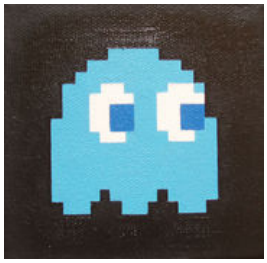
Artificial Intelligence 1: Finite State Machines



Introduction

For an AI agent to appear more interesting, and to present more of a challenge, it is likely to be required to change its behaviour at different times. For example, an enemy which just rotates on the spot firing a gun is pretty dull; equally a character which simply walks back and forth between two way-points is less than engaging; but a character which can choose between these two behaviours starts to get a bit more interesting.

A Finite State Machine (FSM) is a well-established method of representing a range of behaviours within a logical framework. This tutorial introduces the concept, and discusses how to apply it to an AI agent within the context of a game.

What is a Finite State Machine?

According to Wikipedia, a Finite State Machine is:

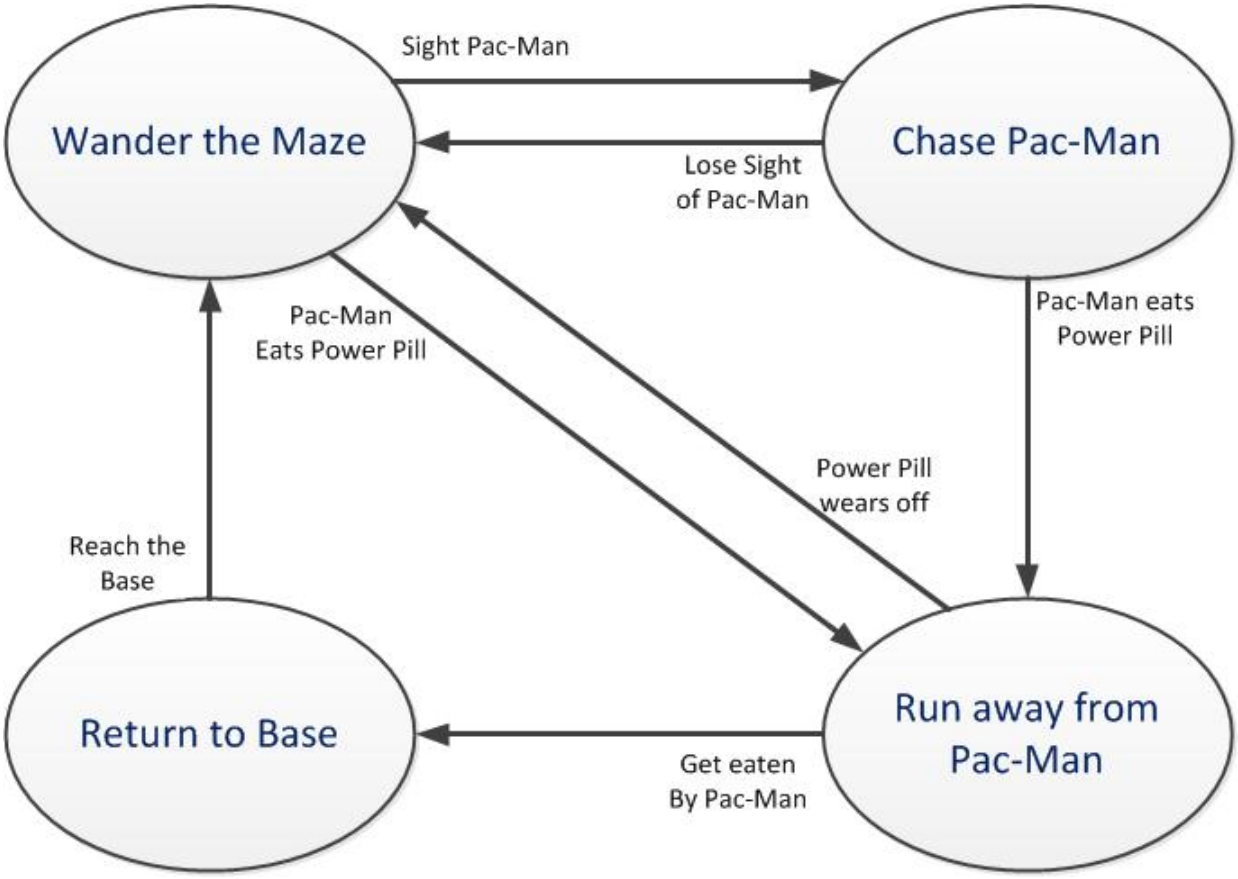
A behavioural model used to design computer programs. It is composed of a finite number of states associated to transitions. A transition is a set of actions that starts from one state and ends in another (or the same) state. A transition is started by a trigger, and a trigger can be an event or a condition.

As can be inferred from this definition, FSMs are used across a wide range of software engineering applications. In order to apply this approach to games technology, the best way to understand Finite State Machines is to think about an example. Well consider one of the ghosts in Pac-man.

The ghosts have four behaviours:

- Wander the maze
- Chase Pac-man
- Run away from Pac-man
- Return to the central base

These are the four states of the AI system. Each ghost can change its behaviour between these states depending on specific things happening in the game. For example, a ghost will change from state 3 to state 4 if it is eaten by Pac-Man. The act of changing state is known as the transition, and the event which has caused that change is known as the trigger. The four states, with their transitions



and triggers can be summed up in a state diagram.

A state machine can also be represented by a table, but in general, the state diagram is more intuitive, making it easier to consider how the system works as a whole. A table can be useful in checking whether any transitions or triggers may be missing from the design. Its also worth noting that the state machine is completely closed (i.e. all eventualities are covered), which is why the word Finite is used in the description.

Programming a State Machine

There are a number of ways of implementing a FSM in code. Choosing the most appropriate for your project is influenced by the expected complexity of the FSM, and how much the FSM is likely to evolve during development. Three methods are discussed in this section, in order of increasing flexibility and complexity.

Hard-coded switch statement

Fundamentally a state machine is a set of conditional statements, so it can be coded as a set of if-then statements, or as a switch statement. A code snippet may look something like this:

```
1 If CurrentState == STATE_1
2 {
3    State1Behaviour();
4    If CheckTrigger1to2()
5    {
6       SetState (STATE_2);
7    }
8 }
```

Hard coded switch

This is the FSM controller code; all of the transitional logic is centralised here. Notice that the actual AI behaviour is contained within a function such as State1Behaviour, and the algorithm to decide whether a particular transition has been triggered is contained in a function such as Check-Trigger1to2. The behaviour code, and the trigger algorithm, could be included in this piece of code rather than in separate functions, but thatll result in a very long function very quickly.

However, coding even a simple state machine (such as the Pac-Man example) as a switch statement will quickly become a tangled mess, which would be very difficult to change if the design evolves, or a new type of AI must be introduced.

Hard-coded State Pattern

A more extendible approach is to set up a basic State class, and a set of specific states which inherit their structure from the basic State class. At any time an AI agent has a state attached to it, when a transition occurs, the state is replaced with the new state. In this case, each state is responsible for determining when a transition should be triggered, so the FSM logic is distributed through the states, rather than existing within a centralised controller function.

In order to make the controller code more centralised, a FSM class may also be implemented which is used to transition between states when triggers occur. In that case a messaging system may be implemented, whereby the FSM is instructed to change states when a trigger is detected within a state behaviour.

Interpreted State Pattern

This is a data driven version of the state pattern. The data for how each state is connected, and how each transition is triggered, is contained in a data file, read by the game at start-up. This data is held in a table which can subsequently be modified at runtime if required.

As the data is in a file, the structure of the FSM can be changed without recompiling the code. In fact, if a sufficiently intuitive tool is developed to generate this data, then it could be put in the hands of the level designers, who could then tweak existing AI state machines, or generate new ones, without programmer input. Of course, if a new type of behaviour is required (ie a new state), or a new logical test for a trigger needs to be introduced, then additional code needs to be added to the game.

State Oscillation

Before moving on, a word of warning about defining the trigger functions in a FSM. Consider a two state FSM. The trigger to move from state A to state B occurs when the AI velocity is more than 10m/s. The trigger for moving from state B back to state A occurs when the velocity is less than or equal to 10m/s. If the AIs velocity is hovering around 10m/s, then the AI state could be switching back and forth from one frame of the game to the next, creating a jittery behaviour. This is known as state oscillation.

Trigger functions should be designed to avoid this happening. In the simple example above, it would be better to trigger the transition from state A to state B when the velocity becomes higher than 11m/s, and to trigger the reverse transition when it becomes lower than 9m/s. Introducing hysteresis like this into complementary trigger functions will reduce the chances of oscillation occurring.

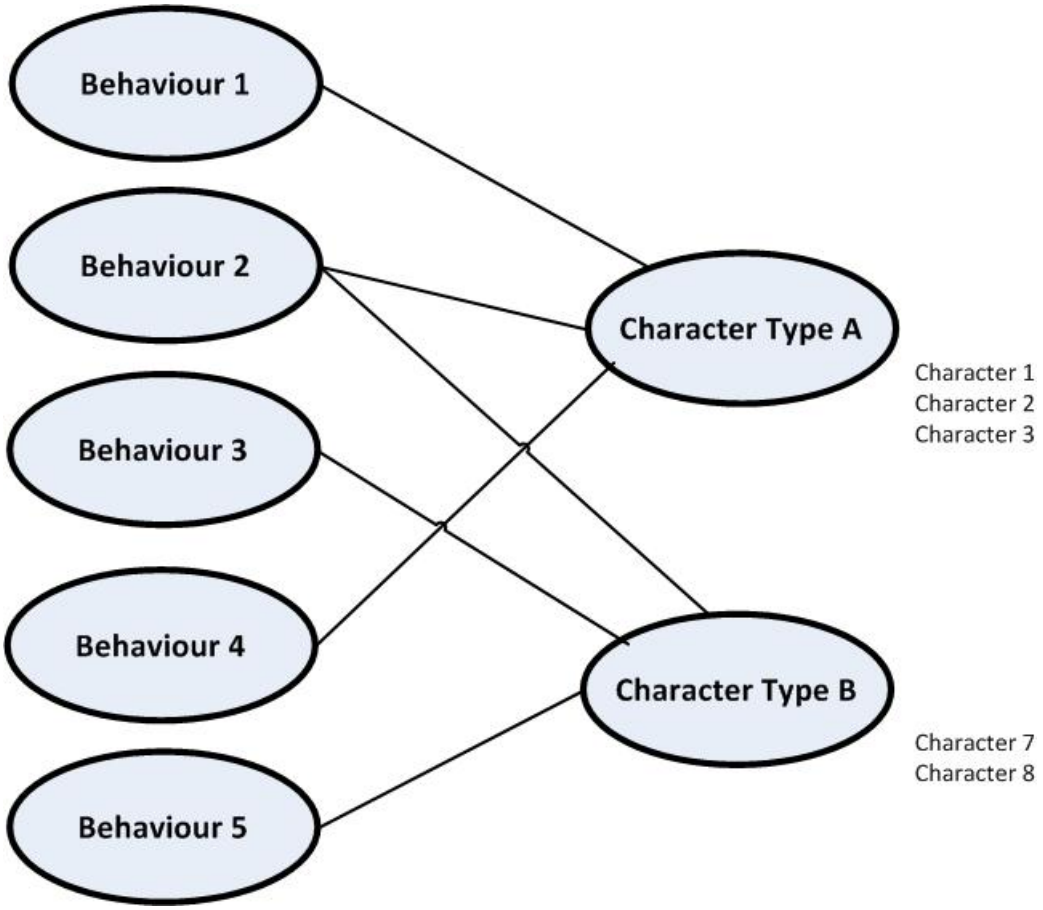
AI Behaviours and AI Types

When designing an AI system it is important to distinguish between AI behaviours and AI types. A well-designed AI system will consist of a set of defined behaviours which are then combined in various different combinations to make the different AI types in the game.

For example, in our Pac-Man example, the AI type is the ghost, and the four states are the behaviours; it should be possible to add a new type of AI which utilises a different combination of those

behaviours (ie a new state machine with different transitions and triggers) without having to rewrite any of the behavioural code.

If a data-driven approach is taken, such as the Interpreted State Pattern, then a designer should be able to add a new AI type to the game, using a new combination of existing triggers and states, with no further input from the software engineer.



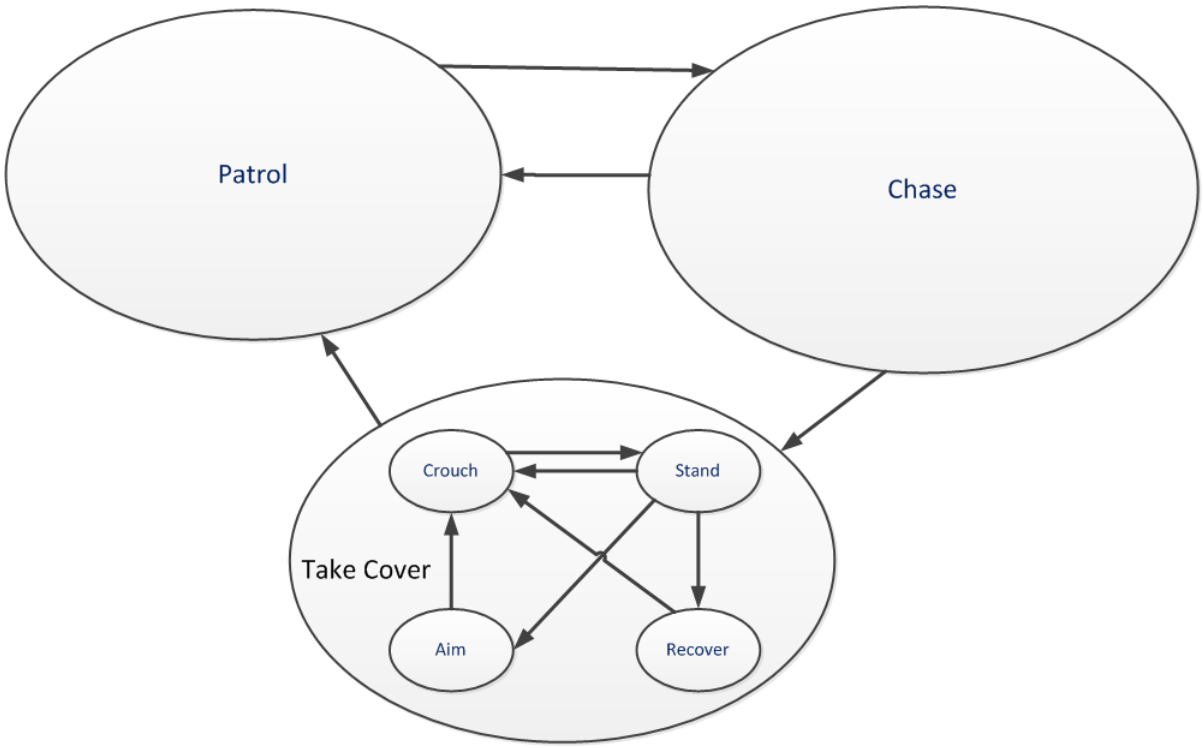
Hierarchical Finite State Machines

Using a Finite State Machine for a game AI agent has a couple of drawbacks:

- A FSM for a complex AI agent can become very complicated, and difficult to maintain for example think of a police character in an open world city-based game who can patrol on foot, pursue the player on foot, decide to get into a car, drive that car in a pursuit or a patrol, run away, try to arrest someone, etc.
- Simpler FSMs can appear predictable and repetitive to the player.

A solution to both these issues is to implement a hierarchical Finite State Machine, whereby each state in the higher level FSM is actually another FSM. The example of a police character may entail a high level FSM with behaviours such as driving vehicle, being a passenger in vehicle and on foot. Each of these states would then be a further FSM defining the various behaviours within that super-state. Those FSMs could then, in turn consist of a set of states, some of which are actually lower-level FSMs.

Note that the structure for such a hierarchy is not the same as the tree structure, with nodes and branches, which we have seen elsewhere. Each level of the hierarchy consists of a FSM with states, triggers and transitions. Focusing on a higher-level state will reveal a further FSM, consisting of states, triggers and transitions.



Fuzzy State Machines

A further way of reducing the predictability of the AI behaviours is to allow an AI agent to combine multiple behaviours at the same time. This is achieved through the use of Fuzzy Logic (rather than binary logic) to implement a Fuzzy State Machine (FuSM).

States in a FuSM are not restricted to being on or off; instead they can hold an intermediate value. This means that at any one time, more than one state may be active and to some degree be on and off. If we go back to our police character AI in the open city game, there may be a chasing the player state which can be combined with either the on foot state or the in vehicle state.

Perhaps counter-intuitively, this approach can actually reduce the complexity of the state machine, while adding more complexity to the behaviour. A FuSM will typically require fewer states, due to the possibility of combinations. In our example, the policeman would need separate states ina finite state machine for chasing player on foot, chasing player in vehicle, patrolling on foot, patrolling in vehicle, fleeing on foot, fleeing in vehicle, etc. Utilising a Fuzzy State Machine allows the combination of on foot or in vehicle with chasing, patrolling or fleeing.

The engine code providing the means to change states based on fuzzy logic is more complex than the straightforward code needed for binary decisions. However, assuming that it is implemented in a suitable manner, the state machines can be expanded upon, or rearranged, with no changes required on the engine.

Implementation

The example code shows a simple FSM encoded as a hard-coded switch statement.

The header file for the state class:

```
1 #pragma once
2
3 #include <stdlib.h>
4 #include <iostream>
5
```

```
6  class State
7  {
8  public:
9     State(){};
10    ~State(){};
11
12    inline bool getWantsFood() { return wantsFood;}
13    inline bool getWantsWalkies() { return wantsWalkies;}
14    inline bool getIsDead() { return isDead;}
15
16    inline void setProperties(bool setupWantsFood,
17              bool setupWantsWalkies, bool setupIsDead)
18       {
19          wantsFood = setupWantsFood;
20          wantsWalkies = setupWantsWalkies;
21          isDead = setupIsDead;
22       }
23
24 private:
25
26    // Here are some simple state properties to get you started. Feel
27    // free to add more! FSMs in the wild are always more complicated
28    // than this example - so make some complex ones of your own!
29    bool wantsFood;
30    bool wantsWalkies;
31    bool isDead;
32 };
```

State.h

The header file for the puppy class:

```
1  #pragma once
2
3  #include <stdlib.h>
4  #include <iostream>
5
6  #include "State.h"
7
8  using namespace std;
9
10 class Puppy
11 {
12 public:
13    Puppy();
14    ~Puppy(){};
15
16    void updateState();
17    inline void eatFood() { energy++; }       // Increases energy
18    inline void haveWalk() { energy--; }      // Decreases energy
19
20    inline State getCurrentState() { return currentState; }
21
22 private:
23    // Energy variable. You might add more variables to make the state
24    // check more complex and interesting. Remember, this is
25    // a simplistic example, have fun with it!
26    int energy;
27
28    //Puppy's current state
```

```
    State currentState;

    //A selection of states reflecting some combinations of state
    //properties. You could arrange these as an array if you like.
    //Similarly, you can add more states - zombified might be
    //a state with wantsFood and isDead set to true!
    State hungry;
    State hunting;
    State bouncy;
    State dead;
};
```

Puppy.h

The cpp file for the puppy class:

```
#include "Puppy.h"

using namespace std;

Puppy::Puppy()
{
    hungry.setProperties(true, false, false);       // Setting
    hunting.setProperties(true, true, false);       // up
    bouncy.setProperties(false, true, false);       // our
    dead.setProperties(false, false, true);         // states

    energy = 3;                     // Default energy value at creation

    updateState();                  // Updates currentState
}

void Puppy::updateState()
{
    if(energy < 1)
    {
        currentState = dead;
        return;
    }
    else if(energy < 4)
    {
        currentState = hungry;
        return;
    }
    else if(energy < 7)
    {
        currentState = hunting;
        return;
    }
    else
    {
        currentState = bouncy;
        return;
    }
}
```

Puppy.cpp

The main cpp file:

```
#pragma once
```

```
2
3  #include <stdlib.h>
4  #include <iostream>
5
6  #include "Puppy.h"
7
8  using namespace std;
9
10 void main()
11 {
12    Puppy wuffles;
13
14    // Declare our input variable
15    int playerInput;
16    // Holding variable - just here to let you follow what's happening
17    int moveOn;
18
19    // While Wuffles is alive...
20    while(!wuffles.getCurrentState().getIsDead())
21    {
22       playerInput = 0;
23
24       while(playerInput < 1 || playerInput > 3)
25       {
26          // Main checks to see if Wuffles currently wants food
27          if(wuffles.getCurrentState().getWantsFood())
28          {
29             cout << "Wuffles looks like he wants something
30                to eat... \n \n";
31          }
32          // Main checks to see if Wuffles currently wants a walk
33          if(wuffles.getCurrentState().getWantsWalkies())
34          {
35             cout << "Wuffles looks like he wants to get out
36                the house... \n \n";
37          }
38          // Note that these are not mutually exclusive. currentState
39          // might support both of these statements - in which case
40          // both are true.
41
42          cout << "If you want to take Wuffles for walkies, enter 1!\n";
43          cout << "If you want to feed Wuffles, enter 2!\n";
44          cout << "If you want to take Wuffles hunting, enter 3!\n\n";
45
46
47          cin >> playerInput;
48
49          system("cls");
50       }
51
52       if(playerInput == 1)
53       {
54          wuffles.haveWalk();
55          cout << "You took Wuffles for a walk! He expended energy!\n";
56       }
57       if(playerInput == 2)
58       {
59          wuffles.eatFood();
```

```
60          cout << "You fed Wuffles! He gained energy!\n";
61      }
62      if(playerInput == 3)
63      {
64          wuffles.haveWalk();
65          wuffles.eatFood();
66          cout << "You took Wuffles out hunting!" << "\n\n" <<
67              "He expended energy to find a bunny rabbit,
68              but gained energy by eating it!\n";
69      }
70
71      cout << "\n" << "Enter a number to move on!" << "\n";
72
73      cin >> moveOn;
74
75      wuffles.updateState();
76
77      system("cls");
78  }
79
80  cout << "You let Wuffles die... How could you? ;_;\n\n";
81
82  cout << "Enter a number to close the program,
83          you heartless monster. \n";
84
85  int wayout;
86
87  cin >> wayout;
88
89  return;
90 }
```

main.cpp

Exercises

- Draw a state diagram for a new AI type in Pac-Man a SuperGhost that cant be eaten but chases Pac-Man when the power pill is eaten, and returns to base if Pac-Man eats a piece of fruit. This should be possible without introducing any new states to the four already used by the normal Ghost.
- Choose a favourite video game, and try drawing the state machine for one of the AI types in that game. If you have chosen a modern game, you may need to think about a specific part of the AI types behaviour.
- Implement the Pac-Man ghost state machine. The behaviours should be implemented as a text output along the lines of I am wandering the maze, with keyboard inputs providing the triggers.