第六章

實驗結果與討論

在介紹完系統的所有運作流程後，接下來我們將用實驗的方法來測試本論文所提出的運動模組是否有效。在這一章裡，我們首先會說明實驗的環境，以及一些相關細節的實作方法。之後我們將實作兩種不同互動機制的系統：虛擬角色操作系統及飛彈射擊遊戲，分別對它們進行測試，藉此來探討這個運動模組的實用性。這裡所要探討的部分，包括角色運動能力的可靠性、系統是否能達到即時效果以及這種方法的優缺點分析等。

6.1 實作與實驗環境

在系統的實作上，我們是以自行開發的遊戲引擎來進行測試。這個遊戲引擎是用C++程式語言來進行開發，開發工具為 Microsoft Visual Studio 2005。其中 3D 圖形介面的部份，是以 Microsoft DirectX SDK 所提供的函式庫來進行開發，其版本為 9.0c。而我們實驗裡所使用到的動作片段檔，是修改自 CMU Graphics Lab 動作擷取資料庫[6]裡所提供的動作片段檔，圖 6.1是我們所使用的動作圖。另外，在場景與角色的碰撞偵測部分，我們是用 AABB(Axis-Aligned Bounding Box)[8]演算法來進行實作，並以角色動作片段的每一個動作格為碰撞檢查的基本單位。動作混合(motion blending)的部分，我們則是使用時間調整(Time Warping) 的方法來進行實作[31]。實驗部份，我們執行測試的機器 CPU 為 P4 3.0G，RAM 為 1.0GB，顯示晶片則是 NVIDIA GeForce 6600。

編號	動作名稱	格數	下一個動作
1	Stop to Walk	46	2, 3, 8, 9, 10
2	Walk forward 1	21	3, 4
3	Walk forward 2	26	2, 5, 8, 9, 10
4	Walk_forward1 to Stop	42	1, 6, 7, 13, 15, 19, 20, 21, 22
5	Walk_forward2 to Stop	47	1, 6, 7, 13, 15, 19, 20, 21, 22
6	Turn left in place	54	1, 6, 7, 13, 15, 19, 20, 21, 22
7	Turn right in place	44	1, 6, 7, 13, 15, 19, 20, 21, 22
8	Walk left 45	47	2, 5, 8, 9, 10
9	Walk right 45	50	2, 5, 8, 9, 10
10	Walk to Craw	57	11, 12, 14
11	Craw forward	55	11, 12, 14
12	Craw to Walk	35	3, 4
13	Stand	20	1, 6, 7, 13, 15, 19, 20, 21, 22
14	Craw	30	11, 12, 14
15	Inverse of Walk_forward1 to Stop	42	16
16	Inverse of Walk_forward1	21	17, 18
17	Inverse of Walk_forward2	26	16
18	Inverse of Stop to Walk	46	1, 6, 7, 13, 15, 19, 20, 21, 22
19	Walk sideway left	40	1, 6, 7, 13, 15, 19, 20, 21, 22
20	Walk sideway right	40	1, 6, 7, 13, 15, 19, 20, 21, 22
21	Hand spring back	60	1, 6, 7, 13, 15, 19, 20, 21, 22
22	Flip turn 90	128	1, 6, 7, 13, 15, 19, 20, 21, 22

圖 6.1：實驗裡所使用的動作圖

圖 6.2是我們實作 AABB 演算法的執行範例。在我們的演算法裡，障礙物中的每個元件都會定義出一個相對的 Bounding Box，用來代表這些元件的幾何資訊。我們將判斷這些 Bounding Box 之間是否有交集，藉此來進行碰撞偵測的檢查。例如在我們的實驗場景裡，房屋障礙物會有十個 Bounding Box，分別用來代表屋頂、門、四根柱子和四面牆壁的幾何資訊。虛擬角色會有十六個 Bounding Box，分別用來代表手、腳、頭部…等十六個身體部位的幾何資訊。當我們要進行角色與房屋的碰撞偵測檢查時，我們會檢查房屋的十個 Bounding Box 是否會跟角色的十六個 Bounding Box 有交集，有交集便代表有碰撞產生，否則便代表沒有碰撞產生。這種碰撞偵測方法有著速度快及容易實作的優點，缺點則是不夠精細。

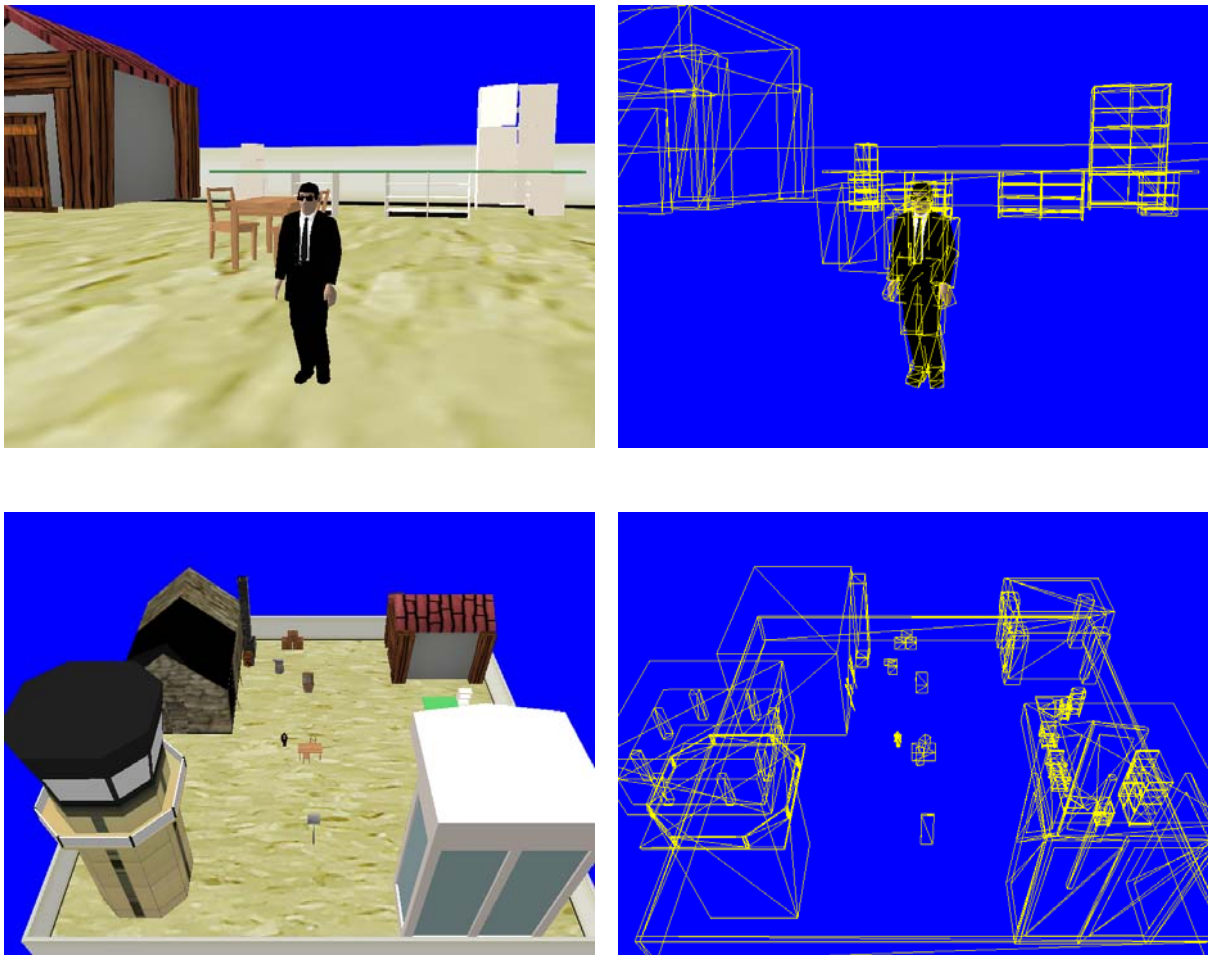


圖 6.2：場景和虛擬角色的 Bounding Box

圖 6.3是描述本系統動作混合方法的簡單示意圖。在我們所使用的方法裡，系統會將目前角色動片段的後 n 個動作格跟下個動作片段的前 n 個動作格進行內插疊合，藉此產生出新的動作格。如圖所示，當角色目前的動作片段進行到最後兩個動作格時，它的混合比重會由 1.0 逐漸降低到 0.0，而下個動作片段的混合比重則會由 0.0 逐漸增加到 1.0，藉此產生出內插疊合的效果。

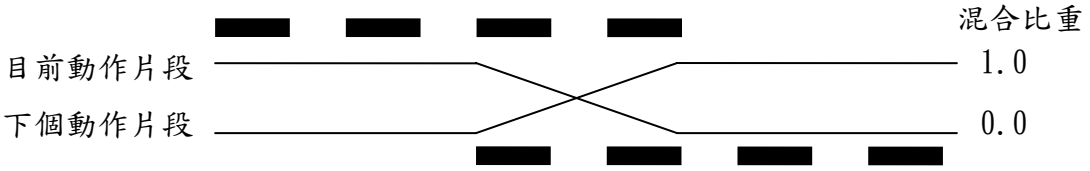


圖 6.3：動作混合方式的示意圖

6.2 角色操作系統

6.2.1 系統設計

在這個系統的使用者輸入部份，我們會允許使用者輸入四個簡單的操作命令：分別是前進、往左、往右和後退。我們希望場景中的虛擬角色可以根據使用者的這些命令來進行移動，並配合場景中障礙物的資訊，自行做出相對適合的反應動作。例如當角色的前方有障礙物的存在，而使用者下達了前進的命令時。這個角色會自行判斷能否穿越這個障礙物，並選擇出各種可行的反應動作來達成使用者的操作命令。例如用蹲下的方式來直線穿越障礙物。或是先閃避過障礙物，再用修正路線的方式來達成使用者的操作命令(如圖 6.4)。這個系統的設計目的，是希望能提供給使用者一個簡單的操作介面。這個操作介面可以簡化使用者的輸入，讓使用者在不需要考慮場景中各種複雜狀況的前提下，靈活的操作場景中的虛擬角色。

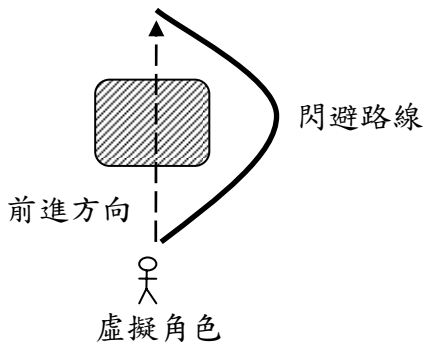


圖 6.4：用閃避的方式來達成使用者的操作命令(前進)

- **擴展優先權的計算**

在擴展優先權的部份，我們是以目標吻合度搭配優先權老化的方式來進行計算上的實作。這裡的目標指的是角色的移動目標，也就是使用者所下達的移動命令。而之所以搭配優先權老化的原因，是因為我們認為使用者在進行角色動作的操作時，所下達的命

令會因為各種狀況而隨時有可能改變。在這種前提下，讓系統思考得太遠並不是件必要的事。因此我們便利用搭配優先權老化的方式，來讓角色不要思考得太遠。

下面是我們所定義的擴展優先權計算公式：

$$\vec{v} = P_{node} - P_{avatar} \tag{7.1}$$

$$M_{position} = \vec{u} \cdot \vec{v} \tag{7.2}$$

$$M_{command} = M_{position} \times \mu_{motion} \tag{7.3}$$

$$M_{path} = \prod_{Ni \in path} M_{command} \tag{7.4}$$

$$M_{aging} = 1.0 - \mu_{aging} \times (t_{node} - t_{avatar}) \tag{7.5}$$

$$W_{expand} = M_{path} \times M_{aging} \tag{7.6}$$

在這個公式裡，$\vec{v}$ 為角色的移動向量，是根據角色目前位置 P_{avatar} 及節點所代表的位置 P_{node} 所計算出來的一個方向向量，$\vec{u}$ 則是使用者指令所代表的移動方向向量，它們的關係如圖 6.5所示。公式裡的 $M_{command}$ 代表節點組態對使用者指令的吻合度，是根據節點位置對使用者指令的吻合度 $M_{position}$ 及角色對節點動作的喜好度 μ_{motion} 所計算出來的。其中 $M_{position}$ 是取 $\vec{u}$ 和 $\vec{v}$ 的內積來代表它的值。μ_{motion} 則是從我們事先定義好的角色動作喜好表中取得，它的值是以動作的複雜程度來決定，全部動作都介於 0.0 到 1.0 之間，其中動作愈複雜，喜好度就愈低。例如在我們的程式裡，跳躍是一種比較複雜的動作，所以我們會給他比較低的偏好值。將這個值加入計算公式，是為了讓系統能將角色的動作喜好反應到動作樹的擴展裡。公式中的 M_{path} 代表角色從目前組態移動到節點組態時，這條移動路徑對使用者指令的吻合度。這裡我們是找出節點到根節點路徑裡的所有節點，並將它們的節點組態吻合度 $M_{command}$ 相乘，藉此計算出 M_{path} 的值。M_{aging} 則是節點的優先權老化值，是根據節點組態裡的系統時間 t_{node}、角色目前組態裡的系統時間 t_{avatar} 及設計者事先定義的老化速率值 μ_{aging} 所計算出來的。最後將 M_{path} 和 M_{aging} 相乘，我們便可以計算出節點的擴展優先權 W_{expand}。

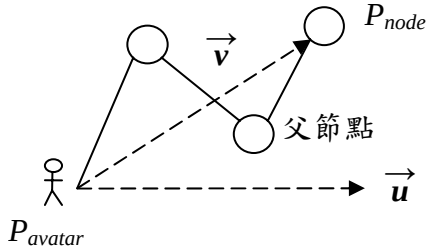


圖 6.5：擴展優先權的計算公式示意圖

- **動作樹的選擇及分析**

在動作樹的分析及選擇部份，我們是以搜尋最佳路徑的方式來進行實作。我們評價一條路徑的方法，是找出屬於這條路徑的所有節點，並將它們的擴展優先權全部加總起來，所得到的值便是這條路徑的品質。它的值愈大代表品質愈佳，角色就會愈優先選擇這條路徑。圖 6.6的 Evaluate_Node 程序是我們在進行角色動作選擇時，系統用來分析節點優劣的演算法。這個演算法是用由下往上(bottom-up)的方式來分析動作樹的所有節點，從中找出最佳的路徑。這裡我們是以遞迴的方式來對這個演算法進行實作。

```
Algorithm: Evaluate_Node(Node)

Input. A node of feasible motion tree.
Output. The worth of node.

1.  // 如果有碰撞發生,回傳評價爲零.
2.  if Node is in collision then
3.    return 0;
4.  end if
5.  // 否則開始計算節點的評價
6.  NodeWorth = Node.Expand_Priority;
7.  BestChildWorth = 0;
8.  // 搜尋Node的所有子節點,找出其中的最佳評價.
9.  for i = 1 to Node.ChildNum
10. begin
11.   ChildWorth = Evaluate_Node(node.Child[i]);
12.   if ChildWorth > BestChildWorth then
13.     BestChildWorth = ChildWorth;
14.   end if
15. end for
16. // 回傳Node的評價
17. NodeWorth = NodeWorth + BestChildWorth;
18. return NodeWorth;
```

圖 6.6：分析節點優劣的演算法

6.2.2 實驗一：實用性測試

- **實驗設計**

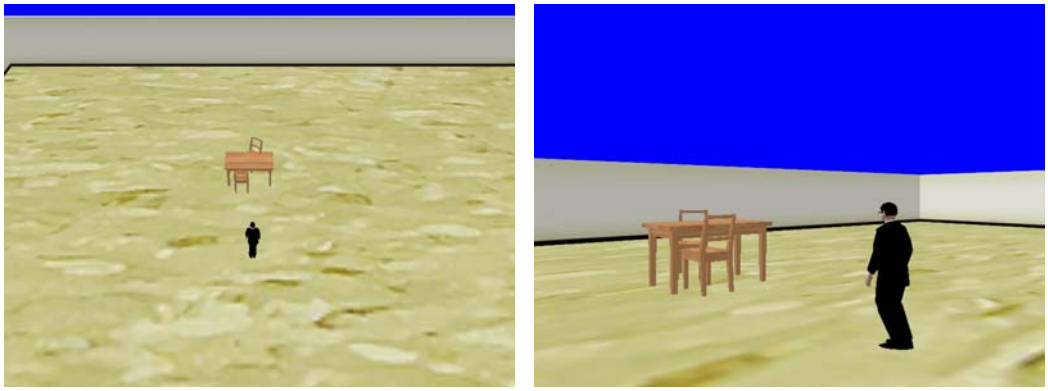
這裡我們設計了四個簡單的場景(圖 6.7)來測試這個運動模組的實用性，這些場景中的障礙物都會擺設在角色前進方向的路上。其中場景 A 是一張桌椅，角色只能藉由繞道的方式來閃避這張桌椅。場景 B 是一個旋轉門，角色可以從其中的一個門進入，並從另一個門出來。場景 C 是由兩個大箱子所組成的夾縫，角色必須用側身橫移的方式才能通過這個夾縫。場景 D 是一個辦公桌，角色可以藉由蹲下前進的方式來直線通過辦公桌，也可以用繞道的方式，從旁邊的缺口通過。在實驗裡，系統將輸入前進的命令來讓角色進行移動。我們將觀察角色的移動過程，確認角色是否能如我們所預期的閃避過場景中的障礙物，並達到使用者的前進命令要求。另外為了能確實達到即時的效果，我們會將系統的畫面刷新率(Frame Per Second, FPS)鎖定在約 86 畫格/秒之間，這也相當於是讓角色有約 0.01 秒/畫格的角色動作預測時間。

- **實驗結果與討論**

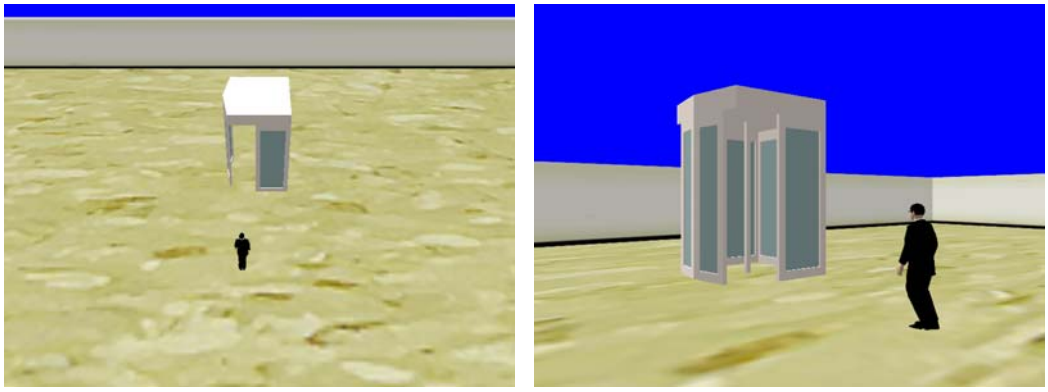
圖 6.8～圖 6.12分別為角色通過 ABCD 四個場景的連續動作圖。從這些連續動作圖裡，我們可以看出角色的確能如我們所預期般的閃避過場景中的障礙物。並且在閃避的過程裡，同時顧及到使用者的命令要求，持續往前移動。

圖 6.13是角色在通過場景 D 的障礙物時，系統所擴展出來的可行動作樹及它的變化過程。從這張圖裡，我們可以看出角色在進行動作的選擇時，動作樹裡會有很多的路徑可以讓角色進行選擇。而角色卻能從這麼多路徑裡，選擇出蹲下往前走的最佳路徑，可見我們的動作樹分析方法的確是夠實用的。

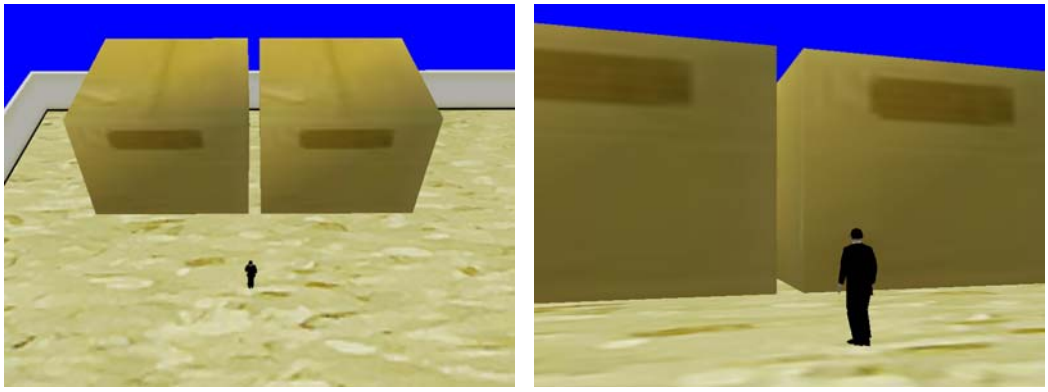
(A)



(B)



(C)



(D)

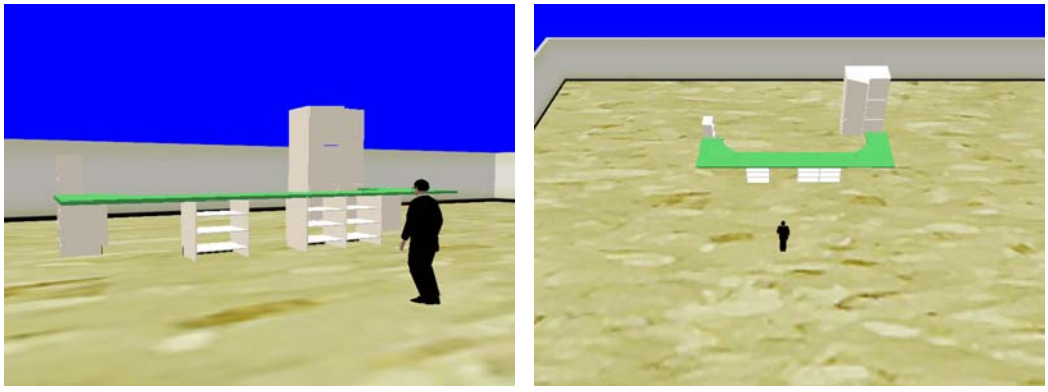


圖 6.7：實用性測試實驗場景

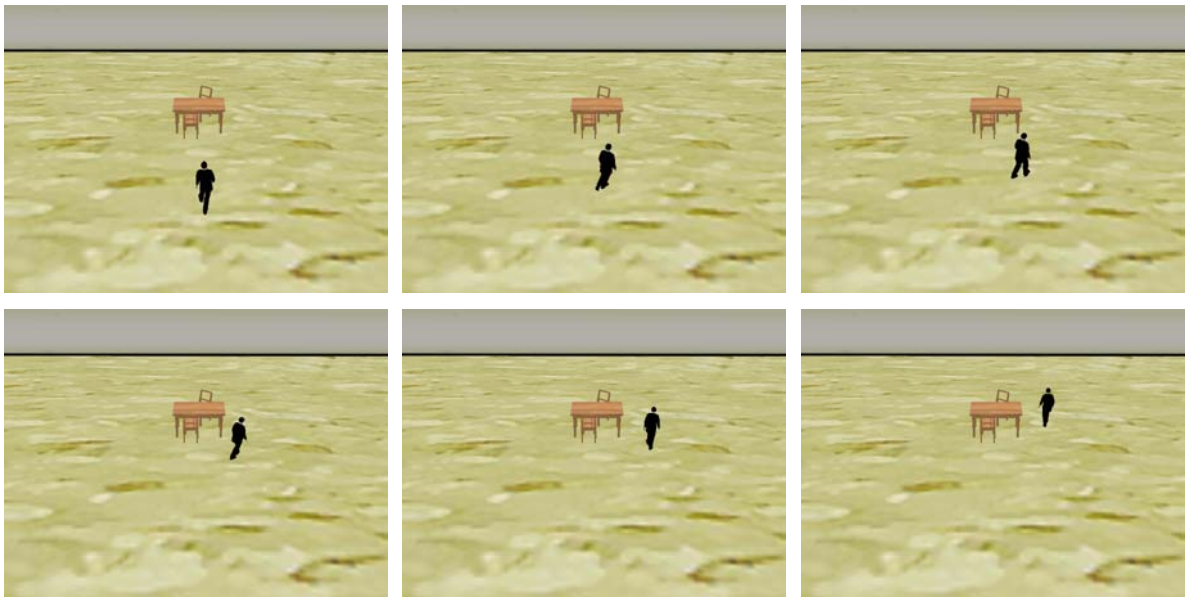


圖 6.8：角色通過場景 A

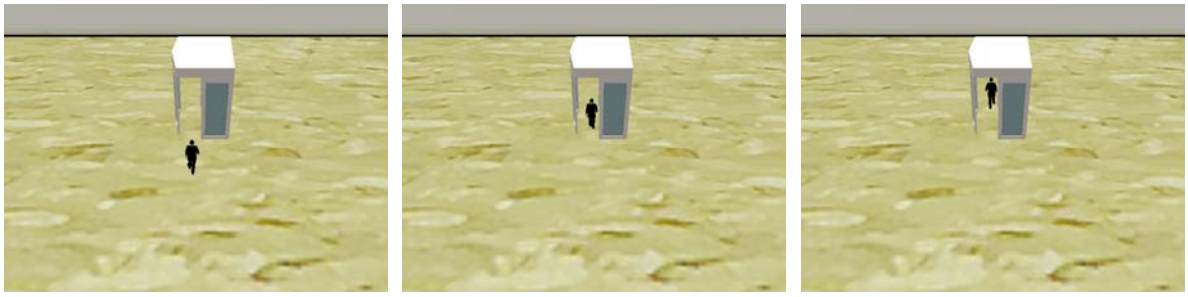


圖 6.9：角色通過場景 B

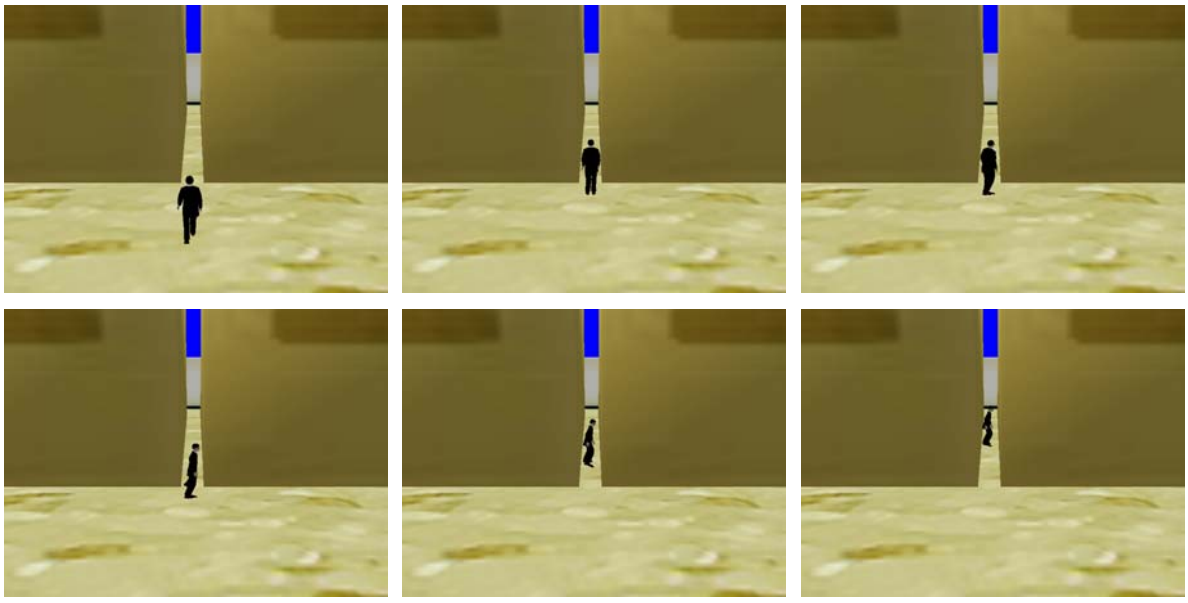


圖 6.10：角色通過場景 C

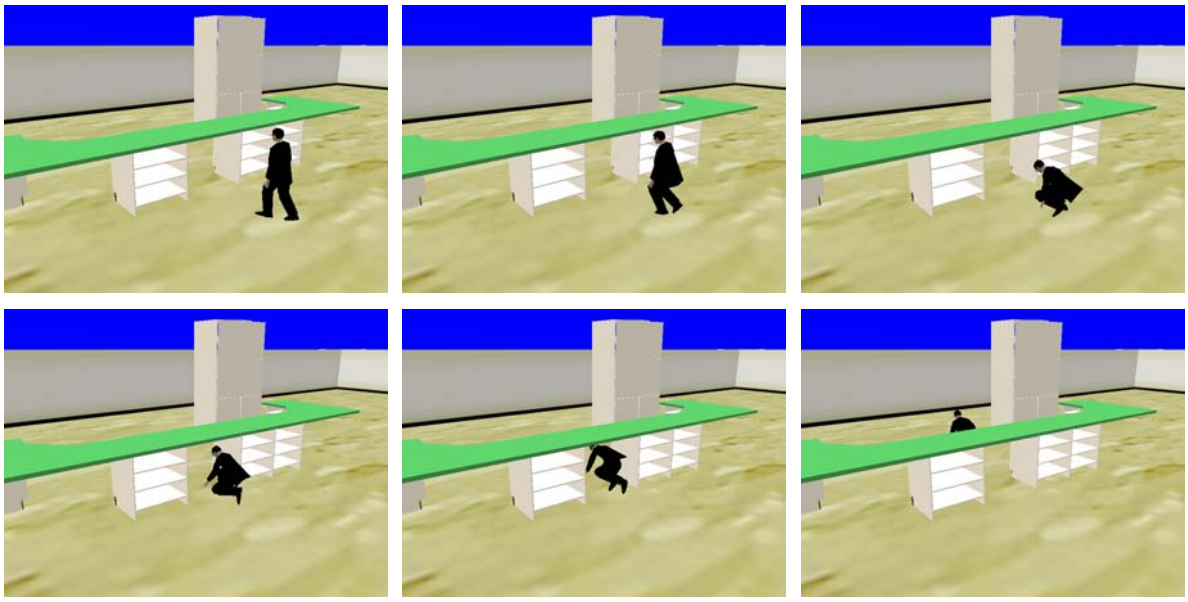


圖 6.11：角色通過場景 D (視角 1)

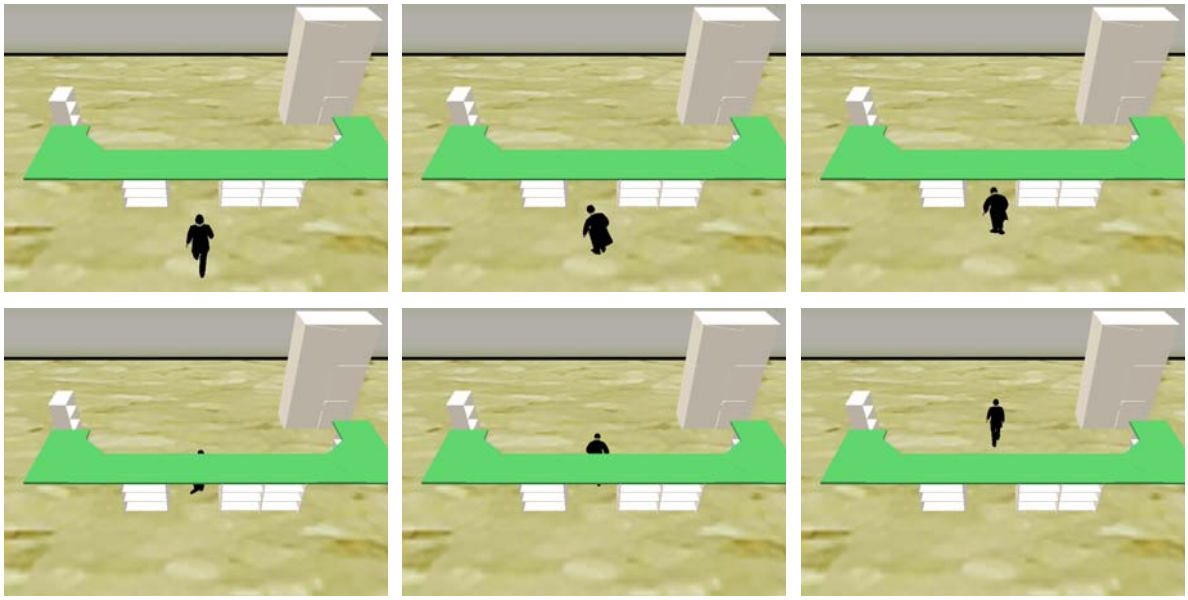


圖 6.12：角色通過場景 D (視角 2)

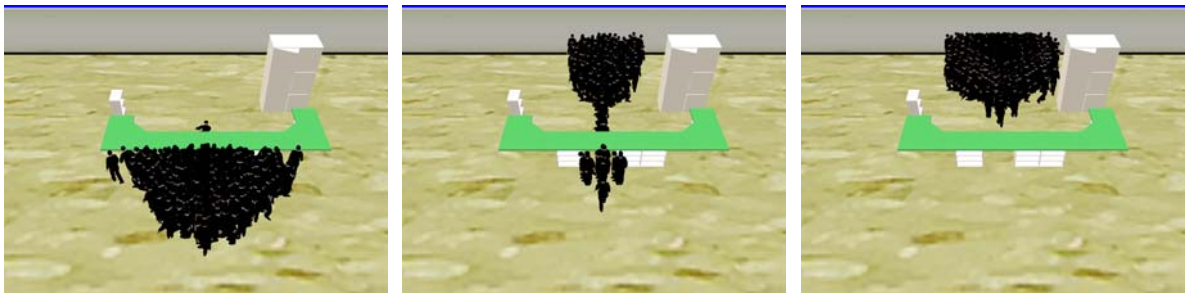


圖 6.13：角色在場景 D 的動作樹擴展範例

6.2.3 實驗二：不同擴展策略的比較

- **實驗設計**

這裡我們將針對三種不同的動作樹擴展策略進行比較。第一種策略是以目標吻合度搭配優先權老化的擴展策略，也就是我們為這個應用所設計的動作樹擴展策略。第二種策略是單純只以目標吻合度來進行動作樹擴展的策略，這種策略可以拿來跟第一種策略進行比較，藉此確認搭配優先權老化的必要性。第三種策略是寬度優先的擴展策略，這種策略可以用來跟目標吻合度的擴展策略進行比較，藉此確認我們所定義的目標吻合度計算公式，是否能提供較佳的預測效果。其中在預測效果的比較部份，我們認為愈好的預測方法，可以有著愈高的預測準確度。而有著愈高的精確度，代表角色每次進行完動作選擇時，可行動作樹的剩餘節點數目也會愈多。因此在實驗裡，我們便是以這個數目的平均值來評估角色的預測能力。

這裡我們設計了一個簡單的迷宮場景(如圖 6.14)來測試角色的預測能力。在這個場景裡，虛擬角色會被放置在左上角的起點位置，使用者必須依照固定的移動路徑來控制虛擬角色，讓這個虛擬角色可以移動到右下角的終點。而在這個移動過程裡，系統會在角色每次在進行完動作選擇時，記錄可行動作樹裡被刪除的節點數目以及剩餘的節點數目。並在移動結束後，計算出它們的平均值，藉此來評估上述擴展策略的預測效果。

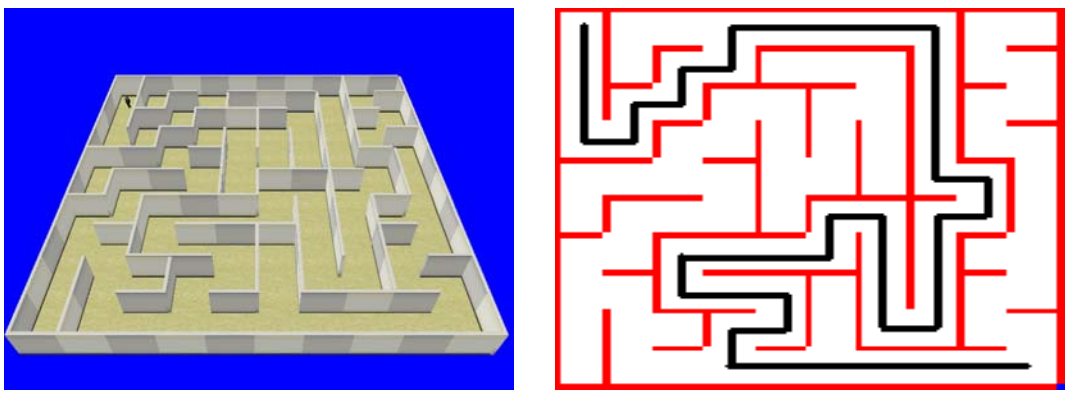


圖 6.14：實驗用的迷宮場景及其路徑

- **實驗結果與討論**

實驗結果如圖 6.15，因為本實驗是以手動的方式來進行測試，所以為了降低誤差值，圖中的數據都是取四次實驗的平均值來完成。其中分析圖表裡的 Type1 是以目標吻合度搭配優先權老化的擴展策略，Type2 是單純只以目標吻合度來進行動作樹擴展的策略，Type3 則是寬度優先的擴展策略。

從這個實驗結果裡，我們可以看出 Type1 的方法會有著比較高的剩餘節點數目，但和 Type2 的方法比較起來，差距並不會太明顯。也就是說搭配優先權老化的計算方式，雖然能對預測結果有所幫助，但並不會有著特別明顯的效能提升。另外在比較 Type2 和 Type3 的結果後，我們可以看出 Type2 的剩餘節點數目會明顯多於 Type3。也就是說我們所設計的目標吻合度計算公式，的確能提拱給角色不錯的預測效果。另外，從實驗數據的比較圖裡，我們也可以看出當角色思考時間愈多，剩餘節點數目也會相對的變多，而且它幾乎是以線性的比例來成長，

這個實驗方法目前還有很多問題是無法從結果裡顯示出來的。例如當系統在用寬度優先的擴展方法來進行動作預測時，雖然它會讓可行動作樹有著明顯較少的剩餘節點數目。但我們並無法從這個結果裡，推論出角色是否就會因此而做出明顯比較差的動作選擇。相同的問題，雖然目標吻合度的擴展方法，的確能夠讓可行動作樹有著明顯較多的剩餘節點數目。但我們也還是無法從這個結果裡，推論出角色是否就會因此而做出明顯比較好的動作選擇。如何判斷角色所選擇的動作是否為合適的動作，是我們未來必須進一步思考的問題。

角色思考時間(秒)	平均的剩餘節點數
0.010	1316.45
0.015	5978.86
0.020	11640.64
0.025	14597.12
0.030	20678.89

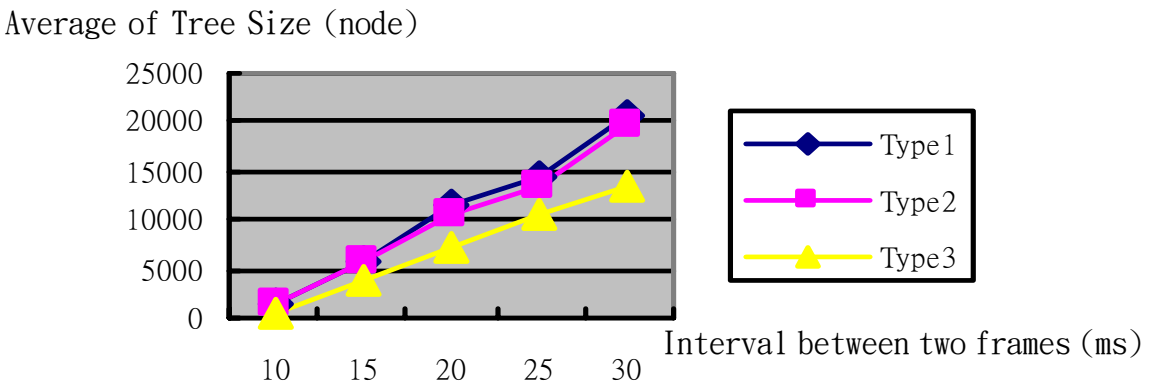
(A) 目標吻合度搭配優先權老化

角色思考時間(秒)	平均的剩餘節點數
0.010	1221.52
0.015	5724.47
0.020	10523.31
0.025	13527.54
0.030	19857.47

(B) 目標吻合度

角色思考時間(秒)	平均的剩餘節點數
0.010	672.78
0.015	3799.33
0.020	7022.04
0.025	10425.97
0.030	13343.69

(C) 寬度優先



(D) 分析圖表

圖 6.15：比較不同擴展策略的實驗數據

6.3 飛彈射擊遊戲

6.3.1 系統設計

在這個系統的使用者輸入部份，我們會讓使用者以第一人稱的視角的方式來和場景中的虛擬角色進行互動。它的整個操作介面就像是在玩目前市面上常見的第一人稱射擊遊戲一樣，使用者可以用前進、往左、往右和後退等功能鍵來控制視角的移動，並用滑鼠的左鍵來對角色發動飛彈攻擊。其中每當使用者點擊滑鼠的左鍵時，場景裡就會新增一個動態的飛彈物體。這個飛彈會以等速移動的方式往使用者點擊的方向飛去，並在和場景中的物體產生碰撞時，產生爆炸並從場景裡面消失。其中如果飛彈是因為擊中角色而產生爆炸時，那麼角色的血量值就會因此而減少。這個系統的設計目的，便是希望角色能在不跟場景產生碰撞的情況下，盡量閃躲這些飛彈。

- **擴展優先權的計算**

在擴展優先權的部份，我們是以目標吻合度搭配深度優先的方式來進行實作。這裡的飛彈指的是場景中的飛彈及目前使用者所處的位置，是角色必須警戒的對象。而之所以搭配深度優先的原因，是因為我們認為在閃躲飛彈的問題裡，盡可能找出一條最遠的活路，應該會有著比較高的生存率。下面是計算擴展優先權的公式：

$$R_{AvoidUser} = \text{Min}(1.0, D_{user} / D_{max}) \quad (7.7)$$

$$R_{AvoidRocket} = \text{Min}(1.0, D_{rocket} / D_{max}) \quad (7.8)$$

$$R_{Avoid} = \text{Min}(R_{AvoidUser}, R_{AvoidRocket1} \sim R_{AvoidRocketN}) \quad (7.9)$$

$$M_{command} = R_{Avoid} \times \mu_{motion} \quad (7.10)$$

$$M_{path} = \prod_{Ni \in path} M_{command} \quad (7.11)$$

$$M_{depth} = 1.0 + \mu_{depth} \times (t_{node} - t_{avatar}) \quad (7.12)$$

$$M_{blood} = 1.0 - \mu_{blood} \times (B_{avatar} - B_{node}) \quad (7.13)$$

$$W_{expand} = M_{path} \times M_{depth} \times M_{blood} \quad (7.14)$$

在這個公式裡，$R_{AvoidUser}$ 為節點對使用者的閃避率，值介於 0.0 到 1.0 之間，是根據 D_{user} 和 D_{max} 所計算出來，計算出來的值愈大，代表角色愈有可能閃避過使用者未來所可能發射的飛彈。其中 D_{user} 是使用者視線到節點位置之間的最短距離，D_{max} 是角色的最大警戒範圍，它們的關係如圖 6.16所示。$R_{AvoidRocket}$ 為節點對飛彈的閃避率，值同樣介於 0.0 到 1.0 之間，是根據 D_{rocket} 和 D_{max} 所計算出來，其中 D_{rocket} 是飛彈移動向量到節點位置之間的最短距離，它們的關係如圖 6.17所示。公式中的 R_{Avoid} 為節點位置的閃避率，它是計算出節點位置對場景中所有飛彈的閃避率 $R_{AvoidRocket1}$～$R_{AvoidRocketN}$ 及節點位置對使用者的閃避率 $R_{AvoidUser}$，從中取出最小值來當成節點位置的閃避率，這相當於是把角色在節點位置的最壞情況當成角色的閃避機率。之後根據 R_{Avoid} 及 μ_{motion}，我們可以計算出節點組態對使用者指令的吻合度 $M_{command}$，並根據 $M_{command}$ 進一步計算出角色從目前組態移動到節點組態的路徑吻合度 M_{path}。最後將 M_{path} 乘上節點的深度優先權 M_{depth} 及血量優先權 M_{blood}，所得到的值便是節點的擴展優先權 W_{expand}。其中在計算 M_{depth} 的部份，$t_{node} - t_{avatar}$ 代表角色從目前組態移動到節點組態的時間差，μ_{depth} 則是由設計者事先定義的深度變化權重，這個公式會讓 M_{depth} 隨著深度的增加而變大。另外在計算 M_{blood} 的部份，$B_{avatar} - B_{node}$ 代表角色從目前組態移動到節點組態的損血量，μ_{blood} 則是由設計者事先定義的血量變化權重，這個公式會讓 M_{blood} 隨著損血量的增加而變小。

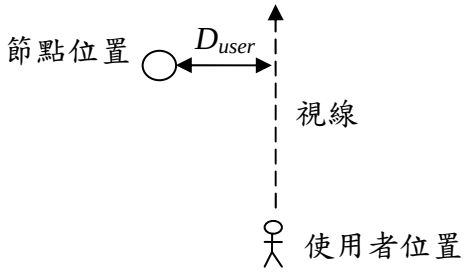


圖 6.16：節點位置對使用者的閃避率

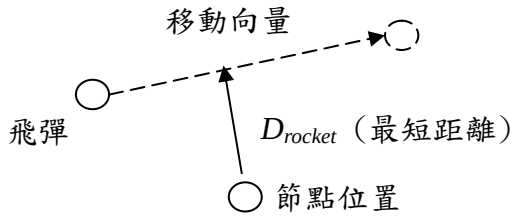


圖 6.17：節點位置對飛彈的閃避率

圖 6.18是我們用來計算節點位置對飛彈閃避率的演算法。在這個演算法裡，我們會計算出飛彈從起點移動到終點的移動向量 v 及節點位置到飛彈起點位置的移動向量 w，並根據這兩個向量來找出飛彈移動過程中離節點位置最近的飛彈位置 ShortestPoint。之後計算出這個 ShortestPoint 跟節點之間的距離 ShortestDistance，並利用這個距離來計算出角色對飛彈的閃避率。圖 6.19是用來計算節點位置對使用者閃避率的演算法，這裡會先計算出節點位置跟使用者位置之間的距離 dist，及計算出節點位置跟使用者視角的弧度差 rad，之後根據這兩個值來計算出當使用者發射飛彈時，這個飛彈跟角色之間的可能最短距離 d。並利用這個距離來算出角色對使用者的閃避率。

```
Algorithm: Evaluate_AvoidRocket(Node, Rocket)

Input. A node of feasible motion tree, and a rocket.
Output. The possibility to avoid rocket.

1.  // 計算節點位置跟飛彈的最短距離.
2.  Vector v = Rocket.EndPosition - Rocket.StartPosition;
3.  Vector w = Node.Position - Rocket.StartPosition;
4.  c1 = dot(w,v);
5.  c2 = dot(v,v);
6.  b = c1 / c2;
7.  ShortestPoint = Rocket.StartPosition + b * v;
8.  ShortestDistance = Distance(Node.Position,ShortestPoint);
9.  // 回傳對飛彈的閃避率
9.  if ShortDistance > MinDistance then
10.   return 1.0;
11. else
12.   return ShortestDistance / MinDistance
13. end if
```

圖 6.18：計算節點位置對飛彈閃避率的演算法

```
Algorithm: Evaluate_AvoidUserAttack(Node, Camera)

Input. A node of feasible motion tree, and user camera.
Output. The possibility to avoid user attack.

1.  // 計算節點位置跟玩家位置的距離.
2.  dist = Distance(Camera.Position, Node.Position);
3.  // 計算節點位置跟玩家視角的弧度
4.  Vector v1 = Camera.Position - Node.Position;
5.  Vector v2 = Camera.ViewVector;
6.  float rad = radian(v1,v2);
7.  // 根據弧度和距離計算出d
8.  d = dist * rad;
9.  // 回傳對未來飛彈的閃避率
10.  if d > MinDistance then
11.    return 1.0;
12. else
13.    return d / MinDistance
14. end if
```

圖 6.19：計算節點位置對使用者閃避率的演算法

- **動作樹的選擇及分析**

在動作樹的分析及選擇部份，我們也還是以搜尋最佳路徑的方式來進行實作。這裡我們將分兩個步驟來評價兩條路徑的品質優劣。第一個步驟是先找出代表這兩條路徑的葉節點，比較這兩個葉節點的血量值，判斷其中那個血量值會比較高。較高的血量值代表當角色走完這條路徑時，他所會受到的損害將會比較小，因此這條路徑就會有著比較高的路徑品質。而如果兩條路徑的血量值是一樣大時，我們便進入步驟二，改以閃避率來評價這兩條路徑的品質。在步驟二裡，我們會找出路徑裡的所有節點，並從這些節點裡找出閃避率最小的值來代表整條路徑的閃避率。這個值代表當角色在走這條路徑時，角色所可能會遇到的最遭狀況，它的值愈大代表路徑的品質愈高。我們將比較兩條路徑的閃避率，選出其中閃避率較高的路徑為最佳路徑。

另外，這裡我們還提出了一種以搜尋最佳子樹來分析動作樹方法。這種方法同樣也是分成兩個步驟來評價兩條路徑的品質優劣，並採用類似的方法來評價子樹。其中第一個步驟是找出子樹裡的所有葉節點，取它們血量的平均值來當成子樹的品質，值愈高代

表品質愈好。而如果這個值是相同的，那麼同樣也是進入步驟二，改以閃避率來評價這兩條子樹的品質。這個步驟是找出子樹裡的所有節點，計算它們閃避率的平均值來當成子樹的品質，值愈高代表品質愈高。

在實驗裡，我們將統一採用搜尋最佳路徑的方式來進行可行動作樹的分析及選擇，因為這是我們認為在這個應用裡效果會最好的方法。我們將在之後的 6.3.5 對這兩種方進行實驗及比較，確認我們的想法是否正確。

6.3.2 實驗一：效用測試

- **實驗設計**

這裡我們的實驗方法是把角色和使用者放置在一個有障礙物存在的場景中，讓使用者可以用第一人稱的視角來對這個角色發動飛彈攻擊。整個實驗過程就像是在玩目前市面上常見的第一人稱射擊遊戲。我們在這個遊戲的過程裡，記錄使用者發射飛彈的位置以及角色被使用者飛彈所擊中的比率等數據，藉此來評估這個系統在遊戲互動上所能發揮的效用。圖 6.20是我們的實驗場景，其中左圖是以第一人稱視角俯瞰整個場景的畫面，右圖是以第三人稱視角和角色進行互動的畫面。

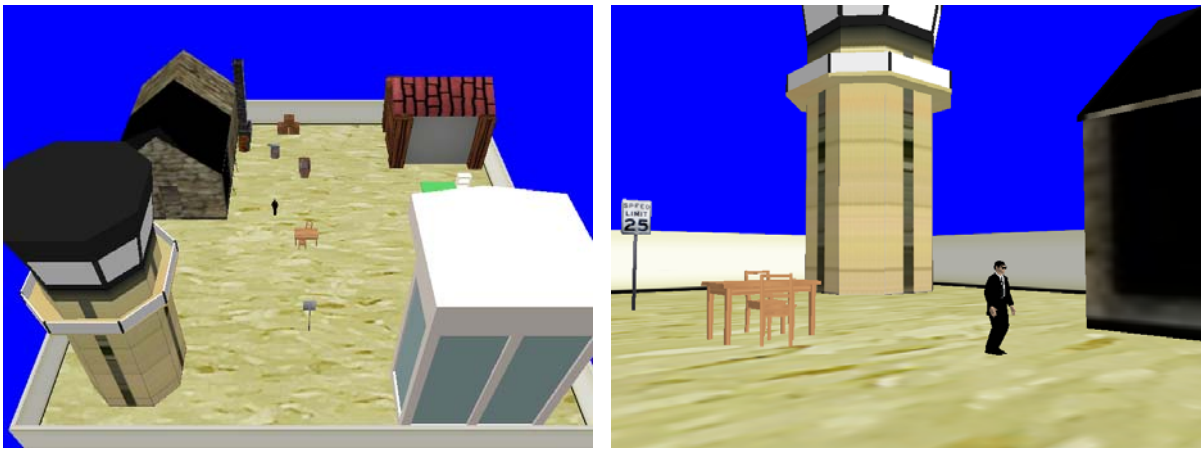


圖 6.20：效用測試實驗場景

● 實驗結果與討論

圖 6.21是我們的實驗結果，這個圖表記錄了使用者在不同距離發射的飛彈，它們所能命中角色的機率。其中不同顏色的線條，代表不同的時間預算，也就是系統在兩個畫格之間所能用來預測角色動作的時間。另外在這個實驗裡，系統限定飛彈的移動速度為 25 m/s。我們可以根據這個飛彈速度和使用者的發射距離，來推算出當使用者發射出飛彈後，角色能有多少時間來對這個飛彈進行反應。

從這個實驗結果裡，我們可以看出當發射距離愈短，飛彈能命中角色的機率就會愈高。其中當飛彈的發射距離為 40m 時，飛彈命中角色的機率不到 20%。也就是我們只要給角色 1.6 秒的反應時間，角色就能有著很高的閃避率。而就算將發射距離調整為 10m，使角色對飛彈反應的時間變成約 0.4 秒，角色也仍然會有著可以被接受的閃避率。這證明了我們的系統的確能提供給角色不錯的閃避能力。另外從這個圖表裡，我們也可以看出當時間預算愈多，系統所能提供給角色的閃避能力也會愈高。但它所能造成的效能差別，只有在約 5m～25m 之間的距離才會有著明顯的不同。其它太近或太遠距離的飛彈，時間預算所能造成的效能影響會非常有限。而從這個實驗裡的結果來看，5ms 時間預算所能提供的閃避能力，在一般的遊戲應用裡，應該已經足夠被使用者接受了。

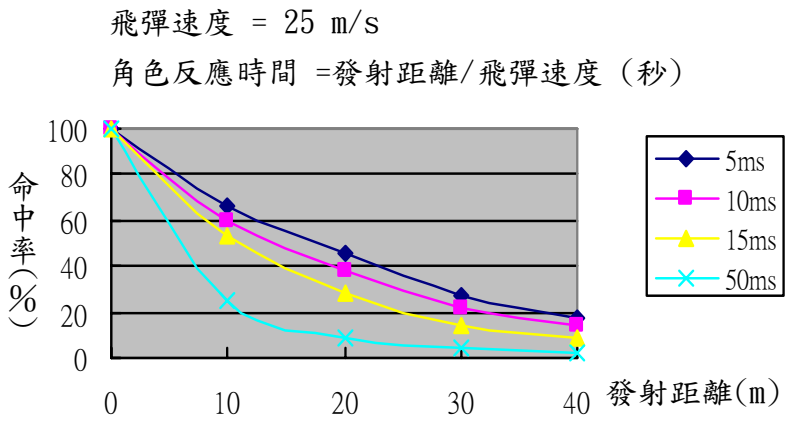


圖 6.21：發射距離和時間預算對命中率的影響

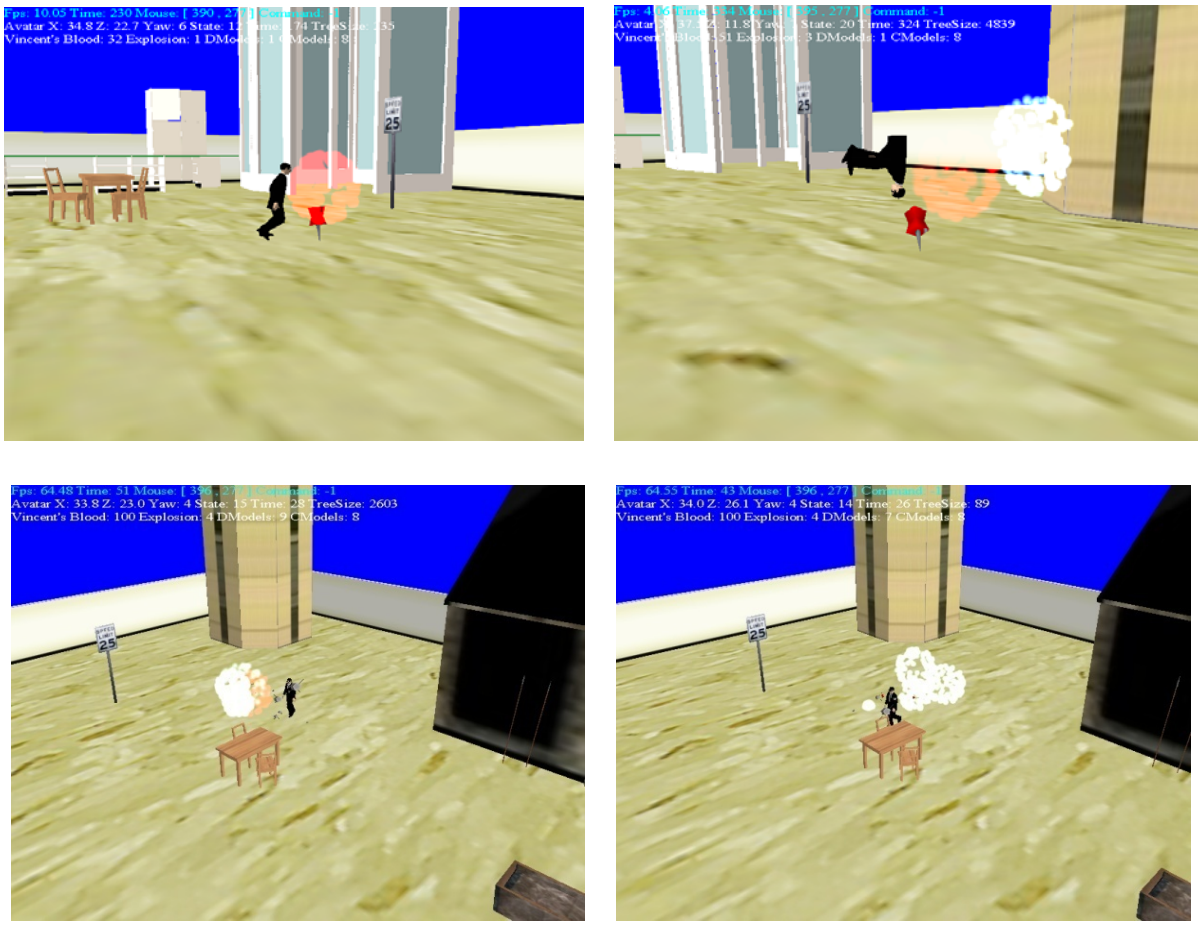


圖 6.22：效用測試實驗過程畫面：上圖為第一人稱視角，下圖為第三人稱視角。

6.3.3 實驗二：不同射擊條件測試

- **實驗設計**

在這個實驗裡，我們將檢查不同射擊條件會對系統效用產生的影響。這裡為了減少人為操作的因素，我們將改用隨機飛彈來測試角色的閃避能力。我們的實驗方法是把角色放置在一個空曠的場景中，讓系統對角色發動隨機且連續的飛彈攻擊。這些飛彈會因為擊中角色或是超過限定的存活時間，而產生爆炸並消失。我們將以各種不同的隨機飛彈參數來進行實驗，並觀察角色對這些隨機飛彈的閃躲率，比較它們之間的差異性。

圖 6.23是實驗裡加入隨機飛彈的方法。在這個方法裡，系統會先隨機產生一個飛彈的發射向量。之後根據角色的位置和這個發射向量，來計算出飛彈的發射位置和它的移動速度，並根據這些資訊來將飛彈加入場景中。這種方法因為是根據角色的位置來決定飛彈的發射點，所以如果角色沒有對這個飛彈進行任何的反應動作，那麼飛彈就一定會擊中這個角色，因此我們可以強迫角色一定要對這些飛彈進行閃躲。圖 6.24是我們的實驗場景，紅色的物體是由系統隨機發射的飛彈，這些飛彈會從各個方位對角色進行攻擊。圖中角色為了閃躲場景中的飛彈，會從左圖的位置移動到右圖的位置。在這個實驗裡，角色動作預測的時間預算為 12.5ms，系統隨機飛彈的最短發射間隔為 0.1 秒。其中當場景裡的存活飛彈數大於我們限定的最多飛彈數時，系統將停止加入彈的動作，直到存活的飛彈數又小於我們限定的最多飛彈數時才繼續加入飛彈。

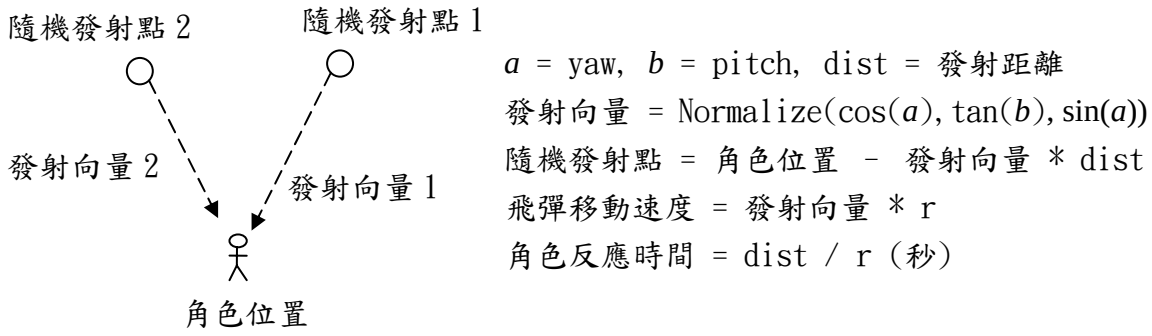


圖 6.23：隨機發射的飛彈和角色位置的關係

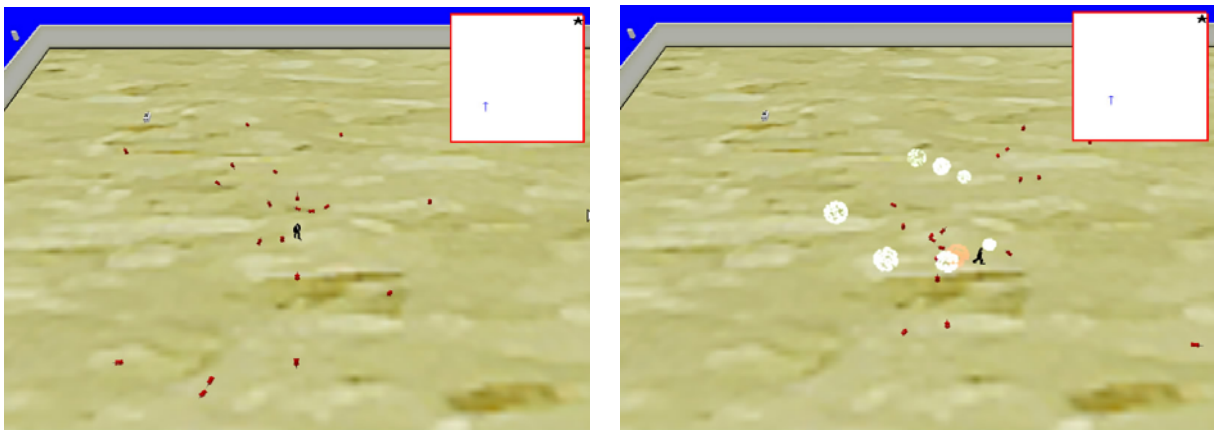
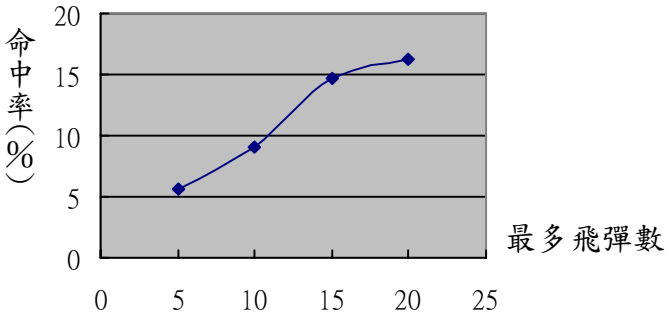


圖 6.24：不同射擊條件測試的實驗場景

(A) 飛彈數目對角色閃避能力的影響

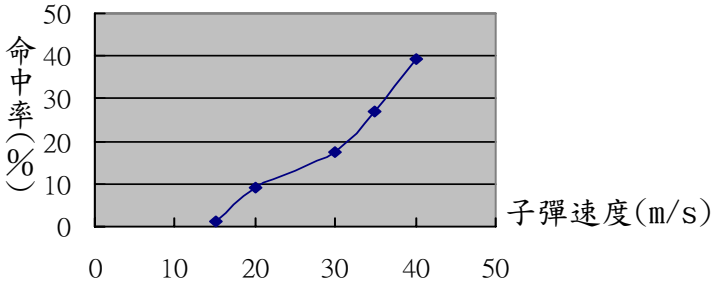
最多飛彈數	命中率(%)
5	5.62
10	9.14
15	14.62
20	16.23



隨機飛彈參數：pitch = 0, yaw = (rand() % 18) * 20, dist = 30, r = 20

(B) 飛彈速度對角色閃避能力的影響

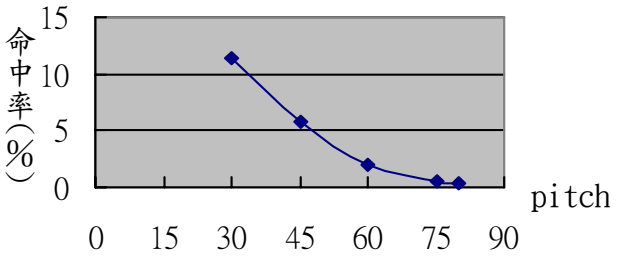
r (子彈速度)	命中率(%)
15	1.14%
20	9.11%
30	17.49%
35	27.06%
40	39.19%



隨機飛彈參數：pitch = 0, yaw = (rand() % 18) * 20, dist = 30, 最大飛彈數 = 7

(C) pitch 對角色閃避能力的影響

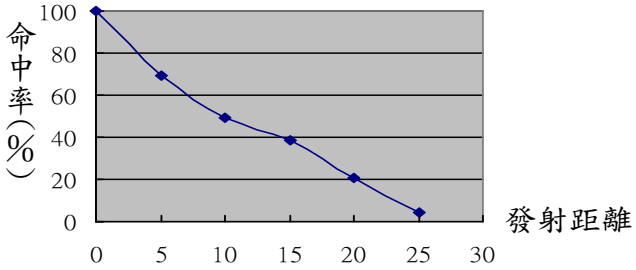
pitch	命中率(%)
30	11.39
45	5.75
60	1.91
75	0.51
80	0.43



隨機飛彈參數：yaw = (rand() % 18) * 20, dist = 30, r = 20, 最大飛彈數 = 20

(D) 發射距離對角色閃避能力的影響

發射距離	命中率(%)
5	69.38
10	48.97
15	38.78
20	20.41
25	4.08



隨機飛彈參數：pitch = 0, yaw = (rand() % 18) * 20, r = 20, 最大飛彈數 = 20

圖 6.25：不同射擊條件測試的實驗數據

- **實驗結果與討論**

圖 6.25是我們的實驗結果。從實驗 A 的數據裡，我們可以看出當飛彈愈多，角色愈容易被飛彈給命中。但即使同時面對著二十個飛彈，飛彈的命中率也只有 16.23%，角色還是有著很高的閃避率。實驗 B 的數據可以看出當飛彈的速度愈快，角色愈容易被飛彈給命中。實驗 C 的數據則可以看出當 pitch 值愈大時，飛彈愈不容易命中角色。這是因為角色的動作圖比較偏向於 XZ 平面的移動，所以當 pitch 值愈小時，飛彈移動範圍和角色移動範圍的重疊部份將會愈多，因此飛彈會比較容易命中角色。實驗 D 的數據能顯示出當發射距離愈短時，飛彈會愈容易命中角色。

在這個實驗裡，角色對飛彈的閃避率會因為隨機飛彈參數的不同而有所改變。而且從我們的分析圖裡也可以看這些改變並不是雜亂無章的變動，而是有著線性增加、曲線遞減或其它相關類型的穩定性變動。所以根據這些分析結果，我們認為本系統的運動模組並不是靠運氣來選擇角色對飛彈的閃避動作，而是能根據情況的不同，提供穩定的閃躲能力給場景中的虛擬角色。

6.3.4 實驗三：不同擴展策略的比較

- **實驗設計**

這裡我們將針對四種不同的動作樹擴展策略進行比較。第一種策略是以目標吻合度搭配深度優先的擴展策略，也就是我們為這個應用所設計的動作樹擴展策略。第二種策略是單純只以目標吻合度來進行動作樹擴展的策略，這種策略可以拿來跟第一種策略進行比較，藉此確認搭配深度優先策略的必要性。第三種策略是以目標吻合度搭配優先權老化的擴展策略，這跟深度優先是完全相反的策略，可以拿來當成對比。第四種策略是寬度優先的擴展策略，用來跟目標吻合度的擴展策略進行比較，藉此確認我們所定義的目標吻合度計算公式，是否能提供較佳的預測效果。

- **實驗結果與討論**

在這個實驗裡，我們同樣還是使用隨機飛彈來測試角色的閃避能力。其中角色動作預測的時間預算為 10ms，系統隨機飛彈的最短發射間隔為 0.1 秒，隨機飛彈參數及實驗結果如圖 6.26所示。從實驗結果裡，我們可以看出 Type1 的方法會讓角色有著比較高的閃避能力，而且它跟 Type2 的效果差別是可以很明顯就被分辨出來的。這可以證明搭配深度優先的策略的確是有一定幫助的。至於 Type2 和 Type3 的結果，雖然它們並沒有很明顯的差別，但我們還是可以看出搭配優先權老化的策略並不會有比較好的效果。Type4 則是所有方法裡效果最差的，它的閃避能力會明顯低於 Type2。這可以證明我們所設計的目標吻合度計算公式，的確能提供給角色不錯的預測效果。

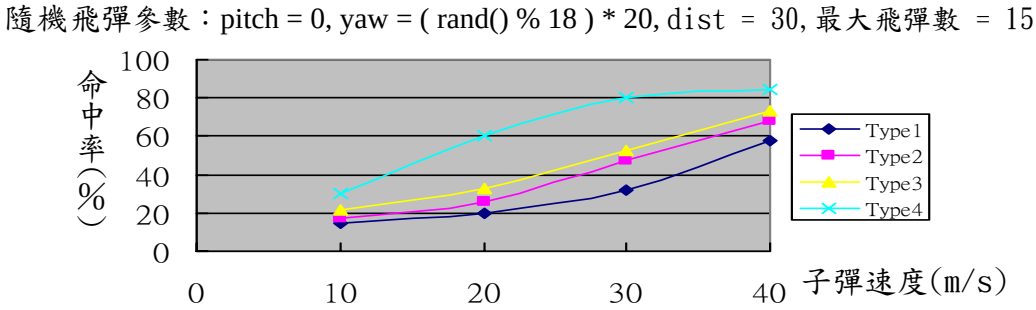


圖 6.26：比較不同擴展策略的實驗數據

6.3.5 實驗四：不同動作樹分析方法的比較

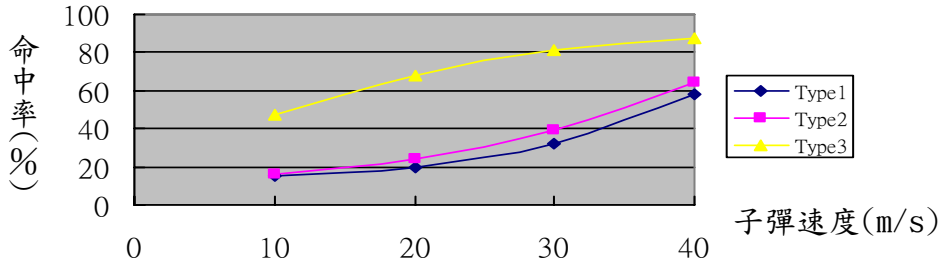
- **實驗設計**

這裡我們將針對三種不同動作樹分析方法進行比較。第一種方法(Type1)是搜尋最佳路徑，第二種方法(Type2)是搜尋最佳子樹，這兩種方法是我們為這個應用所設計的兩種不同動作樹分析方法，它們會分析可行動作樹裡的預測結果，並從中找出最佳的選擇。第三種方法(Type3)則是只搜尋根節點的所有子節點，從中選出最佳子節點，這種方法相當於是只看角色的下一步未來，幾乎沒有使用到可行動作樹裡的預測結果。

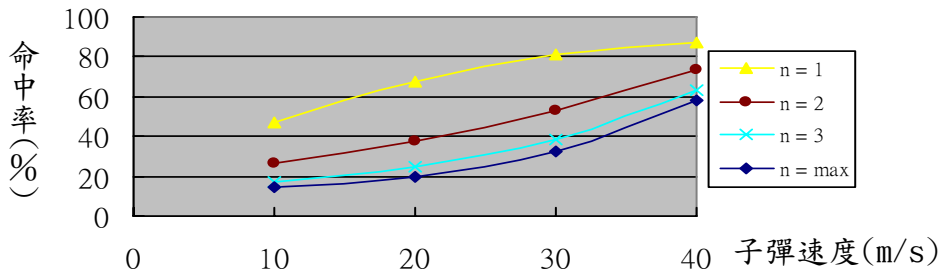
- **實驗結果與討論**

在這個實驗裡，我們同樣還是使用隨機飛彈來測試角色的閃避能力。其中角色動作預測的時間預算為 10ms，系統隨機飛彈的最短發射間隔為 0.1 秒，隨機飛彈參數及實驗結果如圖 6.27。從實驗結果(a)裡，我們可以看出 Type1 的方法會讓角色有著比較高的閃避能力，而 Type2 則相對稍差，但它們之間的差別並不會很明顯，都還有著不錯的閃避能力。Type3 是其中最差的方法，即使在子彈速度最慢的 10m/s 情況下，角色被飛彈命中的機率也還是高達 47%，是比較不能被接受的實作方法。實驗結果(b)則是限制可行動作樹的最大擴展深度，並以搜尋最佳路徑的方式來分析可行動作樹，這種方法相當於限制角色只能看 n 步後的未來。從實驗結果裡，我們可以看出當預測的最大深度為 3 時，角色就能有著不錯的閃避能力，而且效果跟深度為 max 的方法並不會有很明顯的差別。這說明我們的可行動作樹並不需要擴展得太深，系統就能有著不錯的預測效果。

隨機飛彈參數：pitch = 0, yaw = (rand() % 18) * 20, dist = 30, 最大飛彈數 = 15



（a）三種不同類型的動作樹分析方法



（b）最大搜尋深度對搜尋最佳路徑的影響

圖 6.27：不同動作樹分析方法的實驗數據