Programming Language Support for Collaborative Virtual Environments

Clinton Jeffery, Omar El-khatib, Ziad Al-sharif
Department of Computer Science, New Mexico State University
{jeffery,okhatib,zsharif}@cs.nmsu.edu

Naomi Martinez
Department of Mathematics, University of Michigan
naomiam@umich.edu

Abstract

Collaborative virtual environments (CVEs) are a very popular form of entertainment. The high cost of developing CVEs has limited their use in small scale or experimental domains. The 3D graphics, network communications, and audio features of CVEs pose significant challenges for developers. This paper describes very high-level language support for 3D graphics that facilitate rapid development of experimental CVEs.

Keywords: collaborative virtual environments, 3D graphics, programming languages

1 Introduction

Games such as *EverQuest* use computers with 3D graphics cards and internet connections to create virtual environments in which users manipulate complex virtual objects. Collaborative virtual environment (CVE) technology has enormous potential, but the cost of development limits the rate of experimentation and restricts the pool of developers to larger organizations. Very high-level language support for graphics, audio, and networking abstractions can reduce the cost of developing CVEs and make them economical for new application domains. APIs such as OpenGL [7] enable construction of portable 3D applications, but these APIs are very large and complex. Sun's Java3D web site claims that API enables ordinary programmers to write 3D graphics, but the size and complexity of the API (150+ classes) belies that assertion.

This paper describes general purpose programming language features that support the rapid development of CVEs. The features are implemented in a very high-level applications language called Unicon [5], a successor to Icon [4]. Language extension was used to explore the combination of CVE features in optimizing control structures and system support, and to allow the notation for CVE objects to be concise, comparable to arrays or strings.

This paper focuses on language features. Unicon CVEs also use class libraries that model the virtual environment in an extensible, very high-level language; a CVE client/server architecture; tools for building virtual worlds; and a reference implementation CVE; see Section 3.

Features that provide access to general-purpose capabilities that are ubiquitous on modern hardware and software platforms deserve language level support in the form of types, control structures, operators; as well as integration and optimization within virtual machines and runtime systems. The language features added to Unicon to support CVEs are 3D graphics, networking, and audio. They can be used varyingly in many different domains and CVEs. This paper focuses on the 3D graphics aspects.

2 Graphics Support for CVEs

Unicon's 3D interface extends a set of very high-level 2D graphics facilities [3]. Built using OpenGL, Unicon's 3D interface runs on multiple platforms and provides a level of abstraction suitable for programmers with little graphics expertise. Because the design is not a "wrapper" for the OpenGL library, it is not limited by OpenGL's C-based paradigms, but is free to integrate 3D as needed throughout the virtual machine and runtime system of the language.

2.1 Design Rationale

Unicon 3D supports graphics primitives, transformations, lighting, and textures, emphasizing ease of learning and use at a higher level than OpenGL. The use of image files as textures is simplified, the manipulation of colors, material properties and textures is integrated, and a control structure called *transform environment* simplifies matrix stack maintenance.

OpenGL contains 250+ 3D graphics functions; programs also perform many window system calls to open/close windows and handle user input. Unicon 3D reduces OpenGL's API down to 19 new functions and 6 extensions to preexisting 2D graphics functions. The simplification comes from a flexible type system, support for variable numbers of parameters and default parameter values, bundling many OpenGL calls into atomic operations, and omitting seldom-used or unnecessary features. To draw a polygon in OpenGL, the user calls `glBegin()`, followed by calls to `glVertex()` for each vertex; the actual polygon is rendered with a call to `glEnd()`. In Unicon 3D, a call to `DrawPolygon()` performs this task.

2.2 Primitives and Attributes

Scenes are drawn as sets of primitives in an (x,y,z) coordinate system. The objects visible on-screen depend on the eye position and direction. The Unicon 3D primitives are cubes, points, lines, spheres, torii, cylinders, disks, and various polygons. Polygon vertices are supplied in various "mesh modes" (simple, triangle fan, quad strip, etc.). Modest semantic improvements are made compared with OpenGL; for example drawing of concave polygons is not part of OpenGL. Drawing functions use a default window which can be overridden by an an optional first parameter.

Unicon uses attributes, read or set via the function WAttrib(), to control drawing details without introducing a large number of functions or struct/class fields. Attributes are encoded as strings to simplify the interface; they control colors, textures, line styles, fonts, and camera features such as eye position. A change to one of these attributes looks like WAttrib(w,"eyepos=0,0,7.2"). When an eye attribute changes, the scene is redrawn.

2.3 Transform Environments

Matrix multiplications compute rotations, translations, and scaling on objects. Compound objects are rendered by applying transformations hierarchically. OpenGL addresses matrix stacks well and yet Unicon 3D allows substantial simplification over OpenGL. PushMatrix() and PopMatrix() pairs are extremely common, and programmers often fail to match them, motivating a new control structure in Unicon 3D called a *transform environment*. The syntax

```
window : expr
```

evaluates expr, wrapped in a PushMatrix()/PopMatrix() pair. Transform environments may nest. If a window is supplied, it becomes the default window for the evaluation of *expr*. This is analogous to the string scanning control structure that originated in the Icon language. The *expr* in a transform environment is usually a compound expression. The PopMatrix() will be performed even if the code fails or returns from within *expr*, avoiding a common source of bugs in 3D programs. Transform environments also improve code readability.

2.4 Colors, Lights and Materials

Lighting helps a scene appear to be 3D, but adding lighting can be complicated. There are three types of light: ambient, diffuse, and specular, corresponding to non-directional, directional, and reflective properties. Unicon 3D attributes light0..light7 control up to eight lights. Each light is on or off and has diffuse, ambient, specular, and position components.

Each object in a scene may have material properties that describe its color and appearance under various lighting. The material properties are ambient, diffuse, specular, emission, and shininess. Objects can emit light of a specific color. Using combinations of these material properties one can give an object the illusion of being made of plastic or metal. Material properties extend the foreground color attribute used in drawing. In addition to numeric RGB formats, Unicon uses Icon's rich color naming system [3]. Natural color names such as "deep purplish blue" are an alternative to RGB color specifications, and Unicon 3D provides similarly intuitive means of specifying materials in CVEs. Material properties are specified in strings containing up to four semi-colon separated *property components* in the following format, plus an optional numeric *shininess component*.

$$\left[\begin{array}{c}\texttt{ambient}\\ \texttt{diffuse}\\ \texttt{specular}\\ \texttt{emission}\end{array}\right]\left[\begin{array}{c}\texttt{opaque}\\ \texttt{dull}\\ \texttt{translucent}\\ \texttt{subtransparent}\\ \texttt{transparent}\end{array}\right] colorname$$

In order to get red objects a programmer can call Fg("red"); a more full-blown material properties specification can also be expressed conveniently, as in the following example.

Fg("diffuse gray; emission light vivid green")

2.5 Textures

Textures are important in applications such as CVEs, giving a scene its "feel", ranging from cartoon to photorealistic. Textures are 2D images that replace or blend with material properties of 3D objects. Textured objects' appearance depends on the texture image and *texture coordinates* that map parts of the texture image to parts of the object. Unicon 3D texture images can be another Unicon window, an image filename in a format such as JPG or GIF, or a string encoding of the image in a literal format such as "width,palette,data". A rich set of generic textures can be specified by a texture name consisting of the form

material property texture

where *material property* is as described above, and *texture* is one of the following 22 textures: stone, brick, sand, glass, cloth, wood, dirt, marble, concrete, tile, leaf, grass, carpet, skin, hair, metal, water, clouds, grill, iron, plastic, leather. CVE objects can be drawn using generic textures such as `"wood"` or more interesting texture names such as `"dark green brick"`. This aids prototypes, prior to obtaining and refining more precise texture images.

Using another Unicon window as a texture allows the program to create a texture image dynamically. This enables virtual whiteboards and computer screens containing textures that show the dynamic content of other textual or 2D or 3D graphic applications. Textures must have a height of 2^n and width of 2^m pixels where n and m are integers. If not, Unicon 3D scales the texture down to the closest power of 2. Rescaling slows the application and may cause visual artifacts, so it is wise to make textures with appropriate sizes in the first place.

2.6 Animation

Animation is performance sensitive; Unicon is slower than systems programming languages, but nevertheless supports the 3D animations needed by CVEs. Unicon's VM falls somewhere between Java and scripting languages in performance, but VM speed is not a problem when performance is limited mainly by OpenGL rendering speed. The language runtime system serves as the "game engine" and runs at the speed of its C code, and the CVE semantics is running at the speed of the VM.

Animations redraw the entire scene each time any object moves or the user changes point of view. Rather than calling EraseArea() followed by the appropriate graphics primitives, Unicon programs usually allow the runtime system do the redrawing. Unicon maintains an application level display list of graphics operations to execute whenever the screen must be redrawn; the list contains everything since the last EraseArea(). The elements of the list are Unicon records and lists containing the string names and parameters of graphics primitives. For example, a call to DrawSphere(w,x,y,z,r) returns (and adds to the display list) a record gl_sphere("DrawSphere", x, y, z, r).

Instead of redrawing the scene to move an object, Unicon programs modify the object's display list record(s) and call Refresh(). The following code fragment illustrates animation by

sliding a ball up and down. In order to "bounce" the program would need to incorporate physics.

```
sphere := DrawSphere(w,x,y,z,r)
increment := 0.2
every i := 1 to 100 do
   every j := 1 to 100 do {
      sphere.y +:= increment
      Refresh(w)
      }
```

It is easy to modify objects' attributes between frames with high frame rates. Performance results are encouraging; in practice Refresh() is seldom called since changing the camera position via Eye() performs an implicit refresh. The display list (modified implicitly via variable `sphere` above) is also used to insert or remove complex objects in a scene, without rerendering them; it is easy to manipulate because it is an ordinary Unicon list. CVEs maintain subsets of the display list corresponding to rooms, tables, etc. The transform environment control structure produces the sublist of display list objects rendered during its evaluation.

In implementing animations with Unicon 3D, we observed that on the *same hardware*, with vendor drivers, OpenGL scaled to large numbers of objects much better under one operating system than under another. Also, most 3D applications' use of graphics primitives is highly repetitive. A typical CVE display list of 1000 elements is reduced by 50% with the addition of a peephole optimizer to the display list.

2.7 3D Example

This example illustrates many 3D graphics features described above. Because this is program source code and not a data file, programmability and flexibility are retained for CVEs.

Textures can be captured using a digital camera; direct use of image files makes adding textures easy. Editing is usually needed to adjust the image resolution or improve its quality.

```
procedure main()
   &window := open("casa", "gl",
      "bg=black","size=700,700")
   Texture("carpet.jpg") # floor
   FillPolygon(-7,-0.9,-14,-7,-7,
     -14,7,-7,-14,7,-0.9,
     -14,3.5, 0.8,-14)
   Texture("wall1.jpg", # r. wall
          0,1,0,0,1,0,1,1)
   FillPolygon(2,4,-8, 8.3,8,-16,
          8.3,-1.2,-16,2,0.4,-8)
   Texture("wall2.jpg") # l. wall
   FillPolygon(2,4,-8,-9,8,-16,-9,
          -1.2,-16,2,0.4,-8)
   Texture("poster.jpg") # picture
   FillPolygon(1,1.2,-3,1,0.7,
    -3,1.2,0.5,-2.6,1.2,1,-2.6)
   Texture("unicorn.jpg", # pic. 2
          1,0,0,0,0,1,1,1)
   FillPolygon(0.8,2,-9,-3,
     1.6,-9,-3,3.9,-9,0.8,4,-9)

   # Draw lamp in transform environ
   :{ Translate(0.7,0.20,-0.5)
      Fg("emission pale yellow")
      # nested environments
      :{ Rotate(-5,1,0,0,5,0,0,1)
         DrawCylinder(-0.05,.57,-2,
                      .15,.05,.17)}
      Fg("diffuse grey")
      :{ Rotate(-5,1,0,0,6,0,0,1)
         DrawCylinder(0,0,-2.5,.7,
                      .035,.035)}
      :{ Rotate(6, 0, 0, 1)
         DrawTorus(-.02,-.22,-2.5,
                   .03,.05)}
   }

   Texture("t2.jpg","auto") # table
   :{ Rotate(-10, 1, 0, 0)
      DrawCylinder(0,.2,-2,
                   .1,.3,.3) }
   :{ Translate(0,-.09,-1.8)
      Rotate(65, 1, 0, 0)
      DrawDisk(0,0,0,0,.29)}
```

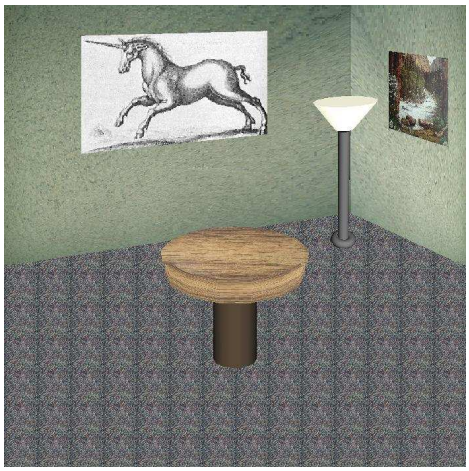

Figure 1: A texture-mapped scene of a room.

```
    Fg("diffuse weak brown")
    :{ Rotate(-20, 1, 0,0)
       DrawCylinder(0,.2,-2.2,
                    .3,.1,.1)}
     while Event() ~== "q"
end
```

3 An Example CVE

Unicon has been used to implement a simple CVE called NSH (New Science Hall) that supports distance education courses, and also enables virtual meetings and labs with local students. The core CVE client was written in about 5K lines of code, demonstrating the potential of these language facilities for CVE construction. Subsequent development of a collaborative IDE and other features added more lines of code than did the core CVE facilities. NSH avatars are simple, customizable communication tools (Figure 2). Avatars can point, have an identifying label, and visibly indicate when chatting is performed. Users provide a GIF or JPG image to present their face inside a rectangle or texturemapped within an egg-shaped head.

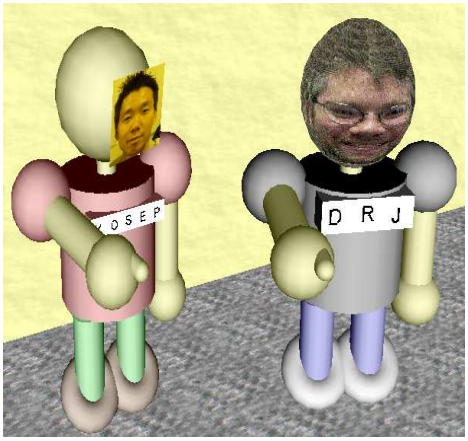

Figure 2: Avatars

NSH's graphical environment is cartoon-like, rather than photorealistic as for CVEs such as *Le Deuxieme Monde*. Students will not have eye contact, but may have less fear of embarrassment while asking questions. Figure 3 shows an example scene from this environment.

NMSU's electronic classroom features custom local software that feeds the virtual whiteboard content up in Adobe SVG (an XML) format via HTTP, where it is available to the

Figure 3: Virtual Academia with Integrated Whiteboards, Chat and Voice

CVE along with other SVG plugin-equipped browsers. A collaborative development environment prototype was also developed, shown in Figure 5. Users edit text, watch each other, and chat. Collaborative views of other areas of software development are under construction. Instructors can walk around a virtual lab, talk with students, and look at their virtual screens, zoom in to the collaborative environment when questions require on-screen details.

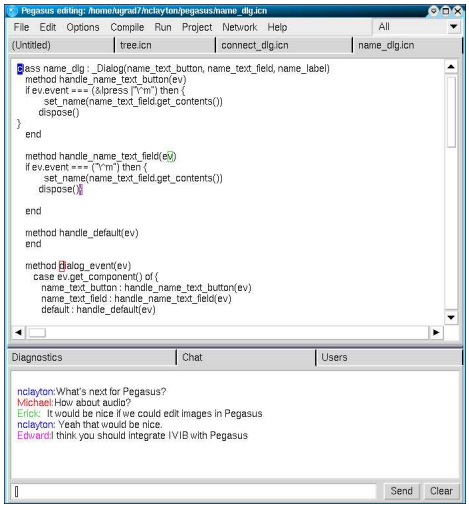

Figure 4: A collaborative IDE

4 Related Work

OpenInventor [9] is a C++ API built atop OpenGL at a high semantic level; unfortunately it is large and complex, and is not as ubiqui-

tous as OpenGL. InvenTcl [1] and iVRS [6] are Tcl bindings for OpenInventor and VRS that allow prototyping 3D applications. InvenTcl and iVRS wrap a C++ 3D toolkit for another language, instead of aiming for higher level features or API simplification.

VRML and X3D are scene description languages based on OpenInventor's object model that are more widely supported than OpenInventor. They run inside a web browser, offering limited capabilities for dynamic content, but despite this have been extended in production CVEs such as Cybertown (cybertown.com) and systems developed at the U.S. Navy's MOVES Institute www.movesinstitute.org

Java's 3D API is a little easier than OpenGL, but learning to write Java 3D programs is difficult [8]; Java 3D introduces a hiararchy of around 150 classes. VPython (vpython.org) provides 3D support that is not a "wrapper" layer on top of a lower-level API; it bundles an IDE for 3D program development, along with a fast vector library, but is not aimed at CVEs since it is not integrated with networking or audio.

The DIVE framework allows rapid CVE development [2]. Rather than extending a VM and runtime system with CVE-oriented features as in this paper, DIVE's CVE engine is extended using the Tcl scripting language. This is a closer-to-mainstream approach to rapid development of custom CVEs, but Tcl is famously slow, discouraging the use of extensive dynamic behaviors by large numbers of objects.

5 Conclusions and Future Work

Unicon requires programmers to learn only basic CVE concepts and a few 3D functions to write CVE client code. This is important when technical staff consists a small number of average programmers devoting fractional time to a project, instead of the army of professional developers that might build a CVE for industry. Unicon is often used for rapid prototyping, but the facilities scale well to production applications except where CPU performance is critical.

The facilities described in this paper reduce the amount of code to implement simple CVEs, while retaining the flexibility of a very high-level language. The features developed will also be useful in experimental visualization or groupware applications. Future work at the language level will include additional performance tuning, support for new 3D platforms, and development of protocol and control structure support for network fluctuations and faults.

6 Acknowledgments

The NSH CVE is being developed by W. Winkler, N. Clayton, A. Dabholkar, K. Tachtevrenidis, Y. Kim, and R. Ramagiri and others. This work was sponsored in part by the Alliance for Minority Participation, and by NSF grants EIA-0220590, EIA-9810732 and DUE-0402572.

References

[1] S. Fels and K. Mase. InvenTcl: a fast prototyping environment for 3d graphics and multimedia applications. In *Advanced Multimedia Content Processing*, pages 161–176, 1998.

[2] E. Frecon. DIVE: A Programming Architecture for the Prototyping of IIS. In *Inhabited Information Spaces*. Springer, 2004.

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

[4] R. E. Griswold and M. T. Griswold. *The Icon Programming Language*. Peer to Peer Communications, San Jose, 1996.

[5] C. Jeffery, S. Mohamed, R. Pereda, and R. Parlett. *Programming with Unicon*. Unicon Project, unicon.org, 2005.

[6] O. Kersting and J. Dollner. Interactive 3d graphics for Tcl. In *2002 FREENIX*, pages 1–12. USENIX, June 2002.

[7] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading, Mass., 1993.

[8] D. F. Savarese. Learning to Fly. *JAVAPro*, June 2003.

[9] J. Wernecke. *The Inventor Mentor*. Addison-Wesley, Reading, Mass, 1994.