SMALLSCRIPT: A USER PROGRAMMABLE FRAMEWORK BASED ON SMALLTALK AND POSTSCRIPT

By Kevin Haaland and Dave Thomas

SCS-TR-149

November 1988

Abstract

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

This research was supported in part by DREO and DRMO.

School of Computer Science, Carleton University

Ottawa, Canada K1S 5B6

Introduction

There are numerous applications in which a user programmable framework (UPF) for user interface development are 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. Despite the wide spread interest in traditional UIMS systems [Pfaff 1985] and standardized toolkits [X, MacApp, ICpak 201] these systems fail to meet the demanding needs of evolving user interface software. The recent trend in CAD applications is to include an embedded programming language. (eg. AUTOLISP) The need for user programmable interfaces has inspired a wide spread increase in Smalltalk which remains the most powerful environment for the interactive exploration of new interface metaphors. Smalltalk's graphics facilities are still based on the low-level **bitblt** primitive. It is unfortunate that a language with such power is still limited by the raster imaging model of the 70s. One of the primary goals of SmallScript is to address this weakness.

Two recent UPFs are NeWS and HyperCard. 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 environment to communicate with the more traditional environment via a well defined (albeit restricted) interface. HyperCard provides a tangible user interface metaphor based on index cards combined with a simple but ill specified command language HyperTalk. HyperCard is clearly the successor to MacPaint and is unfortunately limited by its use of bitmap graphics (MacDraw would have been a better model). HyperCard achieves much of its simplicity by severly restricting the metaphor (eg. single fixed size card). In its current implementation, developing large applications is difficult since HyperCard lacks code browsers and debugging support. It is extremely awkward to interface HyperCard to an existing C/Fortran application. NeWS provides a powerful programming capability based on PostScript. The PostScript interpreter communicates with the application program via a stream connection. In this regard it is much easier to connect a NeWS interface with existing software, although there is no reason in principle that the same approach could not be used with HyperCard via it's XCMD facility. Unlike HyperCard or Smalltalk the NeWS programming environment is reminisent of its forerunner Forth.

In this paper we describe SmallScript, an integrated environment for user interface development. SmallScript like NeWS has a PostScript based graphics imaging model. SmallScript applications 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.

Separating The Application from The Interface

Our primary interest is in applications such as Finite Element Analysis (FEM) which are CPU intensive and produce complex graphical output from graphical input. We therefore must consider a computing enviroment that consists of multiple computers that communicate by message passing. 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 non linear equations in FEM). 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. This approach is used in NeWS, Andrew and X where **clients** and **servers** are used to describe (inconsistently) the two components above. In the case of Andrew and X however the client and the server are assumed to be different machines.

The X Window System

The X window system provides a basic set of display primitives and input routing facilities that are available to application programmers. The X Window system developed at MIT uses asynchronous, stream-based interprocess communication facilities. 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. 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]. Adding new operations to the X Windows server is not possible and applications are responsible for low level activities such as tracking the mouse and restoring windows in the current implementation.

NeWS

The Sun Network extensible Window System (NeWS) is an advanced windowing system that uses PostScript in a distributed window server. PostScript displays text, graphics and sampled images in a uniform and 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.

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

A NeWS application begins by establishing a connection with a NeWS server. After the connection, the client program 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 will respond to events from the server as well as send PostScript code to the server. Unless the client program is written entirely in PostScript, it is divided into two components. The first component generates PostScript code that is sent to the server. The second component implements the rest of the application. Although the presence of PostScript may be hidden by language bindings, the application writer is always aware that PostScript is present.

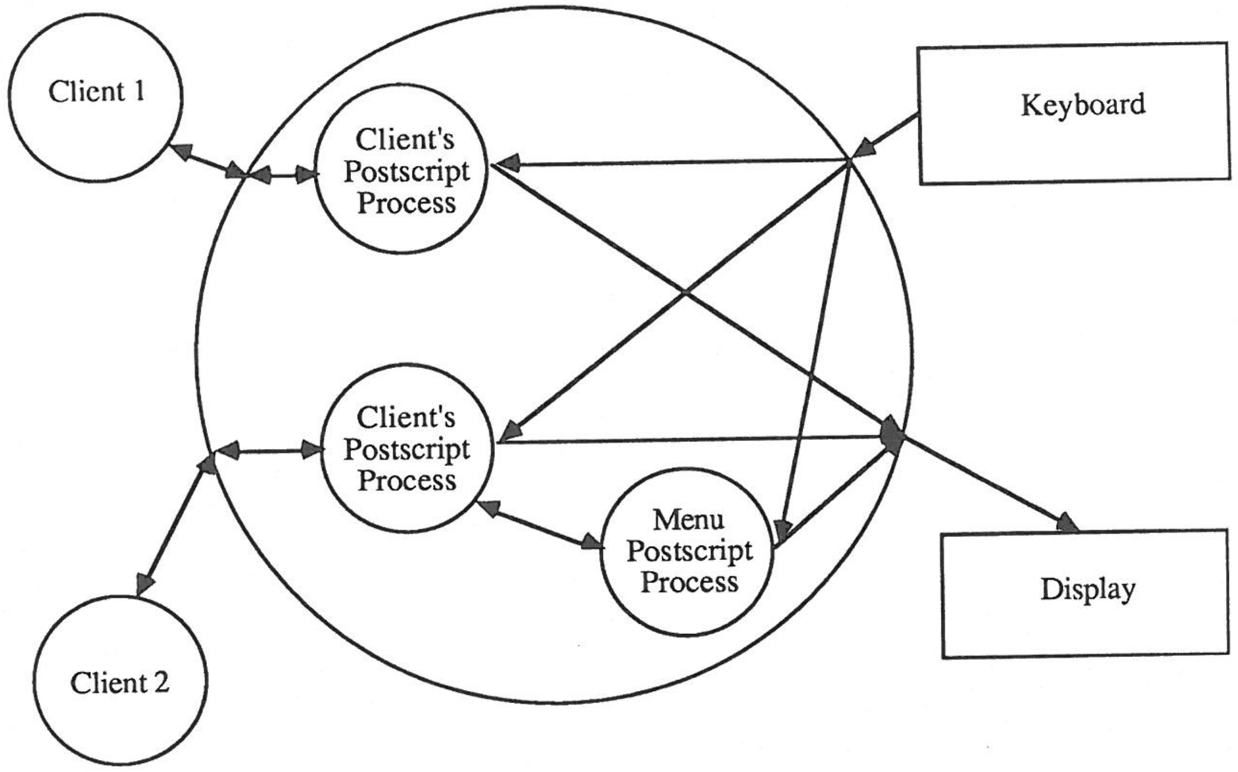

Figure 1 Client server relationship in NeWS

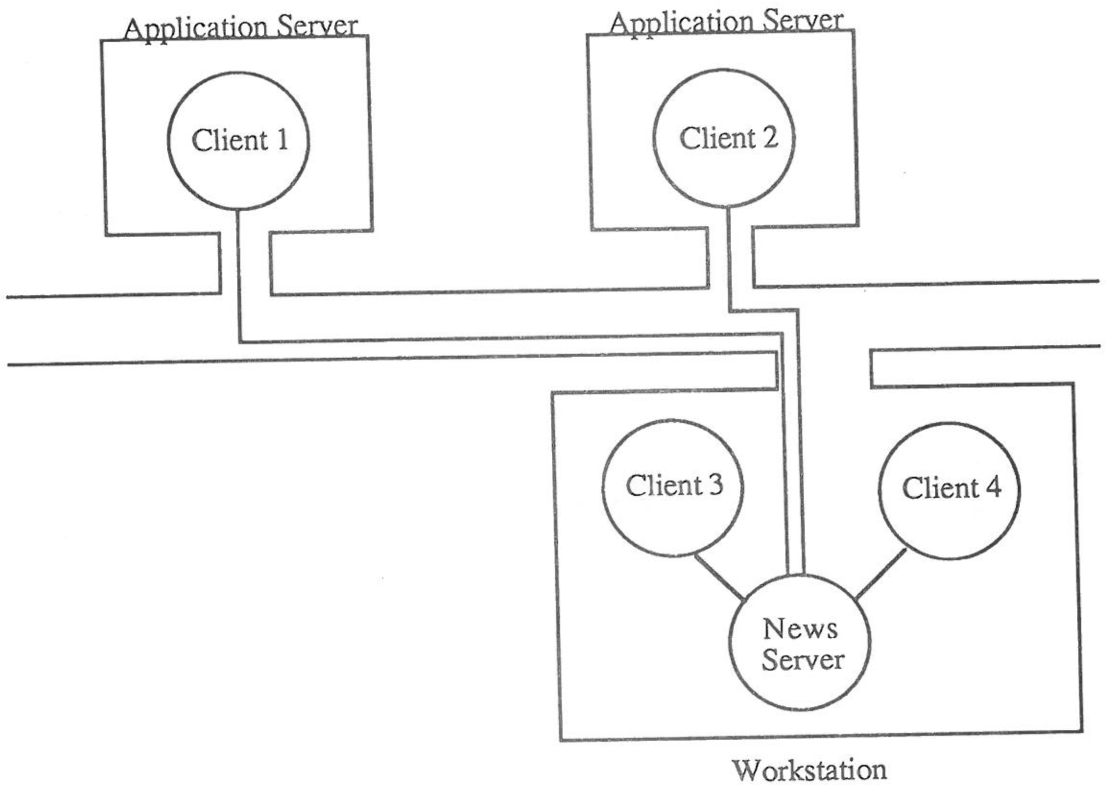

Figure 2 Model of distributed applications in NeWS

User Interfaces Require Object-Oriented Programming

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 the reality of the user. For example a user interface that incorporates file folders, filing cabinets and a garbage can is appropriate for applications that are used by office workers because they are familiar with the behaviour of these objects. Modelling objects complete with their behaviour is a difficult task for programming languages such as Fortran. Object oriented languages such as Smalltalk, Actor [WhiteWater] and Objective-C [StepStone] are well suited to this task. These object oriented programming languages provide the programmer with a set of generic software IC's which can be customized to suit specific needs. This customization process builds on the existing functionality of the system. This ability to perform fast prototyping of ideas is needed to allow tailoring of the user interface. Object oriented languages 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. This modelling capability is the major reason why user interfaces should be written in an object oriented language.

The Changing Structure of Application Programs

Producing a prototype of a large system such as a CAD/CAM package will be much faster when the user interface is implemented in an object oriented language. In a CAD system for electrical systems, the user interface includes objects that correspond to the physical objects being manipulated and an ability to connect electrical objects to form a circuit. The user interface invokes application specific algorithms to do the work. Using this approach, what is commonly referred to as the application can be described more accurately as a set of application specific functions that user interface invokes to do work. The application functions can now be implemented in the language and computer most suitable for their execution.

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 and Andrew implement the windowing routines on the server but 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 events such as bit maps, line drawing commands, key presses and mouse movements. It is often argued that forcing a system to be divided into an application and a user interface is an awkward solution which doesn't work in real systems. This is the same argument used in the seventies when the notion of structured programming was introduced. As with structured programming, 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.

To gain a better understanding of the behaviour of SmallScript, consider an interactive database system. The user of this system will interact using a mouse and high resolution display. The user interface is responsible for presenting the graphical images supplied and implementing their behaviour. For example when the user selects something with the mouse, the selected object highlights itself to inform the user that it has been selected. Then if the user decided to resize a window on his display the user interface does the sizing and displaying of the information with no interaction between it and the application. If during the course of using this system, a query of the database is generated, the user interface forwards the query to the application where it is processed. When the application has finished the query, it sends the results to the user interface. Since the application and the user interface communication only via high level messages, the cost of inter-machine communication is minimized. Until the cost of sending messages between processors is reduced to the point of being equivalent to a procedure call; reducing the number of message sends will improve overall performance. Therefore, the tasks that are assigned to each processor must be disjoint and the time required to perform the task must be significantly longer than the time required to send the message that causes its execution. Both X Windows and Andrew fail in both respects. Applications that use NeWS as their window system send fewer messages but clients and servers still communicate with low level messages for input. Furthermore servers in all three systems will not perform any operation unless it is explicitly told to by a client. The reason why X, Andrew and NeWS are not optimal in a heterogeneous computing environment is due to the fact that the programs that use these system are designed as if they were executing on a single processor system. Clearly for such applications to be efficient an application specific object oriented protocol is far superior to any low level generic protocol [Gentleman 88].

Smalltalk versus Postscript as a UIF Programming Language

For all the reasons why Smalltalk is a good environment to write a user interface, PostScript is not. PostScript lacks a debugger, code browser and class library. PostScript objects are bolted on, instead of integrated with the language. When a new PostScript interpreter is forked by the window server it inherits the address space of its parent. The child process may push and pop dictionaries to and from its private stack, thus controlling the amount of name sharing that takes place. Using this method, a simple object oriented system may be implemented where subclasses inherit behaviour from their superclass, modifying behaviour where necessary. The light window system provided with NeWS uses this capability. Implementing objects with state and behaviour in PostScript is both awkward and incomplete when compared with languages such as Smalltalk, Actor or Objective-C. Unlike PostScript, Smalltalk has an excellent development environment, supports inheritance, class variables, and it has a large assortment of existing classes. PostScript is a language for describing pages and it seems unlikely that a Forth based language can be scaled to suit the needs of a User Interface Framework.

The SmallScript Imaging Model

Imaging models describe, in general terms, the display capabilities of a graphics system. The coordinate system used, the types of lines that can be drawn, support for colour, and text are some of the more important aspects that one should consider when assessing the merits of an imaging model. The ideal imaging model for the system being developed is one that implements an abstract imaging model capable of rendering 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 the programmer taking special measures to use it. 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. Ideally we would prefer to use a 3D model such as PHIGS, however current graphics technology, especially hardcopy favours PostScript.

The problem still remains however, of how to integrate an object oriented programming language such as Smalltalk with a PostScript imaging model. 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[NeWS]; the other is to include only the PostScript imaging model. Using a PostScript interpreter from Smalltalk requires Smalltalk objects to generate PostScript code. We consider translation of Smalltalk methods into equivalent PostScript procedures a convenient approach for the Smalltalk programmer. Implementing this translator would be difficult since Smalltalk supports operator overloading and late binding. Instead, we implemented a Smalltalk class **PostScriptPen**, that encapsulates the interface between Smalltalk and PostScript. PostScriptPen has methods that implement the PostScript imaging model. Smalltalk draws lines and text by sending messages to a PostScriptPen. Programmers can extend the available imaging operations by writing Smalltalk methods that use the imaging operations provided. In a system that generates a large number of complex displays; SmallScript could be moved to a multiprocessor Smalltalk[Thomas D, Lalonde W, Pugh J, 1986], or PostScriptPen could be implemented on a graphics coprocessor. 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.

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]. Proxy objects relieve the programmer from taking special measures to deal with remote objects. 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. Proxy objects reimplement **doesNotUnderstand:** to forward a request to a network manager. A NetworkManager is responsible for building ethernet packets and sending them out the correct socket. For efficiency reasons, a network manager can be told to forward a message and return immediately instead of waiting for the result from the remote machine.

We implemented our own window system to avoid the limitations of existing systems such as EVA [McAffer 1987], Model Pane Dispatcher (MPD) [Digitalk 1986], and Model View Controller (MVC) [Goldberg 1984]. We need to enforce a clean separation between applications that may be written in a language other than Smalltalk and the user interface.

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 that use the data. This combination of data access and application works in single processor systems but is unacceptable in a heterogeneous computing environment since application functions may be written in languages other than Smalltalk. As a result, models in SmallScript perform data access functions only.

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, it displays the character at the insertion point and moves the insertion point 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 but text views register for events such as keyPressed and leftButtonClick.

Separating the model from the view that displays it 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. 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. Coordinating the dependencies between views is the responsibility of a **coordinator.** Multiple coordinators are organized into a directed acyclic graph where the parents are responsible for higher levels of responsibility. Communication between these levels is done by sending messages that correspond 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 an event every time it occurs. In the current implementation, clicking the right mouse button will pop up a menu. 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) To signify that a view is active, the label associated with the window that contains the view is highlighted. All user input is accepted by the active view until the user chooses another view to become active. A view becomes active when the mouse crosses an internal boundary between subwindows or the user clicks the left button inside a window that was not previously the active window. Determining when to highlight the label is the responsibility of the coordinator for the subviews.

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 coordinator tree is the application. Applications do not concern themselves with deciding what to do when the user clicks the left button while the mouse is in a particular window. Instead the application receives high level requests for action such as a query of a database.

The Event System

An event represents an action of importance to the system. 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 to their Smalltalk equivalent. Instances of an EventStream have one public method called **next**, which returns the next event. Events 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 3 Example of a SmallScript primitive event.

Compound events such as button clicks are generated by a button down followed by a button up event within a short time period. When a mouse button down event occurs a future button held event is generated. 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, with the event with the lowest time at the head of the 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 that have expressed an interest in receiving events that match the current event.

Any Smalltalk object can express its interest in being notified when an event occurs. When expressing an interest in a particular event the interested object creates an instance of an **Interest**. Before an event that matches an object's interest is dispatched, the interested field is checked to make sure it is true. This allows for objects to toggle their interest without revoking and rexpressing their interest. Interests may optionally include a block of code which must return true for the event to match. This testBlock is used by a text editor window to express its interest in key pressed events. The text editor only cares about characters being typed in its window and ignores all characters typed outside its window. 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 the objects whose interest matches the current event until all interests have been examined or until an exclusive interest is found. If an 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. When dispatching an event, the event type corresponds to a method name implemented in the class of the object being notified and the data of the event are the parameters to the method.

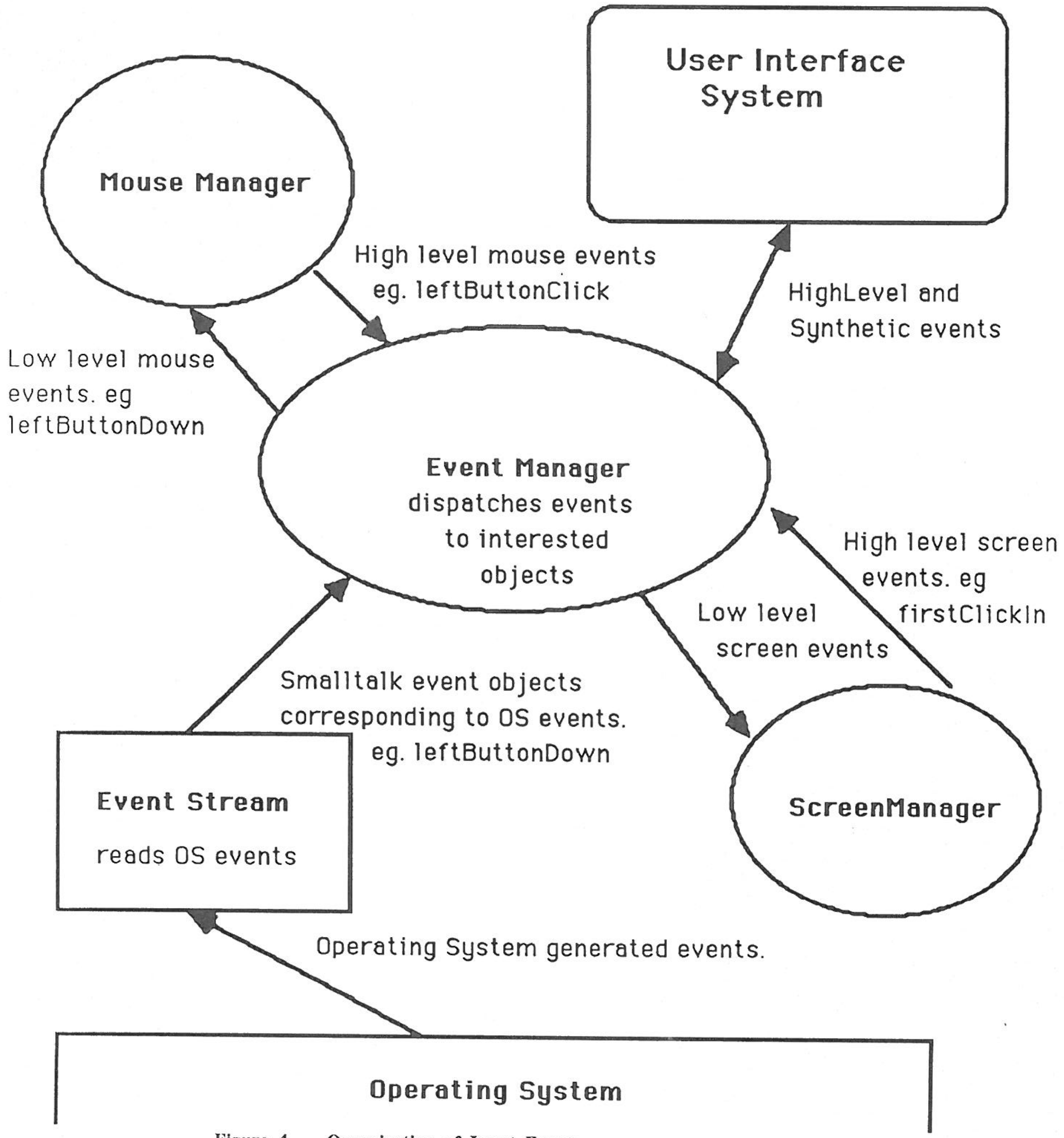
User Interface System
Mouse Manager
High level mouse events eg. leftButtonClick
HighLevel and Synthetic events
Low level mouse events. eg leftButtonDown
Event Manager dispatches events to interested objects
High level screen events. eg firstClickIn
Low level screen events
Smalltalk event objects corresponding to OS events. eg. leftButtonDown
Event Stream reads OS events
ScreenManager
Operating System generated events.
Operating System

Figure 4 **Organization of Input Events**

MouseManager and ScreenManager

Our window system uses higher level events than the ones returned by EventStream. Events such as button clicks and crossing the boundary of a window 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.

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

Moving the mouse across a window boundary generate two events: cursorOut and cursorIn. These events are sent to the window that previously contained the mouse and the window that currently contains the mouse respectively. The screen manager is responsible for generating these events. In addition it generates events that are of interest to specific windows. For example, the first time a leftButtonClick event occurs in a window, the window becomes active and the screen manager is responsible for creating the activate message.

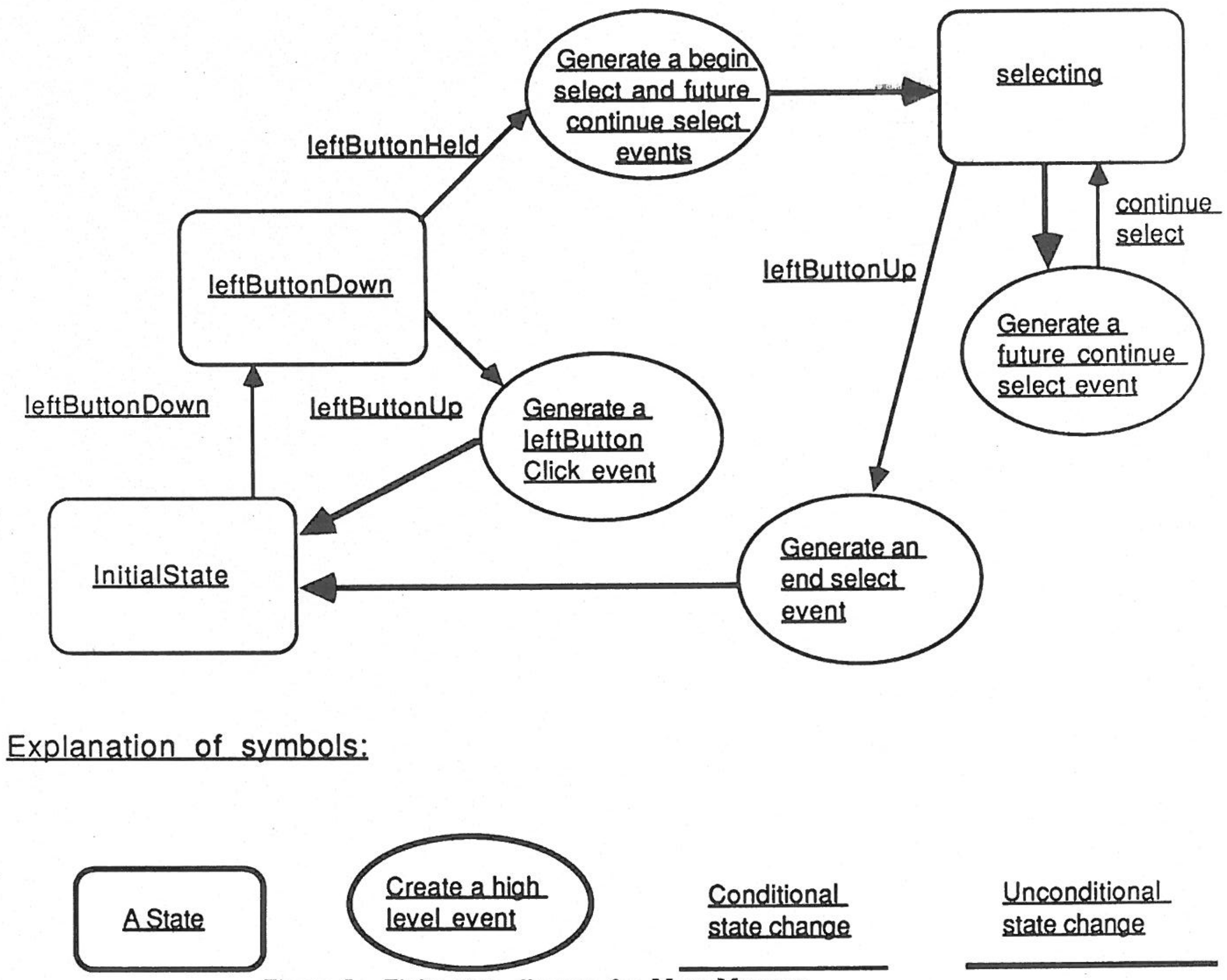
Generate a begin select and future continue select events
selecting
leftButtonHeld
continue select
leftButtonDown
leftButtonUp
Generate a future continue select event
leftButtonDown
leftButtonUp
Generate a leftButton Click event
InitialState
Generate an end select event
Explanation of symbols:
A State
Create a high level event
Conditional state change
Unconditional state change

Figure 5. Finite state diagram for MouseManager

```
ClassBrowserApplication ()

Coordinator  (parent)
           ComplexCoordinator (lable border active)
            MyClassBrowser (classOrInstance selectors text browsedClass
                                        selectedDictionary selectedMethod)
          Editor (text)
      SimpleCoordinator (view menu)
           BorderViewCoordinator ()
           LableViewCoordinator ()
           ListViewCoordinator ()
            MenuViewCoordinator ()
           TextViewCoordinator ()

Event (type sender receiver time data)
      Interest ( interested exclusive proxy testBlock)

EventManager (objectTable interestTable active eventStream currentEvent)

EventStream ()

InternetAddress ()

Model
      TextModel (lines)

MouseEventManager (state)

NetworkManager (tcpSockets bindings serveRequests)

PostScriptPen (clippingPath currentPoint path pen scanner color lineWidth
                    transformationMatrix currentFont)

Proxy (name returnResult)

ScreenEventManager (activeViews state views backgroundView)

Socket ()
      TCPSocket ()
      UDPSocket ()

View ()
      BackgroundView ()
      BorderView ()
      LableView ()
       TextView ( insertionPoint topCorner)
        ListView (selectedLine)
```

Figure 6. Classes that were added to Smalltalk

Summary and Conclusions

SmallScript implements a multiprocessor user interface envitonment using Smalltalk and PostScript. A new design paradigm that places more importance on the user interface, instead of the application, has been developed. An event driven, object oriented window system has been developed. Unlike previous window systems there is a clear delegation of responsibility between objects and as a result, extending the system is easy; with the required methods normally less than fifteen lines. In a system consisting of multiple dependent windows and subwindows, coordinators are arranged in a tree structure with coordinators closer to the root coordinating the higher level functions of the system. The supporting algorithms are implemented in a coordinator that is often the root of the coordinator hierarchy. 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 based PostScript imaging system derived from GhostScript [Deutsch 88].

References

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

ANSIb dpANS Information Processing Computer Graphics Virtual Device Interface (VDI) Functional Description.

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

Deutsch P, GNU Postscript, Free Software Foundation.

Foley, *Interfaces for Advanced Computing,* James D. Foley.

Foley and Vandam 1984, *Fundamentals of Comuper Graphics.* Prentice Hall pg 299.

Gentlemen, M. Private communication 1988.

Intel Inc. (1987) *82786 Graphics Coprocessor User's Manual,* Intel Inc., Santa Clara, California.

ISO (1982) *Graphics Kernel System* (GKS) *Functional Description,* Draft International Standard ISO/DIS 7942, ISO TC97/SC5/WG2 N 163, November 14th, 1982.

Kernighan and Ritchie (1978), *The C Programming Language,* Brian W. Kernighan & Dennis M. Ritchie, Prentice Hall July 1978.

Korth 1986, *Database System Concepts,* Henry F. Korth, Abraham Silberschatz, McGraw-Hill Book Company 1986.

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

Newman 1979, *Principles of Interactive Computer Graphics,* McGraw Hill Book Company 1979.

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

Nova 1987, *GKS Primer. CGI/CGM Primer,* Nova Graphics International Corporation Austin Texas.

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

PHIGS (1985), *American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS)* (draft) , ANSI Document X3H3/85-21, February 18, 1985.

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

RPC, *Remote Procedure Call Programming Guide* (1986), Sun Microsystems Inc., Mountain View California.

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

Smith, R. (1986) The Alternate Realities Kit: An Animated Environment for Creating Interactive Simulations, *Proceedings of: 1986 Computer Society Workshop on Visual Languages*, Dalas, Texas.

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, Actor *Reference Manual*, The WhiteWater Group.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-120	**Searching on Alphanumeric Keys Using Local Balanced Tree Hashing** E.J. Otoo, August 1987.
SCS-TR-121	**An O(√n) Algorithm for the ECDF Searching Problem for Arbitrary Dimension on a Mesh-of-Processors** Frank Dehne and Ivan Stojmenovic, October 1987.
SCS-TR-122	**An Optimal Algorithm for Computing the Voronoi Diagram on a Cone** Frank Dehne and Rolf Klein, November 1987.
SCS-TR-123	**Solving Visibility and Separability Problems on a Mesh-of-Processors** Frank Dehne, November 1987.
SCS-TR-124	**Deterministic Optimal and Expedient Move-to-Rear List Organizing Strategies** B.J. Oommen, E.R. Hansen and J.I. Munro, October 1987.
SCS-TR-125	**Trajectory Planning of Robot Manipulators in Noisy Workspaces Using Stochastic Automata** B.J. Oommen, S. Sitharam Iyengar and Nicte Andrade, October 1987.
SCS-TR-126	**Adaptive Structuring of Binary Search Trees Using Conditional Rotations** R.P. Cheetham, B.J. Oommen and D.T.H. Ng, October 1987.
SCS-TR-127	**On the Packet Complexity of Distributed Selection** A. Negro, N. Santoro and J. Urrutia, November 1987.
SCS-TR-128	**Efficient Support for Object Mutation and Transparent Forwarding** D.A. Thomas, W.R. LaLonde and J. Duimovich, November 1987.
SCS-TR-129	**Eva: An Event Driven Framework for Building User Interfaces in Smalltalk** Jeff McAffer and Dave Thomas, November 1987.
SCS-TR-130	**Application Frameworks: Experience with MacApp** John R. Pugh and Cefee Leung, December 1987.
SCS-TR-131	**An Efficient Window Based System Based on Constraints** Danny Epstein and Wilf R. LaLonde, March 1988.
SCS-TR-132	**Building a Backtracking Facility in Smalltalk Without Kernel Support** Daryl H. Graf and Wilf R. LaLonde , April 1988.
SCS-TR-133	**NARM: The Design of a Neural Robot Arm Controller** Wilf R. LaLonde and Mark Van Gulik, March 1988.
SCS-TR-134	**Separating a Polyhedron by One Translation from a Set of Obstacles** Otto Nurmi and Jörg-R. Sack, December 1987.

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-135	**An Optimal VLSI Dictionary Machine for Hypercube Architectures** Frank Dehne and Nicola Santoro, April 1988.
SCS-TR-136	**Optimal Visibility Algorithms for Binary Images on the Hypercube** Frank Dehne, Quoc T. Pham and Ivan Stojmenovic, April 1988.
SCS-TR-137	**An Efficient Computational Geometry Method for Detecting Dotted Lines in Noisy Images** F. Dehne and L. Ficocelli, May 1988.
SCS-TR-138	**On Generating Random Permutations with Arbitrary Distributions** B. J. Oommen and D.T.H. Ng, June 1988.
SCS-TR-139	**The Theory and Application of Uni-Dimensional Random Races With Probabilistic Handicaps** D.T.H. Ng, B.J. Oommen and E.R. Hansen, June 1988.
SCS-TR-140	**Computing the Configuration Space of a Robot on a Mesh-of-Processors** F. Dehne, A.-L. Hassenklover and J.-R. Sack, June 1988.
SCS-TR-141	**Graphically Defining Simulation Models of Concurrent Systems** H. Glenn Brauen and John Neilson, September 1988
SCS-TR-142	**An Algorithm for Distributed Mutual Exclusion on Arbitrary Networks** H. Glenn Brauen and John E. Neilson
SCS-TR-143	**Time Is Not a Healer: Impossibility of Synchronous Agreement in Presence of Transmission Faults** N. Santoro, November 1988
SCS-TR-144	**Distributed Function Evaluation in Presence of Transmission Faults** N. Santoro and P. Widmayer, November 1988
SCS-TR-145	**Fault Intolerance of Synchronous Networks** N. Santoro and P. Widmayer, November 1988
SCS-TR-147	**On Transparently Modifying Users' Query Distributions** B.J. Oommen and D.T.H. Ng, November 1988
SCS-TR-148	**Smallscript: A User Programmable Framework Based on Smalltalk and Postscript** Kevin Haaland and Dave Thomas, November 1988