DHCI: an HCI Framework in Distributed Environment

Xizhi Li

The CKC honors school of Zhejiang University
College of computer science, Hangzhou, China, 310027
Email: lixizhi@zju.edu.cn

Abstract

Due to web technologies, many software systems, from backbones to front ends, are migrating from standalone applications to cooperative ones. This transform impacts not only the way software works, but also how it looks, or its user interface with human operators. This article proposes DHCI, a framework of HCI, which generalizes a collection of programming and user interfaces in distributed environment. It may be adopted as the top-level interface in distributed software applications that call for extensive interactions with its users. The implementation is illustrated in two software systems we have been working at. One is called WAF, which is a multi-agent system (MAS) based application. The other is called NPL, which is the networking and scripting language used in our 3D computer game engine. They represent respectively two application domains of DHCI. The first is interface software for web users. The second is software development environment for programmers.

1 Introduction

Emerging web technologies such as XML, Web services are raising new opportunities in the virtual world. Many software systems, from backbones to front ends, are migrating from standalone applications to cooperative ones. This transform leads to a fully dynamic and distributed environment in the software world. Fixed client/Server architecture might still be a mainstream choice in the next couple of years. However, more cooperative styles of computing will rise and impact not only the way software works, but also how it looks, or its user interface with human operators. Recently with multiple clients and multiple severs architecture, the functionality of end-user applications is significantly extended. However, HCI concepts used in these applications and their development environment remain mostly unchanged. The lack of distributed HCI concept has, to some extent, limited the creativity of software developers and increased the difficulty of distributed application design.

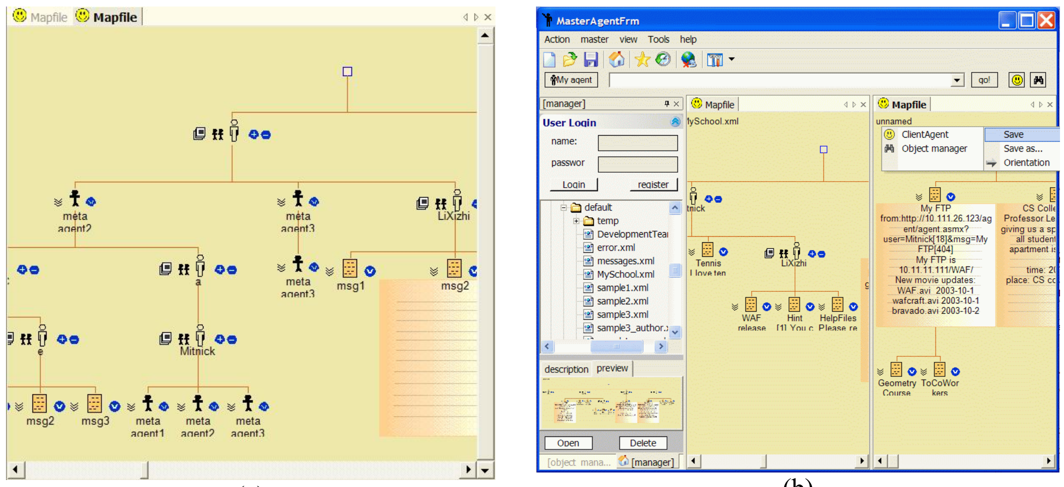
This article proposes DHCI, a framework of HCI, which generalizes a collection of programming and user interfaces in distributed environment. It may be adopted as the top-level interface in distributed software applications that call for extensive interactions with its users. The implementation is illustrated in two software systems we have been working at. One is called WAF, which is a multi-agent system (MAS) based application. The other is called NPL, which is the networking and scripting language used in our 3D computer game engine. They represent respectively two application domains of DHCI. The first is interface software for web users. The second is software development environment for programmers.

Section 1.1 and 1.2 gives the background of the two representative applications, with an emphasis on their interface requirement. Section 2 proposes DHCI in formal terms. Section 3, 4 shows how

DHCI are being applied to the two sample applications. Section 5 concludes the paper. Readers may skip section 2 after finishing reading the introduction part, and start either from section 3 or 4, then going back to section 2 for some references.

1.1 Background of WAF

Our first sample application is called web agent framework (Li & He, 2004) or WAF, which is a web-alike topology (Benyon, 2001) multi-agent system application. It aims to create a visible virtual human relationship network on the Internet. Using agent to represent human beings and provide information to other visitors (including agent) is not a new idea. However, current implementations lack a flexible user interface to convince users that agents are active entities that exist on the network with them. This is due to static user interfaces and conventional object manipulation techniques adopted.



(a) (b)

Figure 1: (a) users leave an off-line tree graph of visited agents and artifacts as they navigate (b) several map files and user's interaction with agents or data in these graph files.

In WAF, user's navigation path can be visualized in an off-line tree graph (See Figure 1.a). The client explorer of WAF will remember each visited agent as well as any downloaded artifacts such as a piece of news or a group of other related agents (e.g. friend agents). It allows reconfiguration of the topology of all these intelligent agents as well as data resources on the client side and save them into local map files. Agents and information in these map files can be updated automatically when they are reactivated or re-opened from the history records kept in the local memory pool(database); they can later be used as the starting point of a new navigation or just provide a group of related web services to its user. See Figure 1.b.

In WAF, although most computing occurs at the place where agents are actually situated, users (including other agents) can customize foreign agent references in different activation maps on their local environment. In our everyday life, we accept the existence of an object only through different perspectives and from many situations in which it used to act. Likewise, in order to let people accept the existence of an agent, the user is allowed to create multiple situations in which the same agent is referenced. In section 3, we will describe it in DHCI terms. One may go to that section directly for further reading.

1.2 Background of NPL

The second application is called NPL, which is the networking and scripting language used in our 3D computer game engine. It is reasonable to believe that the future user interface of the web will go 3D or at least the portion, which might be called networked virtual environment (Singhal & Zyda, 1999), will become a 3D space of interconnected virtual reality worlds on the web. Communities like Web3D is exploiting a new possibility of expressing networked virtual environment that is as distributed as web pages and more interactive than just hyperlinks. One of the biggest parties that are pushing this trend has been in the game industrial.

At Zhejiang University, we are developing a distributed 3D game called Parallel World. We call it distributed game, because each player may have its own game world hosted like a personal website on its PC or other web severs. As a result, a complex game world plot may physically reside on several big servers, whereas a simple game world may be managed by a player on its own PC. Game worlds, big or small, are treated equally in the gaming space. Players might hop from one world to another and complete game tasks spanning several game worlds.

To develop its enabling game engine and using the engine to construct game world logic is a challenging task. Metaphorically speaking, a game engine can be regarded as a local viewport and simulation space for a usually much larger 3D virtual reality space on the web and local disk. Globally referenced art resources and a separate (usually simplified) language system constitute a high-level game programming interface on top of the game engine itself. In some aspects, a game engine can be compared to a web browser, where multimedia files and HTML/XML constitute its high-level programming interface.

An important design issue of the game engine and its high-level programming interface is that, unlike most fixed and pre-compiled distributed applications, distributed game worlds are subject to constant content updates and even deployment site reconfigurations. In our viewpoint, the next generation high-level language may be programmed in a uniform manner independent of its networked runtime environments; the resulting program code will be compiled automatically at their deployment sites; its runtime environments will ensure that the same program will function in the same way under arbitrary deployment scheme; and its development environment may allow visualized design of parallel code and its deployment configurations. In other words, the compiling and deployment process may both be carried out in a distributed manner and environment; whereas the source code is ignorant of this process. This calls for a new language dedicated to this task and a new human-computer interface (HCI) adopted by both its runtimes and the programming interface.

With this vision, we have proposed a neural network based programming paradigm called Neural Parallel Language or NPL to be used in our game engine. It mimics the functioning of neural networks and codes the logics of distributed game world into files that could be deployed to arbitrary runtime locations on the network. Both its programming interface and the game engine platform allows for arbitrary number of visualization and interaction clients running on the huge networked gaming space. The result is that thousands of players simultaneously play in a networked 3D environment which may be hosted on many game servers and even the players' own PCs. In (Li, 2004), I have outlined the basic ideas on its framework design. In section 4, we will describe it in DHCI terms. One may go to that section directly for further reading.

2 DHCI Reference

In previous sections, we have shortly described two application domains of DHCI. Although their HCI differs greatly in both functions and appearance, there is a set of HCI concepts that is common to all of them. These concepts are described in abstract terms in this section. We hope that DHCI could be used as some HCI template, which (1) software designers can use to define similar applications at an early stage and (2) programmers can use to define the high-level programming interface for similar distributed applications.

2.1 Related works

The work contains a major improvement and extension of the short paper (Li & He, 2004). The related works fall into two categories: one at design time, the other at development time.

For the design time HCI, some related works are dedicated to the requirement, analysis and design phase of a distributed application, such as some agent development platform (Bauer, 2001) and UML extensions (like AUML (Odell et al., 2000)). They are focusing on the internal hierarchy of an agent framework. There are also tools and interface models to provide assistance to interface developers (Puerta, 1998). Many methodologies have been developed to support the design phase of Multi-Agent Systems (MAS). Among them are AUML, Gaia and MAS-CommonKADS. These methodologies offer a set of diagrams to help conceptualize and represent the system under development. And there are also tools and network computer games that automatically generate the presence of agent networks over the Internet. Our purposed DHCI is not a concrete implementation to the above helper software, nor is it a substitute for these proposed visual aid tools; instead, we aim to suggest to application developers that much work can be refined and new important functionalities may be needed in future software applications. In the given samples, we demonstrate through concrete applications how these new functionalities are being used and realized.

For the development time HCI issues, DHCI aims to provide a solution to distributed visualization task. We realize that the major impediment to the wide use of distributed systems is the lack of easy-to-use visualization framework. One of its primary reasons is the difficulty of acquiring the information necessary to drive the visual display. A fairly substantial amount of work has been done in the visualization of parallel and distributed programs. However, most of them are in the domain of scientific computation and performance analysis, such as applications built with parallel virtual machine or PVM and the ParaGraph visualization lib (Topol et al., 1995). DHCI promotes the use of distributed computing and visualization for more interactive and widely applicable applications such as web agent framework and distributed 3D computer games. The HCI related programming interface is different from its previous works. Section 2.2.2 gives the details.

2.2 DHCI interface object

The following top-level interface objects are defined in DHCI: DNode, DNodeInstance, DNodeReference, ActivationMap, History, Owner and Runtime. In the following subsections, details of DHCI terms are given with explanations followed. For usage information, please refer to section 3 and 4.

2.2.1 DNode object

DNode := <∑, *L*, UIFunc, In, Out, URI>
∑:= alphabet in the DNode's top-level communication language *L*.
L:= the DNode's internal and communication languages. For formulation simplicity, we assume they are the same language L ($\subset \sum^*$). *L* should contain the function *activate*

> *activate*:=< UIReceivers, DNodeInstance::address, message >, it is an asynchronous function which will send a message to a DNodeInstance object. It returns immediately.

UIFunc:= <UIReceivers, func_name, parameters, …>, the set of definitions to all user interface functions that might appear in *L*. In additional to function name (func_name) and parameters, every UIFunc has an implicit parameter called UIReceivers.

> UIReceivers:={<Runtime::Address, {Owner::key}>}, a list of runtime addresses and keys. It contains information of all runtimes who will receive the UIFunc.

In:= All possible input, which can be further defined as a sub language in *L*
Out := All possible output, which can be further defined as a sub language in *L*
URI := A global resource identifier where the DNode is defined

DNode is an entity prototype defined on the network. When developing distributed applications, it is good practice to define all DNode prototypes beforehand. In most cases, a single DNode prototype is used throughout an application. Only one asynchronous function for sending messages is defined in *L*; synchronous functions are called services in DHCI and are defined in DNodeInstance. The usage of UIFunc is described in details in the DNodeInstance section.

2.2.2 DNodeInstance

DNodeInstance := < DNode, address, program, UIReceivers, lock>
address := <namespace, local path>, a global resource identifier.
program:=<state, code, service>, a program code written in DNode::*L*.

> state := <PublicState, ProtectedState>
>
> > PublicState: = public data or state that is accessible by other DNodeInstance. Public data may be a variable defined in *L*, text and multimedia files in database, etc. They have unique names within a DNodeInstance.
> >
> > ProtectedState:= protected data are accessible within current DNodeInstance. They could be exposed through services (see below) for other DNodeInstance to access.
>
> code:= source code of the program, which may contain asynchronous activate(…) calls to other DNodeInstance.
>
> service:= {<name, IOPairs>}, where IOPairs $\subseteq In \times Out$ AND $\forall (i,o) \in IOPairs\{\exists s1, s2 \in state[\Delta(s1,i) = (s2,o)]\}$, a collection of named synchronous functions written in *L*. The caller will be stalled to wait for the service function to return.
>
> > Δ := A transition function from $(state \times In)$ to $(state \times Out)$

UIReceivers:= DNode::UIReceivers, the current UI receiver list.
lock := {<name, {keys}>} require keys to open them. A program can perform authorizations by accessing the lock content of its associated DNodeInstance.

DNodeInstance is an abstraction for atomic distributed entities on the web. It is instantiated from a DNode prototype. Two major problems confronting most distributed applications are (1) data sharing (2) data visualization. The first problem can be solved by extending the addressing space from local to global and passing any certification information along with data access requests. However, the second problem is not trivial and does not have a universal solution as the first one. One of the design goals in DHCI is centered on the solution to the second problem. See below.

The solution to distributed visualization

One of the big pictures for parallel and distributed computing is to regard all networked computing resources as possessed by one big super computer. However, most applications running on such platforms have only one universal display or one fixed view distribution architecture such as a server providing visualizations for many view clients. It is difficult to schedule a visualization architecture where (1) each distributed atomic program tells how data should be displayed; (2) each client has their own view of the entire distributed computing space (i.e. web) and their own interactions with end users. But such visualization architecture is preferred in many applications and we implemented it in DHCI. For instance, in web agent framework, each agent defines how the result of a request should be displayed on the callers' screen. One request to an agent may activate a chain of other agents, each of which might have something to display for the original caller in additional to data processing request from the last agent in the chain. For another example in the Parallel World game, a player might simultaneously interact with several objects in a game scene. Their events will activate the corresponding scripts on several remote servers, which in turn may further activate other scripts on other severs. All scripts along the activation routes may issue GUI commands to change the 3D view on the original player's computer. A more concrete example is that one player is collided with another player in a game scene. What happens to the first player's runtime environment? One possible computing scenario behind the scene is stated below. The first player's runtime sends a message to the terrain host server, whose scripts will generate GUI commands to update player locations in the caller's view. Meanwhile, another message is sent to the second player's server. This second server might decide to generate a GUI command to display some text on the first player's view. The aggregated view of the first player is that the positions of all visible players are updated and that a piece of text is displayed in a popup dialog box. From the first player's viewpoint, the computing occurs on remote servers; GUI commands are issued in remote scripts, but interpreted and executed in its local view platform.

By contrast, fixed visualization architecture transmits data, instead of visualization commands. Its approach may undermine the independence of distributed atomic entities. E.g. any change made to the entity's deployment scheme will also cause the transmitted data to be altered. However, with visualization commands embedded in each distributed entity, programmers no longer need to worry about to whom visualization data should be sent and a deployment scheme change (e.g. deploying all entities to two web servers, instead of one) will not affect the original source code. Another advantage is that it ensures that the minimum data set that is absolutely required by the data consumer in order to perform the required rendering function is transmitted.

One problem remains that how the same DNodeIntance::program can serve different view clients simultaneously. This is done by the trick of an invisible parameter DNode::UIReceivers, which is passed and modified from one DNodeInstance to another according to the following rules.

- When a DNodeInstance is activated by a Runtime, the network address of this Runtime is inserted in the DNodeInstance's current UIReceivers list.
- When a DNodeInstance is activated by another DNodeInstance, UIReceivers of the caller is merged in to that of the receiver's.
- When a DNodeInstance::program is finished, its UIReceivers list is cleared.
- All UI functions inside DNodeInstance::program shares the same UIReceivers with the DNodeInstance, unless explicitly modified by the DNodeInstance::program from inside.
- A DNodeInstance::program may modify its current UIReceivers list from inside.
- All executed UI functions inside DNodeInstance::program will be batched at the end of the simulation timestamp of the local Runtime and sent to all Runtimes that appeared in the function's UIReceivers parameter.

UIReceivers is attached to every UI functions (DNode::UIFunc) and every DNode::*L*::activate(…) call. For a DNodeIntance::program, its current UIReceivers can be modified from inside the program, such as adding a filter to prohibit UI functions being sent to certain receiver Runtimes or binding one receiver with another. In most cases, though, a DNodeIntance::program does not need to care about UIReceivers.

2.2.3 DNodeReference

```
DNodeReference := <DNodeInstance, visibility>
visibility := A customized appearance of DNodeInstance that is used for display
DNodeReference
```

It is a reference of DNodeInstance, which is used in off-line presentation in an activation map.

2.2.4 ActivationMap

```
ActivationMap := <name, nodes, edges>
nodes := {DNodeReference}
edges := {< in , out>}
in := DNodeReference
out :=  DNodeReference
```

It is a collection of DNodeReferences organized in a directed graph, which represents a user configurable activation topology of DNodeReferences. In DNodeInstance::program, each DNodeInstance may define their own activation topology using the DNode::*L*::activate(…) call. However, this topology is specified by each distributed DNodeInstance and can not be modified by external unauthorized users. To remove this restriction, ActivationMap provides a central place for any user to create additional activation map of DNodeInstances.

2.2.5 History

```
History := {< keywords, object, data>}
keywords := {time | name | address | …}
object := DNodeInstance | ActivationMap
```

It is a historical record of all the above DHCI objects. The history object is meant to provide offline browsing capabilities for the above DHCI objects.

2.2.6 Owner

```
Owner := <UserID, keys, privileges, {DNodeInstance}>
keys := {key}
privileges := {create | delete | modify | …}
```

It is an authorized entity which owns a collection of DNodeInstance.

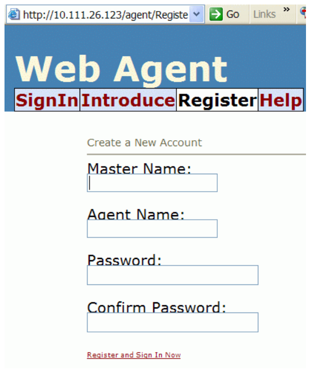
2.2.7 Runtime

```
Runtime := <address, {ActivationMap}, {Owner}>
```

It is the computing, simulation and visualization platform located on a single workstation.

3 WAF illustration

In section 1.1, I have introduced WAF. This section shows how its user and programming interface is mapped to DHCI terms, which are written in parenthesis following their WAF counterparts. In WAF, several agents may be created from the same agent prototype (DNode), which must be made available in the form of a global Internet asset. Because most agent platforms choose to use one of the agent communication languages or ACL and RDF ontology or XML schema in the form of XML encoding, an agent prototype can be defined in XML files on the Internet. An agent instance (DNodeInstance) is usually implemented as web services. A set of high-level UI functions (DNode::UIFunc, Figure 2.c, f) are defined, such as showing message box, expanding current map node, displaying popup menu, etc. The web service code (DNodeInstance::Program) may call other agent instance and send output messages containing UI functions to the receiving client browsers (Runtimes).



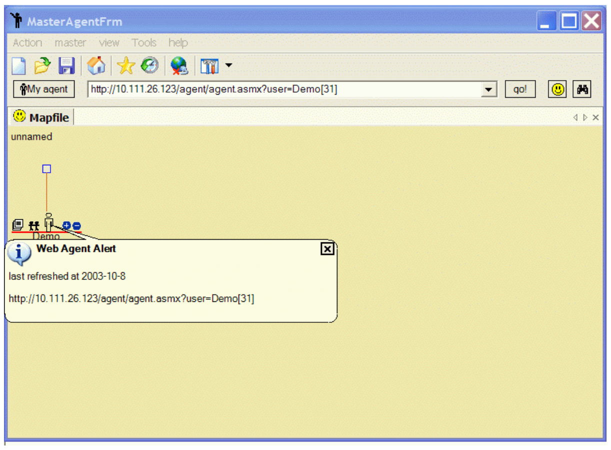
Master Name:

Agent Name:

Password:

Confirm Password:

(a)



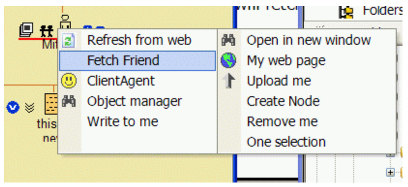
http://10.111.26.123/agent/agent.asmx?user=Demo[31]

Web Agent Alert

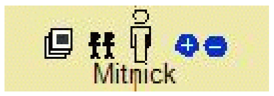
last refreshed at 2003-10-8

http://10.111.26.123/agent/agent.asmx?user=Demo[31]

(b)

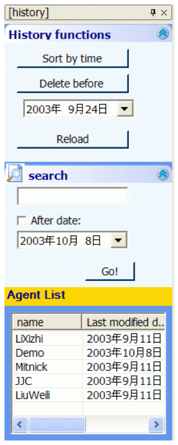


(c)



public message | fetch friend | [Name of the agent] | zoom out | zoom in

(d)

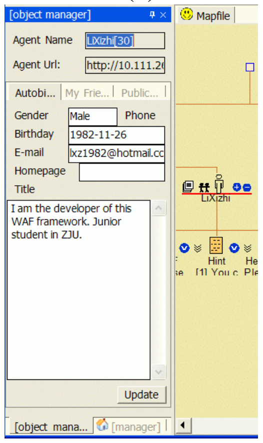


2003年 9月24日

2003年10月 8日

name	Last modified d..
LiXizhi	2003年9月11日
Demo	2003年10月8日
Mitnick	2003年9月11日
JJC	2003年9月11日
LiuWeili	2003年9月11日

(e)



Agent Name: LiXizhi[30]

Agent Url: http://10.111.2

Gender: Male

Phone

Birthday: 1982-11-26

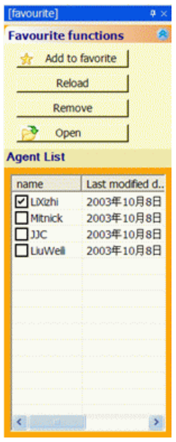
E-mail: lxz1982@hotmail.cc

Homepage

Title

I am the developer of this WAF framework. Junior student in ZJU.

(f)



name	Last modified d..
✓ LiXizhi	2003年10月8日
☐ Mitnick	2003年10月8日
☐ JJC	2003年10月8日
☐ LiuWeili	2003年10月8日

(g)

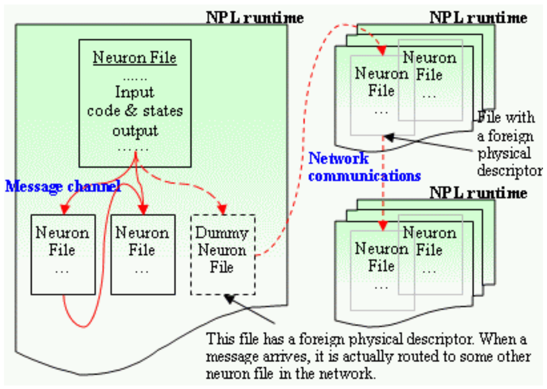
Figure 2: (a) register an owner (b) a downloaded agent reference (c) interact with agent and UI function feedbacks in the form of popup menu (d) a graphical presentation of agent in client browser (e) offline browsing in history (f) UI function feedbacks in the form of pre-defined dialog box (g) other miscellaneous functions provided by the client browsers

In WAF, after registering on an agent provider's website, users are given a master account (Owner, Figure 2.a) and the address of its agent. Once an agent instance has been downloaded to an agent map file (ActivationMap) of a browser, it becomes a reference of this agent on the map (Figure 2.b, d). An agent reference (DNodeReference) is a separate local copy of the web agent. Disk files or a light-weight database system is used as the medium for storing agent map files and agent references. Figure 1.e,g shows some ways in WAF to retrieve off-line objects from the history such as by selecting disk files, entering query strings or date (browsing objects from History). Agent references in a map file can also be edited or annotated (Figure 1) and their topologies in map files can be reconfigured such as by drag and drop operations (reconfiguration in ActivationMap, Figure 1.b).

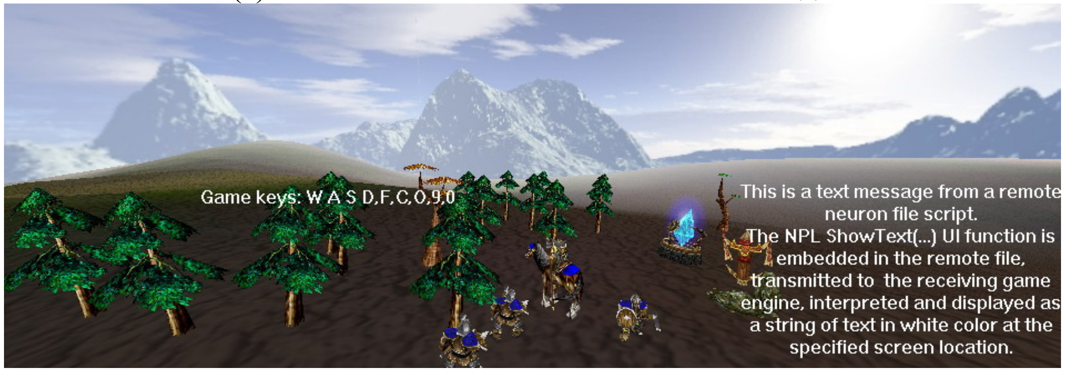
4 NPL illustration

In section 1.2, I have introduced NPL. This section shows how its user and programming interface is mapped to DHCI terms, which are written in parenthesis following their NPL counterparts.

(a)



(b)



This is a text message from a remote neuron file script.
The NPL ShowText(...) UI function is embedded in the remote file, transmitted to the receiving game engine, interpreted and displayed as a string of text in white color at the specified screen location.

(c)

Figure 3: (a) updating players' positions and ordering waving animation from remote scripts (b) NPL runtime and neuron source files (c) a simple remote neuron script demo

Neural parallel language or NPL is the high-level programming language and runtime environment in our distributed game engine. A program written in NPL consists of a collection of independent files called neuron files (DNodeInstance), which can be deployed to arbitrary NPL runtimes (Runtime) on the network. A neuron file may activate other neuron files during its own activation through one-directional asynchronous message passing calls (Figure 3.b). Each NPL runtime may be standalone or be tightly integrated with a game engine (also Runtime). A collection of UI functions (DNode::UIFunc) are defined in NPL. These UI functions cover a complete set of game content controllers, such as loading game scene objects, displaying dialog boxes, playing sounds, updating player positions, animating characters, changing camera mode, etc. Some of these scene objects are associated with specific neuron-file address (these special scene objects are DNodeReference) at their creation time, so that events in the locally simulated game world will activate corresponding neuron files. Human player can built its own game world (reconfigure the ActivationMap) through the game console, such as building houses, planting trees, adding a door portal to someone else's game world, etc. Any visited game worlds can be preserved on the human player's local game server for offline browsing (history). An activation chain of neuron files usually originates from an in-game event, such as an internal timer or a 3D/2D collision event in the local game world (Runtime). The network address (UIReceivers) of this triggering runtime is passed through the activation chain as described in section 2.2.2. Hence, all UI function calls in the chain of files can know their receiving runtimes. This logic will allow the same network of NPL source files to serve any number of game clients (Figure 3.c).

5 Conclusion

We proposed a DHCI framework for economic use of distributed resources. New computing frameworks such as multi-agent system, distributed neural network based computing require new HCI framework designed for their efficient manipulation and visualization. The proposed framework is summarized from two representative applications we have previously implemented. We hope it could provide some useful insights and HCI patterns in the design of similar distributed applications.

References

Bauer., B. "UML Class Diagrams Revisited in the Context of Agent-Based Systems." In the econd International Workshop on Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada, May 28- June 01. 2001. pp 1-8.

David Benyon, "The new HCI? Navigation of information space," Elsevier. Knowledge-Based System (2001).

Li, Xizhi, "Using Neural Parallel Language in Distributed Game World Composing," in Conf. Proc. IEEE Distributed Framework of Multimedia Applications. 2005.

Li, Xizhi and He, Qinming. "WAF: an Interface Web Agent Framework." IJIT. International Conference on Information Technology 2004.

Odell., J., Van Dyke Parunak., H., and Bauer., Bernhard. "Extending UML for Agents." Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, Gerd Wagner, Yves Lesperance and Eric Yu eds., Austin, Tx, pp 3-17, AOIS Workshop at AAAI 2000.

Puerta, A.R. "State-of-the-Art in Intelligent User Interfaces" Knowledge-Based Systems, 10(5), 1998, pp. 263-264.

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

Topol, B. et al., "Integrating visualization support into distributed computing systems." Distributed Computing Systems, 1995., Proceedings of the 15th International Conference on , 30 May-2 June 1995 Pages:19 – 26