

# Visual Control in Quake II with a Cyclic Controller

Matt Parker and Bobby D. Bryant

**Abstract**—A cyclic controller is evolved in the first person shooter computer game Quake II, to learn to attack a randomly moving enemy in a simple room by using only visual inputs. The chromosome of a genetic algorithm represents a cyclical controller that reads grayscale information from the gameplay screen to determine how far to jump forward in the program and what actions to perform. The cyclic controller learns to effectively find and shoot the enemy, and outperforms our previously published neural network solution for the same problem.

## I. INTRODUCTION

Humans are able to accomplish a large variety of complex tasks using a two-dimensional visual image. Camera-mounted vehicles can be controlled remotely by humans. Complex video games are played using a two-dimensional screen as the main informational output to the human players. A doctor can perform surgery remotely using a transmitted camera image attached to a robotic arm. Two dimensional visual information is useful to humans, who have an extremely complex visual system built to quickly process raw visual data. Computer algorithms designed to process raw visual data are generally also very complex, and computationally intensive due to the large number of color values usually mapped over an image. Processing a series of images in real-time, such as may be needed for a camera-mounted autonomous vehicle, is difficult because the visual computation must be fast enough to finish before the next image needs to be processed.

There are some examples of research that use neural networks with raw visual input as controllers in a real-time environment. Pomerleau trained a neural network with backpropagation to drive a van using a 30x32 grayscale image as input [1]. The controller was trained in real time to imitate a human driver. Baluja repeated the experiment, replacing the backpropagation learning with an evolutionary computation model that allowed for more robust control, but limited the training to recorded sets rather than real-time observations of the driver [2]. A car controller was trained to use active vision to race around a track in a realistic driving simulator in research by Floreano et al. [3]. Kohl et al. trained another virtual car-racing controller using a 20x14 grayscale input image and NeuroEvolution of Augmenting Topologies to evolve a vehicle warning system [6]. They then transferred this technique to an actual robot with a mounted camera. Through these experiments neural networks were shown to be a useful controller design for real-time visual control.

Matt Parker (mparker@cse.unr.edu) is a graduate student at the Department of Computer Science and Engineering, University of Nevada, Reno.

Bobby D. Bryant (bdbryant@cse.unr.edu) is an assistant professor at the Department of Computer Science and Engineering, University of Nevada, Reno.

Many vision research experiments that take place in a virtual world use synthetic vision [4][5] to simplify the visual field before processing is performed by the controller. This pre-processing might change the color of objects to simplify the distinction of angles, entities, and movements. Extra-visual data such as distance values might be included with every color input. Enrique et al. re-rendered the visual input as two separate color-coded views; one displayed the color according to the velocity of the object, and one displayed color according to wall-angle or entity identification. They used this high-level visual data to hand-code a robot controller that could navigate a map and pick up boxes of health [4]. In research by Renault et al., an agent was hand-coded to use extra-visual pixel data to walk down a hallway; every pixel contained a distance-from-agent value, an entity identifier value, and a color value [5]. The preprocessing used in Synthetic Vision is useful for simplifying the raw visual input before it gets to a controller, but unfortunately much of Synthetic Vision requires extra-visual information that is difficult to obtain in real-world applications.

First Person Shooters (FPS) are a popular 3-dimensional game genre that immerse a player in an environment to participate in a simulated gunfight (figure 1). FPSs have been used often for AI research. Bauckhage et al. conducted research that used pre-recorded demos of human players to teach separate neural networks: one network learned to aim during combat by changing yaw and pitch, another to change x and y velocity during combat, and another to traverse the map [7]. In research by Zanetti and El Rhalibi, neural networks were trained for Quake III to maneuver an agent through the map, perform combat, and collect items. Their research used neuroevolution to evolve the network to imitate pre-recorded demos by human players, using the player's location and weapon information as inputs [8]. Quake II was used again by Graham et al. along with neuroevolution to learn path-finding strategies and obstacle avoidance [10]. A Neural Gas method was used by Thureau et al. to train an agent in Quake II to navigate waypoints by observing the movements of real players in pre-recorded demos [9].

In our previous research using vision based controllers for Quake II, an agent controlled by a simple recurrent network [12] was trained by neuroevolution to shoot a moving opponent in an open room [13]. The experiment compared two layouts for the visual input. Both layouts used a low-resolution 14x2 grid of input blocks, each activated to the average grayscale value of the pixels covered by the block. A *uniform retina* layout used 28 blocks of identical size, and a *graduated density retina* used the same number of blocks, but varied the size of the blocks in order to provide higher resolution near the center of the the agent's field of vision and

lower resolution at the periphery. Experiments showed that neuroevolution produced more effective controllers for the graduated density retina than it did for the uniform retina. In fact, only one of six attempts to train the agent using a uniform retina produced a controller substantially better than those in the initial random population, whereas all six attempts to train the agent using a graduated density retina succeeded.

Although the graduated density neural network learned to successfully attack the enemy, its behavior was not as lethal as desired and much time was required for the evolution. The agents trained also seemed to lack concise and decisive behavior; they mainly spun in a circle at a certain speed, shooting constantly, and slowed their turn only when the enemy centrally appeared. It also appeared that the network was only using a few of the central visual inputs for its behavior and neglected entirely the peripheral inputs. These unused inputs were nevertheless computed in each calculation of the neural network. These problems inspired us to look for another solution that would learn more decisive behavior faster than our previous neural network and would also use less computation.

For the experiment reported in this paper, we use a genetic algorithm to evolve chromosomes that represent a cyclic program that consists of instructions to look at regions of the screen and to perform actions. The cyclic program encodes the size and location of the visual regions to read. By comparing the grayscale value of the regions to learned thresholds, the program can either jump ahead in the program or step to the next command. Action commands control the full movement of the autonomous agent. This cyclic controller allows for complex behavior that is comparable to a similarly sized neural network, but with minimal computation and faster learning time.

## II. THE QUAKE II ENVIRONMENT

For the work reported here we use the Quake II engine by Id Software, Inc. as our platform for research. Quake II is an FPS that requires players to fight their way out of a maze on an alien world. It supports multiplayer gameplay for a large number of players in a shared environment. The graphics are dreary and realistic (figure 1). There are many tools available to create custom modifications of the game by changing the models, sounds, maps, and game rules.

We chose Quake II out of many possible FPS engines for several key reasons. First, the code for the game engine is open-source, which is essential for modifying code for experiments and distributing the results. The game engine is licensed under the GNU General Public License (GPL), and the code will compile on most UNIX-like operating systems [11]. The second most important feature of the game is that it is able to render the first-person view in software mode. Other open source FPS engines are available, but most of them only render views via a hardware graphics processing unit (GPU). Using an engine that supports software rendering makes it possible to run multiple experiments simultaneously



Fig. 1. An in-game screenshot from the game Quake II.

on a single computer, or to run experiments in a grid or cluster environment where hardware rendering is not supported.

The preprogrammed opponents for Quake II, called *bots*, usually run directly on the game server. These bots cannot access the screen buffer because the server does not render the game world; it only calculates physics and implements game rules. We implemented our visual controller into a client rather than the server, because Quake II clients render their own views of the world based on information provided by the server. The pixel values of the view rendered by the client can be read and preprocessed as needed for experiments with visual controllers.

## III. THE EXPERIMENT

For the current experiment we duplicated our earlier experimental setup for testing retinal layouts with a neural controller [13]. To simplify these initial experiments in visual control, that setup replaces the default low-contrast Quake II environment with a custom high-contrast environment that makes it easy to distinguish the opponent from the wall, ceiling, and floor (figure 2). Our map is a single open room, about the size of half a basketball court.

In multiplayer mode, players and bots enter or re-enter the game through *spawn portals*. If an entering agent spawns at a place occupied by another agent already in the environment, the existing agent is killed by the spawner. To prevent these accidental “spawn kills” we placed the spawn portals for our test environment above the playing area; entering agents drop to the floor without a risk of accidentally killing their opponent.

In this experiment the goal for the learning agent is to kill a hard-coded enemy opponent. The enemy never shoots, but simply walks around the room with a random side movement speed, a random forward and back movement speed, and a random directional changing speed. These values are all re-chosen individually at random time intervals. This creates a random movement pattern that is still reminiscent of a human player’s movement.



Fig. 2. An in-game screenshot of the environment used for our visual control experiments. The ceiling is brown, the walls are gray, and the floor is white. The target agent is a dark blue. The display of the shooter’s own weapon has also been removed. (Cf. figure 1.)

The learning agent is equipped with a blaster weapon that fires lethal energized plasma bolts; one shot will instantly kill the enemy opponent. The blaster shots travel at a non-infinite speed so the aiming often requires leading to hit a moving target. As the blaster is fired, the blaster energy reservoir is depleted by 5. It can charge up to 25 by disuse, and when fully charged it can fire 5 shots in rapid succession, and less than 5 if it is less charged. We added this feature to give the agent more incentive to learn to shoot only when necessary.

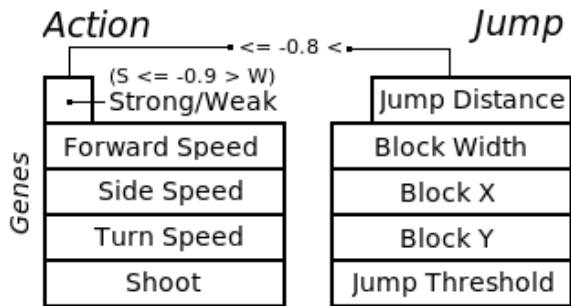


Fig. 3. An instruction consists of five genes. The first gene determines if the instruction is a weak or strong action or a jump and its corresponding jumping distance. The jump instruction looks at a square of pixels and compares the grayscale value to one of its genes to determine whether to jump or step to the next instruction. The action gene controls the agent’s behavior for that frame of gameplay.

#### IV. THE CYCLIC CONTROLLER

The cyclic controller used in this experiment is inspired by those used in previous experiments with Xpilot-AI [14]. The controller’s behavior is specified as a list of instructions, each consisting of five floating-point numbers. There are two types of instructions: jumps and actions. For the visual controller, jump instructions specify a region of the visual field to examine and a conditional jump to another point in the program, depending on the average grayscale value of that region. Action instructions control the agent’s behavior.

The controller is trained by a genetic algorithm (GA). The chromosomes are lists of 455 floating point numbers, which are interpreted as  $455/5 = 91$  instructions for the cyclic controller. We refer to the individual floating point numbers as *genes*. Genes generally fall in the range  $[-1, 1]$ .

The first gene of an instruction indicates if the instruction is a jump or action instruction. In this experiment if the gene is greater than -0.8 then it is a jump instruction; otherwise it is an action instruction. The low threshold favors the creation of jump instructions, to encourage the evolution of programs that examine multiple regions of the visual input before instructing the agent to take an action. The remaining 4 genes in the instruction are used differently depending if the instruction is jump or action, as described below.

##### A. Jump Instructions

The 4 remaining genes in a jump instruction determine where to look in the agent’s 320x240 pixel field of view, and what to do about it. One gene in the instruction specifies the pixel width and height of a square region to examine. The floating point value of the gene is converted into an integer by the equation

$$size = 2 + 20 * (1 + \tanh(g))$$

where  $g$  is the fp value of the gene. Two more genes specify the  $x$  and  $y$  values for the center of the region, as follows:

$$x = 160 + 155 * \tanh(g)^{11} - \frac{size}{2}$$

$$y = 120 + 120 * \tanh(g)^{11} - \frac{size}{2}$$

The equations are exponential to increase the probability that the selected region will be near the center of the agent’s field of view. Visual data near the center of the screen tends to be more important for this task, as shown in the earlier retinal layout experiments.

The grayscale values of the pixels in the selected region are averaged to give a number in the range  $[0, 1]$ ; that average is compared with the final gene of the instruction, called the threshold gene. If the threshold is positive, then if the average grayscale value is greater than or equal to the threshold the program will execute a jump. If the threshold’s value is negative, then if the average grayscale value is less than or equal to the absolute value of the threshold the program will jump. If there is no jump, execution falls through to the next instruction.

To calculate the jump value we use the first indicator gene, which was also used to indicate if it was a jump or action instruction. The range for the jump is between -0.8 and 1.0, so we squash this range down by adding 0.8 to the indicator gene, then dividing it by 1.8, to get a jump value between 0.0 and 1.0. The jump value represents the percentage of the chromosome it should jump, so that, for example, a value of 0.5 jumps forward half the length of the chromosome, 0.0 remains at the same instruction, and 1.0 loops around the entire chromosome back to the original jumping instruction. The chromosome, theoretically, is cyclic, so there is no end or

beginning, but rather loops around indefinitely. In reality the chromosome is a list of floating point numbers and we create the cyclic functionality by using the modulo function to keep the new jump locations within the list. The instructions are strictly separated by every five genes so the jumps always go to the beginning of an instruction.

### B. Action Instructions

An action instruction indicates how the agent should move. Because genes can mutate beyond the range of  $[-1, 1]$  during evolution, we squash their values back into range by applying the hyperbolic tangent to each gene. One of the genes in an action instruction specifies the forward or reverse speed of the agent, with 1.0 indicating the maximum allowable forward speed, -1.0 indicating the maximum reverse speed, and other values interpolated between. A second gene specifies the sideways “strafing” speed, with 1.0 indicating the maximum allowable speed to the right and -1.0 to the left. A third gene, when multiplied by 10.0, specifies the turning speed in degrees. A fourth indicates whether the agent should shoot: if positive then shoot, otherwise hold fire.

We distinguish weak and strong action instructions. The instruction type indicator gene specifies whether the instruction is strong or weak: values less than -0.8 specified that it is an action instruction rather than a jump instruction; values less than -0.9 further specify that it is a strong action instruction, while values between -0.9 and -0.8 specify that it is a weak action instruction. The weak instructions allow the program to decide on actions to be performed when there is no more pressing action to be taken, as follows.

When an agent first enters into the game arena for evaluation, its cyclic program starts execution at the beginning of the list of numbers in its chromosome. Quake II calculates the physics and display of the gameworld at a rate of 40 frames per second; during each frame, the cyclic program executes until it reaches a strong action or any instruction that it has already encountered during the current frame. Any time it encounters a weak action instruction, it saves the instruction in a “last action” variable. If the program ever reaches a strong action instruction it stops, saves that action into the “last action” variable, and sends the agent controls specified by the action to the game server. If a previously encountered instruction is encountered a second time before any strong action instruction, then whatever is currently stored in the “last action” variable is executed: if the duplicate instruction is a weak action, then that action will execute because it was stored in the variable when previously encountered; if the duplicate instruction is a jump then the stored value will still be whatever action was executed during the previous frame of gameplay. At the beginning of the next frame of gameplay the program resumes execution at the next sequential instruction. When the agent first enters the arena the “last action” variable is set to all zeros, which would make the agent sit still and not shoot. This action gene setup allows for diverse and time-dependent behaviors.

## V. TRAINING

We use a Queue Genetic Algorithm (QGA) to evolve the population of chromosomes [14]. A QGA is a first-in-first-out steady-state genetic algorithm. It uses a queue to store the population of chromosomes; the oldest individuals are at the head of the queue and the newest at the tail. Whenever a new individual is needed, it is created by stochastically selecting two parents from the queue by roulette wheel selection and performing crossover between the two parents to create a new child. The child is tested and then added to the tail of the queue, and the oldest individual is dequeued from the head and discarded. The QGA was designed to be easily distributed across a network. This is particularly important with real-time computer games like Quake II, where the game was designed to run at a low speed suitable for human play, and can not be accelerated much without disturbing its network timing and gameplay performance. With a QGA we can easily distribute the fitness evaluations over several simulations running independently on separate computers; the evaluation of one chromosome can begin before the evaluation of the previous one is completed.

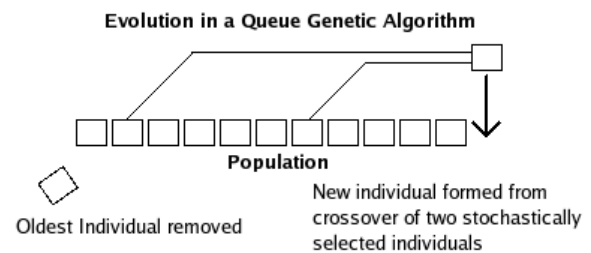


Fig. 4. The Queue Genetic Algorithm (QGA). New individuals are bred from parents chosen from the current population by roulette wheel selection according to fitness. After each new individual is evaluated it is enqueued and the oldest individual is dequeued and discarded.

For this experiment we used a population size (queue length) of 128 chromosomes, each specifying a full 91-instruction cyclic program. The initial population is filled with chromosomes composed of random floating point numbers in the range  $[-1, 1]$ . Breeding uses uniform crossover, with equal chance of drawing each gene from either parent. The chromosome resulting from the crossover is mutated with an independent 10% chance of mutation per gene. The mutations are calculated as a delta drawn from a sharply peaked random distribution,  $\frac{\log(n)}{10} * \text{random}(-1 \text{ or } 1)$ , where  $n$  is a random number in the range  $[0, 1]$ . This produces small deltas with a high probability and larger deltas with a low probability.

The process for testing each individual chromosome is as follows:

- 1) The learning agent appears in the room and drops to the floor.
- 2) The agent is given 24 seconds to kill as many enemy 'bots' as it can.
- 3) Whenever an enemy is killed, it promptly respawns at some random location in the room.

- 4) After 24 seconds the current learning agent is removed and the fitness for its chromosome is reported.

In order to increase selection pressure we report the fitness as  $(5n)^2$ , where  $n$  is the number of kills. Due to the difficulty of the problem and the brief delay between a death of the enemy and its reappearance, the maximum number of kills possible in 24 seconds is about 12.

In our previously published retinal layout experiment, during the start of the evolution the enemy stood completely still. Whenever the population's average fitness reached a certain level the enemy increased slightly in speed. This shaping of the enemy's difficulty enabled the neural network to learn to shoot a quickly moving opponent. Without shaping, with a completely fresh, random population against a quick moving opponent, we found that the network was unable to learn. In order to compare our new cyclic controller to the older neural network we have duplicated the same shaping of the enemy's speed, as well as used the same map and parameters.

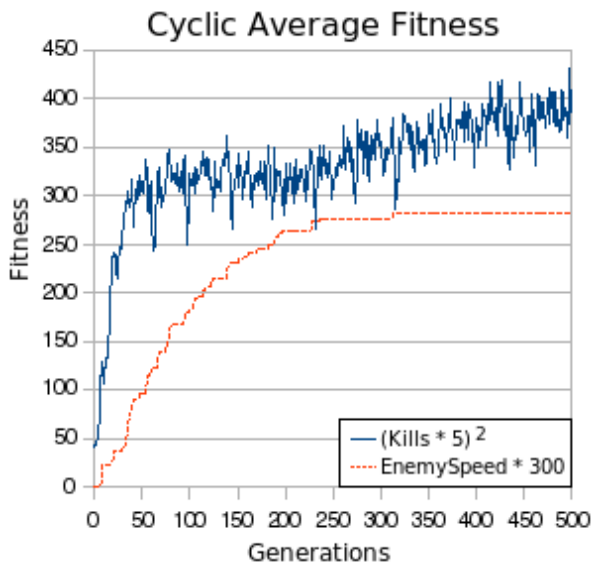


Fig. 5. The average fitness for the cyclic controller of each population averaged over six independent runs. The large solid line represents how well the agents killed the enemy, while the dotted line represents how fast the enemy moved.

Unfortunately, when the cyclic controller was first evolved on this problem with a normal QGA, one out of six tests never achieved a fitness much higher than the initial random population. This can be fixed in several ways. One way is to just ignore the poor tests and use the controllers from the good tests, which would be fine for this experiment since we know that most of the tests will be good. However, we wanted to fix the problem on this experiment so that in future when we try more difficult experiments we will have a technique that will more likely always find some solution. To do this we first looked into Delta-coding, as was used in research by Gomez and Miikkulainen [15]. This technique helps populations when they become stagnant by grabbing the best individual and creating a new population based off of that individual. The new individuals are made up of genes that

are randomly mutated along a Cauchy distribution. Delta-coding is not the best solution for our problem because it is suited for pulling converged populations out of stagnation; our stagnation occurs at the time that the population has yet to converge. Delta-coding also requires a best individual; our "best" individuals, at the early stages in evolution, can often be very lucky individuals with no desirable skill, so it is too inconsistent to build a new population off of a best individual in our case.

We settled on a simple solution that temporarily increases the mutation of a population for one generation at certain times. The population has 40 generations to evolve; after the 40th generation the average fitnesses of the previous 40 generations are averaged together and that average is compared with the average of the current generation. If the current generation's average fitness is less than the average of the previous 40 then the population is assumed to be stagnant and the next generation has a much greater mutation rate. If the current generation's fitness is greater than or equal to the previous 40, then its average fitness is averaged together with the previous 40 and then the next generation is tested normally. The increased mutation rate is like the normal mutation, but there is a 50% chance that each gene will be mutated instead of 10%, and the mutation per gene is calculated by adding a larger delta,  $\frac{\log(n)^2}{10} * random(-1or1)$ , to the gene's original value. This temporary increase in mutation adequately pulls populations out of stagnation and also helps to stimulate more mature populations that are converged but are not improving.

## VI. RESULTS

The cyclic visual controller performed quite well and learned an effective and computationally simple way to kill the enemy. The basic successful behavior is to turn in one direction quickly, looking mostly at one or two blocks near the center of the screen. Whenever one of those blocks darkens the controller goes into a loop of shooting at the enemy. All six tests learned this basic behavior and a shooting behavior. Some tests turned from right to left quickly, showering the enemy with a spray of shots. The more successful tests also side-stepped in one direction while firing at the enemy so that the burst of shots formed a wall that would likely kill the quickly moving enemy.

Figure 5 shows the average of the average fitnesses of the six tests, which begins at generation zero with an average fitness of about 50, or 1.4 kills per turn, and increases up to a little over 400 by the 500th generation; about 4 kills per turn. More importantly, because the average fitness of each test often peaked above the fitness bar of 400, the enemy's speed increased rapidly up to over 90% its full speed. In five of the six tests the enemy reached full speed, and the sixth reached 63% speed, which is about the same as the average speed reached by the neural network tests. That the tests could increase the enemy's speed so rapidly shows that the evolution of these cyclic controllers can quickly make adaptations to adjust to the enemy's faster speeds.

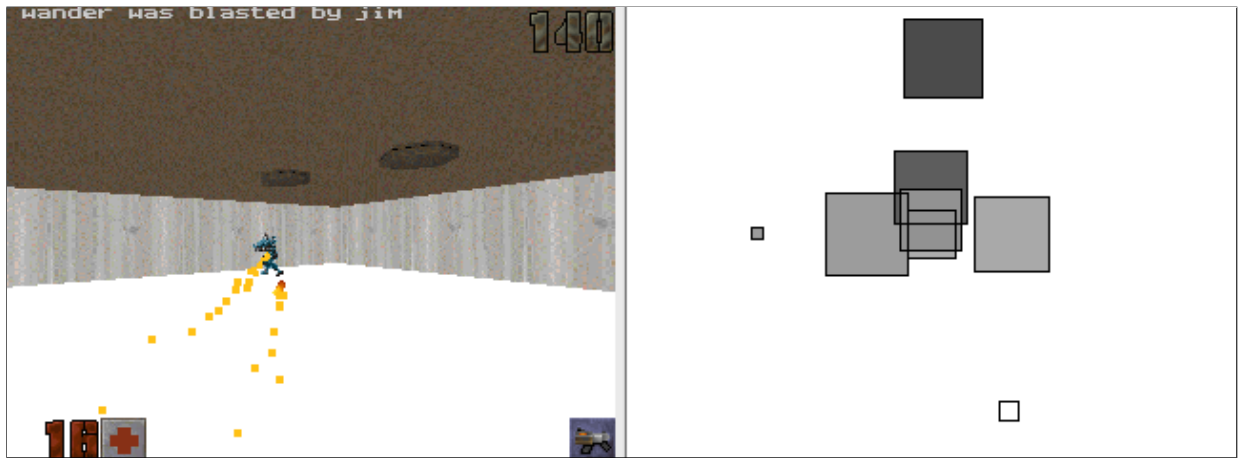


Fig. 6. An in-game screenshot from one of the tests. The left side is the view of the game as generated by the Quake II engine. The right side is the agent's view. Each block where the agent is looking in that frame of gameplay is outlined in black and filled with the grayscale color value of that location on the actual game screen. See Figure 7 for the block-input layout of this controller while it is scanning for the enemy.

There were great variations in the retinal layouts evolved in this experiment. The retinal layouts are generated by the controller executing a series of jump instructions in a frame, either until a jump condition is reached or until an action is performed. Figure 6 shows the in-game screenshot and the coinciding retinal layout for a controller as it is shooting the enemy. This retinal layout differs from the layout generated when the same controller is scanning for the enemy, as seen in Figure 7. The retinal layout used when the controller is shooting the enemy has two extra larger input blocks on either side of the center, which are likely used to determine which direction to turn if the enemy leaves the center. Some of the tests developed a more simplistic solution; the simplest is seen in Figure 8; this controller only uses two block inputs while scanning for the enemy. Allowing the controller to choose its inputs creates an interesting variety of behavior that also executes minimalistic processing.

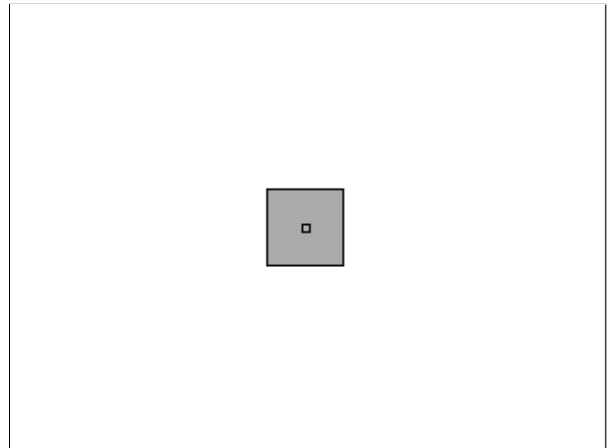


Fig. 8. One test developed a very minimalistic yet effective block-input layout that generally consisted of one large block input and one smaller input, both centered.

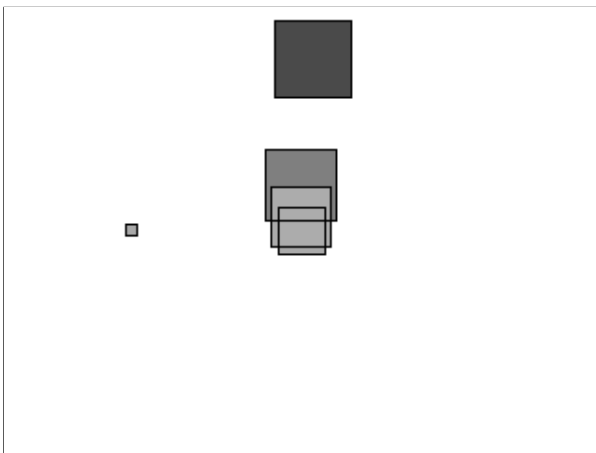


Fig. 7. The evolved block input layout differed between scanning for the enemy and shooting at him. This is a screenshot of the block layout of the same controller from Figure 6, taken while the controller scans for the enemy. In Figure 6, the controller has found the enemy and is shooting at him.

We set up this experiment so that it could easily compare with the behavior of the neural network from our previous

retinal layout experiment [13]. The neural network had 28 block inputs in the layout of a graduated density retina, a recurrent hidden layer [12], and four outputs: forward/back, right/left, turn, shoot. The network learned with neuroevolution, with a chromosome of floating point numbers that represent the weights of the network. The chromosome was evolved using a Queue Genetic Algorithm. Because all of its tests consistently learned, the evolution did not involve any temporary increase in mutation. Figure 9 shows the average of the average fitnesses for the 6 tests using the neural network. The average fitness consistently reaches between 350 and 400 for the second half of the evolution, which is comparable to the cyclic controller. However, the speed of the enemy is rather low, only reaching about 60% of its full speed by the end of the evolution. To graph the combination of fitness and enemy speed, we multiply the fitness by one plus the enemy speed. We add one to the enemy speed so that when the speed is zero the fitness will still appear. Looking at the combination graph for the cyclic controller [Fig. 10] as compared to the neural network [Fig. 11] we can see that

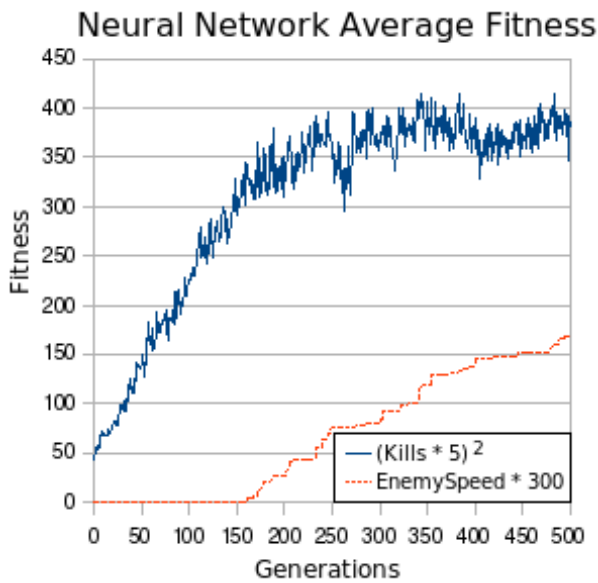


Fig. 9. Average fitness of neural network populations averaged over six tests. The large solid line, which represents the ability to kill the enemy, is comparable to the cyclic controller. The dashed line, which represents the speed of the enemy, was not able to climb as high as in the cyclic tests (Fig. 5)

the combined fitness and enemy speed is much greater than with the neural network.

These results show that the cyclic controller learned faster than our neural network. This does not mean that cyclic controllers are necessarily better than evolved neural networks for real time visual applications; our neural network was “naive” in that it had only a flat hidden layer, which did not preserve any information about the relative positions of the inputs. Not only this, our neural network had a set number of inputs and could not evolve its retinal layout. The cyclic controller could easily prune off its unnecessary inputs and keep its control cycle simple. Because the cyclic controller could remain simple it was easier for it to adapt to changes in the enemy’s speed. For instance, if the new change needed to successfully hit the enemy was a slight side-step to the right when firing, the cyclic controller would need only to change that value for one or two of its dominant action genes. The neural network, however, would likely need to adjust several weights in combination. It would not only need to adjust the weights to do the side-stepping while shooting, but also readjust other weights so that it did not also side-step when not shooting.

The cyclic controllers executed on average only about 18 of their 91 instructions per turn and skipped over the others. The maximum number of instructions used by any chromosome sampled was 48. This small average usage indicates that it may be possible to reduce the chromosome size for this particular problem and still achieve similar success; however, it may be that providing more chromosome space than what is directly used is beneficial for the controllers because it allows the jumps to be clumsier and less precise in order to jump to particular functional regions of the chromosome. Some of the controllers took advantage of the weak and

strong specification for the action instructions. A sampling of the controllers showed that 11.7% of the action instructions that were called were weak action instructions. Some of the tests used the weak action instructions as dummies that were always followed by a strong action instruction. Other tests used the weak instructions to control the agent as intended; in one test 45% of all action instructions called were weak and the strong action instructions only controlled the agent 65% of the time. Probably the reason that weak action instructions were not used more is because only indicators between -0.8 and -0.9 specified that the instruction was weak; anything below -0.9 specified a strong instruction. This provides a fair balance between weak and strong action instructions at the start of the evolution, but eventually the mutations change the instructions to below -1.0 and distort the even balance.

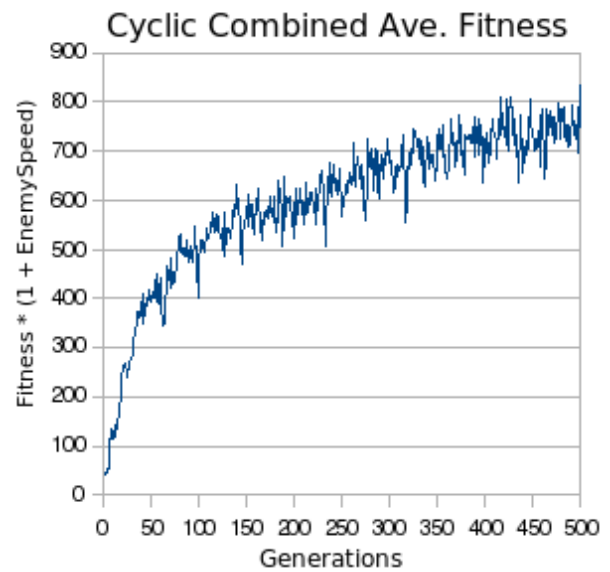


Fig. 10. The average fitness of the cyclic controller combined with the enemy’s moving speed.

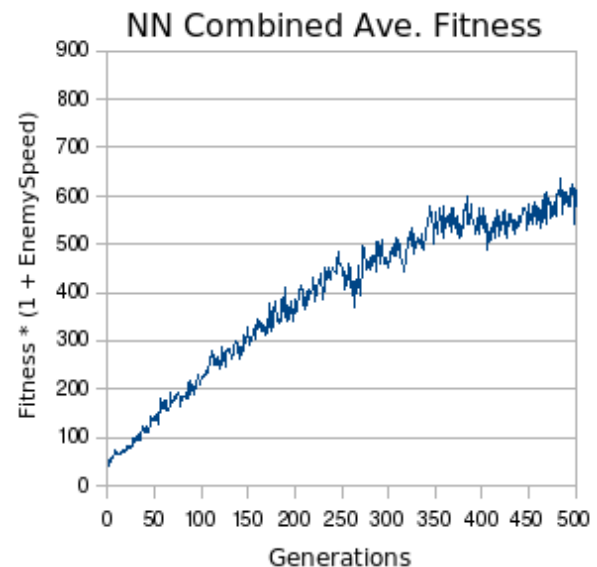


Fig. 11. The average fitness combined with the enemy’s movement speed for the neural network tests.

This type of cyclical genetic structure has successfully been used before to control bots in Xpilot-AI [14]. Ashlock developed a similar system called “ISAc lists”, which are lists that consist of ISAc nodes; each of which contain information about which inputs to compare to one another, how to compare them, and what action or jump to do [16]. ISAc lists were used successfully by Ashlock in a later experiment that simulated grid-drivers in a two-lane traffic simulation [17]. None of these controllers had yet been implemented to control a real-time visual system. Genetic programming and its variants, especially linear genetic programming, has similarities to the cyclic controller used in this paper, but genetic programming is a more generalized evolutionary method and we deemed it excessive for this simple problem [18]. Some of the crossover techniques of linear genetic programming, however, maybe be useful for the cyclic controller and will be explored in future work.

## VII. CONCLUSION

The cyclic controller presented in this experiment uses a genetic algorithm to evolve chromosomes that represent a cyclical program that reads visual input and acts according to its interpretation of that input. The cyclic program is made of two basic instructions: jump and action. A jump instruction looks at a color value of an area on the screen and, after comparing the color value to the instruction’s threshold, either jumps to another place in the chromosome or steps ahead to the next instruction. An action instruction indicates what action the learning agent should take whenever the cyclical program is ended for a frame of gameplay; the weak action instructions set the action and then move on to the next instruction; the strong action instructions set the action, then stop the program and do the action. The cyclic controller will run each frame until it reaches a strong action or the same instruction is visited twice.

In this experiment the agent learned to shoot an enemy who wandered around randomly in a small room. The room was colored so that the walls, floor, ceiling, and enemy were easily distinguishable, even in grayscale. We ran six tests to the 500th generation and all learned successful behaviors. Their fitnesses were much better than those of the neural network used in the retinal layout experiment, which attempted to solve the same problem. The cyclic controller also used much less computation than did the neural network, because it could prune out the unhelpful visual areas of the screen through evolution.

This cyclic controller was overall very successful. It shows that a simple evolved controller may in some cases be more efficient and effective than a neural network for real-time visual problems. It is the first visual controller of its kind and we plan to make many modifications to enhance it in the future, such as by enabling color blocks instead of just grayscale. We also would like to try it in more complex and visually confusing environments and on more difficult tasks to see how well it can scale. We also will experiment with different crossover techniques that may better preserve the

continuity of the cyclic programs, such as by crossing over whole jump sections rather than individual genes.

## VIII. ACKNOWLEDGMENTS

This work was supported in part by NSF EPSCoR grant EPS-0447416. Quake II is a registered trademark of Id Software, Inc., of Mesquite, Texas.

Our modifications to the Quake II source code, including a general-purpose API for client-based AI, is available from the Neuroevolution and Behavior Laboratory at the University of Nevada, Reno, <http://nebl.cse.unr.edu/>.

## REFERENCES

- [1] D. Pomerleau, “Efficient Training of Artificial Neural Networks for Autonomous Navigation”, *Neural Computation*, Vol. 3, No. 1, 1991, pp. 88-97.
- [2] S. Baluja, “Evolution of an Artificial Neural Network Based Autonomous Land Vehicle Controller”, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 26 No. 3, 450-463, June 1996.
- [3] D. Floreano, T. Kato, D. Marocco, and E. Sauser, “Coevolution of active vision and feature selection”, *Biological Cybernetics*, 90(3), 2004, pp. 218-228.
- [4] S. Enrique, A. Watt, F. Policarpo, S. Maddock, “Using Synthetic Vision for Autonomous Non-Player Characters in Computer Games”, *4th Argentine Symposium on Artificial Intelligence*, Santa Fe, Argentina, 2002.
- [5] O. Renault, N. Magnenat-Thalmann, D. Thalmann, “A Vision-based Approach to Behavioural Animation”, *Journal of Visualization and Computer Animation*, Vol.1, No1, 1990, pp.18-21.
- [6] N. Kohl, K. Stanley, R. Miikkulainen, M. Samples, and R. Sherony, “Evolving a Real-World Vehicle Warning System”, In *Proceedings of the Genetic and Evolutionary Computation Conference 2006*, pp. 1681-1688, July 2006.
- [7] C. Bauckhage, C. Thureau, and G. Sagerer, “Learning Human-like Opponent Behavior for Interactive Computer Games”, In B. Michaelis and G. Krell, editors, *Pattern Recognition*, volume 2781 of LNCS, pages 148-155. Springer-Verlag, 2003.
- [8] S. Zanetti, A. El Rhalibi, “Machine Learning Techniques for First Person Shooter in Quake3”, *International Conference on Advances in Computer Entertainment Technology ACE2004*, 3-5 June 2004, Singapore.
- [9] C. Thureau, C. Bauckhage, and G. Sagerer, “Learning Human-Like Movement Behavior for Computer Games”, In *Proc. Int. Conf. on the Simulation of Adaptive Behavior*, pages 315-323. MIT Press, 2004.
- [10] R. Graham, H. McCabe, and S. Sheridan, “Neural Pathways for Real Time Dynamic Computer Games”, *Proceedings of the Sixth Eurographics Ireland Chapter Workshop, ITB June 2005, Eurographics Ireland Workshop Series*, Volume 4 ISSN 1649-1807, ps.13-16
- [11] “Q2 LNX stuff,” Nov 14, 2005. <http://icculus.org/quake2/>
- [12] J. L. Elman, “Finding structure in time”, *Cognitive Science*, 14:179-211, 1990.
- [13] M. Parker, and B. Bryant, “Neuro-visual Control in the Quake II Game Engine”, *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN 2008)*, Hong Kong, June 2008.
- [14] M. Parker, and G. Parker, “Using a Queue Genetic Algorithm to Evolve Xpilot Control Strategies on a Distributed System”, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006)*, Vancouver, BC, Canada, July 2006.
- [15] F. Gomez, and R. Miikkulainen, “Incremental Evolution of Complex General Behavior”, *Adaptive Behavior*, 1997.
- [16] D. Ashlock, and M. Joenks, “ISAc Lists, A Different Representation for Program Induction”, *Genetic Programming 98, Proceedings of the Third Annual Genetic Programming Conference*, San Francisco, 1998.
- [17] D. Ashlock, “Grid-Robot Drivers: an Evolutionary Multi-agent Virtual Robotics Task”, *Proceedings of the 2006 IEEE Symposium on Computational Intelligence in Games*, Reno, NV, 2006.
- [18] M. Brameier. “On Linear Genetic Programming”, *Dortmunder Dissertationen*, Universitt Dortmund, 2005.