ParaEngine: A Game Engine Framework for Distributed Internet Games

[Name of Author]

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 implementations decides the general type of games that could be composed by it. This article proposes a game engine framework called ParaEngine for developing games based on distributed game world data and logic. Its framework design is based on a modified version of Simulation Theory about the human brain. It has long been observed that our brain is both a distributed computing environment and a theatre of multimedia (internal perceptions). The analogy of human cognition to simulation system has been applied to the proposed game engine to construct distributed Internet games. The implementation is illustrated in an RPG game demo called Parallel World.

Key words: game engine, NPL, simulation theory, Web3D

Notes: For anonymous review as required, several web links and author names have been removed from the body of the text and reference section. The appendix A includes the full text of another article on the same topic by the same author. The submitted article is only the first 11 pages with single column formatting.

1 Introduction

If we compare (1) web pages to 3D game worlds, (2) hyperlinks and services in web pages to active objects in 3D game worlds, and (3) web browsers and client/server side runtime environments to computer game engines, then we obtain the first rough picture of distributed Internet games. It is likely that one day the entire Internet might be inside on huge game world. So far, distributed Internet games can still adopt two very different approaches, which lead to (1) web-alike topology based games and (2) neural network topology based games respectively.

The first approach utilizes the topology of the existing Internet. It aims to build lots of 3D game worlds and connect them by hyperlinks. As the number of 3D game world multiplies, it will finally become a virtual version of the planet Earth. For example, Web3D technology [4] 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. Most of its applications involve a static assembly of dynamic scene data from one or several file servers on the Internet. Most virtual reality (VR) or augmented reality (AR) based computer games [7] have inevitably taken this approach. The advantage of the web-alike topology approach is that it is easy to implement and the underlying metaphors are familiar to both developers and users.

The second approach which is what we are about to address in this article has its metaphor found in the human brain. It has long been noticed that our brain is both a distributed computing environment and a theatre of multimedia (internal perceptions). Hence, the analogy of human cognition to simulation system might provide some insights into future distributed applications. In my former literature [1], I have described a neural network based programming paradigm and shown how it is applied to composing distributed computer games. When we examine a programming paradigm, we need to look at how its computing is related to visualization. Taking the Internet (referring to HTML pages only) for instance; firstly different parts of a page are assembled from the network, secondly a visualization is computed, thirdly user interaction occurs, finally a new page is downloaded. Taking object oriented (OO) programming for another instance; it applies well-known patterns such as event driven and MVC (model, view and controller). But how does our brain generate such rich multimedia imageries from its distributed computing mechanism and manipulate them at its will? The *Simulation theory* [3] has long been proposed in cognitive science to explain our behaviors and internal perceptions. This theory will be further developed in this paper in order to be implemented in a computer game engine. By applying a similar pattern, we would have the entire game world 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 a distributed neural network functioning on the Internet.

To summarize the two approaches: The first one has a fixed topology of virtual game worlds on the Internet. A 3D browser or game engine for such games need to ensure that each game object capable of interacting with anything inside and any visitor (user) capable to be connected from anyplace outside. The second one has only a topology of evolving neural network (NN) on the Internet. A game engine for such games must simulate the NN, present any produced multimedia elicitations to the user, and translate any feedbacks (from the user or game rules) to valid input stimuli of the NN. We believe it is more flexible and natural to let the active neural network elicit any visual scene presentations, rather than vice versa.

In this paper, we will first develop the Simulation theory concerning human cognition and see how it can be applied in a computer game engine; then we will cover implementations of our own game engine called ParaEngine. Mayor differences to traditional game engines are highlighted.

2 The "simulation" theory

A recent description of this theory can be found in papers by Hesslow [3]. I will further develop this theory or hypothesis to make it easy to understand from a computer point of view.

A rough description has been previously given by me in [1]. I will shortly review it here. 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 1.

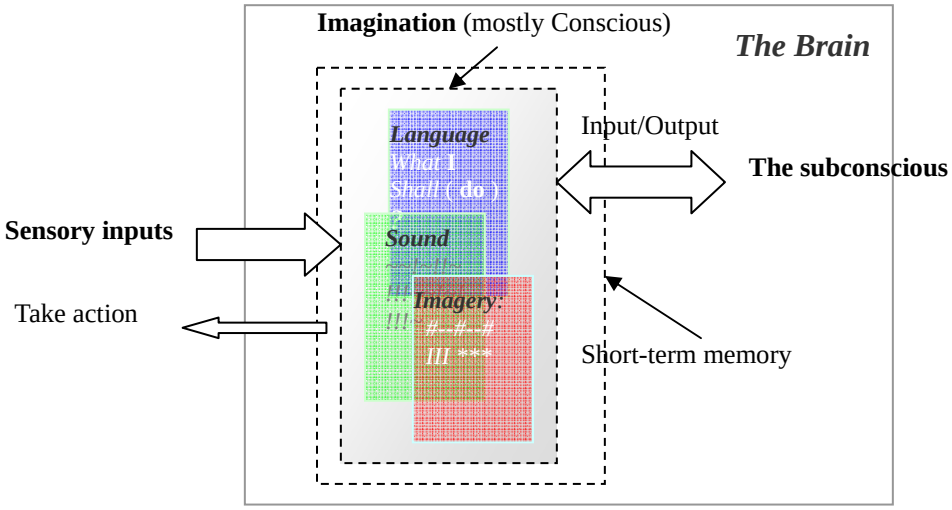

Figure 1. Human Brain: The Imagery-subconscious loop

Metaphorically speaking, the Imagination can be thought of as a multimedia, virtual reality "theatre" [5], 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.

We will now propose a detailed version of Simulation theory (for use in the game engine later). This particular theory has four parts, namely "dimensions of simulation", "concurrent simulation", "imagery-subconscious loop" and "attention and short-term memory". When reading the following parts, readers are encouraged to verify them (i.e. imagine or simulate in the mind) with any computer based simulation software they are familiar with.

(1) **dimensions of simulation.** Any real world situation can be mathematically dissected into infinite number of functions $f_n(t)$ with each evolving over the same time variable *t*. Given N samplings of these functions, we obtain an N-dimensional simulation of a real world situation, i.e. $\{f_n(t) \mid n = 1, 2, ..., N; t \in [T_0, +\infty]\}$. The order of functions in the set is not important, but they must all be synchronized by the same time variable *t*. Generally speaking, the similarity between the simulated situation and the real world situation increases as the number of dimension N increases. A formal description and its proof can be found at [Father, 1980]. With these things in mind, we are now able to define whether a computer program or neural network (the brain) is simulating something or not. This is done by searching a program or NN for any synchronized signals (i.e. time variable *t*) regardless of their physical locations (i.e. *the order of functions in the set*) in that program or NN. So long as the number of dimensions is high enough, the two very different systems (i.e. the simulated

system and real world system) are analogous to each other. We will propose later that the number of concurrent simulations in the human brain is not one, but many; and they are simulated with varying degrees of dimensions, i.e. similarities to the real world. Simulation with the highest dimensions is most analogous to the real world situation and becomes our conscious or internal perceptions. The brain mechanism that controls the rising and dropping of simulation dimensions is called "Attention".
(2) **concurrent simulation.** At any given time, our brain is simulating thousands of visual/auditory/etc. situations concurrently. These simulations are all of high interest or relevancy to the current state of the brain. But only a selected few are enlarged (i.e. their number of dimensions is increased) to be perceptible in our inner world (consciousness). In fact, perception is a natural consequence of increasing simulation dimensions; because the similarity between the internal NN and external real world is also increased.
(3) **imagery-subconscious loop.** This part has been described in many other literatures [3, 5]. It just states that simulation in the human brain can evolve on its own, without interacting with the real world. This is achieved by feeding the result of a simulation to the input of itself, hence forming a loop between imagery and subconscious in the brain.
(4) **attention and short-term memory.** Attention selects only a 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 constitutes what we perceive as a continuous consciousness. 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. I usually compare "attention" to a camera (as in game engine context) or camcorder in our multimedia brain.

3 ParaEngine Framework

In previous sections, we have shown a Simulation theory on how the brain functions. In this section, we will propose software architectures to realize that theory to some extent. Instead of evaluating a theory, our framework is specially designed as a computer game engine for the next generation distributed Internet games. The implementation is illustrated in an RPG game demo called Parallel World (please see Figure 2). 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. In our game demo, players might walk around, talk with NPCs, complete complex tasks all in a continuous infinitely-large distributed game world. On the surface, it looks like browsing 3D web pages [4] on the Internet; however, it is more interactive, purposeful and fun. Beneath the surface, it can be regarded that our own minds are sharing a simulation space with another (artificial) neural network on the Internet. The only difference is that each of us has only one attention, whereas the artificial neural network may have many.

3.1 Overview of ParaEngine

ParaEngine is driven by an active neural network functioning on the Internet. In the engine framework, Neural Parallel Language or NPL [1] is used to construct and simulate 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 [4] which defines the visual elicitations. (We use a dual programming language model [6] in NPL. In this model, there are two distinct language systems: one is host language, the other is extension language. These two language systems could communicate at runtime through user-defined Host API.) The game engine provides a simulation environment. Both the user and the camera system help to select a small portion of all simulated elicitations and present it in cutting-edge multimedia forms to the user. The user might interact with these objects. The game engine then translates such interactions (sensory inputs) to valid neuron stimuli. And the whole system will be functioning as seen in Figure 2 with text, buttons, graphics and sounds.

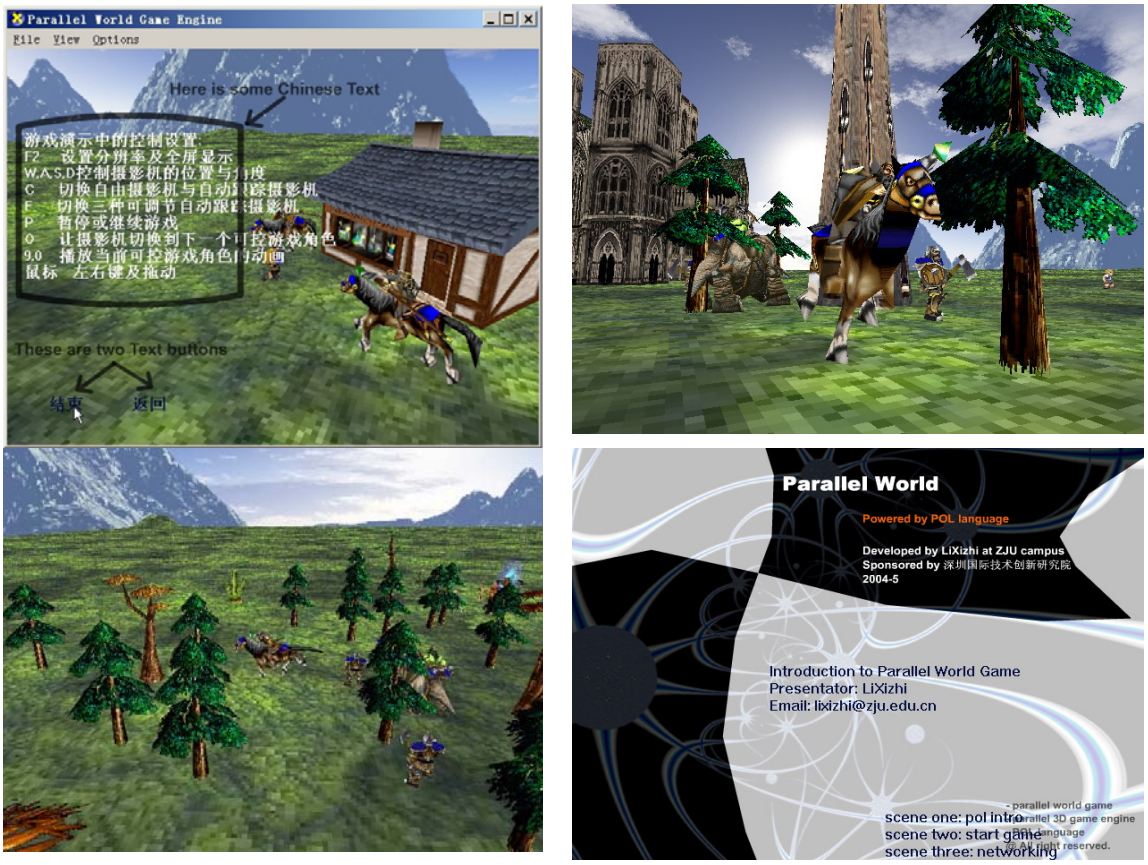

Figure 2. Screen shots from *Parallel World* game

In section 3 of [1], we have proposed Neural Parallel Language (NPL) and its programming paradigm. NPL can be regarded as the script engine in the game engine framework. We will not repeat here how a neural network can be composed and simulated on the Internet. Instead, this paper will focus on all other functionalities and special requirements of the game engine. These requirements as shown below are natural consequences of adopting the framework of the Simulation theory. They have limited the choice of data presentation techniques and common algorithms [8] used by the game engine, such as collision detection and response, path-finding, etc.

List of functionalities and special requirements in the game engine:

- **Neural Parallel Language.** The neuron network and its visual elicitations are all turned into named Internet assets. There will be no explicit network modules in the game engine; instead NPL runtime will handle all distributed computing in a network transparent manner.
- **Scripting.** Everything in the game world should be scriptable, such as resource management, maps, active objects, 2D GUI, camera and object control, etc. This will give NPL full control of the simulated environment through scripting.
- **Real coordinates.** All scene objects are specified in real coordinate system in the game engine. This ensures that there is no boarder or tiles to restrict the positioning of game world objects, although this might cause some troubles in collision detection and path-finding.
- **Absolute positioning.** Although the scene graph is hierarchically structured e.g. in a quad tree, absolute values are used to store the positions of objects in the game engine. This is required by our simulation algorithm. Since we will concurrently simulate a large set of objects which might be geographically far from each other, relative positioning will cause too much computation. More details are given in Section 3.6.
- **Dynamic scene loading.** The 3D world can be dynamically loaded and modified in very fine grains and real-time efficiency should be achieved. This is because the simulation environment will be constantly changing, if attention of NN keeps shifting from one place to another. (e.g. a player

explores through many game worlds on the Internet.) The user, however, must enjoy a continuous visual experience of the changing simulation space, just like in our own imaginative mind.
- **Wide area simulation.** There is no central simulation scheme in the game engine. This is because as the neural network feeds a new elicitation to the game engine it must be able to simulate it together with all other elicitations that are already seen by the game engine. The termination of a simulation depends on time out, memory capacity or explicit commands. (Even the human brain has a maximum capacity in the volume of concurrent simulations.) This also explains the reasons of the above three requirements.
- **Physics and path-finding.** Physics and path-finding are build-in routines of the game engine. For efficiencies, they will not be programmed in NPL. Even our dream worlds are perfectly collision free and follow some basic laws of physics. It is very likely that there is a mechanism in our brain that keeps all of its simulations from making obvious mistakes. And because the mechanism is still unknown and hard to be programmed in NPL, we will realize it in the simulation engine as relatively fixed code.
- **Error tolerance.** The simulation engine should not only tolerate overlapping of rigid objects, but also correct it if one of the objects is movable. In other words, the neuron network run by NPL might make mistakes in positioning objects, yet the game engine should accept such mistakes as valid actions; and automatically and gradually transform group of ill-positioned objects into rational form.
- **Garbage collection.** Because the game engine is viewed as an imagination simulation space, it will not be long before it is filled with too many 2D/3D resources and their instances dumped by the neural network. An automatic garbage collection mechanism must exist so that unused text, textures, sounds, graphics, geometries, animations, etc. can be removed from the memory without human intervention.
- **Efficiency.** The computation for all simulation in the game engine must be roughly proportional to the number of concurrently active objects rather than to the geographical size of the map (game world). Computation means physics, path-finding and rendering. In other words, the selected algorithms must be guaranteed to finish within limited time.

In the following sections, we will cover basic implementations in the ParaEngine for a selection of the above requirements.

3.2 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 3. The color in it denotes the assignment of programming category to computational tasks in composing game world logic.

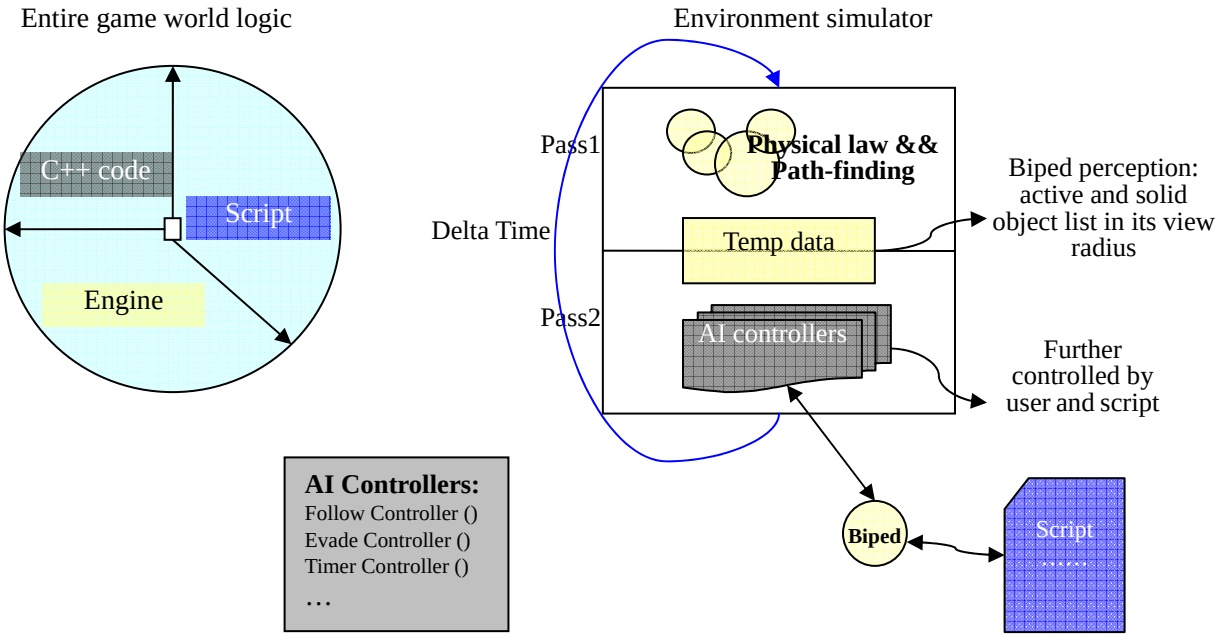

Figure 3. Game world logic in ParaEngine

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 an efficient way. This is one reason why a game engine framework like ParaEngine is still needed to build the aforementioned type of games.

3.3 Timing and engine modules

In ParaEngine, several global timers are used to synchronize engine modules that need to be timed. Figure 4 shows a circuitry of such modules running under normal state. The darker the color of the module is, the higher the frequency of its evaluation.

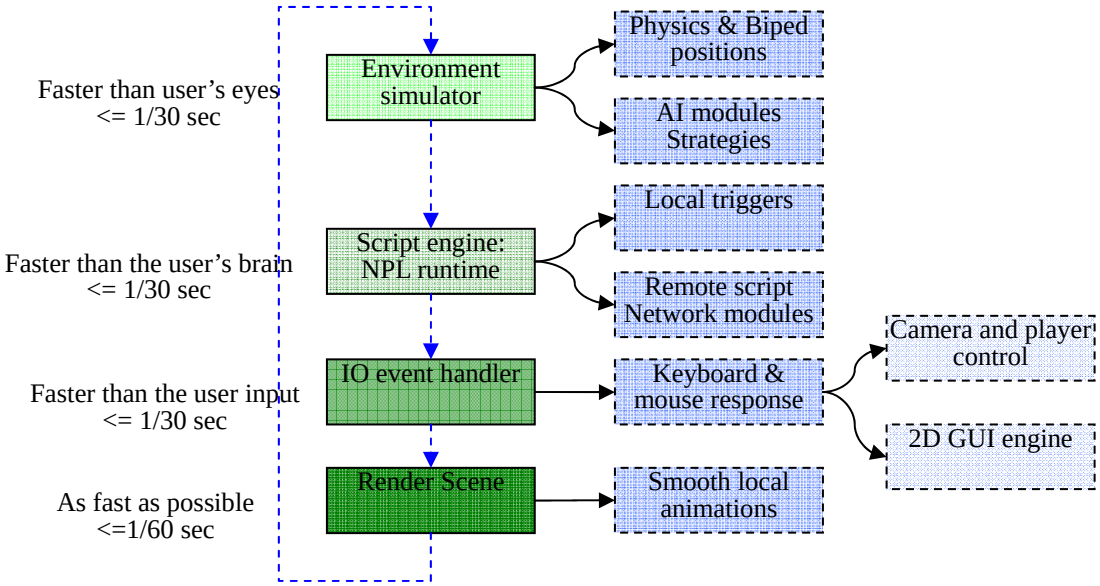

Figure 4. Timing and I/O in ParaEngine

3.4 Scene object and asset entity

The entire 3D game world is composed of 2 sets of objects. The first set is called scene objects and the second set is called asset entities. The scene objects have nothing to do with graphics; it is usually just a mathematical presentation of an object in the game world, such as the size and position of a character. The scene object is NOT concerned with how it is finally rendered to screen by a video card, however, each of them is associated with one or several asset entities. These asset entities are usually textures, mesh and animation data that would tell the rendering pipeline how they will be finally displayed in the screen.

- **Asset entity.** Asset entity can be created at any time during the game. In most cases game assets are batch loaded at the beginning of a new simulation by a script. Asset entity itself must be associated with scene object to take effect. And scene object must be bound to asset entity in order to be rendered. The same asset can be shared among many scene objects. Because asset entity is usually named Internet resource and referenced during a visual elicitation by NPL neuron files, there is an asset manager running in the background, which (1) ensures the same asset can only be loaded once, (2) decides whether to refresh the asset from network or use a local copy, (3) garbage-collects any unused assets in memory.
- **Scene objects.** The scene objects are organized in a tree hierarchy, which is optimized for geographical searching of individual objects. In ParaEngine, all game objects are in 2.5D real coordinate system and roughly organized in a special spatial quad-tree (See Figure 5). This choice might limit the type of game that it could compose, but is suitable for distributed simulation of all mental elicitations by NN and fast collision detection and path-finding in real coordinate system. The scene graph will automatically expand itself as new objects are attached to it. Although most objects are dynamically inserted into the scene graph tree according to their geographical locations, there are some special objects that do not follow this rule. These objects are active objects that need global simulations. For example, some mobile biped object is attached to the Terrain Tile in which it is active, instead of to the tile that best contains it. We know that when the rendering pipeline transverses the scene graph tree, only objects near the camera are visited; however, when the simulation pipeline transverses the tree, all active objects must be covered regardless whether they are near the camera or not. Therefore by moving active objects up near the tree root and using absolute values to store their positions in the scene, it ensures that they can get simulated in simulation pipeline. We will cover the

details of simulation (collision detection and response, path-finding, etc) in later sections.

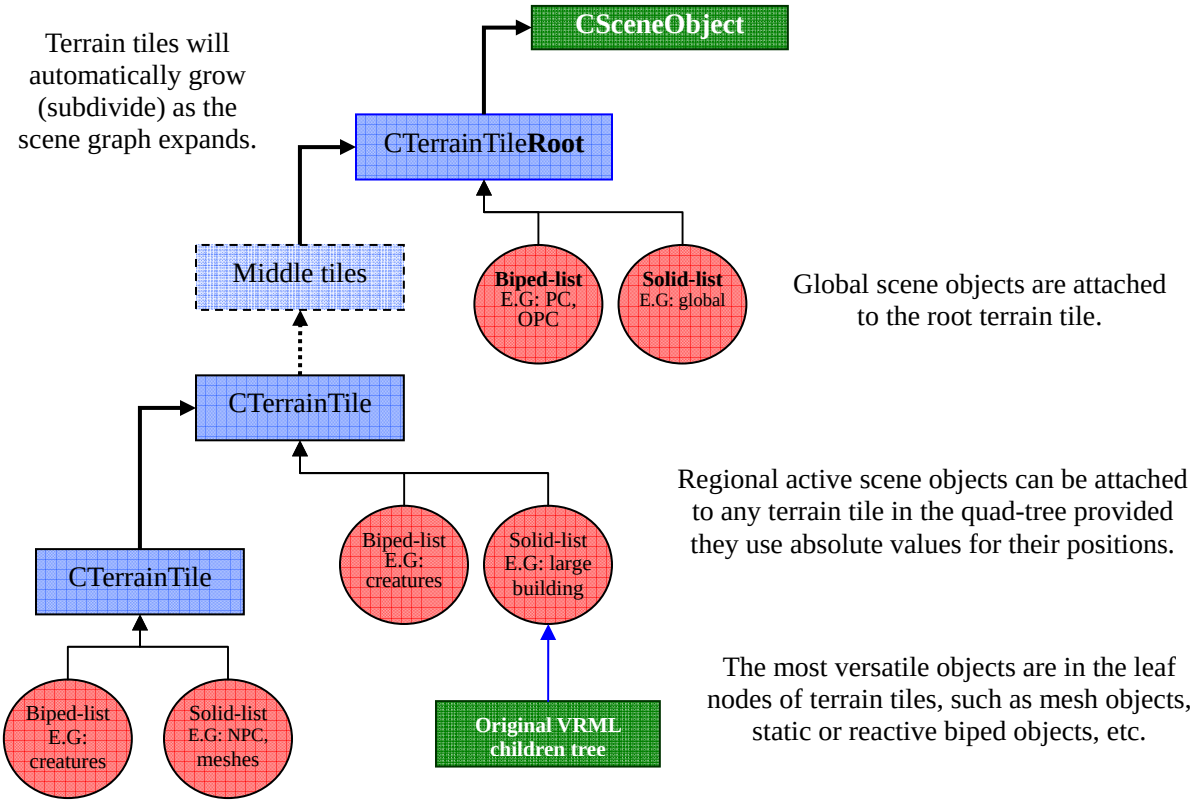

Figure 5. Scene graph tree

3.5 Rendering pipeline

In most cases, there is only one scene graph tree in the engine. Please see Figure 6. To render a new frame, the entire scene is advanced by a time delta. View-culling and sorted post-rendering is also implemented in the rendering pipeline.

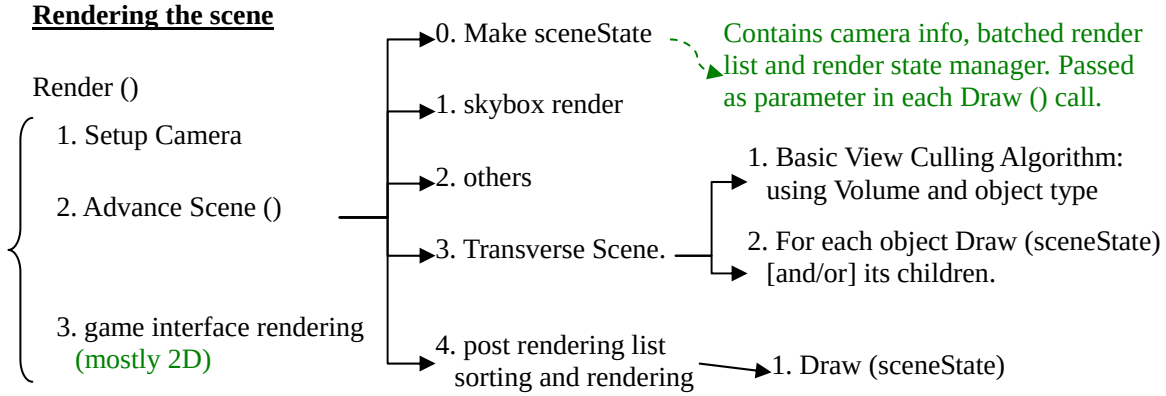

Figure 6. Scene rendering pipeline

3.6 Simulation pipeline

Simulation is the core of ParaEngine. As I have described in Section 2, the Simulation theory has four parts: “dimensions of simulation”, “concurrent simulation”, “imagery-subconscious loop” and “attention and short-term memory”. We will describe their implementations in the game engine one by one. In the sense of “dimensions of simulation”, the rendering pipeline which depends on camera (attention) adds extra “dimensions” to what has already been simulated with but fewer dimensions.

During normal simulation where objects are not seen by a camera, the dimension of simulation is usually limited to object position, size, orientation or perhaps a local frame number; whereas simulation within the camera's view has all the extra dimensions added to be rendered in real life form. "Concurrent simulation" requires the simulation pipeline to cover more about the scene graph than the rendering pipeline during each time slice, so that all active objects will be simulated whether they get the attention (camera focus) or not. As I have described in section 3.2, not all game world logic are written in NPL. Therefore some simulation is done by fixed code in Para Engine, and some is done by distributed neural network in script form. Collision detection and response is one important simulation done by fixed code. In addition, path-finding is also regarded as part of the simulation task performed by the engine code. Hence, in the "imagery-subconscious loop", the "subconscious" is implemented in two programming systems, fixed engine programming and flexible NPL programming. These two systems communicate with each other through predefined triggers in the scene graph. For example, a collision or key stroke upon a scene object might cause a message to be routed to the input field of a neuron script file, and a state change of a neuron script file will cause displacement of objects inside a simulation. Finally "attention" is the camera in the game engine. Objects under "attention" will be simulated with more dimensions than others. The game engine will also keep a short history of camera visited scene nodes. So when the scene graph gets too large, the game engine can selectively release some of them from memory according to this record. This is like in the human brain: imageries that receive more attentions have a better chance to be remembered.

In the following text, I will give some basic ideas on the fixed engine simulation. Please recall that all simulations are completed in a 2.5D unbounded real coordinate system, which means that $\forall p \in \{< x, y, z > | \, x, y, z \in R\}$, p is a valid position in the game world and so it is with orientations. Details of these algorithms can be found in the manual of ParaEngine [Web Link].

3.6.1 Collision detection

First of all, the game engine only generates collision pairs for active objects that are in the simulation region. Second, some active objects usually reside nearer to the scene root than they should be (see Section 3.4), hence we must test against all other active objects in the same tree level as well as the deeper part of that tree node (which might not be transversed in rendering pipeline). An outline of the algorithm is given in Table 1. (Note: Biped means a mobile object in the scene graph. Not everything is explained in it.)

Table 1 Environment Simulator: Collision detection

Clear the global tile list, biped lists, and visiting biped list in each active tile **Pass1:** Select bipeds marked with OBJ_VOLUMN_PERCEPTIVE_RADIUS into the Active Biped List **Pass 2:** generate static collision pairs for each moving active bipeds. "static collision pair" means that one object in the pair is static i.e. not marked with OBJ_VOLUMN_PERCEPTIVE_RADIUS. **Pass 3:** add any active biped that is in the perceptive radius of each active biped into its vicinity biped list. If the biped is moving and is in collision with other active bipeds, add them into its collision list. For each biped in each active terrain tile in the list Test with all other bipeds in the same tile. end **Pass 4:** Animate biped, i.e. move it around the scene, executing High Level Event and/or changing its locations). Solve all collision pairs for each active biped. Path-finding is implemented next; additional way points will be created for mobile objects if necessary. **Pass 5:** Execute AI modules for each active biped.

After implementing the above algorithm, we have a list of bipeds that is active in the current frame, the tile that contains it and any other biped object that is also in the tile. This information is then passed to the AI module of each active biped as its input perceptions. AI module means extensible (high level) simulation tasks which are also available in the fixed engine code.

3.6.2 Path-finding

Path-finding will use the intermediate result generated during collision detection. Path-finding is implemented by each individual biped object. The input information that a biped has is the terrain tile that it belongs to and the distance to all other bipeds and obstacles in its perceptive radius. This information is generated by the environment simulator in an earlier stage. Please see pass 4 of Table 1.The job of path-finding is to generate additional waypoints to the destination. There are several kinds of waypoints that a path-finding biped could generate as a result. See Table 2.

Table 2 Waypoint type

```
enum WayPointType {
        /// the player must arrive at this point, before proceeding to the next waypoint
        COMMAND_POINT=0,
        /// the player is turning to a new facing.
        COMMAND_FACING,
        /// when player is blocked, it may itself generate some path-finding points in
        /// order to reach the next COMMAND_POINT.
        PATHFINDING_POINT,
        /// The player is blocked, and needs to wait fTimeLeft in order to proceed.
        /// it may continue to be blocked, or walk along. Player in blocked state, does not have
        /// a speed, so it can perform other static actions while in blocked mode.
        BLOCKED
};
```

The waypoint generation rule is given in Table 3.

Table 3 Path-finding rule

rule1: we will not prevent any collision; instead, path-finding is used only when there are already collisions between this biped and the environment by using the shortest path.
rule2: if there have been collisions, we will see whether we have already given solutions in previous path-finding processes. If so, we will not generate new ones.
rule 3: we will only generate a solution when the next way point is a command type point. The following steps are used to generate waypoints in path-finding solutions.
Step1: If the biped has already reached a waypoint, then remove it and go on to the next one in the queue. When waypoint itself is in collision with other static objects, the space occupied by these static objects will be used as the destination point; whereas in collision free waypoint, a destination is just a point in real coordinate system.
Step2: Check if there are other moving bipeds in its collision list. If so, block the current biped for some seconds or if the destination point collides with any of them, remove the way point.
Step3: Get the biggest non-mobile object in the collision list. We can give a precise solution. according to its shape, when there is only one object. Any solution should guarantee that the biped is approaching the destination point, so that a group of solutions are guaranteed to reach their goals within limited time.

The core of the algorithm is this: when several objects already collide into each other, only one object is picked to implement path-finding, while the others are put into a blocked state. This moving object will then pick out the biggest object that is currently in collision with it and try to side-step it by generating additional waypoints between its current location and the old destination. This is a fast path-finding algorithm in unbounded real-coordinate system, and is compromised between efficiency and accuracy.

4 Conclusions

In this paper, we introduced the prospect of distributed Internet games, proposed the Simulation theory from a computer point of view, and described a game engine framework to implement it. The idea came from the observation that our brain is both a distributed computing environment and a theatre of multimedia (internal perceptions). The analogy of human cognition to simulation system has been applied to a computer game engine to construct distributed Internet games. Bringing networked virtual game worlds [2] and game world logic to the open Internet will spawn new types of computer games. We hope this article and [1], which complements each other in describing the ParaEngine and Neural Parallel Language, will provide a new prospective to game development as well as general type

distributed applications.

Reference:

[1] [Name of the Author], “Using Neural Parallel Language in Distributed Game World Composing,” in Conf. Proc. IEEE Distributed Framework of Multimedia Applications. 2005 (accepted, to be published).

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

[3] Hesslow, G. (2002) “Conscious thought as simulation of behaviour and perception.” Trends in Cognitive Sciences, 6:242-247

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

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

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

[7] 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.

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

Appendix A: Full Article of Reference [1]

Note: This appendix includes full-text of the literature "Using Neural Parallel Language in Distributed Game World Composing" [1]. It is NOT part of this article, but is appended here for ease of reading and integrity of the proposed framework.

Using Neural Parallel Language in Distributed Game World Composing

[Another article of the author]

Abstract

Bringing networked virtual game worlds and game world logic to the open Internet will spawn new types of computer games. It usually deals with thousands of interactive entities among its web servers. Game engine practitioners have used scripting technology to add soft computing capabilities to a variety of their engine modules. This article proposes a unified approach of using Neural Parallel Language (NPL) in a computer game's scripting engine. We have implemented a reduced version of NPL and our own 3D game engine in a pair. We will show the effectiveness of such a programming language methodology by means of our released game demo. As the web is becoming more and more computable (the semantic web) and intelligent (agent technology), neural network based programming paradigm as described in this article is likely to become the solution to general purpose distributed software applications.

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 [6], 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[5] 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[1][4] and game world logic to the open Internet will spawn new types of computer games[13,14,15,16]. 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. 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 division

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: it is most flexible and can be distributed in many files. Our objective in designing ParaEngine is to let Scripting do as much as possible. (2) C++ programming: it extends basic functionalities already provided in the engine core, such as some computational intensive AI strategies. (3) Engine programming: It is fixed with the release of a game engine. Common functions such as physics and path-finding routines are in this category.

2.2. Timing and networking

In ParaEngine, several global timers are used to synchronize engine modules that need to be timed. Figure 1 shows a circuitry of such modules running under normal state. The darker the color of the module is, the higher the frequency of its evaluation.

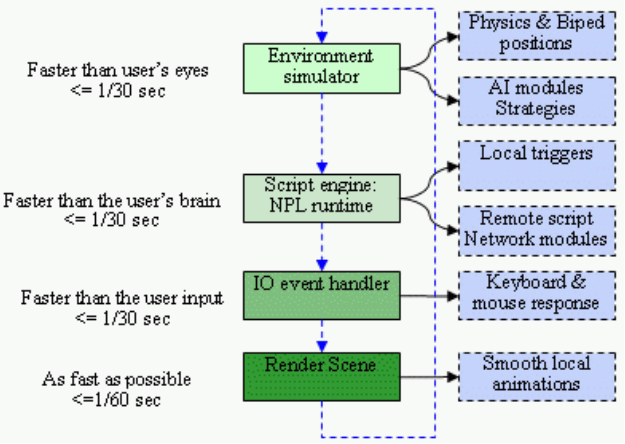

Figure 1. Timing and I/O in ParaEngine

In conventional Internet RPG (Role Playing Game) games, each game-world object like a Non-Player Character (NPC), Player-controlled Character (PC), boxes, doors, weapons, or even terrains is associated with an abstract neuron. The timer (story-line), network command or the human players may stimulate some of its input fields. The stimuli actually come from 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 of doing it. Instead, most game engines explicitly implement a network module [11] which usually relies on Clients/Server (CS) architecture and a single session. A central database is used to hold all the status of its game entities and all events and triggers in the virtual game world reside 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 1), no explicit network modules can be found. Instead, networking is implicitly specified in scripts. As argued previously, scripts are flexible entities that are distributed over the network; and with proper

runtime support at the place where they are situated, complex network logics can be described. Explicitly modeling network logic in distributed script files are not a new idea [8, 9]. 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 [10]. In the ParaEngine framework, we go one step further. Not a single line of networking code needs to appear in any script file, 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 have been deployed in two different runtimes (i.e. two networked computers), the physical location of A and B automatically tells their 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.

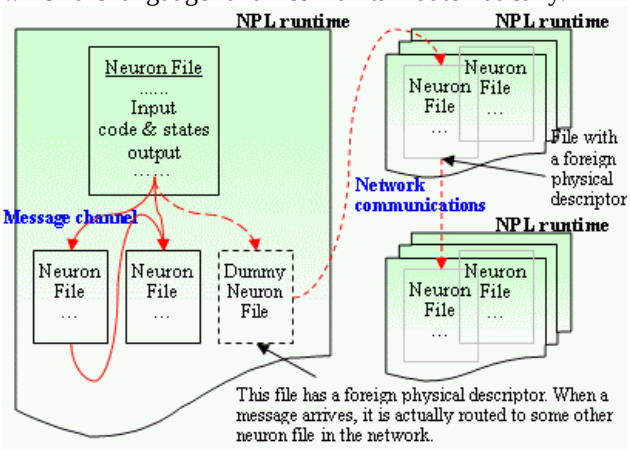

Figure 2. NPL: the big picture

In Figure 2, 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-file 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, terrains, 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

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 | Hand-written script	*.npl (script file)
Plots	Hand-written script | Special visual editor | In-game editing	*.npl (NPL script files to be deployed to the web or just the local runtime)

Let us suppose art elements have been made by the artists, regardless of what ever 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 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 or 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 3, shows the differences. Both models need to map world objects to neuron script files, but the difference is that when in execution which one comes 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 scheme. 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 3).

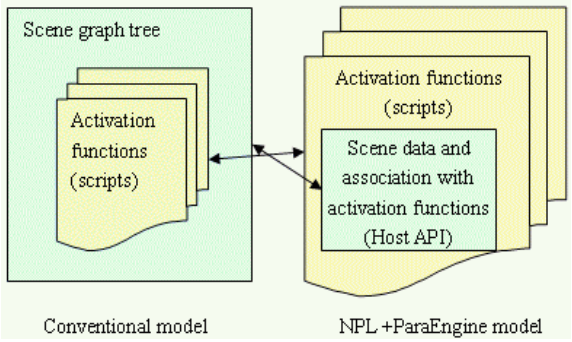

Figure 3. Execution model comparison

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 4.

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.

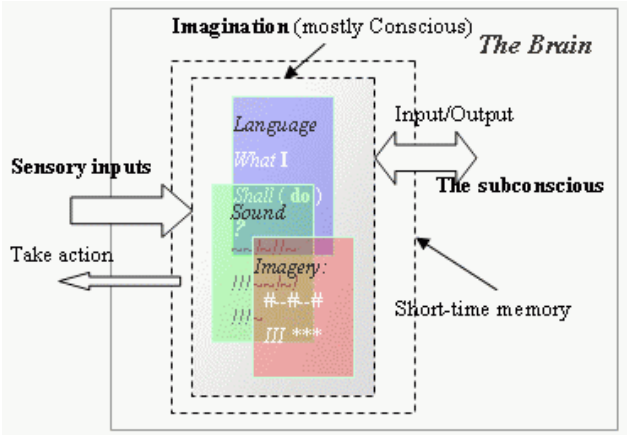

Figure 4. The Imagery-subconscious loop

In the engine framework, NPL is used to simulate 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 elicitations. (We use a dual programming language model [12] in NPL. In this model, there are two distinct language systems: one is host language, the other is extension language. These two language systems could communicate at runtime through user-defined Host API.) 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 translates such interactions (sensory inputs) to valid neuron stimuli. And the whole system will be functioning as seen in Figure 5 with text, buttons, graphics and sounds.

Figure 5. 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. In our game demo, players might walk around, talk with NPCs, 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. We will show the basic steps of composing a game world with our current available tools.

Step1: Preparing 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 are given shorter names. This is done by a short script. A sample code is given below:

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

A GUI tool has 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: Building 3D scenes. 3D Scenes are built using 3dsMax and exported as a VRML file. These scenes (files) are visual entities that might be called by the neural network as visual elicitations. They will cause the game engine to present its imagery to the computer screen; in the meantime, the engine simulates 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 4. In our current implementation, VRML file needs to be further compiled to a more compact script format by a cross-compiler tool.

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. Details of NPL will be presented in section 3. Figure 6 shows a most simple demo: building a movie clip or cinematic with a NPL neural network.

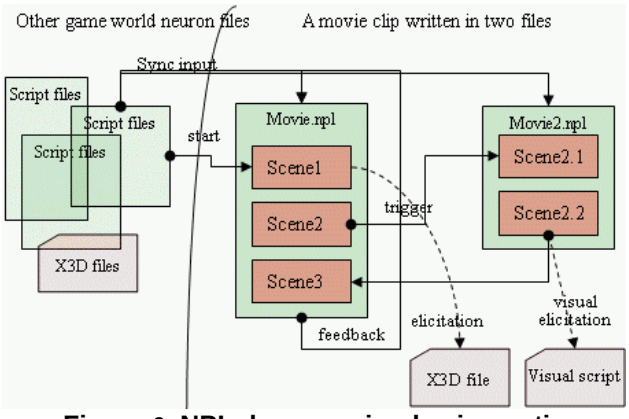

Figure 6. NPL demo: a simple cinematic

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 an X3D file or visual scripts) or call Host API functions (of the game engine) to reset the camera or assign tasks to scene objects. Figure 5 (the second and third picture) shows the execution effect of such a 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 or two computers. When composing a game world, designers can usually divide its 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 servers. An alternative 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) is dealt with by CARE of the NPL language.

3. NPL: Neural Parallel Language

In previous sections, we have shown the role of NPL in ParaEngine. This section will describe NPL as a standalone neural network based programming paradigm.

3.1. Introduction to NPL

In our 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). Distributed software systems generally need to solve two problems: computing and visualization. NPL is associated with a game engine

which provides a visualization platform for the language (See section 3.3).

3.2. 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 [3] eligible for constructing commercial distributed software. NPL tries to solve these problems by means of (1) keeping all communication, network deployment and certification 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 a demonstration.

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 and 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 a management environment where activation and execution of local neuron files occur and messages to external neurons are routed via 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) to 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 simulation environment. This can be regarded as a shared space of virtual reality theatre where stories are evoked or conjured up by a neural network. This virtual reality space may also generate stimuli back to the neuron files. We have and will continue call such a simulation environment a game engine in this paper.

3.3. Computing and visualization in NPL

As a general purpose programming language, NPL must provide patterns for visual presentations. 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 4 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 also 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.

These findings also 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 constitutes what we perceive 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.

3.4. 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 [3]). It divides time into

small slices. Within each slice, there are two phases (see Figure 7) 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. The executable code may include (a) further 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).

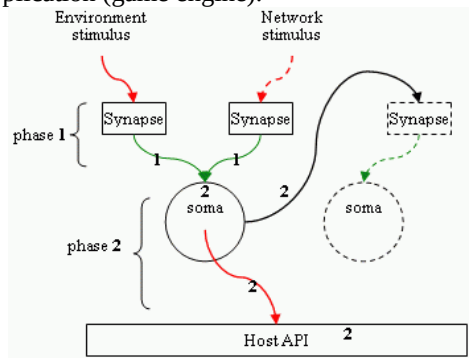

Figure 7. Time slice and phases

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 easy for the neural network to communicate with the game engine, because by running in the same thread, it automatically guarantees mutual exclusive data access to the engine 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 intensive tasks, the graphic (or other real-time function) jerks. NPL generally solves this problem, because execution can always be paused at predictable short interval to free CPU for graphic rendering or IO polling.

3.5. 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 referenced by other files through namespace shortcut. File is used to represent NPL neuron for the following reasons. (1) File is automatically managed in most operating systems; and people are familiar with it. (2) The deployment and configuration of files on the network is easy; and it is supported by operating system. (3) File is the most common Internet resources. Ontology can be created for a domain of neuron files on the network. (4) By using a single file, NPL gets rid of any artificial tokens and syntax that may bewilder programmers at first. E.g. 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. (5) By setting default variable scope to global (of the file), the states of a neuron file can be easily managed.

Object oriented programming is allowed in any place of the neuron file. E.g. the string data type is an internal object that contains both data and functions and can have many instantiations in different places. 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 or its shortcut. The NPL runtimes maintain a mapping from such names to their physical addresses during execution.

3.6. NPL network ontology

Neuron, neuron input/output field, neural network can all be assets on the Internet. 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 topological changes inside a domain of neural network. (3) it enables two unknown neural networks to connect to each other so long as they have agreed upon some IO rules defined in some global files.

Currently we did not implement the ontology approach for locating neuron 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 does not exist on the local computer). It begins from the current directory, then the parent, 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 as the namespace on all runtimes (computers).

3.7. Ongoing work

This article is a mixed description of the full vision of the proposed framework and our current (reduced)

implementations. The evaluation of the Internet RPG game demo has shown good performance of the entire framework (see Figure 5). There are yet many places to be improved. (1) Garbage collection in NPL runtime is one of them. In OO runtime, when the reference count of an object drops to zero, the object can be safely removed; while in NPL runtime, it unloads a neuron file by predicting its probability of being activated in the next simulation cycle. This has something to do with the average hamming distance variance of its input stimuli vector. (2) Visual compiler of NPL is being designed to provide a better 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 becoming more and more computable and intelligent, 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.

References

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

[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.” Conf. on Grand Challenges for Computing Research (2004)

[8] 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.

[9] Cristina Ururahy, Noemi Rodriguez. “Alua: An event driven communication mechanisms for parallel and distributed programming,” PDCS-99, pp.108-113.

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

[11] Joseph Manojlovich, et al. “UTSAF: A Multi-Agent-Based Framework for Supporting Military-Based Distributed Interactive Simulations in 3D Virtual Environments.” Winter Simulation Conference 2003’.

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

[13] Adrian David Cheok, et al. “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.

[14] 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’

[15] Manninen T. “Interaction in Networked Virtual Environments as Communicative Action - Social Theory and Multi-player Games.” IEEE Conf. Proc. CRIWG’2000.

[16] Bowman, D. A., and Hodges, L. F.. “Formalizing the Design, Evaluation, and Application of Interaction Techniques for Immersive Virtual Environments.” Journal of Visual Languages and Computing, 10, p37-53.199