The Andrew Toolkit: An Overview

Andrew J. Palay, Wilfred J. Hansen, Michael L. Kazar, Mark Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, Thom Peters

Information Technology Center
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

The Andrew Toolkit is an object-oriented system designed to provide a foundation on which a large number of diverse user-interface applications can be developed. With the Toolkit, the programmer can piece together components such as text, buttons, and scroll bars to form more complex components. It also allows for the embedding of components inside other components, such as a table inside of text or a drawing inside of a table. Some of the components included in the Toolkit are multi-font text, tables, spreadsheets, drawings, equations, rasters, and simple animations. Using these components, we have built a multi-media editor, a mail system, and a help system. The Toolkit is written in C, using a simple preprocessor to provide an object-oriented environment. That environment also provides for the dynamic loading and linking of code. The dynamic loading facility provides a powerful extension mechanism and allows the set of components used by an application to be virtually unlimited. The Andrew Toolkit has been designed to be window-system independent. It currently runs on two window systems, including X.11, and can be ported easily to others.

1. Introduction

During the past five years, a number of window systems, including the Macintosh [1], SunWindows [2], the Andrew System [3, 4], X Windows [5], and NeWS [6], have been developed for high-resolution, bit-mapped graphics displays. Each of these systems includes a programmer interface for developing applications. These low-level interfaces provide a simple graphics abstraction, a method for receiving input events and perhaps some simple components of the system, for example, menus, scrollbars, and dialogue boxes. Because the lower-level interface provides few guidelines for the developer, it is difficult to build components that can be used by other developers. Different developers continually replicate the same functional body of code - code that, ideally, should be reused, not needlessly recreated. Together these factors result in confusion and the development of inconsistent applications, leading to chaos in the user community. Because of these problems, higher-level interfaces, such as MacApp [7], SunView [8], the Andrew Base Editor [9], and the X Toolkit [10], have been developed.

The Andrew Toolkit (formerly known as Base Environment 2 or BE2) is a new, high-level environment for the development of user interface applications. Built upon the lessons learned over the past four years during the development of a prototype system at the Information Technology Center (ITC), the Toolkit provides a general framework for building and combining components. It is based on a minimal protocol that allows components to communicate with each other about user interface policies, while allowing the developer maximum freedom to determine the actual interactions between components.

The Andrew Toolkit was built using an object-oriented system called the Andrew Class System. This system provides the ability to dynamically load and link code, which in turn provides a powerful extension capability for applications. Furthermore, the dynamic loading facility can be used to add additional components to the basic Toolkit without having to rebuild applications. This feature has already been used to build a generic, multi-media editor, EZ, that can edit a wide variety of components by loading the appropriate code when needed.

The Toolkit provides the usual set of simple components, such as menus and scrollbars, and a number of higher-level, editable components, including multi-font text, tables and spreadsheets, drawings, equations, rasters, and simple animations. The text and table components are multi-media; they allow the embedding of other components within their description. The drawing component also will soon support this feature.

In addition to the editor, a number of basic applications have been developed, including a mail system [11], a help system, a typescript facility that provides an enhanced interface to the C-shell, a ditroff previewer, and a system monitor, Console, that displays status information such as the time, the date, the CPU load, and file system information. Since both the mail and help applications use the text component for the display of information, they automatically inherit the multi-media functionality of the text component. Example snapshots of these applications can be found at the end of this article.

A number of extension packages have also been developed, including a C-language programming component, a compile package, a tags package, a spelling checker, a style editor, and a filter mechanism.

2. Design of the Andrew Toolkit

The original concept of the Andrew Toolkit came from discussions about the development of a text editor that would allow the user to embed and edit other components, such as tables, drawings, and rasters. Some of the problems inherent in building such a system include:

how to handle input of events between components.

how to arbitrate the display of menus.
how to arbitrate the display of the mouse cursor.
how and when to display components.
how to determine the size and placement of embedded components.
how to store the external representation of components.

In examining these problems, a general architecture was developed that allows the inclusion of one component inside another. This allows the developer to build components that can embed other components without detailed knowledge about what is being embedded. Furthermore, new components can be easily embedded in those that already exist without any additional work.

For example, users of the Andrew System can currently compose documents that contain tables, equations, drawings, rasters, and animations. The generic mechanism used to embed one component within another can also be used to embed any components developed in the future. This is an important feature of the system, especially within a university environment. Given our limited resources, we knew from the outset that we could not write all the components desired by different parts of the university. Members of the music department would want to include musical scores inside text just as easily as members of other departments include tables. Members of the electrical engineering department might want to include circuit diagrams inside text. The list is limitless.

The dynamic loading feature is essential to extending the Toolkit’s functionality. If a member of the music department creates a music component and embeds that component into text, the code for the music component is dynamically loaded into the application. Except for a slight delay to load the code, the user of the editor is unaware that the music component was not statically loaded or a part of the original system. The editor did not have to be recompiled, relinked, or otherwise modified to use the new music component. In addition, all users of the text automatically acquire the ability to use the music component; it can be sent in a mail message as easily as it can be edited in a document.

While the Toolkit currently runs under both the original ITC/Andrew Window System and under X.11, of necessity it has been designed to be window-system independent, and, to a great extent, operating system independent. Within the user community, there exist a wide range of machines and window systems that need to be supported. Within the UNIX operating system community, the X Window System is developing into a standard, although other window systems may eventually rival it. Finally, there is the constant development of new operating systems, such as OS/2, for machines between the low-end personal computers and the high-function UNIX workstations.

Since it was impossible to foresee the development of a window system standard over that range of machines, the Andrew Toolkit was made window-system independent. This allows the potential to support a consistent set of applications over a diverse set of window systems. Clearly, the personal computer versions will be more limited, but we consider it a great advantage to be able to use a low-cost machine for simple tasks, then move easily to more powerful machines when required.

3. Basic Toolkit Objects: Data Objects and Views

The Andrew Toolkit is based on the development of components that can be used as building blocks for applications as well as more complex components. Data objects and views are the Toolkit's basic object types; a Toolkit component is usually made of a view/data object pair. While the data object contains information that is to be displayed, the view contains information about how data is to be displayed and how the user is to manipulate the data object. For example, the text data object contains the actual characters, style information, and pointers to embedded data objects, as well as ways to alter the data, such as inserting and deleting characters. The text view is made of information such as the location of the text, what portion of text is currently visible, and what piece of text is currently selected. The text view provides ways to draw the text, handle various input events, and manipulate the visual representation of the text.

The contents of a data object can be saved in a file, but the contents of a view cannot. The information associated with a view is transient and is valid only while the application is running. When the application terminates, that information has no further meaning. However, views do provide the facility for printing within the Andrew Toolkit.

While it is often the case that a view has an underlying data object, there are many cases when a view is used solely to provide a user interface function. In such a case, there is no underlying data object. The scrollbar is one such example; it only adjusts the information contained in another view.

The view/data object distinction has been made to provide a system where multiple views can simultaneously display information contained in a single data object. The design is similar to the Model-View-Controller design used in Smalltalk [12] systems. By comparison, our data objects serve as models, our views are views, and the controller is distributed between the interaction manager (for global decisions) and the individual view (for decisions between children and parent views).

This separation of concerns has brought us many advantages. For example, in building an editor, we wanted to provide the user with the ability to edit the same information in more than one window. Furthermore, we wanted the changes made in one window to be reflected in the other. This case is handled by having two views of the same type, one in each window, displaying information from the same data object. Similarly, we might want to have multiple views of the same type on a single data object in one window. A system like Aldus' PageMaker (TM) could be built under the Andrew Toolkit by allowing the user to specify a set of views and their placement on a page. Some of those views (for example, the text views) would be examining different sections of the same data object.

It is also possible to have two different types of views displaying information contained in a single data object. Currently the text view is a display-based text processing system. It can be characterized as a semi-WYSIWYG or a WYSLRN view. It displays text with multiple fonts, indentations, etc., but makes no attempt to display the information as it would appear on a piece of paper. This view has been used for the basic text editor as well as the mail and help systems. It has been successful, except in cases where the user wants to format the text for printing. In this case, we plan on providing a full WYSIWYG text view. This paper-based text view will be designed to use the same text data object. The user of the system will be able to choose to use either view, or perhaps have one window using the normal text view and the other using the WYSIWYG text view. Again, changes made in one window will automatically be reflected in the other window.

Just as it is possible to have two different views on the same data object in two windows, it is possible to have two different views on the same data object within the same window. A text component could have two embedded views on the same data object. For example, the user might want to display a table of numbers and a pie chart representing the table. This could be done by having one table data object and two views, a normal table view and a pie chart view.

Despite its advantages, there is a cost to separating information into data objects and views. Two particular areas of difficulties are coordinating data objects and views, and maintaining a stable view state. Each is briefly discussed below.

Our system does not encourage a close connection between the changing of the information contained in a data object and the update of the visual appearance provided by the view. Since only one view causes the data object to change, and multiple views may have to reflect the change, a delayed update mechanism must be used. When the user issues a command to a view to alter the underlying data object, the view first requests that the data object modify itself, then requests that the data object inform all its views of the change. When a view is informed that the underlying data object has changed, it must determine what the change is and appropriately update its visual image.

The delayed update mechanism is the trickiest challenge in building a data object/view pair. The developer must create a mechanism with which the view can determine what portion of the data object has changed. This method is normally provided by a set of methods exported by the data object. However, it is not considered proper for the data object to have detailed knowledge of a specific type of view. While this is one way to handle the delayed update, it would preclude the development of other kinds of views on the same type of data object.

The second difficulty we faced was retaining the stable (or permanent) state for a view. In the example of the table and the pie chart discussed earlier, the underlying data object is a table of values. When a file displaying a chart is saved, only those values, and the information that a ‘‘chart’’ is viewing the table, is saved. However, the user may have set certain parameters in the chart, such as the way to label the axes. This information is not part of the table data object, and is not stored in it. Since a view has no permanent state, information kept in the view, such as axis labelling, is not saved. In its simplest form, there is no place to keep this view-specific information.

Our solution to this problem consists of two parts: additional data objects and the concept of an observer. In the example above, the chart view would be viewing not a table data object, but an auxiliary chart data object. The chart data object retains information such as axis labelling, and acts as an observer of the table data object. As information in the table changes, the chart data object is notified; it, in turn, notifies the chart view. There are no specifically defined auxiliary data objects. Rather our update system is based on the observer mechanism, where a data object may be observed by any number of other data objects and views. This design has permitted us a great deal of flexibility and functionality for combining pieces in the Toolkit.

4. Event Processing: The View Tree

Views are used to define the user interface for an application. In defining a user interface, the view must handle both the visual display of information on the screen and the handling of input events that might change the display.

Views are organized in a tree structure. Visually, each view is a rectangle and is completely contained in its parent view. At the top of the tree is a view called the interaction manager. The interaction manager has the responsibility of translating input such as keystrokes, mouse events, menu events, and exposure events from the window system to the rest of the view tree. The interaction manager is also responsible for synchronizing drawing requests between views. By design, it has one child view, of arbitrary type, that may have any number of children. Child views are always visually contained within the screen space allocated to their parent, but the Toolkit does not define any screen relationship between sibling views.

In general, when an event is received by the interaction manager, the event is passed down to the interaction manager’s child. That view determines if it is interested in the event, or if it should pass the event down to one of its children. This process recurs until some view actually handles the event. By passing the event down the view tree, each parent gets the chance to determine the disposition of the event, and can use the semantic information associated with itself to make that determination.

Updates to the visual image of an application are handled in a similar fashion. When a view wants an update to its image, it makes a request to its parent view. This request is usually passed up to the interaction manager, which then sends an update event back down the tree. Since one view might be embedded in another, it does not have complete control over its allocated screen space. For example, the parent might have overlaid some other image on top of the child’s image. By posting an update request up the tree and having the update event come back down, the parent can update its own image and the children’s images in the appropriate order.

The following figure presents a view tree for a window that contains a scrollable text view that contains a table view. The text view is surrounded by a scrollbar, which is surrounded by a frame. The frame provides a message line view. The text and table views reference their respective data objects. The lines around the screen image represent the physical area of the image associated with each view.

scrollbarwidth = .3i messagelineheight = .2i

down WS: box "Window" "System" arrow dashed <-> IM: box "Interaction" "Manager" arrow <-> FR: box "Frame" SB: box "Scroll Bar" at FR - (1i, 1i) arrow <-> TV: box "Textview" arrow <-> TB: box "Tableview" ML: box "Message" "Line" at FR + (1i, -1i) arrow <-> from FR.sw + ((FR.se.x - FR.sw.x) / 3, 0) to SB.n arrow <-> from FR.se - ((FR.se.x - FR.sw.x) / 3, 0) to ML.n

ellipsewid = 0.85i down TDO: ellipse "Text" "Data Object" with .w at TV.e + (movewid, 0) arrow <-> TBDO: ellipse "Table" "Data Object" arrow <-> from TB.e to TBDO.w arrow <-> from TV.e to TDO.w

imagewidth = ML.e.x - SB.w.x imageheight = WS.e.y - TB.e.y tableheight = imageheight / 3 tablewidth = imagewidth * 2 / 3

SD: box ht imageheight wid imagewidth with .ne at (SB.w.x - movewid, WS.e.y) SBI: box ht imageheight - messagelineheight wid scrollbarwidth with .nw at SD.nw MLI: box ht messagelineheight wid imagewidth with .sw at SD.sw DATE: "February 11, 1988" ljust at SD.ne - (1.25i, .5i) GREET: "Dear David," ljust at SBI.ne + (.25i, -1i) TEXT1: "Enclosed is a list of our expenses ..." ljust at GREET - (0, .25i) TABLE: box ht tableheight wid tablewidth at TEXT1.s - (- tablewidth / 2 - 0.1i, tableheight / 2 + .25i) TT: box ht tableheight / 10 width tablewidth with .nw at TABLE.nw right box ht tableheight * .6 width tablewidth / 3 with .nw at TT.sw box same box same box ht tableheight * .3 width tablewidth / 3 with .sw at TABLE.sw box same box same TEXT2: "Hope you have a nice ..." ljust at (TEXT1.e.x, TABLE.sw.y - .25i) SAL: "Sincerely," ljust with .e at (DATE.e.x, TEXT2.s.y - .25i) SIGN: "David" ljust at SAL.se - (0, 0.5i)

dashwid = 0.02i IMBOX: box dashed ht SD.ht + 0.2i wid SD.wid + 0.2i with .nw at SD.nw - (.1i , -.1i) FRBOX: box dashed ht SD.ht + 0.1i wid SD.wid + 0.1i with .nw at SD.nw - (.05i , -.05i) SBBOX: box dashed ht SBI.ht - 0.05i wid SD.wid - 0.05i with .nw at SD.nw + (.025i , -.025i) TXBOX: box dashed ht SBBOX.ht - 0.05i wid SBBOX.wid - scrollbarwidth - 0.05i with .nw at SBI.ne + (.025i, -.05i) TBBOX: box dashed ht TABLE.ht + 0.05i width TABLE.wid + 0.05i with .nw at TABLE.nw - (0.025i, -0.025i) MLBOX: box dashed ht MLI.ht - 0.05i wid MLI.wid - 0.05i with .nw at MLI.nw + (0.025i, -0.025i)

arrow from IM.w to (IMBOX.e.x, IM.w.y) arrow from FR.w to (FRBOX.e.x, FR.w.y) arrow from SB.w to (SBBOX.e.x, SB.w.y) arrow from TV.w to (TXBOX.e.x, TV.w.y) arc -> cw from TB.w to TBBOX.se - (0.25i, 0)

When a mouse event is encountered by the interaction manager, it passes the event down to the frame view. The frame determines if it should handle the mouse event directly, or if it should pass the event down to either of its children. If the mouse event is close to the dividing line between its two children, the frame accepts the mouse event directly. (In this case the user may adjust the position of the dividing line.) If the mouse event is passed down to the scrollbar view, that view will accept the mouse event if the event is over the scrollbar; otherwise the view passes the mouse event to its children. The text view accepts the mouse event if the event is not in any of the text view's subviews; otherwise it too passes the event down.

As the mouse event works its way down the view tree, the view determining the disposition of the mouse event only needs to be aware of the location of its children, not the child's type. Similarly, the child does not need to know its parent's type, or the location in the overall view tree.

The controlling relationship between a view and its children is one area in which the Andrew Toolkit differs from other toolkits. Other systems closely tie the handling of events to the physical relationship of components on the screen; if one component is physically on top of another, the top component blocks the transmission of certain events to lower components. Although this is valid in many circumstances, there are times when it is not. Further, many toolkits use a global analysis of all views in order to process and distribute events. The Andrew Toolkit distributes this authority to each view over its children.

Our Toolkit was designed to overcome the limitations that we had experienced with a global, physical model. An early prototype toolkit, the Andrew Base Editor, tied the handling of events to the physical relationship of components on the screen. During the early design phase for the Andrew Toolkit, we attempted to use this prototype to build a drawing editor that used the text component to display and edit text within the drawings. The text component was a subordinate of the drawing component. The user of the drawing editor might first enter some text, then place a line over the text. When a mouse event occurs near that line, only the drawing component could determine whether the user was selecting the line or the underlying text. This had been impossible to accomplish because the toolkit maintained a strict, global control over the distribution of input events.

A similar case can be seen in the way the frame view handles mouse events. The frame physically divides its image into two areas, which are separated by a thin line. To allow the user to easily drag this line, the frame allocates a slightly larger area to accept mouse events. That area overlaps the space allocated to the frame’s children. If event handling were dictated by screen layout, this interaction would be much more difficult to provide, and would require detailed knowledge of which view tree structure to maintain.

The parental authority is a major architectural concept in the Andrew Toolkit. The same authority for discussing mouse events, discussed above, is used between children and parents to negotiate the contents of menus, the display of cursors, the mapping of keyboard symbols, and the focus of attention.

5. The Graphics Layer

The view tree provides a general mechanism for handling events in the Andrew Toolkit, which hides from the developer the specifics of the input model used by the underlying window system. In a similar fashion, the Toolkit uses a graphics layer to hide the output model of the specific display medium. This display medium is usually the underlying window system, but it can also be a printer.

The graphics layer is built using a third type of object, the drawable. A drawable contains information about the underlying graphics medium. In a window system, that information usually includes:

> the window to draw in.
> the location of the drawable in that window. a
> small graphics state (e.g. current point, line thickness, current font).
> the coordinate system for the drawable.

The drawable provides a set of drawing operations similar to those provided by the X.11 window system.

Each view contains a pointer to a drawable, which is used for all drawing operations. The developer of a view rarely accesses a drawable directly. All methods exported by the drawable are also provided as part of the view interface.

Separating the view and the drawable allows us to provide a simple default printing mechanism. When a view receives a print request for a specific type of printer, it can temporarily shift its pointer to a drawable for that printer type and do a redraw of its image. We expect to provide this facility in a later version of the Toolkit.

6. External Representation

Most of the problems with embedded components are solved by the view interface, except for the external representation of the components. As stated earlier, only data object descriptions are written out to files. The Toolkit architecture places one requirement on the external representation. When a data object writes out its external representation, it is enclosed in a begin/end marker pair. The markers must be properly nested, and it must be possible to find all the data associated with an object without actually parsing the data. Those markers provide a tag denoting the type of the object being written and an identification tag that can be used for referencing the data object by other data objects.

Thus the earlier example containing a table embedded in text would have an external representation that looks like this:

```
\begindata{text, 1}
text data ...
\begindata{table, 2}
the table data goes here ...
\enddata{table,2}
more text data ...
\view{spread, 2}
rest of text data ...
\enddata{text,1}
```

The \view construct is specific to the text object and indicates the exact placement within the text of the view (of type spread) on the table data object.

The use of nested begin/end markers is the only requirement placed by the Toolkit architecture. For external representation we also strongly encourage developers to follow these guidelines:

use only printable, 7-bit ascii characters (including tab, space, and newline).
keep line lengths below 80 characters.

make the representation understandable.

The first two suggestions make it possible to transport files across almost all networks, especially as mail. The final suggestion is an attempt to provide an easier recovery mechanism in the rare case when files are partially destroyed. This suggestion only makes sense in the context of the first two suggestions. If the file is stored using only 7-bit codes, with line lengths under 80 characters, then the overhead in making the file understandable is small. Of course there are some objects, such as rasters, where this requirement is impossible to meet. However, even in those cases it is possible to make the file slightly more comprehensible. For example, the raster format could make sure the bits representing a new row always begin on a new line.

7. The Object-Oriented Environment

The Andrew Toolkit is built using the Andrew Class System (Class). Class provides an object-oriented environment with single-inheritance. The Class language permits the definition of object methods and class procedures. Object methods are similar to C++ [13] methods, and may be overridden in subclasses. Class procedures are similar to Smalltalk’s class methods, only they may not be overridden. C procedures for controlling the initialization and disposition of objects are created by the Class preprocessor. Class also provides for the dynamic loading and linking of code.

Class is a C language-based system. It consists of a small run-time library and a simple preprocessor that only preprocesses Class header files. Class header files are almost identical to standard C header files, except for the inclusion of the class definition. The preprocessor generates two include files, an export file (.eh) used when defining a class, and an import file (.ih) used when referencing a class. The C files written using Class look similar to normal C files, and are not run through a special preprocessor.

Class is similar in nature to C++. We chose to implement our own system for several reasons:

> We wanted to support dynamic loading and linking. C++ generates code that must be statically linked. We considered modifying the C++ preprocessor, but deemed it impractical without having changes made in the official version. We made some initial inquiries, but could not solve this problem quickly enough for development to continue.
>
> We wanted to develop a system that could be debugged easily. C++ preprocesses both include files and source files. Without a debugger that understands the original C++ code, developers have to understand the transformations made by the C++ preprocessor. This poses only small problems for the highly-experienced developer, but would cause problems within the university community.

> We wanted the system to be as simple as possible. We needed an object-oriented environment, but not the other constructs built into C++.
>
> We wanted to be free of any external dependencies that require yet another licensing agreement. Because we hoped to make the Toolkit widely available, requiring another licensed product seemed to be a bad idea.

Even though we did not use C++ to implement the Toolkit, the Class system used C++ as a model for its object-oriented environment. If the above problems were solved (which might now be a possibility), converting to C++ would be a relatively easy, but time-consuming, activity.

8. Extending the Toolkit

A major feature of the Andrew Toolkit is its extensibility. The system has been designed so that the creator of a data object or view does not have to take any special action to allow that object to be embedded in another object. The data object and view interfaces have been designed to provide the necessary set of methods for two objects to communicate without detailed knowledge of each other. Furthermore, those interfaces make it easy for the author of an object to allow it to include other objects. The text object can include any other type of data object. Authors of new objects are strongly encouraged to handle the inclusion of arbitrary objects instead of special-casing the inclusion of specific objects.

The dynamic loading and linking feature also provides a low-level extension language for applications built using the Toolkit. Sophisticated users can use the Class system to write code to implement new commands. These commands can be bound either to key sequences or to menus; when invoked, the code is loaded and executed.

This feature has also given us the ability to run all our applications from a single base program. We have created a program called runapp that contains the basic components of the Toolkit, and dynamically loads in the code for each individual application at run time. Since most UNIX systems do not provide shared libraries, runapp allows multiple Toolkit applications to share a significant portion of code. This leads to performance improvements in many areas:

> paging activity is reduced.
> key portions of code are almost always paged in,
> thus improving user performance.
> virtual memory use decreases.
> file fetch time decreases if running under a distributed file system.
> file size of an application is reduced.

9. Window System Independence

The Andrew Toolkit has been designed to be window-system independent. To port the Toolkit to another window system, six classes must be written, encompassing approximately 70 routines. Of those routines, about 50 are normally simple transformations to the graphics layer of the underlying window system. Once those are written, any Toolkit application should run in the new environment. The six classes that must be written are:

> *Window System*: this class exists to allow the Toolkit to get a handle on the other window system classes listed below.
>
> *Interaction Manager*: this class provides the interface to the event processing mechanism from the underlying window system. This includes the handling of keystrokes, mouse events, and menus.
>
> *Cursor*: this class provides an interface to defining cursors on the underlying window system.
>
> *Graphic*: this class provides the output interface to the underlying window system. All drawing operations are made using this class.
>
> *FontDesc*: this class provides an interface to font descriptions.
>
> *Off Screen Window*: this class provides the facility to draw off screen images that can later be included on screen.

Using this facility we are currently able to run applications on two different window systems without any recompiling. Applications are normally configured for one system. However, using the dynamic loading facility, the modules for the other system can be loaded at run time. The choice of a window system is currently controlled by setting an environment variable. With a little more restructuring of the basic code, we believe that it will be possible to actually open windows on two different window systems at the same time.

10. Current Status

The basic Toolkit applications (EZ, Messages, Help, Preview, Typescript, and Console) have been in general use on the Carnegie Mellon campus since August 1987 using the original Andrew window system. The Andrew System is actively used by approximately 3,000 people, some of whom are beginning to experiment with the multi-media facility which has only recently been advertised.

One of the challenges associated with building user-interface software is to make the system easy to use for the novice user, while making it powerful enough for experienced users. The prototype editor built at the ITC was highly influenced by the goal of making it easy for the novice user. While we were developing that system and until the release to the ITC of EZ, programmers at the ITC used Emacs to edit programs. Since the release of

EZ, the use of Emacs has dramatically decreased. This has been accomplished without sacrificing the usability of the system by our campus user community.

11. Conclusion

UNIX, with its software-tools approach to computing, provided a new paradigm for application building. The idea was that portable, general-purpose modules could be strung together in different ways to create complex applications without great duplication of effort. The Andrew Toolkit is an extension of this concept to the graphic workstation environment.

As an extension of the software-tools approach to computing, the Andrew Toolkit is unique among toolkits. It provides the usual toolkit functions, such as text, scrollbars, and dialogue boxes, but also provides the ability to embed components within each other. The architecture for embedding components has been designed to strongly encourage programmers to build new components that can be used in both new and existing applications. This architecture also encourages programmers to develop objects that can include arbitrary components instead of specific ones.

The separation of information into data objects and views provides a highly modularized structure that also supports the building-block paradigm. In this way, data objects can be used in ways different than those originally envisioned by their creator. New views on existing data objects can be created. Existing data objects can also be used as the building blocks for more complex objects. This can be done without using the existing object’s companion view.

The Andrew Class System is an essential element in supporting this paradigm. The object-oriented nature of the system allows programmers to easily develop new, specialized objects out of existing objects. The dynamic loading facility of Class allows the Toolkit to be easily extended by a large community of developers.

Finally, the Andrew Toolkit is unique among toolkits in its potential to provide a common base of applications across a diverse set of machines and window systems. We have spent considerable effort making the Toolkit window-system independent, and believe that it will be important in the future to support the same software base on many systems.

Acknowledgments

The Andrew Toolkit and applications have been designed and developed over the past two and a half years. Many other people have been involved in various phases of the work including Andrew Appel, Nathaniel Borenstein, Mark Chance, Richard Cohn, Curt Galloway, John Howard, Tom Lord, William Lott, Bruce Lucas, David Nichols, Marc Pawliger, and Adam Stoller. The toolkit could not have been built without the earlier work done at the ITC by James Gosling and David Rosenthal. Documentation for the toolkit has been written by Chris Neuwirth and Ayami Orgura. Help in preparing this document was provided by Linda Branagan, Kristine Subasic Nichols, David Tilbrook, and Jaap Akkerhuis.

We are also grateful to our user community, especially our fellow workers at the ITC, who have been forced to be the front-line testing organization for our software over the past four years.

The ITC is a joint project between IBM and CMU. The ITC also receives support from the National Science Foundation. Development and deployment of the toolkit would not have been possible without the support of these organizations.

Snapshots

1. Snapshot of a full screen image containing a *console*, a *typescript*, and two *ez* windows. The two *ez* windows show editting a document and a C file.

figure pic1.ps

2. Snapshot of a *help* window. Users can see overview information by clicking in the *Overviews* panel on the right. Similarly they can see specific information about a program by clicking in the *Programs* panel.

figure pic2.ps height 2.5i

3. Snapshot of *messages reading* window. The panel on the left gives a list of message folders that can be read. It currently contains a list of all the messages folders available on campus. It can also be set to display the folders a user is subscribed to or just the user's personal folders. The panel at the top left contains the list of messages in the selected folder. The message being displayed contains a drawing within the text of the message.

figure pic3.ps height 4i

4. Snapshot of *messages composition* window. The message being created contains a raster image.

figure pic4.ps height 3.75i

5. Snapshot of an *ez* window containing a number of embedded objects (text, equations, and an animation) within a table that is contained inside of text.

figure pic5.ps height 4i

References

[1] ‘‘Inside Macintosh’’, Addison-Wesley, 1985.

[2] Sun Microsystems, Inc., ‘‘SunWindows System Programmers Guide’’.

[3] Morris, et al., ‘‘Andrew: A Distributed Personal Computing Environment’’, *Communications of the ACM*, March 1986.

[4] James Gosling and David Rosenthal, ‘‘A network window manager’’, *Proceedings of the 1984 Uniform Conference*, Washington, D.C., January 1984.

[5] Scheifler, R.W., and Gettys, J., ‘‘The X Window System’’, *ACM Transactions on Graphics*, Vol 5, No. 2, April 1986, pp79-109.

[6] Sun Microsystems Inc., *NeWS Manual*, 800-1632-10, March 1987.

[7] Schumucker, K., *Object-Oriented Programming for the Macintosh*, Hayden Book Company, Hasbrouck Heights, NJ, 1986.

[8] Sun Microsystems, Inc., *SunView Systems Programmers Guide*.

[9] James Gosling and David Rosenthal, ‘‘The User Interface Toolkit’’, *Proceedings PROTEXT 1 Conference*, 1984.

[10] ‘‘X Toolkit Library - C Language Interface’’, Massachusetts Institute of Technology and Digital Equipment Corporation, 1987.

[11] Borenstein, et. al, ‘‘A Multi-media Message System for Andrew’’, *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume).

[12] Aldus Page Maker - Just note Trade Mark.

[13] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[14] Adele Goldberg and David Robson, *SmallTalk-80: The Language and Its Implementation*, Addison-Wesley, 1983