Lens: A System for Visual Interpretation of Graphical User Interfaces

Kevin Gibbs, Terry Winograd, and Neil Scott
Department of Computer Science
Stanford University
Stanford, CA 94305
Email: kgibbs@cs.stanford.edu

ABSTRACT

Lens enables natural segmentation of graphical user interfaces. Lens is a system that automatically discovers the structure of user interfaces by discovering the hierarchically repeating on-screen elements which structure interfaces. The Lens system was built to facilitate access to modern GUI based computer systems for blind computer users by extracting useful interface information from the visual representation of the computer screen. This paper documents the exploration and design leading to the Lens system, as well as describing the algorithms and our current implementation of the Lens system. Our evaluation of the implemented Lens system in real-world situations shows that a system with no prior information about a specific interface implementation can discover useful and usable interface control structure.

INTRODUCTION

In the two decades since their inception, Graphical User Interfaces have gradually become the *de facto* method of accessing nearly all computer systems. To most users, the thought of doing common computing tasks, such as editing a word processing document or browsing the Internet, from a command line interface, now seems as impossible as it does unlikely. Most personal computing tasks today are so dependant upon the common denominator of a GUI, a graphical representation of data and controls, and a visual display, that the thought of trying to get anything done on a computer without the use of them seems roughly equivalent to trying to navigate through an unfamiliar house with your eyes closed—and your hands tied behind your back.

Which is exactly the computer access problem facing the world's burgeoning population of over 38 million blind people [3]. Although significant efforts have been in the areas of enabling limited applications (such as text processors like EMACS [1] and screen-reading software for the World-Wide Web [2]) for the blind, few if any successful efforts have been made in the area of providing access to the generalized GUI systems that the rest of us use on a day-to-day basis. Admittedly, this problem seems quite intractable. How could a blind user ever be expected to make use of a GUI system, which is at its very core, dependent upon a visual metaphor?

Enter VisualTAS, or the Visual Total Access System. VisualTAS is a system that is under development by our group at Project Archimedes [4] that attempts to make the GUI accessible to blind users (and other physically disabled users) by breaking the screen image down into a series of objects that can be represented to the blind user by a variety of aural, sonic, and physical interfaces. The VisualTAS has two components: a *VisualTAP* (VTAP) that captures screen images and extracts a variety of categories of information from these images, and a *GUI Accessor* that presents each type of information to the blind user in the most appropriate form available [5]. A selection of GUI Accessors will then interact with the user, based on speech, sound, touch, and haptic (force feedback) representations of this information. One example of such a system is The Moose, a haptic interface for feeling visual graphical interface information developed in our lab, which is typical of the type of GUI Accessor that a blind user would interact with [12].

Figure 1. The Moose, a haptic interface [from 12].

The VisualTAP, or Visual Total Access Port, which is the portion of the VisualTAS responsible for extracting data and understanding control information from the GUI, has its work cut out for itself. As the sole source of input available to the VisualTAS, the VTAP must extract enough information from the GUI system to provide a usable system to the blind user. To follow our earlier metaphor, the VTAP is what "unties" the blind user's hand, and allows them to navigate through a visual interface by interacting with the extracted information through a variety of other sensory means.

The overall task of the VTAP, then, is to extract all useful forms of information possible from the GUI system. This task clearly is composed of many subtasks, some of which are more straightforward than others. For example, text and characters found on the screen should be extracted and output the eventual use of speech-synthesis components. Icons onscreen should be discovered and have visual

features extracted from them, so that they can be somehow presented in a non-visual form. Yet these tasks, while quite interesting and significantly difficult, do not broach what is perhaps the single most complex and fundamental problem posed to a system like the VTAP—finding a way to extract and in some way interpret the user interface presented to the user by a GUI. We needed to find some system that could do this, in order to make the VisualTAS a reality.

This problem of gleaning usable interface information from a GUI system appears to be a fascinating and exciting problem for investigation. However, the programmer attempting to actually implement a suitable system for extracting GUI information for a project like the VisualTAS has many initial hurdles to face.

1. The visual descriptions of GUIs vary widely from one operating system and revision to another. Although some visual metaphors are relatively standard (the concept of a "button," a "scroll bar," a "window," ad etc.), the actual visual implementations of these metaphors vary widely. Every OS has it's own "look and feel," and this look and feel even changes amongst mainstream OS revisions, such as Microsoft Windows 2000 and XP, and Apple Macintosh OS 9 and X. The frightening amount of variety found amongst the bevy of UNIX window managers and interface themes can be left to the reader's personal nightmares.

Figure 2. Variety in window controls.

2. The function and location of user interface elements within a system varies in a similar manner. For example, consider how many different ways GUI systems deal with the problem of a window close box. Popular flavors of Microsoft Windows, the Apple Macintosh OS, and UNIX, all use widely different window controls, which vary in not only visual representation ("look and feel"), but also in spatial location.
3. Interactions in software between programs and the visual GUI are very intricate and highly brittle. Operating systems, which ultimate offer GUI services to programs, are amongst the most complex and rapidly changing pieces of software on computers today. Furthermore, the interactions between programs, the OS, and the visual elements that are finally painted on the screen are ill defined, highly implementation dependent, and generally not meant for outside programs to intercept. Any software that actually runs on the host machine and attempts to intercept information from these interactions will likely be similarly highly complex, error prone, and require constant support and revision.
4. Costs of development, of any form, are high. Projects like the VisualTAS are still relatively young and have limited resources and manpower available. Even with available resources, it is quite difficult to find systems programmers of the caliber necessary to solve the highly complex problems that developing a GUI interpretation system like the VTAP requires. Moreover, the developers that are available will likely tend to be spread quite thin amongst the numerous other projects involved in creating a VisualTAS system. Hence, the developer costs of maintaining of any developed system cannot be ignored, as a system that requires constant programmatic upkeep may well become derelict shortly.

MOTIVATION: OUR FRUSTRATING EXPERIENCES WITH CURRENT SOLUTIONS

In our search for a viable solution for extracting GUI information in the VTAP, our experiences all pointed back to the problems outlined above. The various systems that we found that attempted to solve this general problem, that is, of extracting useful control information from a GUI, all suffered from the above problems in various forms.

API "shims." We first investigated systems that attempt to track and interpret all the internal software interactions that ultimately result in the onscreen image. Systems like this typically insert a software "shunt" or wrapper around GUI libraries like Microsoft's MFC, which is responsible for drawing most of the user interface controls in a Microsoft Windows GUI. However, we found that these systems were, as we predicted above, very intricate and very brittle. Every time any major or minor change to the OS or GUI libraries occurred, the software would stop working altogether, and the system would become useless without a great deal of additional development and fixing. Moreover, even when these systems work, they are (obviously) very OS and revision dependent, as an MFC API wrapper provides no use on say, an MacOS system, which uses completely different set of APIs and libraries for its GUI, or on the new version of Windows, which alters the underlying API the system is trying to wrap. Thus, we did not find these systems to be viable due to their extreme brittleness, system dependence, and high development costs, though their accurate and complete information otherwise seemed promising.

Scripting systems. The next, slightly different approach we investigated were OS-level scripting systems. These systems are usually part of the operating system itself, such as AppleScript, or run as a 3rd party application, like the classic QuicKeys. These systems did present a fair number of commands and options for manipulating the GUI, and they also usually have a fair model of what is available on screen for use, such as the available windows, and open programs. However, we found that these systems usually suffered from the same brittleness (albeit, to a lesser extent)

of the "shim" systems, in that the features and options available and implementations were constantly changing amongst revisions and particular programs. These scripting systems were also very dependent on per-program support, which varied widely and was usually limited. Ultimately, and most importantly, these systems seemed to lack the level of detail necessary to actually use the computer in a day-to-day fashion.

SegMan. One system we discovered that seemed to hold promise for our project was the SegMan system, from Robert St. Amant's User Interface Softbot project at NC State [6]. SegMan takes a novel approach to accessing the user interface of the computer [8]. SegMan attempts to visually "see" what is on the screen, through interacting with the screen bitmap, rather than by interacting with system APIs. SegMan accomplishes this through a small .dll for Microsoft Windows that accesses the pixel representation of the screen. From the pixel representation, SegMan forms a series of "pixel groups," which are non-overlapping groups of contiguous pixels that share the same color. SegMan represents these pixel clusters compactly using a novel pixel-neighboring representation [10], and then provides these pixel groups to a Common Lisp development system. This Common Lisp system then contains a series of "pixel-patterns," which are a series of logically conjunctive statements about pixel group properties in written Lisp, which perform the task of recognizing on-screen widgets and controls. A particular "pixel-pattern," for instance, might say that pixel-groups of a certain color and shape that are adjoined by certain other specific pixel groups represent a "Button" onscreen.

7	6	5
0		4
1	2	3

Figure 3. Pixel neighboring. The gray squares are the neighbors of the center.

This programmatic based recognition then continues for many more levels, with much further specific code to model the interface in question. The system eventually produces a model of the screen state that includes what windows are on screen, what controls are available, what their states are, ad etc.

We were initially quite excited by the SegMan system, since it avoids some of the key problems we discussed above. Since SegMan looks only at pixels on screen, it avoids the intricate interactions going on between the OS and particular programs on screen. Also, since the widgets that SegMan discovers are specified by SegMan itself, it can be programmed to discover and make available things that other systems may pass over, like non-standard widgets and controls not found through other interactions. Finally, SegMan seemed very promising to us, because unlike the other systems, it is based upon a *visual* metaphor—it attempts to visually understand the screen, which is exactly the problem we are trying to solve for blind users.

However, we found through further investigation that SegMan, too, was not suitable for our needs. Though it managed to avoid the need to access system APIs, SegMan was very dependent on a full and complete programmatic description of the operating system in question. Every type of widget, button, and desired element had to be programmatically specified in Common Lisp, separately for every OS desired, and we found this to require a very high development time commitment. We felt a better system was possible, one that would be more automatic and less dependent on a programmer carefully investigating an interface and codifying all widgets and controls. Moreover, SegMan's internal representation, while safe from the tempestuous mood swings of the system APIs, was brittle on another level—it was highly dependent on the exact visual description on screen. Change the look and feel of a button slightly, or even change the user-interface color scheme, and SegMan would fail to recognize anything on screen. Since we felt that the visual descriptions of popular user interfaces were changing rapidly, and would continue to do so as time went on, we felt we could find a better solution that would avoid this combination of high development commitment and upkeep for an ultimately brittle interface specification.

Hence, our initial experiences with existing solutions were frustrating. Some systems were more promising than others, particularly the visual aspects of the SegMan system, but none provided a satisfactory solution to our problems.

RESEARCH AGENDA: THE LENS CONCEPT

As a consequence of these problems, we wanted to create a system that enables automatic, *natural* segmentation of the screen space in a visual manner. We set specific goals to try to overcome the problems we had uncovered through our exploration of other tools. Our goals were to create a system that satisfied the following requirements:

- **Automatic.** We wanted a system that would be as *automatic* possible. Rather than depending on large amounts of complex, preprogrammed information, we wanted to try to do discovery of the graphical user interface structure through software. No other tools that we had found had this ability, but we felt it would make a project like the VTAP much more viable.
- **Independent.** All other systems were highly dependent on a particular OS, OS revision, or visual depiction. Since we wanted a generalized GUI accessor system, we wanted to get away from dependency on any particular piece of software or visual representation. Ultimately, combined with the automatic requirement, we wanted to need as little *prior* information about an interface as possible. We

wanted to get away from the specific dependencies that would ultimately lead to complex and brittle interface software.

- **Durable.** Since we recognized the problem of little full-time developer support, and the short, dynamic lifetimes of other software products, we wanted a system that is *durable,* one that would be as zero-maintenance (from the developer point of view) as possible. For a system such as this, the programmer is the single highest cost in the system, and to maximize benefit we wanted something that could last with as little maintenance as long as possible. Many other projects in our own lab, such as experimental tactile interfaces and force feedback mice, were slowly becoming useless because there was no one available to provide the upkeep necessary to keep the systems working with new software revisions and interfaces.
- **Adaptable.** Fitting in the same mold as the goal for the system to be as automatic and durable as possible, we also wanted the system to be *adaptable* – able to reconfigure and change easily. Since the system would likely not be fully automatic, and would require some training for particular interfaces, we wanted the system to be able to do this in the least intrusive, easiest way possible, making adaptation to a new system a natural part of the solution.

Our approach, given these goals, was to build a system based on the same foundational ideas as SegMan, but to extend its underlying "visual" algorithms greatly. We liked the concepts behind SegMan and its visual system of working the pixels directly, but we wanted something that could take the concept of working with the screen visually to the next level, and gain a visual understanding of interfaces independent of prior information about that interface. Essentially, we wanted to create a system that could take the pixel groups found on the screen and discover natural hierarchies of interface segments that repeat themselves on the screen.

We also approached the problem from a hardware only point of view. In other words, we required that no software run on the host machine at all, and that all processing be done from a screen bitmap captured through other means, presumably a frame grabber. This gave us the final step of distance away from the host machine, so the VTAP system would be completely independent of rapidly changing, rapidly breaking host software. (For development purposes, working with screen shots would be acceptable, since a frame grabber would eventually provide the same data.)

Our primary belief was that we wanted a system that could cope with the "real world" as much as possible. We wanted a system that could deal with different OSes, changes in OSes, unknown and unforeseen interfaces, and basically make it possible for the blind user to handle any situation that a seeing user could handle.

As we will see, the Lens system that we developed comprises of a system formed from novel pixel algorithms, automatic learning techniques, a formalized architecture, and a software implementation of these algorithms and techniques that can deal with most real world situations in a way that no current existing system can handle.

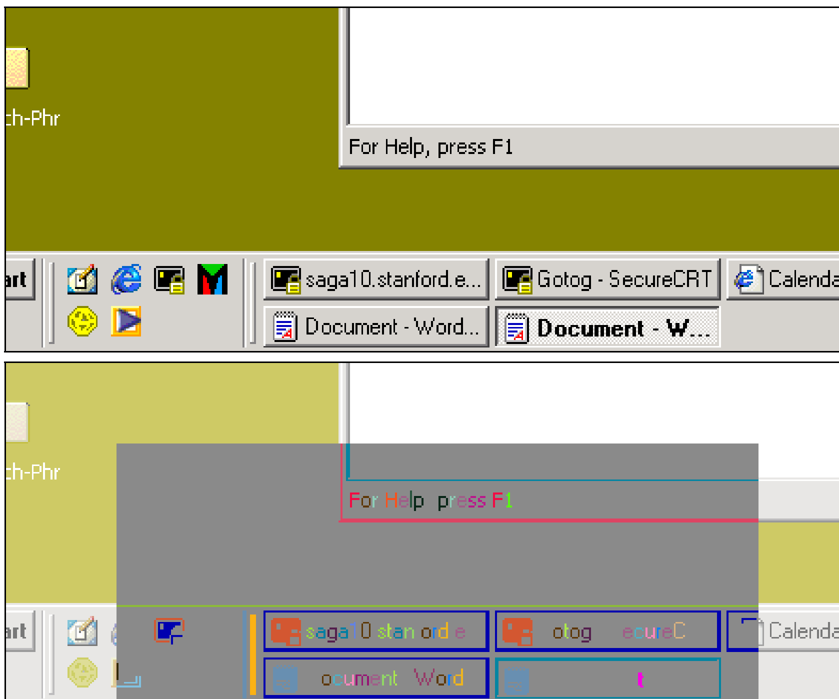

Figure 4. Example of a screen before and after user interface units have been automatically discovered by Lens. Almost as if it is lifting back the skin of the interface, Lens exposes the repeating structure of the GUI.

Lens Requirements

Before beginning development, we set in place a few requirements for how our system will work. These requirements were kept in mind while developing the algorithms and software behind Lens. We required that the resulting system be:

1. Fast enough to eventually run in real time with off the shelf hardware.
2. Very easy to train with new interface data, by a non-technical user.
3. Work from just pixel maps, given from a screen capture. No software or direct information will be provided from the host machine.
4. Software engine to be compact and be based on just a simple image manipulation library and standard C/C++.

5. Make information extracted by the system available for further processing, i.e. provide a good interface for other software tools to do post-processing on the structural interface information discovered.

WHAT WE BUILT: THE LENS SYSTEM

We designed and built a software system for extracting natural information about the structure of graphical user interfaces through a screen bitmap that is able to automatically discover and learn on screen interface units with no prior information, as well as learn from supervised examples given by an instructor in a very simple manner. This system includes both the algorithms that power the screen segmentation, and the architecture and software implementation that Lens fits into.

The Lens system was built in standard C++, and compiles on Windows, UNIX, and anywhere where a C++ compiler and the STL are available. The only external library that Lens depends upon is the ImageMagick library [14], which Lens uses for importation and exportation of screen bitmaps. However, it uses ImageMagick solely for file I/O, and not internal representation, and hence it is very easy to switch to other image processing libraries. (We already changed image manipulation libraries once during the development of Lens, with little necessary modification.)

In addition to the cross-platform processing code, a Windows-based "Interface Explorer" program was developed that allows the user to interact with the Lens recognition process in an online basis, and served as a proof-of-concept for the supervised training portion of the interface understanding algorithm.

Examples of Lens capabilities

Examples of the capabilities of the final Lens system include:

- **Automatic recognition of interface units.** Figure 4 shows an example of Lens' automatic interface recognition in action. Each unique interface unit has been colored a different random color. Note how Lens easily identifies button types, differentiating between selected and unselected, and how other components such as window borders and draggable tabs are identified uniquely.
- **Classification of similar pixel groups.** Figure 5 shows a portion of a complex user interface area with all "similar" pixel groups classified together and colored with the same color. Note how similar groups have different dimensions and scale, but are correctly identified as having the same underlying shape.
- **User training of new interface elements.** Beyond the automatic recognition and classification capabilities, the Lens system also supports simple, end-user training of interface elements, by simply clicking on the desired elements on screen.

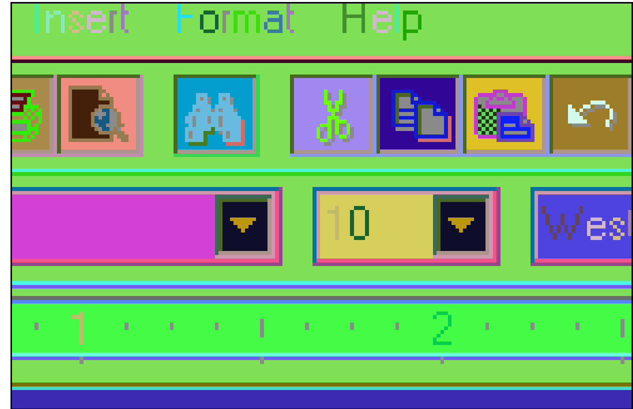

Figure 5. A portion of WordPad with identical PixelClasses colored with the same color. Note how contiguous pixel groups with the same *shape* have been assigned the same color, regardless of *scale*.

Algorithm: The Lens Algorithm

At the core of the Lens system are the algorithms it uses for classifying and grouping pixels on screen into user interface elements. The algorithms take the pixel-neighboring concept from the SegMan system, but extend that concept greatly into a hierarchy of three layers of visual abstraction. The three basic abstractions are that of the *Pixel, Interface,* and *Unit.* Each of these then exists as both an *instance* and a *class*, providing:

PixelGroup	*InterfaceGroup*	*UnitGroup*
PixelClass	*InterfaceClass*	*UnitClass*

To provide an overview of terms,

PixelGroups are individual groups of contiguous, same-color pixels on screen.

PixelClasses are collections of similar *PixelGroups*.

InterfaceGroups represent an adjacent "interface" found between two *PixelGroups* that formed of contiguous, same-color pixels.

InterfaceClasses are, following the system, collections of similar *InterfaceGroups*.

UnitClasses are collections of 3-tuples of the form *(PixelClass, InterfaceClass, PixelClass),* where the second PixelClass may be null. Each tuple represents a particular way that two PixelClasses can be combined. The list of tuples, then, specifies a collection of *PixelClasses* combined in a certain way. A restriction is enforced on the tuple list such that the tuple collection must be *connected*, in other words, there must be a path through tuples from each *PixelClass* to each other *PixelClass* in the tuple list. *UnitGroups* represent on screen instances of *UnitClasses*, and can be thought of being as composed of tuples of the form of *(PixelGroup, InterfaceGroup, PixelGroup)* instances.

To make the grouping by similarity possible, **PixelGroups** record the pixels involved in their group through what is called a pixel-neighboring vector, which represents, for a given contiguous group of pixels, the *shape* and *scale* of pixels within that group, without directly recording the location of the pixels themselves. Each pixel in the group is given an 8-bit value which representing the bit-vector that pixel's neighboring pixels. (See Figure 3.) The sum of these $2^8 = 256$ possible values over each pixel defines the pixel-neighboring vector for a group. **PixelClasses** are classes of similar PixelGroups, where each group has a pixel-neighboring vector that differs from the other neighboring vectors in the class by no more than *toler* of the 256 possible place values. Values for *toler* of 2-3 are typically used.

InterfaceGroups use the exact same pixel-neighboring vector as the PixelGroups for similarity, only they apply it to the next level of abstraction: they represent the *shape* and *scale* of a single-color interface between two PixelGroups. While difficult to grasp conceptually, this essentially applies the pixel-neighboring concept to the next level of abstraction. In other words, instead of a neighboring vector between pixels, Interfaces use a neighboring vector between *groups* of pixels. Following the abstraction, **InterfaceClasses** are then classes of similar InterfaceGroups, using the same similarity measurement for neighboring vectors as above.

These pixel-neighboring vectors are at the heart of the power of the Lens system, because they allow the *similarity* between on screen elements to be easily judged. They are essentially immune to scaling, since scaling affects only the magnitude of one or two of the 256 possible place values in the vector, making them ideal for user interface elements, which are frequently scaled along 1, 2, or 3 dimensions. But neighboring vectors are extremely sensitive to actual arrangement of the pixels (shape), since a disruption in relative arrangement changes many places on the vector. This makes them even more ideal for our task, because the exact drawn shape of a particular on-screen UI element never changes, only its scale.

With the importance of the neighboring vectors explained and the basic units of the system now defined, the general process of the Lens algorithm is as follows:

1. **Sort** the entire screen into PixelGroups, filtering out PixelGroups that are below a certain size.
2. **Discover** PixelClasses by comparing PixelGroups to each other and discovering **similar** classes of PixelGroups.
3. Find **interfaces** between all the PixelClasses, by creating a set of all of the InterfaceGroups that exist between them.
4. **Discover** InterfaceClasses by comparing **similar** InterfaceGroups that exist amongst the same PixelClasses to each other and discovering all similar classes of InterfaceGroups.
5. Now that a **graph** of PixelClasses (nodes) and InterfaceClasses (edges) exists, **search** through this graph for the most statistically common *connected islands* of nodes, and declare these islands to be *UnitClasses*.
6. Individual **instances** of these *UnitClasses* can then be found by perusing through the PixelClass → PixelGroup → InterfaceGroup network, and their location and extents are output.

This algorithm, then, performs automatic discovery of user interface elements, given screen bitmaps.

In addition, user-controlled creation of *UnitClasses* is allowed by simply letting the user select adjoining groups of pixels on the screen. Since a map between the screen bitmap and its classes and interfaces exists, this is a simple task. The PixelClasses and InterfaceClasses associated with the clicked pixels are looked up, and a new *UnitClass* is created with the selected tuples generated from the clicked PixelClasses.

Software Architecture

In addition to the Lens algorithms described above, two things had to be developed to create a working system: the VTAP architecture, which the Lens segmentation system fits into, and an actual software implementation of the Lens algorithms.

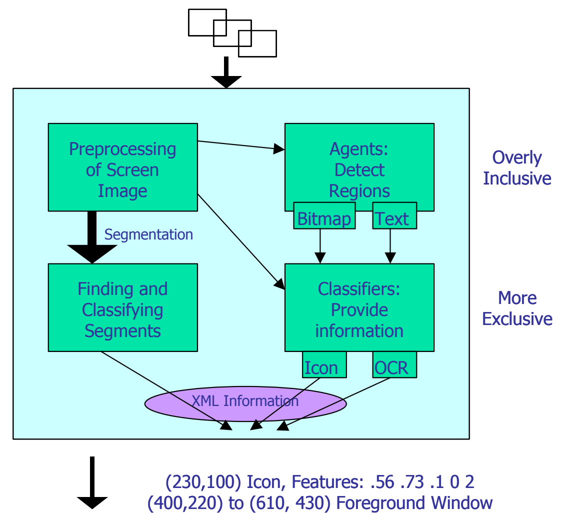

Figure 6. The VTAP architecture, developed with Lens.

The VTAP architecture, developed concurrently with Lens, is pictured here. The Lens segmentation architecture fulfills a major role in the VTAP architecture, by providing the basic interpretation of interface elements. The other portions of the architecture detect and classify items such as icons, text, and other things, which are not directly part of the Lens system's segmentation capabilities.

The relevant point of the above architecture is to define where and how the Lens system exists within the larger VTAP implementation.

The Lens system. The Lens system of algorithms is implemented with classes in the following class hierarchy:

- Master object: *VisTAP*. General system-wide object. Contains, creates, and initializes all other sub-objects.
- Global state object: *VGlobalState*. Holds the state that persists between multiple frames, such as information about the environment and the overall UI state.
- Per-frame state object: *VFrameState*. This is the per-frame state that is flushed and rebuilt with each new screen. Information that should persist is moved from the VFrameState to the VGlobalState at destruction.
- Segmentation manager: *VSegmentor*. Handles all segmentation discovery and classification tasks, and extracts information from the Units/segment table for the final semantic output
- Segment database: *VSegmentTable*. Holds all the Pixel, Interface, and Unit maps and lookup tables. This is kept separate from the VSegmentor so that it can be used directly as a data store for file writing/reading and for access by other classes and components. The VSegmentTable interface is also used to add new Unit information through user interaction.
- Pixel, Interface, Unit classes: *PixelGroup, PixelClass, InterfaceGroup, InterfaceClass, UnitGroup, UnitClass, UnitTuple, RGBData.* As explained above in the Algorithms sections, these classes hold the state and instance information that define them, as well as methods for search and creation from bitmap data.

The Lens system can run in automatic discovery mode, where UnitClasses will be discovered through the automated algorithms above, in an interactive mode where the user can input UnitClasses by simply mousing around the screen, and in a fixed mode where screens are just read in and units are found from the existing unit table, without any learning. The unit table can be written and read to disk upon request, so that state can be stored.

As part of our research, we implemented for Microsoft Windows a "Segment Explorer" program interacts with Lens library as a client, and allows the user to open images, process them, scroll around them, zoom in and out, and perform the mentioned Lens interactive training mode.

EVALULATION

Given our goals and requirements, did the Lens system solve our problem? We stated in our research agenda that we wanted a system that was automatic, independent, durable, and adaptable. Moreover, we required that the system be fast and easy to train. The Lens algorithm and software seemed to have these properties, but tests were in order to show its actual effectiveness.

To test the system, we used some sample screens with from different typical situations and different OSes, and examined what the Lens system was able to with them, given with no prior information about the user interface in question whatsoever. No formal metrics exist to measure Lens' success on each screen, but we wanted to get a feel for what Lens is capable of doing.

We also tested the supervised learning method for Lens' detection of user interface elements. This was tested with screen shots that were then run through the "Interface Explorer" Windows program that we created, and interface units were selected by clicking on pixel groups that comprised user interface units.

Overall Results

Figures 7, 8, and 9 detail our results. As you can see from all the figures, many repeating user interface elements have been automatically identified, and that user-supervised learning of non-automatically discovered elements is straightforward and extremely simple to perform, when compared to programmatic methods for recognizing user interface instances.

In Figure 7, we can see two typical Windows 2000 desktops. In the one on the left, which contains two windows from a simple word-processor application, we can see that a large amount of the user interface structure has been automatically discovered. The task-bar on the lower portion of the screen is of particular note, as you can see the buttons identified identically. Also, Figure 4 from earlier in this paper shows another close-up view of the same information—clearly showing that the structure of the user interface has been identified.

In the other, right-hand screen in Figure 7, we see a similar desktop screen; only this desktop is filled with a large, complex bitmap image. This type of bitmap image represents a common type of "screen noise" that presents a serious problem for systems that are trying to extract useable information from an interface. If such a bitmap poses a problem to the system, then other images will clearly cause a problem in the future. Notice, however, that after Lens has processed this screen, the image completely disappears. All that remains in the screen is the structure of the user interface—magnifying the signal to noise ratio of the screen to a blind user tremendously.

For these screens, with similar units now recognized and identified, and noise removed, specific auditory/tactile feedback can be given for each type of interface element, or for the location of a specific type of element, allowing the blind user using the system to have a much clearer impression about the visual information on the screen. The interface has now been transformed from a blank, featureless screen to a raised, labeled control board—

something a blind user can navigate. Even without additional semantic information about what each unit actually does or represents, we now have a map of what classes of functions are available on the screen, and where. In essence, the user interface has now been turned into a map of controls and their types, essentially classifying and labeling the functional features of the screen for output to auditory, tactile, and other types of interfaces.

Hence, segmentation and automatic discovery of visual interface structure on the screen with Lens appears to be quite viable. Even without any prior information about the operating system, interface elements have been successfully identified. To further test the assumption of independence and durability of the Lens system, Figure 8 shows the output of the Lens system run on a snapshot from Adobe Photoshop on the MacOS.

On this very complex interface, Lens has identified the user interface structure, and has classified most all of the controls on screen uniquely. Moreover, the test was successful on a different operating system (MacOS vs. Windows 2000 in the earlier screens), without any specific prior information about the system in question.

So far, our tests have concentrated on the "automatic" abilities of the system. Adaptability to change from user input, however, is another main goal of our system. To test this adaptability built into Lens, Figure 9 shows an example of supervised learning with the Lens system. Two areas were selected from one of our earlier screen shots, and the user was allowed to click on the relevant components of the desired visual unit, as shown in the figure. From these simple clicks along obvious visual

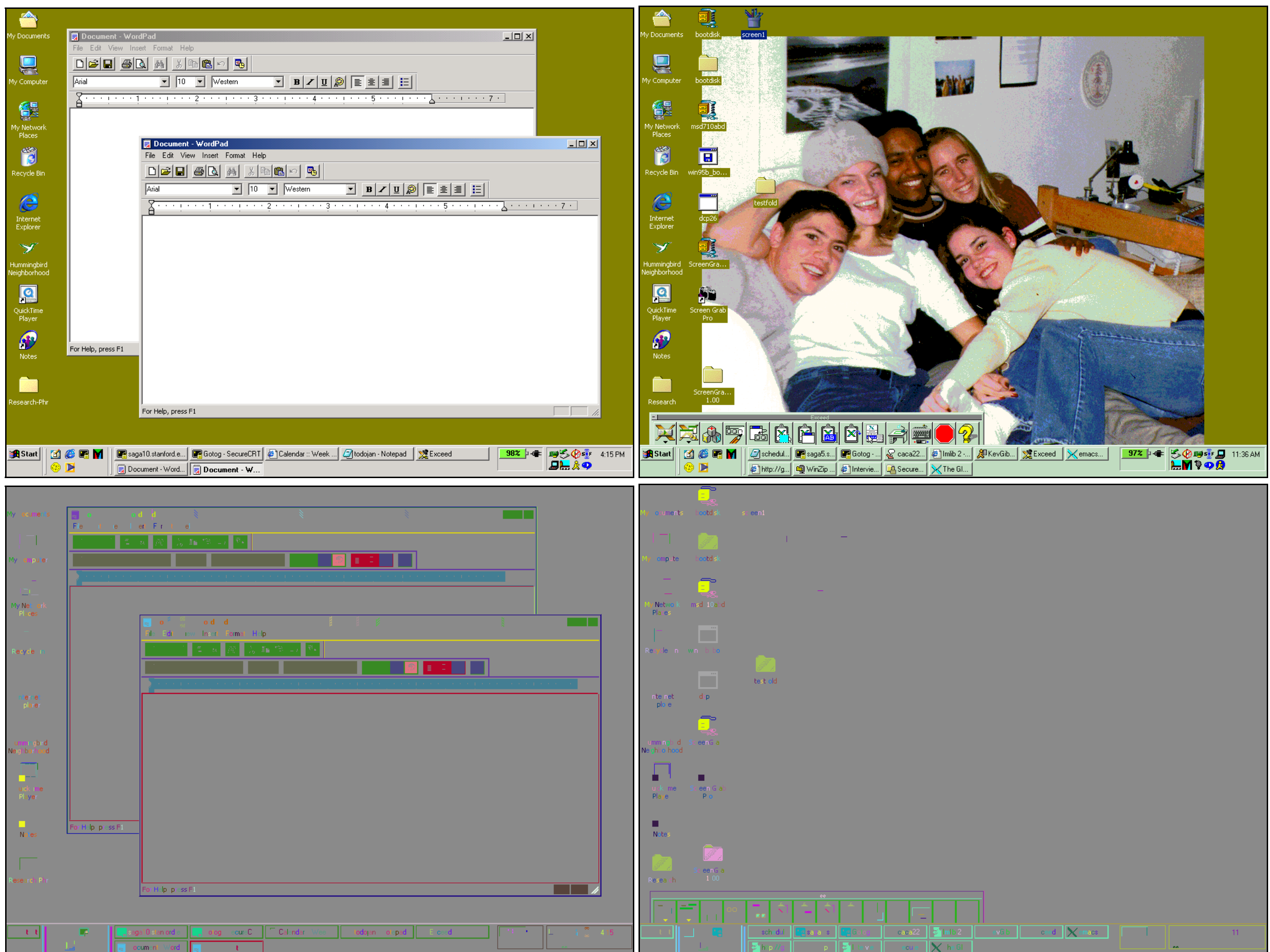

Figure 7. Two shots from a Windows desktop. The left has two simple word processor windows open, and most interface units have been automatically detected. Note in particular the task bar at the bottom of the screen (see Figure 4 for a close-up) and the assorted window and text controls. The right screen has a similar desktop with a complex bitmap photograph in the background. Notice how the photograph disappears after running the segmentation algorithm, while nearly all useful interface structure still remains. This filtering alone increases the interface signal to noise ratio of the screen drastically, and in and of itself almost makes simplistic tactile navigation of the screen possible.

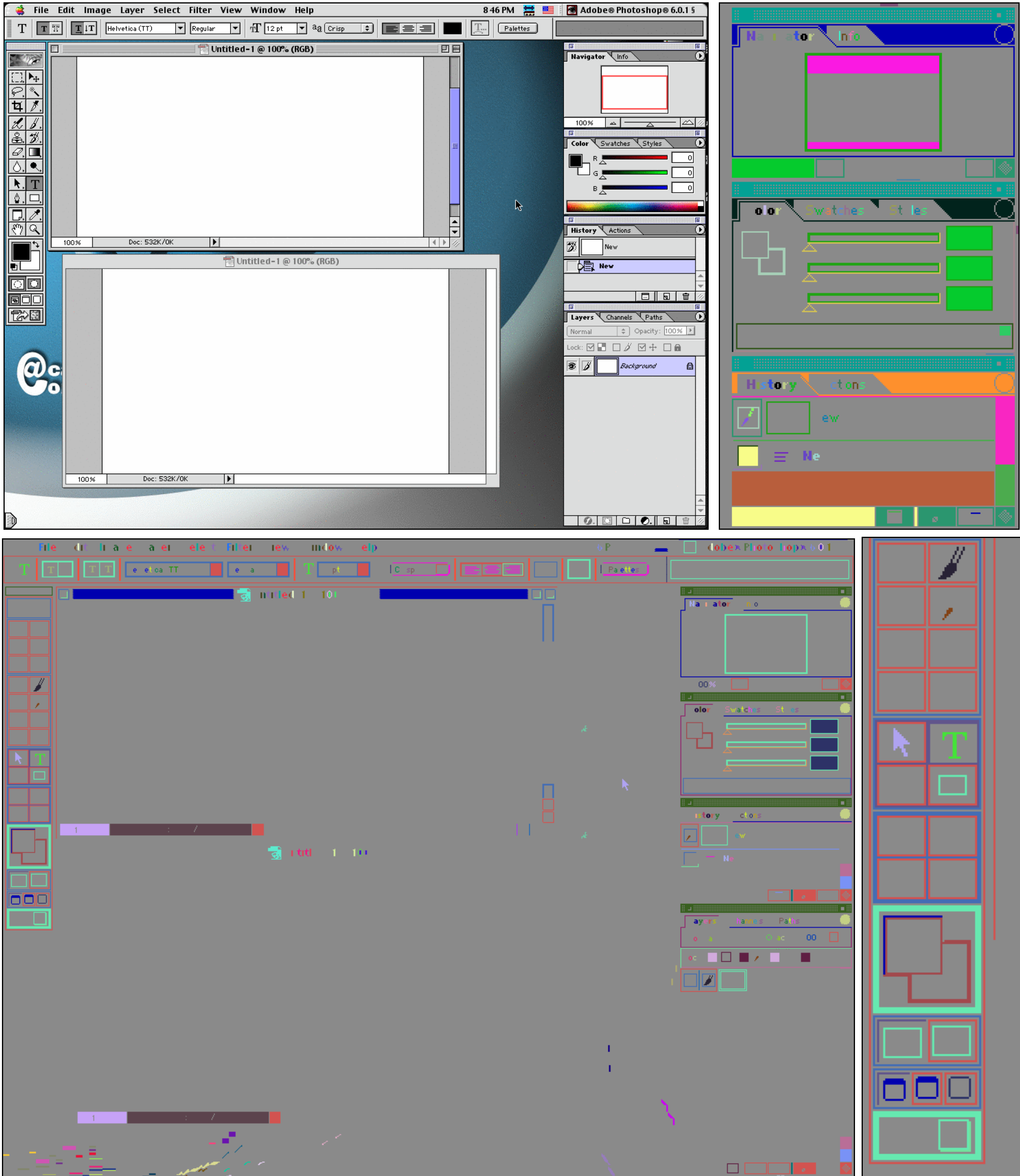

Figure 8. Adobe Photoshop for MacOS 9, run through the Lens automatic unit discovery algorithm. Note the detail of the tool palettes on the right. Similar button types and user interface controls have been colored as the same type of unit.

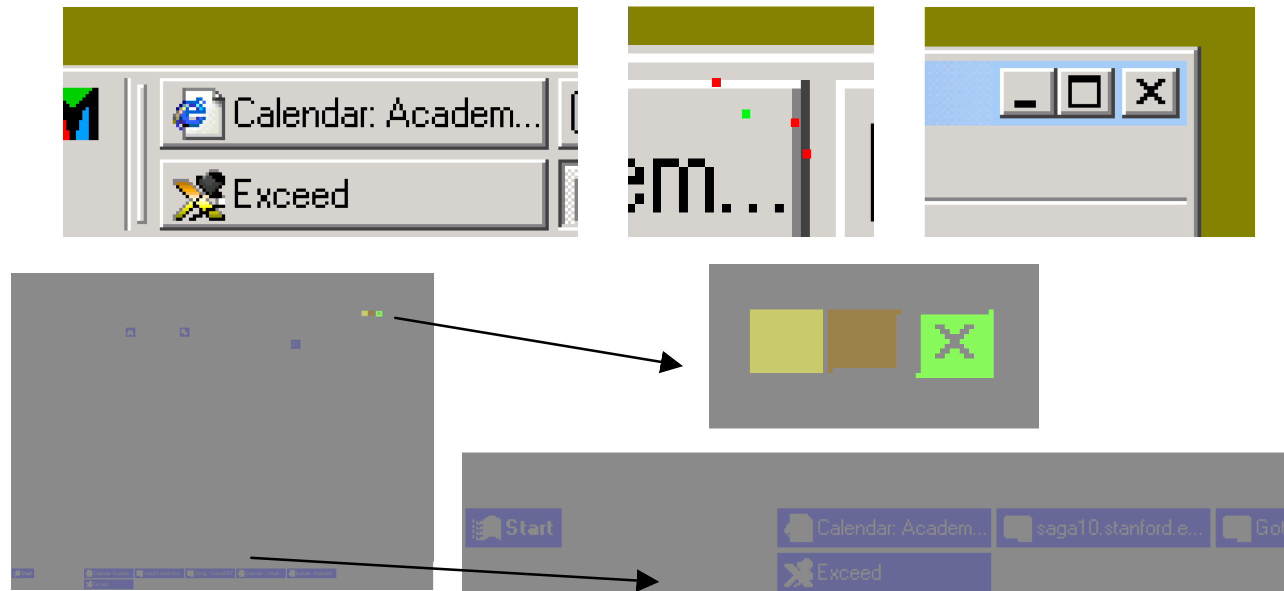

Figure 9. Supervised user interface learning with Lens. The top left and top right panels show a taskbar button and a window control, respectively, that we would like to learn with Lens. To do so, we merely click on each pixel group that makes up the unit, which is demonstrated in the top center frame for the taskbar button. The red clicks specify that the group defines the shape of the unit, while the green clicks specify that the group is associated with the unit, but should not be used to define the shape. The lower three frames show the recognition performed on the screen. The details show how units have been properly "learned" from the user mouse clicks on the groups above.

segments, the Lens system was able to easily encode and recognize instances of those specified user interface elements while processing the screen. The amount of work for inputting a user-interface description with this simple point and click system is many orders of magnitude faster and simpler than inputting a complex programmatic specification, such as those used in SegMan, while apparently just as effective.

One final aspect that we evaluated in the Lens system was that of speed. If processing time is too great, the Lens system could be potentially useless in a real-time user situation. However, in all of the above examples, Lens processed the screens in close to real time, taking at most a few seconds of wall clock time to process a screen from input file to output on commonly available hardware. A 1.4 GHz Athlon CPU with 500 MB of RAM was used for primary testing, while the Interface Explorer was tested on a 800 MHz Pentium III laptop.) Given the early and unoptimized form of the Lens software, this speed would seem to suggest that Lens is capable of performing quickly enough to be a viable system for online, real-time usage.

Evaluation Summary

What should be clear from the screens and examples above is that the system was successful in what we set out to do. In a flexible manner, it was able to discover user interface structure, without prior information. It was also able to learn in a supervised fashion with very simple end-user control. And since the software is independent of prior information about the interface in question, as well as host-software free, thus able to withstand changes over time, we have built a GUI understanding system that is automatic, independent, durable, adaptable, fast, and easy to train. For our problem, the Lens system is a successful solution—it fulfills our requirements, and is able to quickly transform an otherwise indecipherable screen bitmap into a listing of categorized, classified information about its visual interface structure. The Lens system has made the first step toward implementing the larger VisualTAS system a success, and shown that understanding visual interface information in a general way for blind users is a possibility.

RELATED WORK

In our initial description of this project, we discussed some systems that attempted to solve similar problems to this. Of the systems discussed, SegMan [8] is the only one that is directly related to this work. SegMan, and its predecessor VisMap [9], make use of the pixel-neighboring representation that is used by Lens, as well as the concept of pixel groups, or groups of contiguous pixel regions on screen. But that is where the resemblance stops. Lens carries the pixel-neighboring and pixel grouping abstractions 3 levels higher, with the introduction of the concepts of Pixels, Interfaces, and Units, as well as introducing the fundamental division between Groups (individual instances) and Classes for each abstraction. Of course, is vital to note that SegMan is attempting to accomplish a different problem—the facilitation of user interface softbots, or software agents that can explore and interact with a graphical interface. While similar to our problem, the problems are sufficiently different that different approaches and solutions are warranted.

Beyond SegMan, our group has found few other projects that attempt to approach the same type of problem as the Lens system. Henry Lieberman's *Your Wish is My Command: Programming by Example* [7] examines a variety of Programming by Demonstration and Programming by Example systems, many of which have similar problems of user interface understanding to solve. However, in Lieberman's broad survey, few (if any) systems other than VisMap (the predecessor of SegMan) approach the problem of holistic interface understanding through screen bitmap analysis. VisMap and related projects and material are found in Chapter 19, "Visual Generalization in Programming by Example" [13].

Your Wish is My Command does also talk about many of these overarching problems of how to generalize from a user interface, and considers the theoretical implications of attempting to understand a user interface and the inherent dangers and difficulties in such a problem. These thoughts and insights had definite bearings on our goals and understandings coming into this project.

In essence, most all other software that attempts to understand a visual interface functions at a higher level than ours, when considering access to the host system. Even SegMan uses a .dll on the host system to access pixels, where we will use an external video source. And in terms of configuration or prior knowledge, our system is also on the lowest level. It attempts to do as much automatic learning as possible, and even the supervised input learning that it performs is in the simplest possible format. So the Lens system has found a new niche in that it attempts to work in the most general, least intrusive way possible—and succeeds relatively well in its seemingly impossible task of doing so.

CONCLUSIONS

Our main message is that with the Lens software, it is possible to interpret and work with a GUI without an involved, prior knowledge of that particular system beforehand. This key ability allows Lens to be a viable candidate for a truly generic visual interface access product—exactly the goal we are working towards with the VisualTAS project. Moreover, the Lens system allows programmers to concentrate their time and efforts on other areas, since it automates many tedious tasks for understanding a user interface.

Although the Lens system was designed for the VTAP and VisualTAS system for the blind, Lens has many other possible uses that we have not investigated in this paper. With its automatic user interface discovery methods, it could be used to create powerful scripting and automation systems, allowing computer agents to record or control actions in everyday GUIs. It could also be used to analyze usage patterns of a given computer or system, since it knows (or can be taught) precisely what components the user is interacting with, and how. The larger Visual Total Access System project, once completed, also has many usability implications for non-disabled users. Lens with the VisualTAS could make your home computer accessible to you over the phone, or allow pervasive computers to provide access to graphical interfaces in a form appropriate for whatever your current situation is—whether in your car, while on the phone, or during a movie.

There is much left to do. Even with the initial completion of Lens, VTAP and the larger VisualTAS system are by no means finished; Lens fills in only a portion of what it needed in a full system for GUI access to the blind. Lens itself could see much additional development and enhancement. Code and algorithms can be optimized for speed. More sophisticated statistical methods could be applied to determining the set of automatic user interface units. Databases of common interfaces should be created, and wider testing with the Lens should be performed on a variety of systems.

Ultimately, however, the Lens screen understanding algorithms have already provided the vital first step towards a fully competent computer access system for the blind.

Software availability. The Lens implementation mentioned in this paper is available from the Project Archimedes web site, at http://archimedes.stanford.edu/. The Windows Interface Explorer software is also available at the Archimedes site. The ImageMagick library, linked to by the Lens implementation, is available on the Internet at http://www.imagemagick.org/. For more information about the software packages, feel free to contact the author.

Acknowledgements. The helpful input of Terry Winograd's the CS377F HCI research seminar on the content of this paper is thankfully acknowledged. The assistance of Bharath Beedu, Sonal Josan, and the rest of the larger VTAP team at Project Archimedes with the VTAP architecture is also gratefully acknowledged. Sincerest thanks also goes out to Neil Scott, for the vision in getting the VisualTAS project and Project Archimedes itself rolling. The National Science Foundation Grant partially funded this work.

REFERENCES

1. Raman, T.V. "Emacspeak – The complete audio desktop." http://www.cs.cornell.edu/Info/People/raman/emacspeak/emacspeak.html
2. IBM Corporation. "IBM Home Page Reader." http://www-3.ibm.com/able/hpr.html
3. http://www.who.int/archives/inf-pr-1997/en/pr97-15.html blindness stat, 97
4. Project Archimedes. http://archimedes.stanford.edu
5. Scott, N. "VisualTAS Project Proposal Document."

6. St. Amant, R. "User interface softbots." http://www.csc.ncsu.edu/faculty/stamant/ibots.html.
7. Lieberman, H., et al. Your wish is my command. New York: Morgan Kaufmann. 2000.
8. Riedl , M. and St. Amant, R. "Toward automated exploration of interactive systems." *Proceedings of the International Conference on Intelligent User Interfaces (IUI).* To appear.
9. St. Amant, R. and Zettlemoyer, L. "The user interface as an agent environment." *Autonomous Agents.* 2000. Pp. 483-490.
10. Riedl, M. and St. Amant, R. "SegMan technical manual." http://www.csc.ncsu.edu/faculty/stamant/segman-introduction.html.
11. McKinley, J. and Scott, N. "The Development of a Video TAP to Process the Visual Display into Non-Visual Modalities." http://archimedes.stanford.edu/videotap
12. O'Modhrain, M. and Gillespie, B. "The O'Modhrain Moose: A Haptic User Interface for Blind Persons." http://archimedes.stanford.edu/videotap/moose.html
13. R. Potter, et al. *Visual Generalization in Programming by Example.* Printed in Your Wish is My Command, 2001.
14. ImageMagick software package. http://www.imagemagick.org.