



# Evolution of Formidable Player AI in Tactical 3D Games

Stanislav Rost

[stanrost@lcs.mit.edu](mailto:stanrost@lcs.mit.edu)

MIT Laboratory for  
Computer Science  
Cambridge, Massachusetts

Food for one is enough for two,  
Work for two is enough for one

*–Ancient arabic proverb*

## Abstract

### 1 Introduction

Since the dawn of computer entertainment, artificial intelligence played a significant role in the amusement and replayability value of computer games. Designers of the early games realized that humans enjoy competing against opponents with personalities as much as, if not more than, overcoming the challenges offered by the game's environments and their physics. As more and more processing power becomes available to the end-users, game players expect increased sophistication and challenge from the behavior of the computer-driven adversaries.

The question of designing the intelligence of such formidable opponents is then of central importance to the computer entertainment industry. Limited processing power available to game designers and the science's lack of complete understanding of the human mind hamper the game designer's ability to create truly intelligent game entities. However, games have historically approximated intelligent creature behaviors using more traditional computing paradigms.

Developers of computer opponent AI have employed a variety of methods to achieve the impression of intelligence. The simplest and the most unintelligent form of behavior employed by game entities is repetitive. For instance, in the Parker Brothers classic "Montezuma's Revenge"<sup>1</sup> for Atari 2600, skulls roll back and forth along the dungeon floors until hitting a wall of the edge of the platform. In "Space Invaders," the echelon of aliens moves from one edge to the screen to another, while lowering toward the player's ship. Although such behaviors alone are simple and repetitive, a complex system of entities exhibiting such simple behaviors may require a great deal of thought and timing precision in order for human players to succeed.

Another classic approach makes use of random number generators. The perception of intelligence of a creature behaving randomly arises from the notion of a good opponent not telegraphing his true intentions and therefore being difficult to predict. However, unless properly designed, random behavior may result in choices that are not necessarily favorable for the computer-controlled entity<sup>2</sup>. One example of an application of randomness can be seen in "Space Invaders," where every alien has a random probability of firing at any time instant. Often, game designers use random behavior to add temporary imperfections to other-

wise perfect winning algorithms.

If the behavior leading to the win is well-known and can be achieved by a deterministic algorithm, games may explicitly use the algorithm to determine the behavior of its entities [3]. For instance, "Pacman" reused the same version of a pathfinding algorithm, which planned the next step in the path of an enemy ghost necessary to reach a particular coordinate in the maze. This algorithm was used to implement the straightforward predator behavior of the ghost named Blinky, which focused on reaching the player's current position. It was also used to define the predictive predator behavior of Pinky, which focused on reaching the next corner where Pacman is headed. The rest of the ghosts moved in random fashion, changing their direction at the intersections.

And finally, recent games augmented low-level behaviors with high-level algorithms which represent the high-level state of the creature. Examples of such states are "hungry," "angry," "fearful." Depending on the high-level state, a creature may choose to engage in different sets of low-level behaviors, such as running away from the attackers when hurt. Usually, higher-level behaviors are represented by carefully designed finite state machines, whose transitions are triggered by perceptions, player interactions or randomness.

The aforementioned approaches to game creature intelligence, discussed in the context of the classic games, are widely used in modern games. However, the opponents devised with the use of such algorithms still fail to introduce exciting competition to gameplay. Application of such algorithms leads to patterns in enemy actions, and their complete lack of originality. Shortcomings of the traditional approaches evidently drive the industry toward creating multiplayer games to facilitate human-to-human competition.

However, even in multiplayer games the importance of computer-driven AI is paramount. For example, in most multiplayer tactical 3D games, meaningful gameplay can only be achieved if the competing teams have a sufficiently large total number of participants. For instance, a "Magnificent Seven"-style game may provide a setting of a western town. However, contrary to the proverbial challenge, the town may indeed be more than large enough for a small number of human players, which will spend many tedious minutes trying to locate each other before facing off. Additionally, if the participants of the multiplayer match must each fulfill a certain role, teams lacking players in key roles may not be able to achieve balanced gameplay.

To alleviate the shortcomings of small teams, some game designers supplement the teams with AI-driven "bots" to fill the roles of the missing players. We define "bots" as algorithms determining the entity behavior as well as the entity controlled

<sup>1</sup>Incidentally, I owe my interest in computer science to this game. It is the very first piece of software on the very first computer that I ever saw.

<sup>2</sup>"In the darkness, these beasts were capable of anything...even administering first aid." *The 21<sup>st</sup> Century Fairy Tale*

by such algorithms. In order to compete against and along with human players competently, bots must be remarkable well-designed. They must succeed at the tasks of locomotion in complex environments, tactical combat and teamplay. Usually, locomotion is implemented by marking the environments with a large quantity of meta-information, such as “tracks” or “way-points” which the bots may follow. Attacking is often achieved by exposing player locations to the bot logic and applying randomness to making the aim less precise. An example defensive action such as evasion may be achieved by the bot moving toward a position such that the line-of-sight vector from the enemy is obstructed by an obstacle. Bots may achieve teamplay, for example, by being aware of the locations of firefights, moving toward those locations and joining the action.

Although many game designers attempt to create bots which succeed at gameplay, none of existing game bots compare to human players in ingenuity, planning, teamplay, and mastery of weapons and tools. Also, even if exposing internal information such as coordinates of other players and providing the bots with pre-computed paths simplifies the tasks of bot designers, it prevents the bots from discovering their own, unique approaches to gameplay.

More precisely, in order to appear intelligent, a computer-controlled entity must exhibit variation and creativity in its approaches to the following tasks:

**Perception** Discerning dangers and favorable objects and locations, as well as discovering paths away or to such.

**Locomotion** Utilizing the physics of the environment to reach locations necessary for fulfilling the goal, escaping and evading dangerous scenarios.

**Tools** Discovery, mastery and application of various tools and weapons to achieve goals.

**Collaboration** Enlisting and providing aid to the community of friendly entities.

**Opponent Prediction** Ability to anticipate the opponent actions and accounting for them before the opponent performs them.

**Influencing the Opponent** Undertaking actions which force the opponent into non-advantageous actions.

In this project, we argue that, once the bots are abstracted from internal game state and instead perceive the world through sensors as any other embodied creature, the evolutionary process may be applied to discover innovative approaches to achieving game intelligence described above. Our bots do not have access to pre-computed meta-information or player coordinates, but instead use the senses of sight, sound, touch and pain to navigate

around the world and participate in the gameplay. We expect unique behaviors to arise due to optimization of the bot strategies via repeated mutation, cross-breeding and selection.

To test the validity of our conjecture, we develop bots for Counter-Strike, a popular 3D tactical game. To that end, we propose and implement a bot architecture using a hybrid computational model, compilable into native x86 code for greater scalability. After seeding the evolutionary system with an ancestor, we attempt to develop new bot behaviors using an evolutionary approach. We then evaluate the performance of the evolutionary system, and analyze the evolved behaviors and the impact of various factors on the evolution of our AI entities.

## 1.1 About Counter-Strike

Counter-Strike is a multiplayer game based on the Half-Life game engine, developed by Valve Software and published by Sierra Studios. Winner of 8 “multiplayer game of the year” awards, it is cited as the most popular online game, with a record of 30,000 online players competing in multiplayer matches simultaneously [2]. It features realistic physics and complex 3D environments.

The premise of the game is straightforward. The players may join one of the two teams: terrorists and counter-terrorists<sup>3</sup>. Each team has access to a slightly different set of weapons and equipment. Game environments (“maps”) have varying objectives for both sides. For instance, hostage rescue maps feature a number of hostages which counter-terrorist must first rescue from a location secured by terrorists, and then lead to a rescue point. The terrorists’ task is to prevent the rescue of the hostages while preserving their lives. Another game type is conducted on a “detonation” map, where terrorist must plant explosives at a certain site protected by the counter-terrorists, and ensure that the bomb is not defused prior to explosion. Counter-terrorists, in turn, must secure all possible bomb sites or defuse the bomb before it explodes. A match is won by a team when the map objective has been accomplished or if all of the members of the opposing team have been eliminated.

A match typically lasts approximately 5 minutes, and once a player has been eliminated in a match he is removed from the battlefield to observe the remainder of the match as “ghosts”. Upon the next match, the players “respawn” and join the game completely revitalized.

---

<sup>3</sup>This game was developed prior to the tragic events of September 11th, 2001. The creators of this game insist that games and reality exist in different universes, one significant and another not.

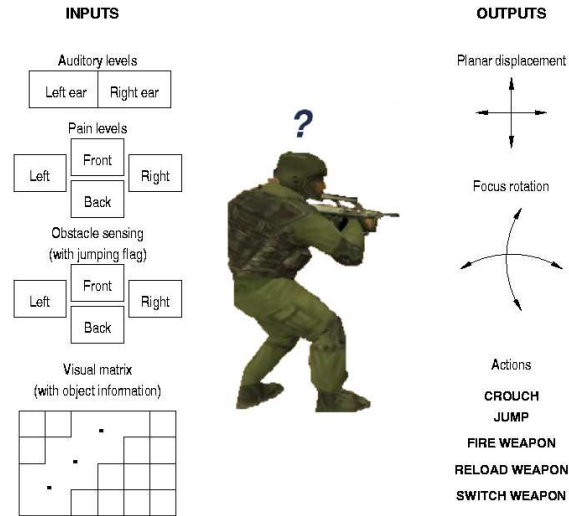


Figure 1: Inputs and outputs of the bot

## 2 Architecture

### 2.1 Integration with Counter-Strike

Integration with Counter-Strike is achieved through Botman's HPB Bot interface, a bridge between the client code and the game engine. In the Half-Life architecture, the client code is responsible for accepting user input, and presentation of the world to the user. The game engine is, in turn, responsible for the management of the global world state, processing object and entity interactions and physics. Normally, client code is loaded through a shared object/dynamic linking library, and is called back by the game engine whenever the game engine is ready to process client behavior. Such pluggable architecture allows Half-Life designers to replace the local client module with another forwarding messages originating from remote networked clients. Botman's HPB Bot interface interposes itself between a default client DLL and the game engine and takes of the function of many callback functions of the client to achieve custom bot behavior.

Our bot evolution system consists of three modules: the breeder, the bot compiler, and the bot handler. The breeder exists outside of the Half-Life/Counter-Strike game engine, and performs the tasks of seeding, evaluation, selection, mutation and cross-breeding. It invokes the bot compiler to create dynamically loadable libraries with bot code. The breeder also invokes the game engine to simulate game matches. Behaviors of characters during the match are determined by the bot handler, compiled into

the extensively augmented HPB Bot framework. The handler is responsible for preparing the input to the bot, running the bot code and applying output behaviors to the creature in the game. More precisely, our evolutionary bot system executes the following cycle:

- The breeder writes descriptions of bots to the bot behavior files, uses the bot compilers to translate bot behavior sources into dynamically loadable libraries.
- The breeder creates a configuration file describing which bots will be used in the simulation by any given copy of the engine.
- The breeder runs multiple (currently four) Counter-Strike game engines. Each engine populates the world with a subset of bots, whose behaviors are determined by bot AI loaded in the dynamic libraries. The bot handler repeatedly delivers perceptions to bots, and, after the AI is done processing, applies their actions to entities. When the match is over, the statistics necessary for determining the evolutionary score are written to a report file.
- The breeder parses the report file, and determines the scores.
- The breeder mutates and cross-breeds bot AI according to the policies described in the Chapter 2.7.

In designing our evolutionary system, we had to accommodate for the idiosyncrasies and limitations of the game engine. The

| Instruction                                                                                                                                                                                                 | Description                                                                                                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;reg&gt; = &lt;reg value&gt; &lt;op&gt; &lt;reg value&gt;;</code>                                                                                                                                  | Allows arithmetic of constants and registers to be stored in a target register. Permitted operators include <code>+</code> , <code>-</code> , <code>/</code> , <code>*</code> , <code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;=</code> . |
| <code>if (&lt;reg&gt;)</code>                                                                                                                                                                               | Execute the next instruction if <code>reg</code> is not zero.                                                                                                                                                                                                                                         |
| <code>transition(&lt;module#&gt;;</code><br><code>module(&lt;module#&gt;;</code>                                                                                                                            | Goto the specified module<br>Declare the beginning of the code segment of the module                                                                                                                                                                                                                  |
| <code>jump();</code><br><code>crouch();</code><br><code>fire();</code><br><code>altfire();</code><br><code>reload();</code><br><code>switchw();</code><br><code>bomb();</code>                              | Make the player jump<br>Make the player crouch<br>Make the player fire the current weapon<br>Use the “alternative fire” button on the current weapon<br>Reload weapon<br>Switch weapon<br>Plant the explosive or defuse it depending on the team                                                      |
| <code>&lt;reg&gt; = sum(&lt;1d or 2d array of registers&gt;;</code><br><code>&lt;reg&gt; = avg(&lt;1d or 2d array of registers&gt;;</code><br><code>&lt;reg&gt; = max(&lt;1d array of registers&gt;;</code> | Obtain the sum of the register values<br>Obtain the average value of the registers<br>Obtain the index of the maximum value of the registers                                                                                                                                                          |
| <code>nop();</code><br><code>halt();</code>                                                                                                                                                                 | Do nothing<br>Finish the execution of the FMM, commit the output register values                                                                                                                                                                                                                      |

Figure 2: Finite Modular Machine instruction set.

following are the limitations and our approaches to accounting for them:

**Maximum number of players:** The game allows for a maximum of 16 players on each team, 32 total. As a result, in our evolution cycle, individuals compete in groups of 32, randomly selected disjoint subsets of the pool of 128 bots.

**Realistic timing:** The timing for player locomotion is realistically modeled after real-life humans. We originally sought a method for increasing simulation speed by increasing the internal game clock speed. This approach has proved infeasible. This constraint leads to the next limitation.

**Round time:** Each match requires at least 5 real-time minutes. This severely limits our evolution rates and our selection pool size.

**Random recoil modeling:** Each weapon features an inaccuracy which is directly proportional to its recoil. The inaccuracy results in the bullet vectors being randomly rotationally offset from the direct line of aim. The longer is the burst of the weapon, the more inaccuracy is introduced in addition to the baseline inaccuracy of the weapon. The evolved bot AI is expected to account for the inaccuracy of the weapons.

Due to the timing limitations, the rate of evolution in our system is limited by 32 individuals every 5 minutes. To mitigate this limitation, we utilize a number of game engines concurrently and achieve a rate of 128 bots (1 generation) per 5 minute.

Therefore, our overall evolution rate is limited to 288 generations per day (barring the time required for mutation, bot compilation and game engine initialization).

In the following sections, we describe our “black-box” approach to specifying bot behavior, define the inputs and the outputs for the bot, describe the computational model for bot logic and its conventions, and talk about the three components of our architecture (the breeder, the compiler and the bot handler).

## 2.2 Formalizing the Evolutionary Game Character AI

We will begin by treating the AI system as a mutable function which maps the character’s perceptions of the environment and other players to actions to be undertaken in reaction. The function, invoked periodically every  $\delta$  time units to determine the course of action within the next  $\delta$ , may choose to preserve some of the prior inputs, outputs or intermediate decisions to base its future decisions on the events in the past, therefore simulating a memory.

In such system, the inputs are presented to the function in read-only registers, a bank of persistent registers is allocated to the function to retain state across invocations, and a number of actions and output registers which directly affect the actions of the bot player.

The computation which takes place inside the “black box” comprises the evolvable component of the bot. The bot interpreter

carries out all of the computation by executing the instructions of the block box.

## 2.3 Inputs and Outputs

The Counter-Strike bot which we propose will need to base its decisions on the inputs and express its decisions using the outputs depicted on the Figure 1. First, the bot is provided with an  $M \cdot N$  matrix of distances to objects in its visual field, effectively obtained through raycasting from the viewpoint of the bot model and labeling of objects encountered by the rays. Each “pixel” in its visual matrix will also be labeled with extended information. For instance, a pixel containing a part of an enemy player will be marked with an “enemy” flag. This approach to the presentation of the visual information is similar to that of the human brain, where semantic information associated with objects is visual view is retrieved from areas associated with language processing. Analogously, a pixel containing a door will be marked with a “door” flag. A pixel aligned in direction with an opponent’s base will be colored with an “enemy base” flag. This is similar to the behaviors of certain birds, which obtain their directional cues from the sun and the stars.

Directional obstacle flags are provided to the bot in those situations when the bot is in direct contact with walls and other obstacles, and if movement in that direction is impeded. However, obstacle information is also augmented with a “can-jump” flag if the obstacle is of small height and the bot can jump on top of it.

Pain levels are provided by the Half-Life engine such that it is possible to obtain the approximate direction from which the attack has originated.

Auditory levels provide useful hints to the location of enemies (through footsteps and weapon fire sounds) and places where firefights occur.

The output of the AI function is expressed in terms of position updates, orientation updates, and actions. Position updates express how far the bot must move in the horizontal plane within the next frame. Orientation updates are expressed in angular differences with respect to the current orientation and change the aim/focus of the bot. Action outputs follow a different model: a bot may choose to execute or not to execute any action at a given moment.

In addition to the inputs shown on the diagram, the bots have access to a register which is furnished with numbers produced by a pseudo-random generator. Upon an access to the random register, its value changes to the next number from the pseudo-random sequence.

## 2.4 Introducing Finite Modular Machines

Upon considering various models used for evolution introduced in class (such as finite state machines of [4], Tom Ray’s Tierra system [6], and artificial neural networks [5]), I decided none of them possess the flexibility necessary for this project.

In particular, finite state machines are poorly suited for processing large quantities of input data. Additionally, finite state machines lack the expressive power to work with input values ranging over a large discrete universe, essentially constraining the designer to the use of the equality/match operator.

Ray-style evolution systems employ mechanisms of transition among sections of code which are easily disrupted by mutations and cross-overs. Admittedly, this weakness can be repaired by introducing code transitions relative to the current instruction counter and providing an alternative method for labeling various positions in the code. However, even after such improvements have been applied, the behavioral code of any entity of considerable complexity is difficult to understand by human designers. Jump-ridden code created by evolutionary systems embodies the worst chaos-spawn spaghetti-style programming the author has ever analyzed.

The use of artificial neural networks in evolutionary systems also poses a number of problems. First, it is unclear how to connect the inputs to the networks correctly. Secondly, it is not obvious what the networks are learning or how to emphasize importance of some features over others. Thirdly, reverse-engineering a neural network to understand its logic is highly non-trivial.

For this project, I decided to employ a novel computational paradigm which combines the positive features of all of the above approaches and eliminate their shortcomings. Our proposed approach, which we label Finite Modular Machines, consist of transitory modules of code similar to the nodes of the finite state machines. However, each module contains arbitrary code able to read from any of the input registers, access any of the temporary/output registers, and perform arithmetic and logic operations. The module’s code is, however, constrained in that it can only transfer control to the beginning of the code segment of another module, either conditionally or unconditionally. No local jumps are permitted within the code of a particular module.

In essence, FMMs combine the modularity and module-to-module transitions of FSMs, random memory access and the computational power of Ray-style machines, and the freedom to combine multiple inputs similar to neural networks.

In addition to input and output registers, FMMs may also be provided with a bank of temporary memory to store variables and values for the duration of the execution of the FMM.

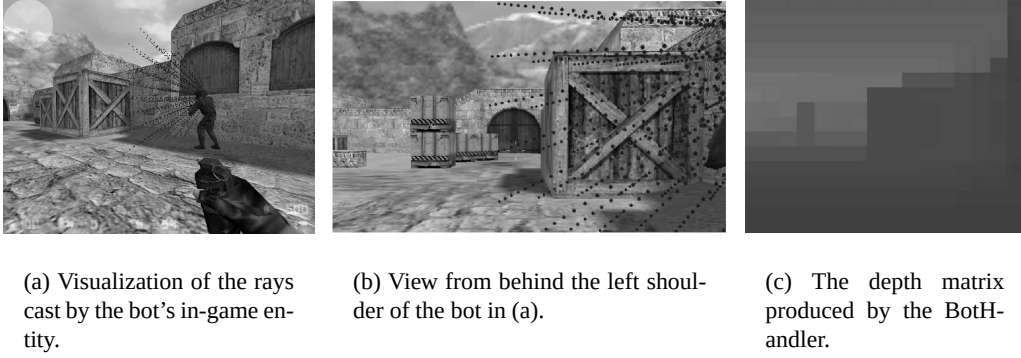


Figure 3: Translation of the 3D world into the distance component of the visual matrix.

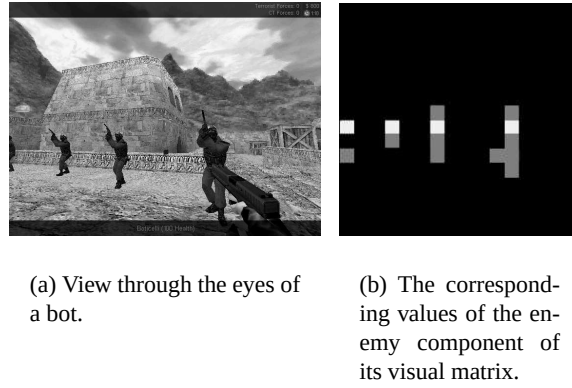


Figure 4: Translation of the enemy perception onto the bot's retina.

## 2.5 Instruction Set and Register Conventions

The instruction set supports basic logic, arithmetic, conditionals, transitions to other modules. The architecture supports a maximum of 256 registers and 256 modules.

It is important to note that bot code exists in our systems according to three different representations. The compact representation is used internally by the breeder to make the task of mutation easier. Following that representation, a single instruction can (somewhat wastefully) be represented in 16 bytes. The bot compiler translates from compact representation to intermediate, C-like representation. Such representation improves the readability of the bot code by humans (similar to use of assem-

bler to clarify machine code). Figure 2 enumerates and describes the instructions in the intermediate notation. Finally, the intermediate code is compiled into native x86 code to improve the performance of our system.

Registers store signed floating point values, and their names follow a simple convention. The input register names begin with *i*, the output register names begin with *o*, and the temporary registers begin with *t*. The registers must be further specified in a notation similar to arrays in C, as follows:

- iRAND** Input a value from a random register.
- iAUDI[L|R]** Input audio in the specified "ear."
- iPAIN[F|B|L|R]** Pain levels in the specified direction.
- iOBST[F|B|L|R][IS|JUMP]** Obstacle sensing in four direc-

tions, flags for whether “is” there an obstacle and whether it can be jumped over.

**iVISU[Y][X][DIST|NME|GOAL|DOOR]** Visual matrix at coordinates  $X, Y$ , the distance or the enemy flag or the goal flag. Either  $X$  or  $Y$  must be a value or a scalar register.

**iVISU\_Y[X][DIST|NME|GOAL|DOOR]** The whole column of at the  $X$  coordinate.

**iVISU\_X[Y][DIST|NME|GOAL|DOOR]** The whole row of the  $Y$  coordinate.

**oPLAN[X|Y]** Delta planar movement.

**oROTA[X|Y]** Delta rotational movement.

**t[I]** Access  $I$ ’th element of the temporary array.

## 2.6 Bot Handler

Bot handler is the part of the bot system which facilitates the bot AI functions to participate in Counter-Strike matches. Whenever the game engine polls the client module for user action, the bot handler provides such response after invoking the bot code.

During initialization of the game, the bot handler loads the dynamic libraries named “EvoBot1”..”EvoBot32” from the library directory, as well as some extended information from the configuration file created by the breeder.

First, the bot handler prepares the bot’s inputs with information as perceived by the character controlled by the bot. That includes raycasting of “probe” vectors from the bot’s current position, computation of their intersections and labeling of the objects which have been intersected. That information is recast into the visual input registers of the bot. The cone of the field of vision of the bot is restricted to 90 deg, in conformance with the default visual field rendered on the screen of the human opponents.

A visualization of the initial segments of rays cast from the visual vantage point of the bot’s in-game entity is shown on the Figure 3(a) with dotted lines. The view from slightly to the left of the bot’s left shoulder is shown on the Figure 3(b). Values of the distance component of the bot’s 16-by-16 visual matrix are rendered in grayscale on the Figure 3(c)<sup>4</sup>. The crate to the right of the bot on the Figure 3(a) can be clearly seen as being closer. The gradient of the floor distance and the “skybox” (which is *significantly* further away) is also evident. The structures on the left are actually two boxes stacked on one another, viewable in the behind-the-shoulder view.

<sup>4</sup>The contrast of this image has been reduced to improve visibility.

In addition to the distance component of the visual matrix, the same rays activate the enemy and the goal components of the visual matrix. However, this information is far more crucial to desired bot behavior than the distance. The probability that coarse-grained rays traced from the bot entity’s vantage point will miss the enemies in the field of vision of the bot increases with the distance to the enemy. This is due to the fact that the ray must precisely intersect a bodily part in order for the enemy component of the visual matrix to activate. We aid the perception of enemies and goals by back-tracing from the centers of mass of all the visible enemies and goal landmarks to the “eye” of the bot, and marking those cells which correspond to the collision of the back-tracing ray with the bot’s “retina” (the visual matrix).

A sample view and the corresponding visualization of the enemy component of the visual matrix are shown on the Figure 4. Centers of mass projected by back-tracing from visible enemies are shown with brighter colors, causing twice the level of activation in the enemy component of the visual matrix cells than other parts of the enemies discovered by field-of-vision tracing rays.

Obtaining the sound levels for the auditory input registers is non-trivial. It is the responsibility of game clients to rendering the sound effects. Therefore, there is no proper method for obtaining the sound levels to supply to bots, which reside and execute within the game engine. Instead, the sound levels are estimated by emitting a simulated signal from the locations of bots. The signal fades off with the square of the distance. The initial signal strength is determined by the speed of the bot, where tip-toeing produces no sound, walking produces medium sound levels, running is significantly louder and discharging a weapon is an order of magnitude louder.

Collision detection is performed by tracing twenty-four short rays from the bot body, twelve at the waist level and twelve at the knee level. Collisions of waist-level rays with external objects determine which obstacle registers get toggled. Collisions of the knee-level rays trigger the values of “can-jump” flags, indicating that the bot can jump over the obstacle in that direction.

Pain data is obtained by intercepting the Half-Life engine’s “pain” messages to the client. Using the pain data, we obtain the inflictor origin, and convert the angle of the vector from the bot to the inflictor into the input register which receives the pain value.

After depositing the sensory information into the bot’s input registers, the bot handler directs it bot runner thread to execute the *think()* method of containing the bot code. The execution time of *think()* is bound by a  $T_{timeslice}$ , reasonably set at 1 milliseconds. The code is structured so that the first instruction of every module acts as a cancellation point, and the

check for whether the time limit is exceeded is performed at that time. High-resolution timestamps for cancellation are procured through the high-performance mechanism of reading the Pentium *rdtsc* register.

After the bot code has finished executing, the bot handler transfers the outputs and applies them to the game entity. The angles and displacements are bound by upper limits so the bot does not possess superhuman powers such as the ability to turn 180 degrees in  $\frac{1}{60}$ th of a second.

In addition to executing bot code, the bot handler also observes the information passed to it by the game engine and records specific subscores for every bot in the game. For instance, if any of the cells in the visual field is marked with the “enemy” flag, the current timeslice is counted toward  $f_{track}$ , the total time spent observing the enemy. The game engine’s pain messages allow to deduce the identity of the player inflicting the damage, so the amount of damage is deducted from the target’s  $f_{selfdamage}$  and deposited to the attackers’  $f_{damage}$ . To tally the distance walked toward the map goal, the Euclidian difference in the distance to the goal at the current position and that at the previous position of the bot is added to the score. Total travel distance component of the bot score is obtained by measuring the Euclidian distance from the position of the bot in the previous frame.

At the end of each match, subscores and various other statistics (such as profiling information) is recorded to the match report file, to be parsed by the breeder.

## 2.7 The Breeder

The evolutionary system houses a population of 128 specimen, initialized with an ancestor program outlined in the next section. The 128 specimen are subdivided into 64 “Counter-Terrorists” and 64 “Terrorists,” and each team evolves separately. Each team is randomly subdivided into disjoint sets of 16, inserted into the game and the two teams of “Counter-Terrorists” and “Terrorists” compete for a single match against each other. All matchings of different teams compete against each other in a match prior to breeding the next evolution.

After the individuals have been scored in a match, the top 50 percent of them in every team are retained. Then, some of the top performers are cross-bred randomly, and their children occupy the slots formerly allocated to the under-performing 50% of the team. Subsequently, all genes are mutated with probability inversely proportional to their score normalized by the sum of all scores.

Our system implements a generic set of mutation and crossover policies. All the instructions in all the modules are equally likely to mutate, however, extra care is taken to ensure that mutations

result in valid instructions. Additionally, the freedom of mutation is constrained so that only writeable registers are written to in the mutated bot code. In half the cases, just the instruction’s operands are mutated, whereas alternatively the instruction is mutated as well. The mutations of scalar arguments are restricted in their distance from the original number. Our system features multipoint crossovers of random arity (in up to 5 places) which occur at random points in the program. Additionally, the breeder may randomly preserve existing module boundaries, split a module, or join two modules together.

The bots are allowed a maximum of 256 modules, each limited to 1024 instructions. Upon initialization, the system loads all population slots with the default ancestor, and fills all the unused modules with the NOP instruction.

## 2.8 The Ancestor and its Subsystems

The ancestor gene is necessary to avoid the lengthy process of start-up evolution required to breed bots with rudimentary skills of locomotion, obstacle avoidance, and basic combat. All of these functions can be provided by the ancestor, and then further improved or changed by the evolutionary selection.

In our scheme, the basic skills are achieved by creating a pipeline of specialized subsystems. The pipeline is organized so that subsystems invoked later in the pipeline may choose to override the output decisions made by earlier stages in the pipeline. The following is a list of the subsystems in the order in which they execute in the pipeline:

**Locomotion** To get to the goalmark, sum all of the vertical goal bits of the visual matrix. Find index of maximum, if found, rotate the heading by a scaled difference between the middle of the screen and the maximum. Step forward. If goalmark not found, walk randomly. <sup>5</sup>.

**Auditory enemy/firefight pursuit** Slightly adjust heading according to the difference in the auditory levels in the left and right ear.

**Damage avoidance** Move sideways (“strafe”) in a random direction if pain is perceived. Crouch sometimes.

**Enemy pursuit/engagement** Integrate over the “enemy” bits of the visual matrix. Adjust aim toward the maximum value. Turn toward pain source. If the middle of the visual matrix has a non-zero “enemy” value, fire.

**Obstacle avoidance/jumping** Evaluate if the obstacle flag is set in the direction in which we are moving. If so, and if

---

<sup>5</sup>Note that the target, once reached by the bot, switches between various landmarks of a particular map

the obstacle can be jumped on, jump. If not, “mirror” the direction of the movement.

The ancestor’s pipeline has been implemented in 50 modules and just over 450 lines of C-like intermediate FMM code. The complete source code for the ancestor bot may be found in the Appendix A.

## 2.9 Bot Evaluation

In designing the bot evaluation routine, I wanted to ensure that bots would have incentives to develop progressively more and more complex behavior. Thus, I have devised a scheme which contributes polynomially higher rewards to bots exhibiting progressively more difficult behaviors. For instance, if walking for the distance of  $X$  contributes to the bot’s score at the rate of  $N$  points per second, moving the same distance toward the goal will contribute  $N^2$  points per second.

Bots are evaluated on the following criteria (where + denotes desirable behaviors and - denotes undesirable behaviors):

- + **Enemy observation**  $f_{track}$  Amount of time spent observing the enemy in its viewpoint.
- + **Enemy damage**  $f_{damage}$  Amount of damage dealt to the enemy.
- + **Enemy kills**  $f_{kills}$  Number of enemies killed in the match.
- + **Distance traveled**  $f_{travel}$
- + **Distance traveled toward goals**  $f_{goal}$
- + **Team score**  $f_{team}$  How well the entire team did.
- **Damage to self**  $f_{selfdamage}$  How much damage the bot sustained.
- **Death**  $f_{death}$  A one-time penalty for dying.
- **Obstacle collisions**  $f_{collisions}$  Number of times the bot walked into an obstacle.

The team score is calculated at the end of the match, and is factored into all the scores of bots on the same team. While this does not skew the ranking of bots in a particular bot batch selected for a given match, it does allow batches of bots which worked well together to outrank other bots at the end of each simulation cycle.

The final evaluation function combines the aforementioned metrics by a weighted polynomial function, with some tweakable constants to account for the contributions of functions of varying units.

$$\begin{aligned} f_{eval} = & (c_1 \cdot f_{track} + c_2 \cdot f_{damage}^2 + c_3 \cdot f_{kills}^3 + c_4 \cdot f_{travel} \\ & + c_5 \cdot f_{goal}^2) - (c_6 \cdot f_{collisions} + c_7 \cdot f_{selfdamage} \\ & + c_8 \cdot f_{death}^2) \end{aligned}$$

## 2.10 Bot Compiler

Although we initially designed the ancestor FMM using the C-like notation described in the section 2.5, the breeder stores the bot code in a compact format, where every instruction occupies exactly 16 bytes. Additionally, during the matches, bot code is compiled into the native x86 format to improve performance. Bot compiler is responsible for translating between compact, intermediate C-like and native representations of the bot code.

Translation between compact and intermediate representations may be performed in either direction. Internally, the breeder keeps all of bot representation in compact format for the ease of mutation. The ancestor code, however, is initially written in the C-like format, for the convenience of the human author, and requires translation into the compact format by the breeder.

C-like representation serves as an intermediate form necessary for bot code to be compiled into the native code. When the breeder invokes the bot compiler to translate the compact bot code into native code, the bot performs the following functions:

- Translate source code into the intermediate C-style code.
- Insert the C-style code into a generic C++ source code template, creating a fully qualified C++ code.
- Invoke GNU C++ compiler to produce a dynamically linking library.
- Copy the DLL to the Counter-Strike library directory.

The bot compiler wraps the intermediate bot code in a template which results in the creation of a C++ class which inherits from the “EvoBot” class, has a name of the form “EvoBot\_ $G\_N$ ,” and is compiled into “EvoBot\_ $G\_N$ .so” (where  $N$  varies for every one of the 128 bots of the same generation  $G$ ). This is necessary to avoid name collisions when the in-game bot handler loads the bot DLL.

The fragment of a EvoBot class header in Figure 5 illustrates how it is possible to simply insert the intermediate bot code into the *think()* method of a class which inherits from the EvoBot class. For debugging purposes, most arithmetic, comparison and assignment operators of the EvoRegister class are overwritten, and implement various safety and range checks to ensure that correct ranges are accessed. The EvoRandRegister class refreshes its value to a random upon access, which is achieved

by overloading the *float* conversion operator. All arithmetic and logic is performed by converting EvoRegister class to its floating point value.

```
#define MAX_MODULES 256
#define MAX_REGISTERS 65536
#define VISU_DIMENSION 16

#define transition(x) goto _evoModLabel##x
#define module(x) _evoModLabel##x: \
if (interrupted()) return
#define nop() /* */
#define halt() return

// Input register indices
enum {RAND,AUDI,PAIN,OBST,VISU};
// Directional indices
enum {L,R,F,B};
// Obstacle flag notifiers
enum {IS, JUMP};
// Visual fields
enum {DIST, NME, GOAL, DOOR};
// Output register indices
enum {PLAN,ROTA};
// Coordinate selector
enum {X, Y};

class EvoBot {
    friend class BotHandler;

protected:    // Can be accessed by friend classes
    // Input registers
    EvoRandRegister iRAND;
    EvoInputRegister iAUDI[2],
        iPAIN[4],
        iOBST[4][2],
        iVISU[VISU_DIMENSION][VISU_DIMENSION][4],
        iVISU_X[VISU_DIMENSION][4][VISU_DIMENSION],
        iVISU_Y[VISU_DIMENSION][4][VISU_DIMENSION];

    // Output registers and action flags
    EvoRegister oPLAN[2], oROTA[2];
    int actions;

    // Temporary registers
    EvoRegister t[MAX_REGISTERS];

    // Make a decision
    virtual void think(timestamp_t delay);

    ...

public:

    ...

    // Math helpers
    EvoRegister sum(EvoRegister *arg);
    EvoRegister avg(EvoRegister *arg);
    EvoRegister max(EvoRegister *arg);

    // Actions
    void jump();
    void crouch();
    void fire();
    void altfire();
    void reload();
    void switchw();
    void bomb();
};
```

Figure 5: Fragment of the EvoBot header definition.

### 3 Experiments



Figure 6: An abbreviated map of de\_dust: the test environment.

Our experiments were conducted on a single “map” of the game, *de\_dust*. It is a detonation scenario, in which terrorists attempt to explode either of the two large crates of military ammunition while counter-terrorist must either prevent the detonation either by eliminating the terrorist team or defusing the bomb once it is activated.

*De\_dust* (whose simplified view from the top can be seen on the Figure 6) is a relatively flat map, with two tunnels connecting the terrorist and the counter-terrorist base camps. One tunnel is a dark hallway with multiple exits, while another is an underground pass.

We ran a total of 288 cycles of evolution in the course of four days. The various system parameters were fixed at the following levels:

|                                                                 |                |
|-----------------------------------------------------------------|----------------|
| Max. modules per bot                                            | 256            |
| Max. instructions per module                                    | 256            |
| Visual matrix dimension                                         | 16             |
| Temporary registers per bot                                     | 256            |
| Fraction of bot modules to split/join                           | 5%             |
| Prob. of a split instead of join                                | 50%            |
| Maximum number of crossovers                                    | 5 (7 segments) |
| Fraction of bots to mutate                                      | 5%             |
| Prob. of mutating whole instruction (not just the arguments)    | 50%            |
| Fraction of instructions mutated                                | 8%             |
| Chance of a mutated argument becoming a number (not a register) | 10%            |
| Maximum distance of mutation for a number argument              | 100            |
| The following weights are assigned to score point values:       |                |
| Tracking score weight                                           | 200            |
| Damage to enemy score weight                                    | 500            |
| Killed enemy score weight                                       | 10000          |
| Travel score weight                                             | 10             |
| Team score weight                                               | 5000           |
| Goal tracking score weight                                      | 10             |
| Collisions penalty weight                                       | 50             |
| Damage to self penalty weight                                   | 250            |
| Death penalty                                                   | 25000          |

The experiments were conducted on 1.8 GHz Pentium 4 with 1 GB of memory. The architecture has been implemented in a number of programming languages. The breeder, the bot handler and the bot compiler are written in C++ and together comprise over 3200 lines of code. The orchestrator process which invokes other components is written in Perl and its source is nearly 500 lines long.

Our implementation is written specifically for Linux on the i586 or higher architectures. It is based on the HPB bot version 2.1 and must be used with the Counter-Strike/Half-Life engine build 2006.

### 4 Analysis

In general, the results of our experiments indicate that more time would be necessary to produce interesting species. There are a total of  $256^2 = 65536$  instructions, and each instruction has slightly less than 20 *billion* combinations, resulting in a total space of 1.3 *quadrillion* possible bots. Naturally, in a universe this large, finding good individuals may take millions of generations. The ancestor bot reduces the length of the mutation path necessary to achieve rudimentary behaviors. The first question of interest then is whether the evolution will be capable of

finding the individuals which perform better than the ancestor designed by a human, or whether it will simply deteriorate the skills of the ancestor, similar to Tom Ray's system. The second question we explore is whether evolution produces any innovative behaviors. In the following sections, we first discuss the results of our evolution simulations, and then focus on how well our questions were addressed by the results.

## 4.1 Evolution Graphs

The subsequent sections contain a number of score graphs. The graphs depict maximum scores obtained by any bot of a particular generation. The average graphs are not shown, however, the averages do not change significantly over the course of the evolution, due to very small population sizes and considerable rates of mutations, which seem to worsen the performance of a fraction of bots. The graphs are of significant variance, mostly due to common use of the random register in the bot logic. In retrospect, introducing randomness and the random walk behavior into the ancestor bot was a poor choice, due to highly non-deterministic and irreproducible behavior, leading to high score variances given the same bot code.

## 4.2 Hierarchy of Abilities

The basic skills of our Counter-Strike bots strongly inter-depend. In particular, the base camps where "terrorists" and "counter-terrorists" begin the match are distant from one another. Therefore, in order to learn how to attack the enemy and defend themselves the bots must first be competent at walking toward the areas where it may be possible to encounter and engage members of the opposing team. Once such behavior is achieved, bots will have the opportunity to evolve the mechanisms of aiming, firing weapons and damage avoidance. Analogously, damage and teamplay components of the bot score only begin contributing to the total score after bots learn to locomote well enough to encounter each other. The only exception may arise due to bots falling off dangerous heights, however, there are few places where this may happen.

## 4.3 Locomotion and Collision Avoidance

Graphs visualizing the locomotion and collision avoidance scores are shown on the Figure 7. It is evident that, initially, evolution tried to reduce the collision score by constraining how much the bot moves in generations 0-70. Once an "equilibrium" between movement and collision levels has been reached, evolutionary selection proceeded to refine both aspects of locomotion.

The ancestor code drives the early generations of bots to walk in a random manner, lacking obstructions in its path. Due to the game engine interpolation of the motion vectors of entities, their movements do not appear jerky or abrupt. However, due to interpolation, the ancestor bot's trajectories often appear to consist of wide arcs. The ancestor bot's collision avoidance seems proficient at directing the bot away from the environmental obstacles, and jumping if a medium-height obstacle is straight ahead. However, in the beginning of the round, the bots appear at their base camp in a closely situated group. As a result, the bots would often get stuck by running into one another. In order to get untangle the bots, the game engine requires that the bot velocity vectors oppose each other, which is highly unlikely in when the bots attempt to walk randomly or inverse their directions as directed by collision avoidance. Therefore, bots would form infrequently dissipating "clusters" of entities.

As one would expect, such clustering drastically reduces the traveling and the goal-tracking components of the score of each bot entangled in the cluster. Bots are not currently capable of detecting friendly entities, only as obstacles in the distance matrix. To counter this affliction, on iteration 141 evolution produced a defense mechanism against entanglement, which reduced the collision scores and improved the traveling score. Emergence of the following code resulted in the sudden reduction in entanglement rates:

```
...
# NME component frequently 0
t[0] = iVISU[12][4][NME] + iVISU[8][7][DIST];
...
t[23] = t[0] < 42.000000;
...
if (t[23])
transition(32);
...
module(32);
...
# Move back
oPLAN[Y] = 0 - 250.000000;
...
# Turn, as well
```

```

OROTA[X] = iVISU[5][14][NME] + oPLAN[Y];
...

```

Equipped with this code, the bot seems to avoid anything directly in front of it. Removing this code increased bot the formation of “clusters,” as well as the collision score.

Reliance of the bots on the distance metric of the visual matrix improved began starting from generation 180. A champion individual from the 191st generation used its visual matrix in a manner similar to cat’s use of “whiskers.” The bot checked the cells near the left and right extremities of its visual field and turned the other way, similar to a directly-connected Braitenberg vehicle:

```

...
# iRAND in [0..1)
t[0] = iVISU[0][14][DIST] * iRAND;
...
# iBST is 0 or 1
t[1] = iBST[R][IS] - iVISU[14][9][DIST];
if (t[1])
OROTA[X] = 0.000000 - 45.000000;
...
t[1] = t[0] <= -29.432014;
...
if (t[1])
OROTA[X] = 0.000000 + 45.000000;
...

```

This code was extremely helpful in avoiding bumping into the walls of hallways, which comprise most of the structures in the test environment. Unfortunately, the strategy de-evolved and lost a considerable share of its effectiveness in the 231st generation.

Goal-tracking behavior has also improved with generations. Even though one goal is randomly chosen by the bot from the set of available goals, goals are generally in front of both groups of entities at the beginning of the match. The big jump around the 150th generation is due to the reduction in the rotational deltas applied by the goal-finding algorithms of the remnant of the ancestor code, which made the bot “overshoot” in turning toward the goal. The score decreased near the 210th generation due to an instruction which was impeding the speed of bot motion mutating the code bot the chapion bot of the 207th generation. By increasing its velocity, bot became effectively less responsive in turning toward the goal.

Tracking the enemy did not factor into the global score until the 112th generation, simply because none of the bots locomoted far enough to encounter the opponents. From generation 147 on, a

number of bots was finally capable of reaching common meeting places, where they mingled while accumulating enemy tracking scores. In generation 207, the improved speed allowed bots to get to the opposing team faster, resulting in increased enemy tracking score. Unlike tracking a static goal landmark, enemy tracking did not suffer from the decrease in bot sensitivity since bot still had abundant opportunities to face their moving opponents.

## 4.4 Combat

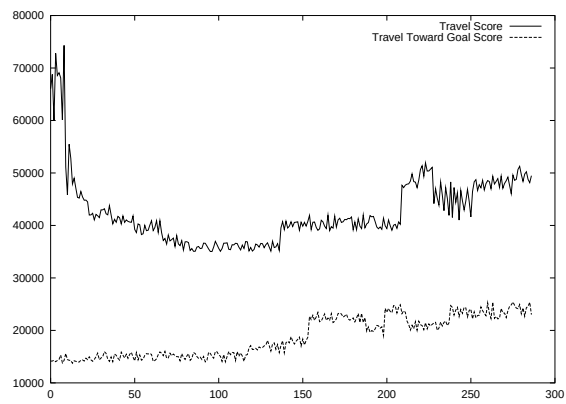
The score graphs for combat-related components are shown on the Figure 8. As expected, the graphs of scores of bots which inflicted the maximum damage are very similar to the graphs of the scores of bots which sustained the maximum damage. A number of exceptions are evident on the “damage to self” graph which arise from bots jumping or dropping off high ledges. The graphs of scores of bots with highest kills and highest number of deaths (only 1 per match per bot allowed) are also nearly the same.



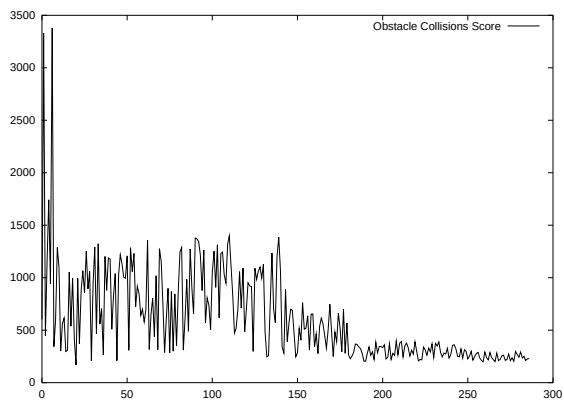
Figure 9: Evolved bot deliberately aiming above the opponent’s head.

In the ancestor code, the combat routine is simplified. First, if the enemy is located anywhere, the bot turns somewhat to face the enemy. Note that the ancestor function does not change the pitch of the bot aim, so only the enemies at the same level are recognized. To aid this, some randomness is added to the aim pitch of the bot. The four cells of the enemy component of the visual matrix are checked, and if the enemy is in either of them, the weapon is discharged with probability of 0.5 every frame, which is necessary for repeated fire.

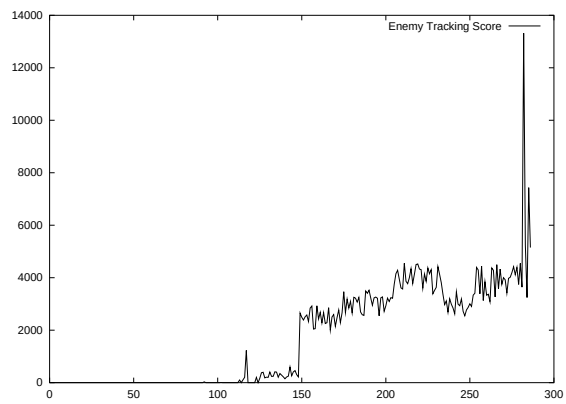
Once the walking behaviors evolved to allow bots from opposing teams to encounter each other, contrary to my expectations, they failed to engage each other. Upon closer inspection I found the combat code from the original ancestor garbled and split into separate modules by mutations. Around the 205th generation, some bots have learned to shoot randomly using *fire()* instruction, and earned some points for hitting enemies. On the 242nd generation, bots have mutated some enemy-tracking logic into



(a)

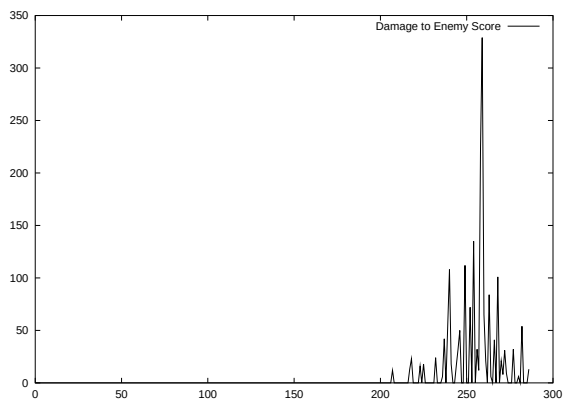


(b)

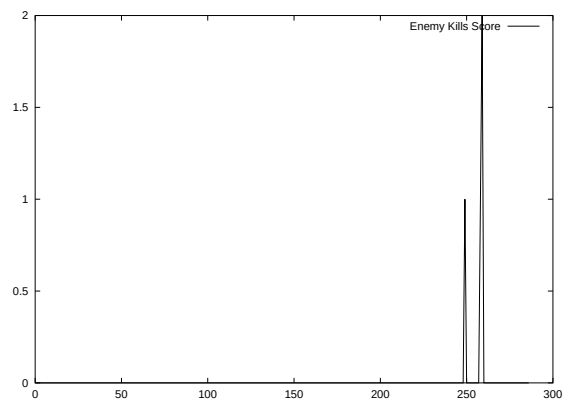


(c)

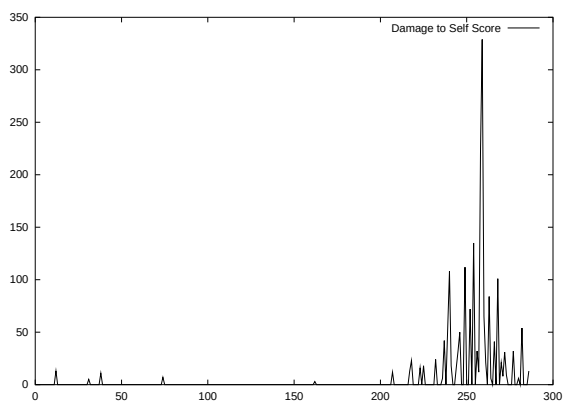
Figure 7: Evolution of locomotion and tracking with generations.



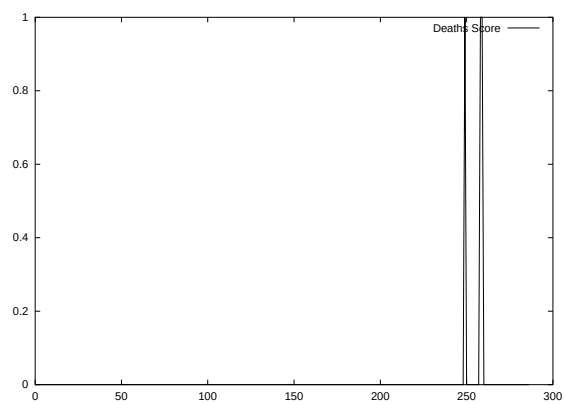
(a)



(b)



(c)



(d)

Figure 8: Evolution of combat abilities with generations.

logic fore the essential tactic of shooting if the enemy cell near the center of the screen is active.

The champion bot of the 260th generation scored 2 kills. The bot was neither extremely accurate in its aim, nor consistent in its tracking of a single enemy. The bot managed to kill two opponents by the feature of the game which dictates that bullets striking upper extremities of humanoid entities in the game inflict greater damage. In particular, a direct shot to the head often kills the opponent instantly, inflicting greater than 100 points of damage.

But the interesting part of the bot's behavior was the method the bot employed to achieve the headshots. Upon detecting an enemy in a central cell of its visual matrix, the bot would shift its pitch by an angle of relatively high magnitude, resulting in an aim at a point considerably higher than the opponent's head, as seen in the following code excerpt:

```
...
if (iVISU[9][7])
transition(101);
...
module(101);
...
OROTA[Y] = iRAND + 24.521147;
...
fire();
...
```

I later pitted champion bot against a remote-controlled opponent and took a snapshot of the bot in action while observing from the first-person view, presented in the Figure 9. The snapshot confirms the method by which the bot was able to accumulate enemy kills.

To discover why shots aimed so high struck the opponent's head, I consulted the members of Counter-Strike Central IRC channel. The conversation revealed that majority of the game's weapons feature an recoil-based inaccuracy such that the recoil from sequential weapons fire shifts trajectories of the consecutive bullets lower than the aim of the bot's entity. Thus, effectively, the bot has learned about the idiosyncrasies of the weapons and the advantages of aiming at the opponent's head.

Damage avoidance is not an ability which evolution chose to pursue in our experiments. This is possibly due to the ancestor bot's instructions for pain avoidance being crippled by splitting the module responsible for such action.

I observed a number of interesting peculiarities when overseeing combat in some of the evolutionary matches. Once, I observed a bot that switched his weapon to a tactical knife (standard for

all players), and in a true Hitchcock-esque fashion preceded to run around frantically stabbing the air in front of him. Other bots seemed to enjoy firing and reloading their weapon. It was also very amusing to observe that dead bodies of bots continue to be controlled by the bot logic for two to three seconds after their death, making the their bodies seemingly "breakdance" on the floor<sup>6</sup>.

## 5 Macroscopic Analysis

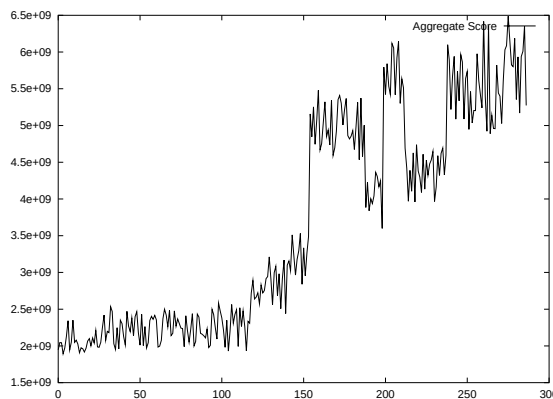


Figure 10: Evolution of the total score.

The graph on the Figure 10 shows that the evolution was somewhat successful in improving the behavior of the ancestor bot, according to our definition of "improving." The graph increases due to gradual improvement in the subscores which comprise the aggregate score's subcomponents. The three sets of "spikes" are due to improvements in methods of locomotion, tracking and attack discussed in the last section.

Our results are both encouraging and discouraging. Our bots have improved upon the ancestor bot's abilities to locomote toward pre-determined landmarks, and learned how use the distance component of the visual matrix to avoid the obstacles ahead. They have learned about engaging the enemy, and accounting for peculiarities of their weapons. However, bots which we evolved are not yet convincingly intelligent. They still stumble around like blind kittens, bumping into obstacles, clustering into tangled groups and getting stuck in tight corners. They infrequently engage the enemy in their viewfield, and are very imprecise in their weapons fire. Human opponents are capable of completely decimating our bots in online matches.

The bots produced by our evolutionary system fell short of our original expectations. Although they did exhibit a great amount

<sup>6</sup>I apologize for all of this morbidity.

of variation in their approaches to locomotion, their randomness made it seem as if the bots got there by accident as opposed to their creativity. The “novel” behaviors discovered by evolution, such as avoiding walls and shooting at opponents, are a matter of common sense to any human Counter-Strike player and bots utilizing pre-computed information about the environment. Despite of all the improvements, the bots failed to achieve truly intelligent behaviors.

A multitude of factors is responsible for the results presented in our paper. The most important factor is the insufficient number of cycles of evolutions, due to the inordinate time demands of each of our evolution cycles. This problem is further aggravated by small population sizes. Combined with the vast size of the universe of possible bots, the chances of producing meaningful bots are minuscule.

On the other hand, I expected the evolution to be more damaging to the functionality of the ancestor bot than it was, given the small population sizes and the relatively high mutation rates. However, due to immense capacity of bot code data structures, many of the mutations occurred in unused spaces filled with NOPs such that a transition instruction preceded them, and the mutated instructions were not ever utilized. Analogously, the rigorous module splitting and joining had much less effect on critical functions than originally expected.

Another factor in the shortcomings of our results is due to the resolution of the sensory information provided to the bots being extremely coarse. Fine-grained values such as vectors toward origins of damage, the objects in the visual field of the bot, et cetera are quantized into large discrete regions. The design trade-off between the simplicity of working with a small quantity of inputs and expecting bots to learn about vector math came at a cost of extreme imprecision. For instance, a bot whose vision is at resolution of 16-by-16 is at a severe disadvantage compared to the humans perceiving the world through an 800x600 monitor (and have the opportunity for precise aiming over long distances). Additionally, a visual matrix of even dimensions poorly serves the needs of a bot which aims at the exact center of its field of vision.

Despite failing to exhibit sufficient intelligence, our project achieved enough progress to support the use of the evolutionary paradigm in computer games. If in 288 cycles of evolution, the bots managed to improve their locomotion and engage the enemies, perhaps it may be possible to further their abilities in carefully constructed evolution systems using improved sensors and computational paradigms. In particular, it is obvious that using evolution to control low-level behaviors such as walking and turning may be both expensive computationally and producing results which do not look convincingly intelligent. Perhaps using

evolution to determine the next point to locomote to and using a pathfinding algorithm to get there may be a better combination. Also, it became clear during our experiments that bots may not learn very much enough from interacting with one another, instead achieving an ecosystemic “intellectual” equilibrium facilitating no further improvement to the intelligence. A better idea may involve a real-time evolutionary system, which learns both from the actions of other human players and from mistakes in actions of its own bots when they lose to human players.

## 6 Conclusions and Future Work

In this work, we designed, implemented and experimented with an evolutionary artificial intelligence system which implements the embodied paradigm for entities in 3D tactical games. The main goals of our research were to determine the fitness of the evolutionary approach to developing bot control mechanisms, including any innovative approaches to gameplay. In our experiments, we achieve sufficient progress to support the claims of suitability of evolutionary approaches to the game intelligence of computer-controlled opponents. Some bot intelligence developed in the course of our evolution significantly expanded on the abilities of the ancestor logic, taking advantage of the sensors and utilizing the tools at the bot’s disposal in novel ways.

In continuing our exploration, we plan on significantly furthering the evolution of our bots. We will attempt to improve the sensory information with more feedback from the engine, for instance making the bot aware of the number of bullets left in the weapon, and the total number of bullets suitable for the bot’s current weapon. In improving the bot’s ability to handle weapons, we intend to enable the bot to purchase weapons at the beginning of the round, be aware of the current weapon, in hopes of enabling bots to specialize in certain weapons.

Additionally, we would like to augment make our system real-time to play alongside human opponents, and learn from the matches. We will soon expose our server to the world-wide clientele of the Counter-Strike players, and populate half the available player slots with bots, evolving in real time. By introducing dual banks of bots, we will be able to perform the CPU-intensive tasks of breeding and compilation in the background on one bank of the bots while another is driving the actions of the bots engaged in the current match. We expect the bots to develop much more complex sets of behaviors by playing along with and against the human players.

## Acknowledgments

Foremost, we would like to thank God for inventing evolution. We would also like to thank Professor Brooks for teaching this fascinating class. Additionally, we would like to thank the class

TA's Jessica Banks and Aaron Edsigner for all their help and clarifications. Special thanks to Botman for making the source to his rudimentary Bot framework available on the Internet.

## Appendix A: Ancestor Bot Source Code

```
# Locomotion

# Find max goal
# Integrate over goal bits
module(0);
t[0] = max(iVISU_Y[0][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[1][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[2][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[3][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[4][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[5][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
```

```
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[6][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[7][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[8][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[9][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[10][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[11][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
```

```

        transition(20);

# Check next
t[0] = max(iVISU_Y[12][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[13][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[14][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Check next
t[0] = max(iVISU_Y[15][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(16);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(20);

# Fall through
transition(24);

# Move right
module(16);
t[10] = iRAND < 0.5;
if (t[10])
    transition(17);
transition(18);
module(17);
oROTA[X] = oROTA[X] - 10.0;
transition(19);
module(18);
oPLAN[X] = oPLAN[X] - 10.0;
module(19);
# Goto move
transition(25);

# Move left
module(20);
t[10] = iRAND < 0.5;
if (t[10])

```

```

        transition(21);
transition(22);
module(21);
oROTA[X] = oROTA[X] + 10.0;
transition(23);
module(22);
oPLAN[X] = oPLAN[X] + 10.0;
module(23);
# Goto move
transition(25);

module(24);
# Random move
t[0] = iRAND + 0.0;
t[1] = t[0] < 0.2;
if (t[1])
    transition(16);
t[1] = t[0] > 0.8;
if (t[1])
    transition(20);
transition(25);

# Move forward
module(25);
oPLAN[Y] = oPLAN[Y] + 250.0;

# Audio discrimination
module(26);
t[0] = iAUDI[R] - iAUDI[L];
if (t[0])
    transition(27);
transition(28);
module(27);
oROTA[X] = oROTA[X] - 5.0;
transition(29);
module(28);
oROTA[X] = oROTA[X] + 5.0;

# Pain avoidance - strafe if hurt
module(29);
if (iPAIN[F])
    transition(30);
if (iPAIN[B])
    transition(30);
if (iPAIN[L])
    transition(30);
transition(31);
module(30);
oPLAN[X] = oPLAN[X] - 150;
transition(32);
module(31);
oPLAN[X] = oPLAN[X] + 150;

# Obstacle avoidance with jumping
module(32);
if (iOBST[F][JUMP])
    jump();
# Very stuck
t[0] = iOBST[L][IS] + iOBST[R][IS];
t[0] = t[0] + iOBST[F][IS];
t[1] = t[0] >= 3.0;
if (t[1])
    oROTA[X] = 0 - 180.0;
if (t[1])
    transition(33);

# Front stuck
t[0] = iOBST[F][IS] == 1.0;

```

```

t[1] = iOBST[L][IS] + iOBST[R][IS];
t[2] = t[0] + t[1];
t[2] = t[2] == 1.0;
if (t[1])
    oROTA[X] = 0 - 180.0;
if (t[1])
    oPLAN[Y] = 0 - 250.0;
if (t[1])
    transition(33);

# Left corner stuck
t[0] = iOBST[L][IS] + iOBST[F][IS];
t[1] = t[0] >= 2.0;
if (t[1])
    oROTA[X] = 0 - 45.0;
if (t[1])
    transition(33);

# Right corner stuck
t[0] = iOBST[R][IS] + iOBST[F][IS];
t[1] = t[0] >= 2.0;
if (t[1])
    oROTA[X] = 0 + 45.0;
if (t[1])
    transition(33);

# Back stuck
if (iOBST[B][IS])
    oROTA[X] = 0.0 + 0.0;
if (iOBST[B][IS])
    transition(33);

# Side stuck
if (iOBST[L][IS])
    oROTA[X] = oROTA[X] - 30.0;
if (iOBST[R][IS])
    oROTA[X] = oROTA[X] + 30.0;

# Enemy location and engagement
module(33);
t[0] = max(iVISU_Y[0][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[1][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[2][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;

```

```

if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[3][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[4][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[5][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[6][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[7][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[8][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
    # Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
    # Move left

```

```

        transition(53);

# Check next
t[0] = max(iVISU_Y[9][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[10][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[11][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[12][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[13][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[14][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Check next
t[0] = max(iVISU_Y[15][GOAL]);
t[1] = t[0] > 8.0;
if (t[1])
# Move right
    transition(49);
t[1] = t[0] <= 8.0;
if (t[1])
# Move left
    transition(53);

# Fall through
transition(59);

# Move right
module(49);
t[10] = iRAND < 0.5;
if (t[10])
    transition(50);
transition(51);
module(50);
oROTA[X] = oROTA[X] - 20.0;
transition(52);
module(51);
oPLAN[X] = oPLAN[X] - 20.0;
module(52);
# Goto move
transition(57);

# Move left
module(53);
t[10] = iRAND < 0.5;
if (t[10])
    transition(54);
transition(55);
module(54);
oROTA[X] = oROTA[X] + 20.0;
transition(56);
module(55);
oPLAN[X] = oPLAN[X] + 20.0;
module(56);
# Goto move
transition(57);

# Check enemy and fire
module(57);
if (iVISU[8][8][NME])
    transition(58);
if (iVISU[7][8][NME])
    transition(58);
if (iVISU[7][7][NME])
    transition(58);
if (iVISU[8][7][NME])
    transition(58);
transition(59);

# Fire
module(58);
t[0] = iRAND < 0.5;
if (t[0])
    fire();

# Finita
module(59);
# Adjust crosshair pitch randomly
t[0] = iRAND * 20.0;
oROTA[Y] = t[0] - 20.0;
halt();

```

## References

- [1] Game developer magazine online.  
<http://www.gdmag.com>.
- [2] Sierra studios press release. <http://www.sierra.com/pr/>.
- [3] Computer games history museum.  
<http://www.videogames.org/html/>.
- [4] COLLINS, R. J., AND JEFFERSON, D. R. Antfarm: Towards simulated evolution. In *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, Eds. Addison-Wesley, Redwood City, CA, 1992, pp. 579–601.
- [5] FAUCETT, L. *Fundamentals of Neural Networks*. Prentice-Hall, 1994.
- [6] RAY, T., LANGTON, C., TAYLOR, C., FARMER, D., AND RASMUSSEN, S. An approach to the synthesis of life. In *Artificial Life II* (Santa Fe, New Mexico, February 1990).