Evolving Third-Person Shooter Enemies to Optimize Player Satisfaction in Real-Time

Conference Paper · April 2012

DOI: 10.1007/978-3-642-29178-4_21

1 author:

Jose Font
Malmö University
53 PUBLICATIONS **447** CITATIONS

SEE PROFILE

Evolving Third-Person Shooter Enemies to Optimize Player Satisfaction in Real-Time

José M. Font

Departamento de Inteligencia Artificial, Universidad Politécnica de Madrid. Campus de Montegancedo, 28660, Boadilla del Monte, Spain. `jm.font@upm.es`

Abstract A grammar-guided genetic program is presented to automatically build and evolve populations of AI controlled enemies in a 2D third-person shooter called "Genes of War". This evolutionary system constantly adapts enemy behaviour, encoded by a multi-layered fuzzy control system, while the game is being played. Thus the enemy behaviour fits a target challenge level for the purpose of maximizing player satisfaction. Two different methods to calculate this challenge level are presented: "hardwired" that allows the desired difficulty level to be programed at every stage of the gameplay, and "adaptive" that automatically determines difficulty by analyzing several features extracted from the player's gameplay. Results show that the genetic program successfully adapts armies of ten enemies to different kinds of players and difficulty distributions.

Keywords: Evolutionary computation, fuzzy rule based system, grammar-guided genetic programming, player satisfaction.

1 Introduction

Game development is a complex and multidisciplinary task that involves character design, environment modeling, level planning, story writing, music composition and, finally, programming [1]. Procedural content generation (PCG) is a research field that studies the application of algorithmic methods from many areas, such as computational intelligence, computer graphics, modeling and discrete mathematics, to the automatic generation of game content. Having access to these tools helps game developers to reduce the design costs when creating games that include huge amounts of content [2].

Search-based PCG is a research area in which evolutionary computation techniques are used to automatically generate game content [3]. Some examples in this area include the on-line generation of weapons for the space shooter Galactic Arms Race based on player preferences [4], the off-line creation of tracks for racing games focusing on diversity and driving experience [5] or personalization to player driving style [6], and the level and game mechanics customization for platform games [7].

Evolution of AI controlled characters in games enhances the creation of intelligent behaviors that raise player interest during gameplay [8]. For example, the NERO video game involves players training non-player characters to learn to target tactical directives by using a neuro-evolution approach [9]. Genetic programming has been applied to the generation of challenging opponents in several competitive games [10,11,12].

Nowadays, shooters are one of the most popular as well as worldwide best-selling game genres [13]. A grammar-guided genetic programming (GGGP) system is proposed here to automatically generate and evolve enemy behavior in a 2D third-person shooter called "Genes of War". GGGP has been successfully applied to the generation of both symbolic and sub-symbolic self-adapting intelligent systems [14,15,16]. GGGP enhances search space exploration because of the usage of the grammatical crossover operator [17], which does not generate invalid individuals during the evolutionary process.

Enemies in Genes of War are controlled by a multi-layered fuzzy ruled-based system. A population of these systems is constantly evolved in real-time in order to maximize player satisfaction at each stage of the gameplay. Satisfaction is measured from an implicit perspective [18] by matching the challenge preferred by the player with the one offered by the game. Two methods are proposed to measure this challenge level: one "adaptive" that fits the enemy population to player skills, and one "hardwired" to match it with programmers preferences.

Results show that the proposed GGGP system adapts the enemy population to two kinds of players, one beginner and one experienced, as well as to two fixed challenge distributions. Modularity and flexibility of the proposed system make it suitable to be exported to other competitive game genres.

2 The Genes of War game

Genes of War is a 2D third-person shooter game with a top-down perspective (Figure 1). In Genes of War, a human player takes the role of a soldier that must defeat an unlimited supply of AI controlled enemies. The player controls the soldier with a simple point and click system, which allows the soldier to move along a fixed size map. The soldier can aim in 360 degrees around itself and right clicking fires the soldier's weapon, even when the soldier is moving. The player can also make the soldier crouch by pushing the left control button in the keypad. Crouching allows the soldier to dodge enemy attacks, but prevents it from moving.

Enemies are AI controlled robots that can move along the map, crouch, and shoot in 360 degrees as the soldier does. Unlike the soldier, enemies are not allowed to shoot while they are moving. The player kills enemies by shooting them until their health points are depleted. Analogously, enemies defeat the player by shooting the soldier until it has no health points left. Friendly fire between enemies does not affect health points. When the soldier or an enemy is killed, it is respawned in a fixed map location with full health points and the default weapon.

Both soldier and enemies start the game equipped with their basic weapons: a machine gun and a cannon respectively. They can upgrade them to more pow-

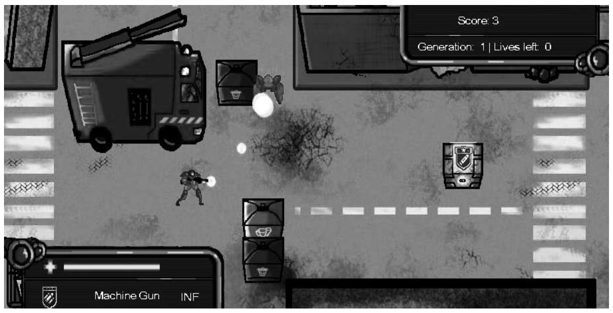


Figure 1: A snapshot from Genes of War.

erful weapons by picking up power-ups that are spawned over the game map. These powerful weapons are classified into short-range (shotgun, laser gun and flamethrower) and long-range weapons (rifle and missile launcher). Each has its own identifying power-up icon and unique values for the following characteristics: power, range, rate of fire, recoil, and ammunition. When the soldier runs out of ammunition, its weapon is automatically downgraded to the basic machine gun that has infinite ammunition. Unlike the soldier, enemies have unlimited ammunition for all weapons. There is a sixth power-up that heals a character (soldier or enemy) by increasing its health points. Power-ups can also be destroyed by firing at them, which causes an explosion that damages any character located close to it.

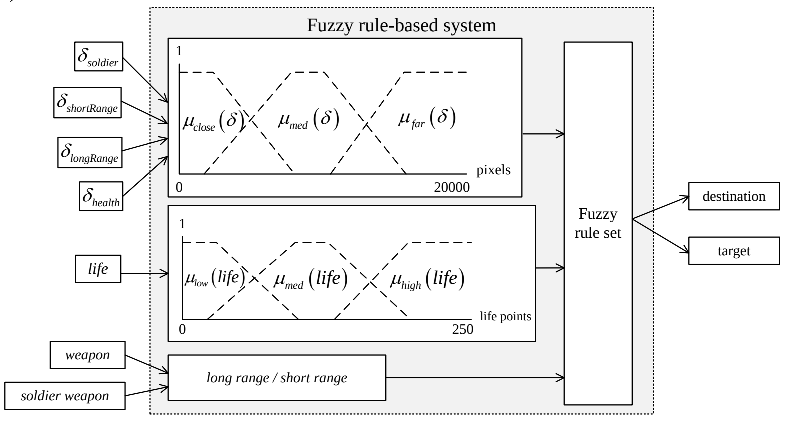
2.1 Multi-layered enemy control system

Enemies are fully controlled by the computer AI. Each enemy has a multi-layered control system composed by two modules. The upper module, called the strategy layer, holds a fuzzy rule-based system (FRBS) that processes a set of environmental variables whose values are extracted from the gameplay. As a result of its inference process, the FRBS determines the strategy that the enemy must follow. A strategy is composed by a destination, the location in the game map where the enemy is heading for, and a target, the location where the enemy aims at.

The lower module, called the action layer, holds a finite state machine composed by five states that represent the five basic actions that an enemy can do during gameplay. These states names are: "move", "fire", "cfire", "stop" and "die". The action layer takes both destination and target variables as input, performing the set of basic actions needed to follow strategy given by the upper layer.

The strategy layer is explained in detail in Figure 2a. The input of the FRBS is a set of seven environmental variables: $\delta_{soldier}$, $\delta_{shortRange}$, $\delta_{longRange}$, δ_{health}, *life*, *weapon* and *soldier weapon*. The first four of them make reference to the distance (in pixels) from the enemy to the soldier, to a short range weapon power-up, to a long range weapon power-up, and to a health power-up respectively.

a)



b)

Fuzzy rule set:

if $(\delta_{soldier}$ is *close*$)$ then $($target = *soldier.location*$)$

if $($*life* is very *low*$)$ then $($destination = *healthItem.location*$)$ AND $($target = *null*$)$

if $($*weapon* is *shortRange*$)$ then $($destination = *longRange.location*$)$

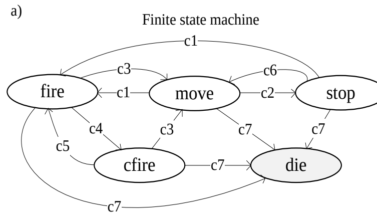
Figure 2: a) Description of the strategy layer in the multi-layered enemy control system. b) Sample fuzzy rule set.

The linguistic labels *close*, *med*, and *far* have been defined for these four input variables, making reference to the three fuzzy sets in which they can take values. The membership of each variable to the fuzzy sets represented by these labels is defined by the membership functions $\mu_{close}(\delta)$, $\mu_{med}(\delta)$, and $\mu_{far}(\delta)$.

Variable *life* contains the enemy's health points. This variable can take values in three fuzzy sets represented by the linguistic labels *low*, *med* and *high*. The membership functions related to these fuzzy sets are $\mu_{low}(life)$, $\mu_{med}(life)$, and $\mu_{high}(life)$. Variables *weapon* and *soldier weapon* describe the kind of weapon that the enemy and the soldier hold, respectively. These are non-fuzzy variables, so the only values they can take are *long range* and *short range*.

The set of input variables together with a fuzzy rule set are processed by the FRBS inference system, producing values for the output variables destination and target. These values are valid game map locations expressed in Cartesian coordinates, such as the location of the soldier or the location of a power-up. A sample fuzzy rule set is displayed in Figure 2b. Fuzzy rules may contain any input variable as an antecedent as well as any output variable as a consequent. Two different clauses can be linked in the consequent by means of the AND operator.

Figure 3a represents the finite state machine (FSM) in the action layer as a graph. During gameplay, a FSM can be placed only in one state (node) at a time. If a fixed transition condition (arc) is verified, the control system jumps a)



b) Transition conditions

c1: if $\left(\Delta_{target} < \textit{weapon range}\right)$ then (fire)

c2: if $\left(\Delta_{destination} = 0\right)$ then (stop)

c3: if $\left(\Delta_{target} > \textit{weapon range}\right)$ then (move)

c4: if (*status* is *crouched*) then (move)

c5: if (*status* is not *crouched*) then (fire)

c6: if $\left(\Delta_{destination} > 0\right)$ then (move)

c7: if $(\textit{life} = 0)$ then (die)

Figure 3: a) Description of the finite state machine in the action layer. b) The set of transition conditions between states.

from its actual state to another one that is connected to it. The basic action that an enemy performs is given by the state in which its control system is placed: “move” towards the destination specified by the strategy layer, “fire” to the target specified by the strategy layer, “stop”, and “die”, the final state only reached when an enemy has no health points. The “cfire” behaves like the “fire” state but makes the enemy shoot in a crouched position.

Figure 3b shows the set of transitions between the states of the FSM written in conditional rule form, in which the antecedent is the condition and the consequent is the arrival state. Every condition matches at least one arc in the graph. The input variables for the FSM are: the distance between the enemy and its destination ($\Delta_{destination}$), the distance between the enemy and its target (Δ_{target}), the *status* of the target (*crouched* or *stand up*) and the enemy’s *life* measured in health points.

3 The evolutionary system

In Genes of War, the action layer is common to every enemy control system. Unlike this, every strategy layer has its own unique fuzzy rule set, making every enemy control system, and therefore, every enemy in the game behave differently under the same environmental conditions. Since fuzzy logic uses a human-like knowledge representation, hand coding different fuzzy rule sets for every enemy in the game seems a feasible task. Nevertheless, the automatic generation of fuzzy rule sets from scratch seems to be a more interesting goal to achieve.

For this purpose, a grammar-guided genetic program has been included in Genes of War. A grammar-guided genetic program is a system that is able to find solutions to any problem whose syntactic restrictions can be formally defined by a CFG. The context-free grammar G_{FRBS} has been designed to generate the language composed by all the valid fuzzy rule sets that match the features required by Genes of War enemy control system. Using G_{FRBS}, a grammar-guided genetic program is able to automatically generate a population of fuzzy rule sets, and thus, a population of enemy control systems, being able to evolve them during the gameplay to create a set of enemies that autonomously adapt themselves to maximize player satisfaction. For more information about encoding FRBS in context-free grammars, please refer to [14].

At the beginning of the gameplay, an army of ten enemies is created. During the initialization step of the genetic program, a population composed by ten individuals is randomly generated. Each of them is assigned to only one enemy in the army. Each individual's genotype is a derivation tree belonging to G_{FRBS}, which codifies a fuzzy rule set. Every enemy decodes its assigned fuzzy rule set and stores it in the FRBS of its strategy layer, creating a control system that is different from any other else in the army.

3.1 Fitness evaluation

A generation is defined as the time period that starts when the soldier is spawned in the game and ends when the soldier has died l times, that is, when the player has lost l lives. When a generation ends, every enemy in the army is assigned a score that measures the performance of its control system. This score is based on several values gathered from the gameplay during that generation. Given an enemy E_i, this score is calculated as $enemyScore\left(E_i\right) = e_1 \cdot Kills_i + e_2 \cdot Damage_i + e_3 \cdot Destroy_i + lifepoints_i$, where $Kills_i$ is the number of times that the soldier was beaten by E_i, $Damage_i$ is the damage dealt by E_i, $Destroy_i$ is the number of items destroyed by E_i, $lifepoints_i$ are the remaining health points of E_i, and e_1, e_2 and e_3 are adjustment constants.

An implicit approach to player satisfaction has been implemented, so it is assumed that satisfaction is achieved by matching the challenge preferred by the player with the one offered by the game. Due to this, the fitness of E_i is calculated as $Fitness\left(E_i\right) = \left|\sigma_{challenge} - enemyScore\left(E_i\right)\right|$, where $\sigma_{challenge}$ is a parameter whose value reflects a target challenge level. The grammar-guided genetic program in Genes of War offers two different methods to set the value of this parameter.

The first, called the "adaptive method", is based on the score obtained by the player during the generation. This score is calculated analogously to an enemy's score: $playerScore = s_1 \cdot Kills + s_2 \cdot Damage + s_3 \cdot Time + s_4 \cdot Life$, where $Kills$ is the number of enemies beaten by the soldier, $Damage$ is the damage dealt by the soldier measured in health points, $Time$ is the duration of the generation in milliseconds, $Life$ is the number of health points recovered by the soldier by picking up health power-ups, and s_1, s_2, s_3 and s_4 are adjustment constants. The value of $\sigma_{challenge}$ is equal to the average player's score obtained in the last five generations.

When using the "adaptive method", individuals in the population of the genetic program (and thus, the enemies in the army) are evolved to minimize the distance between their scores and the score obtained by the player at each generation.

The second method is called the "hardwired method" and allows the shape of the learning curve, that the player must face during gameplay, to be directly programed. This curve is defined by a function $C : \mathbb{N} \rightarrow \mathbb{Q}$, defined in such a way that, given the number of the actual generation g, $C\left(g\right)$ returns the value for $\sigma_{challenge}$ in that generation.

By using this method, enemies evolve to minimize the distance between their scores and the target score programmed in $C(g)$ for each generation.

3.2 Crossover and replacement

After fitness evaluation, the ten enemies are sorted by fitness. The enemies ranked second to ninth are stored in a mating pool. Each member of the mating pool is then submitted to a crossover operation with the fittest enemy.

During a crossover operation, the genotype of both enemies is combined by means of the grammar-based crossover operator [17], creating one new genotype that encodes a new fuzzy rule set. The genotype of the non-fittest enemy is replaced by this new one.

The genotype of the tenth (the last) fittest enemy in the army is replaced by a randomly generated genotype in order to increase the exploration capability of the genetic program. The genotype of the fittest individual remains unchanged.

After crossover and replacement operations, the soldier is respawned with renewed health points and the next generation starts. All values used to calculate player's and enemies' scores are reset, as well as all scores and fitness measures.

4 Experimental results

Two experiments have been run using Genes of War. In the first, the "adaptive method" is used to calculate the target challenge level. Two players, one experienced and one beginner, have been asked to play the game during 30 generations (g) with $l = 1$, that is, one generation lasts one player's life.

Figure 4a shows the results obtained from this first experiment by the experienced player. The bold line displays the values taken by $\sigma_{challenge}$ and the dotted line shows the evolution of the score obtained by the fittest enemy in the army during 30 generations. This score evolves accordingly to $\sigma_{challenge}$ in a way similar to a predator-prey system, in which the fittest enemy chases the player generation by generation.

The value of $\sigma_{challenge}$ increases until generation 12 because the experienced player can easily handle their task (beating enemies). Then, the genetic program evolves the population to produce smarter enemies that obtain higher scores. In generation twelve the enemies are very smart, and thus harder to beat, so the player's score decreases until generation 26. This causes the genetic program to remove smart enemies from the population and produce easier ones. After generation 26 the player is skilled enough to overtake the enemies. Consequently, $\sigma_{challenge}$ increases again and the genetic program starts producing smart enemies that raise the difficulty level.

Figure 4b shows the results obtained by the beginner player. Here, $\sigma_{challenge}$ takes low values because of player's lack of experience. Initial enemies are too smart, so the genetic program evolves the population to obtain easy-to-beat enemies that better fit player's capability. After generation 9 the player is skilled enough to overtake the enemies. This leads to an increase in $\sigma_{challenge}$ that is

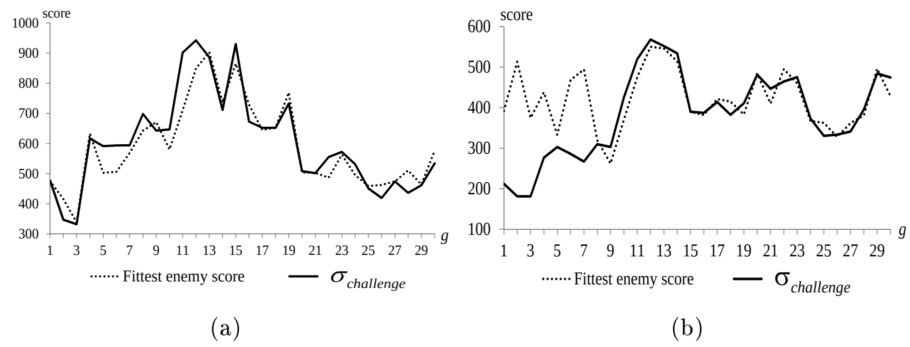


Figure 4: Evolution of the fittest enemy using the “adaptive method” with a) an experienced player and b) a beginner player.

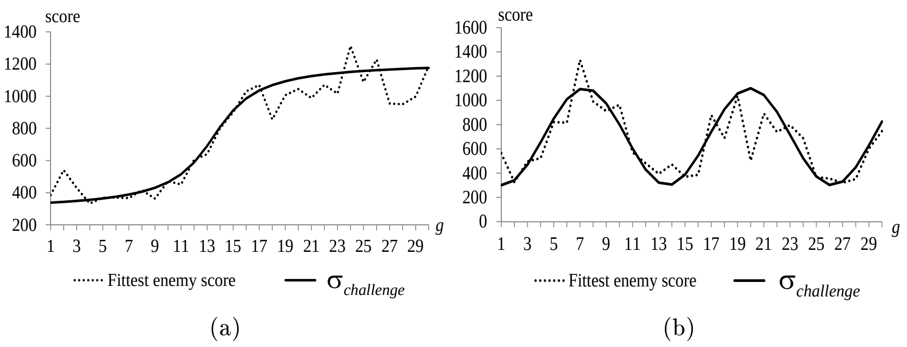


Figure 5: Evolution of the fittest enemy using the hardwired method a) with function $C_1(g)$ b) and $C_2(g)$.

followed by a predator-prey behavior like the one shown in Figure 4a, but with a smaller amplitude in this case.

In the second experiment, the hardwired method has been used to calculate the challenge level. Two different functions, $C_1(g) = 300 \cdot arctan\left(g/2{,}5 - 5\right) + 750$ and $C_2(g) = 400 \cdot cos\left(g/2 + 600\right) + 700$ have been programmed to determine the values taken by $\sigma_{challenge}$ given the generation number (g).

Figure 5a shows the results obtained from the application of $C_1(g)$. As it is shown by the bold line, this function shapes an ideal learning curve in which the challenge level increases slowly during the first stages of the gameplay, then it raises quickly by the middle game to finally return to a slow growth by the end of the game. The dotted line shows how the score of the fittest enemy closely evolves to the values assigned to $\sigma_{challenge}$ by this function.

Figure 5b shows the results obtained from the application of function $C_2(g)$, that represents an oscillatory behavior. This experiment shows how the genetic program is able to adapt the army of enemies to a variable environment, in which the target challenge level in constantly oscillating between two scores: 300 and 1100.

5 Conclusions and future work

The grammar-guided genetic program in Genes of War generates and evolves populations of enemies that match different target challenge levels to optimize player satisfaction. The difficulty in Genes of War can be adapted in real time to fit player skills, when using the "adaptive method", or to match programmers preferences, when using the "hardwired method".

By using this genetic program, no artificial behavior has to be implemented. If using the "hardwired method", programmers only have to set the target difficulty level desired at every stage of the gameplay, and armies of enemies are automatically generated to fit them. Using the "adaptive method" is even easier since the game is capable of finding the challenge level that better fits to every player at every moment. This is a key advantage because it lets programmers focus on any other aspects of game development.

In both cases, changes in game difficulty are only achieved by getting more or less intelligent enemies. Enemies "physical" attributes, like health, speed, endurance or strength, are never modified for this purpose. Smarter enemies develop intelligent behaviors that make them harder to beat. These behaviors include equipping different weapons depending on the situation, looking for health power-ups when an enemy is running low on health points, and running away from the soldier when it is too close.

Insofar as evolution is implemented as a continuous and unending task, the genetic program can operate during the whole gameplay without requiring to stop or restart the game.

The grammar-guided genetic program is a very flexible tool that can be easily modified to include any change made in later development phases. Changing the used context-free grammar makes the system capable of generating more complex fuzzy rule-based systems that deal with more environmental variables, fuzzy sets or membership functions.

The layered design of the presented control system provides it with modularity, granting that any changes performed at the strategy level will not prevent the whole system to work properly.

Due to its flexibility and modularity, it seems feasible to export the presented evolutionary system to other competitive game genres, like 3D third-person and first-person shooters, fighting and racing games.

References

1. J. Togelius, J. Whitehead, and R. Bidarra, "Guest editorial: Procedural content generation in games," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, pp. 169–171, 2011.

2. A. Doull, “The death of the level designer,” Last accessed in November, 2011. [Online]. Available: http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html
3. J. Togelius, G. Yannakakis, K. Stanley, and C. Browne, “Search-based procedural content generation,” *Applications of Evolutionary Computation*, pp. 141–150, 2010.
4. E. Hastings, R. Guha, and K. Stanley, “Evolving content in the galactic arms race video game,” in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on.* IEEE, 2009, pp. 241–248.
5. D. Loiacono, L. Cardamone, and P. Lanzi, “Automatic track generation for high-end racing games using evolutionary computation,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 245–259, 2011.
6. J. Togelius, R. De Nardi, and S. Lucas, “Towards automatic personalised content creation for racing games,” in *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on.* IEEE, 2007, pp. 252–259.
7. C. Pedersen, J. Togelius, and G. Yannakakis, “Modeling player experience in super mario bros,” in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on.* Ieee, 2009, pp. 132–139.
8. S. Lucas, “Computational intelligence and games: Challenges and opportunities,” *International Journal of Automation and Computing*, vol. 5, no. 1, pp. 45–57, 2008.
9. K. Stanley, B. Bryant, and R. Miikkulainen, “Real-time neuroevolution in the nero video game,” *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 653–668, 2005.
10. Y. Azaria and M. Sipper, “Gp-gammon: Using genetic programming to evolve backgammon players,” *Genetic Programming*, pp. 143–143, 2005.
11. A. Benbassat and M. Sipper, “Evolving board-game players with genetic programming,” in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation.* ACM, 2011, pp. 739–742.
12. Y. Shichel, E. Ziserman, and M. Sipper, “Gp-robocode: Using genetic programming to evolve robocode players,” *Genetic Programming*, pp. 143–143, 2005.
13. B. Walton, “Video game chartz,” Last accessed in November, 2011. [Online]. Available: http://www.vgchartz.com
14. J. M. Font, D. Manrique, and J. Ríos, “Evolutionary construction and adaptation of intelligent systems,” *Expert Systems with Applications*, vol. 37, pp. 7711–7720, 2010.
15. J. M. Font and D. Manrique, “Grammar-guided evolutionary automatic system for autonomously building biological oscillators,” in *2010 IEEE Congress on Evolutionary Computation*, July 2010, pp. 1–7.
16. J. Font, D. Manrique, and E. Pascua, “Grammar-guided evolutionary construction of bayesian networks,” *Foundations on Natural and Artificial Computation*, pp. 60–69, 2011.
17. J. Couchet, D. Manrique, J. Ríos, and A. Rodríguez-Patón, “Crossover and mutation operators for grammar-guided genetic programming,” *Soft Computing: A Fusion of Foundations, Methodologies and Applications*, vol. 11, no. 10, pp. 943–955, 2007.
18. G. Yannakakis and J. Hallam, “Real-time game adaptation for optimizing player satisfaction,” *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 1, no. 2, pp. 121–133, 2009.