Search, Path Finding, Steering Behavior, FSM

Game AI

王銓彰
墨匠科技 BlackSmith CEO
cwang001@mac.com
cwang001@blacksmis.com

Content

- Introduction (遊戲人工智慧簡介)
- Search Algorithms
- Path Finding (路徑搜尋)
- Group Steering Behavior (群體運動)
- Finite State Machines (有限狀態機)

Introduction

Introduction

- 課程介紹
  - 參考書籍
    - 遊戲人工智慧, O'REILLY 中文版
    - David M. Bourg & Glenn Seemann 著
    - 陳建勳 譯, 蘇秉豐 編
  - 課程內容適合程式與企劃

Introduction

- 定義
  - Game AI 的表現型式是在遊戲進行時，讓電腦或電腦操控的人物具有一些人工智慧 (AI, Artificial Intelligence) 的表現行為
  - 電腦操控的人物
    - NPC (Non-Playable Character)
    - 非玩家角色
  - Game AI 不等同 AI
    - 我們不見得想讓 NPC 擁有人類水準的智力
    - 我們是想利用 AI 的技術讓 NPC 在預先規劃的模式下行動，展現企劃的設計，呈現出智慧型的反應
    - Game AI 是 Weak AI (弱勢 AI) 的一種
  - Game AI 不只是數值系統

現有常見的 Game AI 技術

- Cheating (作弊)
- FSM*
  - Finite state machine
    - 有限狀態機
  - Fuzzy state machine
    - 模糊狀態機
- Path finding*
  - 路徑搜尋
- Steering behavior*
  - 生物操控轉向行為模擬
- Goal-based planning
  - 目標導向的行為規劃

We will cover the topics with *

Search

Search Algorithms

- Search Algorithms
  - Blind search (盲目搜尋法)
  - Heuristic search (啟發式搜尋法)
  - Adversary search (敵我交換式搜尋法)
- Blind search
  - Breadth-first search (橫向優先搜尋法)
  - Depth-first search (縱向優先搜尋)

Search Algorithms

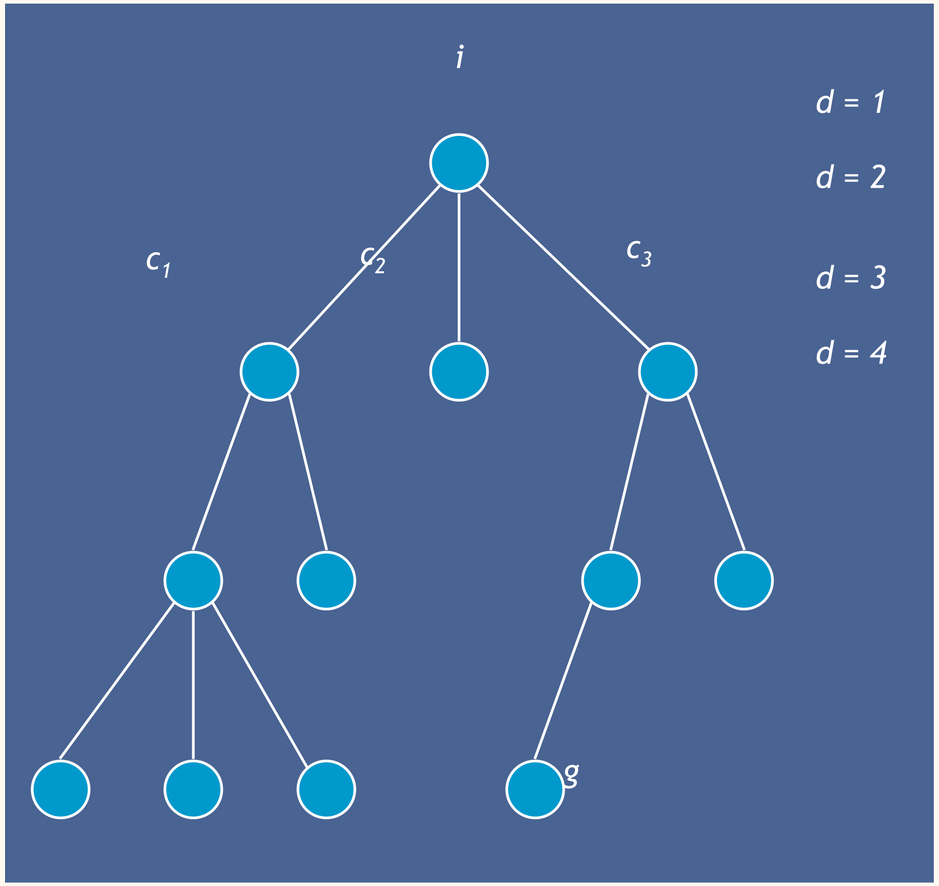
- Heuristic search
  - Search with priority
  - A*
    - A Star
    - 應用於路徑搜尋
- Adversary search
  - Minimax (極大極小值演算法)
  - 棋賽with perfect information

Search Algorithms

- Search是一種解題的方法
  - 尋找一個最"佳"解
  - Search通常搭配Tree的資料結構，而Search tree的方式主要有兩種：
    - Data-driven search，又稱forward chaining
      - 即指從起點出發，尋找終點
      - 我們必須知道每個node的狀態(條件、規則等)。
    - Goal-driven search，又稱backward chaining
      - 是由終點出發，回頭找尋起點
      - 通常我們不清楚展開的規則，不了解問題發生的原因，但知道其結果

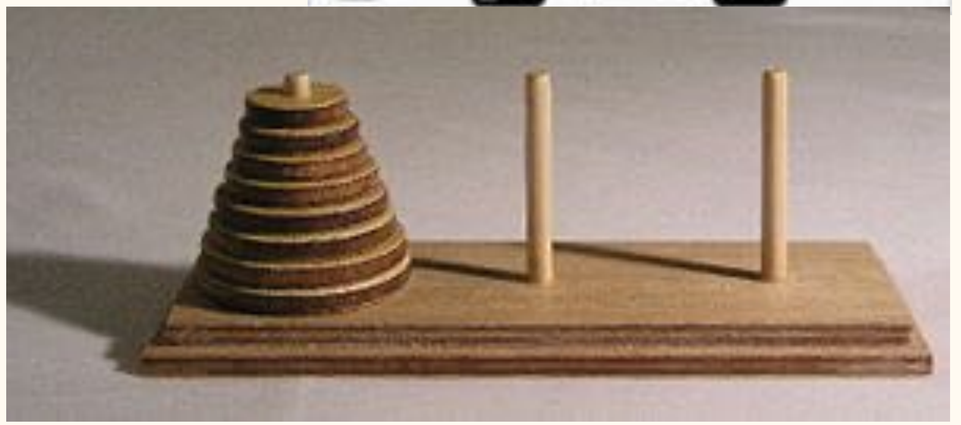
Forward Chaining

- Forward chaining 最常用的兩種演算法為：
  - Depth-First Search
    - 其名的由來在於它是往最深的那個點走，所有路徑都走到底，若沒路再回頭
  - Breadth-First Search
    - 是先將最淺的level的每個node走過，再往下個level走



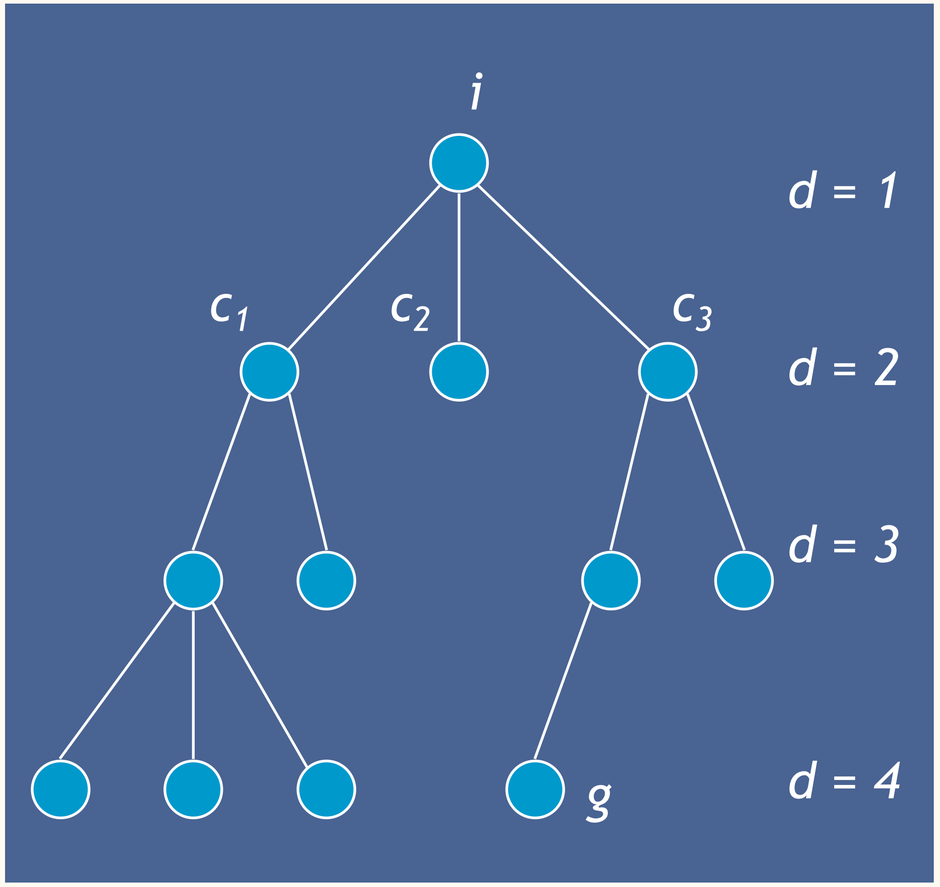
遊戲運用Search的範例

- Game playing
  - Chess
  - Backgammon
- Finding a path to goal
  - The towers of Hanoi
  - Sliding tile puzzles
    - 8 puzzles
- Simply finding a goal
  - n-queens



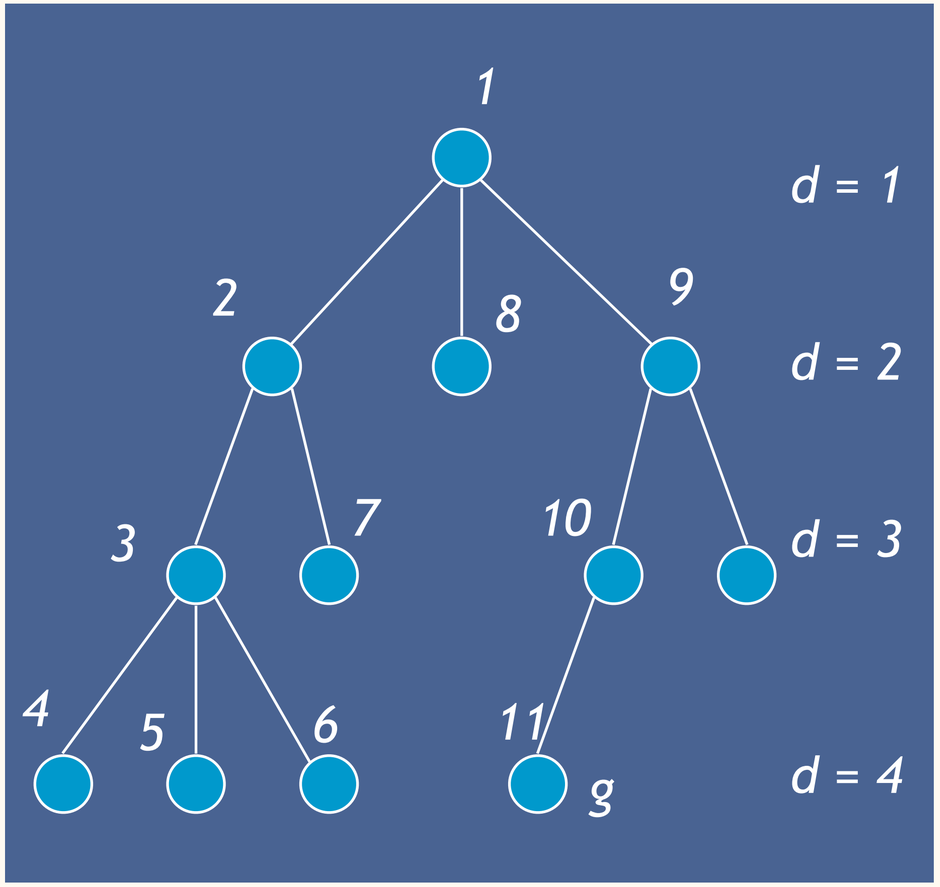
Search

- 專有名詞與定義
  - Search starts (起點)
    - Node i
  - Goal (終點, 目標)
    - Node g
  - Successors (繼承者, 子孫)
    - c_1, c_2, c_3…
  - Depth (代, 深度)
    - d = 1, 2, …



Depth-first Search

- Always exploring the child of the recent expanded node (一定先拜訪目前節點的兒子)
- Terminal nodes being examined from left to right (終端節點是由左往右檢視)
- If the node has no children, the procedure backs up a minimum amount before choosing another node to examine. (如果沒有子代的節點, 往上一代搜尋)
- Stop the search when we select the goal node g. (找到目標節點就結束)

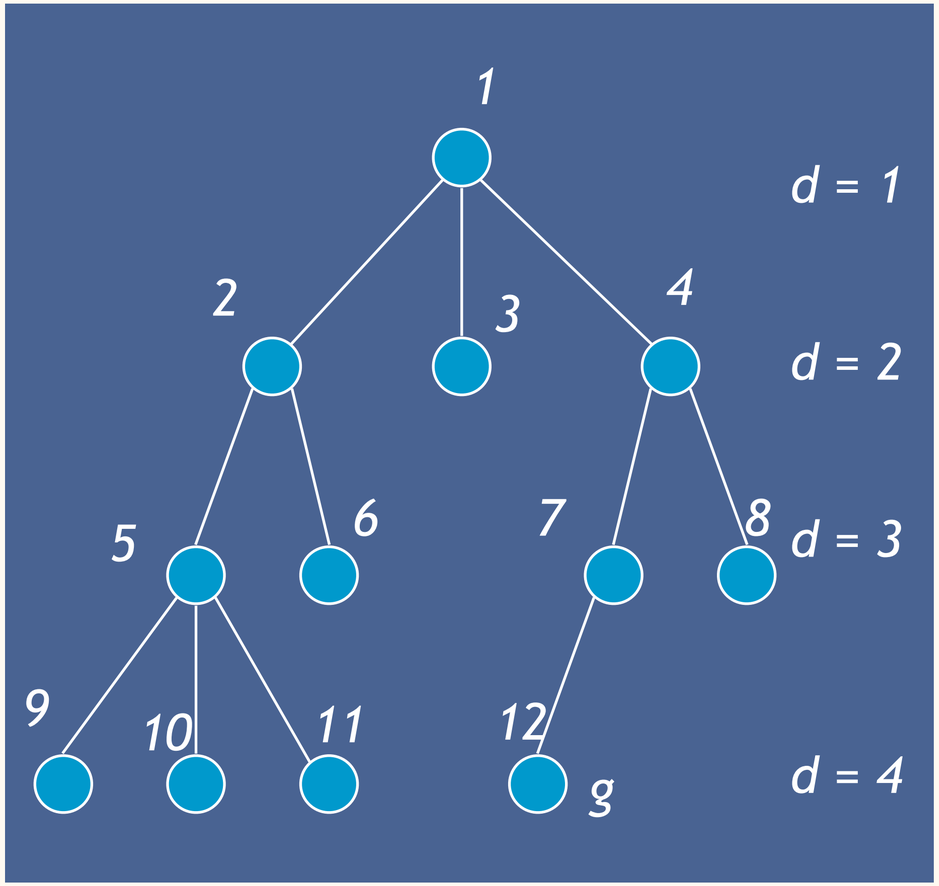


Depth-first Search

- Algorithm
  - Set L to be a list of the initial nodes.
  - Let n be the first node on L. (step 2)
  - If n is a goal node, stop and return it and path from the initial node to n.
  - Otherwise, remove n from L and add to the front of L all of n's children, labeling each with its path from the initial node. Return to step 2.
- Always choosing the first node on L as the one to expand.
  - Depth-first search is implemented by pushing the children of a given node onto the front of the list (用堆疊,Stack,管理, first-in Last-out)

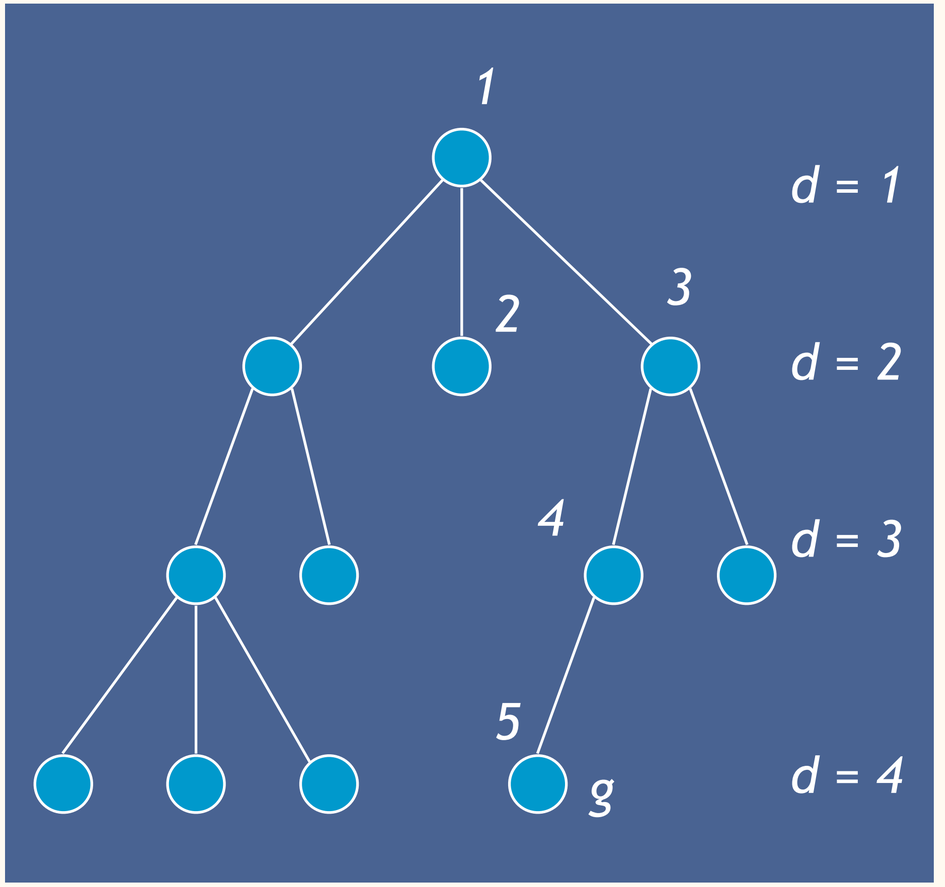
Breadth-first Search

- 是以某一節點為出發點，先拜訪所有同一代相鄰的節點。再依序拜訪與剛才被拜訪過的節點相鄰但未曾被拜訪過的節點，直到所有相鄰的節點都已被拜訪過，再訪問下一代
- The tree examined from top to down, so every node at depth d is examined before any node at depth d + 1.
- L is managed by “Queue” structure.
  - First-in-first-out



Heuristic Search

- Exploring the tree in anything resembling an optimal order. (以最佳的次序展開樹狀結構的搜尋)
- Minimizing the cost(最少的成本)
- When we picking a node from the list L in search procedure, what we do is to remove steadily from the root node toward the goal by always selecting a node that is as close to the goal as possible. (總是挑選最有可能接近目標的節點展開搜尋)
- Estimated and minimizing the cost !? (估算到終點的距離並儘量降低成本)
- A* (A Star 演算法)

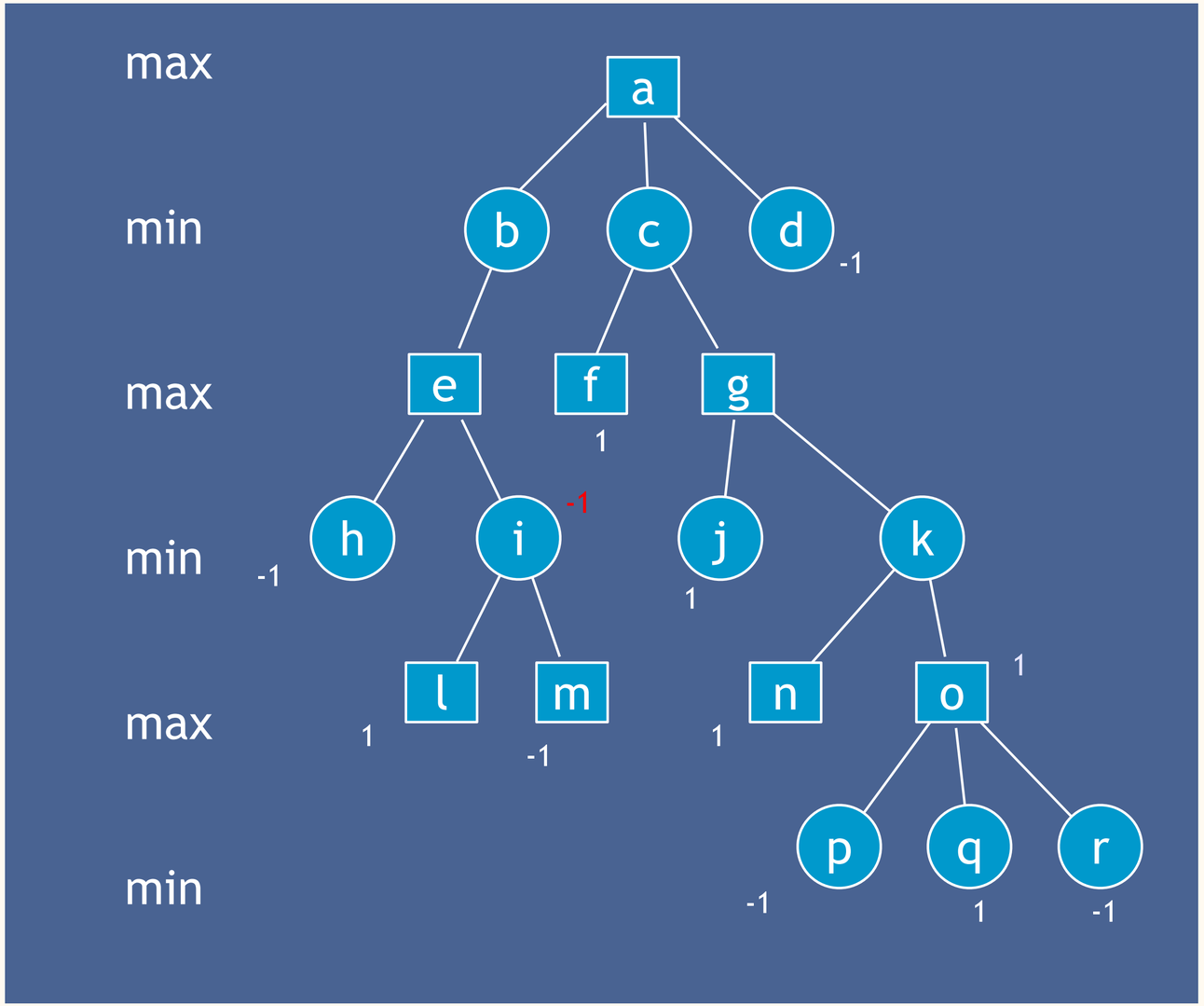


Adversary Search

- 前提
  - 雙人對戰，互相輪流
  - “Perfect” information, where the knowledge available to each player is the same. (雙方公平對奕，無機率計算)
- Examples :
  - Tic-tac-toe
  - Checkers
  - Chess
  - Go
- “Imperfect” information
  - Pokers
  - Bridge

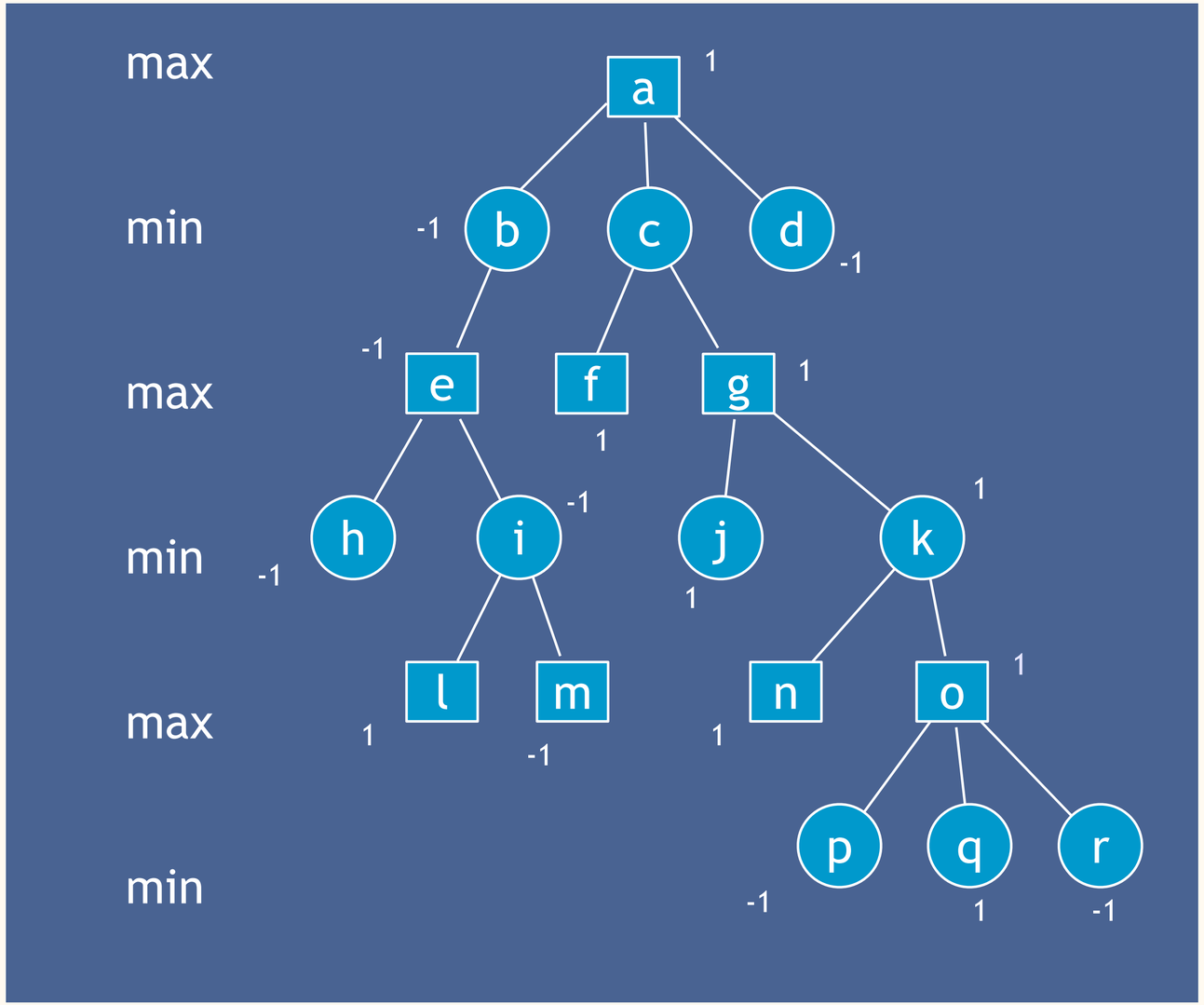
Minimax

- Maximizer to move are in square
- Maximizer to achieve the outcome of 1
- Minimizer to move are in circle
- Minimizer to achieve the outcome of -1



Minimax

- Maximizer win in this case



Minimax

- The algorithm
  1. Set L = { a }, the unexpanded nodes in the tree
  2. Let x be the 1st node on L. If x = a and there is a value assigned to it, return this value.
  3. If x has been assigned a value v_x, let p be the parent of x and v_p the value currently assigned to p. If p is a minimizing node, set $v_p = \min(v_p, v_x)$. If p is a maximizing node, set $v_p = \max(v_p, v_x)$. Remove x from L and return to step 2.
  4. If x has not been assigned a value and either x is a terminal node or we have decided not to expand the tree further, compute its value using the evaluation function. Leave x on L and return to step 2.
  5. Otherwise, set v_x to be $-\infty$ if x is a maximizing node and $+\infty$ if x is a minimizing node. Add the children of x to the front of L and return to step 2.

Minimax

- 議題
  - Draw (平手)
  - Estimated value e(n) : (如何估算輸贏值)
    - e(n) = 1 : the node is a win for maximizer
    - e(n) = -1 : the node is a win for minimizer
    - e(n) = 0 : that is a draw
    - e(n) = -1 ~ 1 : the others
  - When to decide stop the tree expanding further ? (要分析到什麼程度)

Path Finding

Path Finding

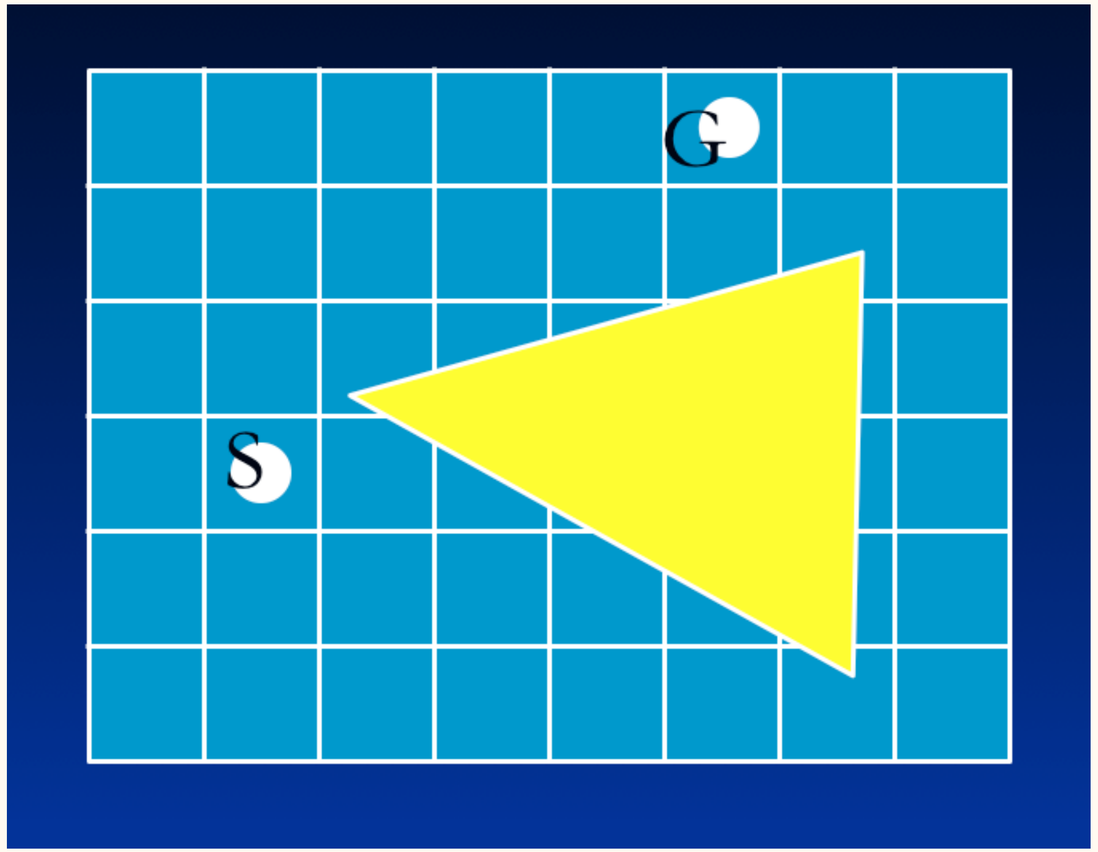
- 因遊戲類型與地形資料而異，與 game play 有密切關聯
  - 從起點出發到目標找一條行走的路徑
  - 過程中要避開障礙物
- A* 是最常用的方法之一
  - 1968
  - Heuristic 搜尋的一種
  - 簡單但效率較差
  - 是一種藉著檢視周圍可行進的方向搜尋一條到目標最低成本路徑的路徑搜尋法
    - Algorithm that searches in a state space for the least costly path from start state to a goal state by examining the neighboring states

Path Finding

- 其他簡易的路徑搜尋法
  - 沿著牆壁
  - 麵包屑搜尋法
  - 固定路線

A*

- State (站的位置)
  - Location
  - Neighboring states (相鄰位置)
- 常用的 Search space (搜尋空間) 型式：
  - Grids (網格)
  - Triangles (三角網絡)
  - Points of visibility (可視點)
- Way points (航點)
  - 搜尋中繼點
- Path (路徑)
- Hierarchical path finding



A* Algorithm
Open : priority queue of search node
Closed : list of search node

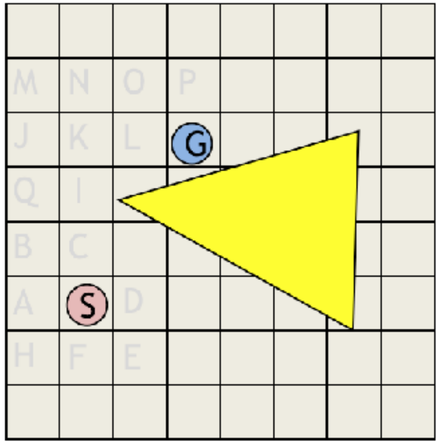
```
AStarSearch( location StartLoc, location GoalLoc, agenttype Agent)
{
    clear Open & Closed

    // initialize a start node
    StartNode.Loc = StartLoc;
    StartNode.CostFromStart = 0;
    StartNode.CostToGoal = PathCostEstimate(StartLoc, GoalLoc, Agent);
    StartNode.TotalCost = StartNode.CostToGoal ;
    StartNode.Parent = NULL;
    push StartNode on Open;

    // process the list until success or failure
    while Open is not empty {
        pop Node from Open  // Node has the lowest TotalCost
```

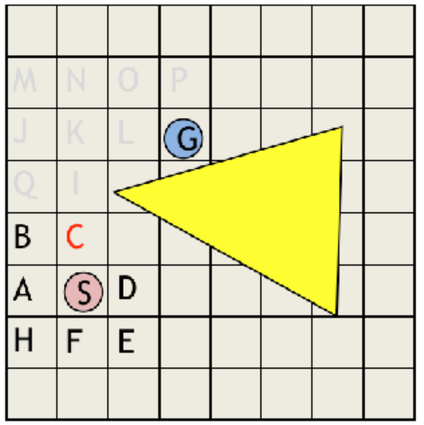
```
// if at a goal, we're done
if (Node is a goal node) {
    construct a path backward from Node to StartLoc
    return SUCCESS;
}
else {
    for each successor NewNode of Node {
        NewCost = Node.CostFromStart + TraverseCost(Node, NewNode, Agent);
        // ignore this node if exists and no improvement
        if (NewNode is in Open or Closed) and
            (NewNode.CostFromStart <= NewCost) {
            continue;
        }
        else {  // store the new or improved information
            NewNode.Parent = Node;
            NewNode.CostFromStart = NewCost;
            NewNode.CostToGoal = PathCostEstimate(NewNode.Loc, GoalLoc, Agent);
            NewNode.TotalCost = NewNode.CostFromStart + NewNode.CostToGoal;
            if (NewNode is in Closed) {
                remove NewNode from Closed
            }
```

```
                if (NewNode is in Open) {
                    adjust NewNode's position in Open
                }
                else {
                    Push NewNode onto Open
                }
            }
        }
    }
    push Node onto Closed
  }
}
```



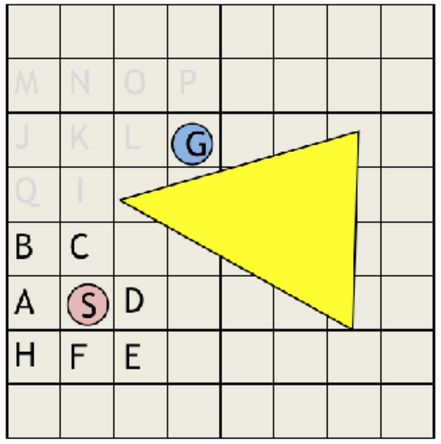
Open (S)
Closed ( )

S [0 + 3.6]



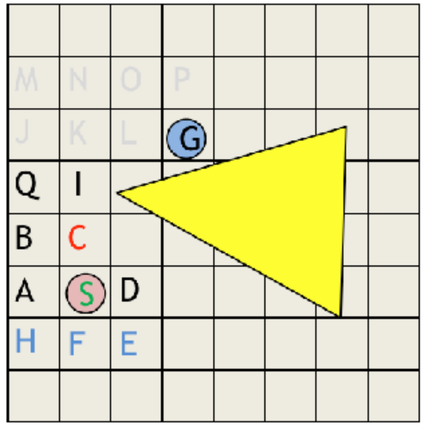
Open (C, D, B, A, E, F, H)
Closed (S)

S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-C [1 + 2.8 = 3.8]
S-D [1 + 3.2 = 4.2]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



Open ( )
Closed ( )

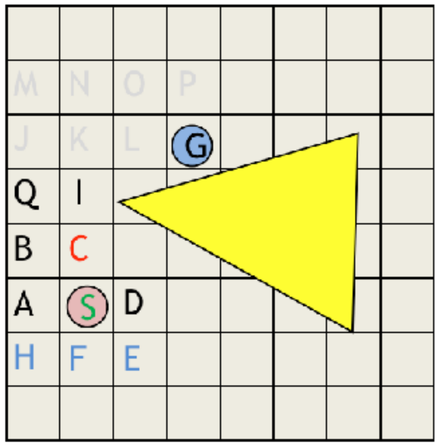
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-C [1 + 2.8 = 3.8]
S-D [1 + 3.2 = 4.2]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



Open (D, I, Q, B, A, E, F, H)
Closed (S, C)

~~S-C-B [1 + 1 + 3.6 = 5.6]~~
S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-C-I [1 + 1 + 2.2 = 4.2]
~~S-C-D [1 + 1.4 + 3.2 = 5.6]~~
~~S-C-A [1 + 1.4 + 4.2 = 6.6]~~

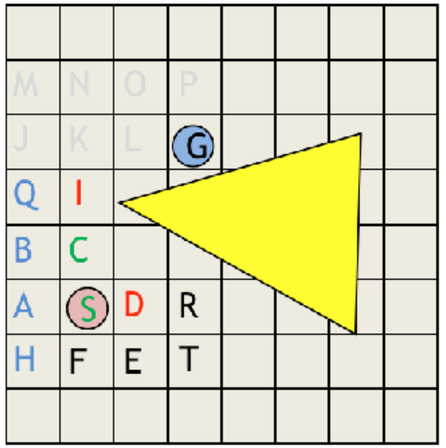
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-D [1 + 3.2 = 4.2]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



Open (D, I, Q, B, A, E, F, H)
Closed (S, C)

S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-C-I [1 + 1 + 2.2 = 4.2]

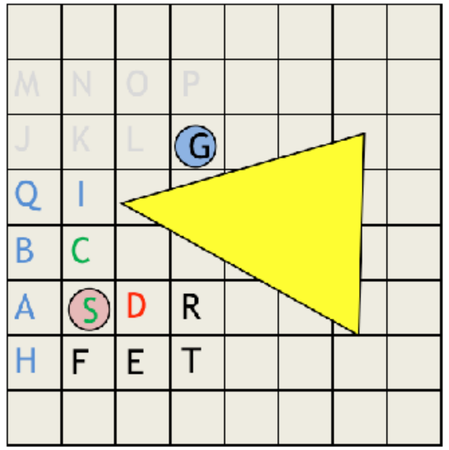
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-D [1 + 3.2 = 4.2]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



Open (I, Q, B, A, E, F, H, R, T)
Closed (S, C, D)

S-D-R [1 + 1 + 3 = 5]
S-D-T [1 + 1.4 + 4 = 6.4]

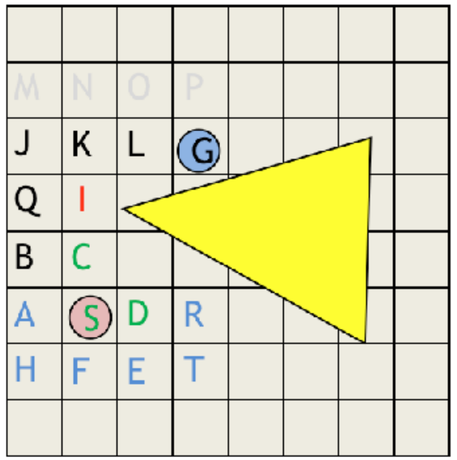
S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-C-I [1 + 1 + 2.2 = 4.2]
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



Open (I, Q, B, A, E, F, H)
Closed (S, C, D)

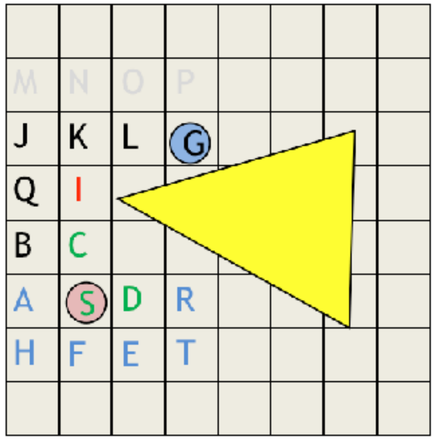
~~S-D-C [1 + 1.4 + 2.8 = 5.2]~~
~~S-D-E [1 + 1 + 4.1 = 6.1]~~
~~S-D-F [1 + 1.4 + 4.5 = 6.9]~~
S-D-R [1 + 1 + 3 = 5]
S-D-T [1 + 1.4 + 4 = 6.4]

S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-C-I [1 + 1 + 2.2 = 4.2]
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]
~~S-C [1 + 2.8 = 3.8]~~



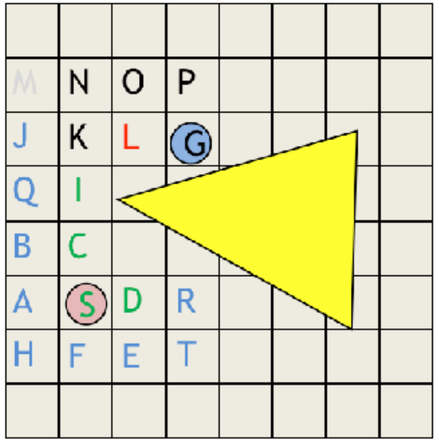
Open (Q, B, A, E, F, H)
Closed (S, C, D, I)

~~S-C-I-Q [2 + 1 + 3.2 = 6.2]~~
S-C-I-J [2 + 1.4 + 3 = 6.4]
S-C-I-K [2 + 1 + 2 = 5]
S-C-I-L [2 + 1.4 + 1 = 4.4]
~~S-C-I-B [2 + 1.4 + 3.6 = 7]~~
S-D-R [1 + 1 + 3 = 5]
S-D-T [1 + 1.4 + 4 = 6.4]
S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



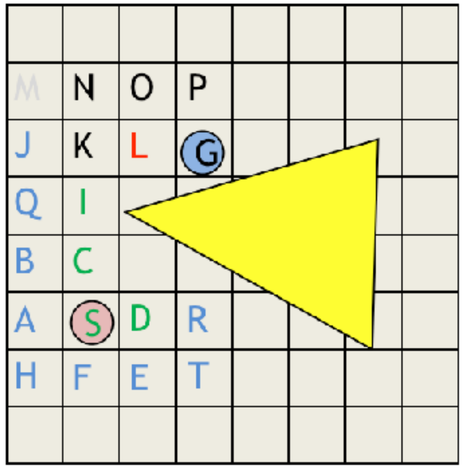
Open (L, J, K, Q, B, A, E, F, H)
Closed (S, C, D, I)

S-C-I-J [2 + 1.4 + 3 = 5.4]
S-C-I-K [2 + 1 + 2 = 5]
S-C-I-L [2 + 1.4 + 1 = 4.4]
S-D-R [1 + 1 + 3 = 5]
S-D-T [1 + 1.4 + 4 = 6.4]
S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



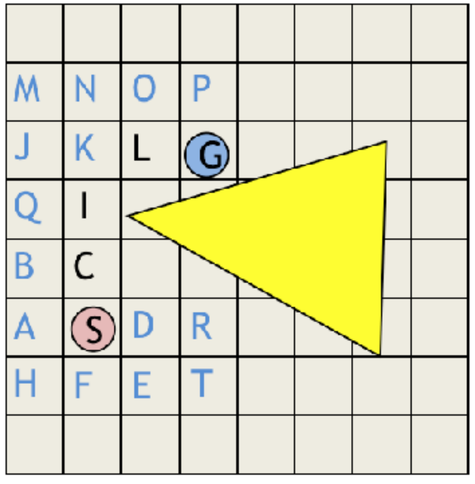
Open (J, K, Q, B, A, E, F, H)
Closed (S, C, D, I, L)

~~S-C-I-L-K [3.4 + 1 + 2 = 6.4]~~
S-C-I-L-N [3.4 + 1.4 + 2.2 = 7]
S-C-I-L-O [3.4 + 1 + 1.4 = 5.8]
S-C-I-L-P [3.4 + 1.4 + 1 = 5.8]
S-C-I-L-G [3.4 + 1 = 4.4]
S-C-I-J [2 + 1.4 + 3 = 5.4]
S-C-I-K [2 + 1 + 2 = 5]
S-D-R [1 + 1 + 3 = 5]
S-D-T [1 + 1.4 + 4 = 6.4]
S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



Open (G, N, O, P, J, K, Q, B, A, E, F, H)
Closed (S, C, D, I, L)

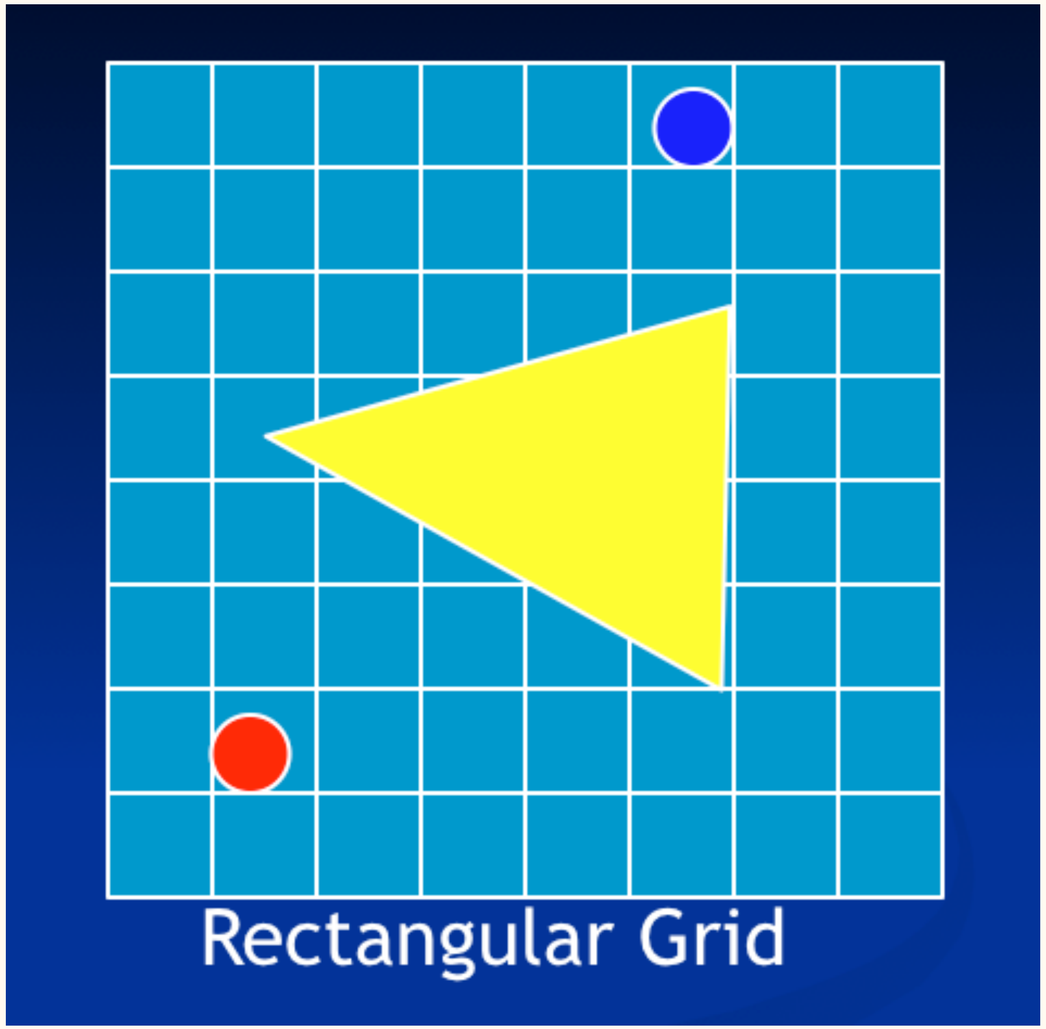
S-C-I-L-N [3.4 + 1.4 + 2.2 = 7]
S-C-I-L-O [3.4 + 1 + 1.4 = 5.8]
S-C-I-L-P [3.4 + 1.4 + 1 = 5.8]
S-C-I-L-G [3.4 + 1 = 4.4]
S-C-I-J [2 + 1.4 + 3 = 5.4]
S-C-I-K [2 + 1 + 2 = 5]
S-D-R [1 + 1 + 3 = 5]
S-D-T [1 + 1.4 + 4 = 6.4]
S-C-Q [1 + 1.4 + 3.2 = 5.6]
S-A [1 + 4.2 = 5.2]
S-B [1.4 + 3.6 = 5.0]
S-E [1.4 + 4.1 = 5.5]
S-F [1 + 4.5 = 5.5]
S-H [1.4 + 5 = 6.4]



S-C-I-L-G

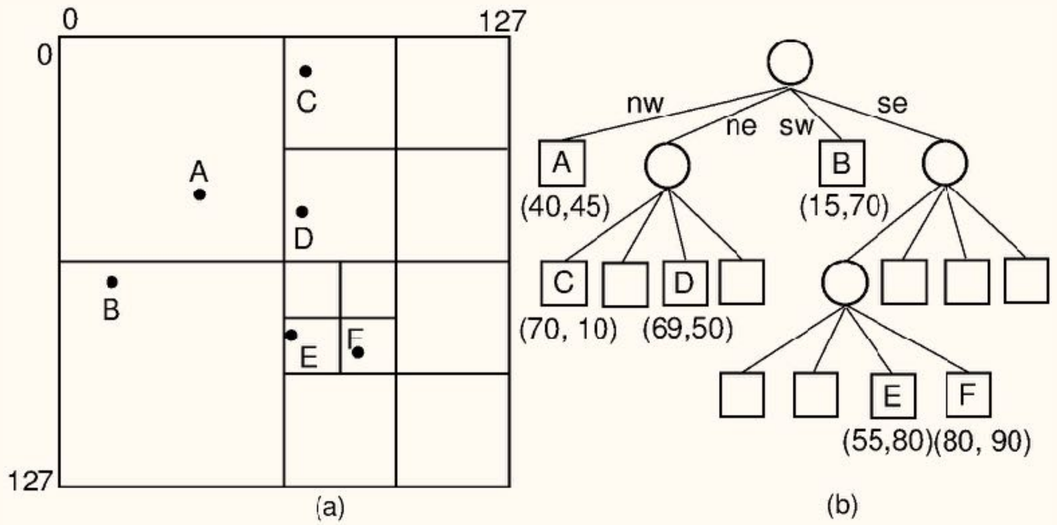
Search Space & Neighboring States

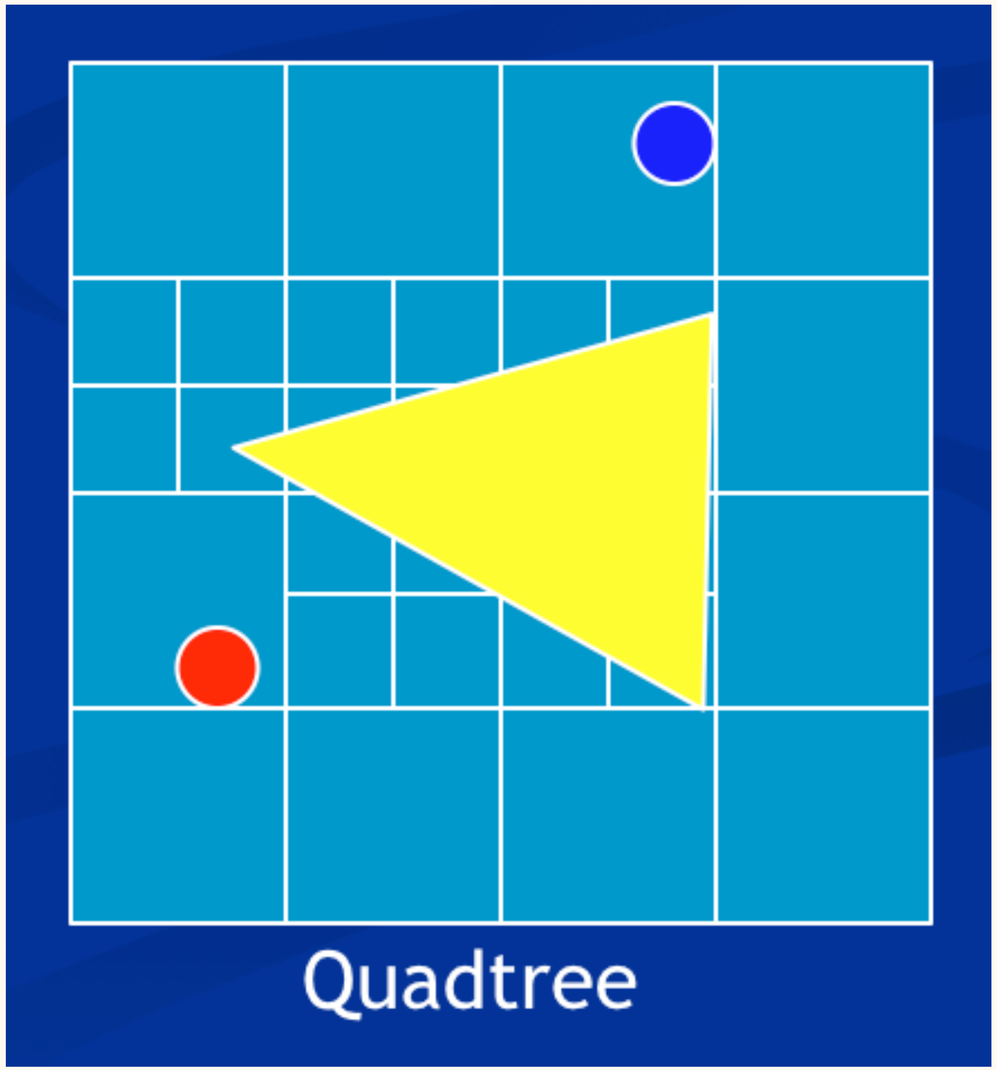
- Rectangular grid
  - Use grid center



Search Space & Neighboring States

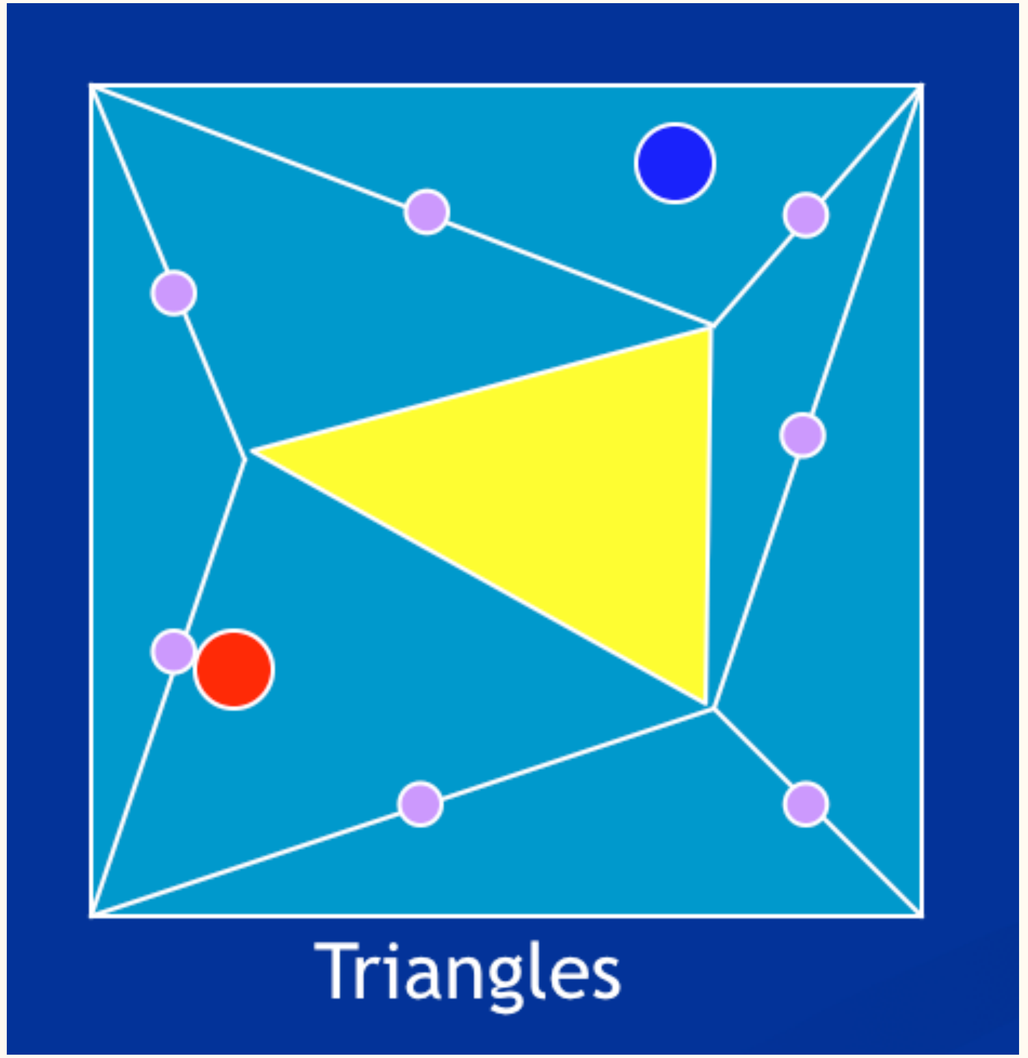
- Quadtree





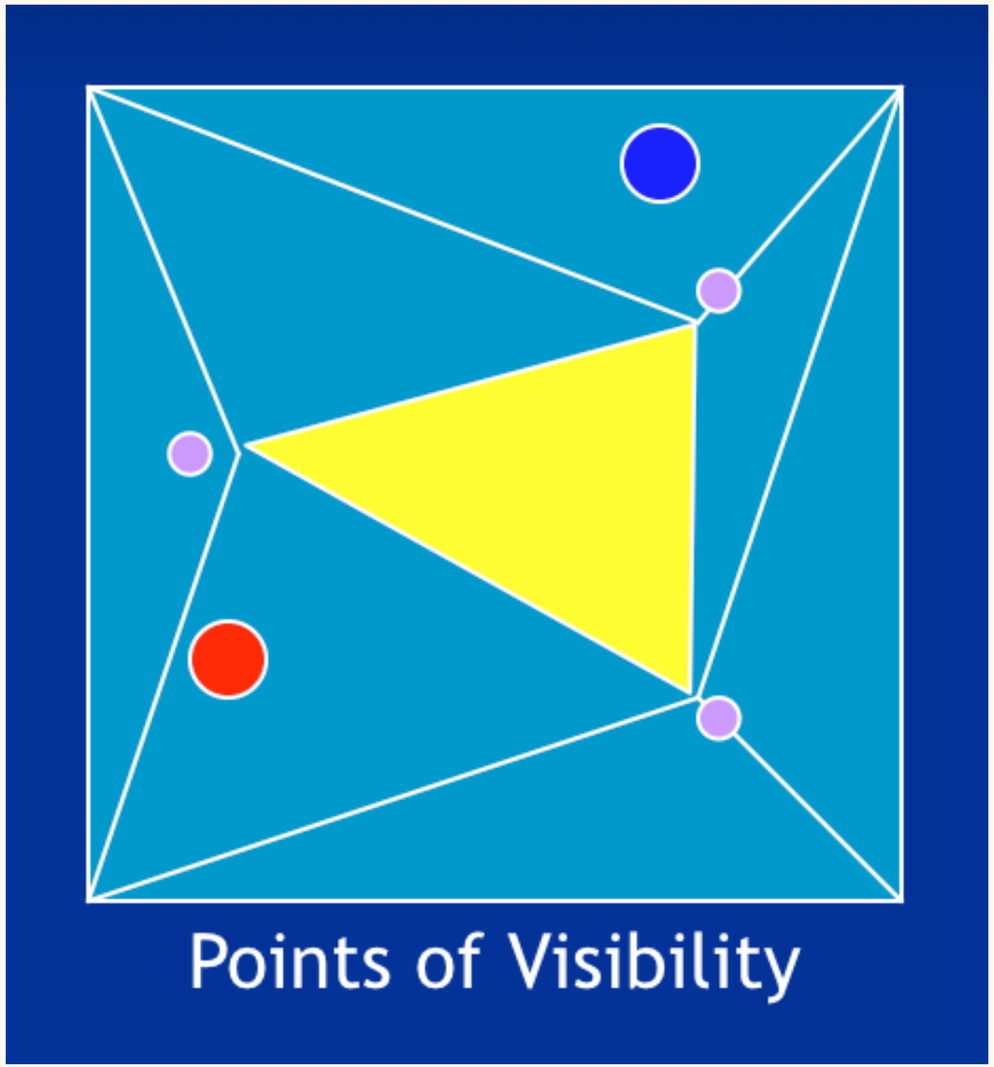
Search Space & Neighboring States

- Triangles or convex polygons
  - Use edge mid-point
  - Use triangle center



Search Space & Neighboring States

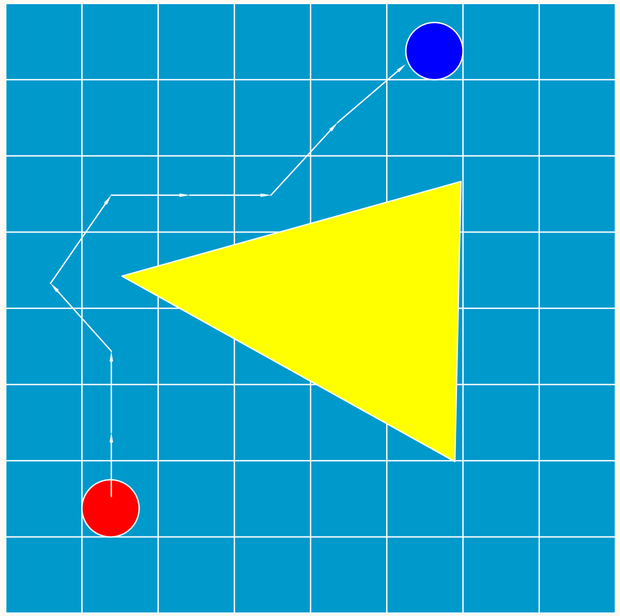
- Points of visibility (POV)



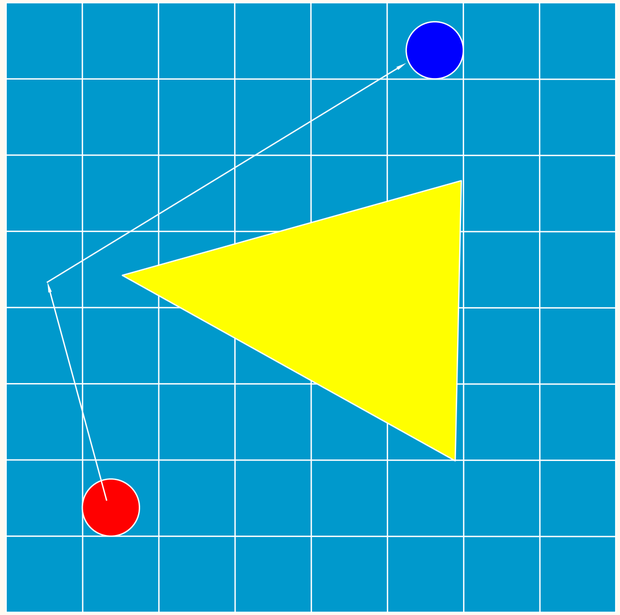
Cost Estimation（成本估算)

- Cost Function
  - Cost-From-Start (從起點到目前的位置)
    - 準確
    - 已經行走的部份
  - Cost-To-Goal (從目前的位置到目標)
    - 估算 / 預測
    - 未完成的部份
  - 常用估算方法
    - 到目標的直線距離

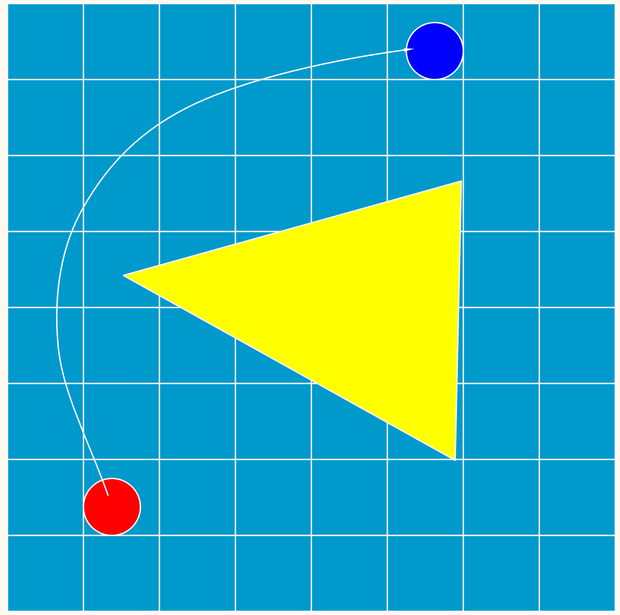
Result Path



Typical A* Path

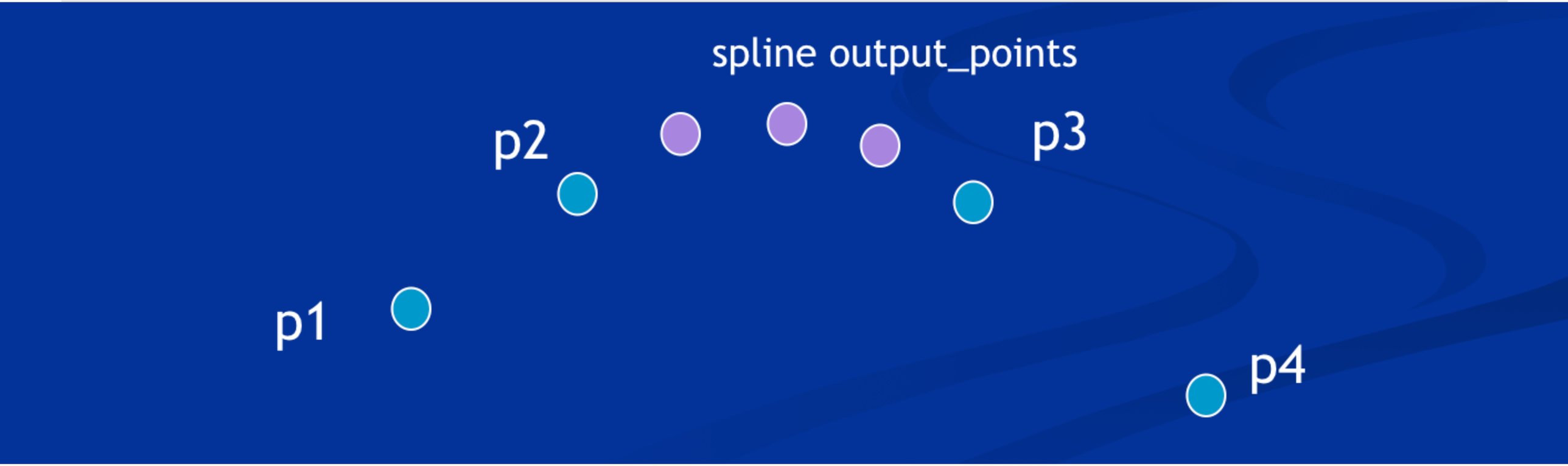


Straight Path



Smooth Path

Catmull-Rom Spline Curve



$$\begin{aligned}\text{Output\_point} = {} & p1*(-0.5u^3 + u^2 - 0.5u) + \\ & p2*(1.5u^3 - 2.5u^2 + 1) + \\ & p3*(-1.5u^3 + 2u^2 + 0.5u) + \\ & p4*(0.5u^3 - 0.5u^2)\end{aligned}$$

Hierarchical Path Finding

- Break the terrain for path finding to several ones hierarchically
  - Room-to-room
  - 3D layered terrain
  - Terrain Level-Of-Detail (LOD)
- Pros
  - Speedup the search
  - Solve the problem of layered path finding

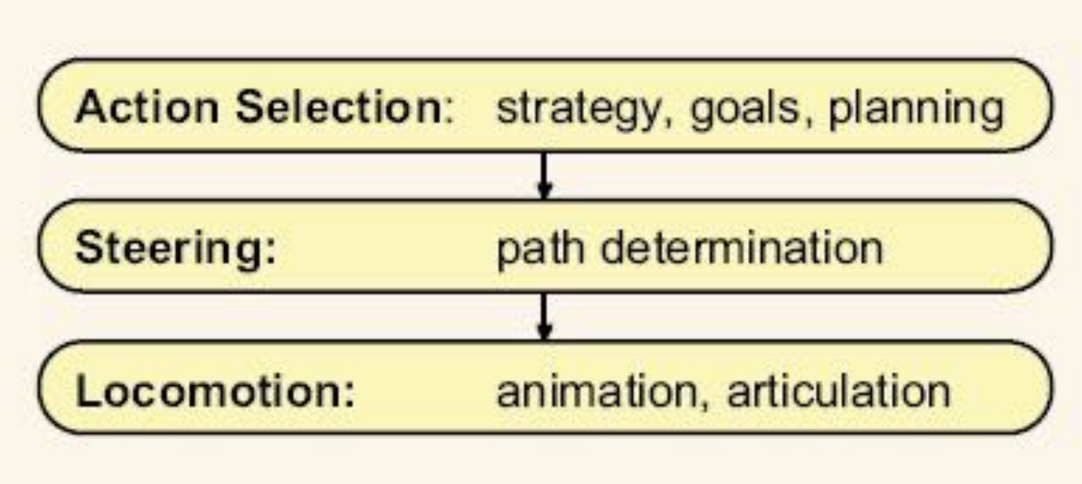
Path Finding 的挑戰

- 移動中的目標
  - Do you need to find path each frame ?
- 移動中的障礙物
  - Prediction Scheme (預測機制)
  - Steering behavior (行走操控行為)
    - Obstacle and collision avoidance (障礙物與碰撞避免行為)
- 複雜的地形
  - Hierarchical path finding
- “Good” path (“好”的路徑)
  - Catmull-Rom spline
  - Use “Steering Behavior”

Steering Behaviors

Motion Behavior（動作行為）

- Action selection（採取行動的決策）
- Steering（操控）
- Locomotion（移動）



動作行為關係圖

Action Selection (行動的選擇與決策)

- Game AI 引擎的主要功能之一
  - State machine (狀態機)
    - Discussed in “Finite State Machine” section (詳見”有限狀態機”章節)
  - Planning (行動規劃)
    - Goal-based planning (目標導向的行動規劃)
  - Tactic & strategy (戰術與戰略)
- Scripting (腳本)
- Assigned by players (玩家操控)
  - Players’ input

Steering（操控）

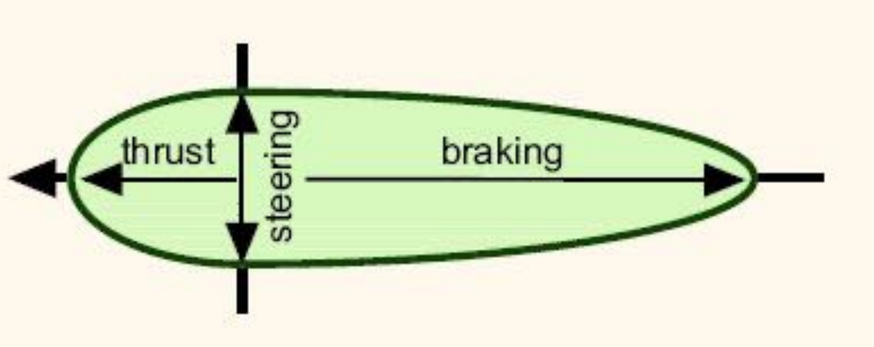
- Path determination
  - Path finding or path planning (路徑搜尋或路徑規劃)
  - Discussed in “Path Finding” (詳見”路徑搜尋”章節)
- Steering Behaviors
  - Seek & flee (尋找與逃跑)
  - Pursuit & evasion (追蹤與逃避)
  - Obstacle avoidance (避開障礙物)
  - Wander (漫步)
  - Path following (路徑跟隨)
  - Unaligned collision avoidance (非特定方向的碰撞避免)
- Group Steering (群體運動)

Locomotion（移動）

- Character physically-based models (人物)
- Movement (移動)
  - Turn right or left (左右轉)
  - Move forward, ... (前進, ... 等)
- Animation (動作)
  - By artists (美術預先製作)
- Implemented / managed by game engine (由遊戲引擎執行與管理)

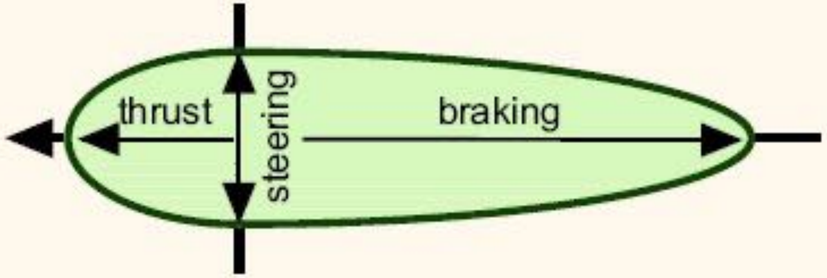
A Vehicle Model (一個簡易車輛的模型)

- Point mass (車子質量集中在一點上)
  - Linear momentum (動量)
  - No rotational momentum (沒有角動量)
- 參數：
  - Mass (質量)
  - Position (位置)
  - Velocity (速度)
    - Modified by applied forces (由力造成改變)
    - Max speed (極速)
  - Max steering force (最大轉向力)
  - Orientation (方位)
    - Car (2D) / Aircraft (3D)



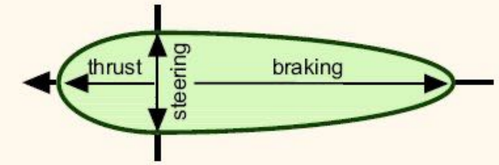
A Vehicle Model (一個簡易車輛的模型)

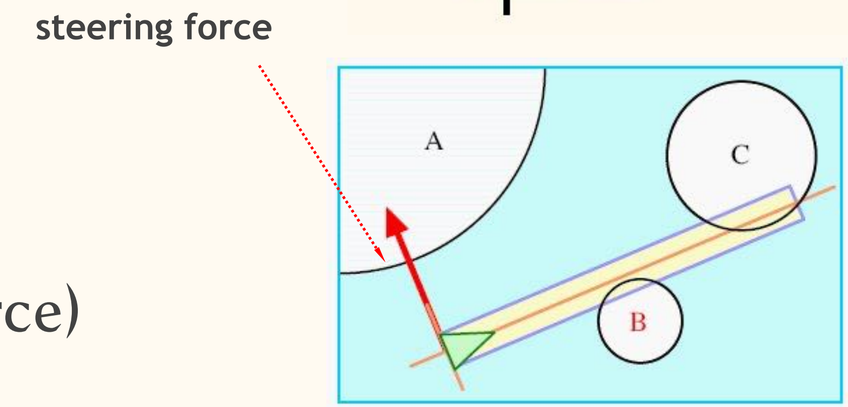
- Local space (區域座標系統)
  - Forward (前進) (z)
- Steering forces (轉向力)
  - Asymmetrical (不對稱)
    - Thrust (推力)
    - Braking (煞車, 制動力)
    - Steering (轉向)
- Velocity alignment (與速度同向)
  - No slide, spin, ... (沒有側滑)
  - Turn (有轉彎)



如何計算 Steering Force?

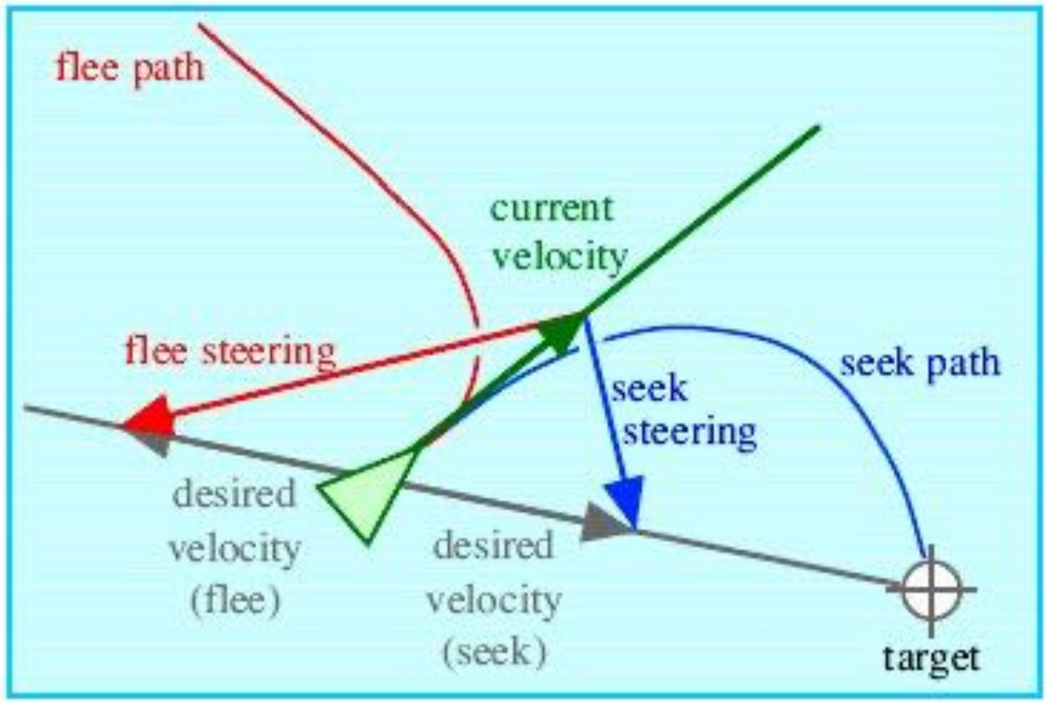
- 方法：
  - Steering Behavior就是一種對NPC的驅動力
  - 先將所有行為(Behavior)的合力求出
  - 將合力分解成行進方向與垂直於行進方向的分力
  - 行進方向的力是影響NPC加速與減速
    - Steer_force = Truncate(streer_direction, Max_force)
    - Acceleration = Steer_force / mass
    - Velocity = Truncate(Velocity + Acceleration, Max_speed)
    - Velocity must >= 0.0
    - Position = Position + Velocity
  - 垂直於行進方向的分力是影響NPC轉向



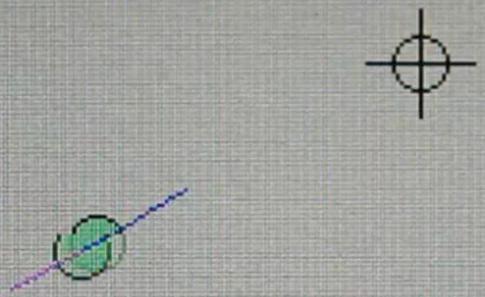


Seek & Flee (尋找與逃跑)

- Pursuit to a static target (追蹤靜態的目標)
  - Steer a character toward to a target position (操控 NPC 朝向目標)
- Flee (逃跑)
  - Inverse of seek (尋找的相反運算)
- Seek Steering force ("尋找"轉向力的計算)
  - vD = normalize(target - position)*speed
  - steering = vD − velocity
- Variants (變化)
  - Arrival (抵達目標)
  - Pursuit to a moving target
    - 跟隨移動中的目標 (領袖跟隨)

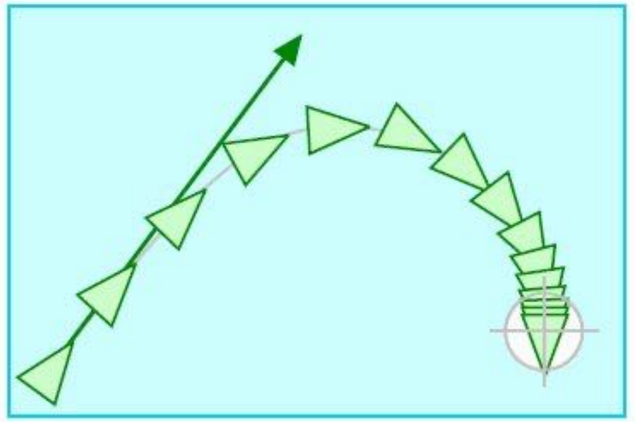


Seek & Flee (尋找與逃跑)

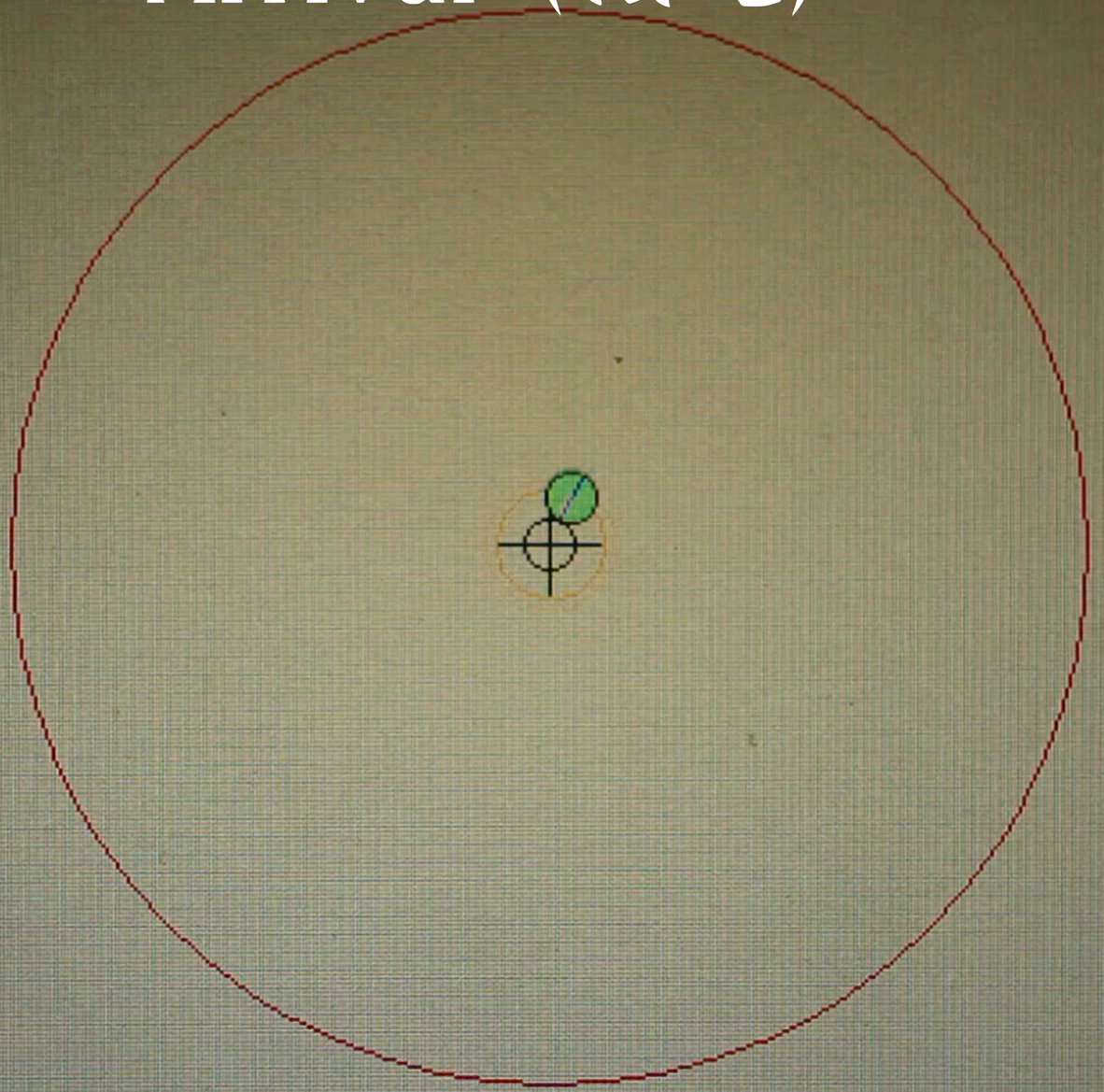


Arrival（抵達）

- 概念：a stopping radius (開始減速的半徑)
  - 半徑外與尋找行為相同
  - 半徑內減速到靜止
    - target_offset = target − position
    - distance = length(target_offset)
    - ramped_speed = max_speed*(distance/slowing_radius)
    - clipped_speed = minimum(ramped_speed, max_speed)
    - desired_velocity = (clipped_speed/distance)*target_offset
    - steering = desired_velocity − velocity

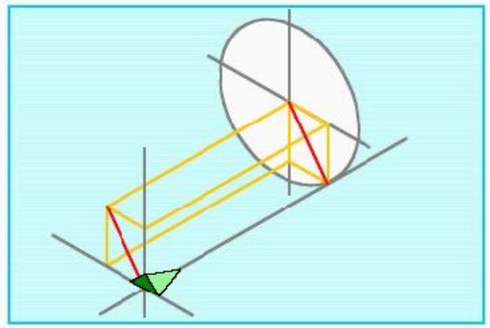


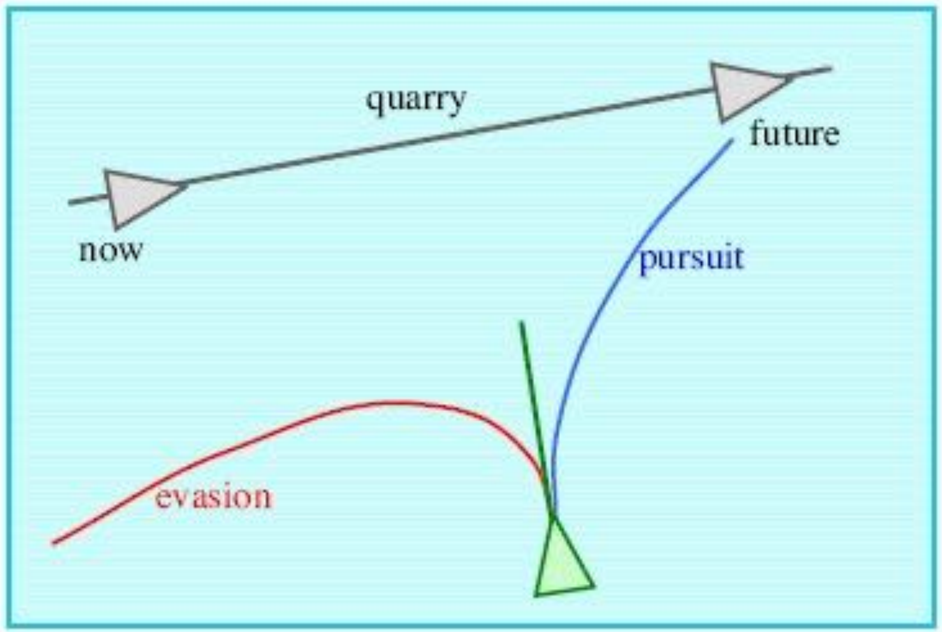
Arrival（抵達）



Pursuit & Evasion (追蹤與逃避)

- Target is moving. (目標是移動中)
  - Apply seek or flee to the target's predicted position
  - 對目標的預定位置做尋找的行為
- Estimate the prediction interval T (目標預定位置的估算)
  - T = Dc
    - D = distance(pursur, quarry)
    - c = turning parameter
- Variants (變化)
  - Offset pursuit (跟飛)
  - "Fly by"

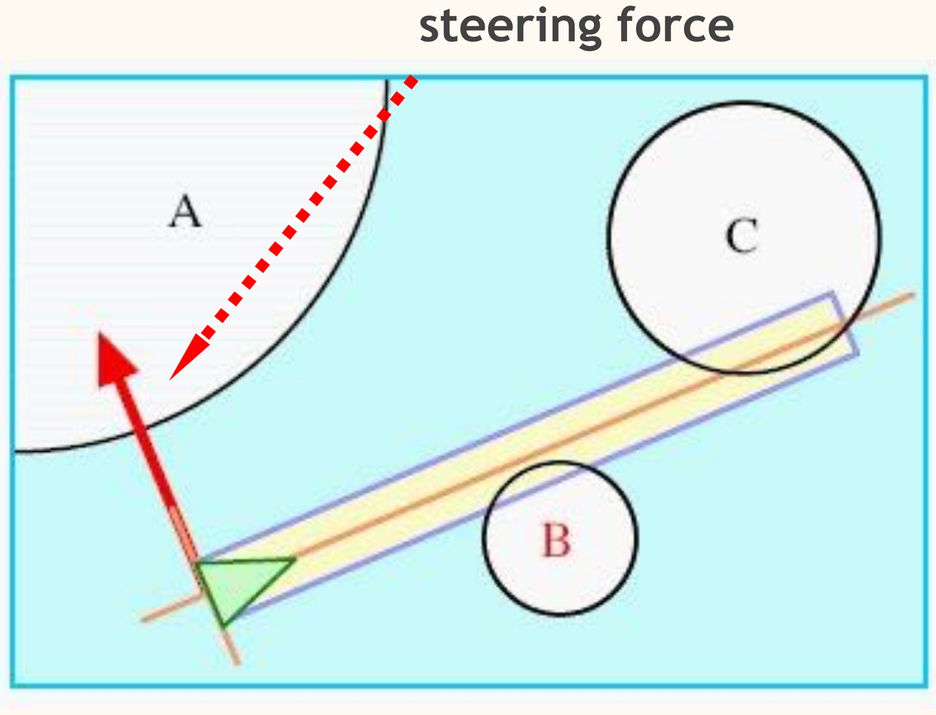




Pursuit & Evasion (追蹤與逃避)

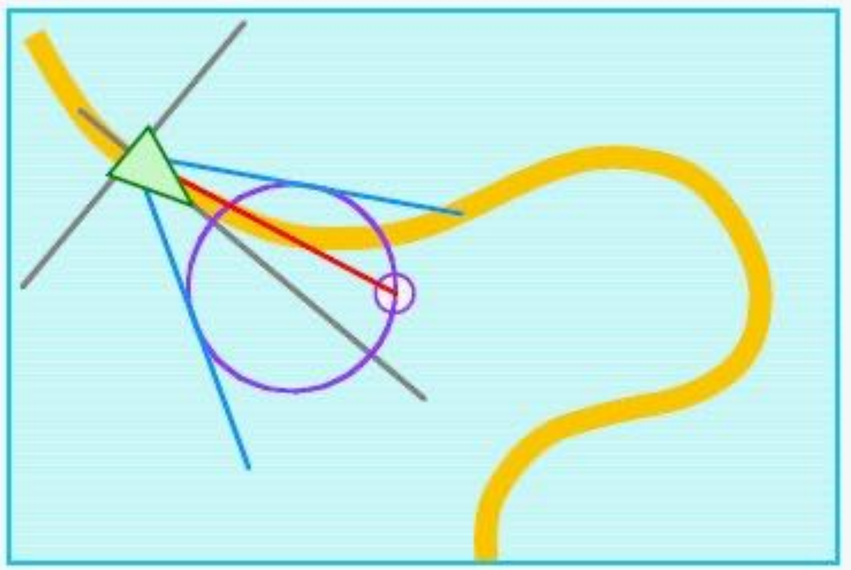
Obstacle Avoidance (障礙物迴避)

- Probe (探針)
  - A cylinder lying along forward axis (從前進方向延伸出的圓柱體範圍)
  - Diameter = character's bounding sphere (探針寬度)
  - Length = speed (means Alert range) (探針長度, 警戒範圍)
- 找出威脅最大的障礙物
  - Nearest intersected obstacle (最近的障礙物)
  - Use bounding sphere 來做碰撞計算
- Steering (算出轉向力)



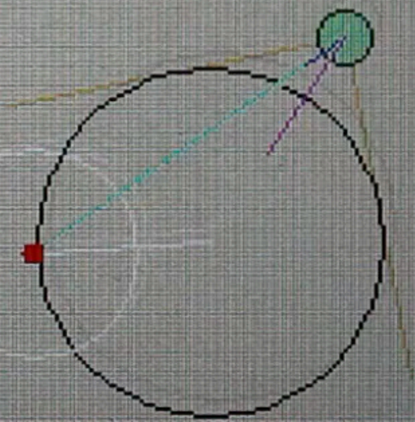
Obstacle Avoidance（障礙物迴避）

Wander（漫步）



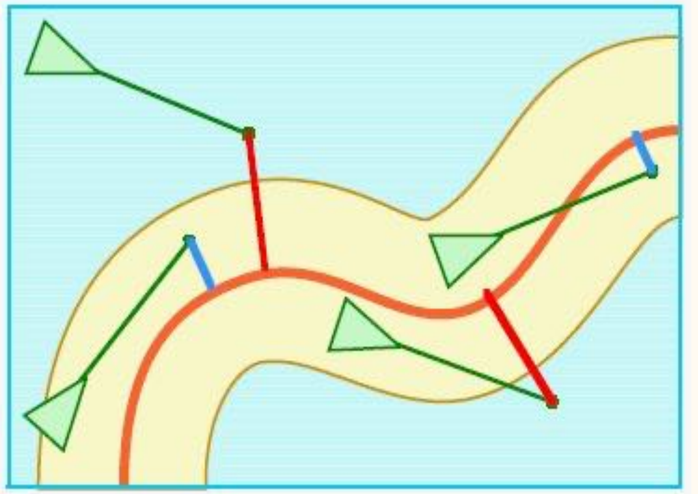
- Random steering（亂數產生的行為）
  - Retain steering direction state（維持轉向的狀態）
  - Constrain steering force to the sphere located ahead of the character
  - 每次只做隨機的角度位移
    - A small sphere on sphere surface to constrain the displacement
- Another solution：
  - Perlin noise
- Variants（類似行為）
  - Explore（探索）

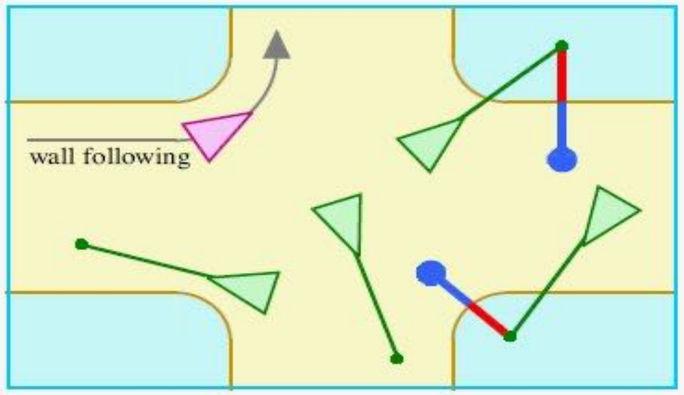
Wander（漫步）



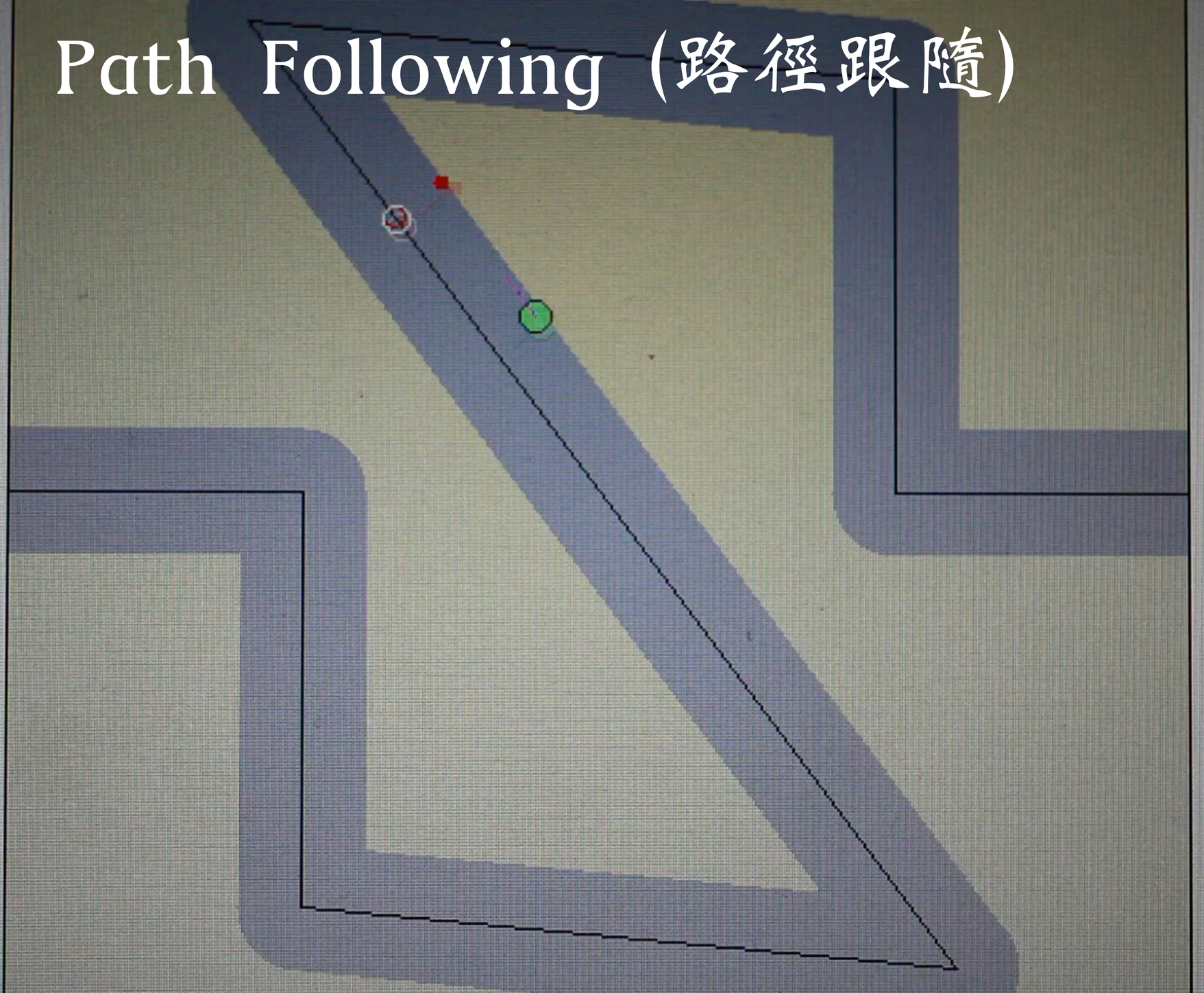
Path Following (路徑跟隨)

- The path (路徑)
  - Spine (軸)：定義路徑的曲線
  - Pipe (通道)
- Following (跟隨)
  - 預測下一步位置
    - 通道內直行
    - 通道外
      - Use “Seek” to the on-path projection
- Variants (類似行為)
  - Wall following (貼牆)
  - Containment (圍牆內)

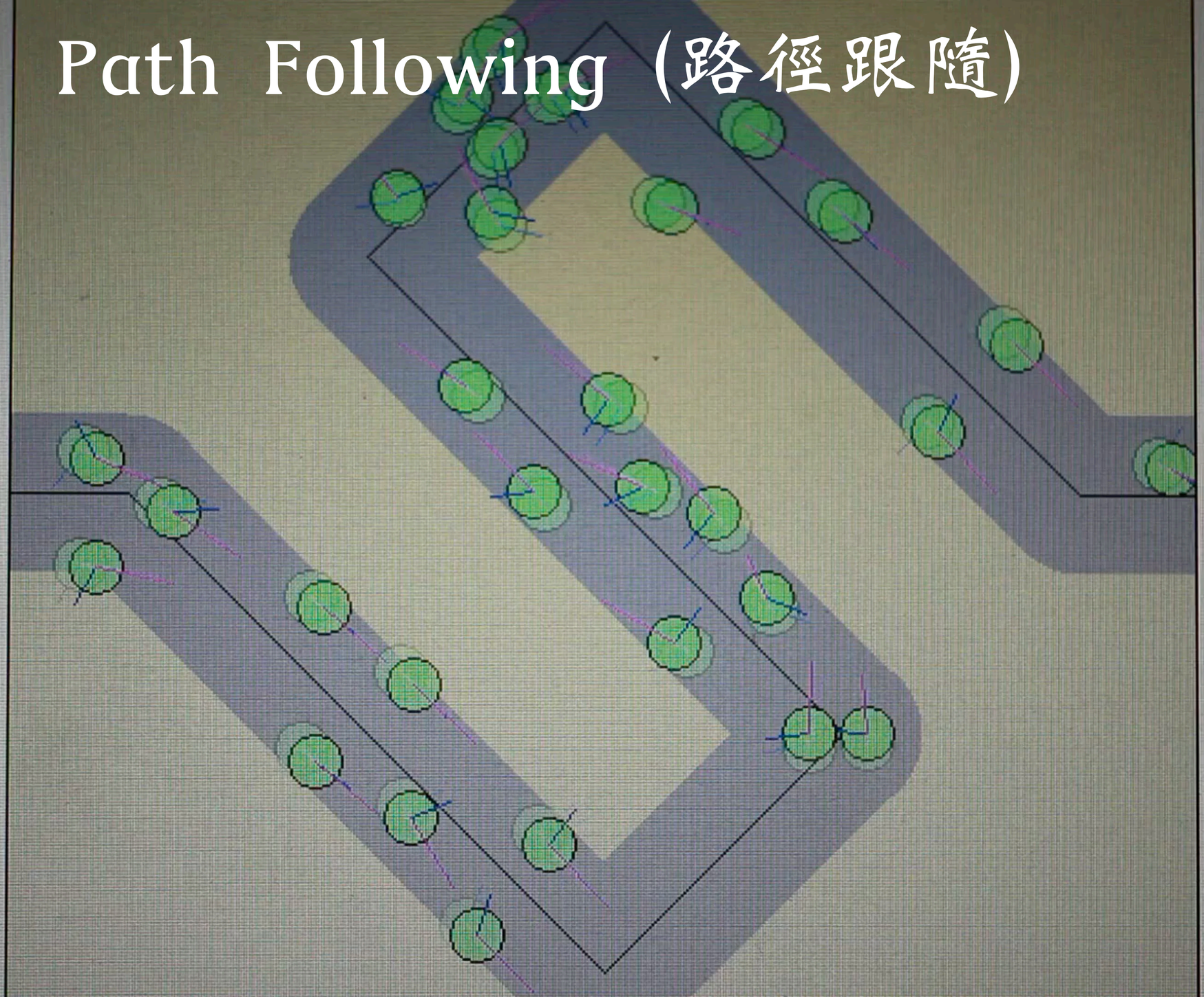




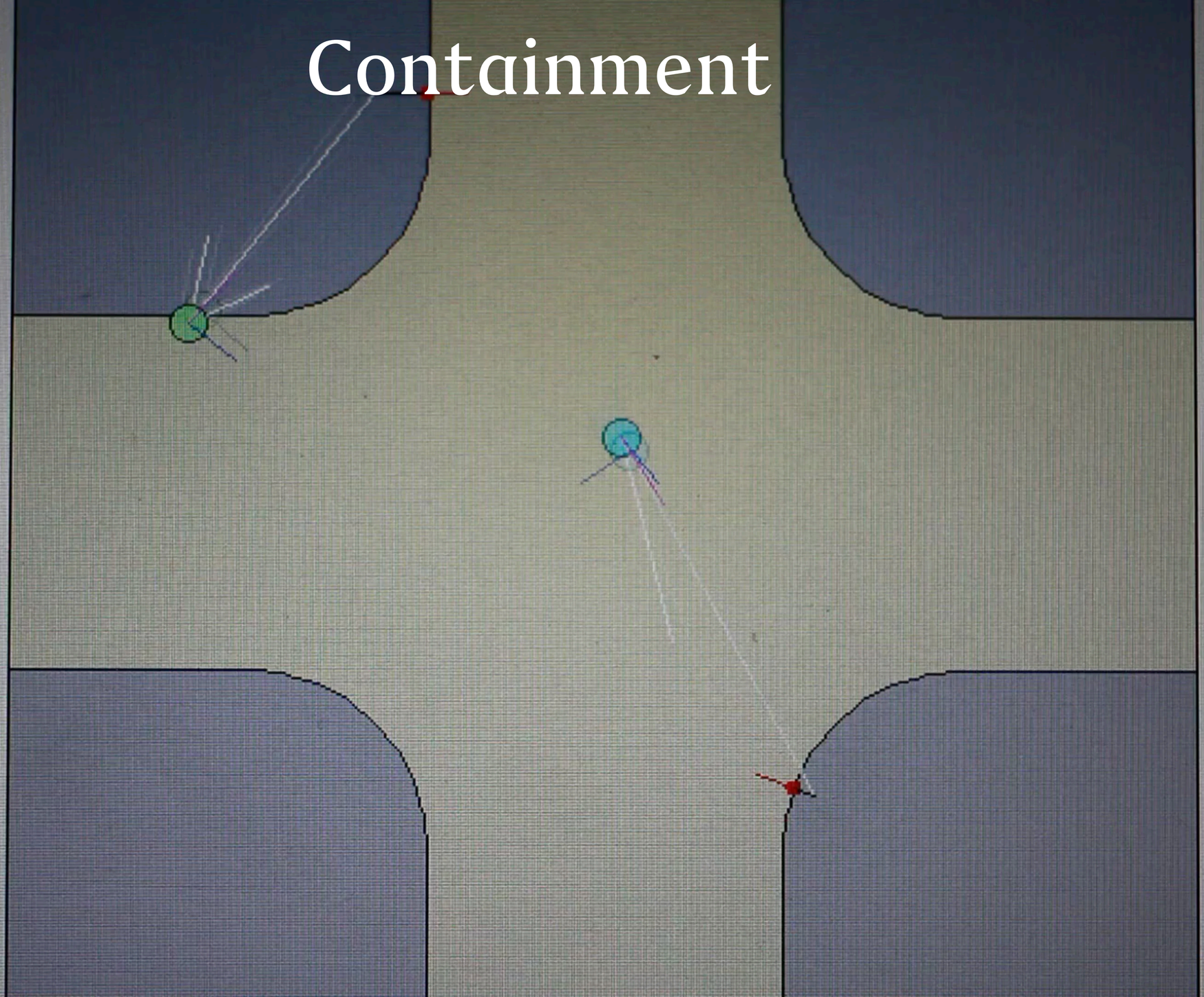
Path Following（路徑跟隨）



Path Following（路徑跟隨）

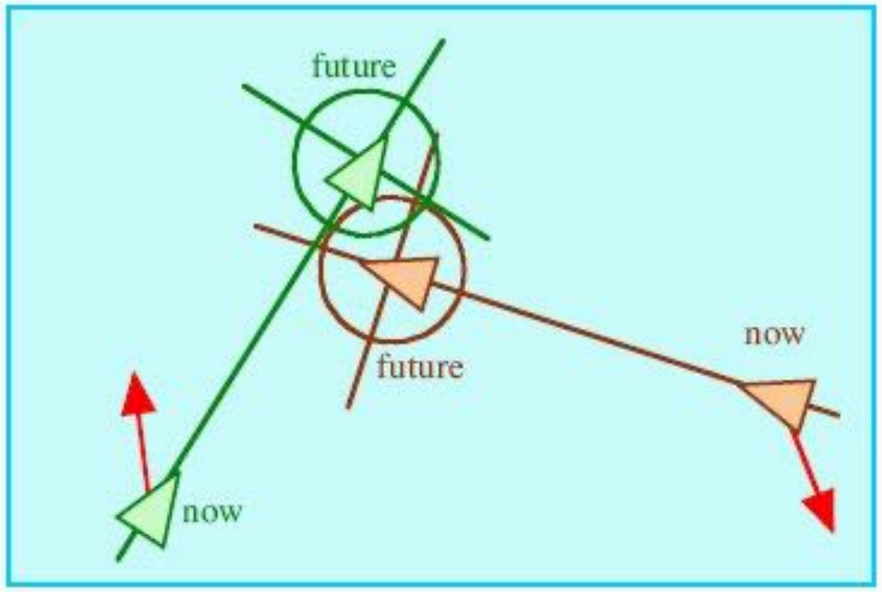


Containment

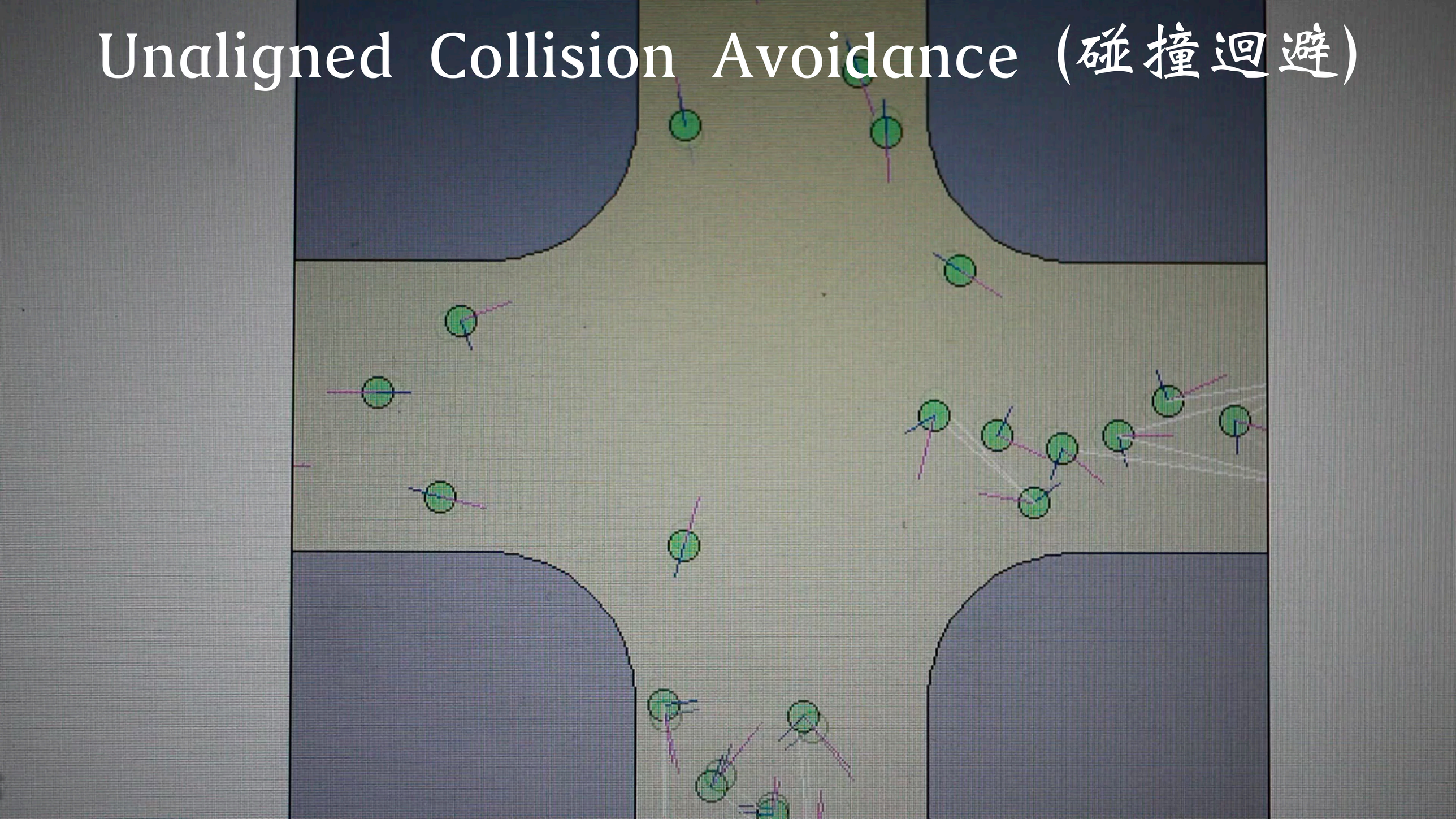


Unaligned Collision Avoidance (碰撞迴避)

- Turn away from possible collision (避免可能的碰撞)
  - Predict the potential collision (預估下一步的碰撞可能性)
  - Use bounding spheres
- 如果可能碰撞, 預估轉向力先行轉向
  - Apply the steering on both characters
  - Steering direction is possible collision result
    - Use “future” possible position
    - The connected line between two sphere centers



Unaligned Collision Avoidance（碰撞回避）

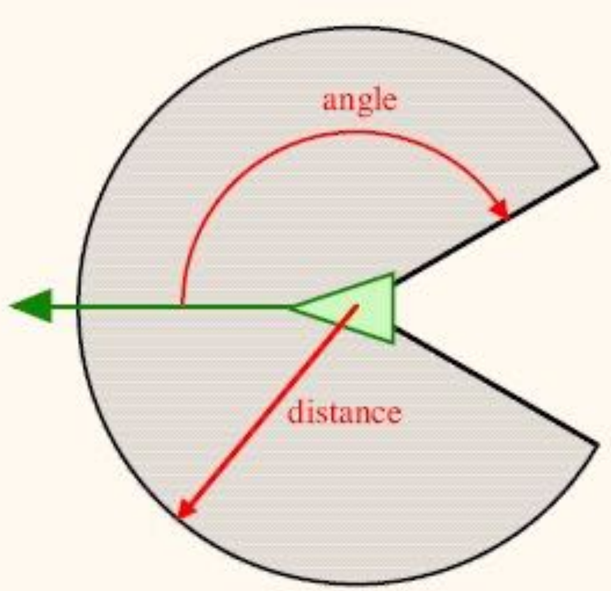


Group Steering Behaviors

- 群體活動的行為模擬必須考慮 NPC 間的相鄰關係
  - Neighboring
- 主要的行為包括：
  - Separation（分開）
  - Cohesion（凝聚）
  - Alignment（同向）

Local Neighborhood（鄰域）

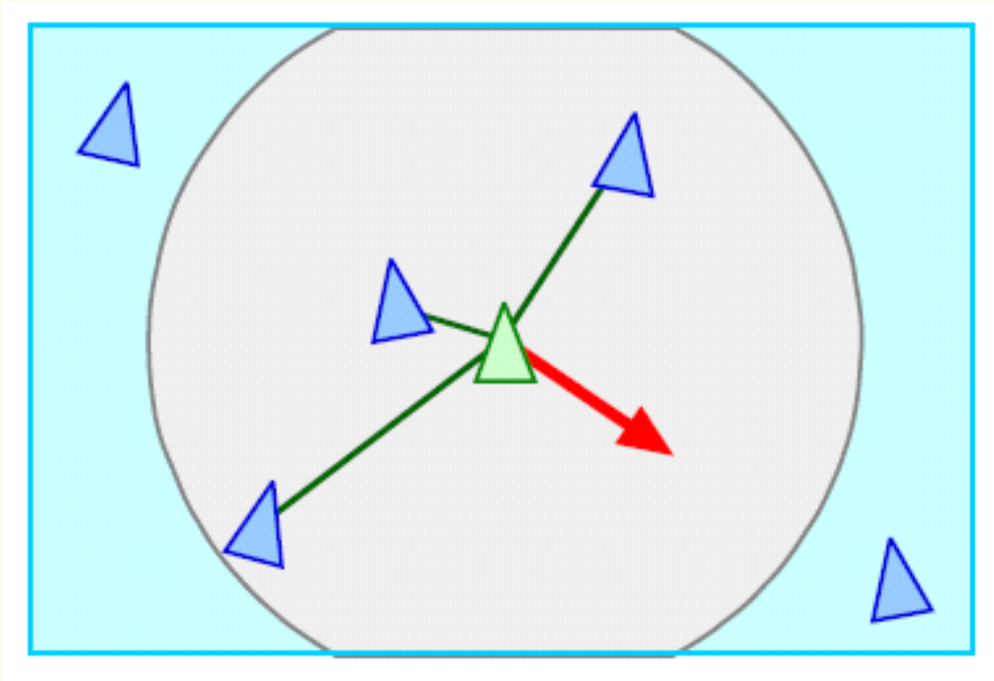
- Local neighborhood of a NPC
  - A distance（距離範圍）
  - Field-of-view（視野）
    - Angle



Neighborhood

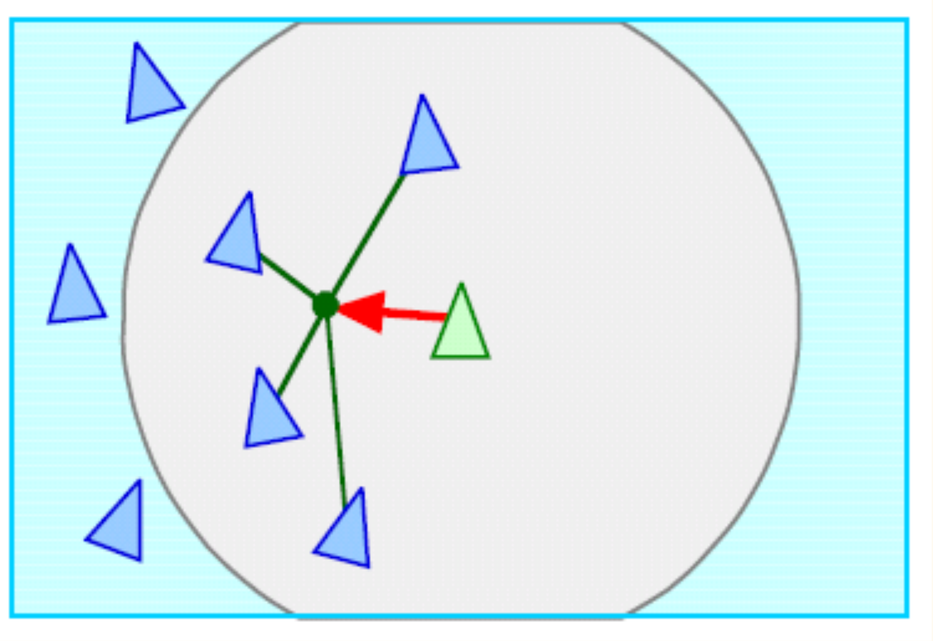
Separation（分開）

- 讓 NPC 與鄰近的夥伴持續保持一定的距離
  - 計算與鄰近的夥伴們的排斥力總合
    - Calculate the position vector for each nearby (計算位置向量)
    - Normalize it (轉成單位向量)
    - 利用距離因素加強排斥力權重
      - 1/distance
    - Sum the result forces (合力)
    - Negate it (反向)



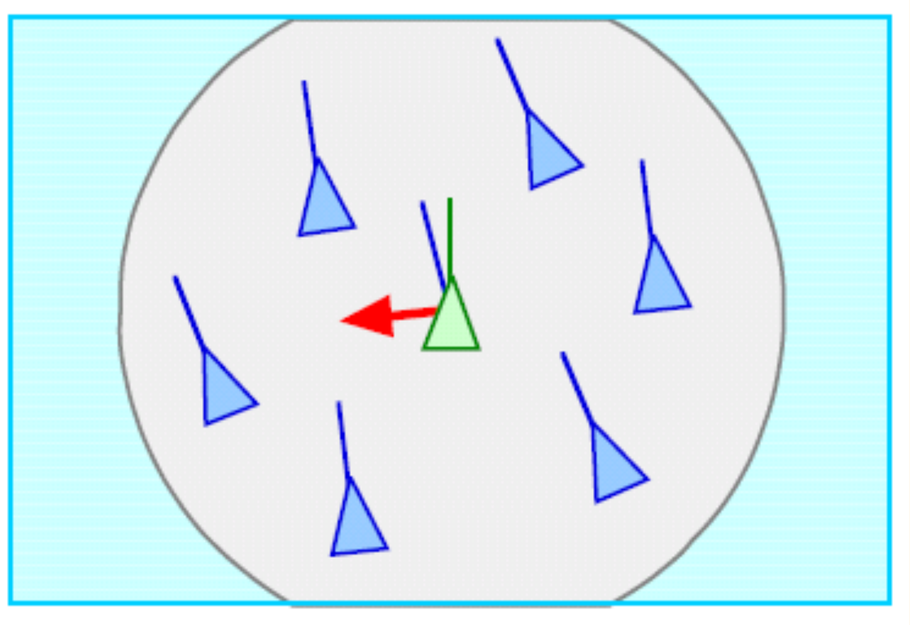
Cohesion（凝聚）

- 朝相鄰 NPC 夥伴移動
  - 計算凝聚力
    - Compute the average position of the others nearby
      - 計算相鄰 NPC 的中心位置
    - Apply “Seek” to the position



Alignment（保持同向）

- 讓 NPC 朝向與相鄰夥伴同方向前進
  - 計算方法
    - 將相鄰夥伴的們的速度方向平均
    - 當做未來的方向
    - 加上轉向力使 NPC 朝向未來方向



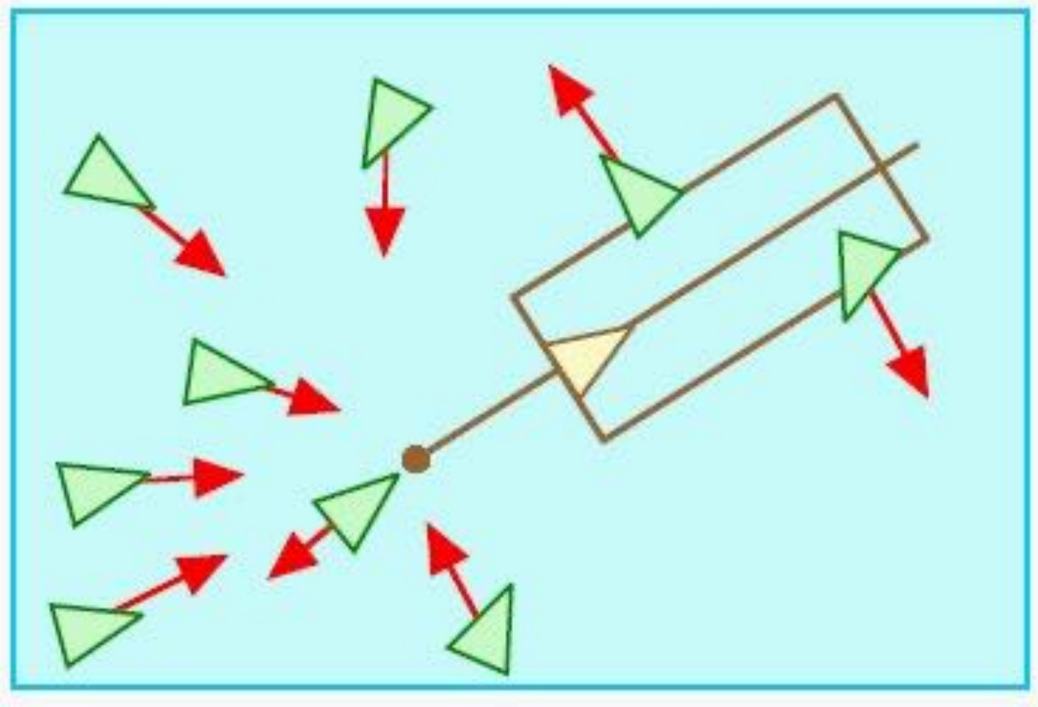
Flocking（群體運動）

- 參考資料
  - “Boids Model of Flocks”
  - [Reynolds 87]
- 三種行為的組合
  - Separation steering
  - Cohesion steering
  - Alignment steering
- 每項組合元件包括
  - A weight for each combination (權重)
  - A distance (鄰近範圍半徑)
  - An Angle (鄰近範圍的視角)

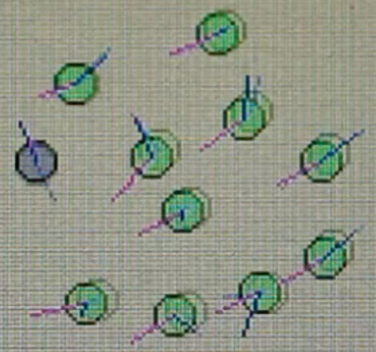
Flocking（群體運動）

Leader Following (領袖跟隨)

- Follow a leader
  - Stay with the leader
    - “Pursuit” behavior (Arrival style)
  - Stay out of the leader’s way
    - Defined as “next position” with an extension
    - “Evasion” behavior when inside the above area
  - “Separation” behavior for the followers



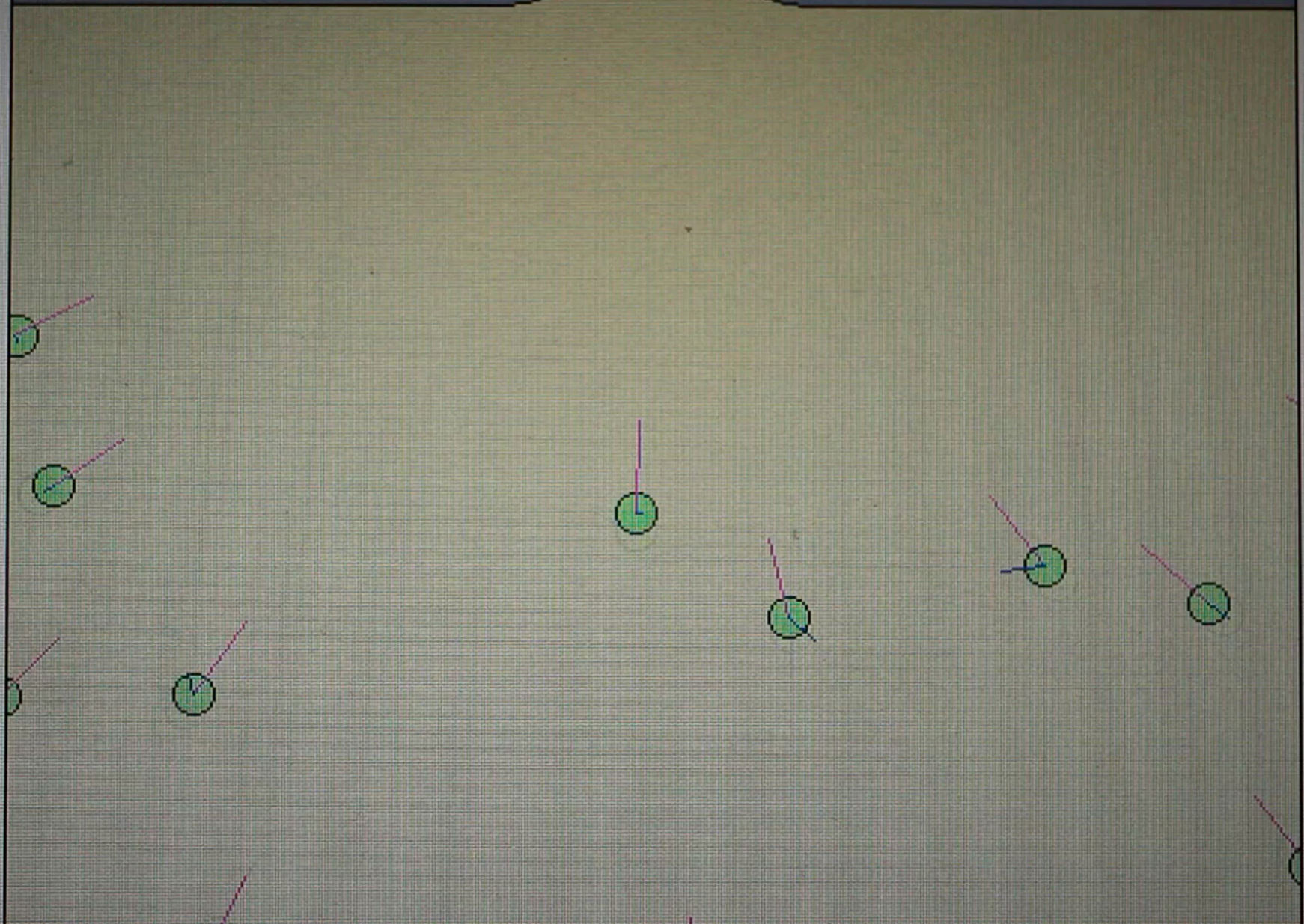
Leader Following（領袖跟隨）



Behavior Conclusion (行為的總合)

- A simple vehicle model with local neighborhood (車輛模型)
- 常見的行為模式：
  - Seek/Flee
  - Pursuit/Evasion
  - Offset pursuit
  - Arrival
  - Obstacle avoidance
  - Wander
  - Path following
  - Wall following
  - Containment
  - Unaligned collision avoidance
  - Separation/Cohesion/Alignment
  - Flocking
  - Leader following
- 視應用而定組成 NPC 的行為

Queue



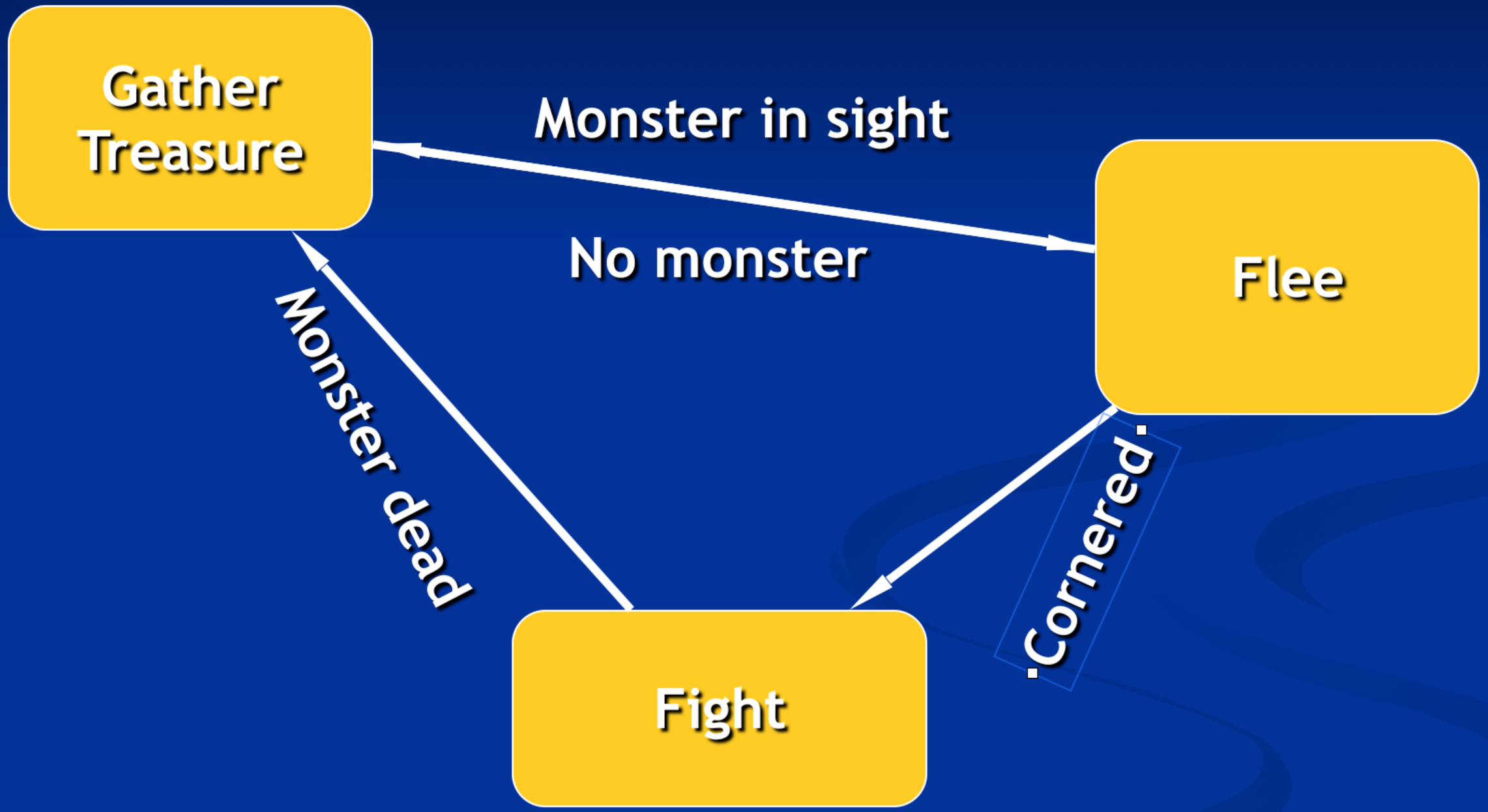
Finite State Machine

Introduction

- Finite State Machine (FSM, 有限狀態機) 是常使用的遊戲人工智慧技術
  - 簡單
  - 有效率
  - 容易延伸
  - 足夠處理大部分的狀態
- 理論 (基本)
  - A set of states, S, (狀態集合)
  - An input vocabulary, I, (輸入的情形)
  - Transition function, T(s, i), (對應函數)
    - Map a state and an input to another state

Introduction

- 實際應用
  - State
    - Behavior（行為）
  - Transition
    - Conditions（條件）
- 使用流程圖來分析
  - UML State chart
    - Arrow
      - Transition
    - Rectangle
      - State


Gather Treasure
Monster in sight
No monster
Flee
Monster dead
Cornered
Fight

FSM in Games

- NPC AI
- “Decision-Action” model (一種 “決策後行動” 模式)
- 條件
  - Players’ action (玩家的動作)
  - The other NPC’ actions (其中 NPC 的動作)
  - Some changes in the game world (遊戲世界的變化)

Implement FSM (FSM 的實作)

- Code-based FSM (以程式碼實作)
  - Simple Code One Up (直接寫在程式中)
  - Straightforward
  - Most common
  - Macro-assisted FSM Language (包裝成巨集式 FSM 語言)
- Data-Driven FSM (視 FSM 為 data 型式)
  - FSM Script Language

Example 1

```
void RunLogic(int *state)
{
    switch(*state)
    {
        case 0:  // Wander
            Wander();
            if (SeeEnemy()) *state = 1;
            if (Dead()) *state = 2;
            break;
        case 1:  // Attack
            Attack();
            *state = 0;
            if (Dead()) *state = 2;
            break;
        case 2:  // Dead
            SlowlyRot();
            break;
    }
}
```

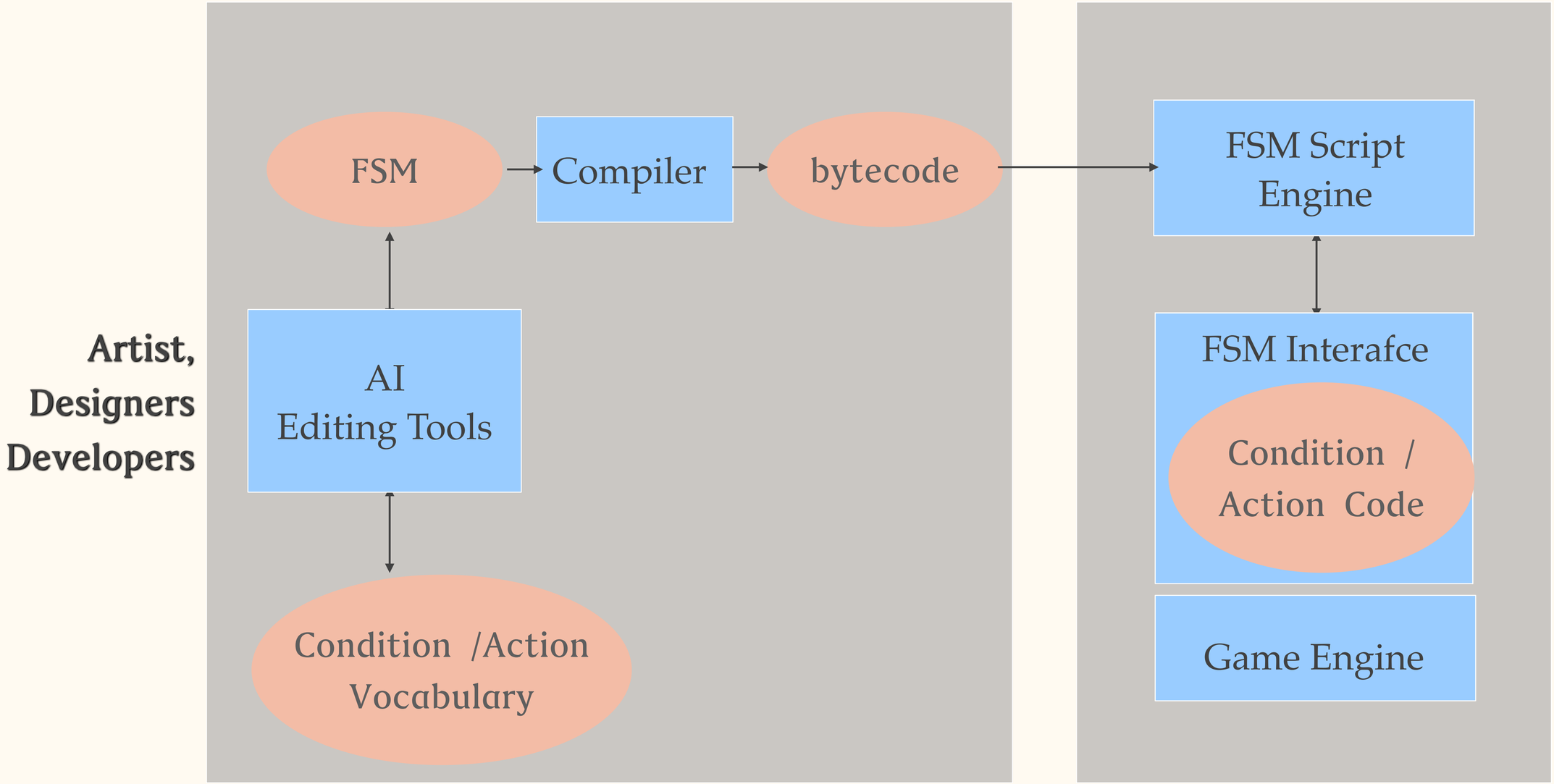
Example 2

```
void RunLogic(FSM *fsm)
{
    // Do action based on the state and determine next input
    input = 0;
    switch(fsm->GetStateID())
    {
        case 0:  // Wander
            Wander();
            if (SeeEnemy()) input = SEE_ENEMY;
            if (Dead()) input = DEAD;
            break;
        case 1:  // Attack
            Attack();
            input = WANDER;
            if (Dead()) input = DEAD;
            break;
        case 2:  // Dead
            SlowlyRot();
            break;
    }
    // DO state transition based on computed input
    fsm->StateTransition(input);
}
```

Data-driven FSM

- Scripting language
  - Text-based script file (撰寫文字碼 Script 檔)
  - 再編譯成：
    - C++整合到遊戲程式碼
    - Bytecode (中間位元碼)於遊戲中解譯
- 編輯器
  - Compiler
  - AI editing tool
- 遊戲端
  - FSM script engine
  - FSM interface

Data-driven FSM



AI Editing Tool for FSM (FSM 編輯器)

- Format
  - Pure text (純文字模式)
    - Syntax (語法)
  - Visual graph with text (圖形式模式)
    - Visual Programming
- Used by Designers, Artists, or Developers (給企劃或美術使用)
- Conditions & action vocabulary (條件與動作的標準詞彙)
  - For examples :
    - SeeEnemy / CloseToEnemy / Attack

FSM Interface (FSM 介面)

- 有助於詞彙與程式碼的對應與結合
  - Glue layer that implements the condition & action vocabulary in the game world
- Native conditions (內建的條件函式)
  - SeeEnemy(), CloseToEnemy(), ...
- Action library (內建的動作函式)
  - Attack(...), ...

FSM Script Benefits (FSM Script 優點)

- 提高生產能
- 給企劃人員使用
- 使用方便
- 擴充性好

Processing Models for FSMs (FSM 處理架構)

- Processing the FSMs
  - 計算目前狀態時各種情況的判定
  - 執行相關的行為
- 何時作？如何作？
  - 視遊戲情況而定
- Three common FSM processing models
  - Polling
  - Event-driven
  - Multithread

Polling Processing Model

- 依照一定的頻率依序處理所有 NPC 的 FSM
  - 鎖在遊戲的頻率上或是特定的更新頻率)
  - 有一次只能轉換狀態一次的限制
  - 限時結束以避免影響效能
- 優點
  - 直接而簡單
  - 易於除錯
- 缺點
  - 效率差
    - 有些 FSM 條件不必每次檢查
- 小心設計為重

Event-driven Processing Model

- Designed to prevent from wasted FSM processing (避免無謂的FSM處理)
  - An FSM is only processed when it's relevant
- 實作
  - A Publish-subscribe messaging system (Observer pattern)
  - 由引擎送相關且必要的訊息給相關的FSM
  - FSM只需訂閱(subscribes)會改變現有狀況的訊息
  - 只有在訊息發生時, 有訂閱的FSM才會被通知
- "As-needed" approach
  - Should be much more efficient than polling ?
- 要小心設計

Multithread Processing Model (多執行緒)

- Both polling & event-driven are serially processed.
- Multithread processing model
  - Each FSM is assigned to its own thread for processing.
  - All FSM processing is effectively concurrent and continuous. (同時執行)
  - Communication between threads must be thread-safe.
- 優點
  - 每個FSM可視為獨立的執行程序
  - 多執行緒架構
- 缺點
  - 小心太多執行緒的效能問題
  - 多執行緒程式開發不易

Interfacing with Game Engine

- FSMs 將複雜的行為邏輯包裝成
  - Decision, condition, action, ...
- Game engine 來執行之
  - Character animation, movements, sounds, ...
- 程式界面：
  - Code each action as a function
    - ie., FleeWolf()
  - Callbacks
    - Function pointers
    - ie., actionFunction[fleeWolf]()
  - Container method
    - ie., actionFunctions->FleeWolf();
    - DLL

An Example

```
class AArmyUnit : public FnCharacter
{
    ...
    void DoAttack(...);
}
```

繼承 Game Engine的人物系統

AI 模組

```
AArmyUnit *army;

army->Object(...);
army->MoveForward(dist, ...);
...
army->DoAttack(...);
```

使用 Game Engine的人物系統操控人物

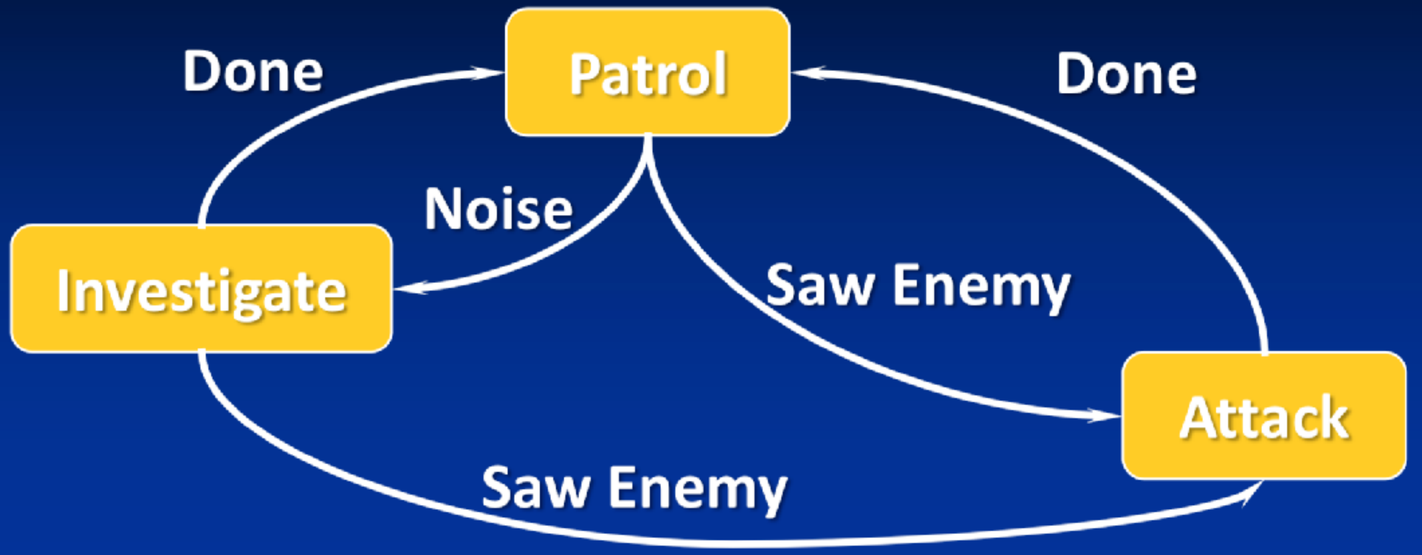
AI

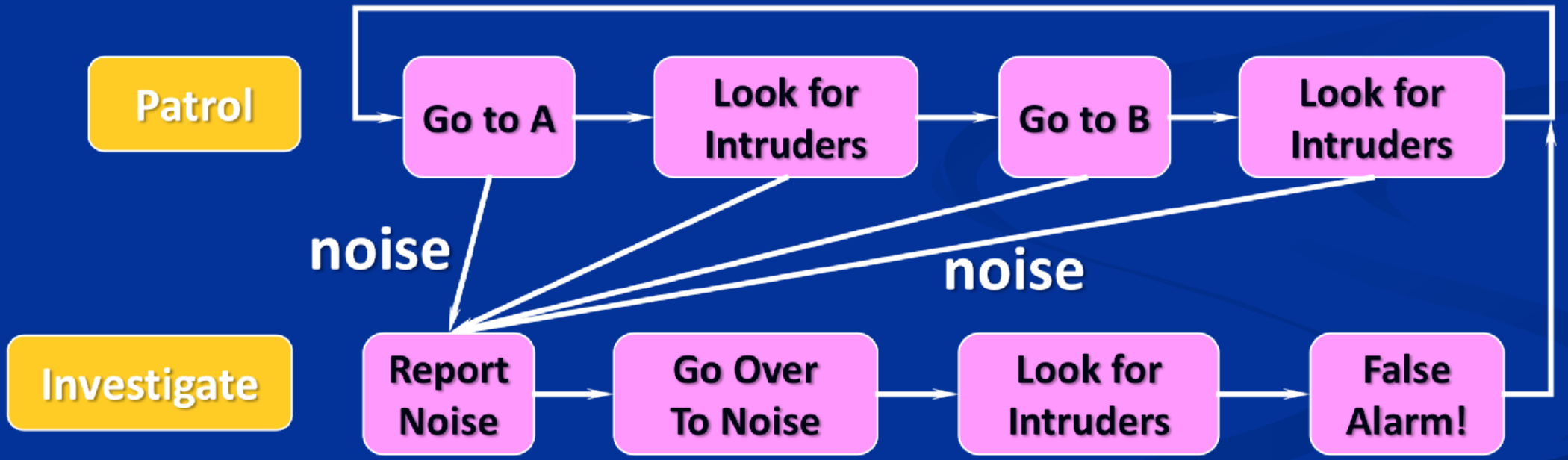
FSM Efficiency & Optimization

- 考量：
  - Time spent (所費的時間) & Frequency (頻率)
- 方法
  - Priority for each FSM (FSM 優先次序)
  - Different update frequency (不同的更新頻率)
- 負載平衡機構
  - Collecting statistics of past performance & extrapolating
- Time-bound for each FSM
- Do careful design at the design level
- Level-of-detail FSMs

Level-of-Detail FSM

- 在玩家沒有察覺變化下簡化 FSM
  - Outside the player's perceptual range (玩家認知範圍外)
- 主要設計關鍵：
  - Decide how many LOD levels (LOD 等級數量)
    - The approximation extent
    - How much development time available ?
  - LOD selection policy (LOD 切換原則)
    - The distance between the NPC with the player ?
    - If the NPC can "see" the player ?
  - Be careful the problem of "visible discontinuous behavior"
  - What kind of approximations (近似的種類)
    - Cheaper and less accurate solution


Done
Patrol
Done
Noise
Investigate
Saw Enemy
Attack
Saw Enemy


Patrol
Go to A
Look for Intruders
Go to B
Look for Intruders
noise
noise
Investigate
Report Noise
Go Over To Noise
Look for Intruders
False Alarm!