PERGAMON

Computers & Graphics 27 (2003) 605–615

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

Technical section

The remote application controller

James R. Miller*

Electrical Engineering and Computer Science, University of Kansas, 415 Snow Hall, Lawrence, KS 66045, USA

Accepted 15 April 2003

Abstract

Many modern design, visualization, and decision support systems involve interdisciplinary working groups. Common requirements to support the efforts of such groups include shared graphical displays and the ability for all involved to interact with the visualizations. Much effort has been expended on the visualization side; somewhat less has been reported on the interactive control side. In this paper, we describe the "remote application controller" (RAC), a Java application running on a personal device that can locate and establish communication with other applications running on the network. The RAC serves two related purposes in this context. When running on personal handheld devices, it allows a group of individuals sharing the same display space to also share control of the applications running in that space. When running either on such devices or on standalone workstations, it allows remote collaborators—in conjunction with another distributed tool we mention—to have separate input and output ports to a visualization application running at a remote location. This effort was initiated before the Jini toolkit from Sun was widely available. Jini provides support for some of the capabilities built into our prototype RAC, hence we are currently reworking those portions to exploit the Jini toolkit and other interactive control and configuration paradigms enabled by this newer technology. Finally, a nice side benefit has been that the RAC architecture provides one easy way to develop effective cross-platform GUIs for arbitrary non-Java-based visualization applications, including the ones originally developed to be run by a single user in a non-collaborative context.
© 2003 Elsevier Ltd. All rights reserved.

Keywords: Distributed interaction techniques; Collaborative work environments

1. Introduction

Collaborative computer-based visual work environments require a mechanism for several cooperating users to control a shared set of applications in a shared context in a convenient and comfortable fashion. On the output side, this requires a large display so that users can be comfortable in the environment instead of being forced to huddle around a single display station. The input side is equally important; a single mouse, or tablet, or keyboard simply does not work well in a collaborative environment. Everyone wants to point somewhere and alter something at the same time. To begin to address this need, users in the environment must have their own personal devices to allow them to interact with objects on the shared display without having to fight over "who gets the mouse".

In this paper, we describe the *collaborative visualization room* (*CVR*), an environment we have created which serves as a part of a testbed for these and other collaborative visualization and interactive techniques. Specifically, we describe the use of a wireless personalized input device which allows multiple users to interact with a shared set of applications in a multi-language (C, C++, and Java), multi-computer, and multi-display environment. We also mention some ongoing projects that are exploiting various portions of this environment.

While our primary motivation was to facilitate the group interactions required in a collaborative decision-making environment, a secondary benefit arose in the context of adopting a common approach for the

*Fax: +1-785-864-3226.

E-mail address: jrmiller@ku.edu (J.R. Miller).

0097-8493/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0097-8493(03)00104-3

development of useful GUIs for general applications. OpenGL [1] has emerged as a cross-platform standard for graphics programming for complex modeling and visualization applications. The OpenGL API is dominated by output graphics capabilities; however, the input side tends to be more platform-dependent with several competing APIs built on top of OpenGL. The Swing utilities associated with Java 2 provide an interesting alternative useful not only for collaborative applications, but for single-user ones as well. Using the approach described in this paper, we can deliver effective user interfaces for single or multiple user applications that are as platform-independent as OpenGL. Our Swing-based remote application controller (RAC) learns the capabilities of a program and presents an appropriate GUI, either on a personal input device (PID[1]) or simply in a window that can be dragged next to the OpenGL application itself.

2. The collaborative visualization room

The CVR contains a wall-sized display using a curved screen 25 ft wide $\times$ 6 ft height (Fig. 1). It provides a 120° field of view when viewed from a distance of 12 ft. Three overhead projectors, each of which is driven by a Silicon Graphics InfiniteReality 2 display subsystem, produce a combined display resolution of 5760×1200 pixels.

Ongoing projects in this room have regularly involved 8–12 active participants. More could be accommodated if necessary. The application illustrated in Fig. 1 is a climatology-based decision support system. It was to support the needs of applications like this that we launched the efforts described here.

3. Related work

To support concurrent collaborative interaction with a group of users, personalized input devices are required. Handheld personal devices are certainly common. Moreover, their use in collaborative work environments is not new. For example, the *NoteLook* system [2] allows users in meetings to use wireless technology to import slides used by the presenter, add annotations to them, import video, and take conventional notes. The primary data flow is one-way, however, and no attempt is made to alter the collaborative environment based on input generated by the personal devices.

The Interactive Workroom (iRoom) is a multi-display collaborative environment in which PDAs represent one of several ways users can collaborate and interact with a variety of shared applications [3]. An *event heap* manages the communication among the various system components, allowing client applications to broadcast messages as well as query for and/or subscribe to various events of interest. Their system architecture "favors platform-neutral languages and development environments", hence they seem to favor Java applications over, for example, C++. By contrast, the vast majority of the applications and tools we use have been developed in C++, and we could not afford to recode those components. Hence we choose to develop an architecture that explicitly handles a mixture of languages, including C, C++, and Java.

Brewer et al. [4] describe a collaborative visualization environment for use in the geosciences for applications very similar to one of our primary motivating applications. They followed a well-established development approach based on various levels of user evaluation and feedback [5]. While their work focused on collaborative interactions, apparently no personalized input devices were used. Instead, all participants were assumed to be in front of workstations when interacting. One interesting part of their user interviews related to the issue of individual control. While acknowledging the potential use, the participants seemed to be worried about conflict arising from this joint control. Two went so far as to assert that having one person in control is preferable [4]. Our own experiences have been different. We frequently deal with the awkwardness of individuals needing to swap seats to get to the mouse to make a point.

The goal of the Pebbles Project is the development of multi-machine user interfaces [6,7]. This work involves a variety of PDAs connected to PCs. Several demonstration applications have been developed. The one closest to what we are doing is their "Remote Commander" application that allows multiple users to share control of a PC application by taking control of the mouse and keyboard. Three different ways of accomplishing this are provided. Each user can take turns controlling the mouse and keyboard; each user can have their own personalized cursor that "floats above" all application windows; or each user can have a custom cursor, but only for custom applications. This approach seems to be closely tied to a PC application architecture and is in any event too low level for our purposes. We wish to export the entire user interface to the PDA, and let users' input devices communicate at a higher level, rather than having them simulate mouse and keyboard input.

4. High-level approach

We assume a PID based on a wireless handheld computer supporting a Java virtual machine and the Swing GUI toolkit. An early prototype used a Xybernaut [8] for this purpose (Fig. 2(a)). As implied above, the RAC software itself is platform-independent, hence

[1] In this paper, we use "PID" as a generic description of a Personal Input Device, one specific type of which is a PDA.

Fig. 1. The CVR, illustrating one of the applications mentioned in this paper.

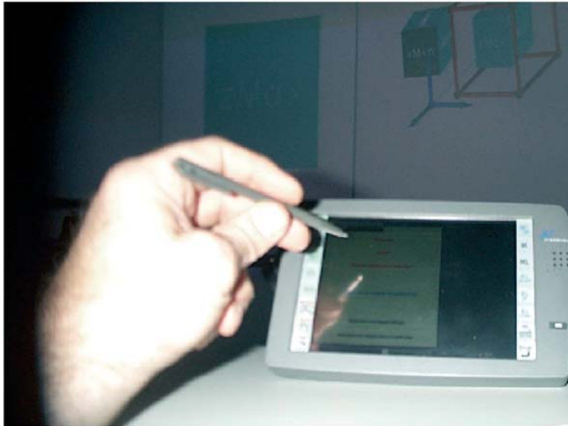
we also run it on various laptop computers and workstations. More recently, we have ported it to the Compaq iPAQ which has become the preferred PID device (Fig. 2(b)).

The RAC itself is a Java-Swing program. Currently, the user interface is identical on all platforms (modulo platform-dependent window dressing). To initiate remote control of an application, the RAC first obtains a specification of the program's control protocol, including menus and basic 3D dynamic transformation control requirements. The RAC has certain hard-wired control panels that it instantiates as needed based on the control protocol specifications it receives. Care was taken to make sure that the panels always appear in their entirety regardless of device. For many basic applications, this has proven to be satisfactory. It is not ideal, however, and some of our more advanced applications are in fact difficult to control due to the small screen space on the handheld and more sophisticated requirements for control protocols. Current effort is targeted at developing a higher-level XML-based description of the control protocol for an application which is then parsed at the handheld device. We expand more on this in Section 8.

The mechanism we adopted to get a handheld device to talk to a remote application involves a directory server that maintains a registry of running programs supporting a common external control interface. When such a program starts, it registers itself with this directory server. When an RAC enters an environment, it knows where to look for this directory server and how to ask it for a list of candidate programs to be controlled. Once a program is selected, direct communication between the program and the RAC takes place. The first communication from the selected program to the RAC is the specification of the program's specific control protocol.

The user need not know anything about servers or how to run programs. They simply execute the RAC program on their device, select an application from a list presented by the RAC, and go about their work. Moreover, an arbitrary number of users in the environment can each have their own such device and thereby share control of a common set of applications running in the environment. We use the low-level socket mechanism to prevent two users from simultaneously controlling the same application. In our experience to date, the sociology of the situation tends to make this sort of sharing work. Individuals relinquish control to allow other people to have a chance. We are currently investigating more flexible approaches to mutual exclusion that could allow multiple users to simultaneously control the same application in a "non-destructive" fashion by having the application place collections of controls into groups such that those in one group will not interfere with those in another. We return to this in Section 8.

Our application registration and discovery process is very similar in spirit to what is now available in Jini


(a)

(b)

Fig. 2. (a) A xybernaut running the RAC; and (b) the RAC running on an iPAQ.

[9,10]. The Jini architecture recently developed for Java applications provides a powerful superstructure on top of the Java Remote Method Invocation (RMI) API. Using Jini, Java service providers can discover network lookup services with whom they can register the services they provide. Java clients also know how to discover the lookup services and query them for providers of desired services. Once a client learns about an appropriate service provider, direct communication between the client and the server is established via special *service proxy* objects using RMI.

As we learned about Jini, we decided to rearchitect those portions of the system to use it since Jini not only provided standard technology for a critical piece of our system, but also we could see several additional advantages that could be realized by incorporating this technology. The major stumbling block was that the Jini architecture was designed for pure Java environments. Our environment is multi-language, primarily C and C++ with some Fortran and Java. By combining the approaches of our current RAC with the distributed Jini tools, we believe we can achieve the best of both worlds.

In the remainder of this paper, we discuss the RAC, first from the perspective of the user, then at the implementation level. The implementation discussion includes not only how the RAC was implemented, but also what users have to do to allow their applications (either legacy or new) to be controlled by an RAC.

5. User's perspective

5.1. Startup and application selection

Interactive applications written to allow external control by RACs are started in the usual way. In our case, this frequently involves running them on the large shared screen in our CVR as illustrated in Fig. 1. Other options include running the application on a traditional workstation with the RAC either running in an adjacent window, on a personal computer, or on a PID.

When running in the CVR, collaborators in the environment launch a copy of the RAC on their personal devices. As an RAC starts, it scans the environment and obtains a list of all interactive applications willing to be externally controlled. It then presents a welcome screen that has a series of tabs corresponding to the various applications detected (Fig. 3[2]). Periodic polling keeps the list current as new applications are launched and/or current ones terminate. Users click the tab for the program they wish to control. Only one user at a time can be using the controls associated with a particular tab. We will define more precisely what we mean and how this works in Section 5.3.

5.2. Controlling an application

The example of Fig. 3 shows tabs for some simple OpenGL programs running in the environment. The samples in that figure are programs used in classes here to illustrate various concepts in curve and surface modeling as well as how shadows can be simulated in OpenGL. Upon selecting one of these tabs, the RAC establishes communication with the corresponding program (actually with a particular *control context* within the program—we will return to this distinction shortly). It then presents the user with a lower-level tabbed pane, each tab providing a different type of

[2] For convenience in generating screen shots of the RAC, we captured its images running on a Macintosh computer.

Fig. 3. The RAC welcome screen.

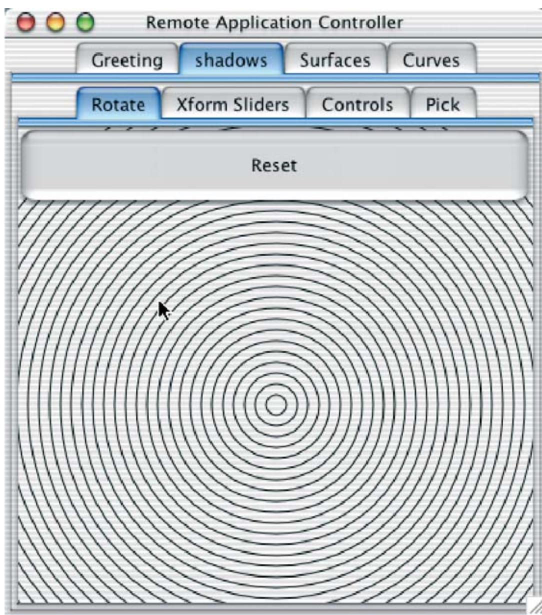


Fig. 4. A rotation pad in an RAC.

control allowed by the program. Figs. 4 and 5 show different control panels available after the user selects the "shadows" application to control.

The "shadows" program itself is illustrated in Fig. 6(a). This program allows dynamic affine transformations of the scene, hence the RAC controls for this program include the pad-based rotation scheme of Fig. 4 as well as the explicit slider-bar transformation control

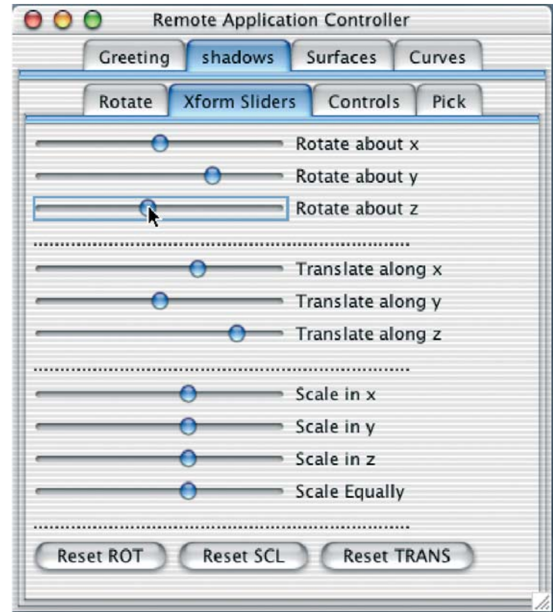


Fig. 5. Slider bars for rotation, scale, and translation.

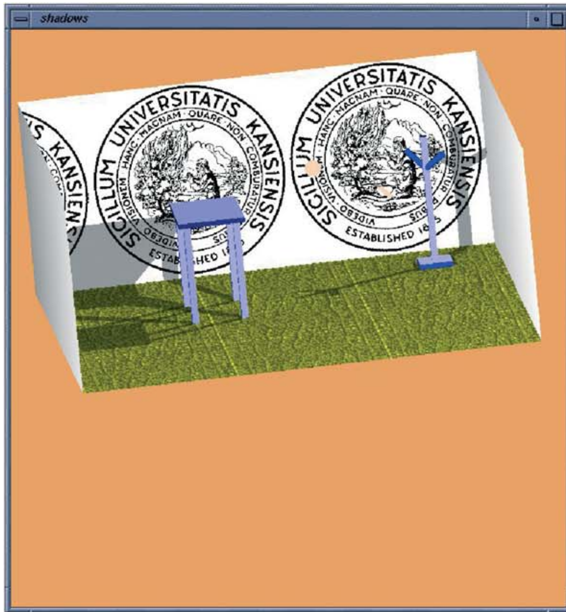
mechanism illustrated in Fig. 5. As the user moves the cursor around the rotation pad, the movements are translated into rotations about the x and y screen axes.

The light sources in the shadows program can be selected and dragged. The RAC provides a pick interface to access this capability. When that tab is selected, a grid pad is presented (Fig. 6(b)) which causes cross-hairs to be drawn on the screen as in Fig. 6(c). Once selected, the light can be dragged to a new position by the RAC as indicated in Fig. 6(d).

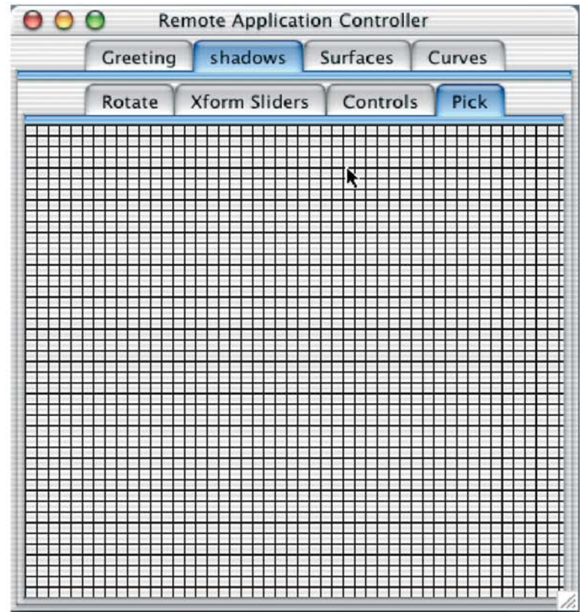
Fig. 7(a) illustrates the "surfaces" rendering context of the "freeform" program. This program demonstrates basic manipulations on Bezier and Rational Bezier surfaces such as degree raising, subdivision, piecewise construction, and so forth. It allows the same dynamic screen rotations as does the "shadows" program, and, like the light sources in the "shadows" program, surface control points can be picked and dragged to modify the surface shape. In addition, the surfaces program defines a menu hierarchy for accessing the various operations. Fig. 7(b) shows a user working with the corresponding control context on the RAC to request that the degree in the u parametric direction of the current surface patch be raised.

5.3. Mutual exclusion issues

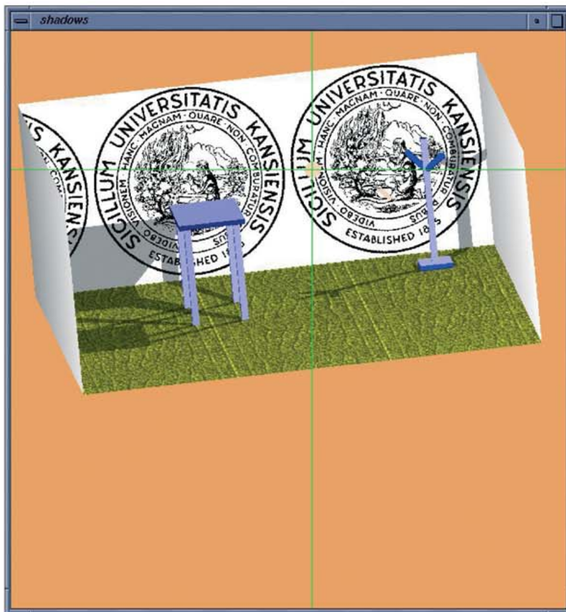
While multiple users can each be controlling a shared set of applications simultaneously, some mutual exclusion is required to avoid destructive interference among the various users. We define a *control context* as the basic unit of mutual exclusion. In the current imple-



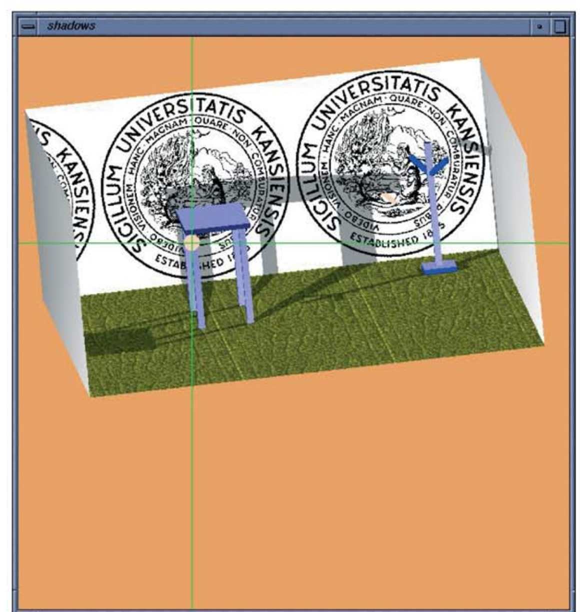
(a)



(b)



(c)

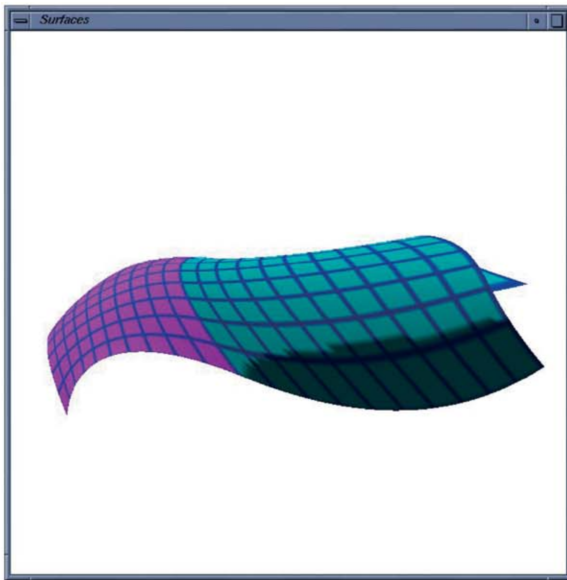


(d)

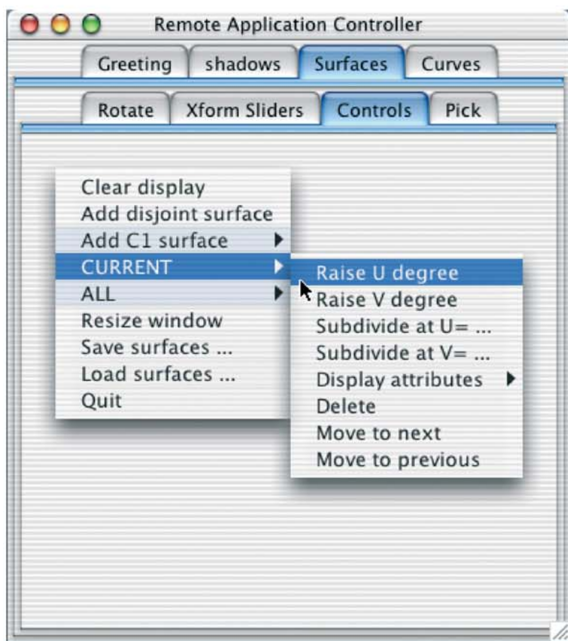
Fig. 6. (a) The "shadows" OpenGL demonstration program; (b) a picking pad in an RAC; (c) selecting a positional light to be moved using the RAC picking pad; and (d) the scene after the light has been dragged to a new location via the RAC.

mentation, we create one control context for each OpenGL rendering context. That is, there can be one control context for each window created by an OpenGL application. The "Curves" and "Surfaces" control contexts shown in Fig. 7(b), for example, actually belong to a single OpenGL program.

Only one user at a time can be interacting with any control context of a given application. A user relinquishes control of a control context, either by selecting a different control context, or by returning to the welcome screen of Fig. 3. In Section 8, we discuss our plans to relax this restriction.



(a)



(b)

Fig. 7. (a) The *Surfaces* rendering context in the *freeform* program; and (b) the control context for the *Surfaces* rendering context.

6. Implementation

Our approach requires that programs be shared in the collaborative space register themselves with a directory server and describe the means by which they are willing to be externally controlled. That is, they describe their menu structure, supported 3D interaction controls, and so forth. The RAC running on each user's PID knows how to find the directory server, search for such programs and control specifications, and establish direct communication on demand. As we will see below, the demands placed on the applications themselves are minimal. A C++ class (one instance of which is instantiated for each control context) hides all but the most application-specific of the tasks from the programmer.

There are three major components to the system: the RAC itself, which is a standalone Java application; an interface mediating communication between an RAC and (a control context of) an application; and the program registry. The interface is largely specified and implemented by a C++ abstract base class called *ExternalIF*. Concrete subclasses of *ExternalIF* are defined for various window interfaces. At this time, we support two: *GlutExternalIF* is created by OpenGL programs using the GLUT window system interface; *SdkExternalIF* is used by programs requiring the OpenGL SDK multi-pipe development environment [11]. The examples in Section 5 used the former; our climatology application discussed in Section 7 uses the latter.

Once created, an instance of the *ExternalIF* object is associated with a control context, manages a shared memory area, interfaces with RACs over socket connections, presents a common interface to modifiable program data in the shared memory area, and provides tools to help a program follow the conventions required in order to be a part of this framework. The remainder of this section discusses these conventions and the use of the *ExternalIF* class hierarchy.

The overall process works as follows. An application willing to be controlled by our RAC first uses the *ExternalIF* object to register itself with the program registry (Fig. 8). When an RAC starts, it asks the program registry for a list of all currently executing programs that are using the *ExternalIF* conventions for remote control. Based on the response it receives, it creates tabbed panels for selection by the user as illustrated in the welcome screen of Fig. 3. A user indicates they want to use a particular control context (for example, the *shadows* control context of Fig. 6) by selecting the corresponding tab. This leads to the lower-level tab pane as discussed above which is then used to control the selected application.

All communication between the RAC and a given control context is performed through sockets. Our socket protocol running on the RAC was derived from a set of Java classes presented in Morelli [12]. The applications we are using in the collaborative environment are generally written in C++ and run in a unix environment. Our socket interface on the C++ side is based on a set of utilities presented in [13].

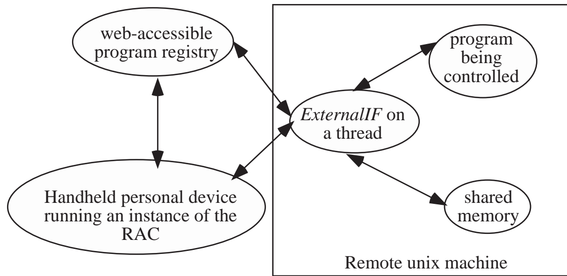


Fig. 8. The RAC architecture.

The interactive application being controlled must be multi-threaded. One thread is dedicated to each control context so that it can listen for communication from an RAC (Fig. 9); the application itself then runs on one or more other threads. The *ExternalIF* object serves as a bridge between these two execution contexts. When something is received from the socket, the listener in the *ExternalIF* object processes the data received, writes appropriate information into the shared memory location associated with the corresponding control context, and signals the main part of the application (by setting a flag in the shared data area) that new data is available for it to process.

An application can request that a control context be terminated at any time. When an application being controlled terminates, the *ExternalIF* object removes each control context description from the web-accessible program registry. The next time an RAC parses the list, it will know to remove the tab for the terminated application while adding new ones for others which may have started since the last time the registry was queried.

Another abstract base class—*Pickable*—is used to specify how the *ExternalIF* handles pick attempts arriving from an RAC. Code in class *Pickable* manages the low-level OpenGL pick process, hiding the render mode transitions and the parsing of the hit array returned. It identifies selected pick identifiers and uses pure virtual methods to handle the application-specific portion of the pick processing. Typically, programmers create their own *Pickable* subclass which would then take over pick processing once pick IDs have been determined.

Summarizing the conventions which must be followed by an interactive application willing to be controlled by an RAC, we note that the application must (refer to Fig. 9):

- register and deregister each desired control context (deregistration is automatic at program termination; earlier deregistration is at the discretion of the application);
- agree to place shared control data in a common location;
- agree to use the mutual exclusion services of the *ExternalIF* class whenever this data is set or used;
- use the interaction control interface of the *ExternalIF* when specifying interactive controls. This interface handles both the specification of the interface to the RAC as well as to the local underlying window manager;

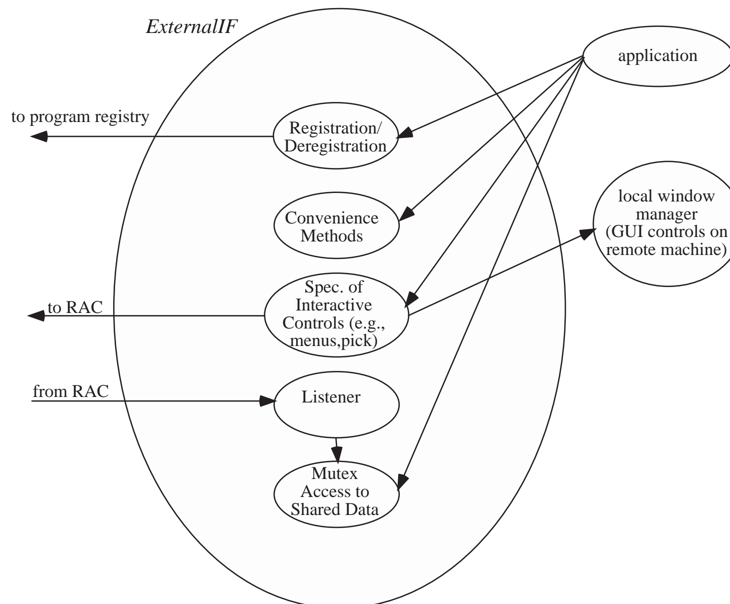


Fig. 9. The architecture of the external interface.

- if picking is desired, define a subclass of *Pickable* and register an instantiated instance with the *ExternalIF*. Different instances can be registered and deregistered at any time during program execution.

The *ExternalIF* class functionality facilitates implementation of these conventions. For example, if an OpenGL application wants to allow itself to be controlled in this way, it does the following operations, typically at initialization time:

- It creates one instance of a concrete subclass of *ExternalIF* for each control context it is allowing to be externally controlled (in our case, either a *GlutExternalIF* or an *SdkExternalIF*).
- It invokes the *start* method of each external interface object when it is ready to allow external control through the corresponding control context. The start method creates the thread on which the external interface will run and registers the control context with the registry server.
- Using an *ExternalIF* class method, the program periodically polls for commands and/or data that may have arrived from an RAC destined for any currently active control context in the program. If it finds any, it processes them and signals the corresponding context in the main part of the application that new control data has arrived.

Finally, we point out that the use of Java for the RAC implementation language has proven to be helpful in a number of ways. For example, we have been able to overcome development problems due to insufficient number of the handheld devices by simply running the RAC in a window on some other workstation or personal computer. This has allowed us to verify issues related to contention and mutual exclusion without having to obtain large number of PID devices.

6.1. Defining the user interface

Like Fox et al. [3], we focused first on integrating this approach into our extensive base of existing legacy applications rather than immediately developing new PID-specific techniques. We wanted to be able to demonstrate quickly the effectiveness of the concept, verify that the approach would be well received by our collaborators, and identify any limiting factors. The design of the *ExternalIF* abstract base class was motivated in part so that legacy applications could simply replace window-manager-dependent function calls with method calls to the appropriate *ExternalIF* subclass. The good news about this approach is that legacy applications required no additional user interface design over and above that already performed, and the level of conversion effort was quite minimal. The bad news of course is that, while the application could then be remotely controlled, no novel PID-specific interface techniques were enabled. We return to this in Section 8.

When designing the user interface to applications, the programmer does not work with Swing or Java at all. In our environment, most applications are written in C++ and use some form of OpenGL for graphics output. To write such an application using the current RAC conventions, one uses the C++ interface presented by the *ExternalIF* abstract base class. At this time, that interface is fairly limited. One can define a hierarchical menu structure and request 3D transformation controls. There is a generic "one size fits all" approach for rendering these GUI controls on the RAC via hard-wired control panels. As discussed in Section 8, now that we have verified the utility of this general approach, we have turned our attention to a more generic interface.

7. Current applications

Among the current research projects employing this environment in the CVR is a scientific visualization application being used for collaborative decision support applications [14]. This application was shown in Fig. 1 in which the area of study was India. The photograph in Fig. 10 illustrates the system being used to examine climatology data for a large part of North America.

The interface between science and decision-making is one of the greatest challenges in information technology research today. Political decision-makers need sound scientific data when making decisions that may have, for example, significant impacts on the environment. Sophisticated computer models can be used to establish scenarios and investigate options, but these simulations are beyond the understanding of most legislators. The collaborative decision support environment we have created allows political decision-makers, members of their staffs, and domain experts to collaborate in an attempt to develop good public policy. During the evaluation of the prototype system, we facilitated several such meetings with groups of varying size and backgrounds [14]. We had politicians, members of their staffs, and various other individuals that could bring different types of expertise to bear. We found that people had to occasionally switch seats so that they could get access to the mouse to make a point. Sometimes people walked up to the screen itself to physically point at some region. Unfortunately, the RAC was not available for use during the evaluations described in [14]. While we have not yet performed as rigorous of a user evaluation as that described in [14], our preliminary experience suggests that this approach is being well received.

A related, but independent effort [15] has led to the development of a technique to distribute 3D scene

Fig. 10. A collaborative climatology scenario analysis application.

descriptions to remote visualization clients. We plan to leverage the RAC and this remote display capability together as a starting point to extend our research in collaborative visualization for decision-making from purely "same time, same place" to "same time, different place" applications. It is our experience that many of the political decision-making contexts of interest to us require such a capability, given that decision-makers and their staffs are frequently not at the same place when they need to confer on some matter.

8. Limitations and future directions

Our initial goal was to develop technology that we could use to evaluate the effectiveness and viability of remotely controlling a shared set of applications using a handheld device. A number of issues were intentionally omitted during this initial evaluation process. For example, menu items that require additional information to be specified (for example, those with "…" on the menu name) are not well supported. When selected on the RAC, dialog boxes or simple prompts simply appear on the remote machine. Obviously this is inadequate, but the problem will be eliminated as we pursue our general future plans discussed in the remainder of this section.

Current effort is targeted at developing a higher-level XML-based description of the control protocol for an application which is then parsed at the handheld device. At the same time, a new interface will be developed for the C++ graphics applications that will cater more towards enabling flexible structured GUI design for new program development as opposed to making it easier to support legacy applications.

On the handheld side, we are developing a smarter RAC that will largely forgo the use of pre-fabricated GUI panels in favor of being able to better understand the control requirements of an application. It can then present realizations of GUI controls specifically designed to minimize required screen space. We are also investigating more flexible approaches to mutual exclusion that will allow a finer level of control than that afforded by a control context.

Ongoing user evaluation will of course be required as we develop these new GUI presentation and control strategies. In our experience to date, we have found remote control of applications via the RAC to be fairly easy, and the turn taking required by the mutual exclusion requirements has not been a significant problem. Nevertheless, we wish to try this out on larger and more diverse groups to see if this experience is more universal.

9. Summary

We have described our CVR and have presented the motivation for a general design of an RAC. Use of an RAC provides one approach for allowing multiple users

in a collaborative environment to share control of a set of concurrently executing applications by using their own personal devices running a copy of the RAC. It also provides a way to achieve platform-independent GUIs for arbitrary applications, collaborative or not. We are currently reworking core pieces of the architecture to exploit the Jini toolkit which promises to improve several management aspects of the RAC architecture. We are also developing more flexible and general application-to-RAC communication strategies that will both allow more generalized control paradigms as well as give responsibility to the RAC to determine the best way to present and monitor user interface components.

Acknowledgements

The work described here was performed in Design-Lab, a multi-disciplinary research laboratory at the University of Kansas funded in part by NSF Grant CDA-94-01021.

References

[1] Woo M, Neider J, Davis T, Shreiner D. OpenGL programming guide, Third ed. Reading, MA: Addison-Wesley; 1999.

[2] Chiu P, Kapuskar A, Reitmeier S, Wilcox L. NoteLook: taking notes in meetings with digital video and ink. In: Proceedings of the ACM Multimedia '99, Orlando, FL, October–November 1999, New York: ACM Press; p. 149–58.

[3] Fox A, Johanson B, Hanrahan P, Winograd T. Integrating information appliances into an interactive workspace. IEEE Computer Graphics and Applications, 2000;20(3): 54–65.

[4] Brewer I, MacEachren AM, Abdo H, Gundrum J, Otto G. Collaborative geographic visualization: enabling shared understanding of environmental processes. Proceedings of the IEEE Symposium on Information Visualization (InfoVis '00), Salt Lake City, UT, October 2000. p. 137–41.

[5] Gabbard JL, Hix D, Swan JE. User-centered design and evaluation of virtual environments. IEEE Computer Graphics and Applications 1999;19(6):51–9.

[6] Myers B. Using handhelds and PCs together. Communications of the ACM 2001;44(11):34–41.

[7] Myers B, Lie K, Yang B-C. Two-handed input using a PDA and a mouse. Proceedings of the Human Factors in Computing Systems (CHI '00), New York: ACM Press; p. 41–8.

[8] Xybernaut. Xybernaut MA IV user's guide. Fairfax, VA, North America: Xybernaut; 1999.

[9] Jini, http://www.jini.org.

[10] Sun, http://www.sun.com/jinni.

[11] Silicon graphics, http://www.sgi.com/software/multipipe/sdk/.

[12] Morelli R. Java, Java, Java! object-oriented problem solving. Upper Saddle River, NJ: Prentice-Hall, 2000 [chapter 15].

[13] Chan T. Unix system programming using C++. Upper Saddle River, NJ: Prentice-Hall, 1997 [chapter 11].

[14] Cliburn D, Feddema JJ, Miller JR, Slocum TA. Design and evaluation of a decision support system in a water balance application. Computers and Graphics 2002;26(6): 931–49.

[15] Tabash J. Towards a framework for distributed scientific visualization. MS thesis, Electrical Engineering and Computer Science, University of Kansas, May 2002.