a i - j u n k i e

State-Driven Game Agent Design

Note: *the text for this tutorial comprises part of the second chapter of the book* Programming Game AI by Example*. Its appearance online is a cunningly disguised attempt to tempt you into purchasing said book. I have no shame. Buy the book, you'll enjoy it*. J

Finite state machines, or FSMs as they are usually referred to, have for many years been the AI coder's instrument of choice to imbue a game agent with the illusion of intelligence. You will find FSMs of one kind or another in just about every game to hit the shelves since the early days of video games, and despite the increasing popularity of more esoteric agent architectures, they are going to be around for a long time to come. Here are just some of the reasons why:

- **They are quick and simple to code.** There are many ways of programming a finite state machine and almost all of them are reasonably simple to implement. You'll see several alternatives described in this article together with the pros and cons of using them.
- **They are easy to debug.** Because a game agent's behavior is broken down into easily manageable chunks, if an agent starts acting strangely, it can be debugged by adding tracer code to each state. In this way, the AI programmer can easily follow the sequence of events that precedes the buggy behavior and take action accordingly.
- **They have little computational overhead.** Finite state machines use hardly any precious processor time because they essentially follow hard- coded rules. There is no real "thinking" involved beyond the if-*this*-then-*that* sort of thought process.
- **They are intuitive.** It's human nature to think about things as being in one state or another and we often refer to ourselves as being in such and such a state. How many times have you "got yourself into a state" or found yourself in "the right state of mind"? Humans don't really work like finite state machines of course, but sometimes we find it useful to think of our behavior in this way. Similarly, it is fairly easy to break down a game agent's behavior into a number of states and to create the rules required for manipulating them. For the same reason, finite state machines also make it easy for you to discuss the design of your AI with non-programmers (with game producers and level designers for example), providing improved communication and exchange of ideas.
- **They are flexible.** A game agent's finite state machine can easily be adjusted and tweaked by the programmer to provide the behavior required by the game designer. It's also a simple matter to expand the scope of an agent's behavior by adding new states and rules. In addition, as your AI skills grow you'll find that finite state machines provide a solid backbone with which you can combine other techniques such as fuzzy logic or neural networks.

What Exactly Is a Finite State Machine?

Historically, a finite state machine is a rigidly formalized device used by mathematicians to solve problems. The most famous finite state machine is probably Alan Turing's hypothetical device: the Turing machine, which he wrote about in his 1936 paper, "On Computable Numbers." This was a machine presaging modern-day programmable computers that could perform any logical operation by reading, writing, and erasing symbols on an infinitely long strip of tape. Fortunately, as AI programmers, we can forgo the formal mathematical definition of a finite state machine; a descriptive one will suffice:

> *A finite state machine is a device, or a model of a device, which has a finite number of states it can be in at any given time and can operate on input to either make transitions from one state to another or to cause an output or action to take place. A finite state machine can only be in one state at any moment in time.*

The idea behind a finite state machine, therefore, is to decompose an object's behavior into easily manageable "chunks" or states. The light switch on your wall, for example, is a very simple finite state machine. It has two states: on and off. Transitions between states are made by the input of your finger. By flicking the switch up it makes the transition from off to on, and by flicking the switch down it makes the transition from on to off. There is no output or action associated with the off state (unless you consider the bulb being off as an action), but when it is in the on state electricity is allowed to flow through the switch and light up your room via the filament in a lightbulb. See Figure 2.1.

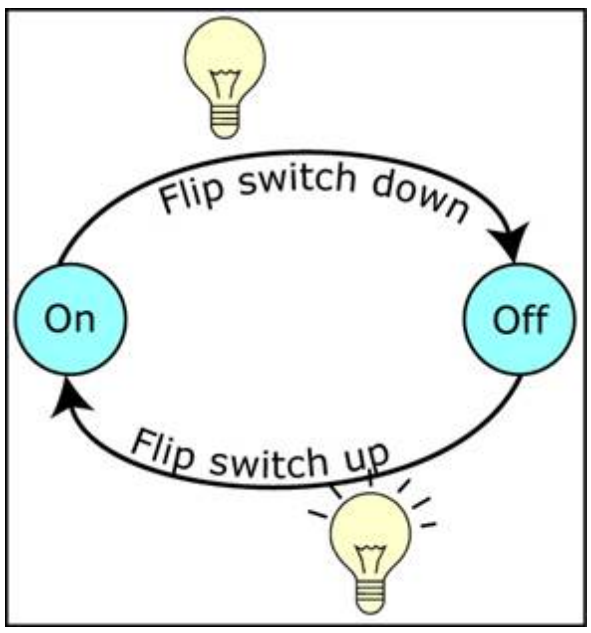

Figure 2.1. A light switch is a finite state machine. (Note that the switches are reversed in Europe and many other parts of the world.)

Of course, the behavior of a game agent is usually much more complex than a lightbulb (thank goodness!). Here are some examples of how finite state machines have been used in games.

- The ghosts' behavior in Pac-Man is implemented as a finite state machine. There is one `Evade` state, which is the same for all ghosts, and then each ghost has its own `Chase` state, the actions of which are implemented differently for each ghost. The input of the player eating one of the power pills is the condition for the transition from `Chase` to `Evade`. The input of a timer running down is the condition for the transition from `Evade` to `Chase`.
- Quake-style bots are implemented as finite state machines. They have states such as `FindArmor`, `FindHealth`, `SeekCover`, and `RunAway`. Even the weapons in Quake implement their own mini finite state machines. For example, a rocket may implement states such as `Move`, `TouchObject`, and `Die`.
- Players in sports simulations such as the soccer game FIFA2002 are implemented as state machines. They have states such as `Strike`, `Dribble`, `ChaseBall`, and `MarkPlayer`. In addition, the teams themselves are often implemented as FSMs and can have states such as `KickOff`, `Defend`, or `WalkOutOnField`.
- The NPCs (non-player characters) in RTSs (real-time strategy games) such as Warcraft make use of finite state machines. They have states such as `MoveToPosition`, `Patrol`, and `FollowPath`.

Implementing a Finite State Machine

There are a number of ways of implementing finite state machines. A naive approach is to use a series of `if-then` statements or the slightly tidier mechanism of a `switch` statement. Using a switch with an enumerated type to represent the states looks something like this:

```
enum StateType{state_RunAway, state_Patrol, state_Attack};

void Agent::UpdateState(StateType CurrentState)
{
  switch(CurrentState)
  {
  case state_RunAway:

     EvadeEnemy();

     if (Safe())

    if (Safe())
    {
      ChangeState(state_Patrol);
    }

    break;

case state_Patrol:

  FollowPatrolPath();

  if (Threatened())
  {
    if (StrongerThanEnemy())
    {
      ChangeState(state_Attack);
    }
    else
    {
      ChangeState(state_RunAway);
    }
  }

  break;

case state_Attack:

  if (WeakerThanEnemy())
```

```
    {
       ChangeState(state_RunAway);
    }
    else
    {
      BashEnemyOverHead();
    }

    break;

  }//end switch
}
```

Although at first glance this approach seems reasonable, when applied practically to anything more complicated than the simplest of game objects, the switch/if-then solution becomes a monster lurking in the shadows waiting to pounce. As more states and conditions are added, this sort of structure ends up looking like spaghetti very quickly, making the program flow difficult to understand and creating a debugging nightmare. In addition, it's inflexible and difficult to extend beyond the scope of its original design, should that be desirable… and as we all know, it most often is. Unless you are designing a state machine to implement very simple behavior (or you are a genius), you will almost certainly find yourself first tweaking the agent to cope with unplanned-for circumstances before honing the behavior to get the results you thought you were going to get when you first planned out the state machine!

Additionally, as an AI coder, you will often require that a state perform a specific action (or actions) when it's initially entered or when the state is exited. For example, when an agent enters the state `RunAway` you may want it to wave its arms in the air and scream "Arghhhhhhh!" When it finally escapes and changes state to `Patrol`, you may want it to emit a sigh, wipe its forehead, and say "Phew!" These are actions that only occur when the `RunAway` state is entered or exited and not during the usual update step. Consequently, this additional functionality must ideally be built into your state machine architecture. To do this within the framework of a switch or if-then architecture would be accompanied by lots of teeth grinding and waves of nausea, and produce very ugly code indeed.

State Transition Tables

A better mechanism for organizing states and affecting state transitions is a *state transition table*. This is just what it says it is: a table of conditions and the states those conditions lead to. Table 2.1 shows an example of the mapping for the states and conditions shown in the previous example.

Table 2.1. A simple state transition table

Current State	Condition	State Transition
Runaway	Safe	Patrol
Attack	WeakerThanEnemy	RunAway
Patrol	Threatened AND StrongerThanEnemy	Attack
Patrol	Threatened AND WeakerThanEnemy	RunAway

This table can be queried by an agent at regular intervals, enabling it to make any necessary state transitions based on the stimulus it receives from the game environment. Each state can be modeled as a separate object or function existing external to the agent, providing a clean and flexible architecture. One that is much less prone to spaghettification than the if-then/switch approach discussed in the previous section.

Someone once told me a vivid and silly visualization can help people to understand an abstract concept. Let's see if it works…

Imagine a robot kitten. It's shiny yet cute, has wire for whiskers and a slot in its stomach where cartridges — analogous to its states — can be plugged in. Each of these cartridges is programmed with logic, enabling the kitten to perform a specific set of actions. Each set of actions encodes a different behavior; for example, `play_with_string`, `eat_fish`, or `poo_on_carpet`. Without a cartridge stuffed inside its belly the kitten is an inanimate metallic sculpture, only able to sit there and look cute… in a Metal Mickey kind of way.

The kitten is very dexterous and has the ability to autonomously exchange its cartridge for another if instructed to do so. By providing the rules that dictate when a cartridge should be switched, it's possible to string together sequences of cartridge insertions permitting the creation of all sorts of interesting and complicated behavior. These rules are programmed onto a tiny chip situated inside the kitten's head, which is analogous to the state transition table we discussed earlier. The chip communicates with the kitten's internal functions to retrieve the information necessary to process the rules (such as how hungry Kitty is or how playful it's feeling). As a result, the state transition chip can be programmed with rules like:

```
IF Kitty_Hungry AND NOT Kitty_Playful SWITCH_CARTRIDGE eat_fish
```

All the rules in the table are tested each time step and instructions are sent to Kitty to switch cartridges accordingly.

This type of architecture is very flexible, making it easy to expand the kitten's repertoire by adding new cartridges. Each time a new cartridge is added, the owner is only required to take a screwdriver to the kitten's head in order to remove and reprogram the state transition rule chip. It is not necessary to interfere with any other internal circuitry.

Embedded Rules

An alternative approach is to *embed the rules for the state transitions within the states themselves*. Applying this concept to Robo-Kitty, the state transition chip can be dispensed with and the rules moved directly into the cartridges. For instance, the cartridge for `play_with_string` can monitor the kitty's level of hunger and instruct it to switch cartridges for the `eat_fish` cartridge when it senses hunger rising. In turn the `eat_fish` cartridge can monitor the kitten's bowel and instruct it to switch to the `poo_on_carpet` cartridge when it senses poo levels are running dangerously high.

Although each cartridge may be aware of the existence of any of the other cartridges, each is a self-contained unit and not reliant on any external logic to decide whether or not it should allow itself to be swapped for an alternative. As a consequence, it's a straightforward

matter to add states or even to swap the whole set of cartridges for a completely new set (maybe ones that make little Kitty behave like a raptor). There's no need to take a screwdriver to the kitten's head, only to a few of the cartridges themselves.

Let's take a look at how this approach is implemented within the context of a video game. Just like Kitty's cartridges, states are encapsulated as objects and contain the logic required to facilitate state transitions. In addition, all state objects share a common interface: a pure virtual class named `State`. Here's a version that provides a simple interface:

```
class State
{
public:

  virtual void Execute (Troll* troll) = 0;

};
```

Now imagine a `Troll` class that has member variables for attributes such as health, anger, stamina, etc., and an interface allowing a client to query and adjust those values. A `Troll` can be given the functionality of a finite state machine by adding a pointer to an instance of a derived object of the `State` class, and a method permitting a client to change the instance the pointer is pointing to.

```
class Troll
{
  /* ATTRIBUTES OMITTED */

  State* m_pCurrentState;

public:

  /* INTERFACE TO ATTRIBUTES OMITTED */

  void Update()
  {
    m_pCurrentState->Execute(this);
  }

  void ChangeState(const State* pNewState)
  {
    delete m_pCurrentState;
    m_pCurrentState = pNewState;
  }
};
```

When the `Update` method of a `Troll` is called, it in turn calls the `Execute` method of the current state type with the this pointer. The current state may then use the `Troll` interface to query its owner, to adjust its owner's attributes, or to effect a state transition. In other words, how a `Troll` behaves when updated can be made completely dependent on the logic in its current state. This is best illustrated with an example, so let's create a couple of states to enable a troll to run away from enemies when it feels threatened and to sleep when it feels safe.

```
//----------------------------------State_Runaway
class State_RunAway : public State
{
public:

  void Execute(Troll* troll)
  {
    if (troll->isSafe())
    {
      troll->ChangeState(new State_Sleep());
    }
    else
    {
      troll->MoveAwayFromEnemy();
    }
  }
};

//----------------------------------State_Sleep
class State_Sleep : public State
{
public:

  void Execute(Troll* troll)
  {
    if (troll->isThreatened())
    {
      troll->ChangeState(new State_RunAway())
    }

    else
    {
      troll->Snore();
    }
  }
};
```

As you can see, when updated, a troll will behave differently depending on which of the states `m_pCurrentState` points to. Both states are encapsulated as objects and both provide the rules effecting state transition. All very neat and tidy.

This architecture is known as the *state design pattern* and provides an elegant way of implementing state-driven behavior. Although this is a departure from the mathematical formalization of an FSM, it is intuitive, simple to code, and easily extensible. It also makes it

extremely easy to add enter and exit actions to each state; all you have to do is create `Enter` and `Exit` methods and adjust the agent's `ChangeState` method accordingly. You'll see the code that does exactly this very shortly.
