Off-Line Evolution of Behaviour for Autonomous Agents in Real-Time Computer Games

Eike Falk Anderson

The National Centre for Computer Animation
Bournemouth University
Talbot Campus, Fern Barrow, Poole
Dorset BH12 5BB, UK

Abstract. This paper describes and analyses a series of experiments intended to evolve a player for a variation of the classic arcade game Asteroids™ using steady state genetic programming. The player's behaviour is defined using a LISP like scripting language. While the game interprets scripts in real-time, such scripts are evolved off-line by a second program which simulates the real-time application. This method is used, as on-line evolution of the players would be too time consuming. A successful player needs to satisfy multiple conflicting objectives. This problem is addressed by the use of an automatically defined function (ADF) for each of these objectives in combination with task specific fitness functions. The overall fitness of evolved scripts is evaluated by a conventional fitness function. In addition to that, each of the ADFs is evaluated with a separate fitness function, tailored specifically to the objective that needs to be satisfied by that ADF.

1 Introduction

In recent years the level of realism in computer games has risen dramatically. While the quality of real-time graphics - mainly due to advances in computer graphics hardware - certainly played the major part in this development, one should not forget that artificial intelligence is another important factor for the attainment of realism in games. The playability of computer games is often measured by the quality of the behaviour of intelligent agents in the game environment. If this behaviour appears natural and human-like, the agent seems to be more life-like and real. There is an obvious solution to satisfy the need for the creation of autonomous agents which seem alive. Genetic programming (GP) produces algorithms by using a process that parallels evolution through natural selection, i.e. a simulation of life. GP has so far been applied to a number of different computer game scenarios. Among these are classic videogames like Pac Man® ([Koza 1994]) or Tetris® ([Siegel and Chaffee 1996]). These experiments evolved game playing behaviour in a modified game environment. Most of these game versions are round-based, i.e. the computation of an action in the game is performed while the game is paused. Gameplay resumes only after those computations have finished, and only until the calculated actions have been executed. This is in contrast to real-time games in which all actions have to be calculated "on the fly". One of the few attempts to apply GP to a real-time game (RoboCup Soccer) is

J.J. Merelo Guervós et al. (Eds.): PPSN VII, LNCS 2439, pp. 689-699, 2002

documented in [Luke et al 1997] and [Luke 1998]. The methods employed for that experiment bear some similarities to the experiments described here. The players are evolved in a modified environment, and not on-line in the game itself. Only the evolved players are used in the real-time application. They also used different algorithm trees for different player objectives, which is similar to the experiments using syntactic constraint to achieve the generation of a separate gene pool for each of the player's objectives as described by [Reynolds 1994]. This approach again resembles the concept of Automatically Defined Functions as described by [Koza 1992]. The experiments described in this paper merge some of these techniques and extend the use of ADFs by calculating a fitness for each of the ADFs in addition to the fitness value calculated for the performance of each individual of the population. The classic arcade game Asteroids is used here as a test case. Asteroids is based on attack and evasion which is a concept that is common to most action oriented video games. The evolution of a successful player could therefore be seen as a proof of concept. It should be possible to transfer a solution of the Asteroids problem to more complex game scenarios by adapting the player's interface with the game to that of another computer game.

In the Asteroids game a "space ship" (the player) has to avoid colliding with a number of "asteroids" to prevent its destruction. At the same time it has to destroy the "asteroids" to win and progress to the next level of the game. For practical reasons a number of modifications to the original game have been made for the version used in the experiments that are described here (Figure 1).

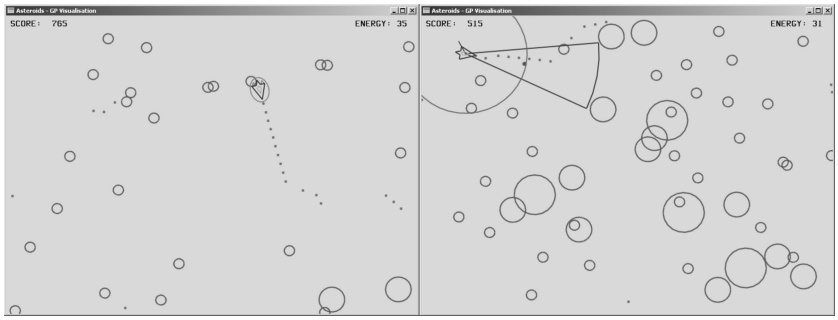

Figure 1. Screenshots of the Asteroids implementation. The left screenshot shows the player's spacecraft avoiding collision by raising its shields. The right screenshot shows the player's sensor information consisting of asteroid proximity and the player's field of view

In the original arcade version of the Asteroids game the player's spacecraft flies through a two-dimensional field of moving asteroids. The player has a pre-defined number of lives. A collision with an asteroid destroys the spacecraft. The player can shoot at asteroids. If hit, a large asteroid will break up into two medium asteroids which in turn can each be split into two small asteroids. Shooting down an asteroid increases the player's score. The player's only means of defence is to jump into "hyperspace", which removes the player's spacecraft from its current location and randomly repositions it on the screen. The game ends when the player has lost all of

his lives. The aim of the game is to stay alive as long as possible and to gain the highest score during that period.

In the implementation used here, the player only has one life to limit the execution time of the game. In addition to that the "hyperspace" escape function has been replaced by a "protective shield", as the results of this feature are unpredictable, i.e. the player's spacecraft could be saved, but just as well be destroyed when using a "hyperspace" jump. The defence "shield" however has a predictable result: it protects the player from destruction during collisions with asteroids by granting invulnerability while it is active. The player is given an initial level of energy. Using shields or firing the gun will drain the player's energy which is replenished over time. The player is therefore encouraged not to waste energy by unnecessary usage of the shields and the gun. Finally a more complex scoring system which gives a more precise reflection of the player's performance is used in this version of the game.

2 Implementation of the Off-Line Game Environment

The environment in which the game is played is a two-dimensional field of 80 x 60 units which continually wraps around. The player starts in the centre. Asteroids are positioned randomly around the player. As a single game of Asteroids can take a few minutes to complete, it is self-evident that trying to evolve the players on-line would take far too long. The obvious solution is to remove the code generation from the real-time application and to evolve the player scripts off-line in a second application which simulates the real-time game. The real-time game itself only interprets and executes the programs that have been evolved by the off-line game simulation. The simulation program contains the GP system that evolves the player scripts. It also contains a copy of the game code that has been stripped of all graphics functions, which is used for evaluating the evolved players' fitness. Without displaying anything on screen, the simulation can run at much greater speed and players evolve much quicker than would be possible in a real-time environment. Whereas it would take several minutes for a single individual of the player population to play a single game in an on-line evolution, using this dual approach, a series of games can be played in a matter of seconds.

To evolve successful players for the game, the code generation program needs to simulate the real-time application as closely as possible. Discrepancies in the frame-rate of the real-time application result in a variable animation step-size for the objects (player, asteroids and bullets). This is simulated in the code generation program by combining the average frame-rate of the real-time game with a random value.

The player interfaces with the game through a LISP like scripting language which implements a number of sensors and controls. In this problem the controls, which are identical to a human player's controls for the space ship, are used as output of the evolved program. The sensors, which reflect the current state of the game, are used as input to the evolved program. The script which controls the autonomous agent is created through evolution, based on the agent's proficiency at playing the game.

3 GP Architecture

The variation of GP used in this project is "strongly typed" GP as introduced by [Montana 1995], which allows for the use of different datatypes. There are two datatypes, one for Boolean values which can be either TRUE or FALSE, and a void datatype which is used for procedures that do not return any data. Three constants (two Boolean: TRUE, FALSE and one void: void) are defined for use in the control structures. The player's sensors and controls make up the terminal set of the GP functions while the control structures are the non-terminal set of functions.

Table 1.

Function	Returns	Description
(targetAhead)	Boolean	TRUE if an asteroid is within the player's field of view, else FALSE
(targetLocked)	Boolean	TRUE if an asteroid is in the player's direct line of fire, else FALSE
(proximityAlert)	Boolean	TRUE if an asteroid is in the player's proximity (within 12 units from the player), else FALSE
(impactAlert)	Boolean	TRUE if the player is about to collide with an asteroid (asteroid is within 3 units from the player), else FALSE
(hasEnergy)	Boolean	TRUE if the player has energy left, else FALSE
(plentyEnergy)	Boolean	TRUE if the player has enough energy for firing more than four shots, else FALSE
(hasShields)	Boolean	TRUE if the player's shields are raised, else FALSE
(lookingAhead)	Boolean	TRUE if the player's direction of movement is identical to the player's heading, else FALSE
(isMoving)	Boolean	TRUE if the player is moving, else FALSE
(accelerating)	Boolean	TRUE if the player has active thrusters, else FALSE
(isTurning)	Boolean	TRUE if the player is turning, else FALSE

3.1 GP Function Set

The sensors of the player's space ship are implemented as a set of Boolean functions. The available sensor information consists of:

- The level of the player's energy.
- The state of the player's movement.
- Approximate positions of targets (asteroids) in relation to the player's position.

The controls for the space ship are implemented as a set of procedures which enable the player to switch its current states. The available instructions are:

- Turning (left, right, not).
- Acceleration (on, off), deceleration (automatically reset for each frame).
- Shields (on, off).
- Firing a single bullet.

A more detailed description of the syntax of these functions and procedures can be seen in Table 1 and Table 2.

Table 2.

Function	Returns	Description
(setThrust)	void	activates the player's thrusters
(noThrust)	void	deactivates the player's thrusters
(decelerate)	void	reduces the player's speed
(setShields)	void	raises the player's shields
(noShields)	void	lowers the player's shields
(rightTurn)/(leftTurn)	void	sets the player to turn clockwise/anti-clockwise
(noTurn)	void	sets the player to stop turning
(fire)	void	fires a single bullet

Player scripts that use this interface are generated using simple Boolean operators (AND, OR, XOR and NOT) which are implemented as non-terminal functions and a small set of control structures which consists of:

- Dyadic selection.
- Comparison.
- Sequence.

See Table 3 for a detailed description of the control structure syntax.

Table 3.

Function	Returns	Description
(if_true b v1 v2)	void	if the Boolean function b returns TRUE the void procedure v1 is executed else if b returns FALSE the void procedure v2 is executed
(if_equal b1 b2 v)	void	if the return values of the Boolean functions b1 and b2 are identical the void procedure v is executed
(sequence v1 v2)	void	executes the two void functions v1 and v2 one after the other

The goal of the game Asteroids is to maximise the player's score. In this implementation of the game the best way to achieve this is to destroy all asteroids as quickly as possible. A precondition for the destruction of all asteroids is the player's survival. This leads to the identification of three distinctive behaviours:

- Aggression - Destroying a target which is in the player's range and line of fire.
- Target Acquisition - Seeking out and finding targets in the shortest possible time.
- Defence - Avoiding collisions with asteroids.

The use of segregated branches of the parse tree for achieving multiple objectives as described in [Reynolds 1994] was the inspiration for the use of ADFs to find a solution that successfully completes the three conflicting objectives of the Asteroids game. This is done by associating each of the objectives with a different ADF. To

ensure that each of these three ADFs specialises in satisfying a different objective, the fitness evaluation of individual players is distributed using task specific fitness functions. The GP system uses a separate fitness function for each ADF which evaluates the fitness of that ADF for a specific task. Each of these fitness functions runs a subset of the game simulation which ignores all factors that are not deemed necessary for accomplishing that particular ADF's objective. The error values that are returned by these fitness functions are accumulated and added to the error value of the overall fitness function which evaluates the performance of the player. All ADFs are terminal functions that return void and take no parameters.

The ADFs can contain all of the available functions, procedures and control structures. In the result-producing branch (RPB) which contains the main program however only the control structures (see Table 3) and the three ADFs (Aggression, Defence, Target Acquisition) are available. It was necessary to impose this syntactic constraint, as early experiments showed that otherwise there was a chance of functions and procedures in the RPB cancelling out the results generated by the ADFs.

3.2 Fitness Evaluation

In the main fitness function which evaluates the RPB of each player, the player's fitness is determined using a progressive fitness measure which is similar to the one described by [Siegel and Chaffee 1996]. In that approach successful individuals of a population were re-evaluated in further test cases. Here however the progression is applied from within the fitness function itself. In the fitness function the player's performance is measured in a series of four games of increasing difficulty. Only successful players may proceed to the following game. The games are time limited to prevent infinite loops from occurring. The fitness cases for each game are "Survival", "Speed", "Marksmanship" (*2 levels*), "Aggression" (*2 levels*) and "Score".

After a series of games has been played further fitness cases are tested. These are "Success", "ADF Usage" (*3 cases*), "Execution Speed" (*4 cases*) and "Vitality" (*7 cases*).

The fitness function for the Defence ADF evaluates a special version of the game that runs over a set time and in which destroyed asteroids are constantly regenerated, so that the game never runs out of asteroids. The fitness cases for this ADF are "Energy Conservation" (*2 cases*), "Evasive Manoeuvres" (*2 cases*), "Distance Checks" (*2 cases*) and "Survival".

The fitness function for the Aggression ADF evaluates a special version of the game which runs over a set time. Destroyed asteroids are constantly regenerated, so that the game never runs out of asteroids. The player is automatically moved across the playing area, so it can concentrate on its shooting skills and collisions between the player and asteroids are disabled. The fitness cases for this ADF are "Energy Conservation" (*3 cases*), "Marksmanship" (*4 cases*), "Use of Guns" (*2 cases*) and "Kill Rate".

The fitness function for the Target Acquisition ADF evaluates a special version of the game in which the player automatically fights and defends against asteroids. The fitness cases for this ADF are "Use of Sensors" (*3 cases*), "Steering" (*4 cases*) and "Movement" (*2 cases*).

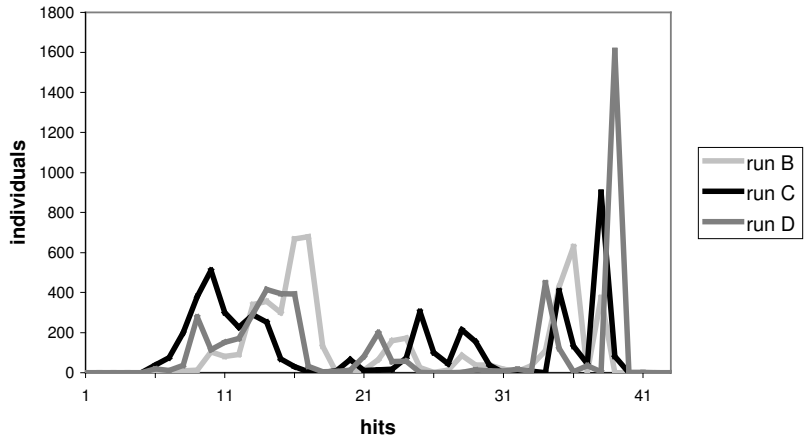

Figure 2. Conditions that have been satisfied (*hits*) for the individuals of runs B, C and D

4 Experiments and Results

In the experiments performed, a large number of evolution runs were carried out, many of which were only used to verify the effectiveness of refinements of the fitness functions. Six of the runs are of particular interest and are discussed below in some detail. However it has to be said that the experiments are still continuing, so this project is still very much work in progress. One of the earlier experiments evolved a population of 5000 individuals over 2 generations, using higher level primitives instead of ADFs (run A). In four of the experiments described here a population of 5000 individuals was evolved over 10 generations (runs B, C, D and E). A single control run with slightly relaxed fitness conditions was performed with a population of 1500 individuals over 10 generations (run F). All experiments used the same random seed for the generation of the initial population and the possibility of a 6% mutation was introduced to counteract a possible loss of diversity in the player's gene pool.

4.1 Early Experiments without ADFs

The earliest test runs of the code generation program did not use ADFs but were instead set to produce programs that consisted of a single tree. The fitness function used for these experiments was almost identical to the one used for later experiments, except for the fact that it also contained fitness cases that were later moved into the task-specific fitness functions for the ADFs. The player scripts produced by these early experiments were weak and hardly ever survived against more than a single asteroid. To find out if the reason for these poor results was that the function set for the player interface was too small, a set of three higher level primitives (seek, autoprotect,

fireAtWill), reflecting the three different objectives of the game and designed to automatically play the game were created as terminal functions. These higher level primitives themselves were just groups of some of the lower level primitives. A hand-coded player script was created by just combining these three higher level primitives with each other using the "sequence" non-terminal function (Table 3). This script proved to be a successful player which never lost a game. If the higher level primitives are used in the code generation program, scripts evolve which are very similar to this initial player script. The following script containing the higher level primitives was generated in run A after two generations (unedited except for addition of comments):

```
(sequence
   seek    ; move until target is in range, then stop
   (sequence
     (sequence
       fireAtWill  ; if a target is in the line of fire, shoot
       autoprotect) ; if collision is immanent raise shields
     autoprotect)) ; if collision is immanent raise shields
=
```

Although possibly not the optimal player, this evolved player employs similar strategies to those used by many human players:

- It keeps turning in one direction until a target is in its line of fire, then stops.
- If a target is in its line of fire, it shoots at the target.
- If no target is in range, it moves forward until a target is in range.

To test if these results could be replicated by just using the low-level primitives, a hand-coded program was created which used three ADFs, each of which emulated one of the higher level primitives.

4.2 Experiments using ADFs with Separate Fitness Functions

When confronted with the problem of trying to achieve a result that equals the success of the hand-coded program, the question arose how to force the ADFs to each tackle a different objective. The approach that was adopted was to create a separate fitness function for each of the three ADFs. Figure 2 shows the fitness distribution within the populations after each of the runs for this experiment. Although only observed over a series of three runs of ten generations each (runs B, C and D), a convergence of the gene pools of the ADFs, reflecting the make-up of each respective fitness function, becomes apparent. Resulting player scripts show the use of a similar strategy to that of the hand-coded player discussed earlier. It should be noted however that there seems to be a larger loss of diversity in the player's gene pool when this method is used, than can be observed in evolution runs in which the ADFs are not evaluated by separate fitness functions. This might be a serious problem that needs to be addressed in future experiments.

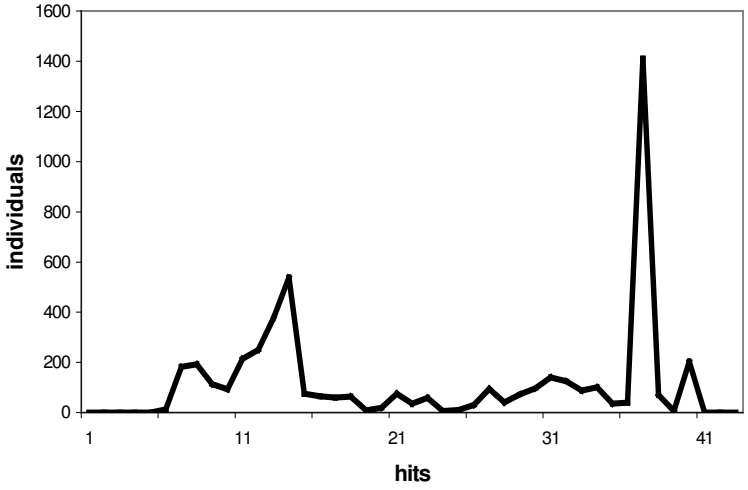

Figure 3. Conditions that have been satisfied (*hits*) for the individuals of control run E

4.3 Control Runs

A single control run E, using the same parameters and fitness function, however without the additional fitness functions for the ADFs resulted in a player with a similar behaviour and capability to the players evolved using separate fitness functions for the ADFs in runs B, C and D. The overall performance of individuals of run E after ten generations was better than that for players evolved in runs B and C. The fitness distribution in the population of this run is shown in Figure 3. The players in run E needed more generations to evolve to a stage where their performance matched that of the players generated by the earlier runs with separate ADF fitness functions. Although the best individual of this control run displayed a similar behaviour to the best individuals of the earlier runs and completed the same objectives, there is no visible task specialisation in the ADFs.

In another control run F the functions for checking the level of energy, as well as all energy consumption by the players were removed from the program. If energy management is disabled, the resulting players seem to be more successful and are much more likely to survive. A single run over ten generations with a population of 1500 individuals evolved two different kinds of successful player. However the evolved strategies appear a lot less intelligent than those observed in the other runs:

One of the players in run F continuously spun its space ship around while firing its gun. This is an obvious solution, as with unlimited ammunition, there is no pressure regarding accuracy, and if the gun continuously fires in all directions, the chance of any asteroid getting close enough to the player to destroy it without being destroyed itself is minute. The strategy adopted by the other player was less obvious but similarly effective. It raised its shields and flew in a straight line, continuously firing its

gun, creating some sort of impenetrable barrier in front of the space ship which destroyed everything in its path. This was only possible with unlimited ammunition and shields.

This illustrates that the pressure created through the player's need for energy management aids the evolution of a more intelligent player. It also shows, that in addition to the three objectives that have been identified, there may be further objectives which need to be completed by a player to succeed, that need to be addressed separately.

5 Conclusion and Future Research

Although these experiments are still work in progress and especially considering the fact that players evolved with ADFs without separate fitness functions do not seem to be any less successful, than those evolved with separate fitness evaluations for each ADF, the latter method looks promising as a possible solution to addressing problems with multiple objectives. The experiments show that this method at least accelerates the evolution of reasonably successful players. However this needs to be verified in further experiments and test runs over more than ten generations each. Furthermore the problem of multiobjective optimisation as described in [Goldberg 1989] needs to be addressed in conjunction with the use of separate fitness functions.

Using GP as a tool for the controlled evolution of autonomous agents for computer games seems an appropriate method for the design of natural agent behaviour. The experiments have shown that GP can be used to evolve the behaviour of an intelligent agent for real-time computer games. The next step will be to improve the agents' intelligence by refining the original problem. Coevolution techniques as used by [Sims 1994], [Reynolds 1994] that use competition between individuals to exert additional evolutionary pressure are a possible solution to this problem, which needs to be explored. Another approach might be to only use GP to evolve the agent's instincts, while other methods could be used to create a higher level intelligence. This might eventually lead to the generation of more complex and realistic agent behaviour than has been achieved so far.

6 Acknowledgements

The author would like to thank Anargyros Sarafopoulos for inspiration, support and the permission to use his GP system for this work. His ideas and suggestions contributed significantly to this project. Additional thanks go to Prof. Peter Comninos for encouragement and help in the preparation of this paper.

References

[Goldberg 1989] Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Massachusetts

[Koza 1992] Koza, J. R. (1992), *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Massachusetts

[Koza 1994] Koza, J. R. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, Massachusetts

[Luke et al 1997] Luke, S. (1997), Hohn, C., Farris, J., Jackson, G. Hendler, J., *Co-Evolving Soccer Softbot Team Coordination with Genetic Programming*, Proceedings of the RoboCup-97 Workshop at the 15th International Joint Conference on Artificial Intelligence, IJCAI

[Luke 1998] Luke, S. (1998), *Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97*, Genetic Programming 1998: Proceedings of the Third Annual Conference, Morgan Kaufmann

[Montana 1995] Montana, D. J., *Strongly Typed Genetic Programming*, Evolutionary Computation, 3(2), 1995, pages 199-230

[Reynolds 1994] Reynolds, C. W. (1994), *Competition, Coevolution and the Game of Tag*, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, MIT Press, Cambridge, Massachusetts

[Siegel and Chaffee 1996] Siegel, E. V. (1996) and Chaffee, A. D., *Genetically Optimizing The Speed of Programs Evolved to Play Tetris*, Advances in Genetic Programming 2, MIT Press, Cambridge, Massachusetts

[Sims 1994] Sims, K. (1994), *Evolving 3D Morphology and Behavior by Competition*, Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems, MIT Press, Cambridge, Massachusetts