AN AGENT-BASED APPROACH FOR CONSTRUCTING SOFTWARE SYSTEMS OF VIRTUAL SIMULATION

Li Hongbing, Meng Bo, Chen Shifu
Nanjing University, P. R. China

ABSTRACT

The design and construction of virtual reality environments involve technologies such as computer graphics, image processing, pattern recognition, intelligent interface, artificial intelligence, voice recognition, network, parallel processing, and high-performance computing. Some researchers insist that object-oriented and agent-oriented technologies are fundamental for virtual reality system design. This paper applies artificial intelligence to the design of virtual reality systems. Agents are constructed by using object-oriented methods and a set of underlying computing models, such as neural networks, genetic algorithms, expert systems, and plan managers. Some object-oriented frameworks of these computing models are presented to illustrate this approach. The example of a spaceship game will illustrate interactions among environments, agents, and underlying computing models. The approach and reusable class library presented herein can be applied to various virtual reality environment simulations and intelligent applications.

1. Introduction

The design and construction of virtual reality environments require technologies such as computer graphics, image processing, pattern recognition, intelligent interface, artificial intelligence, voice recognition, network, parallel processing, and high performance computing. Reference [1] suggests that object-oriented and agent-oriented technologies are the basic methodologies for designing virtual reality systems. In that book vision computing, neuron computing, and evolutionary computing are regarded as the primary advanced virtual computing models. Calderoni and Marcenar argue that many artificial life research areas resort to agent-based simulation, so they try to design a generic platform for allowing scientists to easily build simulation environments [2]. A similar effort has been made by Reignier et al, who also constructed a virtual reality multi-agent platform called *AReVi* [3].

This paper applies artificially intelligent technologies to the virtual reality worlds, and proposes an agent-based approach to virtual reality from the standpoint of software simulation system design. Agents are constructed by using an object-oriented method. At the same time we also present a set of underlying computing models such as neural networks, genetic algorithms, expert systems, and plan managers to support agents. A spaceship game illustrates the interactions among environment, agents, and underlying computing models.

In Section 2 we propose an abstract framework of virtual environments. All of the other work will be based on it. Section 3 presents those underlying computing models in detail. A spaceship game in Section 4 will demonstrate how to use the framework and computing models to develop a virtual environment simulation system. Section 5 concludes this paper.

2. Abstract Framework of Virtual Environments

To explain how to construct agents and underlying computing models using the object-oriented method, we first propose an abstract framework of virtual environments as shown in Fig. 1. It includes agent modules and environment simulation modules (environment, geometric objects, and real objects are represented by the classes *world*, *geometric_object* and *game_object* respectively). These are the rules we abide by in designing those components: (1) The agent module and environment simulation modules are connected only loosely. (2) The agent module should be extensible and adaptable to any new environment. (3) The environment simulation modules are platform independent; they can be used on multiple platforms such as Windows, Unix X Window, OpenGL and Macintosh.

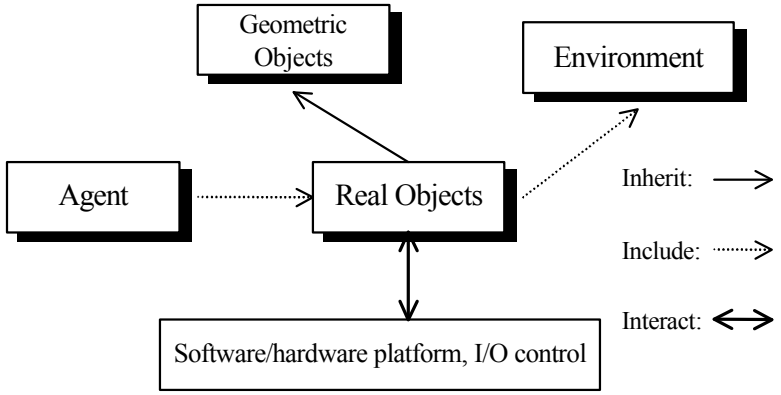

Fig.1. An Abstract Framework of virtual environments

Before giving the detailed class descriptions involved in those computing models, we first simply describe some classes called in them. The class *geometric_object* mainly deals with geometric models. The class *game_object* is a subclass of *geometric_object*. In addition to inheriting the data and methods from the base class, it also includes specific application data, the adding-agent method, the calling-agent method, and the changing self-status method.. The instance of the class *game_object* can refer to one or more instances of the class *Agent*.

The class *World* is a container class of the class *game_object.* An instance of it can contain one or more instances of the class *game_object*. The class *Agent* is a virtual base class. The main functions of it are to provide the interface for inherited classes and executing the agent's function *run_agent,* which supports the dynamic and real time behaviors of the simulation environment systems.

3. Computing Models

3.1 NEURAL NETWORKS

Neural networks perform well in the field of pattern recognition, especially in eliminating noise data and recognizing incomplete patterns [4][5]. In virtual environments these patterns are

sensed by data and triggered by control data. The trained neural networks can be used to match the input and output data patterns.

In section 4 we use a trained neural network to compute the spaceship's speed. It uses the relative position to nearby objects as its input value. The description of the class *Neural* is shown in Fig. 2; it represents one simple neural network.

```
static float Sigmoid(float z);
static float SigmoidP(float z);
class Neural {
public:
    Neural(int input_size, int hidden_size, int output_size);
    Neural(const char *network_file_name);
    float train(int num_training_cases,
        float *training_inputs, float *training_outputs);
    void save(const char *file_name);
    void load(const char *file_name);
    void recall(float *inputs, float *outputs);
private:
    int   num_inputs;
    int   num_hidden;
    int   num_outputs;
    float     w1[MAX_INPUTS][MAX_HIDDEN];
    float     w2[MAX_HIDDEN][MAX_OUTPUTS];
};
```

Fig. 2. Class Neural

The function *Sigmoid* limits the input values of all neural nodes to the specified bound, for example (-0.5, 0.5). It and *SigmoidP* are used to adjust the connection weights of networks during the training procedure. The first constructor function sets the number of neural nodes in input, output, and hidden layers, and it uses random numbers to set the connection weights of networks. The second constructor function constructs the network by using one file that contains the connection weights. The *Train* function is used to train the network and constantly change the weights until the sum of errors for all training examples is minimized. The *Save* and *load* functions are used to save and load the weights of trained networks. The *Recall* function can produce a set of outputs according to a group of neural nodes' inputs.

3.2 GENETIC ALGORITHM

A genetic algorithm is a search algorithm based on natural selection and natural inheritance. It applies a fitness function to each chromosome of the population. The chromosomes that have lower fitness are replaced by the new chromosomes that are generated by crossover and mutation functions among fit chromosomes. Each chromosome includes a specific number of genes. Often real applications represent a gene by only one bit. The two most frequently used operators of genetic algorithm are (1) the crossover operator, which generates new chromosomes through exchanging some genes of two fit chromosomes to replace the unfit chromosomes, and (2) the mutation operator, which changes some genes of those unfit chromosomes.

```
typedef unsigned char gene;
typedef float (*fitness_function)(int chrom_index, int chrom_size,
                        unsigned char *chrom);
class Genetic {
public:
     genetic(int chrom_size, int num_chrom, fitness_function ff);
     genetic(const char *genetic_input_file, fitness_function ff);
      void      do_mutations();
      void      do_corssovers();
      void      sort_by_fitness();
      void      save(const char *file_name);
      void      load(const char *file_name);
protected:
      int   chromosome_size, num_chromosomes;
      gene      *population[MAX_CHROMOSOMES];
      float     fitness_values[MAX_CHROMOSOMES];
      int   p_chrom[MAX_CHROMOSOMES];
};
```

Fig. 3. Class Genetic

Section 4 will introduce a method of using chromosomes to save the spaceship's control strategies in a virtual environment as well as a method of using a genetic algorithm to control the spaceship. The description of the class *Genetic* is shown as Fig. 3. It represents one simple genetic algorithm. The function *do_crossovers* and the function *do_mutations* create new chromosomes in the population. The constructor function of the class *Genetic* uses random numbers as gene values to initialize the chromosomes. The function *sort_by_fitness* calls each chromosome's fitness function, sorts all the chromosomes according to the return values, and saves them in *p_chrom*. The functions *save* and *load* are used to save and load the code of each chromosome in the population.

The instance of class *Genetic* must include a specific number of chromosomes and their related fitness functions. When using the class *Genetic* in Fig. 3, the method of encoding the chromosomes and providing a fitness function for each of them should be determined in advance.

3.3 EXPERT SYSTEMS

Expert systems separate domain knowledge from control. We implement them using a set of "if-then" rules. The separated inference engine interprets and executes those executable rules. It determines the executable rules and uses data of the current status. If the precondition part of a rule can be satisfied, then this rule will be regarded as executable, and the subsequent part of it can be executed by the inference engine. In order to realize a more applicable expert system, multiple knowledge representation methods, monotonic reasoning, fuzzy knowledge handling, and parsing analysis of semantic networks should also be supported [5][6].

In virtual environments an expert system is usually customized to a specific object. However, all instances of this group of objects can share the same instance of an expert system because an expert system is oriented to one group of objects rather than one specific object. Unlike the classes *Neural* and *Genetic*, the class *ExpertSystem* needs the knowledge of the classes *World* and *game_object*, but it will not be changed when the classes *World* and *game_object* have some changes.

```
typedef int(*rule_function)(World*, game_object*);
class ExpertSystem {
public:
    ExpertSystem();
   void add_rule(const char *title, rule_function rf, int priority);
    int excute_highest_priority_rule(World*, game_object*);
    void excute_all_rules(World*, game_object*);
private:
    const char *rule_name[MAX_RULES];
    int      num_rules;
    rule_function  rules[MAX_RULES];
    int     priorities[MAX_RULES];
    const char *titles[MAX_RULES];
};
```

Fig. 4. Class ExpertSystem

Class *ExpertSystem* is described in Fig. 4. The function *add_rules* will add rules to the instances of class, and the rules must match the function prototype

*typedef int (*rule_function) (World*, game_object*);*

A rule should provide at most two functions: (1) The first determines whether the precondition part of the rule matches current status; if it does not, it returns to *FALSE* immediately. (2) If the rule is executable, the subsequent part can be used to change the status of the instances that are presented by the classes *World* and *game_object*.

3.4 PLAN MANAGER

Plan managers are designed to handle plans that are integrated by three functions: (1) the condition judgment function that determines whether the precondition part of the plan can be matched with the current status; (2) the plan execution function that executes at each time step while the plan is active. This terminates when the termination judgment function returns to a *TRUE* value; (3) the termination judgment function that determines if the plan should be terminated.

Although the appearance of the class *PlanManager* is very similar to the class *ExpertSystem*, some differences still exist between them. Expert systems don't save their behaviors to the memo. These behaviors refer to the procedures executed between the previous time and the present time of the calling function to execute the highest priority rule or excute all rules. In contrast, plan managers do save the behaviors. Plan managers manage all status changes during the executing cycle. If specific actions are expected to be executed immediately by an agent, then the class *ExpertSystem* should be selected. If a sequence of actions is expected, then the class *PlanManager* is the appropriate choice.

```
typedef int(*plan_function)(World*, game_object*, int);
class PlanManager {
public:
    void    add_plan(const char *plan_title,
                plan_function pre_condition_rule,
                plan_function plan_execution,
                plan_function termination_condition);
    int   run_plan(World *w, game_object *g);
private:
    const char*   plan_name[MAX_PLANS];
    plan_function     pre_rule[MAX_PLANS];
    plan_function     run_rule[MAX_PLANS];
    plan_function     stop_rule[MAX_PLANS];
    int   num_plans, current_plan;
    int   run_state, time_step_count;
};
```

Fig. 5. Class PlanManager

The class *PlanManager* is described in Fig. 5. The function *add_plan* adds new plans to the instance of the class *PlanManager*. The function *run_plan* first judges if any plans are currently being executed. If not, each plan's condition judgment function is examined to judge if there is a proper plan to be executed.

4. Spaceship Simulation

4.1 GAME DESCRIPTION

There are three types of spaceships in the game: *PlayerShip*, *ProcessorShip*, and *Corsair*. *PlayerShip* has mines on asteroids, and a number of *ProcessorShips* can provide energy to *PlayerShip* in exchange for ore. The exchange process is proposed by *PlayerShip*, and *ProcessorShip* decides whether or not to exchange according to the amount of ore. If *ProcessorShip* has enough ore, it will not exchange with the *PlayerShip*. During the game, there are many *Corsairs*, and they keep stealing energy from the nearby *PlayerShip* and *ProcessorShip*. This game terminates when the *PlayerShip*'s energy is depleted.

4.2 NEURAL NETWORK NAVIGATION

A neural network is used to simplify the problem by controlling the spaceship's speed and acceleration in specific directions. It uses different input values to describe the directions of X, Y, Z, and -X, -Y and –Z. As the input and output values are restricted to (-0.5, 0.5), the positions of all other nearby spaceships should be expressed as input forms of the network. Then the real value outputs should be restored to the speeds and accelerations of the spaceships. The class *NavigationAgent* is used to navigate the *PlayerShip* and various *ProcessorShip*s. It is a derived class from the class *Agent* and can call neural networks, expert systems, and plan managers. The class *NavigationAgent* is shown in Fig. 6.

```
class NavigationAgent : public Agent {
public:
    NavigationAgent(World *w, game_object *g);
    virtual void run_agent();
    ExpertSystem *get_expert_system();
    PlanManager *get_plan_manager();
protected:
    Neural *neural_net;
    ExpertSystem *expert_sys;
    PlanManager *plan_manager;
    World *world;
    game_object *game_obj;
};
```

Fig. 6. Class NavigationAgent

The function *run_agent* first computes the distances between the spaceship and the nearest four objects (including other spaceships and asteroids) and converts the values to the input forms for the network. It then changes the speeds and accelerations according to the suggestion of the network. After that it calls the plan manager to judge if any executing plans are required. If there is no new plan at that point, the function *execute_highest_priority_rule* of the expert system will be called.

4.3 EXPERT SYSTEM CONTROLLING PROCESSORSHIP

The class *ProcessorShip* represents the data and actions of *ProcessorShip*. It is a subclass of the class *game_object* and is shown as Fig. 7. The construction function of the class *game_object* specifies the shape and size for the construction of objects. After that, an instance of the class *NavigationAgent* is created to navigate *ProcessorShip*. At the same time, three rules are added to the expert system that are referred by the newly created navigation agent:

```
nav_agnt = new NavigationAgent(w, this);
add_agent(nav_agnt);
ExpertSystem *es = nav_agnt->get_expert_system();
es->add_rule("rule for fuel use", rule_for_fuel_use, 200);
es->add_rule("rule for motion correction", rule_for_motion_correction, 100);
es->add_rule("rule for avoidng Corsair ships", rule_for_avoiding_Corsairs, 30);
```

The function *rule_for_fuel_use* reduces the energy value at each time step. The function *rule_for_motion_correction* judges the environment status. If *ProcessorShip* is far away from the asteroid center, it will adjust its speed to approach the center. The function *rule_for_avoiding_Corsairs* will move *ProcessorShip* in the opposite direction when *Corsair* is near.

```
class ProcessorShip : public game_object {
public:
    ProcessorShip(World *w, const char * my_name,
                  int appearance_type; int size);
    void set_R4(float new_val);
    float get_R4();
    void set_Magnozate();
    float get_Magnozate();
    int want_to_trade(float trade_radio, int amount);
private:
    float R4, Magnozate;
    NavigationAgent *nav_agnt;
};
```

Fig. 7. Class ProcessorShip

4.4 EXPERT SYSTEM AND PLAN MANAGER AID PLAYERSHIP

Like the class *ProcessorShip* that controls processor ships, the player ship can be aided by class *PlayerShip,* which is described in Fig. 8. It is also a subclass of the class *game_object*. The construction function specifies the shape and size of objects to be constructed. Then an instance where the class *NavigationAgent* will be created to aid *PlayerShip*. At the same time, five rules will be added to the expert system, referred by the newly created navigation agent:

```
nav_agnt = new NavigationAgent(w, this);
add_agent(nav_agent);
ExpertSystem *es = nav_agent->get_expert_system();
es->add_rule("rule for fuel use", rule_for_fuel_use, 200);
es->add_rule("rule for returning to field", rule_for_moving_to_asteroid_field, 100);
es->add_rule("rule for mining asteroid", rule_for_mining_asteroid, 90);
es->add_rule("rule for avoiding Corsair", rule_for_avoiding_Corsairs, 10);
es->add_rule ("rule for trading", rule_for_trading, 92);
```

The function *rule_for_fuel_use* reduces *PlayerShip*'s energy at each time step. If *PlayerShip* is far away from the asteroid center, the function *rule_for_moving_to_asteroid_field* will adjust its speed to approach the center. The function *rule_for_mining_asteroid* searches for the nearest asteroid to mine ore. The function *rule_for_avoiding_Corsairs* finds the nearby Corsairs and sets *PlayerShip*'s speed to take it away from the Corsairs. The function *rule_for_trading* searches for the nearest processor ship, trying to exchange ore for energy.

```
class PlayerShip : public game_object {
public:
    PlayerShip(World *w, const char *my_name,
                int appearance_type, int size);
private:
    float R4;
    int Magnozate;
    NavigationAgent *nav_agent;
    int trading_flag;
    int mining_flag;
    int automatic_flag;
};
```

Fig. 8. Class *PlayerShip*

Except for the rules, the construction function of *PlayerShip* adds two plans to the instance of *PlanManager*.

```
PlanManager *pm = nav_agnt->get_plan_manager();
pm->add_plan("get ore", test_low_ore, go_to_asteroid, end_go_to_asteroid);
pm->add_plan("trade ore", test_trade_ore, go_to_pr_ship, end_go_to_pr_ship);
```

These are the three plan functions for mining ore: (1) *Test_low_ore* creates a plan to move *PlayerShip* to mine ore from an asteroid when judging that PlayerShip's ore is low. (2) *Go_to_asteroid* moves *PlayerShip* to the asteroid. (3) *End_go_to_asteroid* terminates the plan if *PlayerShip* has approached the asteroid or the executing time of the plan is over the threshold.

4.5 CHANGING CORSAIR STRATEGY DYNAMICALLY BY GA

Corsair adopts a strategy different from *PlayerShip* and *ProcessorShip*. Each strategy is decided by three kinds of status parameters: (1) One where *Corsair* moves straight ahead or zigzags. (2) One where Corsair converts energy slowly from asteroids or steals energy quickly from *PlayerShip* or *ProcessorShip*. (3) One where *Corsair* moves slowly or quickly (and thus rapidly consumes energy).

Using 0 and 1 for the determination of the above three status parameters, we have eight possible strategies for each Corsair. Creating a chromosome of three genes for each Corsair satisfies the demand of the problem. The class *Corsair* is a subclass of the class *Genetic*. It is shown as Fig. 9. Only one instance of the class *Corsair* in our example needs to be created. This instance will include many chromosomes, and each chromosome represents the control strategy of one *Corsair*. The list in *game_object* has specified the appearance and status of each Corsair.

```
class Corsair : public Genetic {
public:
    Corsair(World *w, const char *restore_file);
    void move_all();
    void evolve_population();
    static float get_R4_fuel_level(int chromosome_index);
    static float set_R4_fuel_level(int index, float amount);
private:
    World *world;
    game_object *Corsair_object[POPULATION_SIZE];
    static float R4_fuel_level[POPULATION_SIZE];
};
```

Fig. 9. Class Corsair

The function *move_all* changes all *Corsair*s' positions. The function *evolve_population* makes the population evolve periodically through the following three steps: (1) Sort all the chromosomes according to the fitness functions. (2) Apply the mutation operation to the unfit chromosomes. (3) Apply the crossover operation. For these Corsairs, we can simply take their energy values as fitness functions. If the energy of one Corsair is inadequate, its strategy is considered unfit.

5. Conclusion

The paper focuses on the realization of an agent and its underlying computing models based on object-oriented methodology. The ideas presented here can be applied to various virtual reality environment simulations and ordinary intelligent applications. Furthermore, the reusable and extensible classes proposed in this article such as the environment, the geometric objects, the real objects, the agent, the neural network, the expert system, the plan manager, and the genetic algorithm can be applied directly to related applications. However, those computing models are only simple prototypes because the main purpose of this paper is not to develop specific computing models. In practice, based on the demand of specific domain problems, new classes can be derived from those classes or directly extended to improve their functionality. In addition, this article mainly depends on the agents' ability to solve problems, and more complex problems can be solved if cooperation, coordination, and negotiation mechanisms among agents are introduced [10].

REFERENCES

[1] Wang Chengwei, Gao Wen, and Wang Xingren. *Theory, Implementation and Application of Virtual Reality Technology*, Tsinghua Press, Beijing, 1996.

[2] Calderoni, S., and Marcenac, P. "MUTANT: a multiagent toolkit for artificial life simulation". In: Proceedings of Technology of Object-Oriented Languages, TOOLS 26, 1998, pp. 218 –229.

[3] Reignier, P., Harrouet, F., Morvan, S., Tisseau, J., and Duval, T. "AReVi: A Virtual Reality Multiagent Platform", J.-C. Heudin (Ed.): Virtual Worlds 98, LNAI 1434, 1998, pp. 229-240.

[4] Chen Zhaoqian, Li Hongbing, Zhou Rong, and Chen Shifu. "Analysis and Improvement of FTART Algorithm", Journal of Software, Vol.8, No.4, 1997, pp. 259-265.

[5] Masters T. *Advanced Algorithms for Neural Networks*, New York: John Wiley & Sons, Inc., 1995.

[6] Koza, J. R. *Genetic Programming*, Cambridge, MA: MIT Press, 1992.

[7] Li Hongbing, Chen Zhaoqian, and Chen Shifu. "An Object-Oriented Expert System Development Tool", Journal of Software, Vol. 8, Supplement, 1997, pp. 305-311.

[8] Medsker L R. *Hybrid Neural Network and Expert Systems*, Boston: Kluwer Academic Publishers, 1994.

[9] Ghallab M, Milani A. *New Directions in AI Planning*, Netherlands: IOS Press, 1996.

[10] Greg M P, O'Hare Nick R Jennings. *Foundations of Distributed Artificial Intelligence,* New York: John Wiley & Sons, Inc., 1996.

BIOGRAPHIES

Li Hongbing is on the academic staff of the State Key Laboratory for Novel Software Technology, Nanjing University. He received B.Sc., M.S., and Ph.D. degrees from Nanjing University, all in Computer Science. His current research interests include fuzzy systems, machine learning, neural networks, genetic algorithms, expert systems, multi-agent systems, and intelligent CAD. He has published more than 20 papers. He is now visiting the Department of Computing at the Hong Kong Polytechnic University as a research assistant.

Contact information:

Li Hongbing
Department of Computing
The Hong Kong Polytechnic University
Kowloon, Hong Kong
Phone: 852-27667312
Fax: 852-27740842
Email: mailto:cshbli@comp.polyu.edu.hk

Meng Bo is a Ph.D. student of the Computer Science and Technology Department, Nanjing University. He received an M.S. degree from Nanjing University in computer science. His current research interests include intelligent CAD and distributed artificial intelligence.

Contact Information:

Meng Bo
Computer Science and Technology Department
Nanjing University
Nanjing, P. R. China, 210093
Phone: 86-25-3593163
Fax: 86-25-3300710
Email: mailto:russell@263.net

Chen Shifu is a professor of the Computer Science and Technology Department, Nanjing University, once served as the Department Head. Now he is the Deputy Chairman of China Machine Learning Association and a senior fellow of the China Artificial Intelligence Association. His research interests include Machine Learning, Neural Network, Knowledge Engineering, Intelligent Agent, and Intelligent CAD.

Contact Information:

Chen Shifu
Computer Science and Technology Department
Nanjing University
Nanjing, P. R. China, 210093
Phone: 86-25-3593163
Fax: 86-25-3300710
Email: mailto:chensf@nju.edu.cn