Programming Language Support For

Virtual Environments

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

By

Jafar M. Al-Gharaibeh

August 2012

Major Professor: Clinton Jeffery, Ph.D.

Authorization to Submit Dissertation

This dissertation of **Jafar Mohammad Al-Gharaibeh**, submitted for the degree of Doctor of Philosophy with a major in Computer Science and titled "**Programming Language Support for Virtual Environments**" has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: Date:

Dr. Clinton Jeffery

Committee member: Date:

Dr. Robert Heckendorn

Committee member: Date:

Dr. Terence Soule

Committee member: Date:

Dr. Fred Barlow

Department Administrator: Date:

Dr. Gregory W. Donohoe

Discipline's College Dean: Date

Dr. Larry A. Stauffer

Final Approval and
Acceptance by the College of
Graduate Studies: Date:

Dr. Jie Chen

Abstract

Developing 3D virtual environments requires an advanced level of programming expertise in a wide range of programming domains including 3D graphics, networking, user interfaces and audio programming. To compound the problem, virtual environments have strong real time performance requirements. The complexity of developing these kinds of applications comes from two sources: first, the requirements of the virtual environment itself, with its dynamics and size. The second is the programming language used in development, with its strengths and also the limitations it imposes. Unfortunately, most of the tools and libraries necessary for developing virtual worlds are available mainly with low level system programming languages such as C and C++. The complexity and the amount of code required in this family of languages contribute to the overall complexity of virtual world applications making the process of building such applications a challenging process. Because the gap between language and application domain is high, a language for writing virtual environments is needed. Very high level languages such as Python and Unicon, compared to languages such as C, offer very high level programming semantics, syntax, data structures, and rich APIs with built-in support covering a wide range of programming activities such as I/O. With these language characteristics, programs can be made significantly more compact and therefore less complex. However, these very high level languages lack features essential to developing virtual worlds, and more importantly, they fall short of high performance and scalability requirements of virtual environments.

This dissertation presents a language/application co-design approach for software development of virtual world. Both the application and the programming language itself evolve over time to meet new requirements. This approach is used in the development of a collaborative virtual environment called CVE, and its implementation language, Unicon. The focus of the co-design is on language extensions for virtual environment development. These extensions include 3D graphics, concurrent programming and 3D interaction. The language is improved to address the complexities and requirements that arose at the application level.

This dissertation answers two main questions: 1) Is it possible to utilize a very high level language with a legacy virtual machine in virtual worlds development where performance is critical? 2) How and where does such a language need to be extended and modified both at the language level and in its implementation to meet the requirements of a multi-user virtual environment? The goal is to reduce the complexity and cost of developing virtual environments, enabling less experienced programmers to participate in developing such applications. The dissertation does not create a virtual world or a new programming language; instead it uses an existing language and complements it with very high level features. The main contribution of this dissertation is the novel design and integration of new features into a very high level goal-directed language. These features not only provide very high level support for virtual environments, but also meet the language design guidelines, maintain backward compatibility, and have very little impact on the syntax. The benefits of the new features are not specific to virtual worlds.

Curriculum Vita

Jafar Al-Gharaibeh
Department of Computer Science
University of Idaho
Moscow, ID 83844
jafara@vandals.uidaho.edu

Education

University of Idaho, Moscow, ID
Pursuing Ph.D. Degree in Computer Science

Yarmouk University, Irbid, Jordan
Master in Computer Science - Feb 2004

Yarmouk University, Irbid, Jordan
B.Sc. in Computer Science - June 2001

Acknowledgments

This dissertation is coming to a conclusion, and the work could not have been done without the assistance and support of many people. I would like to thank my advisor Dr. Clinton Jeffery for his dedication and encouragement throughout the years of this dissertation work. He set a great example for me in his level of involvement and willingness to provide help and guidance even outside work hours. Also I would like to thank the dear respected members of the committee, Dr. Robert Heckendorn, Dr. Terence Soule, and Dr. Fred Barlow, for taking the time to review this work and provide much appreciated feedback.

I would like to thank the Unicon Language community, especially Kostas Oikonomou, Phillip Thomas and Steve Wampler, for providing feedback and test analysis on various language extensions added to the Unicon programming language as part of this dissertation work.

Finally, I would like to thank the corporate and governmental sponsors of this work. This research was supported in part by a grant from the National Science Foundation under agreement number DUE-0402572. This work was also supported in part by AT&T Labs Research and the Specialized Information Services Division of the U.S. National Library of Medicine.

Dedication

I dedicate this work to my parents who supported my pursuit of education throughout the years. To my wife Hiba who provided comfort and support and daughter Rayanne who brought much joy and happiness.

Table of Contents

List of Figures

List of Tables

PART I Overview

1 Introduction

Virtual worlds have a tremendous impact on the world we live in today. People are spending more and more time in virtual worlds and social media spaces. Moreover, recent years have witnessed the emergence of huge online communities in games and virtual worlds. Millions of people populate these virtual worlds to play, engage in social activities, or do business. Many organizations and institutions use virtual worlds including games for a variety of purposes, such as recruiting and training soldiers in the army [1], or promoting interest in science and engineering [2]. Furthermore, the spectacular success of massively multi-player online role-playing games (MMORPGs or MMOs for short), both in term of number of players and revenue [3], has led to a large amount of interest in educational multi-user virtual environments [4]. World of Warcraft (WoW) [5], and similar games have demonstrated both the mass appeal and the potential of this genre.

One way to build a new virtual world application is to extend and build on top of an existing one. Many virtual environments that are available online for users, such as SecondLife [6], ActiveWorlds [7], or Open Cobalt [8], give users the ability to customize the virtual space and add to it. This approach minimizes the software development cost, but still incurs other costs. In addition to the potential financial cost, users are constrained to the rules, activities, and limitations imposed by these virtual worlds, and content creators are limited to the tools provided by such worlds. For some applications, where domain requirements are substantially different from what is provided by such existing virtual worlds, using an existing virtual environment would require extensive modification that is not feasible for small teams or small projects. Adapting an existing gigantic codebase or using a game engine, in most cases, exceeds the technical skill of inexperienced programmers such as students. That was the case for the Collaborative Virtual Environment project (CVE) covered later in this dissertation.

Another obvious way of building a virtual world is to pick a suitable programming language and start developing from scratch. Usually this is a long and expensive process, defined by the requirements and the goals of the project, and greatly influenced by the programming language used.

In this dissertation, a co-design approach is undertaken to accomplish the task of developing a virtual world. The co-design principle is utilized in many fields such as hardware/software co-design [9]. In hardware/software co-design, the hardware and software are concurrently designed and developed to meet the system objectives [10]. It is popular in domains such as mobile and embedded systems [11] [12], compilers and high performance computing [13]. The popularity of hardware/software co-design suggests an under-utilized analog: language/application co-design. Programming languages are in fact much easier than hardware to co-design alongside the applications that run on them. In many cases this co-design occurs during the creation of a domain specific language, and there are conspicuous examples, such as operating

system/language co-design [14]. The objective of this dissertation is not to develop a language/application co-design methodology, but apply the co-design approach on the domain of multi-user virtual environments.

The Unicon programming language and the CVE virtual environment framework are on-going research projects at the University of Idaho. The author of this dissertation is involved in both projects. The dissertation facilitates virtual world development through language design, and also improves the Unicon language by means of language/application co-design. Any new feature or extension added to Unicon must conform to the language's spirit, and also should be made as general as possible so that it can be used in other domains benefiting the entire Unicon language community.

Using this approach of language/application co-design, CVE and its implementation language Unicon, evolved together over the time of the project development. This approach extends the language virtual machine and the runtime system to serve as a game engine, instead of the more typical approach of embedding a scripting language on top of a more conventional game engine. The application makes use of the language's advanced features, such as powerful data structures, very high level APIs, and concise code. When some new language feature is required by the CVE, or the performance of an existing function in the language does not meet the application requirements, new features are added and/or existing features are enhanced to meet the new requirements.

Extending a language with new features can be done through library bindings or a more integrated built-in approach. Binding provides a direct and usually one to one function mapping to an underlying library. Many languages for example provide bindings for OpenGL. Since OpenGL is implemented in C, most languages cannot call OpenGL functions directly; instead they go through a set of wrapper functions which is referred to as a binding. Built-in support on the other hand, may provide a different vocabulary, constructs, and features that are better integrated with the language and conform to its designs and features. This higher level API does not map directly to the underlying library. One function/feature typically maps to several underlying functions, while other features might not exist in the underlying library. Binding usually provides a more complete access to the underlying library, a more uniform coding style across languages, and better performance than built-in support. But built-in support provides a higher level API which usually translates to shorter and easier to write and maintain source code. In some cases it might also provide extensions that the original library lacks

The Unicon programming language used in this project is an object-oriented descendant of the Icon programming language [15] [16]. Icon integrates Prolog-like goal-direction and implicit backtracking within a conventional syntax and an imperative semantic core. Icon's traditional domain is string and file processing, and the rapid development of experimental algorithms and data structures. Unicon is a superset that extends Icon along two dimensions: features such as classes and packages for larger-scale projects, and

extensive access to modern I/O capabilities such as graphics, networking, and databases. Unicon is an open source project available at unicon.org.

The CVE project discussed in this dissertation is hosted on Source Forge at cve.sourceforge.net. More than 15 students participated in its development. CVE is the driving force behind many new Unicon features and performance improvements. In many cases, iterative revision and development on both the language and the application was done to achieve the required functionality and performance sufficient for smooth animation in CVE. CVE was also extended as part of this dissertation to include computer-controlled non-player characters (NPCs), who serve as tutors and record keepers for users' accomplishments. The design for the NPCs and their quests presented later in this dissertation is called: Portable Extensible Non-player character tutors and Quests (PENQ, pronounced "pink") [17]. Both CVE and its NPCs are used to demonstrate and evaluate the use of the new language features and improvements.

1.1 Motivation and Goals

Building a collaborative virtual world is a huge task involving all kind of activities such as 3D graphics, user interactions, networking, concurrent programming, modeling and art. This makes developing virtual worlds a challenging process, especially because they tend to be large and complex systems requiring large teams. Virtual worlds are also performance critical which adds to the challenges facing developing these kinds of applications. A large part of the complexity of these applications comes from the programming languages such as C and C++ used to develop these virtual worlds.

Languages such as Unicon employ very high level semantics and features hiding a lot of implementation details and allowing developers to write much more compact code compared to languages such as C. The use of a very high level language to develop a virtual world would greatly reduce the amount of code needed, but very high level languages lack many features required by such application. This dissertation tackles the complexity of virtual world by using the very high level language Unicon, and faces the problem of most missing features by building them into the language. The challenge is to meet all of the virtual world's (CVE) requirements using an interpreted language with a legacy virtual machine, and add new features with high level semantics to meet the language standard and make these features general language constructs that can be used by and benefit all kind of applications and not necessarily virtual worlds.

The CVE project described in this dissertation was motivated initially by a goal to support distance education in a virtual world. Enabling distant students to attend lectures and office hours, and do homework assignments and labs within the virtual environment, initial efforts focused on reproducing a local CS education environment, including 3D representations of two physical CS departments, avatars, and chat. The project expanded over time to include domain-specific behaviors and tools such as interactive

collaboration on common CS tasks of editing, compilation, execution and debugging, and later on, game like features.

Within the CVE project are many dissertation-worthy problems, but this dissertation focuses on aspects of the underlying language co-design. The main goal is to build an enabling technology and infrastructures that simplify the process of building a virtual world. This can be achieved by:

- First, using a very high level application programming language that takes a lot less time and code to develop applications than a mainstream systems programming language.
- Second, integrating new features essential to developing virtual worlds into the language.
- Third, addressing any application performance issues mainly in the implementation of the language itself with the least intervention from programmers, or provide the programmers with explicit mechanisms and features that allow them to handle such issues.

Building virtual worlds using an open source very high level programming language opens the door for new capabilities and features that can be built into the language. An open source language is easy to experiment with, and in a very high level language, new features can employ expressive semantics and powerful data structures. The programmer does not have to learn enormous libraries to do tasks such as 3D graphics or networking. This approach hides many of the implementation details in many situations, removing the burden of such low-level details from the programmer and allowing the language to take care of them. Such low level details are usually one of the biggest factors hindering inexperienced programmers from undertaking virtual worlds implementations. With this in mind a new question arises: what features should a programming language have in order to make the task of building a virtual world something feasible even with a limited budget?

CVE is used as a testing platform to validate new language features and also to identify missing crucial features. To further help evaluate new features, a framework was developed that allows easy construction of computer controlled non-player characters (NPCs). NPCs in this framework can be run as part of another process, integrated as threads in the virtual world server for example, or can be run as separate clients giving them a great amount of freedom and independence from the host virtual world, allowing an NPC to be created outside the virtual world and deployed with minimum support from the virtual world itself. These NPCs are tested in an educational setting (the CVE's main purpose) as an experimental domain to create NPC tutors that make the learning process as fun as playing a game.

1.2 Research Hypothesis

The research presented in this dissertation is being undertaken to test two hypotheses. **The first hypothesis is: a very high level interpreted language can be used to build prototype virtual worlds, which are performance critical**. This hypothesis aims to answer the first question of the dissertation. Several standalone tests are used to measure the amount of code needed and also the performance of certain language features essentials for virtual worlds development. In addition to that, experiments are used in CVE to do performance analysis to test if the language meets CVE's requirement on both client and server side.

The second hypothesis is that features required by games and virtual worlds development can be built into a legacy virtual machine, hiding the implementation details and providing a very high level API. This hypothesis aims to answer the second question in the dissertation. There are three points to cover regarding this hypothesis:

- What are the required features?
    - These new features are identified by the requirements of new functionalities needed in CVE
- Can these be built into the language?
    - This will be demonstrated by successful integration of the new features into the Unicon language while meeting the language standard of high level; semantics and maintain backward compatibility
- How to measure these new features?
    - By presenting several example and CVE experiments demonstrating qualitative and quantitative analysis of the new features. Qualitative analysis includes presenting code examples using these features, demonstrating the ease of use of these new features and their integration with the overall language semantics. Quantitative analysis includes conducting experiments to compare the performance of Unicon with and without the new features, presenting how this affects CVE on both server and client sides.

Enabling the very high level language (Unicon) to deliver the requirements of the virtual world (CVE) answers the first question in the dissertation. The need for new features is determined by studying what functionalities virtual worlds provide, what performance level is required, and what features are missing from Unicon. CVE is a test case for this study. The extent to which requirements are met is evaluated in Part III.

Most of the new features are built into the Unicon's language's virtual machine and its runtime system instead of a class library because:

- These features require access to lower level libraries and data that are not available at the language level, such as 3D object selection.
- Some features are performance critical and have a great potential speed up when built into the language, such as array support.
- The language did not have the infrastructure necessary to do them. For example, Unicon language original designed assumed sequential execution in the virtual machine and the runtime system. These had to be extensively changed to support concurrent threads eliminating race conditions and introducing synchronization and communication mechanism. Changing the language was unavoidable in such case.

The second question in the dissertation is answered through identifying problem areas when new functionalities in CVE cannot be added or do not perform well because: 1) the language lacked certain features, such as 3D object selection, necessary to add these new functionalities 2) the language did not deliver enough performance for very demanding functionalities such as 3D models rendering and animation which triggers the addition of arrays and threads.

1.3 Contributions

The research in this dissertation results in several contributions in programming language design with regard to collaborative virtual world development. These contributions can be summarized as follows:

- Demonstrating the use of a very high level interpreted language to develop a real time virtual environment, including advanced features such as 3D models, animation and dynamic texturing.
- Building new language features with a novel design, integrating them into a very high level goal-directed language with a legacy virtual machine. These features not only provide very high level support for virtual environments, but are also consistent with the language design and spirit, and their benefits are not specific to virtual worlds.
- Blending the semantics and syntax of new features with existing language features in most cases. This has the following benefits:
  - Reduce or eliminate the time it takes to learn the new features because they are used in the same way existing features are used. This makes an easy transition for language users to start using the new features.
  - Keeping the language syntax and semantics changes minimal which also contributes to the ease of use of the language.

  This feature “blending” can be witnessed in several features introduced to the language that are discussed throughout this dissertation. A summary where this technique is used can be seen in:
  - Arrays: look and behave like lists

  - Dynamic textures: are drawables like windows
  - 3D selection: operate like a GUI interface
  - Threads: look and feel like co-expressions
  - Thread communication: integrated with network communication.
- Flexible design of new features that balances between simplicity and implicit use versus control and powerful use. Examples include:
  - 3D selection:
    - Direct use of 3D selection API relying on only one function and one keyword. This requires minimal setup, and makes it straightforward for simple scenes with few selectable objects.
    - GUI-like interface for 3D selection, requires slightly more setup and writing event handlers, but more suitable for large projects with hundreds of selectable objects.
  - Data Protection:
    - Thread safe data structures provide a convenient and implicit locking mechanism for fine grained access to data structures.
    - Critical regions and mutexes provide an explicit locking mechanism to protect large regions of code and support atomic behavior to several data accesses.
  - Thread synchronization:
    - Implicit synchronization via use of communication operation which provide blocking and non-blocking semantics controlling when to stop and when to resume a thread without the programmer intervention.
    - Explicit synchronization using condition variables which give programmers full control over when to block threads and when to move them to work.
  - Thread Communication
    - Fully shared memory model by passing references to data structures.
    - Pure message passing by using only communication operators and channels to share data without passing references to data structures.
    - A mixture of shared memory and message passing depending on the situation. For example, threads that work on a huge amount of data can share references to thread safe data structures instead of wasting a significant percentage of time transmitting messages, but when sharing little data with other threads, message passing would suffice.

1.4 Organization of the Dissertation

This dissertation is structured into four parts. The first part (Chapters 1-3) provides introductory background. The second part (Chapters 4-6) covers all of the new language extensions and features introduced to make Unicon suitable for games and virtual environments development. The third part (Chapters 7-9) presents the evaluation of the work that has been done in part II. The last part (Chapters 10-11) concludes the dissertation and proposes future directions and improvement opportunities.

In part II which is dedicated to the programming language support for virtual environments, Chapter 4 presents the 3D graphics features and enhancements incorporated into the language. Chapter 5 covers 3D selection while Chapter 6 discusses the introduction of concurrent threads to the language. Figure 1.1 demonstrates the many features in a programming language that are heavily involved in the development of a virtual environment. It also shows the focus of this research in case of the Unicon programming language. The size of each slice in the figure gives a crude approximation on the amount of work that goes into each part relative to others but it is not meant to be exact.

Part III, dedicated for evaluation, includes two case studies, first, portable extensible non-player characters architecture, and second, CVE as an example collaborative virtual environment development built using the Unicon language. Figure 1.2 presents the aspects of this research study, and demonstrate the fact that changes on any part might affect other parts.

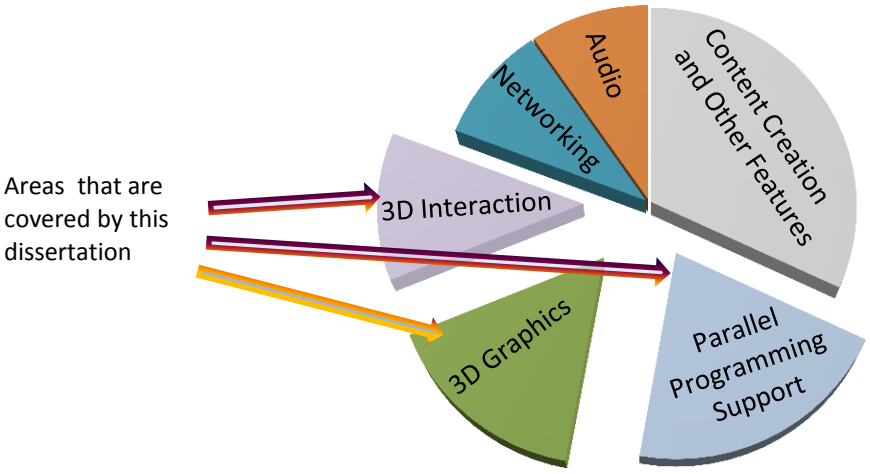

Figure 1.1 The focus of this research on three features set in the Unicon language

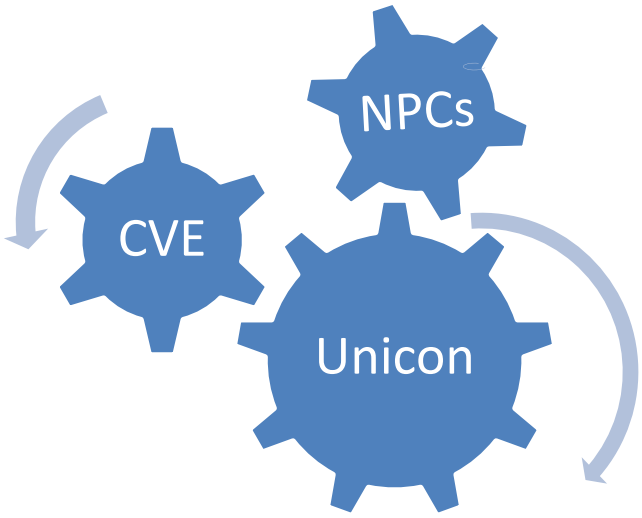

Figure 1.2 The three major aspects of this research

Language features are evaluated by using them to implement substantial components of CVE. The evaluation part (Part III) starts with programming examples and benchmarks in Chapter 7, followed by Chapter 8; NPCs and their role in virtual worlds, including a discussion about quest activities, the design and implementation of the proposed NPC architecture. Chapter 9 introduces CVE, presenting its design and implementation evaluating it and its features as a virtual environment case study for this research.

2 Background and Related Work

This dissertation addresses a broad subject relating to many areas. The work presented in this chapter is carefully selected not only to be related to the work in this dissertation, but also to give context for the key innovations and design decisions of the dissertation as well as to give an idea of the breadth of this research.

The first technical subject area necessary to support virtual environments is 3D graphics. Technologies, techniques and hardware for developing 3D graphics applications have been changing rapidly since the 1990s. Doom [18], the popular PC game released in the early 90s, was able to achieve a level of realism that no other PC game was able to match at the time. In addition, Doom featured the ability to play over the network allowing several players to play in the same virtual world. Since then, the expectations of what a game can provide and how it should look have risen, driving great advances in 3D graphics libraries, game design, and giving birth to game engines [19]. Video games that followed Doom drove advances in graphics cards, bringing them to the masses at increasingly lower prices [19]. Despite these advances, developing these kinds of applications remains a challenge, requiring advanced level of programming expertise, big teams and huge budgets.

Many tools, libraries and languages have been developed or tuned to suit these kinds of applications. A survey of collaborative virtual environment technologies by Wright and Madey [20] covers various tools, techniques and frameworks used to develop virtual worlds. This chapter sheds light on some of the previous work relevant to this dissertation, and gives a background on several subjects that constitute the core interest of this dissertation. Section 2.1 presents the definition of virtual environments and provides example virtual worlds and some research in the subject. Section 2.2 discusses game engines, the building blocks of many virtual worlds and popular games. Section 2.3 covers some of the literature related to language support for virtual environments: mainly graphics, user interaction with the virtual world (3D object selection) and concurrency. The last section in this chapter, 2.4 gives a brief overview of the research in NPCs which are an essential part of virtual worlds, and hence, are covered in this dissertation.

2.1 Adaptable Collaborative Virtual Environments

Virtual environments can be defined as computer-generated, three-dimensional settings in which the users of the technology perceive themselves to be located, and within which interaction takes place. As the technological barriers to creating virtual worlds have decreased, researchers have created many collaborative virtual environments to serve various domains. The popularity of virtual worlds has increased, as well as their numbers and users, with hundreds of virtual worlds available online populated with tens of millions of players.

The term virtual environment is used to refer to a wide range of applications. These applications can be categorized using different criteria as explained in the following paragraphs.

- Based on the number of users:
  - **Single user**: The popular game Super Mario Bros is an example [21].
  - **Multi-user**: a limited number of users (in most cases less than 20) can share the virtual world over a local network. An example from games includes Doom [18].
  - **Massive multi-user**: a large number of users (thousands or more) share a huge virtual world connecting over the internet, such as Second Life [6] and World of Warcraft [5]. This is distinguished from Multi-user above by the large number of users, and also the persistent world which continues to exist and change while the players are not online. The world itself continues to be online.
- Based on appearance:
  - **Textual**: the world is described by words leaving the interpretation to the user's imagination. MUDs (Muti-user Dungeons) are examples of such worlds. AberMUD is an example MUD game [22].
  - **2D**: the world is represented by 2D graphics. Examples include the old arcade games and also many Adobe Flash games [21].
  - **3D**: the world is constructed via 3D graphics. Many popular virtual worlds fall in this category, such as Second Life [6], ActiveWorlds [7] and many other games.
- Based on technology:
  - **Desktop virtual environment**: the virtual world is a conventional application, no special hardware is used. Most computer games belong to this category.
  - **Virtual reality**: includes enhanced elements of immersion, assisted by hardware and devices, such as gloves and head mounted displays. Usually this is used for specific domains like manufacturing and training such as flight simulators (Figure 2.1).

Figure 2.1 Virtual reality: flight simulator [source: www.freeway.org]

- **Augmented reality**: Augmented Virtuality and Mixed Reality are combinations between real and virtual environments. Mixed Reality refers to any environment that has both real and virtual aspects. If this environment is real with added virtual aspects of information it is called augmented reality. If it is virtual with added real aspects it is called augmented virtuality (see Figure 2.2). For example, augmented reality starts with a real physical environment and the computers alter the view of this environment by adding or removing information (visual, sound, etc...) on top of this view. A hypothetical example would be the ability to walk downtown wearing glasses that block all ads replacing them with a photo of a tree. Augmented reality is widely used in military, industry and scientific research. It is also used in sports, while watching a soccer match for example; TV viewers can see lines, numbers and so on projected onto the soccer field explaining an "offside" situation (Figure 2.3) or how far a free kick is from the target .

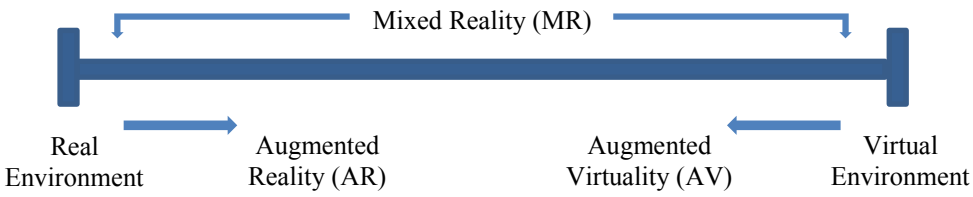

Figure 2.2 Simplified representation of an Reality Virtuality (RV) Continuum [23]

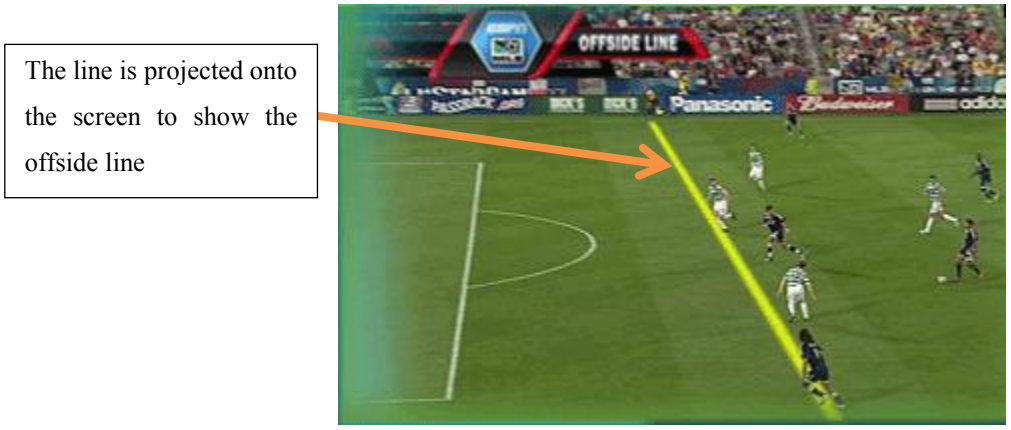

Figure 2.3 The use of augmented reality in a sport show [source: www.totalfootballmadness.com]

Multi-user 3D desktop virtual environments (from now on referred to as virtual environments) are the focus of this dissertation. Many of the ideas and the features discussed here are applicable to other kinds of virtual worlds, but discussing the full range of virtual, augmented and mixed reality research is beyond the interest and capacity of this dissertation. Some closely related research on virtual worlds is presented here.

The following subsections cover a few popular virtual world applications. While there are hundreds of such applications to choose from, those presented here represent unique examples because they can be adapted to different uses. Two virtual world examples are discussed, Second Life and ActiveWorlds. Menneche et al [24] present an introduction to Second Life and other virtual environments. They also offer a road map for research in virtual worlds in general, with the future and potentials of such worlds from different perspectives.

2.1.1 Second Life

Second Life [6] is a 3D online social virtual world. Launched in 2003 by Linden Lab, Second Life acts as a platform, giving its users a virtual world that they can build and hang out in, doing many of the things they do in real life. Unlike conventional games, there is no specific task or mission a user should accomplish, a big monster to kill, or an enemy to fight, making it confusing to users at first who try to identify it as a game.

The virtual world in Second Life is referred to as a grid, and composed of "islands" as can be seen in Figure 2.4. Some of these islands are huge and accessible by the public where others are private and owned by individuals, institutions, or companies. New islands can be created on demand and come with a price in addition to monthly fees. Second Life, charges an initial $1000 setup-fee (April 2012) for having a virtual private region (what is called an island), and $295 monthly lease (April 2012) [25] . The costs add up quickly depending on the size of the land requested and the services provided.

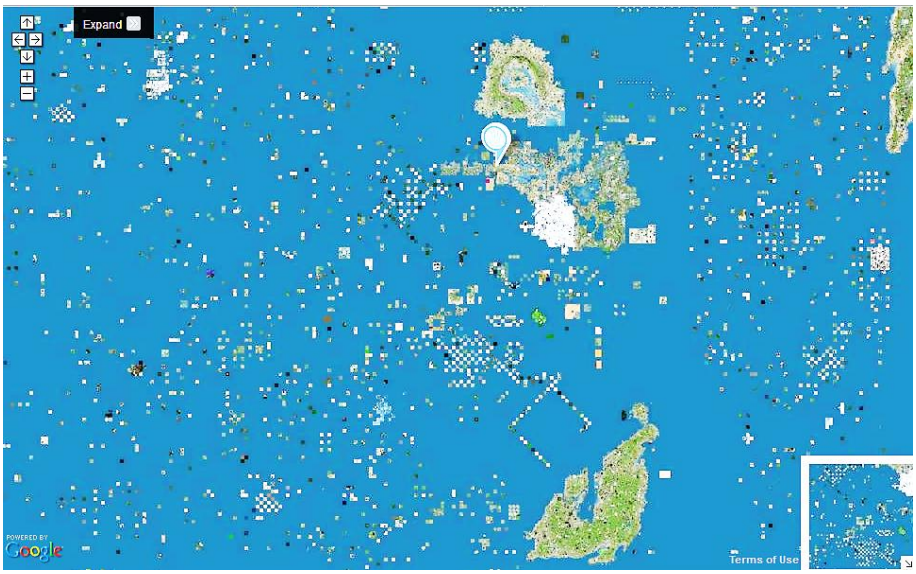

Figure 2.4 A map of some islands of Second Life viewed from Second Life's world map tool.

Second Life features the idea of user-created content. It is a dynamic world where its content is continuously updated and new content is added. The drawback is the high bandwidth requirement, which slows down the virtual world rendering in general. Second Life also includes its own scripting language, allowing users to attach behaviors to certain objects and make them respond to certain events. The scripting language is a useful feature in Second Life, but there are many constraints on what can be done and how much can be done using it. It cannot be used to implement arbitrary new features, and worse, users who wish to add objects to the virtual world have to upgrade their accounts in Second Life from standard to premium membership and buy a piece of property in the virtual world. Even then, the number of objects (or graphic primitives) is limited and constrained by the size of the land. For example, a piece of property approximately the size of the University of Idaho Janssen Engineering Building (JEB) allowed about 450 graphical primitives, enough to represent only a very simple structure representing the outline shape and floors, but not individual rooms, of JEB.

2.1.2 ActiveWorlds

Similar to Second Life in many aspects, ActiveWorlds is another big online virtual world. The project started in the 90s to provide a 3D equivalent of a web browser, and grew up over the years to become a popular online virtual world. Users in ActiveWorlds are called citizens, and they have to pay a monthly fee ($6.95/month (April 2012)) to be able to use all of the features and access all of the contents in the world. There is also a free membership called tourist mode, which provide very limited access to ActiveWorlds [26]. Figure 2.5 is a screenshot of ActiveWorlds showing the place a new user first sees after their first login to the virtual world.

Figure 2.5 ActiveWorlds Gate, the place where new users first login to.

ActiveWorlds is composed of "worlds", similar to Second Life's islands. Many of these worlds are owned by ActiveWorlds and are public for all citizens, while some worlds are owned by citizens. It is up to the world owner to make it public or private. A collection of interconnected worlds composes a universe, which makes an independent, standalone, and large virtual world. ActiveWorlds' main universe is composed of hundreds of worlds and it is the one that ActiveWorlds users have access to. Users such as institutions can buy their own private universes, to create a completely independent, to some degree customizable, virtual world. These universes come with their own servers which have to be leased from ActiveWorlds yearly, adding to the overall cost of owning a universe. There is also a limit on the number of users and the size and content of these universes. Active Worlds Educational Universe is an example service package that can be bought from ActiveWorlds. The package price is $650 with $395 annual renewal fee (April 2012) allowing only 20 simultaneous users.

Second Life and ActiveWorlds can be customized to some degree by changing the existing world content or adding new content. They are popular virtual worlds with a lot of documentation covering them. While these worlds allow customization through content creation and scripting, there is still a hard limit on what and how much users can add. They are not flexible enough to allow the creation of a new arbitrary world or application to allow new kinds of activities or use cases, such as supporting a collaborative editor inside the virtual world. They are not extensible enough to allow plugging in a new technology or the support of a 3D model format for example. They also come with a price and continuous cost as mentioned earlier.

2.2 Game Engines

One of the popular methods to build new virtual worlds is to use a game engine. A game engine is "a system designed for the creation and development of video games" [27]. A short and useful summary of game engines can be found at [28]. The purpose of the game engine is to provide reusable software components and features required when developing a game or virtual world. Usually it provides an abstract and in some cases platform independent layer on top of the underlying implementation and libraries. The philosophy behind the game engine is the idea of separating the game content and game specific behaviors from the game engine itself [29]. Figure 2.6 shows a modular game engine structure and pointing where the game engine fits in the big picture.

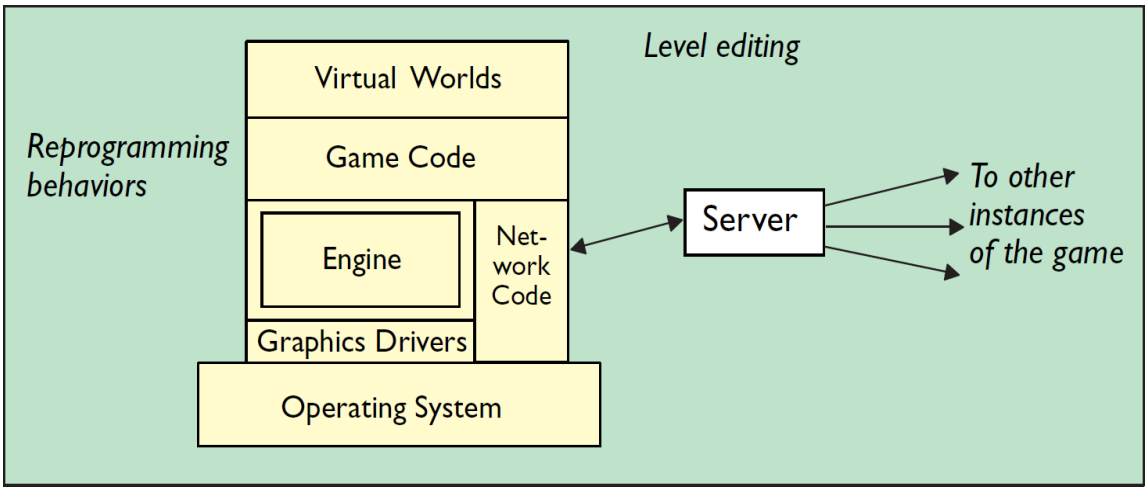

Figure 2.6 Modular game engine structure [30].

Some engines are very specialized and may be used as subsystems of other tools and engines such as the popular Havok physics engine, while other game engines provide several functionalities that include some or all of the following:

- 3D graphics rendering
- 2D drawing
- Graphical user interface (GUI)
- Physics simulation
- Multimedia support (audio, movie playback)
- Scene graph management
- Input/output and networking
- Artificial intelligence (AI)

Bishop et al, summarize the features found in many game engines. They define a game engine so that it includes only those elements that have no effect on the actual game content in addition to the game's main event loop. Those elements are indicated by dashed-lines ovals in Figure 2.7.

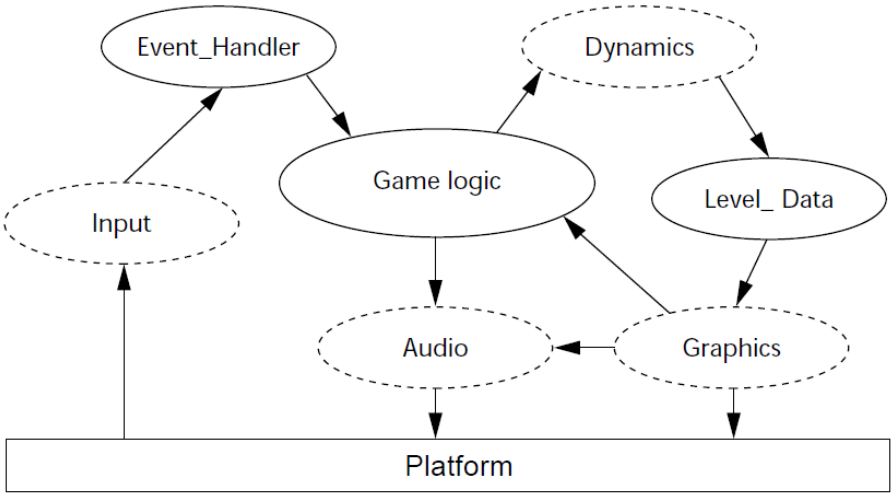

Figure 2.7 Schematic of a game [29]

Game engines are a form of middleware because they are positioned between the game itself and the underlying system. There are hundreds of game engines available for developers to choose from, covering a wide range of game designs and features. Many of these game engines, especially those used in professional games, come with a very high price tag, while other engines are open source, free or have a low price but are less robust and have fewer features. According to [31], *cryEngine* is the most expensive game engine, with a price of more than $200,000.

Among the hundreds of game engines, in addition to *cryEngine,* popular commercial game engines include *Unreal*, *CrystalTools*, *Gamebryo*, and *Jade*, just to mention a few. Open source and free engine examples include *Ogre*, *Panda3D*, *Quake3*, and *OpenSceneGraph*. *Delta3D* is a popular open source game engine used in many projects [32] [33]. Devmaster website [34], provides a comprehensive list of game engines available. A report created by ELIAS project [35], represents a good survey of many open source and low cost game engines. It also discusses the features of game engines with advice on how to pick one to develop a game.

One drawback of most game engines, in addition to the price, is limited portability [32]. Content is usually specific to the selected engine and to suit a specific game genre. The portability issue includes the ability to use the engine in very different game designs, limited portability to move the game engine between different platforms and operating systems, and limited ability to migrate game content to a new game engine if needed.

Another problem with using game engines is that they are usually large and complex black box systems. They do not always give as much of an advantage as is hoped for. According to Fristrom, a technical director and a designer at a game company, when he was talking about using a game engine [36]:

> *"You'd think it would turn a three year project into an eighteen month one, but in my experience it really only saves about six months. You'll find yourself saying things like, 'If only the engine was designed from the start for network play!' and 'If only the engine was designed from the start for a full-state save game!' And so on."*

While this discussion is informal and might not be true for all projects, it still sheds some light on the difficulties game companies face. Six months is a big time saving, but clearly, game engines do not magically solve all of the issues and complexities related to developing games and virtual worlds. They only solve a small fraction of MMO needs, and in many cases development teams have to cope with the inflexibility of the engine.

One more issue in game engines is that, due to performance requirements, the majority of them are written in C++ or C. Few of them provide binding to other languages and/or incorporate the capabilities of scripting languages such as JavaScript and Python. While C and C++ are a lot easier than assembly language, the language of the early games, programmers still require extensive experience and mastery of these languages to be able to write a meaningful game, or develop a virtual world. They lack many features and data structures found in higher level languages that save the programmers a lot of time.

The highly specialized and often expensive game engines are usually geared toward professional programmers and high performance games. A simpler and more manageable approach is needed for less trained programmers and students. Darken et al, point out that the highly specialized game engines model works perfectly for the game industry, but simply does not work for the training community [32]. A new simplified approach can be built around the programming language itself. Building many features found in a game engine into the programming language, and hiding most of the implementation details, and melting those features with the natural flow of the programming language itself, does not require the programmer to learn different tools or incur too much extra complexity in the code. The following section covers some features required by virtual world development, and the support for such features in some programming languages and libraries.

To summarize, game engines are powerful tools, built to be used for one thing as their name implies; i.e. developing games. They encapsulate many of the features and libraries necessary to build virtual worlds, making it easy to integrate many of the virtual world's functionalities. Moreover, most of them are performance tuned to meet the requirements of demanding games. However, most game engines are big, expensive, platform dependent, pieces of software. They are complex and hard to learn and master. They

are not flexible, when a new feature is needed that is not provided by the engine it has to be implemented externally.

While using a language in place of a game engine might not provide the level of specialization or performance provided by a standalone game engine, there are advantages to such an approach. First, with a very high level language, powerful features and data structures can be used to simplify the overall software design. Second, the language approach provides one consistent programming style across all APIs, syntax and features provided by the language. A game engine on the other hand, is separate from the language used in the software under development, making it hard to match what the engine provides with how the language works. This might cause the programmer to keep jumping between the concepts of the engine and the language, unlike the consistent and smooth flow of programming style using the language approach. Third, using the language allows the programmer to build all of the needed features and tools with complete freedom and maximum flexibility, without being restricted to a set of features or mechanisms found in a game engine.

2.3 Language Support for Virtual Environments

The following subsections introduce a set of features in programming languages that are essential to develop virtual environments. Some languages and libraries that provide such features are covered and discussed along the way. [37] presents a previous work aiming to help reduce the cost of building virtual environment.

2.3.1 3D Graphics

The huge advances in graphics hardware in the past two decades allowed for a dramatic leap in the degree of realism that computer graphics can achieve, with richer scenes and larger virtual worlds. However, this also increases the amount of data to manage and code to organize, exploding the complexity of such applications. Computer graphics sits at the center stage when it comes to games and virtual environments. Graphics programming is probably the hardest task in developing such applications both in terms of programming complexity and performance requirements. When creating a game or virtual world, graphics programming usually is done using one of the following five approaches:

1. A game engine.
2. A specialized programming/scripting language.
3. A graphics library.
4. A general purpose programming language that has a built-in graphics API.
5. A mixture of the above.

Depending on the application under development, one approach might be more suitable than others. Each has its own advantages and disadvantages. A game engine for example might give access to advanced features like animation, constructing and managing the scene graph, and provide the game physics, but it is not flexible or customizable; the programmer is constrained as to the languages that can be used with such an engine and the systems that can run it.

2.3.1.1 OpenGL and Direct3D

The two best known graphics libraries are OpenGL and Direct3D. OpenGL (Open Graphics Library) is a standard cross-platform graphics API. It is very well documented and supported on almost all platforms. OpenGL sets the standards for 3D software and is widely used in all kinds of graphics applications, especially in industry, design and education [38]. Direct3D is owned by Microsoft and only supported on Microsoft platforms, but it dominates the PC game industry.

The OpenGL API is available for several languages including C, C++, Java and many others, while the Direct3D API is only available in C, C++, C# and Visual Basic [39]. Both libraries include hundreds of functions that deal with core graphics programming with little or no support for window systems, 3D models, and animation. Programmers have to rely on other libraries and tools to add such support. The huge APIs, and supporting libraries create very complex programming environments. Only experienced programmers become proficient in dealing with the complexities associated with such systems. The following sample OpenGL program written in C draws a rectangle:

```
glBegin(GL_QUADS);                  // Start drawing a quad primitive
  glVertex3f(-1.0f, -1.0f, 0.0f);   // The bottom left corner
  glVertex3f(-1.0f, 1.0f, 0.0f);    // The top left corner
  glVertex3f(1.0f, 1.0f, 0.0f);     // The top right corner
  glVertex3f(1.0f, -1.0f, 0.0f);    // The bottom right corner
glEnd();
```

The code above represents a deprecated (as of OpenGL 3.0) style of rendering a set of vertices in OpenGL. Newer and more efficient functions were introduced in OpenGL version 2.0, but they are neither shorter nor simpler to use. The code leaves a lot of the rendering details out, such as color and relative location. It also does not cover many of the programming details required for this code to work such as creating a window or preparing a frame buffer.

2.3.1.2 Domain Specific Languages for Computer Graphics

Domain specific languages are designed to serve a particular domain, utilizing the theory and vocabulary for that specific domain. Usually this allows the language to be more expressive, more readable, and more productive for such domains, but it is not useful or even not usable outside its domain [40].

Shading languages are a good example of domain specific graphics languages. These languages feature C-like syntax and are used with programmable graphics processing units (GPUs). Shading languages are usually designed to achieve maximum performance utilizing the GPUs available on many computers. Cg, GLSL and HLSL are three popular shading languages. Cg (C for graphics), a language developed by Nvidia, was one of the first languages widely adopted because it is platform independent [41]. GLSL (OpenGL Shading Language), as its name implies, is designed to be used with OpenGL [42]. OpenGL and GLSL work together and share states to maximize the performance of graphics hardware. Microsoft introduced HLSL (High-Level Shader Language) to be used with their DirectX framework [43].

While shading languages can deliver very high performance graphics, they work very close to the hardware. They are very low level, compared to what very high level languages feature. However they represent a great potential with the performance they can deliver for certain graphics operations. Shading languages can be used in the implementation of such operations in the runtime systems of a very high level language. By doing so, certain graphics features gain the boost they need where performance is critical. Doing such integration falls within the interests of this dissertation, but it is left as a future work, largely because of the amount of work involved in such integration, and because it is orthogonal to the primary contributions of this dissertation.

Apart from graphics focused languages, some domain-specific languages were developed to help build virtual environments, in particular, to help develop such worlds to be used on the web, usually using a plugin inside a browser. Two languages are presented here: VRML, and X3D. Both VRML and X3D are closer to model file formats than to programming languages. Files written in these languages are fed to a browser plugin that serves as an engine that renders the data in these files to build virtual worlds.

Virtual Reality Modeling Language (VRML) was the first language to establish itself as a standard for 3D graphics and virtual reality development on the internet [44]. Graphics information is usually saved to text files with .wrl extension. If a VRML plugin is installed in the browser, the browser launches the plugin to translate textual model information into 3D graphics scenes. VRML provides a high level abstraction for 3D scenes. Objects in the scene are organized in a hierarchy and referred to as nodes. The appearance of objects can be controlled via property nodes. The whole scene is rendered by traversing all of the nodes in the graph hierarchy. Figure 2.8 presents an example scene generated using VRML. The following code provides an example of VRML file content:

```
#VRML V1.0 ascii
Separator
{
        Material {
                diffuseColor 0 0 1
        }
        Cube {
                width 3
                height 3
                depth 3
        }
        WWWInline {
                name “http://vrml.test.org/horse.wrl”
        }
}
```

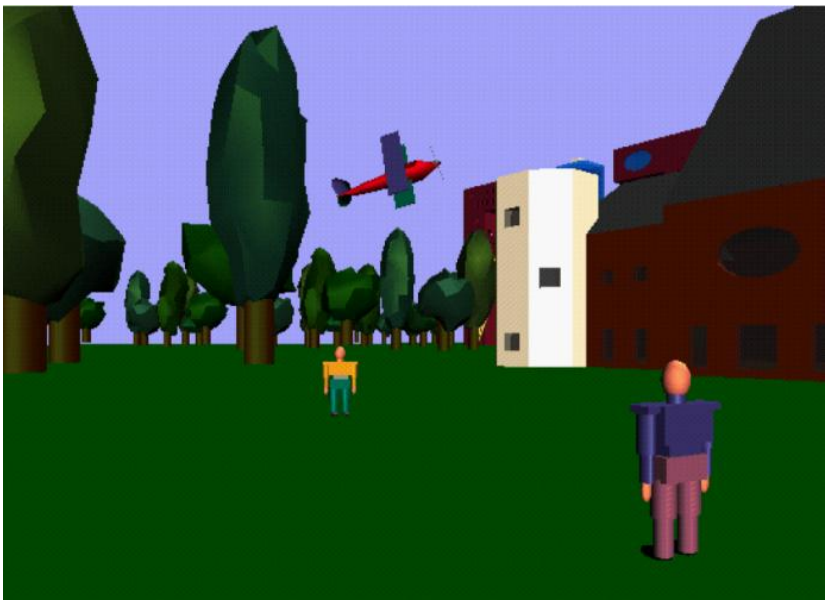

Figure 2.8 A scene with avatars generated using VRML [44]

Extensible 3D Graphics (X3D), a web graphics language based on and backward compatible with VRML, is an international standard for Web-based graphics [45]. X3D uses XML to encode graphics data and scene graph information. This facilitates moving such data between different platforms, and incorporating it into Web services [46].

VRML and X3D are examples of languages that were created specifically to facilitate the process of building virtual worlds. They are specialized tools that describe the content of the world rather than how to build it. What they are missing is the power of a general purpose programming language that is needed to cover the wide range of programming requirements necessary to build a virtual world.

2.3.1.3 Languages with Built-In 3D Graphics Support

Many languages incorporate built-in support for 2D graphics, but few provide built-in support for 3D graphics. That is mainly because most programming languages including mainstream language were designed before the 3D graphics era. Another reason is that there are standard graphics libraries that represent a portable mature alternative for what a language can provide such as OpenGL. Support for 3D graphics in a language can be done through library bindings or a using a built-in API.

Java3D represents another effort from Sun Microsystems and its partners to bring 3D graphics to the Java programming language with the first version released in 1998. Java3D is built on top of OpenGL, Direct3D or recently JOGL rather than a direct binding to a particular underlying library. It has a higher level semantics, as well as more abstractions and features that facilitate 3D graphics programming, separating it from the lower level library such as OpenGL. It supports stand-alone programs as well as Java applets that run in a browser [46]. It includes high level features such as scene graph creation and management, and is capable of loading several model file formats. The default rendering in Java3D is OpenGL. A JOGL rendering exists as an intermediate interface to OpenGL in order to support systems that do not have direct Java3D OpenGL interface such as Mac OS X.

Java3D is a rich 3D graphics API with a wide range of capabilities and features not found in lower level APIs such as scene graph support and 3D model file loading and rendering. The API defines over 100 classes in the core package alone [47]. In addition to that, three extra packages are also required in most Java3D applications bringing the total number of classes to several hundreds. Here are four packages that are usually needed to write an application using Java3D [47]:

- javax.media.j3d: Java3D core classes. Contains the Java3D low level classes such as graphics primitives.
- com.sun.j3d.util: Java3D utility classes. Provides the higher level functionalities such as scene graph creation and management.
- javax.vecmath: Vector math classes provide the math for vector, matrices, etc., used in 3D graphics applications.
- java.awt: Abstract windowing toolkit used to create and manage windows.

Due to the huge number of classes in Java3D, and the constrained and specific programming style that has to be followed when writing code in Java3D, the learning curve is very steep. Java3D provides a powerful set of tools, but the number of classes and the steep learning curve hinders programmers from using it, especially inexperienced programmers who are not accustomed to using such complex tools to write a virtual world. The following program from www.java3d.org demonstrates the use of Java3D to draw a

sphere with a light source. The amount of code and scene setup required (compared with Unicon), shed some light on the complexity of Java3D.

```
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;
import javax.vecmath.*;
public class Ball {
public Ball() {
  // Create the universe
  SimpleUniverse universe = new SimpleUniverse();
  // Create a structure to contain objects
  BranchGroup group = new BranchGroup();
  // Create a ball and add it to the group of objects
  Sphere sphere = new Sphere(0.5f);
  group.addChild(sphere);
  // Create a red light that shines for 100m from the origin
  Color3f light1Color = new Color3f(1.8f, 0.1f, 0.1f);
  BoundingSphere bounds =  new BoundingSphere(new Point3d(0.0,0.0,0.0), 100.0);
  Vector3f light1Direction = new Vector3f(4.0f, -7.0f, -12.0f);
  DirectionalLight light1= new DirectionalLight(light1Color, light1Direction);
  light1.setInfluencingBounds(bounds);
  group.addChild(light1);
  // look towards the ball
  universe.getViewingPlatform().setNominalViewingTransform();
  // add the group of objects to the Universe
  universe.addBranchGraph(group);
}
public static void main(String[] args) { new Ball(); }
}
```

Java was extended by Sun Microsystems with an alternative 3D API in 2003. The extension was done through OpenGL binding that was added to Java and under the name JOGL. The new bindings provide full access to OpenGL, including most of its extensions, and integrate with Java graphics libraries such as AWT and Swing widget sets [46]. JOGL is a binding to OpenGL with one-to-one mapping to its functions.

Using Java frees the programmer from memory management, has windows support, and brings object oriented design to the application, but the programmer still has to deal with all of the low level detail of OpenGL and the complexity associated with all of the setup and maintenance of the application code and scene data that maps very closely to C OpenGL code.

The last example in this section is Fran, a graphics subsystem embedded in Haskell, one of the popular functional programming languages. Haskell is a purely functional language, where the program is composed of definitions that describe what needs to be done rather than describing exactly how it is done [48]. Hudak et al., explain the full details of the Haskell language in their reports in [49] and [50]. Elliott covers the implementation details for Fran in [51].

Fran, (for “functional reactive animation”) extends Haskell with domain specific vocabulary for modeled animation [52]. By using the declarative features of the host functional language, Fran aims to capture and model what the animation is instead of how to present the animation. It adds concepts to Haskell such as 2D images, 3D geometry, transformations and sound. In addition to that it adds some supporting types such as vectors and colors and also adds the notations of reactive behavior to support animation. In Fran’s context, a behavior is defined as a time-varying value. For example, the position of a 3D moving object changes over time, hence the position is a 3D point-valued behavior. The model of the 3D object itself is a 3D geometry-valued behavior. The animation of the object is viewed as an image-valued behavior. The following is a program written using Fran constructed from fragments of code in [52]; the result is displayed in Figure 2.9.

```
importX :: String -> GeometryB
uscale3 :: RealB -> Transform3B
(**%) :: Transform3B -> GeometryB -> GeometryB
type Spinner = RealB -> User-> GeometryB
potSpin1, potSpin2 :: Spinner
withSpin :: Spinner -> User -> ImageB
rotate3 :: Vector3B -> RealB -> Transform3B
time :: RealB
withColorG :: ColorB -> GeometryB -> GeometryB
teapot = uscale3 1.5 **%
            importX "teapot.x"
potSpin2 potAngleSpeed u =
    spinPot potColor potAngle
      ‘unionG‘ light
  where
      light = rotate3 yVector3 (pi/4) **%
              translate3 (vector3Spherical 1 0 time) **%
              uscale3 0.1 **%
              withColorG white ( sphere ‘unionG‘ pointLightG)
       potColor = colorHSL time 0.5 0.5
       potAngle = integral potAngleSpeed
```

Figure 2.9 A sequence of frames, the result of a program written in Haskell using Fran [52]

While Fran results in a very concise code with the expressive power of the host functional programming language, Haskell, it is not necessarily easy to understand in some cases, especially for less experienced programmers. According to [53], the series of implementations that Fran went through increased its complexity with unpredictable performance. It also depends completely on models created externally, allowing them to be imported and composed. In addition, Fran is a very domain-specific language and highly geared to do one thing, that is animation, and it is only implemented on the Windows platform [54]. In contrast this dissertation aims for a more comprehensive set of features that covers not only one very specific domain when developing virtual worlds, but different aspects of this development process. Various subsets of this work are applicable and useful in other domains, not just the domain of virtual environments that provided initial impetus. For example, while concurrency is essential for a virtual world server, and also valuable in a client to boost its performance, it is general enough to be used in all kinds of domains.

This discussion concludes the 3D graphics section that covered several approaches to 3D graphics applicable to virtual worlds. The next section covers related work on the corresponding input/output

problem: language support for 3D Object selection. The ability to click on and pick objects in a 3D scene opens the door for many virtual worlds and games features. These features provide a base for a rich user interaction experience with the virtual world.

2.3.2 User Interaction

Many programming languages have support for user interaction in 3D scenes through 3D object selection. Some of these languages provide very low level APIs that directly reflect functionalities in the underlying graphics libraries. A popular example is the C API for OpenGL. In OpenGL it is up to the programmer to make numerous calls and prepare buffers to collect and process all of the results. Other languages hide some or all of these underlying implementation details to provide a higher level API at the language level.

Different programming languages and graphics libraries have various mechanisms to implement 3D object selection. These include, but are not limited to:

- Color-coding
- Ray tracing
- Special rendering modes

Color-coding involves rendering each primitive in a unique color in an off-screen window buffer so that the user does not notice this process, and then reading the pixel under the current cursor location. The color value of the pixel determines the primitive that the user selected. This technique provides good performance, especially with a small number of selectable objects. The drawbacks of this technique are:

- There is no depth information with the selected object.
- Only the closest objects to the cursor (camera) can be selected. Objects that hide under other objects cannot be selected.
- The unique color must be the same when it was rendered and when it was read back; lighting, dithering or any other setting that might affect how the system interprets the color could affect selection.

Ray tracing selection works by generating a pick ray from the mouse location to the far z-plane and testing if this ray intersects with objects in the scene. The special rendering mode is discussed in the next subsection.

2.3.2.1 OpenGL Selection Mode

OpenGL provides a mechanism for object selection through a special rendering mode. In selection rendering mode, instead of rendering the scene as color values to the color buffer, only "names" are rendered to a selection buffer. In this special mode the programmer supplies storage for results and sets up a special "pick" matrix with a view volume centered around the mouse cursor (**Error! Not a valid bookmark self-reference.**). Objects that are rendered into this selection viewing volume are reported to the user as selection hits [38].

Picking is a special case of selection in which clipping is set up such that only a small region of the screen is visible: whatever is visible is then deemed to have been "picked". The programming steps are:

- Restrict rendering to a small region near the pointer
- Use gluPickMatrix() on the projection matrix
- Enter selection mode; re-render scene and give 3D objects unique integer identifiers
- Primitives drawn near cursor cause hits
- Exit selection; analyze hit records

When using OpenGL selection, the programmer has to work with 6 functions dedicated to selection plus several other functions that must be called to make selection work, for a total of 11 functions. In addition, the programmer must also setup a data structure to hold the selection results.

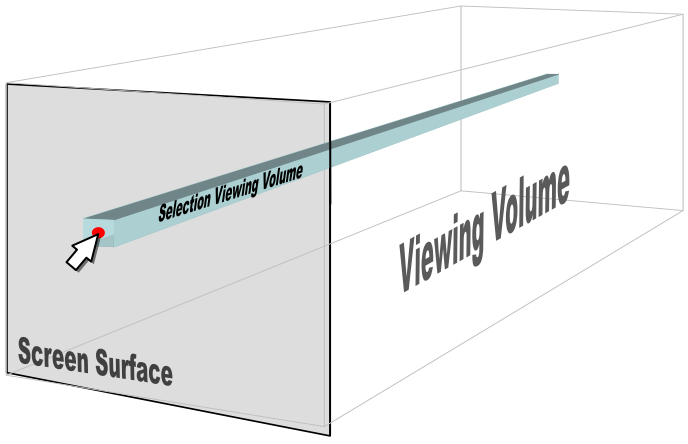

Figure 2.10 Selection viewing volume centered around the mouse cursor

2.3.2.2 Java3D Object Selection

Java3D provides a rich class library for 3D graphics as discussed in subsection 2.3.1.3 which include picking utility classes [55]. These picking utility classes are part of a general interaction framework built into Java3D. This framework revolves around what are called *Behavior* classes, *Behavior* objects and utility classes in Java3D, which form the foundation of animation and interaction. These are used to provide an interface with the keyboard and the mouse facilitating things like world navigation, object interaction and also object animation.

A *Behavior* object is at the heart of interaction and animation in Java3D. It is an instance of an object that implements an abstract *Behavior* class and can be linked to user defined handlers providing sound and animation in the virtual world. These behavior objects are inserted in the scene graph to respond to events. An event can refer to either user-triggered events such as key press or a mouse click, or to an in-world event such a timer or the collision of objects. Table 2.1 gives examples on how a *behavior* object can be used depending on the event or stimulus) and the object affect by the event. For example, a geometry object such as a sphere can use a *behavior* object such that the level of detail of the sphere changes depending on the view location, the closer the view the more detailed the sphere gets, the farther the view, the less detailed the sphere gets.

Table 2.1 Applications of Behavior in Java3D [55]

stimulus (reason for change)	**object of change**			
	TransformGroup (visual objects change orientation or location)	Geometry (visual objects change shape or color)	Scene Graph (adding, removing, or switching objects)	View (change viewing location or direction)
user	interaction	application specific	application specific	navigation
collisions	visual objects change orientation or location	visual objects change appearance in collision	visual objects disappear in collision	view changes with collision
time	animation	animation	animation	animation
view location	billboard	level of detail (LOD)	application specific	application specific

Two basic approaches can be used to achieve picking in Java3D: 1) use objects of built-in special purpose mouse *Behavior* classes for simple actions such as zoom or rotate 2) or use instances of more general custom picking classes that include code with user defined actions. Using the first approach for example, the picking package includes classes for pick/rotate, pick/translate, and pick/zoom. The user can press the mouse button over an object and drag to achieve one of the functionalities of rotate, translate or zoom based

on the class type. The picking classes can be setup to use different mouse buttons to give access to the different functionalities simultaneously. The following steps summarize the process of making a selectable object in Java3D [55]:

- Create a scene graph
- Create a picking behavior object with root, canvas, and bounds specification
- Add the behavior object to the scene graph
- Enable the appropriate capabilities for scene graph objects

The second approach involves adding user-defined actions and creating custom behavior with custom picking classes. It also requires enabling/disabling many attributes for the required objects and their corresponding behavior objects.

In summary, Java3D has a high level object selection mechanism compared to the C OpenGL API, but using it still requires a substantial amount of work by the programmer [56]. There are many packages and classes to keep track of and more attributes to remember and object capabilities to turn on or off. Not to mention the inherent complexity in using the whole graphics library with more than 150 classes, which dictates how to setup the selectable objects, how to add them to a specific canvas and where to insert them in the scene graph before the whole selection process can be made possible.

2.3.3 Concurrency

Pure functional and logic programming languages such as Prolog are famously parallelizable. Many very high level rapid prototyping and scripting languages such as Python, Ruby and (prior to this work) Unicon, feature no concurrency or only user-level concurrency via a global interpreter lock (GIL) [57]. These languages may convey substantial practical advantages in specific application domains, and have large existing code bases that are unable to take advantage of the advances in modern multi-core computer hardware. Such languages must either acquire true concurrency with access to kernel-level threads or face a lingering decline as they become less and less able to utilize the capabilities of the hardware on which they are run.

Different programming languages employ different techniques to achieve parallelism. Some languages such as C and C++ rely on libraries, while others have built-in functions, control structures, and primitives to support concurrency. Among languages that support concurrency, some support only explicit concurrency while others provide implicit concurrency or both, finding parallelism in the compiler or runtime system. Implicit concurrency is appealing in special purpose, domain-specific languages such as functional, dataflow and scientific languages; however, achieving strong performance for implicit

concurrency in general contexts has been elusive. Load-balancing and appropriate granularity selection are difficult to automate.

Haskell is an example functional language for which implicit concurrency has been implemented [58]. The nature of computations in pure functional languages, which are free of side effects and depend heavily on graphs, facilitates such implicit parallelism by assigning different parts of the graph to different threads [59] [60].

Paalvast et al, described Booster [61], a high level parallel programming language that can be translated into lower level languages such as FORTRAN and C. The language features the idea of separation of algorithm description from algorithm decomposition and representation. After the algorithm and the data/code decomposition are described, a transformation can be done automatically to achieve concurrency [61].

ALLOY [62] is an example of a weakly-typed, statically-scoped parallel programming language based on the functional and object-oriented paradigms. ALLOY is similar in many respects to Unicon and its predecessor Icon. It provides features to express parallel algorithms and their related control structures including synchronization and mutual exclusions. Mitsolides and Harrison [63] describe the concept of "replicators" in ALLOY, which are control structures that provide a new view of generator. Replicators help deal with problems related to generators in a concurrent environment, such as expression failure, backtracking and side effects.

Relatively popular modern concurrency-oriented, user-friendly languages include Java (java.com) and Erlang (erlang.org). Java features portable true concurrency and mitigates the pain of writing locking code for concurrency synchronization using monitor semantics. Erlang is arguably higher level, with a more esoteric functional syntax and a message passing model for communication and synchronization [64].

The most famous scripting language, Python, and its popular competitor Ruby feature a Global Interpreter Lock, or GIL [57]. The GIL means that the language's interpreter main loop has a lock (a mutex) such that only one thread is allowed to execute in the interpreter at any given moment by acquiring the lock. This means that even if the program has multiple independent threads, effectively, only one thread at time is running. Switching between threads is done using a scheduler every certain number of instructions or if the running thread blocks for a long operation such as doing an I/O [65]. Experimental implementations such as Jython and Iron Python feature true concurrency, but users tend to stick with original interpreters such as CPython despite their use of a GIL. Several efforts have been made to improve Python performance in multi-threaded applications, however, GIL it is still present and prevents true concurrency in CPython [66].

Stackless Python is a modified version of the Python programming language with a goal of making thread programming easy [67]. The language adds "tasklets" or "microthread" to Python which can be thought of

as user-level light weight threads similar to co-expressions in Unicon which are discussed in section 3.3. In addition to microthreads, Stackless Python features channels to communicate between threads, and also a built-in scheduling mechanism used to schedule threads in the language. Because these micro-threads are not mapped to OS threads, they cannot run in parallel and so, Stackless Python does not have true concurrency. The following short example from [68] demonstrates how tasklets (task and task2) are created in the language, how a communication channel be created (ch) to be used by a sender (Sending()) and a receiver (Receiving()), and how the threads are run (stackless.run())

```
import stackless

def Sending(channel):
   print "sending"
   channel.send("foo")

def Receiving(channel):
   print "receiving"
   print channel.receive()

ch = stackless.channel()
task = stackless.tasklet(Sending)(ch)
task2 = stackless.tasklet(Receiving)(ch)
stackless.run()
```

Lua is a scripting language designed with extensible semantics and intended to be embedded into other applications [69]. Skyrme *et al* in [70] described the design and implementation of a thread library for Lua. Lua allows the host C program to have multiple Lua programs or what is called *Lua states*. Parallelism is exploited by running different *states* on different threads. Threads in Lua do not use shared memory model, instead they only interact using messaging passing through communication channels. Channels have to be created and destroyed explicitly. The restricted communication model used between threads in Lua limits their use and scalability to certain kinds of problems where parallel tasks are independent and shared data is relatively small. Otherwise, the time it takes to transmit large amount of data between threads quickly dominates the execution which eliminates the benefits of running multiple threads.

The SR (Synchronizing Resources) programming language is a language designed specifically for developing parallel programs. The parallelism in SR is mainly exploited through message passing [71], even though it provides mechanisms for techniques like rendezvous and remote procedure call. SR has a highly expressive message passing interface, which is useful in many applications, but it is considered by some to provide a low level abstraction in applications where other concurrency mechanisms can be used [72].

OpenMP (Open Multi-Processing) is a shared memory, multi-platform API. It achieves concurrency through a unique approach using compiler directives, and a combination of library routines and

environment variables [73]. The directives enable the compiler and the runtime system to create several threads when appropriate to accomplish certain tasks, without direct intervention from the programmer. OpenMP provides a simple mechanism for parallelizing some parts of the program. Most of the work is handled automatically. The directives can be skipped by the compiler if a sequential version is required. The following C++ example uses OpenMP to initialize an array in parallel:

```
#include <cmath>
 int main()
 {
   const int size = 256;
   double sinTable[size];
   #pragma omp parallel for
   for(int n=0; n<size; ++n)
     sinTable[n] = std::sin(2 * M_PI * n / size);
 }
```

The line "#pragma omp parallel for" instructs the compile to generate parallel code for the *for* loop. Despite its simplicity, OpenMP requires compiler support, and in some cases can introduce race conditions that are hard to debug, especially with the little control that the programmer has over the threads. These issues and more are addressed in [74] [75].

The last area of related research to the contributions of the dissertation is non-player characters. The following section covers some work in the literature regarding this topic.

2.4 Non-Player Characters

A non-player character (NPC) refers to a character in a game or a virtual world that is controlled by the computer. The term first came from pencil and paper role playing games where some characters in the game are controlled by the game master who organizes and moderates the game. It is used to distinguish NPCs from player characters (PCs) which are controlled by human players. NPCs play different roles in virtual worlds including populating the virtual worlds with different kinds of inhabitants. They can be monsters or enemy characters that a player has to kill. They can be pets or friends that accompany the players helping them achieve some goals in the game. They can also be quest givers or merchants offering variety of goods. All MMOs include NPCs filling in various roles in the game. This section reviews the current state of the art in NPC architecture and establishes the context for the corresponding evaluation chapter of the dissertation.

Redfern and Naughton [76] discuss the use of modern technology and the advances in CVE research in distance education. They propose that CVEs, specifically where users can interact with each other and with NPCs, are suitable tools to improve education.

The Belief-Desire-Intention (BDI) agent architecture tries to enhance the users' interaction with NPCs by developing behavior models that resemble human behavior [77]. Merrick and Maher presented a design for NPC behaviors in computer games based on motivated reinforcement learning [78]. They also presented an adaptive NPC model which considers the impact of the changing environment in open-ended worlds on the NPC. The adaptive model lets the NPC evolve and adapt to the changes in dynamic environments and does not limit its behavior to the pre-programmed rules [79].

CHI Systems, under contract to the U. S. Army Research Institute developed a training system called Virtual Environment Cultural Training for Operational Readiness (VECTOR). In this system they applied experiential scenario-based virtual environments to train solders in cultural familiarization. They incorporated cognitive-model-controlled NPCs that can evaluate and respond to the cultural propriety of the trainee's actions [80].

Art Fossett's blog describes a non-player character created as a virtual object [81]. Making an object look humanoid is a challenge in Second Life, but can be accomplished using sculpted primitives, which are a restricted form of 3D model that graphic artists can produce with commercial grade tools at substantial effort. Fossett couples this humanoid-looking virtual object with an external chat program called a PandoraBot, which implements AIML (an XML-compliant language called Artificial Intelligence Markup Language) [82] and plays a role similar to the PENQ dialogue model which is covered in subsection 8.1.3.

Doron Friedman et al built a Second Life NPC by taking an ordinary user avatar and attaching a virtual object (a ring) to it that turns the avatar into a puppet controlled by an external program. This NPC can move around the environment, albeit with very simple rules for essentially random movement [83].

One popular use of NPCs and quests is to employ them for educational purposes. Such use in multi-user virtual environments for education has been discussed in the Quest Atlantis project [84]. Quest Atlantis is a learning and teaching computer game that leverages commercial game strategies to provide a 3D multi-user environment to immerse children, ages 9-12, in educational activities. It allows children at participating elementary schools to engage in a virtual world and perform educational activities, chat with other users, and build virtual spaces in some designated plots [85] [86].

Creating NPCs that are run and controlled using clients outside the virtual world and its server has been studied in Second Life and enabled by providing an API to communicate with the world server. To summarize, although several interesting prior related experiments have been conducted to add non-player characters to Second Life. At the time of writing this dissertation, no Second Life NPC's are known to be used to deliver tutorial quests from declarative specifications as in the case of PENQ.

3 Unicon Programming Language – Design and Implementation Overview

The Unicon programming language is an object-oriented descendant of the Icon programming language [15] [16]. Icon integrates Prolog-like goal-direction and implicit backtracking within a conventional syntax and an imperative semantic core. Icon's traditional domain is string and file processing, and the rapid development of experimental algorithms and data structures. Unicon is an open source project available at unicon.org. This chapter gives a summary of the language and the features that were built upon or extended in this dissertation.

3.1 An Overview

Unicon forked off the Icon project in 1999, inheriting from it very high level features and concepts, such as string scanning environments, heterogeneous data structures, co-expressions, generators and multi-platform support. The Unicon language is a superset that extends Icon along two dimensions: features such as object oriented programming and packages for larger-scale projects, and richer access to common OS features such as those common to POSIX and Windows platforms with extensive support to modern I/O capabilities such as graphics, networking, and databases. Unicon has active development and user communities, and is used in organizations such as the National Library of Medicine and AT&T Labs Research. According to sourceforge.net, where the Unicon project is hosted, the statistics show about 90 downloads per month in 2011 from several dozen countries around the world. That gives an idea about the attention the language attracts among users around the globe, including some of the features that this dissertation has been introducing into the language.

Unicon has syntax similar to that of Pascal and C. It is an interpreted dynamically typed language. Unlike Icon, which targets small programming tasks, Unicon is well suited for huge projects and event driven applications, helped by many libraries including a complete GUI package and also a GUI builder tool called IVIB [87]. The following is a hello world program in Unicon:

```
procedure main()
   write("Hello World!")
end
```

A Unicon program is composed of a sequence of *expressions*. A Unicon expression might produce a result, in which case it *succeeds*, otherwise it *fails*. Failure here does not mean things went wrong, it means that given the values and the state of the variables and hardware involved in the expression, it cannot produce a result. Expression success and failure is the driving logic of the control flow of a Unicon program. `i > j` for example succeeds and produces the value of the variable `j` if its value is less than that of `i`, otherwise the

expression fails and produces nothing. This allows a simple and intuitive way of writing expressions like "if x < y < z then …".

Unicon also features *generators* and goal directed evaluation with implicit backtracking. A generator is an expression that produces a sequence of results. These results can be generated explicitly by requesting more results from the generator, or implicitly by backtracking on failure forcing the generator to resume and generate new results. Generators maintain a full state of their data and control flow at the point of their suspension allowing them to resume at any point of their execution. The following is a simple example demonstrating the use of a generator and implicit backtracking:

```
if i := !10  & i=5 then write(i)
```

In this expression, !10 represents a generator that can produce the sequence of numbers from 1 to 10. The first value assigned to i is 1, and the evaluation proceeds to the i=5 which fails causing the evaluation to backup, and forcing the generator !10 to produce the next value. The evaluation continues in the same manner until i gets the value 5 which causes the second test i=5 to succeed, which allows the if statement to proceed to the then part. The outcome is a 5 printed to the standard output. The evaluation then moves on to the next expression after the if statement.

Unicon also features keywords. A keyword in Unicon is a predefined global symbol, distinguished from ordinary variables by a leading ampersand, whose value is governed by the language control structures and built-in functions. Keywords are key components in both Icon and Unicon programming languages [15] [16]. is an example keyword in Unicon, used to give the CPU time in milliseconds since the start of the Unicon program.

The following sections give a context for the two main components in Unicon that this dissertation aims to improve and build on, 3D graphics and co-expressions. The new improvements include the addition of many missing features and performance enhancements in the 3D facilities, and enabling co-expressions to run concurrently.

After studying several games and virtual worlds, several features/aspects found in such applications but identified as either missing from Unicon or requiring better support include:

- 3D graphics performance
- 3D object selection
- better image file formats
- 3D model file support
- better texture support, including the ability to modify textures at runtime

This is not a complete list but rather focuses on items that are easily visible in many virtual worlds. Some of these items are more essential than others, but this dissertation aims to address them all to provide a more complete 3D graphics feature set for virtual world development in Unicon.

The last section in this chapter briefly discuss two other important features in games and collaborative virtual environment development: networking and audio/VOIP support. Unicon has a mature networking infrastructure and experimental VOIP facilities. Extending and improving these features, especially on the VOIP side, falls within the interest of this dissertation. However, due to the already long list of tasks that this dissertation promises to accomplish, VOIP will be left for future work. During this work, minor improvements to the Unicon networking facilities were made to support new functionalities in the virtual world, but they are not central contributions for this dissertation.

3.2 3D Graphics

Unicon inherited Icon's 2D graphics [88]. In addition to several enhancements Unicon added to 2D graphics, it also added support for 3D graphics in 2003 using OpenGL as the underlying library [89]. The 3D facilities in Unicon provide a high level subset of OpenGL's capabilities including drawing primitives, transformations, lighting and texturing.

Many of the new 3D graphics features are extensions of Unicon's 2D API. Unicon reduces the number of functions needed compared with the standard OpenGL C interface. Instead of the more than 250 OpenGL [38] functions that C programmers have to learn, Unicon provides about 30 3D functions. For example to draw a point in OpenGL, a call to glVertex() must be made between glBegin() and glEnd(). Depending on the argument number and type, glVertex() takes several forms such as:

glVertex2i()	glVertex3i()	glVertex2iv()	glVertex3iv()
glVertex2f()	glVertex3f()	glVertex2fv()	glVertex3fv()
glVertex2d()	glVertex3d()	glVertex2dv()	glVertex3dv()

These functions are replaced by one function in Unicon, DrawPoint(). The Unicon runtime system handles the different data formats. This is the case for many other functions in OpenGL.

Unlike OpenGL, where the program has to have a display() function to keep track of all the scene content and draw it again every time the screen needs to be refreshed, Unicon does this job implicitly. For this purpose, Unicon attaches a display list to each opened 3D window (Figure 3.1). This list is just a regular Unicon list containing all of the information needed to recreate the whole scene. Every time a new primitive is added to the scene, color is changed, or a transformation function is called, a corresponding element is added to the list remembering what needs to be done, where, and any other information needed with that element.

The display list becomes large for complex scenes, limiting the scalability of 3D programs. Another function named WSection() provides a simple form of scene partitioning. The function allows the programmers to create sub slices that can be marked to be skipped during rendering. When developing a virtual world, this feature enables things like skipping rooms that are either far from or not visible to users.

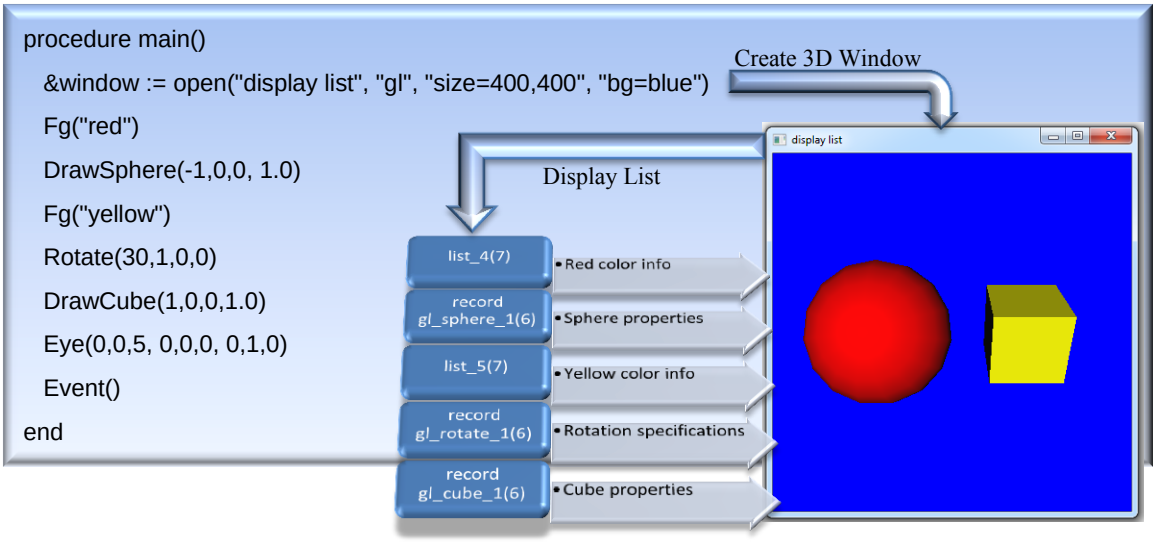

Figure 3.1 A 3D window in Unicon keeps track of all the scene content using a display list

WSection() expands the scalability of Unicon programs but does not solve the 3D performance problem for scenes with very high polygon count, especially those with 3D models loaded from model files, which are introduced to Unicon as part of this dissertation. One limit to performance in such scenarios is the way data is represented in the display list. For a large mesh, thousands of vertices need to be stored in the display list element that corresponds to that mesh. This element is also a Unicon list. Every time the scene needs to be refreshed the data has to be converted to the appropriate data format and copied to a C array. This array is then passed down to OpenGL for final rendering. For scenes with complex meshes, this repeated conversion and data copying takes most of the execution time, consuming more than 80% of the time in some experiments. This was a great opportunity for improving the performance of programs of this kind, which applies to many graphics applications including virtual worlds.

One important missing aspect in Unicon's 3D graphics facilities was the ability to interact with the 3D scenes. The ability to click on objects in the scene, inspect and manipulate them is a very important feature in most games and virtual worlds. 3D object selection support was added to Unicon as part of this dissertation research and is covered in Chapter 5.

3.3 Co-expressions

In a goal-directed evaluation language with generators, the results produced by an expression are limited to where that expression happens to be in the program. In Unicon, once the evaluation of an expression has ended, the expression state including all contained generators and their state is lost; the only way to generate new results is by resetting the expression and restarting its evaluation from the beginning. Co-expressions overcome these limitations, allowing an expression to be resumed at any time and any place [16].

Co-expression work in a similar fashion to that of coroutines found in many languages and described in [90] and [91], except that co-expressions can be composed from arbitrary expressions, making them finer-grained than coroutines, which are limited to the function call level. A co-expression can be thought of as a light weight thread that has a reference to an expression and its environment. Unlike the POSIX threads C implementation, where a thread corresponds to a standalone function, a co-expression can be as simple as one single expression, and is created using the syntax:

```
create expr
```

When the program is first run it starts in the main co-expression and can be accessed in the program via the `&main` keyword. A co-expression can be stored in and used from a regular variable. It can only be run synchronously by explicit or implicit control transfer from another co-expression. Control is transferred between co-expressions by a transmit-and-activate operator. This transfer of control is a form of synchronous communication between threads. At the two ends of the communication, the calling co-expression is blocked and the called co-expression runs. A co-expression `C1` can activate another co-expression `C2`, by doing `x@C2`. `x` is an optional value to be transmitted from `C1` to `C2`. `C1` waits until it gets activated by `C2` or any other co-expression directly or indirectly activated by `C2`. Implicit activation takes place whenever a co-expression produces a value or falls off its end. With implicit activation, the co-expression activates its parent (the last co-expression to activate it).

Despite the limitations that co-expressions overcome and features they provide, they have a limitation of their own, and that is the ability to run only synchronously; at any given moment, only one co-expression can be active, even though many co-expressions might be independent of each other. This is another subject where this dissertation aims to improve the language. With most hardware featuring multi-core capabilities, it is natural for many applications especially demanding ones, such as virtual worlds, to be written with multi-threading support. Two main reasons motivate the addition of concurrency support to Unicon when building a collaborative virtual world:

1. Virtual world client application benefits: the virtual world client performance can be boosted by utilizing multi-core hardware, pushing up the frames/second to maintain a smooth and enjoyable

experience in the virtual worlds. It also allows for bigger and richer scenes with more horse power to handle more data. This is critical in many situations where performance is the limiting factor.

2. Virtual world server benefits: the performance of the server can also be greatly improved by allowing more threads to serve more clients. This is typical in server design and implementation giving the server the flexibility to scale better with more users and more interactions between clients. It also makes the server less prone to problems such as high latency or to what appears as a disconnection because the server is doing a slow disk IO operation.

The introduction of concurrency into the Unicon language is discussed in depth in Chapter 6.

3.4 Other Features that Support Virtual Environments

Unicon includes other features necessary for virtual worlds development. This section covers two of these features, networking and audio, that are not part of this dissertation's contributions; they are discussed briefly here because of their important role in developing collaborative virtual environments, and for the sake of completeness in discussing language support for virtual environments. These features are discussed in detail in [92] and [93].

3.4.1 Networking

Unicon includes a high level networking interface with built-in support for protocols such as TCP, UDP, HTTP and POP. Network connections are opened and closed like regular files using Unicon's open() and close() functions with the appropriate parameters. Data can be read or written to these connections using read() and write() functions by passing the opened connection as their first parameter. The following short example program demonstrates the ease of use and simple interface of HTTP connections, open() with mode "m" For messaging. The program downloads a remote file specified by a URI on the command line, and saves it as a local file.

```
link basename
  procedure main(argv)
  f1 := open(argv[1],"m")
  f2 := open(basename(argv[1]),"w")
  while write(f2, read(f1))
end
```

Despite the simple networking interface that provides access to a wide range of complex capabilities in the underlying implementation, extensions and improvements were still needed to support features in virtual worlds. Real time applications cannot afford to wait for a long time for a connection to open or a slow server to respond. The open() function was extended to allow a timeout parameter that puts an upper limit on how long an application is willing to wait before giving up on a given connection.

For the same reason, a new non-blocking read() function was needed. The new function named ready() was added to serve that purpose. ready() is similar to read() in most aspects except that unlike read(), it returns immediately with whatever data is available on the connection. If no data is available, ready() simply fails (returns indicating no data is available) in Unicon terms. This behavior is critical in real time applications such as virtual worlds.

A good example of language/application co-design is the listener mode, with which a server allows new connections while simultaneously handling existing users on the same thread. Unicon's original "network accept" server mode (open() mode "na") was a blocking operation and required one process (or thread) per user. Empirical use in the CVE system motivated the addition of a "network listener" mode, a non-blocking server open(":port", "nl") that enables a single process or thread to handle new connections while serving multiple existing user connections.

3.4.2 Audio and VOIP

Audio and voice chat are essential parts of games and virtual worlds. Unicon was extended to support such capabilities. Audio files can be played using the function PlaySound(). Existing functions open() and close() were extended to support VoIP session opening and closing respectively. Extending these familiar functions and overloading them with new jobs simplifies the tasks of programmers learning and using the VoIP interface [93]. VAttrib() is similar to an existing Unicon graphics function WAttrib(), was also added to control the attributes of a VoIP session.

Unicon's VoIP interface is a modest extension of the file data type. The function open() with mode "v" for voice, opens a voice session at a specific port and returns a handle to that session. It takes the port number and additional optional parameters that allow the programmer to specify multiple destinations. This function fails if the sound device is reserved by another program or if it cannot open a socket for the RTP protocol. To close a voice session, the function close(x) is passed a voice session handle.

In order to make a meaningful voice connection, a program adds destinations representing other users' voice sessions. The following Unicon program opens a voice session at port 4500 and establishes a voice connection with a destination at the time the session is opened.

```
procedure main()
  local vsession
  vsession:= open("4500","v","jef:128.123.64.48:5000")
  write("Voice session is opened, Press <Enter> to close:")
  read()
  close(\vsession)
end
```

Unicon VoIP sessions can add and drop destinations, and change the voice session settings on the fly using the VAttrib() function. Users do not have to close the voice session and lose the connection just to change one attribute such as the bandwidth level. VAttrib() also allows users to perform queries about those who are listening to the current voice session especially in the case of a multicasting situation or n-user conferencing. VAttrib() takes two parameters: the voice session handle and a string of attributes.

PART II Contributions:
Language Features and Co-design

4 Unicon as a Graphics Engine

Unicon, like many other very high level languages such as Python and Ruby, provides very powerful data structures like lists, tables and sets that are suitable for representing the virtual world state and data. Unicon also provides high-level APIs for graphics and networking facilities [15] [16]. These features make it an attractive starting point for co-design. Although language extension is desirable in many cases, each addition to the language required for virtual environment features should be small and have a minimum impact on the language and its performance, especially if the change is visible at the language level, such as adding a new function. Different parts of the language have been extended or improved throughout the lifetime of the project. Some of these improvements and additions are still ongoing. The next several sections highlight some of these major extensions.

4.1 3D Graphics API

Unicon's 3D graphics facilities discussed in section 3.2, take many burdens off the programmer. Unlike OpenGL, Unicon has a built-in support for mainstream window systems. Opening windows and handling input events comes free of any extra work

Despite the high level of the original Unicon 3D API, several features were either missing, or did not meet the performance requirement while developing the CVE virtual environment. As part of this dissertation research, these features were improved and extended over time to include more capabilities using Unicon 3D graphics. This includes a way to manipulate the OpenGL matrix stack, support for dynamic texturing, texture buffering/caching for better performance, JPEG and PNG image file format support for textures in addition to the language supported GIF format, vertex normals support for better smooth shading, and several other improvements including 3D selection discussed in Chapter 4.

4.2 Improving 3D graphics performance

The 3D performance of programs written in Unicon represents a compromise between the underlying C OpenGL code of the virtual machine runtime system, and the flexibility and ease of programming afforded in the higher-level language. Performance can be lost due to dynamic language representations of data, or by the language's hard-wiring various parameters of the OpenGL semantics.

4.2.1 Data representation

3D graphics makes extensive use of integer and double data types. A 3D model for example, might contain tens of thousands of double and integer numbers for vertex data, indices, texture coordinates, animation and more. These kinds of data are usually stored in arrays that get passed to OpenGL for final processing and

rendering. Unicon's list data type is ideal for storage and manipulation of such data at the language level. Unicon lists are not arrays; they are more general and more powerful. A list can store heterogeneous data types, and can grow and shrink. This means that lists have a different representation and implementation than that of the C arrays used by OpenGL. While their representation makes lists very flexible and easy to use, it also means the data stored in a list cannot simply be passed to OpenGL; it has to be converted to an array format first. The underlying implementation of 3D graphics in Unicon was improved as part of this dissertation work to avoid repetitive conversion for the same data from one frame to the next if the data does not change. The conversion is still necessary whenever the data changes for a specific object, which is the case for many animated objects.

4.2.2 Arrays as Lists

Improving performance by buffering or caching the data converted from a list to an array is only a partial solution. Buffering imposes a memory overhead that might be large for rich scenes; also it does not work for any dynamic object in the scene that requires frequent update to its data. A more general solution is needed that does not require more memory and works well for any objects, including those that involve animation. Animation makes a difference because usually animation is done by key framing and applying an animation transformation to vertex data, generating a new set of world vertices that replaces the old set. In other words, the solution should make the same data visible to both the language level and the underlying OpenGL function, bridging the gap between the language interface and the graphics library.

One way to accomplish this is to add new data types to the language to hold arrays of integer and double data. To have the least impact on the language interface, another route was taken. The list data type was extended to support arrays of data by changing its implementation in the language runtime system. This keeps the design in line with the language spirit and respects a major goal, which is not to have any visible additions to the language interface unless it cannot be avoided.

In the new design of lists, for integer and double data types, an array is just another list that happens to have a fixed initial size and also one type of data, either integer or double. Because of the extensive use of lists in Unicon programs, and to avoid any unintended side effects when using arrays, a temporary new constructor function for arrays is added. A "regular" list is created using the list(size, initial value) function. An "array" style list can be created using the new array(size, initial value). The initial value data type (integer or double) dictates the data type of the returned array list. Beyond the creation of a list, whether "regular" or "array", all other operations are the same. The following code fragment creates two lists of size 10 and initializes their elements from 1.0 to 10.0.

```
L := list(10, 0.0)
A := array(10, 0.0)
every i:=1 to 10 do  { A[i] := L[i] := real(i) }
```

Any non-array operation that is applied to an array list forces it to be converted to a regular list. For example applying pop(), push(), get() or put() at the array A in the code above. This is done by the runtime system without the programmer intervention. The implementation of arrays is evaluated in section 7.3.

4.3 Dynamic Textures

Textures are essential to make realistic scenes. 3D graphics libraries usually have support for texture creation, but once a texture is created little can be done afterward to changes it. In many applications, having textures that can be updated at run time opens the door for new interesting uses. Virtual screens or "live" walls inside a virtual environment can display textures that change over time, simulating a computer screen or displaying visualization directly in the virtual world. In CVE, this can be used to implement whiteboards in a classroom. The whiteboard is captured from the instructor side and broadcast to clients, which then update their whiteboards' textures in the virtual classroom. One of the contributions of this dissertation is to add dynamic behavior support to Unicon's textures.

4.3.1 Reloading Textures

Dynamic texture support in Unicon does not create a new special case texture that can be used dynamically; rather it adds the ability to use any texture in a dynamic way. Once the texture is changed; all objects in a 3D scene using that texture will reflect the new changes in the texture as soon as the window is refreshed.

One obvious way to update a texture is to reload the texture with a new image. This new image can be a slightly modified version of the original image used to create the texture in the first place. Repeating this process results in an animated texture. This is the same procedure used to create scenes in an animated movie, applied to textures in a 3D scene in this case. The function Texture(image_filename) creates a new texture. The function returns a record that holds information about the texture. Another function used to load images, ReadImage(image_filename), is extended to support loading new images into a texture. The following code fragment demonstrates the use of these functions:

```
# Create a texture from a jpeg image
Tex := Texture("shot1.jpg")
Refresh()
every i:= 2 to 10 do {
        # wait for a second before updating the texture
        delay(1000)
        # load a new image in the texture "Tex" each iteration
        ReadImage(Tex, "shot" || i || ".jpg")
        Refresh()
        }
```

The function Texture() creates as texture (Tex) out of the image shot1.jpg This code will update the texture Tex nine times (i loops through 2 to 10 values) with shot2.jpg through shot10.jpg images. Of course the above code should be in a context to be meaningful. All of the images should have the exact dimensions if they are meant to replace each other perfectly, but that is not a requirement. Any subsequent image after the one used to create the original texture will be cropped to fit within the original texture size if they happen to be of larger dimensions. Images are all aligned at their origins (0, 0) which is the top left corner of the image. This can be overridden by passing extra parameters to ReadImage() in the form of ReadImage(texture, image, x, y) where x and y represents the pixel coordinates in the target texture where the new image top left corner will be placed so that the new image will extend right and down from that pixel.

4.3.2 Textures as Windows

Using images to update a texture is a useful feature and can be used to implement many functionalities and also produce nice effects. However, the dynamic behavior of textures can be of greater value if it is not limited to the use of images only. One of the uses of CVE requires software visualization to be plotted into a texture in the 3D world. This can be achieved using the technique described in the previous section. An external visualization program can generate the images and feed them to the CVE program, which then can be used to update the texture in the 3D world continuously with the visualization output. While it is doable, it is a slow process requiring a lot of IO operations and involves moving large quantities of data.

A much more efficient approach can be utilized if a texture can be thought of as a window. The visualizations in question are 2D graphics and mainly involve simple commands including drawing points, rectangles, lines, and changing colors. If a texture can be used as a window, visualization commands can be written directly to a texture resulting in a very useful, very fast feature. For this new feature, no new functions, and no major changes to the existing 2D graphics API are needed. 2D graphics output are all sent to the current active window unless the first parameter to these functions is a window (or a texture) itself. In that case the output is send the window referred to by the parameter.

The new dynamic texture feature involves adding support for a texture as a first parameter replacing a window parameter to several 2D graphics functions, this includes:

DrawPoint()
DrawLine()
DrawRectangle()
FillRectangle()
CopyArea()
Clone()

This represents only the minimal set of functions required to support a few example software visualization programs in the CVE. These visualizations programs are part of Unicon distributions and make use of the functions above to display their results. The evaluation of dynamic texturing will be presented in section 7.2, and also in Chapter 9 when talking about CVE.

5 3D User Interaction

Most 3D applications such as collaborative virtual environments and games are interactive in nature. The user clicks on the 3D scene and picks objects. 3D selection plays a main role in some applications so that it is impossible to build such applications, without 3D selection support. Despite its importance, writing 3D selection code is not always easy. In OpenGL for example, it involves a lot of function calls and low level programming.

The Unicon language did not initially support 3D object selection. CVE developers relied heavily on keyboard bindings to do simple tasks. Opening a door in the virtual world for example, was assigned the combination Ctrl+D. But this is only the beginning of the story, the code had to find the nearby doors and decide which one is the closest to apply the action to it, and even that is not enough, since the user might be facing away from the closest door. This also does not allow the user to apply such actions on objects other than the closest one. Among the hundreds of 3D objects in the CVE virtual word, CVE contains tens of doors which had to be selectable. In addition to that, interaction with NPCs and their quests activities depends heavily on being able to click on an NPC. 3D object selection at that point proved to be a must have feature. This triggered the addition of 3D object selection to Unicon which constitutes another major contribution of this dissertation.

The relatively complicated and low-level object selection mechanisms used in OpenGL were inappropriate for a very high level language. Building the 3D selection mechanism into the VM runtime system makes it possible to hide all of the low level semantics from programs. This chapter describes the design, implementation and use of high level 3D selection facilities that were introduced to the Unicon language as part of this research.

5.1 Language interface

The 3D graphics facilities in Unicon hide most of the underlying implementation details yet preserve a lot of powerful 3D functionality. For 3D selection, the same principle is followed; hiding all of the unnecessary details with minimal changes to the language interface. In Unicon, one existing function is extended for selection, plus a keyword and a window attribute are introduced. The use of a keyword for 3D selection goes along the existing design of handling mouse events in the language. When a user clicks a mouse button for example, keywords like &x and &y correspond to the pixel coordinates of the mouse cursor at the time of click. It is natural to follow the same principles and update &pick with the picked object if there is any.

The use of 3D selection in Unicon requires minimal setup and code additions. The new keyword (&pick) provides access to 3D selection results. A new window attribute (pick) enables and disables 3D selection.

The term “pick” was adopted to denote the 3D selection feature to avoid confusion with other uses of “select”. The term “select” is heavily used in different contexts in programming languages and libraries. Clipboard contents, text regions and TCP sockets are examples of such use. The meaning of “pick” also conforms very well with its role in the language which is selecting or picking objects. A few steps are required to use 3D selection in Unicon. These steps are summarized by the following:

- Enable/disable the selection (on/off)
- Give selectable 3D objects unique string names
- Collect selection results for mouse input events through the keyword &pick

The following sub-sections present the new addition to the language interface, and demonstrate the novelty of this simple interface by making 3D object selection accessible to non-expert programmers.

5.1.1 Controlling the selection state

Turning on/off 3D selection is controlled by the Unicon function WAttrib() [88]. WAttrib() is a generic routine for getting or setting a window's attributes. WAttrib("pick=on") turns on 3D selection. WAttrib("pick=off") turns 3D selection off.

WAttrib(“pick”) returns a string value of "on" or "off" depending on the current 3D selection state. By default 3D selection is turned off. The program can turn on and off the 3D selection depending on the program requirements. For better performance it is recommended to turn off selection for non-selectable objects in the scene. This will ensure that 3D selection code in the underlying implementation is skipped, thus no time is wasted processing objects that are not intended to be selectable when doing the special 3D selection internal rendering. Marking an object as selectable is covered in the next subsection.

5.1.2 Naming 3D Objects

3D Objects are defined by their corresponding rendered primitives. The function WSection() [94] marks the beginning and the ending of a display list section that holds a 3D object. A call to the function WSection(s) with a string parameter s marks the beginning of a 3D object with the string s as its name. Another call to WSection() with no parameter marks the end of the 3D object. All of the rendered graphics between a beginning WSection() and its corresponding ending WSection() are parts of the same object. To be selectable, a 3D object must have at least one graphical primitive, such as a line or a sphere. The string name should be unique to distinguish different objects from each other. Different objects could have the same name if the same action would be taken no matter which of these objects is picked. The following code fragment is an example of named 3D objects. It simply draws a red rectangle and gives it the name redrect.

```
WSection("redrect")       # beginning of object
Fg("red")
FillPolygon(0,0,0, 0,1,0, 1,1,0, 1,0,0)
WSection()                # end of the object
```

In the example above, the call WSection("redrect") marks the beginning of a new object with the name redrect. Fg("red") does not affect selection because it does not produce a rendered object. FillPolygon(0,0,0, 0,1,0, 1,1,0, 1,0,0) on the other hand does affect selection because it produces a rendered object, and it actually represents the object named redrect. WSection() marks the end of the object named redrect.

5.1.3 Retrieving Picked Objects

In general, picking objects is associated with the mouse. In Unicon, mouse events are tracked through a set of keywords. &lpress and &rpress for example denote values that indicate that there was a left click or right click event, respectively. The Unicon function Event() produces these events from the event queue. It also generates other information related to such events such as the x and y coordinates of the mouse cursor, as mentioned earlier in this chapter, at the time of the click. The keyword &pick provides 3D selection information in the same fashion on mouse clicks. If selection is enabled, &pick generates all of the string names of the objects under the cursor, one at a time. The following code fragment writes all of the objects' names that were picked by the mouse left-click:

```
every picked_object := &pick do
  write(" picked object :", picked_object)
```

If there were no selectable objects under the cursor at the time of the event, &pick just fails and produces no results. &pick gets its results from both left-clicks and right-clicks.

5.2 Event-Driven Interface for 3D Object Selection

The WSection() function and keyword &pick provide a simple 3D object interface. The programmer gives objects unique names using WSection(), collects selected objects' names by scanning string names generated by &pick, and then takes the appropriate action based on the selected object. For small programs with few selectable objects this is a trivial task, but as the program and its number of selectable objects gets larger, which is the case in CVE, managing selectable objects and the actions to be taken becomes challenging, especially if the objects are hierarchical. While the new 3D object selection interface is already high level and can serve the CVE needs, adopting the co-design philosophy means that the application and the language can both improve to make use of or provide new features. This philosophy was very apparent in this case. CVE demanded a new feature, and the language provided it. Then with new ideas from the use cases of the new feature in CVE, a new simpler interface was built as part of the CVE application. The new

interface was general and powerful enough to be promoted to be part of the class library in the language distribution.

When using selection, from a high level view, the programmer defines a selectable object and assigns an action to be taken when this object is selected. It can be thought of as a graphical user interface (GUI) object, which is an on-screen entity that responds to user events. When a programmer creates a button for example, he does not worry about the button name (except to make it readable and meaningful), and he does not worry about how or when this button was clicked. All the programmer cares about is to take an action if this button is clicked. This section discusses the introduction of a new class to the Unicon language for the purpose of managing 3D selection and adding a high level abstracted layer for using 3D object selection after lesson learned from the CVE development.

5.2.1 Design for the 3D Selection Class

A new class was introduced with the name Selection3D. The class holds information about what objects are selectable, what events should these objects respond to and the action to be taken when an object is selected and receives an event to which it should respond. Any selectable object can respond to one or more mouse events. A separate table for each one of these mouse events (except for CLICK and DRAG, they simply reuse other events' tables) keeps all of the objects that can respond to that particular event. For example there is a table of all objects that can respond to a LEFT_CLICK event if any of these objects were selected. This table is called Tleft_click. Figure 5.1 shows the mapping from all of the mouse events that are recognized by this class to their corresponding tables.

The Selection3D class uses helper class Listener3D to store information about each selectable object: its name (given by the user), the action associated with it, the class object that holds the action if the action is a method, and the event type (specified by the user). Figure 5.2 shows the two classes and their relationship.

CLICK	➔	Tleft_click and Tright_click
LEFT_CLICK	➔	Tleft_click
RIGHT_CLICK	➔	Tright_click
DOUBLE_CLICK	➔	Tdouble_click
DRAG	➔	Tleft_drag and Tright_drag

Figure 5.1 Events that are recognized by the Selection3D class and the mapping between these events and their corresponding object tables.

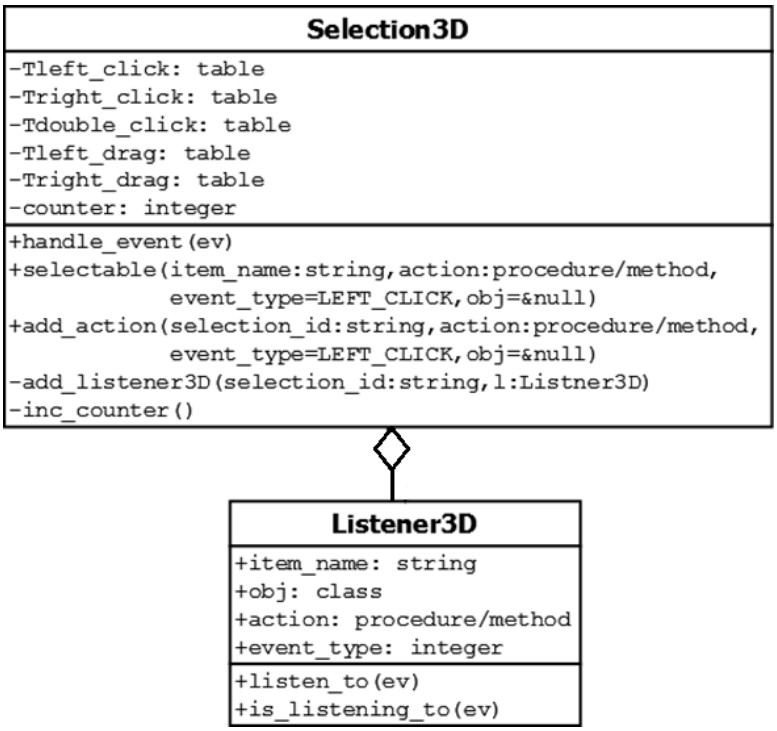

Figure 5.2 UML diagram for the classes used to manage/control 3D object selection

5.2.2 Using the Selection3D class

To make the Selection3D class available for a program in Unicon, the following statement must be added at the beginning of the source code because the class is part of the graphics3d package:

```
import graphics3d
```

The Selection3D class has three methods that can be called to manage the 3D selection in the program. The first method is selectable(), which is used to register new 3D objects to make them selectable. This method takes up to four parameters in the following order:

- A string name of the 3D object. The name does not affect the selection behavior
- A procedure/method name to be called when this 3D object is selected
- An optional event type which could be any of the event types shown in Figure 5.1. The default event type is LEFT_CLICK
- The class object that has the method name (second parameter). This is only valid (and mandatory) if the second parameter is a method which is part of a class object.

The method selectable() returns a string value called a selection id. A selection id is a unique value that can then be passed to WSection() to mark a new 3D object name. The following is an example of such use:

```
select_id := select3D.selectable("red ball", on_red_ball)
WSection(select_id)
```

In this example a 3D object named "red ball" is registered to be selectable. The procedure on_red_ball will be called if this 3D object is selected. The third parameter is omitted which means the 3D object responds to the default event type which is mouse left click. on_red_ball is a procedure (not a method) so nothing should be passed as a fourth parameter. The example also shows how the returned value select_id is passed to WSection().

The second method in Selection3D class is add_action(). This method takes four parameters exactly like selectable() except for the first parameter. Instead of taking an arbitrary user selected string name, it takes a string name that was returned and registered by selectable() to add another action or response to another kind of event. In the example above the following line of code can be added right after the first line to make the red ball respond to right mouse click by calling the procedure on_right_click():

```
select3D.add_action(select_id, on_right_click, select3D.RIGHT_CLICK)
```

The third and final method in the Selection3D class is handle_events(). Normally this method should be part of the event handling loop in the program. At least all types of mouse events in Unicon should be passed to this method. This lets the Selection3D class collect event information and picked objects through &pick and take the appropriate action. Failure to pass any kind of mouse events to the Selection3D class might cause it not to produce the intended behavior. A correct way to use the method handle_events() is shown toward the end of the example in section 7.1.1.

5.3 Implementation details

3D selection in OpenGL requires rendering the scene in a special rendering mode called the GL_SELECT *rendering mode* [38]. In this mode, the scene should be redrawn in the off-screen buffer without swapping it with the front buffer so that this process would be hidden from the user. After setting up the selection environment, objects in the scene must be rendered along with unique integer names generated by OpenGL and assigned to each one of them. This selection mode works by restricting the drawing to a very small area on the screen around the mouse cursor and creating the viewing volume from that area back to the far Z-plane. All of the objects that are rendered in this viewing volume are reported as being selected.

In Unicon, each 3D graphics window has a Unicon list of lists and records to keep track of all objects in a 3D graphics scene. This list is maintained internally in Unicon's window structure when creating a window with "gl" parameter. Each entry in this list contains information about the function name and the parameters

of that function. When a window needs to be redrawn, the window is cleared, all attributes are reset to the defaults, and the Unicon list of lists is traversed to redraw every object in the scene [89]. This implementation is very useful for 3D selection. Unlike the usual use of selection in OpenGL in languages such as C, where the programmer should keep a separate rendering function with selection data for selection rendering mode, in Unicon the same rendering function can be used and the selection code would be executed or skipped dynamically depending on whether the rendering pass is regular rendering mode or selection rendering mode. The key points in the implementation of 3D selection in Unicon include:

- Two variables were introduced to control the execution of the selection code.
    - A state variable to turn on and off the selection in the application
    - A variable to control when to execute the selection code. Even if the selection is on, the selection code should be executed only once after each mouse click (left click or right click do the same)
- A list of string names that are mapped to integer ids (OpenGL names) for selection.
- String names are controlled by `WSection()` calls which also store the integer name mapped to each string name. The string name is used at the source level, and the integer name is used with the underlying library.
- `&pick` generates all of the results from a list of string.

6 Concurrency

Collaborative virtual worlds are very demanding applications. When developing these applications, the need to utilize everything that the hardware has to offer becomes essential to the usability and the success of the software. Both the server and the client sides of these applications can greatly benefit from multi-threaded programming to increase the scalability, reduce the delay and the response time, and increase the overall performance of the application.

In addition to the importance of having multi-threads support when developing virtual environments, adding concurrency to Unicon was also motivated by a collaboration with AT&T Labs Research, where the language has been used to implement a network performability ("performance + reliability") analysis tool called *nperf* [95]. nperf is computationally-intensive, and seriously needed the benefits of parallelization.

This chapter describes a novel design and a major contribution of this dissertation: building true concurrency features into a very high goal-directed interpreted language. The chapter covers major areas of work and extension of the Unicon programming language to support true concurrency. The implementation is built on top of POSIX threads and runs well on UNIX-based operating systems such as Linux, Solaris, and Apple OS X, as well as on the Microsoft platform. While parts of the concurrency features added to Unicon were straightforward adaption of pthreads, parts were planned invention of higher level constructs in order to integrate well with Unicon, and parts were discovered empirically while applying the facilities to real applications and learning from the experience following the codesign approach.

Because Unicon's parent Icon features synchronous, ultra-lightweight threads called co-expressions, the easiest graft of concurrency onto Unicon was to extend that type. Although co-expressions are implemented on top of POSIX threads, because they are not concurrent, they also have a native implementation in assembler code on popular processors that is much faster than the pthreads implementation, on the order of ~100x on modern processors. Unlike the native implementation, the pthreads implementation can be scheduled by the operating system to make use of multiple cores, so it was natural to build Unicon's concurrency facilities on top of the pthread library.

6.1 Language Design

From a language design standpoint, adding concurrency to Unicon consisted of: (1) adding a way to tell co-expressions to execute concurrently instead of waiting to be activated, and (2) extending the activation operator @ and adding new operators to provide a simple mechanism for thread synchronization and communication. These were achieved by introducing a new reserved word thread and extending the @ operator to maintain producer-consumer queues for asynchronous operation in addition to operators for

new forms of synchronization and communication as discussed later in this section. The thread syntax can be used in the following manner:

```
thread write (1 to 10)
```

which launches a thread that writes out to standard output the numbers 1 to 10. The language was also extended with several new built-in functions for high-level access to mutual exclusion primitives and condition variables, in addition to extending a few existing functions to support new semantics. For example, the built-in function wait() was extended to allow a thread argument to support the semantics of "join" in thread programming. Many of the extensions have to do with synchronization. Some problems require using advanced synchronization mechanisms and rely on the language support to achieve full control over the execution of threads and protect shared data. This section covers the synchronization techniques introduced into the Unicon language. It also covers thread communication and the new communication operators.

6.1.1 Critical Regions and Mutexes

A mutex is a synchronization object used to protect shared data and serialize threads in critical regions. For example when two threads compete to increment a variable, the end result might not be what the programmer intended. In such cases a mutex should be used to protect access to the variable. Any operation or data where more than one thread can execute non-deterministically, leading to data corruption or incorrect results is called *thread unsafe*. Thread unsafe code or data structures have to be handled correctly via synchronization mechanisms to achieve correct behavior with correct results.

A mutex object can be created using the mutex() function. The returned mutex object can then be locked/unlocked via the functions lock() and unlock() to serialize execution in a critical region. The following example demonstrates the use of a mutex to protect increments to the variable x:

```
lock(region)
x := x + 1
unlock(region)
```

The mutex object has to be initialized only once and can then be shared between all of the threads accessing the critical region (x := x + 1). lock(region) marks the beginning of the critical region protected by the mutex region. unlock(region) marks the end of the critical region. When a thread calls lock(region), it tries to acquire the mutex region. If region is not "owned" by any other thread, lock(region) succeeds and the thread becomes the owner of the mutex region and then enters the critical region, otherwise the thread blocks until the current owner of the mutex leaves the critical region by calling unlock(region).

The more critical-regions/mutexes a concurrent program has, the slower it runs. The length of the critical region also affects the performance. The longer the critical region, the more time it takes a thread to finish the critical region and release the mutex, which increases the chances other threads get blocked waiting to acquire the mutex and enter the critical region. Locking a mutex and forgetting to unlock it is very likely to lead to a deadlock, a common problem in concurrent programming, where all threads block waiting for each other, and for resources to become available. Because all threads are blocked, resources will not be freed, and the threads block indefinitely.

Unicon provides a special syntax for critical regions which is equivalent to a lock()/unlock() pair, that aims mainly to guarantee that a mutex is released at the end of a critical region, beside enhancing the readability of the program. Here is the syntax:

```
critical mtx: expr
```

This is equivalent to:

```
lock(mtx)
expr
unlock(mtx)
```

The code to increment x in the previous example can be written as:

```
critical region: x := x + 1
```

The critical region syntax only unlocks the mutex if it is executed until the end. If there is a return or break in the critical region expression body, then it is the programmer's responsibility to explicitly unlock the mutex of the critical region. For example:

```
critical region: {
  if x > 100 then { unlock(region); return }
  x := x + 1
  }
```

In some situations, a thread might have several tasks to finish and it is undesirable for it to block waiting for a mutex if it is acquired by another thread. For example, if a thread is creating items that can be inserted in one of several shared queues, the thread can insert every new item in the first queue that it acquires. trylock() is an alternative function for lock(). The only difference is that trylock() is non blocking. If the thread cannot acquire the mutex immediately the function fails. The most suitable way to use trylock() is to combine it with an if statement, where the then body unlocks the mutex after finishing the work on the protected object as follows:

```
if trylock(mtx) then {
    expr
    unlock(mtx)
    }
```

Both lock() and trylock() return a reference to the mutex or the object they acquired (upon succeeding in the case of trylock()). This makes it very convenient to write code like the following, assuming L1 and L2 are both lists that are marked as shared:

```
item := newitem()
if L := trylock(L1 | L2) then {
    put(L, item)
    unlock(L)
    }
```

Note that trylock() may fail to lock any of the lists leaving item unprocessed. Depending on what the code needs to do, if it is required to guarantee that it does not proceed before one of the locks to L1 or L2 succeeds then it can be written as follows:

```
item := newitem()
until L := trylock(L1 | L2)
put(L, item)
unlock(L)
```

Having a loop that is continuously calling trylock() without a delay between successive calls is bad programming practice since it burns CPU cycles. This can be fixed by calling the delay() function in the loop's body. Another solution is to use a condition variable described in subsection 6.1.4, or use an implicit synchronization mechanism such as described in subsection 6.1.5.4.

6.1.2 Initial clause

A procedure in a Unicon program can have an initialization clause at the top of the procedure. This gets executed only once the first time the procedure is entered. The initial clause provides a very convenient way to have local static variables and their initialization in the same procedure instead of relying on global variables and having to initialize them somewhere else. A procedure that produces a sequence of numbers one at each call can be written as:

```
procedure seq()
  static i
  initial i:=0
  i := i+1
  return i
end
```

This initial clause is thread-safe. It can be thought of as a built-in critical region that is run only once. No thread is allowed to enter the procedure if there is a thread currently still executing in the initial block. This can be useful specifically in a concurrent environment to do critical initialization such as creating a new mutex object instead of declaring a mutex variable to be global and initializing it somewhere else or passing it from one function to another where it will be actually used. A concurrent version of seq() would look like this:

```
procedure seq()
  local n
  static i, region
  initial {i:=0; region:=mutex()}
  critical region: n := i := i+1
  return n
end
```

With the use of the initial clause, seq() is self-contained and thread safe. The use of the local variable n to temporarily hold the value of the counter i while still in the critical region. That is because once the thread leaves the critical region, there is no guarantee that the value of i would remain the same before returning the value of i. Using the variable n guarantees that the value returned is correct even if the value of i is changed by another thread.

6.1.3 Thread-safe Data Structures

In Unicon, mutexes do not exist solely as independent objects, they are also used as attributes of other objects, namely attributes of the mutable data types. Any data structure in Unicon that can be used in a thread unsafe manner is better protected by a mutex. Instead of declaring a separate mutex, locking and unlocking it, the structure can just be marked as "needs a mutex/protection" and the language does an implicit locking/unlocking protecting the operations that might affect the integrity of the data structure. For example, if several threads are pushing and popping elements into and out of a list, this creates thread-unsafe operations on the list that require protection.

Using a thread-safe list results in less lines of code compared with doing explicit locking, and also produces a more efficient program doing less locking and unlocking at the language level, or even not doing at all. For example, a list can be protected by passing it to the function mutex() as follows:

```
mutex(L)
```

From this point on, any puts to the list or gets from L are protected by an implicit mutex locking mechanism. This helps make concurrent programming almost as easy as writing a sequential program. The only needed step is to notify the language at the beginning that the data structure is shared, by passing it to the mutex() function. Having to explicitly mark structures as shared ensures that mutexes are only used

where needed, and not wasting time on unnecessary locking/unlocking for structures that are not shared. The locking/unlocking of protected structures slows down structures operations by about 7%. The mutex() function takes a second optional parameter denoting an existing mutex object or an object that is already marked as shared (has a mutex attribute) . Instead of creating a new mutex object for the data structure, the existing mutex is used as an attribute for the data structure. If the second object is a structure that is not marked as shared, a new mutex is created. This is useful when two objects need to be protected by the same mutex. For example, the list L and the table T in the following example share the same mutex:

```
mtx := mutex()
L := mutex([ ], mtx)
T := mutex(table(), mtx)
```

which is equivalent to the following if the mutex does not need to be explicit:

```
L := mutex([ ])
T := mutex(table(0), L)
```

or

```
L := [ ]
T := mutex(table(0), L)
```

In all cases, lock(L) and lock(T) lock the same mutex, serializing execution on both data structures. Not all operations on data structures produce correct results, only “atomic” operations do. In other words, implicit locking/unlocking takes place per operation, which means even if each one of the two operations is safe, the combination might not be. A critical region is still needed to combine the two. For example, if L[1] has the value of 3 and two threads are trying to increment L[1] as the following:

```
L[1] := L[1] + 1
```

the result in L[1] could be 4 or 5. That is because reading L[1] (the right side of the assignment) and storing the result back in L[1] are two different operations (not atomic) and are actually separated in time. The good news is that solving such an issue does not require an extra explicit mutex. If L is marked as shared (passed to the mutex() function) it can be passed to lock()/unlock() functions. It can be used with the critical syntax like the following:

```
critical L: L[1] := L[1] + 1
```

6.1.4 Condition variables

Mutexes protect shared data in critical regions, and block threads if more than one thread tries to enter the critical region. Condition variables provide thread blocking/resumption that is not tied to accessing shared data like a mutex. A condition variable allows a thread to block until an event happens or a condition changes. For example, in a producer/consumer problem, a consumer keeps spinning to get values out of the shared list. In similar situations in real life applications, any spinning could be a waste of resources; other threads including producer threads could be using the resources doing something useful instead. The consumer should block until there is data to process in the list. This is where a condition variable comes into play. A condition variable can be created using the function condvar(). The return object is a condition variable that can be used with wait() and signal() functions. wait(cv) blocks the current thread on the condition variable cv. The thread remains blocked until another thread does a signal(cv), which wakes up one thread blocked on cv. A very important aspect of using a condition variable is that the condition variable is always associated with a mutex. More specifically, the wait() function has to be always protected by a mutex. Unicon provides a built-in mutex for condition variables which can be thought of as an attribute, similar to thread safe data structures. This means a condition variable can also be used with lock()/unlock() functions or the critical clause. It is important to realize that not only wait() has to be protected by a critical region, but also the condition or the test that leads a thread to wait on a condition variable.

The signal() function takes a second optional parameter denoting the number of threads to be woken up. By default, that number is one, but it can be any positive value. From example:

```
every !4 do signal(cv)
```

Can be written as:

```
signal(cv, 4)
```

Furthermore, if all of the threads waiting on cv needs to be woken up, a special 0 (or CV_BROADCAST) value can be passed to signal() causing it to broadcast a wakeup call for all threads waiting on cv:

```
signal(cv, 0)
```

Or

```
signal(cv, CV_ BROADCAST)
```

6.1.5 Thread Communication

Co-expressions, as explained in Chapter 3, have a simple form of communication influenced by their synchronous execution. Threads on the other hand, run in parallel, creating a dynamic environment for communication. In many cases, a running thread must send a value to another thread without waiting for results, or receive a value from another thread if there is one without waiting. The @ operator is not suitable for this kind of (asynchronous) communication. Threads' communication needs to go well beyond co-expressions, for which a single @ operator provided communication bundled with synchronization. Unicon adds four operators for asynchronous communication. These are @>, @>>, <@ and <<@. The operators correspond to send, blocking send, receive and blocking receive.

6.1.5.1 Thread messaging Queues

Thread communication is done through messaging queues that are initialized at thread creation. Each thread maintains two queues called the inbox and outbox that are created with the thread. When a thread sends a message with an explicit destination, the message is queued in the destination's inbox. Otherwise it is queued into the sender's outbox. A thread can receive messages from another thread by dequeuing messages from the source's outbox if there is an explicit source, otherwise it dequeues messages from its own inbox. Figure 6.1 presents two threads with the inboxes and outboxes.

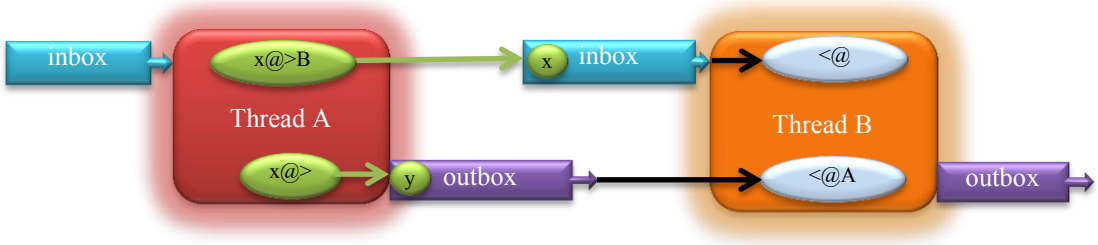

Figure 6.1 Threads messaging queues

6.1.5.2 Send and Receive Operators

The first two operators are @> (send) and <@ (receive). These operators communicate send/receive messages between threads. The semantics support co-expressions as well which also have inboxes and outboxes. It does not matter whether threads or co-expressions are involved in the communication, the semantics are the same. The send operator has the syntax

x@>T

x can be any data type including null, which is equivalent to omitting it. T refers to a thread to which x is transmitted. x is placed in T's inbox. x can be picked by T using the receive operator which is presented later. The send operator can also have no destination such as:

x@>

In this case x is sent to no one, instead it is placed in the sender's outbox. The operator can be read as "produce x". x can then be picked up later by any thread wanting a value from this sender. For example the sender thread in this case might be a thread creating prime numbers and placing them in its outbox to be ready for other worker threads.

The receive operator is symmetric to the send operator and takes two forms, with explicit source or with no source as follows:

<@T

<@

The first case reads: receive a value from T, which gets a value from T's outbox, a value produced by T. Going back to the prime number example mentioned above, <@T would be the way to pick a prime number produced by the prime number generator thread T. <@ on the other hand reads values directly from the receiver's inbox. It reads messages sent directly to the current thread doing the receive.

Both @> and <@ can succeed or fail. In the case of <@ the operator succeeds and returns a value from the corresponding queue (inbox/outbox) depending on the operand if the queue is not empty. If the queue is empty the operation fails directly. In the case of @>, if the value is placed in the corresponding queue, the operation succeeds and returns the size of the queue. If the queue is full, the send operation fails. The inbox/outbox for each thread is initialized to have a limited size (it can hold up to 1024 values by default). This limit can be increased or decreased depending on the application needs. The limits are useful so that queue sizes do not explode quickly by default and also to have an implicit communication/synchronization as explained later in this section.

Each thread has exactly one inbox and one outbox, and each operator call is mapped to only one of these inboxes or outboxes as seen in Figure 1. All messages from all threads coming to thread B in the figure end up in its inbox. All threads trying to receive messages from A compete on its outbox. Both the inbox and the outbox queues are public communications channels, and it is impossible to distinguish the source of a message if there are several threads sending messages to the same thread at the same time. Furthermore, if <@ has an explicit source like A in Figure 6.1 it only looks in A's outbox, and does not see messages from A coming directly to the inbox. Applications that require the sender's address can attach that information to messages by building them as records with two fields, one field for data and the other containing the

sender's address. A better approach for private communications for some applications is the use of lists shared between the two communicating threads or the use of private communication channels discussed later in this chapter.

6.1.5.3 Inbox/Outbox and the Attrib() Function

As seen in previous subsections, communications between threads is done though inbox/outbox queues which are governed by size limits. The size limit (defaults to1024) and the actual size dictate how synchronization is melted with the communication. The size operator * can be used with a thread to query its actual outbox size (how many values it contains, not the maximum limit) and can be used as follows:

```
outbox_size := *T
```

But this is only a single attribute for one queue. To access or change the different attributes, a new function was introduced to Unicon: Attrib(), which uses the form:

```
Attrib(handle, attribcode, value, attribcode, value, ...)
```

The integer codes used by the Attrib() function are supplied by an include file (threadh.icn) with $define symbols for the attributes. This header file is part of a new thread package that is part of the Unicon distribution called package threads. It can be imported to a program via:

```
import threads
```

When values are omitted, Attrib() generally returns attribute values. To get the size of the outbox (similar to what the * operation does in the example presented earlier), the code is:

```
outbox_size := Attrib(T, OUTBOX_SIZE)
```

similarly,

```
inbox_size := Attrib(T, INBOX_SIZE)
```

gets the current size of the inbox. On the other hand,

```
Attrib(T, INBOX_LIMIT, 64, OUTBOX_LIMIT, 32)
```

sets the inbox and outbox size limits to 64 and 32 respectively. Table 6.1 summarizes all of the available attributes and their meanings.

Table 6.1 Thread's communication queues attributes

Attribute	Meaning	Read/Write?
INBOX_SIZE	Number of items in the inbox	Read Only
OUTBOX_SIZE	Number of items in the outbox	Read Only
INBOX_LIMIT	The maximum number of items allowed in the inbox	Read/Write
OUTBOX_LIMIT	The maximum number of items allowed in the outbox	Read/Write

6.1.5.4 Blocking Send and Receive

In many situations, senders and receivers generate and consume messages at different speeds. For example if there is thread that generates prime numbers very fast to be used by other slow threads, the prime number generator thread should not continuously generate prime numbers, but create a batch and wait until it is all used before generating another batch. Instead of overloading slow receivers or busy waiting for slow senders, the two ends of communication require a synchronizing mechanism to tell when to send new messages or when a new message is available. Two more send and receive operators provide such functionality, these are the blocking send operator @>> and the blocking receive operator <<@. These operators can be used in the same way @> and <@ are used, except that instead of failing when the operation cannot be completed, the new operators block and wait until the operation succeeds. This is useful in situations where a value is required to proceed, allowing the thread to block instead of spinning waiting for a value.

In some cases, a thread might want to use a blocking receive to get values from a second thread, but it is not willing to block indefinitely; probably the thread can do some other useful work instead of waiting. The <<@ accepts a timeout parameter to impose a limit on how long to wait for a result before giving up. Here is how <<@ would look like in this case:

```
result := timeout <<@              #  get from the current thread inbox
```

or

```
result := timeout <<@ T            # get from T's outbox
```

The timeout operand is a non-negative integer denoting the maximum time to wait in milliseconds. Negative integers are treated as a null value, an indefinite blocking receive. A 0 operand indicates no waiting time, making the end behavior effectively similar to a non-blocking receive. Table 6.2 summarizes the different forms of the send and receive operators and their operands:

Table 6.2 Summary of the new communication operators

Operator	Operands	Behavior
@> **(send)**	msg@>	Place msg in the current thread's outbox, fail if the outbox is full
	msg@>T	Place msg in T's inbox, fail if T's inbox is full
<@ **(receive)**	<@	Get a message from the current thread's inbox, fail if it is empty
	<@T	Get a message from T's outbox, fail if T's outbox is empty
@>> **(blocking send)**	msg@>>	Place msg in the current thread's outbox, block if the outbox is full
	msg@>>T	Place msg in T's inbox, block if the T's inbox is full
<<@ **(blocking receive)**	<<@	Get a message from the current thread's inbox, block if it is empty
	<<@T	Get a message from T's outbox, block if T's outbox is empty
	n<<@	Get a message from the current thread's inbox, block up to n milliseconds if the inbox is empty waiting for new messages
	n<<@T	Get a message from T's outbox, block up to n milliseconds if T's outbox is empty waiting for new messages to become available

Most applications do not need to use all of these options to operate, using a subset is usually sufficient. In a fast sender slow receiver application for example, the sender would block when the queue is full and unblock when the queue is empty (using @>>). The receiver on the other hand would keep consuming messages from the queue until it is empty. In that case it blocks until there is a new message added to the queue (<<@). For some applications however, this communication scheme might not be optimal, hence many options are provided. The different options in the table above give the programmer a wide range of control over when to block or resume a thread based on the availability of data in the communication queues. This control covers many applications' needs, and provides simple ways to abstract some concurrent programming activities such as load balancing and efficient use of resources.

6.1.5.5 Private Communication Channels

As mentioned in the previous sections, inbox and outbox communication queues are visible by all threads all the time. In some scenarios two or more threads need to communicate with each other without worrying about other threads sending and receiving messages at the same shared queues. While it is possible to build a protocol at the application level on top of the inbox and outbox queues to achieve such behavior, it is simpler and more efficient to have the threads communicate privately. This kind of communication can be done by sharing a list between two threads and protecting it by an explicit mutex, or using a thread-safe list. A more formal way for such communication is to use the channel() function.

Starting a private communication is similar to a network connection, except that this connection is taking place between two threads in the same process instead of two different processes that may be on different machines. A private communication channel between two threads and can be created using the function channel(). channel() is part of the threads package (which was added to Unicon's class library) and takes one parameter, which is the thread with which the connection will be initiated. If channel() succeeds, it returns a list representing a communication channel between the two threads. Representing a bidirectional channel that can be used by the two threads, given that each thread calls the function channel() with the other thread as an argument. Here is an example.

In thread A:

```
chB:= channel(B)  | stop("failed to open a channel with B")
```

In thread B:

```
chA:= channel(A)  | stop("failed to open a channel with A")
```

This channel is meant to be a directional communication channel. In theory one thread would use it as an outbox, and the other would use it as an inbox. That means only one thread will send messages over the channel while the other receive them from the other end. For many problems, one thread produces results while one or more threads consume these results making the directional nature of the channel suitable for such problems. In cases where the communication is needed in both directions, the two communicating threads can open two channels to send messages in both directions. The provided channels can be used with the communication operators (all four of them) with the same semantics as before. The only difference in this case is that the right operand is a communication channel instead of a thread. The channel() function also accepts a second parameter as an identifier for the channel() so that the two ends of the channel find each other easily without depending on the order the channel() function is called.

Up to this point, this chapter covered mostly language extensions and features visible at the source level. The remainder of this chapter focuses on the underlying implementation of concurrency in the Unicon language, presenting lessons learned from the implementation of concurrency within the legacy virtual machine interpreter, known as *iconx*.

6.2 Virtual Machine Enhancements

The Unicon virtual machine is the Icon virtual machine, extended to better accommodate object-oriented programming. The extensions made to support true concurrency would apply equally to the Icon implementation. Other similar languages face analogous situations, where the lessons learned here may be applicable. Two main issues had to be addressed in order to allow concurrent execution of multiple threads in the VM. First, the VM's state had to be replicated per thread, and second, self-modifying instructions

and the *initial clause* had to be protected against race conditions. The two issues are discussed in the following two subsections.

6.2.1 VM Registers

Thread-local storage is utilized to provide each thread with its own copy of the virtual machine "registers", which were formerly global variables in the VM C code. These include the program counter, stack pointer, the several frame pointers necessary to manage calls, returns, suspension and resumption on the stack, and other pieces of virtual machine state. To add concurrency, these elements of state are moved into a struct threadstate that is allocated for each thread, and keeps all thread-specific VM state information. The threadstate is fairly large, under the assumption that memory is cheap whereas thread synchronization is expensive. POSIX threads API calls are used to obtain references to the threadstate structure where it is needed.

A key concern when using thread-local storage is performance. Initially, all references to VM registers were replaced, via macros, by references through a global threadstate pointer variable declared with the compiler storage specifier __thread that is available in some compilers such as gcc and Sun's (now Oracle's) cc. This implementation is straightforward but incurs a substantial performance cost. In addition, the __thread keyword is not currently available in the pthreads implementation on OS X and that of Mingw gcc on Microsoft Windows. Switching to the more portable pthreads API for thread-local storage using pthread_getspecific() allows a faster implementation than __thread. The straightforward implementation invoked pthread_getspecific() once at the top of each C function containing references to VM register and state variables. The many calls to this API wherever the VM state is referenced in the runtime system entail an insignificant performance cost. It is evaluated as part of the overall thread performance in section 7.4.

6.2.2 Self-Modifying Instructions

The Unicon virtual machine has seven instructions that contain integer offsets as operands. These offsets refer to memory addresses within the instruction region, or to static data regions of the bytecode. They are used for several types of literals, globals, static variables, and also for instructions that jump to addresses, such as Op_Goto. For performance reasons, when the opcode is first encountered, offsets are converted to absolute pointers. The opcode is modified to indicate that the operand is now a pointer, and the instruction proceeds at full speed on subsequent executions.

This implementation introduces a race condition when multiple threads execute the self-modifying instruction at the same time. The problem was solved by adding a mutex for each self-modifying instruction opcode. Figure 6.2 shows an example self-modifying instruction code protected with a mutex.

Mutex contention could be reduced further by allocating a separate mutex for each instance of each opcode (requiring a number of mutexes proportional to the size of the program) instead of using just seven mutexes for the seven self-modifying opcodes; it is unlikely that this would be worth implementing, since the number of instances is large and each instance of a self-modifying instruction uses its mutex only once when it replaces itself.

To provide a thread safe *initial clause* (discussed in 6.1.2) at the language level, a similar technique used for self-modifying instructions was utilized, unlike self-modifying instructions, where locking and unlocking take place at the same instructions, with *initial clause*, the locking happens at the beginning of the *clause*, and the unlocking happens at the end. To allow such behavior, a new VM instruction was introduced which takes care of updating the instruction at the beginning of the *clause*, and unlocks the mutex. The updated instruction causes all of the subsequent calls to the function containing the *initial clause* made by any thread to skip over the *clause*. This mechanism effectively eliminates the need to do any locking/unlocking after it is initialized.

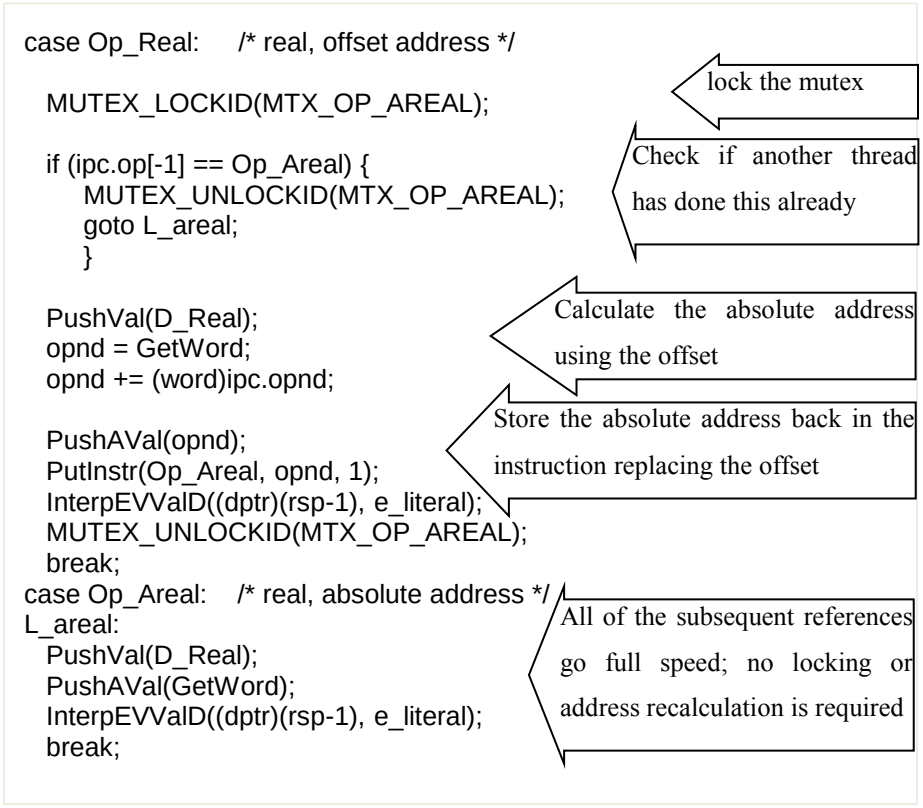

Figure 6.2 Self modifying instruction protected by a mutex

6.3 Runtime System Support

Achieving thread-safety in the implementation without a global interpreter lock required extensive modifications to the runtime system to provide thread-safe environment. This includes safe use of the IO subsystem and the underlying libraries, but most importantly heap management in term of allocation and garbage collection.

6.3.1 Input/Output Sub-system

The Unicon input output sub-system includes very high-level facilities for accessing files, networks and messaging, 2D and 3D graphics, databases, pipes and pseudo-ttys. The underlying C library functions implementing these capabilities are often thread- safe [96]. However, Unicon language-level IO operations usually involve several underlying library calls that must be atomic with respect to threads. Each IO handle is assigned a mutex when it is opened. Any IO operation that uses this handle locks the mutex to guarantee the atomicity of such operations. The cost incurred by such locks is evaluated in section 7.4.

The atomicity of IO is guaranteed only within a single Unicon-level IO operation. For example write(x, y) is atomic and no other thread can write to the same file while write() is running. If multiple writes or reads are required to be made atomic with respect to other threads, an explicit lock is needed. In such a case, the IO handle can be passed to the lock()/unlock() functions to protect it from any access other than from the current thread during the execution of several IO operations.

6.3.2 C Library Thread Safety

In addition to IO, many C library functions called in the Unicon VM were implemented before threads were widely used in operating systems. A subset of those functions is documented in the POSIX standards as thread-unsafe. About 40 function of those are used in the runtime system. Most calls to these thread-unsafe functions were replaced with the thread-safe alternatives available in modern C implementations. A few, infrequently called, thread-unsafe functions were protected by mutexes. A couple of the thread-safe alternatives were not portable; new wrapper functions or implementations were developed in such cases.

6.3.3 Allocation and Separate Heaps

Allocation is a frequent operation for which speed is a top priority in the Unicon virtual machine. Unicon programs start with a heap consisting of one *string* and one *block* (structure) region, and allocate more regions as needed. At any moment, the string and block regions used for allocation are together referred to as the *current heap*. If the memory request is bigger than what is available in the current heap, other regions

are checked. If enough memory is found, the current heap is switched to use the satisfying region; otherwise a new region is allocated.

With multiple threads, safety becomes a concern if only one shared heap is available. When a thread allocates memory in the current heap and updates the heap's state variables, the operation must be protected by a mutex. Otherwise different threads would corrupt each other's allocation requests, leaving the heap in an invalid state. A similar situation occurs when some operations in the runtime system deallocate memory after allocating it for a temporary use.

After determining that the locks required in the shared heap strategy were significantly slowing down the frequent task of memory allocation, the decision was made to use separate, per-thread heaps. This allows threads to use their own *private* heaps at full speed, just like a single-threaded application. Heaps that are not owned by any thread are referred to as *public* heaps. Public heaps are a product of threads out-growing their private heaps. This happens when a thread requests more memory and that request can only be granted by allocating a new private heap. A program starts with no public heaps, and with one private heap for the main thread. A new private heap is allocated for each new thread if none of the public heap is sufficient. Figure 6.3 shows the heap layout with regards to threads. when a thread finishes, its heap goes back to the pool of public heaps.

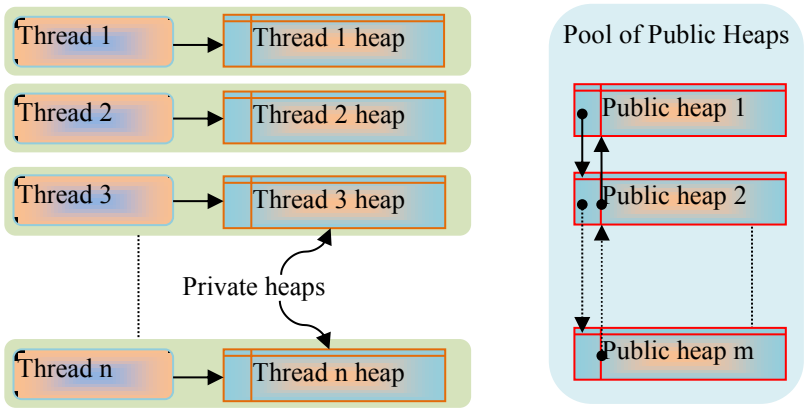

Figure 6.3 Threads, private heaps and public heaps

All heaps (public and private) are visible to all threads, so any thread can access variables and data structures in any heap at any time. The accesses can be controlled or protected at the application level, if needed, in the case of a shared variable. The terms "private" and "public" heaps here are limited to allocating memory in the heaps, and are not to be confused with memory access to heaps in general. In other words, *a thread can allocate memory only in its private heap, but can access (read and write) variables in all heaps.*

Having a full private heap does not mean a thread will automatically allocate a new fresh heap. When a memory request cannot be granted in the private heap, public heaps are checked to see if any of them has enough free memory to grant the request. If so, the private heap is swapped with the public heap, changing the private heap to public and vice-versa. The pool of public heaps is protected by a mutex. If public heaps do not have enough free memory, a garbage collection will take place. If the memory request cannot be granted even after that, the thread allocates a fresh heap.

6.3.4 Garbage Collection

The frequency of garbage collection depends on heap size and application memory usage patterns. Historically, many Icon applications ran to completion without ever garbage collecting. However, modern object oriented event-driven applications run for longer periods of time, and garbage collect proportionally often, despite increases in physical memory and heap size. When garbage collection does occur, it concerns every thread.

Since garbage collection does not happen often (examples are given in Chapter 9), the design philosophy is to keep garbage collection as simple as possible under concurrent execution. To allow safe access to data in the heaps, garbage collection suspends all running threads except the thread which triggered it, referred to as the *GC thread*. The GC thread performs a conventional garbage collection, during which the program runs sequentially.

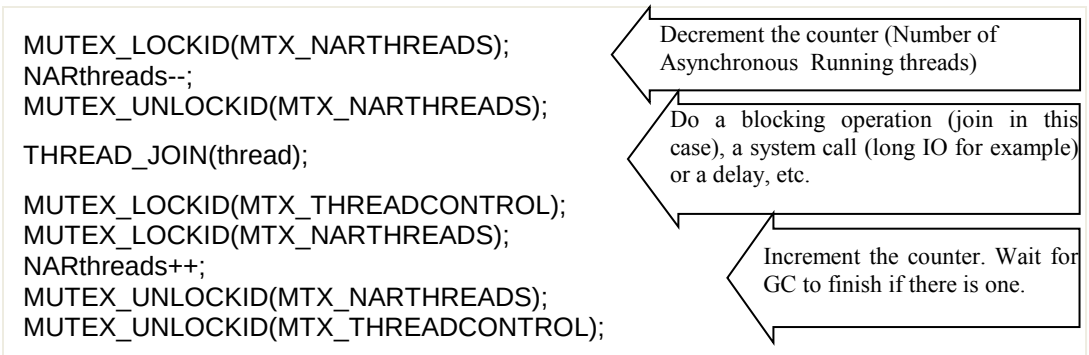

Figure 6.4 Tracking the number of running threads in the system

The GC thread first locks the garbage collection mutex MTX_THREADCONTROL, and then sets a global flag that announces the need to perform garbage collection. It directs all running threads to converge to a special routine called thread_control, the coordination point for thread suspension and resumption. The global flag is checked by all threads once per virtual machine instruction. The runtime system keeps track of how many threads are running using a global counter NARthreads, which is incremented and decremented by the threads depending on their state: before a thread blocks on a mutex it decrements the

counter, and increments it after unblocking. The counter itself is protected by a mutex and can be incremented only if there is no garbage collection request pending or taking place (Figure 6.4).

After a call for garbage collection causes a thread to enter the thread_control function, the thread decrements the running threads counter NARthreads and goes to sleep on the gc condition variable. The GC thread keeps testing the running threads counter until its value decreases to 1, meaning that the GC thread is the only thread running. Then the GC thread proceeds to perform the garbage collection. After finishing, the GC thread unlocks the garbage collection mutex, and does a broadcast to the gc condition variable. This wakes up all of the threads blocked on the gc variable, allowing them to resume their execution and return to where they called the thread_control function in the first place.

In garbage collection-intensive applications, it may happen that two (or more) threads trigger a garbage collection at the same moment. This situation can be handled in various ways. The simplest solution is to block all of the threads requesting garbage collection after the first one and let the first one proceed and finish as described above (block all of the other threads and then wake them up again). A better solution, the one adopted in this design, is to make the threads that are competing for garbage collection aware of each other. Instead of a sequence of "block all others" followed by a "wake up others" performed by each GC thread, a GC thread can hand the control over to the next GC thread in line, if there is one, and then go to sleep. Only the last GC thread in the line wakes up all of the blocked threads. This strategy eliminates intermediate block/wakeup operations, leaving only one "block others" operation by the first GC thread and one "wake up others" operation by the last GC thread. A counting semaphore (gc semaphore) controls the queuing and synchronizing of several threads to garbage collect. The first thread to request a garbage collection does not block on the semaphore, but subsequent requesters do. After finishing garbage collection, each thread signals the semaphore to wake up the next GC thread and goes to sleep. The last GC thread to wake up is responsible for waking up all of the threads after finishing.

An additional measure increases the effectiveness of the strategy just described: triggering an artificial garbage collection if another thread has a garbage collection request pending. After a thread triggers a garbage collection and starts the protocol to suspend all other threads, each thread answers the call, but before going to sleep on the gc condition variable it checks if its heap is "nearly" full. The "nearly full" value depends on the heap size, the number of threads, and the nature of the application. Tests indicate that generally, a value in the range 90%-95% is reasonable for most applications (garbage collection example statistics are given in Table 9.1). Thus if the thread's heap has only 5%-10% of its total size free, the thread queues up to do garbage collection instead of going to sleep on the gc condition variable. This technique forces several garbage collections that might be separated by very short periods of time to cluster together, requiring only a *single* suspension for all of the threads to do the several garbage collections. Figure 6.5 shows an abstract overview of garbage collection and concurrency control.

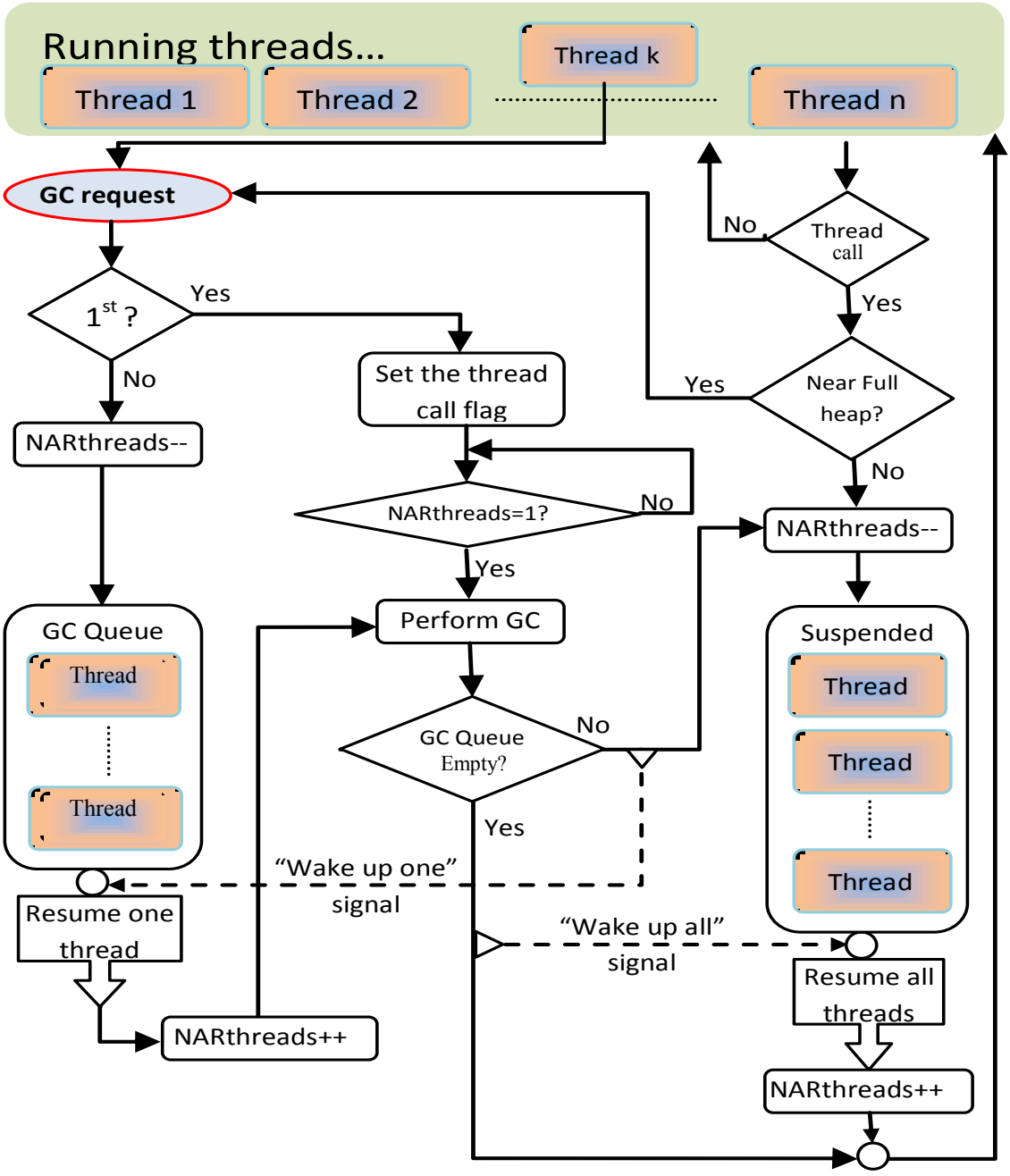

Figure 6.5 A high level view of the dynamics of suspending/resuming threads for GC

PART III Evaluation

7 Programming Examples and Benchmarks

This is the first of the three chapters dedicated to test the new language features and evaluate the contribution of this dissertation. This chapter presents a set of relatively short Unicon programs to evaluate the language additions that were created for this dissertation in a standalone context. Although some of the evaluation such as performance is quantitative, many of these tests or demonstrations allow qualitative evaluation of aspects such as ease of programming interface, or number of lines needed to accomplish a task. The next two chapters focus on evaluating the combination of these features in larger software systems, NPCs and the CVE.

7.1 Using 3D Object Selection

This section demonstrates the two methods introduced by this this dissertation, by which 3D object selection can be used in Unicon. The first method relies on using the 3D selection language interface directly. The second method relies on a standard class library to provide an event-driven GUI-like interface.

7.1.1 Language Interface

This section presents a simple full example program, that demonstrates the use of the 3D selection mechanism in Unicon. Three spheres, red green and blue, are drawn in a 3D graphics window. The red and blue spheres are selectable but the green is not. The user can click on any place in the window and the program reports the picked object to the user. If the user clicks on the red or blue sphere they will get the message “you picked the red ball” or “you picked the blue ball”. If the user clicked anywhere else including on the green ball they will get the message “you picked nothing”. That is because the selection is off for the green ball so it is not selectable.

```
procedure main()
# open a 3D window and make it the default
&window := open("3D selection in Unicon",   "gl","size=500,500")
# begin a new selectable section/object
WAttrib("pick=on")        #turn on 3D selection
WSection("red ball")
  Fg("red")
  DrawSphere(1, 0.5, 0, 0.5)
WSection() # end of the red ball

# Draw a nonselectable green ball
WAttrib("pick=off")       #turn off 3D selection
Fg("green")
DrawSphere(-1, 0.5, 0, 0.5)
```

```
# begin a new selectable section/object
WAttrib("pick=on")          #turn on 3D selection
WSection("blue ball")
   Fg("blue")
   DrawSphere(0, -0.5, 0, 0.5)
WSection() # end of the blue ball

#setup the eye to look at the spheres
Eye(0,0,4, 0,0,0, 0,1,0)
Refresh()

# enter an event loop to handle user events
repeat{
 case Event() of {
  &lpress | &rpress : write("you picked : ", (“the ” || &pick) | "nothing" )
   }
 }
end
```

Making an object selectable involves only one function: WSection(), which takes a string name to be used as an identifier that can be easily distinguished when collected by &pick. This simple API allows 3D selection code to be minimal and readable to help write compact programs. Figure 7.1 demonstrates the reduction in the amount of code needed to setup a selectable object in Unicon compared with C using OpenGL. Nothing prevents a C library from providing the same abstraction as WSection(), but the subsequent input handling and identification of 3D objects associated with event() and &pick provide a compact very high level interface.

7.1.1 Event-Driven Interface

The following program demonstrates the high-level event-driven interface for 3D object selection in Unicon. This is provided through a class library that is part of the Unicon distribution. The programmer writes event handlers and assigns them to objects in the scene. This is very similar to the mechanism employed by GUI applications. Note that it builds upon, rather than replaces the language support for 3D selection.

C/OpenGL Code:

```
glSelectBuffer(size, buffer); /* initialize */
glRenderMode(GL_SELECT);
glInitNames();
glPushName(-1);
```

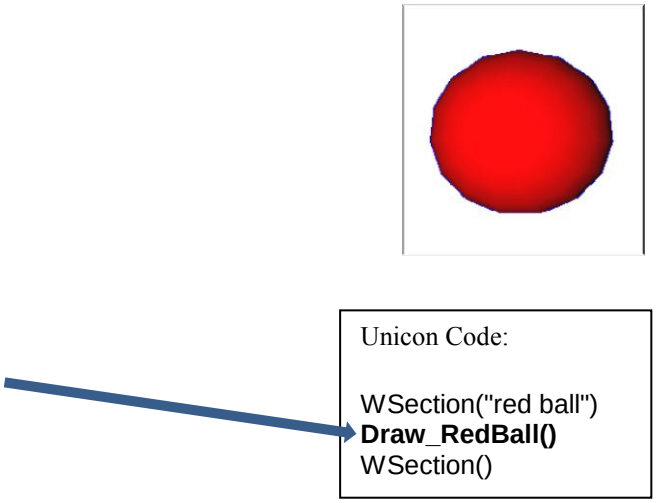

Figure 7.1 Only one function is required to make an object selectable in Unicon, compared to many functions with C/OpenGL

```
import graphics3d
global select3D

procedure on_red_ball()
  write(" You picked the red ball!")
end

procedure on_blue_ball()
  write("You picked the blue ball")
end

procedure main()
&window := open("3D selection in Unicon", "gl","size=500,500")
select3D := Selection3D()

# begin a new selectable section/object
WAttrib("pick=on")  #turn on 3D selection
select_id := select3D.selectable("red ball", on_red_ball)

WSection(select_id)
  Fg("red")
  DrawSphere(1, 0.5, 0, 0.5)
WSection() # end of the red ball

# Draw a nonselectable green ball
WAttrib("pick=off")  #turn off 3D selection
Fg("green")
```

```
DrawSphere(-1, 0.5, 0, 0.5)

# begin a new selectable section/object   WAttrib("pick=on")  #turn on 3D selection
select_id := select3D.selectable("blue ball", on_blue_ball)

WSection(select_id)
  Fg("blue")
  DrawSphere(0, -0.5, 0, 0.5)
# End of the blue ball
#setup the eye to look at the spheres
Eye(0,0,4, 0,0,0, 0,1,0)
Refresh()
# enter an event loop to handle user events
repeat{
  if (ev := Event()) then
    select3D.handle_event(ev)
 }
end
```

The event-driven interface for object selection frees the user from having to worry about names or to which object the name belongs when the time comes to collect selected objects. This is more important particularly in a large project such as CVE. The event-driven interface allows for a modular programming style or having each selectable object have an event handler as a method as part of its class.

7.2 Using Dynamic Textures

Dynamic texturing allows the program to update its textures in real time. This can be used to create effects needed in the virtual world that cannot be achieved otherwise, such as having a virtual computer screen. This section presents examples that demonstrate the use of this feature.

7.2.1 Reloading Textures

One way to use dynamic textures is to reload a texture with another image. The new image does not have to have the same size or format as the original image. The following program renders a scene with a cube textured from all sides with the same texture. The program pauses for a key press or a mouse click before overwriting the center portion of the texture on the cube with a small image. The result is shown in Figure 7.2. Once the texture is updated, all sides of the cube reflect the new change.

```
procedure main(argv)
   &window := open("dyn tex", "gl", "size=512,256", "bg=blue)
   WAttrib("texmode=on")
   PushTranslate(0,0,3)
   Rotate(20, 1, 0, 0)
   Rotate(30, 0, 1, 0)
   Scale( 2,1,2)
   tex := Texture("grapes.jpg")
```

```
   cube := DrawCube(0,0,0, 2.5)
   Eye(0,-1.0,13, 0,-0.25,0, 0,1,0)
   Event()
   # update the texture with a new image
   ReadImage(tex, "setupbig.png", 174, 46 )
   Refresh()
   Event()
end
```

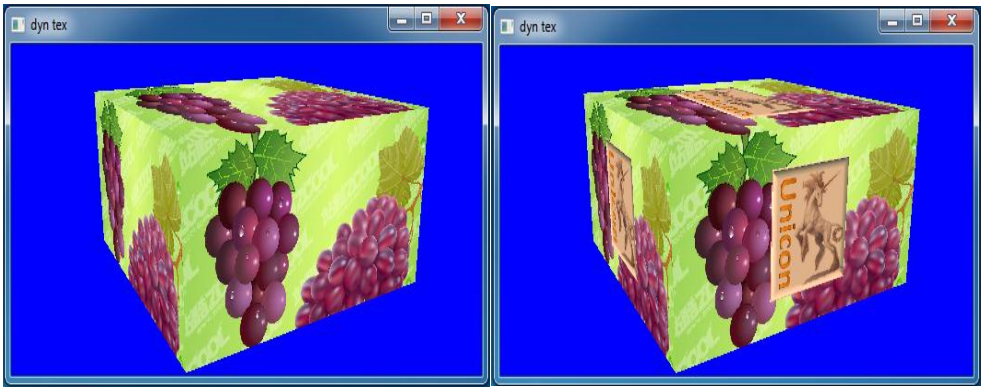

Figure 7.2 Left: the original texture shown on three sides of the cube.
Right: The texture after getting updated with another image

The new image does not have to be an image stored on disk, it can be captured on the fly from a window. The function CopyArea() is extended to support this feature. Instead of copying an area from window to window the function can do the copying directly into a texture in the following manner:

```
CopyArea(win, tex, x,  y, w, h, new_x, new_y)
```

In this case, the function copies an area from the window win to the texture tex. The area copied from win starting at x, y coordinates extending w width and h height. The copied area is placed at new_x, new_y coordinate in tex. if new_x and new_y are omitted, their values default to x and y.

7.2.2 Dynamic Textures as Windows

The other use scenario for dynamic textures is to use them as windows, accepting drawing commands directly. The texture in this case is similar to a window's canvas, where lines, rectangles and other 2D primitives can be rendered to be part of the texture. Instead of passing a window as a first argument to these drawing functions, a texture variable is passed, diverting the drawing to the texture itself. The following example demonstrates such use, and Figure 7.3 presents the screenshots of the program output.

```
procedure main(argv)
   &window := open("dyn tex", "gl", "size=512,256", "bg=blue")
```

```
   PushTranslate(0,0,3)
   Rotate(20, 1, 0, 0)
   Rotate(25, 0, 1, 0)
   Scale( 2,1,2)
   WAttrib("texmode=on" )
   tex := Texture("grapes.jpg")
   cube := DrawCube(0,0,0, 2.5)
   Eye(0,-1.0,11, 0,-0.25,0, 0,1,0)
   Event() # pause waiting for an event
   Fg("yellow")
   # Draw 5 rectangles
   FillRectangle(tex,  2, 2, 508, 10,
                       2, 245, 508,10,
                       2, 2, 10, 250,
                       498, 2, 10, 250,
                       250, 40, 250 , 70)
   Fg("red")
   every i:=!30 do
         DrawLine(tex, i*8+250,50, i*8+250, 100)

   every i:=!20 do
         if i%2=0 then{
            Fg("black")
            DrawRectangle(tex,i^2+i*10,i^2, 25,25)
            }
         else{
            Fg("green")
            FillRectangle(tex,i^2+i*10,i^2, 25,25)
            }

   Refresh()
   Event()
end
```

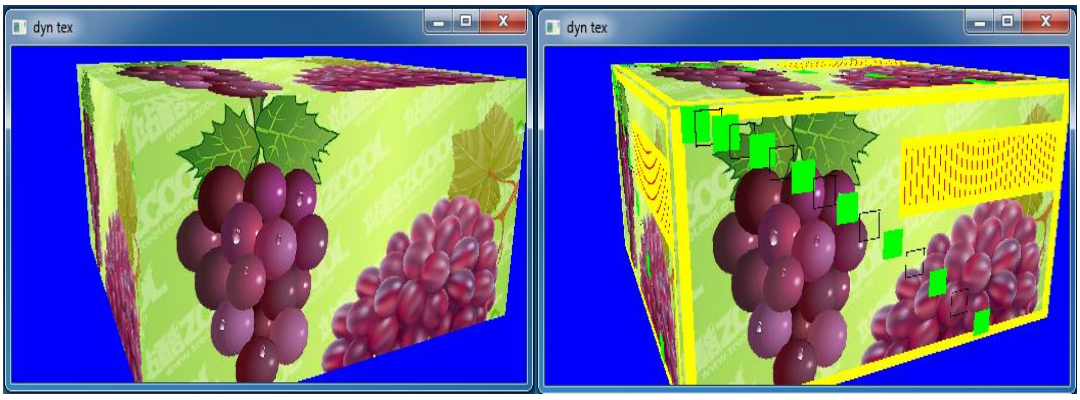

Figure 7.3 Left: Original texture. Right: the result of drawing on the texture dynamically

When using a dynamic texture as window, the difference between the two is blurred at the code level. As shown in the code listing above, the programmers write the code as if the texture is a real window, and the

result is reflected on the texture wherever it is used in the scene as illustrated in the figure. This transparency between textures and windows means that programmers do not have to learn a new API, they just apply what they know about windows to textures. It also means that code written to work with windows or textures, requires almost no modifications except changing a variable initialization from a window to a texture or vice versa to have the code work one way or another. A piece of code that needs to support both use cases at the same time need not to be modified or replicated, moving such code to a separate procedure and passing the right parameter is sufficient, further reducing the amount of code needed.

7.3 Lists as Arrays versus Traditional Lists

Supporting the array representation for Unicon lists not only benefits programs that use such lists directly, but also improve the performance of some programs that use lists of these types under the hood. For example, much of the data stored in the display list of a 3D window is now stored as arrays instead of lists. The new representations not only perform better, but also use less memory compared to using regular lists to store the same data. The following sub-sections demonstrate the use of the new list representation in the 3D graphics facilities in Unicon. Results are also presented comparing the performance and memory requirement of traditional lists and the new list representation. The main focus of the discussion is performance gains (mainly frames per second - FPS). However, since the new representation uses a lot less memory, memory usage is covered for completeness and to give an idea how much arrays representation reduce memory usage. Memory requirement difference is covered in the next subsection. The evaluation of arrays focuses on their impact on virtual environment performance.

7.3.1 Arrays and Memory requirement

Lists not only have variable size, they also can have heterogeneous data types. This makes them very flexible and convenient to use for many kinds of problems. In a 3D graphics pipeline, however, a huge amount of data, which is usually fixed in size and data type must be compactly pushed through the pipeline as fast as possible. As discussed earlier, lists have to be converted to arrays so that they can be processed by underlying libraries such as OpenGL. With the new array representation of lists, this conversion is avoided.

Data types in Unicon are represented internally using *descriptors*; [22] provides a detailed description on data types and their representation in Unicon. A *descriptor* is a combination of two memory *words* also known as *double words*. One word stores data type information, while the second word stores the actual data or a pointer to the data if the data does not fit in one word, such as a list. For the new list representation, the amount of memory saving when using arrays is platform dependent. Typically, storing an integer in a list in Unicon incurs a 1x memory overhead (an integer can be stored in one word, thus

requires one descriptor to represent), while storing a real number incurs a 3x memory overhead. Ignoring the overhead the list structure incurs when aggregating integer or real numbers, this translates to a huge amount of memory when working with a huge amount of data. For example, on a 64-bit system a list of 1024 integer elements would take (ignoring the list overhead):

1024 * (8 * 2) = 16KB

8 is the number of bytes in a *word*, 2 is the number for words in a *descriptor*. If the list is used to store real numbers the size is:

1024 * (8 * 4) = 32KB

When using the array representation for list, the size of both lists would be the same:

1024 * 8 = 8KB

On a side note, a new implementation of real numbers in Unicon introduced in summer 2012 on 64-bit platforms cuts the memory requirement of the real numbers by half. This brings the memory requirements for lists of real numbers in line with lists of integers. This is not true on 32-bit platforms. The memory requirements of real numbers presented throughout this section assumes the old representation of real data type. As mentioned earlier, the emphasis is on performance, not memory usage.

7.3.2 Arrays and Terrain Rendering

Terrain generation and rendering is essential for most virtual worlds. Since a lot of scenes include terrain, it is very important to render them efficiently. A simple way to create terrains is using procedural algorithms that generate height maps (y values) that are interpreted to be the height above the x-z flat plane, or below it representing valleys or sea water. These height maps are usually previewed as grayscale bitmap images. Figure 7.4 is an example of using this technique to create terrain in Second Life virtual world.

Such terrain can be generated on the fly when loading a game's virtual world for example, or before loading a specific scene. They can be also generated and stored in advance to customize them if needed. Figure 7.5 is an example rendered terrain in Unicon loaded from a height map file.

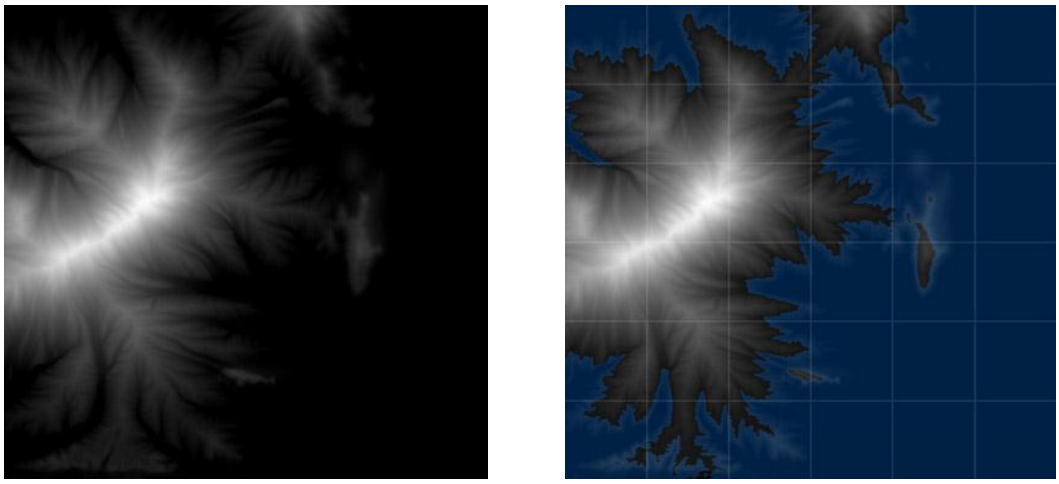

Figure 7.4 Terrain generation and rendering in Second Life using height maps [secondlife.com]

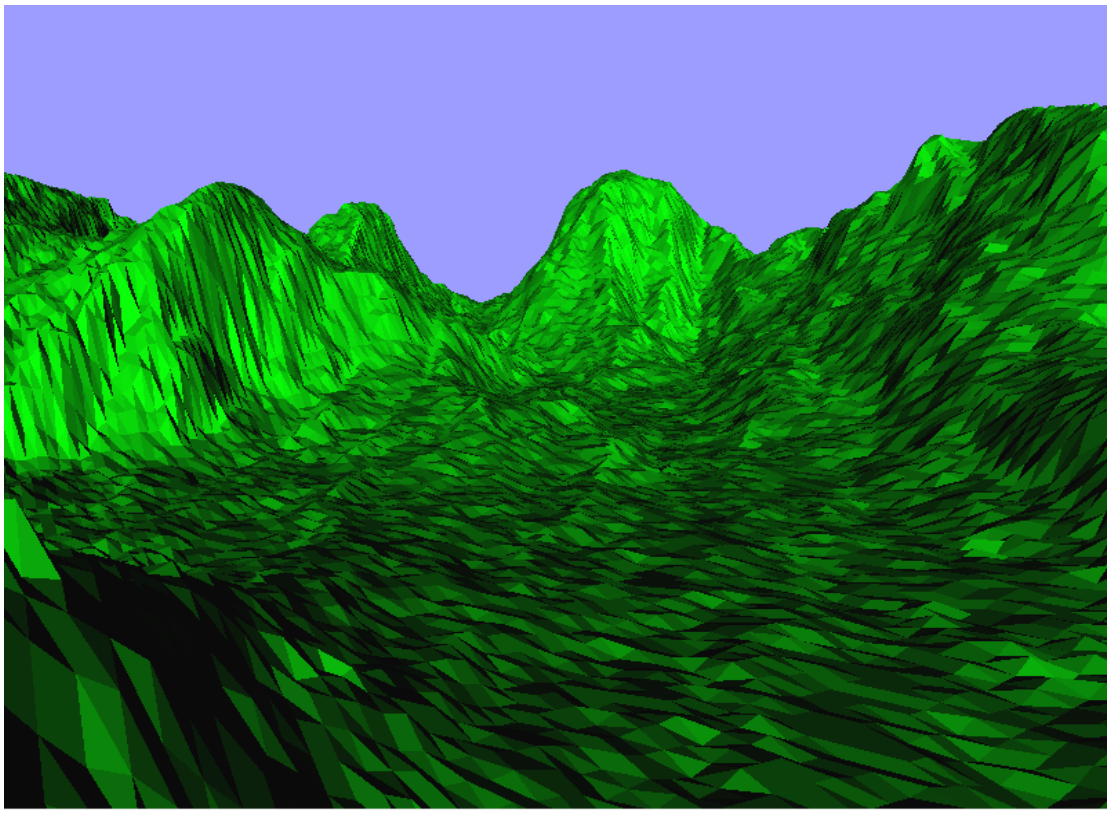

Figure 7.5 A simple example of 3D terrain rendering in Unicon

The terrain in Figure 7.5 is loaded from a 128 by 256 height map. The data is stored in a list before being fed to an algorithm for final rendering. The algorithm generates x and z values at fixed intervals and combines them with a y value from the height map creating a three dimensional vertex. Three vertices are then combined to create a triangle that can be rendered in the scene. Most vertices are replicated up to six times to be used with adjacent triangles sharing the same vertex coordinates. The result is a list of 194310 vertices to build 64770 triangle. The approximate performance of using traditional list compared with using real array list is presented in Table 7.1 below:

Table 7.1 Terrain rendering, traditional list vs. real array list

	Traditional List	Real List
Untextured Render (FPS)	53	170
Textured Render (FPS)	30	120

The benefit of using array representation for lists is obvious in this case. When rendering terrains in Unicon, the memory usage to render terrain data is brought down to 25% that of traditional lists. The rendering speed is also significantly higher with three to four times faster than traditional lists. This not only allows smoother graphics, but also allows more complex scenes and leaves a lot of room to render other objects in the scene.

7.3.3 Arrays and 3D Models Rendering

Similar to terrain rendering, 3D models include manipulating and rendering huge amount of data. The difference between 3D models and terrain is that the former contains structured data to allow animating different parts of the 3D model. This section demonstrates the effectiveness of using array representation of list to store and render such objects. The examples include two Microsoft ".x" 3D model files for a low-polygon model and a high-polygon model presented in Figure 7.6. The terms high-polygon and low-polygon do not refer to a standard or a universal polygon count terminology. The terms are used only to distinguish the two models relative to each other given their polygon count. Both models include skeletal bone system with animation information through key frames. Details about each model are presented in Table 7.2.

The experiment is designed to measure the 3D graphics rendering performance (FPS) of 3D models in Unicon using the traditional list representation and also the new array representation. Each test is run seven times, the lowest and highest numbers are discarded before taking the average of the remaining five numbers. The experiment was conducted on a laptop computer with the configuration described in Table 7.3.

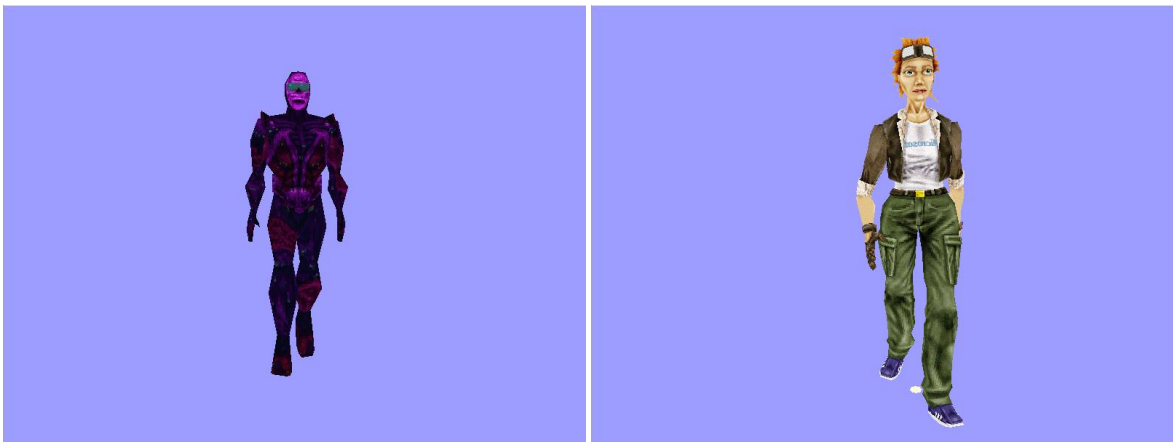

Figure 7.6 Low-polygon model (Left) and high-polygon model (right) rendered in Unicon

Table 7.2 Details of the models used in the experiment

	Low-Polygon	High-Polygon	Ratio (High/Low)
Vertex Count	475	4884	10.2
Triangle Count	528	6841	13.0
Texture Coordinates	475	4884	10.2

Table 7.3 The configuration of the laptop where the tests were conducted

CPU	AMD Llano Quad-Core A8-3530MX Accelerated Processor (2.6GHz/1.9GHz, 4MB L2 Cache)
GPU	Integrated (On-chip) AMD Radeon HD 6620G graphics controller
RAM	8GB DDR3-1600
OS	Fedora 16 64-bit
Screen	1920x1080@60Hz

Table 7.4 presents the result of rendering the two models using traditional lists and array lists. The table is split in half; the upper half shows the results of rendering the models without animation. The lower half shows the results of rendering animated models. Figure 7.7 present some of this information visually to compare the results of rendering the models using lists or arrays. The figure also shows the difference between non-animated models and animated models. The results show about 1.5X speed up with low-polygon model using arrays instead of list to store the model data. The high-polygon model sees about 3.5X speed up. This is an indication that in general, models with more polygons (more data density) benefit more from using arrays. That is because more polygons means more data conversion overhead when using list, and this conversion is avoided when using arrays.

Table 7.4 Low-polygon vs. high-polygon 3D models rendering using the two list formats In Unicon

	3D Model	# Polygons	List	Array	Speed Up
Static	Low-polygon	528	587 FPS	840 FPS	1.43
	High-polygon	6841	112 FPS	393 FPS	3.51
	Ratio	0.08	5.24	2.14	
Animated	Low-polygon	528	375 FPS	585 FPS	1.56
	High-polygon	6841	36 FPS	108 FPS	3.00
	Ratio	0.08	10.42	5.42	

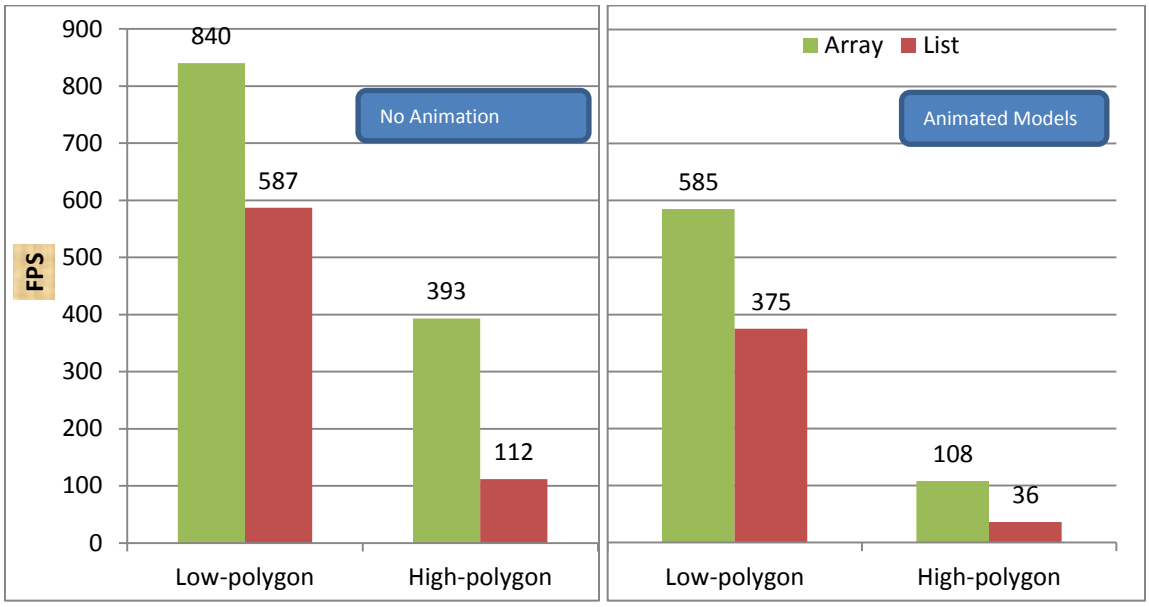

Figure 7.7 3D Model rendering using arrays and lists

Another experiment is to see how the rendering scales up with increasing number of models/polygons in scene, and how arrays improve the scalability of 3D graphics in Unicon. This is done for both animated models and static models. For static models, it can be seen in Table 7.5 that arrays scales the graphics performance up to 4.5X for the low-polygon model (Figure 7.6) used in the experiment. For most cases, the number of models rendered using arrays can be tripled while maintaining the same frames per second rate of that of list (Figure 7.8). Similarly, in Table 7.6, animated models gain a performance boost when using arrays allowing the rendering of about double the number of models compared with lists maintaining the same frame rate (Figure 7.9).

Table 7.5 Rendering Static models (No animation)

# Models in the scene	# Polygons	List - FPS	Array -FPS	Speed Up
1	528	587	840	1.43
4	2112	216	488	2.26
8	4224	134	345	2.57
16	8448	67	198	2.96
25	13200	41	134	3.27
49	25872	22.3	76	3.41
77	40656	13.9	51	3.67
99	52272	9.2	41	4.46
117	61776	8.6	35	4.07

Table 7.6 Rendering animated models

# Models in the scene	# Polygons	List - FPS	Array - FPS	Speed Up
1	528	375	585	1.56
4	2112	107	248	2.32
8	4224	43	105	2.44
16	8448	16.6	42	2.53
25	13200	10.1	23.2	2.30
49	25872	3.4	7.5	2.21
77	40656	1.8	3.8	2.11
99	52272	1.2	2.4	2.00
117	61776	1.1	1.8	1.64

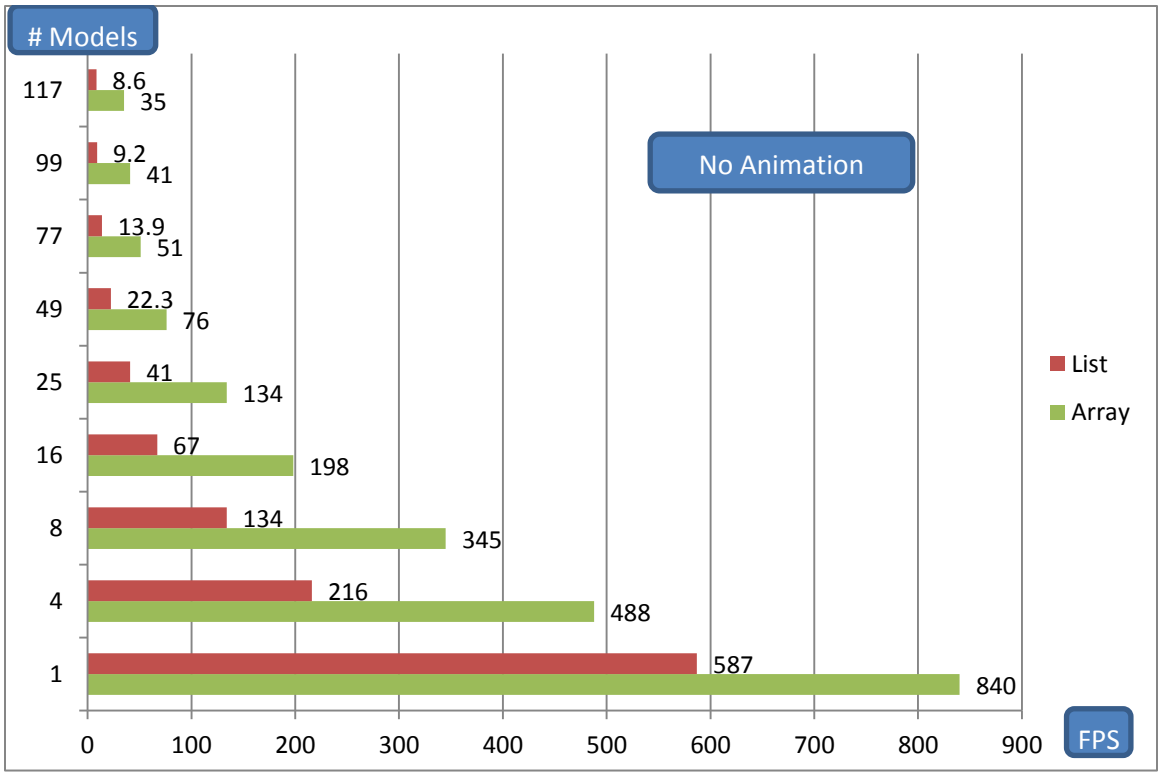

Figure 7.8 Rendering performance of non-animated 3d models using lists vs. arrays

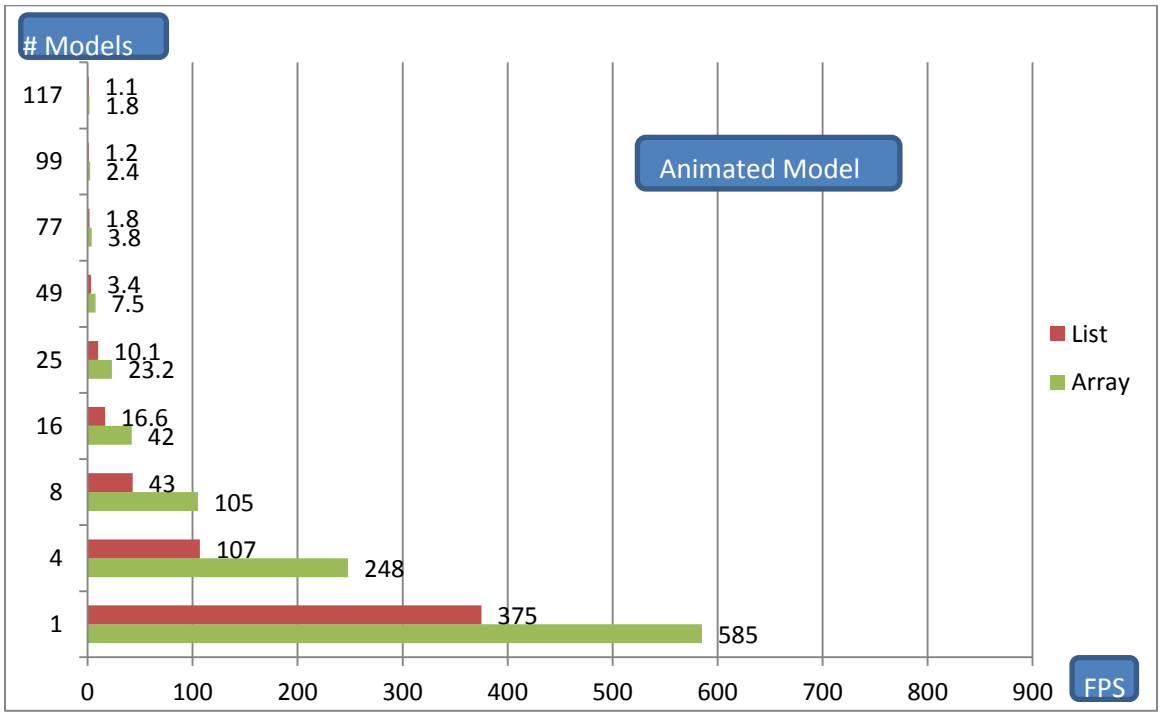

Figure 7.9 Rendering performance of animated 3D models using lists vs. arrays

The last topic covered in this analysis is a comparison between static models and animated models in how they perform using arrays instead of lists. The numbers in previous tables and figures show that there is a big difference in performance between the static and animated models. Figure 7.10 and Figure 7.11 show the performance trends when adding more models to the scene in both cases, static models and animated models. With static models, once the model is rendered the first time, it becomes a matter of passing arrays of data to OpenGL. In case of animated model, the vertex coordinates have to be recalculated between frames. For this reason static models scale better and gain more speed up using arrays. The results in the figure are specific to the low-polygon model and only presented to show the trend, the actual numbers and speed up will differ from one model to another. As presented in Figure 7.7, the more polygons a model has, the more it benefits from using arrays.

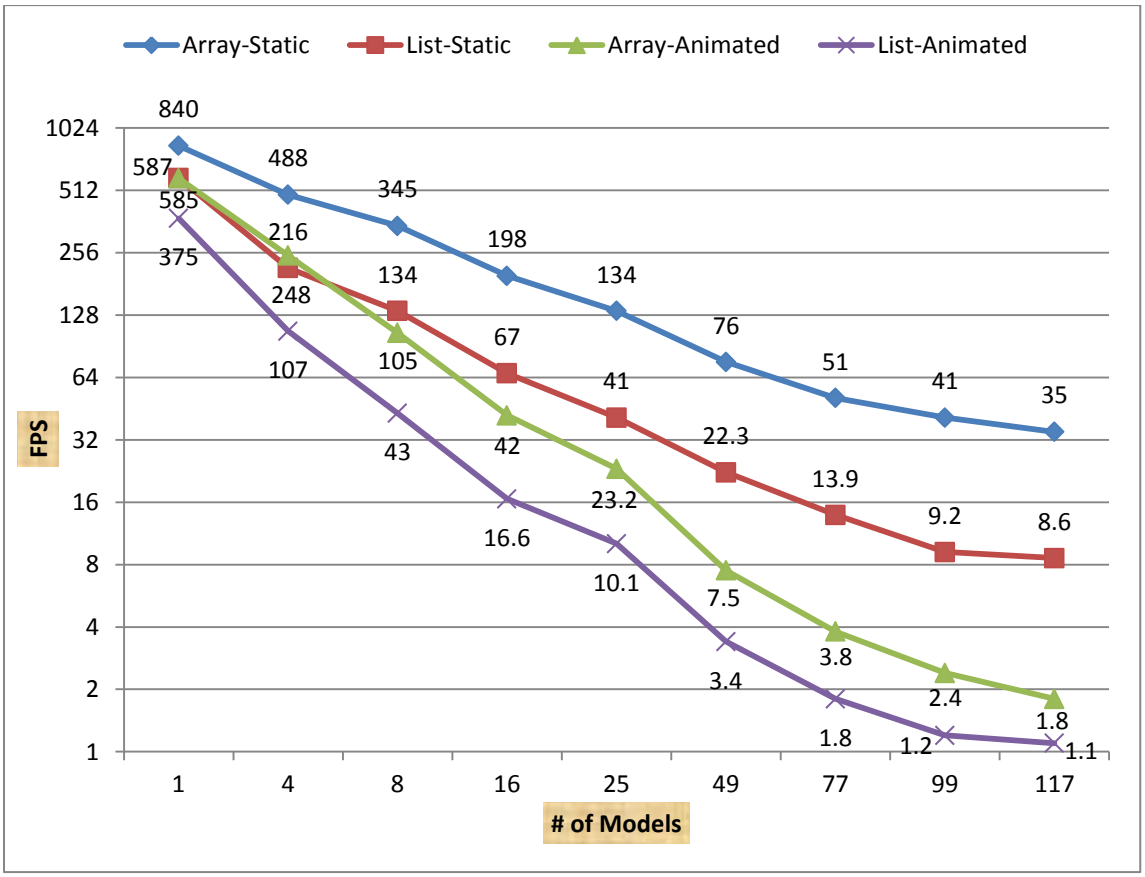

Figure 7.10 The effect of adding more models (static/animated) on frame rate when using arrays and lists

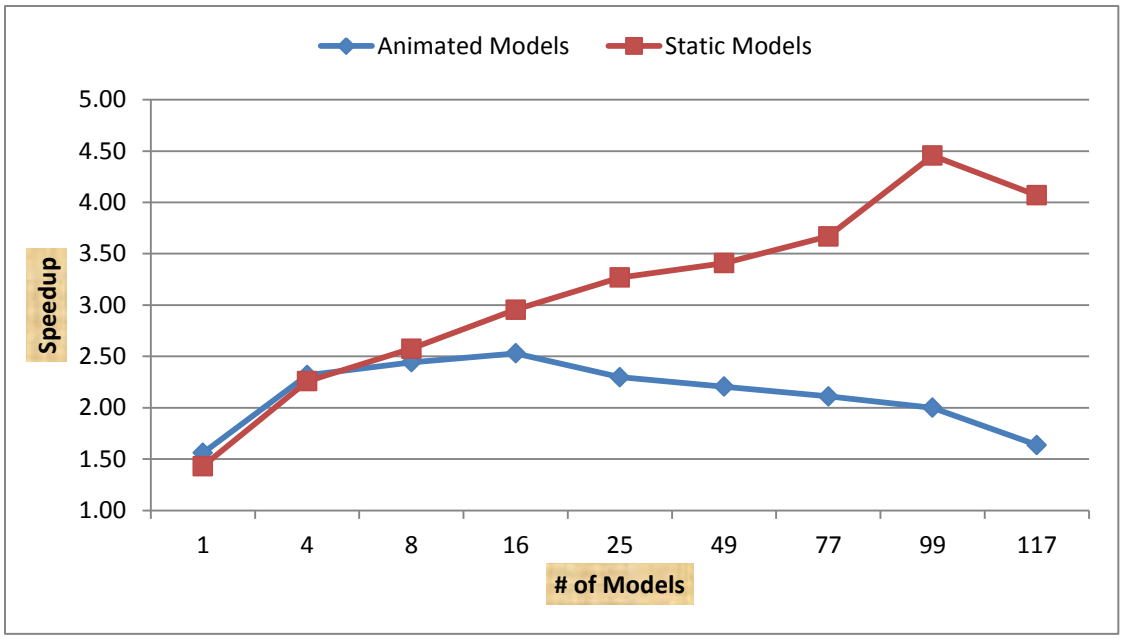

Figure 7.11 Speed up gains of static models and animated models when switching from lists to arrays

7.4 Threads Performance

Even after a successful implementation of concurrency, improving application performance by using threads is not trivial. The performance gain depends on the application coding style (whether the application was a concurrent design starting from a clean slate, or a conversion of an originally sequential program) and the dependencies in the code and data. Synchronization poses a challenge in some applications. Synchronization and thread-local storage are also a source of slowdown for the language infrastructure. Garbage collection remains the biggest factor in preventing some applications from taking full advantage of multiple cores available in a given machine, especially for memory- intensive and long-running programs.

7.4.1 Performance under Varying Conditions

Several experiments were conducted to measure the performance of threads in Unicon under different conditions, varying the number of threads, heap size, garbage collection frequency, and many other factors. Results were obtained on a Sun SPARC M9000 with 128 cores running Solaris 10. Experiments used three simple multi-threaded Unicon programs. The first, *sum,* includes a simple loop that counts to a large integer. The program can split this job among several threads. The loop does not do any memory allocation and so is not affected by garbage collection at all. On the other hand, the program *heavy* is a garbage collection-intensive test that keeps allocating memory and trashing it, filling the heaps very quickly and

forcing very frequent garbage collections. The last program, *sl* finds the sum of a long list of integers. All times are in seconds and were measured using the shell command *time,* which means the time measured include the creation, setup and initialization of the program and the threads, and also the time to setup data in the case of *sl* test. Figure 7.12 shows the performance of these programs with an increasing number of threads.

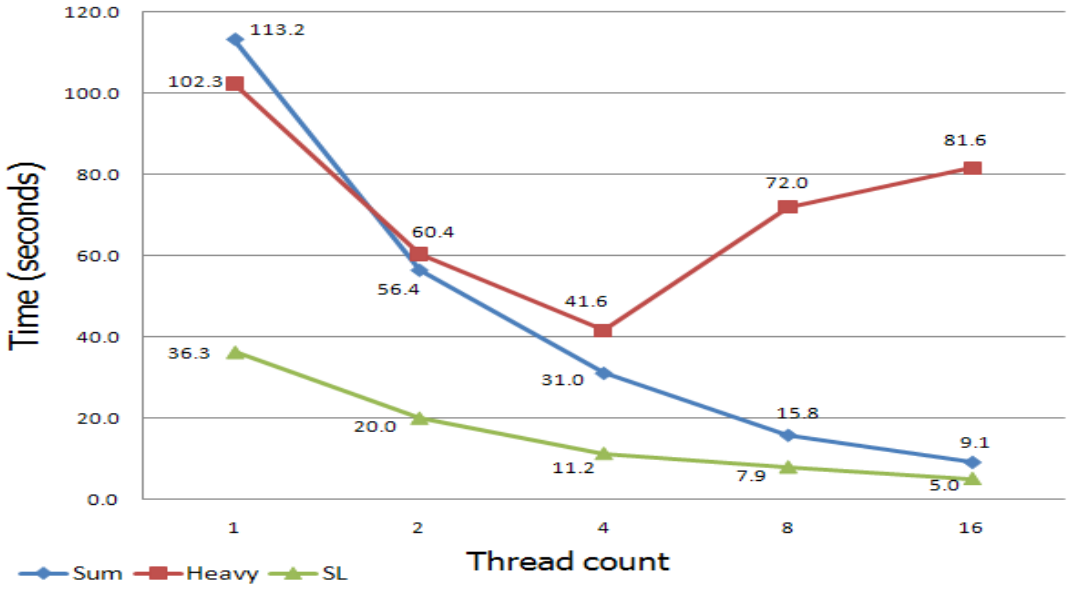

Figure 7.12 The effect of adding more threads in several programs

sum runs twice as fast when the number of threads is doubled, as expected, since the threads are independent and the program does not do garbage collection. *heavy* follows the same pattern with 2 and 4 threads. However, because it is memory-allocation demanding with a very high rate of garbage collection, adding more threads has a negative impact: more threads require more synchronization and more time to suspend and resume for each garbage collection. *sl* gets high speedup with more threads but less than *sum*. That is mainly because *sl* first initializes the big list of integers to be used, and this is done by the main thread before the other threads start. This initialization takes a constant time, independent of the number of threads.

Figure 7.13 demonstrates the effect of the heap size allocated for each thread on the performance of a program. The heap size affects how frequent the garbage collection is. These results depict the garbage collection-intensive *heavy* program. The figure shows that in an extreme case of garbage collection, a program does not benefit, or gets only a modest increase in speed from larger heaps with few threads running. The speedup increases with the number of threads if combined with an increase in heap size.

Figure 7.14 illustrates the value of forcing some threads that have almost full heaps to garbage collect if another thread triggered a garbage collection.

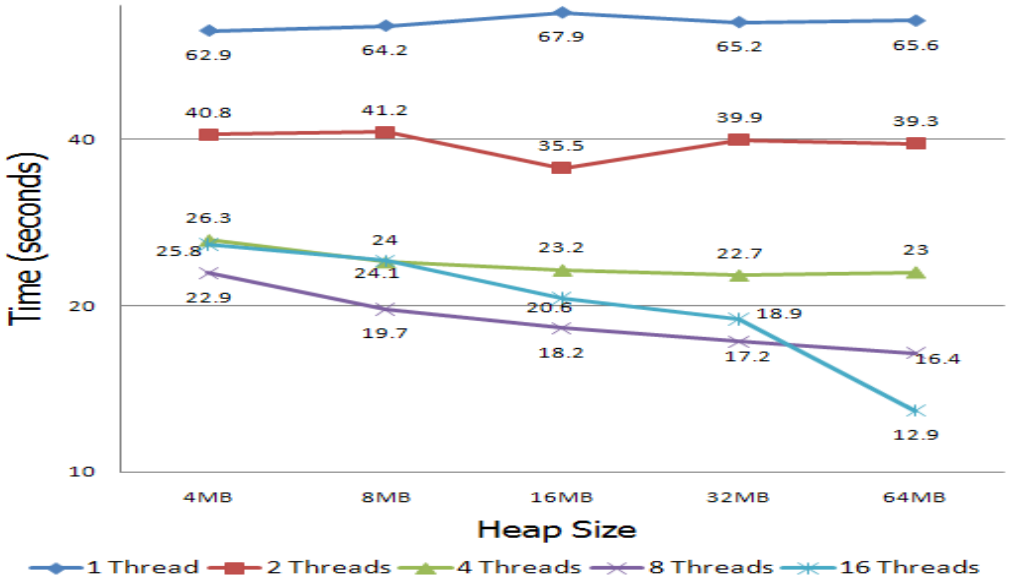

Figure 7.13 Heap size effect on the performance of garbage collection intensive program

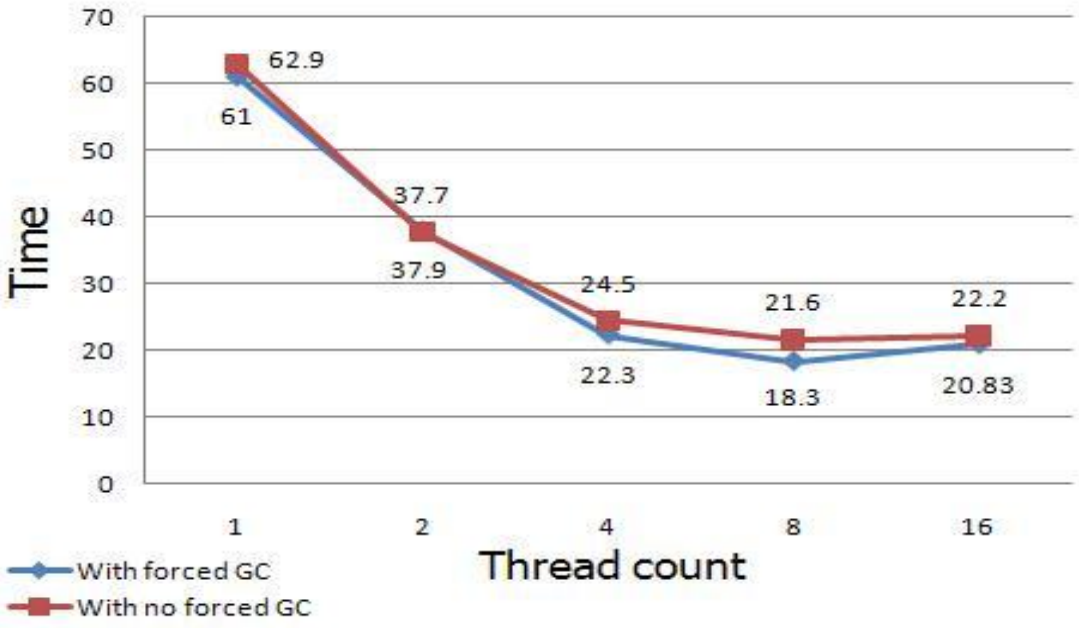

Figure 7.14 Forcing threads with semi-full heaps to garbage collect

7.4.2 Real World Applications Performance

The previous section presented some basic results measuring threads performance under varying conditions such as heap size and garbage collection frequency. This section evaluates the threads facilities in Unicon by presenting results for three classical concurrent problems: matrix multiplication, list sort, and string processing combined with I/O. The implementation of these algorithms is not the most efficient or the optimal solution. Instead, the implementation is kept simple so that it does not diverge from the sequential implementation very much. The idea is to test the performance improvement that can be achieved with little work from the programmer and without extensive knowledge of parallel programming.

Matrix multiplication is one of the easiest tasks to parallelize. Each thread is assigned a slice of the resulting array to compute its elements as presented in Figure 7.15. Since the two inputs arrays are read-only, there is no thread safety issues or data to protect against race conditions. As long as the arrays are large enough, adding more threads scales very well in bringing the time required to compute the result of the multiplication down. The source code for the parallel matrix multiplication is listed in Listing 7.1.

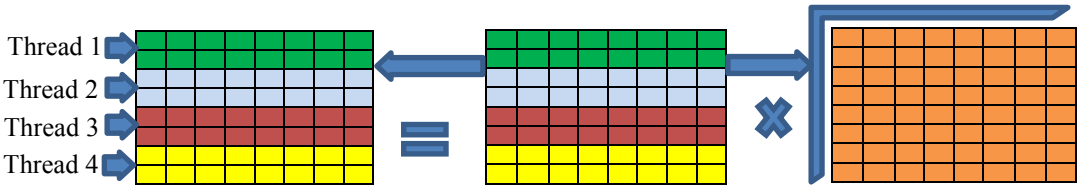

Figure 7.15 Matrix multiplication: each thread works on a slice independently from other threads

For sorting a list the *Quicksort* algorithm is chosen. Unicon's class library includes an implementation of an in-place *Quicksort* function. The same function is utilized with a wrapper function used to divide the work and to spawn new threads to sort different parts. The wrapper function itself is a modified version of *Quicksort*, which is able to launch new threads if possible and appropriate, based on the number of threads and the amount of data to be processed. The function does one *Quicksort* iteration (pick a pivot and swap elements left or right of the pivot if they are less or greater than the pivot), at that stage the left and the right of the pivots can be thought of as two independent lists. The original thread can work on one side, after creating another thread that takes on the other side as explained in Figure 7.16.

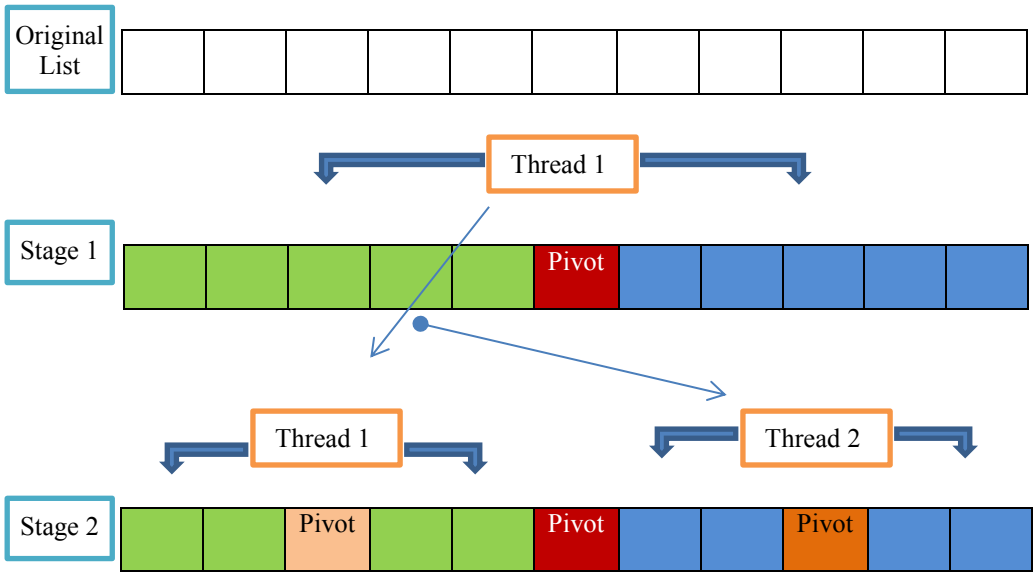

Figure 7.16 Quicksort: at each stage a new thread could be spawned to handle the other half of the list

Unlike matrix multiplication, *Quicksort* is more prone to load balancing due to dependency on the value of the pivot, which is picked from the middle of the list in this implementation. However, to prevent all or most of the available threads from crowding at one side of the pivot, a thread is only allowed to spawn a new thread when the size of the remaining half of the list is greater than a certain value that also takes the number of the maximum threads allowed into account. One shortcoming of this implementation is that once a thread finishes sorting its slice, it exits. A better implementation would allow this thread to go back to a pool of threads where they can be assigned new slices to sort. As mentioned before, the objective of this experiment is not to create the best of implementation of *Quicksort*; rather the objective is to test the performance of a straightforward implementation that is very similar to the sequential one. The source code of the parallel *Quicksort* is listed in Listing 7.2.

The last program in this test is a word counting tool (*Wordcount*) listed in Listing 7.3, similar to the command `wc` on Linux. The tool counts all of the words in all of the files in a given directory including all of its subdirectories, i.e. a recursive tool. The implementation starts by creating the maximum number of threads allowed, then generating all of the file names that need to be processed. The file names are placed in a shared a list where all threads start pulling the file names and counting their words. The result is accumulated at the end.

```
import threads
#args:  1- dimension of the matrix  2- number of threads
procedure main(argv)
  n := integer(argv[1]) | 2
  nthread := integer(argv[2]) | 1
  nthread >:= n
  &random := 0
  t := microseconds()
  M1 := initm(n)
  M2 := initm(n)
  t := microseconds()
  R:= mm(M1,M2, nthread)
  write("Mul time=",microseconds()-t)
end

procedure initm(n, zero)
   M := list(n)
   every M[1 to n] := array(n, 0.0)
   if /zero then{
      every L:=!M do
         every L[1 to n] := ?0
      }
   else{
      every L:=!M do
         every L[1 to n] := 0.0
      }
   return M
end
# do M1 x M2 using nthread threads
procedure mm(M1,M2, nthread)
   R := initm(*M1, 0.0)
   split := *M1/nthread
   L:=[ ]
   # create threads to do other slices
   every i:=1 to nthread-1 do
      put(L, thread submm(M1, M2, R, i*split+1, i*split+split ))

   # do the first slice
   submm(M1, M2, R, 1, split)
   every wait(!L)
  return R
end
# R = M1 x M2. do the slice from s to e
procedure submm(M1,M2, R, s, e)
   every i := 1 to *M1 do
      every j := s to e do
            every k := 1 to *M1 do
               R[i][j] +:= M1[i][k] * M2[k][j]
  return R
end
```

Listing 7.1 Matrix multiplication source code

```
import threads
import lang
global nthread, xthread, LT
#args:  1- size of list  2- number of threads
procedure main(argv)
  n := integer(argv[1]) | 2 ;  nthread := integer(argv[2]) | 1
  nthread >:= n ; &random := 0
  t := microseconds()
  A := initArr(n)
  if nthread=1 then {
    t := microseconds()
    A:= qsort(A)
    write("sort time=",microseconds()-t);   return
   }
  xthread := Shared(1); LT := mutex([ ])
  t := microseconds()
  A:= psort(A)
  every wait(!LT)
  write("sort time=", microseconds()-t)
end

procedure initArr(n)
   A := array(n, 0.0);  every A[1 to n] := ?0;   return A
end

procedure psort(L, first, last)
   local i, j, pivot
   /first := 1; /last := *L
   i := first; j := last
   if i = j then  return L
   pivot := L[(i + j) / 2]
   repeat {
      while L[i] < pivot do i +:= 1;    while pivot < L[j] do j -:= 1
      if i <= j then {     L[i] :=: L[j]; i +:= 1; j -:= 1   }
      if i > j then break
   }
   if xthread.value<nthread then{  # if allowed to create another thread
      lock(xthread)
      if xthread.value<nthread & j-first>(*L/nthread/1.05)then {
         if first < j then
            xthread.value+:=1
            unlock(xthread);  put(LT ,thread psort(L, first, j))
             }
      else{  unlock(xthread);   if first < j then qsort(L, first, j)  }
      if i < last then
          if last-i>(*L/nthread/1.05) then psort(L, i, last)   else qsort(L, i, last)
      }
   else {
      if first < j then qsort(L, first, j);   if i < last then qsort(L, i, last)
      }
   return L
end
```

Listing 7.2 *Quicksort* source code

```
import threads
link strings
global done
procedure main(argv) # arg : 1- # of threads  2- The target directory 3- The word to count
  nthread := integer(argv[1]) | 1;  path := (argv[2]); str := (argv[3]) | "main"
  t := microseconds()
  if nthread==1 then L := count_seq(path, str) else L := count_par(path, str, nthread)
  write("time=", microseconds()-t)
end

procedure count_seq(dir, s)
  local ldir, L, t, L1 := [ ], L2 := [ ]
   chdir(dir); do_dirs(dir, L1); done:=1
   count_s( L1, L2)
   return L2
end

procedure count_par(dir, s, nthread)
  local ldir, L:=[ ], t, L1 := mutex([ ]), L2 := mutex([ ])
  every i:=1 to nthread-1 do put(L, thread count_s(s, L1, L2))
  chdir(dir);do_dirs(dir, L1); done:=1
  count_s(s, L1, L2)
  every wait(!L)
  return L2
end

procedure do_dirs(dir, L)
   ldir := open(dir)
   every d:= !ldir do{ if d== ("." | "..") then next
       d := dir ||"/" || d;
       if stat(d).mode[1]=="d" then do_dirs(d, L) else  put(L, d)
    }
    close(ldir)
end

procedure count_s( s, L1, L2 )
  local file, fs, fin, sum := 0
  repeat{
    if file := pop(L1) then
      if (fin := open(file)) then {
         every words(reads(fin, -1)) do
            sum+:=1
         close(fin)
         }
   if *L1=0 & \done then break
   }
  put(L2, sum)
end
```

Listing 7.3 *Wordcount* source code

Table 7.7 Configuration of the test machine

CPU	4 AMD Opteron (8-Modules/16-core) totaling 32-modules/64-cores running at 2.4GHz
Storage	90GB SSD
Architecture	64-bit
RAM	32GB DDR3-1866
OS	Fedora 16 64-bit

The test machine for running the three test programs has the specifications listed in Table 7.7. Each two cores in the CPUs are called a module which has some shared resources between the two cores (such as a floating point unit) making the machine not a true 64-core. The 64-cores provide an advantage in some programs and the results are included, however, the results will only be taken seriously up to 32 threads, especially on programs that do extensive floating point computations. Figure 7.17 presents a summary of running the different programs using an increasing number of threads.

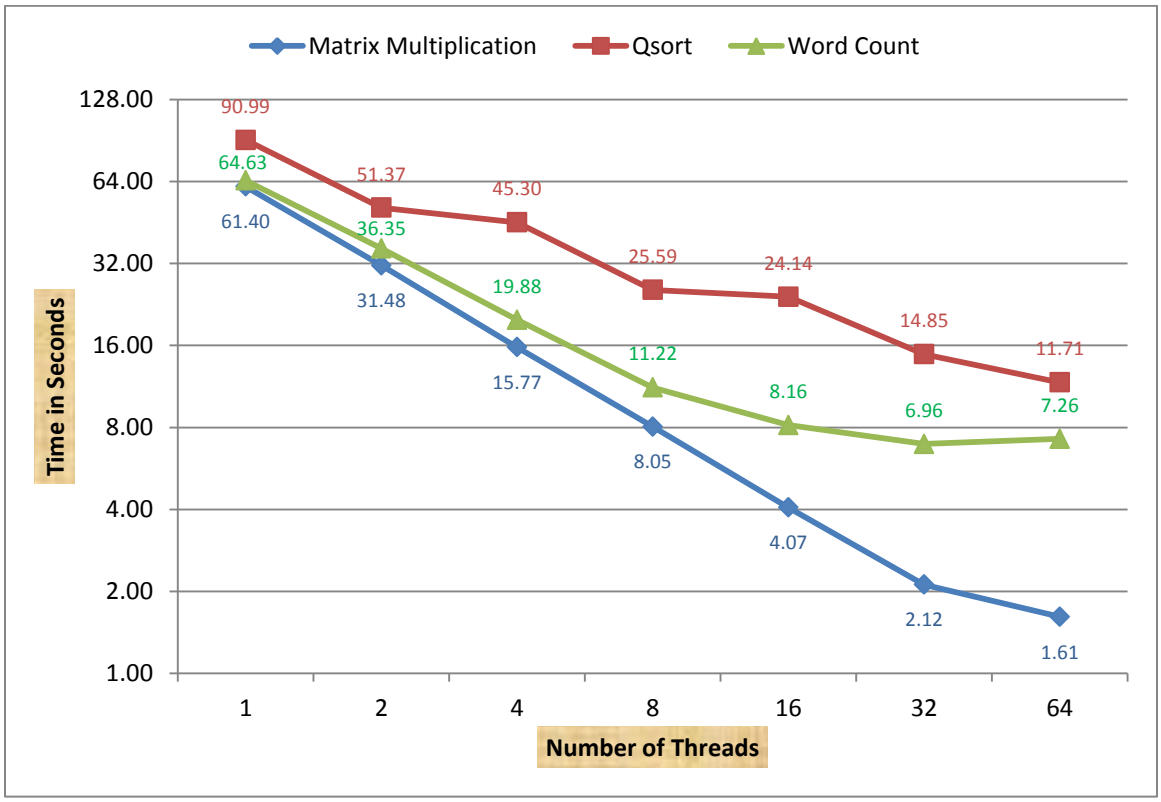

Figure 7.17 Different programs and their response to increasing the number of threads

As expected, matrix multiplication scales very well taking full advantage of new threads. Doubling the number of threads doubles the performance, cutting the execution time in half. This is a strong indication that the language itself does not impose a significant cost on increasing the number of concurrent threads. The sharp drop in performance from 32 threads to 64 threads shows the impact of modules sharing floating point units in a floating point heavy program such as matrix multiplication. For the majority of applications, the performance gain from using threads is only limited by the amount of parallelism in the application itself. The thread safe features such as internal language mutexes and thread local storage used in the virtual machine and the runtime system do not add a significant overhead when adding more concurrent threads.

Quicksort and *Wordcount* greatly benefit from threads, but they do not scale as well as matrix multiplication. Using four threads cut the time in half in the case of Quicksort and by two thirds in the case of *Wordcount* still a big improvement over the sequential versions. *Wordcount* does a lot of I/O which is a limiting factor on how much concurrency improves the program. The specific implementation of *Quicksort* does not help the program to take advantage of the concurrency to its fullest. For example, having one thread do a full iteration at each step before launching a helper thread, and also, not carefully picking a pivot that balances the work load between the two threads at each load. Furthermore, not recycling a thread after finishing sorting its slice, also prevents a better utilization of the available threads. Figure 7.18 summarizes the actual thread/core utilization of each of the three programs compared to the theoretical maximum utilization that can be achieved given the number of threads the program uses.

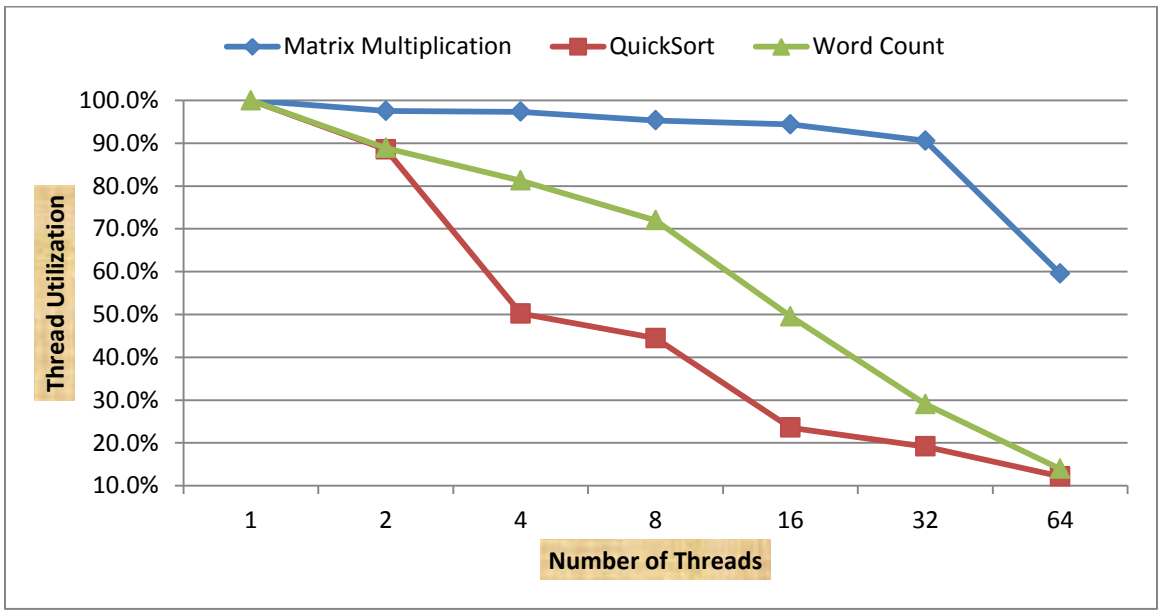

Figure 7.18 Thread utilization in each program given the number of the concurrent threads

The last subject covered in this section is the overhead of concurrency features built into Unicon on sequential programs. Two versions of Unicon are used in this experiment, one built with concurrency enabled (concurrent), while the other is built with no support for concurrent threads (non-concurrent). Figure 7.19 presents the results of running the three programs using the two versions of Unicon. The largest overhead in concurrent Unicon can be seen in *Wordcount*. This is mainly because *Wordcount* processes of tens of thousands of files with intensive I/O. All of these files are protected by mutexes. The difference in the other two programs is less significant, with *Quicksort* scoring slightly better performance using concurrent Unicon due mainly to improved locality in some part of the language implementation that happens to work in favor of *Quicksort*. The average overhead in these three programs is 5%.

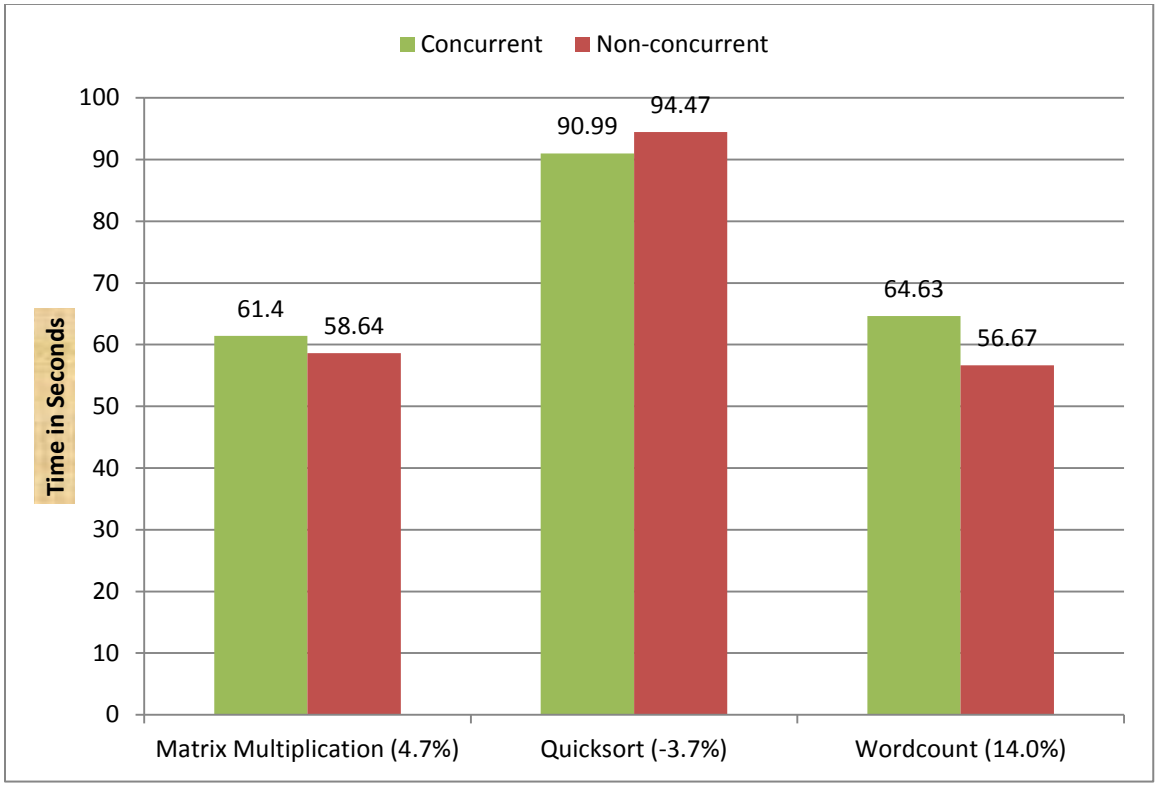

Figure 7.19 Sequential performance of concurrent and non-concurrent versions of Unicon (concurrency overhead between parenthesis)

8 Portable Extensible Non-player Characters and Quest Activities

Non-player characters (NPCs) are an important component of role playing games and massively multi-player online games (MMOs), presenting activities and the storyline of the game. Computer-controlled NPCs can be enemies, allies, monsters, or pets, but their most important role is that of quest giver or assistant to the user who goes on an adventure. Quest givers are also important because educational content can be delivered to users through quests in educational virtual worlds. This technique can be used to draw users into tutorial quests.

As mentioned earlier in section 2.4, a simple NPC architecture has been developed as part of this dissertation called PENQ (Portable Extensible Non-player characters and Quests). PENQ is a prototype. For the purpose of this dissertation, only essential parts are implemented, which make such NPCs viable test subjects for the new language features. These essential parts include basic and partial support for knowledge, dialogue and behavior models described in the following sections. The NPCs can keep track of the quests they have, the quests each user accomplished, can carry on very short conversations with predefined chat messages, and have very basic wandering capability. Providing a full and robust NPC implementation is outside the scope of this dissertation, and will be left as future work.

The main purpose of these NPCs is to add actions in the CVE virtual world, stressing both the clients and the server and test new language features. NPCs in CVE not only enrich the virtual world, they also open the door for more use cases and test scenarios once their implementation is complete. PENQ architecture is independent from the CVE architecture, and can be run as independent processes connecting to the server as regular clients. PENQ NPCs can also be run inside the virtual world server if the server supports such integration as it is the case for the CVE server presented later in this chapter. Figure 8.1 shows several avatars from the CVE virtual world including NPCs that were added to CVE as part of this dissertation. A quest giver NPC in CVE has a red ball over his head. This marks NPCs with available quests as persons in the environment with whom the user has special reason to interact.

The addition of NPCs triggered many of the new language features presented in this dissertation. In most aspects, NPCs are similar to regular users, putting more pressure on the virtual world server and the client. For CVE that means finding new ways to improve the scalability of in terms of the number of users it can handle. Similar things can be said about the client, especially that NPCs brought with them 3D models which put extensive pressure on Unicon 3D graphics rendering pipeline, requiring new features such as array and thread support.

This chapter presents the PENQ architecture including NPCs and their quest system along with a summary of their design and prototype implementation. The chapter briefly introduces NPCs in the CVE with 3D

models support in the CVE client followed by the integration of NPCs in the CVE server as threads. The evaluation of NPC running as threads is presented in section 8.5. The impact of NPCs on the client is left as part of the general CVE evaluation in the next chapter.

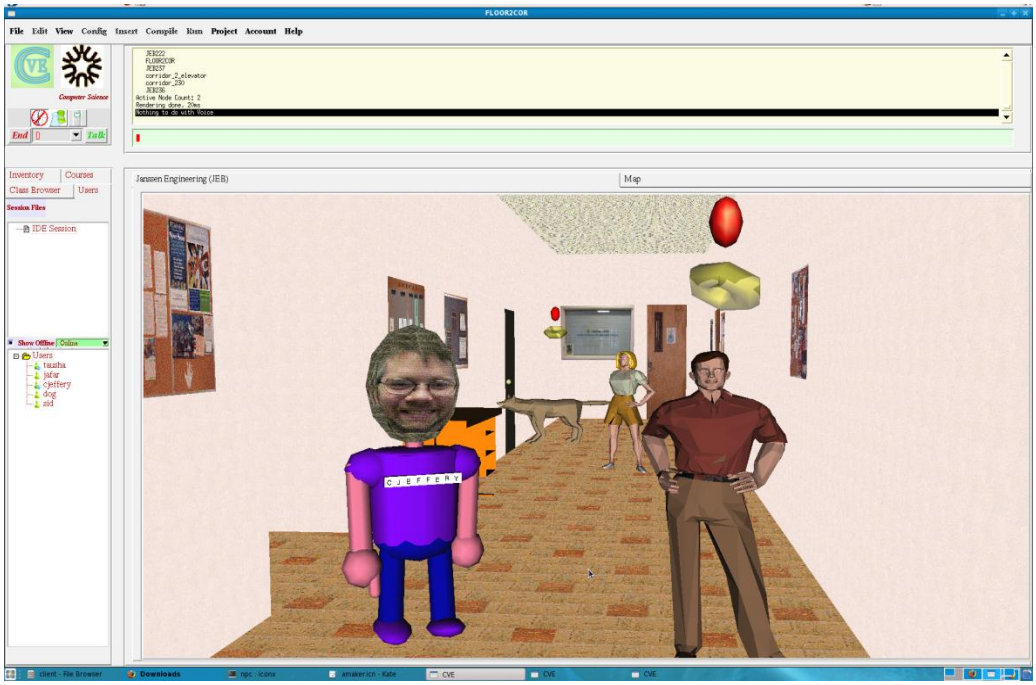

Figure 8.1 NPCs and other users in CVE. An NPC is marked with red ball above their heads

8.1 Non-player Characters

A PENQ NPC is created much as a regular (human-controlled) character. End users can utilize their regular account or an auxiliary account to create an NPC character, adding quests and activities and making them available to other users. The intention is that when the user is logged off, their avatar is still present on the system, controlled from a remote NPC client just like a regular user, interacting with other users as instructed by the player.

8.1.1 NPC Profiles

An NPC profile is a file containing NPC details in simple HTML format. An NPC profile can be created and maintained as a webpage. In HTML they are each given in a named anchor tag. Although a graphical wizard for creating profiles is available, many NPCs can be created manually by copying a template and changing the content details. Table 8.1 lists the major parts in an NPC profile along with a short description about each part.

Table 8.1 The major parts in an NPC profile file

id	An "ID card" presentation of the NPC, suitable for an "inspect details" operation in a game. The id provides an image, name, and basic attributes. Figure shows an example ID card
knowledge	A specification of the NPC's knowledge model consists of a teaching section with a bulleted list of named links to quests. NPC knowledge also includes a more dynamic experience (user model) database that is not part of the profile
dialogue	A specification of this NPC's verbal capability, and thereby its personality
behavior	A specification of this NPC's active (e.g. mobile) behavior. The four kinds of PENQ behavior specification include stationary, wanderer, routine, and companion
avatar	A specification of this NPC's avatar (link to 3d model file, dimensions, and textures)

Figure 8.2 An NPC ID card

8.1.2 Knowledge Model

PENQ NPCs use two types of knowledge: the quests they offer, and what they remember about other users from past quests. For this dissertation, the static knowledge is implemented, but only very basic dynamic knowledge is supported to allow the NPC to keep track of the user's quests accomplishment.

8.1.3 Dialogue Model

NPCs that chat feel more natural but are frustrating when they fail to respond usefully to a player's conversation. The PENQ NPC's dialogue model currently consists of offers to undertake available quests, plus a few chat messages are supported to allow the NPCs to greet other users, introduce themselves and invite users to take quests.

8.1.4 Behavior Model

The PENQ NPC behavior model provides rules for NPC movements and responses to external stimuli. At least four NPC behavior types are needed to support behavior common in current games: stationary,

routine, wanderer, and companion. A stationary NPC does not move from a specified home location. A routine NPC regularly does specific tasks including movements at predetermined times. A wanderer is an NPC that moves randomly within a prescribed domain. A companion NPC accompanies a player on a destination-based quest. For this dissertation, only stationary, and a basic wanderer NPCs are supported.

8.2 Quest Activities

In games, quest activities are used to teach the game itself as well as to entertain. The NPCs and quests described in this chapter of the dissertation are intended to fill an educational role, besides enabling evaluation of the new language features added to Unicon.

Quests are a primary mechanism for tutorial learning. Quest specifications resemble UML use case descriptions [97]. The kinds of steps are limited to those observable by NPCs interacting with users. The main differences between a quest and a use case description are that a quest may contain auxiliary content (such as quizzes and demonstrations) that are used to measure completion of the quest steps, and a quest lists rewards for completion, if any. Quizzes and demonstrations will often need to be external references to pools of questions. The difference between quizzes and demonstrations is that a quiz is delivered and answers interpreted by an NPC agent directly, while a demonstration involves an in-world interaction (in this case, a session with a tutorial UNIX command-line shell) that is monitored by an NPC agent. Evaluation of deeper understanding may require offline human evaluation, or fall outside the realm of what an NPC Tutor can reasonably perform.

8.2.1 Quest Repository

PENQ Quests are maintained in the same way as NPC profiles: they are HTML files linked from the knowledge section in the NPC's webpage. The quest webpage includes several sections. A quest builder tool facilitates the process of creating new quests. Figure 8.3 shows an example quest from the domain of computing. Figure 8.4 is a screenshot of a quest being offered in a response to a user clicking a quest red sphere above an NPC.

8.2.2 Quest Rating and User Reward via Peer Review

Players need to be motivated to perform quests, and their accomplishments need to be recognized. The main kind of reward that matters to PENQ is the experience points in specific topic areas that enable a character to undertake more advanced quest activities. In the ls activity presented in Figure 8.3, two specific previous quests (tutorials on Files and Directories) had to be completed before the NPC offers the

ls quest. Completing the ls quest enables any quest that depends on it specifically, and also awards a point of general UNIX experience.

```
Name    :       ls
Summary :       Learn the basics of the ls command.
Requires :      Files, Directories
Steps    :      Read the UNIX manual page for ls.
Pass a quiz on ls command line options.
Demonstrate "ls" for Tux.
Rewards :       UNIX: 1
Quiz (2/2 to pass)
How can you get a long listing that shows file permissions and size?
> ls -l
How can you list all files in all subdirectories?
> ls -R
Demo (2/2 to pass)
Show me a simple listing of the root directory.
> ls /
Show me a listing of the current directory, sorted by the time each file was last accessed?
> ls -t
```

Figure 8.3 An example quest as it appears in a web page

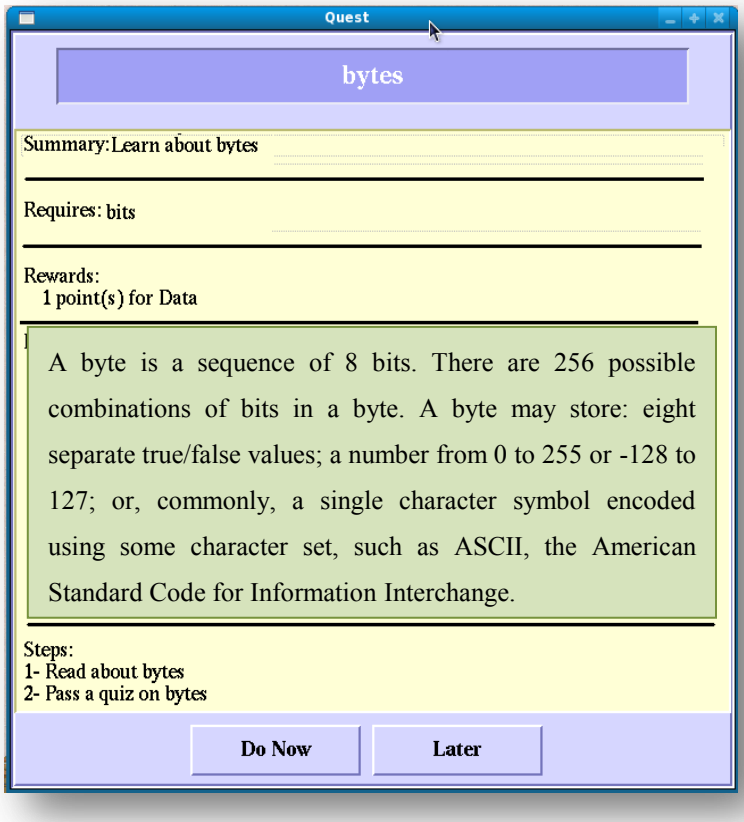

Figure 8.4 A sample CVE quest invitation dialog

8.3 Design and Implementation

The PENQ NPC is designed as a standalone entity; however, they can be plugged in as threads in the server. An NPC client connects to the server like a regular user, with an NPC indicator flag. When run as separate clients, this design frees the server from NPC management, makes the NPC more flexible by allowing the NPC to run from any machine and allows a human to "play" an NPC.

As mentioned in the introduction of this chapter, PENQ provides specifications of NPCs and quests and their envisioned design. The implementation details are left for specific implementations. For this dissertation purpose, only a small fraction of the specification is implemented. The design and implementation is described here to give an idea of what to expect from these NPCs, what functionalities they provide, and what support they require from the language and the virtual world.

8.3.1 NPC's Architecture

The NPC client design mirrors the NPC profile. The following is the list of the NPC's major components:

A knowledge engine: composed of two parts, the relatively static quest knowledge and the dynamic knowledge.

A behavior engine: dictates how the NPC will behave and move around in the world based on the NPC profile.

A chat engine: analyzes the incoming chat messages and generates proper response if possible. Chat messages are categorized either as general chat or messages that involve questions or answers about the quests the NPC is providing.

At the top level there is an I/O interface that manages data transfer between the NPC, virtual world server, HTTP servers and also the disk. Figure 8.5 shows the different NPC components.

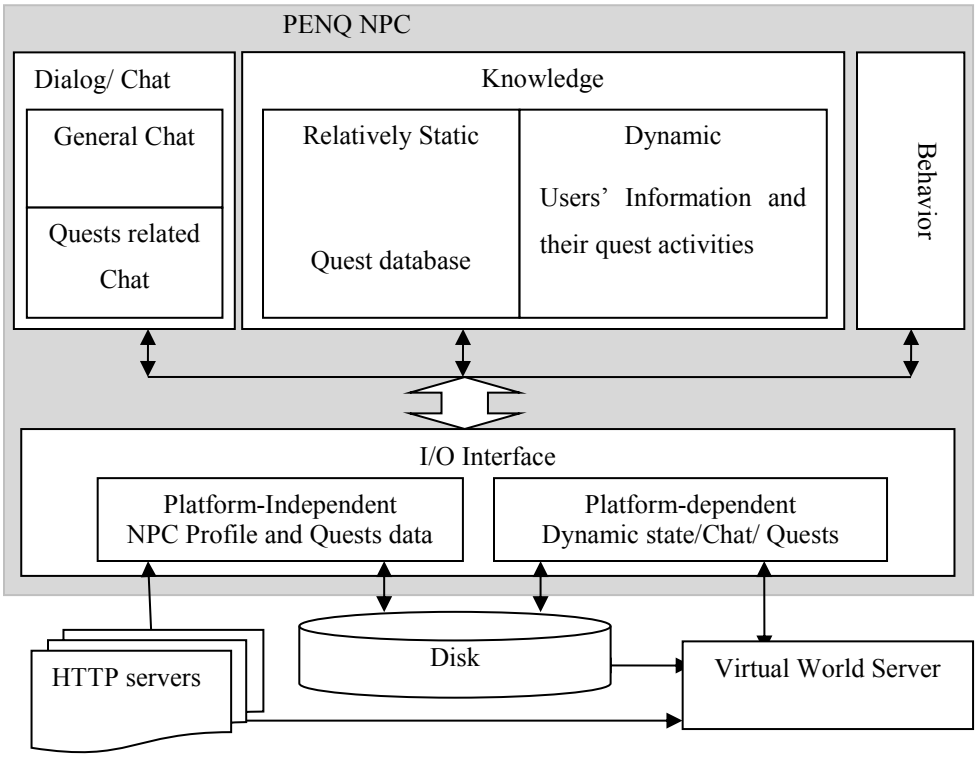

Figure 8.5 PENQ NPC Architecture

8.3.2 Implementation Discussion

The NPC client starts by downloading the NPC home page and quest pages that define the profile of the NPC. It then initiates a TCP connection with the virtual world server. Upon a successful login, the new NPC is available in the virtual world accepting interactions from other users.

8.3.2.1 Network Protocol

The PENQ NPC network protocol uses string messages consisting of a command name followed by arguments. This enables an easy integration with CVE when it comes to network communications since CVE also uses string messages. The NPC client recognizes the following messages coming from the virtual world server and other users: chat messages, messages about other users' locations in the virtual world, and messages about quest requests and activities. All commands begin with two forward slashes (//). Some commands have arguments to pass information to/from the NPC. A summary of the commands are listed in Table 8.2. A summary of the quest command and its options is presented in Table 8.3. The different options communicate information about quest activities between the NPC, server and clients. All of quest commands are prefixed with () and some of them take several arguments containing all of the information necessary for each quest activity.

Table 8.2 Summary of the most important NPC protocol messages

Category	Command	Description
User presence commands	users	brings up a list of all the other users who are currently online
	avatar	informs about a new user who just logged in
	move	informs about a specific user movement (x,y,z and direction)
Chat commands	say	public chat message sent to all users
	tell	private chat message sent to the intended user only

Table 8.3 Summary of quest commands

Prefix	Action	Description
npcmsg Quest	LookFor	asks the NPC for available quests
	Halo	informs the client that the NPC has available quests
	GiveMe	asks the NPC to send the next available quest
	URL	sends the client a quest URL
	Accept	informs the NPC that the user has accepted the quest
	Cancel	informs the NPC that the user has cancelled or abandoned the quest
	Done	informs the NPC that the user has finished the quest

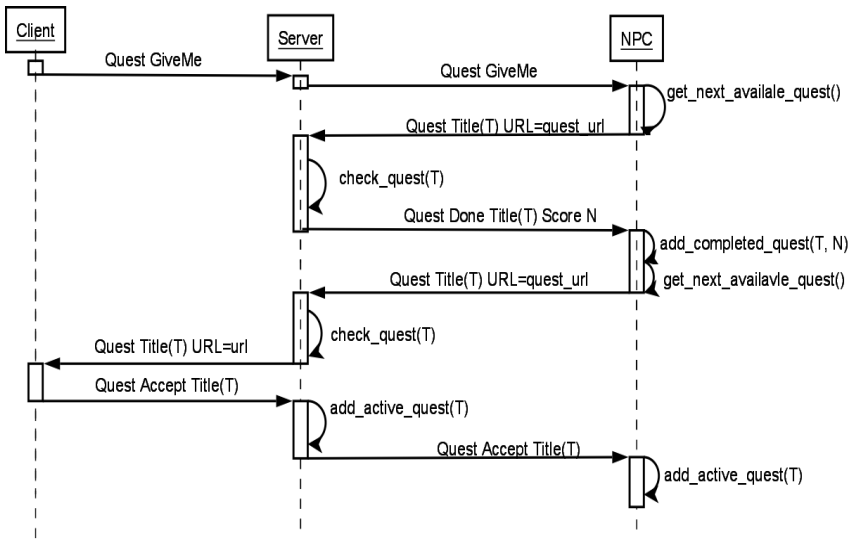

Figure 8.6 PENQ NPC quest messages between the NPC, the server and the client

Figure 8.6 shows a scenario where the quest messages are used to start a new quest. The process starts when a user client clicks the red ball marking a quest over an NPC. The click sends the message "Quest GiveMe" to the server, which forwards it to the specific NPC. The NPC finds the next available quest for that user based on the knowledge it has, then sends back a quest title along with its URL so that it can be downloaded directly from the source webpage. The server gets the message and checks whether the user has already completed the quest or is currently undertaking the quest. The NPC maintains a list of completed and active quests for every user, but if the NPC process gets restarted, the quest protocol allows it to reload user dynamic knowledge on demand.

The scenario in Figure 8.6 shows the case when the NPC needs to get updated. After a quest request message, the server replies to the NPC informing it that this user has completed the quest. The NPC then adds this piece of information to its database and finds another available quest for the user and sends it back to the server, which in turn checks again whether the user has already completed the quest. This time the server approves the new quest and forwards it to the client. The client gets the message and downloads the quest from the specified URL or loads it from the disk if it is stored on the local disk. If the user accepts the new quest, which is the case in the scenario we have here, it sends the message "Quest Accept Title(T)" back to the server. The server adds the specified quest to the active quest list for the user and forwards the message to the NPC which in turn also adds the quest to the active quests list for that user.

8.3.2.2 Source Code Organization

At the top level view, the NPC source code is organized into two major components: the NPC class and the client application that uses it. The NPC class, called ExternalNPC, features a public interface consisting of login() and mainloop() methods and a few other methods that control the NPC activities. The NPC class also holds most of the common features shared between different kinds of NPCs such as quest activities and basic chat capabilities. NPC client applications can customize the NPCs and give them distinguishing characteristics beyond what is modeled in the profile web page to any more advanced dialogue capabilities and behavior that are required for that specific NPC.

Given the ExternalNPC class, if a new NPC named Tux were created, what does the Tux NPC client look like? Tux has a profile on a web page. Listing 8.1 shows a minimal Tux NPC client to instantiate Tux in the CVE virtual world. Tux's profile is downloaded from the specified homepage. The methods handle_msg() and idlefunc() are called to handle the received messages from the server and to set what to do when Tux is idle. Although Tux is not doing anything except standing in a fixed position and logging whatever messages he gets from the server without responding to any of them, this is a building block for an NPC client that may form the basis for interesting NPC dialogue, knowledge, and behavior models.

```
class ClientNPC:ExternalNPC()
method handle_msg(s)
  write("tux received ", s)
end
method idlefunc()
  write("tux is snoozing")
  delay(1000)
end
method run(password)
  srvport := "virtual.cs.uidaho.edu:4500"
  homepage := "www.tux_homepage.com"
  login(password)
  mainloop()
  write("NPC loop finished, good bye")
end  # class ClientNPC

procedure main(arg)
   tux := ClientNPC()
   tux.run(arg[1])   # passing the password
end
```

Listing 8.1 A very simple and compact NPC client example

The source code for the NPC client itself is organized into several classes. The following is a list of the major classes along with a summary about each class:

ExternalNPC: Holds all the information about the NPC. It has methods for downloading and parsing the NPC profile, managing the connection with the CVE server, receiving server messages and giving proper responses and taking proper actions when asked to chat or to give quest activities.

AvatarData: Holds the NPC's knowledge of other users' avatars and their quests activities. The NPC uses this to be aware of other users' locations and take proper actions if any. Some NPCs for example might interact with other users if they come closer or go farther, and also avoid them if the NPC is set to move. The NPC uses this also to know what quests each specific user has already completed, quests that they can take and quests that they are currently working on.

Knowledge: A collection of knowledge categories.

KnowledgeCategory: A collection of quests that belongs to the same category.

Quest: Holds all of the information about a quest, such as readings, prerequisites, and steps. It has methods to read and parse a quest from a webpage or a local file and save it if necessary to a local file. The Quest class also keeps track of the quest activities like the current question the user is answering, quest score, and user's answers for different questions and so on.

8.4 NPCs in the CVE Environment

The CVE virtual environment is primitive, but its simplicity allows easy experimentation. The default CVE avatars are hardwired humanoid, but avatars can be created also from 3D models produced by tools such as 3D Studio Max and exported in Microsoft .x format. Using these tools to create 3D models requires expertise that most users and developers lack. Hence, there is a need to have artists who can use these tools to be part of the teams that develop virtual worlds. Many online sources sell readymade or custom models based on user's demand. For this dissertation, a few models from free online resources were used. Figure 8.7 shows some example avatar models in the CVE environment along with quest activities.

CVE features an integrated collaborative IDE that allows tutorial activities for a range of computing topics. Besides shell commands illustrated in the ls example earlier, these include editing, compilation, execution, debugging, and testing activities for C, C++, Java, and Unicon. CVE is covered in more detail in the following chapter.

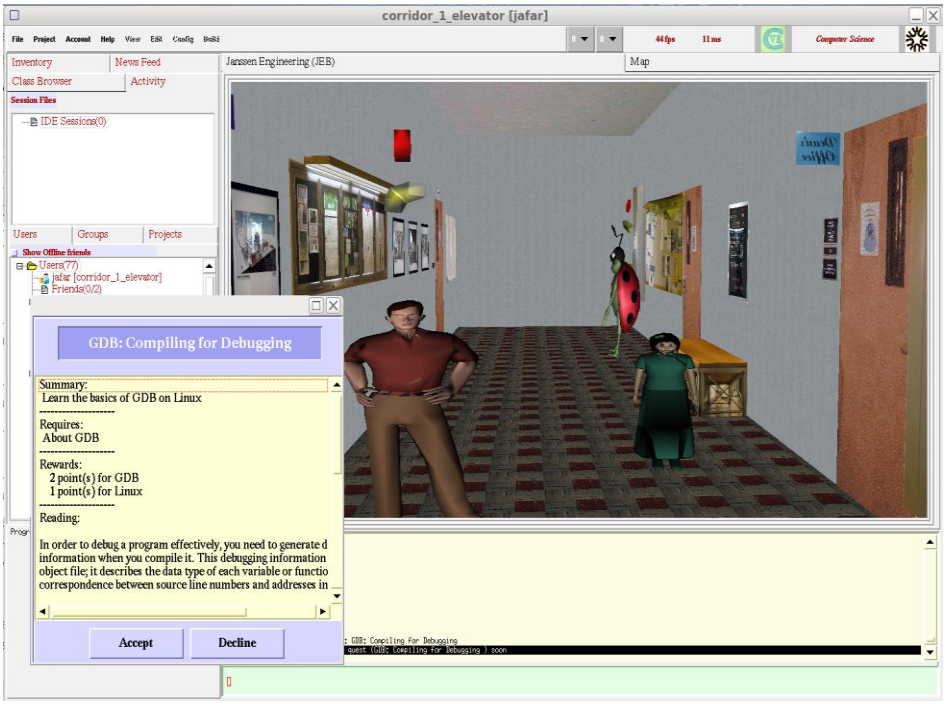

Figure 8.7 NPCs in CVE virtual world

8.5 NPCs as Threads

In most games, NPCs are typically run by the game server. NPCs are generally light tasks with very limited capabilities such that the server can handle a huge number of them. Having them as independent clients in such cases is a waste of resources, with the negative effect of putting extra pressure on the server taking resources that can be allocated to regular users. PENQ NPCs are designed to be able to run as standalone clients, independent from the server or a specific virtual world, but the encapsulated and independent design of the NPC coupled with the language features and extensions make it very easy to have this transition, from a process to a thread. The independent design ensures that the NPC is self-contained and does not share data directly with the server or with other NPCs which help eliminate any data race issues. And the language extensions, namely communication operators, make the communication mechanism between the server and the NPC fully transparent, whether the NPC runs as a thread or as separate client. Even within the separate client, the design allows separate NPCs to run as threads within one client.

All of the NPC running models share an identical code base except for a very thin layer at the top that does not exceed 30 lines of code that sets up the NPCs in the client of the server. This would not be possible without the communication operators in the language that abstract the communication between threads in the same process, or between threads in different processes through sockets. For example, NPCs send messages to the server via the @> operator in the form of:

```
x @> y
```

When the NPC runs as a separate client, y is initialized to refer to an open TCP socket. This causes x to be transmitted to the server via the TCP connection. However, when running inside the server, y is initialized to &null, causing the message x to be queued on the NPC thread, ready to be picked up by the server. y could refer to the server thread inside the server process but that is part of the implementation details. The server picks up the message by doing:

```
msg := <@client
```

With a TCP connection, client refers to an open connection. With an NPC thread it refers to a variable initialized as a reference to thread of the NPC. This allows communication with NPC threads or NPC independent clients via socket to be done transparently at both sides, the server and the NPC. The motivation behind supporting flexible running modes for the NPCs is to allow end users to create custom NPCs and run them as remote clients. However, when it comes to maintaining the CVE and ensuring that some built-in NPCs will always be online, having every NPCs run as a separate client puts extra work on the maintainer of the virtual environment and its server. When running the required NPCs in one process, they are easier to launch and also easier to shut down. When they are built-in to the server as threads, they become part of the routine of maintaining the server itself.

Another benefit of having NPCs as threads is the fact that threads are lighter than processes, allowing them to take less resources which increases the number of NPCs that can be run on a single machine. By running them inside the server, they put less burden on it by having faster communication and leaving more room for TCP connections from regular clients. Furthermore, using thread communication features between the server and NPC threads provides a convenient and implicit synchronization between the server and the NPCs by setting limits on the outbox queues of the NPCs, such that an NPC blocks for a short period of time whenever the queue is full. This ensures that the NPCs do not overload the server with communication messages acting like an implicit braking system for the NPCs, because the NPCs send the commands to the server using the blocking send operator @>>, which blocks the NPC if the message queue is full.

Figure 8.8 presents the results of running the NPCs in different modes: separate clients' processes, all running as threads in one process independent from the server, or all running as threads in the server process. The figure shows the latency of the server (the time it takes the server to reply to a message) increases when increasing the number of the NPCs connected to the server. All of the NPCs keep moving (every 10-20 millisecond), generating move messages that are sent to the server, and the server broadcasts these messages to all connected users and NPCs. The results of latency are measured in one of the NPCs over a period of two minutes, taking the average latency time during that period at the end. The experiment is repeated 5 times and the average is taken at the end.

The results show that when running the NPCS outside the server, the server can handle between 8 to 16 constantly moving NPCs. Adding more NPCs in those cases, floods the server with messages rendering it unusable. When running them as threads inside the server, the server spend a lot less time doing I/O to communicate with the NPCs. This improves the overall performance of the server allowing it to handle up to 64 NPCs. That is more than 4 times the number of NPCs the server can handle when they run as independent clients.

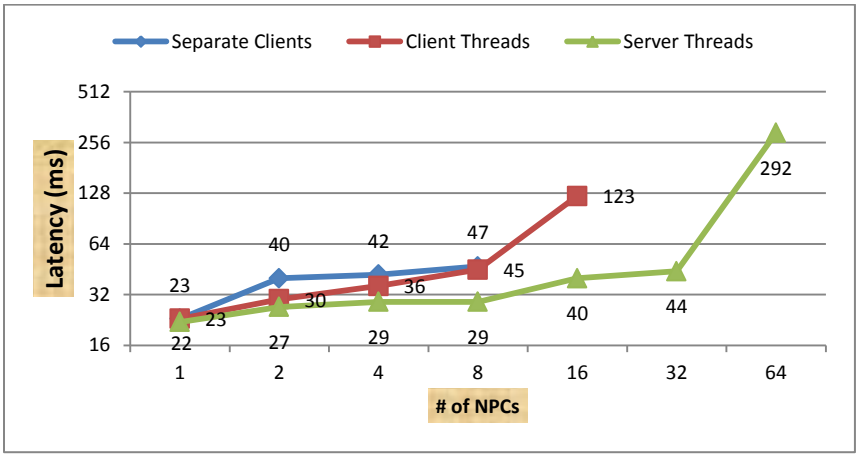

Figure 8.8 The effect the NPCs' running modes on the server latency.

9 CVE: A Collaborative Virtual Environment

CVE (http://cve.sourceforge.net/) is an educational platform that was built primarily to support two uses: (1) distance learning by college computer science students, and (2) software development and group collaboration. Figure 9.1 shows a screen shot from the CVE virtual world. The collaborative virtual environment provides developers with a general view of other users and what they are doing. It allows developers to chat with other team members and with developers from other teams in real time. It also allows users including developers, students and instructors to collaborate in code editing, compilation and debugging (Figure 9.2).

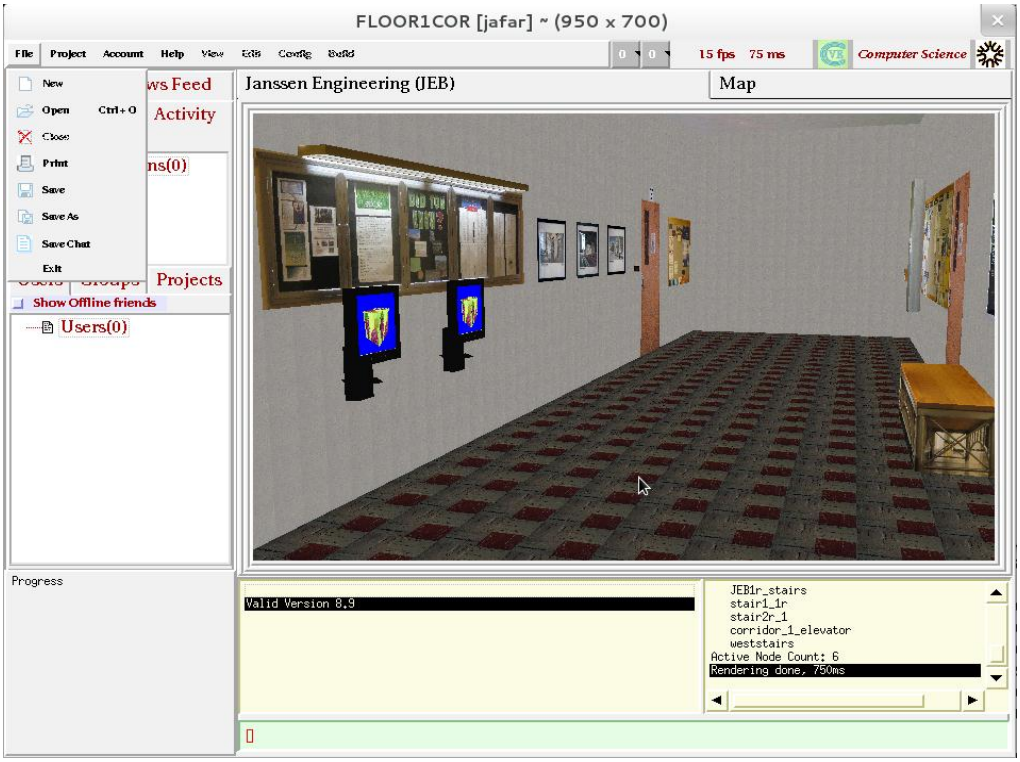

Figure 9.1 A screen shot of the CVE client

This chapter focuses on the interactions between the application (CVE) and the programming language (Unicon) and how they affected each other during the lifetime of this dissertation, with CVE triggering new language features to better facilitate CVE development and support new functionalities. The goals are:

- Demonstrate the aspects of the co-design approach that help shaped the application and also the language and its new features
- Evaluate the new language additions and features as used in the virtual world

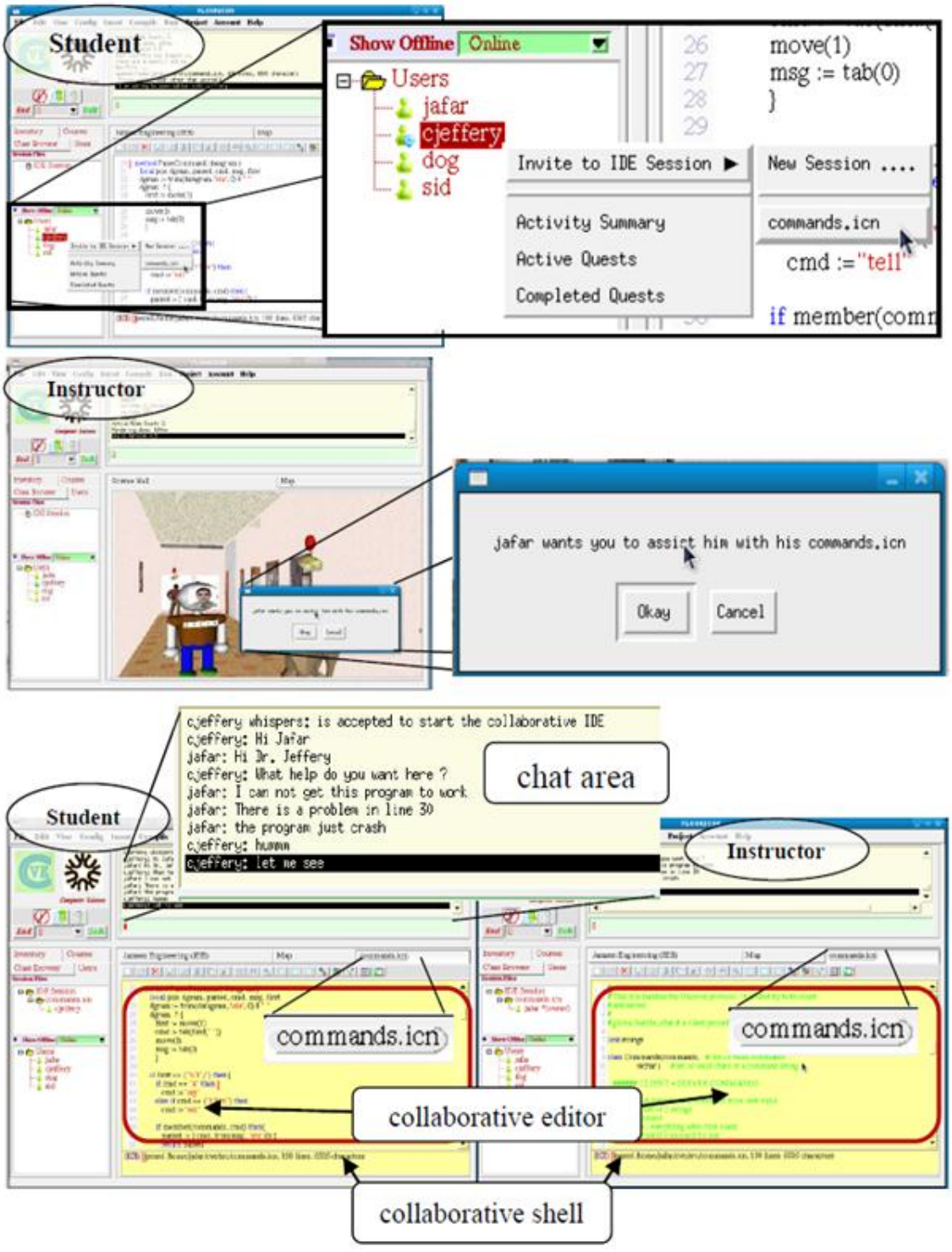

Figure 9.2 Starting a collaborative IDE session in CVE

9.1 Design

Language design was discussed in part II of this dissertation. This discussion of application co-design focuses on features and class libraries added to both the language and the application. When looking at CVE from a design point of view, it consists of three major layers:

1. An application layer
2. A class library
3. The language layer

While the language is not part of the application (CVE in this case), it is listed here as the bottom layer to emphasize the fact that the addition of many new language features was driven by the application requirements. Architecturally in many aspects, the language serves the role of a game engine for the application. The language plays a major role in shaping the design and implementation of the virtual world. As discussed in earlier chapters of this dissertation, the language was augmented with 3D graphics, object selection, concurrent threads, network, and audio API's to facilitate the building of such worlds.

In the CVE middle layer lies an application class library, in addition to the language class libraries that were added to support new features required by the virtual world. The application library provides infrastructure for the networked 3D environment, e.g. the behavior of doors, whiteboards, and avatars.

The top layer is the application layer for CVE which consists of: a 3D model of a virtual world produced semi-automatically using CVE builder tools; a set of domain collaboration tools; and a set of user accounts, created on the CVE server. In addition to that, new NPCs and quests can be added over time create new activities. On the other side, from a functional point of view, the CVE also has three major components:

1. A virtual space that users can explore and also meet and chat
2. A social collaborative IDE subsystem called SCI where users can write and share code and see each other's activities
3. Non-player characters (NPCs) and a quest system that users can interact with in order to go on (educational) quests

The following section covers the evaluation of the features that were added to the language while developing CVE. It also presents examples of where the co-design approach was very apparent when developing both the application and the language.

9.2 Language Features Developed for CVE

This section demonstrates the use of several new Unicon language features in CVE. It also provides an evaluation for many of these features and other various improvements and additions to the language and their effect of performance, usability and ease of use.

In real time virtual world applications such as CVE, performance is critical. A 30+ FPS is preferred, but lower numbers in the range of 15-20 FPS can be tolerated especially in a virtual world, where the purpose of the environment is not to deliver high end realistic 3D graphics but education and collaboration opportunities.

9.2.1 3D Models

One of the challenges in building a 3D virtual world is to populate it with 3D content. Graphics content can be hardcoded into the virtual world; this can be done for content that is static and simple enough that it is feasible to achieve through coding. For contents that change over time or require a high level of art, coding is not an option. Hardcoded objects need to be recoded anytime a change is needed and such changes necessitate rebuilding the application.

Most virtual worlds rely on data files and 3D model files to store the content of the world. Such models can be acquired from third parties or created using applications such as 3D Studio Max or Blender. The challenge lies in loading and manipulating such models. If programmers need to read a specific file type, they will have to find libraries or write code to support that particular file type.

A class library was added to Unicon to support two 3D model formats, the Simple 3D and Microsoft .x formats. The class library also supports terrain generation and rendering using certain terrain data file formats. This enables Unicon programmers to build richer worlds without worrying about the file format, how to load it, or how to render and animate it in case it contains animation data. S3D and Microsoft .x formats were picked because they are open, human-readable text file formats.

A simple tool was also developed to load and view 3D model files to let the programmers preview the rendered models and look up all of the information about the model, such as vertex and polygon counts, whether the model contains animation data and so forth. The tool also supports playing the animations in the model file to let the programmer preview the animations in real time. The information that tool provides can help the programmer not only to be aware of the model information but also better plan how much memory the model will require and how much time it takes to render. Figure 9.3 is a screen shot of Unicon 3D Model Viewer.

3D model support using a class library provides a convenient way for programmers to load and render 3D models without worrying about the details. The library uses some features such as thread support for some of the functionality as discussed in the coming sections. One important aspect about using a class library is that it represents a way to encapsulate features in the language. When a class library is not sufficient, features are integrated into the language. In the case of 3D models, a class library provided the required features, but the performance was not high enough which triggered the addition of new features that the class library builds on, such as the addition of arrays support discussed in the following section.

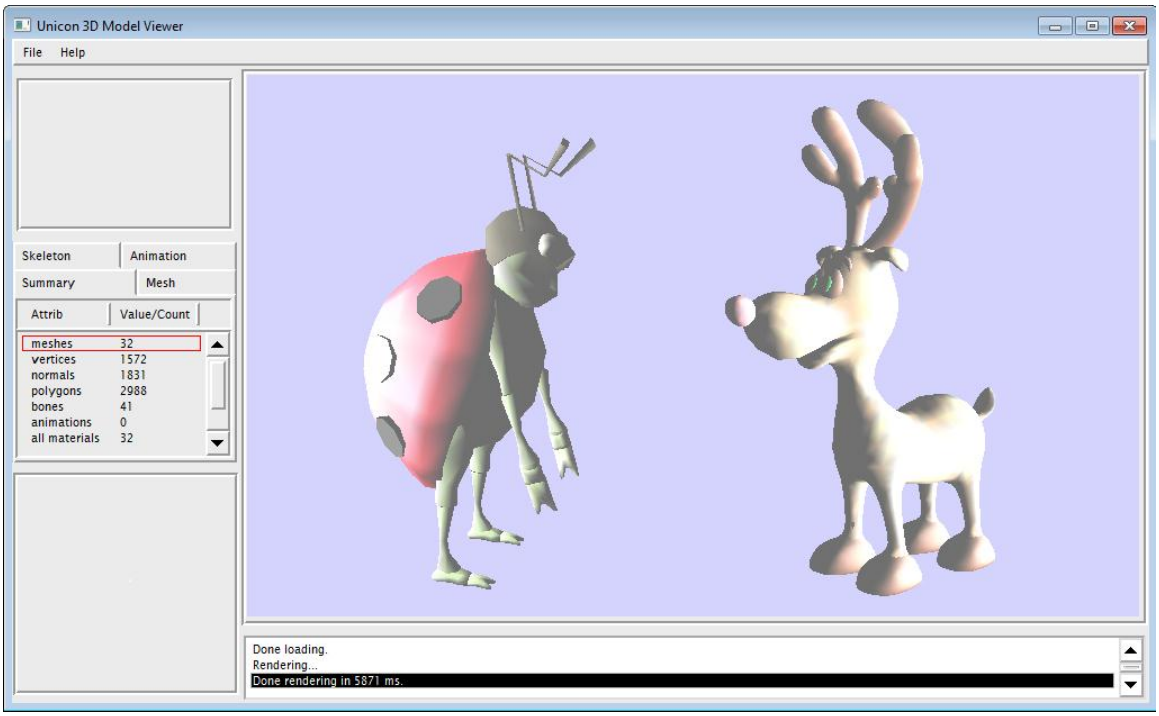

Figure 9.3 Unicon 3D model viewer

9.2.2 Arrays as Lists

In some cases, additions or improvements to the language were not a direct product of a feature addition required by the virtual world, but rather driven by other aspects such as improving the 3D graphics performance. In the early stages of developing CVE, the list data type used to store a lot of the virtual world data. In later stages, especially after adding the data intensive 3D models, it was clear that smooth animation cannot be achieved with most of the data being converted continuously from lists to arrays. By supporting array representation for the list data type in Unicon; this continuous conversion is avoided. While very simple scenes got only a modest increase in performance from this improvement, for scenes that have 3D models, the performance was tremendously improved as presented in section 7.3. In this

section, the effect of these improvements is measured within a context of a virtual environment. Figure 9.4 shows a 3D model for a warrior with animation for a walk cycle loaded from a .x file with a 528 polygon, 428 texture coordinate, and a 256x256 texture. The model can play few animations including things like walk, fight, idle, and a few more. This model is used for avatars in the CVE for testing purposes. The results of these experiments are shown in discussed throughout this section. All of these experiments were conducted on the same machine described in Table 7.3.

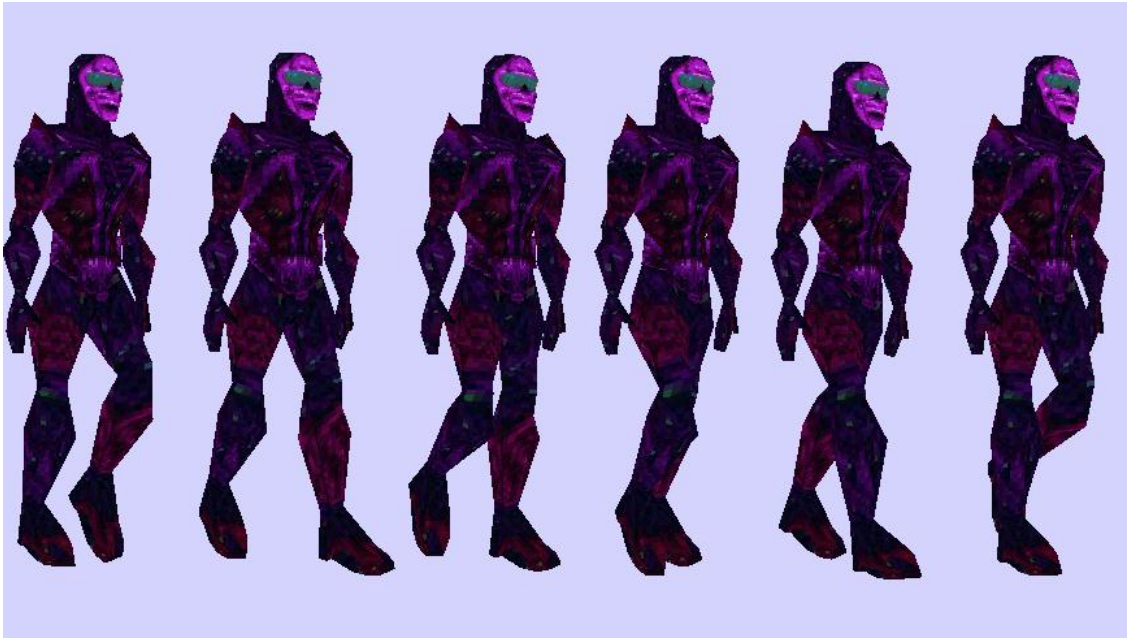

Figure 9.4 An example 3D model with an animation of a walk cycle

The first of these experiments demonstrate the effect of using lists vs. arrays in Unicon on CVE performance. The CVE was run with different number of NPCs logged in as shown in Figure 9.5 and the frames per second were recorded. The duration of each run lasted for about 2 minutes with all of the NPCs constantly moving back and forth in the scene. The final frames per second results presented here are the averages of repeating the test five times. The use of NPCs only and not human users was made for practical reasons. Both NPCs and human users are handled in the same way in the CVE and should not affect the accuracy of the outcome of the experiment, especially that rendering and animation of 3D models is identical for all whether they are regular users or NPCs.

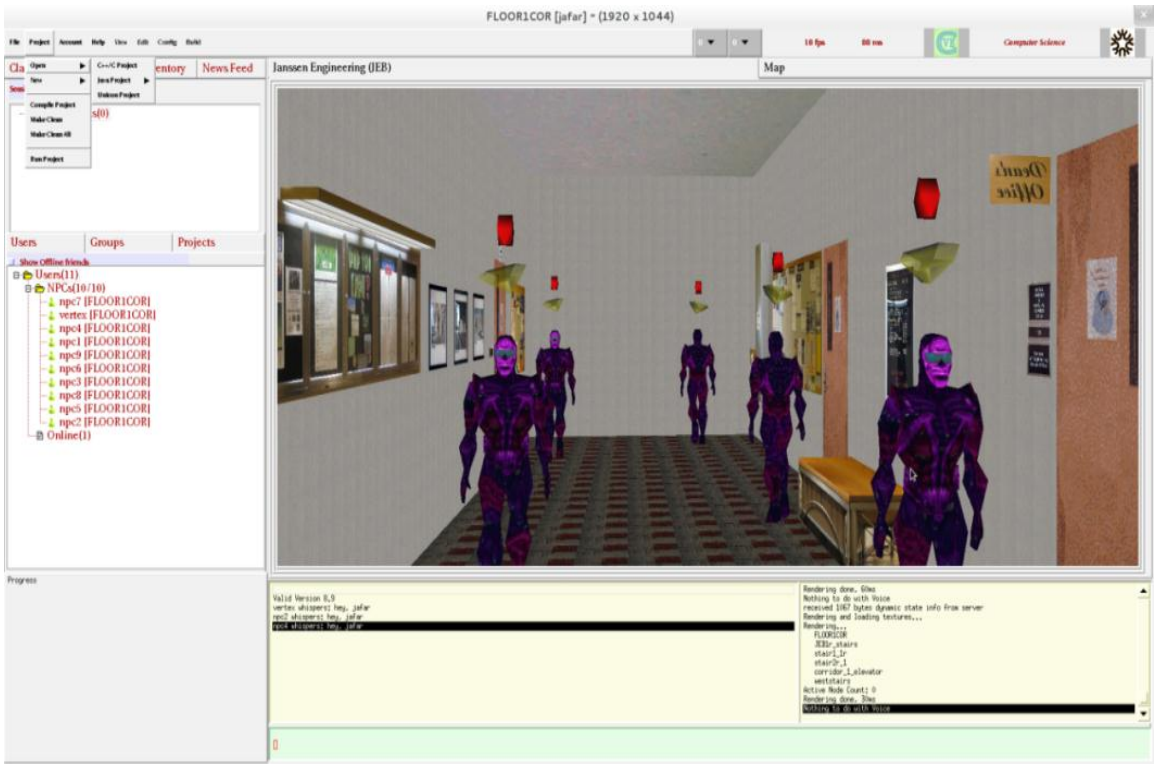

Figure 9.5 A number of NPCs walking around in CVE

Figure 9.6 shows the result of the experiments with improved performance advantage for arrays over lists especially when increasing the number of users in the scene. There is little advantage with new users or with few users because the CVE world itself dominates the rendering in that case, not the users' 3D models. The CVE world model is primarily built using primitives such as rectangles with few data points. The primitives are scattered across a big number of calls to Polygon(), making the move to arrays not as effective because there is a very small amount of data to begin with. When adding more users where thousands of vertices are piled up in a single array or list, the payoff of the arrays representation starts to show up more clearly, reaching almost 3x speed up in the case of 16 users. The number 8 is still low for FPS but it is a lot better than 3 FPS in the case of using lists. This improvement comes to CVE at no cost without any code modification. The improvements are done in the language, so CVE gets it for free.

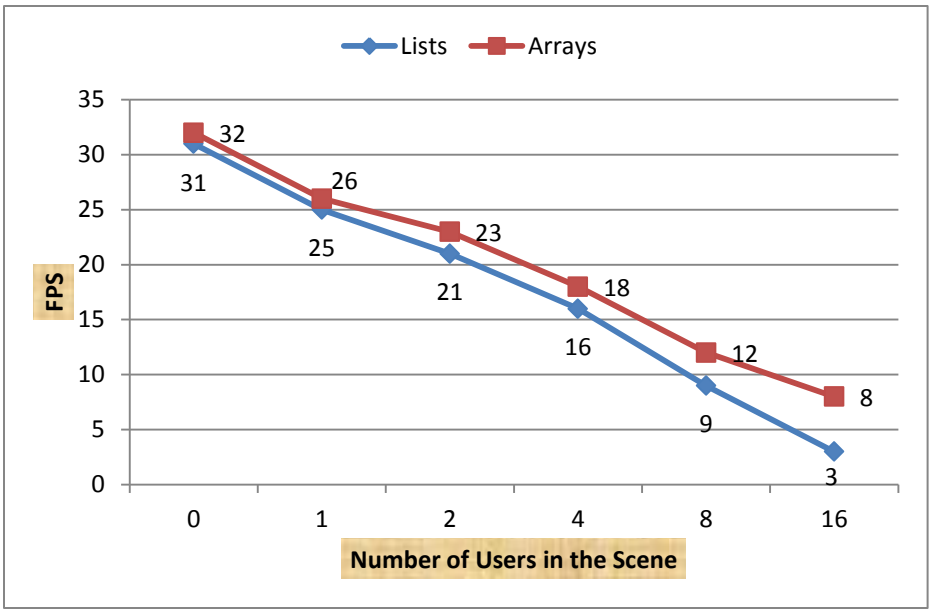

Figure 9.6 CVE performance using arrays and lists

To understand why users' 3D models have this big impact on performance, refer to section 3.2 about 3D graphics in Unicon. A 3D window in Unicon maintains a display list of all rendered objects in scene. When doing animation, the data in the display list is updated and the scene is redrawn by calling the function Refresh(). A continuously changing world in the case of CVE requires continuous calls to Refresh() to maintain smooth animation. The number of calls to Refresh() that can be made in a single second dictates the frames per second (FPS) that can be achieved by the application. Any long operations outside Refresh() means a drop in FPS resulting in a jerky animation. With multiple animated 3D models in the scene, the CVE client has to update the model's vertex data using the animation key frames that are part of the model. The operation includes a huge number of matrix multiplications and vertex transformations. The result is a very computationally intensive operation that leaves very little time for refreshing the screen. Figure 9.7 shows the percentage of time spent in the function Refresh() in the case if lists and arrays. The remaining percentage represent the time the client spend doing other tasks such as reading network messages and also the expensive operation of updating 3D models animation information. A solution for this problem will be presented in the thread section ahead.

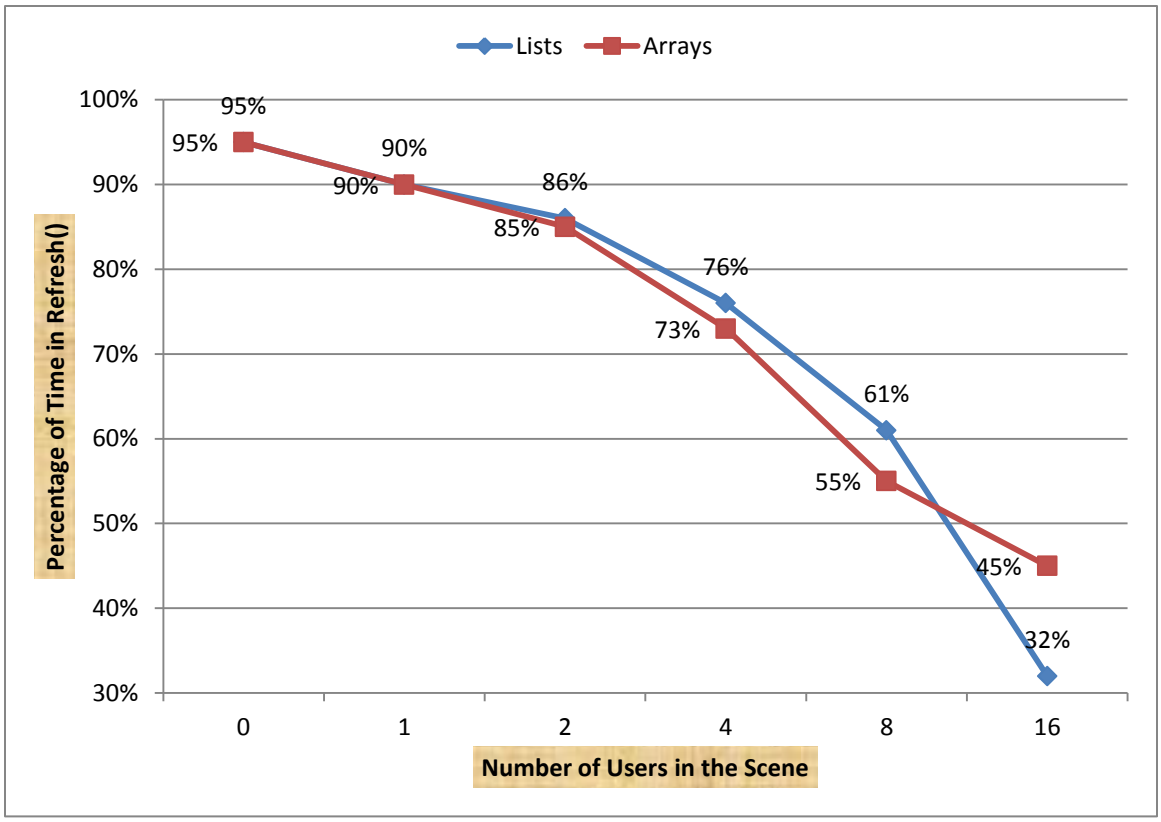

Figure 9.7 The percentage of time spent in the function Refresh()

Figure 9.8 summarizes information from Figure 9.6 and Figure 9.7. The FPS depends on how long Refresh() takes and also how long it takes to finish the work outside Refresh() which is dominated by key frame animation updates. Using arrays, the FPS manages to stay higher since Refresh() takes less time. The rendering with arrays is faster which frees more time to do animation updates outside Refresh(). With a high number of 3D models in the scene, the performance with lists drops dramatically to the low 3 FPS value. The application spends most of the time updating the animating models leaving very little time to do Refresh(), which itself takes a lot more time with more 3D models in the scene. With the faster arrays, the application manages to do more of the less demanding Refresh(), and thus maintains a higher frame rate.

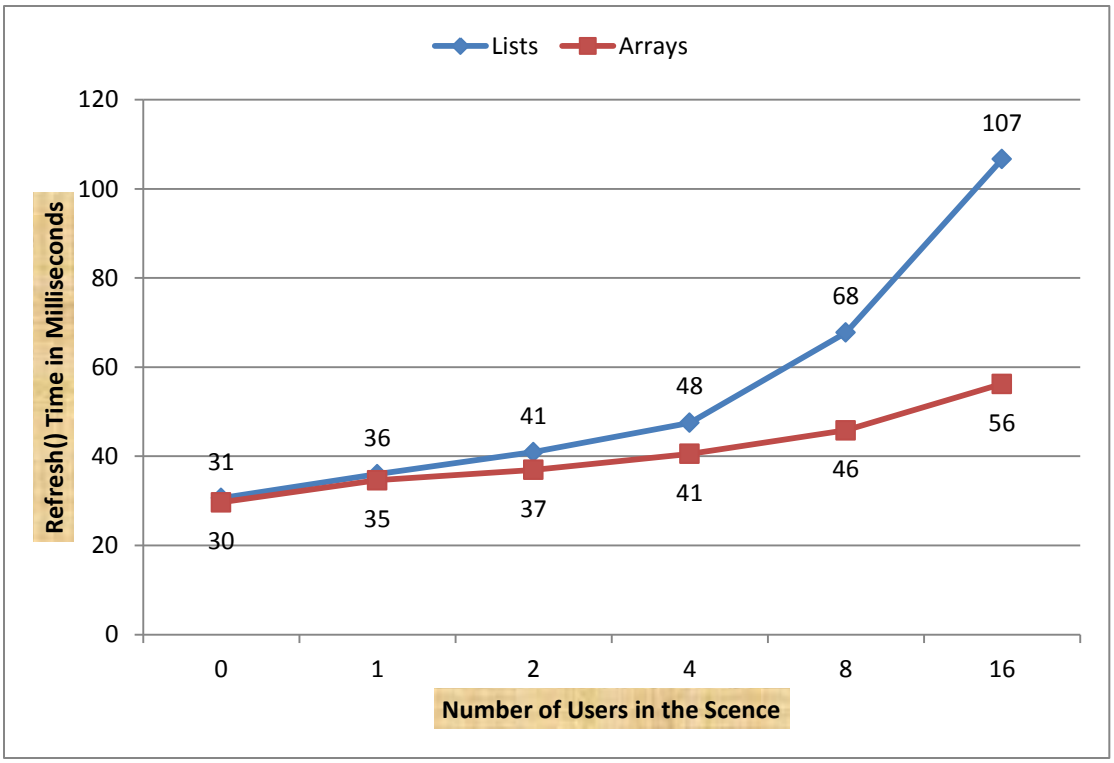

Figure 9.8 The time it takes to do one Refresh()

9.2.3 Selectable objects

Two main uses of the 3D object selection are already incorporated into the CVE. These two use case scenarios were the driving force behind adding this feature into the language. The two selectable objects in the virtual world until now are avatars and doors. The examples presented here demonstrate the use of 3D selection in CVE and the simplicity of adding new CVE functionalities on top of this new feature. The new feature not only has a minimal API requiring minimal changes to CVE to get 3D object selection to work, but in many cases fits with the design of CVE such as the doors example presented below. The 3D selection feature is evaluated qualitatively.

All doors in CVE are given unique string names. A door name starts with the word “door” followed by an integer number representing a unique identifier. This scheme was used prior to introducing 3D object selection, and it was a perfect fit for what 3D selection needed in order to work. When rendering a door, the code in CVE looks like the following:

```
WSection(&window, "door" || id)
   #  the actual door rendering goes here
WSection()
```

If 3D selection is turned on; doors will be on the list of selectable objects. Since door ids begin with the word “door”, a string scan of the output of &pick will tell if the selectable object is a door or not. If it is a door, scanning through the digits after the word “door” will reveal the door integer id. The code in CVE that is added to click events to capture selected doors is approximated by the following code:

```
 if picked_object := &pick then {
   picked_object ? if tab(find("door")+4) then
                picked_door_id := integer(tab(many(&digits)))
   }
```

Getting 3D selection to work with doors requires very little work from the programmer. No setup code, no changes on how to render doors, and no changes on how to handle their behavior. The only changes in this case is a new if statement to collect a picked object, if any, and then check if it is a door before identifying which door was picked. The mechanics of the selection is done and hidden by the language.

A more powerful use of selectable objects in CVE utilizes the event-driven interface for the 3D selection. The following code demonstrates the use of the selection class to register the avatar for selection to respond to a left mouse-click with the event handler on3d_avatar and to a right mouse-click with the event handler on3d_avatar_right.

```
select_id:=select3D.selectable("avatar:"||a_name, "on3d_avatar", select3D.LEFT_CLICK, self)
select3D.add_action(select_id, "on3d_avatar_right",  select3D.RIGHT_CLICK, self)
WSection(select_id)
        # actual render of the avatar goes here
WSection()
```

The programmer has only to specify what is to be done (an event handler), and when it should be done (on mouse clicks left or right). No other setup or rendering details is involved or needed. The following code fragment lists the two handlers specified in the code above:

```
method on3d_avatar()
  world.nsh_dialog.write_to_chat_win("This is: ", image(a_name) )
end

method on3d_avatar_right()
    pop3D.add_menu_item("Tell "  || a_name || " Hello", self, "on_menu_tell")
    if is_afk() then  pop3D.add_menu_item("Away for " ||m||":"||s )
    pop3D.add_separator()
    pop3D.add_menu_item("History ", self, "on_menu_history")
    pop3D.add_menu_item("Active Quests ", self, "on_menu_active_quests")
    pop3D.add_menu_item("Completed Quests ", self, "on_menu_completed_quests")
end
```

The first handler only echoes the avatar name to the chat window whenever it is left-clicked. The second handler is a lot more powerful. It pops up a menu (pop3D) in the 3D window allowing the user to select different actions; each one has it is own corresponding handler. pop3D is constructed on the fly and can be

customized for each avatar, and also for specific NPCs allowing great freedom on how the event will be handled. Adding the support for popup menus in 3D windows to CVE is part of this dissertation work. Popup menus in 3D windows was not something planned for, but was triggered by the addition of the 3D object selection to the language, which in turns was triggered by the requirements of CVE as mentioned earlier. This is a very clear example on how the application and the language evolved together in a co-design fashion, each one triggers new features and new use cases in the other one as shown in Figure 9.9.

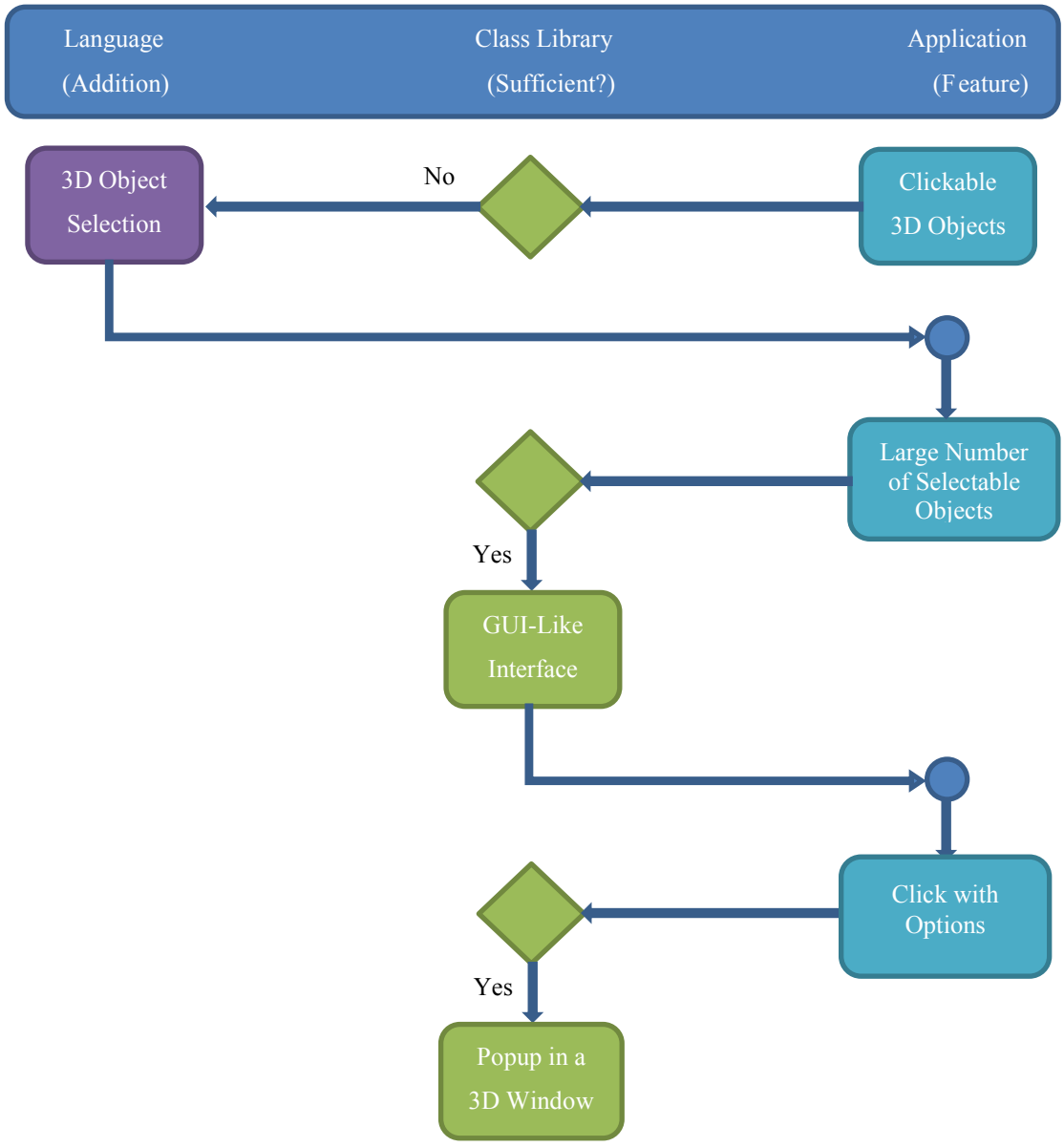

Figure 9.9 Feature evolution of 3D object selection in CVE: language-application co-design example

9.2.4 Visualizations via Dynamic Textures

The addition of dynamic textures to the Unicon Language was covered in sections 4.3.1 and 7.2. The integration of this feature into CVE is still in an early stage and has not made it to an official CVE release. But the prototype is functional enough to allow evaluation. Dynamic textures enable users to share visualizations in the virtual world. This kind of visualization requires more than dynamic texture to have it fully functioning in CVE. It requires a network protocol necessary to transmit images and/or drawing commands over the network to other users. This section provides a brief introduction on using this feature in CVE with demonstrations and examples.

As explained earlier, dynamic textures can be used to create different functionalities. In CVE they can be used to create visual effects such as animated textures, but the most important role is to use them to create virtual whiteboards and virtual computer screens. Updated images and drawing commands can be transmitted to different users to play the same animation on different clients. This falls outside the scope of this dissertation. What this dissertation provides in this regard is language support for dynamic textures, combined with other language improvements and features that facilitate such support. This includes better image files and thread support. Threads are essentials in performance critical situations such as updating a texture or animation in the background. An example of such use of threads is presented in the following section.

To demonstrate the use of dynamic textures in CVE, a simple example is presented here. A virtual computer in the CVE world uses a dynamic texture in rendering its virtual display. The texture is updated whenever needed by copying the content of a window with 3D graphics directly. The window is shown in Figure 9.10. The cube in the window spins creating a simple animation. The result of this animation combined with the dynamic texture of the virtual computer screen is a live animation in the virtual world as shown in Figure 9.11.

Figure 9.10 A window with a 3D spinning cube used as a source for a dynamic texture in CVE

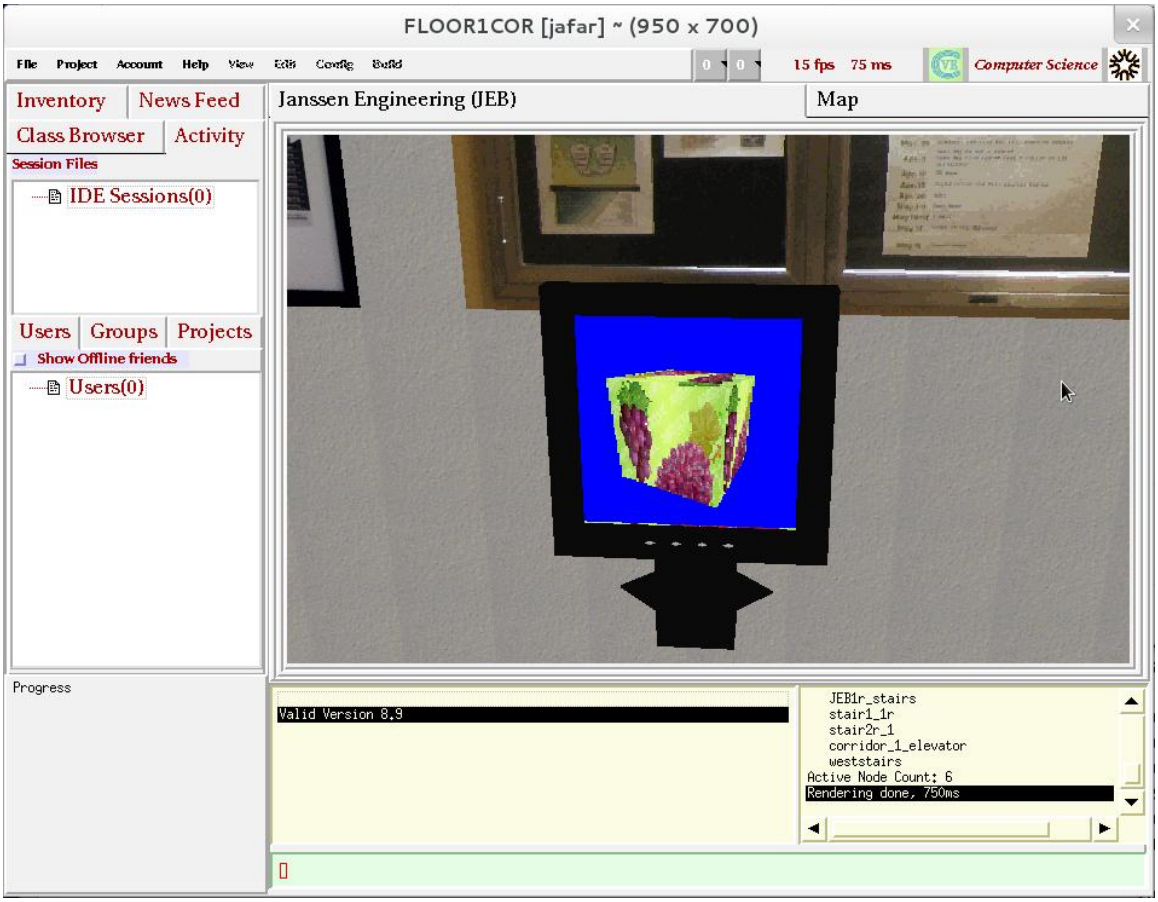

Figure 9.11 A virtual computer uses a dynamic texture showing a 3D spinning cube in the CVE virtual world

The dynamic texture can be shared among different objects, computer screens in this case to play the animation in different places to different users in the virtual world. This adds no performance penalty since the texture is only updated once, but the animation is reflected in different places in the environment. This is another attractive aspect of dynamic textures, creating a visual effect that can be shared among many objects, as many as the programmer wants as seen in Figure 9.12, at a constant cost.

Using dynamic textures in CVE is straightforward. A dynamic texture resembles a window making the API easy to use so the programmers do not have to learn a new API. Updating a texture in most cases involve only one function call such as the examples presented above where CopyArea() is utilized to do the task of copying the window's content into the texture. This allows the CVE developers to write compact code and focus on what to use this feature for in CVE, rather than how to do it.

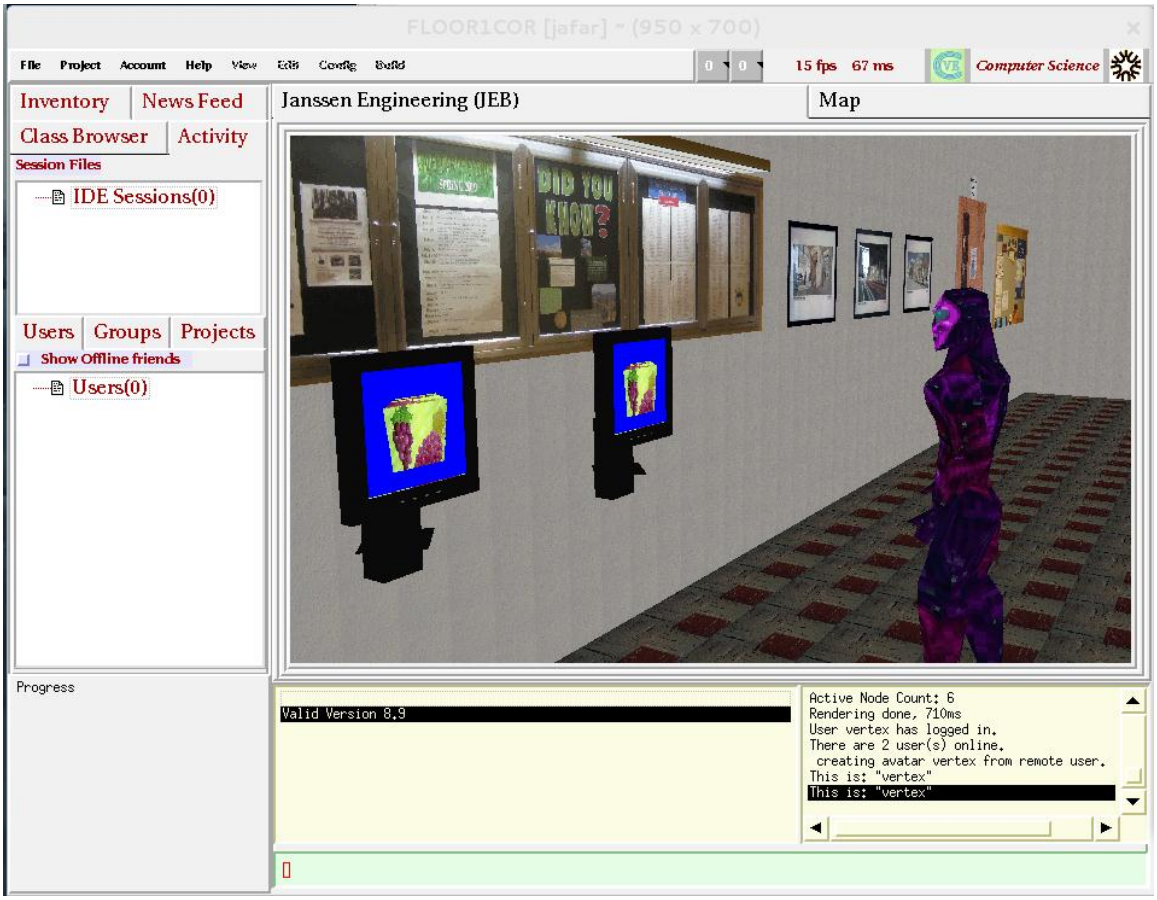

Figure 9.12 A user in CVE watching two virtual computer screens sharing the same dynamic texture

9.2.5 Concurrent threads

Concurrent threads provide a great opportunity to improve the CVE design and performance on both the server and the client sides. The server extensions to support threaded NPCs were covered in section 8.5. The extensions to the client to make use of threads are covered in the following subsections.

9.2.5.1 Multithreading the CVE Client

One way to improve the client performance of CVE was to use arrays instead of lists to represent data especially in the case of 3D models. While there was a big performance improvement when multiple users and NPCs were logged in with 3D models for avatars, the performance still did not scale well. In this section, a multithreaded approach is taken to scale the performance of the client up with an increased number of concurrent online users.

The challenge with animated 3D models is that they require a huge amount of computation to calculate the coordinates of almost every vertex in the model every frame. This has to be done while keeping the frame rate high enough. The tests show that a single thread fails to deliver enough power to maintain an acceptable experience which includes high frame rate and low response time. This means a frame rate of more than 3 or 8 achieved using lists or arrays as shown in section 9.2.2.

The key idea in utilizing threads in the client makes use of the fact that calculating new animation frames of a 3D model can be done independently from the main thread. When the main thread is busy refreshing the screen, other threads can calculate all of the required key frames of the animated object in the scene and have them ready for the main thread to pick them without any waiting time. This process off loads intensive computation from the main thread allowing it to do more screen refreshes and deliver higher frame rate as shown in Figure 9.13. The threads case in the figure builds on top of the array performance.

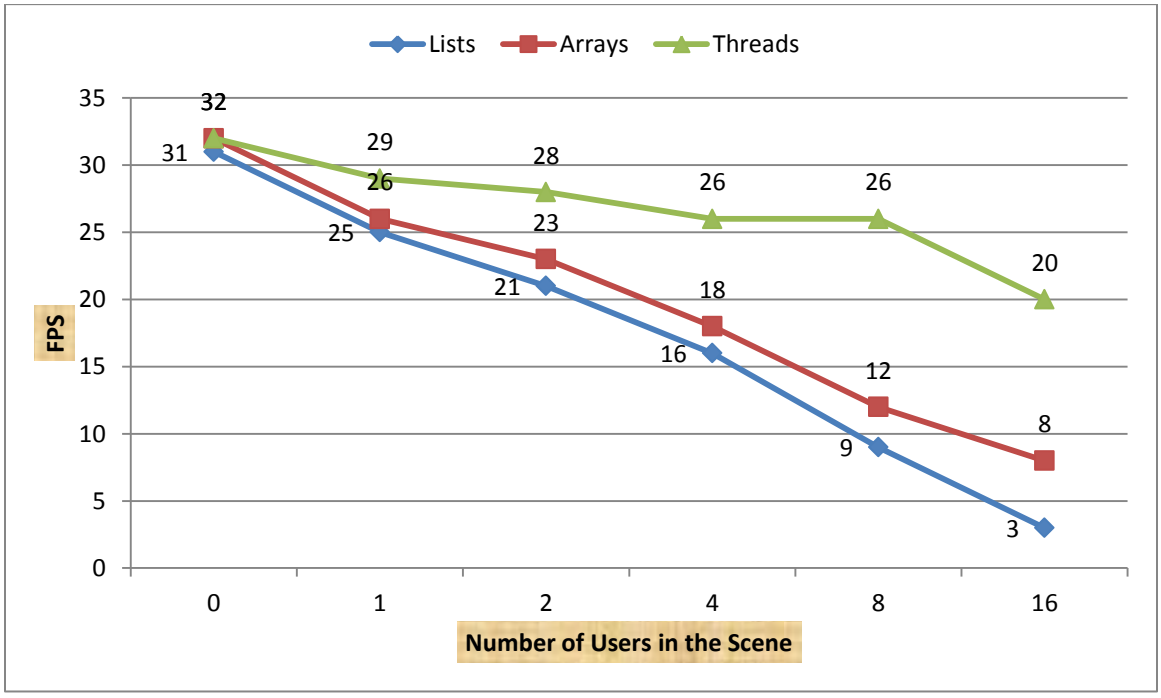

Figure 9.13 The benefits of adding thread support to the CVE client

Thread support is built in the 3D graphics package (graphics3d) in Unicon, which means the CVE client is not affected at all at the source level except for a single line that calls a method telling the 3D model to enable animation computation via threads:

```
model3d.start_animation_thread()
```

Because of the high level thread features Unicon has as a result of this dissertation, the package itself saw minor code updates to support threads changing less than 20 lines of code. The changes are mainly

reorganizing some code and controlling few state variables, in addition to launching a new thread. The method most affected is calc_animation() which is part of 3D model class, and responsible for calculating new animation frames. In the threaded version when the method is called by the main thread, instead of calculating a new frame, the method becomes a swap operation swapping a frame prepared by the thread into the display list and signaling the thread to work on preparing a new frame for the next time around. Because the main thread has to do a lot of work including refreshing the screen making it a lot slower than the thread responsible to calculate the animation frame, the animation frame in most cases will be ready for swapping with no delay. In rare cases where the frame is not ready, the main thread simply skips the swap and continues without affecting the animation which further guarantees that the main thread does not spend long time in operations outside Refresh().

The advantage of using threads is very obvious from the figure. At 16 users, the performance of threaded client is 2.5x of that of arrays alone on the quad core laptop described in Table 7.3, and close to 6.7x that of the single threaded list implementation used before this dissertation work. The figure shows that 20+ FPS is achieved with only two users online in the case of arrays or lists, but with threads 20 FPS is achieved even with 16 users online. Figure 9.14 compares the percentage of execution time spent in Refresh() in the main thread with that of the lists and arrays.

The figure confirms the earlier claim that the main thread has more time to spend in refreshing the screen. In all cases except with 16 users, the main thread spends almost the same percentage of time in Refresh(), and the exact same time compared with arrays as seen in Figure 9.15. The difference from the arrays case, is that the main thread does not do intensive computation outside Refresh(). As mentioned earlier, that is because the main thread is no longer responsible for recalculating the animation key frames. The 11% drop at 16 users can be attributed to the increase of time it takes to do garbage collection and more importantly the increased network traffic that is handled by the main thread due to the increased number of concurrent users. The effect of garbage collection and its frequency on a real time application like CVE is discussed in the following section.

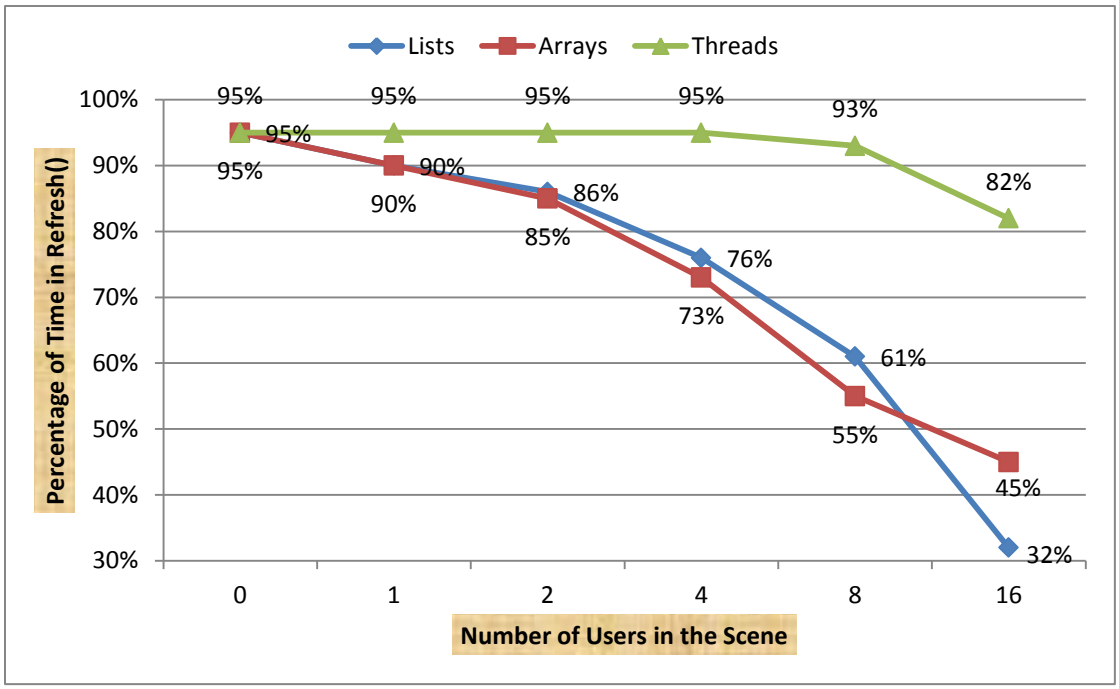

Figure 9.14 The percentage of execution time spent in Refresh() fucntion

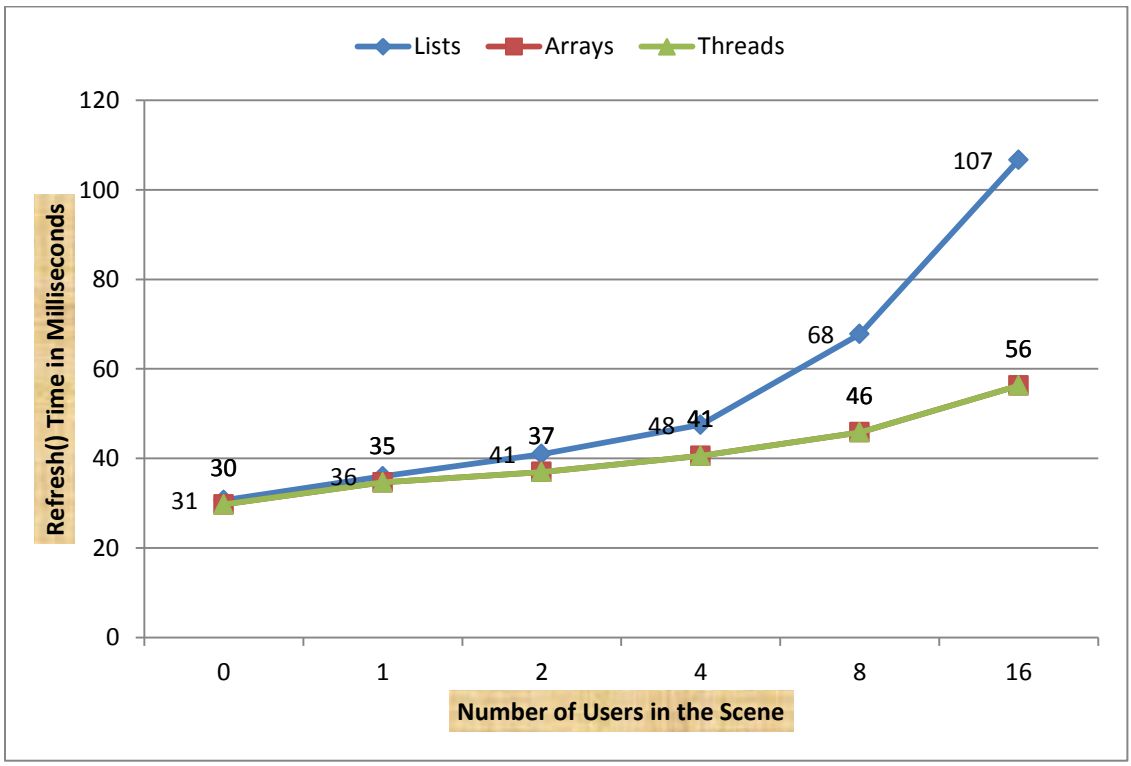

Figure 9.15 The time to do a single Refresh()

9.2.5.2 Garbage Collection and Multi-Threaded CVE

Garbage collection is a process that a long running application goes through periodically. In a real time application like CVE, the duration and the frequency of garbage collection is a serious issue. A real time virtual world that hangs every few seconds would be an unpleasant experience for its users. This section addresses this concern, showing the effect of garbage collection on CVE, especially in a multi-threaded environment, where garbage collection might take more time because of the synchronization required in suspending all threads to do a garbage collection, as explained in section 6.3.4.

Table 9.1 Garbage collection duration and frequency and its impact on CVE with different heap size while having seven online NPCs.

Heap Size / Main Thread (MB)	Heap Size / Thread (MB)	FPS	Worst 5 Seconds- Average FPS	Time Between GCs (seconds)	GC duration (ms)	Thread Suspension Time (ms)
1 Thread (Main)						
128	NA	12	9	22	95	NA
256	NA	12	10	48	140	NA
512	NA	12	10	95	230	NA
1024	NA	12	10	198	405	NA
7 Threads / Default Thread Heap Size (10% of Main's Heap Size)						
128	12.8	24	13	6	220	38
256	25.6	24	17	12	236	35
512	51.2	25	18	20	297	34
1024	102.4	25	19	38	327	31
7 Threads / Custom Thread Heap Size						
512	32	24	17	18	268	35
512	64	25	18	28	326	37
512	128	25	19	50	400	35
512	256	25	19	129	712	32

Table 9.1 presents different statistics about the CVE when running with different heap sizes, and also when running with one thread only vs. seven threads. The FPS refers to the steady frame rate that CVE maintains when running for a long time. The FPS rate is calculated at 5 second intervals. The fourth column refers to the worst FPS recorded in these 5 second intervals. As expected, increasing the heap size reduces the

frequency of the garbage collection, but it also increases the duration of the garbage collection. In most experiments, especially with large heaps, garbage collection is so infrequent that it is not statistically significant. In the case of having the main thread only, the relation between the heap size, garbage collection frequency and duration is predictable. Doubling the heap size cuts the garbage frequency in half, and increases the duration of the garbage collection by a factor of one and a half to two times, and since there are no other threads running, there is no thread suspension (the time it takes to stop all threads before proceeding with garbage collection) time. Having small heaps causes garbage collection to happen more frequently and in some cases it might result in thrashing where the application spends most of its time doing garbage collection. Having large enough heaps helps prevent such a situation, but it has the tradeoff of very long garbage collections causing the application to freeze for a short period, close to half a second with 1GB heaps. Because of this problem, it is not recommended to have very large heaps, especially since the benefits of having larger heaps diminishes beyond a certain point. For CVE with one thread, a 256MB to 512MB heap size delivers an optimal performance without the penalty of having very long garbage collections.

In a multi-threaded CVE, the picture is not that different on how heap size affects garbage collection behavior. However there are more variables added to the mix, creating a more dynamic environment. The frame rate is a lot better than the single threaded case, even in worst case scenarios and more frequent garbage collection. With multiple threads running there are more garbage collections and there is also a price for suspending all running threads. The default heap size at 51.2MB seems to deliver a consistent performance and below a third of a second garbage collection duration that takes place every 20 seconds or so. Having custom values of 64 MB to 128MB deliver a similar and more consistent experience with less garbage collections, but at the upper ends the risk of relatively long freezes (400ms) increases. Nevertheless, having a third of a second freeze every minute or so would pass in many cases unnoticed. Thread suspension time in CVE is not typical in non 3D graphics programs where it takes less than one millisecond only to stop all threads. As shown in the previous section, the main thread spends most of its time in Refresh(), which is a long operation and not interruptible by calls to garbage collection thread suspension. This means that almost in all cases, it is very likely that the main thread is the last thread to suspend and after a long waiting time, 30-40ms in these tests. This seems to be a problem but it is not, since the main thread is the most demanding thread and most other threads finish their work quickly every frame and spend most of their time waiting for the main thread anyway.

Part IV Conclusions and Future Work

10 Conclusions

Collaborative virtual worlds are complex and large programs requiring a variety of programming activities. Using system programming languages such as C makes matters worse because of the low level nature of C compared to Unicon. Two main reasons deter developers from using very high level languages to build virtual environment applications: poor performance and lack of features necessary to build such applications.

This dissertation addressed the complexity of virtual world systems using a very high level language. A co-design approach was adopted where the CVE virtual world and its implementation language, Unicon, evolved together identifying the limitations of the language in this domain. The dissertation overcomes these limitations by a novel design of new features built into the language and a set of class libraries that builds on top of the new features to further abstract many of these features. The effectiveness of this approach was tested by program examples and use in the development of CVE and its NPCs demonstrating the ease of use of the new features and also presenting performance improvement achieved with the new features. The following sections provide conclusions to specific areas of the research conducted as part of this dissertation. A summary of the contributions of this dissertation is presented in section 1.3.

10.1 3D Graphics

Designing and building a very high level 3D graphics API is only one side of the story, it is only good if it serves its purpose and meets the requirements of the applications. Unicon provides a very easy to use and expressive 3D graphics API, but big projects such as CVE demanded more in terms of features and performance. This dissertation addresses many of these requirements and completes the features set in Unicon to meet the requirements of heavy and complex 3D graphics applications such as CVE, without compromising the flexibility and dynamic flavor of high level language.

Many new improvements were made to the language's built-in graphics engine both to introduce new features and also to improve the performance of many existing features. These improvements include the addition or extension of several functions to support new functionalities or make them more efficient, better image file format support, efficient representation of some data types, and the introduction of dynamic textures. In some cases iterated revisions of the design and implementation of features had to be made at the language level and also at the application level to meet the performance and functional requirements of the application.

While the focus of this portion of the dissertation was 3D graphics, in many cases improvements were made to not be limited to 3D graphics. A new image file format support was introduced that can also be used in 2D graphics. A better list data type implementation was introduced to support integer and real

arrays that benefits a wide range of applications beyond 3D graphics. This meets one of the design goals of this dissertation: to make new features as general as possible so that they can be used across a range of application domains.

10.2 3D Interaction

A very high level 3D object selection model was created for the Unicon programming language, extending Unicon's 3D graphics API. The language hides the implementation details which leads to a design that puts fewer burdens on programmers when they use 3D object selection. Driven by the CVE requirements and its use case feedback, the model also adds a layer of abstraction in the form of a class that encapsulates details and makes 3D object selection event-driven, similar to common GUI environments. This new abstraction class blends in nicely within the CVE event-driven programming model. Most programmers are familiar with GUI programming style, where the programmer does not have to worry about how to get the mouse to work or where the mouse pointer is at the time of a click. The programmer needs to worry only about what to do when an object is picked. He provides a handler and connects it to a specific object in the scene with a specific mouse event. This handler then is called whenever the user triggers that event.

10.3 Concurrency

Concurrency was added to the virtual machine and runtime system by extending an existing non-concurrent thread type. A primary design goal was to add explicit concurrency with a minimal impact on the syntax. This goal was achieved initially, but the goal has changed over time into a new goal to find an appropriate balance between simplicity and power for explicit concurrent programming.

The primary contribution of the work consists of adding concurrency support to a goal-directed very high level language, in addition to the invention of thread-safety mechanisms that in many cases avoid the use of mutex-based synchronization. This effort was especially important in the memory allocator and garbage collector. For the majority of the runtime system, the thread-safety mechanism of choice was to migrate data into thread local storage and then focus on reducing the cost of accessing that storage.

The overhead cost of concurrency is non-negligible, especially when it affects non-concurrent execution. When CPython was parallelized in 1999 by Greg Stein, removal of its global interpreter lock reduced sequential performance by 50%, and true concurrency in CPython has been prevented for over a decade due to this negative result [98]. For Unicon, similar (42%) initial negative performance was ameliorated by aggressively reducing the necessity for mutexes and redundant lookups in thread-local storage as described earlier. The average overhead associated with concurrency in Unicon is presently around 5%, and varies depending on the application.

The act of porting Unicon's concurrency to Linux, OS X, and Windows resulted in lessons learned and forced code improvements (such as avoiding __thread) that increased performance, benefitting all platforms. The concurrency features were used and evaluated in different sections in the third part of this dissertation, specifically sections: 7.4, 8.5, and 9.2.5.

10.4 NPCs

PENQ is a framework for portable NPC tutors and quest activities built on top of a multi-platform non-player character architecture. PENQ provides World-of-Warcraft style quests for the purpose of delivering educational content. The architecture provides rudimentary NPCs that offer educational quest activities to users across virtual worlds. Creating a new NPC consists of writing a new web profile for that NPC. Creating new quests follows the same manner, enabling the virtual world to be populated with NPCs and educational content written by regular users

PENQ NPCs and their integration with CVE triggered many additions and improvements to the language such as 3D model support, 3D object selection to allow direct interactions with the NPCs, and also thread support. Thread support was augmented with new communication operators that support both thread communication and TCP communication. The result is a more scalable CVE server and NPCs that can run both as threads in the server and as independent clients and share the same code base.

10.5 CVE

The main innovations in building CVE are: combining its development with the development and enhancement of the host language; building an abstract and general class library and tools that reduce the programming effort necessary to build a virtual world; incorporating a social collaborative IDE into the virtual world, and finally facilitating the process of adding NPCs and quests with the potential of sharing them with other virtual worlds.

Building support into the language includes enhancements to graphics, networking, audio and the integration of these subsystems. Language extension was the choice instead of writing libraries or modules for general features such as adding a non-blocking read function, or when a feature has the need or the potential to interact with other features in the language virtual machine or runtime system. Once a given hardware capability is sufficiently ubiquitous, adding control structures and built-in syntax to access it is not just a notational convenience, but an enabling technology.

Adding new classes was the choice when features were specific to virtual worlds. The CVE class library primarily serves to model virtual environment functionality independent of its views and controls. The research contribution here is not to invent new paradigms, but to explore the simplest implementation

techniques that provide sufficient performance on current hardware. This design bias, combined with the very high level language used, is utilized to add new features and conduct experiments.

Using arrays and threads greatly improves the performance of CVE; combined the CVE performance is improved to up 7 times compared with original CVE before this dissertation work. This performance gain is essential in enabling new features in the CVE such as 3D models. Heap sizes play a major role in the overall performance of CVE. The optimal heap size value is based on actual experimentations. Other virtual world models and different 3D models for avatars will require a different amount of memory. 512MB for both block regions and string regions seems to be a good heap size for the main thread based on the current CVE state with thread heaps having the default 10% of the main heap size. Adding a little extra memory to threads would have a slight increase in performance, and a slightly more consistent experience, but it is not essential. Very large heaps have the undesirable impact of long garbage collections, so a balance has to be kept between garbage collection frequency and garbage collection duration. One solution to this problem is to have fast multi-threaded garbage collection, but this goes beyond the scope of the dissertation.

11 Future Work

The work presented in this dissertation forms a ground for further research directions following the final defense of this dissertation. This chapter sheds some light on possible opportunities where this work could be expanded.

11.1 3D Graphics

The 3D graphics facilities in Unicon provides a simple to use and high level set of features very suitable for developing virtual worlds. This feature set should be combined with an efficient rendering pipeline to make it useful in real time applications such as virtual worlds. This dissertation targeted improvements on both the feature set and also the performance of 3D graphics in Unicon. Yet there are several areas that can benefit from further studies and improvements in both features and performance.

Arrays had a significant impact on performance. Similarly supporting special and optimized types such as vertices and matrices, which is essential in 3D graphics, would have a significant positive impact on performance, especially if combined with SIMD data-parallel operators, which in turn helps boost performance even more and also simplifies writing 3D graphics code by building native support for these data types, and their operations.

11.2 3D Interaction

Unicon's 3D object selection provides a simple API to interact with objects in the scene, whether used directly or through the GUI-like class library. The work can be further improved with features useful for developing virtual worlds. One such feature is to have separate keywords for different mouse buttons such as &leftpick and &rightpick to get the picked objects using a left click or a right click respectively, instead of combining all of the selection result in &pick. This helps avoid having to check other mouse button keywords to extract the information about which mouse button was clicked at the time of picking. Another improvement is to have the ability to extract information about the picked object such as color, texture and world coordinate.

The most interesting improvement would be the ability to pick objects in texture. With dynamic texture, 3D objects or GUI components can be rendered directly into a texture in the 3D world. Supporting clicks polygons or on GUI components in a texture allows for a new way of interacting with objects in the scene such that all GUI events and interaction is done inside the 3D world itself.

11.3 Concurrency Support

While they have already proven useful in an important real-world application such as CVE, the Unicon concurrency facilities will become more useful with refinement. Future work includes further performance tuning, and extension to a broader range of concurrency modes appropriate to Unicon's domain.

Several techniques can be used address the impact of concurrency on sequential programs to further improve their performance. For example, the performance of some programs might be improved by providing concurrent and non-concurrent versions of selected virtual machine functions, such that the concurrent version is not used unless concurrent threads are detected in the program. The switching over to concurrency-supporting versions of the functions happens implicitly without the programmer intervention. Another technique to improve performance is to reduce lookups in the thread local storage. This can be done by passing a pointer to the thread state as an extra parameter on the stack where it is needed. This cost might be somewhat mitigated by passing a pointer to the thread state as an extra parameter on the stack where it is needed. The effect of such a change should be studied and be adopted if it provides a net gain over the current implementation.

Compared with pure functional or logic programming languages, implicit parallelism is a challenge in a pragmatic language such as Unicon. Implicit parallelism is a major opportunity due to the very high level of the language. Both co-expressions and generators represent natural units for analysis; in some instances the code may be parallelized implicitly. Generators are especially exciting prospects for implicit parallelism, as they naturally describe a fine-grained, demand-driven parallel computation.

Another major area for future work is improvement of the garbage collector. Analysis can determine that some threads may garbage collect without stopping all the other threads; for example, producers and consumers that have no shared memory references and communicate purely through message passing. When synchronous collection is required and threads do have to be stopped, the garbage collector uses a mark and sweep algorithm; the marking phase could be parallelized.

Improving the performance of Refresh() represents an attractive opportunity to improve the overall performance of virtual worlds. This is particularly interesting because Refresh() can benefit from having multiple underlying concurrent threads that are not visible at the source level. This would add a form of implicit concurrency that the application gets for free without any modification.

References

[1] "America's Army," U.S.ARMY, [Online]. Available: http://www.americasarmy.com/. [Accessed August 2012].

[2] "MIT's Education Arcade Uses Online Gaming to Teach Science," 17 January 2012. [Online]. Available: http://links.visibli.com/04e74d1c146e7eeb/?web=a74b88&dst=http%3A//education.mit.edu/blogs/louisa/2012/pressrelease. [Accessed August 2012].

[3] "Massively Multiplayer Online Role-Playing Game," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Massively_multiplayer_online_role-playing_game. [Accessed August 2012].

[4] C. Steinkuehler, "Massively Multiplayer Online Games & Education: an Outline of Research," in *CSCL'07 Proceedings of the 8th iternational conference on Computer supported collaborative learning*, 2007.

[5] "World of Warcraft," Blizzard, [Online]. Available: www.worldofwarcraft.com. [Accessed August 2012].

[6] "Second Life," Linden Research, Inc, [Online]. Available: http://secondlife.com/. [Accessed August 2012].

[7] "ActiveWorlds," ActiveWorlds Inc, [Online]. Available: http://www.activeworlds.com/. [Accessed August 2012].

[8] "Open Cobalt," [Online]. Available: http://www.opencobalt.org/. [Accessed August 2012].

[9] G. DeMicheli and M. G. Sami, Hardware/Software Co-Design, Springer, 1996.

[10] G. De Micheli and K. R. Gupta, "Hardware/Software Co-Design," *Proceedings of THE IEEE,* vol. 85, no. 3, pp. 349-365, 1997.

[11] Hardware/Software Codesign Group, "A Framework for Hardware-Software Co-Design of Embedded Systems," [Online]. Available: http://embedded.eecs.berkeley.edu/Research/hsc/abstract.html. [Accessed December 2011].

[12] W. H. Wolf, "Hardware-Software Co-Design of Embedded Systems," *Proceedings of The IEEE,* vol. 82, no. 7, pp. 967-989, 1994.

[13] M. Wolfe, "Compilers and More: Hardware/Software Codesign," HPCWire, 2012.

[14] K. Klues, M. Kazandjieva and P. Levis, "Operating System/Language Co-Design," 2011. [Online]. Available: http://sing.stanford.edu/os_language/. [Accessed September 2011].

[15] C. Jeffery, S. Mohamed, R. Parlett and R. Pereda, Programming with Unicon, 2003.

[16] R. Griswold and M. Griswold, The Icon Programming Language, 3rd ed, San Jose, CA: Peer-to-Peer Communications, 1999.

[17] J. Al-Gharaibeh and C. Jeffery, "PNQ: Portable non-player characters with quests," in *Proceedings of the 2010 International Conference on Cyberworlds (CW2010)*, Waltham, MA, 2010.

[18] "Doom (video game)," [Online]. Available: http://en.wikipedia.org/wiki/Doom_%28video_game%29. [Accessed August 2012].

[19] D. Kushner, Masters of Doom: How Two Guys Created an Empire and Transformed Pop Culture, The Random House Publishing Group, 2003.

[20] T. E. Wright and G. Madey, "A Survey of Technologies for Building Collaborative Virtual Environments," *The International Journal of Virtual Reality,* vol. 8, no. 1, pp. 53-66, 2009.

[21] "Nintendo's Official Home for Mario," Nintendo, [Online]. Available: http://mario.nintendo.com/. [Accessed August 2012].

[22] R. E. Griswold and M. T. Griswold, The Implementation of the Icon Programming Language, Princeton University Press, 1986.

[23] H. Takemura , A. Utsumi and F. Kishino, "Augmented Reality: A Class of Displays On The Reality-Virtuality Continuum," *SPIE Telemanipulator and Telepresence Technologies,* vol. 2351, pp. 282-292, 1994.

[24] B. E. Mennecke, D. McNeill, E. M. Roche, A. D. Bray, M. A. Townsend and J. Lester, "Second Life and Other Virtual Worlds: A Roadmap for Research," *Communications of the Association for Information Systems,* vol. 22, pp. 371-388, 2008.

[25] "Second Life Work FAQ," [Online]. Available: http://wiki.secondlife.com/wiki/Second_Life_Work/FAQs#Cost_and_Billing. [Accessed November 2011].

[26] "Become a Virtual World Citizen!," ActiveWorlds, [Online]. Available: http://www.activeworlds.com/products/citizenships.asp. [Accessed December 2011].

[27] "Game Engine, Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Game_engine. [Accessed December 2011].

[28] J. Ward, "What is a Game Engine?," UBM TechWeb, 29 04 2008. [Online]. Available: http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=2. [Accessed 09 December 2011].

[29] L. Bishop, D. Eberly, T. Whitted, M. Finch and S. Michael, "Designing a PC game engine," *IEEE Computer Graphics and Applications,* vol. 18, no. 1, pp. 46-53, 1998.

[30] M. Lewis and J. Jacobson, "Game Engines in Scientific Research," *Communications,* vol. 45, no. 1, pp. 27-31, 2002.

[31] "Top 10 Most Expensive Softwares in the World," [Online]. Available: http://www.mostcostly.com/most-expensive-software. [Accessed December 2011].

[32] R. Darken, P. McDowell and E. Johnson, "The Delta3D Open Source Game Engine," *IEEE Computer Graphics and Applications,* vol. 25, no. 3, pp. 10-12, 2005.

[33] P. McDowell, R. Darken, J. Sullivan and E. Johnson, "Delta3D: a complete open source game and simulation engine for building military training systems," *JDMS,* vol. 3, no. 3, pp. 143-153, 2005.

[34] "Game Engines," [Online]. Available: http://devmaster.net/devdb/engines. [Accessed December 2011].

[35] LudoCraft, "Open Source & Low Cost Game Engines," [Online]. Available: http://ludocraft.oulu.fi/elias/dokumentit/open_source_game_engines.pdf. [Accessed December 2011].

[36] J. Fristrom, "Manager In A Strange Land: Reuse and Replace," GamaSutra, UBM TechWeb, 09 01 2004. [Online]. Available: http://www.gamasutra.com/view/feature/2020/manager_in_a_strange_land_reuse_.php. [Accessed 09 Desember 2011].

[37] C. Jeffery, O. El-khatib, Z. Al-sharif and N. Martinez, "Programming Language Support for Collaborative Virtual Environments," in *the 18th International Conference on Computer Animation and Social Agents (Casa)*, Hong Kong, 2005.

[38] D. Shreiner, M. Woo and J. Neider, OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd ed, Amsterdam: Addison-Wesley Longman.

[39] J. Yee, "A Survey Of Graphics Programming Languages," 2004.

[40] "Domain-specific language," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Domain-specific_language. [Accessed December 2011].

[41] W. R. Mark, R. S. Glanville, K. Akeley and M. J. Kilgard, "Cg: a Aystem for Programming Graphics Hardware in a C-like Language," *ACM Transactions on Graphics (TOG),* vol. 22, no. 3, pp. 896-907, July 2003.

[42] J. Kessenich, D. Baldwin and R. Rost, "The OpenGL Shading Language Version 1.20," 07 09 2006. [Online]. [Accessed December 2011].

[43] S. St-Laurent, The Complete Effect and HLSL Guide, Redmond, WA: Paradoxal Press, 2005.

[44] W. Broll and T. Kopp, "VRML: Today and Tomorrow," *Computers and Graphics,* vol. 20, no. 3, pp. 427-434, 1996.

[45] D. Brutzman, "Computer Graphics Teaching Support using X3D: Extensible 3D Graphics for Web Authors," in *ACM SIGGRAPH ASIA*, Singapore, 2008.

[46] J. X. Chen and E. J. Wegman, Foundations of 3D Graphics Programming Using JOGL and Java3D, New York: Springer, 2006.

[47] D. J. Bouvier, "Getting Started with the Java 3D API, Chapter 1," 2000. [Online]. Available: http://java.sun.com/developer/onlineTraining/java3d/j3d_tutorial_ch1.pdf. [Accessed January 2012].

[48] P. Hudak and J. H. Fasel, "A Gentle Introduction to Haskell," *SIGPLAN Notices,,* vol. 27, no. 5, 1992.

[49] P. Hudak, "Modular Domain Specific Languages and Tools," in *Proceedings of the Fifth International Conference on Software Reuse*, 1998.

[50] P. Hudak, S. P. Johns and P. Wadler, "Report on the Programming Language Haskell, A Non-Strict Purely Functional Language," *SIGPLAN Notices,* vol. 27, no. 5, 1992b.

[51] C. Elliott, "Functional Implementations of Continuous Modeled Animation," in *Proceedings of the 10th International Symposium on Principles of Declarative Programming*, 1998.

[52] C. Elliot, "An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation," *IEEE Transactions on Software Engineering,* vol. 25, no. 3, pp. 291-308, 1999.

[53] C. Reinke, "FunWorlds/HOpenGL, Functional Programming and Virtual Worlds," in *International Workshop on Implementation of Functional Languages (IFL)*, 2001.

[54] C. Elliott and P. Hudak, "Functional Reactive Animation," in *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, New York, 1997.

[55] Sun Microsystems, "The Java 3D Tutorial (Pages 4.1- 4.52)," [Online]. Available: http://java.sun.com/developer/onlineTraining/java3d/j3d_tutorial_ch4.pdf. [Accessed September 2011].

[56] D. F. Savarese, *Learning to Fly (Intro to Java 3D API),* 2003.

[57] D. Beazley, "Inside the Python GIL. Python Concurrency Workshop," May 2009. [Online]. Available: http://www.dabeaz.com/python/GIL.pdf. [Accessed August 2012].

[58] P. Hudak, "Exploring parafunctional programming: Separating the What from the How," *IEEE Software,* pp. 54-61, January 1988.

[59] S. L. Peyton-Jones, The Implementation of Functional Programming Languages, Prentice Hall International, 1987.

[60] W. Hasselbring, "Approaches to High-Level Programming and Prototyping of Concurrent Applications," *Software-Technik Memo 91,* 1997.

[61] E. M. Paalvast, H. J. Sips and L. C. Breebaart, "Booster: a High-Level Language for Portable Parallel Algorithms," *Applied Numerical Mathematics,* vol. 8, no. 2, pp. 177-192, 1991.

[62] T. Mitsolides, *The Design and Implementation of ALLOY, a Higher Level Parallel Pro-gramming Language,* PhD thesis, New York University, 1992.

[63] T. Mitsolides and M. Harrison, "Generators and the replicator control structure in the parallel environment of alloy," in *PLDI '90 - ACM SIGPLAN conference on Programming language design and implementation*, 1990.

[64] J. Armstrong, R. Virding, C. Wikstrom and M. Willimas, Concurrent Programming in ERLANG, Englewood Cliffs, NJ: Prentice Hall, 1996.

[65] D. Beazley, "Understanding the Python GIL," [Online]. Available: http://www.dabeaz.com/python/UnderstandingGIL.pdf. [Accessed August 2012].

[66] D. M. Beazley, "Inside the New GIL," 14 January 2010. [Online]. Available: http://www.dabeaz.com/python/NewGIL.pdf. [Accessed August 2012].

[67] "Stackless Python," [Online]. Available: http://www.stackless.com/. [Accessed August 2012].

[68] "Channels in Stackless Python," [Online]. Available: http://www.stackless.com/wiki/Channels. [Accessed August 2012].

[69] R. Ierusalimschy, L. H. De Feigueiredo and W. Celes, "Passing a language through the eye of a needle," *Communications of the ACM,* vol. 54, no. 7, pp. 38-43, July 2011.

[70] A. Skyrme, N. Rodriguez and R. Ierusalimschy, "Exploring Lua for Concurrent Programming," *Journal of Universal Computer Science,* vol. 14, no. 21, pp. 3556-3572, 2008.

[71] G. Andrews and R. Olsson, The SR Programming Language: Concurrency in Practice, Benjamin/Cummings, 1993.

[72] H. Bal, J. Steiner and A. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys,* vol. 21, no. 3, pp. 261-322, 1989.

[73] B. Capman, G. Jost and R. van der Pas, Using OpenMP, MIT Press, 2007.

[74] P. Gustafson, "Detecting and Avoiding OpenMP Race Conditions in C++," [Online]. Available: http://developers.sun.com/solaris/articles/cpp_race.html. [Accessed December 2011].

[75] A. Kolosov, E. Ryzhkov and A. Karpov, "32 OpenMP traps for C++ developers," May 2008. [Online]. Available: http://software.intel.com/en-us/articles/32-openmp-traps-for-c-developers. [Accessed December 2011].

[76] S. Redfern and N. Naughton, "Collaborative Virtual Environments to Support Communication and Community in Internet-Based Distance Education," *Journal of Information Technology Education,* vol. 1, no. 3, 2002.

[77] K.-S. Yoo and W.-H. Lee, "An Intelligent Non Player Character Based on BDI Agent," in *Fourth International Conference on Networked Computing and Advanced Information Management NCM '08 2*, 2008.

[78] K. Merrick and M.-L. Maher, "Motivated Reinforcement Learning for Non-player Characters in Persistent Computer Game worlds," in *the International Conference on Advances in Computer Entertainment Technology*, 2006.

[79] K. Merrick and M.-L. Maher, "Motivated reinforcement learning for adaptive characters in open-ended simulation games," in *ACM international conference on Advances in computer entertainment technology 203*, 2007.

[80] C. McCollum, C. Barba and T. Santarelli, "Applying a Cognitive Architecture to Control of Virtual Non-Player Characters," in *2004 Winter Simulation Conference*, 2004.

[81] A. Fossett, "PandoraBot NPC," [Online]. Available: http://artfossett.blogspot.com/2007/07/pandorabot-npc.html. [Accessed September 2011].

[82] R. Wallace, Artificial Intelligence Markup Language (AIML) Version 1.0.1, Working Draft 25 October 2001.

[83] D. Friedman, A. Steed and M. Slater, "Spatial Social Behavior in Second Life," in *Intelligent Virtual Agents LNAI 4722*, Paris, Farnce, 2007.

[84] "Quest Atlantis," [Online]. Available: http://www.questatlantis.org/. [Accessed August 2012].

[85] H. Tuzun, "Quest Atlantis: A Computer Game That Transcends the Computer," [Online]. Available: http://www.e-mentor.edu.pl/_xml/wydania/5/64.pdf. [Accessed September 2011].

[86] S. Barab, M. Thomas, T. Dodge, R. Carteaux and H. Tuzun, "Making Learning Fun: Quest Atlantis, A Game Without Guns," *Educational Technology Research and Development,* vol. 53, no. 1, pp. 86-107, 2005.

[87] C. Jeffery and S. Jeffery, "An IVIB Primer (Unicon Technical Report #6b)," 2006.

[88] R. Griswold, C. Jeffery and G. Townsend, Graphics Programming in Icon, San Jose, CA: Peer to Peer Communications, 1998.

[89] C. Jeffery and N. Martinez, "The Implementation of Graphics in Unicon Version 11 (Unicon Technical Report #5a)," 2003.

[90] D. Knuth, "The Art of Computer Programming," *Fundamental Algorithms,* vol. 1, 1968.

[91] A. L. De Moura and R. Ierusalimschy, "Revisiting Coroutines," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 31, no. 2, pp. 1-31, 2009.

[92] S. E. Lumos, *Messaging Language Extensions for Unicon, Master Thesis,* University of Nevada, Las Vegas, 2000.

[93] Z. Al-Sharif and C. Jeffery, "Adding High Level VoIP Facilities to the Unicon Language," in *Proceedings of the Third International Conference on Information Technology: New Generations (ITNG)*, 2006.

[94] C. Jeffery, N. Martinez and J. Al-Gharaibeh, "Unicon 3D Graphics: User's Guide and Reference Manual (Unicon Technical Report #9b)," 2010.

[95] K. N. Oikonomou, "Network Performability Evaluation," in *Guide to Reliable Internet Services and Applications*, C. R. Kalmanek, S. Misra and C. R. Yang, Eds., Springer, 2010, pp. 113-135.

[96] "Thread-safety and POSIX.1," UNIX white papers, [Online]. Available: http://www.unix.org/whitepapers/reentrant.html. [Accessed September 2011].

[97] G. Booch, J. Rumbaugh and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1998.

[98] "Python FAQ," [Online]. Available: http://docs.python.org/faq/library#can-t-we-get-rid-of-the-global-interpreter-lock. [Accessed September 2011].