Finite State Machines (FSM)
Finite state machines as a control technique in Artificial Intelligence (AI)
Level: beginner to intermediate

Author: Jason Brownlee
June 2002

Forward

Finite state machine is a technique I have been hearing about for some time, probably since I first got interested in first person shooter (FPS) computer games. I never really thought about what the term meant, all I knew was that it was related to the way enemies work in these types of games. As I have progressed through my Information Technology course at university I have found myself increasingly looking back on concepts and techniques I have previously come across with questions, finite state machines being one of these topics.

Finite state machines are said to have been “so widely used” and “so simple”, yet they have not been covered in my course so far in programming or artificial intelligence subjects. I turned to common knowledge resources on the Internet (forever my jump start for new technologies and techniques) for a suitable definition within the scope of artificial intelligence, and was not satisfied. This essay is the product of my research on the topic of finite state machines in the context of artificial intelligence as a control technique, and through that research my goal was to learn something and through writing this essay hopefully be able to teach something.

Introduction

The intent of this essay is to provide a useful and practical introduction of the technique of Finite State Machines (FSM) within the context of artificial intelligence (AI) as a control technique. The emphasis in this essay will be on practicality both in definition and explanation, rather than an emphasis on heavy theoretical and mathematical concepts behind the technique.

This essay will start with a light theory section describing the technique in terms of its elements and usage. The section will introduce the main types of finite state machine and popular enhancements to the basic concept.

The second section will provide two “real world” examples from the computer game domain. These examples will provide insight into how the technique could be used to model specific systems and the type of control it can provide

The final section will take a detailed look at the finite state machine framework implemented in a production quality and commercially released product. It will provide insight into usage of finite state machines in a broader system and how that system could be implemented to support multiple concurrent finite state machines in the same environment.

Section 1: Background
Finite State Machines (FSM), also known as Finite State Automation (FSA), at their simplest, are models of the behaviors of a system or a complex object, with a limited number of defined conditions or modes, where mode transitions change with circumstance.

Finite state machines consist of 4 main elements;

- states which define behavior and may produce actions
- state transitions which are movement from one state to another
- rules or conditions which must be met to allow a state transition
- input events which are either externally or internally generated, which may possibly trigger rules and lead to state transitions

A finite state machine must have an initial state which provides a starting point, and a current state which remembers the product of the last state transition. Received input events act as triggers, which cause an evaluation of some kind of the rules that govern the transitions from the current state to other states. The best way to visualize a FSM is to think of it as a flow chart or a directed graph of states, though as will be shown; there are more accurate abstract modeling techniques that can be used.

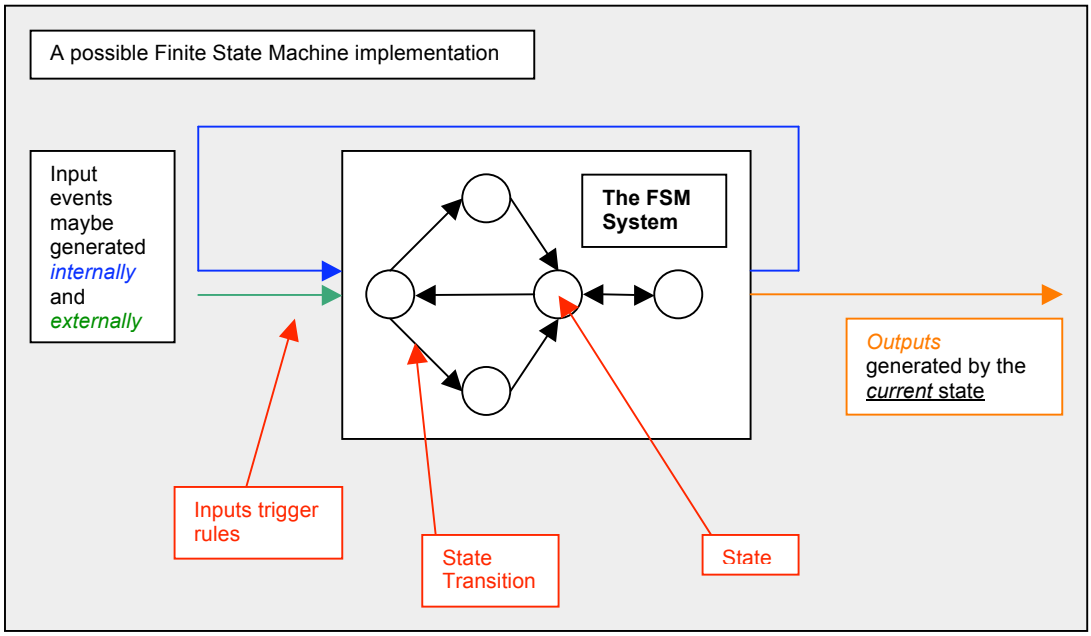

Figure 1.1 – A possible finite state machine control system implementation

FSM is typically used as a type of control system where knowledge is represented in the states, and actions are constrained by rules.

"*…One of the most fascinating things about FSMs is that the very same design techniques can be used for designing Visual Basic programs, logic circuits or firmware for a microcontroller. Many computers and microprocessor chips have, at their hearts, a FSM.*" [1]

Finite state machines are an adopted artificial intelligence technique which originated in the field of mathematics, initially used for language representation. It is closely related to other fundamental knowledge representation techniques which are worth mentioning, such as semantic networks [5] and an extension of semantic networks called state space [5].

Semantic networks were proposed to represent meaning and relationships of English words. A graph is constructed where nodes represent concepts and edges the relationships. State space is an extension on the idea of semantic networks, where a node denotes a valid state and the edges transitions between states. State space, unlike FSM, requires both an initial state and a goal state, and is typically used in problem solving domains where a sequence of actions is required for solving the overall problem (sequence from initial to goal states). Like FSM, state space has rules which constrain state transitions, and are triggered by input events.

Like any rule based systems, if all the antecedent(s) of a rule are true, then the rule is triggered. It is possible for multiple rules to be triggered, and in the area of reasoning systems, this is called a conflict set. There can only be one transition from the current state, so a consistent conflict resolution strategy is required to select only one of the triggered rules to fire and thus performing a state transition.

This brings us to two main types of FSM. The original simple FSM is what's known as deterministic, meaning that given an input and the current state, the state transition can be predicted. An extension on the concept at the opposite end is a non-deterministic finite state machine. This is where given the current state; the state transition is not predictable. It may be the case that multiple inputs are received at various times, means the transition from the current state to another state cannot be known until the inputs are received (event driven).

An implementation of a deterministic finite state machine may see the firing of the first rule that is triggered. This may be ideal for many problem domains, but for computer games, easily predictable behavior is usually not a wanted feature as it tends to remove the "fun-factor" in the game.

"*…a player feels like they are playing against a realistic simulation of intelligence, and not against a reproduction of a sequence of actions.*" [2]

The "sequence" which is one of the key benefits of FSM, should not be blindingly obvious in computer games. There are a number of extensions to FSM and workarounds for "mixing up" the sequence to make it harder to predict actions. One of these non-deterministic approaches involves the application of another proven artificial intelligence technique; Fuzzy Logic, called Fuzzy State Machines (FuSM).

Just like finite state machines there is a lot of flexibility when implementing a fuzzy state machine. A fuzzy value can be applied to various state transitions. When a conflict set is encountered the higher the fuzzy value for a transition, the higher the likelihood of the state transition. This allows the specification of a fuzzy priority to state transitions.

An implementation of FuSM may involve the assignment of fuzzy values to various inputs to represent the degree an input is defined. The fuzzy system would use these weighted input values in the evaluation of rules, triggering only state transitions whose assessed value is above a specified threshold.

Another approach for converting a deterministic FSM into a non-deterministic FSM would be to simply use a random number generator to select a triggered rule. It may not be necessary to implement a deterministic finite state machine to have a perceived level of unpredictability. This can be achieved by a system or object that has a large number of defined states and a complex mesh of transitions, giving the appearance of being unpredictable.

It is important to understand the difference between a state and an action. When designing a computer program, larger functionality are decomposed into a number of smaller actions or activities. This is done so that each can be defined in a function, making the overall solution modular, and easier to maintain. FSM is similar in that it's a decomposition of the behaviors of a

system or object, and even a state can be decomposed into sub-states. The difference is a state may involve one or more actions.

Example 1: a moveUnit() action may be used by both the evadeEnemy state and the attackEnemy state.

Example 2: the evadeEnemy state may consist of many actions, some evaluations, some movement directives, and some actions which can change the entities own state. If the entity was cornered for example, there may be a state transition from evadeEnemy to attackEnemy, where the act of being cornered is the trigger.

The best way I like to think of the terms is an action is an activity that accomplishes something like an evaluation or a movement, and a state is a collection of actions that are used when in a particular mode. A state is the circumstance of a thing, its condition, and the actions are the attributes of that state. It provides the ability to limit the scope of actions or the amount of knowledge to only that required for the current state.

There are two main methods for handling where to generate the outputs for a finite state machine. They are called a Moore Machine and a Mearly Machine, named after their respective authors.

A Moore Machine is a type of finite state machine where the outputs are generate as products of the states. In the below example the states define what to do; such as apply power to the light globe.

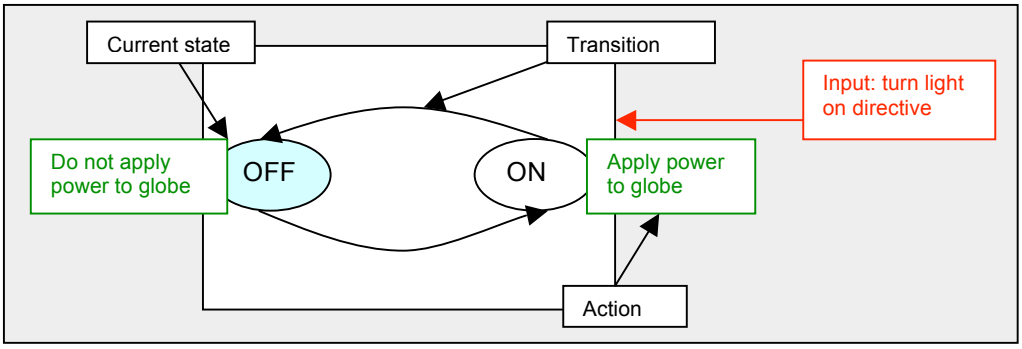

Figure 1.2 – a light system example of a Moore Machine

A Mearly Machine, unlike a Moore Machine is a type of finite state machine where the outputs are generated as products of the transition between states. In below example the light is affected by the process of changing states.

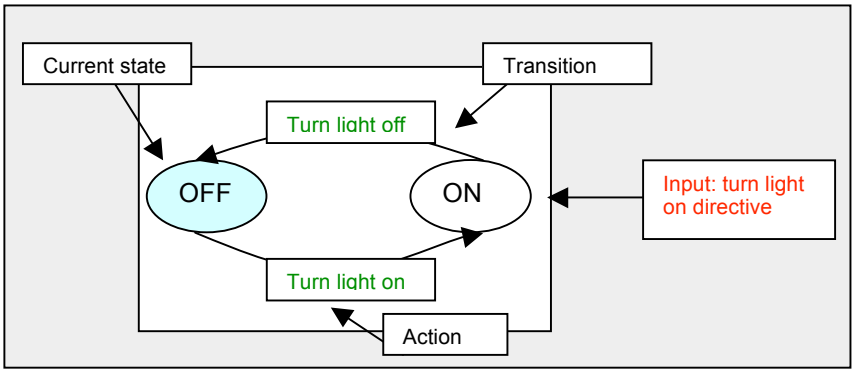

Figure 1.3 – a light system example of a Mearly Machine

Finite state machines is not a new technique, it has been around for a long time. The concept of decomposition should be familiar to people with programming or design experience. There are a number of abstract modeling techniques that may help or spark understanding in the definition and design of a finite state machine, most come from the area of design or mathematics.

- **State Transition Diagram**: also called a bubble diagram, shows the relationships between states and inputs that cause state transitions [1]
- **State-Action-Decision Diagram**: simply a flow diagram with the addition of bubbles that show waiting for external inputs [1]
- **Statechart Diagrams**: a form of UML notation used to show behavior of an individual object as a number of states, and transitions between those states. [3]
- **Hierarchical Task Analysis (HTA)**: though it does not look at states, HTA is a task decomposition technique that looks at the way a task can be split into subtasks, and the order in which they are performed [4]

Advantages of FSM

- Their simplicity make it easy for inexperienced developers to implement with little to no extra knowledge (low entry level)
- Predictability (in deterministic FSM), given a set of inputs and a known current state, the state transition can be predicted, allowing for easy testing
- Due to their simplicity, FSMs are quick to design, quick to implement and quick in execution
- FSM is an old knowledge representation and system modeling technique, and its been around for a long time, as such it is well proven even as an artificial intelligence technique, with lots of examples to learn from
- FSMs are relatively flexible. There are a number of ways to implement a FSM based system in terms of topology, and it is easy to incorporate many other techniques
- Easy to transfer from a meaningful abstract representation to a coded implementation
- Low processor overhead; well suited to domains where execution time is shared between modules or subsystems. Only the code for the current state need be executed, and perhaps a small amount of logic to determine the current state.
- Easy determination of reachability of a state, when represented in an abstract form, it is immediately obvious whether a state is achievable from another state, and what is required to achieve the state

Disadvantages of FSM

- The predictable nature of deterministic FSMs can be unwanted in some domains such as computer games (solution may be non-deterministic FSM).
- Larger systems implemented using a FSM can be difficult to manage and maintain without a well thought out design. The state transitions can cause a fair degree of “spaghetti- factor” when trying to follow the line of execution
- Not suited to all problem domains, should only be used when a systems behavior can be decomposed into separate states with well defined conditions for state transitions. This means that all states, transitions and conditions need to be known up front and be well defined
- The conditions for state transitions are ridged, meaning they are fixed (this can be over come by using a Fuzzy State Machine (FuSM))

Like most techniques, heuristics for when and how to implement finite state machines are subjective and problem specific. It is clear that FSMs are well suited to problems domains that are easily expressed using a flow chart and possess a set of well defined states and rules to govern state transitions.

For a broader discussion of finite state machines I recommend you read; *Finite State Machines – Making simple work of complex functions* [1].

Section 2: Practical Analysis

A Practical Analysis of FSM within the domain of first-person shooter (FPS) computer game

The intent of this section is to use a computer game to illustrate the conceptual workings of a FSM based on a practical rather than theoretical implementation. I will not attempt to provide insight into how the computer game works or even many specifics (code) of the implementation in the computer game. The FSMs discussed in this section have been coded, tested and released in production code, providing real world examples, within the domain of a first person computer game.

This section will provide two examples of finite state machines from a first-person computer game created by id Software [6] called Quake [7]. The reason I chose this product as the basis for the analysis was due the fact that game engine code has been released publicly under the GNU General Public License (GPL) [8] [9]. The other reason for this decision was because at the time the game was released, it was cutting-edge and had its code licensed by other company's that produced further highly-popular titles such as Unreal and Half-Life, thus proving the success of the original product.

We are going to start with the analysis of something very simple. We will see that even through mapping the states of a simple projectile like a rocket, we can learn more about the very nature of FSM. I make no claim at being an expert in regard to Quake game code (this was my first excursion through it), so forgive me if my interpretations based on code do not directly map.

A rocket in Quake is a projectile fired from the Rocket Launcher weapon/item which may be possessed and operated by a human player.

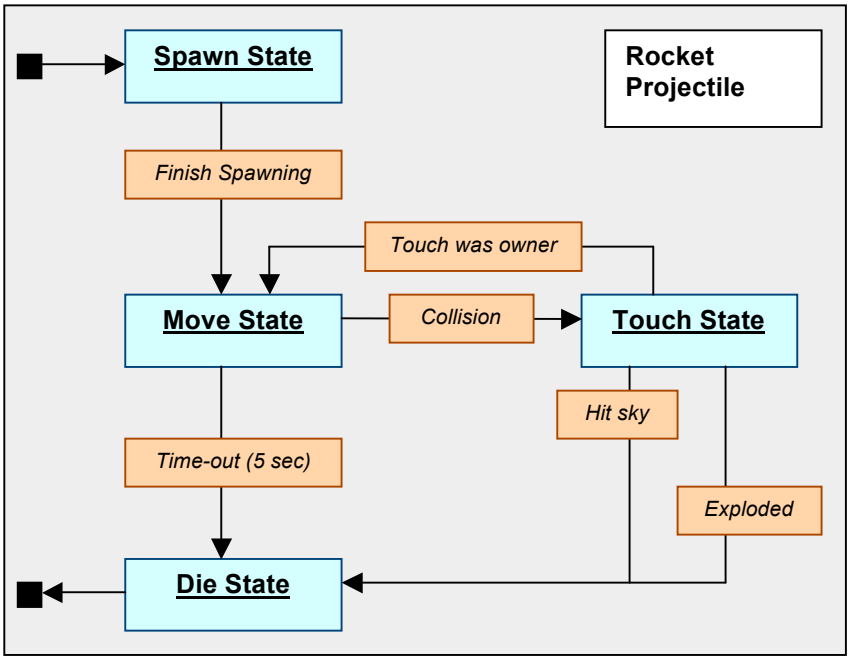

Figure 2.1 – State transition representation of a rocket projectile from Quake

Rather than restricting myself and confusing the reader by using a formal notation as mentioned in the first section of this document, the finite state machine has been represented using an approach very similar to a State Transition Diagram. The blue boxes are the states, the orange the triggers and the arrows are the state transitions. The black boxes show the entry point and exit points of the system.

The diagram shows the full life cycle of the rocket projectile within the game. It is interesting to note that the projectile is spawned into existence as the product of an action of another FSM, namely that of the “rocket launcher” from its “fire” action. When the projectile instance dies it is removed from the game, and no longer exists.

This representation is a subjective interpretation of code. Another valid representation may break the “touch state” down further into touch and explode states. Personally I view explode as an action or effect performed by the rocket object in its touch state.

Another interesting note is that when the projectile is in its touch state, one of its effectors is to attempt to damage whatever it is touching. If it succeeds in damaging another entity in the game world, the damage action becomes an input which can trigger a state change of the effected entity into another state.

Quake makes extensive use of FSMs as a control mechanism governing the entities that exist in the game world. This has been provided by an interesting framework which is tightly related to the way FSM work in the computer game. For further discussion of this framework, please see section three.

Let’s take a look at a slightly more advanced FSM from Quake. A Shambler is a big bad monster entity from the single player component of Quake. Its mission in life is to kill the player, once it is aware of the player.

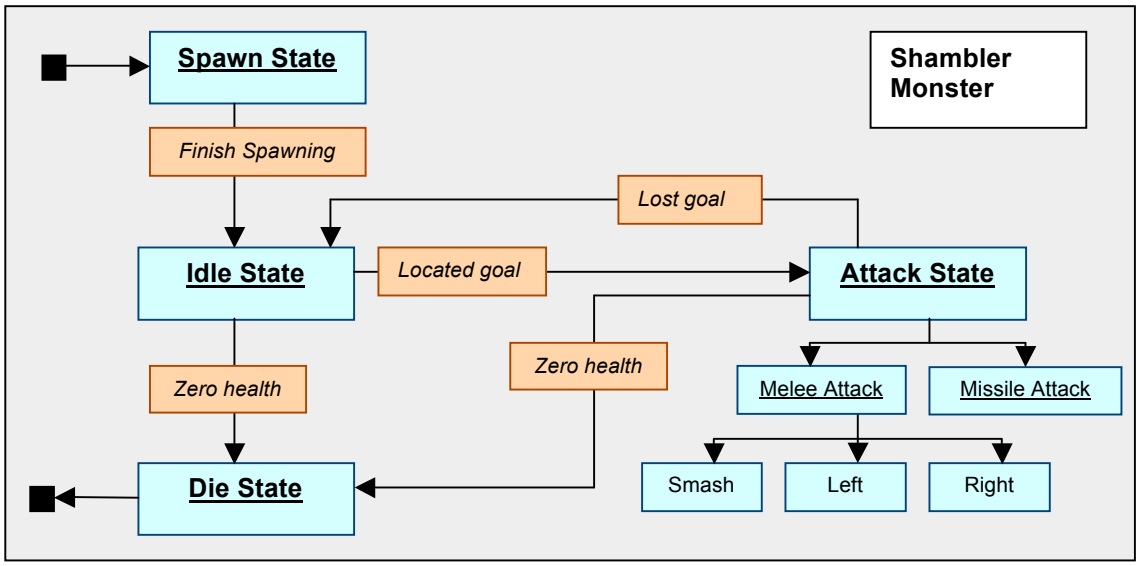

Figure 2.2 – State transition representation of a Shambler monster from Quake

Like the rocket projectile, this entity has an initial state (spawn state), and the system ends when the entity dies (die state). There are only four identified main states, but the Shambler is a good example that illustrates the ability to have a hierarchy of sub-states. Though the sub-states in this example could be considered actions of the “attack state”, they are also sub-states because the monster can only perform one (or be in one) of them per execution of the attack state.

When in the attack state, the Shambler instance makes a decision based on evaluated inputs whether to perform a melee (close up) or missile (long distance) style of attack on its goal. Upon selecting a melee type attack (melee attack state), the inputs are further evaluated, along with a random number to select a melee attack type (smash with both arms, smash with left arm or smash with right arm).

The use of a random number in the selection of a melee attack sub-state adds a level of unpredictability to the selection. Each level in the hierarchy could be considered a sub-finite state machine of the greater monster entity, and in this case the sub-FSM of the melee attack state could be classified as non-deterministic.

It is important to understand the use of layered or hierarchical finite state machines, because when used as seen in the Shambler monster it allows far more complex behaviors. This technique is heavily used in Quake by all entities in the game world. Because of this, a lot of code has been abstracted to be easily be used in many different places, actions such as movement, visibility assessments and so on.

This example has been greatly simplified to make it more readable. An example of this is in the triggers which cause the state transitions. When a state transition occurs from “Attack State” to “Idle State”, the trigger has been simplified as “lost goal”. It is true that the transition occurs due to the loss of the goal entity, but a goal can be lost by the Shambler in a number of ways evaluated at different points in the code, including time-out and damage from another entity.

Another key point regarding this example is the use of goals as the primary motivator for the FSM. This technique has not been discussed, though it is an example of the power and flexibility of FSM as a control technique. As well as possessing a hierarchy of finite state machines, the high level FSM is driven by the entities desire to locate a goal, and attack its goal. The goal is usually a human player or even another monster in the game world. It should be noted that whilst the monster is in its idle state, as well as just standing around it is also looking for goals to walk to (for roaming).

Quake did not provide the best single player experience imaginable, though it was and still is fun and addictive, both key attributes of the games success. It is good example, and a good learning tool that can show the power of both very simple finite state machines such as the rocket projectile, and slightly more complex FSM made up of a hierarchy of FSM and motivated by goals, such as the Shambler monster.

Section 3: A Finite State Machine Framework

In this section we will take a look at the implementation of finite state machines from broader perspective. We will investigate how a finite state machine can be implemented, and take a look at a framework that can facilitate multiple FSMs in a simulated real-time environment.

A single FSM on its own is of little use; therefore we need to investigate implementations from a broader view to understand where a single FSM plugs in. We will analyze portions of the finite state machine framework from the computer game Quake, in attempt to understand how to make use of the technique in a real world application.

One possible way to implement finite state machines is to have a controller of some type which acts as switch box. When the thread of execution swings around to execute code of the FSM, it is pointed at the controller which evaluates or determines the current state, usually through the use of a switch (case) statement or if-then-else statements. Once the current state is determined, the code for that state is executed, actions performed and possibly state transitions for the next time the FSM is executed. The controller may be a simple switch statement evaluating an integer, but an implementation may see the controller performing some pre-processing of inputs and triggering of state transitions before-hand.

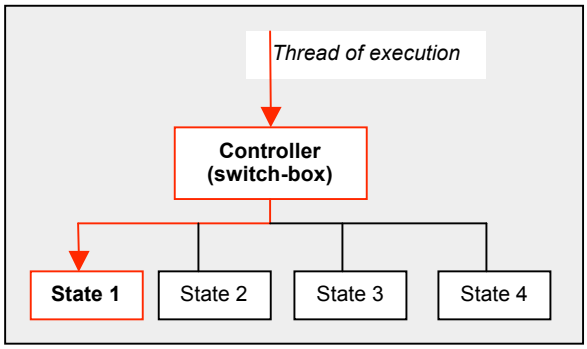

Figure 3.1 – A FSM implementation where the controller acts as a switch box to determine which state to run. The red line denotes the thread of execution.

The implementation that the programmers at id Software have chosen could almost be considered to have Object-Oriented (OO) feel (though the implementation is not OO). As mentioned before, the game world is populated by entities, and as such a generic entity structure is used as their basis. Each entity in the collection of entities is provided with some execution time by calling its “think” function. The entity is executed by the game code in what could be described as a polymorphic manner. Entities have a common interface (because they are all the same data structure), and this interface consists of function pointers which are used to execute entity specific and entity non-specific code as either action outputs or input events for the FSM.

An example; most entities are affected by damage. Damage can be inflicted by many things like a rocket projectile for example. When a damage trigger is transmitted to another entity, its pain function pointer is called, thus triggering a state transition of the effected entity into possibly a death or attack state. Key point: The damage inflicted is an input to the FSM, which may act as a trigger for a state transition.

In essence the same switch-box technique described in figure 3.1 is used, where the entities base data structure provides function pointers which act as the “switches”. When an entity is given a chance to execute its state, its “think” function pointer is called. If previously a damage input was received, the entity may have had a state transition into its “die state”. When the thread of execution runs, the objects “die state” code is then run (via a polymorphic call of the entities think function), removing the entity instance from the game world.

Rather than further explain using theory, let's take a look at a practical example from the game Quake. A dog is a simple monster that wishes's to attack the player or any other monster that angers it, much like the previously seen Shambler monster.

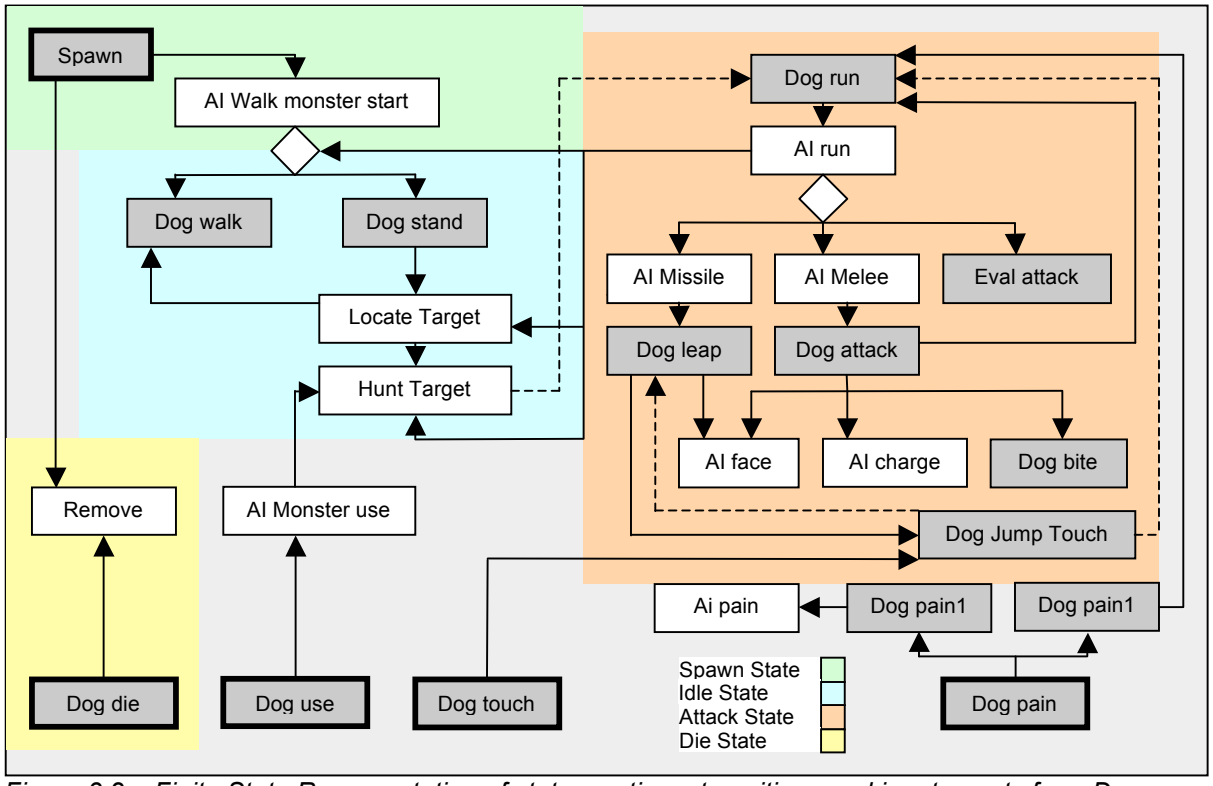

Figure 3.2 – Finite State Representation of states, actions, transitions and input events for a Dog monster in Quake

Figure 3.2 shows the four main states of a Dog monster from Quake. Superimposed over that are the main actions performed in each of those states. Some map directly to functions, others map to chains of function calls. The representation clearly shows the five main input event actions with dark borders which are called as outputs of other finite state machines. There are two types of actions displayed, the gray background represents the dog monster specific code, and the white represent framework code used by all monsters.

The figure is a good diagrammatical tool for showing what monster specific code needs to be written and where it fits into the framework. It is not a complete representation, for example, in regard to the code executed after spawning, there are other "swim" and "fly" start actions used by some monsters that can swim and fly as opposed to our walking dog. Also there is a third attack sub-state called "slide attack" which is not relevant to the dog monster.

It can be seen that although the design of a FSM on paper may be clear cut and seem easy to implement, when it comes to implementing a larger number of them, such as all the monsters in a computer game, it is easy and probably necessary to blur the boundaries of the states. In figure 3.2 we do not see well defined functions representing states as may have been expected, instead functions are shared by states and FSMs. In this example it is clear that core actions have been decomposed and their functionality abstracted to be incorporated into the framework to be used by all monsters of the game. It is a nice approach for speed of execution, maintainability and code reuse, but poor due to a moderate level of complexity (time must be invested to trace the execution path).

The finite machine is provided with execution time through its think function. It evaluates inputs from its inputs from of the game world, but can directly receive specific events as input from the output of actions performed by other FSMs. These include a touch, use, pain and die events. These events can trigger state changes of the effected finite machine, for example, as seen in Figure 3.2 a touch input event is a collision determined by the game as it advances the game physics. When received in the above example, and if the dog monster has a valid “touch” sensor function specified (which may not always be the case), it will run the code in that function and possibly have a transition to its missile attack sub state or to its attack state for a revaluation of its attack sub-states.

It is becoming clear that a FSM in this domain is very useful as a control mechanism and when used on a larger scale as seen, it is powerful. This example shows that a FSM framework can provide the ability for a simple multi-agent system, where each FSM system could be considered an agent (intelligent – uses AI techniques, autonomous – acts independently). The FSMs have sensor functions implemented specifically to handle expected events, and also has effector functions which are simply the actions performed in the game world.

Conclusion

We started out with a definition of finite state machine, learning that it can be used as a control technique for a system, describing states or behaviors of that system, and defining rules or conditions that govern transitions from the systems current state to another state.

Section one showed that a finite state machine is considered to be deterministic which means its actions are easily predictable. A number of extensions to finite state machines such as random selection of transitions, and fuzzy state machines shows us another common type of FSM called non-determinist where the systems actions were not as predictable, giving a better appearance of intelligence.

Next we took a close look at a simple real world implementation and learned how a rocket projectile could be modeled using FSM. This lead us to a more advanced example where we saw the behavior of a semi-intelligent monster modeled using a combination of hierarchical finite state machines and the use of goals as a primary motivator. This showed us a real implementation of a non-deterministic finite state machine where the monsters “melee attack” state was selected using a combination of inputs and a randomly generated number.

Finally we took a broader look at the Quake computer game implementation and examined the framework used to support the monsters that populate the games single player experience. We learned how a polymorphic approach was used for the execution of a monsters “current state” and that input events were also processed using polymorphism allowing any entity finite state machine’s actions to directly effect entities. We saw how a simple framework that supported a variety of state machines could be considered a simple multi-agent system where each individual state machine simple had to plug in relevant monster specific code into the framework.

Finite state machines are a simple and effective artificial intelligence technique for controlling a system and providing the appearance of intelligence. We learned that that the perceived appearance of intelligence is more important than actual intelligence, and that finite state machines are able to provide this perception. This was proven through practical analysis of a computer game, which is a very unforgiving domain when it comes to quality of both product and game playing experience.

Referenced

[1] D. Gibson, *Finite State Machines – Making simple work of complex functions*, SPLat Control Pty. Ltd [http://www.microconsultants.com/tips/fsm/fsmartcl.htm], 1999

[2] A. E. Collins, *Evaluating the performance of AI techniques on the domain of computer games,* [www.dcs.shef.ac.uk/~u8aec/com301], 2001

[3] B. Bruegge & a. H. Dutoit, *Object-Oriented Software Engineering – Conquering Complex and Changing Systems*, Prentice Hall, 2000

[4] A.Dix, J. Finlay, G. Abowd, R. Beale, *Human-Computer Interaction 2nd Edition*, Prentice Hall, 1998

[5] P.K. Winston, *Artificial Intelligence*, Addison-Wesley, 1993

[6] id Software, website: http://www.idsoftware.com

[7] Quake, a computer game by id Software, website: http://www.idsoftware.com/games/quake/quake/

[8] GNU General Public License (GPL): http://www.gnu.org/copyleft/gpl.html

[9] Quake source code released under the GPL, links for engine and game code:
Engine Code: ftp://ftp.idsoftware.com/idstuff/source/q1source.zip
Game & Tool Code: ftp://ftp.idsoftware.com/idstuff/source/q1tools_gpl.tgz

Unreferenced

J. P. Bigus & J. Bigus, *Constructing Intelligent Agents Using Java - Second Edition*, Wiley & Sons, 2001

Rajesh Vasa, *HIT3102 Intelligent Agents Lecture Notes*, Australia - Rajesh Vasa – Swinburne University of Technology - School of I.T., 2002

The alphabets, words and languages of finite state machines
http://www.c3.lanl.gov/mega-math/workbk/machine/mabkgd.html

Finite State Machine Design
http://www.redbrick.dcu.ie/academic/CLD/chapter8/chapter08.doc.html

Finite state machine software, products and projects
http://www.csd.uwo.ca/research/grail/links.html

Whatis: Finite State Machine
http://whatis.techtarget.com/definition/0,,sid9_gci213052,00.html

Finite State Machine
http://wwwacs.gantep.edu.tr/foldoc/foldoc.cgi?Finite+State+Machine

J. M. P. van Waveren, Quake III Arena Bot Thesis,
http://www.kbs.twi.tudelft.nl/Publications/MSc/2001-VanWaveren-MSc.html, 2001