

Evolving AI Opponents in a First-Person-Shooter Video Game

C. Adam Overholtzer

Supervisor: Simon D. Levy

One of the major commercial applications of artificial intelligence (AI) is in the rapidly expanding computer game industry. Virtually every game that is sold today has some sort of AI, whether it is the "computer player" in a chess game, the opposing paddle in a one-player version of Pong, or the machinegun-toting enemies in a first-person shooter. Any virtual being that does not behave in a strictly pre-scripted manner has some sort of AI behind it. Sadly, the multi-billion dollar gaming industry has done very little to advance the field of AI. New games may sport stunning 3D graphics, but a realistic-looking enemy is not worth much if he is a moron – and not even a realistic moron at that. The industry has moved past the day when an AI opponent would not even react to the death of his teammate standing three feet away, but that does not mean these bots, as they are commonly called, have any serious thought processes. Most are completely reactive, and while they may often *appear* to behave sensibly, they still require three-to-one odds to beat a human player despite having perfect aim. The AI in these games often seems like an afterthought, with game designers unwilling to spend the time to implement more complex AI, much less waste the user's precious processing power being spent on rendering the 3D graphics.

My proposal was to take an existing first-person shooter and expand the capabilities of its AI. All of the "hard work," namely the graphics, physics and gameplay system, would be done for us and all I would do is focus on the AI and try to improve the thinking ability of the bots within the current system by implementing fairly orthodox planning algorithms. Some of the most basic planning algorithms have only recently been implemented in these types of games, and so it should be fairly simple to improve the existing AI by taking the planning algorithms to the next level of complexity. But before discussing how to do this, I will go back to the "square one" of bot design and discuss the issues in first-person shooters that every bot since "Doom" has had to deal with.

State of the Art: 1992

All first-person shooters (FPS) are 3D games, meaning the human player is shown a first-person view of a 3D environment that he or she can move about in, using the images on the

computer screen to decide when to turn, shoot at an enemy, etc. As far as I know, no bot in any FPS can “see” the environment as the human player can. A bot is only aware of a very small set of coordinates defining the most critical points on the map, often just his own location and the location of his enemy, the player. A simple line-of-sight function determines if the line between these two points is unobstructed, while another algorithm lets the bot know how close he is to a wall. Since the bot has no representation of the layout of the map, he has no way of knowing that the correct path to the enemy might be to take the first left and then make a right at the fork in the road. Other behaviors common to all but the earliest games include reaction to sound (*e.g.* gunfire), which is usually done in a crude-but-effective manner similar to line-of-sight, and patrolling, which is simply instructing a bot to walk back and forth between two given points. Virtually all of the improvements to FPS AI can be viewed as an expansion on one of these early ideas because most of the problems are due to the bot’s limited knowledge of the world.

State of the Art: 2003

Given the simplicity of the bots in even the most modern FPS, we decided it would be safe to work with one of the more developed AI systems in a popular commercial FPS, Unreal Tournament 2003 (UT2K3). The underlying code for UT2K3 can be viewed as two separate but highly integrated parts: the Unreal Engine and the high-level UnrealScript classes. The “engine” is the top-secret core of any game, and it controls the graphics rendering and physics for the environment. The Unreal Engine is a complicated and state-of-the-art “black box” that should be viewed as the thing that “makes everything work.” The UnrealScript files, on the other hand, are freely downloadable from the Internet and control most everything else: how weapons look and operate, how “items” in the environment work and what they do, how a game is played and what rules apply, and, most importantly, how bots think. It is through these UnrealScript functions that UT2K3 brings two new ideas to the basics of bot AI discussed above: pathfinding and team AI.

Pathfinding takes the idea that a bot can “patrol” between two points and expands it to allow for more realistic movement through the map. The person who designs a map places dozens of “pathnodes” throughout the map, marking all the major corridors, crossroads, hidey-holes, and landmarks with a node. These nodes are then linked in every possible manner so that now when a bot decides it wants to move from point A to point C, and the pathnode network lets

him know he can do so by jumping up to point B and then moving to point C – a big improvement over walking in straight lines and bumping off walls. The paths are given weights to indicate which ones are more likely to be used by the enemy, which are harder to navigate, which require a jump across a chasm, and so on. This pathfinding system is an excellent example of how the bot's abilities can be greatly improved by a relatively simple concept that adds very little to a bot's knowledge or capabilities (note that the pathnode network is hard-coded by the designer, so it is pre-determined and not learned). No longer will a bot walk straight into a wall, not knowing if he should turn left or right to go around it!

Team AI is another recent innovation in bot AI, and it basically allows a group of bots to function as a team by setting group objectives and assigning tasks to different members of the group. Again the logic is predetermined, but essentially the team AI will do things like tell two bots to guard the home base while the other three attack the enemy. If the base comes under heavy attack, the team AI allows the bots to communicate basic ideas such as: "Help, help! We're under attack! Come save us!" This may seem like a trivial improvement, but it is fact one of the major advancements in FPS AI in the last few years. Before, bots were generally oblivious to one another; now they can work as a group.

Given the impressiveness of UT2K3's AI, coupled with the fact that their bots are still pretty darn stupid, and the relative simplicity of the Java-like UnrealScript, we decided to move ahead with our project. Professor Levy and I attended the game publisher's "Unreal University" at North Carolina State University in November 2003. At this two-day conference for amateur Unreal developers, we learned the basics of UnrealScript and saw the work of others (nothing involving AI). We asked about doing AI work and were told that UnrealScript is powerful enough to handle complex algorithms, but that some of what we would want to do may require access to lower-level functions within the Unreal Engine. But no worry, we were assured, because a free academic license for the Engine would be available "soon" (a commercial license for the Unreal Engine costs \$350,000).

Excited by what we were told, Professor Levy purchased a copy of UT2K3 (the UnrealScript modifications require the original game to play) and I began reading the existing AI code to figure out what to do. After spending two weeks pouring over thousands of lines of virtually undocumented code and analyzing the observable behavior of bots in the game itself, I felt ready to recommend a course of action. Because I found the individual bot AI to be

reasonably impressive already and very difficult to modify (the “botAI” file contained close to one hundred functions and was highly dependent on dozens of other classes and hidden functions within the Engine), I decided to focus on the team AI, which is simpler, more accessible, and presumably easier to improve upon.

UT2K3 has several types team-based games in addition to the usual free-for-all game (Deathmatch), the more interesting of which are Capture-the-Flag and Bombing Run. Bombing Run is a game similar to football: each team has a goal they must defend and the object is to seize the “ball” in the center of the map and run it to your enemy’s goal. Of course all of this involves shooting your opponents as well, but anyone who is killed “respawns” back near his goal after a few seconds and continues to play. I quickly noticed that this was the one game that the current AI played *very* poorly. The bots’ ability to defend their own base was fairly limited, and once a bot got the ball, he would invariably abandon his allies and run like crazy towards the enemy goal. Since the enemy spawns near their own goal, a single bot without any cover from his allies does not make it very far. The bots’ tactics were so poor that a human team could easily defeat them and a battle between two teams of bots would go on and on without either team scoring a goal. Here, it seemed, was a game where simply planning (maybe some football strategies and the ability to run interference for the guy with the ball?) would greatly improve the bots’ performance.

The first few modifications I made to the code were extremely trivial – literally nothing more than changing a constant. As time passed, it became clear to me that very few nontrivial changes could be made within the limits of these few UnrealScript files. Too many functions either were so convoluted that I barely got the gist of what they were doing or else were dependent on functions buried inside the Engine. Many improvements *could* be made, but not without access to the C++ header files inside the Unreal Engine for critical features such as pathfinding, lest I waste time “reinventing the wheel,” so to speak. Since we were already a month into the semester, Professor Levy inquired about the free academic license we had discussed at the conference. Our contact informed us that the agreement was still being worked out (this was more than two months after the conference) and that it definitely looked like the academic license would require the “nominal fee” of \$15,000. Rather than continuing to struggle with all the unnecessarily complex code for a relatively simplistic AI, we decided it was “game over” for Unreal Tournament.

We really cannot blame the developers of UT2K3 because their goal is to make a game that works well and makes them a lot of money, nothing more. Their scripting language is designed to only allow the most trivial of changes, such as different 3D models, a new weapon, or a new set of rules for a game. What we wanted to do *can* be done, but it can only be done easily if we have access to the engine itself and selling licenses for their engine is how these developers make money. Thus, our next plan was to go with a game where money was not an issue: the open-source game Cube.

The Cube Engine

Cube is an open-source FPS, written in C++ for Linux and Windows, that is based on a very unorthodox engine that emphasizes simple 3D designs to provide a quickly and cleanly rendered environment. The Cube Engine is, to quote its website, “mostly targeted at reaching feature richness through simplicity of structure and brute force, rather than finely tuned complexity.” Not only is the code for Cube much shorter and simpler than that for UT2K3, but, because it is open-source, we have full access to *all* of the code and thus almost *any* change is possible (given enough time). Since the inner workings of Cube are more clear-cut and better documented, it is relatively easy to understand and modify.

There were downsides to using Cube, of course. The AI for Cube is extremely simplistic – little more than the basics I outlined earlier. The bot, or “monster,” only knows when he has seen the enemy walk in front of him, when he has been shot, or if he has bumped into a wall. There is also more of a problem with omniscience in Cube, because a monster in the HOME state may be instructed to move toward an enemy’s coordinates even when they have never seen him or her, and frankly, that is cheating. The abilities of the monsters in Cube are vastly inferior to those of the bots in UT2K3, and this is reflected in the fact that a single Deathmatch game in Cube commonly pits *thirty* puny monsters against the single human player. They are so easily killed and so stupid that the only way the player will lose is if he or she is unfortunate enough to get mobbed by five or six monsters at one time.

The AI for a monster is contained within a single function (a welcome change from UT2K3) and is very straightforward and completely deterministic. At any given time the monster is in one of several “states”, such as HOME, where he is homing in on his enemy, AIMING, where he is preparing to fire, and SLEEP, where he is standing around waiting for something to

happen (this state is never used in a Deathmatch). HOME is the most important of these states and in a Deathmatch a monster is generally in this state. To change from one behavior to the other the monster simply transitions from one state to the other. How could an AI be simpler?

The first thing I changed was to decrease the number of monsters in a game to ten while increasing their strength and speed to compensate. I also completely eliminated some of the monster types so there was not quite so much variance in the physical abilities of the monsters in a game. I found this setup to be about as difficult as the old Deathmatches with the benefit that a monster no longer dies in one hit and thus can survive long enough to do something interesting.

After reading through some of the code and playing the game for a while, I noticed that the one truly embarrassing problem with the monsters' AI was that when they bumped into a wall, which happens often, they either moved off in a random direction or, occasionally, tried jumping. Since plenty of ledges can be jumped onto, jumping is a good strategy sometimes but the monsters currently did this *infrequently* and *randomly*, meaning that I would see monsters stuck behind 2-foot-tall fences while others tried to jump over a 25-foot wall. So, the first thing I resolved to add was a function called `canjump()` that would look at the space in front of the monster, much like the existing collision-detection algorithm, and see if it was low enough for the monster to jump onto. Unlike any of my trivial improvements in UT2K3, this only took a few hours and now whenever a monster approached a ledge he could jump over, he would cheerfully leap right over it.

This new feature revealed a problem, however, and it is a problem inherent in an entirely deterministic planned system: balance. A human player does not always move in straight lines, and he or she would certainly not move in a straight line and jump over everything in his or her path. After all, if you jump around constantly you would draw attention to yourself and become an easy target... and you would look like an idiot. A human would go around some objects and jump others, depending on the circumstances. In our simple AI, the best we can do is randomize how often the monster will jump over small obstacles, because a monster that *always* jumps whenever he can looks ridiculous. Randomization may seem like a cop-out, but one thing I have noticed is that the difference between random behavior and planned behavior is often hard to distinguish in these games, if for no other reason than the human player is so preoccupied that he or she cannot carefully study the monsters' behavior. The question then, is how often should a monster jump when he runs into a low obstacle? One in three times? Half the time? How can

the AI designer determine these things? A human does not think this way, but in our simple Cube environment, the monsters *must* think this way.

We realized that given the simplicity of Cube, developing a straight hard-coded AI that would show real improvement would be impossible in the remainder of the semester. Rather than spending lots of time trying to figure out how the monsters “should” behave, we decided that it would be easier and more interesting to add an evolutionary algorithm (Mitchell 1996) to the monsters’ deterministic behavior. By evolving over time, the monsters could learn from experience what the best strategies were and would, for example, decide for themselves how often they should jump.

Evolutionary Ideas

The idea is that each of a monster’s possible decisions will be represented by a single value (true, false, or a probability) and all of these values combined will determine his behavior. This string of values, which we can call his DNA, is attached to an individual monster and whenever the monster needs to make a decision, he will now consult both the usual criteria *and* the corresponding value in his DNA. This is best explained by example: Suppose that you are a monster and you have bumped into something. In order to decide whether you should jump or not, you will consult both the `canjump()` function *and* check your DNA to see how often you “like” to jump. Every decision a monster could make, even trivial ones, has a corresponding value in the DNA that will determine his “unique” behavior.

At the end of a game, each monster is given a score based on how well they performed and five of the ten monsters are chosen to be reborn in the next game. Two copies of each monster will be passed on to the next game, but each of these is run through a mutation function that will randomly alter a small fraction of the values in the DNA. This way, the monsters will be changing a little bit each game and the ones that perform the best will live to the next game. Just like biological evolution, our game became a “survival-of-the-fittest” environment where only the most well-adjusted monsters survive and eventually, at least in theory, the monsters will become very good at surviving.

There were three stages to the implementation of our evolutionary algorithm: first, I added a few more behaviors to the monsters, then I designed the DNA and “attached” it to the monsters, and finally Professor Levy wrote the mutation and survival-of-the-fittest algorithms.

The new behaviors I added were fairly straightforward: seeking helpful items (health heals existing damage, armor lessens the damage you take when shot), wandering randomly, retreating when attacked, and “camping” near useful items.

To add the previously player-only ability to seek and use items, I first gave the monsters the ability to gain the benefits of health and armor by moving near the items, just as the human player can (called “picking up an item”). Then I added functions to allow the monsters to know when they were near an item (recall the monster’s limited knowledge of the world), and lastly I added a new state, `SEEK`, that simply points the monster in the direction of an item and has him walk towards it.

Wandering randomly, which gives the monster something to do when he doesn’t know where the human player is, was added by allowing a monster to transition into `SLEEP` in a Deathmatch and to move while in this state. A wandering monster will (probably) `HOME` if he sees the human player or may `SEEK` if he sees an item that he would like. Retreating works much the same way: a monster who has been attacked and “wants to retreat” (yes, the DNA determines that) will either seek a distant item, jump into the nearest teleporter, or simply run off in the opposite direction of the player... whichever is most convenient.

The new state `CAMP` is used to allow for a somewhat less obvious strategy: lurking near an item’s spawnpoint. In a Deathmatch, an item (health, armor, etc.) that has been picked up by someone will, after a short time, reappear (respawn) where it used to be, allowing someone else to pick it up. Since a player might need two health packs to fully heal, a good way to safely get two is to find one and then wait for it to respawn to get it again. This new state may be used after picking up an item to “camp” near that item’s spawnpoint. The final task was to connect all of these new abilities into the current system by adding transitions to `SLEEP`, `SEEK`, and `CAMP` in `HOME` and other appropriate places.

The next step in our implementation was to design the DNA sequence and connect it to the existing system. The DNA is represented by the following struct:

```
struct botDNA {  
  
    // THE BOOLEANS  
    bool caresHealth;    // will seek health or boost if needed  
    bool caresArmour;    // will seek armour at start of game (when he has none)  
    bool seeksArmour;    // will seek armour when he "needs" it  
    bool seeksBoost;     // will seek boost to get extra health  
    bool caresLOS;       // care if we can see the enemy or not  
    bool caresLostLOS;   // care if we saw enemy and now have lost him (NOT USED)  
    bool willFollow;     // will pursue an enemy (move to where we last saw enemy)  
    bool actCowardly;    // if shot, run away from attacker  
    bool willMoveAim;    // move when aiming? (false in original game)
```

```

bool willMoveSleep; // move when sleeping? (this would allow for random wandering)
bool willFaceSleep; // turn randomly when sleeping or walk in straight lines?
bool willFaceHome; // turn towards foe or wander randomly when homing?
bool willFaceSeek; // turn towards item or wander randomly when seeking?

// THE PROPORTIONS (range is 0 to 15)
int willCamp; // stay by health/armour spawn locations rather than search?
int willSleep; // if we've got nothing to do, should we give up just hang out?
int willJump; // if we're blocked, how often should we try jumping?
int willHome; // will we move towards the enemy when homing? (0 means NEVER move)
int willTurn; // will we turn towards the enemy? (tied to "willFace..." booleans)
int healthmin; // the least amount of health we'll tolerate
int armourmin; // the least amount of armour we'll tolerate

// THE DISTANCES (all integers from 4 to 64, increment by 4)
int targetdist; // farthest distance we will look for our enemy
int healthdist; // farthest distance we will look for health
int armourdist; // ... you get the idea...
int boostdist; // ...

};

```

These values were simply tacked on to the existing if-statements, so for example the statement

```
if(canjump() == true)
```

to decide if the monster should jump if he bumps into something now becomes

```
if((canjump() == true) AND (rand15 < monster->willJump)),
    given a randomly generated integer between 0 and 15 called rand15
```

In this way, every choice the monster can make is broken down into all the various options and the values in his DNA determine which choice set to use; *e.g.*, a monster that likes to `actCowardly` will run away after being shot, while a monster with that variable set to `false` will pursue his attacker rather than retreating.

The final state of implementation was for Professor Levy to write the various evolutionary algorithms. The first step was to turn my `botDNA` struct into a “bitstring,” a string of 0’s and 1’s (or bits), which can then be fed into a standard off-the-shelf mutation algorithm. Each Boolean got one bit while the integers got three or four bits apiece, for a total of 57 bits. So, the function `mutate (botDNA *pdna)` takes a pointer to a DNA sequence, converts it to a bitstring and then runs a mutation algorithm Professor Levy took from a textbook on genetic algorithms (Mitchel 1996). Essentially, this algorithm moves through the bitstring and, given a specified mutation rate (a percentage, usually extremely low like 0.01%), flips the occasional bit to its opposite (0 to 1 or 1 to 0). Thus, the bitstring is slightly altered (mutated) from the original. The bitstring is then converted back into a `botDNA` struct and the values are copied into given monster. This `mutate` function is called when a monster is spawned; a DNA sequence is taken from our collection of bitstrings, mutated, and “cloned” onto the newly-spawned monster.

The other important problem is the survival-of-the-fittest algorithm, which we solved in two parts. First, I wrote a function that is run after a game ends and assigned a score to each of the monsters. For each monster, the score (between 0 and 1) is determined by the following:

$$(\text{seconds_alive}/\text{total_game_time} + \text{hits_on_player}/\text{total_hits_on_player})/2$$

unless this monster killed the human player, in which case a 1.0 is also average into the total. This is simply the average of the fraction of the game this monster survived and the percentage of damage to human player caused by this monster, with a bonus if the monster actually killed the player and thus “won”. This is a very crude determiner of skill and the scores are entirely relative, but it serves our purposes.

A score alone should not determine if a monster’s DNA will survive, so next Professor Levy wrote a *fitness-proportionate selection* algorithm that would pick 5 of the 10 monsters based on their scores. This function was essentially like a weighted roulette wheel, with each monster’s score determining what fraction of the wheel is assigned to him. Then, the “wheel” is “spun” and whichever monster’s name is in the selected slot is chosen to move on. So, the monster with the highest score is *likely* to move on, but even the monster with the lowest score *could* move on. This last bit of randomization is important in order to maintain diversity in the bot population: simply choosing the fittest individuals can result in failure to explore potentially useful evolutionary directions whose fitness is not immediately apparent.

Playing Games (Finally)

With all of the evolution code finally in place, it was time to begin. The evolutionary aspect of a single game would progress as follows:

1. Five bitstrings are copied in from a file `dna.cube`, converted into botDNA structs.
2. Two copies are made of each struct, giving us a total of 10 structs, and then the ordering is scrambled to randomize the spawn order of the various DNA types.
3. As each monster is spawned, he is given one of the DNAs and then his DNA is mutated.
4. When the game ends, each monster is assigned a score based on his performance.
5. Using the scores, 5 monsters are probabilistically selected to move on and their DNA structs are converted to bitstrings and written to the file `dna.cube`.

To begin, I created a `dna.cube` file with all the values set to zero, except for the four distance values, which were set to their maximum value, 64. Since all these monsters were identical to

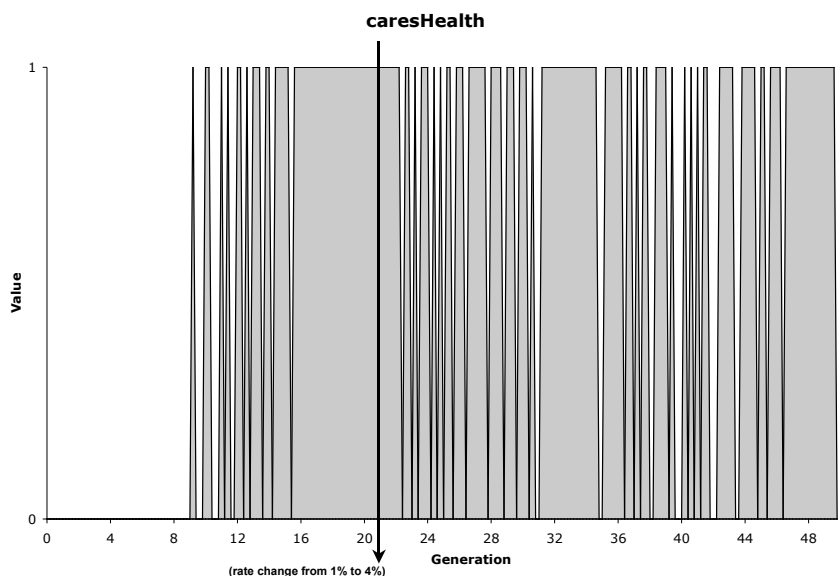
start with, the mutation rate was set to a rather high 1%. Unfortunately for me, the only way to evolve our monsters is to play against them, so I sat down and played the game 21 times.

As you might imagine, the monsters initially played very poorly. A few would get stuck in corners, others would spin in place and not react when I shot at them, and some would shoot but seemingly only at random. After less than 10 games, the monsters had pretty much evolved to where they had been before I made any changes: they would chase me and shoot at me, they would sometimes jump over objects, and they seldom got stuck in corners, but still they were fairly easy to beat. But by game 20, winning was not so easy. The monsters had become reasonably good defensive players: they would retreat when shot at, they would get health when they needed it, and while I could certainly still beat them, it took much more effort to hunt them all down without getting myself killed.

At this point, I decided that the monsters were evolving too slowly (many of the DNA values had never been “flipped”) and so I changed the mutation rate to the appallingly high 4% (imagine if 4% of *your* genes mutated). This would make the evolution somewhat more erratic and increase the likelihood of devolving (loosing good characteristics), but over time it should not really matter. With my new mutation rate, I sat down and played another 29 games. I soon realized that I could no longer count on winning. The monsters had abandoned their more defensive strategy and become very good at hunting me down and killing me. Winning was *hard* and I died often. After just 50 iterations, the monsters had evolved from morons into serious opponents.

Analysis

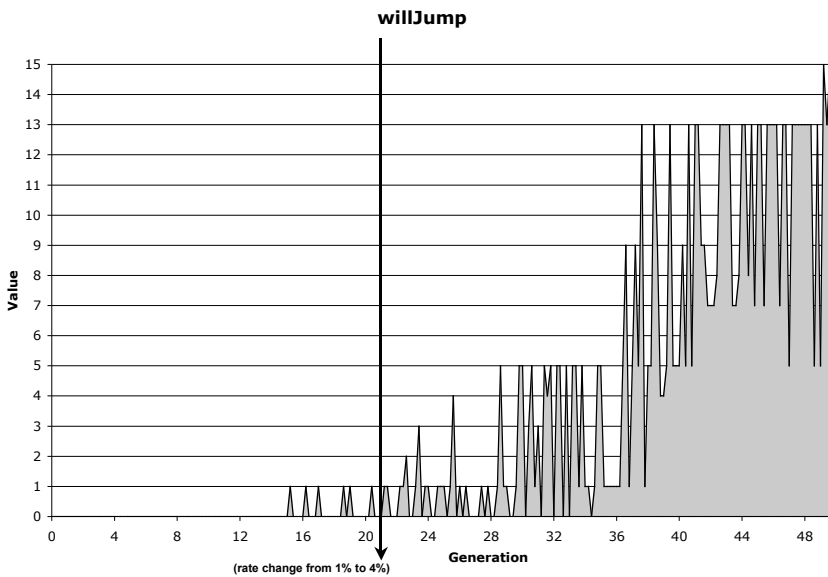
Let’s track the mutations of a few of the more interesting DNA values to see how the monsters evolved. We will start with the fairly straightforward Boolean `caresHealth`, which asks if the monster will care enough about his health to pick up health packs when he needs them. The chart to



the right shows the value of `caresHealth` for each of the five monsters that were selected following each of 50 games, which can be considered “generations.” Since `caresHealth` is a fairly productive behavior, it is not surprising that once some monster gained that ability sometime around game 10, it quickly spread to all of the monsters and remained a popular and common trait from then on. Note that once the mutation rate was increased to 4%, the mutations became far more frequent and erratic, but even so a true `caresHealth` value remains dominant.

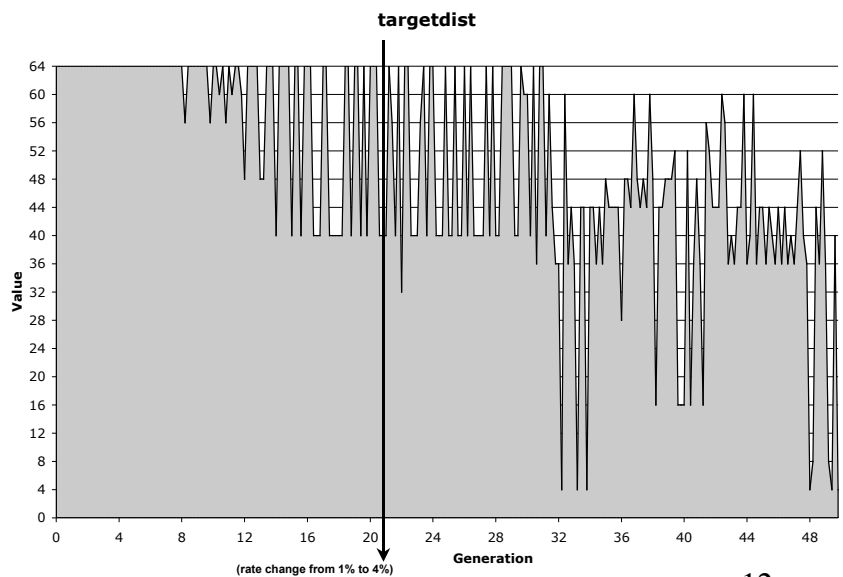
Now let us look at the case we considered earlier: `willJump`. If a monster is blocked by

some obstacle and can jump over, how often should he? Sometime around game 30 they seem to settle on 5, or a probability of 1/3, which is what I originally used when I first wrote the function. But just a few games later, the monsters increase the average to around 1/2 and by game 45 or so they seem all seem to agree on jumping 13 times for every 15 possible jumps. This is much higher



than I would have expected, indicating that jumping over small obstacles, rather than moving around them, may be a better idea than I thought. This is a good illustration of the benefits of evolutionary algorithms: *the monsters had evolved a behavior that might be better than what I would have hard-coded.*

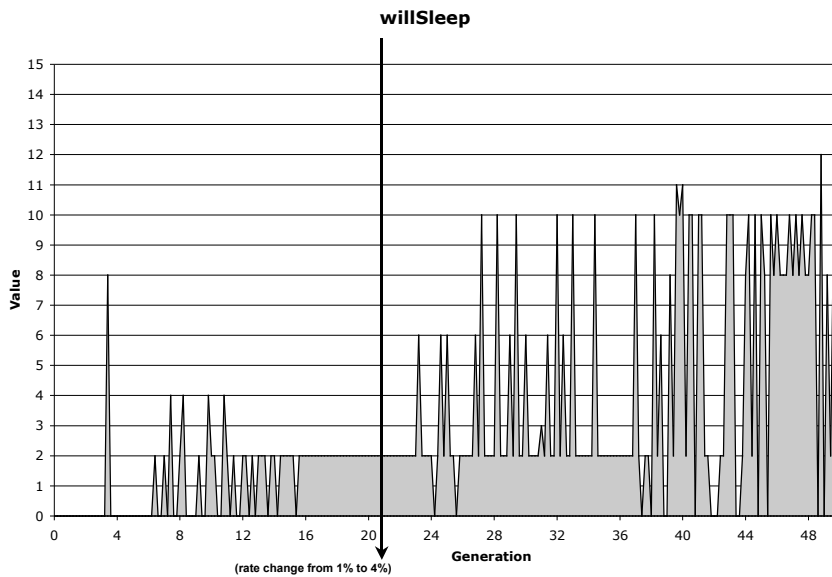
Two more interesting cases where the evolution produced results that one may not have expected are `targetdist` and `willsleep`. The former, at the right, is the farthest distance, in “feet”, that a monster will target and fire at an opponent. The default distance, in both the



original game and our DNA value, was 64. One might think that firing from long range, or at least having the *option* of firing from long range, would be advantageous. However, the monsters decreased this distance steadily until the average was around 40, with some monsters decreasing their `targetdist` as low as 16 or even 4! It is possible, and quite likely for the extremely low values, that playing in some of the cramped indoor levels may have allowed a low `targetdist` to have no real effect on the monsters' performance. However, I think the general trend toward values less than 64 does imply that it is more beneficial to wait until you “see the

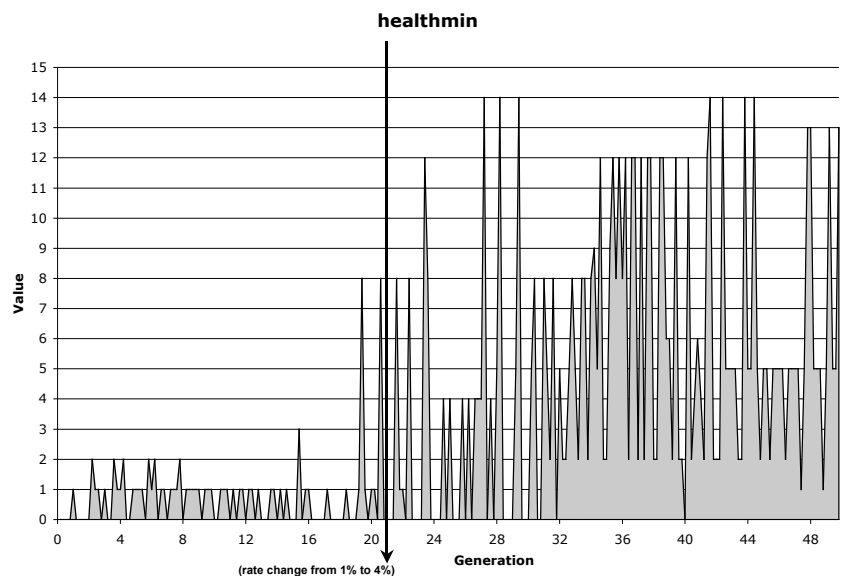
whites of their eyes” rather than firing from far away.

The other case, `willSleep`, is the probability that a monster who cannot find the player will stop searching and instead wander aimlessly, possibly picking up items along the way. Since this behavior is not terribly productive, one would think it would be easier to earn a high score by actively



hunting the human player. As we can see above, the values jump all over the place, meaning the survival-of-the-fittest algorithm had not really determined whether this behavior is good or not, though the upward-moving trend of the values does suggest that it is wise to *occasionally* wander.

There are several other evolved values that can be considered inconclusive, most notably `healthmin`, `armourmin` and the various distances. If we look at `healthmin` at the right, which is the minimum fraction of the monster's original health that he will tolerate, we can see that while there is an



upward trend, the values are generally scattered. In fact, the slightly increasing values alone are not terribly interesting, because a `healthmin` of 0 means that the monster does not care about his poor health until he is dead. Because low values are all worthless and there are concentrations of values around both 12 (a probability of 4/5) and 5 (a probability of 1/3), this chart essentially tells us nothing. I believe at least part of the reason for this is the small population size and the redundancy of having interdependent `caresHealth` and `healthmin` values in the DNA.

Conclusions

Evolutionary algorithms enabled our monsters to move from incompetent to lethal in 50 generations, which is a stunning improvement and proof that evolutionary algorithms can be effectively and rather painlessly adapted to 3D first-person shooter games. This system eliminates the need for high-level planning on the part of the designer because the AI figures out the best behaviors by itself. Furthermore, our simple evolutionary system was extremely fun to play against because the monsters grew more and more challenging over time and essentially learned from their mistakes as a real human opponent would. Instead of increasing the difficulty in the cheap way that most FPS do, either by making the monsters stronger or more numerous, our evolutionary algorithm enabled them to legitimately improve their abilities. Because they are computer opponents, they could improve faster than I could and that is why they could move from morons to masters in relatively few rounds of play.

The other lesson we learned from this project is a more general truth: open source is better for this type of project. This may seem obvious in hindsight, but if an independent study depends on learning and expanding upon what people have already done, then a complex closed system like the Unreal Engine is not the way to go. Cube has fewer features than UT2K3, but given the time we had, that was irrelevant. The work we did may be *technically* less impressive than the existing AI in UT2K3, but the *concepts* we worked on are novel and presumably could be, with varying degrees of difficulty, implemented in *any* FPS (even Unreal) – Cube is simply the best environment for us to demonstrate that.

What to Do Next...

There are many aspects of our implementation that could be improved upon, but I think the biggest one is scale. With a population of just ten, and five of them surviving to the next game, it is far too easy for a single behavior set to dominate the population. Diversity is unlikely in a society with so much “inbreeding,” and even a ridiculously high mutation rate did not disrupt the homogeneity for long. The next major improvement would be to design a gigantic map so that as many as 100 monsters could play at one time. A larger population would allow for more interesting mutations to survive long enough to flourish and would diminish the effect of luck. This would allow different types of behaviors to evolve *simultaneously*, so that maybe 15 of the monsters might play defensively while 20 would play aggressively. That would be quite an improvement over the present, where those two behaviors evolved *sequentially*.

The next step would be to add more capabilities to the monsters, such as better navigation (jump across pits, don’t charge down dead ends, etc.) or the ability to pick up and use ammo in addition to health and armor. I also believe that some of the existing functions could be improved upon, namely the survival-of-the-fittest algorithm and the implementation of integers in the bitstring. It would also be interesting if the transitioning from state-to-state were more under the control of the DNA rather than the way it is now, where the transitions have hard-coded delays and they are triggered by predetermined criteria specified by the current state. We should jump at any opportunity to take power away from the programmer and give it to the evolutionary system.

The long-term goal would be to either add some high-level game algorithms in Cube, such as Unreal Tournament’s team AI or pathfinding, or else implement a similar evolutionary algorithm in a more complex FPS. Our project has shown that an evolutionary algorithm can greatly enhance a first-person shooter, but the real test is whether such a system could push the AI in these games beyond what we have today. If the quality and difficulty of these games can be dramatically improved by a simple off-the-shelf genetic algorithm, then that would be a real achievement.

References

Mitchell, Melanie M. (1996) *An Introduction to Genetic Algorithms*. Cambridge, Massachusetts: MIT Press.