

A Study on Genetic Algorithm and Neural Network for Mini-Games*

SAI-KEUNG WONG AND SHIH-WEI FANG

Department of Computer Science

National Chiao Tung University

Hsinchu, 300 Taiwan

Physics simulation and character control are two important issues in computer games. In this paper, we propose two games which are tailored for investigating some aspects of these two issues. We study on the applications of neural network and the genetic algorithm techniques for building the controllers and the controllers should be able to finish the specific tasks in the two games. The goal of the first game is that the controller can shoot a ball so that the ball collides with the other two balls one after another. The challenge of this game is that the ball should be shot from the proper position and the goal is achieved every time. The second game is a duel game and two virtual characters are controlled to fight with each other. We develop a method for verifying whether or not the skill power of the two virtual characters is balanced. The controllers of both games are evolved based on neural network and genetic algorithm in an unsupervised learning manner. We perform a comprehensive study on the performance and weaknesses of the controllers.

Keywords: artificial intelligence, evolutionary robotics, games, physics simulation, skill balancing

1. INTRODUCTION

Artificial intelligence techniques have been applied for computer games in different aspects, such as solving the problems of path finding [1, 2], controlling the non-player characters with a variety of reactions to players in intelligent and challenging ways, and learning the behaviors of players [3]. Some techniques are developed for dynamically adjusting the game difficulty so as to achieve game balancing [4] or even changing game parameters via online learning algorithms [5]. Despite their different purposes in the development of computer games, the common goal is to make the games more enjoyable and fun.

Physics simulation and character control are two important issues in computer games. In this paper, we aim for applying artificial neural network and the genetic algorithm techniques to handle some aspects of these two issues. We want to evolve the controllers to control virtual characters to fight with each other in a dynamics environment. In order to have a manageable environment, we propose two 3D mini-games which are tailored for our investigation. The goal of the first game is that the controller can shoot a ball so that the ball collides with the other two balls one after another. The challenge of this game is that the ball should be shot from the proper position and the goal is achieved every time. The second game is a duel game and two virtual characters are controlled to fight with each other. We develop a method for verifying whether or not the skill power of the two

Received February 28, 2011; revised August 19, 2011; accepted August 30, 2011.

Communicated by I-Chen Wu.

* This work was partially supported by the National Science Council of Taiwan, under Grant No. NSC 99-2221-E-009-143.

virtual characters is balanced. The controllers of both games are evolved in an unsupervised learning manner. To accelerate the computation, parallel computing technique is adopted.

In physics simulation games, collision detection and collision response are two of the key elements. Furthermore, the characters can move around in an arena and shoot projectiles towards opponents with different effects. Skill balancing between two characters with different skills is important for game play experience. Hence, we study these two aspects based on the two games. The major contributions of this paper include: (1) a method with high aiming accuracy for shooting a ball to multiple targets one after another; and (2) an automatic method for performing verification of skill balancing between two characters in a duel game.

2. RELATED WORKS

Artificial intelligence has been researched extensively and applied in different types of games [6-13]. These game types include action, adventure, sports, role-playing, racing and god-game. Fogel developed an optimization approach for evolving artificial neural network for playing chess [14]. Cole *et al.* presented methods for tuning first-person shooter bots by applying genetic algorithms [15]. The video game *NERO* [16], which was developed by Kenneth *et al.*, was one of the innovative examples. The agents in the game were capable of learning online while the game was being played. The skills of the agents were evolved gradually. The commercial game *Black and White* from Lionhead Studios was an example for imitation. There was one major non-player character which imitated the actions performed by the player.

Genetic algorithm has seen applications in collision detection for rigid bodies [17] and fabric simulation [18]. A genetic algorithm was employed for computing the oriented bounding volume so as to improve the culling efficiency of the bounding volume tests [19]. Riechmann connects the theory of genetic algorithm to evolutionary game theory [20]. Revello and McCartney applied genetic algorithm to war games which contain uncertainty [21]. Cardamone and Loiacono presented controllers for car racing games with neuroevolution [22]. Wong utilized backpropagation neural network for personalised difficulty adjustment in a game system [23].

3. CONTROLLER TRAINING

We employ the artificial neural network (ANN) with the genetic algorithm for training the controllers in an unsupervised learning manner. The structure of a neuron of an ANN is shown in Fig. 1. The implementation of the neural network is simple and acceptable results can be produced by using a small number of parameters.

Genetic algorithm is one of the evolutionary algorithms inspired by the process of natural evolution [24, 25]. It gives solutions for optimizing problems by applying the techniques, such as inheritance, mutation, selection and crossover. Solutions generated by genetic algorithms are usually encoded as a set of genes. A set of genes can be interpreted as one of the possible solutions. By combining genetic algorithms with artificial neural networks, the solutions, *i.e.* the sets of weights of the artificial neural networks, are needed



Fig. 1. One of the neurons of the input layer in a neural network, where x_1 to x_n are the n inputs, w_1 to w_n are the weights for the inputs, w_b is the bias, $+$ stands for summation, and f is the activation function.

to be encoded and treated them as genes for evolving the neural networks gradually. Initially, all the weights of the neural networks of the whole population are generated randomly. During the training process a *fitness value* which is a value measuring the quality of a controller is calculated according to a predefined fitness function. The controllers with the highest fitness values are regarded as the elites of the population. The next generation is created by the process of inheritance, mutation, selection, and crossover. If the fitness function is good, the solutions are expected to be getting better and better for new generations. Usually the fitness function is depended on the game rules. The encoding scheme for a gene (*i.e.* the weights of the neural network) is a set of floating point numbers. The selection is based on Roulette Wheel sampling. Inheritance, crossover and mutation operators are directly applied for modifying the genes. For example, if crossover is performed for two chromosomes, a crossover point is randomly selected and then all the genes beyond the crossover point of the two chromosomes are swapped; if mutation is performed, a subset of weights of the gene is randomly selected and each weight is perturbed randomly.

4. THE METHODOLOGY

In this section, we present our approach for building up our two mini-games and training the controllers.

4.1 Game Building

We build up our gaming environments by using OGRE3D [26] which is one of the most popular open-source graphics rendering engines. The 3D models are texture-mapped and rendered. Two games are created and the controllers of the games are trained independently.

4.2 Multi-thread Parallel Training/Evolving

The training/evolving phase of artificial neural networks is usually the most time-consuming process. The computation cost increases for simulating the movements of 3D

models and computing the game-logic. We need to perform parallel computation to speed up the training process. In order to do so, every controller in each of our games has a unique game space. Each game space stores a set of dynamic data that may be changed over time, such as the positions and velocities of 3D models. There are some static data which are shared by all the game spaces, for example, the 3D mesh data of models. The 3D mesh data are static and they do not change over time. Hence, each game space keeps track of the dynamic data. There are no direct or indirect interactions between the controllers during a training session. Each controller does not affect nor be affected by the other controllers. Hence, they are trained separately. The training results of the controllers are combined when a new generation is created. A new set of weights is then applied to the controllers. The training session is repeated until a certain condition is satisfied and then the entire training process is completed. Branke [27] had concluded that adopting a multi-threading parallel approach for training/evolving controllers could alleviate the computation cost problem [28]. In our approach, a game space may need to compute random numbers. It is therefore necessary that each game space has its own random number generator. We employ Mersenne Twister [29] for computing random numbers.

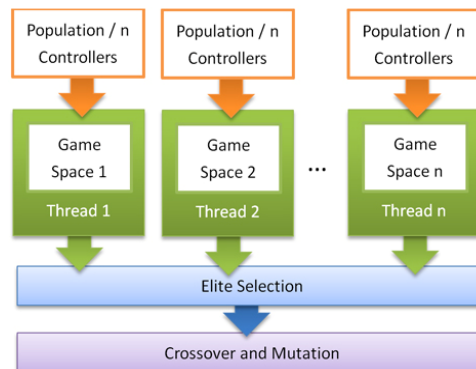


Fig. 2. The architecture for the parallel training session.

In our approach, we create n threads on a multicore system with n cores, as shown in Fig. 2. Each of the threads has a game space. Then we divide the population evenly to the n threads. For each generation, each thread performs the computation for training the assigned controllers one by one during the training session. Once all the threads have finished for training all the controllers, the process of the elite selection starts, followed by the crossover and the mutation processes. In this way, we ensure that all the controllers are trained in the parallel manner correctly and independently. The game space of each controller can be reset after a new generation is created. A new training session is then carried out. The condition for checking whether or not the training is finished is that the maximum number of generations is reached or a best controller can fulfill the game goal.

5. IMPLEMENTATION OF THE GAMES

We describe the approaches for implementing our two games in this section. They

are Snowball Shooting Game and Skill Balancing Game (or Wizard Duel Game). The controllers (bots) of both games are trained by using the neural network with the genetic algorithm on the parallel computing architecture. The goal of the first game is to control a penguin to shoot a snowball and the snowball then collides with the other two balls one after another. The goal of the second game is to perform automatic verification for skill balancing between two virtual characters with different skills.

5.1 Game 1: Snowball Shooting Game

This game is intended for training a penguin to shoot a snowball and make the shiny ball hit and destroy the robot. The challenging task is that the penguin should be able to destroy the robot consecutively. The layout of the objects and the game rules are described as follows. There are a penguin at the lower part of the battlefield, two balls placed in the middle part and a robot at the upper part, as shown in Fig. 3. The penguin can move horizontally to left or right, and it can shoot a snowball along the vertical direction. The penguin should shoot a snowball at the rusty ball for hitting the shiny ball and then the shiny ball must hit the robot in order to score. If the robot is not hit, the position of the rusty ball is reset. If the robot is hit by the shiny ball, a new position of the robot is randomly generated. Collision detection and collision response are performed for the balls based on physics laws. We adopt the genetic algorithm to train the controllers of the penguin. As this is an unsupervised learning, the controllers are evolved according to their fitness values. There is no involvement from the human players. During the training process, the controllers learn the game rules implicitly. It is therefore important to encourage the controllers to perform certain kind of actions in some specific situations or discourage them in other situations.

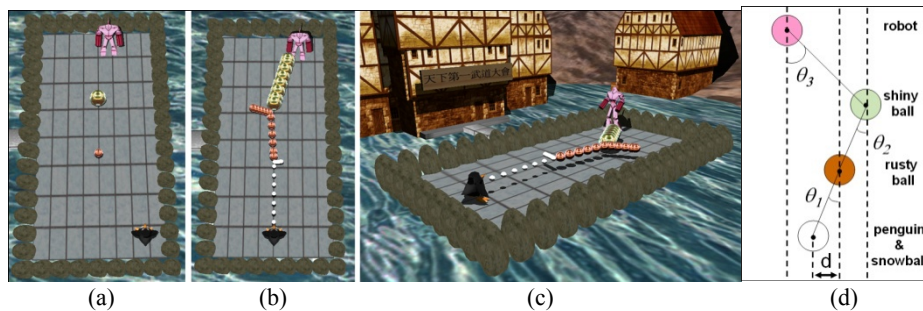


Fig. 3. (a) Objects in game; (b) A snowball is shot and the movement paths of the three balls are shown; (c) A perspective view of the game scene; (d) Layout for computing inputs.

Configurations: There are six inputs including three (normalized) angles (*i.e.* θ_1 , θ_2 and θ_3), the (normalized) signed distance d between the rusty ball and the snowball along the x -axis, the firing status of the penguin and a value k for indicating whether or not the snowball hits the rusty ball if the penguin shoots the snowball at its current position. The value of k is computed as:

$$k = \begin{cases} 0 & \text{the snowball has been shot} \\ 1 & \text{the snowball would hit the rusty ball if it is shot.} \\ 0.5 & \text{otherwise} \end{cases} \quad (1)$$

The signed distance d gives a hint for the movement direction of the penguin. The three angles are used for computing the shooting angle of the snowball. The firing status informs the penguin that whether it has shot the snowball or not.

There are three outputs. The first output is used for computing the movement direction of the penguin (Left/Right), the second output is used for determining whether or not the penguin should shoot the bullet and the third output is used for computing the speed of the snowball. The movement of the penguin is modeled as follows. Let Δt be the simulation time step. Then the velocity of the penguin is computed as $v = v_0 + a\Delta t$ and its position is computed as $p = p_0 + v\Delta t$, where the subscript 0 means the previous frame and a is the acceleration. If the penguin is moving to the left side, a is negative; otherwise it is positive. There is a maximum speed for the penguin. The penguin moves at discrete positions due to the nature of the simulation system.

There is one hidden layer in the neural network and a hidden layer has six neurons. The crossover rate and the mutation rate are 0.25 and 0.4, respectively. We set the maximum perturbation to the weights is 0.3. The number of elite copies is 4 and the population size is 400. Once a training session is done a new generation is created.

During the training session, a penguin can shoot the snowball. We compute the collision status between the shiny ball and the robot by using ball-ball collision check. In each simulation step, the positions and velocities of the balls and the penguin are updated. If the penguin shoots a snowball, it can shoot another snowball until one of the following four termination conditions is satisfied:

TC1: the shiny ball hits the robot.

TC2: the snowball moves out of range and it does not hit the rusty ball.

TC3: the rusty ball moves out of range and it does not hit the shiny ball.

TC4: the shiny ball moves out of range.

An action session begins at the moment the penguin shooting the snowball until one of the termination conditions is satisfied. Denote the three balls (*i.e.* snowball, rusty ball and shiny ball) as p_i ($i = 1, 2$ and 3) and the robot as p_{robot} . Then we proceed to compute a fitness factor as follows,

$$s(p_1, p_2, p_3, p_{robot}, n_{ch}, n_g) = \alpha(n_g)(\alpha h(p_1, p_2) + \beta h(p_2, p_3)) + \lambda + \gamma h(p_3, p_{robot}) + \kappa n_{ch} \quad (2)$$

where $h(x, y)$ is a binary value indicating whether or not x hits y during the current action session, the counter n_{ch} stores the number of consecutive hits at the robot, λ is used for encouraging the penguin to move to the proper position for shooting the snowball, n_g is the current number of generations, $\omega(n_g)$ is one of the forms 0 , $1/(k_g n_g / N_g + 1)$ and $1/((k_g n_g / N_g)^2 + 1)$, k_g is a constant (it is set to 5 in all the experiments), N_g is the maximum number of generations, α , β and γ are weighting values and $\alpha < \beta < \gamma$. Usually, $\gamma h(p_3, p_{robot})$ should be much larger than $\omega(n_g)(\alpha h(p_1, p_2) + \beta h(p_2, p_3))$ so that the penguin has the incentive to

attempt to destroy the robot. Consider that there are 100 generations (*i.e.* $N_g = 100$). For the first 20 generations, $\omega(n_g)$ is set as $1/(k_g n_g / N_g + 1)$. For the generations between 21-80, $\omega(n_g)$ is set as $1/((k_g n_g / N_g)^2 + 1)$. And then in the remaining generations, $\omega(n_g)$ is set to zero. λ is a non-zero value if the penguin waits for a while before it shoots the snowball and the snowball hits the rusty ball. Furthermore, to encourage the penguin to move to the correct position for shooting the snowball, the value of κ should be larger than or equal to γ . The counter n_{ch} is reset if the robot is not destroyed in the current action session. In this way, the penguin has incentive to move to the proper position before shooting. Notice that the penguin takes a while before it reaches at the proper position. If such action was not rewarded, the penguin would simply stay around and shoot the snowball to hit the rusty ball only. The penguin is hardly evolved to achieve the goal. In order to encourage the penguin to move to the correct position before shooting, it is therefore the fitness factor should be increased significantly. We compute the fitness value F_t as follows,

$$F_t = c(\theta_4) s(p_1, p_2, p_3, p_{robot}, n_{ch}, n_g) \phi(\text{dis}(p_3, p_{robot})), \quad (3)$$

where θ_4 is the angle between the velocity of the shiny ball and the vector formed by the shiny ball and the robot, $c(\theta_4)$ is the precision bonus, $\text{dist}(p_3, p_{robot})$ is the distance between p_3 and p_{robot} , and the function $\phi(\text{dist}(p_3, p_{robot}))$ is defined in the form $1/\text{dist}(p_3, p_{robot})$. The term $c(\theta_4)$ is higher for smaller θ_4 . In the early generations, we motivate the penguin to shoot the snowball when it can. We also encourage the penguin that can destroy the robot consecutively as the fitness value is higher for more consecutive hits.

5.2 Game 2: Skill Balancing Game (Wizard Duel)

Skill balancing among different classes/races of player characters is often an important issue in online RPGs (Role-Playing Games) or ARPGs (Action Role-Playing Games). A game is skill-balanced if a class/race should be as powerful as other classes/races. In order to make a game skill-balanced, there are several up to hundreds or thousands of parameters that may need to be carefully adjusted. This is a challenging problem in game design. And for some games, this could be a never-ending tuning problem, which might consume a lot of time, human resources, and money. A game company might deal with this issue by running many tests played by players. We develop an approach based on genetic algorithm for determining whether or not the skills of two classes/races are balanced in a one-on-one PK (Player Killing) system. We then apply the skill balancing system to our Wizard Duel game.

There are two different wizards, the fire-majored and the ice-majored wizards, in an arena. Each of them can cast four different spells. And every spell has its own setting, including the damage power, casting time, cost of mana, the attack range, the CD (cool-down) time and special effects, the values of the settings are defined by us after referencing to several modern online games, such as Dragon NestTM. Different skills may have different ways to attack. For examples, some skills may cause single damage to the target while some may cause multiple hits to the target according to the distance between the target and the caster. The special effects of the skills may have different de-buffing states to the targets, such as burning for a certain duration of time, frostbiting for decreasing the target movement speed, freezing and stunning temporarily. The skill lists of the two wizards are shown in Tables 1 and 2, respectively.

Table 1. Skill list of fire wizard.




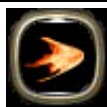




 Fire Ball	Attack Power: Magical Attack Power $\times 501\% + 414$ Cost MP: Basic MP $\times 2.1\%$ Attack Range: 10.5885m Max Damage Hit: 1 Hit	Cast Time: 2 sec. CD Time: 17 sec. Special Effect: Burn, take 10% of last hit damage point every 2 sec. and last for 8 sec.; Stun, last for 1.5 sec.
 Inferno	Attack Power: Magical Attack Power $\times 17.25\% + 23$ Cost MP: Basic MP $\times 3.4\%$ Attack Range: 3.5295m Max Damage Hit: 12 Hits	Cast Time: 0.8 sec. CD Time: 24 sec. Special Effect: Burn, take 50% of last hit damage point every 2 sec. and last for 10 sec.; Stun, last for 4 sec.
 Fire Wall	Attack Power: Magical Attack Power $\times 50.17\% + 162$ Cost MP: Basic MP $\times 3.2\%$ Attack Range: 2.5883m Max Damage Hit: 6 Hits	Cast Time: 1.2 sec. CD Time: 33 sec. Special Effect: Burn, take 5% of last hit damage point every 2 sec. and last for 10 sec.; Stun, last for 3 sec.
 Fire Shots	Attack Power: Magical Attack Power $\times 68\% + 94$ Cost MP: Basic MP $\times 2.5\%$ Attack Range: 10.5885m Max Damage Hit: 3 Hits	Cast Time: 0.1 sec. CD Time: 20 sec. Special Effect: Burn, take 10% of last hit damage point every 2 sec. and last for 15 sec.

Table 2. Skill list of ice wizard.

 Freezing Sword	Attack Power: Magical Attack Power $\times 63\% + 107$ Cost MP: Basic MP $\times 2.3\%$ Attack Range: 2.353m Max Damage Hit: 3 Hit	Cast Time: 0.1 sec. CD Time: 15 sec. Special Effect: Frostbitten, moving speed $\times 50\%$ and last for 2 sec.; Stun, last for 3.5 sec.
 Freezing Field	Attack Power: Magical Attack Power $\times 196\% + 320$ Cost MP: Basic MP $\times 3.1\%$ Attack Range: 5.4119m Max Damage Hit: 1 Hit	Cast Time: 1.1 sec. CD Time: 28 sec. Special Effect: Frostbitten, moving speed $\times 50\%$ and last for 8 sec.
 Blizzard	Attack Power: Magical Attack Power $\times 131\% + 334$ Cost MP: Basic MP $\times 2.3\%$ Attack Range: 258.83m Max Damage Hit: 2 Hits	Cast Time: 0.9 sec. CD Time: 15 sec. Special Effect: Frostbitten, moving speed $\times 50\%$ and last for 1 sec.; Stun, last for 0.5 sec.
 Instant Freeze	Attack Power: Magical Attack Power $\times 76\% + 188$ Cost MP: Basic MP $\times 2.3\%$ Attack Range: 1.8824m Max Damage Hit: 1 Hits	Cast Time: 0.5 sec. CD Time: 24 sec. Special Effect: Frozen, last for 2 sec.

For simplicity, we set all other attributes of the two wizards the same. That is that they have the same health points, mana points, initial movement speed, attacking factors, defending factors and critical hit rate and *etc.* Our system neglects any human-skill related factors so that every spell that a player casts is perfectly-aimed at the opponent.

Table 3. Detail settings of GA in the Wizard Duel game.

Population Size	200	Crossover Rate	0.4	Mutation Rate	0.15
Perturbation Value	-0.3 ~ 0.3	Number of Elites	4	Copies of Elite	1

Configurations: The settings of the artificial neural networks are the same for the fire-wizard and ice-wizard controllers. There are eleven inputs and two outputs, and there is one hidden layer in the network with six neurons. The weights of the network are regarded as the genes of a chromosome. To model mutation, a perturbation value is added or subtracted to the gene (weight). Table 3 details the parameters of GA.

The detailed description of the inputs of the neural network is given as follows,

(1-4) The skill ready-rate of the 4 skills of the training bot is defined as follows,

$$SkillReadyRate_i = (TotalCDTime_{Skill_i} - RemainCDTime_{Skill_i}) / TotalCDTime_{Skill_i},$$

for $i = 1, 2, 3, 4$.

- (5) (Remaining frozen or stunned time of the opponent)/(MAX frozen or stunned time of all skills).
- (6) (Remaining spell casting time of the opponent)/(MAX spell casting time of all skills).
- (7) (Remaining frostbitten time of the opponent)/(MAX frostbitten time of all skills).
- (8) (Estimated movement time to the opponent)/(MAX movement time).
- (9) (The skill ID which is cast by the opponent)/(Total number of skills of the training bot = 4). ID = 0, 1, 2, 3, 4. For ID = 0, it implies that the opponent is not casting any skill at the moment.
- (10) (Estimated time for moving to the safe position)/(MAX movement time). The safe position is the position which is outside of the attack range of the opponent. The value is set to 0 if the opponent is not casting any skill at the moment.
- (11) (The last movement skill ID of the opponent)/(Total number of movement IDs = 3). The opponent moves towards the training bot, stays, or moves away from the training bot.

The description of the two outputs of the neural network is given as follows,

- (1) The first output value is between (0, 1). We divide it into five intervals evenly and each interval corresponds to one of the five IDs of the skill that the wizard attempts to cast. The IDs are 0, 1, 2, 3, 4. For ID = 0, it implies that the wizard does not cast any skills.
- (2) The second output value is between (0, 1), which encodes the movement ID of the training bot. The training bot moves toward the opponent, stays or moves away from the opponent.

The fitness value F_t is given by:

$$F_t = F_{t-1} + G_t - L_t, \quad (4)$$

where F_{t-1} is the fitness value of the training bot at time step $t - 1$. The value G_t (Gained)

is the value for the score gained in the current time step t , and L_t (Loss) is the amount of decreased score. The value G_t is computed as follows,

$$G_t = \text{AtkDmgPt} + \text{SpellCasting} + \text{SpellFiring} + \text{BreakSpell} + \text{SpellEvaded} + \text{WinPt}, \quad (5)$$

$$\text{BreakSpell} = f_{\text{BreakSpell}} \times \text{AvoidDmgPt}. \quad (6)$$

- AtkDmgPt are the damage points made to the opponent by attacking skills.
- SpellCasting is added when the bot has started casting a spell successfully.
- SpellFiring is added when the bot finishes casting a spell successfully rather than the spell is blocked by the opponent.
- BreakSpell is affected by the value of AvoidDmgPt times by a factor $f_{\text{BreakSpell}}$. The value of AvoidDmgPt is the damage points that the training bot has successfully avoided by blocking the spell casted by the opponent.
- SpellEvaded is added when the training bot evades the attack by the opponent. That is that the training bot stays inside the attack range of the opponent while the opponent has started to cast a spell. But after the opponent has finished casting the spell, the training bot has moved out of the attack range of the opponent. SpellEvaded is computed as the damage would be taken if the training bot does not evade from the attack.
- WinPt is added when the training bot defeats the opponent.

The value L_t is computed as follows,

$$L_t = \text{GotHurt} + \text{CastingWhileCD} + \text{SpellBroken} + \text{OutOfRange} + \text{LosePt}, \quad (7)$$

$$\text{SpellBroken} = f_{\text{BreakSpell}} \times \text{MissedDmgPt}. \quad (8)$$

- GotHurt is added when the training bot is hit by the opponent.
- CastingWhileCD is added when the training bot is trying to cast a spell but it is in CD.
- SpellBroken is added when the spell casting of the training bot is blocked by the opponent. The value of SpellBroken is MissedDmgPt times by the factor $f_{\text{BreakSpell}}$. The amount of MissedDmgPt is the damage points that the opponent has avoided because of the blocking the spell.
- OutOfRange is counted if the training bot starts casting a spell while the opponent stays outside of the attack range.
- LosePt is counted for the training bot being defeated.

Note that, for different games, additional weighting factors or fitness terms can be added to the formulas. We train the controllers by the multithreading technique. Both kinds of controllers are trained at the same time. The n th fire-wizard fights against the n th ice-wizard in a game space instead of fighting a wizard performing random actions. In this way, both wizards can have a better chance to improve their skills.

6. EXPERIMENTS AND RESULTS

We performed experiments for the two 3D games. In the following, we describe the experimental setups and then present the results for each game.

6.1 Game 1: Snowball Shooting Game

We conducted two sets of experiments by applying different fitness functions to the game. In each of the experiments, the maximum speed of the penguin was 0.01 and time step was 1. We trained the controllers for 200 generations and performed 25,000 simulation steps for each generation.

Experiment Set One: We implemented all the terms for computing the fitness value. We set $\alpha = 10$, $\beta = 40$, $\lambda = 500$, $\gamma = 1000$, $\kappa = \alpha + \beta + \gamma$, $c(\theta_4) = 1.5\cos\theta_4$, and $\phi(\text{dist}(p_3, p_{\text{robot}})) = 1/(\text{dist}(p_3, p_{\text{robot}}) - (r_2 + r_3) + 1)$. For the first 40 generations, $\omega(n_g)$ was set as $1/(k_g n_g / N_g + 1)$. For the generations between 41-160, $\omega(n_g)$ was set as $1/((k_g n_g / N_g)^2 + 1)$. In the remaining generations, $\omega(n_g)$ was set to zero. The purpose of this experiment is to verify the correctness of the fitness function.

Experiment Set Two: We kept the parameters the same as the first experiment (1). We however removed one of the terms or set a term to constant. In total, there were seven experiments. The purpose of this experiment is to check whether or not some terms of the fitness function can be eliminated without affecting the performance of the penguin.

Observations and Discussions:

Experiment Set One: We denote the radii of the three balls and the robot as r_i , $i = 1, 2, 3$ and 4. We performed experiments with different settings for r_i , such as $\{5, 10, 15, 10\}$, $\{5, 10, 20, 10\}$ and $\{5, 30, 10, 10\}$. The penguin can destroy the robot if it shoots the snowball from the proper position. In the series of experiments with these settings, the penguin moved to the proper positions and shot the snowball. The snowball hit the rusty ball and the rusty ball then collided with the shiny ball. Finally, the shiny ball hit the robot. After that the penguin could move to the proper position and shot the snowball again without wasting any snowballs. The penguin could destroy the robot consecutively. In order to evaluate the successful rate of the penguin, the game was played for five hours. We recorded the number of times that the penguin could destroy the robot. The successful rate was 99%. If the robot was generated at the two corners, it took two or three attempts for the penguin to destroy the robot. These cases occurred rarely.

The speed of the penguin should be small enough so that it is possible for the penguin moving to the correct positions for shooting. The possible positions for the penguin are a finite set of positions due to the nature of a physics simulation system with a fixed time step. For a larger speed of the penguin, the number of possible positions for the penguin to move is smaller. It is possible that the set of the possible positions may not contain the proper position. In this case, the penguin cannot be trained properly. Recall that v is the velocity of the penguin and Δt is the simulation time step. For example, if the $|v\Delta t|$ is larger than or equal to $r_1 + r_2$ (sum of the radii of the snowball and rusty ball), there are at most three possible positions for the penguin to shoot the snowball for hitting the rusty ball. But the shiny ball hardly collides with the robot. Hence, we can either decrease Δt or $|v|$ so that the set of the possible positions for the penguin contains the proper positions.

Experiment Set Two: The performance of the penguin was poor in these seven experi-

ments. The penguin simply shot the snowball as soon as a termination condition was satisfied. The penguin did not attempt to move to a proper position before shooting the snowball. It could not destroy the robot consecutively. In some cases, the penguin simply stayed near a spot and kept on shooting the snowball.

Based on the results of the second experiment, we should design the fitness function which encapsulates the cost of each task. For the tasks taken relatively longer time to accomplish, the fitness value associated with these tasks should be relatively higher. We find out that it is easier to evolve the controllers for achieving a sequence of tasks by letting the controllers obtain much higher fitness values of the latter tasks than from the early tasks. Since the Snowball Shooting game is a physics simulation game, some tasks, such as penguin moving to the proper position and the entire process for hitting the robot, are required higher number of simulation steps to accomplish. If the tasks are accomplished, the fitness value should be added accordingly by considering the duration for accomplishing the tasks.

6.2 Game 2: Skill Balancing Game

It is difficult, in general, to judge whether or not the skills of both wizards are balanced by inspecting only the skill lists. After we had trained the two different kinds of bots for 2000 generations, then we performed the balance test. The balance test is that we pick the best bot of each kind and let them fight until one of them obtains 1000 scores. We repeated the balance test for five times. The snapshot of the game is shown in Fig. 4. The scores of the bots are shown in Table 4.



Fig. 4. Skill balancing game.

Table 4. Skill balancing before adjustment.

Balance Test	1	2	3	4	5	Average
Fire Wizard	1000	1000	773	1000	1000	954.6
Ice Wizard	512	453	1000	876	678	703.8

The set of the skills of the fire wizard is more powerful than the set of the skills of the ice wizard, as shown from the results. Therefore, as an example for skill balancing, we chose to strengthen the attack power of the skill ‘Freezing Field’ and the hit time of ‘Bliz-

zard' of ice wizard but to weaken the attack power of the skill 'Fire Ball' of fire wizard. We made some changes to the settings of the three skills gradually, and once a new setting is made, we performed the balance test to check whether or not the skill power of the two characters was balanced. And after we modified the settings and performed the balance test for a few times, we finally came up with a setting such that the skill power of the two characters was balanced. We decreased the attack power of 'Fire Ball' to $\text{Magical Attack Power} \times 401\% + 414$, increased the attack power of 'Freezing Field' to $\text{Magical Attack Power} \times 296\% + 320$, and increased the maximum damage hit to 3 hits of the skill 'Blizzard'. The new results for five tests are shown in Table 5.

Table 5. Skill balancing after adjustment.

Balance Test	1	2	3	4	5	Average
Fire Wizard	972	1000	1000	929	816	943.4
Ice Wizard	1000	968	909	1000	1000	975.4

The new results shows that the skill sets of both wizards are approximately balanced. The average damage taken by both sides was nearly the same.

7. CONCLUSION

We have applied neural networks with genetic algorithm for two games, one for physics simulation and another for character control. Based on the experiments of the Snowball Shooting game, we should lower our standard and increase the fitness values of the controllers if they make good attempts at the early phase of training. Gradually the standard should be raised step by step. The fitness value for a task required longer time to accomplish should be higher. In this way, we can train controllers with better performance in a relatively short time. We notice that artificial neural networks can be more than just controlling the game play directly. We can employ it for game balancing, such as in the Wizard Duel game. From the training records, game designers would have a better understanding of the skills. For example, which skills are less used than the others? By analyzing the training records, a game designer might retrieve a lot of valuable information and have a better sense about the current balancing state of the game in a shorter time compared to human testing. Then he/she can adjust the parameters of the game accordingly.

In the future, we would like to integrate the two proposed methods into a single game which includes character control and physics simulation. Currently, the manual operations are required for setting different skill attributes in order to perform the verification test for skill balancing in a duel game. We would like to develop a fully automatic method for performing game balancing in a duel game.

REFERENCES

1. K. D. Forbus, J. V. Mahoney, and K. Dill, "How qualitative spatial reasoning can improve strategy game AIs," *IEEE Intelligent Systems*, Vol. 17, 2002, pp. 25-30.
2. R. Graham, H. McCabe, and S. Sheridan, "Pathfinding in computer games," *Institute*

- of Technology Journal*, Vol. 9, 2003, pp. 1-10.
3. C. Thureau, C. Bauckhage, and G. Sagerer, "Imitation learning at all levels of game-AI," in *Proceedings of the International Conference on Computer Games, Artificial Intelligence, Design and Education*, 2004, pp. 402-408.
 4. R. Hunnicke and V. Chapman, "AI for dynamic difficulty adjustment in games," in *Proceedings of Challenges in Game Artificial Intelligence AAAI Workshop*, 2004, pp. 91-96.
 5. A. Pfeifer, "Creating adaptive game AI in a real time continuous environment using neural networks," Master Thesis, Knowledge Engineering Group, TU Darmstadt, 2009.
 6. J. Laird and M. VanLent, "Human-level AI's killer application: Interactive computer games," in *Proceedings of the 17th International Conference on Artificial Intelligence*, 2000, pp. 1171-1178.
 7. R. Koster, *A Theory of Fun for Game Design*, Paraglyph Press, United States, 2004.
 8. J. Togelius and S. M. Lucas, "Evolving controllers for simulated car racing," in *Proceedings of IEEE Congress on Evolutionary Computation*, 2005, pp. 1906-1913.
 9. G. N. Yannakakis, "AI in computer games: Generating interesting interactive opponents by the use of evolutionary computation," Ph.D. Thesis, College of Science and Engineering, School of Informatics, University of Edinburgh, 2005.
 10. J. Togelius, S. Lucas, and R. Nardi, "Computational intelligence in racing games," *Advanced Intelligent Paradigms in Computer Games*, 2007, pp. 39-69.
 11. N. van Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber, "Robust player imitation using multiobjective evolution," in *Proceedings of IEEE Congress on Evolutionary Computation*, 2009, pp. 652-659.
 12. J. Westra and F. Dignum, "Evolutionary neural networks for non-player characters in quake III," in *Proceedings of the 5th International Conference on Computational Intelligence and Games*, 2009, pp. 302-309.
 13. S. H. Jang, J. W. Yoon, and S. B. Cho, "Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary algorithm," in *Proceedings of the 5th International Conference on Computational Intelligence and Games*, 2009, pp. 75-79.
 14. D. B. Fogel, *Blondie24: Playing at the Edge of AI*, Morgan Kaufmann, United States, 2002.
 15. N. Cole, S. Louis, and C. Miles, "Using a genetic algorithm to tune first-person shooter bots," in *Proceedings of International Congress on Evolutionary Computation*, Vol. 1, 2004, pp. 139-145.
 16. K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Evolving neural network agents in the NERO video game," in *Proceedings of IEEE Symposium on Computational Intelligence and Games*, 2005, pp. 182-189.
 17. J. Wu, L. Chen, L. Yang, Q. Zhang, and L. Peng, "A collision detection algorithm based on self-adaptive genetic method in virtual environment," in *Proceedings of the 1st International Conference on Swarm Intelligence*, 2010, pp. 461-468.
 18. B. B. Li and Z. H. Zhao, "Fitness function optimized in genetic algorithm for fabric dynamic simulation," in *Proceedings of IEEE Pacific-Asia Workshop on Computational Intelligence and Industrial Application*, 2008, pp. 59-63.
 19. W. Zhao, L. J. Li, and C. S. Chen, "Research on collision detection algorithm based

- on particle swarm optimization,” *Lecture Notes in Computer Science*, Vol. 6249, 2010, pp. 602-609.
20. T. Riechmann, “Genetic algorithm learning and evolutionary games,” *Journal of Economic Dynamics and Control*, Vol. 25, 2001, pp. 1019-1037.
 21. T. Revello and R. McCartney, “Generating war game strategies using a genetic algorithm,” in *Proceedings of Congress on Evolutionary Computation*, Vol. 2, 2002, pp. 1086-1091.
 22. L. Cardamone, D. Loiacono, and P. Lanzi, “Evolving competitive car controllers for racing games with neuroevolution,” in *Proceedings of Annual Conference on Genetic and Evolutionary Computation*, 2009, pp. 1179-1186.
 23. K. Wong, “Adaptive computer game system using artificial neural networks,” *Neural Information Processing*, 2008, pp. 675-682.
 24. J. H. Holland, “Genetic algorithms,” *Scientific American*, 1992, pp. 66-72.
 25. D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Longman Publishing Co., Inc., Boston, 1989.
 26. OGRE3D, <http://www.ogre3d.org/>.
 27. J. Branke, “Evolutionary algorithms for neural network design and training,” in *Proceedings of the 1st Nordic Workshop on Genetic Algorithms and its Applications*, 1995, pp. 145-163.
 28. E. Cantú-Paz, “A survey of parallel genetic algorithms,” *Calculateurs Paralleles, Re-seaux et Systems Repartis*, Vol. 10, 1998, pp. 141-171.
 29. Mersenne Twister, <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>, last visited on 25th Feb. 2011.



Sai-Keung Wong (黃世強) received his Ph.D. and M.S. degrees in Computer Science from the Hong Kong University of Science and Technology in 2005 and 1999, respectively. He has been an Assistant Professor of the Department of Computer Science of the National Chiao Tung University, Taiwan, since 2008. His research interests include computer animation, collision detection, 3D game engines, visualization and artificial intelligence.



Shih-Wei Fang (方士偉) received his B.S. degree in Computer Science from the National Tsing Hua University, Taiwan, in 2008. He is currently a master student in the National Chiao Tung University, Taiwan. His research interests include 3D games and computational intelligence.