Tabula Rasa
A Multi-scale User Interface System

by

David Fox

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Computer Science
New York University
May 1998

Dissertation Advisor

Discard This Page

Acknowledgements

My thanks go to Nat Kuhn, who came to my eighth grade class in 1973 to recruit members for the R.E.S.I.S.T.O.R.S. computer club at Princeton University. I would also like to thank Andy van Dam for his computer graphics course at Brown University, and Arun Netrivali who guided my research and helped me with my first publications at AT&T Bell Laboratories.

The person most responsible for the successful completion of my degree is the academy award winning Ken Perlin, who lured me back into academia and is both a wonderful advisor and a good friend. Thanks also to Cliff Beshers, Matthew Fuchs, Jon Meyer, Don Mitchell and the many others who have been stimulating and inspiring friends and colleagues.

Thanks go to the members of my thesis committee, Jack Schwartz, George Furnas, Chee Yap, and Ed Schonberg. In particular, Bala Krishnamurthy invested a great deal of time and gave invaluable advice.

I'd like to thank my parents Elaine Fox and David Raymond. Finally, my loving appreciation goes to my wife Lydia Thompson.

Dissertation Abstract

Title: Tabula Rasa: A Multi-scale User Interface System
Author: David Fox
Advisor: Ken Perlin

This dissertation develops the concept of a zoomable user interface and identifies the design elements which are important to its viability as a successor to the desktop style of interface. The implementation of an example system named *Tabula Rasa* is described, along with the design and implementation of some sample applications for Tabula Rasa. We show how programming techniques such as delegation and multi-methods can be used to solve certain problems that arise in the implementation of Tabula Rasa, and in the implementation of Tabula Rasa applications.

Over the past thirty years the desktop or WIMP (Windows, Icons, Menus, Pointer) user interface has made the computer into a tool that allows non-specialists to get a variety of tasks done. In recent years, however, the applications available under this interface have become larger and more unwieldy, taking into themselves more and more marginally related functionality. Any inter-operability between

applications must be explicitly designed in.

The Zoomable User Interface (ZUI) is a relatively new metaphor designed as a successor to the desktop interface. It is inspired by the Pad system, which is based on a zoomable surface of unlimited resolution. Just as the desktop interface has a set of essential elements, a ZUI has a set of elements each of which is vital to the whole. These include

- a zoomable imaging model,
- a persistent virtual geography for data objects,
- *semantic zooming* to optimize the utility of screen space,
- *work-through interfaces* for application objects,
- a constraint system for ensuring the consistency of the interface elements.

These basic elements combine to produce an environment that takes advantage of the user's spatial memory to create a more expansive and dynamic working environment, as well as encouraging finer grained applications that automatically inter-operate with various types of data objects and applications.

Contents

List of Figures

Chapter 1

Introduction

The purpose of this dissertation is to introduce Tab, an example of a new style of user interface system which presents the user with a dynamically zoomable space. This introductory chapter describes Tab and discusses why it is interesting, particularly in relationship to the leading user interface style of the past twenty years, the desktop model. The second chapter considers the historical underpinnings of each of the major elements of system. The third chapter presents a description of the design of the low level mechanisms of an implementation of the system called *Tabula Rasa* (or henceforth *Tab*), while the fourth chapter demonstrates how those low level mechanisms are used to implement some conventional user interface objects and some basic multi-scale applications. Finally, a concluding chapter summarizes and suggests some future directions for zoomable user interface (ZUI) systems.

The most common user interface style used in today's personal computers is the *Desktop Metaphor*. Systems which adopt this style are known as desktop systems because of the resemblance of the overlapping windows that appear on the user's screen to overlapping sheets of paper on a desk. The desktop interface achieves its power by bringing together several elements which, in combination, create a new and compelling experience for the user. The WIMP (Window/Icon/Menu/Pointer) system at once enables the user to perform multiple tasks (by using multiple windows and icons), relieves the user of much of the burden of learning and remembering how to use applications (by providing a menu-driven interface), and provides a more dynamic, tactile and intuitive type of interaction (by using a mouse.) All three of these elements were necessary to create a successful interface – each complements the other two in a variety of ways. Windows multiply the number of applications that can be running, increasing the burden of memorization that menus help to relieve. Without a mouse the manipulation of windows could become an arduous task. The mouse makes it much easier to access menus specific to a particular window, and so on.

In this dissertation a new style of user interface system is presented which embodies a new collection of user interface elements which complement one another in a way similar to the way windows, menus and pointers do. This system has three design elements and an underlying programming language methodology. These are the design elements:

- A resolution independent virtual surface with persistence,
- portals and filters,
- semantic zooming, and
- a constraint system.

The underlying programming language methodology of the *Tab* user interface development system is the use of multi-methods and a form of inheritance called *delegation*. These elements will be discussed in detail in the sections that follow.

1.1 User Interface Metaphors

It has been pointed out by some authors, e.g. Ted Nelson [31], that the similarity of a desktop style user interface to an actual desktop breaks down very quickly (real paper doesn't pop to the top of a stack when you click on it), while others have noted that slavishly attempting to duplicate the features of a real desktop would defeat the whole purpose of putting a system on a computer. As Gentner and Nielson note in [16], if your text editor functions exactly like a typewriter, why not just use a typewriter?

This engenders two observations. First, the word "metaphor" is ill chosen. Metaphor is a poetic device, drawing an absolute identity between two concepts. "The moon is a ghostly galleon", rather than "the moon resembles a ghostly galleon." Brenda Laurel [24] points out that *simile* is a more accurate description

of the relationship between a desktop and a desktop-style user interface system. The second observation is that the only interesting things about any computer system are the ways in which it *departs* from the real-world system on which it is based. We incorporate a simile into a system in order for the user to build a mental model of that system. The user then uses that model to reason about the system and deduce its untried capabilities. The desktop model tells us that "this system is like a desktop, but without the physical limitations of a real desktop, *and* with the capabilities (and many limitations) familiar from older command-line oriented systems." Thus, a reasonable user will not be surprised if the screen representation of a folder fails to fatten as we add documents, even though a real file folder would. A user experienced with older computer systems would correctly expect that a desktop system might run out of storage space suddenly, while a real desktop has a more gradual storage exhaustion failure mode.

Indeed, the passage of time allows greater and greater departure from the desktop model as incremental improvements are discovered and added, and users become accustomed to the departures already incorporated. Nelson rejects the whole notion of metaphor in system design, preferring systems that are designed with a consistency from which the user can build a mental model, but with no real-world referent. He refers to this as the principle of *virtuality* [30], but this appears to be more a rejection of slavishly metaphorical design rather than a serious proposal. Linguistics and cognitive science argue against the ability of

any human to think even a single thought without reference to simile. [23]

1.2 The Virtual Surface

The first element of the Tab system is resolution independence. While the historical underpinnings of each of the elements are discussed in detail in chapter two, the notion of incorporating resolution independence into a user interface system is inherited directly from the Pad system. Pad is a name coined by Ken Perlin in 1989. It has been used to refer to a series of systems developed at New York University, and one system developed jointly with the University of New Mexico, each with a different feature set. A paper by Ken Perlin and myself presented at the 1993 SIGGRAPH conference [33] described the notion and the system as it was at that time.

In his book "Designing the User Interface" [36] Ben Schneiderman writes that there are two user interface techniques that are widely reported to engender not only acceptance, but often an enthusiastic response. The first is direct manipulation ([36], p. 202), which is a central component of the desktop system. The second technique is to design a system which presents the user with a large navigable space ([36], p. 222.) The Pad system is intended to capitalize on both these techniques. The central element of the Pad model is a virtual surface capable of nearly limitless detail. This surface is called *virtual* because such a surface can-

not be adequately represented by any foreseeable display equipment. Even if it could, the human eye cannot apprehend the detail such a display would present. Instead, the system creates the impression of limitless detail by allowing the user to navigate while looking down at the surface. The user can move laterally to see different parts of the surface, or can change altitude, up to see a larger portion of the surface, or down to see a smaller portion in greater detail. surface?

Using a low resolution device (such as a computer monitor) to create a high resolution experience of a surface is analogous to the way humans use their eyes to create a high resolution experience of their environment. The eye has a very small area of high resolution in the center of the retina surrounded by an area of low resolution.[1] It is also able to move very quickly, (about 500 degrees per second), and these two features combine to create the experience of being immersed in a world of great visual detail. This suggests that an important element of a system based on a high-resolution virtual surface is that its navigation system be extremely responsive.

We claim that this model has departed from conventional systems to the point where the desktop simile becomes irrelevant. The amount of space on our virtual surface is so vast that it could be used to represent anything, or perhaps everything. Instead of the few icons and windows we find on the screen of a desktop

[1]There are about 150,000,000 receptors in the human eye, but only about 1,000,000 optic nerve fibers, so the number of "pixels" is several million, about that of a high quality computer monitor.

system, our virtual surface could hold *all* of the user's files. The user zooms in towards the folder that contains the file of interest, and at a certain point the individual files appear and the user zooms in on one particular file. When it becomes large enough to work with, the user can read or edit the file.

Note that this arrangement implies a geography for the user's data objects that is persistent. Desktop systems tend to use their surface for relatively transient objects, just as a real desktop would – a document being edited, a window onto an on-line service and perhaps one or two others. The amount of space available on a virtual surface makes this economy unnecessary, the user travels to the document of interest rather than bringing it onto the desk from someplace "outside". Furthermore, the size of the virtual surface makes a persistent geography natural to the user trying to locate objects on it. Humans have developed a great ability for navigating in a familiar geography, and a Pad style system capitalizes on this ability.

The Pad surface could also be used to hold other types of hierarchies. A shared collaborative version could partition the surface among a large number of users. Individual users would never run out of space because they could always just shrink everything down by half. A distributed version of Pad could represent the Internet as a set of nested domains. Zooming in on a particular machine might reveal areas for that machine's users, analogous to todays World Wide Web sites, and if the viewer is privileged, system administration tools and information might

become available, and so on.

Pad was originally conceived to implement a virtual surface rather than a three dimensional space, and this places it somewhere between desktop systems and virtual reality systems in terms of the "Looking at" vs. "Being in" dichotomy discussed by Schneiderman ([36], p. 223.) Pad space is sometimes referred to as a 2.5 dimensional space, which can be thought of as a three dimensional space where the viewer is always looking straight down, and the objects in the space are all flat and face straight up. Of course, these flat objects could be displaying perspective drawings of three dimensional objects. The distance of the objects from the viewer has no effect on their size, just as windows in a conventional desktop interface stay the same size even if they are pushed to the bottom of the stack. This is to say, the system uses an orthogonal projection rather than a perspective projection. As in a desktop system, there is a total ordering on the objects in the space which is called the stacking order. Rather than being determined by the viewer's distance, each object contains a bounding rectangle that defines its size and position independent from its altitude, and a transformation matrix that converts the coordinate system of the space the object exists in to the object's private coordinate system.

We believe that this type of space, where the size of an object is decoupled from its distance, has advantages over a true three dimensional space as an environment for displaying and storing information. It means the visual size of an object can

be controlled by the user, it becomes a design element of the user's graphical computing environment. While we might add the same feature to a fully three dimensional space, it would be that much more confusing because the depth cues in a such a space are that much stronger than in our 2.5 dimensional space. This 2.5 dimensional space also reduces the tendency one has of becoming disoriented or lost in a fully three dimensional space, while still freeing the user from some of the restrictions of the two dimensional space embodied in conventional desktop systems. Finally, we decided that the additional computer power required to implement a fully three dimensional system would be better devoted to improving the responsiveness of this simpler imaging model.

Tab is an attempt to carefully consider what is necessary to make a successful user interface system based on Pad's virtual surface; that is, a system that might successfully compete with the commercial systems. This goal brings up a number of new considerations that would not apply to a system that used the Pad virtual surface to implement a particular application. The main task is to devise an application programmer interface (API) which allows programmers to author interactive software to inhabit this virtual surface. The API must address the questions of how to express both the appearance and the interactive behavior of Pad applications.

1.3 Portals and Filters

One of the features of the desktop system is the ability to juxtapose various objects so they can be compared, or so one can be referred to while working on the other. The Tab system must match this feature within its own paradigm. In a desktop system this is accomplished by loading the objects into windows controlled by applications and dragging windows around. Because we wish to maintain a permanent geography and avoid moving objects, a mechanism called a *portal* has been adopted from the Pad system to allow users to relate objects distant in position or scale or both. In the Tab system, a portal can be linked to any object, and it displays whatever is below that object (Figure 1.1.) Portals are usually linked a simple invisible object type called a *camera*. Not only can a user see through a portal, but also can interact with the objects there just as one interacts with objects directly, that is, if the user's input events hit the monitor they are "transported" to the camera and pass to the objects below it.

We also need to decide how the notion of applications translates into the Pad model. By this we mean tools that can be applied to data such as text, images or a spreadsheet. Most desktop interfaces associate a primary tool with each data object. For example, under Microsoft Windows every document whose filename ends with `.doc` is usually assigned Microsoft Word as a default application that is invoked when the icon for that document is double clicked. However, there

may be a number of tools each of which operate on a particular data object in a different way. There may be one tool that analyses the document's grammar and computes word count statistics, another which could be used to edit figures the document contains, and so on. There seems to be a tendency for a desktop model application to try to do all of these tasks in a single package, while the older command-line model encouraged separating these tasks out into smaller packages each of which (as the slogan goes) "do one thing and do it well."[2] *It is as if in order to escape the deficiencies of the desktop interface all the functionality of the system has drained down into each individual application.* One reason for this tendency may be that there is no intuitive idiom in the desktop model for applying one of several tools to an object, let alone an idiom for applying several tools in "sequence" as in a Unix pipeline.

In order to bring this capability to the user interface it was decided to allow the data to remain stationary on the surface and bring the tool to the data object using a variant of the portal called a *portal filter* or simply *filter*. This is a portal which "understands" a certain type of object and causes objects of that type to

[2]While a desktop user interface has been added to Unix and desktop-style applications are available for it, there is still a thriving Unix sub-culture that remains loyal to command-line oriented tools such as Knuth's TeX [22], the Emacs editor [38], `wc` for doing word counts, and so on. [34] Emacs is interesting as an example of a tool which goes well beyond the command line interface but in a direction entirely different from the desktop model. If the desktop model is a "point and click" interface, the Emacs model might be called a "cut and paste" interface. The user can capture text strings or sequences of keystrokes and combine them into powerful operations. If that is not sufficient, there is a complete implementation of Lisp that can be used to extend the system. This type of programmability is an area in which the desktop interface is notoriously weak. [16]

have a different appearance when viewed through it. A filter may also "mediate" the input events that pass through it on the way to that object, changing them into commands the object understands.[3] By convention, filters in pad usually directly display the area they cover, rather than some remote area as portals often do. (This is enforced using the constraint system, another Tab subsystem.)

As an example, a user might place a word processing filter on top of a text file. When this happens, a cursor appears in the text. Now when the user's mouse is positioned over both the filter and the text file (that is, inside their intersection), text is inserted into the data file as the user types. If more than one text file appears beneath the filter, each has a cursor, and editing commands go to whichever one contains the mouse cursor.

In order to achieve the most versatile model for filter behavior it is important to carefully consider what information belongs to which object. In this example, the text belongs to the data object, while the cursor and its position should be a part of the filter. We don't want the data object to have to carry any information which is specific to the filter's application. There might be another filter which when placed over the text file highlights spelling errors or displays a grammatical analysis of the document – such a filter would have no use for a cursor, and it would be a waste for this information to be stored in the document. This also means that the filter needs to store a set of cursors, one per text file that appears

[3]In object oriented terms, the filter intercepts the input events and directly invokes the methods of the data object.

below it. A general mechanism for managing per-filtered-object information is discussed in section 1.6.

If we are editing a document and we move the editor filter off that document and then back on, we would probably like the cursor to re-appear in the same position we left it. This means we need to retain the cursor information associated with the document even if the filter moves away from it. One might worry that a filter will accumulate thousands of cursor positions, but this is less likely to become a problem if we use semantic zooming to insure that no cursor is generated for documents that occupy too little of the screen to be comfortably edited. The cursor is a small piece of data, but other filters might generate much larger amounts of information associated with each data object – for example, an image processing filter might generate modified versions of the image it is covering. This issue is domain dependent, so it is important that the API provide tools to allow the Pad application programmer to implement various policies regarding when to retain or discard this data. This includes notification of when an object enters or leaves the filter, timers which notify the filter when an object has been absent for a certain period, and a least recently used queue to decide which data can most safely be discarded.

It should be noted that portals and filters, like data objects, are persistent objects in the Pad system. In fact, there is no reason to introduce the notion of a "transient" object. Such an object is an artifact associated with the idea

of turning off the computer, and with the distinction between temporary and permanent memory. We do not believe that this distinction has any value to the user, and therefore its existence should be made transparent. This decision means that filter based applications have no *save* command - the document should always be in a "safe" state, and older versions are made available through use of undo commands and the ability to revert to check-pointed versions of the document.

1.4 Semantic Zooming

In designing a Pad API, our first observation is that rendering commands and event positions must be performed in a floating point space. This brings up some difficulties with regards to rendering operations which have generally been regarded as pixel-oriented - for example, images are usually stored as a grid of pixels, and thus have a "natural size" with relation to the screen's pixel grid. Also, text rendering is usually tuned carefully to the exact size in pixels that the character will appear – even if the characters are stored as filled shapes constructed from polygons and splines, the font will contain hints about how to make sure that artifacts don't show up when converting to a pixel representation. A Pad system needs to take on some tasks that are usually handled by the application in systems where the rendering model gives access to the screen's pixels.

Furthermore, applications still need to know how large they appear on the

user's screen in order to avoid wasting time and screen space rendering detail that is too small to be seen, or too large to be interesting. For example, a Pad representation of a text file might show just the file's name while the user was at a distance where the file is less than an inch across. When the file is a couple of inches across it might display some details about the file such as size, permissions and ownership. Only when it became five or six inches across would it display the textual content of the file. Finally, it would fade out entirely if it got too big, perhaps when the characters exceeded an inch in height.

We have coined the term "semantic zooming" to describe this behavior. It is intended to try to optimize the information carrying capacity of the user's screen in a multi-scale system, just as anti-aliasing optimizes a screen's image displaying ability. The API must include a mechanism by which the application can determine its screen size. Beyond that, however, this topic is mostly one concerning the design of Pad applications, rather than how to solve technical issues. A couple of examples of applications that illustrate some issues regarding semantic zooming are:

Computer Cartography (figure 1.3) - generating maps from a cartographic database at an arbitrary scale involves a number of decisions such as which features to keep or omit and how to do label placement with the features that are selected. The representation of a given feature may also change as the scale

changes – a small city will be a filled shape on a small scale map, while at a scale where it is reduced to less than a pixel size it might continue to appear as a labelled dot due to the fact that the map reader's interest in the town may outweigh its apparent size.

A spreadsheet is visually a table, but it contains a data dependency graph that is usually only visible through interpreting the row and column labels or through typographic conventions. In most spreadsheets, several rows of data will be summed, and the sum will be placed in a row below a double line. Then several of these sums will themselves be summed and the grand total will appear at the bottom. A similar arrangement occurs in the columns. Two levels of summing is usually the limit for a given spreadsheet, and at that point the sum might be carried over to a row of another document.

In Pad this hierarchy might be represented differently, in a single document. The rows of the top level spreadsheet would resolve into multiple rows as you approached, allowing an unlimited depth. You would want separate control of the horizontal and vertical zooming so you could adjust the temporal resolution separately from the conceptual hierarchy represented in the rows. Portals could be used to keep the row and column labels visible even when you were deep within the table – one portal would display the cell entries while the constraint system would cause the other two to track the row and column labels.

This example points up the importance of transitions in semantic zooming – one of the most important things to be conveyed by the spreadsheet is the relationship between the summed rows and the sum. Both must be clearly visible at the same time for the viewer to relate the two.

1.5 Constraints

We have already encountered one situation where we need to maintain a constraint – the convention that the cameras should remain directly below their associated portal if that portal is a portal filter. A constraint system makes it easy to tie various components into a cooperating system, and help system components from falling prey to the syndrome of increasing complexity described in section 1.3.

Actually, the resolution independence of the Tab system makes the satisfaction of constraints much simpler than in many pixel-based systems. This is because it is never necessary to use complicated algorithms to position object in a pleasing way - things can always be fit into the allotted space, and the user can move closer if necessary. Therefore, the Tab constraint system has so far remained quite rudimentary, implemented using a simple call-back keyed to three important types of events in the system: an object being inserted, an object being removed, or an object whose geometry changes. These are the principal types of changes whereby one object might affect another. It is also possible for an object to control another

object of which it has direct knowlege.

1.6 Multi-Methods, Multiple Inheritance and Delegation

Multi-methods, also known as multiple dispatch, is a feature of some object-oriented programming systems whereby a call to a generic function is dispatched to a particular method based on the types of *all* of its arguments - not just the first argument as in languages such as C++ or Java. One language which provides multi-methods is CLOS (the Common Lisp Object System). To dispatch a call, a list of all the methods which are applicable to the argument types is made, and then this list is sorted by the specificity of each method's formals in relation to the types of the call's actuals. Then the most specific method is invoked. Multi-methods are a perfect mechanism for implementing filters, because it is necessary to dispatch each call on the basis of the type of object and the type of filter.

Multiple inheritance is another feature of the object-oriented programming model which allows classes to inherit data and methods from more than one superclass. This mechanism is well suited to composing various types of interactive behaviors in a single object, particularly in combination with multi-methods. For example, in Tab the interactive behavior of an object is determined by a component of the class `<handler>` (as in "event handler.") There are many subclasses

derived from `<handler>` to describe different types of interactive behavior. For example, the `<drag-bindings>` class implements the ability to drag and scale an object using the mouse, while we might have a class `<text-editor-bindings>` which implements keyboard commands to edit text, such as backspace, delete, cursor motion and so on. If we want an object to have both these behaviors we can simply create a new class with both `<drag-bindings>` and `<text-editor-bindings>` as super-classes and allow the multi-method mechanism to locate the correct method for each event.

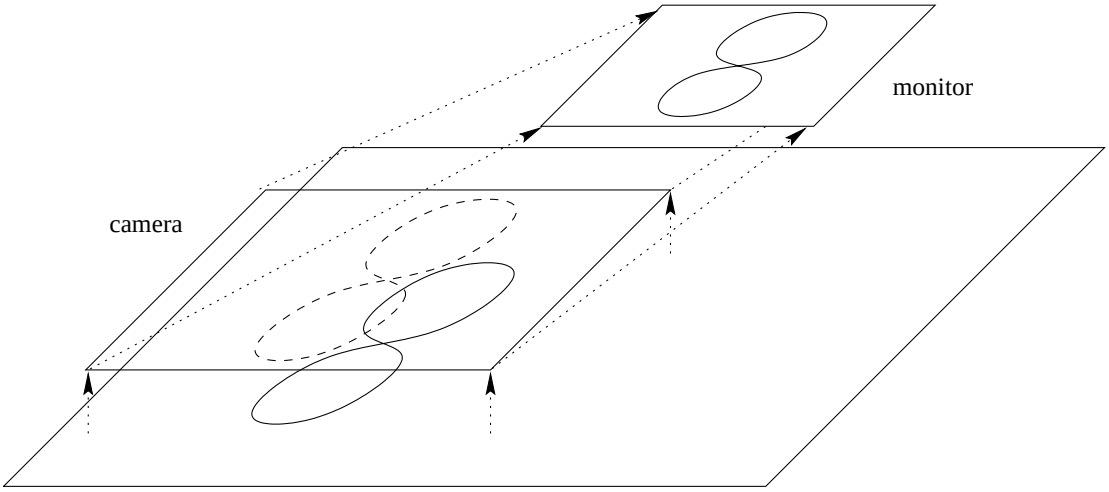
The third and most unusual object-oriented feature used in the Tab implementation is delegation. This is a form of inheritance which is per-instance rather than per-class. Few object oriented languages implement delegation – there is actually no built-in support for delegation in CLOS or STklos, our implementation language. However, we will see in section 3.15.1 how delegation can be implemented in STklos. It allows us to implement a macro similar to `define-class` named `define-filtered`, in honor of its central role in implementing the Tab filtering mechanism. An instance of a class created using `define-filtered` is constructed from an *existing instance* of the parent class, along with the usual constructor arguments. References to the slots the new instance inherited from the parent class are actually references to the slots of the parent *instance*. This corresponds to the relationship between an object and that object as seen through a filter.

As mentioned above, the filtering mechanism seems to hold some promise of restoring the ability to compose tools lost in the move from the Unix command line environment to the desktop environment. This mechanism should enable us, for example, to place a "find the adjectives" filter on top of a document, and a "count the words" filter on top of that to display the number of adjectives in our document. The same "count the words" filter should, if placed directly on the document, display the total word count. This can be accomplished by having the adjective filter derive a filtered class from the text object which adds slots and to and overrides methods of the original class so that it behaves as a document which contained only the adjectives of the original document.

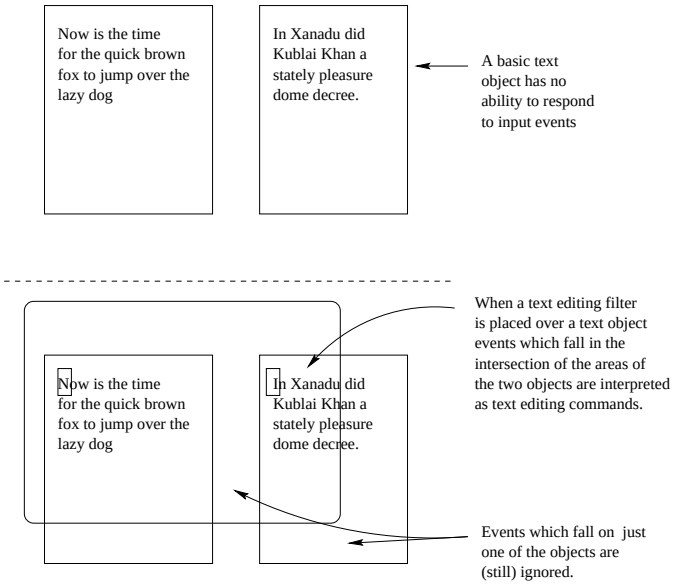
The filtered object is created the first time the system requests it – that is, the first time an event hits the filter responsible for creating it. As shown in figure 1.4, filter 1 might create filtered object 1 from the text object, while filter 2 might create filtered object 2 from the still-exposed portion of the text object. Furthermore, if filter 2 happened to understand filtered object 1 it would then create from it filtered object 3. The filter's decision of whether to create the filtered object is made via another use of multi-methods - if there is a `get-filtered-object` method for that particular combination of filter and original object the return value is the filtered object.

1.7 Summary

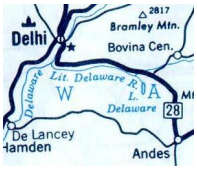
This introductory chapter has introduced Tab in the context of existing user interface metaphors and similes. The basic elements of the Tab system have been discussed at some length, including the virtual surface, portals and filters, semantic zooming, and constraints. It is important to stress that the value comes from combining all these techniques in a single system, just as the combination of windows, icons, pointers and menus let to the success of the desktop interface style.



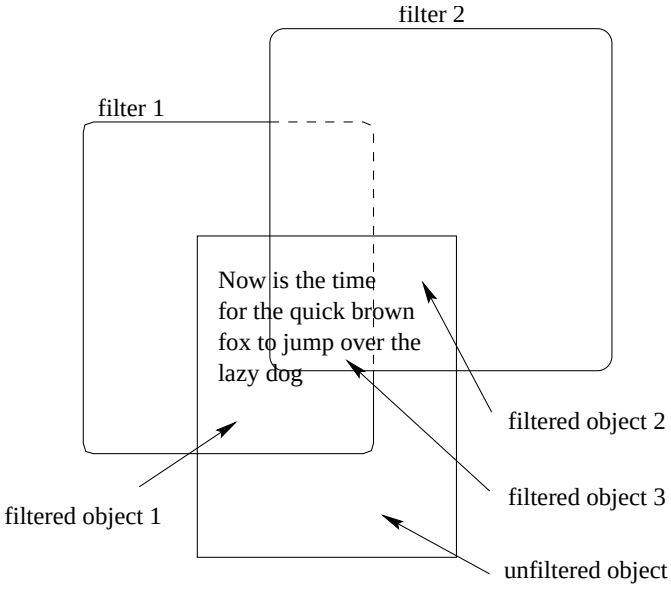
A portal is a camera/monitor pair. Objects visible below the camera appear on the surface of the monitor, scaled to fit. Furthermore, events which hit the surface of the monitor are transmitted to the objects below the camera.



A text editing filter is placed over two text objects.



Semantic Zooming in Cartography



Composed filters can create multiple filtered objects.

Chapter 2

Review of Literature

This chapter reviews the literature relating to each of the topics identified in the introduction as being central to the design of the Tab system. The earliest known examples of real time interative computer graphics include the SAGE Air Defence System (1955) [11], which allowed the user to point at targets with a light pen, and the DAC-1 (Design Augmented by Computers) a computer drawing system created by General Motors and IBM which did real time 3-D rotation of a description of an automobile. It was finally unveiled at the Joint Computer Conference in Detroit in 1964. Apart from these secret projects, interactive computer graphics a scientific discipline began in 1962 with Ivan Sutherland's Sketchpad system [39]. With his doctoral dissertation, the notions of direct manipulation, display lists and instancing, and constraints were introduced. In 1968 Douglas Englebart, then of SRI, showed a user interface system at the Joint Computer Conference which

included a keyboard, keypad, mouse, and windows, and demonstrated a word processor, a hypertext system, and a system for remote collaborative work. This is the first user interface system based on direct manipulation. By "user interface system" we mean the system which mediates all the user's interactions with the computer, as opposed to an interactive computer graphics program. In 1972 work began on the Xerox Alto, a computer system intended for research which incorporated a bit-mapped raster display, three button mouse, and Ethernet network connectivity. Its graphical user interface (GUI) was implemented using the Smalltalk object-oriented programming language. This machine evolved into the desktop style of interface which is prevalent today - examples include the Xerox Star (the first commercial system, introduced in 1981), the Apple Macintosh (the first commercial success), Microsoft's Windows (the biggest success), the X Window system [35], and others.

2.1 User Interface Metaphors

In 1984, just after the introduction of the Macintosh, Andries van Dam was asked in an interview in Communications of the ACM what major improvements still need to be made to bring the user-computer interface to maturity [42]. His answer, in part:

> "[...] We also need faster image dynamics on the screen – it is still

> too slow right now. [...] On the software front, the applications aren't really integrated yet; you've got to switch between them. Also, not all data can be transferred from one program into another."

Since then, phenomenal progress has been made in the area of computer rendering and imaging. In fact, progress in rendering has been so great that it is mostly beyond the scope of this dissertation. Most of the rendering techniques used in the Tab system are simply ways of harnessing capabilities already available in the underlying software and hardware, such as shared memory, compiler in-lining and loop unrolling, font rendering technologies, polygon filling algorithms and so on. On the other hand, it is fair to say that relatively little progress has been made in being able to transfer data from one program into another. It is for this reason that I will concentrate on the qualities of the Tab interface that will allow applications and users to cooperate and better share data – i.e., the features discussed in the previous chapter. Taken together, these features constitute the "metaphor" of the Tab system. This section covers the topic of user interface metaphors in general, while the following sections will cover the components of Tab's metaphors, which were defined in the first chapter.

Considerable care must be taken to create a successful virtual world. While simplicity is a virtue, creating the illusion of simplicity can involve a great deal of complexity. Furthermore, a simple model can seem appealing at first, but inadequate later. As an example, some automobiles have an Automatic Braking

System (ABS) the objective of which is to prevent the brakes from locking, while keeping the car going straight and slowing it down. Sometimes this works well, but on a slippery surface the driver may wish to simultaneously slow down and make a turn. This possibility is outside the user interface metaphor of the of the ABS as described above, and the author has found himself on certain occasions sliding straight out into the middle of an intersection instead of turning left. An adequate ABS system may need to take the position of the steering wheel into consideration. To paraphrase Einstein, a user interface metaphor should be as simple as possible, but no simpler.

By the same token, the spatial "metaphor" used in Pad is quite compelling on its own, but in designing an actual system other elements must be added to the system – portals, filters, semantic zooming, constraint solving. These solve important problems which arise in actual use, and without which the system is just a toy. It is impossible to evaluate a new style of user interface if the implementation is not complete, so creating a radically new system becomes a chicken and egg problem. You want to evaluate your system for guidance as it is being developed, but meaningful evaluations can't be performed until it is complete. This may be the reason that comparisons between the spatial and textual metaphor such as [18] which show that the spatial metaphor is somewhat inferior are so at odds with what we see in the real world, where textual interfaces are in danger of extinction.

2.2 Virtual Surfaces in User Interface Systems

The Pad style of virtual surface, which is adopted by Tab, is one which is capable of representing essentially limitless detail, and which allows the viewer to observe the surface at any location from any distance. However, the restriction is made that the view direction be always perpendicular to the surface, and the viewer cannot rotate.

This means that the Pad virtual surface is a special case of a general three dimensional rendering system, which makes all such systems the ancestors of Pad and Tab. The first such system was created in 1959 by General Motors and IBM. The DAC-1 (Design Augmented by Computers) converted a 3-D description of an automobile to an image which could be viewed from different directions. [27] The algorithms used in Tab to implement the virtual surface are essentially the same as those used in any 3-D computer graphics program, chosen with an eye towards real-time performance. The Tab system also allows gradual refinement of the image, so a variety of algorithms with different time/quality trade-offs can be selected. Such algorithms can be found in any computer graphics or image processing text book, such as [13].

Pad is about creating a user interface system which hides the pixel nature of the user's display and creates a virtual space which embodies a continuous coordinate system. This has not been the practice up until now because of the

computational costs involved in doing so. All drawing operations have an additional level of abstraction which requires the transformation of coordinates in the virtual space to the physical screen space. Rendering images into a virtual space requires re-sampling and reconstruction in the physical coordinate system. Other rendering operations such as drawing text require even more complex types of transformations. In the past these transformations have been the responsibility of the particular application which needed it. To impose these costs on every application which ran on the system would have imposed an intolerable computational burden.

2.2.1 SDMS

In *Spatial Data-Management* [7] Richard Bolt describes a system that incorporates a scalable virtual surface. He calls the principle of using spatial cuing as an aid to memory the "Simonides Effect", named for the ancient Greek poet famous for his technique of memorizing long recitations by assigning each topic a specific location in the floor plan of an imaginary temple, then recalling the entire speech by mentally walking through the temple.

The description of the Spatial Data Management System (SDMS) includes a large number of advanced techniques: navigation over a large two dimensional workspace, screens giving an overview of the workspace (portals), directional audio cues, applications that activate as you approach them (semantic zooming),

gestures for controlling audio volume and for turning document pages, hierarchical traversal of a book's contents, and live video in a zoomable window. However, this is primarily an "idea" paper, and there is little discussion of how these features might be integrated into the user interface of a a personal computer system.

2.2.2 InterViews

InterViews is a system by Mark Linton [9] which is based on a floating point coordinate system. In [26] Linton discusses a number of issues which are important when implementing a resolution independent layer on top of a resolution-dependent graphics system such as X Windows. One such issue concerns whether to apply the current drawing transform to an object's size or its position. Transforming the size of two adjacent objects can cause gaps between nearly adjacent objects to appear and disappear, a very disturbing effect in an animated display. It is less disturbing to convert positions and allow the objects to vary in size by a pixel. Of course, the best route would be to anti-alias the objects, but as we discuss in section 2.6.1 we need to implement algorithms that cover the whole range of speed/quality combinations.

On the other hand, for some display elements it is more important to maintain a constant size than to maintain a constant relative position. The thickness of a frame around an object might be one example. In InterViews an object desiring a particular size examines its position in the pixel grid and computes the position

it needs to pass to achieve a particular size. This technique is easily adapted to Tabula Rasa because the current drawing transformation always takes coordinates back to screen pixel coordinates rather than to some physical coordinate system. This is because an object's size in pixels is usually the best measure of visible detail, rather than the physical size of an object on the screen, because the screen size is often not available to the software, and the viewer can always change distance resulting in a change of apparent size. This bias is partly the result of Tab being a less printer-oriented system than InterViews, which is designed so that any object can generate printable Postscript output.

One of Linton's more significant conclusions for our purposes is that the inherant cost of doing resolution independent rendering is about 45% over that of pixel-based rendering. Given the rapid advances in computing technology, this is quite acceptable.

2.2.3 Pad++

In [3] Bederson lists the following techniques used to maintain efficiency in a zoomable interface (quoting):

- **Spatial Indexing** – Techniques to speed up culling of off-screen objects
- **Spatial Level of Detail** – Avoid rendering detail which is too small to see (below the Nyquist limit[1])

[1] [13], p. 627

- **Clipping** – Avoid rendering any invisible portion of an object
- **Refinement** – Do low quality rendering initially, then high quality once the image is stable.
- **Adaptive Render Scheduling** – Keep the zooming rate constant even as the frame rate changes.

In addition to these issues, some primitive rendering operations must be adapted to a resolution independent context, particularly images and text. Low quality techniques for image scaling such as those used in computer games software are necessary to maintain interactive speed on conventional hardware. One such game development system which inspired the Tab fast image rendering algorithm (later incorporated into the Pad++ system) was the WT ("What's That?") system. [25] This system does simple orthogonal projection 3-D rendering, which means there is no vertical perspective distortion of the images. In an appendix we will see how these techniques and others are adapted for use in Tab.

2.3 Portals and Work-Through Interfaces

> "When the user is building a trail, he names it, inserts the name in his code book, and taps it out on his keyboard. Before him are the two items to be joined, projected onto adjacent viewing positions. At the bottom of each there are a number of blank code spaces, and a

pointer is set to indicate one of these on each item. The user taps a single key, and the items are permanently joined..." – *Vannevar Bush - As We May Think - The Atlantic Monthly, July 1945* [8]

One of the purposes of portals is to serve as the visual equivalent of a hyperlink, and as such inherits from Ted Nelson's ideas about hypertext in [30]. When they are constrained to look at the same place they are located, they become magnifying glasses. If they look at the place they are located without changing the scale, they can be considered filters. These are only interesting if they somehow change the thing being viewed.

2.3.1 Toolglass and Magic Lenses

In "Toolglass and Magic Lenses: The See-Through Interface" [5] Bier et. al. describe a system for implementing filters which is quite similar to that offered by Pad. They emphasize the desirability of using filters as a two-handed interface, where the non-dominant hand positions the filter and the dominant hand performs work through it, much as one might work with a ruler or French curve. The paper contains an interesting collection of design ideas for work-through interface tools, including both visual and interactive tools.

Three approaches are described for implementing filters. The first, *Recursive Ambush*, obtains from each object a procedural description of itself (e.g. a Postscript program) and interprets that description using modified primitives. For

example, the DrawLine primitive might be modified to always draw red lines. The authors mention three disadvantages of this approach: the need to re-implement numerous graphics primitives to implement a given filter is onerous; the results of composition can be mystifying, and the performance of composed filters deteriorates rapidly because the entire computation must be re-done for each pair of filters that see each other – and because they are placed by hand each filter usually has at least a peek at each of the filters below it, causing a doubling of computation time for each added filter.

The *Model-In Model-Out* (MIMO) approach creates a modified copy of the object, or a new object of a different type based on the original object (the line between these is not entirely clear.) This modified model is what the user sees and interacts with through the filter. This approach trades storage space for a substantial performance advantage. It also frees the filter from the domain of the graphics language – filters can be designed to operate in the application domain. Other advantages mentioned include performance and relative ease of authoring and debugging. Disadvantages include very high storage requirements due to making complete copies of all the filtered objects, and problems associated with propagating changes made to the model back to the original object. (This last issue does not appear to have been addressed.)

The third approach is called *Re-parameterize and Clip*, where instead of performing the rendering itself, the filter alters the parameters and clip area of a

renderer and thus directs its output. The performance of this approach is similar to the MIMO approach.

The advantages and deficiencies of these approaches can be combined to point towards the ideals we seek in the implementation of a filtering mechanism for Tab. To allow composition without an undue performance penalty we need to maintain state information about the objects being filtered. The mechanism should not be limited to the domain of graphics primitives; it should operate in the domain of the methods of the objects which are being modeled. We want to avoid the excessive storage requirements of the MIMO approach and we want to operate on the original object, not a copy. And finally, we want filter composition to happen in a natural way, without the need to consider the implications of each possible filter combination. In section 3.15 we will see how the use of delegation-style inheritance greatly facilitates many of these goals.

2.3.2 Delegation

Delegation style object inheritance is not present in the languages commonly used for user interface system development, such as C++, Java, and even CLOS. Delegation is discussed by Brooks Conner in [10], who considers it a technique which is particularly important to interactive graphics programming. He notes that its absence in any commonly used languages has led to unnecessarily complexity in the design of user interface development systems such as Xt, Garnet

and many others. One language that implements delegation is Self [41] [37], but unfortunately development of Self has been discontinued and it is not available for common platforms.

2.4 Semantic Zooming

In describing Pad as a "resolution independent", one might think that matters concerning the size of an object on the screen are hidden from the application programmer, but actually these issues are re-cast at a different level of abstraction. Instead of being told what the size of the drawing area is in pixels, the application always draws in the same area, and has *access* to the scaling factor that converts the drawing coordinates to pixels. Techniques that use this information are termed "semantic zooming", and this term covers many existing techniques that place a layer of abstraction over the task of drawing into a pixel grid. The aim of semantic zooming is to maximize the amount of useful information on the screen while minimizing the amount of useless information.

The most prominent ancestor of Semantic zooming is anti-aliasing, a technique where the rendering of an image is optimized for a particular pixel resolution. With anti-aliasing, the optimization criteria is to eliminate artifacts such as jaggies or Moire patterns, which are not present in the original image or model being rendered and thus constitute useless information.

The Tree-map, presented in [17] and [40], is a display technique for hierarchical information in which a rectangular area is first allocated to hold the representation of the tree, and this area is then subdivided into a set of rectangles that represent the top level of the tree. This process continues recursively on the resulting rectangles to represent each lower level of the tree, each level alternating between vertical and horizontal subdivision. The stopping point for the recursive subdivision must be based on the visual size, so as you examine different subtrees, semantic zooming occurs as the depth of the traversal changes.

There are other examples of systems, either real or envisioned, that embody characteristics of semantic zooming. In Computer Lib ([30]) Nelson describes "Stretch-text", a form of text document which you could ask for longer or shorter editions, and when you did the system would produce a document of the requested size which covered the same subject matter in greater or lesser detail. In Filmfinder [1] described by Ahlberg and Schneiderman the full title of a film appears when the user gets close enough to the surface that fewer than 25 films are on the screen. Other examples of systems which adapt the displayed information to the scale of the view include some such as [29], which mention the notion of Semantic Zooming explicitly.

2.5 Visual and Interaction Conventions

2.5.1 Event Processing

Some systems, such as the X window system, simply present the event to the object and allow it to handle it as it wishes. The event is passed as a C structure, and information such as the type of event (KeyPress, MouseMotion, etc.), the mouse cursor position, the object which received the event, the text string associated with the key that was pressed (e.g. `"Backspace"`) is available by examining that structure.

The Tk system provides a grammar whereby sets of events can be described with a text string, and then allows you to "bind" actions to the set of events described by such a string. For example, `"<KeyPress-a>"` (the letter "a"), `"<Control-d>"` (the letter "d" with the control key down), `"<Any-KeyPress>"`, `"<Double-Button-1>"` (double click of mouse button one), etc. Once an event has matched and the binding is invoked, other information about that event can be inserted into the action specification using special character sequences which the system performs substitutions on: `%x` gives the X coordinate of the mouse, and so on.

This approach is clearly better than the X windows approach from the standpoint of the application programmer. However, if we examine the tasks that are being performed we can see that the Tk system seems to be an ad-hoc solution

to the problem. First, we are partitioning the set of all events into a number of categories, where each event in a particular category is dispatched to a different handler. The expressiveness of the Tk event category syntax is fairly limited here; for example there is no way to describe the set of all digit key press events. The system is also doing some rudimentary parsing on the event stream when it recognizes a double-click event. The question is, could a more formal approach to the specification of event sets and the grammar of the event stream produce a superior system?

This is a dangerous question for a system implementor to ask. The beauty of the Tk event description system is that the sets of events that it can describe are (most of) those that have been shown to be useful to the application programmer. While providing more would increase the power of the system in the abstract, the resulting system might increase the effort required to describe those original sets which are so useful. In fact, the added expressiveness would obscure some useful information which is available to the Tk application programmer – that these are the sort of events that are generally preferred by users.

The Tab system adopts an event dispatching scheme which assigns to each event a list of symbols which range from most to least specifically descriptive. The application programmer can define methods the names of which are any of these symbols, and the most specific method will be invoked.

2.6 Maintaining System Responsiveness

Due to the dynamic nature of the user's interaction with the system, A Pad-style user interface system needs be able to do real-time full-screen redrawing. This section discusses techniques employed in the Tab system to achieve this on conventional computer hardware.

2.6.1 Gradual Refinement and Adaptive Render Scheduling

Two important techniques used in Bederson and Hollan's Pad++ system ([3], [4]) are gradual refinement and adaptive render scheduling. Gradual refinement allows the system to use rougher but faster rendering techniques while the image is in motion. Due to the short time these frames are visible, there is less need for high-quality rendering - there is no time for the user to examine them.

Adaptive render scheduling is an interaction technique which computes the scale factors used while zooming on the basis of elapsed wall-clock time rather than on any fixed per-frame factor. If the system frame rate is reduced, the size of the zoom steps is increased to compensate, so the expected interactive characteristics of the interface are kept as consistent as possible.

2.6.2 Asynchrony

A third technique which will be implemented in future versions of Tab is asynchrony or multi-threading. This brings the multi-tasking familiar from operating system design to the application level. Many opportunities for parallel execution are present in the Tab system, for example, using separate threads for reading and writing objects and other time consuming peripheral activities. This technique may be most familiar from its use in the Netscape web browser. Another use is to decouple event processing, which should occur as the events arrive, from screen update, which need not exceed the rate of about thirty frames per second. Even more sophisticated alternatives to the event loop are described in Matthew Fuchs' paper [14].

Chapter 3

Tab Low Level Design

This chapter discusses the design decisions which led to the current implementation of Tab. This current implementation was preceded by a large number of others, including implementations of Pad by Ken Perlin [33] and Matthew Fuchs. It is also informed by the past and current design decisions made in Ben Bederson's Pad++. Before starting a discussion of the internal design of Tab, it is important to consider the tools which have been used to build it, as these have had a profound effect on the final result.

3.1 Tab Externals

The first decision one makes in designing a new system is often what hardware and software environment the system will inhabit. Many considerations go into

this decision, and not all of which are technical:

- What features the environment provides,
- what platforms the resulting system will run on,
- the stability of the environment,
- how familiar, proven and widely accepted the environment is,
- the cost of the environment software,
- the cost of licensing the environment's run-time system to end users,
- how likely the environment's vendor is to survive

and so on. Often, the importance of the environment's features are downplayed relative to other factors, either by the lack of evidence showing the benefits, or even by citing the Turing equivalence of all programming languages. As an academic system we felt that the it was important to take advantage of any benefits the programming environment could provide in the design of the Tab system, even at the cost of basing our work on systems with a shorter track record and less support.

The earliest versions of Pad were implemented for single plane, monochrome Sun workstations, using the Sun Pixmap library. Due to the lack of processing power all scaling was done by powers of two; no attempt was made to do smooth zooming. This early version was also implemented entirely in C, with no provision for any interpreted extension language. As time went by, the amount of computing

power and the sophistication of the video subsystem of the typical workstation continued to increase rapidly, so later designs began to support pseudo-color and full color imaging models, and it became possible to do real time zooming of the tab surface, including images. Of course, it will always be possible to devise a Tab application of sufficient complexity as to make smooth zooming impossible, so the *refinement* mechanism describe in section 3.11 was provided. At the same time as these advances in hardware technology were occurring, other changes were taking place in the world of Unix software. The decision was made to base future versions on the X Window system, and to implement the low level code in C++ rather than C.

Later, it became clear that an interpreted extension language was necessary to enable the creation of domain specific Pad applications. One early attempt used the Tcl [32] programming language, but the lack of sophisticated programming language constructs soon led to the adoption of Scheme as the Tabula Rasa extension language. At this time development of Pad++ began, and Bederson decided to rewrite the system from scratch. A low level substrate was written in C++, and the Tcl programming language was adopted for an extension language. In addition, the substrate was designed as a Tk widget. This allowed the embedding of Pad in a user interface which could use the Tk tools (though the Tk tools only existed outside of the Pad space) and eased the development process by providing pre-built widgets for controlling the Pad++ system. Since that time

other extension languages have been added to Pad++, including Scheme.

In the meantime, Scheme was adopted as the extension language for the Tab system. Although the Tcl language has been found to be quite acceptable to end users as a scripting language, it was judged that a more powerful language was necessary to implement a flexible general purpose user interface construction environment. [14] A Scheme interpreter named STk [15] was chosen because it included an interface to the Tk toolkit, and has an object oriented programming system which supported both multiple inheritance and multi-methods. Though use of the Tk toolkit was dropped soon after, the power of the object system remains a driving force in Tab's design.

As development continued, features tended to creep out of the C++ layer up into the Scheme layer. This was due both to the ease of development in an interpreted environment, and to the appeal of the more elegant, general and powerful implementations which resulted when coding in object oriented Scheme. Unfortunately, this also led to a worsening of the system's performance. After some time spent attempting to re-partition the Scheme/C++ split to preserve both speed and flexibility, the STk interpreted-only Scheme environment was abandoned in favor of Joel Bartlett's `Scheme->C` Scheme compiler. [2]

3.1.1 The Taboo Object System

The first task in the transition from STk to `Scheme->C` was to extract the object system from STk and make it portable. The intention was to keep the code from becoming dependent on any specific compiler, and this goal has been met with mixed success, for reasons which will be described below. The STk object system, STklos, was based on TinyCLOS [20], a simplified version of the Common Lisp Object System [21] written entirely in Scheme. The portions of STklos which had been translated into C for speed were rewritten in Scheme and the resulting code was made to work under two Scheme compilers: `Scheme->C` and Marc Feeley's Gambit Scheme System [12]. The resulting subsystem has been named *Taboo*, for "TAB Object Oriented programming system."

These two Scheme compilers were chosen because they provide several features which are not part of the Scheme standard, but which turn out to be important in implementing Tab. The first two of these non-standard features in the list below were necessary for implementing the object system itself, while the remainder are needed to implement other parts of the Tab system.

- CLOS style keywords,
- access to the contents of a closure,
- loading a startup file for customization,
- ability to compile under both Unix and Microsoft Windows,

- object finalization,
- exception handling,
- a suitable foreign function interface

Currently, Tab can be compiled by a version of `Scheme->C` which has been modified by the author to implement the first three features in the list above. It can also be compiled by Gambit, which has all these features except for exception handling. The Gambit version works well most of the time. Currently, all the other compilers which are available to us lack one or more of the required features.

After the object system was ported to `Scheme->C` and a compiled version of Tab was built, it was found that the expected speedup did not occur. Investigation revealed that the object system's mechanism for accessing slots was consuming the majority of the system's time. This was especially unfortunate because the mechanism for dispatching a generic function call itself made several slot references. The problem turned out to be the data structure used to implement multiple inheritance in TinyCLOS. In a single inheritance object system it is possible to store all the slots of an object in a single vector, and each slot will always have a fixed offset in that vector which is known at compile time. Thus, each slot access becomes a single vector reference. However, in a multiple inheritance system this is not possible, because the graph induced on the classes by the superclass relation is no longer a linear list. Even a class with no super classes can't assume its first

slot is at offset zero, because it could still appear anywhere in inheritance tree of a given instance.

TinyCLOS uses the following data structure to store an instance's slots: it traverses the tree of super classes and makes a list of slot names. It then stores an association list in the class object containing each slot name and the slot's offset. It then allocates a vector in the instance whose length equals the number of slots. To access a given slot, the association list is searched for the slot name, and then the resulting index is used on the instance's slot vector. The system was spending most of its time traversing slot name association lists.

One way to improve the slot access performance is described by Kelley Murray in [28]. It involves generating a perfect hashing function for all the slot names in the program and then accessing a two dimensional array of slot offsets using that and an index number assigned to the class. This method's main drawback is the size of the array that might be needed for a large system. A variant of this has been implemented for Taboo where instead of using perfect hashing functions an integer sequence number is assigned to each slot name as the program starts up, and all references to slots by symbol constants are converted to array references. A similar technique is also used to avoid looking up classes by name, which speeds method dispatch.

This improvement to the object system has brought the hoped-for speed-up. Although method dispatch is still fairly slow, the ability to access slots quickly

allows the hand optimization of inner loops, at least until further improvements are made to the object system implementation.

3.1.2 Portability

As noted in the table above, the choice of compilers was made with an eye towards portability to Microsoft Windows. It is expected that the code which the Tk library uses to map X library calls to Microsoft Windows equivalents can be used to port the underlying C++ code, thus rendering the entire system portable.

3.2 Tab Internals

The remainder of this chapter discusses the low level Scheme layer of the Tab implementation. This layer is primarily concerned with implementing the geometry of Pad space. The central element of this is the update algorithm – the procedure whereby the user's input is ultimately translated into changes on the user's screen.

The challenge in designing the system is to efficiently implement a scalable virtual surface while providing a elegant application programmer interface in an interpreted language for the developer of Pad applications. The application designer is concerned with the domain of the application being designed, and should be shielded as much as possible from issues which lie outside that domain. This means the API should embody decisions concerning the appearance of menus, the

implementation of the event handler for dragging and zooming objects, and so on. In cases where the designer can't be completely shielded from such decisions, a set of alternatives should be provided each of which has been shown to have merit in practice – several types of borders, several styles of line drawing, etc. Default values for such choices should be carefully selected by the API designer. Mechanisms for making deeper alterations to the normal look and feel should be provided, but they should be transparent to the designer who doesn't need to use them.

Creating an efficient system while addressing these concerns is the primary goal of Pad system. Though "separating mechanism and policy"[1] is a valid goal, the mechanism must be designed in such a way that policy alternatives are not eliminated, while at the same time the efficiency gained by "compiling in" certain behavior is not lost. This chapter is primarily concerned with the mechanism, but that mechanism has evolved to its present state because of the need to balance these concerns. One former design was implemented entirely in C, while another was implemented primarily in Scheme with only calls to the rendering primitives implemented in C++. Neither of these designs were acceptable, and they led to a more recent design, where the update mechanism (event distribution, damage propagation and damage repair) is implemented in C++, with Scheme hooks for the application developer attached. Unfortunately, this design became unworkable when more sophisticated uses of semantic zooming were attempted. The

[1] See [35], p. 7

update mechanism was not flexible enough to allow objects to decide procedurally whether to be visible or invisible, opaque or transparent. This led to the current design, which is again primarily in Scheme, with certain key rendering algorithms implemented in C++. The issue of execution speed is addressed by careful attention to algorithms and adoption of improved compiler technology.[2] For the didactic purposes of this implementation the greater expressiveness of a higher level language is the most important issue.

We begin the description by defining the lowest level objects that inhabit Tab space, from which all others are derived. There follows a description of how these objects interact in the course of processing user input and updating the screen. We then gradually add more complex objects such as portals and filters. After that we proceed to a discussion of sub-topics of the update cycle – how rendering primitives are implemented, how actions are bound to events, and how gradual refinement of rendering quality is implemented. A discussion of the use of asynchrony is also included, though the implementation is incomplete due to lack of widespread support for threads in current operating system libraries.

3.3 Tablet

As shown in figure 3.1, the root of the tab inheritance tree is the *tablet* class. This class encapsulates the minimum functionality necessary for an object to exist in

[2] And, of course, improved hardware technology.

Tab space. Each tablet has three basic attributes: A *shape*, a *transform*, and an *element*.

The *shape* of a tablet is the portion of the Tab surface that it occupies. The data structure of the shape is described in [13] (pp. 992-996), and has been chosen because it is easy to compute the unions, intersections and differences required in the algorithms described below. It is also supported by most window systems once the coordinates are converted to integers (*exactified*.)

A tablet's *transform* is the transformation from the *public coordinate system* to its *private coordinate system*. Giving each tablet a private coordinate system allows it to operate in a coordinate system which is natural to the application domain. In the current implementation the transform is not a general linear transformation, only scaling and translation are permitted. This avoids some difficulty in implementing graphics operations which are not supported by the underlying window system and for which we have no efficient replacement, such as rotated shapes and images.

The *element* of a tablet is a pointer to the container object that represents its stacking order on the surface. This can be used to locate adjacent tablets in the stacking order in constant time. This is an important operation for the system, which frequently needs to traverse the tablet stack.

3.4 Viewer

The second basic object that figures into the initial description of the update algorithm is the *viewer*. This object represents the user's input and output devices. Because the input/output devices are represented by a type called *window* in the underlying Zoom graphics library, the viewer object has the window class as one of its super classes. The viewer's private coordinate system is the pixel coordinates of the screen or window, and the image it displays there is of whatever lies inside the viewer's shape and below it in the stacking order.

Note that the viewer's private coordinate system has an inverted Y coordinate for the window systems we have encountered so far, with Y increasing towards the bottom of the screen, while in Tab the Y coordinate increases upwards. This means that care must be taken when transforming shapes to insure that the Y spans remain in increasing order and the heights remain positive.

There can be several viewers in the Tab system, each displaying a different portion of the Tab space and each displaying themselves in a different window. These viewers could even be connected to windows on different user's screens, providing a primitive sort of shared/collaborative system. More sophisticated types of multi-user Tab systems could be constructed by implementing viewers which used other protocols besides X-windows.

The basic *update cycle* is activated when a portion of a viewer becomes dam-

aged and needs to be redrawn. *Damage* occurs in several situations: when the viewer is first created, when a portion of the window becomes exposed, or when something in Tab space transmits damage to the viewer. If the damage originates in Tab space it is transmitted upwards through the stack of tablets. If it encounters a tablet, the shape of that tablet is subtracted from the damage shape and the remaining damage continues upwards. If and when the damage hits a viewer the intersection of the damage and viewer shapes is converted to the viewer's private coordinates (i.e. pixel coordinates) and added to the viewer's damage shape.

The emphasis here is on updating the smallest possible amount of screen space, because the act of updating can trigger expensive operations. As we will see in the next chapter, one way of preventing invisibly small detail from being rendered is by putting an object in front of it which is opaque at certain scales. If the system doesn't carefully avoid unnecessary rendering this technique will fail.

The viewer contains two stacks which it uses during all phases of the update cycle: a *clip stack* and a *transform stack*. At any time the top element of the transform stack will take the private coordinates of the tablet which is being operated on to pixel coordinates. Similarly, the top element of the clip stack tells us what part of the screen coordinate system the current tablet affects. These stacks will see heavier use once the *group* type is introduced, but they are presented here because we now need access to their top element.

At some point when damage has accumulated it is time to *render* the damaged

portion of the viewer. This might be at the end of an event handling routine, or if the Tab implementation is asynchronous (implemented using multiple threads of execution) it might be after a specific amount of time has passed. To render the damaged portion of a viewer, the viewer's transform is first pushed onto its own transform stack, and the damage shape is pushed on the viewer's clipping stack. Now each tablet below the viewer is examined in turn. The top element of the viewer's transform stack is applied to each tablet's shape to convert it to the coordinate system of the shape on top of the clip stack. the tablet's render method is invoked to redraw any intersection between the two shapes. After that, the tablet's shape is subtracted from the top element of the clip stack and the procedure is repeated for the next lower tablet.

For each tablet, the intersection of the tablet shape and the shape on top of the damage stack is computed. If this intersection is non-empty the current clip boundary is set to this shape transformed back into viewer coordinates. Next the composition of the tablet's transform and the top element of the transform stack is computed and pushed onto the transform stack. Then the tablet's render method is called. Afterwards, the area just rendered is subtracted from the damage region and the system continues to the next tablet down.

An input **event** is handled in a similar way. Like a rendering request, it originates from a viewer, which represents the user's mouse and keyboard as well as the screen. There are two important differences, however. First, instead

of having a shape an event occurs at a single point. Second, an event is first propagated down through the stack of tablets, and if it is stopped or reaches the bottom without being used it then comes back up, giving the tablets a second chance to intercept the event after it has been rejected by the objects below it. This becomes important once transparent objects and portals are introduced.

3.5 Group

A *group* is a tablet which contains an ordered collection of tablets. Groups define what is referred to above as the stacking order. The transform of the group affects all the tablets inside it – the private coordinate system of the group is the public coordinate system of the tablets it contains. This means that the transform of the group can be used to move and scale the group's elements *en masse*.

The shape of the group is the union of the shapes of its constituent tablets, so events or damage that don't intersect the group's shape can be considered to have missed all the group's elements as well. This means the group can be used to implement spatial indexing algorithms to enhance the system's efficiency. Groups can be nested to implement even more sophisticated schemes.

Although users consider the tablets on the screen to be sitting on a surface, this notion actually breaks down into more primitive elements. A surface is a group of tablets, the bottommost member of which is a **backdrop**. A backdrop

is simply an opaque tablet of unlimited extent.

The element slot of the tablet holds an object which represents the tablet's membership in a group, specifically an instance of the *ordered-set-element* class. As mentioned above this gives us a way to locate the adjacent tablets in the stacking order, as well as giving access to the group itself. Giving the tablet only a pointer to the group and having it search for itself amongst the group's elements proved to be an unacceptable performance penalty.

The presence of groups as the tablet ordering mechanism motivates the existence of the transform stack in the viewer. Whenever an event or a rendering request encounters a group the viewer pushes its transform stack and composes the group's coordinate system with its own. When leaving the group the inverse of its transform is then composed.

Having nested groups also means we need a way of determining whether an event that enters a group has hit anything while it was inside, and whether or not the object it hit used it. This allows the system to decide whether to propagate the event downwards or upwards when it leaves the group. A slot which holds a symbol is provided in the event object for this purpose. This symbol is initially `unused`, and it remains so on its trip down the tablet stack. When it hits an object it becomes either `used` or `stopped`, depending on whether the object has consumed that event.

3.6 Transparent and Never Transparent Objects

It is desirable to be able to create objects which are transparent or semi-transparent, not only visually but also transparent to events. For this we define a predicate `transparent?` on tablets to distinguish these objects. By default, tablets are opaque, but viewers are transparent, they are invisible to other viewers and do not receive or block events.

For opaque objects the rendering proceeds in top to bottom[3] order. First the clipping shape is set to the intersection of the object's shape and the remaining damage shape. The object then draws itself and then subtracts its shape from the remaining damage. This is to ensure that nothing outside of the damage shape is drawn, and nothing is drawn more than once. However, if a transparent object is present we need to use a bottom to top drawing order to render it, so it can overlay itself onto the already rendered image of the objects below it. These two requirements combine to form a total ordering on the set of tablets to be rendered at any given time and scale.

We define transparency using a method rather than a slot to allow for the possibility of one object appearing simultaneously in both an opaque and transparent state. This could happen if its transparency depended on its scale and it was visible both directly and at another scale through a portal.

The one difficulty that arises from objects which are sometimes transparent

[3]Often called front to back.

and sometimes opaque is that although the scale is known when propagating events and doing rendering, it is not known while propagating damage. This is because damage originates from tablets and travels upwards towards the viewer, while events and rendering requests originate at the viewer and travel downwards. The viewing scale of the damaged area is not known until it hits the viewer, so while propagating damage we may not have enough information to decide whether a tablet is transparent. In this case, we must assume that the tablet *is* transparent, and as a result there may be more damage than necessary.

To mitigate this problem, another predicate `never-transparent?` is added to allow tablets to indicate that they can *never* become transparent. This method returns true by default, so unless a subclass of tablet defines this method to return false, the tablet will always obscure damage which encounters it on the way to a viewer, even if the `transparent?` predicate sometimes returns true. Special attention should be paid to the case where a viewer is inside of a group. If the group's `never-transparent?` method is true then damage which occurs below the group will not be propagated to the elements inside the group, as it is quicker to subtract the shape of the group from the damage shape and continue to the tablets above. In this case, the viewer will never see the damage, and will not be updated. It is assumed this situation will not occur.

3.7 Translating Events Into Actions

Previous sections have discussed how events are distributed to objects during the update cycle. Once the event reaches its destination the tablet needs to determine what actions to take in response to that event. There must be a simple and powerful interface for the application programmer to specify the relation between events and actions.

A tablet's event handling behavior is controlled by adding one or more **handler** classes to its list of super classes. This may seem inadequate on first consideration – how do you handle tablets which handle events differently on different parts of its surface, or at different times? This type of task can be accomplished by creating special handlers which contain two or more other handlers, and look at the event's properties to decide which of them to forward the event to. Note that a full implementation of multiple inheritance is needed for this to be practical, as each handler may include state. The "multiple interface" approach used in Java is not sufficient.

Once an event is delivered to a specific handler associated with a tablet, the event's type is converted into a list of symbols, each of which describe the event in more and more general terms. For example, the upper case "Q" keyboard event would yield the list `(shift-key-q any-key)`. Then the object system's `find-method` function is used on each of the symbols on this list in turn, to see

if there is a method by that name which takes this handler type as an argument. If no method is found, or all the methods leave the event unused, the default method simply named `handle` is called. This arrangement allows the application programmer to simply define a method whose name is that of the desired event, with an argument of some type derived from handler, and that method will automatically receive exactly the events requested.

3.8 Portal and Camera

A *portal* is an object which can be linked to any other object. It is usually linked to a *camera*, which is invisible and transparent to events. Whatever is directly below the camera is visible on the portal's surface. Furthermore, events and damage also pass between the camera and portal. (This is similar to the relationship between a viewer and the user's screen.)

Though a portal is not transparent in the sense conveyed by the `transparent?` predicate, it does pass events through to its associated camera. On the downward trip an event will contain `unused` in its state slot, while on the upward trip it will contain either `stopped` or `used`. It is important for transparent objects to distinguish between these cases.

3.9 The Timer Heap

Applications and various other parts of the tab system can use the `set-timer` method to request a timer event be sent at a fixed time in the future. The global variable `timer-heap` is a heap which organizes all the requests for timer events into a single priority queue. The item at the front of this queue is always the timer which is going to lapse soonest. The elements of the queue are instances of class `<timer-event>`, and a `<heap>` structure is used to keep them sorted.

The `(set-timer <real> <list>)` function creates a timer event carrying the info field given in the second argument, set to go off after the number of seconds given in the first argument. When it goes off the generic function `alarm` will be invoked and the objects in the list passed as the second argument will be used as its arguments. Before the objects go into the heap the the current time is added to the number of seconds to determine the specific time the timer will lapse.

When the `next-event` method is called from the main loop, it first checks whether there are any timer events in the priority queue. If there are it uses the next one as the timeout value for getting the event. If no other event occurs before the timeout a `TimerNotify` event will be returned. In this case we remove the timer from the heap and copy its `info` slot into the event.

3.10 Grabbing

The Grab mechanism allows one or more objects to take control of the event stream from the usual dispatching system. This is necessary in situations where the user is dragging an object around the screen. The intent is for the object to follow the motion events of the user. However, if the usual dispatching system is used some of the motion events might take the mouse pointer outside the dragged object's boundary. This event would never be delivered to the object, and the drag would stop. Another situation where grabbing is necessary is when dragging an object across something which is above it in the stacking order. Without grabbing, the system would drop the object being dragged in favor of the object above.

When an object is passed to a viewer's `grab` method, all subsequent events from that viewer are delivered directly to the object until the viewer's `ungrab` method is called. Unlike most window systems, more than one object can grab the input stream. The viewer maintains a list of grabbers, and as long as that list is not empty each event is delivered to each grabber. Allowing multiple grabs is an alternative to the *enter* and *leave* events used by many systems. To illustrate with an example, a pop-up menu consisting of a column of items needs to grab the event stream because it needs to be notified in the event that the user releases the mouse button while the pointer is outside of the menu – in that case the

pop-up is dismissed. Because the menu items are themselves Tab objects, they also need to grab the event stream so they know when the pointer enters and exits their surface and can change their border relief accordingly. The use of multiple grabs allows similar functionality to the enter/leave event model while avoiding the overhead of associating with the generation of enter/leave events.

The viewer's `grab-list` slot holds a list of *(handler, transform)* pairs that are grabbing the event stream. The transform is the one that was in effect when the grab was initiated. This allows the system to re-create the correct environment when sending events to a grab. It is necessary to copy the grab list before the event is sent, in case new grabs are added or old ones deleted during the handling of the event.

3.11 Refinement

Refinement is a technique whereby areas of the screen are first rendered using rougher and less time consuming techniques, and then again using more sophisticated and time consuming algorithms. The purpose of refinement is to provide good response when the user is interacting with the system, and then to provide a higher quality image when the system is at rest.

To achieve this we keep the shapes to be refined in a priority queue, and subtract any newly damaged shapes from the shapes already in the queue. If

a queue element matures without having been reduced to an empty shape, that shape is re-rendered using a higher refinement level. This is implemented by adding three slots to the viewer object: one to hold the current refinement level, one class slot to hold the amount of time to wait between refinement levels, and a priority queue (heap) to hold the refinement events.

Each heap element is an instance of the `refine-event` class, which has three slots: a shape which holds the area of the viewer to be re-rendered, a timestamp which holds the time at which the refinement is to be performed (usually about half a second after the object was created,) and the viewer on which the refinement is to be performed.

The `insert-refine-event` method creates a new refinement event and inserts it into the heap. The first thing it must do is use `cancel-shape` to subtract its own shape from those of the existing elements. In this way refinement of the area covered by this newly damaged shape must be postponed until this new event lapses.

In addition to being inserted into the refine queue, the refine event is also inserted into the main event queue. When the timer event goes off it invokes an `alarm` method which performs the actual re-rendering. This is necessary to cause the system to wake up when it is time to do the refinement. An earlier version of the system only checked the `refine-heap` at the end of each iteration of the update loop, but when the system became idle the refinement queue would never

be checked. In the present system, the only purpose served by having a refine-heap separate from the timer heap is to provide easy access to the list of pending refine events for canceling re-damaged shapes.

When the timer event expires it invokes the `alarm` function, to which it passes the timer event's info field which contains the `refine-event`. This call is dispatched to a method which increases the viewer's refine level and renders the requested shape.

3.12 Saving and Restoring

The design goal for the save/restore mechanism is that it place as little burden on the application programmer as possible. The design requirement is that it write an expression to a file which, when read back into the interpreter, creates a configuration as close as possible to the one that was written, given the conditions at the time the configuration is read back in.

The reason we say "as close as possible" rather than identical is that there may be differences in the program's environment when it is restarted, and some aspects of the running environment may not be suitable for saving. For example, the type of visual might be different when the configuration is loaded. This means that we need to re-quantize the images, so simply writing the image data structures would not be acceptable. Furthermore, writing the image data to the save file would

waste disk space, it might be more appropriate to simply write the image's file name or URL. This suggests that devising a general mechanism which could write any object without any specific knowledge of that object would be impractical.

To implement save/restore for a new class the application programmer only needs to write a **config** method for that object. The config method takes one argument, the original tablet that is to be configured, and it returns a function of one argument, the object that is to be turned into a copy of the original. As an example, here is the config method for the camera class:

```
(define-method config ((orig <camera>))
  `(lambda (copy)
     (,(next-method) copy)
     ,(if (<camera>-portal orig) (refer (<camera>-portal orig)))))
```

The config function looks almost like a standard STklos initialize method, except that

1. the entire function is wrapped in a `(lambda (copy) ....)` so that it returns a function that can be saved and then loaded and applied to the copy, and

2. instead of simply calling (next-method) the expression `(,(next-method) copy)` is used, which expands into code that initializes the super classes, and

3. all references to objects are wrapped in `(refer ...)` which expands into code that looks up that other object in a hash table, so that circular structures and multiple references are handled properly.

3.13 Constraints

The constraint system is implemented in the same way that a tablet's event handling characteristics are selected, by adding handler classes to the tablet's list of super classes. In this case the objects are **constraint** objects, classes which implement one or more of the three "hook" methods, `insert-hook`, `geometry-hook`, and `remove-hook`. The system calls these methods when various attributes of the tablet change. These three events are chosen because they are the primary ways in which the geometry of Tab space can change.

The `geometry-hook` is called whenever a tablet's shape changes, but not when only its transform changes. This is because changing the transform only affects that tablet's appearance, it should not affect any objects outside itself. Therefore the damage mechanism is sufficient to handle such a change. The `insert-hook` is run whenever a tablet is inserted into a group, and the `remove-hook` is run whenever a tablet is removed from a group.

Several constraint classes are provided which implement some basic and useful types of constraints. These can then be combined using multiple inheritance. It

is vital that the programmer ensure that running a hook not cause an infinite recursion in the hook itself. Because hooks are usually simple, no built-in mechanism is provided to prevent this, though a hook method could be implemented in such a way that it checks for and prevents this itself.

Here are a few examples of useful basic constraint objects. The `<no-magnify>` constraint is used to keep the center of a portal's camera at the same position as the center of the portal, and the magnification at one. This gives the portal the appearance of a non-magnifying piece of glass. Note that the constraint object has a slot for the camera, rather than assuming that because it is mixed into the portal class it can retrieve the camera from itself. This is because it might be part of a frame which contains the portal, in which case the camera should stay centered below the frame.

```
(define-class <no-magnify> () ())


(define-method geometry-hook ((self <no-magnify>))
  (set-transform (<portal>-camera self)
                 (<tablet>-shape (<portal>-camera self))
                 (<tablet>-shape self)))
```

The `<magnify-hooks>` class relaxes the "no magnification" constraint so the object can be used to enlarge or reduce:

```
(define-class <magnify> (<no-magnify>) ())


(define-method geometry-hook ((self <magnify>))
  (let ((camera (<no-magnify>-camera self)))
    (set-transform
      (<tablet>-shape camera)
      (make-shape :center (center (<tablet>-shape self))
                  :size (size (<tablet>-private-shape self))))))
```

The `<follow>` class is used to cause a camera to always sit just below its associated portal. When the portal is removed, the camera is removed; when the portal is inserted, the camera is inserted just below it:

```
(define-class <follow> () ())


(define-method insert-hook ((self <follow>))
  (insert-below self (<portal>-camera self)))


(define-method remove-hook ((hooks <follow>))
  (remove (<portal>-camera hooks)))
```

Finally, some common combinations of constraint classes:

```
(define-class <magnifier> (<magnify> <follow>) ())
```

```
(define-class <filter> (<no-magnify> <follow>) ())
```

3.14 Drawing Primitives and Display Lists

The graphics primitives implemented in Tab correspond to those implemented in X windows, but they are all adapted to take arguments which are in floating point coordinates rather than integer. This conversion is mostly straightforward due to the restriction that the transformation is not allowed to include rotation. However, there are a few issues involved in the conversion from floating pointer to integer, and a few cases which are more difficult due to limitations of the underlying graphics system.

Most of the issues are covered by Mark Linton in "Adding Resolution Independence on Top of the X Window System" [26]. For example, to avoid gaps between adjacent objects one should avoid "exactifying" lengths, but instead exactify positions.

The most important cases where the limits of the underlying graphics system are encountered are in the rendering of scaled images and scaled text. The networked model used by the X Windowing system is particularly ill-suited for implementing real-time image scaling due to the assumption of a relatively low bandwidth connection between the client application and the display.

One technique used to speed image scaling is shared memory. Specifically,

the MIT shared memory extension is used whenever the display is on the same machine that Tab is running. This allows direct access to the memory where the screen image is constructed. If this is not present it is necessary to maintain a synchronized copy of the screen image on both the client and server machines, so that image scaling can occur on the client side while the built-in X rendering operations can occur in the server.

The algorithm used for image scaling is adopted from gaming software [25] to allow real-time performance for full resolution image scaling. Although each pixel of the scaled image is constructed from a single sample of the source, the technique has been found acceptable for most purposes, particularly for non-refined views. The important speed-ups in the code use

- Loop unrolling,
- Special casing using the compiler's inlining feature. Below we see an example function where the cases 8, 16 and 32 are selected for optimization by passing a constant to an inline function. Assuming that the `do_row_inline` function does a fair amount of work, this will result in improved performance (at the expense of increased code size) due to the constant folding performed by the optimizer on the function argument:

```
void do_row(int bpp) {
  switch(bpp) {
```

```
case 8: do_row_inline(8); break;
case 16: do_row_inline(16); break;
case 32: do_row_inline(32); break;
default: do_row_inline(bpp); break;}}
```

Using these methods the inner loop of the simplest case of non-dithered image scaling is reduced to about six Intel 386 instructions per pixel

- If the image is to be scaled much larger than the original, the destination image is drawn by filling a rectangle for each pixel. This approach uses the window system's rendering primitives to achieve better performance than a pixel-by-pixel approach.

- For pseudo-color displays, a fast color ordered dither technique is employed. Applying the optimizations mentioned above to a dithering algorithm results in about 22 instructions per dithered pixel, about one third of the non-dithered speed.

Two techniques for drawing scaled text are provided. One is a stroke font, which provides good performance but only mediocre quality. The other is code to interpret Adobe Type 1 outline fonts, which provides good quality at larger scales.

A display list data structure is provided to allow the rendering of composite objects without having to execute function calls for each graphics element. The

elements of the display list are rendered in floating point coordinates so they can be scaled and translated like other Tab objects.

3.15 Filtering

As discussed in section 1.3, implementing filtered object behavior requires that we create surrogate objects whose behavior can be overridden method by method. What makes this difficult is that we don't want the surrogate to be an entirely new object, we want it to have as its parent the *original instance* of the object. When we look at or modify a slot in our derived object we want to see or modify the parent instance when the slot is inherited from the parent. Consider our text document/word processor example from section 1.3. When we place the word processor filter over our document it creates a "text document being edited" surrogate object which is derived from the text document object. However, with C++ style inheritance this object would be an entirely new instance; modifying its text would have no effect on the text of the original document. Instead, we want our new object to have the original text document *instance* as its parent. Then the surrogate object created by the filter would, when asked for the fifth word in the text, refer the request to the original *instance* of the text document. When asked for the cursor position it should refer to data stored only in the surrogate.

If we tried to implement the surrogate object in C++ we would have to first create a new object containing a pointer to the old object and the new cursor data. Then for each method implemented by the original object we would have to provide a stub method that invoke that method when it is called in the surrogate. We would also have to dereference any uses of the original object's data members through the pointer to the old object. If the old object was itself a surrogate object we would have to continue recursively. This approach is not suitable for applications programming.

3.15.1 Implementation of Delegation in STklos

Fortunately, the behavior we seek turns out to be easily implemented using CLOS [6], or in this case a variant called STklos [15]. Normally classes are created using a `define-class` macro, but filtered objects are created using a `define-filtered` macro which creates classes with the instance inheritance property we need. The approach is to first create a new class with any new slots we need but with no parent class. The macro adds to these slots a `super` slot to contain a reference to the parent instance. It also creates a list of "virtual slots", one for each slot of the parent class. A virtual slot is one which is not allocated in an instance, but instead procedures are provided to get and set its value. These procedures access the slot in the parent instance. Next, the macro "manually" sets the parent class of our filter by modifying its `direct-supers` list and re-

computing its class precedence list. The ability to do this is a feature of the Meta-Object Protocol[21], in which the filter class itself is an instance of the `class` class, which can be manipulated within the framework of the programming language. Finally, it creates an initialize method which initializes the member variables but bypasses the initialize method of the new superclass, because the super instance is already initialized.

```
(define-macro (define-delegator name super slots)
  `(begin
     (define-class ,name ()
       (,@slots
        (delegate :init-keyword :delegate))
       ,@(map (lambda (slot)
                `(,(car slot)
                  :allocation :virtual
                  :slot-ref (lambda (self)
                              (slot-ref (slot-ref self 'delegate)
                                        ',(car slot)))
                  :slot-set! (lambda (self v)
                               (slot-set! (slot-ref self 'delegate)
                                          ',(car slot) v))))
              (slot-ref (eval super) 'slots)))
```

```
    (slot-set! ,name 'direct-supers (list ,super))
    (slot-set! ,name 'cpl (compute-cpl ,name))
    (define-method initialize ((self ,name) args)
      (%initialize-object self args))))
```

One remaining weakness of this implementation of delegation is that if the `initialize` method is overridden, calls to `next-method` will incorrectly modify the delegate object. Instead of calling `next-method`, the function `%initialize-object`, which initializes the object's direct slots, must be called directly.

3.15.2 Delegation Example

As an example, consider a simple point class which also stores the point's distance to origin:

```
[delegation-example.scm]
(require "delegation")


(define-class <point> ()
  ((x :init-keyword :x :accessor x)
   (y :init-keyword :y :accessor y)))
```

Lets create a class that filters `<point>` which adds a slot to store the distance to the point's origin:

```
[delegation-example.scm]

(define-filtered <3d-point> (<point>)

  ((z :init-keyword :z :accessor z)))


[delegation-example.scm]

(define p (make <point> :x 3. :y 4.))

(define p3 (make <3d-point> :super p :z 5.))
```

After executing the above code, we can look at the description of `dp` and see it does indeed have the slots of `p`. Furthermore, if we change a slot of `p` the corresponding slot of `p3` changes as well:

```
STk> (describe p3)

#[<3d-point> 8044120] is an instance of class <3d-point>

Slots are:

     super = #[<point> 8044b88]

     z = 5.0

     x = 3.0

     y = 4.0

#f

STk> (slot-set! p 'x 1.)

#[undefined]

STk> (slot-ref p3 'x)
```

```
1.0

STk>
```

3.15.3 Implementing Filters

Whenever an event reaches a tablet, or a tablet's render method is called, the first thing that happens is that the **portal-stack**, the stack of portals that were passed through on the way from the viewer to the object, is re-traversed from bottom to top. Each portal is given the opportunity to handle the event or do the rendering. If one of them chooses to do so, it will give the object an alternative behavior or appearance.

This decision is based on the presence or absence of a method to handle the event or perform the rendering which takes an additional portal argument along with the usual viewer, tablet and event. This technique is simpler than delegation based filtering, and can only be used to implement types of filtering which do not involve additional per-object state. For example, you could implement a bar-graph rendering filter for a numerical data object without using delegation, but you could not implement an editor filter because editors need extra state, such as the cursor position.

The class `<filter>` is the subclass of `<portal>` which implements delegation based filtering. It has a `delegate-class` slot, which holds the class of the objects it filters, and a `filtered-class` slot which holds the delegator class, the subclass

of `delegate-class` created by `define-delegator`. It also contains a dictionary of the delegate objects it has seen, each one paired with the associated delegator which it created.

During the re-traversal mentioned above, if the filter finds or creates a delegator object for the object which the event hit, the event is handled by the delegator object. To create a delegator the system also continues the re-traversal and obtains the delegate from the first filter which can provide one. This allows the construction of multi-level delegate/delegator pairs to implement filter composition. If no filter provides a delegate the original object is used.

3.15.4 Filter Composition

Now let us consider how these filters will compose. We would like a filter which is placed on top of the `text-editor` filter to use the `text-being-edited` objects as delegates for its own delegator type (if it knows how.) Filters you would place over a text editor filter might implement various input methods for international character sets.

For composition to work correctly, it is important that we locate the delegator by starting our search from the topmost filter, which is also the first filter the event encountered. In this way we will locate the most derived delegator, which includes the most functionality. In terms of our example in figure 3.2, the delegate object is a `text` object and the delegator created by filter f_1 is a

`text-being-edited` object, while f_2 has built a delegator d_2 that extends delegate d_1, the `text-being-edited` object. Delegator d_2 might be of a class that overrides the event handling methods of the `text-being-edited` class to implement a specialized text input method, such as the Pin Yin method for inputing Chinese text. Filter f_3 does not participate, as it does not recognize `text` as a delegate class.

There is one final bit of complexity to be noted here. If an event arrives before either of the delegator objects in figure 3.2 have been created, d_1 must be created *before* d_2, so that d_2 can use d_1 as its delegate. This is accomplished by a recursive call to the delegator creation procedure.

Now suppose we place one `text-editor` filter on top of another. The topmost filter will see the `text-being-edited` delegators, but it will treat them as `text` delegators and generate its own `text-being-edited` delegators. Each filter will manage its own cursor for each `text` object, and events sent to each delegator will perform modifications on the underlying `text` object. The two should be able to co-exist.

It should also be noted that composed filters are able to build on the results of other filters, but also co-exist peacefully if they lack knowledge of one another. This technique has all the advantages of the MIMO (Model In - Model Out) approach without the drawbacks of wasted storage and the difficulties of propagating changes back to the original object. It is also a simple matter to implement the

Recursive Ambush technique (discussed in section 2.3.1) by overriding the delegator's draw method.

3.16 Creating Delegator Classes Dynamically

There is still a problem with the way that filter composition will behave using the implementation described above. Suppose we have two filters with the same delegate class but different delegator classes, for example, a text editor filter and a grammar checker filter. If we put the grammar checker filter on top of the text editor filter the resulting delegators will have the grammar checker behavior but not the text editor behavior. This is because the `text-being-grammar-checked` delegator class is a subclass of `text` rather than `text-being-edited`. If we swap the filters we will be able to edit the text but we won't see the grammar checker's output.

In abstract terms, the problem is that the delegator class of the top filter is not a subclass of the delegator class of the filter below it. Instead, it is a subclass of the delegates at the bottom level. We need the delegator of the top class to be a subclass of the delegator class of the filter below it so it can then further extend the functionality of the delegators that filter produces.

This problem can be solved by using the ability of a CLOS-style object systems to create new classes at run time. Instead of creating a single delegator class

for a filter, we create a new one whenever we see an object which is of an unfamiliar subclass of the delegate class. These `(delegate-class, delegator-class)` pairs can be stored in a hash table associated with the class of the filter. Now a new class will be created to represent a `text-being-edited` object which has a grammar checker filter over it, a `text-being-edited-being-grammar-checked` class. The render method of this new class will invoke the `text-being-edited` render method before making its annotations.

The dynamic creation of delegator classes allows filters which were not designed to work with each other to cooperatively produce combined views.

3.17 Conclusion

The design elements presented in this chapter fulfill the requirements set out in chapter 1 for a zoomable user interface system. The features of the implementation language have had a strong influence on the implementation of many of the features, and the result is an interface with a high degree of flexibility and power. The use of an object system that supports multiple inheritance and multiple dispatch allows components to be written in a highly abstract fashion, which results in a flexible system with many reusable components. This is done without losing performance by using a Scheme compiler and a highly optimized object system.

The most basic feature of the Tab system is the fact that every Tablet has a

private coordinate system. This is analogous to the transformations used in 3-D graphics systems. The Group object uses its transformation to exert control over the objects which it contains, and can be used for the implementation of spatial indexing schemes. The mechanisms used in the update cycle (event/damage/render) also bear some similarity to the mechanisms used in 3-D rendering, but more emphasis is placed on careful damage analysis to avoid as much re-rendering as possible. This is because the objects in Tab are applications, and rendering can involve expensive operations in the application domain.

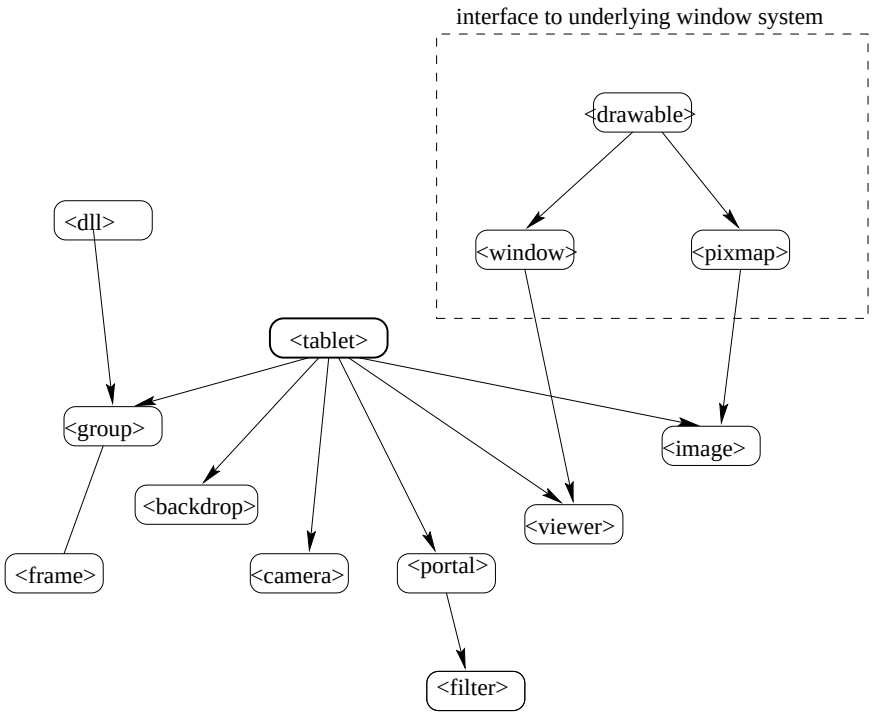
Each Viewer object represents a connection to the outside world, and this provides an intuitive model for multi-view and multi-user systems. Because this is the only avenue to the real world, damage is only interesting if it eventually affects a viewer. Damaged areas are accumulated in the viewer until the rendering portion of the update cycle is entered, at which point the viewer's render operation is invoked. The refinement process is also initiated at this point by attaching the damaged shape to timer event which will expire about half a second in the future. At that time another render request is initiated with the refinement parameter set to a higher level, so that better quality but more expensive rendering will be done. During the period between the initial and refined rendering any new damage may cancel some or all of the refinement shape. The refinement mechanism trades image quality for responsiveness during the times when the screen is changing too quickly for the user to notice.

The inclusion of portals create the possibility of an object being visible in a viewer more than once, perhaps at different scales. This introduces some issues for the implementation of semantic zooming. In particular, it is vital that an object never assume that it has a single current scale. If it needs to maintain scale specific state information, it must be able to maintain several sets for each of the currently visible scales. Some assistance in meeting this requirement is provided by the transparency mechanism, which can be used to implement semantic zooming at a more sophisticated level than simply responding to requests to "draw yourself at this scale." By creating a "shutter" that is opaque at lower magnifications and transparent at higher magnifications, more detailed objects can be revealed only when appropriate.

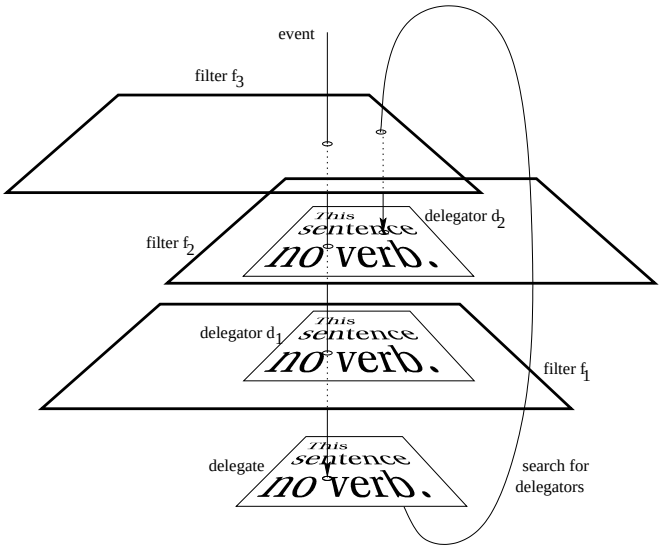
The timer event mechanism used to schedule timer events is also used to implement any animation. Objects simply attach themselves to a timer event, and the object system's dispatch mechanism invokes the proper method of the `alarm` generic at the proper time. The object system's dispatching mechanism is also used to distribute events by generating some generic function names from the event's type and then locating a method that matches the type of object the event hit.

Also presented were mechanisms for saving and restoring a configuration, a constraint mechanism for allowing objects to cooperate, and several techniques for more efficient rendering, such as display lists. Finally, the filtering mechanism

was described, whereby both the appearance and interactive behavior of objects can be augmented and overridden. This is how the Tab mechanisms for communication between objects, including operations such as inspection and editing, are implemented.



The root of the tablet inheritance graph.



An event passes through filters f_3, f_2 and f_1 and hits an object. Then the filter stack is searched for that object's topmost delegator, d_2, which is found in f_2.

Chapter 4

Designing Tab Applications

This chapter describes how the basic classes described in Chapter 3 are used and extended to provide a set of user interface elements. The objects presented here are intended to provide a basis for beginning to explore the system's potential, they do not exploit the full potential of a multi-scale user interface system. They include the set of familiar user interface objects - buttons, menus, and so on. The main goal of this chapter is to show that the mechanisms defined in the previous chapter yield implementations of conventional user interface elements that are simple and straightforward.

This chapter is divided into four sections. The first covers some simple visual elements such as backdrops, text labels, and frames. The next covers some basic event handler objects - one for dragging and zooming with the mouse, one with some bindings for changing the stacking order of objects, etc. The third section

introduces some familiar interactive objects such as buttons and menus. The fourth section covers some basic multi-scale interactive elements.

4.1 Basic Visual Elements

Backdrop – A `<backdrop>` is an opaque tablet of unlimited size. It is usually inserted at the bottom of a group to provide a uniform visual background, by default solid gray. In the default initial configuration it is mixed with an event handler that controls the pop-up menu, zooming and dragging of the view, etc. The only unusual requirement for implementing backdrops is the existence of a distinct value in our shape data structure for an "unbounded shape", one that extends infinitely in all directions. The `initialize` method simply sets the area to this value and a render method is provided which just erases any area it is passed.

Text Label – A `<label>` is a single line of text whose font, foreground color, and background color can be selected. If the background color is set to boolean false (`#f`) the background will be transparent, whatever is below it will show through the UN-drawn areas of the text. This means the `transparent?` predicate must be defined to return the boolean negation of the background color, so that the system will always redraw the area below the label before drawing the label itself:

```
(define-method transparent? ((self <label>) (v <viewer>))
   (not (get-background self)))
```

Frame – A `<frame>` is used to draw a visible border around an object. The border can be raised, flat, or sunken. It is implemented as a subclass of `<group>`, with a special render method that draws the border. It is frequently used to visually separate an object from its surroundings, or as part of an object such as a button that needs to switch from a raised to a flat appearance when activated.

A frame's thickness may be given using several different units of measure, either as a fraction of the frame's dimensions, or in terms of either the frame's public or private coordinate system. If given as a fraction, it can be either a fraction of the width, a fraction of the height, or else each thickness can be computed separately: horizontal as a fraction of the width and the vertical as a fraction of the height.

If the constructor is passed a tablet to be inserted into the frame then the frame's geometry will be adjusted so that the tablet just fits within the frame's inner border.

Besides `<group>`, the `<frame>` class has another superclass `<frame-handler>` which is an event handler to decide whether an event fell onto the frame's border or inside the frame. Based on this decision, it forwards these events to one of two handlers. The `<frame-handler>` also changes the cursor to a special shape when

it is over the frame. An example of the use of a handler for events that fall on the frame's border is to drag and scale the frame and its contents.

Image – One important type of tablet is the `<image>` class, which is a subclass of both the `<tablet>` class and the `<pixmap>` class. The `<pixmap>` class corresponds to a window system object containing a grid of pixel values. An image also has a path slot which holds the name of the file from which the image data is to be loaded. The `<pixmap>` constructor is able to recognize a variety of image formats by examining the file's header, and it applies various filters to convert the image to a common format. Finally, the image is loaded into memory and rasterized so that its pixel values are appropriate for the current window.

The image's private coordinate system is equal to its pixel coordinate system, so to render an area of an image we simply convert the area to integer coordinates and call the `copy-shape` method of the `<pixmap>` class. This method does a fast scaled copy of some sub-shape of the image to the window.

4.2 Basic Event Handlers

This section covers several objects derived from the `<handler>` class, which can be associated with a tablet to give it various types of useful interactive behavior. One of the most important features of the Tab API is the extent to which the interactive behavior is decoupled from the visual appearance. Each bit of interac-

tive behavior is implemented in a different subclass of `<handler>`, and they can all be mixed and matched at will.

4.2.1 Dragging and Zooming

In the interest of encouraging a consistent user interface across Tab applications, a class derived from `<handler>` is provided called `<drag-bindings>`. This class implements a standard interface for dragging and scaling objects. Any object can be given these bindings by simply creating a `<drag-bindings>` instance and assigning it using the `set-handler` method.

First we need to choose a set of bindings. We reserve mouse button one for functions specific to objects, or for popping up a menu if the event hits the background. Then we will assign

- Hold mouse button 2 to zoom in
- Hold mouse button 3 to zoom out
- Holding both buttons 2 and 3 to drag the view

This handler class has a number of slots, both class slots and instance slots, which hold values that control the interactive behavior. The first of these is `zoom-in-velocity`, a class slot. This holds a real number which expresses the zoom speed in terms of the change in scale per second. The default value is 2, meaning that the size will double each second when zooming in and halve each

second when zooming out. When the mouse goes down at the start of a zoom a timer is set, and every time that timer goes off the view is updated and the timer is reset.

When the mouse comes up the view is updated and the timer is canceled. The zoom factor of the last step must be based on the time the mouse up event occurred, not the current time when the event gets handled. Even having taken these measures, the events can be delayed while they are waiting to be read from the socket connecting X server and client, and they aren't assigned a time until they are retrieved by the client side library. The only solutions to this problem are to keep the rendering from causing long delays by using semantic zooming, or to use multi-threading to retrieve the event asynchronously.

The zoom factor at any time is determined by the elapsed wall clock time that the mouse button has been down up to that point. By basing the zoom on the actual elapsed time we get a constant rate of zooming, even if sporadic delays occur. For a given zoom velocity v and time period t, the calculation is the same as figuring what interest rate is required to produce a given return – the desired zoom factor is f in the equation $f^t = v$ (as can be seen by setting t to either zero or one,) thus,

$$\begin{aligned} t \log f &= \log v \\ f &= e^{(\log v)/t} \end{aligned}$$

Other slots related to zooming include an `action` slot, which holds the symbol `zooming` while a zoom is occurring. This is used to decide whether a zoom needs to be canceled when the mouse button comes up. The slot `fixed-point` holds the mouse position at the time of the last timer event, and is used as the position around which the zooming occurs. The `zoom-factor` slot holds the amount by which the object is currently zoomed, and the `zoom-velocity` slot holds either `zoom-in-velocity` if we are zooming or its inverse if we are zooming out.

Note that the object being zoomed grabs the event stream during the zoom, so that the zoom continues even if the mouse happens to end up outside of the object's shape after a zoom step.

One more refinement that can be useful is to have the zoom start out at a slower rate to allow fine tuning of scale, later increasing to a higher rate for coarse adjustment.

Dragging is similar to zooming, but in some ways simpler. Like zooming, we save the initial mouse position as our fixed point. At any time, the position of the drag is given by the difference between the mouses current position and the fixed point. This avoids cumulative errors that might result from repeatedly adding incremental mouse movements to the object position. As in zooming, the event stream is grabbed during dragging, so that the mouse can move off screen without dropping the object if necessary.

There is also a subclass `<tablet-bindings>` of `<drag-bindings>`, which in-

cludes methods that are only appropriate for regular objects, not for cameras. These include an action to change the tablet to its natural size, and one to raise a tablet to just below the viewer. Note that these actions ignore the prefix, because they don't actually change the tablet's geometry, just the view or the stacking order. In order to save users the annoyance of getting lost, there is also a `<home-handler>` which is bound to the logo, and returns the main view to its original position when it receives a click event.

4.2.2 Event Symbols

The `<event>` object has a method which returns a list of symbols that describe the event. This list starts with the most specific description (e.g. `shift-buttonpress-2`) and ends with the least specific (e.g. `any-buttonpress`.) To handle an event a search is made for a generic function that can handle the tablet that received the event as an argument. If no method is found in the most specific generic function, the less specific ones are tried. For example, the `buttonpress-3` method of the `<drag-bindings>` handler starts a zoom out, so this method will be activated whenever a tablet that has `<drag-bindings>` as a superclass receives such an event, unless a more specific method exists in that generic function.

The `<drag-bindings>` object also has a `prefix` slot which holds a prefix which is prepended to all the event handling methods that it creates. This can be used to create a drag binder which only reacts when the `shift-` key or the `control-`

key is down. By convention, actions that only affect the view are bound with no modifiers, actions that move or scale an object are bound with the `shift-` prefix, and objects that modify the view of a portal are bound with the `control-` prefix.

4.3 Basic Interactive Elements

4.3.1 Buttons

A button consists of a tablet, a frame around the tablet with some special bindings, and an action. The button is initially up, meaning that its frame is in "raised" mode. When a buttonpress or motion event with a mouse button down occurs inside the frame, the button goes into "pressed" mode, meaning that the frame takes on a "sunken" appearance. When a button release event occurs inside the frame the action is performed and the button goes back into raised mode. If the pointer leaves the button while it is down it goes up again without triggering the action.

Tab provides three classes derived from a basic `<button>` class:

- A `<momentary-button>` goes down when the mouse button is pressed, or when the mouse pointer enters its face while the mouse button is pressed. It comes back up when the mouse button is released, or when the mouse pointer leaves its face. It has an action function which is invoked only when the mouse button is released while the mouse pointer is on its face.

- A `<toggle-button>`'s state inverts only when the mouse button is pressed while the mouse pointer is on its face. Its action function is invoked only when its state goes from up to down.

- A `<radio-button>` is one of a set of `<toggle-button>`s, each of which automatically go into the up state whenever any of the other members go down.

A button is a frame with an object inserted into it to display the button's face – often a text label. When the button is pressed (receives a buttonpress event or a motion event with the button down) the frame relief is changed from raised to flat. The button also grabs the event stream. This is to ensure that the button knows when the mouse pointer leaves its surface so it can do a release.[1]

4.3.2 Menus

A `<menu>` is a `<group>` containing several buttons arranged in a column or row. The menu constructor lays out the buttons in either a row or a column, and inserts them into the group beneath the camera associated with the menu. In vertical menus all the buttons are equal in both height and width, narrower buttons are extended to the width of the widest. Horizontal buttons are all the same height, but they are allowed to be of different widths.

[1]Many systems use enter and leave events to achieve this functionality. Tab allows multiple simultaneous grabs of the event stream to achieve the same effect.

A <popup-menu> is a <menu> which appears when an object (usually a <back drop>) which implements <popup-handler> receives a buttonpress-1 event. It then grabs the event stream and remains up until a buttonrelease-1 event is received. If that event occurs on one of the buttons it will be activated. A popup menu has a pop-marker slot, which holds the object below which the popup menu will be inserted.

When a menu is popped up, the position of the mouse and the scale of the view is saved. This position can then be retrieved from the menu to position objects created by an action in the menu. The menu is first positioned using the saved position and scale factors. Then it is moved so that the mouse is near the left side of the top button. (It is assumed here that the menu's orientation is vertical.)

4.3.3 Tabview

The <tabview> is a subclass of the <viewer> object that provides support for panning and zooming over the surface. It has become traditional for a Pad system to provide such an interface for navigating Pad space and the objects on it when the system is started, and to provide a logo and other instruments that do not move – they remain "stuck to the glass" of the user's monitor. This is accomplished by creating an arrangement of portals and cameras which are managed by a <tabview> object. This section describes how a <tabview> object is initialized.

The arrangement is shown in figure 4.1.

The viewer is fixed – that is, it cannot be dragged around using the mouse. Below it is the logo, and below that a portal which is attached to a floating camera. The floating camera responds to various mouse events to allow the user pan and zoom around any objects between it and the backdrop. The `<tabview>` creates and manages this arrangement.

4.4 Basic Multiscale Elements

The next task is to begin designing applications that take advantage of Tabula Rasa's multi-scale features. Several other design issues can be identified:

Hysteresis – The scale at which an object appears should be larger than the scale at which it disappears. If for some reason an object is more expensive to render the first time its appearance is delayed until it reaches a larger scale than it would normally appear at. This might be due to the necessity of loading an image from disk into memory.

Text Labels – In Tab you can draw text at any aspect ratio, but only some ratios are pleasing to the eye. In general, ratios wider than the text's natural size are unacceptable, while a certain range narrower than natural size are acceptable. Therefore, text which does not fill the allotted space is horizontally centered in

that space, while text that cannot be narrowed enough to fit is truncated on the right.

Aspect Ratio – The optimal aspect ratio of an object can change as its scale changes. A text label representing a file name generally has a wide aspect ratio, while the image that file contains usually has an aspect ratio fairly close to one. How can we make the transition from label to content without confusing the user or wasting space?

We begin with a few simple (but useful) applications.

4.4.1 A Magnifying Glass

A `<magnifier>` is a framed portal. The frame is given bindings (via inheritance) to allow dragging and scaling of both itself (via its border) and its look-on (by using the mouse while holding the control key.) The portal is connected to a camera which inherits from a constraint object to keep it centered on and directly below the frame, as shown in figure 4.2

4.4.2 Filtered Rendering Example

A simple example of filtered rendering is a set of x, y data points which we wish to be able to see using different types of visualizations, such as a line graph or a bar graph. Suppose we have defined a class `<xy-data>` which can contain a set

of data and which displays itself as two columns of numbers.

We will now define two subclasses of `<filter>` called `<xy-linegraph-filter>` and `<xy-bargraph-filter>`. When these filters are placed over an `<xy-data>` object we want them to give different views of the data. One of the filters will display the data as a line graph, the other as a bar graph.

The line graph and bar graph are constructed identically, except for the details of the render method. For the bar graph, a class `<xy-bargraph-filter>` is derived from `<filter>`. Its constructor initializes its delegate class to `<xy-data>` and its delegator class to `<xy-bargraph>`. The `<xy-bargraph>` class is derived from `<xy-data>`, and a render method for this class is created which scans the data to locate its extremes and draws the bar graph. That is all that is needed for a filter which simply changes the appearance of a tablet.

4.4.3 Filtered Event Handling

In the same way that the previous section was a demonstration of filtered rendering, this section is a demonstration of filtered event handling. A text buffer object is presented, and a filter which implements a simple work-through editor. When the filter is placed over a text buffer a new "text being edited" object is created whose type has a delegation style inheritance relationship with the text buffer's type. It extends the text buffer's functionality by adding a cursor and an event handler to interpret the various editing commands.

The <text> object is derived from <tablet>, and it contains all the information to describe the content and appearance of the text: a file name, the lines of text, a font, foreground and background colors, the widths of the margins and the aspect ratio of the page. It has a constructor method that reads the text from a file. It also has a render method that draws the text according to the appearance parameters described above. The render method is divided into two parts, a render-background method and a render-text method. This allows subclasses of <text> to use the text drawing method after drawing a different background. The <text> class also has methods to insert and delete text.

The <text-filter> object is a subclass of <filter> whose delegate class is <text> and whose delegator class is <text-being-edited>. The <text-being-edited> class is a delegation subclass of <text>. It adds two slots to the <text> class to hold the integers cursor-row and cursor-col, as well as a cursor-color slot.

The <text-being-edited> render method first fills the background with a slightly darker color than the <text> class background color, to visually distinguish editable text. It then determines the location of the character that the cursor is positioned over and fills the character's bounding rectangle with the cursor color. It then invokes the <text> method to draw the characters of the text.

The <text-being-edited> class also has methods up, down, left and right

which move the cursor by damaging the old cursor location, changing the values of `cursor-row` and `cursor-col`, and then damaging the new position. Finally, the events are bound to actions by defining event-named methods for the cursor keys, the text keys and so on.

4.4.4 A Drawing Filter

The next type of object is a filter for editing drawings, which are collections of drawing primitives. It demonstrates how controls can be attached to a filter by giving the user buttons to select the type of object to draw and the color to use to draw it. Because we need to associate some controls with the filter (to select the current drawing style and color), we derive it from a `<group>` rather than a `<filter>`. Inside the group go the control menus and a `<filter>`, to which we assign the delegate class `<drawing>` and the delegator class `<drawing-being-edited>`. A `<drawing>` is essentially a display list, while a `<drawing-being-edited>` adds to that some state variables to hold information about the object being drawn. For example, if we are drawing a rectangle the position of the initial corner is stored so a "rubber band" rectangle can be drawn until the final position is selected. At that point primitive `<drawing>` methods are called to add the rectangle to the display list data structure.

4.4.5 Directory Viewer

This section describes a tablet that provides a view of a directory tree. This is the first semantic zooming application, in that it displays a hierarchical data structure and allows the user to move closer and closer, revealing more detail in the process.

The viewer will lay out the elements of a directory in a rectangular grid, maintaining a certain aspect ratio and element spacing. Each member of the directory is represented by a separate tablet of a type suitable to the type of file it represents – images are represented by image tablets, text files by text tablets, and so on. In particular, each subdirectory is itself represented by the same type of directory viewer as the top level object.

Semantic zooming must be employed by the directory viewer to prevent any attempt to instantly load the entire content of each of its elements, which would cause a recursive loading of every file in the entire subtree. Instead, a directory is initially populated by `<loader>` objects, each of which are assigned to a particular path name and will load that object the first time it is displayed above a certain scale. Until that time they simply display a plain background with the name of the file they represent.

We also want the objects to disappear when they once again become small, as rendering even a small version of an object can be computationally expensive in relation to the amount of screen area they cover. For this reason we don't

simply replace the loader object with the real object, but instead the real object is inserted into the loader, which is derived from the group class. The loader's render method checks the scale of any render requests and invokes its own simple rendering method at small scales and the default group rendering method at larger scales when we want to see the contained object.

We want the directory editor to be able to use standard Tab objects to represent directory elements – we don't want to invent special objects for the directory editor's use, or to make the directory a single object that has its own way of displaying files. For this reason, we derive the `<directory>` class from a `<group>` class, (initially `<oblist>`) and add a caption giving the folder's name. Then the elements representing the directory's files will be inserted into the group.

Directory Layout Now we need to determine a suitable lay out for the file nodes. Each file will have the aspect ratio given in `page-aspect`, and we want gutters between each row and column, but not around the edge. The vertical and horizontal gutters should be as nearly equal as possible. Our approach is to find the best layout with no gutters, and then to scale the nodes down until both vertical and horizontal gutters exceed a minimum size.

One approach is to try different numbers of rows until the amount of wasted space is minimized. Suppose we have n nodes to be laid out. We can compute the number of columns c for a given number of rows r as follows. We know that

the first $r-1$ rows will each have c nodes, and the last will have between 1 and $c-1$, so

$$c(r-1)+1 \leq n \leq c(r-1)+(c-1)$$

$$r \leq \frac{n+c-1}{c} \leq r+\frac{c-2}{c}$$

Which, for $c \geq 1$, can be expressed as

$$r = \lfloor \frac{n+c-1}{c} \rfloor$$

Suppose now our desired aspect ratio (height divided by width) is a, and we have a rectangle of width w and height h. We need to choose the value for c which yields spaces in our layout whose aspect ratio is closest to a. For a given choice of c the actual aspect ratio of the spaces a' is

$$a' = \frac{h/r}{w/c} = \frac{hc}{wr}$$

$$a' = \frac{hc}{w\left(\lfloor \frac{n+c-1}{c} \rfloor\right)}$$

For a given choice of c, the resulting layout will either yield spaces for nodes which have too large an aspect ratio, in which case horizontal space will go unused, or to small an aspect ratio, in which case vertical space will go unfilled. As the

number of columns increases, the aspect ratio will increase. We want to stop when the aspect ratio first exceeds the the target aspect ratio, and return either this or the previous layout, depending on which wasted less space. For an area w by h, the amount of wasted space will be

$$\max(h(w - ch/ra), w(h - rwa/c))$$

An adequate method of computing the best value of c is to try each value until the amount of wasted space begins to increase.

4.4.6 Animation

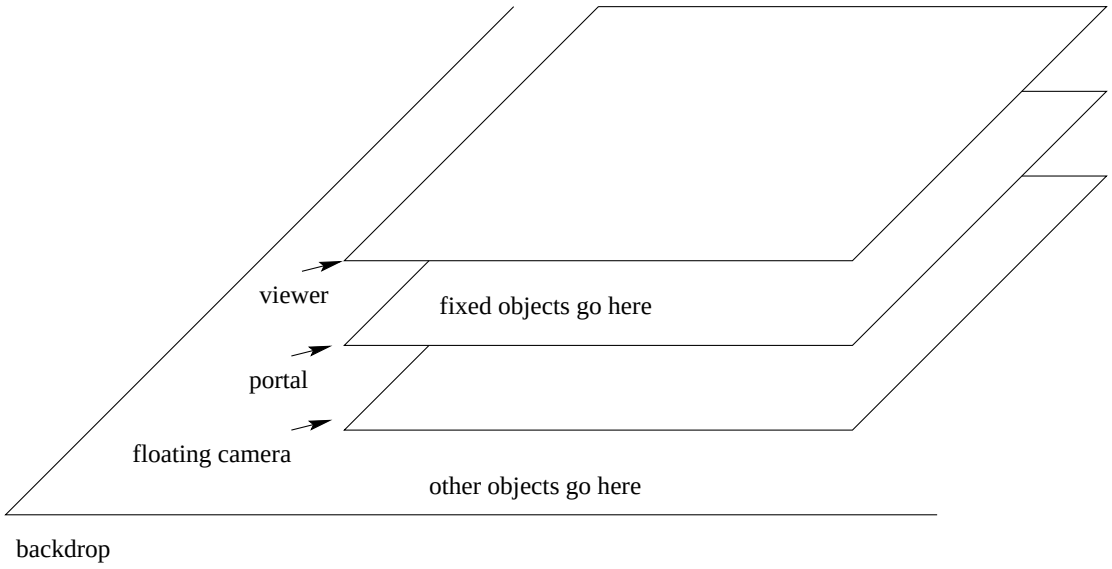
An `<animation>` is a tablet containing a vector of images, a current offset, and a frame rate. The constructor first checks that there are at least two images in the animation. It then locates the viewer argument so it can loop through images turning them from path names into `<image>` objects. Finally, it sets up timer loop by passing itself to `set-timer` and defining an `alarm` method which increments the frame number (modulo the number of frame), damages the animation, and resets the timer. The render method simply calls the render method of the current image.

4.5 Conclusion

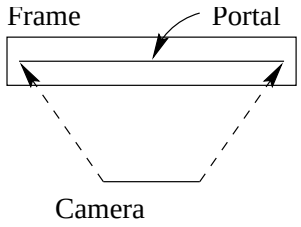
This chapter has described the most basic Tab objects which have been implemented on top of the framework laid out in the previous chapter. With the creation of these objects we begin to explore the fundamentals of multi-scale application design.

Apart from their multi-scale behavior, the basic visual elements are similar to those in non-zooming systems: text, images, frames, etc. Interactive behavior is assigned by mixing event handling super classes into an object using multiple inheritance. Timer events are used to achieve animated effects, zooming in and out in particular. Filtered rendering and filtered event handling provide a means of communication between objects.

An examination of the implementation of Tabula Rasa will show that each of these objects are implemented quite simply on top of the framework provided in chapter 3.



Initial system configuration.



A magnifier is a framed portal whose camera is constrained to stay directly below the frame. When the camera is smaller than the portal (as shown here) it is enlarging.

Chapter 5

Conclusions

This dissertation has presented a working design for a new style of user interface development system called "Pad". Just as certain elements are combined to create the Desktop interface, this new style interface has several distinguishing characteristics:

- It is a zoomable system, all data objects exist in a two dimensional floating point coordinate system,
- applications adopt a "work-through" mechanism, meaning that they adopt the metaphor of a transparent filter which is placed over the objects upon which they are to operate,
- a meaningful and consistent semantics is provided for the composition of multiple filters.

By adding these capabilities to the user interface substrate underlying the system's applications, it gives the applications a different flavor than those of conventional systems such as Microsoft Windows. It is expected that applications will tend to be of a finer granularity and greater number, each one doing a smaller job.

Certain existing commercial systems already incorporate some subset of these principles. The success of plug-ins in the Adobe Photoshop image processing package is an important example. Here the task of image processing makes filter composition a natural way of representing the task at hand. An important element of the success of the Unix operating system is based on the composition of filters of text streams. Again, the key is the ability to compose active components in a natural and meaningful way.

5.1 Tab

An example implementation named *Tab* has been created to experiment with the Pad style of user interface system. Tab has evolved into a system which makes heavy use of advanced object oriented techniques, including multiple inheritance, multi-methods and delegation. As a result, the components of Tab are written in a very abstract fashion, and are consequently highly reusable.

In the Tab system, filters are composed using an object-oriented methodology.

Placing a filter over an object creates a new object, but the object shares both the methods and the data slots of the old object, so that changing the value of a slot in the new object changes the value of that slot in the old object, and vice versa, using delegation style inheritance.

The type of the new object is defined in the filter, and it can have added slots and methods just as any subtype does. This allows a very general type of filtering of the visual and interactive characteristics of the old objects. The same delegation mechanism can be applied to the subtypes to create sub-subtypes, allowing the composition of multiple filters.

What are the uses of this type of composition? It is difficult to predict what sorts of applications might be developed using such a system. However, the close relationship between delegation and other popular types of inheritance suggests that the technique could have widespread applications. Consider an interface to a set of cartographic databases. The base object could be a geometric model of the globe, implementing the type of projection which maps latitudes and longitudes onto a plane. Then there could be a filter which represented a set of geographic data – it would derived a new filtered object from the globe object to create the display. Multiple filters representing could be placed over the same globe, and all the data sets would be visible. Other types of filters could be used to select or modify different sets of data, or implement semantic filtering to adjust the amount of detail based on the viewing scale.

5.1.1 Comparison to Pad++

The system most closely related to Tab is Bederson's Pad++[3]. However, there are significant differences in design philosophy between to two systems. There is little emphasis on filters in Pad++, and the mechanisms provided for implementing them are more primitive. Also, the primitive objects in Pad++ are somewhat "heavier" than those in Tab – for example

- portals always have visible borders in Pad++, while the two functions are separated in Tab. This gives portals a more flexible role in Tab, where they can be assembled into a variety of different derived forms.

- Pad++ portals always look at the top level, instead of being attached to a camera object which can be anywhere in the stacking order. This again makes Tab portals more flexible, where they can be used as filters or magnifying glasses which see only the objects below themselves.

- Pad++ objects are assigned a layer number rather than being inserted into different groups. This means a hierarchical structure can be constructed in Tab to implement features such as spatial indexing.

- Pad++ objects have a "sticky" field which determines whether they move or stay still when the main view moves. This functionality is implemented in Tab using a combination of a viewer, and portal and a camera.

- In Pad++ damage is not clipped to the shape of the portals it passes through; portal damage is on an all or nothing basis.

Thus, many of the features of Pad++ are constructed from simpler and more general elements in Tab. On the other hand, more effort has gone into Pad++ to implement spatial indexing to handle large numbers of objects, and to implement better and faster rendering algorithms for text. It may also be that some elements of Tab are inherantly less efficient that the corresponding but less general elements in Pad++.

5.2 Remaining Issues and Future Work

5.2.1 Efficiency

One area where the current implementation could be improved is speed. Although the system has been ported from an interpreted Scheme system to a compiler, a number of optimizations still remain to be performed. The slot access and method lookup algorithms are considerably slower than the best available (for example, those described in [19] and [28]), and these operations currently dominate the system's execution time.

The use of delegation is a step forward compared to the approaches to work-through interfaces described in [5]. However, it is quite likely that a data-driven architecture might be required to give adequate performance in domains where

the methods of filtered subclasses do complex calculations to produce intermediate results that could be cached. This is in some sense related to the *Model-In Model-Out* approach to filtering describe in section 2.3.1, but it is superior in that better heuristics can be applied in selecting which data to cache and when and how to update it.

5.2.2 Portability

Another area of interest in the short term is portability. The system could reach many more users if it was ported to a Microsoft operating system, and the 32 bit Windows NT and Windows 95 systems offer sufficient support to make this task fairly easy.

The most important work to be done is to design and implement applications to inhabit this framework. This is the only way to truly demonstrate the viability of the concepts. It is also the best way to develop the idioms and conventions that are needed to make a new interface paradigm serve all the user's needs.

5.2.3 Asynchrony

There are various areas where an asynchronous implementation could improve the usability of the system. The system should be split into two cooperation threads, the first would process user input, distribute events and propagate damage, the second would do only the rendering. By putting the rendering thread in a timed

loop, we could prevent the system from spending its time updating any areas more often than visually necessary. This would also make it somewhat easier to implement interruptible rendering, where new damage causes any refined rendering to abort even after drawing has begun.

5.2.4 Coordinate Space

One problem which may emerge as zooming user interfaces come into heavier use is a peculiar aspect of the coordinate system which they typically define. Although we have called it a two dimensional real number coordinate system, all current ZUI's actually implement a *floating point* coordinate system. It is tempting to think of Pad space as vast and uniform, but while floating point numbers have a very high resolution near the origin, the local resolution declines quickly as you travel away from the origin. If a collaborative version were implemented in which users began building persistent structures, it would probably not take long for someone to notice this granularity. Some system for defining a local "sub-origin" with a nested floating point coordinate space is probably the easiest way to overcome this problem.

5.2.5 Application Design

Chapter 4 presented a sample of some basic Tab application design techniques. These include using constraints to make a magnifying glass, or deriving a editable

text object from a regular text object. The future of the Tab project depends on developing design techniques which exploit the multi-scale nature of Tab space and the benefits of work-through interfaces.

5.3 Lessons Learned

Finally, a summation of the most important lessons learned in the building of the Tab system and in this disseration. The desktop interface has come to dominate our computer screens over the last quarter of a century, and has progressed little in that time. The advances in hardware speed during that time have been extremely large; it may be that the time is approaching for a sudden shift to a new metaphor. There is a tendency towards large, monolithic applications, a trend which is largely due to the lack of any mechanism for communication in the desktop metaphor between screen objects. These large applications limit the user's opportunities for customizing the computing environment, and they raise the entry barriers for software companies by greatly increasing the effort required to produce a marketable product.

A user interface metaphor is not a single idea, but rather a set of complementary mechanisms. The desktop metphor is windows + icons + menus + pointer, while the Pad metaphor is semantic zooming + filters + constraint system. This new metaphor increases the size of the space in which the user operates, and also

provides a uniform metaphor for communication between screen objects.

To support this type of interface, an implementation language which supports multiple inheritance, multi-methods and delegation inheritance is advantageous. item The mechanisms required to implement a Pad system are more complex than those of desktop systems, so attention to implementation efficiency is important. Increasing the efficiency of the language mechanisms is particularly beneficial. Multi-scale interfaces are a step towards a style of interface which exhibits the qualities of direct manipulation, spaciousness, and animation.

Bibliography

[1] Christopher Ahlberg and Ben Schneiderman. Visual Information Seeking: Tight Coupling of Dynamic Query Filters with Starfield Displays. In *CHI Conference Proceedings*, page 313, 1994. http://www.acm.org/pubs/citations/proceedings/chi/191666/p313-ahlberg/.

[2] Joel F. Bartlett. `SCHEME->C`: a Portable Scheme to C Compiler. Technical Report 89.1, Digital Equipment Corporation, January 1989. http://www.research.digital.com/wrl/techreports/abstracts/89.1.html.

[3] Benjamin B. Bederson and James D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *UIST Symposium Proceedings*, pages 17–26, 1994. http://www.acm.org/pubs/citations/proceedings/uist/192426/p17-bederson/.

[4] Benjamin B. Bederson and James D. Hollan. Pad++: A Zoomable Graphical Interface. In *CHI Conference Proceedings*, pages 23–24, 1995. http://www.acm.org/pubs/citations/proceedings/chi/223355/p23-bederson/.

[5] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. Toolglass and Magic Lenses: The See-Through Interface. In *SIGGRAPH Conference Proceedings*, pages 73–80, 1993. http://www.acm.org/pubs/citations/proceedings/graph/166117/p73-bier/.

[6] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification X3J13. *SIGPLAN Notices Special Issue*, 23, September 1988.

[7] Richard A. Bolt. Spatial Data Management, 1979.

[8] Vannevar Bush. As we may think. *The Atlantic Monthly*, July 1945.

[9] Paul Calder and Mark Linton. Glyphs: Flyweight Objects for User Interfaces. In *UIST*, pages 92–101, Snowbird, Utah, October 1990.

[10] D. Brookshire Conner. Supporting Graphics Using Delegation and Multi-Methods. Technical Report CS-93-33, Brown University, September 1993. http://www.cs.brown.edu/publications/techreports/reports/CS-93-33.html.

[11] R. R. et. al. Everett. Sage: A data-processing system for air defense. In *Proceedings of the Eastern Joint Computer Conference*, Washington, D.C., 1957.

[12] Marc Feeley. *Gambit Scheme System*. http://www.iro.umontreal.ca/ gambit/.

[13] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughs. *Computer Graphics: Principles and Practice, Second Edition*. Addison-Wesley, 1990.

[14] Matthew Fuchs. Escaping the Event Loop: An Alternative Control Structure for Multi-Threaded GUIs. In C. Unger and L.J. Bass, editors, *Engineering for HCI*. Chapman & Hall, 1996. http://www.cs.nyu.edu/phd_students/fuchs.

[15] Erick Gallesio. Embedding a Scheme Interpreter in the Tk Toolkit. In *First Tcl/Tk Workshop*, pages 103–109, Berkeley, CA, 1993. http://kaolin.unice.fr/STk.html.

[16] Don Gentner and Jakob Nielson. The Anti-Mac Interface. *Communications of the ACM*, 39(8), August 1996.

[17] Brian Johnson and Ben Schneiderman. Tree-Maps: A Space-Filling Approach to the Visualization of Heirarchical Information Structures. *IEEE Visualization*, page 284, 1991.

[18] William P. Jones and Susan T. Dumais. The Spatial Metaphor for User Interfaces: Experimental Tests of Reference by Location versus Name. *ACM Transactions on Office Information Systems*, page 42, January 1986.

[19] James Kempf, Warren Harris, Roy D'Souza, and Alan Snyder. Experience with CommonLoops. In *OOPSLA Conference Proceedings*, 1987.

[20] Gregor Kiczales. TinyCLOS, 1992. ftp://parcftp.xerox.com/pub/openimpls/tiny/.

[21] Gregor Kiczales and Daniel Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1992.

[22] Donald E. Knuth. *The TeX book*. Addison-Wesley, Reading, MA, 1984.

[23] George Lakoff and Mark Johnson. *Metaphors We Live By.* University of Chicago Press, Chicago and London, 1980.

[24] Brenda Laurel. *Computers as Theatre.* Addison Wesley, 1991.

[25] Chris Laurel. wt - A 3-D Game Engine, 1994. ftp://magoo.uwsuper.edu/pub/wt.

[26] Mark A. Linton. Implementing Resolution Independence on Top of the X Window System. In *Proceedings of the Sixth X Technical Conference*, pages 109–116, Boston Massachusetts, January 1992.

[27] Michael Morrison. *Becoming a Computer Animator.* Sams Publishing, 1994.

[28] Kelly E. Murray. Under the Hood: CLOS. *Journal of Object-Oriented Programming*, pages 82–86, September 1996.

[29] Jeyakumar Muthukumarasamy and John T. Stasko. Visualizing Program Executions on Large Data Sets Using Semantic Zooming. Technical Report GIT-GVU-95-02, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, January 1995. http://www.cc.gatech.edu/gvu/research/techreports95.html.

[30] Ted Nelson. *Computer Lib / Dream Machines 2nd ed.* Tempus Books of Microsoft Press. First edition, 1987.

[31] Theodor Holm Nelson. The Right Way to Think about Software Design. In Brenda Laurel, editor, *The Art of Human-Computer Interface Design.* Addison-Wesley, Reading, Mass, 1990.

[32] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[33] Ken Perlin and David Fox. Pad: An Alternative Approach to the Computer Interface. In *SIGGRAPH Conference Proceedings*, 1993. http://www.acm.org/pubs/citations/proceedings/graph/166117/p57-perlin/.

[34] D. M. Ritchie. The UNIX System: The Evolution of the UNIX Time-sharing System. *Bell Systems Technical Journal*, 63(8):1578, October 1984.

[35] Robert W. Scheifler and James Gettys. *X Window System, 3rd ed.* Digital Press, 1992.

[36] Ben Schneiderman. *Designing the User Interface, 2nd ed.* Addison Wesley, 1992.

[37] Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP*, Aarhus, Denmark, August 1995.

[38] Richard Stallman. EMACS: The Extensible, Customizable Self-Documenting Display Editor. In *Proc. SIGPLAN/SIGOA Symposium on Text Manipulation*, Portland, OR, 1981. ACM.

[39] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communications System. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346, Baltimore, MD, 1963. Spartan Books.

[40] David Turo and Brian Johnson. Improving the Visualization of Heirarchies with Treemaps: Design Issues and Experimentation. *IEEE Visualization*, page 124, 1992.

[41] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA Conference Proceedings*, pages 227–241, Orlando, FL, October 1987.

[42] Andries van Dam. Computer Graphics Comes of Age (Interview). *Communications of the ACM*, 27(7):638–648, 1984.