

7

Using Fuzzy Logic to Make Your AI Seem Alive

Fuzzy logic is a fantastic way to represent the rules of your game in a more nuanced way. Perhaps more so than other concepts in this book, fuzzy logic is a very math-heavy topic. Most of the information can be represented purely in mathematical functions. For the sake of teaching the important concepts as they apply to Unity, most of the math has been simplified and implemented using the Unity's built-in features. Of course, if you are the type who loves math, this is a somewhat deep topic in that regard, so feel free to take the concepts covered in this book and run with them! In this chapter, we'll learn:

- What fuzzy logic is
- Where fuzzy logic is used
- How to implement fuzzy logic controllers
- What the other creative uses for fuzzy logic concepts are

Defining fuzzy logic

The simplest way to define fuzzy logic is by comparison to binary logic. In the previous chapters, we looked at transition rules as true or false or 0 or 1 values. Is something visible? Is it at least a certain distance away? Even in instances where multiple values were being evaluated, all of the values had exactly two outcomes thus, they are binary. In contrast, fuzzy values represent a much richer range of possibilities, where each value is represented as a float rather than an integer. We stop looking at values as 0 or 1, and we start looking at them as 0 to 1.

A common example used to describe fuzzy logic is temperature. Fuzzy logic allows us to make decisions based on non-specific data. I can step outside on a sunny California summer day and ascertain that it is warm, without knowing the temperature precisely. Conversely, if I were to find myself in Alaska during the winter, I would know that it is cold, again, without knowing the exact temperature. These concepts of cold, cool, warm, and hot are fuzzy ones. There is a good amount of ambiguity as to at what point we go from warm to hot. Fuzzy logic allows us to model these concepts as sets and determine their validity or truth by using a set of rules.

When making decisions, people, as it is common to say, has some gray area. That is to say, it's not always black and white. The same concept applies to agents that rely on fuzzy logic. Say you hadn't eaten in a few hours, and you were starting to feel a little hungry. At which point were you hungry enough to go grab a snack? You could look at the time right after a meal as 0, and 1 would be the point where you approached starvation. The following figure illustrates this point:



When making decisions, there are many factors that determine the choice. This leads into another aspect of fuzzy logic controllers – they can take into account as much data as necessary. Let's continue to look at our "should I eat?" example. We've only considered one value for making that decision, which is the time since the last time you ate, however, there are other factors that can affect this decision, such as, how much energy you're expending and how lazy you are at that particular moment. Or am I the only one to use that as a deciding factor? Either way, you can see how multiple input values can affect the output, which we can think of as the "likeliness to have another meal".

Fuzzy logic systems can be very flexible due to their generic nature. You provide input, the fuzzy logic provides an output. What that output means to your game is entirely up to you. We've primarily looked at how the inputs would affect a decision, which, in reality, is taking the output and using it in a way the computer, our agent, can understand. However, the output can also be used to determine how much of something to do, or how fast something happens, or for how long something happens. For example, imagine your agent is a car in a sci-fi racing game that has a "nitro-boost" ability that lets it expend a resource to go faster. Our 0 to 1 value can represent a normalized amount of time for it to use that boost or perhaps a normalized amount of fuel to use.

Picking fuzzy systems over binary systems

As with the previous systems we covered in this book, and with most things in game programming, we must evaluate the requirements of our game and the technology and hardware limitations when deciding on the best way to tackle a problem.

As you might imagine, there is a performance cost associated with going from a simple yes/no system to a more nuanced fuzzy logic one, which is one of the reasons we may opt out of using it. Of course, being a more complex system doesn't necessarily always mean it's a better one. There will be times when you just want the simplicity and predictability of a binary system because it may fit your game better.

While there is some truth to the old adage "the simpler, the better", one should also take into account the saying "everything should be made as simple as possible, but not simpler". Though the quote is largely attributed to Albert Einstein, the father of relativity, it's not entirely clear who said it. The important thing to consider is the meaning of the quote itself. You should make your AI as simple as your game needs it to be, but not simpler. Pac-Man's AI works perfectly for the game—it's just simple enough. However, rules say that simple would simply be out of place in a modern shooter or strategy game.

Take the knowledge and examples from this book and find what works best for you.

Using fuzzy logic

Once you understand the simple concepts behind fuzzy logic, it's easy to start thinking of the many, many ways in which it can be useful. In reality, it's just another tool in our belt, and each job requires different tools.

Fuzzy logic is great at taking some data; evaluating it in a way similar to how a human would (albeit in a much simpler way) and then translating the data back to information usable by the system.

Fuzzy logic controllers have several real-world use cases. Some are more obvious than others, and while these are by no means one-to-one comparisons to our usage in game AI, they serve to illustrate a point:

- **Heating ventilation and air conditioning (HVAC) systems:** The temperature example when talking about fuzzy logic is not only a good theoretical approach to explaining fuzzy logic, but also a very common real-world example of fuzzy logic controllers in action.
- **Automobiles:** Modern automobiles come equipped with very sophisticated computerized systems, from the air conditioning system (again) to fuel delivery to automated breaking systems. In fact, putting computers in automobiles has resulted in far more efficient systems than the old binary systems that were sometimes used.
- **Your smartphone:** Ever notice how your screen dims and brightens depending on how much ambient light there is? Modern smartphone operating systems look at ambient light, the color of the data being displayed, and the current battery life to optimize screen brightness.
- **Washing machines:** Not my washing machine necessarily as it's quite old, but most modern washers (from the last 20 years) make some use of fuzzy logic. Load size, water dirtiness, temperature, and other factors are taken into account from cycle to cycle to optimize water use, energy consumption, and time.

If you take a look around your house, there is a good chance you'll find a few interesting uses of fuzzy logic, and I mean besides your computer, of course. While these are "neat" uses of the concept, they're not particularly exciting or game-related. I'm partial to games involving wizards, magic, and monsters, so let's look at a more relevant example.

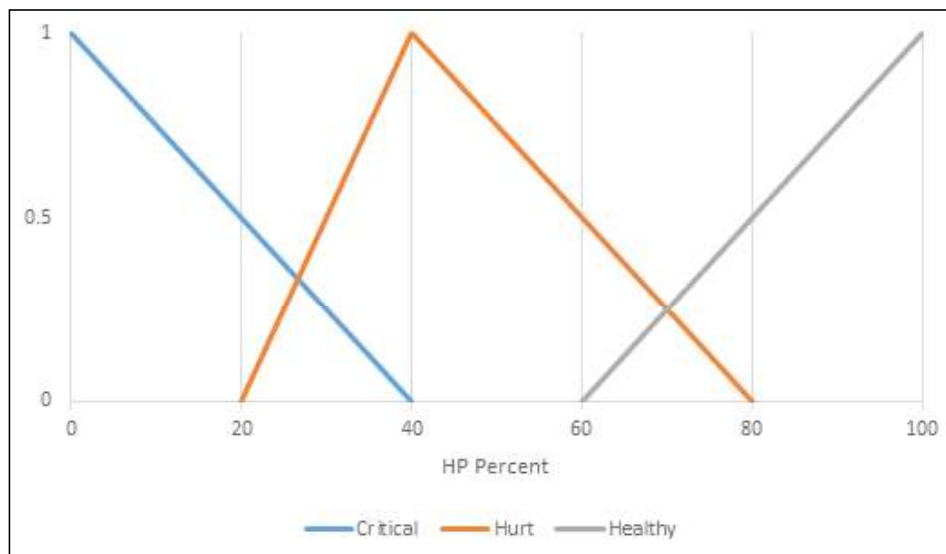
Implementing a simple fuzzy logic system

For this example, we're going to use my good friend, Bob, the wizard. Bob lives in an RPG world, and he has some very powerful healing magic at his disposal. Bob has to decide when to cast this magic on himself based on his remaining **health points (HPs)**.

In a binary system, Bob's decision-making process might look like this:

```
if(healthPoints <= 50)
{
    CastHealingSpell(me);
}
```

We see that Bob's health can be in one of the two states — above 50 or not. Nothing wrong with that, but let's have a look at what the fuzzy version of this same scenario might look similar to, starting with determining Bob's health status:



A typical function representing fuzzy values

Before the panic sets in upon seeing charts and values that may not quite mean anything to you right away, let's dissect what we're looking at. Our first impulse might be to try to map the probability that Bob will cast a healing spell to how much health he is missing. That would, in simple terms, just be a linear function. Nothing really fuzzy about that — it's a linear relationship, and while it is a step above a binary decision in terms of complexity, it's still not truly "fuzzy".

Enter the concept of a membership function. It's sort of the key to our system as it allows us to determine how true a statement is. In this example, we're not simply looking at raw values to determine whether or not Bob should cast his spell, but instead we're breaking it up into logical chunks of information for Bob to use in order to determine what his course of action should be.

In this example, we're looking and comparing three statements and evaluating, not only how true each one is, but which is the most true:

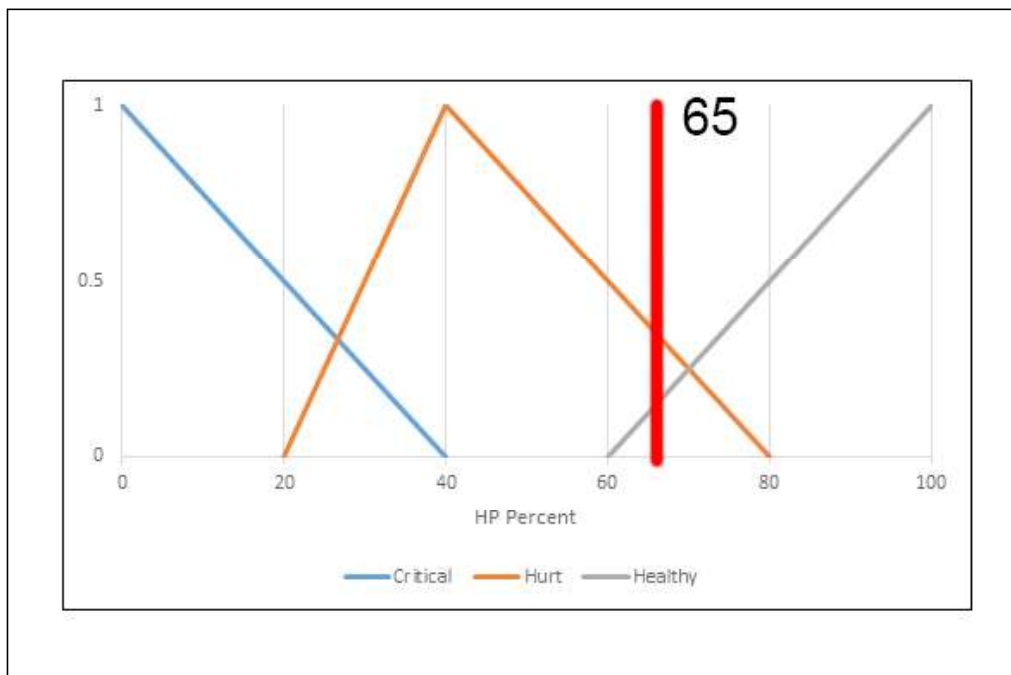
- Bob is in critical condition
- Bob is hurt
- Bob is healthy

If you're into official terminology as such, we call this determining the degree of membership to a set. Once we have this information, our agent can determine what to do with it next.

At a glance, you'll notice it's possible for two statements to be true at a time. Bob can be in a critical condition and hurt. He can also be somewhat hurt and a little bit healthy. You're free to pick the thresholds for each, but in this example, let's evaluate these statements as per the preceding graph. The vertical value represents the degree of truth of a statement as a normalized float (0 to 1):

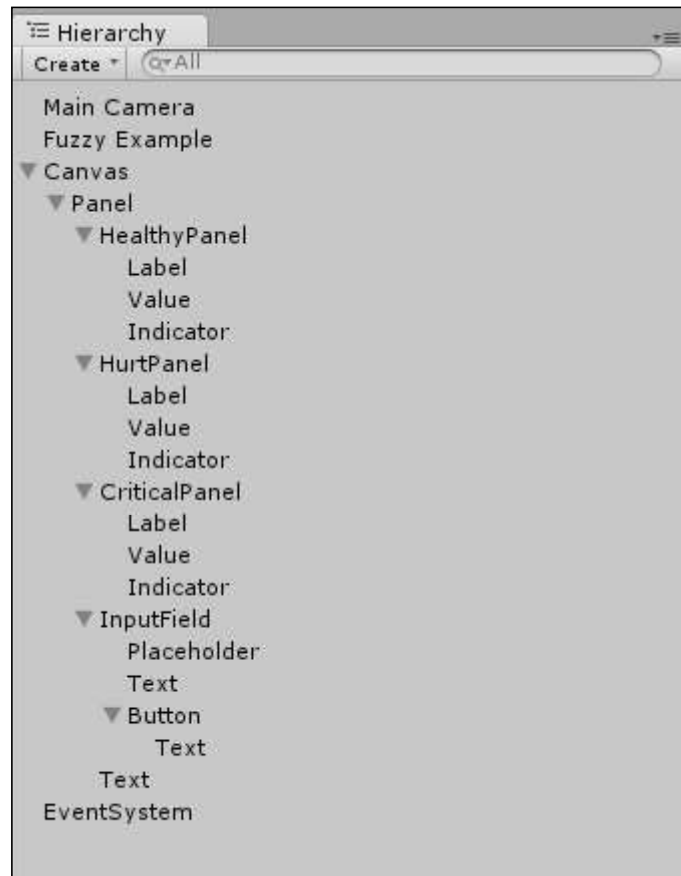
- At 0 percent health, we can see that the critical statement evaluates to 1. It is absolutely true that Bob is critical when his health is gone.
- At 40 percent health, Bob is hurt, and that is the truest statement.
- At 100 percent health, the truest statement is that Bob is healthy.

Anything outside of these absolutely true statements is squarely in fuzzy territory. For example let's say Bob's health is at 65 percent health. In that same chart, we can visualize it like this:



Bob's health at 65 percent

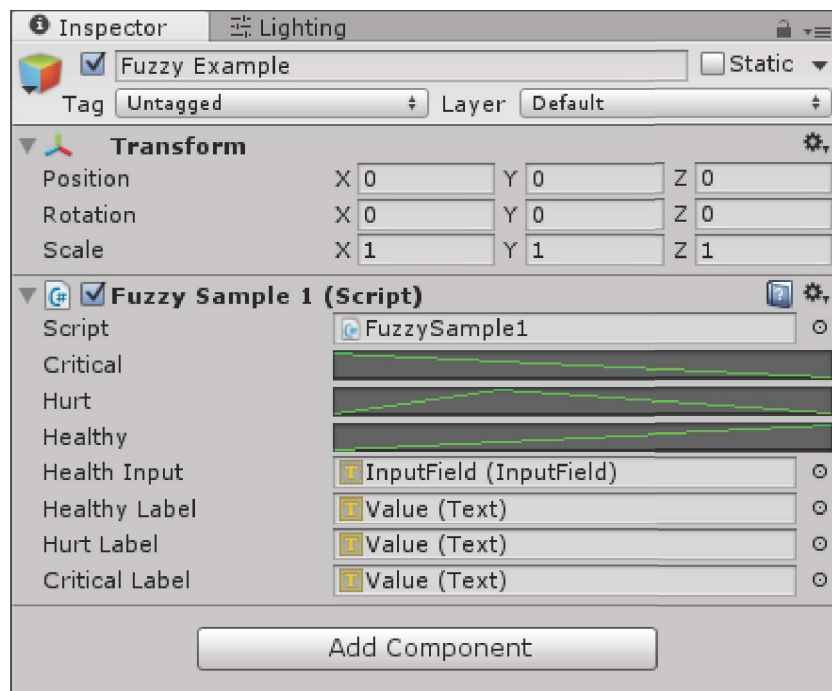
The vertical line drawn through the chart at 65 represents Bob's health. As we can see, it intersects both sets, which means that Bob is a little bit hurt, but he's also kind of healthy. At a glance, we can tell, however, that the vertical line intercepts the "hurt" set at a higher point in the graph. We can take this to mean that Bob is more hurt than he is healthy. To be specific, Bob is 37.5 percent health hurt, 12.5 percent healthy, and 0 percent critical. Let's take a look at this in code; open up our `FuzzySample` scene in Unity. The hierarchy will look like this:



The hierarchy setup in our sample scene

The important game object to look at is `Fuzzy Example`. This contains the logic that we'll be looking at. In addition to that, we have our `Canvas` containing all of the labels and the input field and button that make this example work. Lastly, there's the Unity-generated `EventSystem` and `Camera`, which we can disregard. There isn't anything special going on with the setup for the scene, but it's a good idea to become familiar with it, and you are encouraged to poke around and tweak it to your heart's content after we've looked at why everything is there and what it all does.

With the `Fuzzy Example` game object selected, the inspector will look similar to the following image:

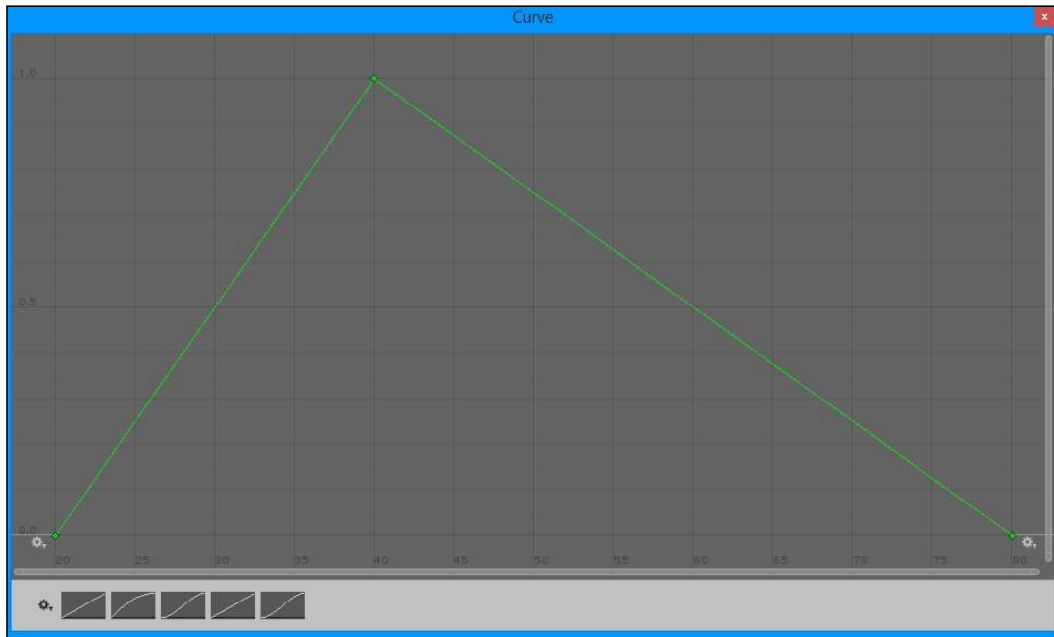


The Fuzzy Example game object inspector

Our sample implementation is not necessarily something you'll take and implement as it is in your game, but it is meant to illustrate the previous points in a clear manner. For the different sets, we use Unity's `AnimationCurve` for each one. It's a quick and easy way to visualize the very same lines in our earlier graph.


Unfortunately, there is no straightforward way to plot all the lines in the same graph, so we use a separate `AnimationCurve` for each set. In the preceding image, they are labeled "Critical", "Hurt", and "Healthy". The neat thing about these curves is that they come with a built-in method to evaluate them at a given point (t). For us, t does not represent time, but rather the amount of health Bob has.

As in the preceding graph, the Unity example looks at a HP range of 0 to 100. These curves also provide a simple user interface for editing the values. You can simply click on the curve in the inspector. That opens up the curve editing window. You can add points, move points, change tangents, and so on, as shown in the following screenshot:



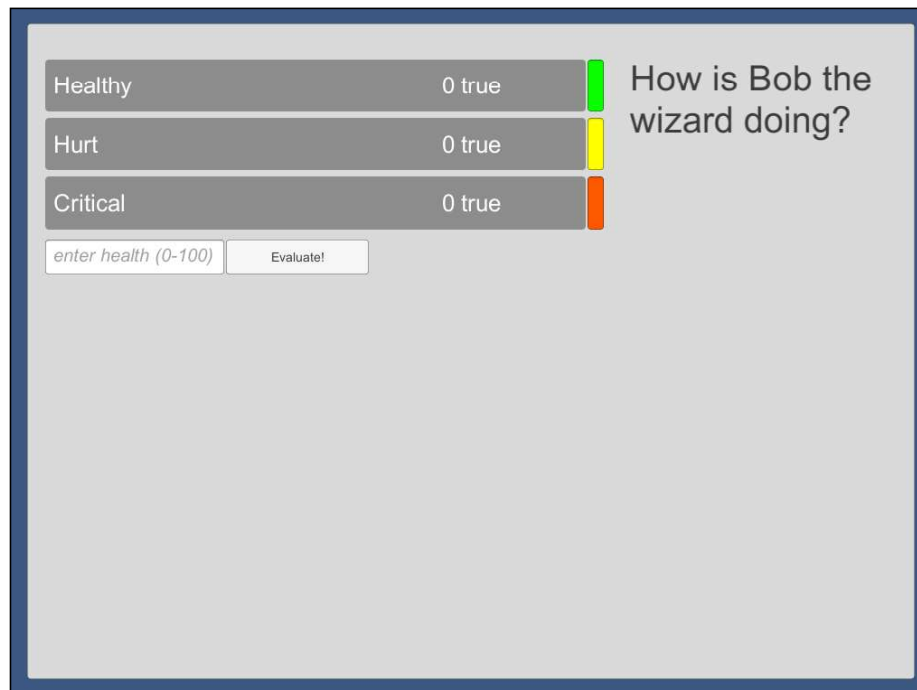
Unity's curve editor window

Our example focuses on triangle-shaped sets. That is, linear graphs for each set. You are by no means restricted to this shape, though it is the most common. You could use a bell curve or a trapezoid for that matter. To keep things simple, we'll stick to the triangle.

[ You can learn more about Unity's AnimationCurve editor at <http://docs.unity3d.com/ScriptReference/AnimationCurve.html>.]

The rest of the fields are just references to the different UI elements used in code that we'll be looking at later in this chapter. The names of these variables are fairly self-explanatory, however, so there isn't much guesswork to be done here.

Next, we can take a look at how the scene is set up. If you play the scene, the game view will look something similar to the following screenshot:



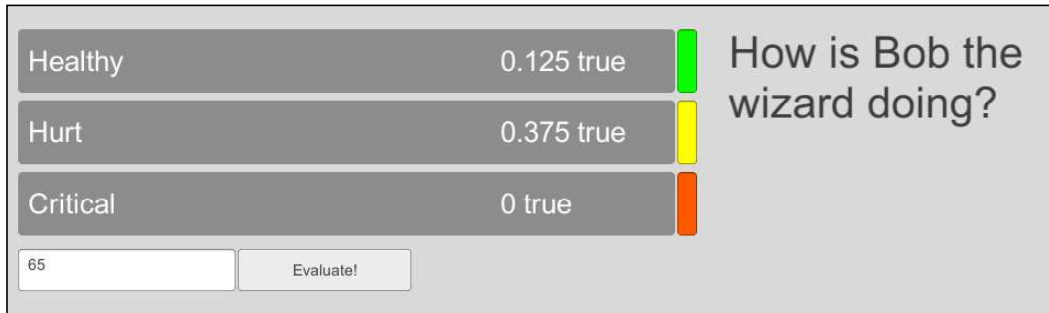
A simple UI to demonstrate fuzzy values

We can see that we have three distinct groups, representing each statement from the Bob the wizard example. How healthy is Bob, how hurt is Bob, and how critical is Bob? For each set, upon evaluating, the value which starts off as "0 true" will dynamically adjust to represent the actual degree of membership.

There is an input box in which you can type a percentage of health to use for the test. No fancy controls are in place for this, so be sure to enter a value from 0 to 100. For the sake of consistency, let's enter a value of 65 into the box and then press the **Evaluate!** button.

This will run some code, look at the curves, and yield the exact same results we saw in our graph earlier. While this shouldn't come as a surprise (the math is what it is, after all), there are fewer things more important in game programming than testing your assumptions, and sure enough, we've tested and verified our earlier statement.

After running the test by hitting the **Evaluate!** button, the game scene will look more similar to the following screenshot:



This is how Bob is doing at 65 percent health

Again, the values turn out to be 0.125 (or 12.5 percent) healthy and 0.375 (or 37.5 percent) hurt. At this point, we're still not doing anything with this data, but let's take a look at the code that's handling everything:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class FuzzySample1 : MonoBehaviour {
    private const string labelText = "{0} true";
    public AnimationCurve critical;
    public AnimationCurve hurt;
    public AnimationCurve healthy;

    public InputField healthInput;

    public Text healthyLabel;
    public Text hurtLabel;
    public Text criticalLabel;

    private float criticalValue = 0f;
    private float hurtValue = 0f;
    private float healthyValue = 0f;
```

We start off by declaring some variables. The `labelText` is simply a constant we use to plug into our label. We replace the `{0}` with the real value.

Next, we declare the three `AnimationCurve` variables that we mentioned earlier. Making these public or otherwise accessible from the inspector is key to being able to edit them visually (though it is possible to construct curves by code), which is the whole point of using them.

The following four variables are just references to UI elements that we saw earlier in the screenshot of our inspector, and the last three variables are the actual float values that our curves will evaluate into:

```
private void Start () {
    SetLabels();
}

/*
 * Evaluates all the curves and returns float values
 */
public void EvaluateStatements() {
    if (string.IsNullOrEmpty(healthInput.text)) {
        return;
    }
    float inputValue = float.Parse(healthInput.text);

    healthyValue = healthy.Evaluate(inputValue);
    hurtValue = hurt.Evaluate(inputValue);
    criticalValue = critical.Evaluate(inputValue);

    SetLabels();
}
```

The `Start()` method doesn't require much explanation. We simply update our labels here so that they initialize to something other than the default text. The `EvaluateStatements()` method is much more interesting. We first do some simple null checking for our input string. We don't want to try and parse an empty string, so we return out of the function if it is empty. As mentioned earlier, there is no check in place to validate that you've input a numerical value, so be sure not to accidentally input a non-numerical value or you'll get an error.

For each of the `AnimationCurve` variables, we call the `Evaluate(float t)` method, where we replace `t` with the parsed value we get from the input field. In the example we ran, that value would be 65. Then, we update our labels once again to display the values we got. The code looks similar to this:

```
/*
 * Updates the GUI with the evaluated values based
 * on the health percentage entered by the
```

```
        * user.  
        */  
    private void SetLabels() {  
        healthyLabel.text = string.Format(labelText, healthyValue);  
        hurtLabel.text = string.Format(labelText, hurtValue);  
        criticalLabel.text = string.Format(labelText, criticalValue);  
    }  
}
```

We simply take each label and replace the text with a formatted version of our `labelText` constant that replaces the `{0}` with the real value.

Expanding the sets

We discussed this topic in detail earlier, and it's important to understand that the values that make up the sets in our example are unique to Bob and his pain threshold. Let's say we have a second wizard, Jim, who's a bit more reckless. For him, "critical" might be below 20 rather than 40, as it is for Bob. This is what I like to call a "happy bonus" from using fuzzy logic. Each agent in the game can have different rules that define their sets, but the system doesn't care. You could predefine these rules or have some degree of randomness determine the limits, and every single agent would behave uniquely and respond to things in their own way.

In addition, there is no reason to limit our sets to just three. Why not four or five? To the fuzzy logic controller, all that matters is that you determine what truth you're trying to arrive at, and how you get there; it doesn't care how many different sets or possibilities exist in that system.

Defuzzifying the data

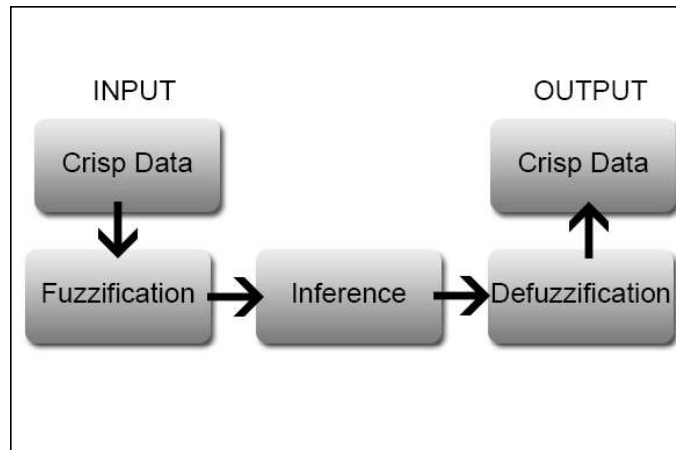
Yes, that's a real (sort of) word. We've started with some crisp rules, which, in the context of fuzzy logic, means clear-cut, hard-defined data, which we then fuzzified (again, a sort of real word) by assigning membership functions to sets. The last step of the process is to defuzzify the data and make a decision. For this, we use simple Boolean operations, that is:

```
IF health IS critical THEN cast healing spell
```

Now, at this point, you may be saying, "Hold on a second. That looks an awful lot like a binary controller," and you'd be correct. So why go through all the trouble? Remember what we said earlier about ambiguous information? Without a fuzzy controller, how does our agent understand what it means to be critical, hurt, or healthy for that matter? These are abstract concepts that mean very little on their own to a computer.

By using fuzzy logic, we're now able to use these vague terms, infer something from them, and do concrete things; in this case, cast a healing spell. Furthermore, we're able to allow each agent to determine what these vague terms mean to them at an individual level, allowing us not only to achieve unpredictability at an individual level, but even amongst several similar agents.

The process is described best in the following diagram:



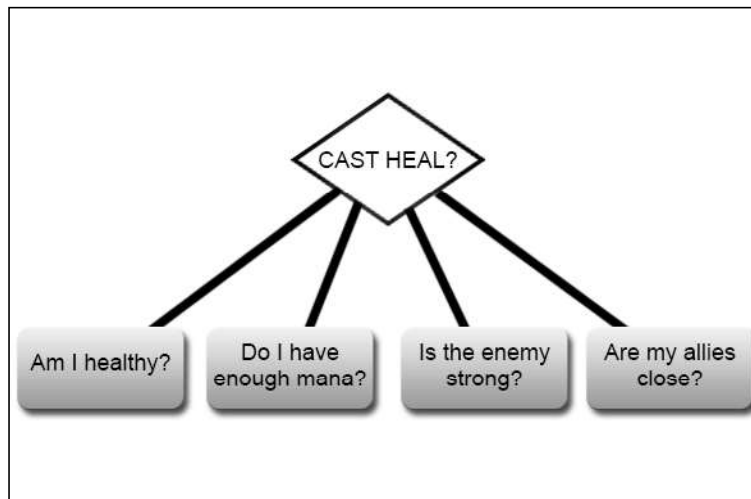
The fuzzy logic controller flow

At the end of the day, they are still computers, so we're bound to the most basic thing computers understand: 0s and 1s:

- We start with, crisp data, that is, concrete, hard values that tell us something very specific.
- The fuzzification step is where we get to decide the abstract or ambiguous data that our agent will need to make a decision.
- During the inference step, our agent gets to decide what that data means. The agent gets to determine what is "true" based on a provided set of rules, meant to mimic the nuance of human decision-making.
- The defuzzification step takes this human-friendly data and converts it into simple computer-friendly information.
- We end with crisp data, ready for our wizard agent to use.

Using the resulting crisp data

The data output from a fuzzy controller can then be plugged into a behavior tree or a finite state machine. Of course, we can also combine multiple controllers' output to make decisions. In fact, we can take a whole bunch of them to achieve the most realistic or interesting result (as convincing as a magic-using wizard can be, anyway). The following figure illustrates a potential set of fuzzy logic controllers it can use to determine whether or not to cast the heal spell:



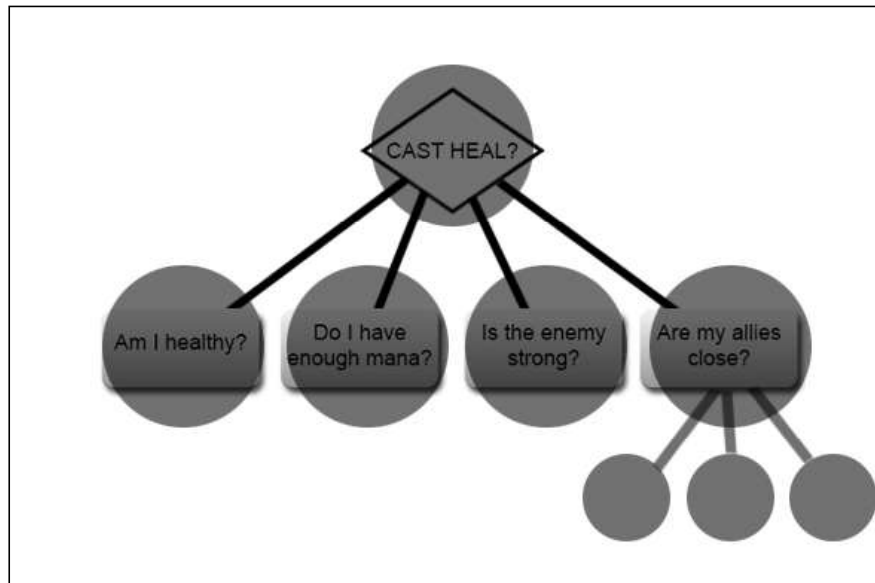
We've looked at the health question already, but what about the rest? We have another set of questions that really don't mean much to our agent on their own:

Do you have enough mana? Well, you can have a little bit of mana, some mana, or a lot of mana. It would not be uncommon for a human player to ask this question as they choose to cast a magic spell in a game or use an ability. "Enough" may literally be a binary amount, but more likely, it would be "enough to cast heal, and have some left for other spells". We start with a straightforward crisp value—the amount of mana the agent has available that we then stick to our fuzzy logic controller and get some crisp data at the other end.

What about the enemy's strength? He could be weak, average, strong, or unbeatable. You can get creative with the input for your fuzzy logic controllers. You could, for example, just take a raw "strength" value from your enemy, but you could also take the difference between your "defensive" stat and the enemy's "attack power", and plug that into your fuzzy logic controller. Remember, there is no restriction on how you process the data before it goes into the controller.

Are my allies close? As we saw in *Chapter 2, Finite State Machines and You*, a simple distance check can do wonders for a simple design, but at times, you may need more than just that. You may need to take into account obstacles along the way – is that an ally behind a locked gate, making him unable to reach the agent? These types of questions could even be a nested set of statements that we need to evaluate.

Now, if we were to take that last question with the nested controllers, it might start to look a little familiar.



The preceding figure is quite tree-like, isn't it? Sure enough, there is no reason why you couldn't build a behavior tree using fuzzy logic to evaluate each node. We end up with a very flexible, powerful, and nuanced AI system by combining these two concepts.

Using a simpler approach

If you choose to stick with a simple evaluation of the crisp output, in other words, not specifically a tree or an FSM, you can use more Boolean operators to decide what your agent is going to do. The pseudo code would look like this:

```
IF health IS critical AND mana IS plenty THEN cast heal
```

We can check for conditions that are not true:

```
IF health IS critical AND allies ARE NOT close THEN cast heal
```


And we can also string multiple conditions together:

```
IF health IS critical AND mana IS NOT depleted AND enemy IS very  
strong THEN cast heal
```

By looking at these simplified statements, you have noticed yet another "happy bonus" of using fuzzy logic – the crisp output abstracts much of the decision-making conditionals and combines them into simplified data.

Rather than having to parse through all the possibilities in your `if/else` statements and ending up with a bazillion of them or a gazillion switch statements, you can neatly bundle pockets of logic into fewer, more meaningful chunks of data.

In other words, you don't have to nest all the statements in a procedural way that is hard to read and difficult to reuse. As a design pattern, abstracting data via a fuzzy logic controller ends up being much more object-oriented and friendlier.

Finding other uses for fuzzy logic

Fuzzy data is very peculiar and interesting in that it can be used in tandem with all of the major concepts we introduced in this book. We saw how a series of fuzzy logic controllers can easily fit into a behavior tree structure, and it's not terribly difficult to imagine how it can be used with an FSM.

Merging with other concepts

Sensory systems also tend to make use of fuzzy logic. While seeing something can be a binary condition, in low-light or low-contrast environments, we can suddenly see how fuzzy the condition can become. You've probably experienced it at night – seeing an odd shape, dark in the distance, in the shadows, thinking "is that a cat?" which then turns out to be a trash bag, some other animal, or perhaps even your imagination. The same can be applied to sounds and smells.

When it comes to pathfinding, we run into the cost of traversing certain areas of a grid, which, a fuzzy logic controller can easily help fuzzify and make more interesting.

Should Bob cross the bridge and fight his way through the guards or risk crossing the river and fighting the current? Well, if he's a good swimmer and a poor fighter, the choice is clear, right?

Creating a truly unique experience

Our agents can use fuzzy logic to mimic personalities. Some agents may be more "brave" than others. Suddenly, their personal characteristics—how fast they are, how far they can run, their size, and so on, can be leveraged to arrive at the decisions that are unique to that agent.

Personalities can be applied to enemies, allies, friends, NPCs, or even to the rules of the game. The game can take in crisp data from the player's progress, style of play, or level of progression, and dynamically adjust the difficulty to provide a more unique and personalized challenge.

Fuzzy logic can even be used to dole out the technical game rules, such as number of players in a given multiplayer lobby, the type of data to display to the player, and even how players are matched against other players. Taking the player's statistics and plugging those into a matchmaking system can help keep the player engaged by pitting him against the players that either match his style of play in a cooperative environment or players of similar skill level in a competitive environment.

Summary

Glad to see that you've made it to the end of the chapter. Fuzzy logic tends to become far less fuzzy once you understand the basic concepts. Being one of the more math-pure concepts in the book, it can be a little daunting if you're not familiar with the lingo, but when presented in a familiar context, the mystery fades away, and you're left with a very powerful tool to use in your game.

We learned how fuzzy logic is used in the real world, and how it can help illustrate vague concepts in a way that binary systems cannot. We also learned how to implement our own fuzzy logic controllers using the concepts of member functions, degrees of membership, and fuzzy sets. Lastly, we explored the various ways in which we can use the resulting data, and how it can help make our agents more unique.

In the final chapter, we will look at several of the concepts introduced in the book working together.