Past, Present and Future of User Interface Software Tools

Brad Myers

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
412-268-5150
Fax: 412-268-1266
bam@cs.cmu.edu
Carnegie Mellon University

Scott E. Hudson

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
412-268-2429
Fax: 412-268-1266
hudson@cs.cmu.edu
Carnegie Mellon University

Randy Pausch

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
412-268-3579
Fax: 412-268-1266
pausch@cs.cmu.edu
Carnegie Mellon University

9/1/2000 12:31 PM

To appear in:
John M. Carroll, ed. *HCI In the New Millennium*. Addison-Wesley, 2001. To appear.
Revision of a longer article:
Brad Myers, Scott E. Hudson, and Randy Pausch,
"Past, Present, and Future of User Interface Software Tools," *ACM Transactions on Computer Human Interaction*, vol. 7, no. 1, March, 2000. pp. 3-28.

Abstract

A user interface software tool helps developers design and implement the user interface. Research on past tools has had enormous impact on today's developers—virtually all applications today were built using some form of user interface tool. In this chapter, we consider cases of both success and failure in past user interface tools. From these cases we extract a set of themes that can serve as lessons for future work. Using these themes, past tools can be characterized by what aspects of the user interface they addressed, their threshold and ceiling, what path of least resistance they offer, how predictable they are to use, and whether they addressed a target that became irrelevant. We believe the lessons of these past themes are particularly important now, because increasingly rapid technological changes are likely to significantly change user interfaces. We are at the dawn of an era where user interfaces are about to break out of the "desktop" box where they have been stuck for the past 15 years. The next millennium will open with an increasing diversity of user interfaces on an increasing diversity of computerized devices. These devices include hand-held personal digital assistants (PDAs), cell phones, pagers, computerized pens, notepads and watches, and various kinds of desk and wall-size computers, as well as devices in everyday objects (such as mounted on refrigerators, or even embedded in truck tires). The increased connectivity of computers, initially evidenced by the World-Wide Web, but spreading also with technologies such as personal-area networks, will also have a profound effect on the user interface to computers. Another important force will be recognition-based user interfaces, especially speech, and camera-based vision systems. Other changes we see are an increasing need for 3D and end-user customization, programming, and scripting. All of these changes will require significant support from the underlying user interface software tools.

1. Introduction

There is no question that research in the area of user interface software tools has had an enormous impact on current practice of software development. Virtually all applications today are built using window managers, toolkits and interface builders that have their roots in the research of the 70's, 80's and 90's.

These tools have achieved a high level of sophistication due in part to the homogeneity of today's user interfaces, as well as the hardware and software platforms they run on. Whereas the 70's saw a tremendous amount of experimentation with a variety of input devices and user interface styles, much of the diversity is now gone from user interfaces. Almost all applications on Windows, Unix or the Macintosh look and work in a very similar fashion, primarily using a small set of constructs invented 15 or more years ago. Further, the hardware platform has largely stabilized on the now familiar desktop machine – one with a large (typically color) bitmap screen, a keyboard, and a mouse (with between one and three buttons).

This stability has had important positive benefits. For end users, the consistency of interfaces now available makes it possible for them to build skills that largely transfer between applications and platforms – knowing one graphical user interface provides skills that apply to many others. For tool builders this relative lack of change has also allowed them to go through significant refinement of concepts. In many respects tools have been able to mature and "catch up" with an otherwise moving target.

However, many feel that significant opportunities for improved interfaces are being lost to stagnation. In addition, conventional GUI (Graphical User Interface) techniques appear to be ill-suited for some of the kinds of interactive platforms now starting to emerge, with *ubiquitous computing* devices [42] having tiny and large displays, *recognition-based user interfaces* using speech and gestures, and requirements for other facilities such as end-user programming.

As predicted by Mark Weiser at Xerox PARC, the age of *ubiquitous computing* is at hand. Personal Digital Assistants (PDAs) such as the Palm Pilot and personal organizers such as the Sharp Wizard are already popular. Digital cell phones are merging with digital pagers and PDAs to form portable, wireless communication devices that support voice, along with electronic mail, and personal information such as schedules and contact lists. Wall-size displays are already available as projection devices such as the SMART Technologies SmartBoard or large plasma

panels such as ImageSite from Fujitsu, which is 42 inch wide. It is inevitable that the costs of hardware will continue to drop, and that new computational opportunities will arise. For example, connectivity will become easier due to new wireless technologies such as the BlueTooth in-room radio network [9].

Interfaces on these very large and very small displays cannot typically use the standard desktop model, and people will not necessarily expect these devices to act like "regular" computers. Reviews comparing the 3Com Palm Pilot with Windows CE devices often make the point that the Windows user interface style created for the desktop does not work well on palm-size devices. And it clearly does not work for a tiny display on a phone. Similarly, the standard Windows widgets such as pull-down menus are not appropriate on large wall-size displays since, for example, the menus may be literally too high for short users to reach. Furthermore, people will be interacting with multiple devices *at the same time* so the devices will need to communicate and coordinate their activities.

The implication of these changes is that we can expect a dramatic increase in the diversity of both the types of computing devices in use, and the task contexts in which they operate. This in turn implies that we are poised for a major change in user interfaces, and with it dramatic new needs for tools to build those interfaces. It is especially important to explicitly consider the effects our tools will have on what we can and will build, and to create new tools that have the properties needed to meet a new generation of demands. There are many examples that show that tools have significant impact on the styles of interfaces that are created. For example, in the World-Wide Web, it is actually easier to use pictures as buttons rather than to use "real" button widgets. Therefore, designers created elaborate, animated user interfaces with rich visual design and high production values.

Why are tools so important and successful? In general, tools help reduce the amount of code that programmers need to produce when creating a user interface, and they allow user interfaces to be created more quickly. This, in turn, enables more rapid prototyping and therefore more iterations of iterative design that is a crucial component of achieving high-quality user interfaces. Another important advantage of tools is that they help achieve a consistent look and feel, since all user interfaces created with a certain tool will be similar.

This paper briefly discusses the history of successes and failures of user interface software tool research to provide a context for future developments. It then discusses the implications on tools

of the impending changes. Finally, it discusses the requirements for the underlying operating system to support these tools.

2. Historical Perspective

2.1. Themes in Evaluating Tools

In evaluating past and future tools, we have identified some themes that seem to be important in determining which are successful.

- **The parts of the user interface that are addressed:** The tools that succeeded helped (just) where they were needed.
- **Threshold and Ceiling:** The "threshold" is how difficult it is to learn how to use the system, and the "ceiling" is how much can be done using the system. The most successful current systems seem to be either low-threshold and low-ceiling, or high threshold and high ceiling. However, it remains an important challenge to find ways to achieve the highly desirable outcome of systems with both a low threshold and a high ceiling at the same time.
- **Path of Least Resistance:** Tools influence the kinds of user interfaces that can be created. Successful tools use this to their advantage, leading implementers towards doing the right things, and away from doing the wrong things.
- **Predictability:** Tools that use automatic techniques that are sometimes unpredictable have been poorly received by programmers.
- **Moving Targets:** It is difficult to build tools without having significant experience with, and understanding of, the tasks they support. However, the rapid development of new interface technology and new interface techniques can make it difficult for tools to keep pace. By the time a new user interface implementation task is understood well enough to produce good tools, the task may have become less important, or even obsolete.

2.2. What worked

User interface tools are an area where research has had a tremendous impact on the current practice of software development [21]. Of course, window managers and the resulting "GUI style" comes from the seminal research at the Stanford Research Institute, Xerox Palo Alto Research Center (PARC), and MIT in the 1970s. Interface builders were invented in research laboratories at BBN, the University of Toronto, Xerox PARC, and others. Now, interface builders

are widely used for commercial software development. Event languages, as widely used in HyperTalk and Visual Basic, were first investigated in research laboratories. The current generation of environments, such as Microsoft's OLE (Object Linking and Embedding) and Java Beans, are based on the component architecture that was developed in the Andrew environment [31] from Carnegie Mellon University. The following sections discuss these successes in more detail. A more complete history appears elsewhere [21] and another reference contains a comprehensive survey and explanation of user interface tools [20].

2.2.1. Window Managers and Toolkits

Many research systems in the 1960s, such as NLS [7], demonstrated the use of multiple windows at the same time. Alan Kay proposed the idea of overlapping windows in his 1969 University of Utah Ph.D. thesis [16] and they first appeared in 1974 in his Smalltalk system from Xerox PARC. Many other research and commercial systems picked up the idea from there, notably the Apple Macintosh and Microsoft Windows.

Window managers provide a basic programming model for drawing and screen update (an *imaging model*) and for accepting user input (an *input model*). However, programming directly at the window manager level tends to be time consuming and tedious. Further, when each programmer creates all interface components from scratch, it is practically impossible to provide much widespread consistency for the user. To address these issues, user interface *toolkits* were developed on top of the abstractions provided by window managers. Toolkits typically provide both a library of interactive components, and an architectural framework to manage the operation of interfaces made up of those components. Employing an established framework and a library of reusable components makes user interface construction much easier than programming interfaces from scratch. As first demonstrated by the Apple Macintosh toolbox [1], the fact that a toolkit makes the programmers' job much easier can be used as leverage to achieve the difficult goal of maintaining consistency. Thus by achieving the goal of making the programmer's job simpler, toolkits provide a *path of least resistance* to also achieve the goal of supporting widespread interface consistency.

2.2.2. Event Languages

With *event languages*, the occurrence of each significant event – such as manipulation of an input device by the user – is placed in an *event record* data structure (often simply called an *event*).

These events are then sent to individual event handlers that contain the code necessary to properly respond to that input. Researchers have investigated this style in a number of systems, including the University of Alberta User Interface Management System [8] and others. This led to very popular uses of the event language in many commercial tools, such as the HyperTalk language of Apple's HyperCard, Microsoft's Visual Basic, and the Lingo scripting language in Macromedia's Director.

Event languages have been successful because they map well to the direct manipulation graphical user interface. These systems generate *events* for each user action with the mouse and keyboard, which are directed to the appropriate application that then must respond. Event languages also help encourage the mode-free style of interfaces since the user is in charge of generating events that the application handles. However, as will be discussed later, the *recognition-based* user interfaces that are emerging for modalities such as gestures and speech may not map well to this event-based style, so we may need a new paradigm.

2.2.3. Interactive Graphical Tools

Another important contribution of user interface research has been the creation of what has come to be called *interface builders*. These are interactive tools that allow interactive components to be placed using a mouse to create windows and dialog boxes. Examples include Visual Basic and the "resource editors" or "constructors" that come with Microsoft's Visual C++ and most other environments. Early research on this class of tools includes Trillium from Xerox PARC [10] and MenuLay from the University of Toronto [4]. The idea was refined by Jean-Marie Hullot while a researcher at INRIA, and Hullot later brought the idea with him to NeXT, which popularized this type of tool with the NeXT Interface Builder.

An important reason for the success of interface builders has been that they use graphical means to express graphical concepts (e.g., interface layout). By moving some aspects of user interface implementation from conventional code into an interactive specification system, these aspects of interface implementation are made available to those who are not conventional programmers. Even the programmers benefited, as the speed of building was dramatically reduced. These properties of interface builders can be thought of as providing a *low threshold* to use, and avoiding making it difficult to learn (at least initially). In these systems, simple things can be done in simple ways.

2.2.4. Component Systems

The idea of creating applications by dynamically combining separately written and compiled *components* was first demonstrated in the Andrew system [31] from Carnegie Mellon University's Information Technology Center. Each component controlled its rectangle of the screen, and other components could be incorporated inside. For example, a drawing inside a text document would be controlled by a drawing component, which would be independent of the text editor component. This idea has been adopted by Microsoft's OLE and ActiveX, Apple's OpenDoc, and Sun's Java Beans [14]. One reason for the success of the component model is that it addresses the important and useful aspect of application building: how to appropriately modularize the software into smaller parts, while still providing significant capabilities and integration to users.

2.2.5. Scripting languages

It is no accident that the first toolkits were developed using programming languages that were interpreted: Smalltalk [39] and then Dlisp [38], which were developed by researchers at Xerox PARC, had small toolkits. The interpreted language enables the developer to rapidly prototype different user interface ideas and immediately make changes, which provides fast turn-around. With the rise of C and C++, most user interface development migrated to compiled languages and these capabilities were lost. Researchers have investigated ways to bring these advantages back, resulting in scripting languages such as tcl/tk [30], Python [19] and Perl [41]. Now, these research languages are seeing increasing commercial use, and popular languages such as Visual Basic and Javascript are providing interpreted capabilities.

Combining scripting capabilities with components and an interface builder has proven to be a particularly powerful approach. For example, there are thousands of components for Visual Basic available from third-party vendors. Using the interface builder of Visual Basic for the layout and the Visual Basic language for scripting the "glue" that holds everything together enables people who are not professional programmers to create sophisticated and useful interactive applications. Visual Basic shows that a little programming – if packaged properly – can make it possible for domain experts to create interfaces that reflect their domain and task knowledge.

2.2.6. Hypertext

The World-Wide Web (WWW) is another spectacular success of the research on user interface software and technology. It is based on the *hypertext* idea. Ted Nelson coined the term "hypertext" in 1965 and worked on one of the first hypertext systems called the "Hypertext Editing System" at Brown University. The NLS system [6] also had hypertext features. The University of Maryland's Hyperties was the first system where highlighted items in the text could be clicked on to go to other pages [18]. HyperCard from Apple was significant in helping popularize the idea for a wide audience. For a more complete history of Hypertext, see [26].

Hypertext did not attain widespread use, however, until the creation of the World-Wide Web system by Berners-Lee in 1990, and the Mosaic browser a few years later. Some of the elements of the success of the WWW are the ease of use of Mosaic, the simplicity and accessibility of the html language used to author pages, the ease of making pages accessible on the web, and the embedding of pictures with the text. The WWW provided a *low threshold* of use for both viewers and authors. Viewers had a simple mechanism that provided access to many of the existing network resources (e.g., ftp, telnet, gopher, etc.) within a hypertext framework, and authors used the very simple html textual specification language. This allowed the system to be used by content providers with a minimum of learning. Second, the success of the Mosaic browser clearly demonstrated the power and compelling nature of visual images (and more generally, rich content with high production values).

2.2.7. Object-Oriented Programming

Object-oriented programming and user interface research have a long and intertwined history, starting with Smalltalk's motivation to make it easy to create interactive, graphical programs. C++ became popular when programming graphical user interfaces became widely necessary with Windows 3.1. Object-oriented programming is especially natural for user interface programming since the components of user interfaces (buttons, sliders, etc) are manifested as visible objects with their own state (which corresponds to instance variables) and their own operations (which correspond to methods).

2.3. Promising Approaches That Have Not Caught On

In addition to the lessons learned from successes, a great deal can also be learned from looking at ideas which initially seemed to show great promise, but which did not in the end succeed (or at least have not *yet* succeeded) in delivering on that promise. Many of these succumbed to the moving-target problem: as these systems were being researched, the styles of user interfaces were changing towards today's standard GUI. Furthermore, many of these tools were designed to support a flexible variety of styles, which became less important with the standardization. This section considers several such examples including the concept of a User Interface Management System (UIMS), language-based approaches, constraints, and model-based systems.

2.3.1. User Interface Management Systems

In the early 80's, the concept of a *user interface management system* (UIMS) was an important focusing point for the then-forming user interface software community. The term "user interface management system" was coined [15] to suggest an analogy to database management systems. Database management systems implement a much higher and more useable abstraction on top of low-level concepts such as disks and files. User interface management systems were to abstract the details of input and output devices, providing standard or automatically generated implementations of interfaces, and generally allowing interfaces to be specified at a higher level of abstraction.

However, this separation has not worked out well in practice. For every user interface, it is important to control the low-level pragmatics of how the interactions look and feel, which these UIMSs tried to isolate from the designer. Furthermore, the standardization of the user interface elements in the late 1980's on the desktop paradigm made the need for abstractions from the input devices mostly unnecessary. Thus, UIMSs fell victim to the moving target problem.

2.3.2. Formal Language Based Tools

A number of the early approaches to building a UIMS used techniques borrowed from formal languages or compilers. For example, many systems were based on state transition diagrams (e.g., [13]) and parsers for context free grammars (e.g., [28]). Initially these approaches looked very promising. However, these techniques did not catch on for several important reasons that can serve as important lessons for future tools.

The use of formal language techniques was driven in part by an early emphasis in UIMS work on the task of *dialog management*. At the time that these early user interface management systems were being conceived, the dominant user interface style was based on a conversational metaphor (in which the system and user are seen as conversing about the objects of interest). In this setting, dialog management takes on a very central role, and in fact a number of formal language based systems did a very good job of supporting that central task. Unfortunately, just as these early systems were being developed, the direct manipulation style of interface [35] was quickly coming to prominence. In direct manipulation interfaces, the role of dialog management is greatly reduced because structuring of the dialog by the system is typically detrimental to the concept of directness. As a result, these early user interface tools quickly became ones that had done a very good job of solving a problem that no longer mattered, thus falling victim to the moving target problem.

There are other problems with this class of tools. In these systems it is very easy to express sequencing (and hard to express unordered operations). As a result, they tend to lead the programmer to create interfaces with rigid sequences of required actions. However, from a direct manipulation point of view, required sequences are almost always undesirable. Thus, the *path of least resistance* of these tools is detrimental to good user interface design. Another reason that some systems did not catch on is that had a *high threshold* for using them because they required programmers to learn a new special purpose programming language (in addition to their primary implementation language such as Pascal, C, or C++). Even though the dramatically improved power of the tools seemed to justify allowing it to be difficult to learn, many potential users of these systems did not adopt them simply because they never got past initial difficulties.

2.3.3. Constraints

Many research systems have explored the use of *constraints*, which are relationships that are declared once and then maintained automatically by the system, for implementing several different aspects of a user interface. Examples include Sketchpad [37], ThingLab [3], HIGGENS [11], Garnet [22], Amulet [23], and subArctic [12]. With constraints, the designer can specify, for example, that a line must stay attached to a rectangle, or that a scroll-bar must stay at the right of a window. Once these relationships have been declared, a constraint solving system responds to changes anywhere in the system by updating any values needed to maintain the declared constraints.

Constraint systems offer simple, declarative specifications for a capability useful in implementing several different aspects of an interface. Further, a range of efficient algorithms has been developed for their implementation. However, constraint systems have yet to be widely adopted beyond research systems. One of the central reasons for this is that programmers do not like that constraint solvers are sometimes *unpredictable*. Once a set of constraints is set up, it is the job of the solver to find a solution – and if there are multiple solutions, the solver may find one that the user did not expect. If there is a bug in a constraint method, it can be difficult to find. Furthermore, the declarative nature of constraints is often difficult to master for people used to programming in imperative languages — it requires them to think differently about their problems, which also contributes to having a *high threshold.*

One area of user interface design for which constraints do seem successful is the *layout* of graphical elements. Systems such as NeXTStep provided a limited form of constraints using the metaphor of "springs and struts" (for stretchy or rigid constraints) that could be used to control layout. This and other metaphors have found wider acceptance because they provided a limited form of constraints in a way that was easier to learn and more predictable for programmers.

2.3.4. Model-based and Automatic Techniques

Another thread of user interface research that has shown significant promise, but has not found wide acceptance, is the investigation of automatic techniques for generating interfaces. The goal of this work is to allow the designer to specify interfaces at a very high level, with the details of the implementation to be provided by the system. Motivations for this include the hope that programmers without user interface design experience could just implement the functionality and rely on these systems to create high-quality user interfaces. The systems might allow user interfaces to be created with less effort (since parts would be generated automatically). Further, there is the promise of significant additional benefits such as automatic portability across multiple types of devices, and automatic generation of help for the application.

Examples include *model-based* systems, such as Mike [27], Jade [40], UIDE [36], and ITS [43]. These systems used techniques such as heuristic rules to automatically select interactive components, layouts, and other details of the interface.

Automatic and model-based techniques have suffered from the problems of unpredictability. Programmers must also learn a new language for specifying the models, which raises the

threshold of use. In addition, model-based systems have a low ceiling. Because automatically generating interfaces is a very difficult task, automatic and model-based systems have each placed significant limitations on the kinds of interfaces they can produce. A related problem is that the generated user interfaces were generally not as good as those that could be created with conventional programming techniques. Finally, an important motivation for model-based techniques was to provide independence of the input-output specification from the details of the specific user interface characteristics, but with the standardization of the user interface elements, this separation became less important. As we will discuss later, a new requirement for device independence is emerging, which may raise the need for model-based or related techniques in the future.

3. Future Prospects and Visions

The next sections discuss some of our predictions and observations for the future of user interface tools. It is impossible to discuss the tools without discussing the user interface changes that will require the new tools. Therefore, these sections are organized around the new user interface paradigms that we see emerging. We see important implications from computers becoming a commodity, ubiquitous computing, the move to recognition-based and 3D user interfaces, and end-user customization.

3.1. Computers Becoming a Commodity

As Moore's law continues to hold, computers available to the general public have become fast enough to perform anything that researchers' computers can do. The trick that has been used by computer science researchers of buying expensive, high performance computers to investigate what will be available to the public in five or ten years no longer works, since the computers available to the public are often as fast or faster than the researcher's. This may have a profound impact on how and what computer science research is performed. Furthermore, the quantitative change in increased performance makes a qualitative change in the kinds of user interfaces possible. For example, it has now enabled the production of inexpensive palm-size computers, and single chip microprocessors the power of a 68000 that cost only about 30 cents and can be embedded in various devices. Another impact of the high performance is that user interfaces are becoming more *cinematic*, with smooth transitions, animation, sound effects, and many other visual and audio effects.

3.2. Ubiquitous Computing

The idea of ubiquitous computing [42] is that computation will be embedded in many different kinds of devices, on many different scales. Already we are seeing tiny digital pagers and phones with embedded computers and displays, palm-size computers such as the Palm Pilot, notebook-size panel computers, laptops, desktops, and wall-size displays. Furthermore, computing is appearing in more and more devices around the home and office. An important next wave will appear when the devices can all easily communicate with each other, probably using radio wireless technologies like BlueTooth [9]. What are the implications of these technologies on the tools that will be needed? The next sections discuss some ideas.

3.2.1. Varying Input and Output Capabilities

Virtually all of today's interaction techniques have been highly optimized over the last twenty years to work with a fairly large display and a mouse with one to three buttons. Virtually none of the devices envisioned for ubiquitous computing have a mouse, some have no true pointing devices at all, and many have different kinds of displays. The first important issue is how the interaction techniques should change to take these varying input and output hardware into account.

The most obvious impact is that developers will now have to create user interfaces that work with vastly different sizes and characteristics of displays. Whereas screens on desktop machines have only varied from 640x480 to 1280x1024 pixels (a factor of 4) and their screen vary from about 8 inches to about 20 inches in diagonal (a factor of 2.5), in the future, screens will be from cell phone displays (60x80 pixels at about 2 inches in diagonal) to wall size displays (a wall-size display at Stanford is using a screen that is 3796x1436 pixels and 6 feet by 2 feet [44]). These variations are factors of about 625 in resolution and 100 for size. Current techniques have often made implicit assumptions about device characteristics. For example, standard widgets like pull-down menubars cannot generally be used, since they may not fit on small screens and on large screens, they might be too high for some short users[33].

Also the input modalities differ—cell phones have a numeric keypad and voice, palm-size displays have touch-sensitive screens and a stylus but no keyboard, other PDAs have tiny keyboards, and wall-size displays can often be touched or pointed at. Changing from a mouse to a stylus on a touchpad requires different interaction techniques. Some devices will also provide

high-quality speech, gesture and handwriting recognition. Many tools cannot handle a different number of mouse buttons, let alone the change from a mouse to a stylus, and the move to new input modalities such as speech, gestures, eye tracking, and video cameras is completely out of the question for such tools.

Thus, the same user interfaces obviously cannot be used on all platforms. If a developer is creating an application that should run on a variety of platforms, it becomes much more difficult to hand-design the screens for each kind of display. There is not yet even standardization within devices of the same class. For example, different kinds of PDAs have quite different display characteristics and input methods.

This may encourage a return to the study of some techniques for device-independent user interface specification, so that developers can describe the input and output needs of their applications, vendors can describe the input and output capabilities of their devices, and users can specify their preferences. Then, the system might choose appropriate interaction techniques taking all of these into account.

3.2.2. Tools to Rapidly Prototype *Devices*, not just Software

An important consideration for the new devices is unlike desktop machines that all have the same input and output capabilities (mouse, keyboard, and color screen), there will be a great variety of shapes, sizes, and input-output designs. Much of the user interface will be in built into the hardware itself, such as the physical buttons and switches. Therefore, the designer will have to take into account not only the software, but also the physical properties of the devices and their capabilities. Thus we will need tools that support the rapid design and prototyping of the hardware. It will not be sufficient to use screen-based techniques for prototyping the hardware since the pragmatics and usability cannot be evaluated solely from a simulation on a screen.

3.2.3. Tools for Coordinating Multiple, Distributed Communicating Devices

With the rise of the use of the Internet and World-Wide Web, the computer is becoming a technology for *communication* more than for *computation.* This trend will significantly increase with the rise of ubiquitous computing. Many of the devices (cell-phones, pagers) are already designed for communication from one person to another, and the computation is bringing added functionality and integration. As room-area wireless networks increasingly enable multiple

devices to communicate, it will be more important that computers no longer are used as islands of computation and storage, but rather as part of an integrated, multi-machine, multi-person environment. Furthermore, as people increasingly distribute their own computation across devices in their office, their home, their car, and their pocket, there will be an increasing need for people to communicate with *themselves* on different devices. Supporting collaboration with other people will be more complicated than with current multi-user applications such as GroupKit [34] or Microsoft NetMeeting, since it can no longer be assumed that all the people have comparable computers. Instead, some users might be on a cell-phone from a car while others are using PDAs on the train, or wall-size displays at home. Sometimes, users may have *multiple* devices in use at the same time. For example, the Pebbles project is investigating how PDAs can be used effectively at the same time as PCs [25]. Devices and people will have different levels of connectedness at different times, from disconnected, to connected with slow links, to being on a high-speed network. Decisions about what information is to be sent and shown to the user should be based on the importance and timeliness of the information and the capabilities of the current connection and devices. Furthermore, the user's interaction will need to take into account information from the Internet and information generated by sensors and people from all over the world. The status from the environment should also be taken into account (some call these "context-aware" user interfaces [5]).

The implications of all this on tools are profound. It will be important for tools to provide facilities to manage data sharing and synchronization, especially since it must be assumed that the data will go out of date while disconnected and need to be updated when reconnected. Since data will be updated automatically by the environment and by other people, techniques for notification and awareness will be important. Furthermore, the tools will be needed to help present the information in an appropriate way for whatever device is being used. If the user has multiple devices in use at the same time, then the content and input might be spread over all the active devices. All applications running in this future environment should be cognizant of multiple people and sharing, so that group work can be easily accommodated. This implies that security issues will become increasingly important, and better user interfaces for monitoring, specifying and testing security settings will be crucial.

Since these capabilities should be available to *all* applications, it will be important for them to be provided at a low level, which suggests that the supporting capabilities be included in the underlying operating system or toolkits.

3.3. Recognition-Based User Interfaces

Whereas most of today's tools provide good support for widgets such as menus and dialog boxes that use a keyboard and a mouse, these will be a much smaller proportion of the interfaces of the future. We expect to see substantially more use of techniques such as gestures, handwriting, and speech input and output. These are called *recognition-based* because they require software to interpret the input stream from the user to identify the content.

These new modalities will fundamentally change how interfaces are developed. For example, to create speech user interfaces today requires learning about vocabularies, parsers, and Hidden-Markov-Models. Tools will be needed that hide all of this complexity and provide an easy-to-use interface to programmers.

Recognition-based interfaces have a number of fundamental differences from today's interfaces. The primary difference is that the input is uncertain—the recognizer can make errors interpreting the input. Therefore, interfaces must contain feedback facilities to allow the user to monitor and correct the interpretation. Furthermore, interpreting the input often requires deep knowledge about the context of the application. For example, to interpret "move the red truck to here," the system needs access to the objects in the application (to find the red truck) along with timing information about pointing gestures (to find "here").

There are significant implications for tool design. First, a conventional event-based model may no longer work, since recognition systems need to provide input continuously, rather than just in discrete events when they are finished. For example, when encircling objects with a gesture and continuing to move, the objects should start moving immediately when the boundary is closed, rather than waiting for the end of the gesture. Similarly, in speech, the interpretation of the speech should begin immediately, and feedback should be provided as soon as possible. Temporal coordination with other modalities (such as pointing) requires that the continuous nature of the input be preserved.

Another issue is the separation of knowledge about the application's contents. Today's user interface tools work without any deep knowledge about what the application is doing—they only need to know the surface widgets and appearance. To support the new modalities, this will no longer work. We believe an appropriate architecture is for the tools to have access to the main application data structures and internals. Otherwise, each application will need to deal with its own speech interpretation, which is undesirable. Increasingly, future user interfaces will be built

around standardized data structures or "knowledge bases" to make these facilities available without requiring each application to rebuild them. The current trend towards "reflection" and the "open data model" is a step in this direction, but this is a deep unsolved problem in artificial intelligence systems in general, so we should not expect any solutions in the near term.

3.4. Three-Dimensional Technologies

Another trend is the migration from two-dimensional presentation space (or a 2½ dimensional space, in the case of overlapping windows) to three-dimensional space. Providing tools for 3D is a very difficult problem. Researchers are still at the stage where they are developing new interaction techniques, gestures and metaphors for 3D interfaces. We predict the need to settle on a set of 3D interaction techniques and 3D widgets before high-level tools for interactive behaviors will be possible. Some research systems, such as Alice [32] are exploring how to hide the mathematics, which will be increasingly important in the tools of the future. Providing support in a 3D toolkit in 1999 suffers from the problems that 2D toolkits had 15 years ago. First, the need for performance causes the underlying implementation details to be visible. Second, we are not sure what applications will be developed with this new style. In this regard, 3D interfaces are probably in worse shape than 2D interface were, because 2D interfaces were able to adopt many paper conventions, for example, in desktop publishing. Until a breakthrough occurs in our understanding of what kinds of applications 3D will be useful for, it will be extremely difficult for toolkits to know what abstractions to provide.

3.5. End-User Programming, Customization, and Scripting

One of the most successful computer programs of all time is the spreadsheet, and the primary reason for its success is that end users can program (by writing formulas and macros).[1] However, *end user programming* is rare in other applications, and where it exists, usually requires learning conventional programming. An important reason that the World-Wide Web has been so

[1] It is interesting to note that spreadsheet formulas are a form of constraint and are tremendously successful, whereas constraints for programmers have not been successful. We conjecture that this is because constraints fit with the business person's mental model who just needs a computation and an answer, but not with the programmer's who needs more control over performance and methods. A key for any tool is to fit with the users' model.

successful is that everyone can create his or her own pages. However, for "active" pages that use forms, animations, or computation, again programming is required, usually by a professional programmer using a programming language like PERL or Java. "Productivity applications" are becoming increasingly programmable (e.g., by writing Visual Basic scripts for Microsoft Word), but only to those with some affinity for programming. Other systems use what are called *scripting languages*, such as Lingo for MacroMedia Director, but these are often fairly conventional programming languages.

End-user programming will be increasingly important in the future. No matter how successful interface designers are, systems will still need to be customized to the needs of particular users. Although there will likely be generic structures that can be shared, such systems and agents will always need to be tailored to meet personal requirements. Better interfaces and understandings of end-user programming are needed. Furthermore, these capabilities should not be built into individual applications as is done today, since this means that the user may need to learn a different programming technique for each application. Instead, the facilities should be provided at the system level, and therefore should be part of the underlying toolkit.

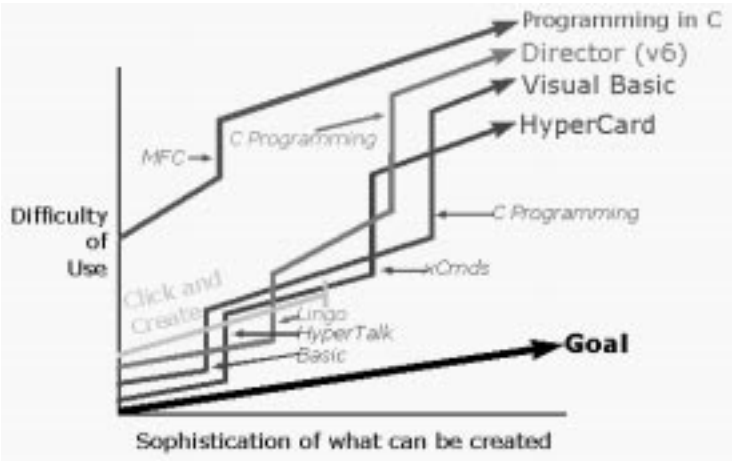

Figure 1: The intent of this graph is to try to give a feel for how hard it is to use the tools to create things of different levels of sophistication. For example, with C, it is quite hard to get started, so the Y intercept (threshold) is high up. The vertical walls are where the designer needs to stop and learn something entirely new. For C, the wall is where the user needs to learn the Microsoft Foundation Classes (MFC) to do graphics. With Visual Basic, it is easier to get started, so the Y intercept is lower, but Visual Basic has two walls—one when you have to learn the Basic programming language, and another when you have to learn C. Click and Create is a menu based tool from Corel, and its line stops (so it has a low ceiling) because it does not have an extension language, and you can only do what is available from the menus and dialog boxes.

The important research problem with scripting and customization is that the threshold is still too high—it is too difficult to learn how to program. The threshold and ceiling issue is illustrated by the research on "Gentle Slope Systems" [24] that are systems where for each incremental increase in the level of customizability, the user only needs to learn an incremental amount. This is contrasted with most systems that have "walls" where the user must stop and learn many new concepts and techniques to make further progress (see Figure 1).

3.6. Further Issues for Future Tools

In addition to the issues discussed above, we see some additional trends in the near future that will contradict assumptions built into today's tools.

- *Skill and dexterity levels of users*. Most current interfaces assume an average level of dexterity and manual skill on the part of the user. However, based on current demographics we know that as time passes there will be many more older adults using interactive systems (not to mention younger, but disabled persons). With aging comes an inevitable decline in motor, memory, and perceptual skills. This may require redesign of many accepted interactive techniques and the tools that support them.

- *Non-overlapping layout or rectangular and opaque interactive components*. Early toolkits (such as the Macintosh toolbox) assumed that interactive widgets such as buttons and text fields would not overlap. Other toolkits (most notably those coming with the X-windows system and systems that were heavily influenced by it such as the Java AWT) instead assumed that overlap was possible, but that all components were rectangular and opaque. These assumptions worked well for early GUI interfaces. However, they have more recently become rather limiting and they typically preclude techniques such as translucency and Magic Lens interactions [2] that show great promise and are now technically feasible.

- *Using fixed libraries of interactive components*. Most toolkits have long assumed that a fixed library of interactive components covered the vast majority of interfaces that were to be built. As a result, they have placed an emphasis on making components from the library easy to employ, while generally neglecting the issue of making it easy to create new interactive components. The implicit or explicit assumptions made by a system, significantly limit the kinds things that can be (easily) accomplished with it.

- *Interactive setting*. Much of current user interface design knowledge (and hence the supporting tools) also implicitly makes assumptions about the setting in which a user acts. For example, most interfaces assume that the user is sitting, has two hands available, and can look at the interface while operating it. However, some of these properties do not hold in many situations in the world where computational support could be valuable (e.g., while driving, or lying under a machine being repaired).

- *Requiring the user's full attention.* Almost all current interfaces assume that they have the user's full attention—they typically do very little unless the user pays attention to them and acts upon them. However, with ubiquitous computing, the number of devices for each user is multiplying. If each of these demands a small piece of the user's attention, the aggregate result may be quite unpleasant. As a result, there is a clear need for new interaction paradigms that minimize the amount of attention demanded. With these interfaces will be a need for tools that explicitly consider human attention as part of the design criteria. This may require a strong integration of cognitive science knowledge that to date has not been directly employed in user interface tools.

- *Support for Evaluation*. Today's tools focus on the design and implementation of user interfaces. However, achieving the general goal of supporting rapid iteration of designs requires rapid evaluation, as well as rapid implementation. Unfortunately, few tools have provided explicit support for evaluation. This is partially because tools that have tried, such as MIKE [29] discovered that there are very few metrics that can be applied by computers. A new generation of tools is trying to evaluate how people will interact with interfaces by automatically creating cognitive models from high-level descriptions of the user interface [17], but this work is very preliminary and much more research is needed.

- *Creating usable interfaces*: Going even further, tools might enforce or at least encourage user interfaces that were highly usable, rather than today's stance that tools should be neutral and leave the design mostly to the human designer.

4. Operating System Issues

What is the "Operating System" in today's systems? Does the Window Manager count as part of the operating system? What about the toolkit that provides access to the standard drawing capabilities and widgets (e.g., Win32 and the Microsoft Foundation Classes)? Today, there is even a legal debate about whether the web browser can be considered part of the operating

system. Furthermore, the distinction between the design-time tools (used for creating the user interface) and the run-time tools (to execute the interface) blur as more services are used by both. It seems clear that the facilities that are provided by the "operating system" will continue to expand, and many of the features and tools that this article discusses will be included in what is called the operating system, for good or ill.

Some of the capabilities *must* be provided at a very low level. For instance, access to information about the input, output and communication capabilities of devices must be provided to the application software, so it can make intelligent choices about the user interface. For instance, Windows CE for palm-size devices seems to make it impossible to find out what kinds of hardware buttons are available on the PDA. Another example is that none of today's networking interfaces make it possible for an application to decide how fast a connection is available. In fact, it is usually impossible to find out if you are connected *at all*, so applications freeze up for minutes waiting to see if the network might respond, when the operating system could easily tell that the machine is disconnected. This is unacceptable. Interfaces of the future will need to make intelligent choices based on knowledge about the current capabilities of the device and its connections to the environment.

Other capabilities might be provided on top of the operating system, but alternatively might be part of it. For applications to run on a variety of devices, a portable infrastructure must be provided, possibly like the Java run-time library. There will also need to be high-level protocols for accessing information on distributed files. End-user programming support will be needed across multiple applications, so this should be provided in a universally applicable form. Ideally, all of these required capabilities will be available, but not bundled with the operating system. More progress will be made if multiple vendors and research groups can compete to provide the best possible tools.

5. Conclusions

Generally, research and innovation in tools trail innovation in user interface design, since it only makes sense to develop tools when you know for what kinds of interfaces you are building tools. Given the consolidation of the user interface on the desktop metaphor in the last 15 years, it is not surprising that tools have matured to the point where commercial tools have fairly successfully covered the important aspects of user interface construction. It is clear that the research on user interface software tools has had enormous impact on the process of software development. However, we believe that user interface design is poised for a radical change in the near future,

primarily brought on by the rise of ubiquitous computing, recognition-based user interfaces, 3D and other technologies. Therefore, we expect to see a resurgence of interest and research on user interface software tools in order to support the new user interface styles.

We believe that these new tools will be organized around providing a rich context of information about the user, the devices, and the application's state, rather than around events. This will enable end-user programming, recognition-based user interfaces and the data sharing needed for ubiquitous computing. It will be important to have replaceable user interfaces for the same applications to provide different user interfaces on different devices for ubiquitous computing, and to support customization. This will include having a procedural interface to everything that can be performed by the user. We recommend that tools aim to have a low threshold so they are easy to use, but still provide a high ceiling. Predictability seems to be very important to programmers, and should not be sacrificed to make the tools "smarter." All of these challenges provide new opportunities and research problems for the user interface tools of the future.

Acknowledgements

The authors wish to thank Brad Vander Zanden, Dan Olsen, Rob Miller, Rich McDaniel and the referees for their help with this article.

References

1. Apple Computer Inc., *Inside Macintosh.* 1985, Addison-Wesley.

2. Bier, E.A., *et al.* "Toolglass and Magic Lenses: The See-Through Interface," in *Proceedings SIGGRAPH'93: Computer Graphics.* 1993. **25**. pp. 73-80.

3. Borning, A., "The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory." *ACM Transactions on Programming Languages and Systems*, 1981. **3**(4): pp. 353-387.

4. Buxton, W., *et al.* "Towards a Comprehensive User Interface Management System," in *Proceedings SIGGRAPH'83: Computer Graphics.* 1983. Detroit, Mich: **17**. pp. 35-42.

5. Dey, A.K., Abowd, G.D., and Wood, A. "Cyberdesk: A Framework for Providing Self-Integrating Context-Aware Services," in *Proceedings of the 1998 International Conference on Intelligent User Interfaces (IUI'98).* 1998. pp. 47-54.

6. Engelbart, D. and English, W., "A Research Center for Augmenting Human Intellect." *Reprinted in ACM SIGGRAPH Video Review, 1994.*, 1968. **106**

7. English, W.K., Engelbart, D.C., and Berman, M.L., "Display Selection Techniques for Text Manipulation." *IEEE Transactions on Human Factors in Electronics*, 1967. **HFE-8**(1)

8. Green, M. "The University of Alberta User Interface Management System," in *Proceedings SIGGRAPH'85: Computer Graphics.* 1985. San Francisco, CA: **19**. pp. 205-213.

9. Haartsen, J., *et al.*, "Bluetooth: Vision, Goals, and Architecture." *ACM Mobile Computing and Communications Review*, 1998. **2**(4): pp. 38-45. Oct. www.bluetooth.com.

10. Henderson Jr, D.A. "The Trillium User Interface Design Environment," in *Proceedings SIGCHI'86: Human Factors in Computing Systems.* 1986. Boston, MA: pp. 221-227.

11. Hudson, S. and King, R., "A Generator of Direct Manipulation Office Systems." *ACM Trans. on Office Infomation Systems*, 1986. **4**(2): pp. 132-163.

12. Hudson, S.E. and Smith, I. "Ultra-Lightweight Constraints," in *Proceedings UIST'96: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1996. Seattle, WA: pp. 147-155. http://www.cc.gatech.edu/gvu/ui/sub_arctic/.

13. Jacob, R.J.K., "A Specification Language for Direct Manipulation Interfaces." *ACM Transactions on Graphics*, 1986. **5**(4): pp. 283-317.

14. JavaSoft, *JavaBeans.* Sun Microsystems, JavaBeans V1.0, December 4, 1996. http://java.sun.com/beans.

15. Kasik, D.J. "A User Interface Management System," in *Proceedings SIGGRAPH'82: Computer Graphics, 16(3).* 1982. Boston, MA: **16**. pp. 99-106.

16. Kay, A., *The Reactive Engine.* PhD Thesis, Electrical Engineering and Computer Science University of Utah, 1969, 327.

17. Kieras, D.E., *et al.* "GLEAN: A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs," in *Proceedings UIST'95: Eighth Annual Symposium on User Interface Software and Technology.* 1995. pp. 91-100.

18. Koved, L. and Shneiderman, B., "Embedded menus: Selecting items in context." *Communications of the ACM*, 1986. **4**(29): pp. 312-318.

19. Lutz, M., *Programming Python.* 1996, O'Reilly & Associates. ISBN: 1-56592-197-6. http://www.python.org/.

20. Myers, B.A., "User Interface Software Tools." *ACM Transactions on Computer Human Interaction*, 1995. **2**(1): pp. 64-103.

21. Myers, B.A., "A Brief History of Human Computer Interaction Technology." *ACM interactions*, 1998. **5**(2): pp. 44-54. March.

22. Myers, B.A., *et al.*, "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces." *IEEE Computer*, 1990. **23**(11): pp. 71-85.

23. Myers, B.A., *et al.*, "The Amulet Environment: New Models for Effective User Interface Software Development." *IEEE Transactions on Software Engineering*, 1997. **23**(6): pp. 347-365. June.

24. Myers, B.A., Smith, D.C., and Horn, B. "Report of the `End-User Programming' Working Group," in *Languages for Developing User Interfaces.* 1992. Boston, MA: Jones and Bartlett. pp. 343-366.

25. Myers, B.A., Stiel, H., and Gargiulo, R. "Collaboration Using Multiple PDAs Connected to a PC," in *Proceedings CSCW'98: ACM Conference on Computer-Supported Cooperative Work.* 1998. Seattle, WA: pp. 285-294.

26. Nielsen, J., *Multimedia and Hypertext: the Internet and Beyond.* 1995, Boston: Academic Press Professional. 480.

27. Olsen Jr., D.R., "Mike: The Menu Interaction Kontrol Environment." *ACM Transactions on Graphics*, 1986. **5**(4): pp. 318-344.

28. Olsen Jr., D.R. and Dempsey, E.P. "Syngraph: A Graphical User Interface Generator," in *Proceedings SIGGRAPH'83: Computer Graphics.* 1983. Detroit, MI: **17**. pp. 43-50.

29. Olsen Jr., D.R. and Halversen, B.W. "Interface Usage Measurements in a User Interface Management System," in *Proceedings UIST'88: ACM SIGGRAPH Symposium on User Interface Software and Technology.* 1988. Banff, Alberta, Canada: pp. 102-108.

30. Ousterhout, J.K. "An X11 Toolkit Based on the Tcl Language," in *Winter USENIX Technical Conference.* 1991. pp. 105-115.

31. Palay, A.J., *et al.* "The Andrew Toolkit - An Overview," in *Proceedings Winter Usenix Technical Conference.* 1988. Dallas, Tex: pp. 9-21.

32. Pausch, R., *et al.*, "Alice: A Rapid Prototyping System for 3D Graphics." *IEEE Computer Graphics and Applications*, 1995. **15**(3): pp. 8-11. May.

33. Pier, K. and Landay, J., *Issues for Location-Independent Interfaces.* Xerox PARC, Technical Report ISTL92-4, December, 1992. Palo Alto, CA. http://www.cs.berkeley.edu/~landay/research/publications/LII.ps.

34. Roseman, M. and Greenberg, S., "Building Real Time Groupware with GroupKit, A Groupware Toolkit." *ACM Transactions on Computer Human Interaction*, 1996. **3**(1): pp. 66-106.

35. Shneiderman, B., "Direct Manipulation: A Step Beyond Programming Languages." *IEEE Computer*, 1983. **16**(8): pp. 57-69. Aug.

36. Sukaviriya, P., Foley, J.D., and Griffith, T. "A Second Generation User Interface Design Environment: The Model and The Runtime Architecture," in *Proceedings INTERCHI'93: Human Factors in Computing Systems.* 1993. Amsterdam, The Netherlands: pp. 375-382.

37. Sutherland, I.E. "SketchPad: A Man-Machine Graphical Communication System," in *AFIPS Spring Joint Computer Conference.* 1963. **23**. pp. 329-346.

38. Teitelman, W., "A Display Oriented Programmer's Assistant." *International Journal of Man-Machine Studies*, 1979. **11**(2): pp. 157-187. Also Xerox PARC Technical Report CSL-77-3, Palo Alto, CA, March 8, 1977.

39. Tesler, L., "The Smalltalk Environment." *Byte Magazine*, 1981. **6**(8): pp. 90-147.

40. Vander Zanden, B. and Myers, B.A. "Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces," in *Proceedings SIGCHI'90: Human Factors in Computing Systems.* 1990. Seattle, WA: pp. 27-34.

41. Wall, L. and Schwartz, R.L., *Programming perl.* 1992, Sebastopol, CA: O'Reilly & Associates.

42. Weiser, M., "Some Computer Science Issues in Ubiquitous Computing." *CACM*, 1993. **36**(7): pp. 74-83. July.

43. Wiecha, C., *et al.*, "ITS: A Tool for Rapidly Developing Interactive Applications." *ACM Transactions on Information Systems*, 1990. **8**(3): pp. 204-236.

44. Winograd, T., *A Human-Centered Interaction Architecture.* Working paper for the Interactive Workspaces Project, Stanford University, 1998. http://graphics.stanford.EDU/projects/iwork/.