ATK + 8859 = Multi-Lingual Text and Mail

A Study in Expanding the Andrew Toolkit

Thomas Neuendorffer
Information Technology Center Carnegie Mellon University Pittsburgh, Pa. 15213

ABSTRACT

The Andrew Toolkit and Andrew Message system have been available on the X distribution tape since X11 release 2, providing multi-font text with embedded objects (drawings, rasters, animations, etc.) that can be edited, printed or mailed between users. The recent addition of the ISO 8859 fonts to the X- distribution has prompted the ITC to add some additional support to allow multi-lingual text and mail that takes advantage of the expanded character set.

This paper provides an overview of what facilities are available, and how they can be customized to fit the needs of a particular site. The goal of this paper is to illustrate some of the features of ATK that allow it to be adapted to specific needs or purposes. There is also a technical overview of what was involved in adding the ISO support to ATK while maintaining backward compatibility.

1. Introduction

The true test of a system is its ability to adapt to external needs and opportunities that arise after the system was conceived. Unix has certainly passed this test in the way it has adapted to systems and computer architectures far beyond the pdp 11's on which it was first implemented. For a user interface toolkit, the challenges are different. Several recent papers [Hayden1990, Williams1990, Cattey90] have described the use of Andrew as the basis for major programming efforts. The goals of this paper are to present the expansion methods of ATK that allow significant customization and application development without significant programming effort, and to illustrate the line of thinking that goes into producing an ATK application. In the process, we hope to demonstrate how the problem of accented characters can be solved relatively easily given a system with reasonable power and flexibility.

1.1. The problem/opportunity

While graphical workstations have been available for several years now, the writers of software for these machines have been surprisingly slow in taking advantage of their capabilities. This is particularly true when it comes to text oriented systems, accented characters and electronic mail. There are relatively few systems that provide on-screen editing of accented characters. Those that do exist typically do not deal with electronic mail, or the problems of exporting these capabilities to other applications. Nor has there been much work in the area of toolkits that support the creation of applications that handle accented characters. There is an obvious need to deal with this problem on a system-wide basis. A user should be able to read ‘‘Flygande bäckasiner söka hwila på mjuka tufvor’’ in electronic mail and copy that into a text being edited or paste it in a spreadsheet.

In 1987, the International Organization for Standardization (ISO) published a set of 8-bit single byte encoded fonts [ISO87] to support the Latin alphabet used by many western countries. The fonts included with X -Window system [Scheifler86] release of January 1990, adhere to part one of this standard. With these basic resources in place, the Andrew toolkit was well situated to attack this problem

1.2. Issues

The key issues in dealing with the ISO fonts in ATK were.

1. Screen display
2. Internal representation
3. External representation, in a form compatible with electronic mail.
4. Printing.
5. Ease and flexibility of user entry.

Of these issues, the first three are basic support issues that needed to be dealt with in the lower levels of the system, while the last two call for good basic support within the system to allow their development as extensions. Our goal has been to see that a good general capability in these areas is provided, while, at the same time, the system remains open for local conventions, modifications and customization, including graphical interfaces.

2. Background: The Andrew Toolkit

The Andrew toolkit (ATK) [Palay88] provides a object-oriented environment wherein objects can be dynamically-loaded. In addition to containing multiple fonts, text can contain embedded raster images, spreadsheets, drawing editors, equations, simple animations, etc.. These embedded objects could themselves contain other objects, including text. These resulting multi-media documents can be edited, printed, or sent in mail messages via the Andrew Mail System [Borenstein88]. AMS takes advantage of the fact that the ATK external file format (*datastream*) is defined to be short line seven-bit ascii, so that it can safely send ATK files around in electronic mail, regardless of the fact that they may contain esoteric objects, like animations or electronic pianos. Because AMS uses the same text object that the ATK editor does, any modifications to that text object to deal with ISO fonts will automatically add that functionality to both the editor and the messages program, as well as the many other applications that use the text object. In addition, since the cut- bufffer also uses datastream format, it will also be possible to cut and paste ISO characters between applications, just as it is currently possible to cut and paste styled text.

In order to understand ATK programming, it is important to grasp the concepts of the two main hierarchy within the system, the view hierarchy and the class hierarchy.

2.1. The Class Hierarchy

The class hierarchy should be familiar to those with experience in object oriented programming. Whenever a programmer codes a new object, he/she writes it to be a sub-class of another object, whose methods (functions) it will inherit and can override. Two children of the top-level observable class are the *view* class and the *dataobject* class. The *dataobject* class is designed to be the parent of those objects that will store and read data from a file. One dataobject is the *text* class, that will store text strings and style information to a file. When one is editing text with the ATK editor *ez*, it is the *text* object that stores and saves that information.

Most *dataobjects* have a corresponding *view* object. The *view* is responsible for being able to display the dataobjects information on the screen, and provides the interaction with the menu and keystroke packages that allows the user to edit the *dataobject's* information. The *textview* object provides this capability for *text*. The combination of a *dataobject* and its *view* is called an *inset.* See Figure 1.

2.2. The View Hierarchy

The view hierarchy is more unique to ATK. ATK provides a general mechanism that allows insets to be embedded in one another. A text inset could contain a spreadsheet inset that could itself contain a raster , a drawing, or even another text inset. The standard ATK editor *ez* does not, in fact, know anything about editing text. Text just happened to be one of the insets that it can embed in its *frame*, which is a view that manages the message line, as well as the underlying views being edited. *Ez* is just as capable of viewing and editing a spreadsheet or a raster image as it is text. Figure 2 illustrates the editor editing text that contains an animation inset, along with a diagram showing the view hierarchy. Requests for services, menu postings and keystroke requests are passed up the tree, and events (mouse hits, redraw requests, etc.) are passed down. The parent can override a child's requests, or add their own on in addition. Events are passed to the parents who have the option of passing them on to their children.

For our purposes, this is particularly important in dealing with key sequences and menu choices. The view at the top of the view tree handling all of this is the inset manager or *im*. When a view receives the input focus, the *im* is the object that sees that each request to map a procedure with one or more keystrokes or menu entries is handled.

2.3. Customizing the Interface

During this process of adding the procedures requested by the view, the im also checks to see if the user has added or overridden any procedures in his init file for that viewtype. If so,the user specified procedures are added. In addition, many objects check for attributes in the user’s preference file that will customize a specific interface. The system manager can also install global init and preference files. These can allow for significant site customization without any changes to the source.

3. Approaches to supporting ISO characters in ATK

3.1. Creating an ISO inset

The typical approach to extending ATK is to write a new inset. To apply that approach here would mean treating ISO characters the same way that we treat drawing, raster images and other objects that can be inserted into text. Whenever an ISO character would be needed, there would be an interface that would insert character sized inset containing that one character into the text. Printing could also be supported, since insets generally provide their own print methods. This approach has the advantage of being completely backward compatible, requiring no interface changes to the text object. It would also allow these characters to be inserted in insets other than text. The disadvantage comes in the fact that insets are somewhat heavyweight approach to inserting single characters in a document. It would also make certain common actions, such as searching a text for a string containing on or more ISO characters, a much more difficult (though not impossible) routine to implement. The ability to add insets is a powerful mechanism unique to ATK, however since ISO characters, for most operations, should be treated the same as other text characters, the use of an ISO inset was rejected.

3.2. Subclassing text

Another approach would be to create a sub-class of text with the additional functionality of being able to deal with ISO characters. This is the approach taken by the specialized editors that have been created to edit c, modula 3, and lisp code. It is also the approach taken by various other objects, like one that displays a list of choices and provides feedback to an application when a user clicks the mouse on a choice. Again, while this is an excellent approach for some applications, it would not be appropriate for dealing with ISO characters. For one thing, the use of ISO characters should be ubiquitous. All of the sub-classes of text, like the message line or the list object mentioned above, should be able to deal with ISO characters. ATK does not currently provide the capability for subclasses to masquerade as their parent class as far as other sub-classes are concerned. Add to this the fact that when text was written, certain decisions were made regarding what methods needed to be available for overriding by the subclass. Certain things (such as displaying an individual character on the screen) are going to have to be handled by the parent class, for efficiency if no other reason.

There is ongoing work at Kieo University, Japan [Kitagawa90] that is using this approach to create a Japanese text inset. Given the completely different model that is needed to support Japanese characters, this is a reasonable approach.

3.3. Expanding the text object via new proctable entries

ATK provides the capability for programmers to write modules to expand the capabilities of existing objects. These modules provide new functions that can be invoked by the user via key-strokes or menu commands. Because of its flexibility, and the fact that it allowed dealing with the existing text object, it was decided to use this approach, combined with backward compatible modifications or bug fixes to the text inset itself where necessary and/or appropriate.

3.4. Using Ness

Ness, an object expansion language that is shipped with ATK, includes the capability of defining new user level procedures that modify text. Though it could have been used to implement some of the desired routines, C based extensions had the advantage that they could deal with parts of the system that ness doesn't provide interfaces for, the buffer help facility for example. Still, much customization work is possible with Ness. See [Hansen90] for more information.

4. Providing the basic support

4.1. Screen display

This in fact was already dealt with. Since X had always allowed for more than 128 characters per font, the ATK graphics layer was already prepared to deal with them, including a call to indicate if a given character was in a given font. Using the X font aliasing feature, it was relatively simple to replace the standard ATK fonts with their ISO equivalent. A few minor bugs regarding the different handling of chars and unsigned chars had to be dealt with.

4.2. Internal representation

This also presented little problem. The ATK text object was not using the 8th bit of its character string for any other purpose, except for the placeholder character that was placed in the location of an embedded inset. And, even in this case, it was only used as a hint that a view may be held by a surrounding style object. So internally, no changes were made to store the 8-bit characters. Again, some changes had to be made for modules like the search routine, but these were relatively minor.

4.3. External representation

During the initial work, it was sufficient to store the resulting files in 8-bit form, however this would not be reasonable for shipped code. The ATK datastream is defined to be 7-bit ascii, with no long lines. This was done primarily to allow it to be sent as electronic mail with standard mailers. All ATK objects are expected to save themselves according to these guidelines. Now of course there are many ways to save 8 bits in seven bit form, however we needed a way that would be backward compatible with older versions of

ATK. The form chosen is a variation on the way styled text is stored. An ´a’ with the high bit set will store as

\ˆ{a}

Compare this to the way that a bold ’**a**’ would store.

\bold{a}

This format has the advantage that older unpatched versions of ATK will just treat high bit characters as an unknown style with the name ’ˆ’. Thus if a new text is read and written out again by and old version of ATK , the high-bit information will preserved, even though the display will be incorrect.

4.4. Printing

ATK generates troff code for high-quality multi-font printing. The decision to use troff was made at a time when it was the only commonly available system under Unix capable of printing high quality text on laser printers. Though troff is not distributed with ATK, it is generally available on most Unix systems. ATK’s use of troff for printing has, from the beginning, been both a blessing and a curse. It has been treated as the assembly language for creating printed documents and for the most part, has worked quite well. Where it falls apart is where we don’t really have enough information to do what needs to be done, the primary example of this being the proper handling of tabs. Through the years though, we have been able to adapt it to provide additional facilities, such as the printing of tables of contents, indices, and footnotes, as well as being ably to deal with embedded images, either through insets that generate troff drawing , or insets that generate postscript that is placed in the troff stream and passed through to the printer. While there has been talk of getting ATK to generate postscript directly for printing, a large amount of work would be required to write code to take care of all of the things we are currently using troff to do.

Since raw troff does not handle accented characters, we needed to look into producing macros to deal with them. In the process, we decided to modify the text-to-troff conversion object *txttroff* to look for a user-defined procedure for translating high-order bit characters to usable troff strings. This allowed us to experiment with various options, while allowing for routines that dealt with local printing conventions.

4.5. Regarding the basic support

At this point the lowest level of support is in place. One could argue that the above changes fall into the category of bug fixes to the underlying system, and that the bugs were historical artifacts of the fact that ATK was developed on a window system that assumed a 7 bit character set. What is left is system expansion work that can be done without changes to the underlying system.

5. The first application, Swedish.c

In development efforts of this type it is often useful to attack a more specific problem as part of the process of finding a general solution. We began to concentrate efforts on a specific target audience, a group at Chalmers University of Technology in Sweden.

The Chalmers group had been working with ATK for several years, however they used the standard Swedish conventions when they needed accented text. For example, a left open bracket '}' would be used to represent an 'å'. By adapting their printing programs to convert back to the accented form, they were able to get properly printed documents, but this solution could obviously be improved upon. There should be no reason to deal with these sorts of limitations in a graphical workstation environment .

In response to their needs, the module Swedish.c was written. It provides dynamically loadable routines for each of the 6 most common Swedish ISO characters into text, and procedures for searching through a section of text and replacing all of the 'special' characters with their ISO equivalents. Since text has now been modified to store the ISO characters in 7-bit format, two users, both using the ATK messages program to send and receive mail, could now exchange native language electronic mail. Of course not everyone they correspond to will be using ATK, so the conversion routines can be used to easily convert to and from the local conventions.

Users are able to call these routines by adding the following lines to their .atkinit file

```
addkey swedish-o-diaeresis-small \S textview
addkey swedish-o-diaeresis-cap \T textview
addkey swedish-a-diaeresis-small \U textview
addkey swedish-a-diaeresis-cap \V textview
addkey swedish-a-overcircle-small \W textview
addkey swedish-a-overcircle-cap \P textview
addmenu swedish-convert-to-ATK ‘‘Swedish,Use Swedish characters’’ textview
addmenu swedish-convert-from-ATK ‘‘Swedish,Use brackets for Swedish’’
textview
```

The *addkey* requests will call the given procedure from a textview when the given key-sequence is typed. In this case, an ESCAPE W (or F5 on an IBM keyboard) will insert an aa into the text being edited, but the user could modify the file for any desired key or key-sequence. Similarly, the *addmenu* requests will add a line to a new or existing card that, when chosen, will call the named procedure. Here, a ‘‘Use Swedish characters’’ will be added to the card titled ‘‘Swedish’’ to call the convert-to-ATK routine. These routines can also be added for all users at a site by adding them to the global .atkinit file. An additional proctable was added to deal with printing that would take advantage of the hook added to *texttroff*. This function needed only to do the same conversions that convert from ATK did, except that the backslash character had to be escaped.

This relatively simple module (included in its entirety in Appendix 1) did much to make both ATK and X far more useful to the people at Chalmers. They have expanded it to included a few more ISO characters and expect to begin introducing the messages program to their students in the fall.

6. Attacking the general problem

With the success of Swedish.c, the next step was to attack the problems encountered in a generally applicable way.

6.1. Several solutions to character input.

If a site has a keyboard set up to send the proper ISO-codes for the ISO characters on its keys, then character entry should be straight forward. However not all sites have these luxuries. In order to support a variety of interaction styles, several solutions to the general problem of character input were implemented.

6.1.1. Character modifying procedures

The first set of methods simply added an accent to the last character typed. Each procedure was linked to a specific accent and could be bound to a menu or keystroke. When invoked, the procedure would look at the last character entered, do a table lookup, and replace it with the appropriate ISO character.

Another set of procedures allowed the accent character to be entered first. This was possible because the key binding package allows for overrides to be set so that a programmer can cause keys to be interpreted by a special function before possibly being interpreted by the normal methods. So an override was set such that the next key typed would enter the code for the typed character with an appropriate accent.

6.1.2. Compose character, version 1

The problem with these solutions was that since the mapping of character to accent was done by the general keymap facility, there was no central knowledge of accent-specific mappings that could conveniently be set or looked up to provide the user help. The next step was to provide a translation table and a compose character function.

For flexibility, it was decided use a user-replaceable table to specify the translations between character combinations, a high-order-bit character to input, and the troff string for printing. The location of this table is specifiable in the users preference file. This again allows for customization by both sites and individuals. An example table entry that would specify the translation of an ’a’ follow by a ’ˆ’ to an ’â’ (ISO character 226) looks like:

aˆ 226 a*ˆ

This entry also specifies ‘‘a*ˆ’’ as the string to be sent to troff for printing this character (see Printing below).

It was now possible to provide a compose character function that provided online, interactive help. The string input facilities provided by the editor package includes the ability to display a help buffer when a ’?’ is typed and to fill in part of an input string in response to a space character. Since the strings to be entered were all ˜3 characters, providing completion was perhaps somewhat redundant. The help facility, however, allows the user to grow familiar with the character set provided by compchar quickly and easily as they are not forced to look up the character in a help file or other documentation. Figure 3 shows the help buffer for completing the letter a.

6.1.3. Compose character, version 2

The primary failure of this help mechanism occurs in Andrew messages because while it does use some of the facilities provided by the editor package, other facilities, including the help buffer, are disabled since it does not use the buffer package. In addition, using the string input facilities had the drawback of requiring a newline to terminate the name of the character desired. This rapidly became an annoyance even in testing so a way to avoid this was clearly needed. A second compchar function was implemented using the key overrides to allow the input of a character to be terminated automatically upon the completion of the character. Help could then be provided in the message line as each character was typed. So with this new approach, when one types the ‘‘compose character’’ key, followed by an ’a’, the message line would display the following message prompting for the completion

Possible completions: Æ-E, æ-e, å-o, ä-:, ã-˜, â-ˆ, á-a, à-g

When the completion character was typed, the character would be immediately entered in the document.

6.2. Printing

After implementing the local solution above, we started looking at more general solutions to printing accented characters with troff. It turns out that two of the standard macro packages for troff, mm and ms, provided troff macros for printing the accented characters and we were able to adapt these macros for use with ATK. For initial testing, we used the expansion method added to the text-to-troff translation object for Swedish to implement this, along with another preference option that allowed us to modify the troff macros that ATK applies to the troff it generates. Once this proved a general enough solution, we installed it as the part of the default printing scheme, leaving in place the capability for user override.

7. Adding a graphical interface

7.1. A Scenario

Let us return to Sweden for the moment and consider the following scenario. A site administrator has a large number of users (both experienced and novice) using AMS for sending and receiving mail. Not all of the users have Swedish keyboards and he/she would like to provide simple graphic interface that would allow them to take advantage of these newly available non-ascii characters. One option would be to provide a ’soft’ on-screen keyboard that would display the commonly used characters, and allow the users to enter them into a mail message by clicking the mouse on the desired character. It would also be useful to provide buttons to do the transition to and from the local conventions. On most systems, this would be a fairly difficult project, requiring a lot of programming and major modifications to the underlying mail program, assuming that it would be even doable. Andrew provides support to allow this kind on interface to be added with under 100 lines of user written code.

7.2. Online creation of the interface

The Andrew Development Environment Workbench (Adew) described in [Neuendorffer89] and [Beer90] allows the programmer to quickly prototype the application on the screen by pasting together a collection of insets. It then provides support for turing the prototype into a finished application. The sample Adew programs have been mostly stand-alone application like a calculator , or interfaces to external devices, like the mailable electronic piano mentioned above. With a little additional knowledge of the system however, Adew can be used for graphical extensions to new or existing insets. So to deal with the scenario, let’s start with creating the soft keyboard. Adew provides a set of buttons, switches, sliders and the like that can be pasted into a prototype. This prototype can be based on text, spreadsheet, of any other inset that allows embedded objects. For this application, we can use the tiling *lset* object to quickly create eight panels to hold the buttons. The Arbcon (Figure 4) can then be used to create references to new button objects that are simply pasted into place. Each of these buttons has various attributes that can be set via the Arbcon, including a string to place in the button. Since we have already modified text to allow input of ISO strings, we can now use to compose character function to input the desired characters to be displayed. The resulting ’soft keyboard’, next to a sendmessage window can be seen in Figure 5.

We now have an easily modifiable visual prototype on the screen. This prototype is stored as a editable file, not unlike any other file in the system. The next step is to think about how to implement the application. Adew provides a program called createcon that will take one of these prototypes and writes for it a dynamically loadable controller object. This object, when associated with the prototype, will contain pointers to all of the dataobjects and views in the application, and callback routines for all of the buttons, sliders and switches. Normally each button will be given a different call-back, based on the logical name of the object, provided by the prototyper as one of the object attributes. We will use this to set up the bottom two translation buttons, that need only to call the existing procedures in swedish.c. Another mechanism allows the same call-back to be assigned to all objects with the same logical name prior to a postfixed number (example: name_23). In this case , the value of the number will be provided as an argument to the call-back. This provides us with a simple way to implement the remaining keys, by giving each button the same name, followed by the number of the ISO character to insert. When a button is pressed, the call-back procedure will insert the proper character into the sendmessage text. The last problem is one of integration. How does one creating the button panel and controller in such a way that they have a pointer to the sendmessage object.

7.3. Tying it all together

Normally the controller is automatically tied to the prototype by means of a control button, that has, as one of its attributes, the controller's name and a function in the controller to call to initialize it. This is also how the createcon program normally knows what prototype object to create. In this case, we want to use a different initialization method that can be called from *sendmessages* to create the interface.

The first step is to create the controller object by running createcon with the -c flag to indicate the name of the class we wish to create.

```
createcon -c msbcon buttons.lset
```

The result will be a new msbcon.c file, a new msbcon.ch file (to define the new object), an imakefile, and a shellscript (makemsbcon) that will build the Makefile and the application. What we need to add is a proctable entry, not unlike we did for swedish.c above, that can be called from *sendmessage* to create a new window containing the buttons, and a new instance of the controller object. Since this procedure will have pointers to all of the necessary objects, it can be responsible for providing the controller object with the needed pointer to the sendmessage object.

The accompanying code (appendix 2) shows all of the user written code needed to write the application in this scenario. Without comments, it is well under 100 lines of user-written C. While some pieces could have been more straight-forward, we are continually looking for examples like this where new general functionalities should be added to the lower level's of the system.

It could be noted that this is a fairly specialized application, dealing only with sendmessage and the Swedish situation. The philosophy is that graphical interfaces should be easily enough to implement so that they can be specialized for specific applications, or even custom made for specific users. In this case, since Adew separates the prototype from the application itself, one could easily write a controller that bound the button to another application. Another application that provided a graphical interface for adding accents to existing characters would not be difficult.

8. Conclusion

ATK offers a variety of expansion methods, each one appropriate for a certain set of problems. Because these sets overlap, there is often more than one approach to a given challenge. While this can lead to confusion at times, it also lends a large degree of flexibility to the programmer who, with a variety of approaches available, can often choose either the most direct path to a specific problem, or a more general solution to a set of challenges. This ability to support the implementation of fast prototypes, complex long term solutions, and most everything in between is a major strength of ATK. In addition, the modular approach of ATK often means that a solution applied to one application will work on others with no additional implementation effort.

ATK is not the system to end all systems, and not all problems are as easy to deal with as the ones described above. However, it is hoped that some of solutions that ATK has pioneered will make their way into commercial systems. In the meantime, ATK provides a viable alternative for many users who may be dissatisfied with the limitations of the current alternatives.

9. Acknowledgments

Thanks to Rob Ryan who did much of the work of implementing the general solutions described above. The work described in this paper is available as part of the patches to ATK currently being distributed by the ITC.

Thanks also to ..

Christer Bernerus of Chalmers University of Technology for his interaction with regard to swedish.c.

Bill Cattey of Project Athena at MIT for some useful discussions and his work on the troff macros.

Mark Pawliger for his information on foreign language keyboards.

Andy Palay for review and advice.

And all the other implementors, supporters and users of ATK. Thanks folks.

REFERENCES

[Beer90] Beer, Martin D., "Developing Document Management Systems Using the ANDREW Toolkit.," *Proceedings of UKUUG Summer 1990 Conference*, London, July 1990.

[Borenstein88] Borenstein, Nathaniel S., C. F. Everhart, J. Rosenberg, A. Stoller, "A Multi-media Message System for Andrew", *Proceedings of the USENIX 1988 Winter Technical Conference*, Dallas, February, 1988.

[Cattey90] Cattey, William, "The Evolution of Turnin. A Classroom Oriented File Exchange Service," *Proceedings of the USENIX Summer 1990 Conference*, pp. 171-181, Anaheim CA, June, 1990.

[Hansen90] Hansen, Wilfred J., "Enhancing documents with embedded programs: How Ness extends insets in the Andrew ToolKit," *Proceedings of IEEE Computer Society 1990 International Conference on Computer Languages*, March, 1990, New Orleans, IEEE Computer Society Press (Los Alamitos, CA), pp. 23-32.

[Hayden90] Hayden, Charles C., John. C. Mitchell, Jishnu Mukerji, Frederick A. Schmidt, "The Software Development Assistant," *AT&T Technical Journal*, March/April 1990.

[ISO87] Technical committee ISO/TC97, *Information Processing - 8-bit single byte coded graphic character sets*, International Organization for Standardization (ISO), Switzerland, 1987.

[Kitagawa90] Kaz Kitagawa, Private communication, Kieo University, Japan, 1990.

[Neuendorffer89] Neuendorffer, Thomas, "ADEW: The Andrew Development Environment Workbench: An Overview," *Proceedings of the 3rd Annual X Window conference*, Boston, 1989.

[Palay88] Palay, A., W. Hansen, M. Kazar, M. Sherman, M. Wadlow, T. Neueundorffer, Z. Stern, M. Bader, T. Peters, "The Andrew Toolkit: an Overview", *Proceedings of the USENIX Technical Conference*, Dallas, February, 1988.

[Scheifler86] Scheifler, R.W., J. Gettys, "The X Windows System", *ACM Transactions on Graphics 5*, 2 (April 1986) pp. 79-109.

[Williams90] Williams, Nick, William Cattey, "The Educational On-Line System", *Proceedings of the EUUG Spring 1990 Conference*, Munich, April 1990