第三章
系統實現

先前我們已經介紹過我們要實現此系統所需的硬體和軟體設備，接下來我們討論如何利用這些軟體和硬體來設計出這套安全監控模擬系統，提供輔助策略及方法協助使用者操作此系統。

3.1 輔助策略

爲了讓使用者在操控時，可以讓互動過程更順利流暢，在這我們利用輔助策略，讓自動車上的 PTZ 攝影機能夠對準目標物件；此外還提供多視角輔助視窗來觀看場景。在與環境互動前，虛擬實境必須要做碰撞偵測，以及產生碰撞行爲，分述如下。

A.碰撞偵測

3-D 引擎需要知道某個物件何時會撞到環境，並予以阻隔。如果要檢查某個多邊型是否擋在從一點到另一點的路線上，我們可以在再這兩點間投射一條假想的光線，看看它是否會與某個平面產生相交的結果。多邊型是個有限大小的平面。透過建立一個代表這個多邊形的平面，我們可以運用代數的計算，判斷是否相交，如圖 3.1 所示。除了偵測物件網格何時會與環境碰撞之外，我們也要避免物件與物件之間的碰撞情況，例如我們不希望自走車與桌子發生彼此穿透現象，所以我們要加入物件對物件的碰撞偵測。一般的作法是使用邊界方塊(bounding box)[14]-[15]的方法，也就是利用一個最小的立體方塊將一個物體包圍住；但在這裡我們不採用與環境碰撞的偵測方法，而改用簡單的計算，進行物件對物件的

碰撞偵測。我們要做的判斷就是處理中的物件的範圍之內是否有交疊存在。每個物件的範圍大小與它們的長度有關，從圖 3.2 中我們可以發現若移動物件的話，一定會發生兩個範圍會有相交的情形發生，即使這兩個物件並沒有接近。我們計算這兩個物件的中央點距離，若它們的距離小於或等於兩個半徑之合，則表示這兩個範圍會相交，利用上述的方法我們就可避免自動車在環境中發生穿透的現象。

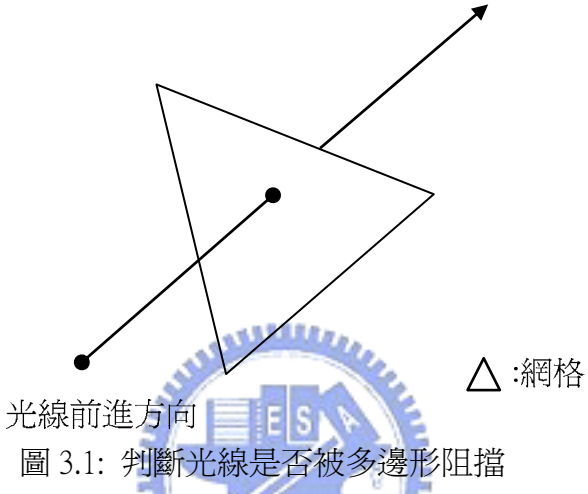


圖 3.1: 判斷光線是否被多邊形阻擋

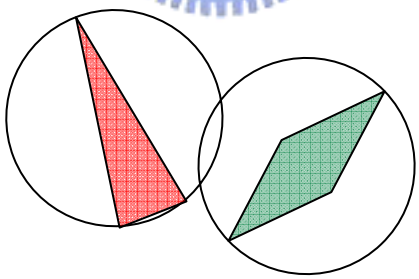


圖 3.2: 物件與物件的碰撞情形

接著就是將輔助策略加入虛擬實境中，以利使用者操控；當受控端距離較遠時，利用網路將畫面傳送回主控端，往往會受到時間延遲而造成影響，使我們無法做即時的控制，所以運用遠端呈現模擬系統來達成即時的控制，接著再將遠端的影像系統所擷取的影像資訊做更近一步的利用，若影像資訊可以提供環境中物件的位置、頂點和邊線，則可判別出物體的方位。

若影像系統擷一物件的影像，在經由影像處理過後，我們得到此物體在空間中的位置、頂點座標以及邊線資訊[22]，如圖 3.3 所示，其爲 DirectX 中的座標系統，但由於目前必須從虛擬場景中來求得，現在只擁有物體的中心座標以及邊線長度的資訊，這些需經一些計算以求得初期頂點座標，圖中 Point 1 到 Point 4 爲此物體的上表面頂點座標，Point 5 到 Point 8 爲此物體的下表面頂點座標，由於我們要判別的是物體在空間中的方向，所以在這兩組座標點我們只需選擇其中一組就可以，且因 Y 軸的資訊用不到，所以可以把這 3 維座標圖轉換成 X-Z 平面的 2 爲座標圖，圖 3.4 所示。

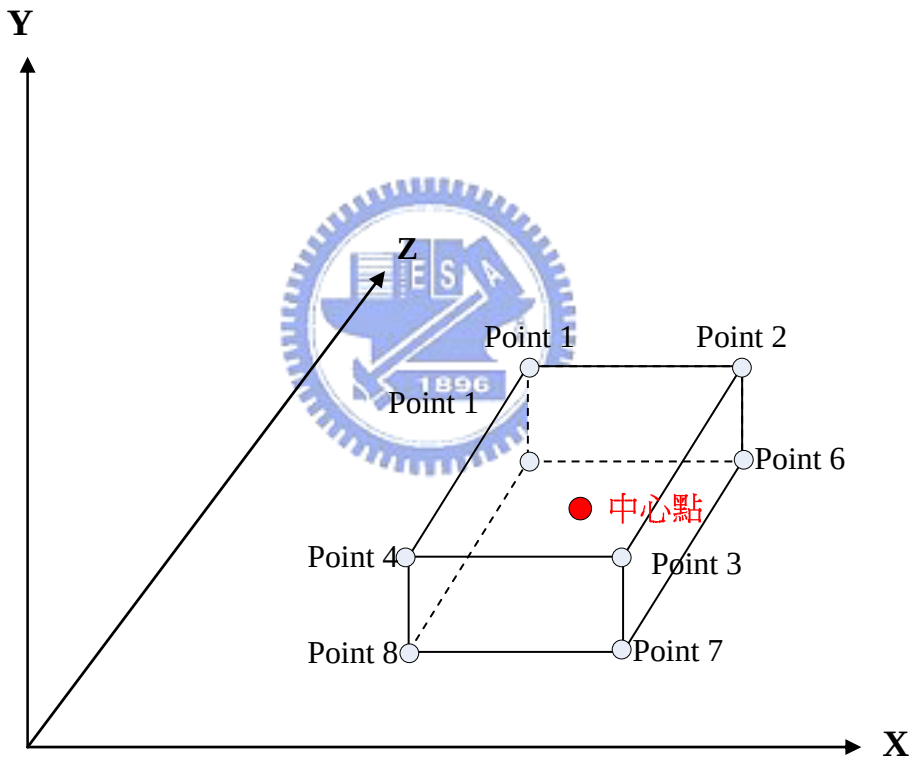


圖 3.3:物體在空間中的頂點座標圖。

圖 3.4 中所示，中心點到四個頂點的距離一樣，其距離都爲 r，所以當中心點確定後，無論物體的方向爲何，其四個頂點座標一定會相對出現在圓心爲中心點，半徑爲 r 的圓上面其中四點；現在假設物體的中心點在 (x_0, z_0)，因此時 Point 1 的座標點爲 (x_1, z_1)，將 x_1 和 z_1 表示成參數形式，若此時中心點 (x_0, z_0) 爲圓心，可用參數式寫成:

$$x_1 = x_0 + r\cos\theta \quad (3.1)$$
$$z_1 = z_0 + r\cos\theta \quad (3.2)$$

其中 r 是中心點和 Point 1 之間的距離，θ視角度參數。

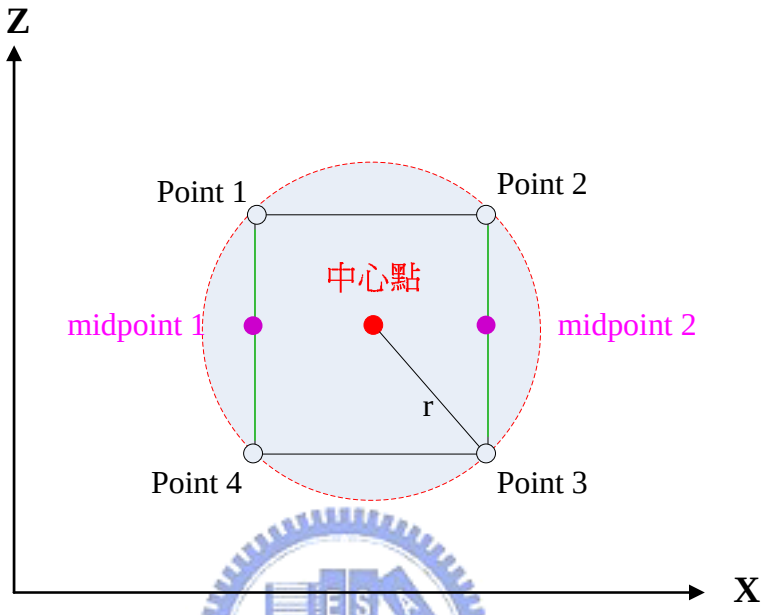


圖 3.4:物體在空間中之 X-Z 平面圖。

當物體沿著 Y 軸旋轉一角度$d\theta$後，Point 1 的座標點變成爲(x_2, z_2)，其可表示成:

$$x_2 = x_0 + r\cos(\theta + d\theta) \quad (3.3)$$
$$z_2 = z_0 + r\cos(\theta + d\vartheta) \quad (3.4)$$

上面兩式可再利用和角公式改寫成:

$$x_2 = x_0 + r\cos\theta\cos d\theta - r\sin\theta\sin d\theta \quad (3.5)$$
$$z_2 = z_0 + r\sin\theta\cos d\vartheta + r\cos\theta\sin d\theta \quad (3.6)$$

接著將(3.1)與(3.2)式，代入(3.5)與(3.6)式，則可得

$$x_2 = x_0 + (x_1 - x_0)\cos d\theta - (z_1 - z_0)\sin d\theta \quad (3.7)$$
$$z_2 = z_0 + (z_1 - z_0)\cos d\vartheta + (x_1 - x_0)\sin d\theta \quad (3.8)$$

最後 Point 1 可表示成(3.7)及(3.8)式；所以當物體在經過互動後，其方向與位置也隨之改變，此時物體的頂點座標可經由(3.7)及(3.8)式求得；在影像系統傳回的

資訊方面，還須包括邊線的判別，由於邊線長度都不同或是頂點過多，例如一梯形立方體或是一個多面立方體，所以不能只從其頂點的位置來計算球得出物體的目前方向。假若影像系統可以判別出一對平行邊線，則我們可以利用構成這一對邊線上的四個頂點座標進而求得物體的方向；假設現在透過影像系統去判別出 Point 1 和 Point 4 構成一邊線，Point 2 和 Point 3 構成一邊線，如圖 3.4 中綠色邊線所示，我們再分別求出其個別邊線的中間點，分別爲 midpoint $1(mx_1, mz_1)$ 和 midpoint $2(mx_2, mz_2)$，再利用這兩點來判別出物體的方向角度，其可表示成

$$angle = \pm\cos^{-1}\left(\frac{|mx_1 - mx_2|}{\sqrt{(mx_1 - mx_2)^2 + (mz_1 - mz_2)^2}}\right) \quad (3.9)$$

或

$$angle = \pm 180° \mp \cos^{-1}\left(\frac{|mx_1 - mx_2|}{\sqrt{(mx_1 - mx_2)^2 + (mz_1 - mz_2)^2}}\right) \quad (3.10)$$

經由上式可以求得出物體目前的方向角度，我們利用這角度來讓自動車上的攝影機能夠旋轉到此方向。

B.多視角輔助視窗

本實驗室之前也開發過類似的操作介面，如圖 3.5 所示。它一次只能提供一個視角畫面，導致使用者在操作上有些不方便，需要不停地切換視角。有鑑於此我們這在虛擬實境上面增加一些輔助視窗來協助操作者，如圖 3.6 所示；圖 3.6(a) 的主視窗能夠完整地得知自動車狀態，爲了能讓使用者不會迷失方向，我們提供一個 2D 的平面地圖(圖 3.6(b))，此地圖能夠讓使用者明確瞭解自動車在環境中的位置，可以自由地規劃路徑以到達目的地；PTZ 攝影機(圖 3.6(c)) 、環場攝影機 (圖 3.6(d))可用來增加視角，讓使用者能夠更清楚地了解環境資訊，接下來的內容分別對這些視窗的功能詳加說明。

圖 3.5: 前一代虛擬實境介面。

(a)主視窗

(b)場景地圖

(c)PTZ攝影機視角

(d)環場攝影機視角

圖 3.6: 本論文所開發的虛擬場景介面。(a)主視窗，(b)場景地圖，(c)PTZ 攝影機視角和(d)環場攝影機視角。

圖 3.6(a)主視窗是以能夠觀測到自動車的狀態爲主，所以在畫面會呈現出自動車完整的狀態，能夠顯示出升降平台的上升下降狀態與自動車行駛的狀態，並且呈現出自動車周圍環境的資訊，讓使用者不會撞到障礙物，如圖 3.7 所示。另外還能夠接收固定式攝影機所傳來的訊息，告知使用者是否發現入侵者，進而決定因應的策略，此外主視窗還能依照使用者的要求，切換到固定式攝影機的鏡頭，如圖 3.8(a)-(b)所示。

圖 3.7: 顯示自動車目前狀態

(a)固定式攝影機視角之一　　(b) 固定式攝影機視角之二

圖 3.8: 環境中固定式攝影機的視角，(a)固定式攝影機視角之一和(b) 固定式攝影機視角之二

這裡我們在此說明加入場景地圖的原因，在[3]-[4]這二篇論文在介紹他們利用一救災機器人(圖 3.9(a))來協助消防人員救出受難者，如圖 3.9(b)所示，他們讓機器人深入火災現場尋找受難者，把受難者正確位置和被阻擋的路線回傳到消防人員的電腦上，讓消防人員能夠以最短的時間解救出受難者。基於上述的優點，我們也在虛擬實境中加入場景地圖來協助操作者。場景地圖提供了定位的資訊，例如自動車在場景中的相對位置，讓使用者不會迷失方向，另外圖中也呈現固定式攝影機的狀態，可以清楚地知道攝影機鏡頭朝向哪個方向(圖 3.6(c)中的紅色三角型是代表自動車，藍色三角型是代表 PTZ 攝影機的視角，藍色三角型是代表 PTZ 攝影機的視角粉紅色三角型是代表固定式攝影機，黃色三角型代表固定式攝影機視角) 。 除此之外我們還提供了偵測入侵者的功能，只要入侵者的位置是在固定式攝影機拍攝的範圍，則地圖就會呈現出入侵者的位置，如圖 3.10 所示，並把此區域做記號，讓使用者能夠注意到此區域，進一步採取相應的策略。

(a)救災機器人[3]　　(b)救災機器人的操作介面

圖 3.9:利用場景地圖來協助救災行動: (a)救災機器人和(b)救災機器人的操作

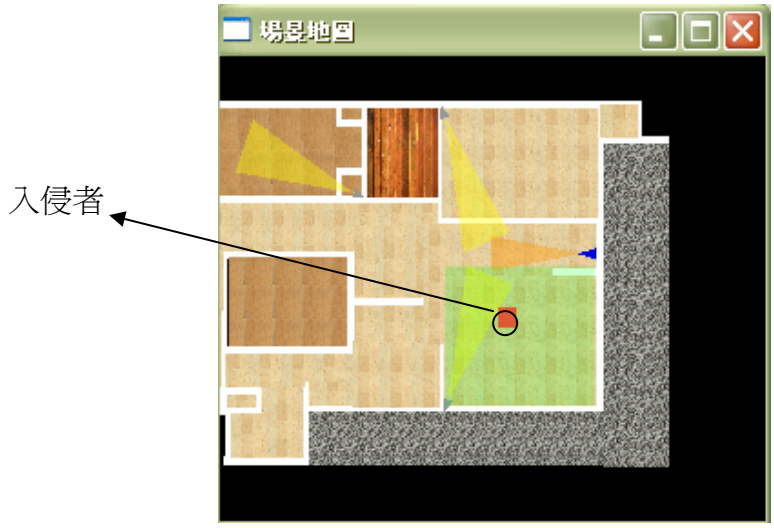
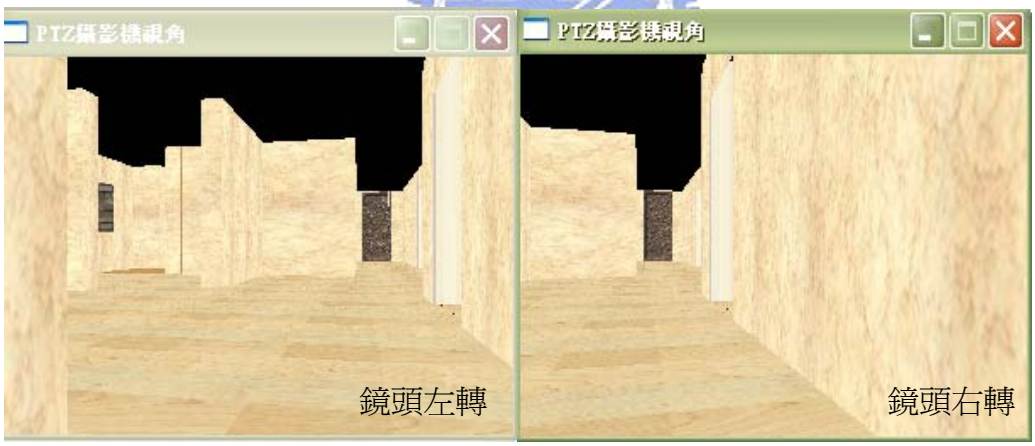


圖 3.10:固定式攝影機範圍內偵測到入侵者

當我們要詳細觀察環境中某個物件時，就必須利用 PTZ 攝影機，由於 PTZ 攝影機視角的功能使鏡頭能夠左右轉動(圖 3.11 (a))、上下擺動(圖 3.11(b))、鏡頭縮放(圖 3.11(c))，因此我們可以由畫面來仔細觀察此物件。



(a)視角左右轉動的狀態

圖 3.11: PTZ 攝影機視角:(a)視角左右轉動的狀態，(b)視角上下擺動的狀態，(c)視角縮放的狀態(cont.)

(b)視角上下擺動的狀

(c)視角縮放的狀態

圖 3.11: PTZ 攝影機視角:(a)視角左右轉動的狀態，(b)視角上下擺動的狀態，(c)視角縮放的狀態

3.2 路徑規畫

完成了以上的基本功能之後，我們接下來討論路徑規畫，在學界方面已提出多種方法，其中一較爲廣泛應用於自動車之路徑規畫的方法稱爲視圖法(Visibility graph method)[16]，這個方法基於以下的原理，首先將自動車的所有可移動空間標示成藉由兩兩相連之點所結成的網路，剩下的問題則是搜尋這個網路，找到起始點到終點的最短距離。搜尋的方法大致分爲兩類，一類稱爲全域搜尋(Global planning)，是由詳細地搜尋所有的網路而得出答案之方法；而另一類稱爲區域搜尋(Local planning)，藉由某種直覺的探索方法來引導其搜尋之方向，而得出答案之方法。一較具代表性的全域搜尋法稱爲 Dijkstra algorithm[17]，它從起始點開始向外搜尋，直到達到終點爲止。另一有名的區域搜尋法爲 BFS(Best-First-Search) algorithm[18]，它以最接近終點的點作爲搜尋的下一目標，減少其搜尋之時間。比較 Dijkstra algorithm 和 BFS algorithm 的搜尋範圍及所找出的路徑，我們可以發現 BFS algorithm 的搜尋範圍通常比 Dijkstra algorithm 小，這是因爲 BFS algorithm 有利用搜尋方法，但是 BFS algorithm 無法保證得出最佳路徑。

較廣爲人應用的 A*演算法[19]，是在 1968 年由 Hart 等三人提出，只要路徑存在，它可以保證回傳最佳路徑，同時擁有搜尋範圍較小的優點。當使用者輸入自動車的目的地後，系統就可以用 A*演算法搜尋格點化之後的地圖，同時將最佳路徑回傳。由於要執行 A*演算法時要用到場景的矩陣地圖，此矩陣可以呈現出場景的資訊，所以我們必須先利用下述沿著牆走的方法來取得此矩陣才能執行 A*演算法。

• 沿著牆走:

步驟一: 若自動車前方無障礙物、右方有障礙物、左方及後方不考慮，則自動車直線前進。

步驟二: 若自動車的右方及前方有障礙物時，則自動車朝左邊旋轉，如圖 3.13 所示。

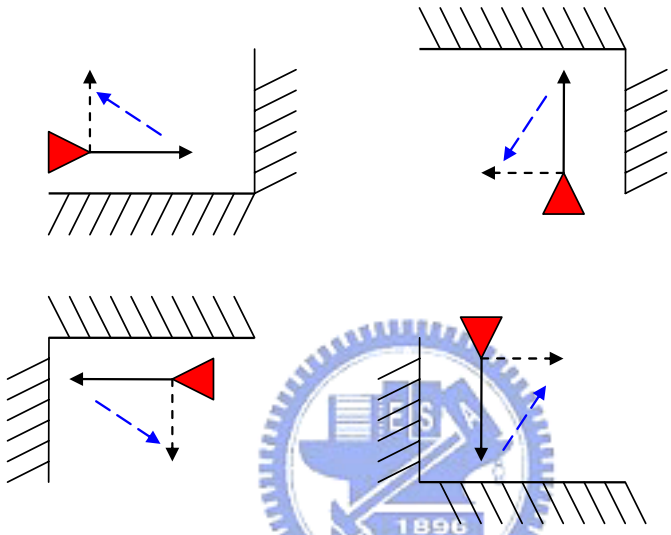


圖 3.12: 自動車避障之情形。(紅色三角型爲自動車，實線箭頭代表目前自動車前進的方向，虛線箭頭代表所要改變的方向)

步驟三: 當自動車的前方、左方及右方有障礙物時，則自動車旋轉 180 度。如圖 3.14 所示。

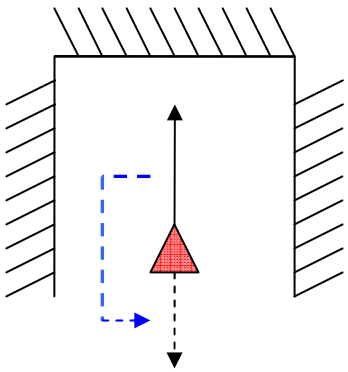
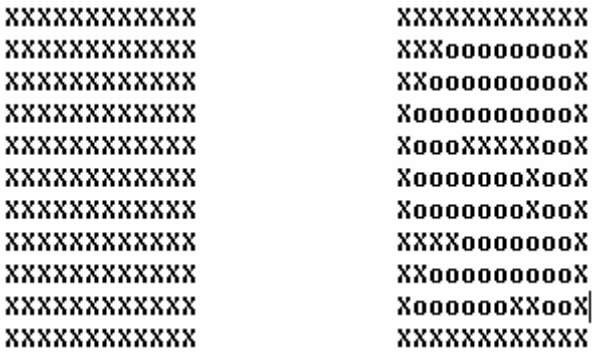


圖 3.13: 前、前左及右方皆爲障礙物的避障方法

製作場景地圖的一開始，我們先提供一個符合場景大小的初始矩陣，如圖 3.14(a)所示，其中 X 代表障礙物，接下來讓自動車利用沿著牆走的方式將場景完整地行駛一次，用 o 來代表車子通過的地方，也就是無障礙的區域，完成行駛完成之後就可得到完整的矩陣地圖，如圖 3.14(b)所示。



(a)初始場景地圖　　(b)已完成之場景地圖

圖 3.14: 場景地圖: (a)初始場景地圖和(b)已完成之場景地圖

有了上述的矩陣地圖之後，接下來我們介紹 A*演算法。A*演算法沿著從起始點 N_{init} 所產生的路徑(path)，在每個反覆運算過程的一開始，會有一些演算法已經拜訪過(Visited)的點。對每一個已拜訪過的點 N，都有一條或數條路徑連接起始點 N_{init} 和 N ，但 A*演算法只記憶最小成本的路徑[20]。這些路徑都是藉由指標存在樹狀結構 T 中，在 A*演算法中，每一個點 N 都可以經由成本函數來估算，從起始點 N_{init} 經過點 N ，到終點 N_{goal} 的最小成本路徑，成本函數的計算如下:

$$f(N) = g(N) + h(N) \tag{3.11}$$

其中， $g(N)$ 是 N_{init} 到 N 之間的最小成本， $h(N)$ 是 N_{goal} 到 N 之間所估算的最小成本。A*演算法的輸入包括了與 N 相鄰的節點 G、 N_{init} 、 N_{goal} 、 h 、 k ，而 k 是判斷兩點間成本之函數，所有在 G 中的點，一開始都是位拜訪(Unvisited)的狀態。A*演算法使用稱爲 Open list 的名單，藉由 f 的挑選，將 G 中的點儲存於其中。

A*演算法的過程如下:

步驟一: 開始:A*由起始點 N_{init} 開始，通常將 $g(N_{init})$ 設爲 0，並計算 $f(N_{init})$，並將起始點放入 Open list 中。

步驟二: 演算法接著從 Open list 中挑 $f(N)$ 最小的點出來，假如 Open list 是空的，則 N_{init} 到 N_{goal} 之間並無任何路徑，演算法結束，假如挑出來的點是 N_{goal} ，則演算法由樹狀結構 T 中重建從 N_{init} 到 N_{goal} 之間的路徑並傳回，演算法結束。

步驟三: 若挑出來的點不是 N_{goal} 且未被拜訪過(Unvisited)，將此點加入 Closed list 中，則對每個相鄰點，演算法用指標指向原本之點並存入 T 中，並把這些點放入 Open list 中，並且標上已拜訪(Visited)。

步驟四: 檢視目前節點的相鄰節點，若這些相鄰的節點不在 Open list 中且不在 Closed list 中也不爲障礙物，則將此點加入 Open list 中。

步驟五: 路徑完成後，從目標點依循母磚塊節點走回起始點，最後就會得到完整的路徑。

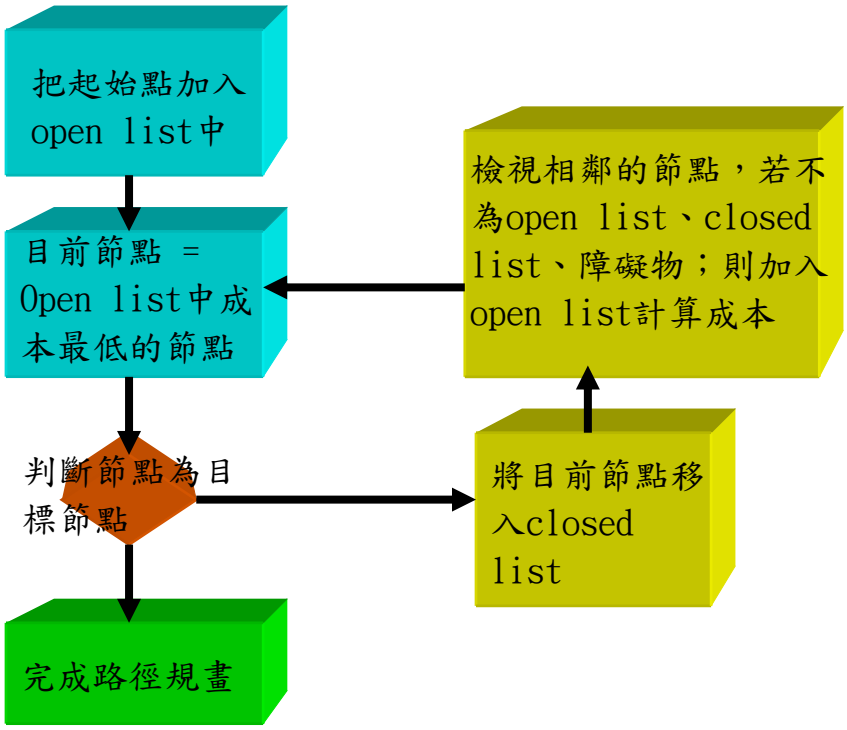


圖 3.15:A*演算法流程圖。

探索函數 $h(N)$ 估算 N_{goal} 到 N 之間最小成本。探索函數的選擇相當重要，因爲它影響到 A*演算法的結果，若是我們將 $h(N)$ 直接令爲 0，則 A*演算法就會變成 Dijkstra 的演算法，雖然 Dijkstra 的演算法可以保證永遠找到最佳路徑，但它的效率較差，事實上，若 $h(N)$ 永遠小於或等於 N_{goal} 到 N 之間的最小成本，則 A*演算法可以保證永遠找到最佳路徑，但是 $h(N)$ 愈小，A*演算法所需搜尋的點愈多，所以效率愈差，圖 3.15 爲整個演算法之流程圖。