SmallScript: A User Programmable Framework
Based on Smalltalk and PostScript

Kevin Haaland and Dave Thomas
School of Computer Science
Carleton University

Abstract

This paper describes the design and implementation of SmallScript. SmallScript combines the power of object oriented programming and an advanced imaging model, based on PostScript, to provide an integrated, interactive development environment for multiprocessing graphical applications. All applications which use SmallScript are divided into user interface and application components which interact through an object-oriented message protocol.

1 Introduction

There are numerous applications in which a user programmable framework (UPF) for user interface development is essential. The large number of shell scripts and menu shells on current systems are a strong testimony to the need and utility of such systems. Traditional user interface management systems (UIMS) [Pfaff, 1985] and standardized toolkits such as X Windows [Scheifler, 1986] and MacApp [APDA, 1986] despite wide spread interest, fail to meet the demanding needs of evolving user interface software. To overcome this problem, embedded programming languages are often provided (eg. AUTOLISP). The need for user programmable interfaces has inspired a wide spread increase in the use of Smalltalk, [Goldberg, 1983] which remains the most powerful environment for the interactive exploration of new interface metaphors. It is unfortunate, however, that a language with such power is based on the low-level bitblt raster imaging model of the 70's. One of the primary goals of this research is to address this weakness.

In this paper we describe SmallScript, an integrated environment for user interface development. SmallScript, like NeWS [NeWS, 1987], has a PostScript [Adobe, 1984] based graphics imaging model. User interfaces are written in Smalltalk, a language and environment acknowledged to be among the most advanced available. SmallScript extends Smalltalk with a modern, device independent imaging model and an alternative user interface framework. We describe our prototype design and implementation of SmallScript.

Our primary interest is CPU intensive applications such as finite element analysis, (FEA) and computer aided design (CAD). Ideally, we see each program being composed of an application and a user interface. The application component implements the algorithms necessary to make the program function (eg. determining the optimal circuit board layout in a CAD system, or solving a large set of nonlinear equations in a FEA). The user interface component mediates the interaction between the application components and the user. By dividing the problem into an application and a user interface, we hope to exploit the advantages of distributed processing in a heterogeneous computing environment. Furthermore, we would like to reduce the amount of user interface code that has to be written for each application. This approach is used in NeWS and X, where clients and servers are used to describe both components.

2 User Programmable Frameworks

Two recent user programmable frameworks (UPF) are NeWS and HyperCard [Goodman, 1987]. These systems offer the end user, as well as the application developer, the ability to configure a rich graphical interface using an interpretive language. Both systems allow the user interface to communicate with a more traditional application environment via a well defined interface. HyperCard provides a tangible, user interface metaphor based on index cards combined and

a simple command language, HyperTalk. HyperCard is unfortunately limited by its use of bitmap graphics and its inability to add new object types. In its current implementation, developing large applications is difficult since HyperCard lacks code browsers and debugging support. Providing a HyperCard user interface to an existing application requires the addition of external pieces of code, called resources, to the basic HyperCard system, via a resource mover utility program. It is also possible to use the XCMD facility to execute code not written in HyperTalk.

NeWS provides a powerful programming capability based on PostScript. The PostScript interpreter communicates with the application program via a stream connection. If an existing application can be modified to generate PostScript code, porting it to a NeWS environment is relatively easy.

3. Distributed Window Systems

3.1 The X Window System

The X window system provides a basic set of display primitives and input routing facilities. This system uses asynchronous, stream-based interprocess communication. Applications may use the facilities provided by X Windows to present information, on any display in the network, in a device independent and network transparent manner. An application calls procedures in the X library to perform low level operations on the display device. The requests are sent to the server for execution via the interprocess communication network. The server sends event notifications, corresponding to mouse movements and key presses, to the application. Client and server processes communicate via a potentially large number of low level messages. For example, printing "Hello world" in a window requires 40 executable statements and 25 calls to the standard X11 library [Rosenthal, 1988]. The implementers of X Windows argue that the cost of interprocessor communication is minimal. Thus combining the user interface with the window server is not required [Scheifler, 1986]. It is also argued that most application writers do not need to be concerned with low level interactions between clients and servers since the X11 distribution contains a user interface toolkit which allows the "Hello world" program to be written in 5 lines of code.

3.2 NeWS

Sun's Network extensible Window System (NeWS) is an advanced windowing system using PostScript in a distributed window server. PostScript displays text, graphics and sampled images in a device independent manner. NeWS is unique in its incorporation of a full programming language as the means of describing the appearance of objects on the display.

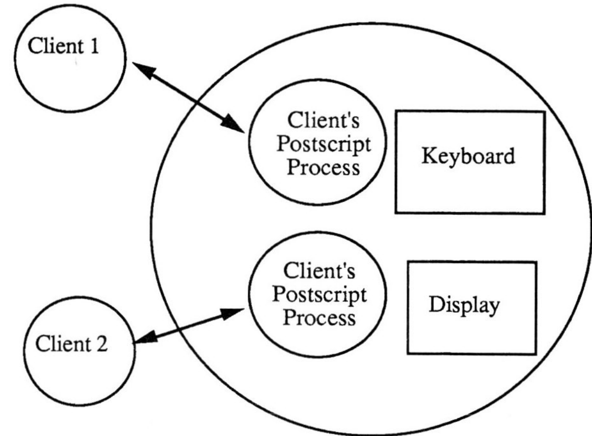

Figure 1 Client server relationship in NeWS

Multiple clients may have open connections with a single window server. Each client communicates with a lightweight process that executes inside the window server. (See figure 1) In NeWS, these processes are PostScript interpreters. NeWS clients send PostScript code via a byte stream which connects the client and a PostScript interpreter running in the server. Since PostScript is a programming language, clients can extend the capabilities of the window server by defining new PostScript procedures.

A NeWS application begins by establishing a connection with a NeWS server. Tthe client program then sends PostScript code to the server. This code defines the application specific routines required by the client. During the course of execution the client program responds to events from the server and sends PostScript code to the server. Unless the entire client code exists as a PostScript program, executed by the server, it must be divided into two components. The first component generates PostScript code that is sent to the server. The second implements the rest of the application. Although the presence of PostScript may be hidden by language bindings, the application developer is always aware that PostScript is present.

4 Object Oriented Approach

4.1 Introduction

Efficient exploration of new ideas in user interface design requires an ability to produce prototypes of the desired system. A good user interface often attempts to model techniques and entities familiar to the user. Object oriented languages such as Smalltalk, Actor [WhiteWater, 1987] and Objective-C [StepStone, 1986] can model real world objects (such as integrated circuits and welds), as well as abstract objects (such as windows, icons, menus, buttons and scroll bars) much better than traditional languages. For example, a user interface that incorporates file folders, filing cabinets and a garbage can is appropriate for applications used by office workers since they are familiar with the behaviour of these objects. Most object oriented programming languages provide the programmer with a set of generic software components which can be customized to suit specific needs. By using inheritance and polymorphism (where possible), the programmer builds on the existing functionality of the system.The ability to perform fast prototyping of ideas is needed to allow tailoring of the user interface.

4.2 Heterogenous Computing Environments

Local area networks and standardized communication protocols have allowed the creation of heterogenous computing environments. Typically, these networks are used for electronic mail, file transfer, remote login, and distributed file systems. Effective use of the resources available on a network can provide reliability and performance improvements that would be either impossible or too expensive on a single CPU system. On a network there may be specialized hardware devices, such as database machines and super computers, which can dramatically improve system performance. In NeWS, an application consists of a client and a window server process. This process is a user interface assistant for the client. By writing more of the system in PostScript, the user interface can be made to perform more tasks that would otherwise be done by the client. In NeWS, the application (client) controls the user interface (server). We propose that the user interface should be given control of the system and that developing an interface should be made as easy as possible. Furthermore, the user interface should be able to exploit multiple application assistants that, where appropriate, execute on specialized hardware.

In order to minimize the number of messages sent between CPUs, dependencies between components must be minimized. Dividing any system into application and user interface components is a nontrivial problem. The client-server model of distributed user interfaces used in X Windows implements the windowing routines on the server. It is the responsiblity of the client to implement the rest of the user interface. As a result, the client and server are tightly coupled and exchange a large number of low level messages, such as bit maps, line drawing commands, key presses and mouse movements.

It is often argued that forcing the division of a system into an application and a user interface is an awkward solution which doesn't work in real systems. This is the same argument that was used in the seventies when the notion of structured programming was introduced. Separating application and user interface has many benefits: program modularity, software reusability, better system design and improved programmer productivity. The application now becomes a toolbox of support procedures that the user interface calls when it requires work to be done.

Modern user interfaces are highly reactive and account for a large percentage of the code in most systems. To allow the user to customize his environment, many systems (eg. Autocad) include an embedded programming language. Since Smalltalk is both a programming language and an environment, we achieve the benefits of an embedded programming language with all systems developed in SmallScript.

To gain a better understanding of the behaviour of SmallScript, consider an interactive database system. The user interacts with this system using a mouse and high resolution display. The user interface is responsible for presenting graphical images and implementing their behaviour. For example, when the user selects an item with the mouse, the selected object highlights itself to inform the user that it has been selected. If the user decides to resize a window, the interface does the sizing and displaying of the information without help from the application. If, during the course of using this system, a database query is generated, the user interface forwards the query to the application. When the application has performed the query, the results are sent to the user interface. Since the application and the user interface communicate only via high level messages, the cost of inter-machine communication is minimized. An application specific object oriented protocol can often be superior to any low level generic protocol [Gentleman, 1988].

5 Implementation

5.1 The SmallScript Imaging Model

Imaging models describe the display capabilities of a graphics system. The coordinate systems used, the types of lines which can be drawn, support for colour and text are some of the more important aspects consided when assessing the merits of an imaging model. The ideal imaging model for our system renders objects composed of curved as well as straight lines, colour, and multiple fonts. The imaging model must also support hardware assisted image generation, if it is available, without requiring the programmer to take special measures to use it. Ideally we would prefer to use a 3D model such as PHIGS. However, current graphics technology, especially hardcopy, favours PostScript. PostScript has been used successfully as the basis of the graphical output of NeWS. The PostScript imaging model, with extensions for rendering three dimensional objects, satisfies all of our criteria.

Integrating an object oriented programming language, such as Smalltalk, with a PostScript imaging model can be approached in several ways. Central to this problem is the question of how much PostScript should be incorporated in the system. One solution is to include the entire PostScript language, as done in NeWS; the other is to include only the PostScript imaging model. Using a PostScript interpreter from Smalltalk requires that Smalltalk objects generate PostScript code. Automatic translation of Smalltalk methods into equivalent PostScript procedures is a convenient approach (from the Smalltalk programmers point of view). However, implementing this translator would be difficult; since Smalltalk supports operator overloading and late binding.

Instead, we have implemented a Smalltalk class **PostScriptPen**, that implements the PostScript imaging model in Smalltalk. PostScriptPen encapsulates the interface between Smalltalk and PostScript. Messages are sent to a PostScriptPen when graphical output is desired. Programmers can extend the available imaging operations by writing Smalltalk methods that use the imaging operations provided. At the present time, a subset of the PostScript imaging model is implemented in SmallTalk. Since all imaging operations are based on the same primitives as PostScript, producing a hardcopy version of a display is simply a matter of sending the same operations to a PostScript printer. The classes that have been added to the original Smalltalk/V image are listed in Appendix A.

5.2 SmallScript User Interface Organization

In order to distribute the user interface and the application, interprocess communication primitives based on the Berkeley socket model were added to Smalltalk/V. These primitive operations are used by a network transparent interface based on Proxy objects [Bennett, 1987]. A proxy object is responsible for receiving all requests for service from the remote machine and ensuring that a message is sent to the computer that actually contains the object being referenced. By reimplementing doesNotUnderstand in Proxy, references to a remote object are automatically converted into Ethernet messages. (Figure 2 provides a high level overview of our system)

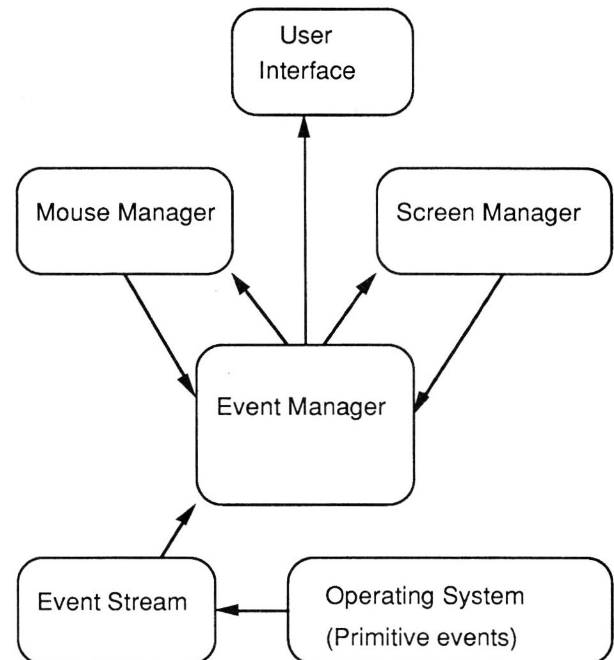

Figure 2. SmallScript Organisation

There are three classes in our window system. There are objects that supply information (**models**), objects that display information (**views**) and objects that coordinate the behaviour between views (**coordinators**). Models have two responsibilities in existing Smalltalk window systems (eg MVC, MPD): supplying data and implementing application methods which use the data. This combination of data access and application works in single processor systems. It is unacceptable in a heterogeneous computing

environment since application functions may be written in languages other than Smalltalk.

Views display information stored in a model. Typical views are textViews, labelViews, listViews, menuViews and borderViews. Views are responsible for implementing their visual behaviour. For example, when a textView is the active view and the user types a character, the character is displayed at the insertion point which then moves to the right. When the mouse is moved within a listView, the selection underneath the mouse will highlight itself, and return to normal when the mouse leaves.

Views register for the events that they require to implement their visual behaviour. For example borderViews, once displayed, do not change their appearance; therefore they do not register for any events. Text views however, register for events such as keyPressed and leftButtonClick and listViews register for mouse movement and button click events.

Separating the model from the view allows multiple views on the same model. By using only models and views, a window system could not handle dependencies between windows. Two or more windows are dependent if actions in one affect the behaviour of another. For example, selecting an element in the selectors pane of a class hierarchy browser changes the browser's text pane. Dependency management is a significant feature missing from the first implementation of EVA [McAffer, 1987]. Coordinating the dependencies between views is the responsibility of a coordinator. Multiple coordinators are organized into a directed acyclic graph. Communication between these levels is achieved by sending messages corresponding to events. Views inform their coordinator when a significant event has occurred. The definition of what constitutes a significant event depends on the application. The default implementation of listView treats the selection of an entry in the view as a significant event and will generate a selection event every time it occurs. These events are handled by the coordinator for the view. At any one moment there is only one active view. (This will be eliminated when SmallScript is moved into the Actra environment [Thomas D, Lalonde W, Pugh J, 1986]) To signify that a view is active, the label associated with the window containing the active view is highlighted. All user input is accepted by the active view until the user activates another view.

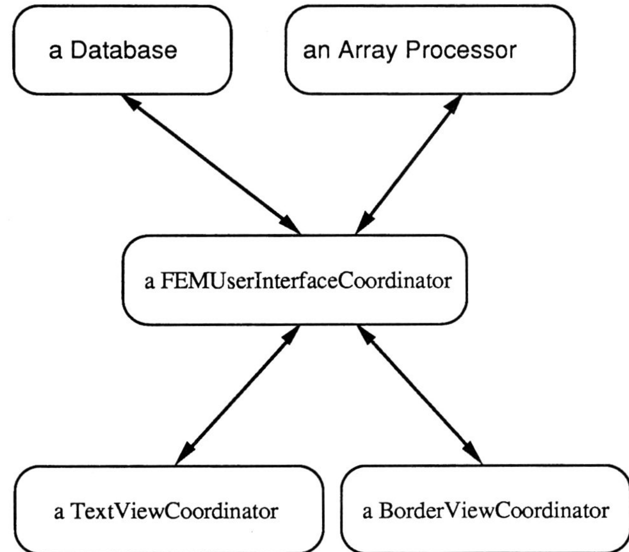

Figure 3. Design of a Simple Application

As previously mentioned, coordinators are arranged in a directed acyclic graph, with coordinators closer to the root coordinating higher levels of behaviour. Coordinators at the leaves of the tree coordinate the behavior of one view and at the next level, they coordinate dependencies between multiple views. The root of the coordination tree is the application. The application(s) receive high level requests for actions such as a query of a database. For example, in Figure 3, a high level design for a system requiring access to a database and manipulation of matricies (eg. PHIGS) is presented.

5.3 The Event System

An event represents an action of importance to the system (See figure 4). Pressing a key on the keyboard or moving the mouse are examples of physical events. The event system in SmallScript is composed of four classes: **Event**, **EventStream**, **EventManager** and **Interest**. The event stream interacts with the underlying operating system to translate physical events into their Smalltalk equivalents. Instances of an EventStream have one public method (called next), which returns the next event. Event streams convert the operating system dependent information into Smalltalk events. Events have instance variables that identify the type, originator, data and time of the event.

```
An Event
 type:      leftButtonDown
   sender:  anEventStream
   data:    100@100
    time:    12332
```

Figure 4. Example of an event.

Compound events such as button clicks are generated by a button down followed by a button up event. When a mouse button down event occurs, a future, button-held event is generated. A future event is an event whose time is greater than the current time. Generating events that will occur in the future is simply a matter of specifying the time at which the event should occur and posting it.

Event managers are responsible for distributing events to all interested objects. Events that have been posted but not yet dispatched are stored in a priority queue. When the time of the event at the head of the queue is less than or equal to the current time, it is dequeued and dispatched to all objects which have expressed an interest in receiving events matching the current event.

Any Smalltalk object can express an interest in being notified when an event occurs. When expressing an interest in a particular event, the interested object creates an instance of **Interest**. Before an event matching an object's interest is dispatched, the interested field is checked to make sure its value is true. This allows for objects to toggle their interest without revoking and reexpressing their interest. An Interest may optionally include a block of code which must return true in order for the event to match. Text editors, since they are only concerned about characters being typed in their window, uses this filtering facility. An event manager adds the interest to a collection of interests associated with the event.

When the event manager dispatches an event, it goes through the list of interests associated with the event type, informing all objects whose interest matches the current event. If a matching interest is exclusive, no other object is told about the event. When expressing its interest, a proxy can be specified. The proxy for an interest performs actions on behalf of the interested object.

5.4 MouseManager and ScreenManager

Our window system uses higher level events than the ones returned by EventStream. Events such as button clicks and window boundary crossings are important to the window system. Transforming low level events into higher level events is the responsibility of a manager. Managers express interest in low level events and combine one or more of them into higher level events. For example, a leftButtonDown and a leftButtonUp will be combined to form a leftButtonClick. There are two types of managers in the current system: MouseManagers and ScreenManagers. A mouse manager generates high level mouse events, and a screen manager generates high level screen events.

A finite state machine is used by the MouseManager to transform left button activities into leftButtonClick, and select events. A similar finite state machine is used to transform right button actions into rightButtonClick and scrollEvents. Extending this finite state machine to transform two clicks into a doubleClick is simply a matter of adding a few more states and transition paths. The ability of the event kernel to accept events occuring in the future is used extensively by the mouse manager.

6 Conclusion

SmallScript addresses two major issues: how to exploit a heterogenous computing environment and how to provide a flexible user interface. Effective use of available computing resources requires that a system be decomposed into functional components. SmallScript is the user interface component. Event objects are used to communicate between components in a SmallScript application. Event objects can describe physical events, such as the user typing a character, as well as abstract events, such as a database query. Since components (coordinators) communicate via these events, distributing an application over a network is easy. Furthermore, coordinators do not have to be written in Smalltalk, which allows the developer to exploit existing code libraries.

The current prototype is implemented in Smalltalk /V on an IBM AT. In our prototype the PostScript pen is simulated using bitblt. The next version will use a TI34010 [Texas Instruments, 1986] based PostScript imaging system derived from GhostScript [Deutsch, 1987] and will be moved to a multiprocessor Smalltalk [Thomas D, Lalonde W, Pugh J, 1986].

Appendix New Smalltalk Classes

ClassBrowserApplication ()

Coordinator (parent)
- ComplexCoordinator ()
  - MyClassBrowser ()
  - Editor ()
- SimpleCoordinator (view menu)
  - BorderViewCoordinator ()
  - LableViewCoordinator ()
  - ListViewCoordinator ()
  - MenuViewCoordinator ()
  - TextViewCoordinator ()

Event ()
- Interest ()

EventManager ()

EventStream ()

InternetAddress ()

Model
- TextModel ()

MouseEventManager ()

NetworkManager ()

PostScriptPen ()

Proxy ()

ScreenEventManager ()

Socket ()
- TCPSocket ()
- UDPSocket ()

View ()
- BackgroundView ()
- BorderView ()
- LableView ()
- TextView ()
- ListView ()

References

Adobe 1984 *PostScript Language Manual*, Adobe Systems Inc., Palo Alto, California.

APDA 1986, *MacApp: The Expandable Macintosh Application,* Apple programmers and developers association, APDA #KMSAPD, 290 SW 43rd Street, Renton WA 988055

Bennett John 1987, *The Design and Implementation of Distributed Smalltalk,* OOPSLA 87 Conference Proceedings.

Deutsch 1987, GNU Postscript, Free Software Foundation.

Gentlemen 1988, M. Private communication 1988.

Goldberg 1983, *Smalltalk-80: The language and its implementation,* Adele Goldberg and David Robson, Addison-Wesley 1983.

Goodman 1987, *The Complete HyperCard Handbook,* The Macintosh Performance Library, Bantom Computer Books 1987.

McAffer, J. and Thomas D.A. 1987, *Eva: An Event Driven Interface for Smalltalk* , Graphics Interface Conference 1988.

NeWS 1987, *NeWS Technical Overview* , Sun Microsystems Inc., Mountain View, California.

Pfaff G.E. (Ed) 1985. *User Interface Management Systems,* Proc. *Seeheim Workshop on User Interface Management Systems,* Nov 1983, Springer-Verlag, Berlin(1985)

Rosenthal 1988, *A Simple X11 Client Program or How hard can it really be to write "Hello World"?* David S. H. Rosentahl Sun Microsystems 1988.

Scheifler Robert W 1986, *The X Window System,* Robert W Scheifler, Jim Gettys, MIT October 1986.

StepStone 1986,*Objective C Reference Manual,* StepStone Inc.

Texas Instruments Inc. 1986, *TMS34010 User's Guide,* Texas Instruments Inc., Houston, Texas.

Thomas, D., Lalonde, W. and Pugh J. 1986, *ACTRA, A Multitasking/Multiprocessing Smalltalk,* Technical Report No. SCS-TR-92, Carleton University, School of Computer Science, Ottawa, Ont.

WhiteWater 1987, Actor *Reference Manual* , The WhiteWater Group.