A Game Engine Framework Utilizing Web 3D Technology

Xizhi Li
The CKC honors school of Zhejiang University
Computer Science Department
email: LiXizhi@zju.edu.cn

Abstract:
Modern computer game engine has evolved to become a complete suite of virtual world constructing tools and runtime environment. The latter is usually a tightly integrated framework of 3D rendering engine, scripting engine, physics simulation and networking. This framework design as well as individual component implementation decide the general type of games that could be composed by it. Web3D technology such as X3D language is exploiting a new possibility of expressing networked virtual environment that is as distributed as web pages and more interactive than just hyperlinks. Bringing networked virtual game worlds and game world logic to the open Internet will spawn new types of computer games. This article presents a game engine framework called ParaEngine for developing games based on distributed game world data and logic. The enabling technique lies in the role of its scripting engine which is called Neural Parallel Language or NPL. NPL makes it possible to compose game world logic (which might physically exist on arbitrary places on the Internet) in a network transparent manner; X3D plays a descriptive role in dynamic scene rendering and association of interactive scene objects with NPL neuron file. Unlike general purpose X3D or VRML visualizer, X3D file is not directly executed but serves as mental imagery and multimedia elicitations of a distributed Neural Network functioning on the Internet. This framework makes it possible to compose and run active and evolving game world and its logics spanning a network. The implementation is illustrated in an Internet RPG game demo called Parallel World.

Key words: game engine, NPL, X3D, Web3D

1 Introduction

Modern computer game engine has evolved to become a complete suite of virtual world constructing tools and runtime environment. The latter is usually a tightly integrated framework of 3D rendering engine, scripting engine, physics simulation and networking. The balance of efficiency and flexibility is the primary issue that is weighed constantly in these many different places in an engine designer's mind. It is usually such compromises drawn by the designer that determined the characteristics of the engine and hence the type of games that could be composed by it. Scripting is the symbol of flexibility and has become ubiquitous in modern computer game engines. Scripting alone means two things: (1) script files are automatically distributed and logics written in a script can be easily modified; (2) script code may be generated by dedicated visual language and software tools. In a computer game, almost all kinds of static data and most dynamic logic have text-based presentations outside the hard core of its engine. The adoption of scripting technology makes level design or game world logic composing easier than ever.

Data exchange on the Internet is also largely text-based. An entity on the Internet with or without computing capabilities automatically becomes a global resource and can be referenced by other resources. A great deal of web technologies and recommended standards have been recently proposed to make the web more and more meaningful, interactive and intelligent. As envisaged by web3d, web service and ubiquitous computing research, etc, software applications in the future are highly distributed and cooperative. Computer games and other virtual reality applications are likely to become the most pervasive forces in pushing these web technologies into commercial uses. It is likely that one day the entire Internet would be inside one huge game world. However, two related issues must be resolved first, which are distributed computing and visualization.

While some current effort on Semantic Web/Grid tells a computer program exactly what to compute and visualize on the Internet, there still lacks formal approaches on telling it how to compute or visualize. Web3D technology such as X3D language is exploiting a new possibility of expressing networked virtual environment that is as distributed as web pages and more interactive than just hyperlinks. X3D code generally describes a tree hierarchy of nodes with routes or stimuli specified for their input fields. Nodes can be associated with script files or other Internet assets. Script files contain logics and hence logics can be distributed on the Internet (the latter needs special runtime environment support where scripts are situated). Although most X3D applications involve only a static assembly of dynamic scene data from one or several file servers on the Internet, the existence of scripts and dedicated runtime environment on both client and server makes it possible to construct active virtual environment spanning the network. However, X3D alone is not sufficient or in some cases suitable to handle all distributed computing and visualization tasks required in a computer game.

Bringing networked virtual game worlds and game world logic to the open Internet will spawn new types of computer games. This article introduces a game engine framework called ParaEngine for developing games based on distributed game world data and logic. The enabling technique lies in the role of its scripting engine which is called Neural Parallel Language or NPL. NPL makes it

possible to compose game world logic (which might physically exist on arbitrary places on the Internet) in a network transparent manner; X3D plays a descriptive role in dynamic scene rendering and association of interactive scene objects with NPL neuron file. Unlike general purpose X3D or VRML visualizer, X3D file is not directly executed but serves as mental imagery and multimedia elicitations of a distributed Neural Network functioning on the Internet. This framework makes it possible to compose and run active and evolving game world and its logics spanning a network. The implementation is illustrated in an Internet RPG game demo called Parallel World. In the following sections, we will first present the general framework of ParaEngine and a demonstration of constructing a distributed game world with it, then we will cover NPL programming paradigm in further details.

2 ParaEngine Framework

A complete description of the game engine is not in the scope of this article, but can be found here[http://www.lixizhi.net/projects.htm#paraengine]. This section will focus mainly on the following relevant aspects: (1) how the major modules of the game engine (graphics rendering, I/O, scripting, physics, AI, networking) relate to each other, (2) how computational tasks (path-finding, collision detection and response, intelligent creature strategies, game world logics or stories, etc) required by the game engine are allocated to one of the three possible programming choices namely script, extended binary code, and Engine core code, (3) how multiple instances of the game engine function on the Internet to exhibit one huge game world.

2.1 Game World Logic

One of the major tasks of a computer game engine is to offer language and tools for describing Game World Logic in an engine digestible format. Usually the logic of the entire game world can be further partitioned into three subcategories of programming: (1) script programming which is most flexible and can be distributed in many files (2) C++ programming which extends basic functionalities already provided in the engine core, (3) Engine programming which is fixed with the release of the engine. Our objective in designing ParaEngine is to let Scripting do as much as possible. Please see Figure 1. The color in it denotes the assignment of programming category to computational tasks in composing game world logic.

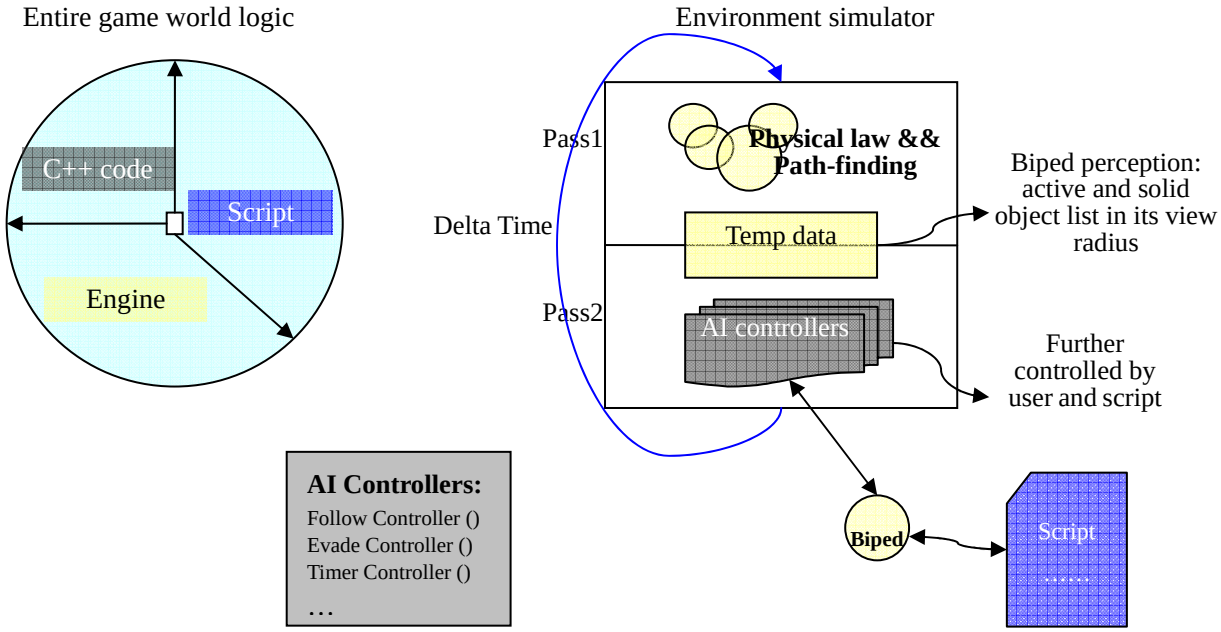


Figure 1. Game world logic in ParaEngine

In the figure, AI controllers are C++ code based AI modules that should not be confused with Script-based AI. AI controller is the best choice whenever performance is most critical. They can be assigned to Biped Scene Object. The association of biped object and AI controller is entirely arbitrary and reassignment is also allowed during game play through scripting. For example, in the game demo Parallel World, a "Follow" Controller and "Evade" Controller were written in C++ code, so that when assigning a NPC creature to both of them, it will have the ability to follow automatically a moving target (avoiding all obstacles in its view perception) as well as evade a target according to its unit type (melee or ranged). Both the AI controller assignment and controller-control are available as host APIs in the script language used by the game. A useful conclusion of this section is that although we hope to build everything from scripts, there still exist some places where current scripting technology is not eligible or suitable to use. In other words, languages such as (VRML + Java) might alone be capable of static and/or interactive 3D information visualization on the web, it is not sufficient, though, to implement all game world logics required by a modern Internet computer game in a efficient way. This is one reason why a game engine framework like ParaEngine is still needed to build the aforementioned type of games.

2.2 Timing and Networking

In ParaEngine, several global timers are used to synchronize engine modules that need to be timed. Figure 2 shows a circuitry of such modules running under normal state. The darker the color of the module, the higher the frequency of its evaluation.

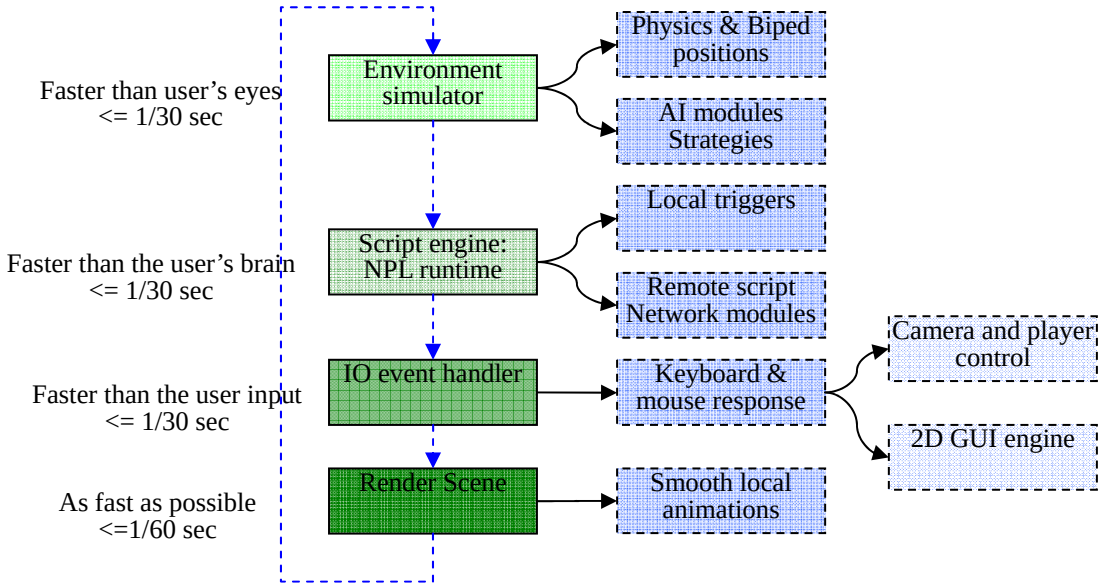


Figure 2. Timing and I/O in ParaEngine

In conventional Internet RPG (Role Playing Game) games, each game-world objects like a Non-Player Character (NPC), Player-controlled Character(PC), boxes, doors, weapons, or even terrains is associated with an abstract neuron. Both the timer(story-line), network command or the human players may stimulate some of its input fields. The stimuli might be generated by the game's physics, network or GUI engine. During each simulation cycle on a local game engine, it executes any activated neurons (a script code associated with a certain game object), which usually read their stimuli, compute according to its current state, generate new stimuli to other neurons and sometimes even take actions (also a script code). On the whole, the driving force of a game engine is the constant firing of stimuli in a neural network constructed by scripts.

Game-engine practitioners have used scripting technology to add soft computing capabilities to a variety of their engine modules; so that commercially released games will still enjoy a certain degree of online-reconfiguration. Unfortunately, there has been no unified approach for solving this problem. Instead, most game engine explicitly implement a network module[9] which usually relies on single Clients/Server (CS) architecture and a single session at any given time. A single one-table central database is used to hold all the status of its game entities and all events and triggers in the virtual game world resides only on the local computer. ParaEngine overcomes these limitations and provides further flexibilities by means of using NPL in constructing game world logics.

2.2.1 The Absence of Network Modules and NPL

In ParaEngine (Figure 2), no explicit network modules can be found. Instead, networking is implicitly specified in script. As argued previously, scripts are flexible entities that are distributed over the network; and with proper runtime support at the place where scripts are situated, complex network logics can be described. Explicitly modeling network logic in distributed script files are

not a new idea [17][18]. For example, there are several standards such as DIS or Distributed Interactive Simulation in X3D language [6], as well as some ad hoc approaches in a few computer game engines[19]. In the ParaEngine framework, we goes one step further. Not a single line of networking code needs to appear in any script files, yet network logics can be described in a run-time transparent manner. To get a quick idea of how this can be done, we can image two script files A and B which represent two neurons with a message channel from A to B. If A and B has been deployed in a single runtime (i.e. one computer), then whenever A is activated it will route a message directly to the input field of B and no network communication occurs. Now if A and B has been deployed in two different runtimes (i.e. two networked computers), the physical location of A and B automatically tells the runtimes that network communication is needed to route the message. Therefore, the code of script file A and B stays the same for both situations. To make this possible, each neuron file is turned into a hierarchically named network resource which the language runtimes maintain automatically.

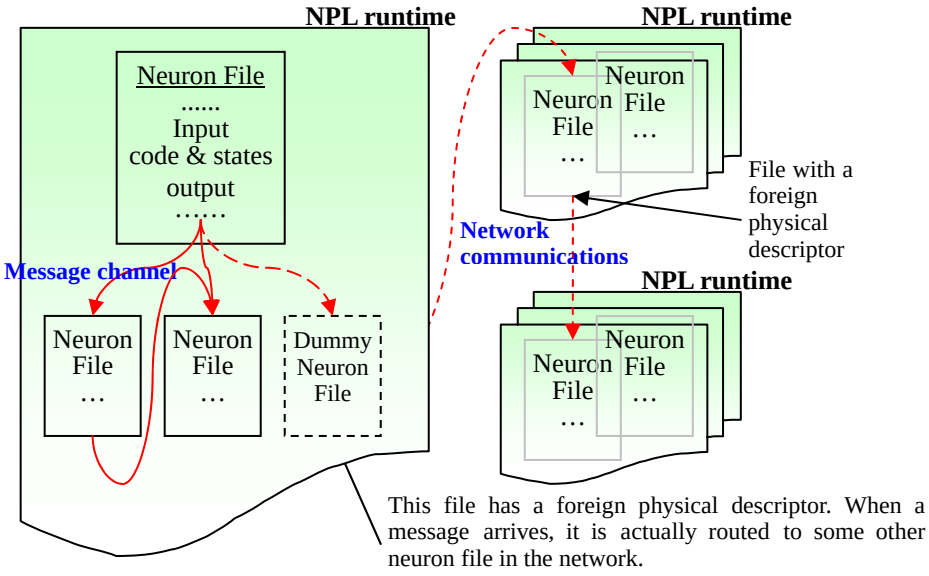


Figure 3. NPL: the big picture

In Figure 3, NPL language runtime is embedded in each local game engine and manages communication between neuron files. The big picture is given below. When writing or debugging a neuron-files network, programmers do not need to concern about the actual physical environments where these files will be eventually deployed (e.g. a huge distributed virtual game world might exist over 1000 servers on the Internet and would be ever expanding). Instead, tasks concerning the actual hardware, communication protocol, certification and ontology (such as for exchanging meaningful information) will be specified not in program code, but in the visual Compiler And Runtime Environment (CARE) of the NPL language. For example, it is the task of CARE to distribute/deploy an integral neuron-file network written in NPL to separate locations (runtimes) of the physical network. In theory, the only atomic structure that is unbreakable by CARE is a single Neuron file. Details of NPL will be given in Section 3 after the introduction of the game engine framework.

2.3 Composing distributed game world

This section shows the basic steps of composing distributed game world using the proposed game engine framework and Neural Parallel Language. There are many ways by which a game world can be composed. A game world consists of graphical models, animation models, terrain, AI creatures, plots and sequences, etc. It is difficult to have all of them fit into one world editor, not to mention the various ways that the same thing can be created. Table 1 shows a few of them as implemented in our ParaEngine.

Table 1 Game world composing tools in ParaEngine

Genre	**Tools and methods**	**Product digestible by the engine**
graphical models	3dsMax + exporter	X file, MDX file, *.tga, *.bmp
animation models	3dsMax + exporter MDX + exporter	X file, MDX file.
Terrain and scene	3dsMax →VRML(*.wrl) → Script converter	*.npl (NPL script file), *.wrl (VRML)
AI creatures	C++ code + NPL scripts	CAIModuleBase derivatives, *.npl
Sequences (Movie)	Visual script maker or Hand-written script	*.npl (script file)
Plots	Hand-written script With special visual editor In-game editing	*.npl (NPL script files to be deployed on the Internet or just the local runtime)

Let us suppose art elements have been made by the artists, regardless of what ever third-party tools (3Dsmax, Photoshop, etc) may be used. This can be tremendous work. Fortunately, these file based assets can be easily shared on the Internet, which the original VRML standard already achieved. What we will focus here, however, is to compose distributed game world out of them, writing game stories and designing logics of the game.

The novelty of the proposed game engine framework lies in that: the entire game world is viewed as existing inside a huge brain spanning the Internet. Visual presentation such as a game scene or movie is but mental imagery and multimedia elicitations of the distributed Neural Network functioning on the Internet. By contraries, in conventional game world composing techniques, the virtual world is modeled in a (possibly networked) 3D space with scripts (some logic) attached to interactive scene objects. Figure 4, shows the differences. Both models need to map world objects to neuron script files, but the difference is that when in execution who came first: the static 3D space or active neural network. In our vision, future software would be designed under the premise that all front end software run in a distributed environment and (co)operate in a manner similar to neural networks. The high level programming paradigm proposed by NPL matches this trend. And we believe it is more flexible and natural to let the active neural network elicit any visual scene presentations, rather than vice versa (see Figure 4).

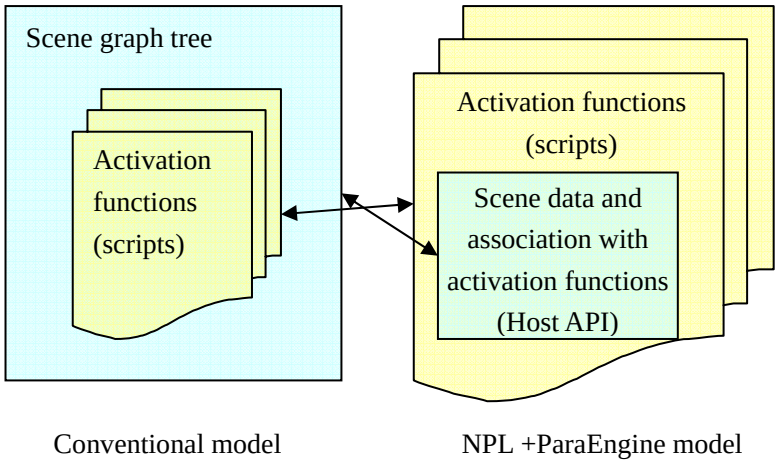


Figure 4. Execution model comparison

2.3.1 An analogy to Imagery

In order to better illustrate the relationships between the game engine, world logics, and the Neural Networks; I will draw an analogy of the programming paradigm of NPL to a theory (hypothesis) of the human brain concerning Imagery [2] on cognitive science. In my own version of this theory, it states that human imagination and visual/auditory perceptions are in essence the same thing in our conscious mind, and that they are both the input and output of the unconscious mind which does the work of recognition, memorization and deduction. The cycle of imagination and the subconscious forms mostly a closed loop when we are asleep, and a biased loop (by what we perceive) when we are awake. See Figure 5.

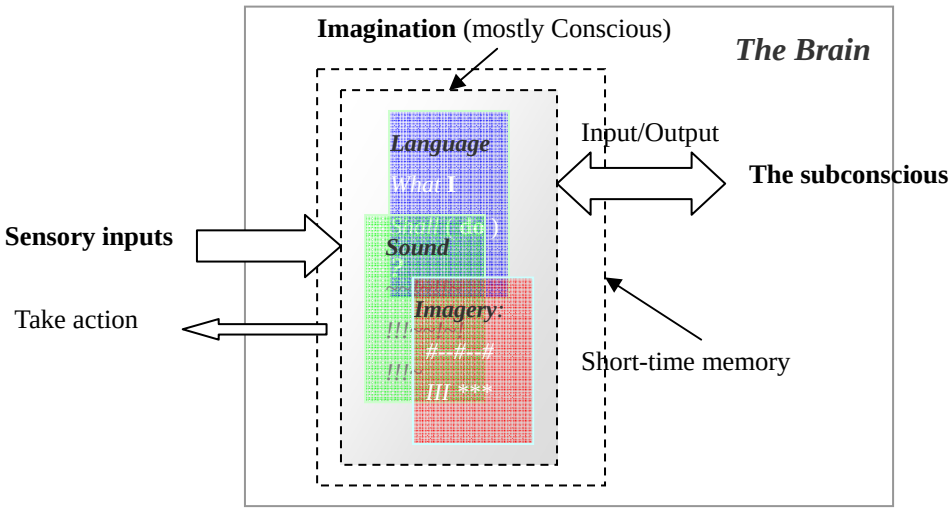


Figure 5. Human Brain: The Imagery-subconscious loop

Metaphorically speaking, the Imagination can be thought of as a multimedia, virtual reality "theatre" [7], where stories about the body and the self are played out. These stories,

- are influenced by the present situation according to perception,
- elicit the subconscious activities accordingly,

- and thereby influence the decisions taken by action.

In the engine framework, we use Neural Parallel Language (NPL) to construct neural network that maps to both the imagination and subconscious part of the human brain. However, the imagination represented by the state of a Neural Network can not be visualized by itself. Instead the responsible neurons must tell the Game Engine game-related information upon activation whenever an internal state has been reached. This is done through a set of game specific API called Host API or by feeding to the engine an X3D (VRML) file which defines the visual elicitation. We use a dual programming language model in our architecture, please see the following sections for more information. The game engine will then present the Imagination in cutting-edge multimedia forms to the user. The user might interact with these objects. The game engine then translate such interaction (sensory inputs) to valid neuron stimuli. And the whole system will be functioning as seen in Figure 6 with text, buttons, graphics and sounds.

Figure 6. Screen shots from *Parallel World* game

Everything in the figure is mental elicitation of a neural network constructed by NPL. The neural network may be deployed on as many computers as the number of neurons used in composing the game world itself. *Parallel World* game is an Internet RPG game we constructed using our game engine. Players might walk around, talking with other NPCs(None-player-characters), complete complex tasks all in an continuous infinitely-large distributed game world. It is like browsing 3D web pages[6] on the Internet, however, it is more interactive, purposeful and fun.

2.3.2 A streamline of composing game world

One possible streamline of constructing game worlds is given here. We only show the tools and steps that are used during the time of composing the demo game. There are many other (perhaps easier) ways to do it if better helper tools have been available.

Step1: Build all game assets.

These include mesh models, biped animation sequences, sounds and textures. Or one can collect URI of such models if they are Internet resources. Instead of using URI or file path name for their instantiation or elicitation later in scripts or VRML file, they must be given shorter names. This is done by using scripting and Host API of the ParaEngine. A sample code is given here:

```
function wrl_movie1_res()
  -- X file terrain 200*200
  ParaCreateAsset("MS", "terrain200", "xmodels\\terrainPH.x"); --terrainG
  --anim:[0]stand,[1]stand hit,[2]death,[3]Birth,[4]Spell EatTree,
  --radius:1.346700 meters
  ParaCreateAsset("MA", "tree0", "Doodads\\Terrain\\AshenTree\\AshenTree3.mdx");
end
```

A GUI tool have also been created for exporting groups of resource files into such script code. The above script is actually generated by this tool. Functions started with “Para” are Host APIs. Therefore, by referring to this script, all runtimes on the network know where to find these named resources.

Step2: Build 3D scenes.

3D Scenes are built using 3dsMax 6.0 as shown in Figure 7. The figure shows top view of a 200*200 meters scene dotted with all kinds of trees, stones, buildings and creatures. Each object might reference a named asset as defined in step1.

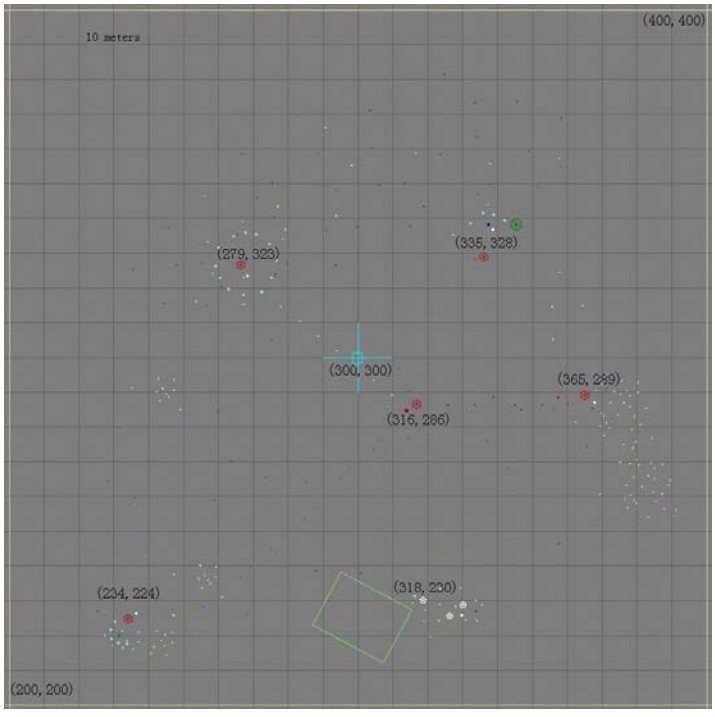
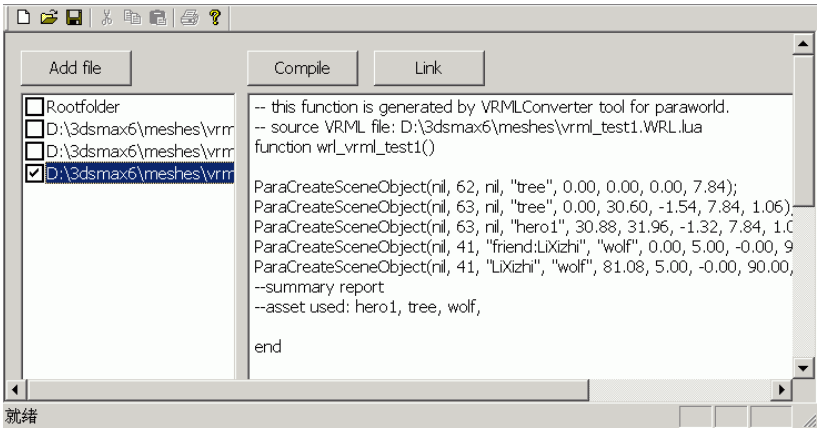


Figure 7. Composing game scenes

The scene is exported as a VRML file. These scenes (files) are visual entities that might be elicited by the neural network that will be constructed later. In our current implementation, VRML file needs to be further compiled to a more compact script format by a cross-compiler tool called VRML converter. Please see Figure 8. In future version, VRML or XML based X3D data will be directly used in neural network's visual elicitation. If the resulting script or X3D data is activated by a neural network, it would cause the game engine to present such imagery to the computer screen; in the meantime, the engine simulate the imagery which might cause new stimuli to be generated to the neural network. Hence this forms the imagery-subconscious loop previously mentioned in Figure 5.

```
#VRML V2.0 utf8
# Produced by 3D Studio MAX VRML97 exporter, Version 6, Revision 0.56
# MAX File: vrml_test1.max, Date: Mon Aug 02 00:59:13 2004
WorldInfo {
   title "Map for paraworld"
   info "LiXizhi"
}
DEF _tree_62_01 Transform {
   translation 0 0 0
   children [
      Shape {
         appearance Appearance {
            material Material {
               diffuseColor 0.102 0.6941 0.5804
            }
         }
         geometry Sphere { radius 7.843 }
      }
   ]
}
… Other text…
```



GUI tools for cross-compiling

Figure 8. VRML converter

Step3: Constructing Neural Networks using NPL

This is the most important and high-level part of distributed game world composing. In our framework, neural network defines the behavior of the game world and its logics. For example, one can create NPL neuron-file network that functions as message broad-casting portals, reactive agent (like RPC or remote procedure call), memory block, a sequence of cinematic, complex logic circuits with feedbacks, or a router or switcher, etc. Since details of NPL is presented in Section 3, advanced demonstration such as building client/server architecture will not be given here. Instead, Figure 9 shows a most simple demo: building a movie clip or cinematic with a neural network.

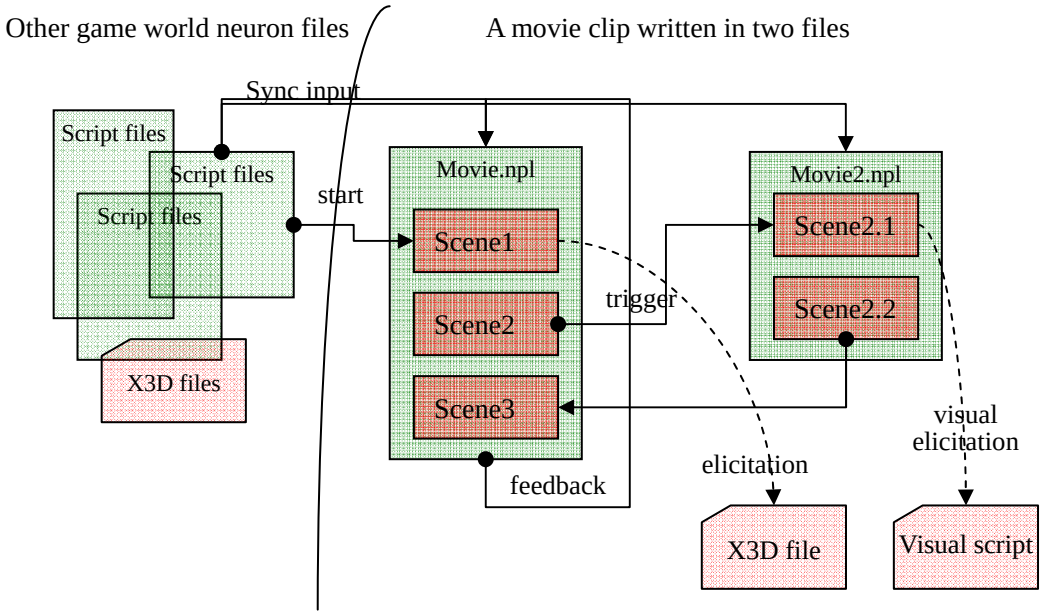


Figure 9. NPL demo: a simple cinematic

This network contains only two neuron files: Movie.npl and Movie2.npl. The small block inside the file box denotes a kind of input activation conditions. Each activation condition is symbolically named as SceneX. The *sync input* field is one method of synchronizing the beat (activation) of the two neuron files. The execution of the cinematic is rather like real movie shooting. Each SceneX will either elicit a complete visual imagery (by referring to a X3D file or visual scripts) or call Host API functions (of the game engine) to reset the camera position and target or tasks of biped scene objects. Figure 6 (the second and third picture) shows the execution effect of such neuron-file network.

Step4: Deploying neuron files on the physical network

Neural network deployment shows one benefit of using neural network based programming paradigm in distributed application development such as composing game world logic on the Internet. Neuron files can be arbitrarily distributed on the physical network. For example, in Step3, the two movie files can be deployed in one computer or two separate computers. In game world composing, designers can usually divide game world logic into two general categories: client side and server side. Client side neuron files will be shipped with the game; while server side will be distributed to many host computers. An alternative choice might be regarding each computer (peer) as both client and server. Information unspecified in the neuron source code (such as the topology of neuron files) will be dealt with by the visual Compiler And Runtime Environment (CARE) of the NPL language.

3 NPL: Neural Parallel Language

We have shown in previous sections the role of NPL in the game engine framework. This section will describe NPL as a *standalone* neural network based programming paradigm.

3.1 Introduction to a new programming paradigm

Software applications in the future will be highly distributed and cooperative. Current

programming paradigm is, however, difficult to use in designing and implementing software systems that will function gracefully in the dynamic network environment where they are situated. In other words, conventional programming gives programmers limited patterns in constructing next-generation computer software; and that the computing capabilities of the resulting software are automatically fragile to the changing environment. It is hardly possible for the web to function as one giant brain without soft computing patterns being applied to various levels of existing programming paradigm. We present a Neural-Network based programming paradigm that will change the way that programmers model and implement their software applications.

In my viewpoint, the compiling of code (that targets distributed environment) may also be carried out in a distributed manner (from command-line compiler to rich HCI enabled ones with network capabilities); the next generation high-level language may be able to express adaptive and distributed behaviors with its own language primitives; its compiler may be able to generate low-level code that runs on any part of the network; and its development environment may allow visualized design of any parallel-code and deployment-scheme. In other words, the coding and compiling process may both be carried out in a distributed manner and environment. This calls for a new language dedicated to this task and a new human-computer interface (HCI) adopted by its compiler and runtime environment.

With this vision, we proposed a unified approach of a neural network based programming paradigm called Neural Parallel Language (NPL). We implemented it together with a 3D computer game engine. The game engine is specially designed in corporation with NPL. Distributed software systems generally need to solve two problems: computing and visualization. The game engine serves as the visualization platform for the language. The combination shows promising results in composing distributed game world and logics.

3.2 Related works of NPL

The relevant works falls into two categories. One is from Neural Network Simulation researches on methodology (such as discreet event simulation paradigm [8]) and tools (GENESIS [3]). The other is from projects on Networked Virtual Environment[1] (Net-VR) (such as DIVE, NPSNET-V [4]) and languages such as X3D, which have overlapping visions with us. First of all, NPL is not another Neural Network Simulation toolkits. The essence of NPL is a neural-network based programming paradigm for general purpose (commercial) software development such as computer games. In NPL, traditional (fixed) neuron prototype as in NN simulator is expanded broadly to be more like a software agent. Current NPL implementation as in our game engine concerns more on the following issues: neuron input fields specification (Ontology), cross-network neural message transmission (XML) and security(NPL runtime certification), automatic neural network memory management (database support) and learning, neural network deployment, dynamic neural connection establishment on the network, etc. Secondly, NPL is not a component or a library that is released for an existing language platform such as Java or C++. For example, there are many Distributed Virtual Environment implementations which are built directly on top of Java. They took advantages of the java platform, however, they sacrificed the directness in expressing a neural network in a clean NN language. Java and many other popular programming platforms are designed with one dominant top-level paradigm: that is

Object-Oriented. In this paradigm, everything is modeled as data and function pair. To construct a neural network in Java, one must indirectly call functions through an object-oriented software interface. This leads to complex interface design and is difficult to use and upgrade. In NPL, we will have a clean Neural network based syntax which resembles the actual network and hide all communication details from its code. By doing so, there will be no complex object interfaces that programmer needs to learn. NPL principle is that: as software is becoming more and more intelligent, so shall our programming language. In reality, the current NPL implementation adopts a dual-programming model [10]. In this model, there are two distinct language systems: one is host language, the other is extension language. These two language system could communicate at runtime through user-defined Host API. In our game engine, the host language is C++, and the extension language is NPL. C++ is chosen for its high performance and DirectX SDK support as required in our game engine.

3.3 The NPL Methodology

The key idea of NPL is that software system in the future functions more like one giant brain spanning across the entire Internet. It could form new neuron connections, learn from experiences, remember patterns and perform many other functions resembling the human brain. Current object-oriented programming language lacks the directness in composing such kind of software systems, nor is any existing Neural Network Simulation Language eligible for constructing commercial distributed software. NPL tries to solve these problem by means of (1) keeping all communication, network deployment and certifications details out of the program code, (2) presenting programmers a very clean neural network based programming paradigm, (3) preserving all previous familiar paradigms such as object oriented or functional programming. By using NPL, software is constructed like designing a brain network. Section 2.3 shows some demonstrations.

The components of NPL include:

- Neuron file: The source file that programmers used to code the function of an abstract neuron. NPL does not distinguish between a single neuron or a network of neurons. Both can be modeled inside one neuron file. No explicit instantiation is needed in the code, so long as files are deployed into a runtime environment. More details will be given later.
- Runtime environment: It is the runtime environment of neuron files. It is responsible to manage communication of neuron messages both inside the local runtime and between other runtimes on the network. It maps resource names (e.g. of neuron files) to their physical locations on the network, and automatically update any topological changes to this mapping.
- Visual compiler: It compiles neuron files into intermediate code to be executed by the runtime. It has a GUI front end to allow a group of neuron files to be deployed (compiled) on multiple runtimes on the network.
- Visualization and simulation Engine: It provides a complete class of multimedia functions (Host API) which neuron files can be used to read/write to/from a networked virtual environment. This can be regarded as a shared space of virtual reality theater elicited by a neural network. This networked virtual reality will also generate input stimuli back to the neuron files. We have and will continue call this engine a game engine in this paper.

3.4 Computing and visualization in NPL

As a general purpose programming language, NPL must provide patterns for visual presentation. In object oriented programming, we have well-known patterns such as MVC(model, view, controller). In neural-network based programming paradigm, the relationship between computing and visualization resembles our cognition process. Figure 5 shows this analogy.

For years, researchers have suspected that the binding task (mind and brain) is accomplished by nerve cells in distinct areas of the brain communicating between themselves by oscillating in phase (40 hertz) -- like two different chorus lines kicking to the same beat even though they're dancing in different theatres. These oscillations have been detected in everything from the olfactory bulb of rabbits to the visual cortex of cats and even conscious humans. IBM, Birmingham and Saint Mary's researchers believe they have explained not only how the oscillations come about but how the oscillatory rhythm is communicated from one area of the brain to another. These two findings are critical to understanding how the complex electric signals of large numbers of nerve cells generate awareness and perhaps even consciousness.

Indeed, these researches provide us with patterns that are applicable in NPL. Section 2.3 provides a concrete example of such computing and visualization pattern. In the game engine, camera plays the role of attention in the mind. Attention selects only limited mount of imagery at any given time, despite there might be millions of other stories that are being played (simulated) in the mind at the same time. It is the constant selection of our attention that constitute what we perceived as a continuous consciousness. The same thing happens in a game engine, the camera only present a portion of the simulated world to the viewport. Attention replays a previously unseen imagery in the same sequential order as it was generated a short time ago, therefore reinforced it into memory; it signifies the importance of such imagery by bringing it to our internal perceptions, which in turn, makes it easier to affect subsequent imagery generation and selection of attention. So does it in the game engine.

3.5 Message driven model

Many interpreted extension languages are event driven. However, a 100% event-driven language can not simulate parallel behaviors, unless it’s been explicitly programmed as multi-threaded. This is because functions or nested functions must be fully executed before it can release control of its execution thread. Another extreme is that functions can be suspended at any point of execution, at the cost of maintaining mutual exclusive access of any shared data structures. For functional and performance concerns, none of these methods is used by NPL message driven model. Instead, NPL runtime environment adopts a hybrid approach (this is not new). It divides time into small slices. Within each slice, there are two phases (see Figure 10) called (1) **synapse data relay phase**, (2) **neuron response phase**. In phase 1, stimuli from the environment, network and/or local neurons activate the synapses of any connected neuron; and data is passed to the soma, cached, but not executed. In phase 2, NPL examines the list of potential reacting neurons, which is generated in phase 1, and executes it if any of its input field condition tests has passed. Any single execution should be guaranteed to exit within the rest of the time slice by the programmer; otherwise it will be terminated abruptly by the runtime. The executable code may include (a) further complex activation test, (b) generating stimuli to some other neurons wherever they may be, (c) calling

Host API functions provided by the host application(game engine).

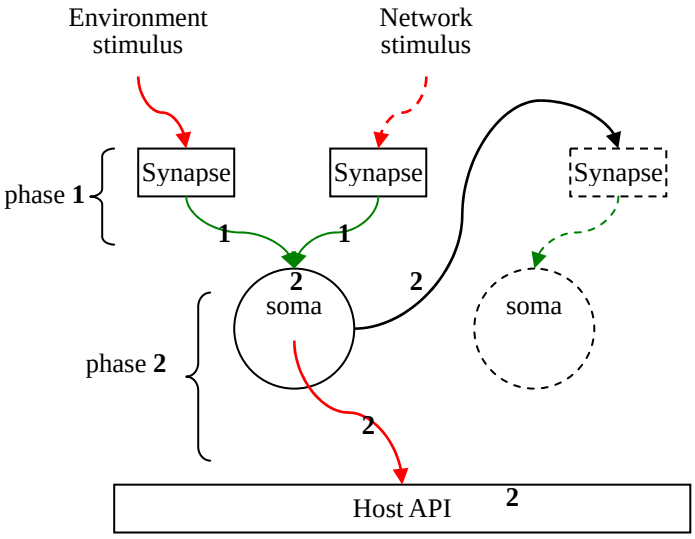


Figure 10. Time slice and phases

With this simple approach, there are three obvious advantages. (1) The time interval during which the neuron state is changed is fully predictable (it is always in phase 1). NPL can efficiently handle mutual exclusive data access to any input field data. (2) External stimuli (from network) are handled transparently as internal stimuli. (3) The execution of NPL never stalls the CPU; even it is running in the same thread as the host application (game engine). This feature also makes it easier for the neural network to communicate with the host application, because by running in the same thread, it is automatically guaranteed mutual exclusive data access to the host application and vice versa. The current implementation of NPL only supports this single threaded mode.

In computer game engine (or other discrete time interactive applications), one annoying problem is that when dealing with some computation extensive tasks, the graphic (or other real-time function) jerks. NPL generally solves this problem. Execution can always be paused at predictable short interval to free CPU for graphic rendering or IO polling.

3.6 Neuron file

In NPL, neuron file plays an important role. It is the building block of neural network. Each neuron file represents an abstract neuron and can have one activation function and many helper functions. Files are referred by other files through namespace shortcut. File is used to represent NPL neuron for the following reasons. *(future version may abandon or have new options for this approach.)*

- File is automatically managed in most operating systems; and people are familiar with it.
- The deployment and configuration of files on the network is easy; and it is supported by operating system.
- File is the most common Internet resources. Ontology can be created for a domain of neuron files on the network.

- By using a single file, NPL gets rid of any artificial tokens and syntax that may bewilder programmers at first. A blank file in NPL is also a valid neuron file and is able to receive and store (overwrite only) any incoming signals, but not producing anything.
- By setting default variable scope to global (of the file), the states of a neuron file can be easily managed.

The content of a neuron file generally contains one *activate* function, some helper functions, many input fields and output channels. Activation function can have none or several input field test conditions. Each condition can be assigned an arbitrary name. See the following pseudo-code.

```
Namespace//Neuron1.npl
function activate()
   Public command input1; -- public input field: a program code that will be executed
   Public string input2;      -- public input field of type string
   Public double input3;     -- public input field of type double
   int a[][], b, c;   -- accessible only in this neuron file, including helper functions.

   [Condition =“Open door of”] if (a && b) then –a test condition: return true if a, b has changed
      … code …
      Output (“#channel1”, “a=1, b=1”);     -- a named output channel
      … code …
      Output (“namespace/neuronX.lua”, “”);   -- an implicitly named routing.
      Output (“namespace/neuronX.lua#input1”, “c=1”); -- an explicitly named routing
      … code …
   end
   [Condition =“shows map”] if (input2 && b!=1) then – return true if input2 has changed and b!=1
      …code… end
   … no condition… -- always executed.
end
function foo()
 -- do something. It has access to all global variables.
end
```

Once a *if clause* has been attributed by a named *condition*, the expression that it evaluates can only be comprised of input fields of this neuron. Test condition is so commonly used that it becomes a language primitive. Names in conditions are for two purposes (1) tracking the status of an active neural network. For instance, we can ask the run time questions like: when does “Open door of” occurs or is it followed by “shows map”. (2) With careful design, we can query the possibility of possession of a certain knowledge from a neural network in a offline mode: such as <“terrain X” “contains” “A”>?, <”A” “is a friend of” “B”>?. Connections between input and output can be specified either in the neuron file, an external file, or dynamically in the NPL runtime. Visual tools can be designed for constructing these connections.

Object oriented programming is allowed in any place of the neuron file. In fact the *string* data type is an internal object that contains both data and functions; and can have many instantiations in different files. However, each neuron file is immediately an instance of itself so long as it has been assigned a path name in a namespace, which is done by CARE. All other neuron files can begin referencing this neuron instance by this unique name. The NPL runtimes maintain a mapping from such names to their physical addresses during execution.

3.7 NPL network ontology

Neuron, neuron input/output field, neural network can all be assets on the Internet. These assets contains network topological information and relationships between them. RDF is an ideal framework of expressing such ontology, and neural network could be made universally available. Discovering and generating ontology might be a joint job of CARE and the human user. With an

ontology framework, (1) Neuron files could be referenced by namespace shortcut (i.e. shorter and invariable names) rather than physical addresses (2) it allows the runtime environment to quickly update network topology changes inside a domain of neural network. (3) it enables two unknown neural networks to connect to each other, provided they have agreed upon some input/output rules defined in ontology description files.

Currently we did not implement the ontology approach; hence NPL runtimes do not have a centralized location for fetching ontology description files (it is an ongoing work). For simplicity, the current NPL runtime searches the file directory for a configuration file of any dummy neuron (a neuron file that exists not on the local computer). It first searches the current directory, then the parent directory, etc. This is a distributed approach, but is not very convenient. For example, files belonging to the same namespace must be deployed to a file directory (with the same short name of the namespace) on all runtimes(computers). One configuration file sample is given here.

Config.pol syntax
locality = {local | network} descriptor = *IPV6 address* connectiontype = {winsock_server| winsock_client| winsock_serversclient } address = *TCP/IP address:port* DestRedirect = {*string** | *string*} SrcRedirect = {*string*}
Sample code
locality = network descriptor = 0.0.0.0 connectiontype = winsock_server address = 10.111.26.123:8051 DestRedirect = script*

3.8 Ongoing Work

This article is a mixed description of the full vision of the proposed framework and the details of our current (reduced) implementations. The evaluation of the Internet RPG game demo has shown good performance of the entire framework (see Figure 6). There are yet many places to be improved. (1) Garbage collection in NPL runtime is one of them. NPL runtime will load a neuron file, once a message has been routed to any of its input field. But how will NPL runtime decide how much space it should reserve for the execution of a certain neuron file or when a neuron file should be unloaded. (2) Visual compiler is being designed to provide a GUI interface for file deployment. (3) We are also designing a detailed ontological framework for discovering, referencing and describing unknown neural networks.

4 Conclusions

Distributed software systems (including the web) need a unified solution to two related problems: computing and visualization; our brain is both a distributed computing environment and a theater of multimedia (internal perceptions). Hence, the analogy of human cognition to programming paradigm might provide some insights into future application development. In this paper, we proposed a unified approach of composing distributed game world based on that analogy. The implementation shows promising results in the game demo. As the web is coming more and more computable (the semantic web) and intelligent (agent technology, MAS), neural network based programming paradigm as described in this article is likely to become the solution to general purpose distributed software applications. NPL is one recent effort in bringing general purpose programming to a level that resembles the human brain.

Reference:

[1] Singhal, S., and Zyda, M. (1999). Networked Virtual Environments: Design and Implementation, ACM Press.

[2] Henry C.Ellis and R.ReedHunt. Fundamentals of Cognitive Psychology. MC Graw Hill. ISBN: 0-697-10543-1

[3] GENESIS : The GEneral NEural Simulation System Version 2.2.1. http://www.genesis-sim.org/GENESIS

[4] Capps, M.; McGregor, D.; Brutzman, D.; Zyda, M. NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments. IEEE Computer Graphics and Applications 20(5): 12-15 (2000).

[5] Jed Hartman and Josie. The VRML 2.0 Handbook: Building Moving Worlds on the Web Wernecke (1996) Addison-Wesley. ISBN 0-201-47944-3.

[6] Web3D Consortium. http://www.web3d.org/

[7] Murray Shanahan. The Imaginative Mind A Precis. Conference on Grand Challenges for Computing Research (gcconf 2004)

[8] Simon J.E. Taylor, Boon Ping Gan, Steffen Straßburger and Alexander Verbraeck. HLA-CSPIF Panel on Commercial Off-the-Shelf Distributed Simulation. Winter Simulation Conference 2003'.

[9] Joseph Manojlovich, Phongsak Prasithsangaree, Stephen Hughes, Jinlin Chen, and Michael Lewis. UTSAF: A Multi-Agent-Based Framework for Supporting Military-Based Distributed Interactive Simulations in 3D Virtual Environments. Winter Simulation Conference 2003'.

[10] R. Ierusalimschy, L. H. de Figueiredo, W. Celes. Lua-an extensible extension language. Software: Practice & Experience 26 #6 (1996) 635-652.

[11] Adrian David Cheok, Siew Wan Fong, Kok Hwee Goh, Xubo Yang, Wei Liu, Farzam Farzbiz, and Yu Li. Human Pacman: A Mobile Entertainment System with Ubiquitous Computing and Tangible Interaction over a Wide Outdoor Area. Mobile HCI 2003, LNCS 2795, pp. 209–224, 2003.

[12] Ruck Thawonmas and Takeshi Yagome. Application of the Artificial Society Approach to Multiplayer Online Games: A Case Study on Effects of a Robot Rental Mechanism. ADCOG 2004'

[13] Rogelio Adobbati, Andrew N. Marshall, Andrew Scholer, Sheila Tejada, Gal Kaminka, Steven Schaffer, and Chris Sollitto. Gamebots: A 3D Virtual World Test-Bed For Multi-Agent Research, Proceedings of the International Conference on Autonomous Agents (Agents-2001) Workshop on Infrastructure for Agents, MAS, and Scalable MAS, Montreal, Canada, 2001.

[14] Tveit, A., Rein, O., Iversen, J.V. and Matskin, M., Scalable Agent-Based Simulation of Players in Massively Multiplayer Online Games, Proc. The 8th Scandinavian Conference on Artificial Intelligence (SCAI2003), Bergen, Norway, Nov., 2003.

[15] Manninen T. Interaction in Networked Virtual Environments as Communicative Action - Social Theory and Multi-player Games. In proceedings of CRIWG2000 Workshop, October 18-20, Madeira, Portugal, IEEE Computer Society Press

[16] Bowman, D. A., and Hodges, L. F. (1999). "Formalizing the Design, Evaluation, and Application of Interaction Techniques for Immersive Virtual Environments." Journal of Visual Languages and Computing, 10, 37-53.

[17] N.Rodriguez.C.Ururahy.R.Ierusalimschy, and R.Cerquera. The use of interpreted languages for implementing parallel algorithms on distributed systems. Euro-Par' 1996 Parallel Processing. Pages 597-600. Vol I.

[18] Cristina Ururahy, Alua: An event driven communication mechanisms for parallel and distributed programming, 2000

[19] Daniel Sánchez-Crespo Dalmau, Core Techniques and Algorithms in Game Programming, New Riders Publishing, ISBN : 0-1310-2009-9, 2003-9