October 31, 1995

SRC Research Report

130a

Visual Obliq: A System for Building Distributed, Multi-User Applications by Direct Manipulation

Krishna Bharat and Marc H. Brown

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984, we have performed basic and applied research to support Digital's business objectives. Our current work includes exploring distributed personal computing on multiple platforms, networking, programming technology, system modelling and management techniques, and selected applications.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

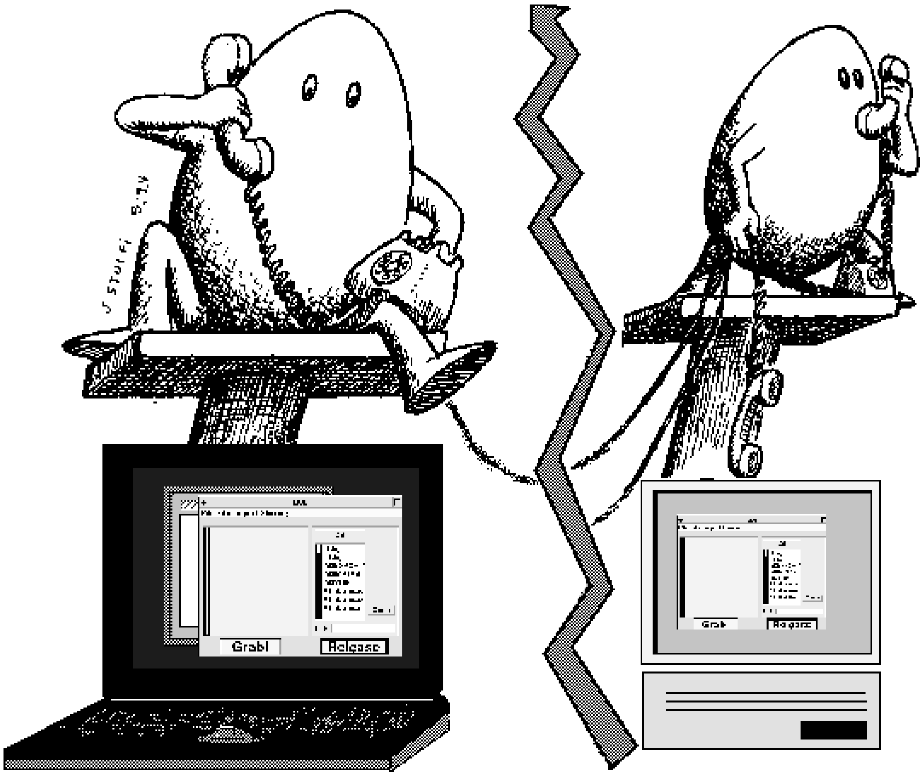
We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Robert W. Taylor, Director

Visual Obliq: A System for Building Distributed, Multi-User Applications by Direct Manipulation

Krishna Bharat and Marc H. Brown

October 31, 1995



Original artwork by Jorge Stolfi;
modified without permission by Krishna Bharat.

Publication History

This report appeared as “Building Distributed, Multi-User Applications by Direct Manipulation,” in the *Proceedings of the ACM Symposium on User Interface Software and Technology* (UIST’94), November 1994, pages 71–81.

The accompany videotape appeared as “Building A Distributed Application Using Visual Obliq,” in the *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI’95), pages 415–416 of the *Conference Companion*.

Author Affiliation

Krishna Bharat is a PhD student in the Graphics, Visualization, and Usability Center of the College of Computing at the Georgia Institute of Technology. Krishna’s electronic mail address is `kb@cc.gatech.edu`. The work described in this report was performed while Krishna was supported by a research internship at SRC during the summer of 1993.

Abstract

This report describes Visual Obliq, a user interface development environment for constructing distributed, multi-user applications. Applications are created by designing the interface with a GUI-builder and embedding callback code in an interpreted language, in much the same way as one would build a traditional (non-distributed, single-user) application with a modern user interface development environment. The resulting application can be run from within the GUI-builder for rapid turnaround or as a stand-alone executable. The Visual Obliq runtime provides abstractions and support for issues specific to distributed computing, such as replication, sharing, communication, and session management. We believe that the abstractions provided, the simplicity of the programming model, the rapid turnaround time, and the applicability to heterogeneous environments, make Visual Obliq a viable tool for authoring distributed applications and groupware.

Contents

1 Introduction

The recent explosion in the quantity and quality of user-interface development environments is simplifying (and making more enjoyable!) the job of programming the GUI part of an application. However, current tools address only single-site and single-user applications. Distributed, multi-user applications are much more complex. Pieces of such applications typically run in separate address spaces, often on heterogeneous machines over a network. The pieces must communicate and synchronize with each other, sharing and replicating data as needed, and handle users at each site.

This report describes Visual Obliq, a user-interface development environment for building multi-user applications. To a first approximation, think of Visual Obliq as a state-of-the-art user-interface development environment, such as Microsoft's Visual Basic, extended to handle distributed applications but *without complicating the programmer's task.*

Visual Obliq consists of a GUI-builder for interactively designing an interface, and run-time support for handling distribution. The GUI-builder allows the user to construct the interface in a standard direct manipulation fashion, and to attach callback code to each widget. The callback code is written in an interpreted language, and can access the Visual Obliq runtime library to handle issues specific to distributed computing. The user who is building the application can run the application from within the GUI-builder, or can have the GUI-builder output a stand-alone executable program. Figure 1 shows the Visual Obliq GUI-builder while creating a shared-editor application.

We believe that the abstractions provided, the simplicity of the programming model, the rapid turnaround time, and the applicability to heterogeneous environments, make Visual Obliq a viable tool for authoring distributed applications.

The organization of this report is as follows. The next three sections describe the system from three perspectives: the end-user, the interface-designer, and the client-programmer. The *end-user* is the person who runs a program built using Visual Obliq. The *interface-designer* is the person who uses the GUI-builder for interactively designing the user interface. The *client-programmer* is the person who actually writes the code for the application. The distinction between these three people is primarily one of nomenclature. Because the GUI-builder integrates the design and testing phases of application development, the interface-designer and the client-programmer is often the same person.

Section 5 presents the infrastructure upon which our implementation is built. Section 6 describes how distribution is implemented, and Section 7 describes how the system can be extended with more widgets. In the remaining sections, we re-

This space is available for a small, negotiable fee.

. (dir)
.. (dir)
GUI–builders.txt
about–Tcl–DP
cover.tex
coversheet.sty
design
fig5.idraw
fig:burger.gif
fig:burger.ps
fig:example.gif

Name : editzone
Type : textedit
Bg Color : VeryLightYellow
Rim : 0
Fg Color : Black
Border : 1
Font : -*-helvetica-bold-*R-*120-*

☐ ReadOnly
☐ Clip
☒ Has Scrollbar
☐ From File :

This space is available for a small, negotiable fee.

Figure 1: The Visual Obliq GUI-builder.

view related efforts, and offer some concluding thoughts.

An accompanying videotape shows Visual Obliq in action, building the multi-user text editor. The code that a user must write to implement this editor appears in Appendix A. Finally, Appendix B contains the Visual Obliq Home Page as of the printing of this report.

2 End-User Perspective

Most prior work on groupware has focused on sharing a traditional single-user application by replication at some level—the display, the interface, or the whole application. The degree of coupling and the level of replication may vary, but the functionality provided at each site is the same. We call such applications *homogeneous*. We address a larger class of distributed applications, *heterogeneous* applications, wherein the functionality provided at each site may be different. In Visual Obliq, the client-programmer can decide what goes on at each site.

A distributed application spans multiple sites linked by a network. The program code need exist only at the site where the application starts up, which is called the *server-site*. Then it spreads to sites where users are located, called *client-sites*. Typically the server-site is also a client-site.

An instance of a distributed program in execution is called a *session*. Users at various sites may join or leave a session from time to time, whereas the session itself continues to exist until it is explicitly terminated. Each session has a name which is of the form

```
<program-name>@<server-site-name>
```

A session is created when a program starts running at a server-site. Typically, this happens by hitting the “Run” button in the GUI-builder, or by invoking an executable that was output by the GUI-builder. The executable is a shell script containing the code generated by the GUI-builder, prefixed by a one line shell command,

```
#!/bin/vorun -r
```

As we shall discuss in Section 4, client-programmers write code in Obliq [5], an interpreted language. The `vorun` program is a version of the Obliq interpreter with various libraries pre-loaded for handling Visual Obliq’s distribution.

The most common way that a user joins a session is by invitation by other members. Users wishing to be invited to a session must be running a *Visual Obliq Agent* (VOA). The VOA responds to invitations from other sites by displaying a window at the invitee’s workstation (see Figure 2) that allows the user to accept or decline the

Document Title: Digital Systems Research Center Home Page

Document URL: http://src-www.pa.dec.com/src.home.html

The Systems Research Center, lo… …out 40 full-time researchers. We work in several a… …computing, networking, programming techno… …n.

We want our ideas and experienc… …ew products and to help solve technical problems, wh… …aboration with advanced development and produ… …sts for technical help.

SRC

- Finding things
  - ☐ What's new at SRC
  - ☐ Fun Stuff
  - ☐ Full-text searching: Digital's Web – Gatekeeper archive
  - ☐ Gateways (NI – ELF – bboards – abwho – argo – J-Video – man – finger – others)
- Research
  - ☐ SRC project plans
  - ☐ SRC publications, talks, and patents
- Staff

Session Request

You are invited to a session of

TicTacToe @ qilbo.dec.com

by krishna @ tsktsk.dec.com

Figure 2: Inviting another user to join an existing session.

invitation. Issuing the actual invitation is the responsibility of the client-programmer, who would typically provide the end-user with a dialog box to input the name of the site where the invitation should be sent.

The other way that a user joins a session is on his own accord, having found out about it somehow. For example, to join a session called `TicTacToe` at the machine `qilbo.dec.com`, an end-user would issue the command

```
vorun -join TicTacToe@qilbo.dec.com
```

Although the system provides no access-control on users wishing to join an existing session, an application may implement some policy of its own.

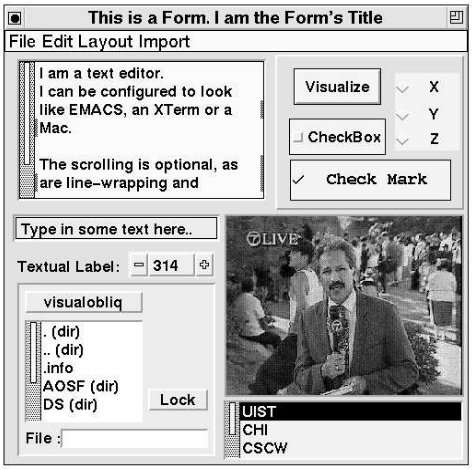
3 Interface-Designer Perspective

The GUI-builder is a tool that integrates all stages of the development cycle. It allows the interface-designer to graphically specify the interface, attach code at appropriate places, and execute the resulting application within the environment for testing and debugging. The GUI-builder can also output a file containing a standalone application.

The interface-designer interactively designs a set of top-level windows called *forms* using the GUI-builder. Forms contain widgets and popup windows. Most importantly, each widget can contain callback code that will be invoked by the system in response to end-user actions. When a session is in progress, multiple instances of each form may exist, at various sites. Thus, forms are both the unit of design and the unit of distribution. This provides a simple and elegant conceptual model for the client-programmer, and it is an important contribution of our system.

Figure 1 is a screen dump showing the GUI-builder in action. There are two top-level windows: the design window (left) is for building the interface, and the attribute sheet (right) is for manipulating properties of widgets.

The *design window* has a palette of widgets that may be used to compose the interface, and a design area where the interface is embedded while it is being designed. The interface-designer may have a number of design windows open while designing a given application. Currently, the design palette contains the following types of widgets (see Figure 3): browsers, buttons, choices, file browsers, forms, frames, numeric fields, scollbars, static text, text editors, toggles, typein fields, and video players. These widgets are common in modern GUIs, with the exception of forms, frames, and video players. A form is either a top-level window or a popup-window within some other form. A form may also have a menu bar. A menu bar is a property of the form widget, not a separate widget. A frame is used to geomet-



I am a text editor.
I can be configured to look like EMACS, an XTerm or a Mac.

The scrolling is optional, as are line-wrapping and

Figure 3: Widgets supported by Visual Obliq.

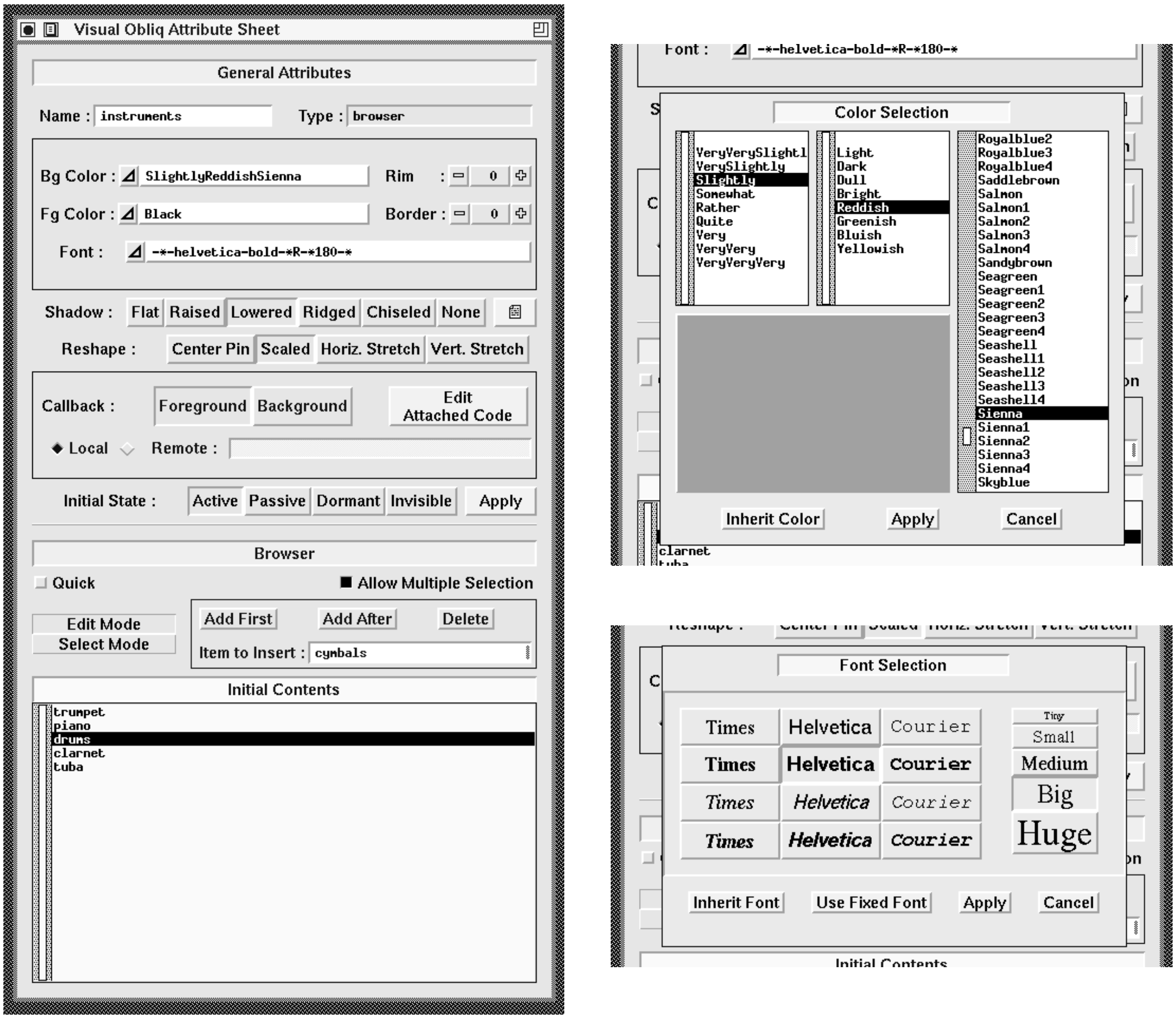
rically constrain the location of other widgets; it has no interactive behavior. The video player is used to playback audio-visual input from a specified host. Section 7 describes how Visual Obliq can be extended with more widgets.

The user-interface of the GUI-builder is similar to other GUI-builders. The size and location of widgets are set in the design window; all other properties are specified in the attribute sheet. In the design window, a single-click is used to select a widget, and a double-click causes the attribute sheet to be loaded with the selected widget's properties. Editing operations, such as cut and copy, apply to the selected widget. Resizing and moving a widget are subject to the bounds of the frame or form in which it lies. Moving a frame causes all of its children to move with it (indeed, this is the primary purpose of a frame). Resizing a frame or a form is further constrained to not clip any of its children. Finally, there are ways to align, shape, and distribute widgets under the same frame or form.

The top half of the *attribute sheet window* displays properties that are applicable to all widgets. These properties mostly relate to the visual characteristics (e.g., font and colors) and to how the callback code should be invoked (foreground or background, local or remote). Some widgets may not use some properties in any way (e.g., a scrollbar doesn't use the font information; a frame never invokes a callback), and such properties are disabled appropriately. The bottom half of the attribute sheet has fields for widget-specific attributes. For example, the browser-specific properties are the initial contents (a set of strings or a file containing text), a flag indicating whether multiple selections are supported, the initial selection, and a check-box toggle for indicating what type of mouse clicks should cause the callback to be invoked (e.g., on every down-click or only on the up-transition of a double-click). Figure 4 shows the attribute sheet and the popup dialogs that allows the interface designer to select a font and a color.

A novel part of the Visual Obliq GUI-builder is the simple, but expressive, model for specifying a widget's behavior when its parent is reshaped at run-time. This model is a simplification of Cardelli's stretching model [4], with which we had considerable experience using as interface-designers. Widgets may be thought of as rectangular sheets of rubber, attached to their parent by one or more pins. The following settings are supported: *pinned*—a single pin at the center; *scaled*—pins at the four corners; *vertical stretchy*—pins at the top and bottom edges; and *horizontal stretchy*—pins at the left and right edges. Each widget has a default setting. For example, buttons use pinned, text editors use scaled, and typein fields use horizontal stretchy. The setting can be changed in the attribute sheet.

Thus far, we described the GUI-builder's *design* mode. The GUI-builder also has modes corresponding to other phases in the application-development process: testing, building, and running.



Name : instruments
Type : browser
Bg Color : SlightlyReddishSienna
Rim : 0
Fg Color : Black
Border : 0
Font : -*-helvetica-bold-*R-*180-*
Item to Insert : cymbals

```
trumpet
piano
drums
clarnet
tuba
```

```
VeryVerySlightl
VerySlightly
Slightly
Somewhat
Rather
Quite
Very
VeryVery
VeryVeryVery
```

```
Light
Dark
Dull
Bright
Reddish
Greenish
Bluish
Yellowish
```

```
Royalblue2
Royalblue3
Royalblue4
Saddlebrown
Salmon
Salmon1
Salmon2
Salmon3
Salmon4
Sandybrown
Seagreen
Seagreen1
Seagreen2
Seagreen3
Seagreen4
Seashell
Seashell1
Seashell2
Seashell3
Seashell4
Sienna
Sienna1
Sienna2
Sienna3
Sienna4
Skyblue
```

Figure 4: The attribute sheet of the Visual Obliq GUI-builder (left), and its popup dialogs for selecting a color (top-right) and a font (bottom-right).

In the *test* mode, the user is allowed to interact with the interface without handles over the widgets intercepting the input. No callbacks are invoked, but widgets react to user input, menus are active and the resize model takes effect so the interface-designer gets a pretty good idea of the look-and-feel of the interface.

In the *build* mode, the GUI-builder generates a file that can be executed. This file has three logical components: The first is a textual description of the interface in the FormsVBT language [2]. The second component is a program in the Obliq language [5] that embeds the client-programmer's callback code and builds the interface from the description. The third part is a one-line shell command to invoke the interpreter, as discussed in Section 2. The contents of this file, though textual, are not intended to be viewed or edited by the client-programmer. Section 5 describes the FormsVBT and Obliq languages, and the next section describes the facilities available to the client programmer.

In the *run* mode, a file is not written disk. Instead, the contents of the file are given to an Obliq interpreter that is linked into the GUI-builder, thereby providing rapid turnaround. As a convenience, the GUI-builder provides a way to invite other clients to join the current session. This is done using the typein field in the upper right of the design window. However, most applications provide such a facility themselves; it is quite easy to implement, as we shall describe later.

4 Client-Programmer Perspective

The bulk of the code that a client-programmer writes is *widget callbacks*. Callback code is invoked in response to user action in a widget. To support distributed and heterogeneous applications, three other categories of code can also be specified: form code, global code, and session-constructor customization code. These are discussed later.

All client code is written in Obliq [5], an interpreted language that supports distributed object-oriented computation. Section 5.1 describes the language in more detail. The Obliq code that is generated by the GUI-builder (and then fed to an Obliq interpreter for execution) contains a complete Obliq program. However, client-programmers never actually see the complete Obliq program; the various code fragments, such as callbacks, are written in the appropriate parts of the GUI-builder's attribute sheet window.

To get a feeling for coding in Visual Obliq, we first examine the callbacks for a non-distributed application. Consider a game of Tic-Tac-Toe, as in Figure 5. There are 11 widgets within a form: nine buttons comprising the game board, one static text showing the string "Current player:", and a typein field. Initially all buttons

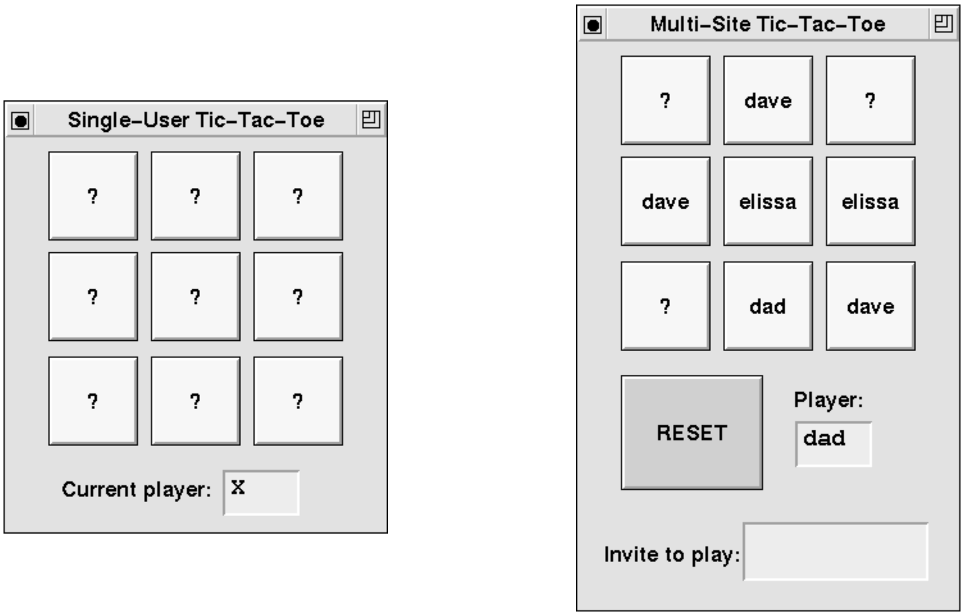


Figure 5: A single-user, non-distributed game of tic-tac-toe (left), and a multi-user, distributed version (right).

display a question mark. When one of the nine buttons is clicked for the first time, that button's label changes from a question mark to the string in the typein field. Through the GUI-builder, the buttons were given the names `b1`, `b2`, ..., `b9`, the typein field the name `who`, and the top-level form the name `ticktack`. The typein field and the static text don't have callbacks; the callback for each button is similar. It looks as follows for `b5`:

```
let curr = SELF.b5.getText();
if text_equal(curr, "?") then
  let sym = SELF.who.getText();
  SELF.b5.putText(sym);
end;
```

Although we haven't explained the syntax or semantics of Obliq, the callback code is hopefully easy to follow.

The variable `SELF` is defined by Visual Obliq to be a handle to the form-instance in which the user clicked, causing the callback to be invoked. `SELF` is actually an object with fields for each widget in the form. For example, `SELF.who` refers to the typein field. Each widget in turn is an object, with methods to retrieve and modify its properties. For example, all typein widgets have a `getText` method to re-

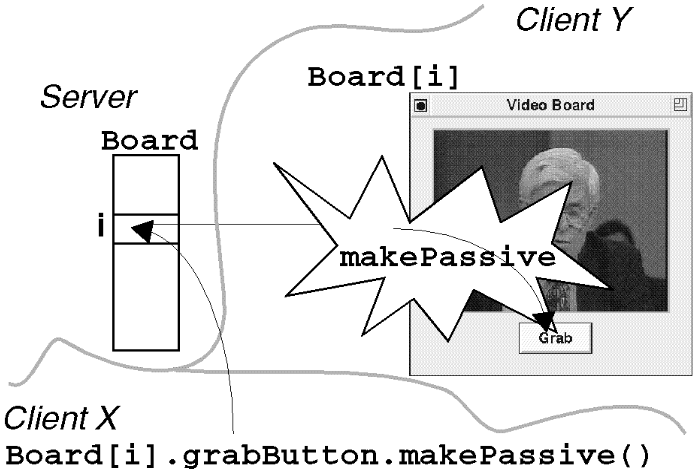


Figure 6: Remotely accessing a widget.

trieve their contents. They also have methods `putText`, `getFont`, `putFont`, and so on.

As mentioned before, the unit for distribution in Visual Obliq is a form. Visual Obliq creates an array of handles to form-instances; each element of the array is a handle to an instance running, typically at a distinct site in a separate address space. The array is stored at the server-site. What makes this strategy a big win is that there is a single, global name-space, allowing one part of the distributed application to access and modify objects in other parts, without caring about their physical location. Such *location transparency* applies to the user-interface, and also to any additional fields that the programmer may attach to the form. Consequently, both the state of the form-instance and the widgets within it may be remotely accessed and manipulated.

Figure 6 shows how location transparency can be used by one client-site to disable a button at another client-site. The top level form is named `Board`, and the array of handles to the form-instances is stored in an array named `Board` at the server-site. There are two client-sites, x and y. Client x can disable the button named `grabButton` on client y by the one line

```
Board[i].grabButton.makePassive();
```

just by knowing that client-site y is the ith instance in the current session.

Let's now consider a distributed version of the Tic-Tac-Toe application, where all sites are playing the same game (see Figure 5). The callback that changes the button's label needs to be changed to the following

```
let curr = SELF.b5.getText();
if text_equal(curr, "?") then
  let sym = SELF.who.getText();
  foreach f in ticktack do
    f.b5.putText(sym);
  end
end;
```

in order to cause the labels to change at all sites. (The `foreach` statement iterates through an array, setting the iteration variable to each successive element of the array.) A new player can be added to the game by including a typein field in which a player types the name of a site to be invited to join the game. The callback for the new typein field, `invitee`, is quite simple:

```
let destination = SELF.invitee.getText();
installAt(destination);
```

The procedure `installAt` is provided by Visual Obliq to cause an application to spread to another site. The effect of calling `installAt(site)` is for the runtime to contact the Visual Obliq Agent at the client-site `site` to popup a "You are invited ..." message (see Figure 2. If the user declines, `installAt` returns false; otherwise it returns true. Here the return value is ignored. Most importantly, if the user accepts the invitation, the Visual Obliq Agent will install a new copy of the application at the client-site `site`, and the global array `ticktack`, stored at the session's server-site, will grow by one element. Section 6 describes how this happens in more detail.

Although all code originates at the server-site and moves to the client-sites, the display of the game is *not* the same at all sites! In this simple Tic-Tac-Toe example, the contents of the `who` typein field will probably be different (each user will type in his own marker!), and the size of the window is under the user control at each site. Of course, the colors, fonts, and other screen resources may be different, depending on how the X server at each site resolves the resource requests. In general, a session may be configured so that certain forms get instantiated at some sites and not at others. For instance, in the distributed multi-user editor described in Appendix A, the server-site has an instance of a "moderator" form. This form allows the user at the server-site to yank the "floor" from one participant and give it to another.

4.1 Other client-code

We now consider the other types of code that the programmer can write. These provide client-programmers with a wide-range of choices of distributed programming techniques to work with. Visual Obliq takes care of inserting the code into the appropriate places in the generated Obliq program.

4.1.1 Form code

The client-programmer can extend a form object with additional data-fields, procedures, and methods. It's important to understand that although a handle to each form-instance is stored in an array of handles at the server-site, each instance itself is located at a client-site. Each client-site is a separate address space, typically on a separate machine. This is the technique used by the client-programmer to associate "local code" with client sites. Any code associated with a form gets replicated in each of its instances and is hence local to the site where its instance is created.

Data-fields, procedures and methods within a form-instance can be can be accessed remotely. When a data-field is accessed remotely, the data gets copied over. Likewise, invoking a procedure which is part of a remote form-instance causes the procedure to be copied over and executed locally. However, when a method of a remote instance is invoked it executes at the remote site. For example, if client-site c were to execute

```
x = DeptEmps[5].avgRaise(salary);
```

the code for procedure `avgRaise` would be copied from instance 5 to client-site c, and executed at client-site c. If `avgRaise` happened to be a method instead, it would execute at the site where instance 5 resides.

4.1.2 Global code

The client-programmer can designate additional data-fields, procedures, and methods to be stored globally for the session, rather than on a per-form basis. Data-fields, procedures, and methods respond to remote accesses as in the previous case. This is where the client-programmer will place the "global" (i.e., shared) portion of the application code.

4.1.3 Session-Constructor Customization code

The GUI-builder generates a procedure called the *session-constructor* to engineer the spread of the session from site to site. The session-constructor is executed at each client-site when it joins the session, taking the local site-name as an argument, and bootstraps the session at the site. The session-constructor is responsible for instantiating the forms needed by the site initially.

The default session-constructor creates one instance of each form that has been designed in the GUI-builder. The client-programmer can edit the constructor to decide how many instances of each form should be initially created at each site. For heterogeneous applications, the session-constructor can be made conditional.

This procedure may be viewed as a ticket that permits a site to join a session. It is exported when invitations are extended, and may be imported from a known purveyor, when admission to the session is desired.

4.2 Remote Callbacks

Earlier we described how location transparency allows remote objects to be accessed as if they were local. Visual Obliq also makes it easy to write distributed callbacks. The primary purpose of such distribution is load-sharing, and exploiting the hardware (e.g., a supercomputer) or software (e.g. a database) of some particular machine.

As part of the attribute sheet, the client-programmer can specify that a callback will be executed remotely and provide an Obliq expression that, when evaluated at run-time, will designate an instance of a form (not necessarily the same type of form!) where it should execute. Visual Obliq copies the callback to the remote instance, and executes it there. References to `SELF` will still pertain to the instance where the callback originated, and hence the semantics of the callback are unaltered. If fields within the remote instance also need be accessed, they may be referenced using the prefix `REMOTE` instead.

Since shipping a callback to a remote site is not necessarily more expensive than a single remote reference, such remote callbacks provide an efficient alternative to making multiple remote accesses.

4.3 Background Callbacks

The client-programmer can specify within the attribute sheet that a callback should be executed in the background. This will cause a separate thread to be forked in order to execute the callback. This feature is orthogonal to remote execution. Background callbacks are useful when a callback is known to take a long time and can be executed asynchronously with other callbacks.

4.4 Synchronization

Although a global name-space simplifies distributed programming a great deal, it does not address the issue of synchronization. Through the GUI-builder, it is possible to specify that each callback method should be executed mutually exclusive of all other callbacks in the session. However, many other styles of protection and serialization are possible, and may be implemented using Obliq's repertoire of synchronization primitives.

For instance, here's a way to make a callback conditionally execute. That is, it will execute only if no other callback in the session, protected by the same lock, is being executed. First, we add the following global code:

```
var Ness = { serialized,
  locked => false,
  TestAndSetLock => meth(s)
    if not (s.Locked)
      then s.Locked := true; true
      else false end
    end,
  Unlock => meth(s)
    s.Locked := false
    end
}
```

The code defines a new object, `Ness`, with a data field called `locked`, and two methods. The keyword `serialized` is an Obliq primitive to indicate that there is a hidden mutex. The mutex will be acquired on entry to a method and released upon completion. Then, we modify the callback call as follows:

```
if Ness.TestAndSetLock() then
  ...
  Ness.Unlock();
end;
```

The code indicated by the ellipses is executed only if no other callback in the session is executing code protected by the `Ness` lock.

Given location transparency, implementing custom synchronization schemes is no more complicated than in a single address-space.

4.5 Robustness

Failures are a common cause of concern in distributed applications. Any remote access is capable of raising an exception, which could crash the session if not properly handled. The Visual Obliq GUI-builder protects code it generates with exception-handlers. The client-programmer is expected to do the same at some granularity.

5 System Support

In the previous section we presented Visual Obliq's programming model for distributed applications. The model makes many demands on the underlying system. Data and code need to be copied and shared across address spaces, methods need to be invoked on remote objects, and session-constructors must be made publicly available. In this section we describe how the SRC Modula-3 programming environment supports the implementation.

5.1 Obliq and Network Objects

To quote Cardelli [5]: "Obliq is a lexically-scoped untyped interpreted language that supports distributed object-oriented computation. An Obliq computation may involve multiple threads of control within an address space, multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet. Obliq objects have state and are local to a site. Obliq computations, in the form of procedures, can roam over the network, while maintaining network connections."

In Obliq all entities are composed of *constant values*, which are immutable, and *locations*, which are mutable. When an entity is transmitted to a remote site, the constant values within it are replicated, while locations within it are transmitted as *network references*. Operations on a network reference will happen transparently at the original site. The fields and methods in an object are locations. So are the elements in an array and variables in general. Thus, objects, arrays and variables, for all practical purposes, remain rooted in the address space where they are created. However, when a procedure is transmitted, the code for the procedure is copied over to the remote site (because it is a constant) and gets run there, while variables in its scope become network references. Thus, when a procedure is transmitted in Obliq, its scope gets transmitted as well. This permits the procedure to retain its original semantics in the new address space.

We use this technique in Visual Obliq to spread the session from site to site. Objects can be instantiated in another address space by transmitting procedures which construct them. This is how we instantiate forms at client-sites. We create them using form construction procedures called *form-constructors*, transmitted from the server-site. These procedures retain their scope in the client address space. At the same time they are able to acquire a handle to the local scope, which is passed in as an argument called `LOCAL`. This enables them to get access to local widget creation and management code.

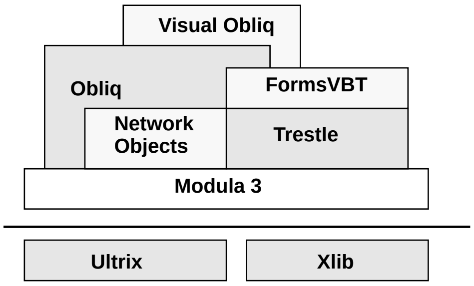


Figure 7: The SRC Modula–3 programming environment.

Obliq's static scoping is heavily utilized in Visual Obliq. Objects take on the scope of the procedures that create them. If the procedures that create a set of objects were transmitted from the same address-space, then the objects will share a common scope. This is how we implement a global name-space in Visual Obliq. Form-instances inherit the scope of the form-constructors that create them, and hence get access to the scope at the server-site. Instance arrays, form-constructors and other forms of global code are placed in the initial, common scope at the server-site, and are hence globally visible.

The Obliq distribution and data sharing mechanisms are built using the Modula-3 Network Objects package. Network Objects [3] are special Modula-3 objects that support remote access to fields and methods. Obliq uses a *Network Object Daemon* to make object references publicly available. In Visual Obliq we use this mechanism to make the session-constructor publicly available. Any site that starts a Visual Obliq must must also be running a Network Object Daemon, to make the session public. Obliq has support for exception-handling and synchronization, which are invaluable for building robust and deterministic distributed applications.

Obliq (and hence Visual Obliq) can serve as an embedded command language for Modula-3. Further, Obliq is extensible, and Modula-3 types and packages can be integrated as needed. One such extension is the FormsVBT package.

5.2 FormsVBT and Trestle

FormsVBT [2] is a system for building graphical user interfaces. It consists of a language for describing an application's user interface, and a runtime library for communicating between an application's code and the user interface. (FormsVBT provides a stand-alone application for constructing the user interface, akin to Visual Obliq's GUI-builder. This part of FormsVBT is not used by Visual Obliq.)

FormsVBT is implemented in Modula-3 and uses the Trestle UI toolkit [9] running on X windows.

A user interface in FormsVBT is a hierarchical arrangement of *components*. These include passive visual elements, basic interactors, modifiers that add interactive behavior to other components, and layout operators that organize other components geometrically. In the FormsVBT language, the arrangement is written as a symbolic expression (S-expression). The outermost expression is the *form* or top-level component, and subexpressions are either properties that modify a component or other, subordinate components.

The runtime library provides the communication between an application and its user interface. There are procedures to convert an S-expression into a window object, procedures to register *event-handlers* that will be invoked in response to user actions, procedures to retrieve and modify the values of the components, procedures to change the appearance (and even the hierarchy) of the components, and so on.

Each component in FormsVBT is implemented by a window class, called a VBT, provided by Trestle. Most of the things that a programmer would want to do with a component can be done via the FormsVBT interface. However, there may be occasions when the programmer would like direct access to the underlying VBT. FormsVBT provides such access.

The GUI shown in Figure 2 is described by the following FormsVBT S-expression:

```
(Shape (Width 350) (Height 400)
  (VBox
    (Rim (Pen 10) (BgColor "Black") (Color "White")
      "Session Request")
    Fill
    "You are invited to a session of"
    Fill
    "TicTacToe @ qilbo.dec.com"
    Fill
    "by krishna @ tsktsk.dec.com"
    Fill
    (HBox
      Fill
      (Button %accept (BgColor "PaleGreen") "Accept")
      Fill
      (Button %reject (BgColor "PaleRed") "Reject")
      Fill))))
```

The two buttons have been given names so that callbacks can be registered for them.

6 Run-Time Picture

Figure 8 illustrates the spread of a session from the site named `ash.pa.dec.com` to two others.

Imagine that the user at `ash.pa.dec.com` types

```
vorun -r foo
```

The command `vorun` runs the Obliq interpreter with Visual Obliq support code (in the file `vo-library.obl`) pre-loaded. The Obliq program in `foo.obl` is loaded, creating the session `foo@ash.pa.dec.com`. The file `foo.obl` also causes the creation of a session-constructor and a form-constructor. The session-constructor is registered under the name `foo` with the local Network Object Daemon; it will be needed if a remote client-site wishes to join the session. The form-constructor is a procedure that is used to build and install a top-level window from the FormsVBT textual description. There is one form-constructor for each top-level window defined in the application. In this case, there is a single top-level window, and it was given the name `Board`. The final step in the execution of `foo.obl` is the invocation of the session-constructor. The default session-constructor instantiates a single form by invoking each form-constructor. The machine `ash` is now both the server-site and a client-site.

At another site, `bay.pa.dec.com`, the user types

```
vorun -join foo@ash.pa.dec.com
```

This `vorun` program is a version of the Obliq interpreter, with the `vo-library` already loaded. The `-join` option uses a procedure in the library to import a reference to the session-constructor named `foo`, from the Network Object Daemon running on `ash.pa.dec.com`, and executes the session-constructor. The session-constructor causes the form-constructor procedure to be copied over and executed. The form-constructor creates and initializes an Obliq object to manage the user-interface. The object computes the S-expression for the its interface from the attributes of its components. The rest of the form-constructor converts the S-expression into a VBT tree, attaches callbacks to the VBTs and installs the tree in the window system. A handle to the window object is added atomically to the global array called `Board` maintained at `ash` (the server-site), and the window object is installed in the window system at `bay`.

Finally, the user at `bay.pa.dec.com` interacts with the application and eventually performs an operation that invites the user at `oak.crl.dec.com` to join the session. This is brought about by invoking the built-in routine `installAt` from a callback. The `installAt` procedure imports a handle to the Visual Obliq

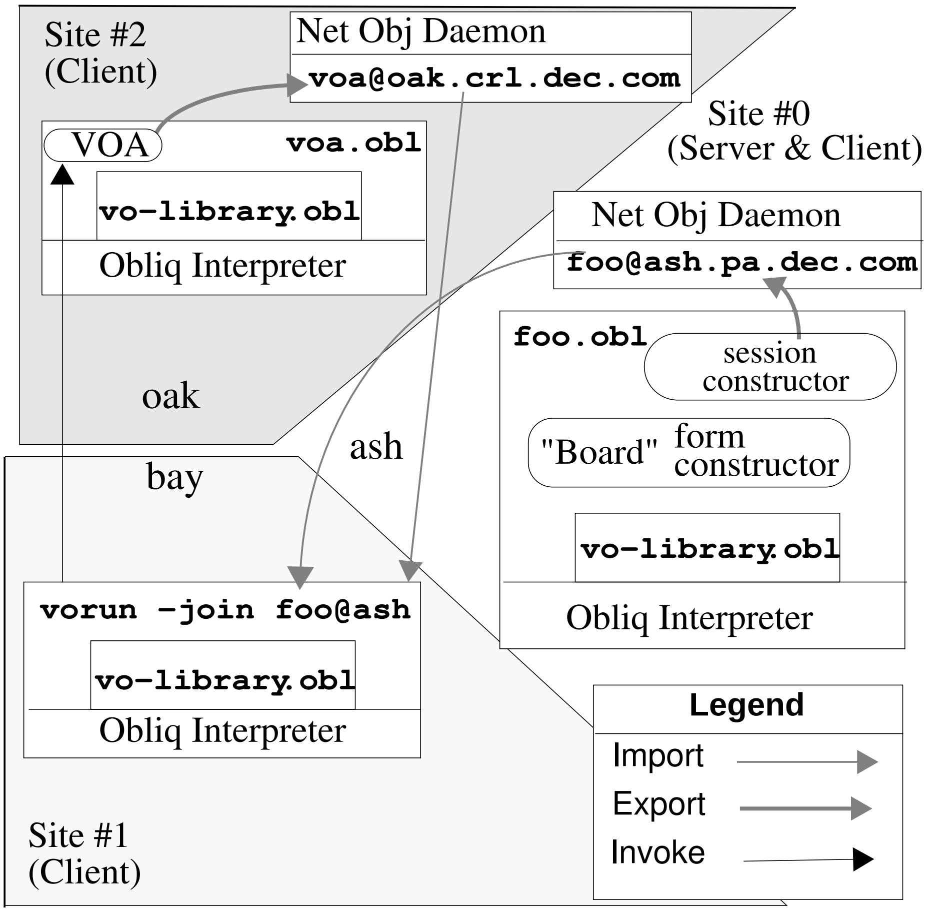


Figure 8: An example scenario.

Agent (VOA), which is an Obliq object registered with the Network Object Daemon running on `oak.crl.dec.com`. Then, `installAt` invokes a method on this object, passing in the session-constructor for the application. The method pops up an invitation window (shown in Figure 2) to find out if the user at `oak` wishes to join the session. If the user agrees, the session-constructor (passed as a parameter from `bay`) is executed as before.

7 The Extensible GUI Builder

All widgets in Visual Obliq are built out of FormsVBT components. The widget set in Visual Obliq is extensible and any composite FormsVBT object can be easily integrated.

Here is the FormsVBT S-expression that defines a (simplified version of the) Visual Obliq button widget named `b`:

```
(Border %bBorder (Pen 1)
  (Button %b
    (TSplit =0
      (Text %bText "Label")
      (Pixmap %bPixmap "/dev/null"))))
```

(The TSplit component is used to display exactly one of its children. Here, the TSplit is used to decide whether to display some text or a pixmap in the button.) A variant of this basic S-expression is used to generate the widget both at design-time and at run-time.

At design-time, the GUI-builder will modify the S-expression to reflect the values that the user has specified for various attributes. Also, additional components are wrapped-around the S-expression, in order to block mouse clicks and to support moving and resizing widgets. At design-time, there is a *property table* associated with each widget. The property table maps property names to property values. This information is copied into the attribute sheet when a user double-clicks on a widget; it is updated from the attribute sheet when the user clicks ”Apply”.

Specifically, only four elements are required to add a new component to Visual Obliq:

1. A Modula-3 procedure for generating a FormsVBT S-expression for a widget from a property table. For the widgets currently implemented in Visual Obliq, this procedure has been implemented in a table-driven fashion. In each case, it replaces place-holders in a template S-expression with values from the property table.

2. A FormsVBT S-expression (for the bottom half of the attribute sheet) that contains ways to modify widget-specific attributes.

3. Three Modula-3 procedures for interacting with the attribute sheet. Procedure `loadAttributes()` copies a widget's property table into the attribute sheet; procedure `checkAttributes()` validates data in the attribute sheet and generate relevant error messages; and procedure `copyAttributes()` copies attributes from the attribute sheet back into the widget's property table.

4. A widget-constructor, which is an Obliq procedure, stored in `vo-library.obl`. This procedure is invoked each time that a new instance of the widget is created in order to instantiate an object to manage the instance. The object thus created provides the application programmer's interface to the widget. For example, the video player widget has methods `play(source)`, `clear()`, and `volume(level)`.

The first three elements are needed by the GUI-builder; the last element is needed so that callback code can access the properties of the widget.

8 Related Work

There has been considerable work in CSCW in the area of multi-user applications, or groupware. General purpose groupware systems come in two flavors, application sharing systems and groupware toolkits. The "Unofficial Yellow Pages of CSCW" [8] provides a comprehensive list of both sorts.

Application sharing systems allow a traditional single-user application to be shared by replication at some level. Usually it happens at the level of the display, as in XTV [1], and Shared-X [6]. This conforms to our notion of homogeneous applications.

Groupware toolkits such as Rendezvous [10] and GROUPKIT [11], on the other hand, provide the flexibility needed to author heterogeneous groupware. These systems differ from Visual Obliq in the abstraction provided for distribution. In Rendezvous the client-programmer makes use of a "sharing object" which the runtime system is responsible for maintaining in a consistent state across address-spaces. Slots in interface components may be constrained to those in sharing objects to achieve what-you-see-is-what-I-see. In GROUPKIT, messaging objects called "readers" and "writers" are used to provide the client-programmer with a message-passing facility. We believe that the abstraction of Visual Obliq's "location transparency" is more powerful and simpler to use (since it is tantamount to a single address space) than the abstractions found in contemporary groupware toolkits.

Fresco [7], currently under development, will allow the distribution of user interface components using CORBA. GUI components will have IDL interfaces defined for them so that they may be accessed remotely, allowing interface components in different address spaces to be pieced together to form a GUI hierarchy.

The combination of Obliq and FormsVBT most closely resembles Tcl/Tk with the Tcl–DP extension [12]. Obliq provides a cleaner framework for implementing groupware than Tcl because the language inherently supports the notion of multiple address-spaces, remote references and procedure migration. Visual Obliq could be written using Tcl–DP rather than Obliq for writing callbacks, at the cost of simplicity in the client-programmer's model.

Visual Obliq could probably also use General Magic's Telescript rather than Obliq, and MagicCap probably provides an environment much like that of Visual Obliq's GUI-builder. Unfortunately, technical details about General Magic's systems are not public.

Finally, we should point out that there are many interesting GUI-builders that have been reported in the literature, and many are available commercially. We modeled our GUI-builder on Microsoft's Visual Basic.

9 Summary

Visual Obliq was implemented mostly during the summer of 1993, and is reasonably complete. Thus far, it has been used only for "hacking around" and building toy applications. As the accompanying videotape verifies, simple (yet non-trivial) distributed applications can be built in a matter of minutes.

Implementing distributed applications is notoriously difficult, and yet the popularity and importance of collaborative tools is growing. Given the right abstractions the task of the programmer can be greatly simplified. We believe that abstractions that mirror a user's mental model of a distributed application are bound to work well. This was the goal behind developing Visual Obliq. The programmer is able to design windows, specify how many instances should appear at each site, and how they interact. For each type of window, the programmer can find out dynamically how many instances there are, and access each one by name. The fact that this is happening in separate address-spaces is hidden.

Visual Obliq provides a stand-alone environment for creating distributed applications rapidly and simply. It is the first system that combines the convenience of a GUI builder with a groupware toolkit, thereby providing groupware programmers and distributed application programmers with an integrated "design-code-and-go" solution.

Acknowledgments

We are indebted to Luca Cardelli for developing Obliq, for answering many (sometimes silly) questions, and offering lots of (always) sound advice. Paul McJones helped improve the readability of this report.

References

[1] H. M. Abdel-Wahab, and M. A. Feit. XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration, *Proceedings of the IEEE Conference on Communications Software*, pages 159–167, April 1991.

[2] Gideon Avrahami, Kenneth P. Brooks, Marc H. Brown. A Two-View Approach To Constructing User Interfaces. *Computer Graphics*, 23(3):137–146, July 1989.

[3] Andrew D. Birrell, Greg Nelson, Susan Owicki, and Edward P. Wobber. Network Objects. *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 217–130, December 1993.

[4] Luca Cardelli. Building User Interfaces by Direct-Manipulation. *1st Annual ACM Symposium on User Interface Software and Technology*, pages 152–166, October 1988.

[5] Luca Cardelli. Obliq: A language with distributed scope. Research Report 122 Digital Equipment Corporation, System Research Center, Palo Alto, CA, March 1994.

[6] P. Gust. Shared-X: X in a Distributed Group Work Environment, Second Annual X Technical Conference, January 1988.

[7] Mark Linton and Chuck Price. Building Distributed User Interfaces with Fresco, *Proceedings of the Seventh X Technical Conference* Jan 1993, pages 77–87.

[8] P. S. Malm. The unOfficial Yellow Pages of CSCW. *Classification of Cooperative Systems from Technological Perspective*, University of Tromsø, to appear.

[9] Mark S. Manasse and Greg Nelson. Trestle Reference Manual. Technical Report 68, Digital Equipment Corp, System Research Center, Palo Alto, CA, December 1991.

[10] John F. Patterson, Ralph D. Hill, Steven L. Rohall, and W. Scott Meeks. Rendezvous: An Architecture for Synchronous Multi-User Applications, *CSCW '90*, October 1990, pages 317–327.

[11] Mark Roseman and Saul Greenberg. GROUPKIT—A Groupware Toolkit for Building Realtime Conferencing Applications, *CSCW ’92*, November 92, pages 43–50.

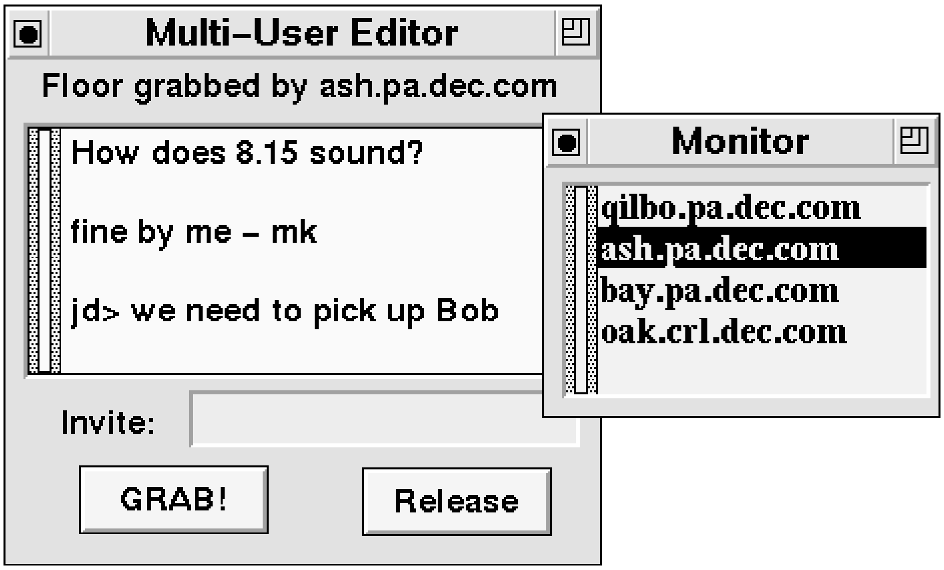
[12] Brian C. Smith, Lawrence A. Rowe, and Stephen C. Yen. Tcl Distributed Programming, *Proceedings of the Tcl Conference*, 1993.

Appendix A: Application Built in Videotape

The screen dump below shows a rudimentary multi-user editor constructed during the accompanying videotape. The window on the left is the shared editor; it appears at all client-sites. At any given time, at most one participant (i.e., a user at a client-site) "owns the floor," and this is the only person who is able to type into the text editor; the text editor widget on every other participant's form-instance is dormant. The owner of the floor may relinquish the floor by clicking on the "Release" button. At that point, the floor can be grabbed by another participant by pressing the "GRAB!" button. The typein field labeled "Invite:" allows participants to invite other users into the session.

The window on the right is a monitor for a moderator, the user at the server-site. The monitor window contains a browser with a listing of participants. Whenever a user grabs the floor, the appropriate entry in the browser is highlighted. Also, whenever an entry in the browser is selected by the moderator, the floor is yanked from the current owner and given to the selected party. (The videotape tape does not show the code necessary for supporting the monitor window.)

The named widgets in the shared-editor form (named `MUE`) are as follows: `grab` is the "GRAB!" button; `release` is the "Release" button; `invitee` is the typein



Floor grabbed by ash.pa.dec.com

How does 8.15 sound?

fine by me – mk

jd> we need to pick up Bob

gilbo.pa.dec.com
ash.pa.dec.com
bay.pa.dec.com
oak.crl.dec.com

Figure 9: MUE: A multi-user editor with monitor.

field; `editor` is the text editor; and `msg` is the status message at the top of the form. All of the callbacks are set to be run mutually exclusively. Initially, `editor` is dormant and `release` is passive. The monitor form is named `Monitor`.

The callback for the “GRAB!” button does the following: The “GRAB!” buttons at all client-sites are disabled and the status messages at all client-sites are updated appropriately. Also, the user who clicked on the “GRAB!” button has his “Release” button and text editor enabled, and the appropriate entry in the moderator’s browser is selected to show who has the floor. Here is the actual Obliq callback code:

```
foreach f in MUE do
  f.grab.makePassive();
  f.msg.putText( "Floor grabbed by " & LOCAL.HOSTNAME)
end;
SELF.release.makeActive();
SELF.editor.makeActive();
Monitor[0].list.select(LOCAL.HOSTNAME);
```

The callback for the text editor copies the entire contents of the editor to all other sites after every keystroke:

```
let contents = SELF.editor.getText();
foreach f in MUE do
  if not (f is SELF) then
    f.editor.putText(contents)
  end
end;
```

The callback for the “Release” button causes the user’s interface to return to the initial state (the “Release” button is passive and the text editor is dormant), and the “GRAB!” button is enabled on all client-sites. Also, the selection is removed from the monitor’s browser, since there is no longer any participant who owns the floor. Here is the callback code:

```
foreach f in MUE do
  f.grab.makeActive()
end;
SELF.release.makePassive();
SELF.editor.makeDormant();
Monitor[0].list.deselect(LOCAL.HOSTNAME);
```

The callback to invite a new user into the session is as follows:

```
let destination = SELF.invitee.getText();
installAt(destination);
```

Whenever a client-site starts up, it must be added to the moderator's browser. This is done through the following initialization code that is part of the `MUE` form:

```
Monitor[0].list.add(LOCAL.HOSTNAME);
```

The monitor form has one widget, a browser named `list`. Whenever the moderator clicks on an element of the browser, the floor is yanked from the current owner and given to the selected party. Here is the code to do that:

```
let owner = SELF.list.getIndex();
let ownerName = SELF.list.getEntry();
foreach f in MUE do
  f.grab.makePassive();
  f.msg.putText("Floor given to "&ownerName);
  if (f.index is owner) then
    f.release.makeActive();
    f.editor.makeActive();
  else
    f.release.makePassive();
    f.editor.makeDormant();
  end
end;
```

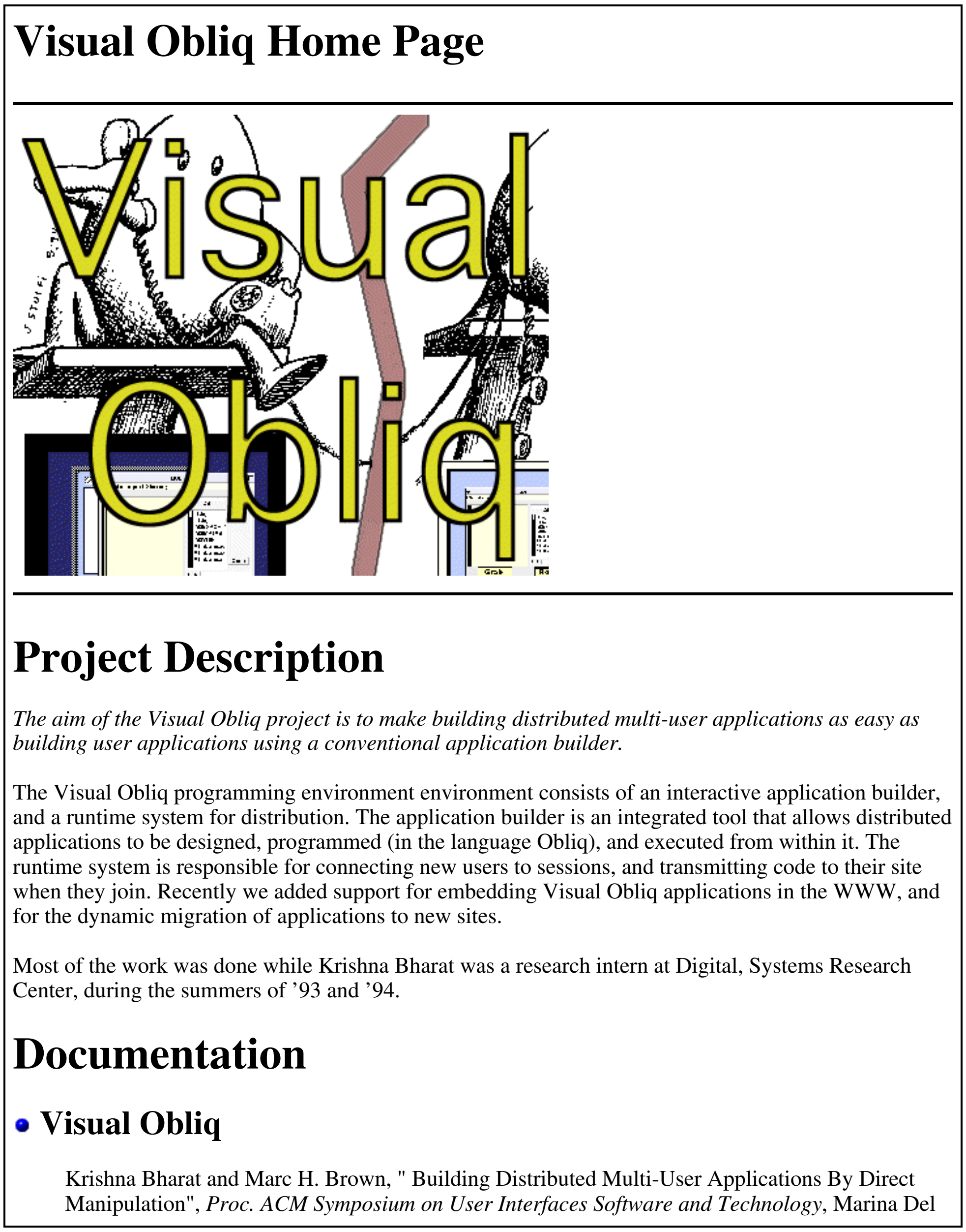
Finally, the application's session-constructor needs to install a `Monitor` form at the server-site, in addition to the default action of installing an `MUE` form at all sites in the session:

```
if SERVERSITE then MonitorNew(LOCAL) end;
MUENew(LOCAL);
```

The procedure `MonitorNew` is the form-constructor for the `Monitor` form, and the procedure `MUENew` is the form-constructor for the `MUE` form.

Appendix B: Visual Obliq Home Page

`http://www.cc.gatech.edu/gvu/people/Phd/Krishna/VO/VOHome.html`



Visual Obliq Home Page

Project Description

The aim of the Visual Obliq project is to make building distributed multi-user applications as easy as building user applications using a conventional application builder.

The Visual Obliq programming environment environment consists of an interactive application builder, and a runtime system for distribution. The application builder is an integrated tool that allows distributed applications to be designed, programmed (in the language Obliq), and executed from within it. The runtime system is responsible for connecting new users to sessions, and transmitting code to their site when they join. Recently we added support for embedding Visual Obliq applications in the WWW, and for the dynamic migration of applications to new sites.

Most of the work was done while Krishna Bharat was a research intern at Digital, Systems Research Center, during the summers of '93 and '94.

Documentation

- **Visual Obliq**

  Krishna Bharat and Marc H. Brown, " Building Distributed Multi-User Applications By Direct Manipulation", *Proc. ACM Symposium on User Interfaces Software and Technology*, Marina Del

Rey, CA, Nov 1994, pp. 71-82.

Visual Obliq (Video)

Krishna Bharat and Marc H. Brown, " Building A Distributed Application Using Visual Obliq", To appear in *CHI '95, Video Proceedings*.

The Obliq Programming Language

Luca Cardelli, " A Language with Distributed Scope", *Proc. of the 22nd Annual ACM Symposium on Principles of Programming Languages*, Jan 1995, pp. 286-297.

Embedding Visual Obliq Applications in WWW

Krishna Bharat and Luca Cardelli, " Distributed Applications in a Hypermedia Setting", To appear in *Proc. of the International Workshop on Hypermedia Design*, Montpellier, June 1995.

Migratory Interactive Applications in Visual Obliq

Krishna Bharat and Luca Cardelli, " Migratory Applications" (postscript), To Appear in *Proc. ACM Symposium on User Interfaces Software and Technology*, Pittsburgh, PA, Nov. 1995.

Availability

Visual Obliq is included in the current Digital SRC, Modula-3 Distribution.

Members

- **Marc H. Brown, Digital, Systems Research Center**
- **Luca Cardelli, Digital, Systems Research Center**

- **Krishna Bharat, Graphics, Visualization and Usability Lab, Georgia Tech**
