Building a Collaborative Virtual Environment: A Programming Language Codesign Approach

Jafar Al-Gharaibeh, Clinton Jeffery and Hani Bani-Salameh
Computer Science Department
University of Idaho
Moscow, Idaho USA
jafara@vandals.uidaho.edu, {jeffery, hani}@uidaho.edu

Abstract— **Developing 3D virtual environments requires an advanced level of programming expertise. In many cases, working on such an application involves a wide range of programming activities such as 3D graphics, networking, user interfaces and audio programming. At the same time, such applications are usually real time and performance critical. The complexity of developing such an application comes from two sources: first, the programming language used in development with its strengths and also the limitations it imposes. The second is the requirements of the virtual environment itself, with its dynamics and size. Some languages are more suitable than others for any given application domain, but in general once a programming language is selected, the main focus is the application itself and not the language used. This paper presents an approach where a virtual environment (CVE) and its implementation language (Unicon) evolved together over the course of the project development, derived from emerging functional and performance requirements. The Unicon language was improved and new features were added over time to meet new demands and address the complexities that arose at the application level. This approach was combined with developing a framework to build virtual worlds with a social integrated development environment and to populate those worlds with non-player characters.**

Keywords-component; language design; cve; 3D models; social IDE; object selection; language-application codesign

I. INTRODUCTION

Cyberworlds have a tremendous impact on the physical world. People are spending more and more time in virtual worlds and social media spaces. From entertainment through army and industry to education, cyberspaces have found uses in many diverse applications. Yet the cost and complexity of building virtual worlds remain a challenge in terms of the programming expertise required and long development time needed to build such worlds. Such resources are usually only available in big companies and large institutions. Graphics APIs such as OpenGL [10] enable the development of portable 3D applications. Such APIs are very large, complex and low level. Higher level Java3D is claimed to enable ordinary programmers to write 3D graphics, but the size and complexity of the API (150+ classes) contradicts that claim.

The CVE project described in this paper was motivated initially by a goal to support distance education in a virtual world. The project expanded over the years to build a more general framework for rapid prototyping of virtual worlds, not just the geometry and appearance of the virtual space, but potentially extensive domain-specific behavior needed for virtual worlds to be adapted to new uses. The goal became to build an enabling technology and infrastructures that make the task of creating a virtual world something manageable by a small team with limited resources.

A secondary goal is to learn lessons from building virtual worlds using an open source very high level programming language, where new capabilities and features can be built into the language. Such features can employ the very high level semantics and powerful data structures where the programmer does not have to learn enormous libraries to do tasks such as 3D graphics or networking. This approach hides a lot 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?

The programming language used in this project is called Unicon; an object-oriented descendant of the Icon programming language [3, 4]. 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.

II. MOTIVATION

The obvious way to build a new virtual world application is to use an existing virtual environment such as SecondLife, OpenSim, ActiveWorlds, or OpenCobalt, avoiding the construction costs which can easily run into the tens of millions of dollars[1]. However, at the onset of this project it was realized that the application domain requirements were substantially different from existing game-oriented virtual worlds. Adapting an existing gigantic codebase was rejected

for this project because it was believed that this option would exceed the technical skill of the students involved.

The popularity of hardware/software codesign suggests an under-utilized analog: language/application codesign. Programming languages are in fact much easier than hardware to codesign alongside the applications that run on them. In many cases this codesign occurs during the creation of a domain specific language, and there are conspicuous examples, such as operating system/language co-design [2]. As vendors of the very high level language Unicon, the project team decided to view the project as an opportunity to improve the language by means of language/application codesign.

This approach extends a language virtual machine 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.

III. Programming Language Support

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 very suitable for representing the virtual world state and data. In addition, high-level graphics and networking features are major emphases in Unicon's research agenda. These combined to make it an attractive starting point for virtual environment codesign. Although language extension was an explicit goal, the goal was to find the smallest, most elegant set of language additions necessary in order to support the development of virtual environment, especially if the change is visible at the language level, such as adding a new function or introducing a new syntax. Different parts of the language have been extended or improved throughout the life time of the project. Some of these improvements and additions are ongoing. The next several sections highlight some of these major extensions.

A. 3D Graphics API

Unicon has a high level 2D graphics API inherited from Icon programming language with many extensions and also the addition of a comprehensive GUI class library [5]. This API was extended to support 3D graphics built on top of OpenGL [6]. Unicon's 3D API is not a wrapper around the OpenGL C library interface, but rather a high level abstraction that enables programmers to use whatever the language provides transparently with the underlying OpenGL implementation.

Instead of the more than 250 OpenGL [10] functions that C programmers have to learn, Unicon provides about 30 3D functions, and that includes several 2D graphics functions that were extended to have 3D semantics. For example the DrawLine() function was extended to take three coordinates per vertex when working with 3D graphics instead of only two in the case of 2D graphics.

In addition to the big reduction of function count, Unicon also relieves the programmer of many other burdens. Unlike OpenGL, Unicon has a built-in support for many window systems. Opening windows and handling their events comes free of any extra work. OpenGL programmers use third party libraries and tools to handle the window system. Another simplification by the language is loading and handling texture files. Unicon handles all of that with support for several image file formats. OpenGL on the other hand, does not have any built-in support for image formats. Programmers have to either write their own code or rely on third party libraries to handle image files.

The original Unicon 3D API was improved and extended over time to include more features after new 3D applications, especially CVE, and experiments were conducted 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, vertex normals support for better smooth shading, a way to control the visibility of different parts of the scene, and several other improvements including 3D selection discussed in section III.C

B. 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 in particular 3D model files to store the content of the world. Such models can be created using applications such as 3D Max and Blender. Likewise, many websites sell or in some cases provide for free all kinds of 3D models. The challenge lies in loading and manipulating such models. OpenGL has no built-in support for reading model files. If programmers need to read a specific file type, they will have to find libraries or write code to support that particular file type. Direct X provides support for Microsoft .x file format, but Direct X is Windows specific and so it is platform dependent.

A class library was added to Unicon to support two 3D model formats, namely 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 or not 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 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. Fig. 1 is a screen shot of the tool which is called: Unicon 3D Model Viewer.

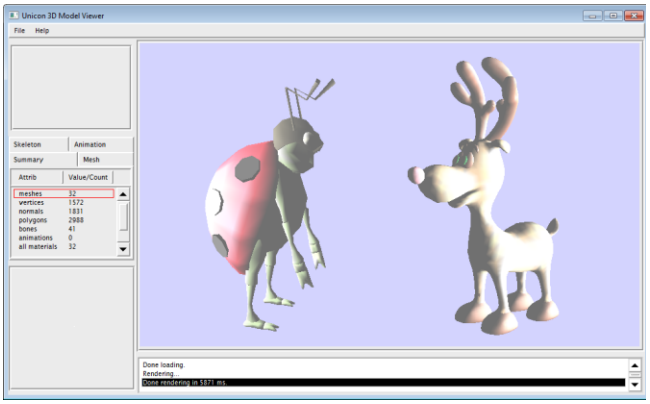

Figure 1. A 3D Model Viewer was developed to help Unicon programmers use 3D models in games and virtual worlds

C. 3D Object Selection

User input is essential in virtual worlds. Much of this input comes through direct interaction with the virtual world's content, usually using a mouse. 3D Object selection was added to Unicon for this project, complementing its 3D graphics API. OpenGL's low-level object selection mechanism was inappropriate for Unicon. A higher-level mechanism was built into the Unicon virtual machine, hiding all of the low level details and leaving a very simple interface at the language level. The interface hides a total of 11 OpenGL functions that are usually used for 3D object selection. In Unicon, one existing function was extended for selection, plus a keyword and a window attribute were introduced. 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 important components in the Icon and Unicon programming languages.

Working with 3D selection in Unicon is easy and straight forward. 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

1) Controlling the selection state

Turning on/off 3D selection is controlled by the Unicon function WAttrib() [5]. WAttrib() is a generic routine for getting or setting a window's attributes. To turn on 3D selection at any point in the program, the following statement is inserted:

```
WAttrib("pick=on")
```

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 any non-selectable object in the scene.

2) Naming 3D Objects

3D Objects are defined by their corresponding rendered primitives. The function WSection() marks the beginning and the ending of a section that holds a 3D object. A call to the function WSection with a parameter string marks the beginning of a 3D object with the strings as its name. Another call to WSection() with no string parameter marks the end of the 3D objects. 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 named 3D object. 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.

3) Retrieving Picked Objects

In general, picking objects is associated with the mouse. In Unicon, keyboard events are (mostly) corresponding one letter strings and mouse events are small negative integers. Mouse and keyboard state information can be accessed thought 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 at the time of the click. &pick was designed to work in the same fashion with 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. It is up to the programmer to decide and assign specific behaviors or event handlers to different mouse buttons based on the picked object.

4) Complete Example

This section presents a simple full example program. The example 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 he will get the message "you picked red ball" or "you picked blue ball". If the user clicked anywhere else including on the green ball he 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 : ", &pick | "nothing" )
    }
  }
end
```

D. 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.

1) Data respresentation

3D graphics makes extensive uses 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 this 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 designed 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.

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, but also it does not work for any dynamic object in the scene that requires constant update to its data. A more general and optimal solution was needed that did not require more memory and worked well for any objects, including those that involve animation. Animation makes a difference because usually animation is done by key farming and applying an animation transformation to vertex data, generating a new set of world vertices that replaces the old set. In addition to that, the same data would be 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. To avoid any accidental array creation, a new constructor function for arrays was added. A "regular" list can be created using the list(size, initial value) function. An "array" style list can be created using the new array(size, initial value). For an array list, 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)
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 would force it to be converted to a regular list. For example applying pop(), push(), get() or put() at the list array A in the code above. This is done by the runtime system without the programmer intervention.

E. 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". The program downloads a remote file specified by a URI on the command line, and saves it as a local file. The Icon Program Library module basename is used to extract the filename from the URI.

```
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 long times 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 in Unicon terms. This behavior is very critical in real time applications such as virtual worlds.

A good example of language/application codesign 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 would require 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() that enables a single process or thread to handle new connections while serving multiple existing user connections.

F. Audio and Voice Over IP

Audio is an essential part of games and virtual worlds' experience. Voice chat is also becoming an integral part such applications. Unicon was extended to support such capabilities. Only one function, VAttrib(): similar to an existing Unicon graphics function WAttrib(), was added. Two existing function, 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 [11].

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. When passed a voice session handle, the function close(x) terminates the session and releases all resources related to that session.

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. Changing the voice session settings on

the fly adds a very important feature to the Unicon's VoIP APIs. Users do not have to close the voice session and lose all the connections 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.

IV. COLLABORATIVE VIRTUAL ENVIRONMENT

Virtual environments can be defined as computer-generated, three-dimensional settings in which the users of the technology perceive themselves to be and within which interaction takes place [8]. As the technological barriers to creating virtual worlds have decreased, researchers have created many collaborative virtual environments to serve various domains. Virtual environments provide the user with the amazing experience of moving around and interacting with a simulated world [9].

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. Fig. 4 shows an example scene developers might see in this environment. The collaborative virtual environment provides developers with a general view of other users and what they are doing. It allows developers to chat via text or VoIP with other team members and with developers from other teams in real time.

CVE was built as a general prototyping framework rather than a particular virtual world. From a design point of view, the CVE has three major layers:

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

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 quest system that users can interact with and take on quests

The following section discusses the CVE design layers. The remaining two sections discuss SCI and NPCs subsystems.

A. CVE Design Layers

Out of the three CVE layers, the bottom one, a very high level language, plays the major role in shaping the design and implementation of the virtual world. As discussed in the earlier sections of this paper, the language was augmented with very simple 3D graphics, object selection, network, and audio API's to facilitate the building of such worlds.

In the CVE middle layer lies a class library, in addition to the language class libraries that were added to support new features required by the virtual world, providing infrastructure for the networked 3D environment, e.g. the behavior of doors, whiteboards, and avatars; the CVE server enables user interaction and shares state between clients; CVE also provides simple "builder" tools to generate a virtual environment from inputs such as 2D floor plan data and extract textures from digital camera photos.

The last and top layer is the application layer for any particular virtual world generated using CVE which typically consists of: a 3D model 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 virtual spaces that can be created dynamically from within the virtual world, new NPCs and quests can be added over time to expand the world and create new activities.

B. Social Collaborative IDE

A subsystem of CVE called Social Collaborative IDE (SCI) supports communication and collaboration within a distributed software development community, and addresses their needs in a variety of different phases in a team software development process.

Fig. 2 shows the integration of the presence information, collaboration tools, and software development facilities in a single environment.

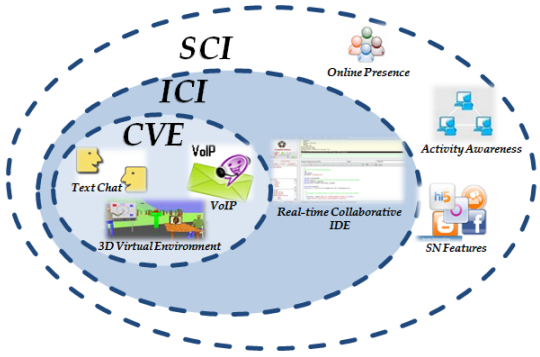

Figure 2. The SCI architecture

The inner oval represents the CVE collaborative virtual environment where users can interact with each other within a 3D virtual world. CVE provides developers with a general view of other users and what they are doing. In the middle oval, ICI developers use synchronous collaborative software development tools that extend CVE's generic virtual environmental capabilities to communicate, interact, and collaborate in solving their programming problems. The outer oval provides the developers with groups', users', projects', and sessions' presence and awareness information. SCI's asynchronous features help users to select and coordinate their active synchronous collaborations. In general, the asynchronous tools drive the use of the synchronous tools, and the two categories complement each other. CVE, ICI, and SCI are complementary tools that work together to provide a unique single development environment. Fig. 3 shows the structure of the current SCI space.

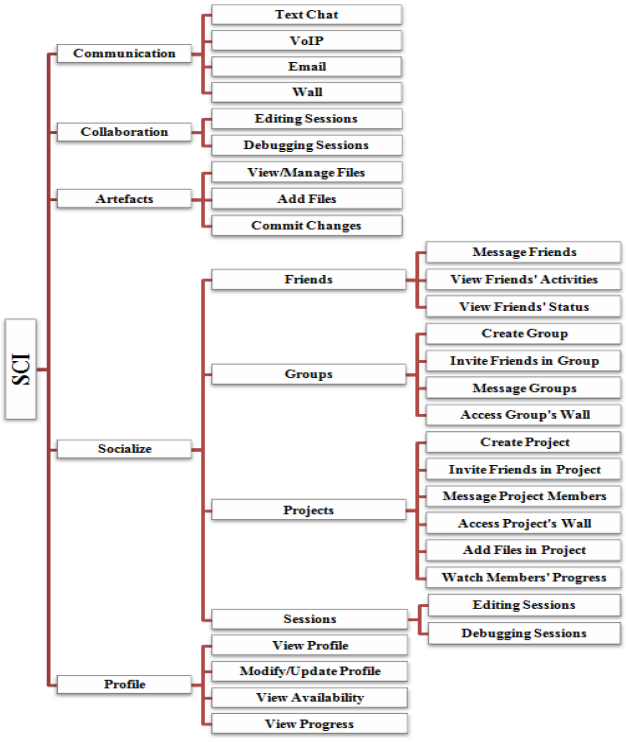

Figure 3. The structure of the SCI space

SCI started with a basic standalone IDE that lives inside the Unicon programming language called UI. Considering the fact that CVE was initially built for the purpose of supporting distance learning and improve software engineering instruction, UI was extended to support multiple programming languages and integrated inside the CVE virtual environment (see ICI in Fig. 2). To allow users to get the benefit from the combination of the online presence supported by the CVE and the software development, the system was extended by adding awareness, presence, and social networking features (see SCI in Fig. 2).

C. Non-Player Characters and Quests

NPCs are an integral part of many virtual worlds especially role playing games and massively multiplayer online games (MMOs). NPCs usually perform the task of presenting activities, guiding and completing the storyline of the game. NPCs usually take the role of enemies, allies, monsters or pets, but their most important role is that of quest giver or assistant to the user who goes on adventures.

In order to populate the CVE virtual world with NPC and quest activities a new framework for creating them was proposed called PNQ [7]. The framework is independent from the CVE system enabling a PNQ NPC to be part of several virtual worlds. The goal is to promote virtual worlds' content reuse. It also enables users to create NPCs and quests outside the virtual world without the need to make changes to the source code of the virtual world. New NPCs and quests can be added using template files or the quest builder which is part of the PNQ framework. The user can customize an NPC and then import it to the virtual world. CVE NPCs' avatars can be created from 3D models produced by tools such as 3D Studio Max and exported in Microsoft .x format. Fig.4 shows some example NPC avatars models available in the CVE environment along with a quest activity.

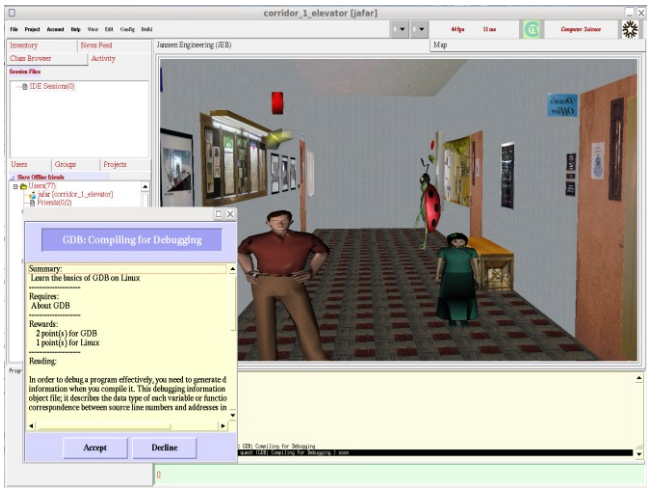

Figure 4. NPCs in CVE virtual world

V. DISCUSSION

The main innovations in building CVE are: combining its development with the development and enhancement of the host language; building 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 both, graphics, networking, audio and the integration of these subsystems. Language extension was the choice instead of writing libraries or modules when a feature is general like adding a non-blocking read function, or when it 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, make it easy to add new features and conduct experiments.

In some cases, new extensions 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 was the data type of choice to store a lot of the virtual world data. In later stages and 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. While some very simple scenes got a modest increase in performance, for some scenes especially those that have 3D models, the performance was tremendously improved with up to 100x speed up. Fig. 5 shows a 3D model for a warrior with animation for a walk cycle. The model has a 485 vertex count and 528 face count. Without using arrays the animation runs at less than 5 frames per second. This number was pushed up to 80 frames per second after introducing arrays as a special case of the list data type in Unicon. This model runs comfortably at 400 frames per second in C code, but the programmer has to deal with the complexity of writing C. One of the primary goals of this research was to overcome that complexity by using a very high level language.

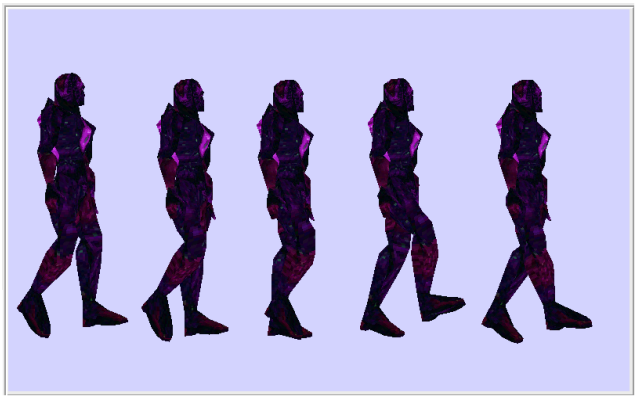

Figure 5. An example 3D model with an animation of a walk cycle. The performance of such a model was greatly improved by introducing arrays to be part of the list data type in Unicon

VI. CONCLUSIONS

This paper discussed aspects of the codesign between the Unicon programming language and the CVE collaborative virtual environment. On the language side this codesign included several extensions to Unicon's 3D graphics and networking facilities. On the application side, the CVE virtual environment was modified to utilize several new language features as they were added, including 3D object selection, VOIP, and non-blocking I/O. The CVE application's 3D rendering model was specifically tailored to take advantage of Unicon's "render sections" control structure for both object selection and the ability to dynamically include or exclude portions of the rendered scene on the fly.

The results of this work can be seen on both sides of the research, the language and the virtual world. In the first side, the language has many new extensions and improvements that benefit wide range of applications and not limited to virtual worlds development only, even though many extensions were driven by that. On the other side, the CVE platform is very suitable for quickly prototyping a new virtual world serving like an engine. While CVE is usually tested with a small number of simultaneous users, 30 or less, it has been gradually improved in term of graphics and network traffic to scale to larger numbers of users. CVE has not been developed with the kind of hardware or software engineering resources that go into commercial games that handle hundreds or thousands of users.

ACKNOWLEDGMENT

This work was supported in part by NSF DUE- 0402572 and in part by the Specialized Information Services division of the National Library of Medicine.

REFERENCES

[1] A. Carpenter, "Applying risk analysis to play-balance RPGs," http://www.gamasutra.com/view/feature/2843/applying_risk_analysis_to_.php. (accessed July 2011)

[2] K. Klues, M. Kazandjieva, and P. Levis, "Operating system/language co-design," http://sing.stanford.edu/os_language. (accessed July 2011)

[3] C. Jeffery, S. Mohamed, R. Pereda, and R. Parlett, Programming with Unicon, 2004. Unicon Project at http://unicon.org.

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

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

[6] C. Jeffery, O. El-khatib, Z. Al-sharif, and N. Martinez, "Programming language support for collaborative virtual environments," Proc. 18th International Conference on Computer Animation and Social Agents. CGS. 2005.

[7] J. Al-Gharaibeh, and C. Jeffery, "PNQ: Portable non-player characters with quests,". Proc. 2010 International Conference on Cyberworlds, CW2010, 2010.

[8] Dictionary.com. available at http://dictionary.reference.com/browse/virtual environment.

[9] J. Langton, T. Hickey, and R. Alterman, "Integrating tools and resources: a case study in building educational groupware for collaborative programming," Journal of Computing Sciences in Colleges, 2004. 19(5): pp. 140 – 153.

[10] D. Shreiner, M. Woo, and J. Neider, OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2, 3rd ed. Addison-Wesley Longman, Amsterdam, 1999.

[11] Z. Al-Sharif, and C. Jeffery, "Adding high level VoIP facilities to the Unicon language," Proc. Third International Conference on Information Technology: New Generations, ITNG 2006, pp. 524-5