ParaEngine: A Game Engine Framework for Distributed Internet Games

Xizhi Li
The CKC honors School of
Zhejiang University, P.R. China
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 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

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, we will obtain the first rough picture of distributed Internet games. It is likely that one day the entire Internet might be inside one huge game world.

Most distributed Internet games today have a web-alike topology. It aims to build lots of 3D game worlds and connect them by hyperlinks. 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.

In this paper, another approach for constructing distributed Internet games is discussed. It 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 and simulations). Hence, the analogy of human cognition to simulation system might provide some insights into future distributed 3D applications. In [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 (such as for the game engine to construct the game worlds), we need to look at how its computing is related to visualization. Taking the HTML pages for instance, firstly different parts of a web page are assembled from the network, secondly their visualization is computed, thirdly user interaction occurs, and finally new pages are downloaded. Taking object oriented (OO) programming for another instance; it applies well-known patterns such as event driven and MVC (model, view and controller). In our game engine framework, the programming paradigm is motivated by the introspection that human beings are capable of generating first-person and third-person simulations in its brain. This capability leads us to one of the famous hypothesis concerning the human brain, which is called the Simulation-Theory (ST) [3]. The theory will be reviewed in this paper with regards to its selected implementations in the computer game engine. According to the ST theory, by performing simulations in a similar manner, it is possible to synthesize automatic animations which react to external stimuli. Because animations are generated (elicited) from dynamic combinations of simulations (which might physically located at different places on the web), the game world and logic are also distributed.

In this paper, we will first review the Simulation theory concerning human cognition and see how it can be applied in a computer game engine; then we will discuss the special requirement imposed to the game engine; finally we will give more details on some selected implementations.

2 Review of the Simulation-Theory

The Simulation-Theory (ST) is originally a theory about how the human brain generates visual imagery. A recent description of this theory can be found in papers by Hesslow [3]. A more formal mathematical formulation of ST is given by me in [9].

ST 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**.

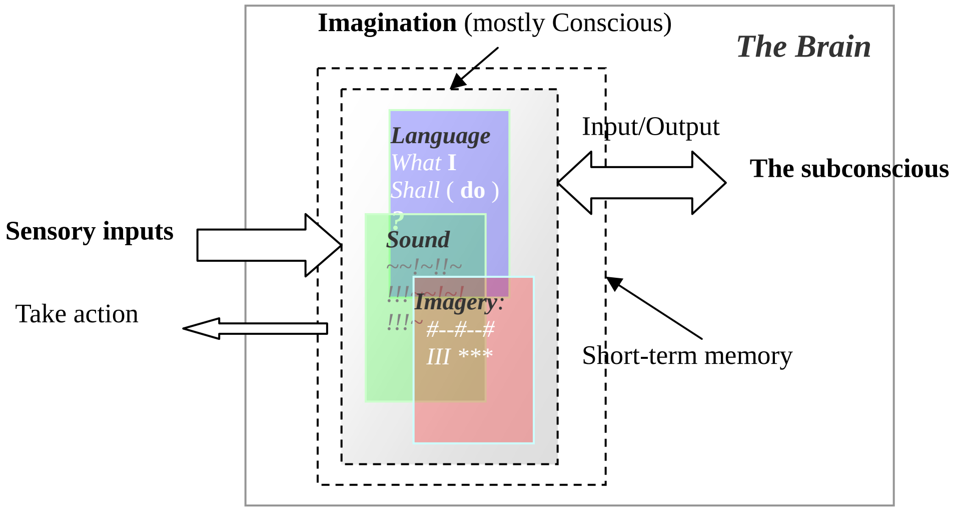

Figure 1. Human Brain: The Imagery-subconscious loop

Metaphorically speaking, the Imagination can be thought of as a multimedia, virtual reality "theatre" [9], 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.

The following parts are the key ingredients in the ST theory. They are "dimensions of simulation", "concurrent simulation", "imagery-subconscious loop" and "attention and memory".

(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. $\{f_n(t) \mid n = 1, 2, ..., N; t \in [T_0, +\infty]\}$. Generally speaking, the similarity between the simulated situation and the real world situation increases as the number of dimension *N* increases. 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 or 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 controlling the selection of simulation dimensions is called "Attention".
(2) **concurrent simulation.** At any given time, our brain is simulating thousands of visual/auditory/etc. events concurrently. A dimension f(t) may be reused for different simulations at different times. Concurrent simulations are 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).
(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 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 unmagnified simulation or memory clip in

the same sequential order as it was generated some time ago, therefore reinforced it in the 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.

3 ParaEngine Framework

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 distributed Internet games. The implementation is illustrated in an RPG game demo called Parallel World (see Figure 2). Everything in the figure is mental elicitation of a neural network constructed by NPL (The scripting language and runtime environment for the game engine). Script files contain the logic for individual neurons and may be deployed on as many computers as the number of files used in composing the game world itself. In the demo, players may complete complex tasks in a very large animated game world. It can be regarded that the players' 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 is used to construct and run the neuron network. The logics defined in the NPL scripts decide how different combinations of neurons are synchronized to perform special simulations and how old simulations are broken and new simulations are formed. The continuous visualization of many concurrent simulations performed by the NN constitutes the animation that the player may see and interact with. However, a realistic visualization needs lots of other information such as mesh and texture data of 3D models. Therefore, the responsible neurons (or script files) 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 corresponding 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 also provides a local simulation environment, which is different from the simulation performed by the NPL. (Local simulation treats everything as an object with certain physical properties.) Both the user and the camera system help to select a small portion of the locally simulated environment and present it in cutting-edge multimedia forms to the user. The user might interact with these objects. The local simulation engine then translates such interactions (sensory inputs) back to valid neuron stimuli (which will affect NPL based simulation). 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 is the script engine in the game engine framework. We will not review here how a neural network can be composed and simulated on the Internet. Instead, the rest of the paper will focus on how other modules of the game engine should be designed to satisfy the NPL scripting paradigm. Some key requirements for the game engine are shown below. They are natural consequences of adopting the framework of the Simulation theory, which has 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 game engine requirements:

- Neural Parallel Language. The neuron network (script files) 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 flies through a long distance in the game world, causing many simulations (on the Internet) to be aroused and decayed.) 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 (a translated simulation) to the game engine it must be able to run 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 this 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.

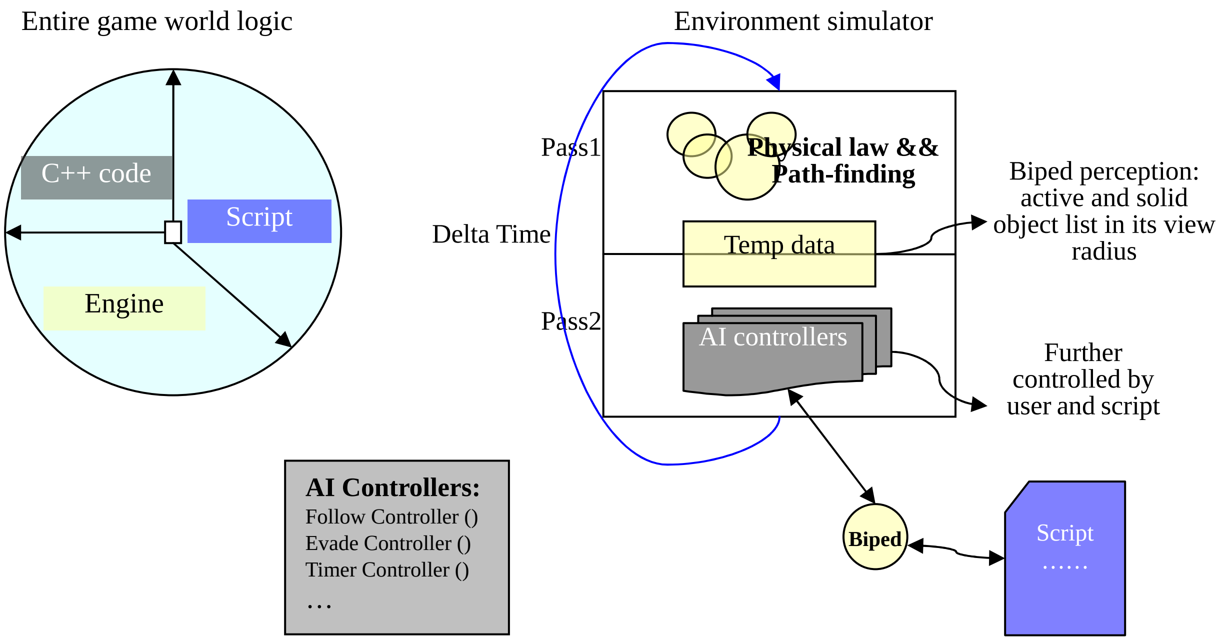

Figure 3. Game world logic in ParaEngine

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.

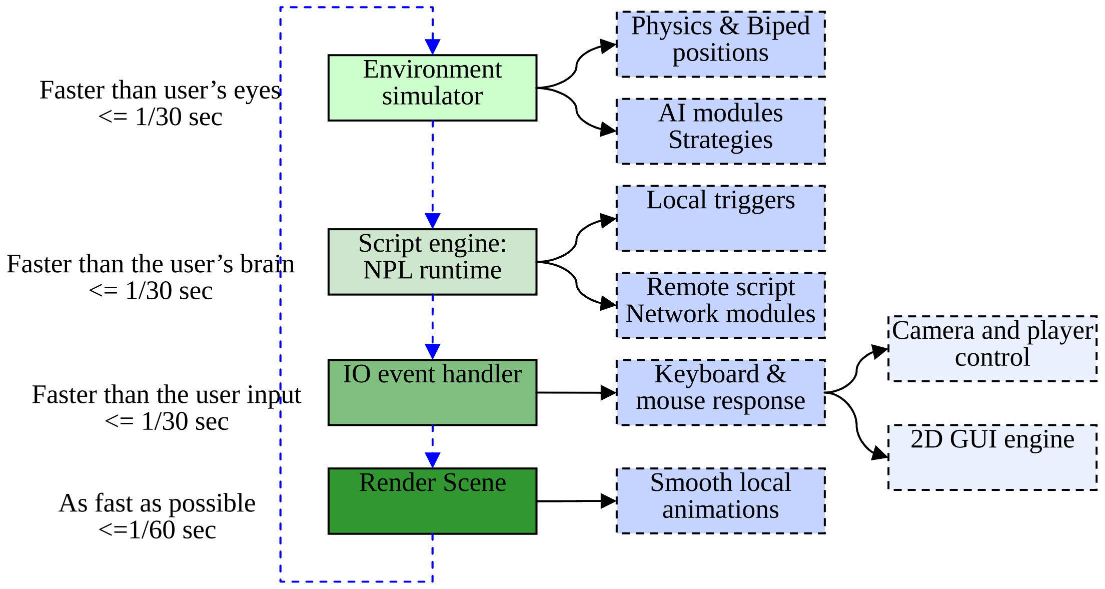

Figure 4. Timing and I/O in ParaEngine

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.

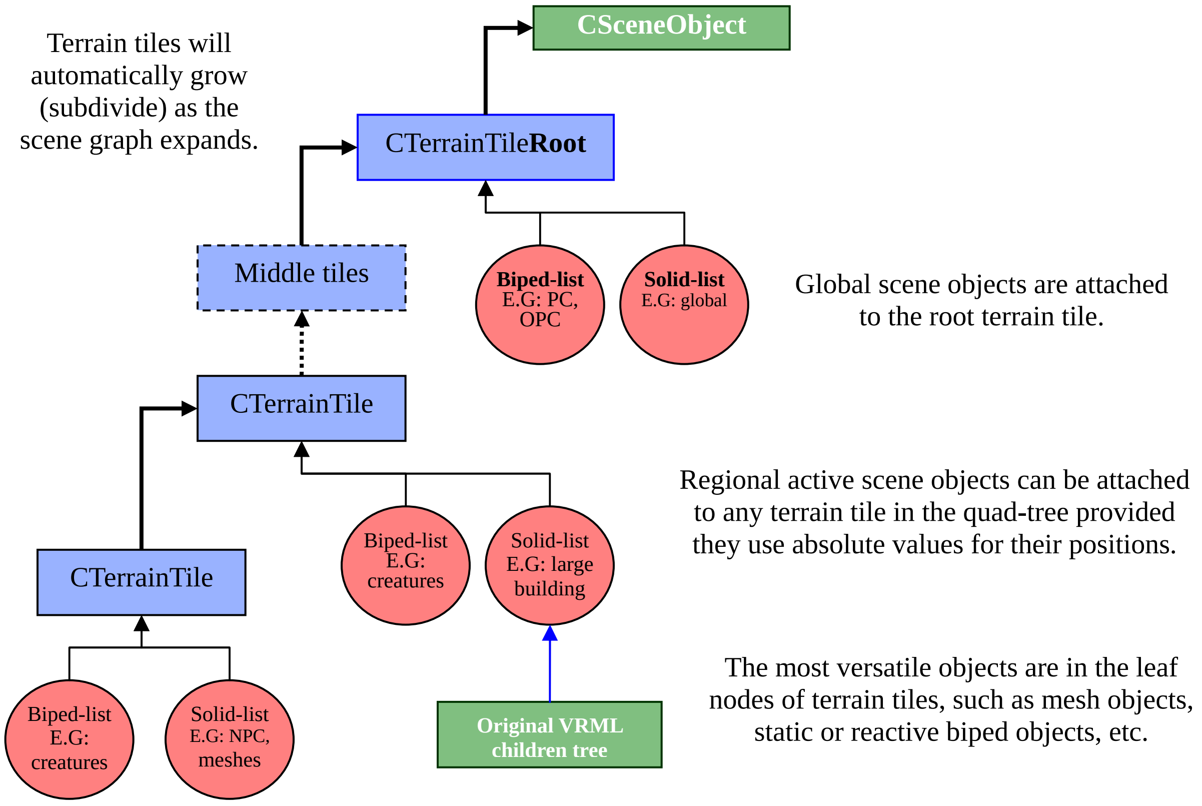
Terrain tiles will automatically grow (subdivide) as the scene graph expands.
CSceneObject
CTerrainTileRoot
Middle tiles
Biped-list E.G: PC, OPC
Solid-list E.G: global
Global scene objects are attached to the root terrain tile.
CTerrainTile
Biped-list E.G: creatures
Solid-list E.G: large building
Regional active scene objects can be attached to any terrain tile in the quad-tree provided they use absolute values for their positions.
CTerrainTile
Biped-list E.G: creatures
Solid-list E.G: NPC, meshes
Original VRML children tree
The most versatile objects are in the leaf nodes of terrain tiles, such as mesh objects, static or reactive biped objects, etc.

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.

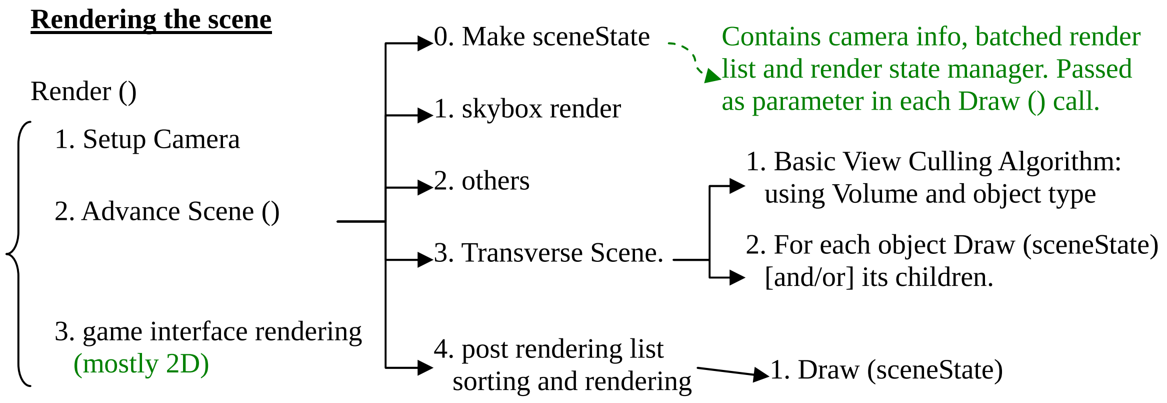
Rendering the scene
Render ()
1. Setup Camera
2. Advance Scene ()
3. game interface rendering (mostly 2D)
0. Make sceneState
1. skybox render
2. others
3. Transverse Scene.
4. post rendering list sorting and rendering
Contains camera info, batched render list and render state manager. Passed as parameter in each Draw () call.
1. Basic View Culling Algorithm: using Volume and object type
2. For each object Draw (sceneState) [and/or] its children.
1. Draw (sceneState)

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. 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. Physically based simulation is done by fixed code in Para Engine, and high-level simulation is done by distributed NN in script forms. 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 fixed engine code. Hence, in the “imagery-subconscious loop”, simulations in the “subconscious” are 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 new configurations to an existing simulation or spawning new ones. 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[1]].

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.
```

[1] http://www.lixizhi.net/projects.htm#paraengine

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, reviewed the simulation-theory from a computer point of view, and discussed a game engine framework based on them. 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] Xizhi Li, “Using Neural Parallel Language in Distributed Game World Composing,” in Conf. Proc. IEEE Distributed Framework of Multimedia Applications. 2005 (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

[9] Xizhi Li. Synthesizing Real-time Human Animation by Learning and Simulation. (unpublished, for review only. Please download full paper from http://www.lixizhi.net/publications.htm)