3.12

A Generic Fuzzy State Machine in C++

Eric Dybsand

edybs@ix.netcom.com

Fuzzy logic was ably presented in the original *Game Programming Gems* article titled "Fuzzy Logic for Video Games" by Mason McCuskey [McCuskeyOO]. Likewise, a generic Finite State Machine was introduced in that same book, in the article "A Finite-State Machine Class" written by me [DybsandOO]. This gem will serve to marry these two concepts into a generic Fuzzy State Machine (FuSM) C++ class that you can use in your games, and to provide you with an additional overview of fuzzy logic, as well as some ideas on how to use fuzzy logic in your games.

First, let's briefly review the FAQ definition of fuzzy logic:

> *"Fuzzy logic is a superset ofconventional (Boolean) logic that has been extended to handle the concept ofpartial truth—truth values between "completely true" and "completelyfalse" [FAQ97]."*

Thus, instead of the states of *ON* and *OFF,* or *TRUE* and *FALSE,* a fuzzy state machine can be in a state of almost *ON or* just about *OFF,* or partially *TRUE or* sort *of FALSE.* Or even both *ON and OFF or TRUE and FALSE,* but to various degrees.

What does this mean to a game developer? It means that a Non-Player Character (NPC), for instance, does not have to be just *AMD* at a player, but that the NPC can be almost *MAD,* or partly *MAD,* or really *MAD,* or raging *MAD* at a player. In other words, by using a FuSM (implementing fuzzy logic), a game developer can have multiple degrees of state assigned to a character (or a concept within the game—more on this later). This could also mean that states in a game do not have to be specific and discrete (often referred to in the fuzzy logic world as *crisp),* but can be, well, fuzzy (less determined). The real advantage to this is discussed in the next section.

Status in a FuSM is typically represented internally using the range of real numbers from 0.0 to 1.0; however, that is not the only way we can represent a fuzzy value. We can choose literally any set of numbers, and consider them fuzzy values. Continuing the NPC attitude example [DybsandOO], let us consider how to maintain the "dislike portion" of the attitude of an NPC toward a player within a FuSM. We could use the range of integers from 1 to 25 to indicate that an NPC has a variable feeling of

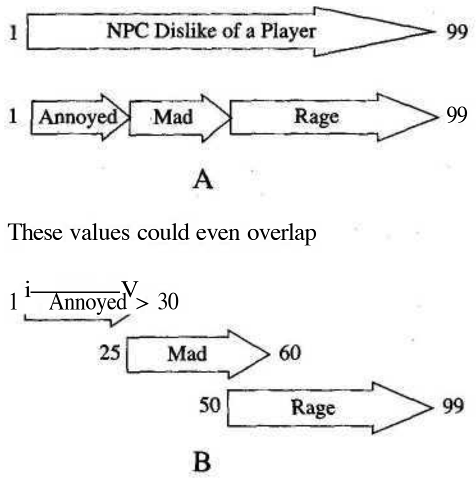

FIGURE 3.12.1 *A) Fuzzy values for dislike attitudes toward player. B) Fuzzy values that overlap.*

ANNOYANCE toward a player, and the range of integers from 26 to 50 may reflect an NPC's variable feeling of *MAD* toward a player, while the range of integers from 51 to 99 may indicate the NPC's variable feeling of *RAGE*. Thus, within each type of attitude toward a player, the NPC may possess various degrees of dislike such as *ANNOYANCE, MAD* or *RAGE* (Figure 3.12.1).

Before we leave this brief introduction to FuSMs, let's clear up one common misconception often associated with fuzzy logic: there is no specific relationship between fuzzy values and probability. *Fuzzy* logic is not some new way to represent probability, but instead represents a degree of membership in a set. In probability, the values must add up to 1.0 in order to be effective. Fuzzy logic values have no such requirement (note the overlap example just presented). This does not mean that fuzzy logic values can't happen to add up to 1.0, it just means that they don't have to for a FuSM to be effective.

Why Use a FuSM In a Game?

In this author's opinion, the number-one reason to use FuSMs in a computer game is that it is an easy way to implement fuzzy logic, which can broaden the depth of representation of the abstract concepts used to represent the game world and the relationships between objects in the game.

In essence, to increase gameplay!

How do FuSMs increase gameplay, you ask? FuSMs increase gameplay by providing for more interesting responses by NPCs, by enabling less predictable NPC behavior, and by expanding the options for choices to be made by the human player.

Thus, a player does not encounter an NPC that is just *MAD* or not *MAD* about being attacked by the player. Instead, the player must deal with an NPC that can be various degrees of being *MAD*. This broader array of considerations increases gameplay by adding to the level of responses that can be developed for the NPC, and seen by the human player.

Another effect of adding FuSMs to computer games is to increase the replayability of the game. By broadening the range of responses and conditions that the player may encounter in given situations during the game, the player will be more likely to experience different outcomes in similar situations.

How To Use FuSMs in a Game

Actually, various forms of FuSMs have been used a lot in computer games!

One example of where a FuSM has been used in computer games is for the health or hit points of an NPC or agent. Instead of the agent simply being healthy or dead (finite states), using a range of hits points can reflect the agent being anything from completely healthy to partially healthy to almost dead to totally dead (fuzzy states). Another example of using a FuSM in a computer game can be found in the control process for accelerating or braking an AI-controlled car in a racing game. Using a FuSM in this case would provide various degrees of acceleration or braking to be calculated in lieu of the finite states of *THROTTLE-UP* or *THROTTLE-DOWN* and *BRAKE-ON* or *BRAKE-OFF actions*. And as our ongoing example of representing an attitude shows, a FuSM is perfect for representing NPC emotional status and attitude toward the player or other NPCs.

Applying fuzzy logic to states in a computer game is relatively straightforward, as noted in the previous examples. Those decision processes that can be viewed as having more than two discrete outcomes are perfect candidates for the application of fuzzy logic, and there are many of those processes to be found.

Now let's consider putting fuzzy logic into a generic C++ class, a FuSM.

Review of Game *Programming Gems'* Generic Finite State Machine in C++

The original Generic FSM in C++ [Dybsand00] consisted of two classes: FSMclass and FSMstate. The FSMclass class object encapsulated the actual finite state machine process, maintained the current state of the FSM, supported the container for the various FSMstate class objects, and provided control for the state transition process.

The FSMstate class object encapsulated a specific state and maintained the arrays of input and output states for which a transition could be performed for the state of the object.

Inputs to the FSM were presented to FSMclass::StateTransition(), which determined the appropriate FSMstate object that was to handle the input (based on the current state), and then the input was passed to FSMstate::GetOutput() to obtain the new output state.

The new output state was returned to the FSM user process by FSMclass::State-Transition() and was also made the new current state of the FSMclass object. Thus, the generic FSM provided a crisp and discrete state transition in response to an input.

Adapting the Generic FSM in C++ to FuSM in C++

There are only a few changes needed to transform the Generic FSM into a FuSM. The first is to add support to the FSMclass class object for multiple current states. The next change is to modify the FSMstate class to support degrees of being in a state. Finally, we need to modify the state transition process within both class objects to support a transition to multiple states and a degree of being in the new state. During this refinement process, we will morph *FSMclass* into FuSMclass and the FSMstate class into the FuSMstate class.

ON THE CD

The reader is invited to review the Fuzzy/Finite State Machine project found on ,-ke companion CD-ROM, and to follow along in that project code as the various classes are referenced.

The adaptation process begins with the FuSMclass class that, while similar to FSMclass, is now capable of supporting multiple current states (the new FuSMclass and FuSMstate class members are shown in bold.) This capability is provided by the FuzzyState_List m_list member, which is an STL list object that contains pointers to FuSMstate objects. A pointer to any active fuzzy state (based on the current input value) object is saved in this list. This way, multiple current states can be supported. As in FSMclass before, FuSMclass also maintains an STL map object (the Fuzzy-State_Map m_map member) for containing pointers to all possible FuSMstate objects that could be considered by FuSMclass.

We continue the adaptation process by developing an access member function (called GetNextFuzzyStateMember()) that will provide an accessing service to the FuSMclass object. The GetNextFuzzyStateMember() member function maintains a pointer to the next FuSMstate pointer in the FuzzyStateJ-ist m_list, so that all active current states can be accessed by processes outside of FuSMclass. Thus, this service is how you can get access to the active current states by your program. By continuing to call GetNextFuzzyStateMember() until you receive a NULL pointer, your program can determine all the active fuzzy states.

The next step in the adaptation process is to modify the FuSMstate class to support various degrees of membership. This support is provided by adding the new member variables of int m_iLowRange and int m_iHighRange. For simplicity, this design views the fuzzy membership range as whole positive numbers, and could be easily adapted to view membership as a set of real numbers. For convenience, this adaptation also maintains two additional attributes of the FuSMstate object: the value of membership in the set for this FuSMstate object (int m_iValueOfMembership), and the degree of membership in the set (int m_iDegreeOfMembership).

Notice that the biggest difference between the finite state object (FSMstate) and our new fuzzy state object (FuSMstate) is that a state transition array is no longer needed. This is because in fuzzy logic, it is possible to be in one or more states at the same time; while in finite logic, it is possible only to be in one state at a time.

Concluding the adaptation process involves modifying the state transition process in both the FuSMclass and FuSMstate objects to support the possibility of multiple current states and to support various degrees of membership within a given state. For FuSMclass, this means modifying StateTransition() to process the Fuzzy-State_Map m_map member containing all possible states, giving each FuSMstate object an opportunity to effect a transition based on the accumulated input value (the int m_iCurrentInput member found in the FuSMclass). Those FuSMstate objects that do transition have their pointers saved off in the FuzzyState_List m_list member, thus indicating that the FuSMstate object is an active current state.

For the FuSMstate class object, the adaptation process involves replacing FSM-state: :GetOutput () (from the FSM) with a new transition function. The new FuSM-state: :DoTransition() member function accepts the input value maintained by FuSMclass and considers the degree of membership this input value represents. If membership within the fuzzy state exists, the member function returns a *TRUE* and maintains the status of membership for any future access.

ON THE CD

This completes the adaptation process. For more details and the code listings, see tne FuSM project on the companion CD-ROM.

Now Fuzzy Up Your Games!

Using the FuSMclass and FuSMstate classes as a guide, you are now ready to start making your games more fuzzy! Doing so will enrich the gameplay experience of your players and broaden your own understanding of how to deploy one of the more flexible forms of artificial intelligence tools available to game developers.

References

[DybsandOO] Dybsand, Eric, "A Generic Finite State Machine in C++," *Game Programming Gems,* Charles River Media, 2000.
[FAQ97] "What is fuzzy logic?" FAQ: Fuzzy Logic and Fuzzy Expert Systems 1/1 Monthly Posting, www.faqs.org/faqs/fuzzy-logic/partl/, 1997.
[McCuskeyOO] McCuskey, Mason, "Fuzzy Logic for Video Games," *Game Programming Gems,* Charles River Media, 2000.

3.13

Imploding Combinatorial Explosion in a Fuzzy System

Michael Zarozinski, Louder Than A Bomb! Software

michaelz@LouderThanABomb.com

Fuzzy logic, when you come right down to it, is just a bunch of "if-then" statements. One of the biggest problems in using fuzzy logic is that the number of "if-then" statements grows exponentially as you increase the number of fuzzy sets you "if" together. This is called *combinatorial explosion,* and can make fuzzy systems slow, confusing, and difficult to maintain. In games, speed is essential, and combinatorial explosion can make the use of fuzzy logic impractical.

For an introduction to fuzzy logic see "Fuzzy Logic for Video Games" by Mason McCuskey in the first *Game Programming Gems* [McCuskeyOO]. For this gem, we'll provide some definitions, as there is little agreement on fuzzy logic terminology.

- **Variable.** A fuzzy variable is a concept such as "temperature," "distance," or "health."
- Set. In traditional logic, sets are "crisp"; either you belong 100 percent to a set or you do not. A set of tall people may consist of all people over six feet tall, anyone less than six feet is "short" (or more appropriately, "not tall"). Fuzzy logic allows sets to be "fuzzy" so anyone over six feet tall may have 100-percent membership in the "tall" set, but may also have 20-percent membership in the "medium height" set.

The Problem

Table 3.13.1 shows the effects of combinatorial explosion as more variables and/or sets are added to the system.

This exponential growth in the number of rules can bring any system to its knees if every possible rule must be checked on each pass.

Table 3.13.1 The Effects of Combinatorial Explosion

Number of Variables	Sets Per Variable	Number of Rules
2	5	$5^2 = 25$
3	5	$5^3 = 125$
4	5	$5^4 = 625$
5	5	$5^5 = 3{,}125$
6	5	$5^6 = 15{,}625$
7	5	$5^7 = 78{,}125$
8	5	$5^8 = 390{,}625$
9	5	$5^9 = 1{,}953{,}125$
10	5	$5^{10} = 9{,}765{,}625$

The Solution

William E. Combs, an engineer at Boeing, developed a method for turning the exponential growth shown above into linear growth known, appropriately enough, as the "Combs Method." This results in a system with 10 variables and 5 sets per variable having only 50 rules, as opposed to 9,765,625 rules.

It is important to note that the Combs Method is not an algorithm for converting existing "if-then" rules to a linear system. You should start from the bottom up, creating rules that fit in with the Combs Method.

If you're interested in the theory behind the Combs Method, see the proof at the end of this gem.

The Real World

To bring this theory into the real world, we'll look at a trivial system for calculating the *aggressiveness* of a unit in a game. For now, we'll consider a one-on-one battle with three variables, ignoring any surrounding units (friend or foe):

- Our health
- Enemy health
- Distance between us and the enemy

The *health* variables have three sets: Near death, Good, and Excellent.

The *distance* variable has three sets: Close, Medium, and Far.

Finally, our output *(aggressiveness)* has three sets: Run away, Fight defensively, and All-out attack!.

Traditional Fuzzy Logic Rules

If we were using a traditional fuzzy logic system, we'd start creating rules in a spreadsheet format as shown in Table 3.13.2.

Table 3.13.2 Some Traditional Fuzzy Logic Rules

Our Health	Enemy Health	Distance	Aggressiveness
Excellent	Excellent	Close	Fight defensively
Excellent	Excellent	Medium	Fight defensively
Excellent	Excellent	Far	All-out attack!
Excellent	Good	Close	Fight defensively
Excellent	Good	Medium	All-out attack!
Excellent	Good	Far	All-out attack!
Excellent	Near death	Close	All-out attack!
Excellent	Near death	Medium	All-out attack!
Excellent	Near death	Far	All-out attack'
		• • •	
Good	Good	Close	Fight defensively
Good	Near death	Close	Fight defensively
		• • •	
Near death	Excellent	Close	Run away
Near death	Excellent	Medium	Run away
Near death	Excellent	Far	Fight defensively

Note that Table 3.13.2 only shows 14 of the 27 possible rules. While a trivial example such as this is fairly manageable, combinatorial explosion quickly conies into play. In a game, we may need to take into account more variables such as the relative health of our forces and the enemy forces that may join in the battle. If we were to represent these two additional variables (bringing the total number of variables to five), the table would grow from 27 rules to 243 rules. This can quickly get out of hand. Fortunately, the Combs Method only needs 15 rules to deal with the same five variables.

Combs Method of Fuzzy Logic Rules

Building rules in the traditional system, we look at how the combination of input sets relates to the output. To build rules using the Combs Method, we look at each individual set's relationship to the output and build the rules one variable at a time (Table 3.13.3).

In a Combs Method system, it is recommended that all variables have the same number of sets as the output variable. This is not an absolute rule, but it gives each output set the chance to be paired with an input set for each variable.

Table 3.13.3 Each Individual Set's Relationship to the Output

Our health	Aggressiveness
Excellent	All-out attack!
Good	Fight defensively
Near death	Run away

Enemy health	Aggressiveness
Excellent	Run away
Good	Fight defensively
Near death	All-out attack!

Distance	Aggressiveness
Close	Fight defensively
Medium	Fight defensively
Far	All-out attack!

Concrete Example

To test the system, we'll use the following values:

- Our health: 76.88
- Enemy health: 20.1
- Distance: 8.54

Figures 3.13.1 through 3.13.3 show the "Degree of Membership," or DOM, for the input values in the system (note that the DOMs for a variable do not have to sum to 100 percent).

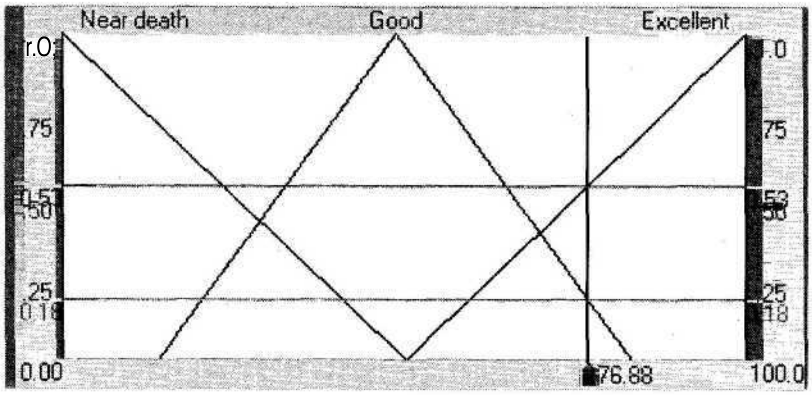

FIGURE 3.13.1 Our Health>r *a value of 76.88. Near death: 0%, Good: 18%, Excellent: 53%.*

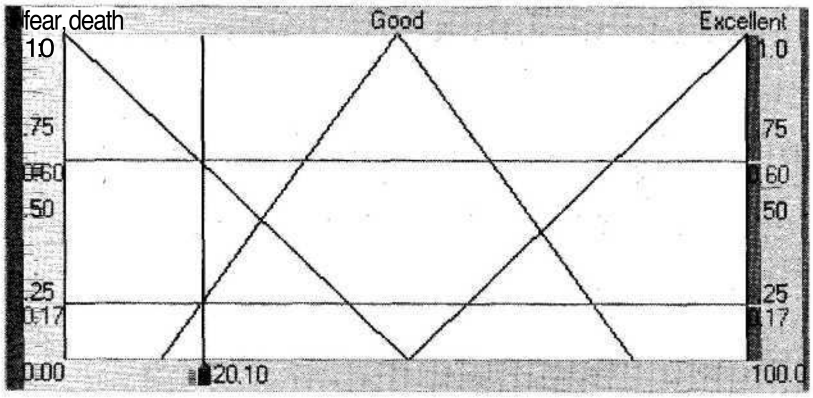

FIGURE 3.13.2 Enemy Healthy»rd *value of20.1. Near death: 60%, Good: 17%, Excellent: 0%.*

Figures 3.13.1 through 3.13.4 were taken from the Spark! fuzzy logic editor, which allows you to visually create a fuzzy logic system, integrate it into your game, and change the AI in real time without having to recompile.

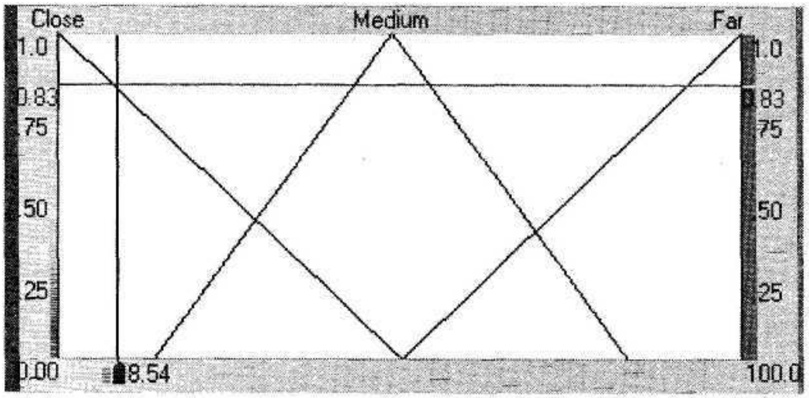

FIGURE 3.13.3 *Distance fir a value of 8.54. Close: 83%, Medium: 0%, Far: 0%.*

Concrete Example with the Traditional System

Using the rules we created earlier, the rules listed in Table 3.13.4 are activated in the traditional system.

The DOM of the input sets are ANDed together to get the output set's DOM. The ANDing is logically equivalent to taking the MINIMUM of the three input values.

The denazification method we're using—center of mass—takes the MAXIMUM value for the output set then finds the center of mass for the output sets (Figure 3.13.4).

Table 3.13.4 These Rules Are Activated in the Traditional System

Our Health	Enemy Health	Distance	Aggressiveness
Excellent (53%)	Good (17%)	Close (83%)	Fight defensively (17%)
Excellent (53%)	Near death (60%)	Close (83%)	All-out attack! (53%)
Good (18%)	Good (17%)	Close (83%)	Fight defensively (17%)
Good (18%)	Near death (60%)	Close (83%)	Fight defensively (18%)

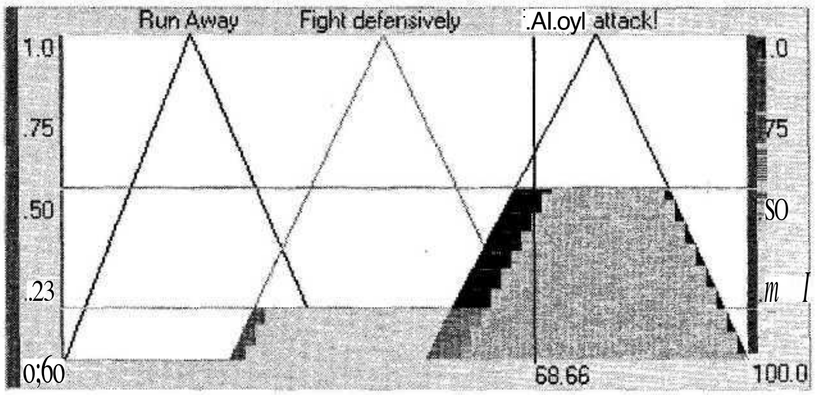

FIGURE 3.13.4 *Traditional system output. Fight defensively: 18%; All-out attack: 53%; Aggressiveness: 68.66.*

Concrete Example Using the Combs Method

Using the same input values, the Combs Method rules are listed in Table 3.13.5.

The Combs Method result of 60.39 is not exactly the same as the traditional method (68.66), but we wouldn't expect it to be as we're using a different inference method. The Combs Method is ORing the values together, which is the same as taking the MAXIMUM (Figure 3.13.5). Traditional fuzzy logic ANDs values together, which takes the MINIMUM; hence, the difference in the *fight defensively* output sets.

Note that (in this case) if we took the MINIMUM of the output sets (and there is no rule saying we can't) we would get the exact same result as the traditional fuzzy logic method. This is a result of the rules we selected for the Combs Method. Since there is not an algorithm to convert traditional fuzzy logic rules to the Combs Method, we cannot say that by simply taking the MINIMUM you will always get the same results as in the traditional method.

The Proof

It is not essential that you understand why the Combs Method works in order to use it. Formal logic can be confusing, so this proof is provided for the reader who wishes to have a deeper understanding of the theory behind the Combs Method. The Combs

Table 3.13.5 Combs Method system output. Fight defensively: 83%; All out attack: 53%; Aggressiveness: 60.39.

Our Health	Aggressiveness
Excellent (53%)	All-out attack! (53%)
Good (18%)	Fight defensively (18%)
Enemy Health	**Aggressiveness**
Good (17%)	Fight defensively (17%)
Near death (60%)	Fight defensively (60%)
Distance	**Aggressiveness**
Close (83%)	Fight defensively (83%)

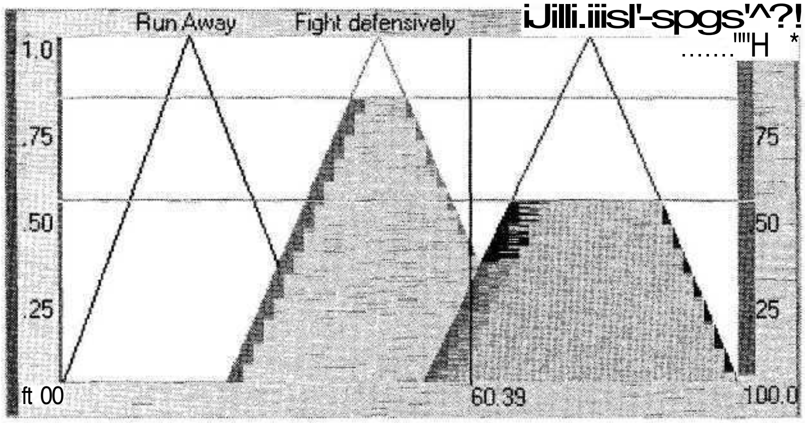

FIGURE 3.13.5 *Combs Method system output. Fight defensively: 83%, All out attack: 53%. Aggressiveness: 60.39.*

Method is based on the fact that the logical proposition *(p* and *q)* then *r* is equivalent to *(p* then *r)* or *(q* then r).

Since fuzzy logic is a superset of formal logic, we can ignore "fuzziness" and just prove the equivalence of the Combs Method to the traditional method. In Table 3.13.6, *p* and *q* are antecedents and *r* is the consequence. The antecedents are statements such as "if Jim is tall" or "if Jim is healthy." The consequence is a *potential* result such as "Jim can play basketball".

This proof is straightforward, except for the *x* thenjy type clauses. These are standard formal logic propositions, but their truth table is confusing—especially when *x* and *y* are false but the proposition is true. See [aiGuruOl] for some examples that will help clarify this proposition's truth table.

Table 3.13.6 Antecedents and Consequences

P..__	J7 _.,.	pandq	*T*	*Ip* and *q)* then r	*p* then *r*	q then *r*	(p then *r)* or (Q then *r)*
T	T	T	T	T	T	T	T
T	T	T	F	F	F	F	F
T	F	F	T	T	T	T	T
T	F	F	F	T	F	T	T
F	T	F	T	T	T	T	T
F	T	F	F	T	T	F	T
F	F	F	T	T	T	T	T
F	F	F	F	T	T	T	T

If you need visual proof, the Venn diagrams are shown in Figures 3.13.6 and 3.13.7. Since Venn diagrams can only show AND, OR, and NOT relationships, the following conversions are made by material implication:

- Traditional Logic: *(p* and *q)* then *r* is equivalent to (not *(p* and *q))* or *r*
- Combs Method: *(p* then *r)* or *(q* then *r)* is equivalent to ((not/>) or *r)* or ((not *q)* or *r)*

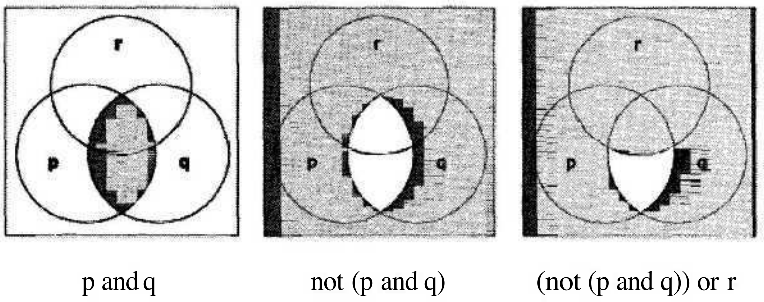

FIGURE 3.13.6 *Venn diagramfor Traditional Logic.*

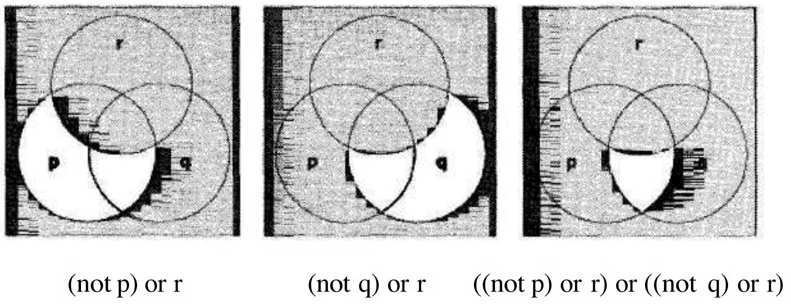

FIGURE 3.13.7 *Venn diagramfor Combs Method.*

Conclusion

If forced to make a choice between fast and intelligent game AI, fast will win almost every time (ignoring turn-based games). The Combs Method allows you to create AI that is not only fast, but also complex enough to result in rich, lifelike behavior, thus providing a more engaging experience for the player.

Next time you're faced with creating a complex behavior system, give the Combs Method a try. You may be surprised by the depth of the behavior you can get with such a small number of rules.

References

[aiGuruOl] Zarozinski, Michael, "if p then q?" available online at www.aiGuru.com/logic/if_p_then_q.htm, March 5, 2001.

[Andrews97] Andrews, James E., "Taming Complexity in Large-Scale Fuzzy Systems," PC AI (May/June 1997): pp.39-42.

[Combs99] Combs, William E., *The Fuzzy Systems Handbook 2nd Ed*, Academic Press, 1999.

[McCuskeyOO] McCuskey, Mason, "Fuzzy Logic for Video Games," *Game Programming Gems*, Charles River Media, 2000.