如下：

1. 道路間相鄰像素亮度差不會大於 5
2. 陰影和道路的交界會有一條邊界，且兩者亮度差大於 5

基於這兩個特徵，加上本文是基於車道線的前車偵測，偵測目標為距離畫面較近車輛，因此本文決定使用由下而上，由左而右的掃描方式掃描陰影的輪廓，當輪廓是連續時判斷為同一物件，記錄在車輛儲存資訊裡，陰影輪廓掃描結果如圖 Fig.18。

(a).



(b).



Fig.18 由於處理時需由下而上，因此使用旋轉後的水平濾波灰度圖做處理，結果座標會使用座標轉換回正向的圖的座標，紅色為陰影輪廓掃描結果，結果顯示近車和遠車皆可陰影輪廓掃描，(a).為遠距離車輛 (b).為近距離車輛。

pseudo code 如下：

輸入：hahaha.TempRotate(水平濾波灰度圖(旋轉)) 輸出：車輛資訊陣列 參數： ShadowMaxluminance = 40(預設) MinShadowluminance = 5 SearchBottom = 5(預設)
<pre>VehicleShadow () begin     count = 0;     for( j = 左道路線畫面中最上面y座標; j &lt;= 右道路線畫面中最下面y座標; j++ )         begin             for( i = SearchBottom; i &lt; 左右道路線x = j的y值; i++ )                 begin                     if( 像素[i, j] 亮度-像素[i+1, j] 亮度 &gt;= MinShadowluminance &amp;&amp; 像素[i+1, j] 亮度 &lt;= ShadowMaxluminance )                         begin                             if(flag == 0)                                 begin                                     紀錄陰影輪廓起點座標                                     紀錄上一個陰影輪廓特徵點座標                                     count++;                                     flag = 1;                                     breaknum = 0;                                     break;                                 end                             end                             if( 現在陰影輪廓特徵點和前一個陰影輪廓特徵點連續 )                                 begin                                     count++;                                     breaknum = 0;                                     紀錄上一個陰影輪廓特徵點座標                                 end                             end                         end                     end                 end             end         end     end</pre>

<pre> end else begin     if( count &gt; 3 )    //三點連續才算陰影     begin         紀錄陰影輪廓點終點座標及中點座標     end     if( breaknum &gt;= 5 )    //找到不連續點5次     begin         if( count &gt; 3 )             車輛陰影輪廓物件找到，換記錄下一個             flag = 0;             count = 0;             j = 上一個陰影輪廓特徵點座標y值;             continue;         end     end end end if( flag == 1 ) begin     if( breaknum &gt;= 5    j == 右道路線畫面中最下面y座標 )     begin         紀錄陰影輪廓點終點座標及中點座標         車輛陰影輪廓物件找到，換記錄下一個     end     count = 0;     flag = 0; end if( 現在陰影輪廓特徵點和前一個陰影輪廓特徵點不連續 )     breaknum++; end end </pre>
<p>表 4-5 陰影輪廓偵測的 pseudo code</p>

#### 4.1.4 陰影資訊排序

在有陰影資訊後，由於物件不一定只找到一個，而本文基於車道線的前車偵測目標是車道線間最近的一台車，為了顯示時判斷方便，因此需對陰影物件進行排序，陰影物件離底部越近的排在陰影資訊的越前面，排序方法為氣泡排序法。

#### 4.1.5 車輛左右邊緣偵測

在排序之後，即是偵測車輛左右邊緣。由於本文偵測車輛左右邊緣需找到比

較正確的陰影中點，才能往左右偵測車輛左右邊緣，再加上車道線間的陰影可能是不完全的陰影，如切換車道的車輛可能只有一部份陰影在車道線間，因此需對每個可能的陰影物件向左右畫面延伸，找到較完整車輛的陰影物件（可能超出車道線範圍），並算出此陰影中點，在此次延伸中，本演算法使用範圍偵測陰影的方法偵測陰影最左邊和最右邊，如表 5-6 的 pseudo code。

pseudo code 如下：

輸入：hahaha.TempGray，車輛資訊陣列 輸出：車輛資訊陣列 參數： ShadowMaxluminance = 40(預設)
<pre> findvehiclePossiblerange () begin   for( k = 0; 車輛資訊[k]為空; k++ )   begin     以車輛資訊[k]中點座標為起點     for( i = 車輛資訊[k]中點座標x值; i &gt;= 0; i-- )     begin       if( 起點和起點上方10個像素中所有像素亮度 &gt; ShadowMaxluminance )         break;       else         紀錄車輛資訊[k]的左陰影座標     end     for( i = 車輛資訊[k]中點座標x值; i &lt; 800 0; i++ )     begin       if( 起點和起點上方10個像素中所有像素亮度 &gt; ShadowMaxluminance )         break;       else         紀錄車輛資訊[k]的右陰影座標     end   end end end </pre>
表 4-6 陰影大小校正的 pseudo code

接著便是偵測車輛的左右邊緣。本文提供兩種車輛左右邊緣的方法可供選擇，第一種車輛左右邊緣的方法優點在於對於遠距離車輛左右邊緣偵測結果較佳，但由於依據車輛左右下角垂直邊緣特徵，在有些狀況下不一定會有，如畫面經過對比調整後，陰影特徵吃掉輪胎的垂直邊緣特徵，導致偵測不到陰影，如圖 Fig.19。



**Fig.19 陰影吃掉輪胎旁的垂直邊緣特徵，導致第一種左右邊緣偵測方法失效**

而第二種左右邊緣偵測方法，則是略過底部垂直邊緣特徵，偵測主要是依據車身的垂直邊緣特徵，在 Fig.19 中不會受到過黑陰影的干擾偵測車輛左右邊緣，但是缺點為速度較慢，在本文中依序介紹這兩種左右邊緣偵測方法。

第一種左右邊緣偵測是利用對垂直邊緣特徵採範圍判斷方式累積，示意圖如 Fig.20，假設紅點是車輛垂直邊緣特徵，針對某行找垂直邊緣資訊時，將參考鄰近的四行資料，把這五行的垂直邊緣做統計，統計方式為，假設這五行中的一列至少有一個垂直邊緣特徵，則該行投影累計加一，由下往上依序一定高度的所有列之後，把累計的數字存到垂直投影陣列中代表該行的位置，當對垂直投影陣列中每個位置（即每行）做累積後，由陣列中間往左右找一定投影累計數量比例的行位置，此  $x$  座標為新左右車輛邊緣的座標。

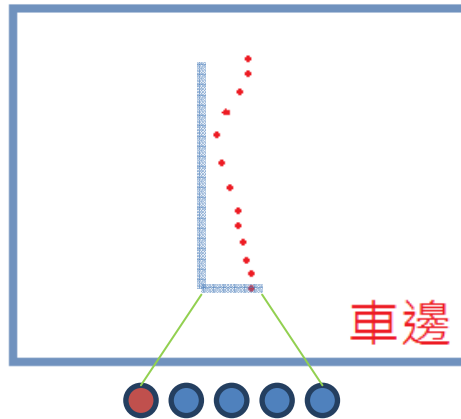


圖 Fig.20 第一種左右邊緣偵測原理示意

pseudo code 如下：

```

輸入：hahaha.TempSobelV，車輛資訊陣列
輸出：車輛資訊陣列
參數：
    EdgeRateH = 0.7
    CloseVehicleWidth = 200

findvehicleaccuratewidth1 ()
begin
    for( k = 0; 車輛資訊[k]為空; k++ )
        begin
            verticalproject(k);    //可得到垂直投影陣列，大小為三倍車寬
            以車輛資訊[k]中點座標x值-垂直投影[0]代表的座標x值為起點
            if( 陰影寬度 < 50 )
                EdgeRateH = 0.7;
            else if( 陰影寬度<100 && 50 <= 陰影寬度 )
                EdgeRateH = 0.5;
            else
                EdgeRateH = 0.7
            for( i = 車輛資訊[k]中點座標x值-垂直投影[0]代表的座標x值; i >= 0; i-- )
                begin
                    if( 垂直投影[i] >= 陰影寬度/2*EdgeRateH )
                        begin
                            紀錄車輛資訊[k]的左下角座標
                            break;
                        end
                    end
                end
            for( i = 車輛資訊[k]中點座標x值-垂直投影[0]代表的座標x值; i < 垂直投影陣列大小;
i++ )
                begin
                    if( 垂直投影[i] >= 陰影寬度/2*EdgeRateH )
                        begin
                            紀錄車輛資訊[k]的右下角座標
                            break;
                        end
                    end
                end
            end
        end
    end
end

```

```

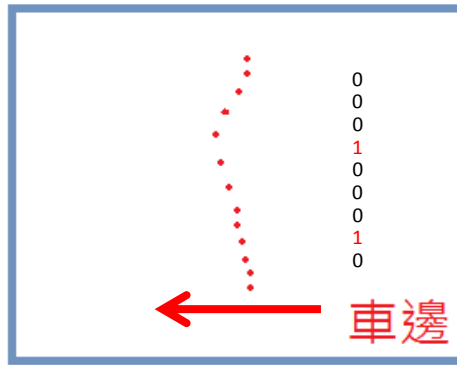
verticalproject(k)
begin
  if( 車輛資訊[k]陰影寬度 > CloseVehicleWidth )
    mode = 0;    //近車
  else
    mode = 1;    //遠車
  for(j = 0; j < 車輛陰影寬度/2; j++)
    begin
      for(i = 車輛資訊[k]的中點座標x值; i >= 垂直投影[0]代表的座標x值; i++)
        begin
          if( 像素[i, j]~像素[i+7, j]有一個點是垂直邊緣特徵 && mode == 0 )
            垂直投影[i-垂直投影[0]代表的座標x值]++;
          else if( 像素[i, j]~像素[i+2, j]有一個點是垂直邊緣特徵 && mode == 1 )
            垂直投影[i-垂直投影[0]代表的座標x值]++;
        end
      end
    end
  for(j = 0; j < 車輛陰影寬度/2; j++)
    begin
      for(i = 車輛資訊[k]的中點座標x值; i < 垂直投影[垂直投影陣列大小]代表的座標x值;
i--)
        begin
          if( 像素[i, j]~像素[i-7, j]有一個點是垂直邊緣特徵 && mode == 0 )
            垂直投影[i-垂直投影[0]代表的座標x值]++;
          esle if( 像素[i, j]~像素[i-2, j]有一個點是垂直邊緣特徵 && mode == 1 )
            垂直投影[i-垂直投影[0]代表的座標x值]++;
        end
      end
    end
end
end

```

表 4-7 左右邊緣偵測方法一的 pseudo code

第二種左右邊緣偵測是利用對垂直邊緣特徵採垂直陣列更新的方式判斷，示意圖如 Fig.20，假設紅點是車輛垂直邊緣特徵，使用一個初始值為 0 的垂直陣列，由陰影中點往左右方向平移，當相對應的座標是垂直邊緣特徵，則將陣列中該位置的值由 0 更新為 1，當垂直陣列由 0 到 1 的陣列元素達到一定比例的數量時，此位置對應的座標為車子邊緣座標。





**Fig.21 第二種左右邊緣偵測原理示意圖**

實際上程式在實作時，會遇到一些車輛背後的垂直特徵干擾，以致於左右邊緣偵測不是很完美，因此稍作修正，搜尋起始位置為車輛陰影中點往左右車輛陰影寬度 0.3 倍的位置，向左右方向更新垂直陣列(0 $\rightarrow$ 1)，當垂直陣列由 0 到 1 的陣列元素達到一定比例的數量時，代表已經找到垂直邊緣的一個邊界，此時繼續向左右方向更新垂直矩陣，但是這次反過來是當原來為 1 的位置的像素為非垂直邊緣特徵，則 1 $\rightarrow$ 0，當有一半的 1 被更新回 0 時，此位置為車輛左右邊緣位置的另一個邊界。

第二種左右邊緣偵測 pseudo code 如下：

輸入：hahaha.TempSobelRotateV，車輛資訊陣列 輸出：車輛資訊陣列 參數： EdgeRateH = 0.7
<pre> findvehicleaccuratewidth2 () begin   mode = 0;   for( k = 0; 車輛資訊[k]為空; k++ )   begin     for( j = 車輛資訊[k]的中點座標x值-車輛資訊[k]寬度*0.3; j &gt;= 車輛資訊[k]的中點座標-車輛資訊[k]寬度*1.5; j-- )     begin       if( mode == 0 )       begin         當j減少，換到上一列，更新垂直陣列，並累計0變1的數量         if( 累計0變1的數量達到車輛資訊[k]寬度*0.6*EdgeRateH )           mode = 1;       end     else if( mode == 1 ) </pre>

<pre> begin     當j減少，換到上一列，更新垂直陣列，並累計1變0的數量     if( 累計1變0的數量達到車輛資訊[k]寬度*0.6*EdgeRateH*0.5 )         紀錄車輛資訊[k]的左下角座標     end end mode = 0; for( j = 車輛資訊[k]的中點座標x值+車輛資訊[k]寬度*0.3; j &gt;= 車輛資訊[k]的中點座標+車輛資訊[k]寬度*1.5; j++ )     begin         if( mode == 0 )             begin                 當j增加，換到下一列，更新垂直陣列，並累計0變1的數量                 if( 累計0變1的數量達到車輛資訊[k]寬度*0.6*EdgeRateH )                     mode = 1;             end         else if( mode == 1 )             begin                 當j減少，換到下一列，更新垂直陣列，並累計1變0的數量                 if( 累計1變0的數量達到車輛資訊[k]寬度*0.6*EdgeRateH*0.5 )                     紀錄車輛資訊[k]的左下角座標             end         end     end end end end </pre>
表 4-8 左右邊緣偵測方法二的 pseudo code

#### 4.1.6 車輛頂部偵測

車輛頂部偵測主要是找車輛頂端的水平邊緣特徵，由於搜尋車輛頂部可能會受到擋風玻璃上的圖示影響，因此本文採由外而內的偵測車輛頂部，方法是選定某個畫面中的水平高度，由上而下找尋車輛頂部的偵測高度，若找不到車頂資訊，則再提升起始水平高度，重新由上而下尋找車頂邊緣，起始水平高度總供提升八次，如果在八次內有找到佔車輛寬度一定比例的水平邊緣特徵，則判斷為車輛頂部。

pseudo code 如下。

輸入：hahaha.TempSobelH，車輛資訊陣列
輸出：車輛資訊陣列
參數：
EdgeRateV = 0.7
HorizontalProjectNum = 15
findvehicleaccuratewidth2 ()
begin

```

mode = 0;
for( k = 0; 車輛資訊[k]為空; k++ )
begin
    flag = 0;
    for(l = 0; l < 8; l++)
    begin
        水平投影陣列初始化
        horizontalproject(k, l);    //得到水平投影陣列 大小為HorizontalProjectNum
        for(i = 0; i < 投影陣列大小; i++)
        begin
            if( 水平投影[i] >= 車輛資訊[k]的車輛寬度 * EdgeRateV )
            begin
                儲存車輛資訊[k]的車輛左右上角座標
                flag = 1;
                break;
            end
            else
                車輛資訊[k]設定為非車輛
            end
            if( flag == 1 )
                break;
        end
    end
end
horizontalproject(k, l)
begin
    距離車輛資訊[k]車輛底部座標上方車輛資訊[k]的車輛寬度*0.1*l的高度為起始高度
    for(j = 0; j < HorizontalProjectNum; j++ )
    begin
        for(i = 車輛資訊[k]左下角座標x值; i < 車輛資訊[k]右下角座標x值; i++)
        begin
            if( 像素[i, j]為水平邊緣特徵 )
                水平投影[j]++;
        end
    end
end
end

```

表 4-9 車輛頂部偵測的 pseudo code

#### 4.1.7 車輛驗證

在更新完所有的車輛資訊後，需對車輛進行驗證，驗證主要是依據車輛的特徵，判斷各個框選範圍是否為車輛。車輛最主要的特徵即為車輛底下會有陰影特徵、左右邊緣會有明顯的垂直邊緣特徵且車輛頂部會有明顯的水平邊緣特徵，由於本文對車輛偵測有進行陰影偵測、左右邊緣偵測和頂部偵測，這些皆是依照車輛特徵進行偵測，非車輛的偵測資訊皆會在上述階段判斷為非車輛，因此本文只需利用簡單的方法對偵測道的物件進行最後確認，並不需要重新將特徵數據化，

帶入複雜的 SVM 進行驗證，或是重新判斷一次所有特徵，而最後簡單的判斷即為判斷物件是否有一條以上的水平直線的水平邊緣特徵，此特徵可能為車子底部的橫桿又或是車輛車牌附近會有很多水平的直線，像是保險桿或是車牌和車燈上的水平邊緣特徵，而且物件寬度相較於車道線寬度，屬於一個合理的範圍，只要符合上述存在一定長度水平邊緣且起物件寬度和車道線寬度之比值在合理的範圍內，則判斷此框選物件為車輛，不符合的皆設為非車輛，由於方法簡單，在這就不將演算法用 pseudo code 表示了，示意圖如 Fig.22，其中白色線條代表框選物件的水平邊緣。



Fig.22 車輛背後水平特徵

## 4.2 連續圖片偵測

在完成偵測流程後，則開始測試多張連續圖片，再執行多張圖片偵測前，須先完成車輛追蹤部分，也就是說，車輛偵測是由兩部分組成，一部分是偵測模式，一部分是追蹤模式，也就是藉由運用這兩個模式交互切換，完成一部影片的連續偵測。

### 4.2.1 車輛追蹤

車輛追蹤的方法是使用時間軸上前後畫面比對的方法來找出偵測畫面對於參考畫面中的目標物在偵測畫面的畫面哪個地方的誤差最小，代表偵測物移動的位置，使用移動向量(Motion Vector)來代表追蹤目標在偵測畫面會移到哪個位置。要追蹤目標必須使用畫面搜尋演算法來搜尋畫面，而畫面搜尋演算法有很多種，像是最精確也是最慢的全域搜尋法、設定局部區域的局部搜尋演算法、或是其他快速搜尋演算法，本文由於畫面是連續產生的，因此車輛如果在畫面中並不會移動到很遠的位置，一定在前面畫面目標物座標的附近位置，因此可以使用一些快速搜尋演算法來追蹤目標物。本文使用的是最知名的三步搜尋演算法(three-step search method)，而畫面比對方式是使用 MSD(Mean Square Error)，三步搜尋法的作法是先搜尋目標物和它周圍正方形的 9 個 patterns，每個 pattern 和目標物上下左右距離三格，一個 pattern 代表一次畫面比對，找出最像的一個 pattern，並縮小搜尋範圍，搜尋最像的 pattern 和它周圍正方形的 9 個 pattern，每個 pattern 和目標物上下左右距離兩格，並再找出最像的一個 pattern，在縮小一次範圍，搜尋最像的 pattern 和它周圍正方形的 9 個 pattern，每個 pattern 和目標物上下左右距離一格，找出最像的 pattern，此位置為目標物移動的位置，如圖 Fig.23。

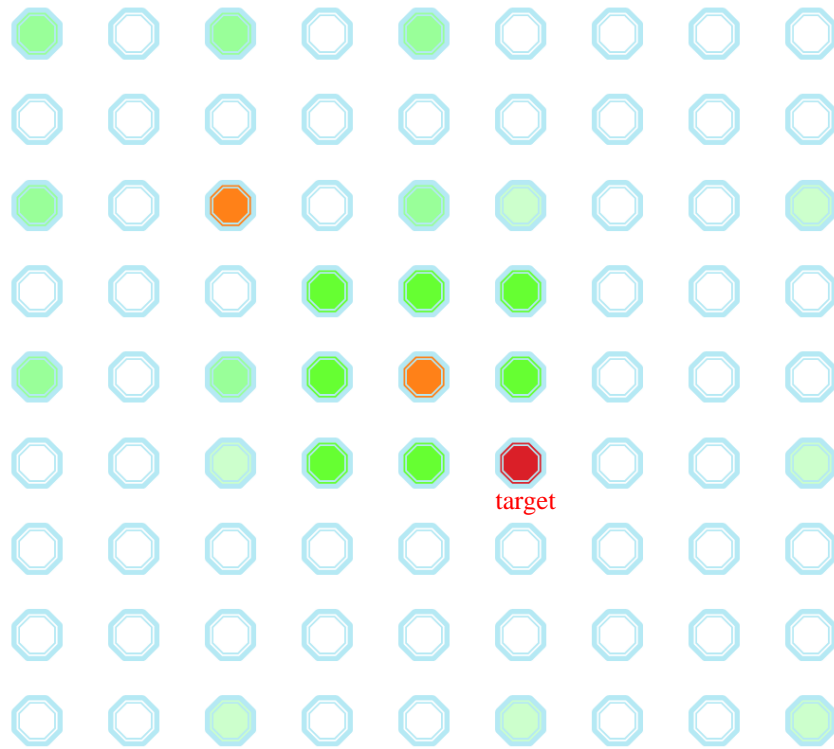


Fig.23 三步搜尋法示意圖

畫面比對的方式有很多種，像是 MAD 或 MSD 等，而本文使用的是 MSD，原因是因為對誤差平方有助於加大誤差值的量，希望可以更精確的追蹤到正確的目標。

#### 4.2.2 三種偵測模式

本文裡一共想出三種不同的執行模式可以選擇，純偵測模式、追蹤為主模式和偵測為主模式。純偵測模式就是對連續圖片每張都用偵測方式偵測車輛，不管有沒有找到。追蹤為主模式就是以畫面追蹤為主要方法偵測車輛，偵測只是找出畫面追蹤需要的目標給追蹤模式使用。偵測為主模式就是以畫面偵測為主要方法偵測車輛，當偵測模式偵測不到車輛時，使用追蹤模式補追蹤同一張畫面，這種模式在偵測模式有一定偵測能力的情況下可以達到很好的偵測效果，當然缺點是計算時間比較久，因為可能每一張畫面需要都需要做偵測車輛和追蹤車輛。

在下面使用流程圖來表示三種不同執行方法：

1. 純偵測模式：由於方法僅是每張圖都使用偵測模式偵測車輛，因此就不使用流程圖來表示。
2. 追蹤為主模式：如圖 Fig.23

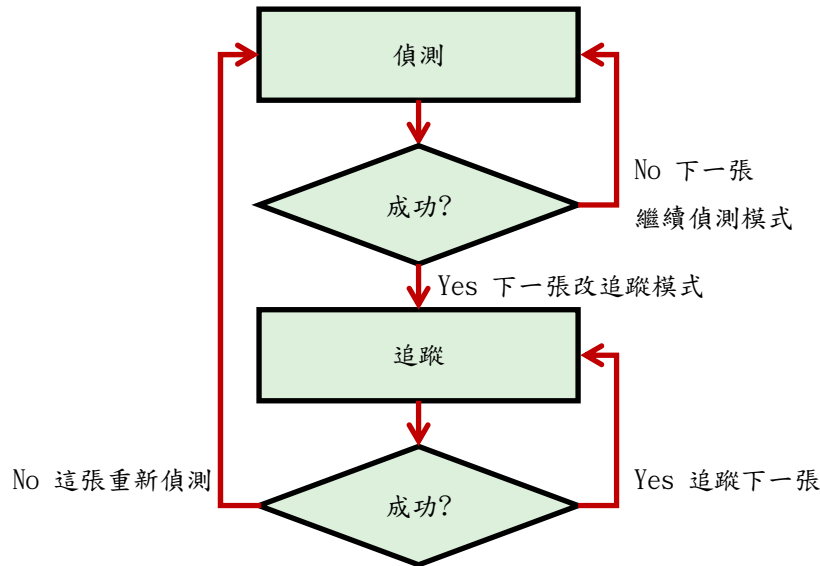


Fig. 24 追蹤為主模式流程圖

3. 偵測為主模式：如圖 Fig.24

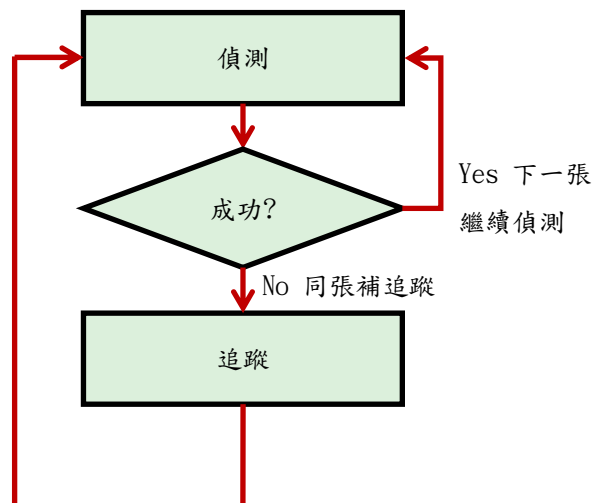


Fig. 25 偵測為主模式流程圖

由於本文三種方法整合在一起，因此在這裡使用有限狀態機來表示整合後的模

式切換流程，如圖 Fig.26。

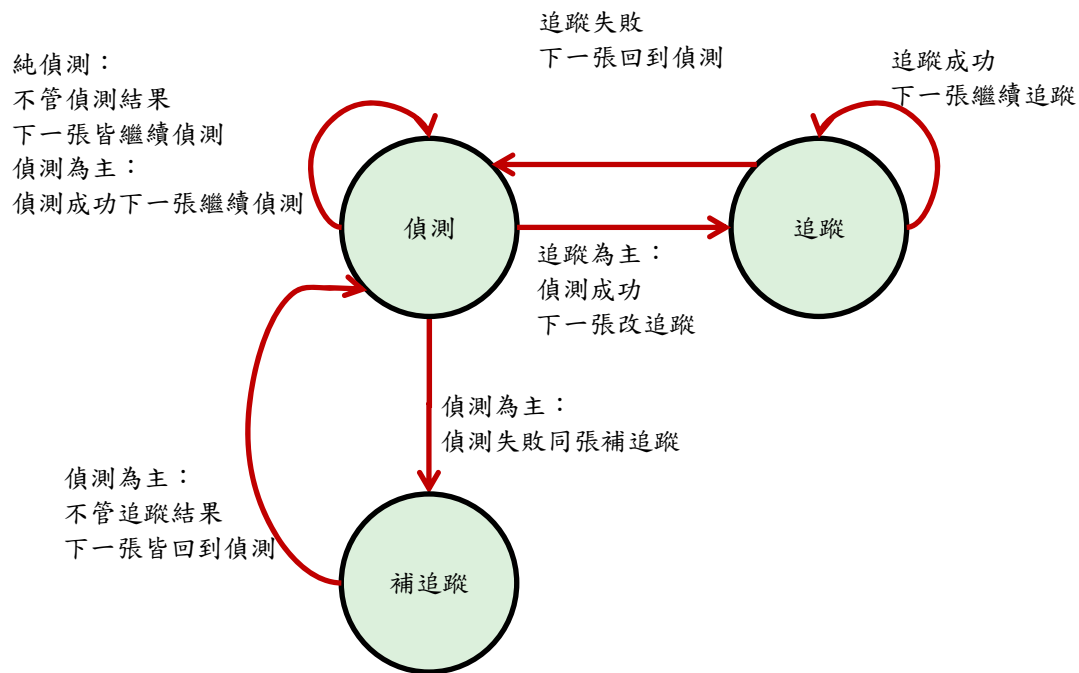


Fig. 26 三種模式整合流程圖

### 4.2.3 車輛偵測資訊

雖然本文中部份處理方式經後面方法加速後在 i7 - 2600 3.4Ghz 上直接執行可以達到每秒 25 張以上(平均)，但是為了除錯方便以及顯示結果方便，本文將車輛偵測與追蹤資訊寫入文字檔中，在連續圖片偵測完後，利用載入圖片的方式，和文字檔中的偵測與追蹤資訊框出車輛，並顯示其他輔助資訊，文字檔中儲存的資訊如圖 Fig.26。



模式	圖片名稱								
0	00000002.jpg								
框中的四個頂點座標									
左下X	左下Y	左上X	左上Y	右下X	右下Y	右LX	右LY		
355	274	355	220	415	274	415	220		
追蹤圖片名稱		追蹤圖片編號		交點座標		斜率		與原點垂直距離	
				X	Y	M左	M右	B左	B右
00000001.jpg		0		355	220	-1.013X	0.794X	625.46X	-97.02X
找到左線道		找到右線道		左線道交點		右線道交點		兩線道交點	
				Y=450	X=0	Y=450	X=800	X	Y
1		1		173	450	689	450	401	222

Fig.27 文字檔中儲存每行資訊

### 4.3 前車距離、距離標線及車輛偏移警告

在可以執行執行連續圖片偵測與追蹤後，接著就是偵測結果的輔助資訊，輔助資訊有前車距離，距離標線及車輛偏移警告。前車距離就是計算偵測出的前車與攝影機的距離。距離標線就是每隔一段距離顯示一條橫向的標線並標出距離。車輛偏移警告就是當車輛切換車道或是偏離道路正中央時，顯示警告訊息，畫面如圖 Fig.28。



Fig.28 車輛偵測輔助資訊畫面

### 4.3.1 回歸曲線

照正確做法，要求出車輛距離及距離標線需使用攝影機裝置焦距等參數[8]推算現實中車輛位置的距離，但本文測試了多個影片，皆是不同的行車紀錄器，也沒有不同影片使用攝影機裝置焦距等參數，因此並沒有使用這種方式標出距離資訊。由於由畫面觀察出，車輛陰影寬度的大小（約等於車寬）與現實中距離遠近呈反比，且陰影在畫面中放大倍數與陰影在畫面距離底部的高度呈指數的關係，因此本文使用統計的方式利用此關係

利用 Excel 找出回歸曲線，利用此回歸曲線推算畫面中的實際距離。步驟如下：

1. 統計畫面資訊
2. 利用 Excel 算出回歸曲線
3. 利用此曲線推估畫面距離關係

首先要統計畫面資訊，統計的資訊有：

1. 陰影最下方的座標 y 值
2. 以像素為單位測量出陰影寬度大小
3. 估算此陰影在現實世界中的長度（約等於車輛寬度）

有了這三樣數據即可推出下面兩個參數：

1. 陰影最下方距離畫面底部的高度(像素)
2. 陰影放大倍數，公式為：
3. 陰影放大倍數 =  $\frac{\text{估算此陰影在現實世界中的長度}}{\text{以 pixel 為單位測量出陰影寬度大小}} = \frac{cm}{pxiel}$  (10)

使用上述方法統計一部影片中挑選出的 20 張不同距離的車輛畫面，完成步驟一。

第二步驟把所有畫面數據，利用 Excel 的 LOGEST() 這個函式算出曲線  $y = b \times m^x$  的係數  $b$  和  $m$ ，其中橫座標  $x$  為陰影最下方距離畫面底部的高度，縱座標  $y$  為陰影放大倍數，算出  $b$  和  $m$  後即可得到下面的回歸曲線，如圖 Fig.29。

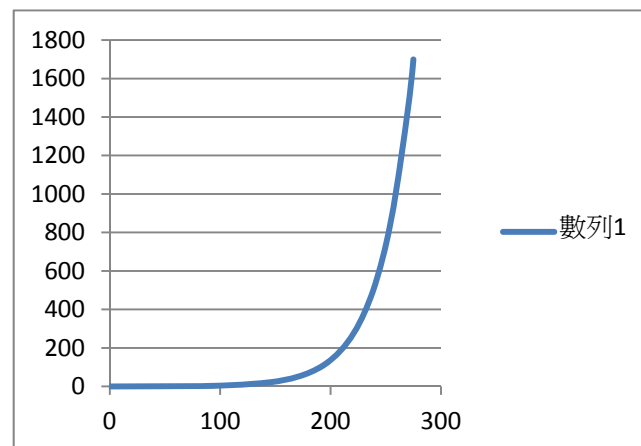


Fig.29 此為某影片統計出的回歸曲線，其中  $m = 1.034118$ ， $b = 0.167287$ ， $x$  軸單位為距離底部的高度像素值， $y$  軸單位為(公分/像素)

第三步驟即利用此關係回推畫面中現實世界的距離，由圖 Fig.29 可看出，當距離底部多少個像素時，此像素在現實世界中佔多少公分，也就是此指數曲線下面由低到高累計的面積，即為該高度換算現實世界中的距離。

#### 4.3.2 前車距離與距離標線

有了上面的距離關係，很簡單就可以算出前車距離，即是把前車框選出的範

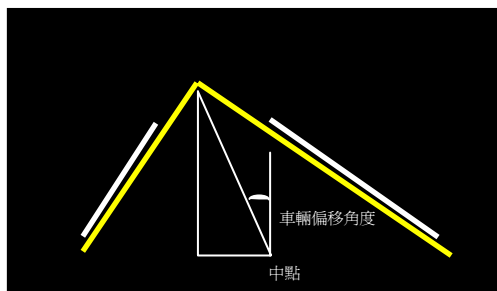
圍距離底部的最小高度代到上面回歸曲線中算面積，即可得到實際距離，結果如圖 Fig.28。

本文裡標出幾段距離，即是用回歸曲線由低往高慢慢累積曲線下的面積，當面積為所要的距離時，標出那個畫面高度的橫線，並標上距離數字，結果如圖 Fig.28。

#### 4.3.4 車輛偏移警告

利用地平線的消失點與左右道路線的中點，可以算出車輛偏移角度，如圖 Fig.30(a).當車輛向左右偏移超過 $15^{\circ}$ ，則顯示靠近左車道線或靠近右車道線，如圖 Fig.30(b).

(a)



(b)



Fig.30 (a).車輛偏移角度示意圖, (b). 當車輛向左右偏移超過 $15^{\circ}$ ，顯示車輛偏移警示

# 第五章 執行加速與 GUI 介面設計

## 5.1 執行加速

本文由於執行速度在未加速前十分的慢，就算用 i7-2600 3.4Ghz 跑數據，使用最基本的 C#指標處理方式，偵測一張圖要花上 242ms，追蹤要 17ms，也就是偵測一張圖最慢(偵測加追蹤)要 259ms，更不用說速度較差的 CPU(本文程式撰寫使用的 CPU - Intel Core Duad Q9500 2.83Ghz)，偵測一張圖要花上 400ms，追蹤要 29.33ms，也就是偵測一張圖最慢(偵測加追蹤)要 429.33ms，要達到即時處理是不可能的，當然在實際處理時不會這麼慢，這是最壞打算，但是光跑一部影片就要花很久時間，雖然有較快速的處理方法，也就是使用 OpenCL 平行的做法加速前處理，用 i7-2600 3.4Ghz 偵測一張圖只要花 45ms，追蹤要 17ms，也就是偵測一張圖最慢(偵測加追蹤)要 62ms，每秒處理約 16 張，但是還是不能夠達到即時，因此本論文花一大部份時間在進行加速及程式碼優化，目標希望程式能達到更高的處理效果。本文在實際研究過程中並不是按照後面小章節順序進行加速及程式碼優化，這裡僅是把對程式進行加速及程式碼優化的幾個部分分章節介紹而已。

### 5.1.1 C# TPL 加速

TPL 其實算是另外一種加速方法，它可將原來 C#的指標運算，經過簡單的程式修改，即可將單執行緒的程式改成由電腦管控執行緒的多執行緒程式，修改方式如第 2.3.11 章節那樣。在使用 TPL 加速車輛偵測的前置處理（包括彩色轉灰階和產生邊緣資訊）後，偵測一張圖可以再快 20ms。

值得一提的是，多執行緒使用的最大效益是執行緒等於核心數的時候，此時加速最快，也就是執行緒太少不夠快，太多反而會因執行緒的切換而導致降低處

理速度，而由於 TPL 是採自動管理執行緒，如果圖形是二維的畫面，在程式上就必須用兩個迴圈的方式處理，如果將兩個 for 迴圈皆改為 Parallel.For 的形式，實際在執行時似乎會先在外部迴圈分配足夠執行緒，而當程式執行到內部迴圈時，會再配置其他的執行緒，因此執行緒會過多，導致實際上的加速達不到上述的 20ms，可能只有 10ms 而已，因此正確作法只需將外部迴圈改為 Parallel.For 的形式即可，分配的執行緒會剛剛好，實際測試後的效果最高。

### 5.1.2 OPENCL 優化

本文 OpenCL 在未優化前速度並不夠快，一般來說經由常識判斷會以為速度的瓶頸是在主機 CPU 以及顯示卡 GPU 之間的傳輸，實際上並不是，本文使用 NVIDIA 的 Visual profile 對 OpenCL 的運作時間進行時序分析：結果如圖 Fig.31。

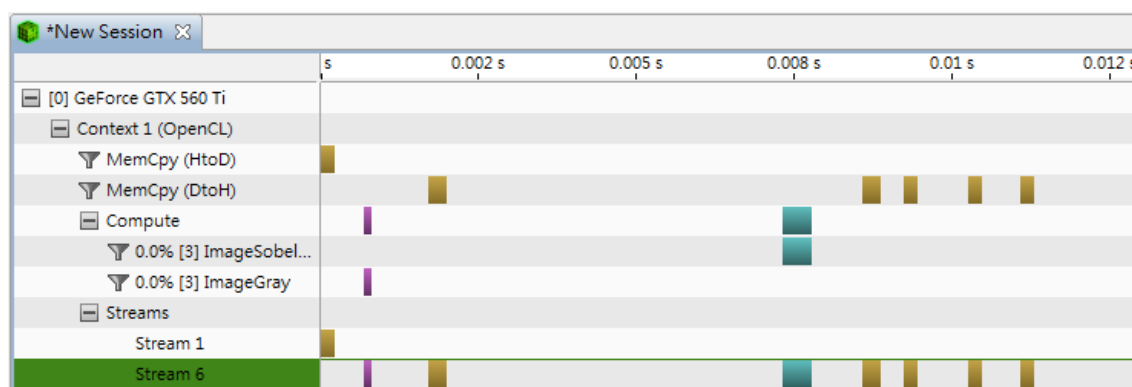


Fig.31 OpenCL 為優化前的時序分析圖

MemCpy 代表的是資料複製，HtoD(Host to Device)也就是從主機傳送資料給設備的時間，DtoH(Device to Host)則是設備回傳資料給主機的時間，Compute 就是核心計算時間，由上圖可以發現其實傳輸和核心計算時間佔整體時間並不是很多，時間的瓶頸在傳輸和計算中間的空洞，也就是 OpenCL 指令執行完有大部分的時間再處理其他 C#的指令。上述程式碼的主要動作為：

1. 執行灰階化的核心(kernel)
2. 把灰階圖傳回主機

3. 執行水平濾波及邊緣檢測的核心
4. 把剩餘所有處理得到的圖傳回主機(一共四張)

初步認為程式瓶頸是 OpenCL 呼叫指令時會有一段等待指令執行的時間，而由於水平濾波及邊緣檢測的核心在計算時並不需要使用主機的資料，也就是它可以拿灰階化的核心處理完的輸出直接當作輸入進行計算，因此可以把執行順序對調，改成下面那樣：

5. 執行灰階化的核心(kernel)
6. 執行水平濾波及邊緣檢測的核心
7. 把所有處理得到的圖一次傳回主機(一共五張)

經過修改後，OpenCL 的時序分析圖變為圖 Fig.32。

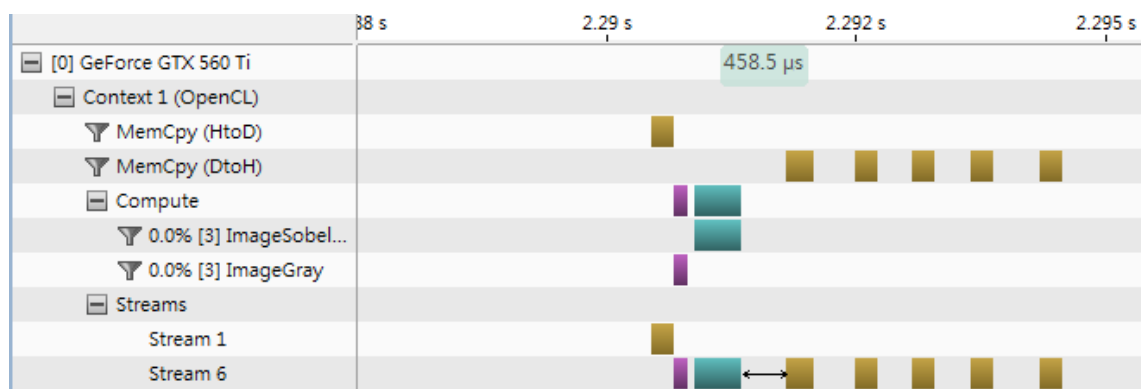


Fig.32 調整順序後的 OpenCL 時序分析圖

可以發現回傳圖片間的空檔間隔有縮小，這和一開始的作法已經加速一倍以上了。由於傳輸時間實際上是不可能縮短，核心運算時間除非核心程式有優化或是處理方法有更好的，不然也不可能再縮小了，因此將目標轉移回主機的指令上。接下來仔細的分析是否還有加速的方法，發現回傳圖片的空檔除了要等待主機回應外，其他時間是在做鎖定記憶體和解鎖的動作，也就是 GDI+指令 LockBits() 和 UnLockBits()，這部分有關程式碼優化的地方留在 8.4 再說。

### 5.1.3 多執行緒及多執行緒載入圖片

本文在圖片旋轉的動作是呼叫 C# 內建函數旋轉動作，一共要旋轉兩張圖。因此如果多執行緒能加速處理速度，對於內建指令應該也有加速效果，因此將旋轉的動作，配置另外兩條多執行緒，分別對水平濾波的灰度圖及垂直邊緣圖做旋轉。另外，經由上網查，使用多執行緒的時候可以使用 `sleep()` 讓某執行緒睡覺，也就是不監視執行此執行緒，如果此執行緒在使用一些較慢的設備，例如硬碟，也就是讀取圖片，可以讓執行緒睡覺，把時間讓給其他運算執行，讓這條睡覺的執行緒默默地去載入圖片。這裡可以使用在連續圖片處理時，使用另外一條執行緒預先載入下一張圖片到某個暫存的變數，並讓此執行緒睡覺一段時間，並在此執行緒睡覺的時候，執行偵測程式碼或是追蹤程式碼，而當圖片載入記憶體後，便結束此條執行緒，在下一張圖片偵測前，把暫存的變數 Assign 給要處理圖片使用的變數，並同樣的再用另外一條執行緒，再載入下一張圖片，這種做法可以使執行速度大約快 1fps。

### 5.1.4 程式碼優化

本文對於程式碼的優化主要在兩個地方，都跟 GDI+ 有關係，因為一般指令執行時間都差不多快，但是使用到 GDI+ 的指令時，不管是查圖片資訊，或是前面提到的 `LockBits()` 或是 `UnLockBits` 都需要較久的時間，同樣一條指令，GDI+ 的函數可能要普通指令的 10 倍以上的時間，呼叫很多次時時間會增加很多，更別說把這種指令放在迴圈中，可能就要執行上萬次了，增加的時間非常可觀。

其實本文對程式碼優化的方法相當簡單，但是要找到此問題相當不容易，尤其對剛入門或是一般的程式設計師，往往都會忽略這個東西導致程式效能變慢。本文優化的第一個地方即是迴圈內的 GDI+ 指令，由於本文對於圖像是使用 `Bitmap` 作為儲存的東西，而進行圖像處理時往往會因為方便或是為了要程式自動化，要查圖像的寬度或高度時，直接在迴圈內就使用 `Bitmap.Width` 等指令就



查了，這就是問題所在，這種查法就是呼叫 GDI+ 的指令，所以只要把這些跟 Bitmap 有關的呼叫，移出迴圈外，速度可以達到意想不到的加速，以本文的例子，把處理過程中迴圈內的 GDI+ 函數都移出迴圈外，使用變數代替，在 CPU - Intel Core Duad Q9500 2.8Ghz 的主機上使用 C# 指標的方法處理原先需要約 400ms，現在只需要 120ms。

而第二個地方是前面提到的 LockBit() 和 UnLockBits()，這兩個指令像前面所說是鎖定記憶體和對記憶體解鎖，速度約慢一般指令約 100 倍，一般指令在較差的主機可能只要 5ns，它需要 500ns。另外在進行圖像處理時其實並不需要一直鎖定記憶體和解鎖記憶體，因此本文將各個程式片段的這兩個指令，統一移到偵測前和偵測後，也就是原先偵測流程是：

1. LockBit()
2. 前處理
3. UnLockBits()
4. LockBit()
5. 偵測程式片段
6. UnLockBits()
7. LockBit()
8. 偵測程式片段
9. UnLockBits()
10. ...

現在偵測流程改為：

1. LockBit()
2. 前處理
3. 偵測程式
4. UnLockBits()

較原先方式簡潔，由於這兩個指令全部加起來呼叫 40 餘次，因此做完這個動

作後偵測模式快了約 20ms。另外在追蹤模式也有這個問題，也就是追蹤時間大約 2~3ms 就完成了。

另外值得一提的是，前面 OpenCL 的這個部分，為了要做統一鎖定記憶體和對記憶體解鎖這樣修改，本文把核心輸入跟輸出從原先的 CL.image 改為 Bitmap 的 ByteArray，並修改核心程式碼，核心程式碼附在附錄二。改成這樣還有另一個好處，在於原先 OpenCL 核心讀取資料是一次 4ByteRGBA，其中 A 是空的，且是使用 read\_imageui() 方式讀取像素值，雖然在 OpenCL 對於圖像用這種方式處理有進行加速，但是改為輸入是 ByteArray 後一次是讀取 3Byte 的 BGR 像素值，但是是連續的方式讀入，所以這種方式處理的速度還是比之前 CL.image 當作輸入的處理方式快約一倍以上，結果如圖 Fig.33。

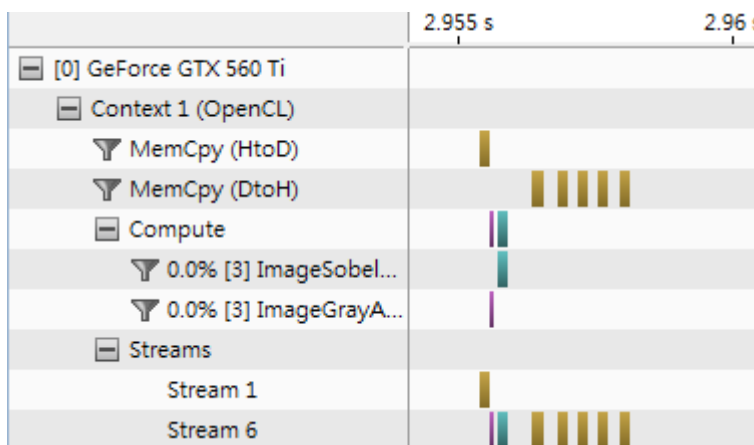


Fig.33 統一鎖定記憶體和對記憶體解鎖修改後的時序分析圖

修改後比原先一開始作法整整快了四倍左右，應該能再加速的空間已經不多了，最終數據結果將在第 10 章介紹。

## 5.2 GUI 介面設計

在做完主要程式設計及優化後，本文亦對程式 GUI 介面進行設計及美化，

可以顯示邊緣檢測的圖，也將調整介面色彩的程式碼集中設定，方便 GUI 介面修改畫面，另外也把參數製成設定檔，可以對不同影片訂製一套不同的參數。

## 5.2.1 介面設計

在這裡介紹本文程式 GUI 介面的幾個設計：

1. GUI 程式介面可以最大最小化，如圖 Fig.34



Fig.34 GUI 主要畫面最大化和最小化

2. 可活動式和可切換進階參數設定面板，進階參數設定面板可在畫面大小內移動，如圖 Fig.35



Fig.35 可活動式和可切換進階參數設定面板

## 5.2.2 功能介紹

所有功能如下：

主要功能		單張圖片偵測及時間計算
------	--	-------------

		連續圖片偵測與追蹤及時間計算
		載入車輛資訊
分解動作 功能		單張圖片偵測
		載入圖片
		前處理
		道路線偵測
		車輛
		平均亮度偵測計算
		道路間平均亮度計算
		道路間平均亮度加權計算
		版本
進階設定 功能		主要設定
		方法設定
		參數設定
		直線與曲線設定
撥放功能		第一頁
		前一頁
		播放
		暫停
		下一頁
		最後一頁
		前處理設定
		邊緣檢測

### 5.2.3 程式碼參數化

本文將所有相關的參數整理在一起，使用全域變數儲存，因此可在程式碼前面的地方對參數進行調整，影片參數也一樣，避免除錯及修改上的麻煩，如圖 Fig.36。

```
//////////主要設定參數//////////
public static int sobelthreshold = 50;          //邊緣檢測臨界值
public static int ShadowMaxluminance = 40;      //車輛陰影臨界值

public static int VehicleLampFindHeight = 330;  //車燈最高搜尋y值
public static double FindTopStartRatio = 0.9;   //起始搜尋頂部的起始高度(佔車寬的比例)
public static double EdgeRateH = 0.7;          //左右邊緣找到邊緣的比率
public static double EdgeRateV = 0.4;          //頂部邊緣找到邊緣的比率
public static double EdgeVerityRateV = 0.7;     //驗證用的比率
public static int SearchBottom = 5;             //起始搜尋底部
public static int Minluminance = 4;             //道路和線最小亮度差
public static int MinShadowluminance = 4;       //陰影與道路最小亮度差
public static int MinLineLength = 20;          //道路線最長長度
public static int MaxHorizonY = 320;           //最大地平線交點Y軸
public static int MinHorizonY = 190;           //最小地平線交點Y軸
public static double MaxAvgError = 5;          //道路線單點與線垂直截距
//陰影threshold參數
public static double AutoThresholdont1 = 20, AutoThresholdont2 = 14, AutoThresholdont3 = 0.05, AutoThresholdont4 = 0.08;
public static double curvem = 1.034118, curveb = 0.167287; //垂直加權用的回歸曲線  $y = curveb + curvem \wedge x$ 
public static double m1default = -0.98968, m2default = 0.79625;
public static double b1default = 618.5208, b2default = -97.7465;
public static int MaxAvgMSD = 800;             //MSD臨界值
public static double TrackErrorStartRatioY = 0.35; //追蹤只比對車中央
public static double TrackErrorEndRatioY = 1;    //追蹤只比對車中央
public static int FrameWidth = 800, FrameHeight = 450;
//////////標記色彩//////////
public static Pen PenDefaultLane = new Pen(Color.FromArgb(170, 130, 90, 190), 2); //紫色 預設道路顏色
public static Pen PenFindLane = new Pen(Color.FromArgb(170, 240, 240, 130), 2); //淡黃色 找到道路顏色
public static Pen PenBeforeLane = new Pen(Color.FromArgb(170, 230, 180, 20), 2); //橘色 前一張道路顏色
public static Pen PenFrame = new Pen(Color.FromArgb(180, 255, 0, 0), 4); //紅色 框體顏色
public static Pen PenTargetLine = new Pen(Color.FromArgb(125, 0, 125, 0), 1); //綠色 目標座標
public static Pen PenDistHorizonLine = new Pen(Color.FromArgb(125, 255, 0, 0), 1); //紅色 距離標線
public static Pen PenVehicleDist = new Pen(Color.FromArgb(125, 255, 0, 0), 1); //紅色 車輛距離
public static Color DistHorizonLineFont = Color.FromArgb(180, 230, 230, 70); //黃色 距離標線字
public static Color VehicleDistFont = Color.FromArgb(180, 230, 230, 70); //黃色 車輛距離字
public static Color ColorFacePlate = Color.FromArgb(255, 210, 240, 210); //淡綠色 底色
public static Color ColorTargetFacePlate = Color.FromArgb(255, 190, 240, 190); //深綠色 目標面板
public static Color ColorToolStrip = Color.FromArgb(255, 180, 240, 180); //更淡綠色 工具列
public static Color ColorAdvancePanel = Color.FromArgb(255, 220, 240, 220); //更亮綠色 進階設定
public static Color ColorVersionFacePlate = Color.FromArgb(255, 220, 240, 220); //版本TextBox背景色彩
//////////
```

Fig.36 參數化程式碼內容

上面的部分參數的設定的說明如下：

- VehicleLampFindHeight (預設值=300)：

此參數是在近車偵測時，判斷車尾燈位置之 y 座標最上方邊界（圖片最上方為 0）。

- FindTopStartRatio (預設值=0.9)：

判斷車頂位置時，依目前偵測出的陰影寬度乘上此參數，作為第一次尋找車頂位置的起始位置，從此位置由上往下尋找，若未找到車頂，下一次重新尋找車頂時，會斟酌往上再調整車頂搜尋的y座標起始值。。

- EdgeRateH (預設值=0.7)、EdgeRateV(預設值=0.4)：和(EdgeVerifyV)(預設值=0.7)：

在判別車輛的左右邊緣（採用EdgeRateV）或是車頂邊緣（採用EdgeRateH）時，會根據目前偵測出的陰影寬度乘上此參數，作為判斷是否找到邊緣之依據。（採用EdgeRateH）參數EdgeVerifyV是做Hypothesis Verification時，驗證偵測到的物件是否為車輛時，根據此參數乘上目前偵測出的物件寬度，判斷物件內是否有超過此值之水平邊緣。

- SearchBottom(預設值=5)：

此參數代表畫面最底端的y座標值。y座標比此值還小，代表是車輛前方的引擎蓋。此參數的目的是為了要從引擎蓋上方開始偵測。

- Minluminance(預設值=4)：

車道線偵測時，車道線位置和非車道線位置（即道路）之亮度差值門檻，亮度差超過此值，才代表車道線存在。如果道路線的邊緣比較模糊的時候，此參數可設小一點；如果地面上有很多相鄰像素亮度差很大的時候，可以考慮調高此參數。

- MinShadowluminance (預設值=4)：

車輛底部陰影偵測時，陰影位置和非陰影位置（即道路）之亮度差值門檻。當陰影邊緣比較模糊時，這個參數要調小。

- MinLineLength的(預設值=20)：

此參數代表找到的車道線的最大可能寬度。此參數的目的是為了避免找尋道路線時把近距離的斑馬線（寬度通常超過此值）也當作是道路的特徵。

- MaxHorizonY (預設值=320) 和 MinHorizonY(預設值=190)：

地平線（左、右車道線交點）的y座標範圍。此參數的目的是避免道路線越找越水平。

- MaxAvgError (預設值=5)：

此參數是內插求出的車道線邊緣直線和所有內插點的距離最大值。此參數設大一點，會允許找出比較彎曲的道路線。

- AutoThresholdcnt1(預設值=20)、AutoThresholdcnt2(預設值=14)、



AutoThresholdcnt3(預設值=0.05)、AutoThresholdcnt4(預設值=0.08):

論文中第三章所提出的四種自動陰影臨界值方法所需之參數。

- curvem(預設值=1.034118)和curveb(預設值=0.167287):

估算各種行車記錄器中畫面y座標和車輛距離之關係時，代入 $d_i = \text{curveb} * (\text{curvem})^{y_i}$ ，算出每個 $y_i$ 座標所對應的 $d_i$ 值，而y座標為 $y_i$ 所對應的車輛距離為 $d = d_i + d_{i-1} + \dots + d_0$ 。

- m1default, m2default, b1default, b2default :

在執行第一張畫面時，若未找到車道線，則根據此四個參數決定起始的左、右車道線。往後畫面若未找到車道線，則以最近找到的車道線代入。

- TrackErrorStartRatioY(預設值=0.35)、TrackErrorEndRatioY(預設值=1)：

車輛追蹤時，為了避免車輛左上角和右上角可能是畫面背景（因大部分車輛之左上角和右上角有弧度）造成比對錯誤，因此比對時所考慮的物件的區域，會從車輛物件高度的TrackErrorStartRatioY開始到車輛高度的TrackErrorEndRatioY結束。

- MaxAvgMSD(預設值=800)：

車輛追蹤時，以三步搜尋法做目前畫面和前張畫面內車輛物件比對，計算出的mean square difference (MSD)之最大值。超過此值，代表未找到追蹤對象。此參數若設大一點的話，可以放寬追蹤時畫面比對容許的誤差和大小

- FrameWidth (預設值=800)、FrameHeight(預設值=450)：

所有的不同解析度的影片，都先轉成此參數所設的固定畫面解析度，採進行後續的所有運算和處理。

## 5.2.4 影片參數設定

本文對不同的測試影片有個別製作設定檔，因此在測試不同影片時只需要選擇所要測試的影片即可，可以不必一項參數一項參數的設定，如圖 Fig.37。參數檔內設定的參數為 Fig.36 中主要設定參數。





Fig.37 參數設定檔位置

## 第六章 實驗數據及分析

本文除了對自己的不同方法實驗數據進行分析比較，也跟不同影片的不同偵測片段進行分析比較。在這裡先介紹偵測率及錯誤警告率正確的計算方法，公式如下：

$$\begin{aligned}\text{Detection Rate} &= \frac{T_p}{T_p + F_n} \\ \text{false alarm Ratio} &= \frac{F_p}{T_p + F_p}\end{aligned}\tag{11}$$

其中

$T_p$  正確偵測（前方有車或無車，且都能正確判別）的車輛數

$F_p$  錯誤偵測（前方有車，但偵測不到）的車輛數

$F_n$  遺漏偵測（前方無車，但有偵測到）的車輛數

如下表：

偵測率判斷	前方有車	前方無車
偵測有車	$T_p$	$F_n$
偵測無車	$F_p$	$T_n$

表 6-1 偵測率判斷方式表格

也就是說會有偵測率和錯誤警示率兩個數據，偵測率也就是在範圍內辨識出車輛的機率，而錯誤警示率即是非車輛正確判斷成車子的機率，也就是假設行車紀錄器有錯誤警示的功能，也就是當車輛發生危險時，程式會發出嗶嗶嗶的聲音，所以當錯誤警示率太高，車子會時常發出亂叫的聲音。

本文測試主機主要為 i7-2600 3.4Ghz。由於在此主機上執行較快，因此測試數據時皆是在這台上跑的。另外還有附程式開發使用的電腦 Intel Core Quad Q9500 2.8Ghz 地單張測試時間，作為比較對象，當然正確率並不會因為主機速

度而有所影響。另外執行平台為 .Net Framework 4.0，程式為 32bit 應用程式，作業系統為 Win7 64bit。

## 6.1 單張偵測時間

本文時間測量方式是使用 C# 的 Stopwatch 物件測量，精準度較高，測試結果如圖 Fig.38

單張偵測	Core Quad Q9500	i7 - 2600
Scan0	67	35
Scan0 + 多執行緒	54.7	27.7
OPENCV	103.4	56.3
OPENCL	59.4	31.4
OPENCL + 多執行緒	54.9	29.32

**Fig.38 單張測試時間，分別在 Intel Core Quad Q9500 2.8Ghz 和 i7-2600 3.4Ghz 測試時間，時間單位為 ms(毫秒)**

實際執行時間可看出 CPU 時脈越高，速度越快，圖 Fig.38 的各欄差別為前處理使用的方法，其他偵測計算皆使用相同方式完成。Scan0 也就是純指標運算，而 Scan0+多執行緒是混合使用 TPL 加速及前面所敘的多執行緒加速方法，而 OpenCL+多執行緒就是混合使用 OpenCL 及前面所敘的多執行緒加速方法，可以發現，最快執行的方法為 Scan0+多執行緒，其次是 OpenCL+多執行緒，可以發現純軟體在多執行緒加速及優化後，在處理的圖不夠大時，速度還是快於使用 GPU 運算的，而如果多執行緒和 OpenCL 皆是使用 CPU 運算，其實多執行緒和 OpenCL 速度是差不多的。

另外比較有趣的是 OpenCV 的部分，在未進行優化前，速度順序是 OpenCL>OpenCV>Scan0，但是經過前述方法加速及優化完，實際處理速度竟比

不過純指標運算，所以 OpenCV 實際上對進階程式設計師來說，由此看來似乎是使用上比較簡單而已，但是總執行速度並不會比較快，原因在於它必須先轉 Bitmap 型態為 Image<,>格式，運算完再轉回 Bitmap，這兩個動作影響 OpenCV 整體執行時間，但是並不代表 OpenCV 運算較慢。

## 6.2 同影片不同偵測方法連續 1000 張運算結果

本文有 2 種左右邊緣偵測方法，3 種陰影偵測方法可供選擇，但是由於要全部數據都跑出來組合形式太多，因此本文按照下面設定跑出一部份數據進行比較。

主機為 i7-2600 3.4Ghz，使用第三種陰影方法，由於純 CPU 在使用多執行緒後目前是最快的，因此使用的純軟體的方式，其餘設定如圖 Fig.39。

影片	編號	模式	左右邊緣方法	偵測率(%)	錯誤警示率(%)	fps
DOD F500 HD_超誇張	1	偵測為主	1	100	0	34
	2	偵測為主	2	100	0	39
	3	追蹤為主	1	100	0	49
	4	追蹤為主	2	100	0	46
	5	純偵測	1	100	0	40
	6	純偵測	2	100	0	40
DAS_test	1	偵測為主	1	100	4.8	34
	2	偵測為主	2	100	0.9	36
	3	追蹤為主	1	100	0.1	48
	4	追蹤為主	2	100	0.8	51
	5	純偵測	1	100	1.5	38
	6	純偵測	2	99	1.3	37

Fig.39 不同設定下的影片執行結果

先從時間來看，追蹤為主的模式執行時間由於絕大部分時間在做追蹤，因此

影片處理速度非常快，原則上可以到 46fps 以上，甚至 51fps，而使用偵測為主的模式，在此設定底下，最慢也有 34fps，也就是使用最慢的處理模式可以達到每秒 34fps 以上，也就是即時的速度。另外在偵測率方面，皆有 99% 偵測率以上，但是由於這兩段連續圖片是比較簡單的車輛狀況，因此偵測為主的模式的偵測率並不會比追蹤為主的偵測率還來的高，實際上跑完整部影片，偵測為主的模式的偵測率一定會比追蹤為主的模式的偵測率還來的高(因為追蹤為主的模式如果偵測錯誤會一直追蹤錯誤目標，導致偵測率大幅下降，但如果在錯誤警示率較低的狀況，其實追蹤為主的模式是最好的)。

而兩種左右邊緣方法，其實是效果差不多的，好壞的差別在第五章的描述，在這裡就不再描述。

### **6.3 不同影片連續 1000 張運算結果**

主機為 i7 - 2600 3.4Ghz，使用純軟體多執行緒的方法，自動陰影使用方法三，左右邊緣偵測使用方法二，設定參數是使用各自影片調出來的設定檔設定的，而影片是使用 youtube 上的不同行車紀錄器影片片段，如圖 Fig.40。

編號	影片	fps	天氣	畫格數	場景	特殊狀況 1	特殊狀況 2	特殊狀況 3	偵測率(%)	錯誤警示率(%)
1	DAS test	37	晴天	1280	高速公路	前車切換車道	車前陰影干擾		98.2	0
2				474	高速公路隧道	亮度大幅變化			99.2	0
3				391	高速公路	高架橋干擾			89.5	0
4				2942	高速公路	連續高架橋干擾	前車或本車切換車道	轉彎出交流道	94.8	0.4
5	DOD F500 HD 超誇張	34	陰天	2110	鄉村道路	建築物陰影干擾	轉彎	近距離偵測	99.4	0.4
6				1704	鄉村道路	過高架橋			97.1	0
7				1173	國道	遠距離車輛偵測			93.4	0.5
8				2050	國道	下坡遠距離車輛	本車切換車道	近距離偵測	99.6	0
9				791	鄉村道路	橋下黑暗道路			85.0	4.5
10	PAPAGO! P1 第二批 V2	35	陰天	1513	隧道內外	鏡前反光	地面較髒	出隧道口	95.0	0
11	進階版 早上二高隧道			1264	高速公路	鏡前反光	大型車		95.3	0
12	行車記錄器、這也太清	35	陰天	1554	市區	本車快速切換車道			88.0	4.1
13	楚了吧 (記得選1080p喔)			3849	市區	本車快速切換車道	等紅燈		99.3	0.1
14				1384	市區	路口等紅燈			X	X
15	清晨陰天F500HD試拍 - YouTube	31	陰天	1456	郊外	本車轉彎	長距離前方無車		97.2	0
16				3347	郊外	蜿蜒道路	長距離前方無車		1	0
17				4380	郊外	蜿蜒道路	跟車		96.1	0
18				2717	郊外	蜿蜒道路	住宅區		1	0
19				1952	郊外	住宅區	多路人干擾		94.0	1.9

Fig.40 不同影片各段數據結果

以上是各種不同影片的不同段，可以觀察到大致上的片段偵測率皆有在 90% 以上，只有少數片段比較差。上圖編號 9，是因為陰影臨界值的關係，由於一部影片只能有一個參數設定，因此陰影臨界值必須取大多數地方適用的值，而編號 9 這段因為在高速公路正下方的道路，路面非常黑暗，跟一般路況不一樣，所以在此段偵測率只能片片斷斷的偵測到車輛。而在上圖編號 14，沒有偵測率和錯誤警示率的原因，是因為車子在十字路口等紅綠燈，前方很多橫向來來去去的車輛，十分難判斷數據，因此在本文裡並沒有去算出數據。而本文中的偵測率在一些簡單狀況非常高，如高速公路上簡單的前車切換車道(上圖編號 1)、連續的前車偵測(上圖編號 5)、近距離車輛偵測(上圖編號 5、8)，可以說是接近完美，而本文有使用道路線來過濾車輛(車輛在某位置左右道路線內一個範圍的車輛大小才算車輛)，使的車輛的錯誤警示率非常低，例如在前方長距離無車情況(上圖編號 15~19)，錯誤警示率幾乎都是 0，因此在前方長距離無車輛狀況下，不會莫名

其妙的出現車輛警示，駕駛者可以安靜地行駛。本文的車輛偵測算法在鏡前反光的狀況下也不會受到嚴重干擾(上圖編號 10、11)，在市區雖然車輛比較密集，但偵測率還是有 88% 以上(上圖編號 12、13)，甚至 99.3%(上圖編號 13)。

## 第七章 結論、未來研究方向

本文在最後面對此研究做最後的結論，並且說明未來研究方向，且目前已經有非常初步的夜間偵測結果，也在此章節呈現。

### 7.1 結論

本文研究的目標是基於車道線的前車偵測，最終目的是希望藉由行車紀錄器得到的影像，藉由畫面的多特徵，使用演算法來對車輛偵測及追蹤，並提供一些輔助訊息，幫助人們在實際駕駛時能減少發生意外，增加行車的安全性。

本文主要是對日間道路線及車輛進行偵測與追蹤，並提供不同的模式，及偵測方法供使用者選擇。而本文的車道線偵測方法，可以避免在道路線中間的字的干擾道路線偵測，而本文的自動陰影臨界值，可以適應大部分畫面自動亮度變化的影像，皆具有找出陰影的最大亮度的功能，而本文的演算法利用道路線的比例和簡單驗證的方法，可以大幅地降低錯誤警示率，使的偵測效果更好。

### 7.2 未來研究方向

由於本文目前皆是以軟體開發角度對程式進行開發，希望未來有一天能套用在實際的攝影機上面，並在高階的 CPU 上能達到即時的處理能力。

在下面列出其他未來研究目標：

1. 多目標偵測：



其實本研究目前可以多目標偵測，但是陰影必須有一部份在車道內，未來希望能直接對全畫面進行車輛偵測。

2. 不同的道路線描述方法：

希望可以使用曲線的方式來描述車道線，提高道路線的準確率。

3. 雨天車輛偵測：

希望能在雨天偵測車輛

4. 夜間車輛偵測：

希望能在夜間偵測車輛

## 參考文獻 (References)

- [1]. M. Herbert, "Active and Passive Range Sensing for Robotics," Proc. IEEE Int'l Conf. Robotics and Automation, vol. 1, pp. 102-110, 2000.
- [2]. S. Park, T. Kim, S. Kang, and K. Heon, "A Novel Signal Processing Technique for Vehicle Detection Radar," 2003 IEEE MTT-S Int'l Microwave Symp. Digest, pp. 607-610, 2003.
- [3]. C. Wang, C. Thorpe, and A. Suppe, "Ladar-Based Detection and Tracking of Moving Objects from a Ground Vehicle at High Speeds," Proc. IEEE Intelligent Vehicles Symp., 2003.
- [4]. J. Hancock, E. Hoffman, R. Sullivan, D. Ingimarson, D. Langer, and M. Hebert, "High-Performance Laser Ranger Scanner," Proc. SPIE Int'l Conf. Intelligent Transportation Systems, 1997.
- [5]. R. Chellappa, G. Qian, and Q. Zheng, "Vehicle Detection and Tracking Using Acoustic and Video Sensors," Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing, pp. 793-796, 2004.
- [6]. Zehang Sun, George Bebis and Ronald Miller, "On-Road Vehicle Detection: A Review", IEEE Transactions On Pattern Analysis And Machine Intelligence, Vol. 28, No. 5, MAY 2006.
- [7]. M. Bertozzi, A. Broggi, M. Cellario, A. Fascioli, P. Lombardi, and M. Porta, "Artificial Vision in Road Vehicles," Proc. IEEE, vol. 90, no. 7, pp. 1258-1271, 2002.
- [8]. 張毅, "即時路面標線、車輛偵測與距離估計", 淡江大學資訊工程研究所, 碩士論文, 2002
- [9]. 沈信良, "先進安全車輛的道路線及前車視覺偵測技術", 國立中央大學資訊工程研究所, 碩士論文, 2007
- [10]. 石博宇, "駕駛助理視覺系統之日間高速公路前車及鄰接車輛偵測", 國立中正大學電機工程研究所, 碩士論文, 2003
- [11]. 孫宗瀛, "結合慣性道路線標記與車輛陰影之前車偵測演算法", 國立東華大學電機工程研究所, 碩士論文, 2011
- [12]. 鄭凌軒, "DSP-Based之車路視覺系統之研究", 國立中山大學電機工程研究所, 碩士論文, 2005
- [13]. 林全財, "影像處理與電腦視覺技術應用於駕駛輔助系統之研究", 國立交通大學電機與控制工程研究所, 碩士論文, 2009
- [14]. 維基百科, <http://zh.wikipedia.org/wiki/Wikipedia:%E9%A6%96%E9%A1%B5>
- [15]. 逍遙文工作室, <http://cg2010studio.wordpress.com/2011/06/06/transform-image-to-gray-level/>

- [16]. 中央大學資訊工程學系 MIAT(機器智慧與自動化技術)實驗室網頁，  
<http://140.115.11.235/~chen/course/vision/ch5/ch5.htm>
- [17]. 微軟msdn，<http://msdn.microsoft.com/>
- [18]. [C#] 將影像轉為Byte Array - 個人新聞台- PChome，  
<http://mypaper.pchome.com.tw/middlehuang/post/1321779350>
- [19]. 撰寫影像處理程式難不倒我!! - 簡單的數位影像處理( C# 篇)，  
[http://debut.cis.nctu.edu.tw/~ching/Course/AdvancedC++Course/\\_Page/Advanced\\_PChome/03%20ImageProcessing%20using%20C%20sharp/ImageProcessing\\_Using\\_C\\_Sharp.pdf](http://debut.cis.nctu.edu.tw/~ching/Course/AdvancedC++Course/_Page/Advanced_PChome/03%20ImageProcessing%20using%20C%20sharp/ImageProcessing_Using_C_Sharp.pdf)
- [20]. Heresy's Space，  
<http://kheresy.wordpress.com/2008/08/18/opengl-%e7%b0%a1%e4%bb%8b/>
- [21]. 邊克老狼2012，  
<http://www.cnblogs.com/mikewolf2002/archive/2011/12/17/2291239.html>
- [22]. OpenCL.NET，  
<http://www.hoopoe-cloud.com/Solutions/OpenCL.NET/Default.aspx>
- [23]. <http://www.mindfiresolutions.com/Coalescing-Global-Memory-Access.htm>
- [24]. OpenCL 規範 1.0 版 - 出自 Khronos OPENCL Working Group

## 附錄一（前處理未優化程式碼）

//灰階化 輸入：原圖 輸出：灰階圖

```
public void GrayLevel()
{
    int i, j;

    Bitmap Bmp = hahaha.TempOrigin, BmpGray = new Bitmap(800, 450,
PixelFormat.Format24bppRgb);

    BitmapData ByteArray = Bmp.LockBits(new Rectangle(0, 0, Bmp.Width, Bmp.Height),
ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);

    BitmapData ByteGrayArray = BmpGray.LockBits(new Rectangle(0, 0, BmpGray.Width,
BmpGray.Height), ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);

    int BmpStride = ByteArray.Stride;
    int BmpByteOfSkip = BmpStride - Bmp.Width * 3;

    unsafe
    {
        byte* ptr = (byte*)ByteArray.Scan0;
        byte* ptrgray = (byte*)ByteGrayArray.Scan0;
        for (j = 0; j < hahaha.TempOrigin.Height; j++)
        {
            for (i = 0; i < hahaha.TempOrigin.Width; i++)
            {
                double gray = 0.114 * ptr[0] + 0.587 * ptr[1] + 0.299 * ptr[2];
                ptrgray[0] = ptrgray[1] = ptrgray[2] = (byte)gray;
                ptr += 3;
                ptrgray += 3;
            }
            ptr += BmpByteOfSkip;
            ptrgray += BmpByteOfSkip;
        }
    }

    Bmp.UnlockBits(ByteArray);
    BmpGray.UnlockBits(ByteGrayArray);
    hahaha.TempGray = BmpGray;
}
```

//水平濾波及所有邊緣檢測 輸入：灰階圖 輸出：水平濾波值、水平及垂直邊緣檢測值存在一張Bitmap中

```
public void SobelandLowPass()
{
    int i, j;
    double tempH, tempV, tempLowpass;
    hahaha.tempSobelandLowpassInfo = new Bitmap(hahaha.TempGray.Width,
hahaha.TempGray.Height, PixelFormat.Format24bppRgb);
    BitmapData byteArray = hahaha.TempGray.LockBits(new Rectangle(0, 0,
hahaha.TempGray.Width, hahaha.TempGray.Height), ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    BitmapData byteInfoArray = hahaha.tempSobelandLowpassInfo.LockBits(new Rectangle(0, 0,
hahaha.tempSobelandLowpassInfo.Width, hahaha.tempSobelandLowpassInfo.Height),
ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
    int bmpstride = byteArray.Stride;
    int bmpByteOfSkip = bmpstride - hahaha.TempGray.Width * 3;
    unsafe
    {
        byte* ptr = (byte*)byteArray.Scan0;
        byte* ptr1 = null, ptr2 = null, ptr3 = null;
        byte* ptr4 = (byte*)byteInfoArray.Scan0;
        for (j = 0; j < hahaha.TempGray.Height; j++)
        {
            if (j == 0)
            {
                ptr1 = ptr + bmpstride * (hahaha.TempGray.Height - 1);
                ptr2 = ptr;
                ptr3 = ptr2 + bmpstride;
            }
            else if (j == 1)
            {
                ptr1 = ptr;
                ptr2 = ptr1 + bmpstride;
                ptr3 = ptr2 + bmpstride;
            }
            else if (j == hahaha.TempGray.Height-1)
                ptr3 = ptr;
```

```

for (i = 0; i < hahaha.TempGray.Width; i++)
{
    tempH = 0;
    tempV = 0;
    tempLowpass = 0;
    if (i == 0)
    {
        tempH += 1 * ptr1[3 * (hahaha.TempGray.Width-1)] + 2 * ptr1[0]
+ 1 * ptr1[3];
        tempH += 0 * ptr2[3 * (hahaha.TempGray.Width-1)] + 0 * ptr2[0]
+ 0 * ptr2[3];
        tempH += -1 * ptr3[3 * (hahaha.TempGray.Width-1)] - 2 * ptr3[0] -
1 * ptr3[3];
        tempV += -1 * ptr1[3 * (hahaha.TempGray.Width - 1)] + 0 * ptr1[0]
+ 1 * ptr1[3];
        tempV += -2 * ptr2[3 * (hahaha.TempGray.Width - 1)] + 0 * ptr2[0]
+ 2 * ptr2[3];
        tempV += -1 * ptr3[3 * (hahaha.TempGray.Width - 1)] + 0 * ptr3[0]
+ 1 * ptr3[3];
        //tempLowpass += 1 * ptr1[3 * (hahaha.TempGray.Width - 1)] + 1 *
ptr1[0] + 1 * ptr1[3];
        tempLowpass += 1 * ptr2[3 * (hahaha.TempGray.Width - 1)] + 1 *
ptr2[0] + 1 * ptr2[3];
        //tempLowpass += 1 * ptr3[3 * (hahaha.TempGray.Width - 1)] + 1 *
ptr3[0] + 1 * ptr3[3];
    }
    else if (i == hahaha.TempGray.Width-1)
    {
        tempH += 1 * *(ptr1 - 3) + 2 * ptr1[0] + 1 * *(ptr1 - 3 *
(hahaha.TempGray.Width - 1));
        tempH += 0 * *(ptr2 - 3) + 0 * ptr2[0] + 0 * *(ptr2 - 3 *
(hahaha.TempGray.Width - 1));
        tempH += -1 * *(ptr3 - 3) - 2 * ptr3[0] - 1 * *(ptr3 - 3 *
(hahaha.TempGray.Width - 1));
        tempV += -1 * *(ptr1 - 3) + 0 * ptr1[0] + 1 * *(ptr1 - 3 *
(hahaha.TempGray.Width - 1));
        tempV += -2 * *(ptr2 - 3) + 0 * ptr2[0] + 2 * *(ptr2 - 3 *
(hahaha.TempGray.Width - 1));
    }
}

```

```

tempV += -1 * *(ptr3 - 3) + 0 * ptr3[0] + 1 * *(ptr3 - 3 *
(hahaha.TempGray.Width - 1));
//tempLowpass += 1 * *(ptr1 - 3) + 1 * ptr1[0] + 1 * *(ptr1 - 3 *
(hahaha.TempGray.Width - 1));
tempLowpass += 1 * *(ptr2 - 3) + 1 * ptr2[0] + 1 * *(ptr2 - 3 *
(hahaha.TempGray.Width - 1));
//tempLowpass += 1 * *(ptr3 - 3) + 1 * ptr3[0] + 1 * *(ptr3 - 3 *
(hahaha.TempGray.Width - 1));
}
else
{
tempH += 1 * *(ptr1 - 3) + 2 * ptr1[0] + 1 * *(ptr1 + 3);
tempH += 0 * *(ptr2 - 3) + 0 * ptr2[0] + 0 * *(ptr2 + 3);
tempH += -1 * *(ptr3 - 3) - 2 * ptr3[0] - 1 * *(ptr3 + 3);
tempV += -1 * *(ptr1 - 3) + 0 * ptr1[0] + 1 * *(ptr1 + 3);
tempV += -2 * *(ptr2 - 3) + 0 * ptr2[0] + 2 * *(ptr2 + 3);
tempV += -1 * *(ptr3 - 3) + 0 * ptr3[0] + 1 * *(ptr3 + 3);
//tempLowpass += 1 * *(ptr1 - 3) + 1 * ptr1[0] + 1 * *(ptr1 + 3);
tempLowpass += 1 * *(ptr2 - 3) + 1 * ptr2[0] + 1 * *(ptr2 + 3);
//tempLowpass += 1 * *(ptr3 - 3) + 1 * ptr3[0] + 1 * *(ptr3 + 3);
}
//tempLowpass = tempLowpass / 9;
tempLowpass = tempLowpass / 3;
ptr4[0] = (byte)Math.Abs(tempH);
ptr4[1] = (byte)Math.Abs(tempV);
ptr4[2] = (byte)tempLowpass;
ptr1 += 3;
ptr2 += 3;
ptr3 += 3;
ptr4 += 3;
}
ptr1 += bmpByteOfSkip;
ptr2 += bmpByteOfSkip;
ptr3 += bmpByteOfSkip;
ptr4 += bmpByteOfSkip;
}
}
hahaha.TempGray.UnlockBits(byteArray);

```

```

        hahaha.tempSobelandLowpassInfo.UnlockBits(byteInfoArray);

    }

//水平濾波圖及所有邊緣檢測圖 輸入：水平濾波值、水平及垂直邊緣檢測值的Bitmap中輸出：
四張圖
public void SobelHVAllandLowPass()
{
    int i, j;
    hahaha.TempSobel = new Bitmap(hahaha.TempGray.Width, hahaha.TempGray.Height,
PixelFormat.Format24bppRgb);
    hahaha.TempSobelH = new Bitmap(hahaha.TempGray.Width, hahaha.TempGray.Height,
PixelFormat.Format24bppRgb);
    hahaha.TempSobelV = new Bitmap(hahaha.TempGray.Width, hahaha.TempGray.Height,
PixelFormat.Format24bppRgb);
    Bitmap BmpLowPass= new Bitmap(hahaha.TempGray.Width, hahaha.TempGray.Height,
PixelFormat.Format24bppRgb);
    BitmapData byteInfoArray = hahaha.tempSobelandLowpassInfo.LockBits(new Rectangle(0, 0,
hahaha.TempGray.Width, hahaha.TempGray.Height), ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    BitmapData byteSobelArray = hahaha.TempSobel.LockBits(new Rectangle(0, 0,
hahaha.TempGray.Width, hahaha.TempGray.Height), ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    BitmapData byteSobelHArray = hahaha.TempSobelH.LockBits(new Rectangle(0, 0,
hahaha.TempGray.Width, hahaha.TempGray.Height), ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    BitmapData byteSobelVArray = hahaha.TempSobelV.LockBits(new Rectangle(0, 0,
hahaha.TempGray.Width, hahaha.TempGray.Height), ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    BitmapData byteLowPassArray = BmpLowPass.LockBits(new Rectangle(0, 0,
hahaha.TempGray.Width, hahaha.TempGray.Height), ImageLockMode.ReadWrite,
PixelFormat.Format24bppRgb);
    int bmpstride = byteInfoArray.Stride;
    int bmpByteOfSkip = bmpstride - hahaha.TempGray.Width * 3;

    unsafe
    {

```



```

byte* ptr = (byte*)byteInfoArray.Scan0;
byte* ptr1 = (byte*)byteSobelArray.Scan0;
byte* ptr2 = (byte*)byteSobelHArray.Scan0;
byte* ptr3 = (byte*)byteSobelVArray.Scan0;
byte* ptr4 = (byte*)byteLowPassArray.Scan0;
for (j = 0; j < hahaha.TempOrigin.Height; j++)
{
    for (i = 0; i < hahaha.TempOrigin.Width; i++)
    {
        if (Math.Abs(ptr[0]) >= hahaha.sobelthreshold || Math.Abs(ptr[1]) >=
hahaha.sobelthreshold)

            ptr1[0] = ptr1[1] = ptr1[2] = 255;
        else
            ptr1[0] = ptr1[1] = ptr1[2] = 0;
        if (Math.Abs(ptr[0]) >= hahaha.sobelthreshold)
            ptr2[0] = ptr2[1] = ptr2[2] = 255;
        else
            ptr2[0] = ptr2[1] = ptr2[2] = 0;
        if (Math.Abs(ptr[1]) >= hahaha.sobelthreshold)
            ptr3[0] = ptr3[1] = ptr3[2] = 255;
        else
            ptr3[0] = ptr3[1] = ptr3[2] = 0;
        ptr4[0] = ptr4[1] = ptr4[2] = ptr[2];
        ptr += 3;
        ptr1 += 3;
        ptr2 += 3;
        ptr3 += 3;
        ptr4 += 3;
    }
    ptr += bmpByteOfSkip;
    ptr1 += bmpByteOfSkip;
    ptr2 += bmpByteOfSkip;
    ptr3 += bmpByteOfSkip;
    ptr4 += bmpByteOfSkip;
}
}

hahaha.tempSobelandLowpassInfo.UnlockBits(byteInfoArray);
hahaha.TempSobel.UnlockBits(byteSobelArray);

```

```
hahaha.TempSobelH.UnlockBits(byteSobelHArray);  
hahaha.TempSobelV.UnlockBits(byteSobelVArray);  
BmpLowPass.UnlockBits(byteLowPassArray);  
hahaha.NinjaHideT = hahaha.TempGray;  
hahaha.TempGray = BmpLowPass;  
}
```

## 附錄二(OpenCL kernel code)

灰階化(二維圖像)    輸入：input    輸出：output

```
__kernel void ImageGray(__read_only image2d_t input,__write_only image2d_t output)
{
    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    uint4 pixel = read_imageui( input,smp,coord );
    float gray = 0.299 * pixel.x+0.587 * pixel.y+0.114 * pixel.z;
    uint4 pixelnew = { gray , gray , gray , 1.0f };
    write_imageui( output,coord,pixelnew );
}
```

水平濾波及全部邊緣檢測圖(二維圖像)輸入：input    輸出：output, output1, output2, output3

```
__kernel void ImageSobelandLowpass(__read_only image2d_t input, __write_only image2d_t output,
__write_only image2d_t output1, __write_only image2d_t output2, __write_only image2d_t output3,
int threshold)
{
    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int width = get_global_size(0);
    int height = get_global_size(1);
    int2 coord1, coord2, coord3, coord4, coord5, coord6, coord7, coord8;
    uint4 pixel, pixel1, pixel2, pixel3, pixel4, pixel5, pixel6, pixel7, pixel8;
    float SobelH, SobelV, Lowpass;
    uint4 pixelSobel, pixelSobelH, pixelSobelV;
    if(isequal( (float)coord.x, (float)0 ) == 1)
    {
        coord1.x = width - 1; coord2.x = 0; coord3.x = 1;
        coord4.x = width - 1; ; coord5.x = 1;
        coord6.x = width - 1; coord7.x = 0; coord8.x = 1;
    }
    else if(isequal( (float)coord.x, (float)(width - 1) ) == 1)
    {
        coord1.x = width - 2; coord2.x = width - 1; coord3.x = 0;
        coord4.x = width - 2; ; coord5.x = 0;
        coord6.x = width - 2; coord7.x = width - 1; coord8.x = 0;
    }
    else
```

```

{
    coord1.x = coord.x - 1; coord2.x = coord.x; coord3.x = coord.x + 1;
    coord4.x = coord.x - 1;                ; coord5.x = coord.x + 1;
    coord6.x = coord.x - 1; coord7.x = coord.x; coord8.x = coord.x + 1;
}
if(isequal( (float)coord.y, (float)0 )==1)
{
    coord1.y = height - 1; coord2.y = height - 1; coord3.y = height - 1;
    coord4.y = 0                ;                ; coord5.y = 0                ;
    coord6.y = 1                ; coord7.y = 1                ; coord8.y = 1                ;
}
else if(isequal( (float)coord.y, (float)height-1 )==1)
{
    coord1.y = height - 2; coord2.y = height - 2; coord3.y = height - 2;
    coord4.y = height - 1;                ; coord5.y = height - 1 ;
    coord6.y = 0                ; coord7.y = 0                ; coord8.y = 0                ;
}
else
{
    coord1.y = coord.y - 1 ; coord2.y = coord.y - 1 ; coord3.y = coord.y - 1;
    coord4.y = coord.y    ;                ; coord5.y = coord.y    ;
    coord6.y = coord.y + 1; coord7.y = coord.y + 1; coord8.y = coord.y + 1;
}
pixel  = read_imageui( input,smp,coord );
pixel1 = read_imageui( input,smp,coord1 );
pixel2 = read_imageui( input,smp,coord2 );
pixel3 = read_imageui( input,smp,coord3 );
pixel4 = read_imageui( input,smp,coord4 );
pixel5 = read_imageui( input,smp,coord5 );
pixel6 = read_imageui( input,smp,coord6 );
pixel7 = read_imageui( input,smp,coord7 );
pixel8 = read_imageui( input,smp,coord8 );

SobelH = 1 * (int)pixel1.x + 2 * (int)pixel2.x + 1 * (int)pixel3.x +
          0 * (int)pixel4.x + 0 * (int)pixel.x    + 0 * (int)pixel5.x
          - 1 * (int)pixel6.x - 2 * (int)pixel7.x - 1 * (int)pixel8.x;

SobelV = -1 * (int)pixel1.x + 0 * (int)pixel2.x + 1 * (int)pixel3.x

```

```

-2 * (int)pixel4.x + 0 * (int)pixel.x      + 2 * (int)pixel5.x
-1 * (int)pixel6.x + 0 * (int)pixel7.x    + 1 * (int)pixel8.x;
Lowpass = 1 * pixel4.x + 1 * pixel.x + 1 * pixel5.x;
Lowpass = Lowpass / 3;
SobelH = fabs(SobelH);
SobelV = fabs(SobelV);
if(isgreaterorequal( SobelH, threshold ) == 1 || isgreaterorequal( SobelV, threshold ) == 1)
{
    pixelSobel.x = 255;
    pixelSobel.y = 255;
    pixelSobel.z = 255;
    pixelSobel.w = 1;
}
else
{
    pixelSobel.x = 0;
    pixelSobel.y = 0;
    pixelSobel.z = 0;
    pixelSobel.w = 1;
}
if(isgreaterorequal( SobelH, threshold ) == 1)
{
    pixelSobelH.x = 255;
    pixelSobelH.y = 255;
    pixelSobelH.z = 255;
    pixelSobelH.w = 1;
}
else
{
    pixelSobelH.x = 0;
    pixelSobelH.y = 0;
    pixelSobelH.z = 0;
    pixelSobelH.w = 1;
}
if(isgreaterorequal( SobelV, threshold ) == 1)
{
    pixelSobelV.x = 255;
    pixelSobelV.y = 255;

```

```

        pixelSobelV.z = 255;
        pixelSobelV.w = 1;
    }
    else
    {
        pixelSobelV.x = 0;
        pixelSobelV.y = 0;
        pixelSobelV.z = 0;
        pixelSobelV.w = 1;
    }
    uint4 pixelLowpass = { Lowpass, Lowpass, Lowpass, 1.0f };
    write_imageui( output, coord, pixelSobel );
    write_imageui( output1, coord, pixelSobelH );
    write_imageui( output2, coord, pixelSobelV );
    write_imageui( output3, coord, pixelLowpass );
}

```

灰階化(ByteArray)    輸入：input    輸出：output

```

__kernel void ImageGrayArray( global uchar* pInput, global uchar* pOutput)
{
    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int2 size = (int2)(get_global_size(0), get_global_size(1));
    float gray = 0.299 * convert_float( pInput[ 3 * (coord.y * size.x + coord.x) + 2] )
    + 0.587 * convert_float( pInput[ 3 * (coord.y * size.x + coord.x) + 1] )
    + 0.114 * convert_float( pInput[ 3 * (coord.y * size.x + coord.x) ] );
    uchar pixelnew = convert_uchar_sat_rte(gray);
    pOutput[ 3 * (coord.y * size.x + coord.x) + 2] = pixelnew;
    pOutput[ 3 * (coord.y * size.x + coord.x) + 1] = pixelnew;
    pOutput[ 3 * (coord.y * size.x + coord.x) ] = pixelnew;
}

```

水平濾波及全部邊緣檢測圖(ByteArray)    輸入：input    輸出：output, output1, output2,

output3

```

typedef struct PixelColor
{
    uchar x,y,z;
}ColorRGBA;

__kernel void ImageSobelandLowpassArray(global uchar* pGrayInput, global uchar* pSobelOutput,
global uchar* pSobelHOutput, global uchar* pSobelVOutput, global uchar* pLowpassOutput, int
threshold)
{
    int2 coord = (int2)(get_global_id(0), get_global_id(1));
    int2 size = (int2)(get_global_size(0), get_global_size(1));

    int2 coord1, coord2, coord3, coord4, coord5, coord6, coord7, coord8;
    ColorRGBA pixel, pixel1, pixel2, pixel3, pixel4, pixel5, pixel6, pixel7, pixel8;
    float SobelH, SobelV, Lowpass;
    ColorRGBA pixelSobel, pixelSobelH, pixelSobelV;

    if(isequal( (float)coord.x, (float)0 ) == 1)
    {
        coord1.x = size.x - 1; coord2.x = 0; coord3.x = 1;
        coord4.x = size.x - 1; ; coord5.x = 1;
        coord6.x = size.x - 1; coord7.x = 0; coord8.x = 1;
    }
    else if(isequal( (float)coord.x, (float)(size.x - 1) ) == 1)
    {
        coord1.x = size.x - 2; coord2.x = size.x - 1; coord3.x = 0;
        coord4.x = size.x - 2; ; coord5.x = 0;
        coord6.x = size.x - 2; coord7.x = size.x - 1; coord8.x = 0;
    }
    else
    {
        coord1.x = coord.x - 1; coord2.x = coord.x; coord3.x = coord.x + 1;
        coord4.x = coord.x - 1; ; coord5.x = coord.x + 1;
        coord6.x = coord.x - 1; coord7.x = coord.x; coord8.x = coord.x + 1;
    }
    if(isequal( (float)coord.y, (float)0 )==1)
    {
        coord1.y = size.y - 1; coord2.y = size.y - 1; coord3.y = size.y - 1;

```

```

        coord4.y = 0 ; ; coord5.y = 0 ;
        coord6.y = 1 ; coord7.y = 1 ; coord8.y = 1 ;
    }
    else if(isequal( (float)coord.y, (float)size.y-1 )==1)
    {
        coord1.y = size.y - 2; coord2.y = size.y - 2; coord3.y = size.y - 2;
        coord4.y = size.y - 1; ; coord5.y = size.y - 1 ;
        coord6.y = 0 ; coord7.y = 0 ; coord8.y = 0 ;
    }
    else
    {
        coord1.y = coord.y - 1 ; coord2.y = coord.y - 1 ; coord3.y = coord.y - 1;
        coord4.y = coord.y ; ; coord5.y = coord.y ;
        coord6.y = coord.y + 1; coord7.y = coord.y + 1; coord8.y = coord.y + 1;
    }

    pixel = (ColorRGBA){ pGrayInput[3 * (coord.y * size.x + coord.x) + 2] , pGrayInput[3 *
(coord.y * size.x + coord.x) + 1] , pGrayInput[3 * (coord.y * size.x + coord.x)] };
    pixel1 = (ColorRGBA){ pGrayInput[3 * (coord1.y * size.x + coord1.x) + 2] , pGrayInput[3 *
(coord1.y * size.x + coord1.x) + 1] , pGrayInput[3 * (coord1.y * size.x + coord1.x)] };
    pixel2 = (ColorRGBA){ pGrayInput[3 * (coord2.y * size.x + coord2.x) + 2] , pGrayInput[3 *
(coord2.y * size.x + coord2.x) + 1] , pGrayInput[3 * (coord2.y * size.x + coord2.x)] };
    pixel3 = (ColorRGBA){ pGrayInput[3 * (coord3.y * size.x + coord3.x) + 2] , pGrayInput[3 *
(coord3.y * size.x + coord3.x) + 1] , pGrayInput[3 * (coord3.y * size.x + coord3.x)] };
    pixel4 = (ColorRGBA){ pGrayInput[3 * (coord4.y * size.x + coord4.x) + 2] , pGrayInput[3 *
(coord4.y * size.x + coord4.x) + 1] , pGrayInput[3 * (coord4.y * size.x + coord4.x)] };
    pixel5 = (ColorRGBA){ pGrayInput[3 * (coord5.y * size.x + coord5.x) + 2] , pGrayInput[3 *
(coord5.y * size.x + coord5.x) + 1] , pGrayInput[3 * (coord5.y * size.x + coord5.x)] };
    pixel6 = (ColorRGBA){ pGrayInput[3 * (coord6.y * size.x + coord6.x) + 2] , pGrayInput[3 *
(coord6.y * size.x + coord6.x) + 1] , pGrayInput[3 * (coord6.y * size.x + coord6.x)] };
    pixel7 = (ColorRGBA){ pGrayInput[3 * (coord7.y * size.x + coord7.x) + 2] , pGrayInput[3 *
(coord7.y * size.x + coord7.x) + 1] , pGrayInput[3 * (coord7.y * size.x + coord7.x)] };
    pixel8 = (ColorRGBA){ pGrayInput[3 * (coord8.y * size.x + coord8.x) + 2] , pGrayInput[3 *
(coord8.y * size.x + coord8.x) + 1] , pGrayInput[3 * (coord8.y * size.x + coord8.x)] };

    SobelH = 1 * (int)pixel1.x + 2 * (int)pixel2.x + 1 * (int)pixel3.x +
0 * (int)pixel4.x + 0 * (int)pixel1.x + 0 * (int)pixel5.x
- 1 * (int)pixel6.x - 2 * (int)pixel7.x - 1 * (int)pixel8.x;

```



```

SobelV =  -1 * (int)pixel1.x + 0 * (int)pixel2.x  + 1 * (int)pixel3.x
          -2 * (int)pixel4.x + 0 * (int)pixel.x    + 2 * (int)pixel5.x
          -1 * (int)pixel6.x + 0 * (int)pixel7.x  + 1 * (int)pixel8.x;

Lowpass = 1 * (int)pixel4.x + 1 * (int)pixel.x + 1 * (int)pixel5.x;
Lowpass = Lowpass / 3;

SobelH = fabs(SobelH);
SobelV = fabs(SobelV);

if(isgreaterequal( SobelH, threshold ) == 1 || isgreaterequal( SobelV, threshold ) == 1)
{
    pixelSobel = (ColorRGBA){ 255, 255, 255 };
}
else
{
    pixelSobel = (ColorRGBA){ 0, 0, 0 };
}
if(isgreaterequal( SobelH, threshold ) == 1)
{
    pixelSobelH = (ColorRGBA){ 255, 255, 255 };
}
else
{
    pixelSobelH = (ColorRGBA){ 0, 0, 0 };
}
if(isgreaterequal( SobelV, threshold ) == 1)
{
    pixelSobelV = (ColorRGBA){ 255, 255, 255 };
}
else
{
    pixelSobelV = (ColorRGBA){ 0, 0, 0 };
}
ColorRGBA pixelLowpass = (ColorRGBA){ convert_uchar_sat_rte(Lowpass),
convert_uchar_sat_rte(Lowpass), convert_uchar_sat_rte(Lowpass) };

```

```

pSobelOutput[3 * (coord.y * size.x + coord.x) + 2 ] = pixelSobel.x;
pSobelOutput[3 * (coord.y * size.x + coord.x) + 1 ] = pixelSobel.y;
pSobelOutput[3 * (coord.y * size.x + coord.x) ] = pixelSobel.z;
pSobelHOutput[3 * (coord.y * size.x + coord.x) + 2 ] = pixelSobelH.x;
pSobelHOutput[3 * (coord.y * size.x + coord.x) + 1 ] = pixelSobelH.y;
pSobelHOutput[3 * (coord.y * size.x + coord.x) ] = pixelSobelH.z;
pSobelVOutput[3 * (coord.y * size.x + coord.x) + 2 ] = pixelSobelV.x;
pSobelVOutput[3 * (coord.y * size.x + coord.x) + 1 ] = pixelSobelV.y;
pSobelVOutput[3 * (coord.y * size.x + coord.x) ] = pixelSobelV.z;
pLowpassOutput[3 * (coord.y * size.x + coord.x) + 2 ] = pixelLowpass.x;
pLowpassOutput[3 * (coord.y * size.x + coord.x) + 1 ] = pixelLowpass.y;
pLowpassOutput[3 * (coord.y * size.x + coord.x) ] = pixelLowpass.z;
}

```