CONNECTIONS IN CONTEXT:
The Intermedia System

Nicole Yankelovich, Bernard J. Haan and Steven M. Drucker

Institute for Research in Information and Scholarship (IRIS)
Brown University
Box 1946
Providence, RI 02912

Abstract

Intermedia is a hypermedia system developed for use in university research and teaching. It provides a framework for object-oriented, direct manipulation editors and applications, and the capability to link together materials created with those applications. Instructors are able to construct exploratory environments for their students as well as use applications for their day-to-day work, research and writing. Building such an interactive user environment containing multiple applications and user-level utilities requires full, effective cooperation among programming team members in order to produce a system with an entirely consistent user interface.

This paper presents a general discussion of hypermedia followed by a description of the Intermedia system. The description focuses on several important user interface features and illustrates the operation of the system through a sample session. The final section of the paper explores how object-oriented programming techniques and a complementary set of software development tools helped programmers work together and maintain a consistent user-interface throughout the system.

1. Introduction

At Brown University, we have both formally and informally examined the way faculty, especially in the humanities, teach with currently available tools and technology. We observed that university instructors do not "tutor" students nearly as much as they try to guide them through bodies of material, help them to analyze and synthesize that material, and encourage them to make connections and discover meaningful relationships.

We have kept these methods in mind, therefore, while formulating the requirements for software systems appropriate for the scholarly community. We have focused our efforts on building tools for synthesizing, arranging, indexing, connecting, sharing, customizing, visualizing, integrating and retrieving information.

This paper describes Intermedia, one such tool designed for use in the university environment. The system, developed at Brown University's Institute for Research in Information and Scholarship (IRIS), contains multiple applications and provides mechanisms for linking together the contents of documents created with those applications. The bulk of this paper describes the linking, or hypermedia, functionality provided by Intermedia and the user interface techniques designed to support that functionality.

Hypertext and Hypermedia

In the early 1960's, Theodor Nelson coined the word hypertext to describe the idea of non-sequential writing. A hypertext system is one which allows authors or groups of authors to link information together, create paths through a corpus of related material, annotate existing texts, and create notes that point readers to either bibliographic data or the body of the referenced text. With a computer-based hypertext system, students and researchers are not obliged to search through library stacks to look up referenced books and articles; they can quickly follow trails of footnotes without losing their original context. Explicit connections — links — allow readers to travel from one document to another, automating what one does when following references in an encyclopedia. In addition, hypertext systems that support multiple users allow researchers, professors and students to communicate and collaborate with one another within the context of a body of scholarly material. For a survey of early hypertext systems refer to [Conk86 and Yank85].

Hypermedia is simply an extension of hypertext that incorporates other media in addition to text. With a hypermedia system, authors can create a linked corpus of material that includes text, static graphics, animated graphics, video, sound, and so forth. Examples and descriptions of existing hypermedia systems can be found in [Back82, Bend84, Brow87, Conk87, Fein82, Good87, Hala87 and Weye85].

2. The Intermedia System

Intermedia is both an author's tool and a reader's tool. The system, in fact, makes no distinction between types of users providing they have appropriate access rights to the material they wish to edit, explore or annotate. Creating new materials, making links and following links are all integrated into a single modeless environment.

The Applications

Five applications currently exist within the Intermedia framework: a text editor (InterText), a graphics editor (InterDraw), a scanned image viewer (InterPix), a three-dimensional object viewer (InterSpect), and a timeline editor (InterVal). Any number of documents of different types may be open on the desktop at one time along with the folders containing the documents. These applications conform as closely as possible to the Macintosh' interface standards detailed in [Appl85]. Both programmer-level tools and well-defined user interface concepts contribute to the high degree of consistency exhibited across all Intermedia applications.

The InterText word processing application is similar to Apple's MacWrite [Appl84a] with the addition of style sheets for formatting text rather than MacWrite-style rulers. Using style sheets, the user can define a set of styles for a particular document (e.g., paragraph, title, subtitle, indented quote, numbered point) and apply those styles to any unit of text between two carriage returns, called an entity. When the user edits the definition of a style, all the entities to which that style is applied reformat accordingly.

With InterDraw, a structured graphics editor similar to Apple's MacDraw [Appl84b], users can created two-dimensional illustrations by selecting tools from a palette attached to each InterDraw window. InterPix is a utility program that displays bitmap images entered into the system using a digitizing scanner. These images can be cropped, copied and pasted into InterDraw documents. The InterPix application is being extended to provide full bitmap editing capabilities.

Like InterPix, InterSpect is a viewer rather than an editor. It converts files containing three-dimensional data points into three-dimensional representations of that data. Users can manipulate the three-dimensional image by rotating it, zooming in or out, or hiding parts of the model.

The fifth application, InterVal, provides interactive editing features for creating chronological timelines. As the user enters pairs of dates and labels, the application formats them on a vertical timeline according to user-defined styles. Like a charting package, the display of the data is determined by a certain set of parameters which the user can modify.

Figure 1, taken from a corpus of material created for an English literature course [Land86], shows two InterText documents, two InterVal documents and two InterDraw documents open on the screen. Both InterDraw documents contain scanned images cropped, copied and pasted from InterPix documents.

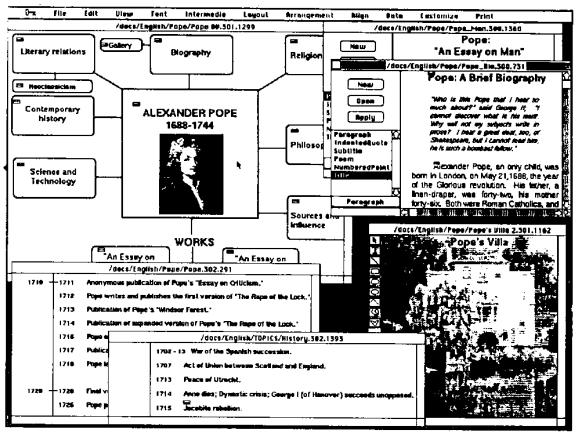


Figure 1. Example of materials from an Intermedia corpus designed for an English literature course.

Hypermedia Functionality

The hypermedia functionality of the Intermedia system is integrated into each application so that the actions of creating and traversing links may be interspersed with the actions of creating and editing documents.

The act of making links between Intermedia documents has been modeled as closely as possible on the Smalltalk/Macintosh copy/paste paradigm [Appl85, Gold84] in an effort to fit the link-making process into a conceptual model already familiar to users. If links are to be made with frequency, they must be a seamless part of the user interface. The smooth integration of this new function into an already ingrained user interface model is apparent in the sample session following this section.

Unlike some other hypertext or hypermedia systems that only allow links to be made to entire documents [Good87, Know86 and Shne86], Intermedia allows users to create links from a specific location in one document to a specific location in another document. These "anchor points" in the documents are called blocks. In designing the Intermedia linking functionality, one of the development team's design goals was to allow anything that could be selected to be made into an anchor for a link. The size of a block, therefore, may range from an entire document to an insertion point, depending on the selection region a user identifies as the block's extent. For example, in an InterText document, a block might consist of an insertion point, a single character, a word, a sentence, two paragraphs, and so forth. Small marker icons are placed near the source and destination blocks to indicate the existence of a link (Figure 2).

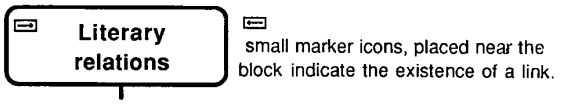


Figure 2. An arrow enclosed in a box signals the existence of a link.

To help manage a large corpus of linked documents, links and blocks are assigned a set of descriptive properties. Some of these, like user I.D. and creation time, are assigned automatically while other properties are user-defined. Users access and edit link and block property information through property sheet dialogs. These dialogs contain a field for the user to enter a one line "explainer," similar to the subject field in some electronic mail programs. From a reader's perspective, link explainers are particularly important. If a single block has more than one link emanating from it, users choose the path they wish to follow from a list of link explainers presented in a dialog box.

Property sheets also contain fields for adding keywords. Although still under development, these keywords, along with the default information assigned to links and blocks, will provide users with a mechanism for searching the document corpus. The result of a keyword search will yield a list of explainers associated with all the blocks or links meeting the search criteria. Each item in the list will automatically be linked to its corresponding block, or, in the case of links, to the corresponding source block of each link. For example, a student would be able to search for all the links created by the professor after a certain date that contain the keywords "Pope and Heroic Couplet."

Some users may want to enter both block and link properties during the link creation process, while others may not want to take the time to fill them in at all. By default, property sheets are not automatically presented to the user. Intermedia provides a Viewing Specifications dialog as a utility for setting user preferences. To change the default behavior, users alter settings in the Viewing Specifications dialog, turning link and/or block creation to "verbose." If users have created links with the "fast" rather than the "verbose" settings, they can still edit block and link properties after a link has been established by selecting a marker icon and choosing the "Link Properties" or "Block Properties" command from the menu.

Link and block properties help to manage complexity within the Intermedia environment, but the notion of context is even more crucial. In some systems, links are global; all links are available at all times to all users. In such systems, links become an integral part of the documents. In Intermedia, block and link information is not stored within individual documents, but rather is superimposed on them. *Webs* are provided to maintain the block and link information, allowing users to work within their own context undistracted by documents, blocks and links created by others sharing the same computing resources. In the future, webs will also serve as the focus for filtering operations. Currently, opening a web causes a particular set of blocks and links to be imposed on a set of documents while that web is opened. Thus, webs allow different users to impose their own links on the same document set. For example, an academic department might purchase all of Shakespeare's works in electronic format. Rather than duplicating every work each faculty member wants to link to, by introducing webs, Intermedia allows users to all create links to the same documents, without having to see each other's links. Although only one context may be viewed at a time, users can easily switch contexts by closing one web and opening another. Of course, if users do want work together, any number may share a single web.

In addition to webs, Intermedia also has a system of user access rights that help to manage multiple users sharing large bodies of connected material. Due to the hypermedia functionality of Intermedia, the access right scheme is slightly more complex than in other UNIX environments where users either have read permission or write permission to files and directories. Intermedia adds annotation permission to the other two forms of access rights. This allows students to add links to a document which they are not allowed to edit.

3. Common User Interface Concepts

Several user interface concepts stressed throughout Intermedia enable users to learn new applications quickly and predict the behavior of features they have never used before. Like the copy and paste operations in Macintosh and Smalltalk programs, some operations in the Intermedia system behave identically across all applications. The linking functionality is a prime example. In any document, users can specify a selection region and choose the "Start Link" command from the menu, and in any other document, regardless of type, users can define another selection region and choose one of several "complete link" commands. Likewise, to follow a link, a user exploring a web can select a marker icon in any type of document, and choose the "Follow" command from a menu. As a short cut, a user can "double-click" on a marker icon to initiate the follow, just as he or she might double-click on an icon in a folder to open a document. Since following a link usually entails opening a document, we anticipated that users would expect to be able to follow a link by double-clicking on the marker icon. In a system that encourages rapid transitions between applications, it is essential to limit the amount a browser must learn in order to be a successful user of the system and to capitalize on those conventions he or she may already be familiar with.

Other features in the system, while not exactly identical to one another, are conceptually similar. Most applications, for instance, allow users to control the format or the display characteristics of data. The interface techniques for conceptually similar operations have been designed to capitalize on the likenesses. One such example is the style paradigm [Inte86, Smit82]. Styles are sets of properties or characteristics that govern the appearance of data within a document. Users can define or modify a style by filling out or editing a "form" called a style sheet (sometimes refered to as a property sheet). Both the InterText application and the InterVal application contain style sheets which can be used to specify different text formats such as paragraphs, indented quotes, lists, and titles, or different timeline formats such as position of dates relative to tick marks and position of labels relative to dates. In the

graphics editor, different styles may be applied to shapes such as line width, pen style, or fill style. Although not yet implemented, InterDraw will also allow users to "borrow" the style of one shape and "apply" it to any others. This means that the combination of line width, pen style and fill style can be defined once and then applied as a unit to any existing objects or to any new objects. The concept of styles will also be included in the InterSpect three-dimensional viewer in the future. In this case, users will be able to define a set of viewing parameters that specify a particular camera position. Just as styles in the text editor do not alter the content, but simply the presentation, of the text, styles in InterSpect will be used to define the manner in which the data is viewed. By storing all presentation parameters in style sheets, it is possible to substitute styles with the same name but different parameters for all types of data in the system.

Related to the use of styles, is the frequent use of palettes — sets of controls attached to a document window. Along with style sheet dialogs, palettes provide a means for defining and applying styles. In InterText, for example, all the styles defined for a particular document are stored in a style palette (see InterText documents shown in Figure 1). Two mouse clicks are all that is necessary to either change the style of an existing text entity or to change from one style to another before beginning a new entity. With a large screen and the ability to have 30 or 40 windows open at one time, it is essential that all the tools needed for common operations are close at hand rather than in the pull-down menus. (Pop-up menus might have been another solution, but we chose to adhere as closely as possible to the Macintosh interface, as it is defined in [Appl85], and use palettes instead.) When they are not needed, all palettes can be hidden from view to unclutter the screen and to improve the manner in which material is presented to a persor browsing through the system.

Another example of a standard user interface concept that permeates the system and is made possible in a workstation environment with virtual memory capabilities is the use of "infinite" undo and redo commands. Instead of only being able to retract the last action performed, the user is able to incrementally undo the effects of all actions performed since the last time a document was saved. Any single action or set of actions that the user has undone may then be incrementally redone. There are no actions in any Intermedia applications that modify the content of the document which cannot be undone. This fosters a sense of security in users, permitting them to experiment freely with their document with the knowledge that they can return it to its former state at any time.

4. Sample Session

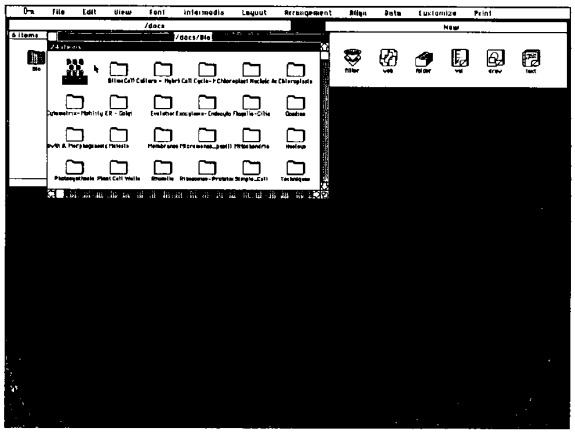
To illustrate the user interface features and the linking functionality discussed above, this section of the paper takes the reader on a tour of the system designed to simulate the interaction that takes place during a hands-on Intermedia session. The screen illustrations should aid in visualizing the system, while the text should supply the action.

The example below is taken from an Intermedia corpus of materials, *Bio 106: Cell Biology in Context* [Heyw87], created for a plant cell biology course in which students use Intermedia's editors, utilities, and linking functionality to write term papers and to explore materials about the cell and its processes.

Prior to the beginning of the semester, the professor wrote, collected, organized and linked together primary source material for the corpus, including current research papers, digitized electronmicrographs, diagrams, lecture notes, and three-dimensional models of cells.

In designing the course, the professor envisioned that students would write their papers using Intermedia's editing tools and would also include links to relevant primary source material in the corpus. His plans included annotating student papers at various stages in the writing process, beginning with their outlines. These annotations might consist of notes about the content of the paper and links to material that he believed the student should read, examine or explore before committing to a particular assertion or argument. Finally, after the papers are "submitted" for a grade, the students in the course would be asked to read and comment on each others' papers. Like the professor's comments, the students' annotations might include reactions and criticisms, or might point to other student papers or to primary source material. Since the comments would become part of the corpus, each student would have the opportunity to read any or all of the comments left by others and comment on the comments, thus engaging in an electronic dialogue. At the end of the course, the professor would evaluate the quality of each student's research paper as well as the quality of his or her comments.

The following scenario steps the reader through a sample Intermedia session from the perspective of the biology professor in the midst of creating course materials.

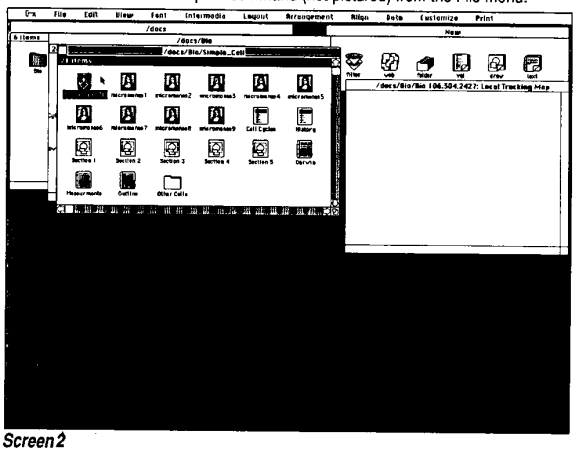


Screen 1

As you can see from Screen 1, the Intermedia desktop includes a window manager, a graphical folder system, a menu bar, and a mouse interface. The contents of the folders reflect the underlying hierarchical directory structure.

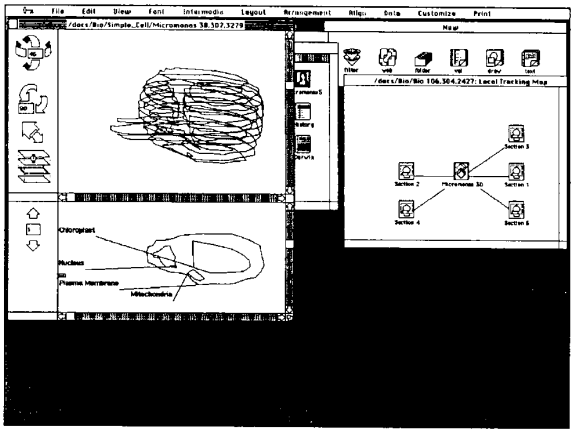
Unlike the Macintosh, Intermedia does not store application icons in the same folders with documents; instead, they are stored with several other special-purpose tools in an application, or New, window which you can see in the upper right corner of the screen. The reason for this is twofold. First, users do not have to search through folders to find the applications. Even if the New window is hidden from view by overlapping windows, selecting the "New" command from the File menu reveals it. Secondly, in a networked environment, it is easier to have a single set of applications in an agreed upon place that can be maintained and updated by a system adminstrator.

Before we browse through the documents contained in the folders, follow links or create links, we must first define a context by opening an existing web or by creating a new one. If a web is not open, we can still open and edit the documents; however, no link and block information will be visible. Rather than beginning a new web, we select the icon titled "Bio 106" and choose the "Open" command (not pictured) from the File menu.



Screen2

After opening the web, which is indicated by an empty local tracking map window (described below), we opened a folder contained in the "Bio" folder called "Simple Cell." The icons in the "Simple Cell" folder represent a number of documents of different types (one InterSpect, nine InterPix, two InterVal, five InterDraw, and three InterText documents). Any of these document icons can be selected, opened and edited. We selected the InterSpect document called "Micromonas 3D" to open it.

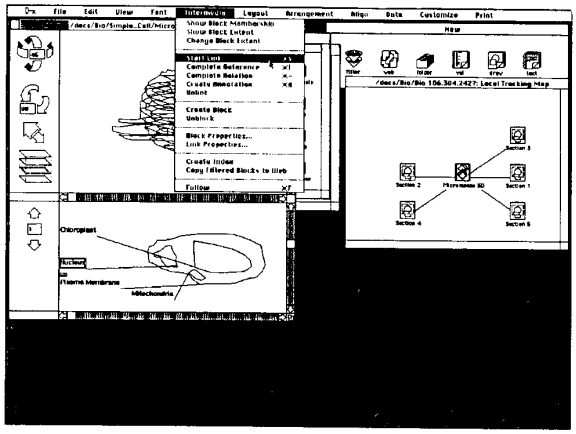


Screen 3

When the InterSpect document opens, the application displays the entire Micromonas cell in a three dimensional view (above) and a single section of the cell in a two dimensional view (below). Students can use the tools in the palette to rotate the three-dimensional reconstruction, to zoom in or out, to highlight the location of the two-dimensional section currently displayed in the bottom view, and to scroll through all the two-dimensional sections associated with the cell. Menu commands allow the user to selectively hide and display different components of the cell and/or the labels.

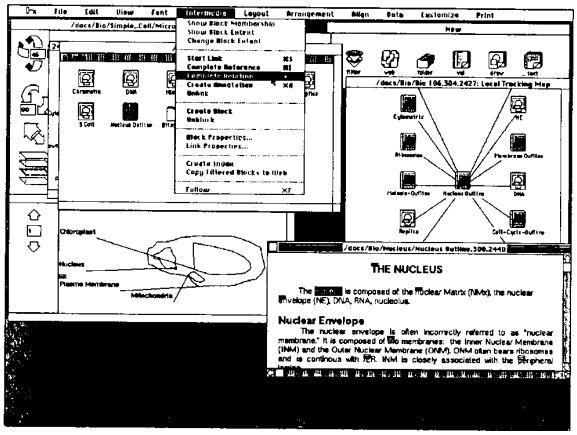
The local tracking map, empty in the previous screen, now shows the currently active document and the links that emanate from it. This is one of two visual representations of the link structure currently provided by the Intermedia system. Local maps are analogous to detailed street maps. They show you your current location and where you can travel to in the immediate vacinity. As you change locations, you require a new local map as a guide. In Intermedia, when the user activates a different document, either by following a link or by opening one from a folder, the local map updates, or tracks the users progress, to display the new current document and its direct predecessor and successor links.

Global maps, not shown in this scenario, are analogous to road maps of the entire country. They depict every document in a corpus; one line between two document icons indicates that at least one link exists between the documents. In a large corpus, users can only see a small portion of the map at any one time. The global map provides a broad overview; just as a traveler can see at a glance which cities are the biggest hubs by the number of roads emanating from them, an Intermedia user can see which documents are central to the corpus by observing the number of links connected to and from them



Screen 4

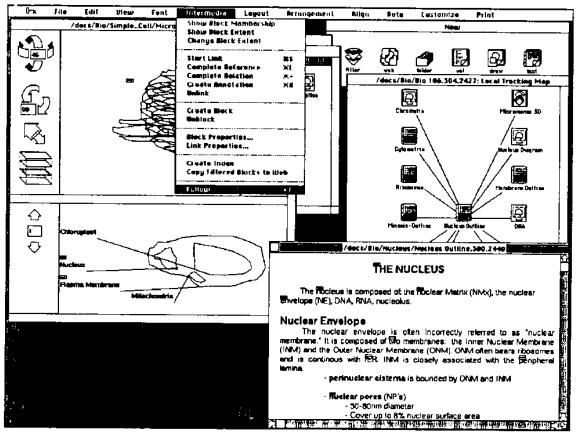
When we last worked on the Micromonas 3D document, we created a number of links connecting the plasma membrane to five different InterDraw documents, each containing a scanned photograph of one of the sections of the Micromonas cell used as data for the three-dimensional reconstruction. Before we connect the plasma membrane to the remaining photographs, we decide to connect the nucleus to general information about nucleii. The first step in creating a link involves defining a block to serve as the anchor for the link. We select the label "Nucleus" as the source block of the new link (the selection is indicated by a rectangular box) and choose the "Start Link" command from the menu. While a link is pending, we may perform any number of other actions unrelated to link-making. Like the "Copy" operation common to all Macintosh-like applications, the "Start Link" operation is completely modeless.



Screen 5

Before completing the pending link, we browse through the folders, locate and open an already existing InterText document called "Nucleus Outline." Once the text is displayed, we select the word "nucleus" in the first sentence of the document to serve as the destination block of the link and choose "Complete Relation" from the menu. You will notice two different complete commands in the menu. These are similar in function, but each creates a different type of link. The "Complete Relation" command which we chose, indicates a primary path, whereas the "Complete Reference" command signifies that the information contained in the source block is of more importance than the information contained in the destination block.

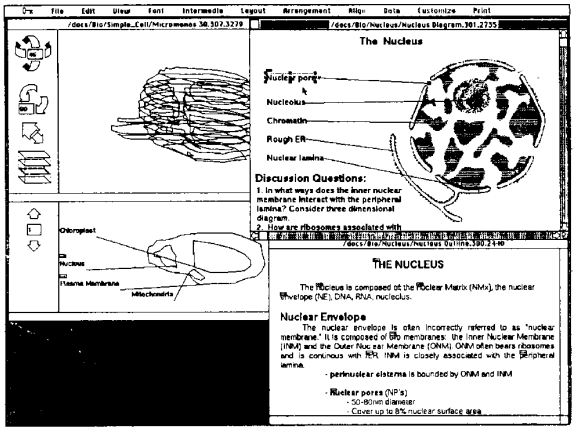
Notice in Screen 5 that the local tracking map has updated to show the links which emanate from "Nucleus Outline," since this is now the currently active document.



Screen 6

Once the link is established, both ends are indicated with markers (arrows enclosed in rectangular boxes) and the new link is added to the local tracking map.

To find other relevant material to connect to the nucleus in the InterSpect document, we enlarge the InterText window and read through the text. Since pores are important when studying simple cells such as the Micromonas cell, we select the link marker above the words "nuclear pores" and choose the "Follow" command from the menu to traverse the link.

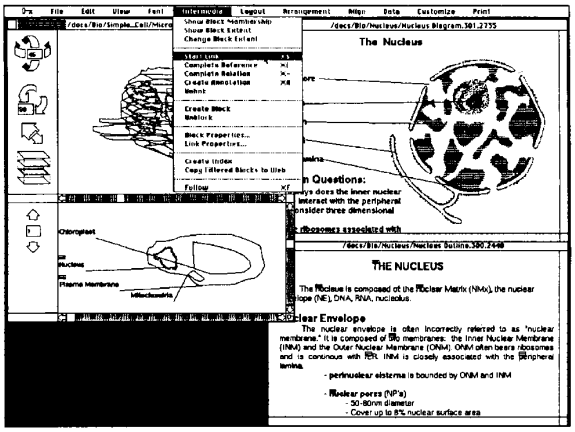


Screen 7

Following the link causes an InterDraw document containing a diagram of the nucleus to open. Notice that when a link is traversed, Intermedia automatically highlights the extent of the block at the other end of the link, indicating a particular scope of information to the reader. In this case our attention is drawn to the label "nuclear pore" and its associated label line.

The illustration in Screen 7 was entered into the system using a scanner. The bitmap was then displayed by the InterPix application, cropped and pasted into this InterDraw document and the text and lines were added to complete the diagram.

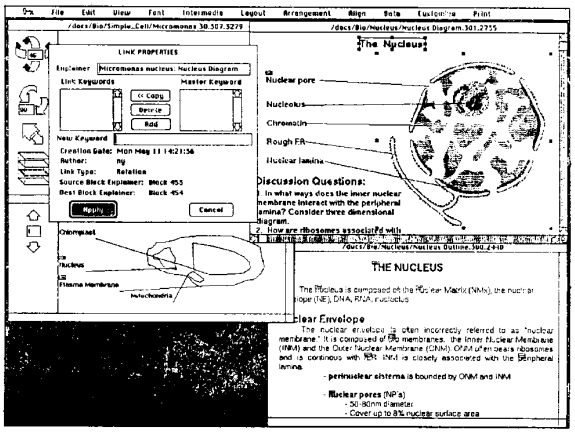
Before we continue making new links, we decide to change the default "viewing specification" for link creation to "verbose" (not pictured). With the verbose option, Intermedia automatically presents a property sheet for each new link as it is created.



Screen 8

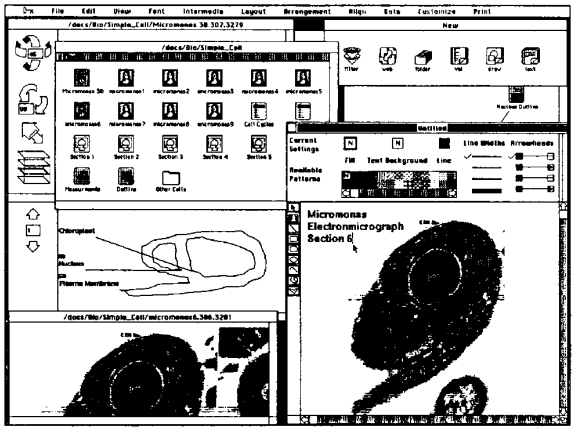
We activate the InterSpect document by clicking once in the window and select the nucleus. This time we choose to select the component itself rather than the label. When students follow the link from the InterDraw diagram to the three-dimensional representation, their attention will be drawn to the nucleus (the source block) in both the two- and three-dimensional views. As in Screen 4, we select the "Start Link" command to initiate a new link.

(NOTE: The Nuclear Envelope featured in Screens 7–9 is reprinted with permission from Garland Publishing, Inc. The Micromonas electronmicrograph, pictured in Screens 10 and 11, is reprinted with permission from the Journal of Cell Biology.)



Screen 9

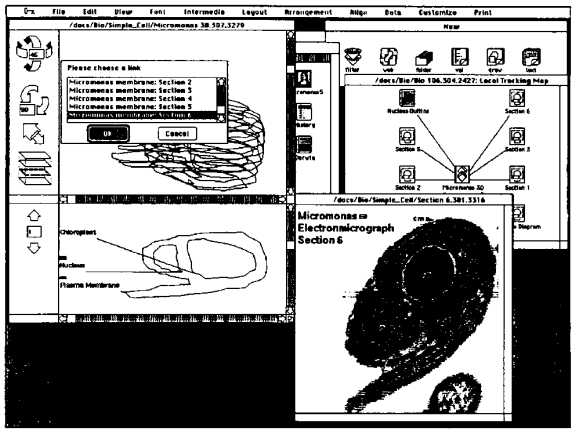
Next, we reactivate the "Nucleus Diagram," select the title of the diagram and the scanned illustration as the destination block for the link, and choose "Complete Relation" from the menu (not pictured). After we issue the complete command, a link property dialog appears, allowing us to fill in descriptive information about the link. We replace the default text, "Link 35," with the more meaningful explainer shown in Screen 9.



Screen 10

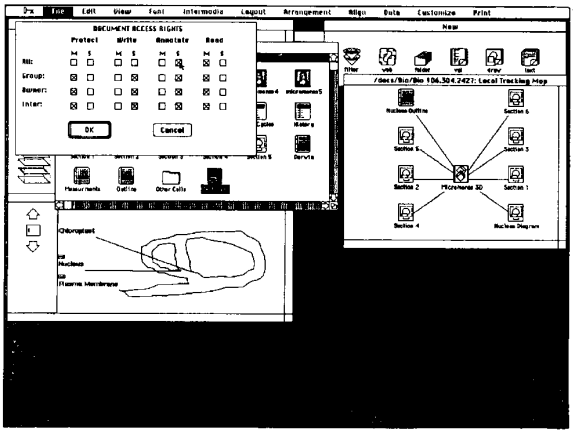
Now we will skip ahead in time a few steps. After creating the link from the nucleus in "Micromonas 3D" to the InterDraw diagram, we reactivated the InterSpect document and used the bottom tool in the palette to scroll to the next two-dimensional section. Since the label "Plasma Membrane" has already been defined as a block for another link, we decided to selected the existing marker as the source point for our new link.

Before we are ready to complete the link, we have to create a new document. We return to the "Simple Cell" folder, open an InterPix document containing a photograph which corresponds to the section currently displayed in the InterSpect document window, and crop and copy a portion of the photograph into the clipboard. We paste this image into a new InterDraw document, created by double-clicking on the "draw" icon in the New window. Finally we add some text to accompany the photograph.



Screen 11

In Screen 11, we have completed editing the new InterDraw document and have hidden the palettes to unclutter the screen. We also completed the pending link, using the text "Micromonas Electronmicrograph Section 6" as the destination block of the link. After the link was established, we double-clicked on the marker associated with "Plasma Membrane" in the InterSpect document to traverse the new link. Since more than one link is associated with the selected block, Intermedia presents a dialog box containing the explainers for each link. We select the link we just created and click on "OK." Since the document at the other end of the link is already open on the screen, following the link will simply activate the document and highlight the extent of the destination block (not pictured).



Screen 12

Before ending our session, we save and close the new InterDraw document, select its icon and choose the "Access Rights" command from the menu. The dialog which appears allows us to add or subtract access rights for different groups of users. For this document, we decide to add "Annotate" rights for all users of the system. This means that any user may create links to or from the document, but may not edit its content. Before exiting the system, we save and close the open InterSpect document and the Bio 106 web.

5. Architecture

While the scenario presents a user's view of the Intermedia system, this section describes the system from a programmer's perspective.

Consistency is the user interface principle most frequently violated by software developers [Shne87]. In part, this may be due to carelessness, but more often, interface inconsistencies are the result of not-quite identical reimplementations of features that have already been implemented elsewhere in a system. Developers writing programs for the Apple Macintosh present a prime example. The Macintosh is a system with clearly defined user interface paradigms, and new programs almost always make use of a number of the same functions that exist in hundreds of other Macintosh programs. Even so, software developers are forced to reimplement most of the "standards" (selection, resizing, dragging, etc.) from scratch. Often these programmers miss an important feature or user-interface detail that users immediately notice. The Macintosh Toolbox, however, does little to encourage adherence to the defined interface standards; adherence is purely in the hands of the programmer.

To build Intermedia, we needed a development environment which would aid programmers in creating a multi-user system containing consistent, direct-manipulation applications and functionality for linking together the contents of documents created with those applications. In designing the Intermedia system, we believed that consistency among applications was crucial since the system encourages quick transitions from one application to another.

Faced with the task of developing a relatively large, interactive system in an ambitiously short timeframe, we needed a set of development tools that would help us:

- remove the burden of user interface consistency from the application programmer;
- adopt an existing user interface standard;
- allow small groups of programmers to work on different parts of the system in parallel;
- facilitate the integration of modules developed by different groups;
- avoid as much duplication of effort as possible; and
- create a system that would be extensible and suitable for prototyping new applications.

We were able to create such an environment by building some of the pieces ourselves and adapting and integrating tools developed by others, resulting in a layered set of tools that allow programmers to develop applications that conform to the Macintosh user interface standards [Appl85].

In particular, we start with an implementation of the Macintosh toolbox under the 4.2 BSD UNIX* operating system, supplement it with an object-oriented programming language, add Apple's MacApp — a set of classes for creating "generic" Macintosh-like applications — and superimpose several crucial "building blocks" from which any number of end-user applications and utilities can be constructed. [Meyr86] provides a detailed technical description of the architecture of the development environment.

In this section, we first focus on key features of the development environment and then provide two specific examples of how the development tools were used. In the first part, we explore the concepts of object-oriented programming which facilitate simultaneous parallel development. Next we describe MacApp, the generic application framework, and then we discuss our text and graphics building blocks. The second part describes how these pieces of the development environment were used to create the Intermedia applications framework and some of the features of the InterSpect application. (For those unfamiliar with object-oriented programming, we provide a short appendix containing a review of basic object-oriented programming principles which you should refer to, if necessary, before reading further.)

Object-Oriented Programming

Object-oriented programming (OOP) has been gaining a great deal of recognition as a superior approach to programming tasks. Studies have shown significant reductions in both development time (by 75%) and size of source code (by 90%) when object-oriented techniques were used [Cox84]. One criticism of programs written using OOP techniques is that they tend to be slower than comparable conventional programs. This, however, does not have to be the case, especially since large portions of the program can be written and optimized by system-level programmers if one uses an appropriate object-oriented system [Stro86].

We selected an object-oriented programming language for the Intermedia project partly for the reduction in development time it promised, but mostly for the benefits it afforded to a group development effort. A team of developers can agree on a shared set of parent classes and their corresponding abstract methods. The use of

abstract methods, or templates, clarifies the behaviors an application programmer must implement. Then, working individually, the developers can create appropriate subclasses that will respond to a single set of messages and that can be easily integrated into one program. In dividing up a programming task, one person can implement the code that will coordinate the sending of messages to objects and the other members of the team can work on defining and refining the objects themselves.

The Applications Framework — MacApp

While an object-oriented programming language provides a number of features that facilitate team efforts, supplementing those features with a set of classes that define common behaviors — an applications framework — insures a greater degree of user-interface consistency in the system under development.

Our choice, Apple's MacApp, represents such a companion to an object-oriented programming language and provides a framework for constructing Macintosh applications [Schm86, Simo86, Doyl86]. MacApp defines classes with a combination of abstract and non-abstract methods that encapsulate the behavior of the Macintosh user interface. An eight-line program suffices to create a skeletal Macintosh-like application with menus, blank windows that can be moved, resized and scrolled, data that can be stored and retrieved, and laser printing. To write an application with "views" in the windows, the programmer must subclass several base classes provided by MacApp. The most important of these classes include:

- The Object class, which manages the freeing of memory and the cloning of new objects. It is the parent of all other classes.
- The Application class, which contains methods for launching an application, displaying the menu bar, managing the main event loop, and creating and initializing appropriate Document objects.
- The Document class, which maintains the data model for the program. Document objects contain all the basic information for saving and restoring the data and managing several other objects — such as Window objects, Frame objects and View objects — that are involved in viewing the information contained in the document.
- The Window class, which manages all the operations pertaining to windows, including opening, closing, resizing, moving, activating and redrawing.
- The View class, which manages the rendering of the data that is contained in the document and passes on mouse-hit events to the appropriate objects within the view.
- The Command class, which manages most actions generated by the user from the menu, the mouse or the keyboard. Because almost all interaction is handled through Command objects, multi-level undo and redo is easily implemented.

By subclassing these and other MacApp classes, a programmer builds a model for the data in an application, creates the windows and frames in which the information will be viewed, and describes how the user can interact with that information.

The Application class has perhaps the greatest impact on the developer. This class contains the methods necessary for an application's most basic behavior. For example, there are methods for launching an application, running the main event loop, dispatching events to the appropriate event handler, creating new documents, closing documents, and deleting documents. In the case of a user selecting a command from a menu, the Application object interprets the mouse press and sends a message to the currently selected object's DoMenuCommand method. Since an application's Application object handles all user-initiated events such as mouse presses, keystrokes, menu selections, etc., a programmer does not have to be concerned with flow-of-control issues.

As an applications framework, MacApp promotes consistency in a multi-application environment by eliminating the need for programmers to reimplement any of the user-interface features required for the shell of a Macintosh application. The framework insures that each application will have windows, menus, and other basic components that look and behave in the same manner as all others in the system.

Building Blocks

While OOP provides the structure and methodology for cooperative development, and MacApp provides a set of base classes from which to build an application, these two components alone are not enough to create a fully functional development environment for a group of cooperative developers. Still missing is a component that helps developers render and manipulate the data for their particular application. To this end, we implementated several building blocks — sets of reusable, redefinable classes that implement basic functions common to multiple applications. Instances of these building blocks can be incorporated directly into an application, and the application programmer can use part or all of a building block's functionality as it exists or modify the functionality to suit a specific application. To support the development of applications within the Intermedia system, we initially implemented two building blocks — a Text Building Block and a Graphics Building Block — and later found the need for a third — a Table Building Block.

The Text Building Block permits the inclusion of text anywhere within an application. The Text Building Block makes it possible to provide exactly the same interface for displaying, editing and formatting multi-font text throughout the whole system. Any application can incorporate an instance of the Text Building Block which inherits full programmer and user interface functionality. An entire application, such as a text editor, or some part of an application, such as the input field of a dialog box, can be based on the Text Building Block.

Just as the Text Building Block allows the inclusion of text anywhere within an application, the Table Building Block facilitates the incorporation of tabular data. A programmer may use the Table Building Block as the backbone of an application such as a spreadsheet program, or may use it to integrate one or more tables into any other type of application. For example, Release 3.0 of Intermedia includes a videodisc application with tables for storing data such as frame numbers, sequence names and playing times.

The Graphics Building Block (GBB) lets programmers incorporate graphics such as lines, rectangles, circles, icons, polygons, etc. into their applications. This building block defines a number of "shape" classes with methods for drawing, highlighting, selecting, resizing, and moving. The building block also subclasses MacApp's View class so that the subclassed graphics GView contains a list of all objects to be rendered on the screen. To illustrate how building blocks are used in general, we will focus on the GBB.

Programmers can take advantage of a building block such as the GBB in one of four ways. First, a programmer can use the building block functionality in its entirety. For example, to create a structured graphics editor similar to Apple's MacDraw [Appl84b] in which users can draw a variety of different shapes on the screen, rearrange them, group them and perform various other editing operations, a programmer could subclass each of the GBB's shape classes and then work from there to add features such as alignment, tool palettes, style palettes, and grids.

In the second case, a programmer can override methods to eliminate functions in building block classes. Intermedia's timeline editor, which renders chronological timelines based on sets of date/label pairs, contains lines, that unlike lines in a graphics editor, cannot be moved by the user [Garr86]. By subclassing the GBB's graphical line object and overriding the method that performs the move function, the programmer is able to incorporate lines into the timeline editor that can be selected and highlighted but not moved. Since the selection and highlighting functionality is inherited from the GBB, the programmer does not have to write any code to perform these functions.

Thirdly, methods may be overridden to add increased functionality to building block classes. The use of icons in Intermedia's "desktop" application illustrates the addition of functionality to a building block's method. The desktop application subclasses GBB icons — simple bitmaps — and overrides the Draw method so that a text string, representing a document's name, is always printed below the icon (see Screen 1).

In the fourth case, a programmer can overrride methods to change the behavior of building block classes. An example of this is clearly illustrated in Intermedia's three-dimensional viewer, InterSpect. To indicate that an object has been selected, the GBB uses "handles" as a method of highlighting. In InterSpect, however, the GBB's GSelection method is overriden to substitute bold outlines for handles as a method of highlighting.

With these four options available, application developers are supplied with enough flexibility to create innovative interfaces, but are not burdened with the implementation of functions that should behave identically across applications. The building blocks complement MacApp by providing a means to achieve internal as well as external consistency among applications.

Adding Shared Functionality

By using the tools described above — an object-oriented programming language, MacApp and building blocks — a programmer could create applications which adhere exactly to the Macintosh user interface paradigms. In the requirements for our Intermedia system, however, we identified the need to run multiple applications on the desktop as well as the need to link the contents of documents together to form trails through collections of documents [Meyr85]. These two requirements made it necessary for us to extend, and in some cases alter, the existing Macintosh user interface paradigms. To this end, we kept MacApp as the first layer of our system and then subclassed some of the MacApp classes to create an Intermedia layer. The manner in which the Intermedia layer extends the functionality of MacApp illustrates the ease with which features shared by many applications can be implemented using our object-oriented development base. Like MacApp, a number of Intermedia layer classes consist of abstract methods. It is in the building blocks, which sit on top of both MacApp and the Intermedia layer, that most of the abstract methods are implemented, leaving a minimal number for the application programmer to implement.

Briefly, the type of additional functionality the Intermedia layer supports includes the creation of links between a selection in a source document and a selection in a destination document. These selection regions, called blocks, can be of any size from an insertion point to the entire contents of the document. The existence of a link is indicated by a marker icon placed near the source and destination blocks. After creating links, users can follow the links to travel from one document to another, and they can assign properties such as names and keywords to the links and to their corresponding blocks. When a link is traversed, the block at the other end of the link is scrolled to and highlighted. Like other standard Macintosh interface paradigms, the behavior of links and blocks must be consistent across all Intermedia applications.

To attain the desired consistency, the Intermedia layer subclasses MacApp's Document, View and Application classes. In MacApp, the Document class manages the reading and writing of an application's data model while the View class manages the rendering of that model. The Intermedia layer's document and view classes, IntDoc and IntView, extend the MacApp functionality to include the reading and writing of link information to a relational database and the rendering of the link model. In addition, IntView provides abstract methods for displaying block markers which are implemented at the building block layer. The Intermedia layer's application class, IntAppl, adds functionality necessary for interfacing with Intermedia's desktop application.

The linking functionality itself is implemented largely in the Block class which exists at the Intermedia layer and inherits from MacApp's Object class. Since there must be a block at either end of a link, blocks are created each time a link is made, unless the user attaches one end of the link to a block that already exists. A Block object keeps track of all links that emanate from it. Therefore, users can access links through one of their blocks in order to follow the link or view its properties. The methods of the Block object include ones for starting links, completing links, following links, viewing link and block properties, and several others. When a block marker is selected, certain appropriate menu items are enabled. If the block has no links, it is possible to show the extent of the block, start a link, delete the block, and show the block properties. Similarly, if the block has at least one link, all of the previously mentioned options are available along with following, viewing of link properties, and deleting the link. All of these actions are done using block methods.

This portion of the linking functionality is shared in its entirety by all Intermedia applications, leaving only a few details for the developer to implement in order to integrate linking into a new application. With a non-object-oriented development base, each application programmer would have been forced to identify and call procedures from appropriate subroutine libraries for saving, restoring, creating, deleting and viewing links. With the Intermedia layer augmenting MacApp, it is the applications framework that essentially calls the programmer, who must implement only those methods that are specific to his or her application. All other functionality is shared as standard fare by all developers.

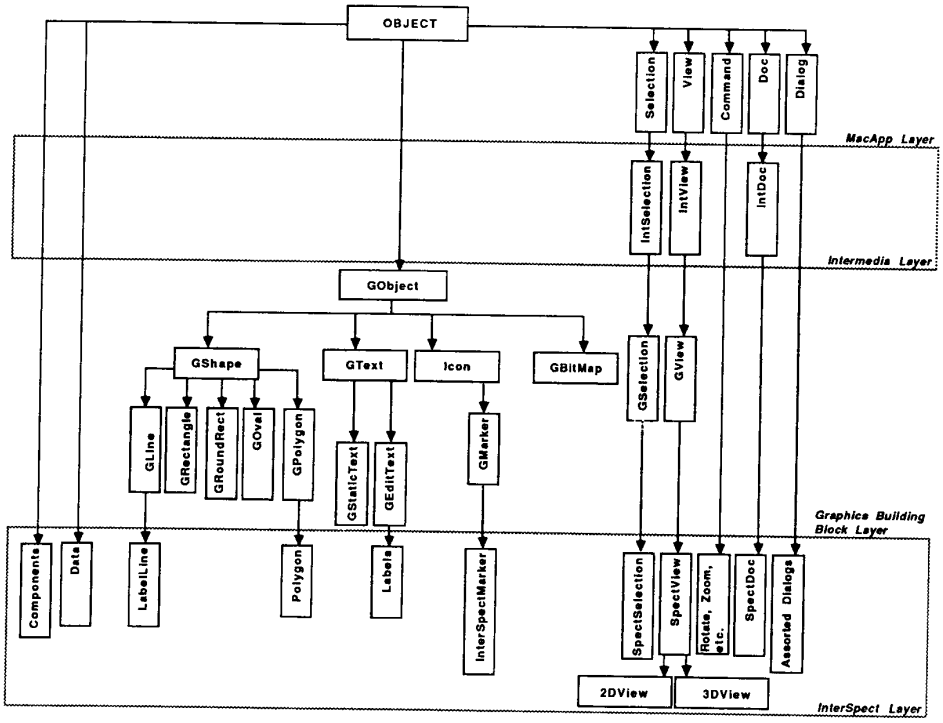


Figure 3. Inheritance hierarchy for the InterSpect unit.

Building an Application

While the block methods in the Intermedia layer handle the core of the linking functionality, the manner in which blocks are displayed in different applications is handled by methods of the individual applications' View objects. The display methods include the creation and display of marker icons, scrolling to a block after a follow, and highlighting the extent of a block. These methods, which were defined at the Intermedia layer as abstract methods of IntView, are implemented in the Graphics Building Block's subclass, GView. An application which, in turn, subclasses GView, inherits a whole hierarchy of functionality, part of which includes the display of blocks.

Some Intermedia applications, like InterSpect, modify the common functions inherited from higher-level layers. For example, one method associated with polygon objects in the GBB allows users to move polygons by clicking on them and dragging. Since InterSpect uses data files which define each polygon in a three-dimensional object relative to other polygons, the developers override GPolygon's Move method to eliminate the dragging functionality. In all other respects, polygons behave in InterSpect as they are defined in the GBB (see Figure 3).

Every application added to the Intermedia environment has a number of features, like zooming and rotation in InterSpect, which are not the same or similar to functions implemented at the MacApp, Intermedia or building block layers. In this case, the programmer creates new objects which are simply subclasses of MacApp's generic Command or Object classes. While the implementation of these features is not directly simplified by the development tools, general object-oriented techniques help to structure programmers' thinking about their application-specific problems.

Developing In Parallel

When applications such as InterSpect are being built in parallel with other applications that run in the same environment, each application is initially considered as a separate entity. The programmers build and debug their programs as independent applications, each taking advantage of the inheritance hierarchy provided by the development tools. The extreme modularity of the object-oriented environment makes it possible to integrate applications which have been developed separately into a single system without requiring recompilation.
For purposes of application integration, the Intermedia system contains a Framework application. When programmers are developing an application, they compile the code for each application object separately from the code for all other objects. The compiled code minus the application object is called a unit. For example, InterSpect has an application object, called SpectAppl, which is a subclass of the Intermedia layer's application object, IntAppl. The SpectAppl object contains a method called DoMakeDocument for creating InterSpect document objects exclusively. To test InterSpect, the programmer binds the InterSpect unit with the SpectAppl object. When ready to integrate InterSpect into the Intermedia system, the programmer does not bind with the SpectAppl object, but instead relies on the Framework's application object (FrameworkAppl) — also a subclass of IntAppl — which contains a DoMakeDocument method for creating many different document objects. When integrating InterSpect into the Intermedia system, the programmer adds InterSpect to the list of document types specified in FrameworkAppl's DoMakeDocument method and then binds InterSpect and all other application units with the FrameworkAppl object rather than the SpectAppl object.

Being able to reuse the same units without recompilation for independent testing purposes and for creating an integrated system was instrumental in the success of our parallel development effort. Developers could work on their portions of the project independently, knowing that the effort of integration would be minimal as long as they worked within the structure provided by the building block and the Intermedia layer classes. In particular, developers could count on inheriting all the linking functionality as soon as their application was bound and run with the FrameworkAppl object.

6. Conclusion

With the generic MacApp layer, the Intermedia layer and three crucial building blocks in place, we are able to build and integrate new applications into the Intermedia environment in a matter of weeks. By systematically implementing abstract methods and overriding other methods found in layers above the application layer, developers are able to create applications guaranteed to have common functionality that is consistent with all other applications. While it is, of course, possible to institute inconsistencies, more effort is required to be inconsistent than to be consistent.

We can directly attribute the successful and rapid development of the Intermedia environment to object-oriented programming techniques. These techniques, in combination with other factors, enabled us to attain each of our goals for the project. We were able to remove the burden of user interface consistency from the application developer, adopt an existing user-interface definition, allow groups of programmers to work in parallel, integrate separate modules without recompilation, avoid considerable duplication of effort, and create an extensible system suitable for the rapid development of new applications.

While the benefits of our development base are substantial, as with every system, there are also problems and drawbacks. Our most serious problem stemmed from working in an extremely layered environment. Although inheritance certainly saves programmers an enormous amount of work, it can prove to be quite time-consuming in an environment that does not support incremental compilation. When working with a UNIX system using the C programming language (with an object-oriented preprocessor), changes in one layer are not automatically propagated to other lower-level layers. In our case, when methods in a parent class such as IntView were changed, all layers inheriting from that class had to be recompiled, often taking 45 minutes or longer. Although we attempted to minimize the number of recompilations, they were often unavoidable during the period we were working on all layers of the system simultaneously. We did manage to structure the working environment so the recompilations would not prevent other people from working, but this scheme added the expense of keeping duplicate copies of the source code and producing new releases of the system every week. Even though we used the RCS source code control system to facilitate release tracking [Tich82], we still had to be extremely careful to include the most up-to-date versions of every layer in each release.

Through the use of abstract methods, OOP provided us with a concrete structure that could be shared by each of the applications in the system. Finally, with OOP and MacApp, we were able to structure the whole system in such a way that each part could be worked on independently with the guarantee that integration would be easily accomplished and common functions would behave identically in each of the applications.

Measuring Success

To assess the power and utility of hypermedia, IRIS is conducting a series of experiments at Brown that introduce the Intermedia system into existing courses and work settings. In these experiments, the Intermedia system is being used throughout the authoring process, providing facilities to create, organize, reorganize and cross-reference free-form text and graphics. The experiments are also designed to examine how a group of authors and editors can use a hypertext or hypermedia system to work collaboratively, using the tools to communicate as well as create.

Acknowledgements

Intermedia is the culmination of two years of intense effort by a large team of developers led by Norman Meyrowitz. We would like to thank Helen DeAndrade, Tim Catlin, Page Elmore, Charlie Evett, Matt Evett, Ed Grossman, Nan Garrett, Karen Smith, Tom Stambaugh, and Ken Utting for their tireless contributions to the Intermedia system. We would also like to thank Peter Heywood, Scott Buchanon, and Chris Scott for the many hours they spent constructing the Bio 106 materials, and George Landow, David Cody, Glenn Everett, Rob Sullivan and Suzanne Keen Morley for producing the enormous corpus, *Context 32: A Web of English Literature*. In addition, we are grateful to Larry Tesler of Apple Computer, Inc., Jeff Singer and Stan Fleischman of Cadmus Computer, Inc., and Mark Nodine of Bolt Beranek and Newman for their assistance in making available software from which we were able to construct our development enviroment.

The work described in this paper was sponsored in part by a grant from the Annenberg/CPB Project and a joint study contract with IBM.

References

[Appl83] Apple Computer Inc., Lisa Office System, Cupertino, CA, 1983.

[Appl84a] Apple Computer Inc., MacWrite, Cupertino, CA, 1984.

[Appl84b] Apple Computer Inc., MacDraw, Cupertino, CA, 1984.

[Appl85] Apple Computer Inc., *Inside Macintosh*, Volumes I, II, and III, Addison-Wesley Publishing Company, Inc., Reading, MA, 1985.

[Back82] D. Backer and S. Gano, "Dynamically Alterable Videodisk Displays," *Proceedings of Graphics Interface 82*, Toronto, May 17-21, 1982.

[Bend84] W. Bender, "Imaging and Interactivity," Fifteenth Joint Conference on Image Technology, Tokyo, Nov 26, 1984.

[Brow87] P.J. Brown, "Turning Ideas into Products: The Guide System," *Hypertext '87 Workshop Proceedings*, Chapel Hill, NC, November 13-15, 1987.

[Conk87] J. Conklin, "Hypertext: An Introduction and Survey," IEEE *Computer*, Vol. 20, No. 9, September, 1987.

[Cox84] B. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology," IEEE *Software*, Vol. 1, No. 1, January, 1984.

[Doyl86] K. Doyle, B. Haynes, M. Lentczner, L. Rosenstein, "An Object Oriented Approach to Macintosh Application Development," *Proceedings of the 3rd Working Session on Object Oriented Languages*, Paris, France, January 8-10, 1986.

[Fein82] S. Feiner, S. Nagy, and A. van Dam, "An Experimental System for Creating and Presenting Interactive Graphical Documents," *Transactions on Graphics*, Vol. 1, No. 1, 1982.

[Garr86] L. Garret and K. Smith, "Building a Timeline Editor from Prefab Parts: The Architecture of an Object-Oriented Application," *OOPSLA '86 Proceedings*, Portland, Oregon, September, 1986.

[Gold84] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1984.

[Good87] D. Goodman, *The Complete HyperCard Handbook*, Bantam Books, Inc., New York, 1987.

[Hala87] F.G. Halasz, T.P. Moran, and R.H. Trigg, "Notecards in a Nutshell," *CHI + GI 1987 Proceedings*, April 1987.

[Heyw87] P. Heywood, S. Buchanon, and C. Scott, *Bio 106: Cell Biology in Context*, Brown University, Providence, RI 1987.

[Inte86] Interleaf, *Workstation Publishing Software: User's Guide*, Ten Canal Park, Cambridge, MA, 1986.

[Know86] KnowledgeSet Corporation, Laser Facts, Monerey, CA, 1986.

[Land86] G. Landow, D. Cody, G. Everett, and R. Sullivan, *Context 32: A Web of English Literature*, Brown University, Providence, RI, 1986.

[Meyr85] N. Meyrowitz, et al, "The Intermedia System: Requirements," Institute for Research in Information and Scholarship, Brown University, September, 1985.

[Meyr86] N. Meyrowitz, "Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework," *OOPSLA '86 Proceedings*, Portland, Oregon, September, 1986.

[Schm86] K. Schmucker, *Object-Oriented Programming for the Macintosh™*, Hayden Book Company, Hasbrouck Heights, NJ, 1986.

[Shne86] B. Shneiderman and J. Morariu, "The Interactive Encyclopedia System (TIES)," Univeristy of Maryland, College Park, MD, 1986.

[Shne87] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1987.

[Simo86] J. Simonoff, *MacApp Programmer's Manual*, Apple Computer Inc., Cupertino, CA, 1986.

[Smit82] D.C. Smith, C. Irby, R. Kimball, and B. Verplank, "Designing the Star User Interface," *Byte* Publications Inc., April 1982.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, Inc., Reading, MA, 1986.

[Tesl85] L. Tesler, "An Introduction to MacApp 0.1," Apple Computer Inc., Cupertino, CA, February 14, 1985.

[Tich82] W.F. Tichy, *Revision Control System (RCS)*, Purdue University, W. Lafayette, IN, 1982.

[Weye85] S. Weyer and A. Borning, "A Prototype Electronic Encyclopedia," ACM *Transactions on Office Information Systems*, Vol. 3, No. 1, January 1985.

[Xero82] Xerox Corporation, 8010 Star Information System *Reference Guide*, Dallas, TX, 1982.

[Yank85] N. Yankelovich, M. Meyrowitz, and A. van Dam, "Reading and Writing the Electronic Book," IEEE *Computer*, Vol. 18, No. 10, October, 1985.

†*Macintosh is a trademark of McIntosh Laboratory, Inc., licensed to Apple Computer, Inc. UNIX is trademark of AT&T.*

APPENDIX

Review of Object-Oriented Programming (OOP) Principles

The fundmental notion in object-oriented programming is, not surprisingly, that of objects. An object-oriented program is a system of interacting objects. Objects encapsulate data and the algorithms that specify the behavior of that data. Operations on an object can take place only through a well-defined interface to the object's behavior; the actual implementation of the behavior is hidden from all but the designer.

The data structure part of an object is known as its fields or slots. The routines that can act upon an object of a particular type are called methods. These methods are the primary way that the data within an object may be modified. Other objects invoke an object's method by sending a message to the object. The object then interprets the message and performs the appropriate method.

Classes (also known as object types) are templates defined by programmers that describe the properties and behaviors of a set of common objects. An object is actually an instance of a class template, created as a program is running. Each object is a copy of the class template, differing from other objects in the class only in the data and implementation assigned to the fields and methods. While class templates provide a basis for modularity, subclassing — the ability to define one class of objects in terms of other classes — is one of the OOP concepts from which much leverage is gained. Subclasses inherit the characteristics of higher-level ancestor classes. An object in a subclass contains all the same fields and methods as an object in the parent class. In defining a subclass, a programmer can then add fields and methods, or redefine methods existing in one of its superclasses. A redefined method can implement a behavior completely different from the original method, or it can merely modify or extend the behavior of its parent slightly. Refining, or overriding, a method of a class makes it possible to considerably reduce the amount of code that an application programmer needs to write; only code that explains how a method differs from the parent method is necessary. The programmer is guaranteed that the methods he or she did not override will respond properly to any messages sent to the object. This process of redefining and extending a class of objects in terms of another class is important for it enables programmers to use existing parts of a system without having to understand the details of their implementation.

For example, say we define a class called Rectangle. All the objects that are instances of this class will have two "Point" fields (the top-left corner of the rectangle and the bottom-right corner). The class will have methods for calculating area and drawing the rectangle. If we wish to draw a rectangle on the screen, we first create a Rectangle object, and then send a message to invoke the "Draw" method. To create a more specialized object that draws a rectangle and prints text inside the rectangle, we would define a subclass of Rectangle, called TextRect. No existing code has to be rewritten; instead, in the definition of TextRect we can add a character string field and override the Draw method inherited from the parent class in order to refine its behavior to draw the rectangle plus the text. Like Rectangle objects, TextRect objects will also respond to "FindArea" messages even though we did not add any code for calculating area in the TextRect class.

In the above example, the Rectangle class served as a template for the subclass TextRect; however, it is often useful to define templates that are less specific than the Rectangle class. For instance, a parent class of Rectangle might have been created, called Shape, with two abstract methods, Draw and FindArea. An abstract method contains no code; it exists only for the purpose of being overridden. To create a new shape, a programmer would subclass Shape, add appropriate fields and override the Draw and FindArea methods. Likewise, a program which displays many different shapes on the screen might contain an object list. Each object in the list would inherit the Draw method from the Shape superclass, but would override it with code appropriate for drawing the specific object. Since all shapes are guaranteed to understand the same message protocol, we can display a whole screenful of shapes by merely sending the same draw message to each object in the list without knowing exactly what type of shape objects are in the list. Objects, like the different shape objects, descending from the same parent class are essentially "plug compatible;" each understands the same messages as the others, yet each performs the task in its own way. This modularity allows us to create and insert new subclasses into the program transparently.