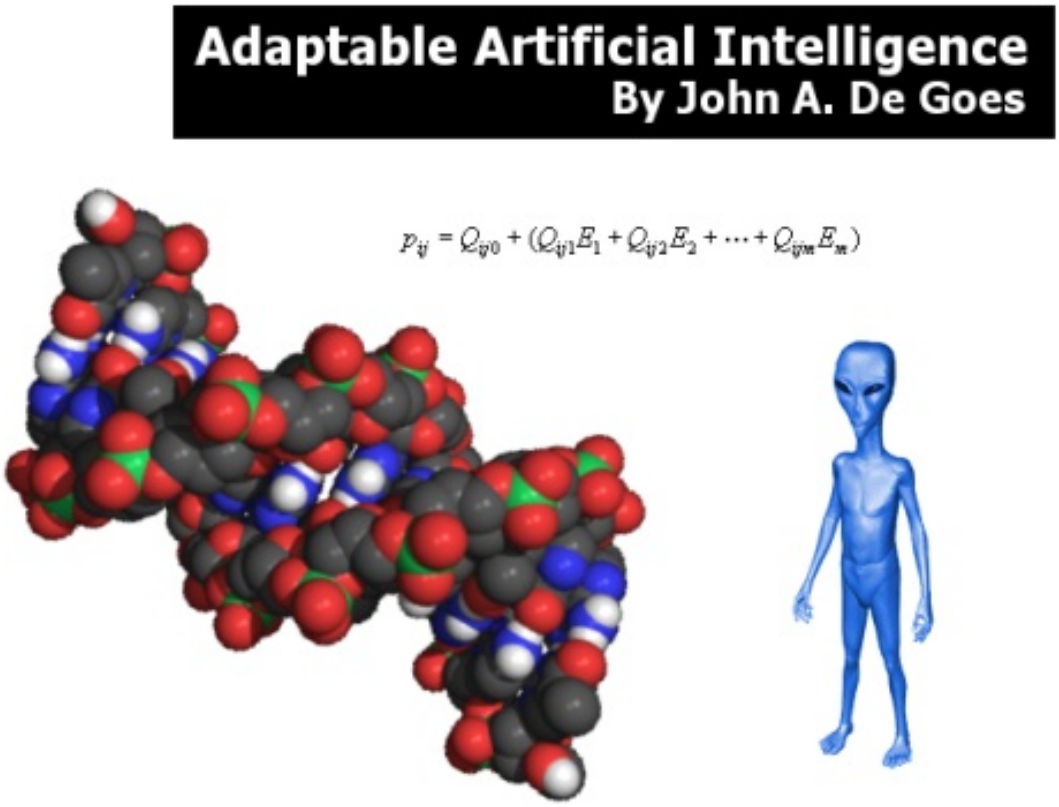

Adaptable Artificial Intelligence

Written by John A. De Goes

Table of Contents

Introduction

Three or four years ago, I remember playing a first-person shooter in which the hero's mission was to wipe out armies of the undead with an array of high-tech weapons. My weapon of choice was the minigun--a perennial favorite of gamers ever since its debut in the classic Wolfenstein 3D (or at least, that's the first game I can remember sporting a minigun).

At one point in the game I placed my minigun-inflicted carnage on hold long enough to study a zombie who looked like he was having a particularly bad day. Not because of the decaying skin falling off of his exposed bones, or the brain matter oozing out of the hole in his head, but because this zombie had the misfortune of getting stuck behind a fence.

The zombie struggled against the fence in a vain and pitiful attempt to reach me, quite oblivious to the fact that five feet to his left, the fence ended, and there was a clear line of attack from that point to me. "Must be the hole in his head," I joked to myself, as I proceeded to do the only humane thing possible, and put the poor beast out of its misery (with my trusty minigun, no less).

Of course, the real problem with our zombie friend wasn't the hole in his head--it was with the overly simplistic marching orders given to it by its creator (basically, go in the direction of the player, and attack when you get close enough).

Fortunately, thanks to much better path-finding algorithms, games these days don't suffer much from the "zombie" problem. But if you look closely enough, you'll still see a number of problems caused by simplistic artificial intelligence.

For example, after single-handedly clearing a room full of baddies, if you run into some stragglers, chances are, they won't run, they'll attack you blindly--showing no more intelligence than the bad guys in your average Schwarzenegger flick ("Yeah, sure, this funny-talking muscular dude has just smoked 2 bazillion of our guys, but I'll go ahead and attack him anyway!"). A thinking opponent would run for cover or join up with surviving comrades.

In most games, the computer opponents never communicate with one another, never conspire together toward a common end, never run when they are hopelessly outnumbered, never ambush the player, never learn from their mistakes, and are otherwise pretty dull and uninspiring (their stunning 3D appearances notwithstanding). It's no wonder the game world is dominated by multiplayer games--human opponents are a heck of a lot more challenging and more interesting than their digital counterparts.

This sorry state of affairs is unlikely to resolve itself completely anytime soon. By our best estimates, computers won't have anywhere near the processing power of the human brain until around the year 2020.

So for the time being, anyway, we are stuck with opponents significantly more dimwitted than humans (and, quite often, significantly more dense than your average cockroach). On the bright side, however, today's blazing fast graphics cards (with built-in support for everything from geometry transformation to shadows and realistic lighting) have given game developers a little more room to play around with such luxuries as physics simulation and--more pertinent to the topic of this seminar--artificial intelligence.

One particular approach to artificial intelligence that I'm fond of requires only a minimal amount of effort to implement, but still gives rise to some fairly sophisticated (as well as unexpected) behavior. This approach borrows from concepts in artificial intelligence, biology and statistics.

In this seminar, I'm going to develop the principles behind this hybrid approach, provide you with source code to try on your own, and finally show you an award-winning artificial life simulator I built using the method.

Biology 101

The artificial intelligence approach I'm introducing in this seminar is firmly rooted in the empirical framework of biology. Which means, for one thing, that some of those boring facts you were taught in high school biology class are actually useful after all.

At that point in your life, however, you were probably more interested in flirting with the hottie sitting three seats to your left than you were in dreaming up ways to apply biology to artificial intelligence. But no matter--the biology is easy enough to relearn. In fact, there's really only one aspect of biology that's relevant to our discussion, and that is *genetics*.

The ideas that make up modern genetic theory are important to us because the method I'm describing relies quite heavily on what's called a *genetic algorithm*.

A genetic algorithm tries to emulate the process by which species evolve. Why? Mainly because species evolve on their own. There's no need for an intelligent, guiding force to drive the evolution process. All that's needed are the properties of inheritance and randomness (I'll explain why in a moment). These two properties are trivial to implement on computers, and yet, as we see in real-life, they can give rise to some very sophisticated, exquisitely adapted systems (including you and me).

The general approach of genetic algorithms is to start from some initial system, and randomly modify the system. If the new system performs as well or better than the old one, according to some standard of performance, then the new system is kept, and the old one discarded. The process repeats until the system reaches some specified performance level.

In the case of artificial intelligence, a genetic algorithm might start with a fairly unsophisticated computer opponent, and randomly modify some of the factors that

determine its behavior. If the new computer opponent is better at killing the player than the old one, then the new factors are kept. The process repeats until the computer opponent reaches a high skill level.

Ultimately, all genetic algorithms have their theoretical basis in the empirical phenomenon of *natural selection*--especially the algorithm I'm writing about, which models natural selection even more closely then the approach just described.

In the next section, I'll introduce you to all the biology you'll need to know to understand what natural selection is and how it works (don't worry, I promise to make it so painless you won't even know you're learning something useful until it's already too late).

In the Beginning…there was Darwin

You remember Charles Darwin--the gray-headed, bearded guy whose mug is usually plastered on the front of *Origin of Species*. Darwin's ticket to fame and his great contribution to science was his realization that *natural selection* is responsible for the birth and development of species.

Before describing what natural selection is, it may be instructive to describe what it is *not*. Hollywood has it all wrong. Contrary to what you see in the movies, there is no driving force that compels species to evolve into higher states of being. Nor is it possible for a species to evolve rapidly--changes from one generation to the next tend to be extremely small, genetically speaking. Nor do organisms themselves evolve (*species* evolve, but any given organism's genetic material is fixed from birth).

What is natural selection, then? Simply defined, *natural selection is the automatic selection process that arises in organisms that replicate in an imprecise way.*

Imprecise replication among a group of organisms introduces a lot of individual variations. Because of these variations, some of the organisms are better at reproducing than others. These organisms have progeny that are similarly good at reproducing. In this way, the "fittest" organisms come to dominate the population (where fitness is defined as a measure of reproductive success). Conversely, unfit organisms tend to represent a smaller and smaller percentage of the population (and, depending on just how unfit they are and how scarce the resources are, they may just die out altogether).

This is the essence of natural selection--so-named because the selection process that favors the fittest organisms is entirely natural (not mediated by external agents, like humans).

Now I'm going to briefly introduce the biological mechanisms that enable the selection process, since as I mentioned before, the approach I'm introducing stays very close to the underlying biology.

What Makes it Tick?

The heart of natural selection is inheritance--specifically, inheritance of traits from organisms to their offspring. Inheritance is mediated through *DNA*, which is a biological instruction set (or "blueprint") for creating and maintaining an organism over the span of its life.

It is DNA that is replicated and passed to offspring, and hence, DNA that is ultimately responsible for the inheritance of traits.

An organism's DNA is divided into one or more *chromosomes* (humans, for example, have 23 unique chromosomes). Chromosomes are linear strands of DNA, usually packaged in a compact shape. Much of the DNA in chromosomes doesn't encode anything, but the rest of the DNA is comprised of *genes*. Roughly speaking, a gene is a span of DNA that encodes a particular characteristic of the organism.

Virtually every cell in your body contains a complete copy of your genetic code. In fact, almost all your cells contain *two* copies of each chromosome. The two copies, referred to as *homologous chromosomes*, are very similar but not identical to each other. One copy you inherited from your father, and the other, from your mother.

G*ametes* (sperm in males, eggs in females) are an exception to this rule, as they contain only one copy of each chromosome. Why just one copy in this type of cell? So that offspring (which result from the union of sperm and egg) receive one chromosome from their mother, and one from their father. Just which copy they receive, as well as the interplay between the two inherited copies, determines the characteristics of the offspring.

Gametes are created in a process known as *meiosis*, which is critical to the evolution process. In meiosis, the chromosomes of the gametes are created from the chromosomes of the parents. In a process known as *crossover*, the gametes randomly receive some genes from the father, and some genes from the mother. (The way crossover occurs is actually more constrained than I've made out, but the constraints aren't relevant for our purposes.)

Replication errors and the randomness of meiosis introduce genetic change--they give rise to all the individual variations that are required for the evolution of a species.

Now that you have some feel for the underlying biology, let's take a look at how we can model the natural selection process on computers.

From the Wet World to the Digital Domain

The two key properties that make natural selection possible are inheritance and randomness--and, as the saying goes, there are as many ways to implement these properties as there are programmers.

The particular approach I have chosen closely parallels what happens in the biological world. I model genes, chromosomes, and genomes. I also allow two digital organisms to mate, producing an offspring that combines the traits of the parents. This allows for some very interesting possibilities.

Under this approach, a gene is modeled as an array of floating-point values. These floating-point values encode some trait of the digital organism, just as real genes encode traits of living organisms.

In straightforward fashion, a chromosome is modeled as an array of genes, and a genome, as an array of chromosomes. I give organisms only *one* copy of each chromosome, which is less general than the biological correlate, but still flexible enough for our purposes.

When creating an offspring's genome, I merge the genomes of the parents by combining their constituent chromosomes. Two homologous chromosomes can be combined in one of two ways: the genes of the child chromosome can be formed from the numerical average of the parent genes (an option that wouldn't make any sense in nature); or the genes of the child chromosome can be randomly made up of the genes of the parents, with a 50/50 chance that any given gene will come from either parent. The latter method is called the *crossover* method, named after the biological process it loosely emulates.

After merging the parent genomes, the offspring's genes are then randomly modified according to a mutation rate. This rate is described by two numbers: a mutation chance, which represents the probability that any given floating-point value will be mutated, and a mutation rate, which represents the maximum amount of change possible in a single mutation.

I allow mutation rates to vary on a per gene basis, partially because this approach mirrors the underlying biology (some regions of DNA are more stable than others), but also because of the additional flexibility-- some genes propagate best with a high mutation rate, whereas others propagate best with a lower mutation rate.

You can see my implementation for yourself. In the Downloads section, you'll find a class library I've nicknamed "Darwin AI", which implements genomes, chromosomes, and genes as I've described them. Though far from optimized (there's a lot of dynamic memory allocation going on, for example), you should find the code relatively straightforward. I won't document the code here because you won't use it directly. Rather, you'll be using higher-level code I'll cover later.

Where to from here?

I've discussed the genetic component of adaptable artificial intelligence--what's missing now is the actual artificial intelligence. You can theoretically store any information you want to in digital genes. However, for artificial intelligence, what we really want to store are factors that completely characterize the behavior of computer characters. This will

allow us to evolve intelligent characters, possibly even characters that can adapt their strategies to the nuances of the players they fight against.

Now I'm going to introduce an elegant way to characterize the behavior of digital organisms, one that can be encoded entirely in digital genes. The beauty of this approach is that it relies on very simple concepts in artificial intelligence and statistics, yet it allows for an incredible degree of character complexity.

Finite State Machines

You can't wander the landscape of artificial intelligence very long without running into *finite state machines*. Formally, finite state machines are deterministic machines capable of taking on a finite number of states based upon a set of rules, which operate on events external to the machines. If this definition seems a bit abstruse, think of a finite state machine as a robot. Whenever an event happens, the robot looks at the rules it has to determine what state it should be in.

Every finite state machine consists of two sets: a *state set* and a *rule set*.

The state set is simply a set of different states the machine can be in. For a computer character (a particularly dumb one, at that), the states might be "attacking player" and "evading player".

The rule set is a set of rules that dictates which events are associated with which states. For example, a rule might specify that the event of "seeing player" leads to the state of "attacking player".

To give you a better feel for finite state machines, I've constructed a hypothetical one in Tables 1-2. Table 1 contains the state set for the imaginary computer character, and Table 2, the associated rule set.

State	State Description
State #1	Patrol.
State #2	Attack player.
State #3	Sound alarm.
State #4	Run away from player.

Table 1: The state set for the finite state machine.

Rule	Rule Description
Rule #1	As soon as game starts, execute State #1
Rule #2	If alarm is sounded and health is greater than or equal to 10%, execute State #2.
Rule #3	If player is visible and alarm is not sounded, execute State #3.
Rule #4	If health is less than 10%, execute State #4.

Table 2: The rule set for the finite state machine.

The computer character of Tables 1-2 wouldn't win any computer gaming awards--this is the level of intelligence exhibited by the guards in Wolfenstein 3D, a game more than a decade old. Still, the simplicity with which it's possible to construct a finite state machine is quite alluring. Let's examine the weaknesses of our imaginary computer character to see if we can find some way of improving it.

Building a Better Finite State Machine

The main problems with the finite state machine of Tables 1-2 are as follows:

1. The number of states is too limited.
2. The number of events the machine can respond to is too limited.
3. The rules are too rigid. The machine cannot adapt to the player and the player can predict exactly what the machine will do under all circumstances.

Of course, we don't have to abandon finite state machines to solve the first and second problems. We can solve them by adding more states and rules. This solution does come at a cost, however: when you add more states and rules, you need more code. For finite state machines, the additional code will be in the form of tedious "if-then-else" statements, which can easily span hundreds of lines of code for a character of even moderate complexity.

The second problem is trickier. How can we make a machine that is unpredictable and adaptable? One answer to this question can be found in the mathematics of *Markov chains*.

Markov Chains

A Markov chain is a process that consists of a finite number of states, each denoted by a number from 1 to n, where n is the total number of states. The process is specified entirely by n^2 transition probabilities, which dictate the likelihood of transitioning from

one state to another. The probability for any transition can be described by a single real number, often symbolically denoted p_{ij}.

Markov chains have a natural matrix representation. This matrix, often called a *transition* or *stochastic* matrix, is shown below for an *n*-state chain:

$$\begin{pmatrix} p_{11} & p_{12} & p_{13} & \cdots & p_{1n} \\ p_{21} & p_{22} & p_{23} & \cdots & p_{2n} \\ p_{31} & p_{32} & p_{33} & \cdots & p_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & p_{n3} & \cdots & p_{nn} \end{pmatrix}$$

Markov chains have all sorts of cool properties. One is that you can calculate the probability a process will have gone from one state to another after *m*-state changes simply by raising the matrix to the *m*th power. But from our perspective, the greatest thing about Markov chains is that they introduce unpredictability into state changes--not adaptability, mind you, which is a much more complicated property, but simple unpredictability. Since the state changes are governed randomly, you can't say with definite certainty what the next state will be.

Applying Markov chains to game development is pretty straightforward. Just let the states of the process be the states of the computer character, and assign transition probabilities accordingly. For example, we could say the probability of transitioning from the state of "attacking player" to the state of "evading player" is 1% (perhaps the character is very proud and rarely runs from a fight).

The problem with this approach is that the character is completely oblivious to the state of its environment. Nowhere in a Markov chain can we specify that the probability of attacking the player increases if the player is visible (and yet clearly, that's the kind of behavior we need). We really don't want the transition probabilities to be fixed, as it would be in a Markov process. Rather, we want the transition probabilities to vary depending on the state of the environment. For example, if the character can see the player, then the probability of transitioning to the state of "attacking player" should be relatively high, whereas otherwise, it should be very low. What we need is a hybrid between Markov chains and finite state machines--something with the strengths of both and the weaknesses of neither. In the next section, we'll explore just such a method.

How to Make a Thinking Zombie

To construct our hybrid approach, we're going to borrow from finite state machines the idea that the character can take on a finite number of states. And we'll borrow from Markov chains the idea that instead of moving from one state to another based on a set of inflexible rules, we'll move between states based on transition probabilities.

Our unique contribution will be to vary the transition probabilities with the state of the environment, in a way that lends itself to the genetics-based tools developed in yesterday's material.

To do this, I must introduce the concept of an *environmental sensor*. An environmental sensor is a function that maps from some aspect of the environment to a real number. For example, a sensor could return 1 if the player is visible from the point of view of the character, and 0 if the player is not visible. Another sensor could return the distance from the player to the character. Another could return the number of computer characters the player has killed thus far.

Essentially, environmental sensors are the means by which the character senses the environment. The more sensors you have, the smarter you can make the character (in general; obviously this is not true if you choose useless sensors, like sensing the color of nearby grass!).

How can we use these sensors? The approach I have developed relies on a set of probability coefficients. If there are m sensors, then there are $m + 1$ probability coefficients for each state transition (and there are n^2 state transitions, where n is the total number of states). These probability coefficients are defined in such a way that the relative probability of transitioning from state i to state j is given by the following equation:

$$p_{ij} = Q_{ij0} + (Q_{ij1}E_1 + Q_{ij2}E_2 + \cdots + Q_{ijm}E_m)$$

where Q_{ijk} is the kth probability coefficient for the transition from i to j, and E_n is the value of the nth environmental sensor.

The interpretation of this equation is simpler than you might think. Basically, the probability of transitioning from one state to another depends (in a linear way) on the value of all the environmental sensors, and on the value of a "base" probability Q_{ij0}. The exact way in which the transition probability depends on the sensors is dictated by the values of the probability coefficients (for example, if a coefficient is zero for a given sensor, then the probability doesn't depend on the value of that sensor at all).

The beauty of this approach is that it expresses the logic for all state transitions in a single, unified framework, one that is relatively simple to code (no hundreds of lines of if-then-else statements as required for finite state machines) and yet is still extremely powerful. Moreover, this approach suits itself perfectly to the genetics algorithm introduced earlier (as I'll show you in a moment).

Let's take a closer look at environmental sensors and how they influence transition probabilities.

The Mechanics of Environmental Sensors

There are an infinite number of ways we can assign real numbers to environmental factors, but some of these ways allow for greater flexibility than others. Why? Principally because the transition probabilities are *linear* combinations of the environmental sensors, and this intrinsic linearity limits the ways in which the environmental sensors can influence the transition probabilities.

For example, say we want a character that is just standing around to start patrolling the area if the player is not visible to the character. This means we want the transition probability from the state of "standing around" to the state of "patrolling" to be high if the player is not visible, but low otherwise. But if our only environmental sensor describing the visibility of the player is equal to 1 if the player is visible, and 0 otherwise, then we're in a bit of a quandary. We can't directly increase the probability when the player is not visible, since the sensor will be 0 (and any coefficient choice, multiplied by zero, is still zero).

It turns out there is a solution to this particular problem: use a large base probability for the transition and a *negative* coefficient for the visibility sensor. This way, when the player is visible, the relative probability will be low or zero, but when the player is not visible (and the sensor equal to 0), the probability will be equal to the base probability.

However, there would be no solution to this problem if we made it slightly more complicated. All we have to do to make it insoluble is throw in another Boolean sensor, such that when it's equal to 1, we want the transition probability from "standing around" to "patrolling" to be high, and when it's equal to 0, we want it to be low. Now since the base probability is high (as required for the visibility sensor), when the new sensor is equal to 0, we cannot force the transition probability to be low by modifying the coefficient (since, again, any coefficient choice, multiplied by zero, is still zero). You might think, "Invert the value of the new sensor," but what if other transition probabilities won't allow this? For a complex character, one that has dozens of states and at least as many sensors, the likelihood of conflicting requirements is quite high.

There are two solutions to this problem. The first is to use not just linear combinations of the environmental sensors, but rather, all sorts of combinations, ones that permit a higher degree of flexibility. The drawback to this approach is that it tremendously increases the number of coefficients we need, and this presents a technical problem if we want to give the character the ability to adapt (more on this later).

The second solution is to add extra sensors that express the same data in different ways. For example, we can have one environmental sensor equal to 1 when the player is visible, and 0 otherwise, and another equal to 0 when the player is visible, and 1 otherwise, and so on, for various other sensors. This approach is much more computationally efficient, but it does require you think carefully about the sensors you are using. I've chosen to go with this latter solution, since I see no feasible way of implementing the former (but don't let that stop you from trying!).

To make sure you have a firm grasp of these ideas, let's revisit the finite state machine introduced earlier and see if we can approximate it using our hybrid approach.

A Practical Example

Recall the finite state machine constructed earlier responded to four different environmental stimuli: the start of the game, the health of the character, whether or not the alarm is sounding, and the visibility of the player. This means we'll need at least four separate environmental sensors.

We can define these sensors as follows:

Sensor 1: Boolean sensor equal to 1 if the game has just started, equal to 0 otherwise.
Sensor 2: Equal to the health of the character, ranging from 0 to 1.
Sensor 3: Boolean sensor equal to 1 if the alarm is sounding, equal to 0 otherwise.
Sensor 4: Boolean sensor equal to 1 if the player is visible, equal to 0 otherwise.

Our finite state machine could respond in four ways: patrolling, sounding the alarm, attacking the player, and evading the player. So we'll define four states for the character:

State 1: Patrolling.
State 2: Attacking the player.
State 3: Sounding alarm.
State 4: Evading the player.

Now let's step through the rules of the finite state machine one by one and construct their equivalent transition probabilities.

As soon as game starts, execute State #1.

We can reformulate this as follows: regardless of the current state of the character, if the game has just started, then the probability of transitioning to State 1 is very high. Mathematically, this means that p_{i1} should be high if the game has just started, and low otherwise. One choice for the probability coefficients that meets these requirements is shown below:

$$p_{i1} = \underset{0}{Q_{i1,0}} + \underset{(1)E_1}{Q_{i1,1}} + \underset{(0)E_2}{Q_{i1,2}} + \underset{(0)E_3}{Q_{i1,3}} + \underset{(0)E_4}{Q_{i1,4}}$$

You can see if $E_1 = 1$, as it will if the game has just started, then the probability of transitioning to the first state (patrolling) is equal to 1, and otherwise, the probability is equal to 0.

Now we proceed to the second rule.

If alarm is sounded and health is greater than or equal to 10%, execute State #2.

This rule cannot be implemented exactly, only approximately. Here's one choice of many:

$$p_{i2} = \underset{Q_{i2,0}}{0} + \underset{Q_{i2,1}}{(0)E_1} + \underset{Q_{i2,2}}{(3)E_2} + \underset{Q_{i2,3}}{(0.3)E_3} + \underset{Q_{i2,4}}{(0)E_4}$$

As you can see, if the alarm is sounded, and the health is equal to 10%, then the probability of transitioning to State 2 (attacking) is 60% (and it increases from there with increasing health). This means that even if the alarm is sounded, and the health is greater than 10%, there's no guarantee the character will attack. Conversely, there is some (small) probability that the character will attack even if the alarm is not sounded, or the health is less than 10%. This adds an element of unpredictability to the character.

If player is visible and alarm is not sounded, execute State #3.

This rule can be implemented exactly with the following choice of coefficients:

$$p_{i3} = \underset{Q_{i3,0}}{0} + \underset{Q_{i3,1}}{(0)E_1} + \underset{Q_{i3,2}}{(0)E_2} + \underset{Q_{i3,3}}{(-1)E_3} + \underset{Q_{i3,4}}{(1)E_4}$$

Note that in some cases, the probability will be negative. This should be interpreted as zero probability.

If health is less than 10%, execute State #4.

This rule is another one that we can only approximate. Here's one choice of coefficients:

$$p_{i4} = \underset{Q_{i4,0}}{1} + \underset{Q_{i4,1}}{(0)E_1} + \underset{Q_{i4,2}}{(-3)E_2} + \underset{Q_{i4,3}}{(0)E_3} + \underset{Q_{i4,4}}{(0)E_4}$$

For these particular coefficients, the probability of transitioning to State 4 (evading) is 70% if the health is equal to 10%, and greater still with decreasing health. This means that sometimes the character will run away even if it has plenty of health, and other times, it will fight it out to the bitter end. However, most of the time, the character will follow the rule, running away when it is very weak, and fighting otherwise.

You may be wondering if we've gained anything by using our hybrid approach. The answer is yes. For one, our character has made the transition from a rigid, inflexible automaton to a character whose behavior cannot be predicted, even by the designer.

Moreover, this behavior isn't simply random, but rather, it's randomness *directed* by the character's environmental sensors, ensuring reasonable (if varying) responses to environmental stimuli.

Another benefit is that we have plenty of room to increase the complexity of our character, even without adding more states or sensors. All we have to do is fiddle with the probability coefficients.

For example, we could make it very unlikely that the character will switch states if it's in the process of sounding the alarm--even if it's health drops below 10%. The way it stands now, the lower the health of the character, the greater the chance it will interrupt whatever it's doing and flee (which may or may not be the behavior we want).

We could also give the character a "guilty conscience" by tweaking the coefficients so that when the character is currently fleeing, the probability of switching to attack mode increases. These subtle nuances (and many more!) can be easily incorporated into the character, at virtually no cost. All the code is the same--only the probability coefficients change. Such is the extraordinary power of our hybrid approach. And when you add more sensors and states (as you would for a real character), the complexity of the character and depth of intelligence increase exponentially, completely unlike finite state machines.

Now that you know how probability coefficients work, let's see how we can package these coefficients into a digital genome, which is something we'll need if we want to evolve digital organisms using the genetic framework presented earlier.

Expressing Transition Probability Coefficients in Digital DNA

Recall that in organisms, genes encode particular characteristics. Moreover, functionally related genes often fall on the same chromosome (which makes sense from an evolutionary perspective). Taking these considerations into account, I chose to organize the coefficients in the manner shown in Figure 1.

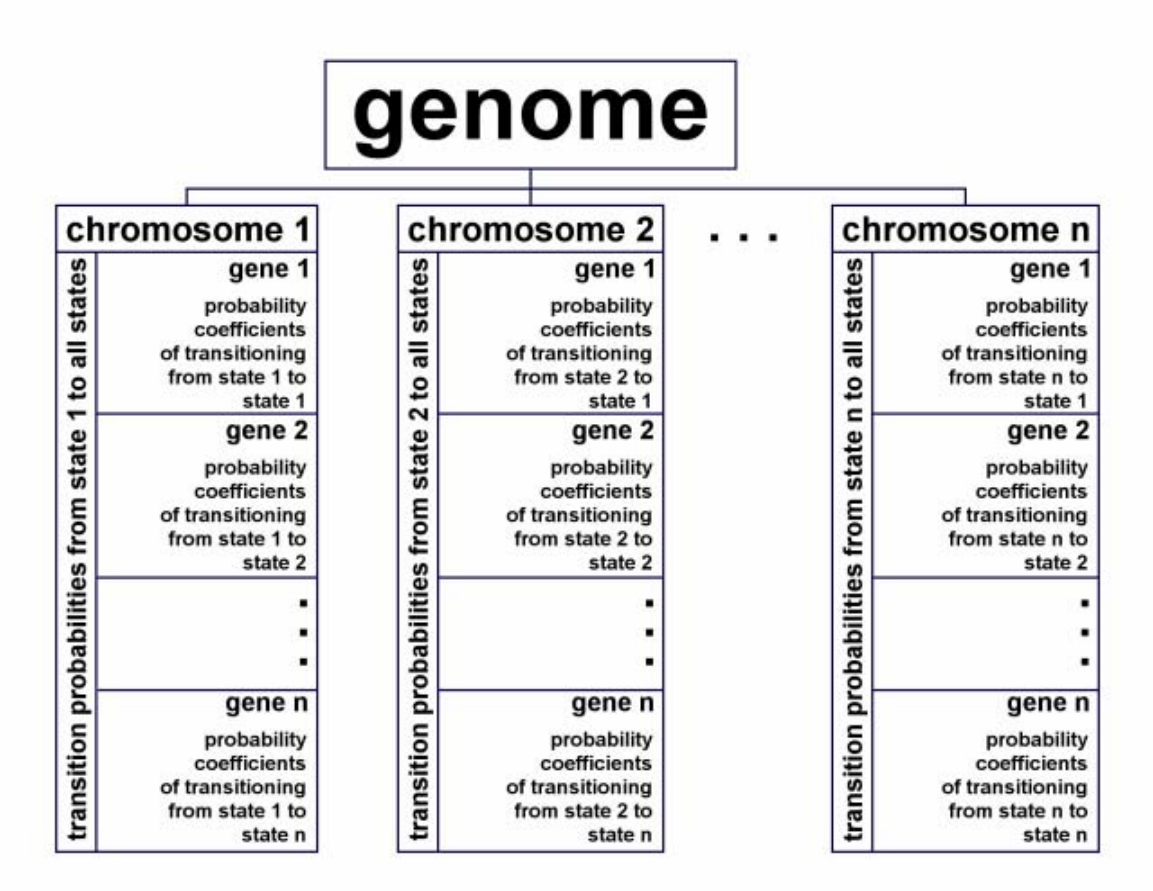

Figure 1: The genome for a computer character.

As in real life, each gene encodes a particular characteristic of the organism (namely, the probability coefficients of transitioning from one state to another). Further, genes are grouped by similar function (e.g. collectively, all genes on chromosome 1 represent the probability coefficients of transitioning from state 1 to all other states).

Since the probability coefficients completely describe the behavior of the character, we can effect any possible change by modifying the genome--a highly desirable property. In the next section, I'll describe how we can use this property to evolve intelligent computer characters.

Creating Intelligent Characters

The process of evolving intelligent computer characters begins with the character's states and environmental sensors. The more of these two building blocks you supply, the smarter your characters can become. This doesn't mean you should overdo it, however. The more states and sensors you have, the longer the evolution process takes. With m sensors and n states, there are $(m + 1)n^2$ probability coefficients that need to be determined--which is no small task.

Clearly, you need enough states to cover all possible behaviors you would like to see in the game. What might not be obvious, however, is that it's often beneficial to break up one behavior into several. For example, instead of having "attack the player" be the only attack state, you could use the combination, "attack from behind," "attack from in front", and "attack from the side." You could also throw in attacks with different weapons, or with different dodging patterns.

Similarly, you do need standard sensors, such as those for the player's visibility, the character's health, and so on, but you shouldn't stop at the obvious ones. A sensor that counts how many of the character's allies are living would enable the character to intelligently decide whether to fight or to flee. Likewise, a sensor that counts the player's remaining ammunition would give the character a better idea of what strategy to pursue (if the ammo is practically unlimited, no special strategy will work; but if it's low, the character might choose to fight it out, or perhaps dodge in an effort to waste the remaining ammo, even if the character's health is low).

Other creative sensors include the distance from the character to the player, whether or not the character is being damaged, which areas of the world the player is familiar with, the health of the player, and the weapon the player is using. And this is just a small sampling. There are literally dozens of other possible sensors that can enable your character to respond more intelligently.

Choosing good states and sensors is the single most difficult part developing intelligent characters. To illustrate this point, let's take a look at one example where there is a very good and a very bad way of choosing sensors and states that enable a specific behavior.

Choose Wisely

Suppose you want the character to evolve the ability to lure the player into unfamiliar territory. What states and sensors do you need? Well, there's one obvious choice: have a state "lure player into unfamiliar territory". This is the obvious choice, but it's also a very poor one, because if you create this state, you haven't gained very much by using our hybrid method. Sure, the character can "lure player into unfamiliar territory", but what else can it do? Maybe it will learn to lure the player into unfamiliar territory when its health is low, or when other conditions are met. But those benefits pale in comparison to the rewards that can be reaped by thinking about the problem a bit more creatively.

Instead of having a mega-complex state, "lure the player into unfamiliar territory", a better approach is to replace it with *K* simpler states (where *K* is the total number of territories), each one in the form, "lure player into territory *Y*." Then add Boolean sensors, one for each territory, in the form, "Is territory *Y* familiar to the player?" This way, the character can lure the player into territories based on different considerations--not merely on the basis of familiarity.

Under this approach, the character might lure the player into a territory with an unusually large number of comrades (providing the character has sensors to count the number of comrades in each territory). Or the character might lure the player into a territory because, historically, players have performed poorly there (perhaps because the lighting is low, or because the environment provides cover for the character). Or the character might lure the player into a territory because it has plenty of health packs the character can use to rejuvenate. The possibilities are endless--but only because the choice of states and sensors give the character the requisite flexibility.

You may not even need states to be as complex as "lure player into territory *Y*." You might get by with states as simple as "go to territory *Y*." An evolved character might figure out the luring part on its own, rapidly switching from the state "go to territory *Y*" to the state "attack player" (which is effectively what luring is). This switching would occur by having a high probability of transitioning from "attack player" to "go to territory *Y*", and visa versa. However, one drawback to this approach is that characters that have evolved the ability to lure will *always* lure; they won't go to territories or attack the player for any other reason. You can solve this by adding multiple states for "attack player" and "go to territory *Y*". Only a pair of these states will be necessary for luring; the rest are free to evolve for other functions.

I should point out that just because a character *can* evolve the ability to lure, doesn't mean it *will* evolve that ability. Sure, if you choose the right coefficients, it will lure, but if you evolve the character, then you can't absolutely guarantee specific behavior (maybe luring doesn't lead to the fittest characters). Nonetheless, your choice of coefficients and choice of a selection rule can greatly influence character behavior, as we'll see later on.

Once you have chosen your states and environmental sensors, you're ready to begin evolving your own digital organisms. Let's take a closer look at how this is done.

Selecting for Smarts

In the real world, if you wanted to evolve smarter animals, you'd let a group loose in a complex environment and let nature take its course. In the virtual world, the process is pretty much the same. You let your characters battle it out with the player, and then reproduce the best of them, repeating the process until the characters reach a high state of evolution.

There are two main differences between the real world and the virtual world that pose an implementation problem. First, in the real world, we're starting from animals that already

possess some degree of intelligence. But if all you do is initialize your digital genomes with random values, you're basically starting from primordial goo. Consequently, none of the characters will do well, and procreating the losers will just produce more losers (in theory, you might eventually stumble onto an intelligent character, but it would probably take millions of computer years).

Second, in the real world, the fit organisms come to dominate the population, while the unfit organisms become extinct. It all happens naturally, which, if you remember, is why the process is called "natural selection." In the virtual world, there's no such "natural" selection process. Or rather, any natural selection process that exists isn't likely to produce interesting and intelligent characters.

The solution to the first problem is straightforward: you just need to manually initialize the genome. You shouldn't worry about creating the smartest possible character (which you couldn't do even if you wanted to--there are too many coefficients for your best guess to be accurate). Instead, just create one with some modicum of intelligence, enough that the selection process has *something* to work with. This also has the advantage of giving you some degree of control over how your character behaves, at least initially (keep in mind you can't absolutely guarantee that evolution will keep your preprogrammed behavior).

The solution to the second problem is more complicated. Theoretically, you could just let the characters run around, reproducing of their own accord, while the player hunts them. After the player has killed them all, you could proceed to resurrect and mate the ones that reproduced most often. However, this will result in characters that are quite good at running away from the player and reproducing--since after all, the ones stupid enough to waste their energy by attacking the player will die out quickly. Unless you're creating a deer hunting game (and heaven knows we don't need any more of those!), this probably isn't the behavior you want.

The first step toward correcting this problem is to prevent characters from reproducing on their own. Instead, you measure the "fitness" of individuals in a group, and after the player has killed them all, you resurrect and mate the best of the lot. However, even this approach requires considerable care, since there's the whole question of how you define "best of the lot".

If you say the best are those that survive longest, then chances are, characters that are quite good at evading the player will evolve, since these characters will out-survive the ones that attack the player (alas, cowards generally live longer than heroes!). But playing a game where all the characters are extremely good at running away is no fun at all. Consequently, the selection criterion must be more sophisticated than simply duration of survival.

The criterion I suggest is the total amount of damage the character does to the player, with possibly some extra "bonus" points thrown in for desirable traits such as duration of survival (characters that last longer are more interesting than those that don't, even if they

do the exact same amount of damage). This selection rule will evolve characters that are good at damaging the player, but not necessarily good at avoiding player damage (not unless it enables them to damage the player more).

You can use this criterion to determine the fitness of all the characters. Choose the top 10 or so, reproduce them by merging their genomes and randomly modifying the result, and repeat as long as you like.

That's the essence of my approach to adaptable artificial intelligence. Before I introduce the included source code and show you the award-winning application I developed around these principles, we need to spend time covering some technical details that you'll run into if you try this method in your own games.

Devil in the Details

Evolution takes place on a large timescale. We've substantially reduced the timescale by limiting the size of the genome (which is equal to the number of probability coefficients). However, for characters of any respectable complexity, the number of undetermined factors is still large enough that evolution of an intelligent character will take quite some time--easily thousands of game hours. Manually initializing the genome is critical, but it isn't enough. These manually initialized coefficients need to be finely honed through thousands of generations of digital characters.

The long time it takes to refine the digital genomes may or may not be a problem. If you have hundreds of beta testers who all have access to the Internet, then you can distribute the problem among all the gamers, and the results will be excellent (as long as the players are willing to put up with some character stupidity during the early hours of play).

If you're writing a smaller title, and have only a handful of beta testers, then you have a real problem. You won't be able to evolve characters in any reasonable timeframe. One thing you could do is manually program a simulated player, and then have it battle the evolving characters. However, the intelligence of the characters will be limited by the intelligence of the simulated player, which means you don't gain a thing by using evolution (after all, you could just use the simulated player AI).

About the only option you have is to evolve two or more species independently, and pit them against each other. Why battle by species, instead of every character for itself? Because if you group the characters by species, they can evolve cooperation (assuming the states and sensors allow this).

Even with this solution, however, you are guaranteed of only one thing: the species will be good at killing each other, not necessarily good at killing the player. However, there should be a large degree of overlap between the skills needed to kill other species, and the skills needed to kill the player (as long as they behave in similar ways). This means that the evolved species will probably be good enough at killing the player that your beta testers (however few) can take it from there.

Once you have evolved an intelligent genome, you have two options. You can keep the genome fixed over the course of the game, or you can allow it to adapt to the player. The latter option is more interesting, but there's one potential problem you should be aware of. Sometimes, characters will achieve superior fitness scores not by virtue of superior skill, but by blind luck. Reproducing these characters won't increase their intelligence, and in fact it may decrease it. The way to minimize this is to look at fairly large numbers of characters before you decide to reproduce, and then when you do reproduce, always do so with the top 10 or so, which will virtually guarantee that most of the characters are worthy of reproduction.

Another detail you should be aware of involves how much control you have over character behavior when you manually initialize the genome. You may be thinking to yourself, "Does it really matter how I initialize the genome? Won't all genomes converge to the fittest genome possible, as defined by my selection rule?" The answer may surprise you: *not necessarily*, in fact, almost certainly not.

This strange phenomenon is due to the shape of something called a *fitness landscape*. The fitness landscape, technically an *n*-dimensional hypersurface (where *n* is the number of probability coefficients), describes the fitness of our digital organism for all possible coefficient choices. The "height" of this hypersurface at a point **v** represents the fitness of the digital organism for the choice of coefficients described by the *n*-dimensional vector **v**.

To visualize the fitness landscape, let's assume there are only two probability coefficients. In this case, the hypersurface becomes an ordinary surface in three-dimensional space. The height of this surface represents the fitness of the organism for all possible choices of probability coefficients. So if you wanted to find out how fit the organism was with a choice of 0.5 for the first coefficient, and 1 for the second coefficient, you would find the height of the surface at the point (0.5, 1). This would give you a numerical value for the fitness of the organism for that particular choice of probability coefficients.

In theory, we could use the fitness landscape to choose the best coefficients. In the two-dimensional case, all we have to do is look for the highest peak on the surface--the Mt. Everest of our fitness landscape, if you will. The coefficients represented by that point correspond to the fittest possible organism.

The problem is we don't know the shape of the fitness landscape. We could compute it explicitly, but doing so would take far too long for a character of any reasonable complexity. Hence we use the genetic algorithm introduced earlier. We start from some point on the surface (which corresponds to the initial values of the genome), and then we move in a random direction (which corresponds to randomly modifying the genome, or merging it with another genome and making small random changes). If our new elevation is higher than the old one (i.e. the new organism is more fit than the old one), then we

stay at this higher point, and continue the process. Otherwise, we go back to where we started from and try again.

This process continues until our fitness no longer improves from one generation to the next. This corresponds to finding a peak on the fitness landscape. However, just because we find "a" peak doesn't mean it's the *highest* peak out there. In fact, it probably isn't. Most fitness landscapes will have numerous peaks. And using the method I just described, we generally won't find the highest peak, since our elevation increases continuously until we hit a peak--we never go down the landscape, which would be necessary if there were any ravines or dips between the highest peak and us.

Figure 2 shows you what a fitness landscape might look like in the case of a character defined by two probability coefficients (obviously, a real character will have more, but there is no way to visualize the fitness landscape in such cases). Note that virtually anywhere we start, we won't find the highest peak using the algorithm I've described.

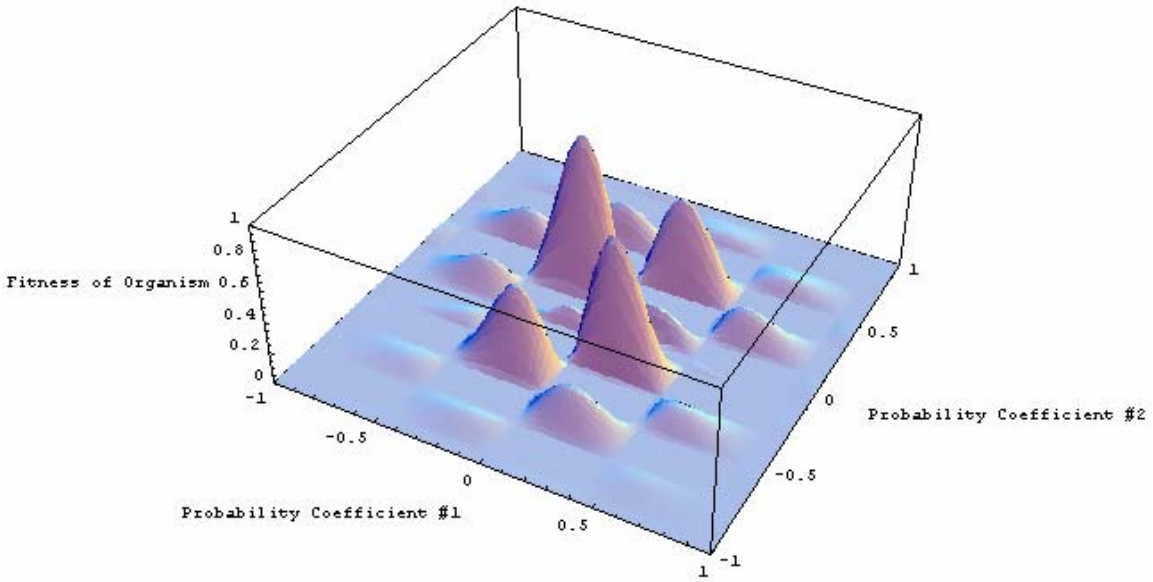

Figure 2: A representation of the fitness landscape.

So what does this all mean? For one, it means that the characters we evolve probably won't be as fit as they could possibly be. There are other choices of probability coefficients that have better fitness scores. However, this same drawback is also a plus,

since it allows you some degree of control over the character's behavior. If you *really* want a character that can lure the player into unfamiliar territory, for example, then initialize the genome appropriately. If you're lucky, the fitness landscape will have a peak close enough to your initial choice so that the luring behavior is preserved. If so, you get the best of both worlds: a highly evolved character that still does what you want it to. It may not be the fittest character possible, but then again, the fittest character possible may not exhibit the behaviors you want.

Astute readers will notice that my description of our genetic algorithm isn't quite correct. Recall that we assign a fitness score to the characters based on some criterion such as damage done to the player. However, as I mentioned before, sometimes the characters will (by blind luck) score higher than they should. This random noise enables our existing algorithm to traverse dips and ravines in the fitness landscape. But you shouldn't count on this behavior to find you the fittest possible character, as it almost certainly won't (especially if your scoring function is quite good).

The last technical detail I need to cover is the issue of *state updating*. Over the course of the game, you need to compute the transition probabilities and update the character's state. But when should you do this? My recommendation is to update at regular time intervals (say, once per second). If you want certain actions to be carried out until completion (say, you want the character to sound the alarm, with no chance of interruption), then you can temporarily suspend state updating over that interval. Keep in mind this behavior is artificial and may result in sub-optimal adaptation. In the case of sounding the alarm, for example, the character can evolve the ability to break off sounding the alarm and fight if the player attacks it (providing the character can sense when it's being attacked). But if you prevent such actions from being interrupted, then this behavior cannot evolve (obviously).

That sums up the technical coverage of the hybrid artificial intelligence method. Before I show you how I used this method in one of my own programs, I'll acquaint you with the source code, which includes a full implementation of the ideas discussed in this seminar.

The DarwinOrg Package

In another tribute to our favorite graying bearded one, I've named the artificial intelligence package *DarwinOrg*. Written around DarwinAI, the package implements everything you need to add intelligent characters to your games.

The only class in the DarwinOrg package is *Organism*. To use this class, set the number of states and sensors (through **SetStateCount ()** and **SetSensorCount ()**, respectively), give the sensors and states ASCII names, if you wish (using **SetSensorName ()** and **SetStateName ()**), and then set all the probability coefficients through the **SetTransition ()** function. Then at regular intervals, update the sensor values by calling **SetSensorValue ()** and perform automatic state updating by calling **UpdateState ()**. To determine the current state, call **GetCurrentState ()**.

The class supports many other functions. You can save a character to disk by calling **Save ()**, and load it from disk by calling **Load ()**. You can mutate the genome by a random amount by calling **Mutate ()**, and mutate the factors that govern mutation (such as mutation rate, etc.) by calling **MutateMutationFactors ()**. And most importantly, you can mate two *Organism*'s by using the overloaded '+' operator (as in, **Baby = Mommy + Poppy**, or, if you swing that way, **Baby = Mommy + Mommy**).

You'll find the complete source code to DarwinOrg in the Downloads section. I suggest you spend some time perusing the source code--there's a heck of a lot going on in there.

Now let's take a look at how I used this technology to develop an award-winning artificial life simulator.

Bugs!

Bugs! Screen Saver is an artificial life simulator that masquerades as a screen saver (see Figure 3). This screen saver simulates a mini-ecosystem, filled with evolving bugs and plants. You'll find the screen saver in the Downloads section. I recommend playing with the software for an hour or more to see just how powerful our hybrid artificial intelligence method is.

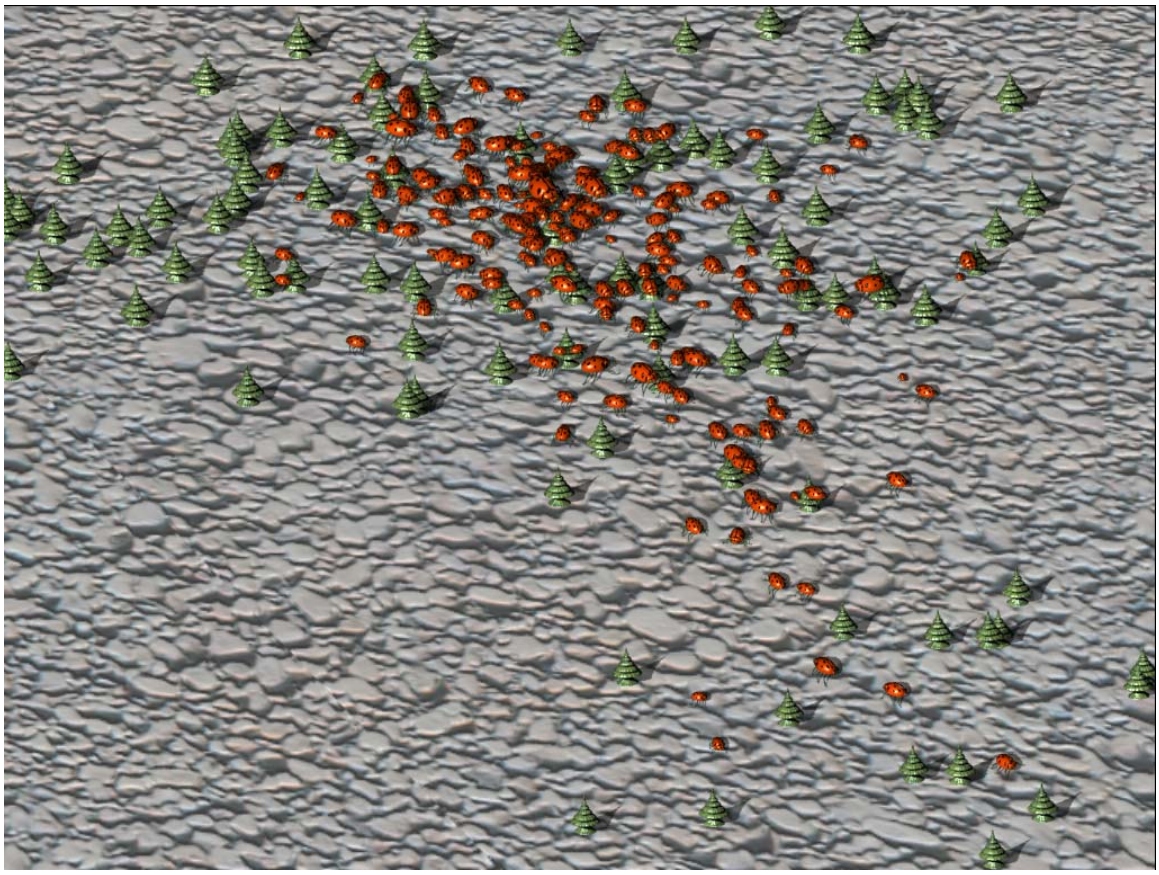

Figure 3: A screen shot from Bugs! Screen Saver.

The bugs and the plants in the screen saver are built using the techniques presented in this seminar. Both organisms have genomes that completely characterize their behavior, and both have multiple states and a number of sensors with which to sense the environment. For more information on the states and sensors, refer to the screen saver's online help system.

It's possible to watch the screen saver for hours and never see the same behavior twice. This is a testament both to the method the screen saver uses and the choice of sensors and states (recall that poorly chosen sensors or states would place severe limits on the maximum possible intelligence).

That concludes the seminar. Hopefully you'll be able to take the concepts presented here and use them in your own games, creating the next generation of computer characters--which, while not on par with human opponents, will at least be more challenging than our zombie friend.

See you in the chat session!