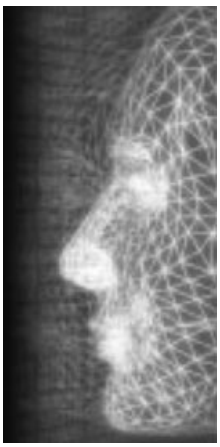
COMPUTER ANIMATION AND VIRTUAL WORLDS
Comp. Anim. Virtual Worlds 2010; **21**: 453–461
Published online 25 May 2010 in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/cav.360

A genetic-fuzzy system for optimising agent steering



By Anton Gerdelan and Carol O'Sullivan*

Fuzzy controllers are a popular method for animated character perception and control. A drawback of fuzzy systems is that time intensive manual calibration of system parameters is required. We introduce a new Genetic-Fuzzy System (GFS) that automates the tuning process of rules for animated character steering. An advantage of our GFS is that it is able to adapt the rules for steering behaviour during run time. We explore the parameter space of the new GFS, and discuss how the GFS can be implemented to run as a background process during normal execution of a simulation. Copyright © 2010 John Wiley & Sons, Ltd.

KEY WORDS: agent navigation and steering; genetic algorithms; fuzzy logic

Introduction

Fuzzy logic controllers are an elegant solution for agents controlling the motion of real time animated characters, creatures and vehicles because fuzzy controllers produce outputs that transition smoothly, rather than stepping between output states, and because they can be implemented to allow agents to react to changing environments and moving obstacles in real time, with minimal processing overheads[1]. This approach is ideal for modern games and 3D simulations that need to simulate large crowds of animated characters, where each character requires its own realistic motion[2].

An outstanding problem with the use of fuzzy controllers for steering and moving animated characters is that the controllers need to be tuned to suit each new type of agent's combination of rôle, physical and performance characteristics, and operating environment. This means that, while the essential kernel of the system—the fuzzy decision making process—applies broadly, all the parameters of the fuzzy systems need to be tailored to suit the peculiarities of each new type of agent.

Building and calibrating fuzzy controllers consists of the following steps[3]:

(1) Designing the size and scale of fuzzy set functions that classify ranges of real (crisp) values, such as distances in meters, into members of discrete fuzzy sets such as *near*, *medium* or *far*. This process is called fuzzification.
(2) Designing the value of fuzzy sets for outputs from the controller. These could be steering angles *light*, *sharp*, and *very sharp*. Using a singleton model, each set must correspond to a single crisp output value, in radians for example. Converting fuzzy output sets to crisp values is known as defuzzification.
(3) Designing a complete set of rules that match all combinations of fuzzy input sets to fuzzy output sets.
(4) Adjusting either fuzzy sets or fuzzy rules to improve the performance of the system.

This process is very time consuming for the designer. Modifying a system by trial and error based on test cases has taken 5–10 hours in our applications. If there are various types of agent involved in a simulation, or an agent has to cope with a large variety of different cases then the required tuning time multiplies. There is no guarantee that hand tuned rules and sets are optimal. The fuzzy controllers are also not able to adapt to any change to the environment after tuning.

Our specific aims in this work are:

(1) To automate the tuning process of fuzzy controllers used for steering and moving agents in a 3D simulation, saving developer time and improving manually calibrated controllers.
(2) To design an architecture that allows this tuning process to happen in real time, giving the agents an adaptive quality.

*Correspondence to: A. Gerdelan, GV2, Computer Science Dept., Trinity College, Dublin 2, Ireland.
E-mail :gerdelaa@cs.tcd.ie

(3) To enable the use of available multi core hardware during the calibration process.

A potential solution to the fuzzy tuning problem could be to use an evolutionary algorithm to automatically tune the fuzzy controllers. The novel contribution of this work is that it provides an automatic calibration mechanism for fuzzy controllers specific to 3D animated agents. This gives fuzzy controllers some of the advantages of the other 3D game evolutionary systems, but does not share the disadvantage of being a closed system 'black box', meaning that the rules evolved by the system are visible and easily manipulated by a user during the process. Future work will have to be done to analyse the performance of this approach within the broad domain of agent steering and movement control.

Related Work

Optimisation of intelligent agents' behaviour using genetic or other evolutionary algorithms is usually performed in staged iterations; one iteration per generation. This has been used to evolve bipedal animated character motion with promising results[4]. Each generation will be evaluated by running the full generation's complement of individuals through a contrived environment. These iterations will continue until a desired fitness or arbitrary generation evaluation limit has been reached. Complete training is usually performed prior to use of the agent in its intended environment, and then no further optimisation takes place. This commonly used training paradigm provides a solution to the time intensive design problem, but it does not address the problem of dealing with varied or changing environments.

An alternative approach is employed in the 3D graphical NERO (Neuroevolution of Robotic Operatives) game[5,6]. NERO is a research platform based on an algorithm called rtNEAT (real time Neuroevolution of Augmenting Topologies). The agents driven by rtNEAT in NERO have behaviours structured on Artificial Neural Networks (ANN). These networks are evolved in real time using a genetic algorithm, the evolution of which is guided by a human player through a reward and punishment scheme. NERO is used to evolve several complex behaviours including reactive navigation. Rather than running the agents through staged, repeatable experiments, the characters are evaluated for about 1 minute, after which they are destroyed and replaced by an agent of the next generation. This speed up of the evolutionary process allows the genetic algorithm to run during simulation execution. NERO also trains 50 agents simultaneously. Although we are interested in fuzzy systems rather than neural networks, we build on the continuous evaluation paradigm, which addresses the problem of adapting training to a changing environment. We also incorporate the parallel approach to reduce training time. Whilst NERO's 'human in the loop' is suitable for directors or game designers that want to interactively direct the evolving behaviour, we are interested in optimising existing steering behaviour whilst minimising human effort, and so aim to fully automate the evaluation process.

Genetic-Fuzzy System (GFS) hybrids combine a genetic algorithm with fuzzy controllers, and have long been used to solve optimisation problems inherent in fuzzy systems by evolving the fuzzy set functions or through tuning of fuzzy rules[7]. However, each new GFS requires a unique, problem dependent architecture and fitness function. Fuzzy-Genetic algorithms have been used for training mobile robots to avoid obstacles. This application domain is inherently similar to animated agent steering, and the GFS approach has been shown to generate reliable fuzzy rules[8]. Although a GFS has been recently proposed for autonomous agent motion in very basic stochastic applications[9], using it for automatic training of 3D animated character navigation is a new approach which requires a new GFS framework specific to the domain.

Table 1 provides a comparative overview of features of different agent steering approaches. The basic Fuzzy system is manually optimised, the Genetic-Neural Network (GNN) is trained in staged batches prior to use, and the final two systems are trained dynamically in real time. An important aspect of differentiation between approaches using evolutionary algorithms (offline GNN, rtNEAT and GFS) is that those based on neural networks are 'black box' systems, i.e. the rules evolved as ANNs are not easily visible or manipulated outside the training

	Fuzzy	GNN	rtNEAT	GFS
Training	Manual	Batch	Online	Online
Black box	No	Yes	Yes	no
Adaptive	No	No	Yes	Yes
Interactive	No	No	Yes	No

Table 1. Comparison of features of agent control systems. A fuzzy controller is compared to a Genetic-Neural Network (GNN), the rtNEAT algorithm, and a Genetic-Fuzzy System.

DOI: 10.1002/cav

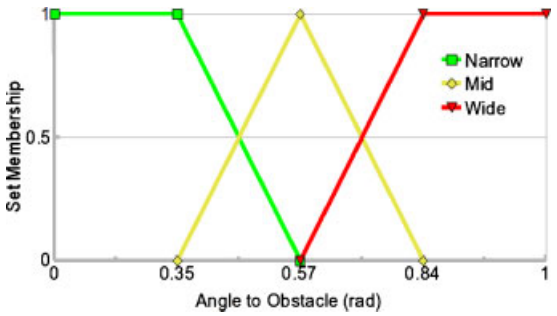


Figure 1. These fuzzy input set functions take a real angle to an obstacle in radians, and fuzzify this into either full or partial membership of the narrow (0–0.57 rad), mid (0.35–0.84 rad) and wide (> 0.84 rad) fuzzy angle values.

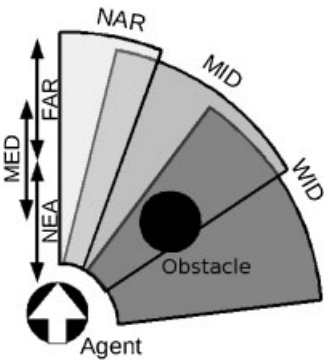


Figure 2. Overlapping fuzzy input sets in a spatial example. The agent has classified an obstacle as a partial member of both mid and wide angles from its heading direction.

process by a human designer. The rtNEAT algorithm as implemented in NERO counters this problem by allowing the designer to interact during the training process, but this decreases the automation.

Background: Fuzzy Controllers in Agent Steering

The primary use of fuzzy controllers is to simplify an agent's understanding of its environment. Instead of classifying distances and angles in terms of meters and degrees, for example, we classify angles in human-like terms as being members of fuzzy sets[3]. For example, we use *narrow, mid* or *wide* and distance sets *near, medium* or *far* to define an agent's fuzzy sets. Creating such a classification is called *fuzzification* and is done by taking the real input values and evaluating their fuzzy equivalents using fuzzy set membership functions. An example of this procedure is illustrated in Figure 1. For a spatial representation of this classification see Figure 2, where in this case the agent at the bottom of the image is classifying the angle to an obstacle from its heading direction in fuzzy terms; *narrow*, *mid* and *wide*. Objects covered by more than one set are considered to be a partial member of both; thus the obstacle in the image is at a partially *mid* and partially *wide* fuzzy angle.

Once we have simplified perceptions into discrete forms thus, we can then perform some human-like reasoning by matching inputs with a fuzzy rule. The result of each rule is also a fuzzy set, but each one of these values is assigned a single midpoint value (also known as a singleton value). For example, a *sharp* turn might have a mid value of 2 rad s^{-1}, and a *very light* turn might have a mid value of 0.5 rad s^{-1}. One such fuzzy rule is:

*if a car is **near** and the angle to it is **narrow** then steer **sharply** away.*

Most intelligent agent systems would use a mathematical function to match inputs to outputs, but with a fuzzy system we can use a table to look up our rules very quickly. This process, which is known as a Fuzzy Inference Engine or Fuzzy Associative Memory Matrix matches each fuzzy distance and angle to fuzzy output values. As an example, the inference engine that we are using for change to steering in the route following component of a simulated car in our traffic simulation is given in Table 2, where output fuzzy steering adjustments are given for each fuzzy input distances and angle combination.

To cover cases where inputs are a partial member of multiple input sets because the input values fall inside overlapping set membership functions (such as the obstacle in Figure 2) we evaluate all of the rules, and using the fuzzy Zadeh union operator[10] we select the minimum of the input memberships for each rule, and then apply the Zadeh complement operator to select the maximum output value of each rule.

To obtain our final crisp output value we aggregate all of the fuzzy output sets together using a centre of mass

	Narrow	Mid	Wide
Near	Sharp	Medium	Very light
Medium	Medium	Very light	Zero
Far	Very light	Zero	Zero

Table 2. Rules for change to steering in the obstacle avoidance component of a simulated car.

DOI: 10.1002/cav

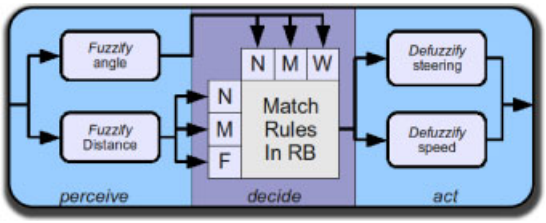


Figure 3. The component architecture has two fuzzy perception inputs (angle and distance), a 3 × 3 rule table which matches input fuzzy sets to output fuzzy sets, and two fuzzy motion defuzzifiers which convert fuzzy outputs into real speed and steering instructions.

function, weighted by the degree of membership of each fuzzy input set. The aggregation process is modeled by the equation:

$$y = \frac{m_0 * w_0 + m_1 * w_1 + \cdots + m_n * w_n}{w_0 + w_1 + \cdots + w_n} \quad (1)$$

Where y is our crisp output, m is a mid value for a fuzzy output set, and w is the weight value between 0 and 1; the degree of membership of a particular fuzzy output set. In our car steering example this crisp output will be a single steering adjustment value in radians. Where most if-then-else decision making systems produce stepped outputs as cases change, this aggregation procedure allows us to smoothly transition outputs between cases.

The basic fuzzy decision making architecture that we are using for all of our animated characters is illustrated in Figure 3. Our systems comprise two fuzzy decision making nodes:

- A reactive obstacle avoidance fuzzy controller
- A target seeking or route following fuzzy controller

Using both of our decision-making modules, environment elements (obstacles and destinations) are fuzzified into input values for angle and distance. Once these have been obtained we match the fuzzy distances near (N), medium (M), and far (F) to the fuzzy angles narrow (N), mid (M), and wide (W) in a rule table called a Fuzzy Associative Memory Matrix (FAMM). We have found that three sets for each input is sufficient (which gives us a 3 × 3 rule table), and that the larger, more detailed FAMMs are superfluous in this kind of application, but we are using a 5-set fuzzy output value for greater rule flexibility. Our FAMM contains the complete matching for every input, and provides an output fuzzy set for each rule, for both change in steering, and for desired speed. Therefore, with two fuzzy decision making modules that have 3 × 3 fuzzy inputs each, we have 18 combinations of inputs, and as each of these combinations is used for two fuzzy outputs then in total our fuzzy system requires 36 fuzzy rules.

Architecture of the GFS

In a fuzzy system all of the problem specific variables and parameters are grouped into what is called a fuzzy knowledge base (KB), which consists of two distinct parts:

- the data base (DB), which determines the size and shape of fuzzy set functions (used for fuzzification and defuzzification);
- the rule base (RB), which contains a list matching every possible combination of fuzzy inputs to a valid fuzzy output.

A GFS is designed to optimise either the scale and shape of set functions in the DB or to tune the rules in the RB, as simultaneous optimisation is conflicting[7]. In this architecture we have chosen to only concentrate on optimising the RB, and not the fuzzy DB as output fuzzy values can be treated as discrete numbers, and are therefore easy to increment or decrement during mutation. As DB optimisation is a more complex operation we will investigate scaling and modification to fuzzy set functions in future works.

Chromosome Structure

The first requirement for a GFS is to create a genetic structure representation (chromosome) for fuzzy rules. The genetic operators (selection, crossover and mutation) can then treat our fuzzy rules as if they were biological processes dealing with genetic code[11]. In our encoding method each rule is recorded as a three-digit cluster. The first digit represents a fuzzy input value for fuzzy input angle, the second for fuzzy input distance, and the third is a fuzzy output value. Both input and output values are expressed as integers rather than a name; e.g. *zero* is expressed as 0, and a *very light* turn as a 1. This makes manipulation very easy, as the entire rule set of 36 rules is simply a string of digits, whilst retaining some human readability. To aid in identifying individuals later we have added a header to the front of the code, which indicates which generation and batch of runs the individual came from.

An example individual's chromosome (individual number 0 (I:0) of generation 1 (G:1)) is shown in Table 3, with rules separated by spaces for clarity.Each 3-digit

DOI: 10.1002/cav

G:1	I:0							
021	122	223	011	113	214	001	014	025
225	124	023	214	113	012	201	102	001
000	011	022	100	112	123	200	213	224
005	014	022	104	113	121	202	211	220

Table 3. A sample chromosome used by our genetic algorithm. Note that we are not using a binary encoding, but rather our own custom discrete encoding where every third digit is a rule output that can be modified by the GA.

cluster represents a fuzzy inference rule. Each set of nine rules is a complete mapping for a fuzzy output. Our genetic algorithm operates by changing the value of the third digit (the rule output) of the 3-digit clusters using our genetic algorithm.

Simulation Component Design

One drawback of existing genetic algorithm models is that they require extensive changes to simulation infrastructures when implemented directly in code. We have therefore designed our architecture with a view to making minimal intrusions on the target simulation. We suggest separating the genetic algorithm from the simulation architecture with only a small simulation plug-in used to interface with an external genetic algorithm. This approach separates the evaluation and breeding functions into separate programmes, leaving only the performance logging and rule distribution functions running as part of the simulation. This means that the whole architecture has only a very small dependency on the target simulation's programming language and implementation.

Our simulation plug-in is illustrated in Figure 4. Referring to the figure, a 'Run Manager' (RM) component loads all available chromosomes into the simulation on programme execution. During run time it distributes these to the agents, and once all runs are completed it looks for a new generation of rule sets to load and distribute on the fly. The count of runs completed for each individual are also recorded so that if the simulation is stopped evaluation can resume from the same point. As indicated in the figure, agents log their performance (fitness function components) at regular mileage intervals. These can then be compiled at a later stage for fitness evaluation. This method allows continuous evaluation that is interruptible, which makes it ideal for computer games where a game might not last for an entire evaluation cycle. Because the runs are logged incrementally the training can be halted and resumed at a later stage with very little loss of training time. We found that using flat files as input and output makes negligible impact on simulation performance. A network loop could easily be used in place of file I/O to reduce processing time, or to distribute training across multiple machines.

Figure 4. Simulation plug-in architecture. The 'Run Manager' (RM) intermittently loads new rules from the gene pool. The trucks represent the agents, which append their performance to logs at small mileage intervals.

Breeding Pipeline

The breeding of new generations based on genetic operators is handled in an external pipeline, as illustrated in Figure 5. This pipeline is independent of the implementation of the target simulation. The CPU demands of the whole system are therefore very low, as the breeding pipeline does not need to remain in-synch with the updates of the simulation. It also readily takes advantage of any available multi core hardware as it occupies a separate operating system process. This separate tool

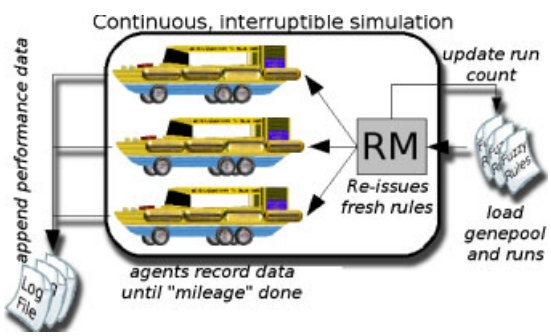


Figure 5. Vehicles under the control of our GFS move through the obstacle-strewn test environment.

DOI: 10.1002/cav

chain reads in all of the logged evaluation results output by the simulation and compiles them into complete runs. The runs are then evaluated with a fitness function, and when all of the runs for an individual are compiled that individual is ranked according to its fitness score. The gene pool in our architecture retains the best individuals from earlier generations, and new individuals are ranked against these. As for genetic operators, our selection operator takes the best p parent individuals from the top of the fitness rank and these are used for breeding the new generation. For the sake of broad applicability (so that the algorithm functions consistently even when we experiment with very small population sizes) we have only used a value of $p = 2$ in our experiments so far. The parent chromosomes are then crossed over to produce a population size of n children. The crossover mechanism moves along the chromosome one rule at a time and has a set $\frac{1}{p}$ probability of choosing either parent's output fuzzy set for each rule. Each rule has an r ('radiation level') probability of being mutated by 0 to m fuzzy output levels. We have attempted to exhaustively explore the complete range each of these genetic algorithm variables.

Fitness Function

The fitness function that we have used for our initial experiments combines two heuristics; firstly a 'crash rating' - the mean penetration or intersection of the character with obstacles during the run in meters, and secondly a heuristic representing the mean speed of the vehicle out of its maximum desired speed given current conditions. Each of the equation components are multiplied by a weighting factor which can be used to reward obstacle avoidance behaviour or expediency to a higher or lower degree. For our initial experiments we have set these weights to 1. Our perfect fitness is a score of 0 so we aim to minimise the fitness. Our fitness function is given in Equation 2 as the fitness awarded to an individual i:

$$\text{fitness}_i = \bar{c}^2 * w_c + \left(1 - \frac{\bar{v}}{v_{\max}}\right) * w_v \qquad (2)$$

Where $\bar{c}$ represents our crash rating and v represents the speed of the vehicle. The weights for the crash rating heuristic and speed heuristic are represented by w_c and w_v, respectively. We represent our crash heuristic in a squared form, as distance comparisons in most simulations are usually squared to avoid use of the CPU expensive square root operator. In the speed heuristic we are taking the mean speed of the vehicle over the maximum desirable speed as given by the environment limitations or vehicle's top speed, i.e. the ideal speed, $v_{\max}$. For an example fitness evaluation, if we have a crash rating of 0.4 m^2 and an average speed of 20 out of an ideal 30 k h^{-1}, then we award the individual a fitness of 0.7333.

Experiments and Results

In order to establish a range of guideline parameters for implementation of the GFS, we designed a range of experiments to exhaustively explore the larger part of the parameter space of the genetic algorithm. All of our experiments were conducted in a 3D graphical simulation, where vehicles traversed an obstacle strewn environment similar to that found in many modern computer games. The vehicles were driven by a simple physics simulation of braking, acceleration, turning friction, hill climbing and descent. Thus, our fuzzy controlled agents had to perceive and operate in a relatively complex environment using a very simple fuzzy system. We found that we were able to quite comfortably distribute our training over 40 agents simultaneously on an Intel Core2 Quad CPU 2.40 GHz desktop machine with a NVIDIA GeForce 8800 GTX card without adversely affecting the frame rate of the simulation or overly cluttering our test environment with moving vehicles.

Experiment Design

At the start of each experiment agent-controlled vehicles were randomly scattered around the landscape, and continuously given pseudo-random way points to move to. To evaluate each agent we used our fitness function as introduced in the previous section. As depicted in Figure 6 the vehicles had to avoid a large variety of shapes and sizes of static and dynamic obstacle, including long walls, and other moving vehicles, whilst being forced to move through steep craters, hills and flat areas. The sim-

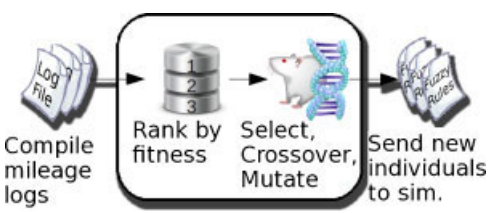


Figure 6. Operation of the breeding tool chain. Firstly fitness scores are created from agents' 'mileage' logs. Individuals are ranked in order of fitness. Genetic operators are applied to create a new generation. This is sent to the simulation.

DOI: 10.1002/cav

ulation was not restarted between evaluation runs, but rather the new runs were awarded randomly to available characters in a continuous fashion. In all of the experiments we started all of the agents with the same default hand crafted fuzzy RB, which was capable of motion but could be improved, i.e. we were using a 'rule base tuning' approach. Our first experiment was designed to established how many runs were required to evaluate each individual reliably, a baseline which would would use in further experiments. Subsequent experiments evaluated our genetic algorithm's variables: mutation range (m), where each fuzzy output value to be mutated would be adjusted $\pm m$ levels, probability of gene mutation (r) and population size (n).

Finding A Good Evaluation Limit

Because we want to continuously evaluate individuals during run time of a simulation, we need to know how many runs to compute before we could stop evaluating each individual. No evolution took place in this experiment, and all agents used the same fuzzy KB. We were interested in how many simulation runs of this duration needed to be accumulated before a reliable estimation of fitness was found. Our fitness evaluation converged after 25–30 runs of this type, which gives us a baseline for the minimum number of runs that we need to record in similar simulations before each individual can be evaluated.

Probability of Mutation Parameter

This experiment was designed to find the ideal probability of mutation r for each gene in the chromosome for each new individual created. All of the other genetic algorithm variables were kept constant, with mutation level $m = 3$, and population size $n = 4$. As we were evolving the RB during a continuous simulation we included a control case $r = 0$ to observe how the changing condition of the simulation affected the fitness evaluation itself. We ran our experiment at several different levels of mutation probability, and the results of this are presented in Figure 7. The control case showed that our simulation stabilised after three generations as the characters tended to spread themselves further apart. At this point we have a base fitness level of 0.86. The extreme cases $r = 10$ and $r = 80$ were slowest learners, with all results tending to a local minimum around a fitness of 0.3. Only $r = 20$ passed this minimum by 45 generations. Overall the results indicate that the genetic algorithm significantly improved fitness over the control case. The

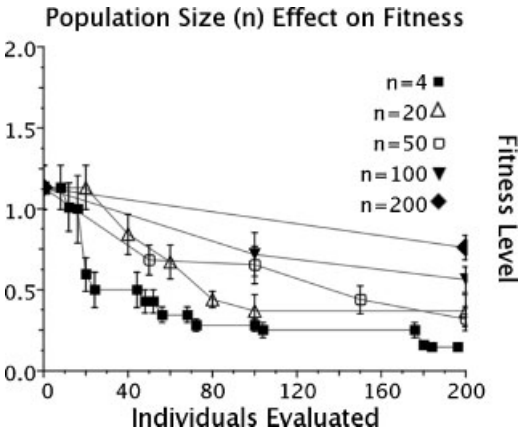


Figure 7. The graph shows us the effect that the probability that each gene is mutation (r) has on fitness. The results are clustered together, indicating that (r) has little effect on fitness. A control plot with no mutation is also shown.

difference in the rate of improvement to fitness made by probability of mutation is marginal, with best results at $r = 20$.

Level of Gene Mutation Parameter

Our next experiment was designed to explore the parameter space of the mutation level m. We assumed that changing our m variable would make a bigger difference to fitness than other parameters, and that higher mutation levels would pass minima encountered at lower levels. Our experiment conditions were the same as in our earlier experiment, but with r held constant at 20%. The results of the experiment are presented in Figure 8.

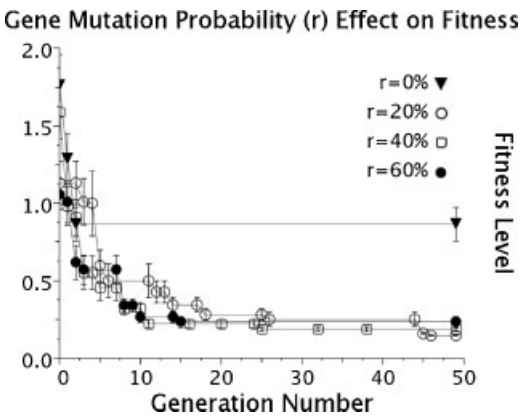


Figure 8. Shown here are the efficiencies of population sizes (n), where we find that lower populations improve fitness more quickly that larger populations.

DOI: 10.1002/cav

Contrary to our assumption, we found that lower mutation levels of 1–2 were adequate for rapidly tuning the RB. All of the different levels were trapped in the familiar 0.3 local minimum of the previous experiment, with only $m = 3$ passing this within 50 generations. An important observation of higher mutation levels is that they produce more agents that become completely stuck and are unusable. Whilst these RBs can be quickly identified and culled (given a very high fitness score) this behaviour would be highly undesirable in an interactive computer game. For run time training where consistently good motion is desired, a mutation level set relatively low (between 1 and 3 in our GFS) is ideal.

Population Size Parameter

Our last experiment was designed to explore the population size (n) parameter space, specifically to find the population size per generation that allows us to improve our fitness score most rapidly. We assumed that larger population sizes would improve our rate of fitness improvement beyond the very low default of $n = 4$. Our results are presented in Figure 9, where we can see a clear indication that bigger population sizes evolve more slowly. Of course, where the number of agents being trained in parallel can train more than an entire generation of runs simultaneously then we can increase the population size to scale. As this is a unique observation we intend to run this experiment with a broader range of agents to investigate if this result holds true across all conditions.

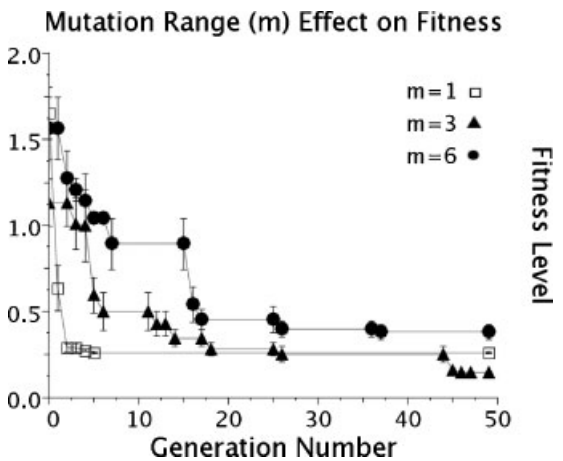


Figure 9. This graph shows mutation range (m), with the best result at ±3 fuzzy sets.

Genetic parameter	Guideline value
Minimum runs	30
Population size n	4
Mutation level m	±3
Mutation chance r	20%

Table 4. Baseline genetic parameters for GFS.

Conclusions and Future Work

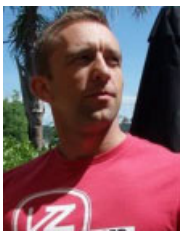
In this work we have found that our GFS architecture can improve the reactive navigation behaviour of 3D animated characters over our initial hand trained set, assuming that our fitness function is a valid measure of performance. We can see this in Figure 7 where all of the evolved results surpass the fitness of the control case. This suggests that the GFS is a useful tool for reducing the human calibration time requirements of agents using fuzzy controllers. We have also found that a GFS can operate dynamically, in a real time simulation, and evolve with a small population size. This offers an alternative to other evolutionary algorithms for games, and suggests that it might be capable of tuning intelligent agents that are based on much more complex fuzzy systems. In addition to presenting our complete GFS architecture, we have also identified a range of developer baseline parameters for use. These are presented in Table 4. Our next steps will be to benchmark performance of the GFS compared to other agent steering systems, and to investigate application of the GFS to a broader range of animated character types, and to investigate the capability of the GFS to adapt to changing conditions in real-time.

References

1. Dougherty M, Fox K, Cullip M, Boero M. Technological advances that impact on microsimulation modelling. *Transport Reviews* 2000; **20**(2): 145–171.
2. Won J, Lee S, Lee S, Kim TH. Establishment of car following theory based on fuzzy-based sensitivity parameters. *Advances in Multimedia Modeling* 2006; **4352/2006**: 613–619. DOI: 10.1007/978-3-540-69429-8
3. Passino KM, Yurkovich S. *Fuzzy Control*. Addison Wesley Longman: Menlo Park, CA, 1998.
4. Allen BF, Faloutsos P. Evolved controllers for simulated locomotion. In *2nd International Workshop, Motion in Games*, Zeist, The Netherlands, 2009.

5. Miikkulainen R. Creating intelligent agents in games. *The Bridge* 2006; **36**(4): 5–13.
6. Stanley KO, Bryant BD, Miikkulainen R. Real-time evolution of neural networks in the NERO video game. In *The 21st National Conference on Artificial Intelligence*, Boston, Massachusetts, 2006.
7. Córdon O, Gomide F, Herrera F, Hoffmann F, Magdalena L. Ten years of genetic fuzzy systems: current framework and new trends. *Fuzzy Sets and Systems, Elsevier* 2004; **141**(1): 5–31. DOI: 10.1016/S0165-0114(03)00111-8
8. Mohammadian M, Stonier RJ. *Fuzzy Logic and Genetic Algorithms for Intelligent Control and Obstacle Avoidance*, chapter 2.6, IOS Press: Amsterdam, The Netherlands, 1994, 149–156.
9. Neves R, Netto ML. Evolutionary search for optimization of fuzzy logic controllers. In *1st International Conference on Fuzzy Systems and Knowledge Discovery*, Singapore, 2002.
10. Bellman RE, Zadeh LA. Decision-making in a fuzzy environment. *Management Science* 1970; **17**(4): B141–B164.
11. Herrera F, Lozano M, Verdegay JL. Tuning fuzzy controllers by genetic algorithms. *International Journal of Approximate Reasoning* 1995; **12**: 299–315.

Authors' biographies:



Anton Gerdelan is completing a PhD in computer science at Massey University in Albany, New Zealand. As part of his research, Anton is working at the Graphics, Vision, and Visualisation group (GV2) at Trinity College Dublin in Ireland. Anton received a BE/Hons in Software Engineering from Massey University in 2007, and was as a researcher for New Zealand's national grid computing project (BeSTGRID) 2007–2009. Research interests include vehicle navigation algorithms, machine learning, 3D graphics and complex simulations.

Carol O'Sullivan leads the Graphics, Vision and Visualisation group (GV2) in Trinity College Dublin. After receiving a BA in Mathematics from Trinity College in 1988, she worked for several years as a software engineer in industry (mainly in Germany), followed by a Masters degree from Dublin City University in 1996 and a PhD in computer graphics from TCD in 1999. Her research interests include perception, animation, virtual humans and crowds. She was elected a Fellow of Trinity College Dublin and of Eurographics in 2003 and 2007 respectively. She has been a member of many IPCs, including the Eurographics and SIGGRAPH papers committees, and has published over 100 peer-reviewed papers. She has organised and co-chaired several conferences and workshops, including Eurographics'05 in Dublin, the SIGGRAPH/EG Symposium on Computer Animation 2006 and the SIGGRAPH/EG Campfire on Perceptually Adaptive Graphics 2001. She is the programme co-chair of the SIGGRAPH Symposium on Applied Perception in Graphics and Visualization 2009, is the co-Editor in Chief of ACM Transactions on Applied Perception and an editorial board member of IEEE Computer Graphics & Applications and Graphical Models.

DOI: 10.1002/cav