Diagram Development Plan

by

Peter Bishop
6/7/85

1. Introduction

This memo records the overall development plan for our Overhead Slide Presentation system, currently called Diagram. We expect the name of the system to change. This document will be under development for the next few months.

2. Milestones

There are several milestones that can be used to see whether the development is on schedule. Each milestone marks the successful completion of one phase of development. Thus we may talk about a particular phase of development or of the milestone that marks the end of the phase. These are the kinds of milestones that would be expected in almost any development project:

1. Product Proposal

During this phase, a fairly complete proposal is made for the user interface of the product, including a storyboard for most of the different screens that the user will see while using the product. In addition, data is gathered about the marketability of the proposed product and the financial outlook for the entire project. A rough estimate is made of the manpower and other resources needed to complete the project. The milestone consists of three documents, a Development Plan, a Product Specification, and a Marketing and Financial Analysis. The plan is to complete this phase by Sept. 1, 1985. These documents should be adequate to decide whether to proceed with full product development.

2. Product Design Specification

The first phase of the implementation of the product is to create an overall design, including an overall module breakdown and a specification of the interfaces between the modules. It is most important to specify in detail the interfaces to modules that will be needed by all or most of the programmers on the project.

The milestone consists of a document that specifies the design of the product, gives the detailed interfaces to the widely-used modules and utilities, and gives somewhat more general design principles to the rest of the modules

in the system. At this point, it is possible to add additional people to the project.

3. Initial System

This phase consists of actually building the foundations of the product. The milestone consists of a complete running system that implements all of the detailed interfaces specified in the Product Design Specification. In addition, it contains enough of the rest of the product to serve as a building-block for further development. It is now possible to see the system actually running, but the number of features that have been implemented is very small.

4. Intermediate System

During this phase, we are building the bulk of the basic features into the product. The Development Plan will generally specify what features will be working at this point in the system. We will not pause for a QA-cycle, however, so this intermediate system will still be quite buggy. It is still just a tool for further development, and a significant milestone to ensure that the development process is working on the right time-table.

The milestone consists of a demonstration of the system with the features specified in this memo working, with perhaps some bugs.

5. Final System

This phase marks the completion of the features of the product. During this phase, it will be highly desirable to finish all of the experimentation that is needed to be certain that we have specified and implemented the correct product. This is the last phase of development during which features may be changed, added, or deleted from the product.

The milestone will be a demonstration of the complete product, with bugs.

6. Debugging

This phase begins with a freeze on all features in the product. During this time, the manuals may be sent to the printers. The entire development staff switches from the coding phase to the debugging phase. This does not mean that no debugging has occurred up to this time, but merely that now it is the only activity. Each developer first works on fixing all known bugs in their own code. As soon as a point is reached when no bugs are known, the first available developer immediately proceeds with further testing, submitting reports for all unknown bugs as soon as they are found. This developer stops additional testing at approximately the point when he or she finds the next bug that is in his or her section of the code.

During this phase of development, versions will appear that will begin to be usable for actual, real work, but with bugs. It is not clear whether this will occur early in this phase or rather late in this phase.

The milestone consists of a demonstration of the product during which no bugs are found, even when visitors attempt to use the product.

7. Golden Master

This is the Quality Assurance phase of the product. The product may be given out to friendly beta testers and to any external quality assurance organization that we might wish to use. In addition, the entire internal development team and anyone else who may wish to, will be spending full-time trying to find any remaining bugs in the product. We will not divert all development people away from testing toward fixing bugs at any point in this process, because it is important to have as few versions as possible that require additional testing. Any new versions that are released during this period must be subjected to the full range of tests that had already been given to previous versions. Rather than releasing many new versions, development will keep an internal version that will not be subjected to a rigorous regression test, unless it is automated. This internal version will remain internal until it is felt that a complete test has been performed on the previous version, then all bugs found will be fixed and another version released.

This process will not stop until a version is accepted as the Golden Master version of the product. This is the milestone for this phase.

3. Basic Design

This section describes the general structure of the software needed to impelement the product. This may appear as an early draft of the Product Design Specification, but it concentrates on specifying the nature of the separable pieces of the product. It is this structure of the product that opens the possibility of placing additional personpower on the project.

4. Resources Needed

This section will describe the resources needed to complete the product. This section cannot really be started until we have gotten further on the product specification itself. In addition, this section will specify the plan for hiring additional people and will show how that will affect the timetable for the development of the product.

5. Experimentation Required

In any new product, there are always questions about how something should be done. We must identify the areas of the product in which questions exist and

carefully manage the resolutions of these questions. In many cases, significant work will be needed to resolve these questions. Sometimes it will be necessary to implement several different approaches to the question and see which one we like the best. In other cases, we can identify simpler ways of settling the questions. This section will need to be a continuing document that is continually changing throughout the entire project as issues are raised and settled. We must start with an initial list of the areas of known uncertainty.

5. Summary

This development plan will allow the project to be tracked by any interested party. It explains why it is important to complete both the specification and the basic implementation of the product so early. Anyone wishing to contribute to the project will need to have this document in order to understand at what point in the project their particular kind of contribution will be needed.

All of the later portions of the project will continually be thought about from the very beginning. Anyone who has an insight about potential problems later in the project should be able to identify the person with whom the point should be raised. All these points will be open for consideration at any time, since there will not be much time for contemplation during the later phases of the project. It is often important to allow thinking time to resolve the hard questions. We can be accomplishing other tasks while that thinking is happening. So be sure to raise problems early, especially after the initial decisions have been announced.

Dennis

Presenter

Dennis Austin
August 19, 1985

These notes describe the basic plan for Forethought's new product. You will find this version very similar to that of July 12, but there is a great deal of subtle refinement. The only substantial change is a new section about tables. I have also not changed the illustrations to the Microsoft style, but the specifications should be complete enough to make up a Windows story board.

Theme

Presenter is a color drawing program designed primarily for producing overhead slides. It is distinguished from MacDraw-type progams primarily by specializing in textual elements:

- Paragraphs with mixed fonts, sizes, and styles.
- Finely adjustable line spacing and placement.
- Wide choice of typefaces and large sizes suitable for graphic displays.
- Word-processor-like manipulation of text.

It can draw diagrams and insert decorative art, but is really aimed more narrowly at graphic text displays. With the addition of text flowing into multiple elements, it would almost make a simple page layout program. Other differences from MacDraw make it more suitable for preparing presentations:

- Multiple images (slides) that can be rearranged within a single file (presentation).
- Multiple windows that can show different images from the same presentation. ~?
- Notes that can be attached to a drawing and viewed selectively.
- A variety of printing methods: full size, two on a page, with or without notes, etc.
- Model image overlays that allow pages share borders and similar features.
- Special help for making bulleted lists and tables.

To keep it simple, the drawing potential of the program is less capable than that of MacDraw. For example, *Presenter* has:

- No polygons or smoothing.
- No free-hand curves.
- No stacking order changes. no circles or curves
- No rotate or flip.
- No grouping or locking.
- No rulers.

These features are useful and I hate to see them go, but in user interfaces, as in programming languages, the problem is not so much what to put in as what to leave out. I'm not sure that this is the right list of omissions, but simplification is important.

Presentations

Unlike other graphics programs, *Presenter* does not produce files or documents that contain a single picture. A *Presenter* document comprises an entire presentation, or perhaps a section of a very long presesentation. There is no limit on the number of images (slides) in a presentation, but the product will not be designed to handle large volumes, say over one hundred. When an actual presentation contains more than about fifty images, it can be better handled by dividing it into sections, each in its own document. one hundred

Notes, titles, and printing formats. Each image has a title and about one-half page of notes associated with it. The title appears on the image itself, but notes are selectively visible. Images can be printed by themselves to make actual slides, or the they can be printed (in reduced size) along with their notes to make handouts for people who miss the presentation, etc. Reduced size printing with two or three slides per page is also available to make handouts for the audience.

master
~~Model~~ slide. There is a special image called the model slide that adds common features to all images in the presentation. This allows for the addition of borders, logos, dates, and other material that appears on every image. The format of titles is also determined here, but unlike the other features, it can be changed on the individual images.

Slide Sorter. The individual images are normally viewed one at a time, or one per window if multiple windows are in use. In slide sorter mode, however, up to about sixteen miniature images can be viewed in a single window. Although the actual images can be hard to distinguish in this view, their titles appear in readable type whenever they are selected. Slides can be cut and pasted in order to reorder the presentation or to move slides between files. This view also provides a handy method of finding and viewing a particular image. An analogous view called Title Sorter has the same function, but shows only the slide titles instead of their images.

Slide show. Although most presentations are printed for final display, *Presenter* allows you to use the computer as a "projector" if you like. The images are shown one at a time ~~full-size~~ (landscape only) on the computer's screen without tools, menu bar, or other windows. The show can be presented manually, pressing the mouse button or keyboard keys to change slides, or automatically timed. Read notes

Drawing Slide Images

Even though a presentation may be entirely text with no diagrams or other graphics, the slide images are still "drawn" to some extent. *Presenter* is, at its heart, a structured graphics editor like MacDraw. It works with objects such as lines and rectangles that can be moved, resized, and overlapped to produce many different effects. Text is held as the contents of objects, typically borderless rectangles.

Objects. There are two basic types of drawing objects: two dimensional (boxes) and one-dimensional (lines). Lines have width, of course, and in fact they have a number of other attributes, but they do not have contents as boxes do. Boxes and lines both have location and dimensions that are specifiable with two points. A new object is drawn by first selecting a tool (determining the type of the object) and then positioning the pointer, clicking, and dragging. For lines the beginning and end of the drag determine the end points of the line, and for boxes they determine opposite corners of a rectangle.

Once the object is drawn it can be moved, resized, or have its attributes changed. Below are some attributes that can be selected by menu commands or other means.

Boxes
- Shape: rectangle, round rectangle, ~~ellipse~~
- Border: thickness, style (dotted, etc), color [or none]
- Fill: color or pattern [or none]
- Contents: text, label, picture [or none]

hidden, outline, solid
pattern
color
line ~~type~~ width

Lines
- Thickness, ~~style~~ pattern, color
- ~~Curved or straight~~
- Special form: Arrows, braces

Selection

When the cursor is an arrow, it is used for selection, moving, and resizing objects. Clicking selects the ~~smallest~~ closest enclosing box, or, if the cursor is close to a line, it selects the line. When an box is selected, a dotted box appears around it with four control points, or "handles", at the corners. When a line is selected, a dotted box surrounds the line and handles appear at both ends.

Moving and resizing. A selected object can be moved around the slide image by clicking in the dotted selection box and dragging. The handles are for resizing. Clicking in a handle and dragging picks up and moves the handle. The handle at the opposite corner (or opposite end for a line) is fixed, so the object must change size accordingly. It is redrawn as the control point is dragged.

Shift and option keys. When moving ~~or resizing,~~ the shift key can be used to constrain the direction of motion to horizontal, vertical, ~~or 45° angles~~. For resizing, ~~the 45° direction~~ pressing shift key allows an object to be resized without changing its shape (aspect ratio). When resizing, the option key has the effect of anchoring the center of the object rather than the opposite control point. The effect is to change the size of the object without changing its position on the page. Shift and option can be used simultaneously.

maybe 45 shouldn't work on same key

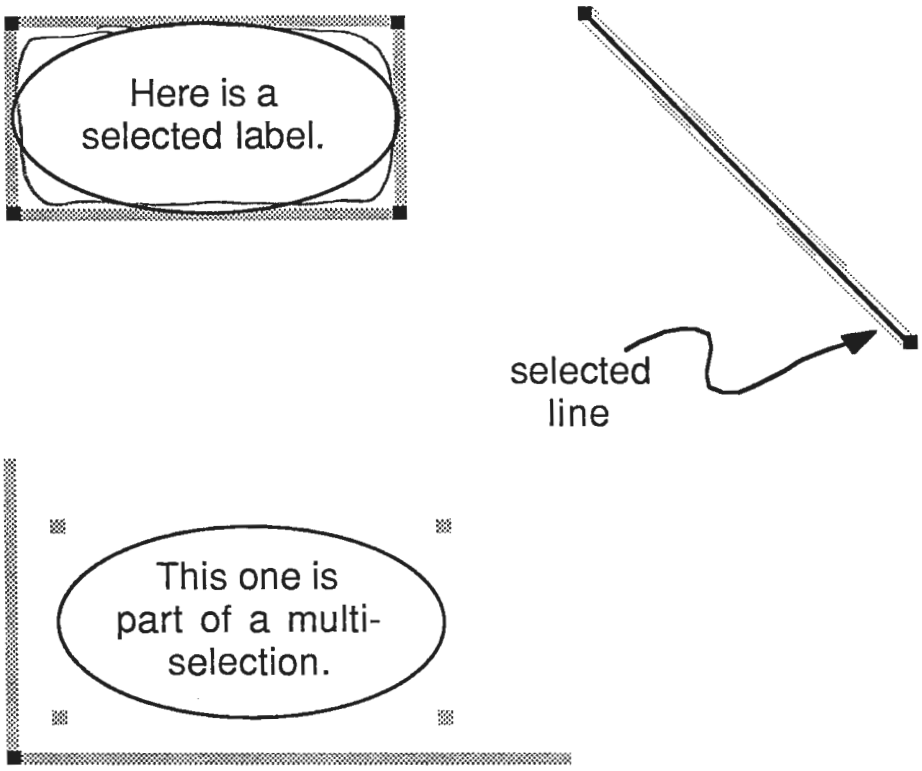
Both keys are also effective when drawing an object. The shift key resticts rectangles to be squares and lines to h, v, d ~~and elipses to be circles~~. The option key allows you to begin drawing by selecting the center of an object instead of one corner. The object then stretches out around the center as you drag.

Marquee selector. Selection of multiple objects is made with a marquee, or with shift-clicking, much as in Macintosh applications. In MacDraw, marquees select only objects that are completely within the box. This differs from the Finder convention of selecting any icon that is partially within the box. MacDraw's method seems designed to simplify selection when objects are stacked in various ways, a situation we expect to be less common in *Presenter*. ~~For the time being therefore, we plan to stick with the Finder convention.~~

~~Multiple selections are shown with the selection box around the entire group of objects. They may all be resized as a group. Because of shift-clicking, it is possible that some of the objects within the selection box are included in the group and some are not. The included objects are distinguished by~~

~~showing them with grayed handles.~~ move bars but no resize handles

Examples of selected objects are shown below.



You will note that this scheme differs considerably from the MacDraw style. Its advantage is clarity. In MacDraw you often pick an object up when you meant to select another and vice versa. Here there is also no confusion about whether the shift key is being used for motion restriction or multiple selection: you must point to a gray move bar or a resize handle to get motion restriction.

Text Boxes

Text boxes are the most distinctive feature of *Presenter*, intended to bring more of a word processor into a drawing program. Within a text box the style of the word processor prevails, but the box as whole can be moved and reshaped as a graphics object on the page. Text boxes can be placed side by side, have a border, or be placed within a drawing. In many ways their manipulation is more natural, more analogous to pen and paper, than working with a word processor.

Paragraphs and characters. A text box contains a series of paragraphs, in the sense that MacWrite and MS-Word use the term: sections of automatically "word-wrapped" text, separated by return characters, with a special indent for the first line. Paragraphs have attributes like line length

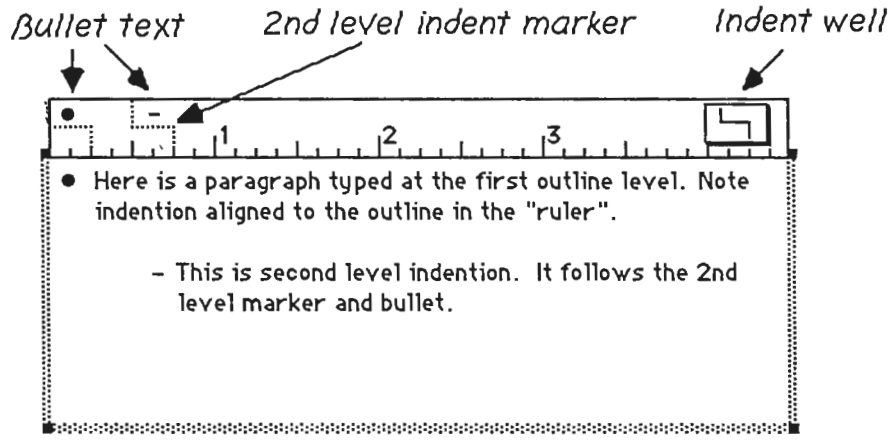
and alignment that do not change within a box. Character properties *can* vary within the box, allowing emphasis of words, etc. In addition to the normal font and style, *Presenter* permits the color of text to be varied. (See Pen menu.)

Editing. To make a text selection, you need an I-beam cursor. You can get this by using the I-beam tool, or you first select the box itself with the pointer. Whenever a single text box is selected (~~or label box—see next section~~) the cursor turns into an I-beam when it is within the box. Editing works in the usual way. Resizing the box reformats the text to fit. Text boxes are expandable so that when more text is inserted they can automatically grow taller to make room for a new line.

Outlines and properties. Although we normally think of a paragraph containing running text, text boxes have special abilities for making paragraphs that form bulleted lists. (Next to titles, lists are thought to be the most common construction used in overhead slide presentations.) The lists generalize to outlines with up to five levels of indentation. Each level has its own properties:

- Amount of indent for first line of an item
- Amount of indent for subsequent lines of an item
- Text used as a bullet, if any, for an item
- Line spacing, within an item and between items

Ruler. All properties but the line spacing are set through a special ruler device that can be displayed at the top of any text box. The ruler has sliding markers to adjust outline indentation :



Indention. Each dotted, offset line is an *indent marker*. Each marker defines the characteristics of paragraphs (list items) at its outline level. The indent markers can be slid back and forth on the ruler, ~~and new ones can be dragged out of the well.~~ The entire marker is dragged by its horizontal section. Its top section defines the indent of the first line of a paragraph; its bottom section defines the indent of subsequent lines. The top and bottom sections can be dragged independently. A hanging indent, where the first line starts to the left of subsequent lines, is shown in the illustration above.

Bullets. Each indent marker can also define a default bullet to be used in lists at its outline level. A few characters can be typed at each indent marker. When you are typing in the text box, a Return starts a new list item (paragraph). As the paragraph is created, the bullet characters are automatically typed for you. The bullet can include a tab character in order to line up following text at the wrapped line indent. The last character of the bullet text also determines the initial font and style for that

outline level.

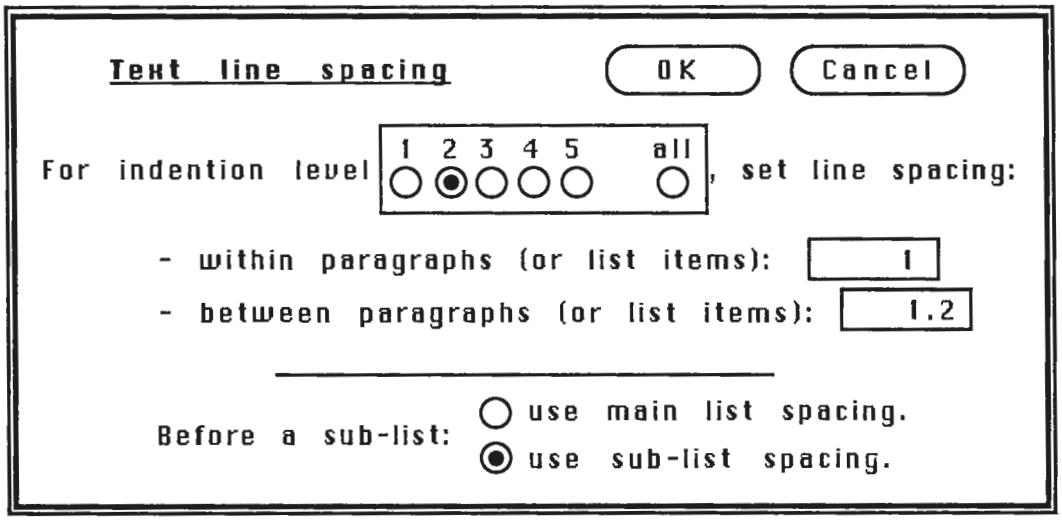
Changing level. When a new paragraph (list item) is created by typing Return, it is at the same outline level as the previous paragraph. To change the outline level, type Command-L or Command-R (left or right) or use the corresponding menu commands. Existing paragraphs can have their level changed by selecting (multiple paragraphs may, of course, be selected) and giving the same command.

Tabs and line breaks. Two special characters may be useful in text boxes. The tab character leaves space up to the next indent marker position. It does not alter the outline level, but acts as an ordinary word processor-style tab using the markers as tab stops. The return character, as already mentioned, begins a new paragraph, but shift-return can be used to cause a line break without starting a new paragraph. Shift-return breaks the word-wrap, but line spacing and indention is unaffected.

tab after the last tab goes to new line?

Auto-increment. If the bullet text contains a number or an isolated letter, it may be possible to have *Presenter* automatically increment the number or letter as new paragraphs are created. It is not clear whether this is the right thing to do, however, since it does not automatically renumber on insertions or deletions, nor alter paragraphs when the bullet is changed in the ruler. These features are not very important, so we plan to resolve them at least partly by experimentation.

Line spacing. Line spacing can be adjusted both within paragraphs and between paragraphs. As mentioned above, different spacing can be set for each level when the box is used as an outline. Spacing is set in a dialog box brought up from a menu command.



The spacing is set for a particular outline level, or for all levels, by selecting radio buttons. The leftmost button corresponds to the leftmost outline level. Selecting a single level displays the current spacing for that level. Selecting all levels shows no values unless all levels are the same.

The numbers you enter are ratios of your desired spacing to the normal spacing for the font. A spacing of 1 means to use single spaced lines. The 1.2 means to increase the spacing by twenty percent; 2 would mean double spacing. Numbers smaller than one reduce the amount of spacing and squeeze the lines together.

explain TT spacing

Spacing between levels. When a sublist has different inter-paragraph spacing from its parent list, the final two buttons give you the opportunity to place it using the parent spacing or the sublist spacing.

Graphical method. There should also be a graphical method that adjusts the line spacing by changing the overall size of the text box. Pulling the bottom of the box down could, for example, adjust the leftmost level's interparagraph spacing. Details of the mechanism are yet to be decided.

Label and Picture Boxes

Any box can be a text box, but boxes used in drawing are usually label boxes. These have no indent rulers and are designed to serve as labels rather than lists or paragraphs. Typing is centered in the box both horizontally and vertically. There is no word wrap; lines are broken by returns. Labels are all in the same font and style.

A label is placed by selecting a box and then typing. If the text fills the box, the box will automatically expand to accommodate new text as it is typed. The box expands in the direction in which text expands: left-aligned text expands on the right, right-aligned text expands on the left, and centered text expands in both directions. Return characters can expand the box at the bottom. To edit the label text in a box, first get an I-beam cursor. You can either use the I-beam tool, or simply select the whole box with the pointer. When a label box is selected, the cursor automatically changes to an I-beam when it is inside the box.

Pictures. In addition to its other characteristics, any box may have a ~~bitmap~~ picture pasted into it. Copy a ~~bitmap~~ to the clipboard, select a box in *Presenter*, and choose Paste. The picture will act like a background for the text. Unlike most Macintosh programs, adjusting the size of a box containing a ~~bitmap~~ does not stretch the ~~bitmap~~—it crops. The picture is pasted in the center of the box. Adjusting the ~~move~~ handles does not move the bitmap on the screen, so all four sides can be cropped independently. Making the box larger than the bitmap adds fill space between the edge of the bitmap and the border of the box. When a new selection is made (after Undo is no longer possible, actually), the cropped portion of the bitmap is discarded so the size cannot be later increased.

Tables

A table is a special kind of box that behaves like an array of label boxes. It serves to group the cells together as one object, and provides several added conveniences. Ruling lines, for example, can be manipulated and colored much as any line, but are actually part of the table itself. The text of any block of cells can be selected, cut, and pasted—especially nice for exchanging information with spreadsheet files.

Creating tables. Any box can be converted into a table by means of a menu command called "Table". The command subdivides the original box into cells approximately 1 1/4 inches wide by 3/8

inch high. It then presents a dialog box in which to specify the number of columns and rows. Initial values depend on how many standard cells fit in the original box. The dialog also offers a check box to make all rulings visible. Invisible ruling shows as dotted lines when the table tool is in use (see below.) The Table command can be given again to change the number of rows or columns.

A table acts somewhat like a big box composed of smaller label boxes. The overall table is selectable as an ordinary graphics object and can be moved or resized as usual. The cell boxes contain text in the same way a label does, but they will not automatically expand to accommodate new typing.

Selecting cells. In a table, text can be selected within a single cell box, or the entire text of any rectangular group of boxes can be selected. You select cell text with the I-beam cursor as you would in a label box. Edit as usual, changing font, etc. Keyboard commands (cursor keys?) can be used to jump the selection to neighboring boxes. If you begin a selection in one cell and drag to a neighboring cell, a block of cells is selected—or rather the text contents of a block of cells is selected. Any rectangular block of cells can selected this way. Their selection is shown by inverting the color of the entire cell. (A shortcut might be nice for selecting entire rows or columns, but I'm not sure.)

Cut and paste. Selected cells can have text commands applied and can be cut and pasted. Cutting (or copying) text concatenates the text from the selected boxes, in row-major order, saving the cell boundaries as tab characters. Pasting such text into a table selection places it into the selected cells the same way. If there are fewer cells selected than there are tab characters in the clipboard, the extra is not pasted in. Pasting into non-tables works in the usual way with tab characters remaining in the text. If the cells are pasted into a spreadsheet, they are all transferred in their original relationship.

Table tool. To construct attractive tables, the ruling lines can be adjusted with a special selection tool. They can be selected either singly or in groups by shift-clicking or marquee. Once selected, they can be colored with the Pen menu or have their width changed with the Lines menu. Chosing a color of none makes them invisible except when the table tool is in use.

Changing a table. A selected line, or group of lines running the same direction, can be dragged within the table to change the dimensions of the rows and columns. Dragging the lines at the edges of the table changes the size of the overall table. A row or column disappears entirely (contents too?) if one of the lines is dragged onto its neighbor or if the Clear command is given. Selecting a line and giving the Duplicate command creates a new row or column, below or to the right of the line. The Distribute command can be used to evenly space a group of lines in the table.

The Drawing Window

On the Macintosh, the window looks like this:

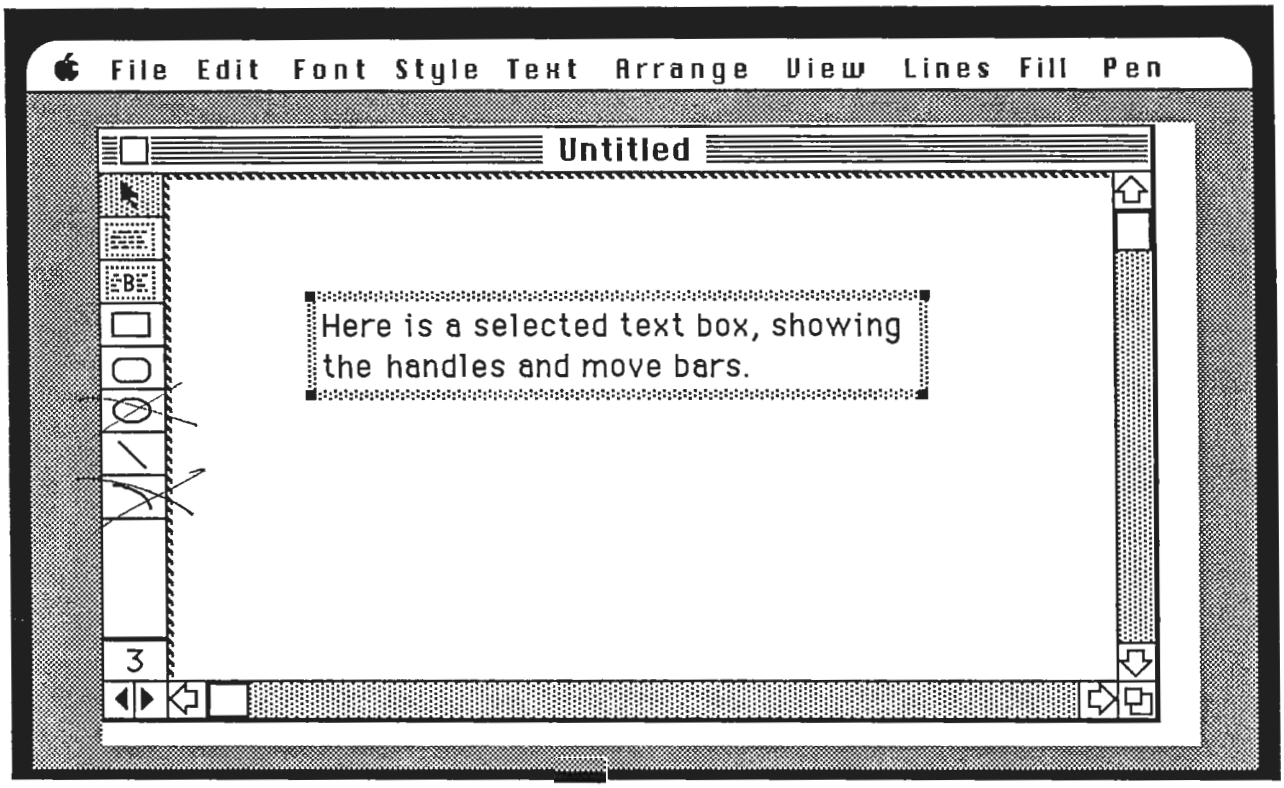


Image number. The similarity to a MacDraw window is obvious. In the lower left corner, though, note that a slide number is shown as well as two buttons for going forward or backward by one slide. To go to a particular number, click on the current number and type the new one. Going forward from the last slide adds a new slide to the file. Inserting slides elsewhere can only be done in Slide Sorter or Title Sorter view.

Tools. The tools column on the left is similar to the one in MacDraw, but it is missing the freehand and polygon tools and includes text box tools. The arrow is the selection tool. When it is in use, the pointer is shaped like the arrow. The other tools are used for drawing. When a drawing tool is in use, the cursor is a cross-hair. The current tool button is overlayed with a gray pattern.

There is one default text tool, but others, labeled with a letter like the *B* box here, can be added by the user—see the Canonize command. The default text tool is preset to have no borders or fill and has a ruler suitable for making bulleted lists. The curve tool does not draw eliptical arcs like the one in MacDraw. It draws curves specified by three points like the curve tool in MS Paint.

Drawing objects. A tool is selected by clicking on its button. To draw with the tool you normally press the mouse button at one corner and stretch (drag) to an opposite corner. If you draw while

when have drawing tool
click without drag converts to arrow.

holding down the Option (control) key, the initial point is the center of the box and all sides are stretched at once. If you hold down the shift key, you can drag only in horizontal, vertical, or 45° directions. ~~Once the object is drawn the selector arrow automatically becomes the current tool.~~

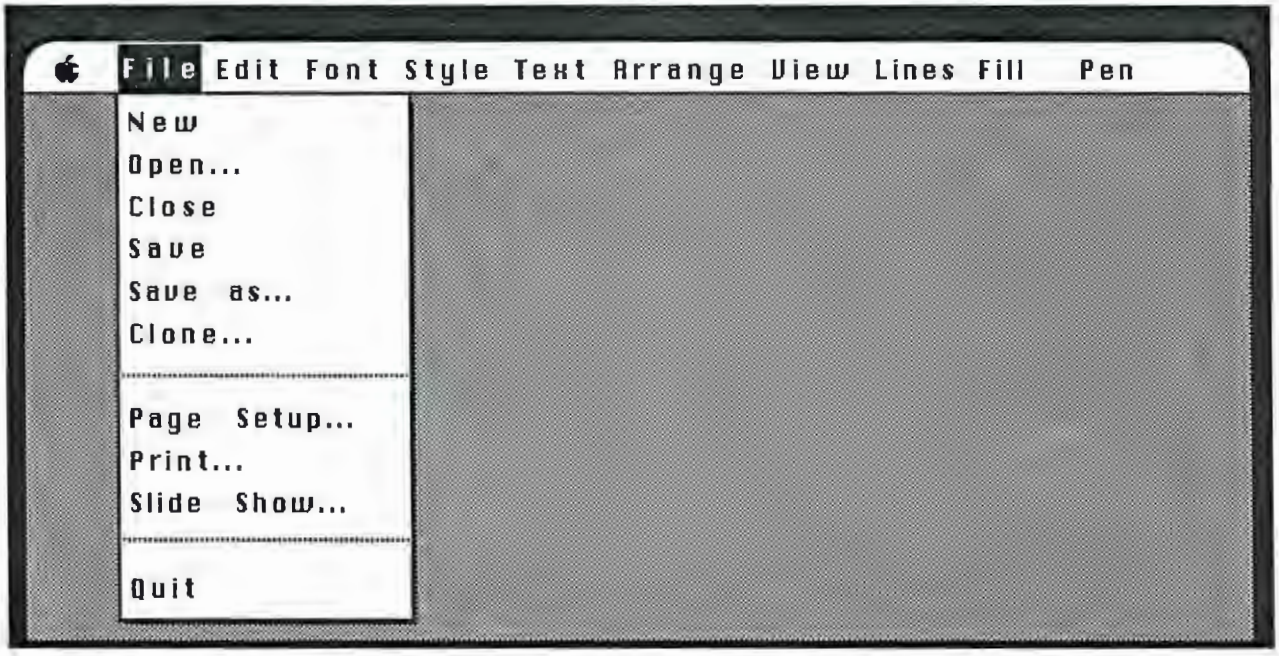
You can also double click a drawing tool button. This immediately draws the corresponding object in a preset size and at a preset position. This is not very useful for the built-in tools, but for for the user-defined tools it may be helpful.

Reshaping. Unlike MacDraw, there are really only two types of objects, lines and boxes. Any of the box types can be converted into one of the others, ~~and any line can later be curved.~~ These changes are made by the Reshape command in the edit menu. ~~The curve tool actually works by first drawing a straight line and then immediately throwing you into Reshape in order to make the curve.~~

On Microsoft Windows, the *Presenter* window will look different. The toolbox may be a popup window instead of a part of the main window. The pen and fill menus may be combined into a single popup window as well.

Commands

The menu bar is designed to be familiar to Macintosh users. The major titles bear some similarity to those in MacDraw, but there are many differences in the actual command set.



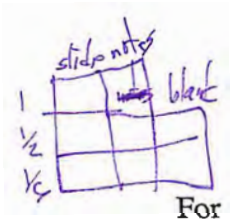
The top of the **File** menu contains all the usual commands. **Open** and **New** are active only when no presentation is open. Except for **Quit**, all other commands in this menu are active only when a presentation *is* open.

As explained under the View command, *Presenter* operates in a number of different modes or views. Opening and closing presentation files does not change the operating mode.

Clone is similar to Save As, but does not save any of the actual images. It saves a presentation that has the custom tools, the chosen fonts, and the model slide, but it has no images. Its purpose is to make it easy to prepare presentations with the same style.

Page Setup determines page size and orientation. All slide images in one presentation have the same orientation. It also permits a choice of aspect ratio for the images. The normal ratio of 4:3 is used on any letter size paper. It gives an image size about 10 by $7^1/2$ inches. The alternative ratio of 3:2 is primarily for compatibility with 35 millimeter slides. It is used whenever legal size paper is chosen. Depending on the printer, it gives an image area about 12 by 8 inches or 10 by 7 inches.

The **Print** command, in addition to the usual options, includes a choice styles:

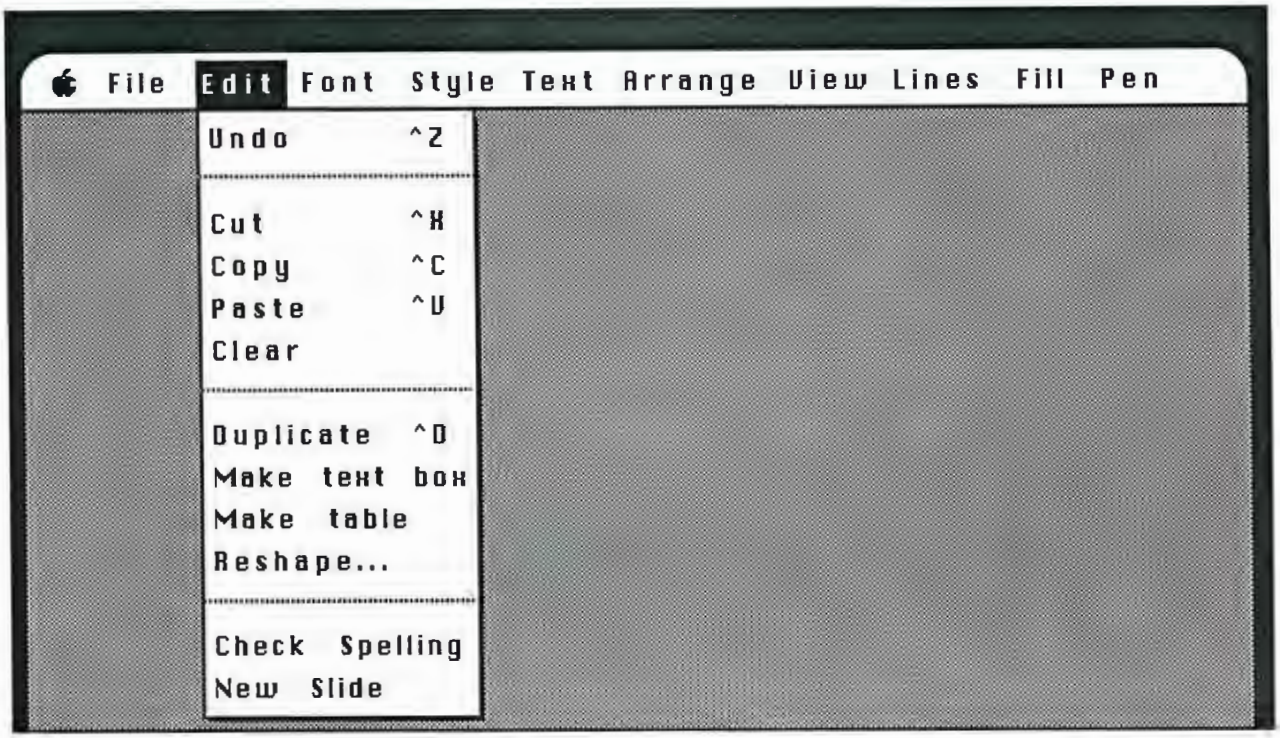


- One full-sized slide per page.
- One half-sized slide per page, with notes for that slide.
- Two half-sized slides per page.
- Three slides per page with room for listeners notes.
- Six (eight?) per page for reference use.

For the half-sized formats, two landscape slides are printed on a portrait page. When notes are included, a landscape slide appears at the top with notes below. If the slides have portrait orientation, they are printed side by side on a landscape page, or with slide to the left and notes on the right.

Slide Show displays the images one at a time on the computer screen. No window borders, scroll bars, menu bars, etc. are shown and the slides are full size. Some cropping may be necessary to fit on the screen, but it should be minimal unless the images have portrait orientation. A dialog box offers a choice of slide changing control: automatically timed at a chosen interval, or by pressing the mouse button.

Save → saves whole pres
Opening a pres that is already open asks if you want to revert?
Is that the way Word works



The **Edit** menu is mostly standard. **Undo** will work after any editing operation, either from the edit menu or from resizing/reshaping with the selection pointer. It can also cancel any change of attributes such as fill, font, etc. Both text and graphics may be cut and pasted. When text is selected with the I-beam pointer, it goes to the clipboard along with its character attributes. When the clipboard contains only text, pasting will create a new, borderless rectangle to contain the text as a label. ? Graphics objects can be cut and pasted much as they are in McDraw. For cut and paste to apply to graphics objects, there must not be any text selection or flashing insertion point.

could use Write format

Graphics and text can also be cut and pasted between *Presenter* and other applications. Standard formats for structured graphics, text (no font/style/size), and bitmaps are supported. Between *Presenter* documents, whole slides can be cut and pasted together with their notes. This can only be done in special views called Slide Sorter and Title Sorter (see below). ~~No other data types can be cut and pasted in sorter views.~~ Can't cut + paste objects

what about changing aspect ratio etc

A presentation contains an environment consisting of the model slide, a set of fonts (see Font menu), etc. When a slide is copied from one presentation into another, it aquires these characteristics from its new environment. They are ~~not~~ carried across from the source. model slide from dest fonts from source

Duplicate creates an exact copy of an object, or group of objects, offset by a small amount. If the duplicate is immediately moved and a second duplicate is made, the new duplicate will be positioned like the one that was manually moved. This feature, borrowed from MacDraw, is useful for creating rows and columns of objects, for example. Once a series or table of boxes is constructed, text may be typed into them. ~~If necessary, boxes can be converted to text boxes to get the benefit of word wrapped paragraphs, etc. Contents of boxes are not copied by the Duplicate command, so new rows or columns can be cloned from existing ones without copying the existing text.~~

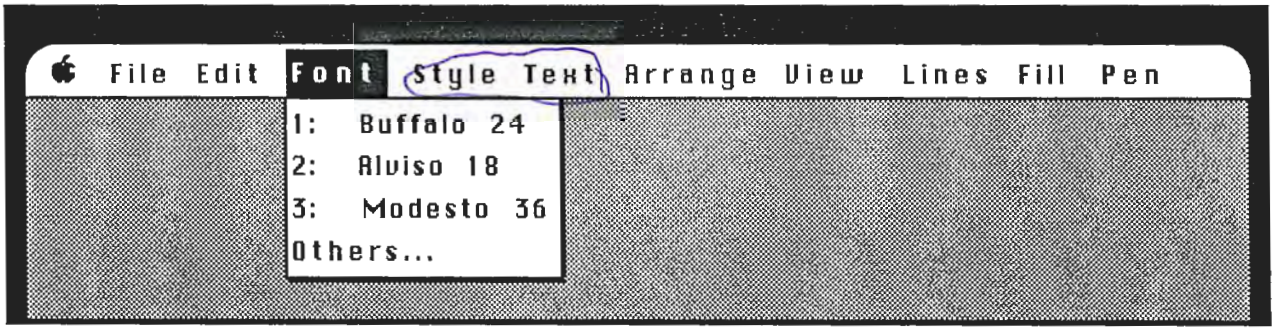
Make text box and **Make table** convert ordinary (label) boxes into these special types. The table command offers a dialog box in which the number of columns and rows, as well as the ruling, can be specified. If it is applied to an existing table, these parameters can be changed.

In *Presenter*, all lines are equivalent, and all boxes are equivalent. The **Reshape** command allows differently curved lines or boxes to be converted into another type. For boxes, a dialog allows changing the roundness of a box, converting rectangles into round-corner boxes of various rounding radii, or converting all the way to an ellipse. For lines, a control point appears that permits its curve to be reshaped.

Check Spelling activates the built-in spelling checker. It presents a dialog asking whether the entire presentation should be checked or only a range of slides. When checking is under way, another dialog box shows the buttons Cancel, Next, and Add-to-Dictionary. While checking, only Cancel is active. If a possible error is found, it is highlighted and the other two buttons are enabled. You may retype the word if you wish. If you make other edits before proceeding with the spelling check, the check will resume where you left off. The spelling dialog remains on top until canceled or completed.

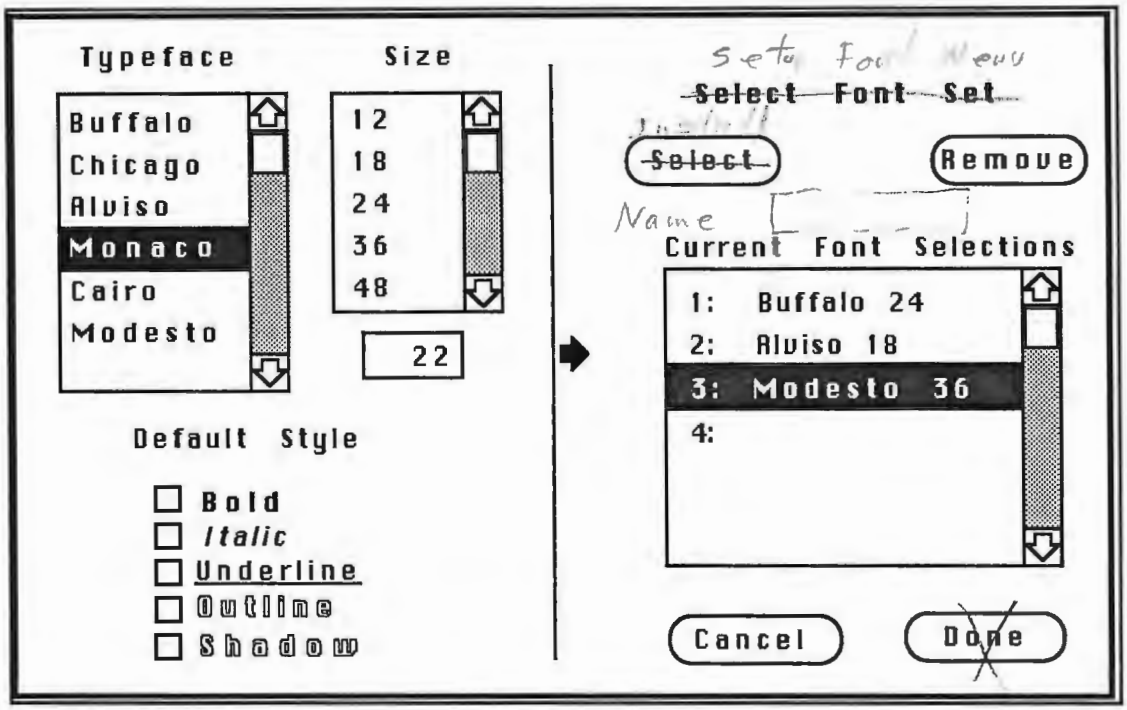
New Slide is for inserting a new slide into the middle of the presentation. It adds a slide after the current one, or if slide sorter or table sorter are in use, at the selected position.

The **Style** menu is exactly like MacPaint, permitting bold, italic, etc. It does not contain type sizes or any other options. The **Font** menu, however, is more unusual.



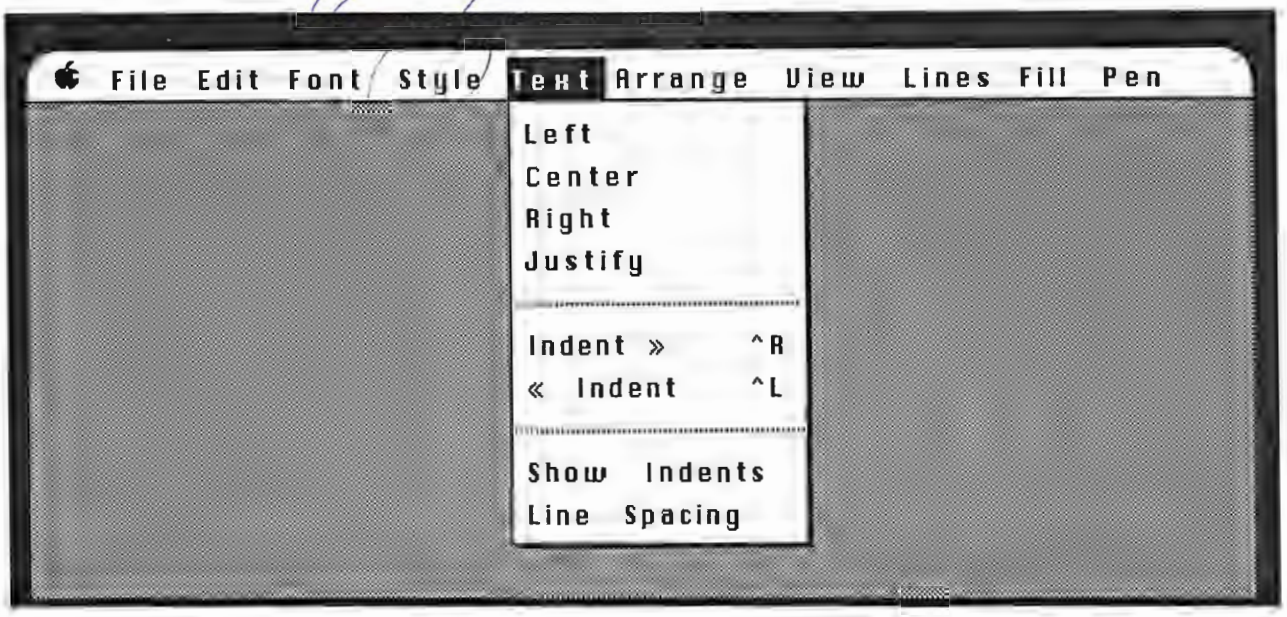
Unlike Macintosh applications where the font menu in fact offers a list of typefaces, the *Presenter* menu offers actual fonts—size included. Any font used in the document appears in the font menu. The fonts are numbered, and the numbers are important. They allow you, for example, to later change font 2 from Alviso 18 to Monaco 18. Everywhere font 2 is used in the document, the font will be changed. This makes it much easier to maintain a consistent style thoughout a presentation.

Fonts can be added to the menu, or changed or removed, with the **Others...** command. It brings up a dialog box like the one below.



The dialog box works somewhat like the font selection in Microsoft Word. You select a typeface, a size, and a default style. Clicking the Select button replaces the highlighted font with your selections. A spare slot (number 4 in this case) is always available at the end of the font list so you can add new ones. You can also select a font from the list and remove it ~~if it is not in use.~~ Removed fonts do not appear in the font menu, but the fonts are not renumbered. You can continue updating the font list until you are satisfied, and then click Done to install the new list.

move to Font

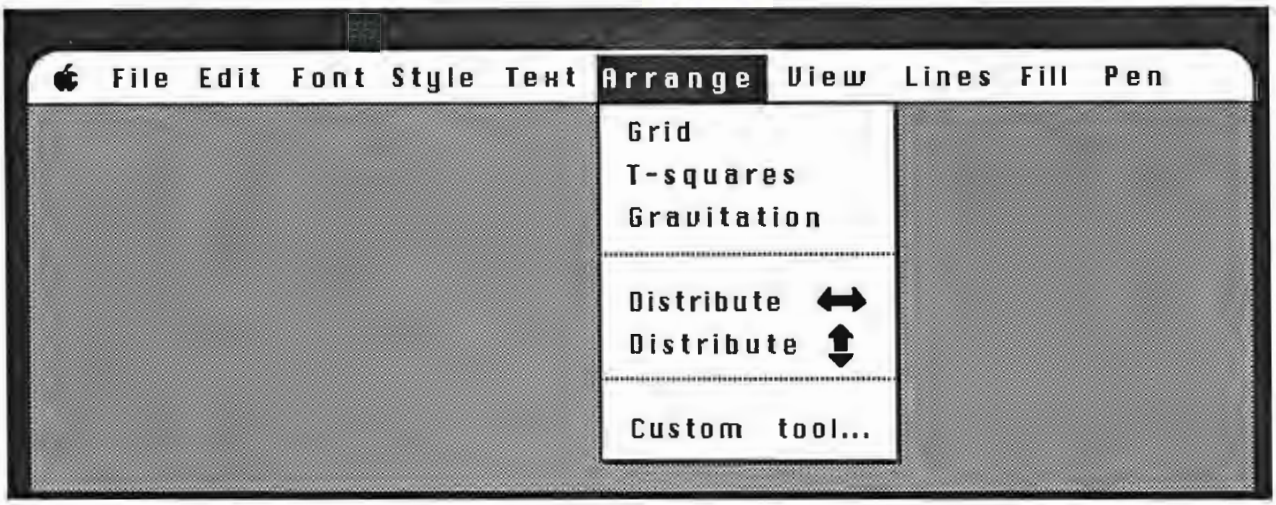


Text controls the attributes of text boxes. The first four items control the alignment in the usual way. They apply to both text boxes ~~and label boxes~~ and are active whenever one of these is selected. They always apply to the box as a whole. — should apply to our para

The **Indent** commands change the outline level of a paragraph (same as command-L and command-R). They are enabled only when there is a text selection in a text box. They work on the paragraph containing the selection. If the selection spans multiple paragraphs, all the selected paragraphs are affected. The command is disallowed ~~if one of the~~ paragraphs is already at outermost or innermost level and thus can't be moved further. ineffective on a that para

Show Indents brings up the indent ruler on the selected text box. It can be used only when a single text box is selected. The ruler disappears when a new selection is made so there is no Hide Ruler command. Turn to hide indent. object

Line Spacing allows inter-line leading to be finely controlled. It brings up the dialog box discussed earlier in the section *Text Boxes*.



Arrange is for positioning objects in common ways. The first three commands are checkmark options that enable various alignment gadgets. Grid and Gravitation are on by default, but their state is carried with the document. The distribute commands move objects.

Grid turns on an invisible 1/16 inch grid. New objects are constrained to have their centers and sides (or end points in the case of lines) lie on the grid. Note that this means that the objects will be an *even* number of grid divisions vertically and horizontally; dimensions are thus in multiples of 1/8 inch. When an object is moved, its center can only be moved to a grid point. When an object is resized, the handles can only be moved to grid points. When moving groups, selection rectangle remains grid-aligned.

T-squares displays T-squares like FileMaker's. The numeric position of each square can be shown either at end of its line or over by toolbox, depending on what is easier and looks better. Position is shown relative to center of page, not upper left corner. If Grid is on, T-squares can only be moved to grid lines.

Gravitation pulls objects to an guide line. When T-squares are off, the edges and centers of objects are "magnetically" attracted to an invisible vertical line running down the center of the page. When drawing or moving an object, its edge or center will snap to the center line if it lies within about 1/4 inch (or 1/4 the length of the object, if smaller than one inch.)

When T-squares are on, it causes both T-squares to act as the invisible center line does, attracting edges and centers of objects. The center line is not used while T-squares are on, since the vertical square takes its place.

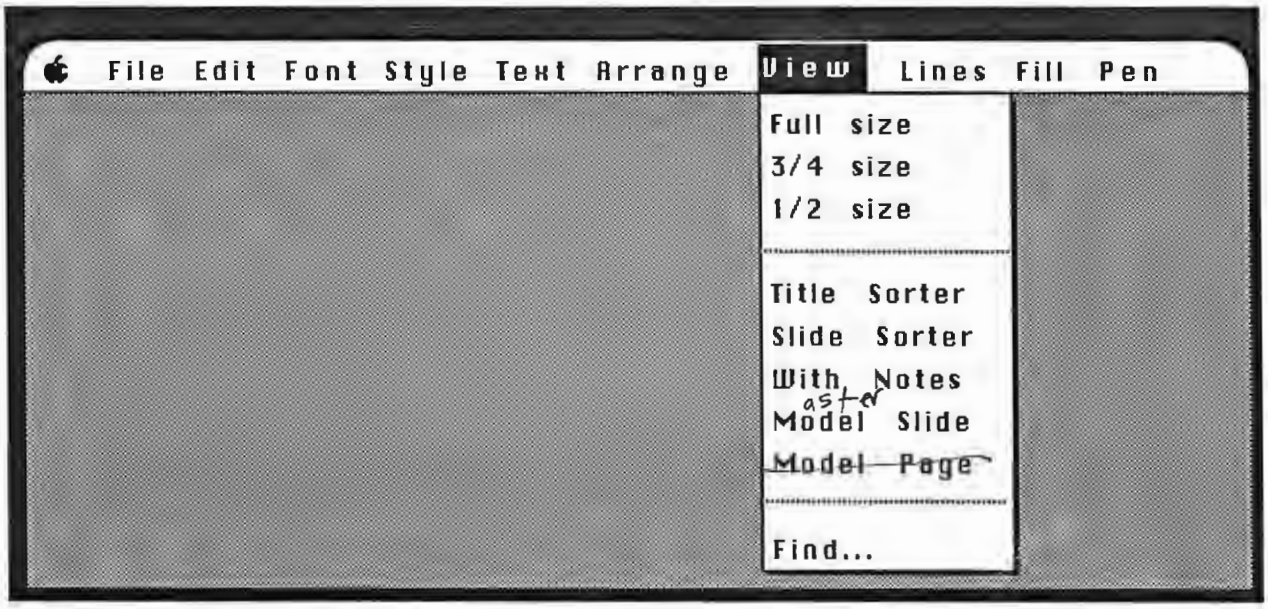
The gravitation effect is more powerful than the grid. If an object is not grid aligned, moving it normally aligns its center on-grid. If a T-square line attracts the edge of such an object, the edge will will be aligned even if its center goes off-grid.

Distribute is for evenly spacing a group of objects, either horizontally or vertically. To see how the commands work, we'll look at the horizontal version; the vertical is the same except for orientation. First select the group of objects, then give the command.

The leftmost and rightmost objects do not move. The other objects have their horizontal position adjusted so that the amount of space between each pair is the same. Their ordering from left to right is unchanged and their vertical positions are unchanged. You should note that the leftmost and rightmost objects may be the same object, say when one of the objects is a box enclosing the others. In that case the objects are distributed across the box. If only one object and its containing box are selected, this has the effect of centering it in the box. A useful special case comes up if the group contains only one object: the object is simply centered on the page.

Custom tool (canonize) is the command that installs a custom drawing tool, particularly useful for text boxes. You can draw a text box, display its ruler, and adjust its properties as you like. Then you use Canonize to turn that box into a drawing tool. A dialog box gives you the opportunity to identify it with a letter such as O for outline. The dialog box also allows you to discard tools you no longer want. Any text box, regardless of the tool that was used to create it, can be altered to suit the circumstances. Custom tools are provided solely as a convenience to creation.

Any group of objects can be canonized. Select the objects and then give the Canonize command to install the new tool. When you double click on a custom tool, the object(s) are immediately drawn at the same location and in the same size that they had when canonized. This makes it easy to repeat a common feature on many slides.



View handles scaling of the image to the window, and is also the key to several unique features. The first two commands reduce the image of a slide to fit within the current window dimensions, either horizontally or vertically. (Depending on the shape of the image and the shape of the window, one of the two commands will show the entire slide.) Return to full size with the Normal Size command.

Title Sorter changes the window to a scrolling list of all image titles. Double click a title to display that slide. Single click and drag to select sequences of images that can be cut or copied. Pasting images inserts them in front of the selected images. This mode allows the same operations as the

slide sorter view discussed below, but works on titles instead of miniature images.

The **Slide Sorter** command brings up a multi-slide view of the document, allowing slides to be viewed together and rearranged. About 12 to 16 miniature slides should be viewable at one time. They are arranged in the window in rows, the number in each row depending on the width of the window. A vertical scroll bar permits scrolling through the entire presentation. Individual slides are selected by pointing and clicking. Selecting the slide pops up its title to help identify the miniature image. A continuous groups of slides is selected by clicking and dragging. Clicking between slides selects an insertion point between slides. The slides can be cut, copied, and pasted much as if they were words in a text document. Only slides can be pasted in this view, however. No other type of data can be used, even if it is from *Presenter*, but slides can be freely copied or moved between presentations.

The slide sorter is also a handy means of jumping around from slide to slide: double-clicking a slide returns to normal view at the clicked slide.

The **With Notes** command changes the view to include the half-size slide and accompanying notes. The notes consist of a single text box occupying the remainder of the page. The text box can be altered in any way, but cannot be deleted. No other objects can be created. The slide image, though visible, cannot be edited. When notes are being shown, the Notes command is checked on the menu. You can go to Titles or Slide Sorter views and back without effecting Notes Mode. To turn off notes, give the command again.

Model Slide brings up the model slide for this document that contains decorative frames, page numbers, and a title box. Features of the model slide are shown, but are not editable, on the actual slides. They are drawn on the model in the same way they would be on an ordinary slide.

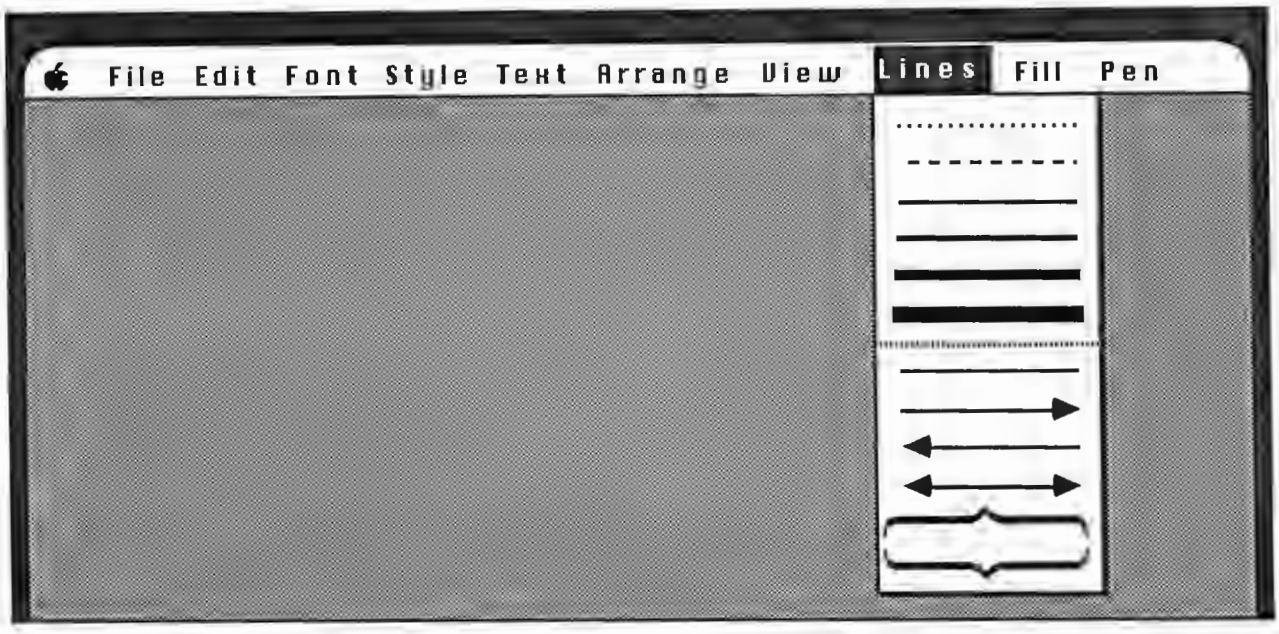
The model slide contains an indestructable text box containing the word "Title". You can change the characteristics of the box to suit your taste—position, typeface, etc. An empty copy of this box will appear on every new slide you create. The text you type into it, if any, will be the title for that slide.

The slide number is simply a box containing the text "#". When a slide is printed, the two characters are replaced by the slide number. The font, etc. is determined by the font of the "#".

When viewing the model slide, the Model Slide command shows a checkmark. You can return to view the image you had before selecting Model Slide by selecting it again. You can also return by pressing either the forward or backward buttons in the window. Finally, you can go to a specific slide by entering it s number in the window box, or using the Titles or Slide Sorter views.

Model Page serves a purpose analogous to model slide, but allows the printed page to be decorated with page number, running heads, etc. It applies only on the printed pages and its image is not superimposed on the on-screen slides. Its purpose is not to apply to the slide images printed full size but to the condensed printed formats where added page info may be useful.

The **Find** command searches all the slides in the presentation for text. It is similar to the search commands in word processors, but there is no "replace" option. The dialog box does, however, offer a choice of whether to search notes or not. The dialog disappears from the screen if the search is successful.



The **Lines, Fill**, and **Pen** menus have the same purpose as those in MacDraw, but there are a number of differences. The lines menu does not offer an invisible line, but it does offer two types of dotted lines. [Note: lines styles available will depend partly upon features offered in Microsoft Windows.] In addition to the arrow types, a line can also be turned into a brace, a very handy type of line in text slides.

The **Fill** menu chooses the fill for boxes, but it is primarily for choosing colors rather than patterns. On monochrome screens, the colors will appear as patterns. It additionally offers a small choice of cross-hatching patterns and none (transparent). The default is none.

Pen determines the color of box edges, lines, and text. The pen menu can be applied to a text selection or graphics selection. If color is not in use, only solid black and solid white may be available, but this also depends on Microsoft Windows' graphics.

Presenter

Dennis Austin
August 19, 1985

These notes describe the basic plan for Forethought's new product. You will find this version very similar to that of July 12, but there is a great deal of subtle refinement. The only substantial change is a new section about tables. I have also not changed the illustrations to the Microsoft style, but the specifications should be complete enough to make up a Windows story board.

Theme

Presenter is a color drawing program designed primarily for producing overhead slides. It is distinguished from MacDraw-type progams primarily by specializing in textual elements:

- Paragraphs with mixed fonts, sizes, and styles.
- Finely adjustable line spacing and placement.
- Wide choice of typefaces and large sizes suitable for graphic displays.
- Word-processor-like manipulation of text.

It can draw diagrams and insert decorative art, but is really aimed more narrowly at graphic text displays. With the addition of text flowing into multiple elements, it would almost make a simple page layout program. Other differences from MacDraw make it more suitable for preparing presentations:

- Multiple images (slides) that can be rearranged within a single file (presentation).
- Multiple windows that can show different images from the same presentation.
- Notes that can be attached to a drawing and viewed selectively.
- A variety of printing methods: full size, two on a page, with or without notes, etc.
- Model image overlays that allow pages share borders and similar features.
- Special help for making bulleted lists and tables.

To keep it simple, the drawing potential of the program is less capable than that of MacDraw. For example, *Presenter* has:

- No polygons or smoothing.
- No free-hand curves.
- No stacking order changes.
- No rotate or flip.
- No grouping or locking.

- No rulers.

These features are useful and I hate to see them go, but in user interfaces, as in programming languages, the problem is not so much what to put in as what to leave out. I'm not sure that this is the right list of omissions, but simplification is important.

Presentations

Unlike other graphics programs, *Presenter* does not produce files or documents that contain a single picture. A *Presenter* document comprises an entire presentation, or perhaps a section of a very long presesentation. There is no limit on the number of images (slides) in a presentation, but the product will not be designed to handle large volumes, say over one hundred. When an actual presentation contains more than about fifty images, it can be better handled by dividing it into sections, each in its own document.

Notes, titles, and printing formats. Each image has a title and about one-half page of notes associated with it. The title appears on the image itself, but notes are selectively visible. Images can be printed by themselves to make actual slides, or the they can be printed (in reduced size) along with their notes to make handouts for people who miss the presentation, etc. Reduced size printing with two or three slides per page is also available to make handouts for the audience.

Model slide. There is a special image called the model slide that adds common features to all images in the presentation. This allows for the addition of borders, logos, dates, and other material that appears on every image. The format of titles is also determined here, but unlike the other features, it can be changed on the individual images.

Slide Sorter. The individual images are normally viewed one at a time, or one per window if multiple windows are in use. In slide sorter mode, however, up to about sixteen miniature images can be viewed in a single window. Although the actual images can be hard to distinguish in this view, their titles appear in readable type whenever they are selected. Slides can be cut and pasted in order to reorder the presentation or to move slides between files. This view also provides a handy method of finding and viewing a particular image. An analogous view called Title Sorter has the same function, but shows only the slide titles instead of their images.

Slide show. Although most presentations are printed for final display, *Presenter* allows you to use the computer as a "projector" if you like. The images are shown one at a time full size (landscape only) on the computer's screen without tools, menu bar, or other windows. The show can be presented manually, pressing the mouse button to change slides, or automatically timed.

Drawing Slide Images

Even though a presentation may be entirely text with no diagrams or other graphics, the slide images are still "drawn" to some extent. *Presenter* is, at its heart, a structured graphics editor like MacDraw. It works with objects such as lines and rectangles that can be moved, resized, and overlapped to produce many different effects. Text is held as the contents of objects, typically borderless rectangles.

Objects. There are two basic types of drawing objects: two dimensional (boxes) and one-dimensional (lines). Lines have width, of course, and in fact they have a number of other attributes, but they do not have contents as boxes do. Boxes and lines both have location and dimensions that are specifiable with two points. A new object is drawn by first selecting a tool (determining the type of the object) and then positioning the pointer, clicking, and dragging. For lines the beginning and end of the drag determine the end points of the line, and for boxes they determine opposite corners of a rectangle.

Once the object is drawn it can be moved, resized, or have its attributes changed. Below are some attributes that can be selected by menu commands or other means.

Boxes

- Shape: rectangle, round rectangle, ellipse
- Border: thickness, style (dotted, etc), color [or none]
- Fill: color or pattern [or none]
- Contents: text, label, picture [or none]

Lines

- Thickness, style, color
- Curved or straight
- Special form: Arrows, braces

Selection

When the cursor is an arrow, it is used for selection, moving, and resizing objects. Clicking selects the smallest enclosing box, or, if the cursor is close to a line, it selects the line. When an box is selected, a dotted box appears around it with four control points, or "handles", at the corners. When a line is selected, a dotted box surrounds the line and handles appear at both ends.

Moving and resizing. A selected object can be moved around the slide image by clicking in the dotted selection box and dragging. The handles are for resizing. Clicking in a handle and dragging picks up and moves the handle. The handle at the opposite corner (or opposite end for a line) is fixed, so the object must change size accordingly. It is redrawn as the control point is dragged.

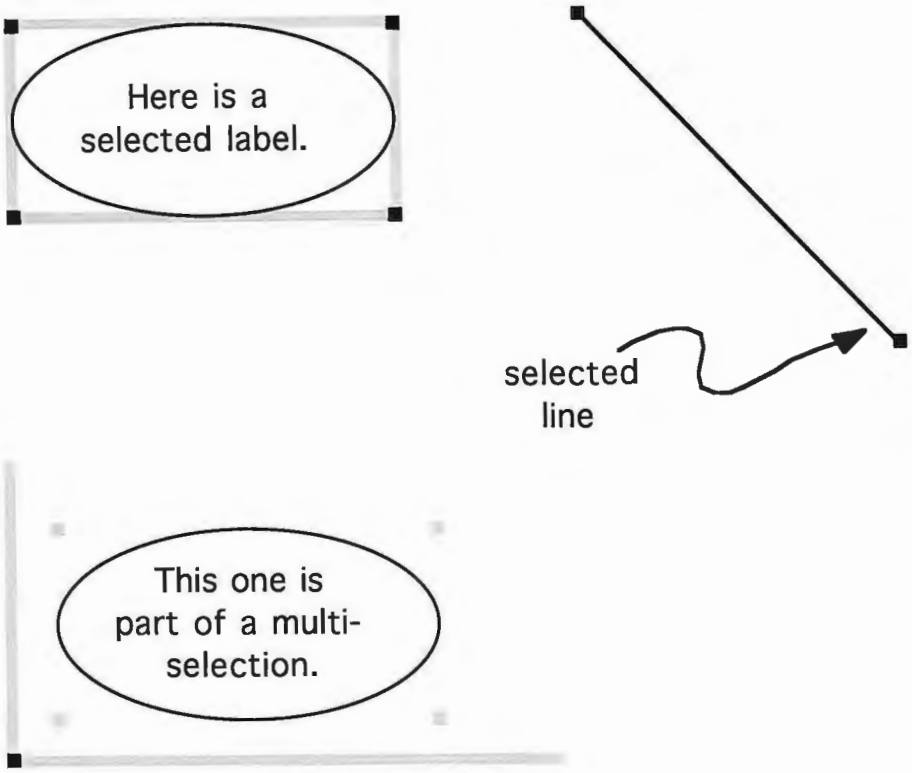
Shift and option keys. When moving or resizing, the shift key can be used to constrain the direction of motion to horizontal, veritcal, or 45° angles. For resizing, the 45° direction allows an object to be resized without changing its shape (aspect ratio). When resizing, the option key has the effect of anchoring the center of the object rather than the opposite control point. The effect is to change the size of the object without changing its position on the page. Shift and option can be used simultaneously.

Both keys are also effective when drawing an object. The shift key resticts rectangles to be squares and elipses to be circles. The option key allows you to begin drawing by selecting the center of an object instead of one corner. The object then stretches out around the center as you drag.

Marquee selector. Selection of multiple objects is made with a marquee, or with shift-clicking, much as in Macintosh applications. In MacDraw, marquees select only objects that are completely within the box. This differs from the Finder convention of selecting any icon that is partially within the box. MacDraw's method seems designed to simplify selection when objects are stacked in various ways, a situation we expect to be less common in *Presenter*. For the time being therefore, we plan to stick with the Finder convention.

Multiple selections are shown with the selection box around the entire group of objects. They may all be resized as a group. Because of shift-clicking, it is possible that some of the objects within the selection box are included in the group and some are not. The included objects are distinguished by showing them with grayed handles.

Examples of selected objects are shown below.



You will note that this scheme differs considerably from the MacDraw style. Its advantage is clarity. In MacDraw you often pick an object up when you meant to select another and vice versa. Here there is also no confusion about whether the shift key is being used for motion restriction or multiple selection: you must point to a gray move bar or a resize handle to get motion restriction.

Text Boxes

Text boxes are the most distinctive feature of *Presenter*, intended to bring more of a word processor into a drawing program. Within a text box the style of the word processor prevails, but the box as whole can be moved and reshaped as a graphics object on the page. Text boxes can be placed side by side, have a border, or be placed within a drawing. In many ways their manipulation is more natural, more analogous to pen and paper, than working with a word processor.

Paragraphs and characters. A text box contains a series of paragraphs, in the sense that MacWrite and MS-Word use the term: sections of automatically "word-wrapped" text, separated by return characters, with a special indent for the first line. Paragraphs have attributes like line length and alignment that do not change within a box. Character properties *can* vary within the

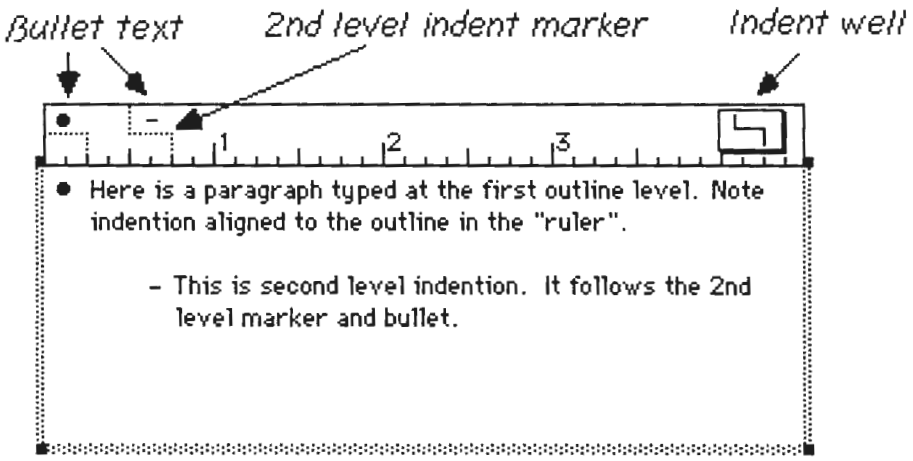
box, allowing emphasis of words, etc. In addition to the normal font and style, *Presenter* permits the color of text to be varied. (See Pen menu.)

Editing. To make a text selection, you need an I-beam cursor. You can get this by using the I-beam tool, or you first select the box itself with the pointer. Whenever a single text box is selected (or label box—see next section) the cursor turns into an I-beam when it is within the box. Editing works in the usual way. Resizing the box reformats the text to fit. Text boxes are expandable so that when more text is inserted they can automatically grow taller to make room for a new line.

Outlines and properties. Although we normally think of a paragraph containing running text, text boxes have special abilities for making paragraphs that form bulleted lists. (Next to titles, lists are thought to be the most common construction used in overhead slide presentations.) The lists generalize to outlines with up to five levels of indentation. Each level has its own properties:

- Amount of indent for first line of an item
- Amount of indent for subsequent lines of an item
- Text used as a bullet, if any, for an item
- Line spacing, within an item and between items

Ruler. All properties but the line spacing are set through a special ruler device that can be displayed at the top of any text box. The ruler has sliding markers to adjust outline indentation :



Indention. Each dotted, offset line is an *indent marker*. Each marker defines the characteristics of paragraphs (list items) at its outline level. The indent markers can be slid back and forth on the ruler, and new ones can be dragged out of the well. The entire marker is dragged by its horizontal section. Its top section defines the indent of the first line of a paragraph; its bottom section defines the indent of subsequent lines. The top and bottom sections can be dragged independently. A hanging indent, where the first line starts to the left of subsequent lines, is shown in the illustration above.

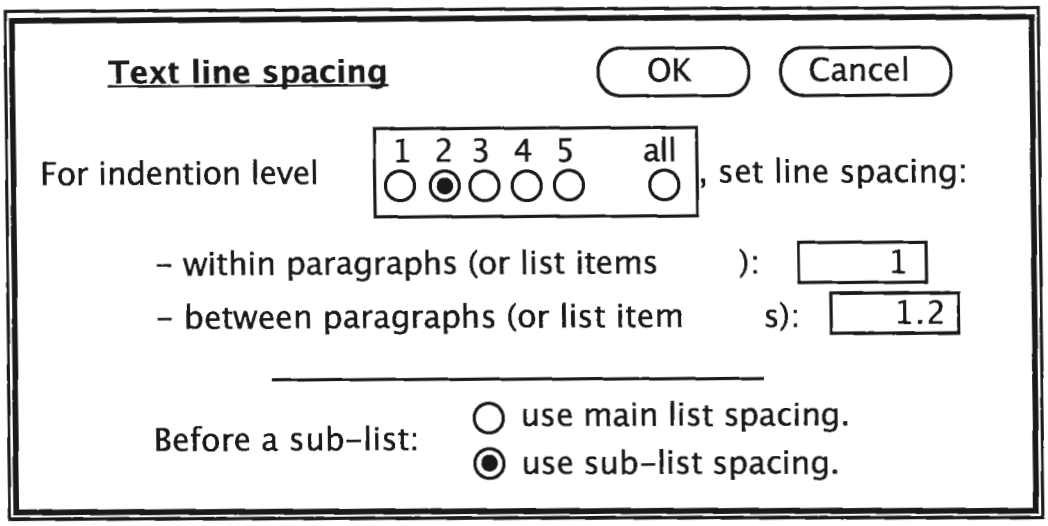
Bullets. Each indent marker can also define a default bullet to be used in lists at its outline level. A few characters can be typed at each indent marker. When you are typing in the text box, a Return starts a new list item (paragraph). As the paragraph is created, the bullet characters are automatically typed for you. The bullet can include a tab character in order to line up following text at the wrapped line indent. The last character of the bullet text also determines the initial font and style for that outline level.

Changing level. When a new paragraph (list item) is created by typing Return, it is at the same outline level as the previous paragraph. To change the outline level, type Command-L or Command-R (left or right) or use the corresponding menu commands. Existing paragraphs can have their level changed by selecting (multiple paragraphs may, of course, be selected) and giving the same command.

Tabs and line breaks. Two special characters may be useful in text boxes. The tab character leaves space up to the next indent marker position. It does not alter the outline level, but acts as an ordinary word processor-style tab using the markers as tab stops. The return character, as already mentioned, begins a new paragraph, but shift-return can be used to cause a line break without starting a new paragraph. Shift-return breaks the word-wrap, but line spacing and indention is unaffected.

Auto-increment. If the bullet text contains a number or an isolated letter, it may be possible to have *Presenter* automatically increment the number or letter as new paragraphs are created. It is not clear whether this is the right thing to do, however, since it does not automatically renumber on insertions or deletions, nor alter paragraphs when the bullet is changed in the ruler. These features are not very important, so we plan to resolve them at least partly by experimentation.

Line spacing. Line spacing can be adjusted both within paragraphs and between paragraphs. As mentioned above, different spacing can be set for each level when the box is used as an outline. Spacing is set in a dialog box brought up from a menu command.



The spacing is set for a particular outline level, or for all levels, by selecting radio buttons. The leftmost button corresponds to the leftmost outline level. Selecting a single level displays the current spacing for that level. Selecting all levels shows no values unless all levels are the same.

The numbers you enter are ratios of your desired spacing to the normal spacing for the font. A spacing of 1 means to use single spaced lines. The 1.2 means to increase the spacing by twenty percent; 2 would mean double spacing. Numbers smaller than one reduce the amount of spacing and squeeze the lines together.

Spacing between levels. When a sublist has different inter-paragraph spacing from its parent list, the final two buttons give you the opportunity to place it using the parent spacing or the sublist spacing.

Graphical method. There should also be a graphical method that adjusts the line spacing by changing the overall size of the text box. Pulling the bottom of the box down could, for example, adjust the leftmost level's interparagraph spacing. Details of the mechanism are yet to be decided.

Label and Picture Boxes

Any box can be a text box, but boxes used in drawing are usually label boxes. These have no indent rulers and are designed to serve as labels rather than lists or paragraphs. Typing is centered in the box both horizontally and vertically. There is no word wrap; lines are broken by returns. Labels are all in the same font and style.

A label is placed by selecting a box and then typing. If the text fills the box, the box will automatically expand to accommodate new text as it is typed. The box expands in the direction in

which text expands: left-aligned text expands on the right, right-aligned text expands on the left, and centered text expands in both directions. Return characters can expand the box at the bottom. To edit the label text in a box, first get an I-beam cursor. You can either use the I-beam tool, or simply select the whole box with the pointer. When a label box is selected, the cursor automatically changes to an I-beam when it is inside the box.

Pictures. In addition to its other characteristics, any box may have a bitmap picture pasted into it. Copy a bitmap to the clipboard, select a box in *Presenter*, and choose Paste. The picture will act like a background for the text. Unlike most Macintosh programs, adjusting the size of a box containing a bitmap does not stretch the bitmap—it crops. The picture is pasted in the center of the box. Adjusting the move handles does not move the bitmap on the screen, so all four sides can be cropped independently. Making the box larger than the bitmap adds fill space between the edge of the bitmap and the border of the box. When a new selection is made (after Undo is no longer possible, actually), the cropped portion of the bitmap is discarded so the size cannot be later increased.

Tables

A table is a special kind of box that behaves like an array of label boxes. It serves to group the cells together as one object, and provides several added conveniences. Ruling lines, for example, can be manipulated and colored much as any line, but are actually part of the table itself. The text of any block of cells can be selected, cut, and pasted—especially nice for exchanging information with spreadsheet files.

Creating tables. Any box can be converted into a table by means of a menu command called "Table". The command subdivides the original box into cells approximately 1 1/4 inches wide by 3/8 inch high. It then presents a dialog box in which to specify the number of columns and rows. Initial values depend on how many standard cells fit in the original box. The dialog also offers a check box to make all rulings visible. Invisible ruling shows as dotted lines when the table tool is in use (see below.) The Table command can be given again to change the number of rows or columns.

A table acts somewhat like a big box composed of smaller label boxes. The overall table is selectable as an ordinary graphics object and can be moved or resized as usual. The cell boxes contain text in the same way a label does, but they will not automatically expand to accommodate new typing.

Selecting cells. In a table, text can be selected within a single cell box, or the entire text of any rectangular group of boxes can be selected. You select cell text with the I-beam cursor as you would in a label box. Edit as usual, changing font, etc. Keyboard commands (cursor keys?) can be used to jump the selection to neighboring boxes. If you begin a selection in one cell and drag to a neighboring cell, a block of cells is selected—or rather the text contents of a block of cells is selected. Any rectangular block of cells can selected this way. Their selection is shown by inverting the color of the entire cell. (A shortcut might be nice for selecting entire rows or columns, but I'm not sure.)

Cut and paste. Selected cells can have text commands applied and can be cut and pasted. Cutting (or copying) text concatenates the text from the selected boxes, in row-major order, saving the cell boundaries as tab characters. Pasting such text into a table selection places it into the selected cells the same way. If there are fewer cells selected than there are tab characters in the clipboard, the extra is not pasted in. Pasting into non-tables works in the usual way with tab characters remaining in the text. If the cells are pasted into a spreadsheet, they are all transferred in their original relationship.

Table tool. To construct attractive tables, the ruling lines can be adjusted with a special selection tool. They can be selected either singly or in groups by shift-clicking or marquee. Once selected, they can be colored with the Pen menu or have their width changed with the Lines menu. Chosing a color of none makes them invisible except when the table tool is in use.

Changing a table. A selected line, or group of lines running the same direction, can be dragged within the table to change the dimensions of the rows and columns. Dragging the lines at the edges of the table changes the size of the overall table. A row or column disappears entirely (contents too?) if one of the lines is dragged onto its neighbor or if the Clear command is given. Selecting a line and giving the Duplicate command creates a new row or column, below or to the right of the line. The Distribute command can be used to evenly space a group of lines in the table.

The Drawing Window

On the Macintosh, the window looks like this:

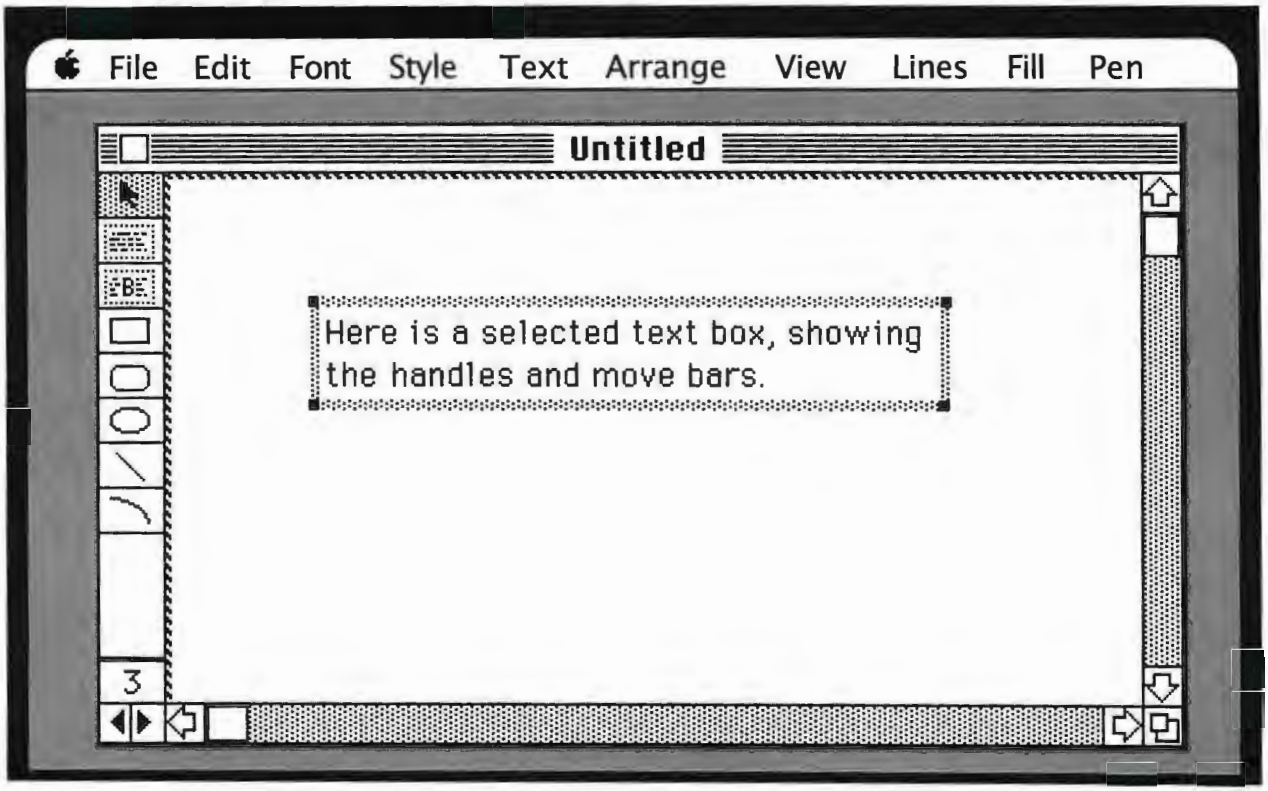


Image number. The similarity to a MacDraw window is obvious. In the lower left corner, though, note that a slide number is shown as well as two buttons for going forward or backward by one slide. To go to a particular number, click on the current number and type the new one. Going forward from the last slide adds a new slide to the file. Inserting slides elsewhere can only be done in Slide Sorter or Title Sorter view.

Tools. The tools column on the left is similar to the one in MacDraw, but it is missing the freehand and polygon tools and includes text box tools. The arrow is the selection tool. When it is in use, the pointer is shaped like the arrow. The other tools are used for drawing. When a drawing tool is in use, the cursor is a cross-hair. The current tool button is overlayed with a gray pattern.

There is one default text tool, but others, labeled with a letter like the *B* box here, can be added by the user—see the Canonize command. The default text tool is preset to have no borders or fill and has a ruler suitable for making bulleted lists. The curve tool does not draw eliptical arcs like the one in MacDraw. It draws curves specified by three points like the curve tool in MS Paint.

Drawing objects. A tool is selected by clicking on its button. To draw with the tool you normally press the mouse button at one corner and stretch (drag) to an opposite corner. If you draw while holding down the Option (control) key, the initial point is the center of the box and all sides are stretched at once. If you hold down the shift key, you can drag only in horizontal, vertical, or 45° directions. Once the object is drawn the selector arrow automatically becomes the current tool.

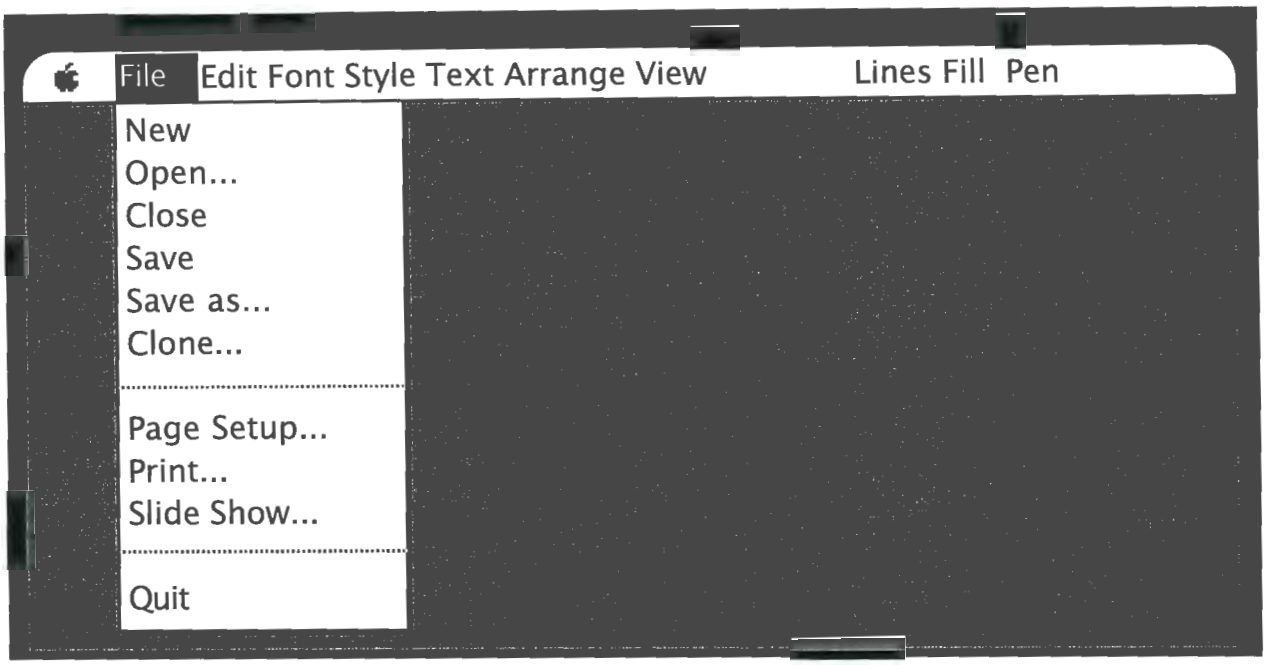
You can also double click a drawing tool button. This immediately draws the corresponding object in a preset size and at a preset position. This is not very useful for the built-in tools, but for for the user-defined tools it may be helpful.

Reshaping. Unlike MacDraw, there are really only two types of objects, lines and boxes. Any of the box types can be converted into one of the others, and any line can later be curved. These changes are made by the Reshape command in the edit menu. The curve tool actually works by first drawing a straight line and then immediately throwing you into Reshape in order to make the curve.

On Microsoft Windows, the *Presenter* window will look different. The toolbox may be a popup window instead of a part of the main window. The pen and fill menus may be combined into a single popup window as well.

Commands

The menu bar is designed to be familiar to Macintosh users. The major titles bear some similarity to those in MacDraw, but there are many differences in the actual command set.



The top of the **File** menu contains all the usual commands. **Open** and **New** are active only when no presentation is open. Except for **Quit**, all other commands in this menu are active only when a presentation *is* open.

As explained under the View command, *Presenter* operates in a number of different modes or views. Opening and closing presentation files does not change the operating mode.

Clone is similar to Save As, but does not save any of the actual images. It saves a presentation that has the custom tools, the chosen fonts, and the model slide, but it has no images. Its purpose is to make it easy to prepare presentations with the same style.

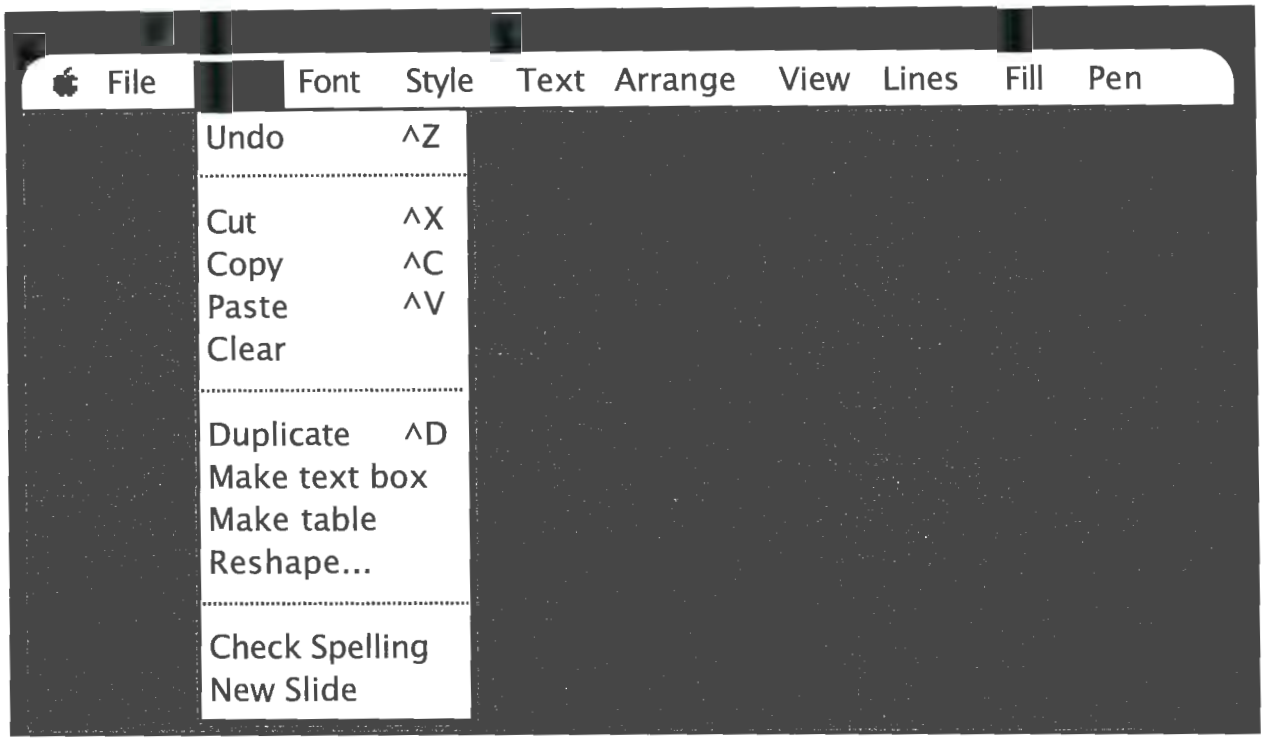
Page Setup determines page size and orientation. All slide images in one presentation have the same orientation. It also permits a choice of aspect ratio for the images. The normal ratio of 4:3 is used on any letter size paper. It gives an image size about 10 by 7 1/2 inches. The alternative ratio of 3:2 is primarily for compatibility with 35 millimeter slides. It is used whenever legal size paper is chosen. Depending on the printer, it gives an image area about 12 by 8 inches or 10 by 7 inches.

The **Print** command, in addition to the usual options, includes a choice styles:

- One full-sized slide per page.
- One half-sized slide per page, with notes for that slide.
- Two half-sized slides per page.
- Three slides per page with room for listeners notes.
- Six (eight?) per page for reference use.

For the half-sized formats, two landscape slides are printed on a portrait page. When notes are included, a landscape slide appears at the top with notes below. If the slides have portrait orientation, they are printed side by side on a landscape page, or with slide to the left and notes on the right.

Slide Show displays the images one at a time on the computer screen. No window borders, scroll bars, menu bars, etc. are shown and the slides are full size. Some cropping may be necessary to fit on the screen, but it should be minimal unless the images have portrait orientation. A dialog box offers a choice of slide changing control: automatically timed at a chosen interval, or by pressing the mouse button.



The **Edit** menu is mostly standard. **Undo** will work after any editing operation, either from the edit menu or from resizing/reshaping with the selection pointer. It can also cancel any change of attributes such as fill, font, etc. Both text and graphics may be cut and pasted. When text is selected with the I-beam pointer, it goes to the clipboard along with its character attributes. When the clipboard contains only text, pasting will create a new, borderless rectangle to contain the text as a label. Graphics objects can be cut and pasted much as they are in McDraw. For cut and paste to apply to graphics objects, there must not be any text selection or flashing insertion point.

Graphics and text can also be cut and pasted between *Presenter* and other applications. Standard formats for structured graphics, text (no font/style/size), and bitmaps are supported. Between

Presenter documents, whole slides can be cut and pasted together with their notes. This can only be done in special views called Slide Sorter and Title Sorter (see below). No other data types can be cut and pasted in sorter views.

A presentation contains an environment consisting of the model slide, a set of fonts (see Font menu), etc. When a slide is copied from one presentation into another, it aquires these characteristics from its new environment. They are not carried across from the source.

Duplicate creates an exact copy of an object, or group of objects, offset by a small amount. If the duplicate is immediately moved and a second duplicate is made, the new duplicate will be positioned like the one that was manually moved. This feature, borrowed from MacDraw, is useful for creating rows and columns of objects, for example. Once a series or table of boxes is constructed, text may be typed into them. If necessary, boxes can be converted to text boxes to get the benefit of word wrapped paragraphs, etc. Contents of boxes are not copied by the Duplicate command, so new rows or columns can be cloned from existing ones without copying the existing text.

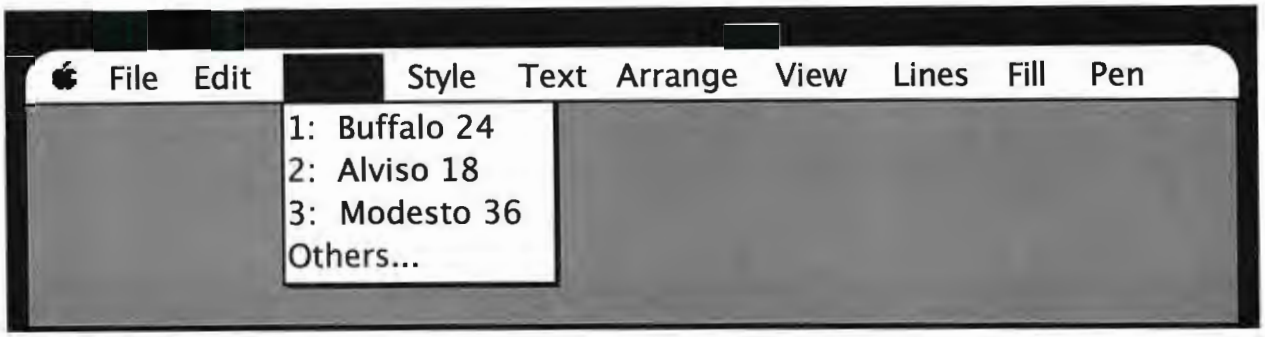
Make text box and **Make table** convert ordinary (label) boxes into these special types. The table command offers a dialog box in which the number of columns and rows, as well as the ruling, can be specified. If it is applied to an existing table, these parameters can be changed.

In *Presenter*, all lines are equivalent, and all boxes are equivalent. The **Reshape** command allows differently curved lines or boxes to be converted into another type. For boxes, a dialog allows changing the roundness of a box, converting rectangles into round-corner boxes of various rounding radii, or converting all the way to an ellipse. For lines, a control point appears that permits its curve to be reshaped.

Check Spelling activates the built-in spelling checker. It presents a dialog asking whether the entire presentation should be checked or only a range of slides. When checking is under way, another dialog box shows the buttons Cancel, Next, and Add-to-Dictionary. While checking, only Cancel is active. If a possible error is found, it is highlighted and the other two buttons are enabled. You may retype the word if you wish. If you make other edits before proceeding with the spelling check, the check will resume where you left off. The spelling dialog remains on top until canceled or completed.

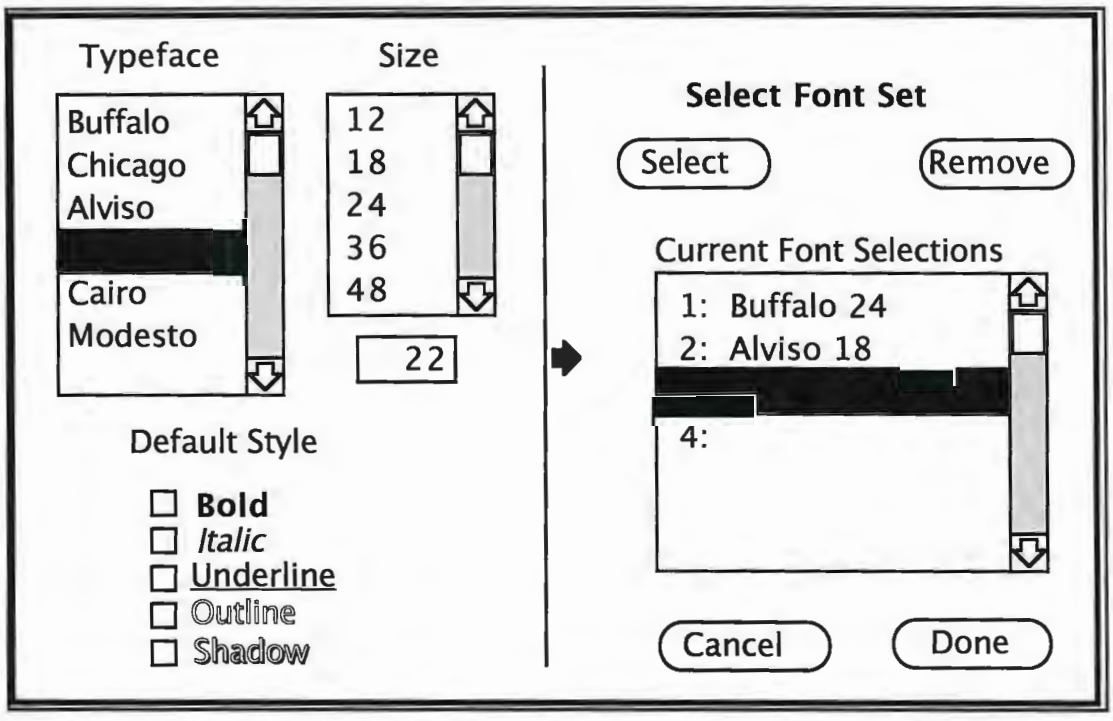
New Slide is for inserting a new slide into the middle of the presentation. It adds a slide after the current one, or if slide sorter or table sorter are in use, at the selected position.

The **Style** menu is exactly like MacPaint, permitting bold, italic, etc. It does not contain type sizes or any other options. The **Font** menu, however, is more unusual.



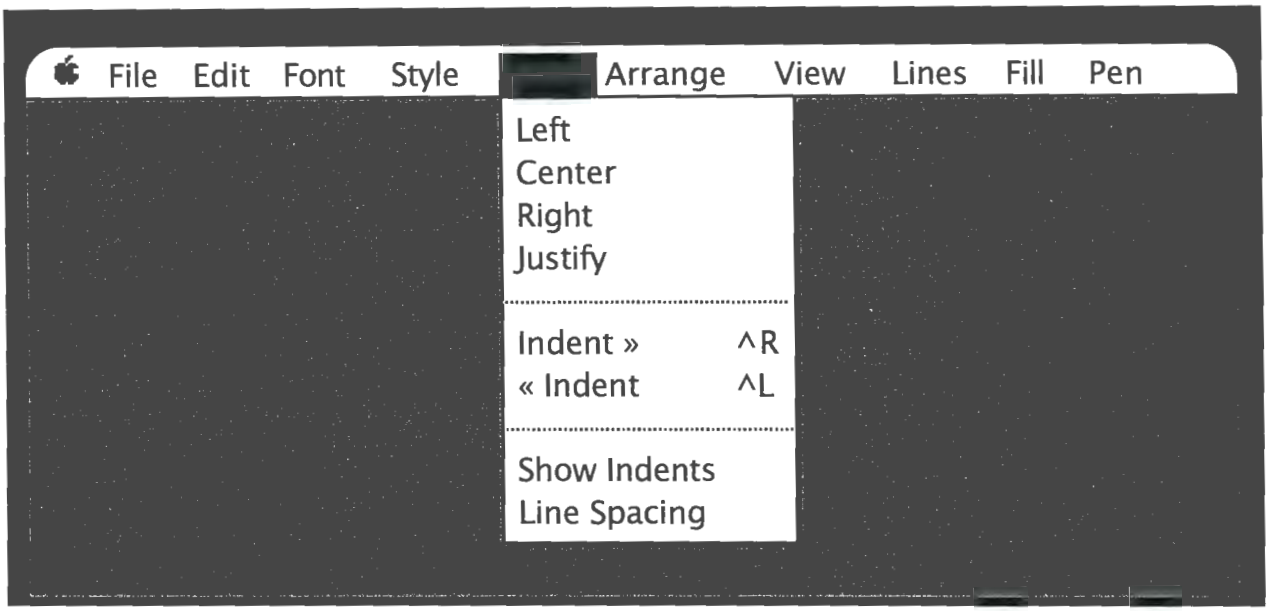
Unlike Macintosh applications where the font menu in fact offers a list of typefaces, the *Presenter* menu offers actual fonts—size included. Any font used in the document appears in the font menu. The fonts are numbered, and the numbers are important. They allow you, for example, to later change font 2 from Alviso 18 to Monaco 18. Everywhere font 2 is used in the document, the font will be changed. This makes it much easier to maintain a consistent style thoughout a presentation.

Fonts can be added to the menu, or changed or removed, with the **Others...** command. It brings up a dialog box like the one below.



The dialog box works somewhat like the font selection in Microsoft Word. You select a typeface, a size, and a default style. Clicking the Select button replaces the highlighted font with your

selections. A spare slot (number 4 in this case) is always available at the end of the font list so you can add new ones. You can also select a font from the list and remove it if it is not in use. Removed fonts do not appear in the font menu, but the fonts are not renumbered. You can continue updating the font list until you are satisfied, and then click Done to install the new list.

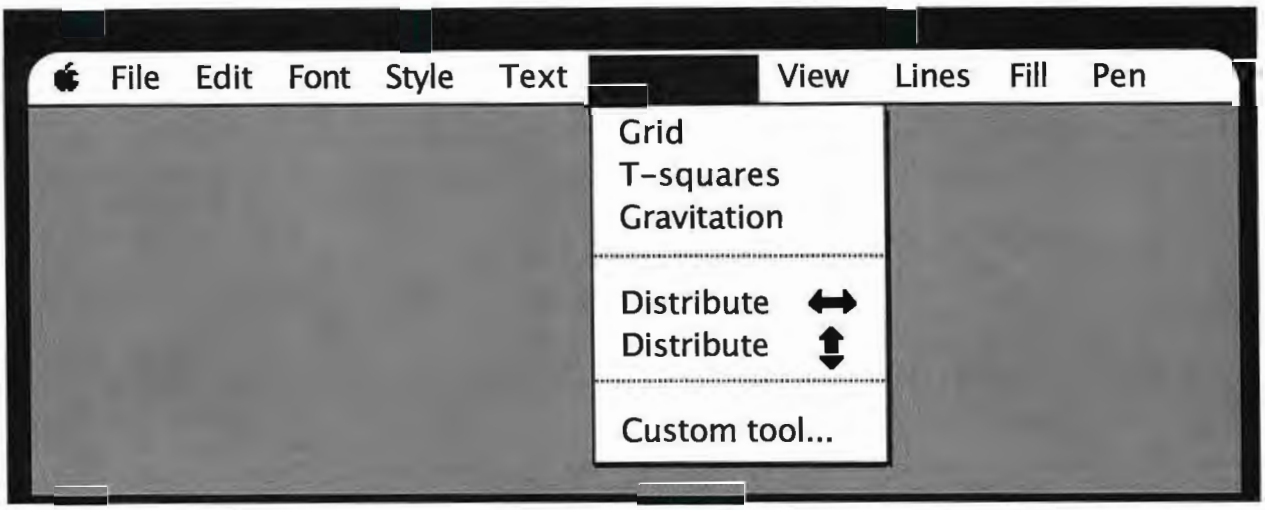


Text controls the attributes of text boxes. The first four items control the alignment in the usual way. They apply to both text boxes and label boxes and are active whenever one of these is selected. They always apply to the box as a whole.

The **Indent** commands change the outline level of a paragraph (same as command-L and command-R). They are enabled only when there is a text selection in a text box. They work on the paragraph containing the selection. If the selection spans multiple paragraphs, all the selected paragraphs are affected. The command is disallowed if one of the paragraphs is already at outermost or innermost level and thus can't be moved further.

Show Indents brings up the indent ruler on the selected text box. It can be used only when a single text box is selected. The ruler disappears when a new selection is made so there is no Hide Ruler command.

Line Spacing allows inter-line leading to be finely controlled. It brings up the dialog box discussed earlier in the section *Text Boxes*.



Arrange is for positioning objects in common ways. The first three commands are checkmark options that enable various alignment gadgets. Grid and Gravitation are on by default, but their state is carried with the document. The distribute commands move objects.

Grid turns on an invisible 1/16 inch grid. New objects are constrained to have their centers and sides, or end points in the case of lines, lie on the grid. Note that this means that the objects will be an *even* number of grid divisions vertically and horizontally; dimensions are thus in multiples of 1/8 inch. When an object is moved, its center can only be moved to a grid point. When an object is resized, the handles can only be moved to grid points. When moving groups, selection rectangle remains grid-aligned.

T-squares displays T-squares like FileMaker's. The numeric position of each square can be shown either at end of its line or over by toolbox, depending on what is easier and looks better. Position is shown relative to center of page, not upper left corner. If Grid is on, T-squares can only be moved to grid lines.

Gravitation pulls objects to an guide line. When T-squares are off, the edges and centers of objects are "magnetically" attracted to an invisible vertical line running down the center of the page. When drawing or moving an object, its edge or center will snap to the center line if it lies within about 1/4 inch (or 1/4 the length of the object, if smaller than one inch.)

When T-squares are on, it causes both T-squares to act as the invisible center line does, attracting edges and centers of objects. The center line is not used while T-squares are on, since the vertical square takes its place.

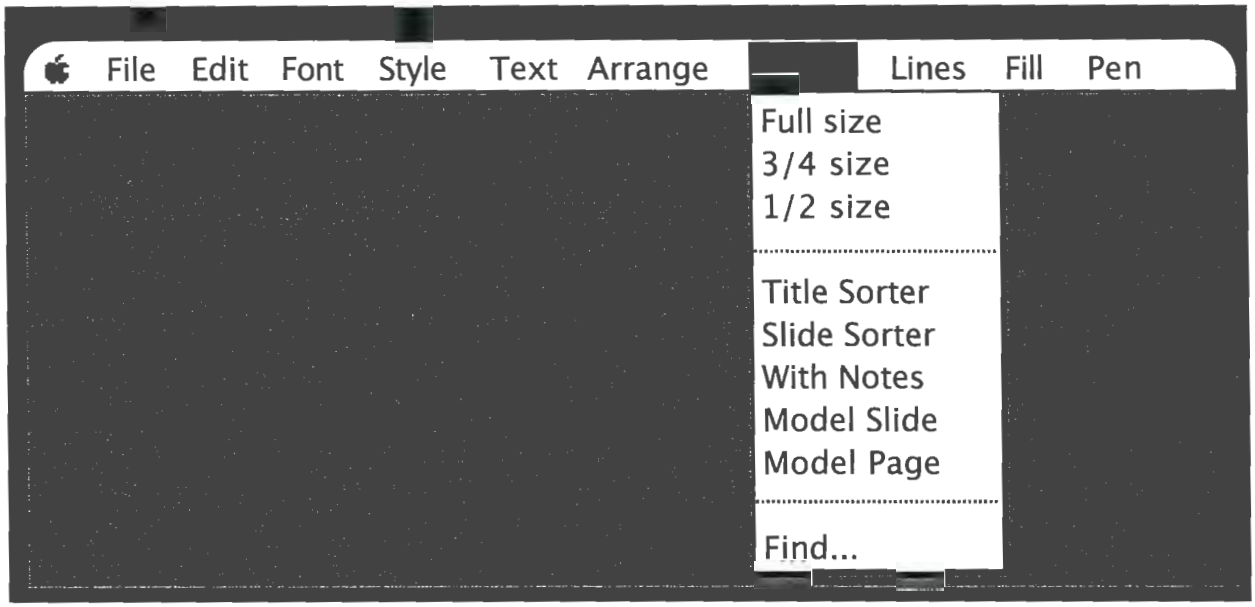
The gravitation effect is more powerful than the grid. If an object is not grid aligned, moving it normally aligns its center on-grid. If a T-square line attracts the edge of such an object, the edge will will be aligned even if its center goes off-grid.

Distribute is for evenly spacing a group of objects, either horizontally or vertically. To see how the commands work, we'll look at the horizontal version; the vertical is the same except for orientation. First select the group of objects, then give the command.

The leftmost and rightmost objects do not move. The other objects have their horizontal position adjusted so that the amount of space between each pair is the same. Their ordering from left to right is unchanged and their vertical positions are unchanged. You should note that the leftmost and rightmost objects may be the same object, say when one of the objects is a box enclosing the others. In that case the objects are distributed across the box. If only one object and its containing box are selected, this has the effect of centering it in the box. A useful special case comes up if the group contains only one object: the object is simply centered on the page.

Custom tool (canonize) is the command that installs a custom drawing tool, particularly useful for text boxes. You can draw a text box, display its ruler, and adjust its properties as you like. Then you use Canonize to turn that box into a drawing tool. A dialog box gives you the opportunity to identify it with a letter such as O for outline. The dialog box also allows you to discard tools you no longer want. Any text box, regardless of the tool that was used to create it, can be altered to suit the circumstances. Custom tools are provided solely as a convenience to creation.

Any group of objects can be canonized. Select the objects and then give the Canonize command to install the new tool. When you double click on a custom tool, the object(s) are immediately drawn at the same location and in the same size that they had when canonized. This makes it easy to repeat a common feature on many slides.



View handles scaling of the image to the window, and is also the key to several unique features. The first two commands reduce the image of a slide to fit within the current window dimensions, either horizontally or vertically. (Depending on the shape of the image and the shape of the window, one of the two commands will show the entire slide.) Return to full size with the Normal Size command.

Title Sorter changes the window to a scrolling list of all image titles. Double click a title to display that slide. Single click and drag to select sequences of images that can be cut or copied. Pasting images inserts them in front of the selected images. This mode allows the same operations as the slide sorter view discussed below, but works on titles instead of miniature images.

The **Slide Sorter** command brings up a multi-slide view of the document, allowing slides to be viewed together and rearranged. About 12 to 16 miniature slides should be viewable at one time. They are arranged in the window in rows, the number in each row depending on the width of the window. A vertical scroll bar permits scrolling through the entire presentation. Individual slides are selected by pointing and clicking. Selecting the slide pops up its title to help identify the miniature image. A continuous groups of slides is selected by clicking and dragging. Clicking between slides selects an insertion point between slides. The slides can be cut, copied, and pasted much as if they were words in a text document. Only slides can be pasted in this view, however. No other type of data can be used, even if it is from *Presenter*, but slides can be freely copied or moved between presentations.

The slide sorter is also a handy means of jumping around from slide to slide: double-clicking a slide returns to normal view at the clicked slide.

The **With Notes** command changes the view to include the half-size slide and accompanying notes. The notes consist of a single text box occupying the remainder of the page. The text box can be altered in any way, but cannot be deleted. No other objects can be created. The slide image, though visible, cannot be edited. When notes are being shown, the Notes command is checked on the menu. You can go to Titles or Slide Sorter views and back without effecting Notes Mode. To turn off notes, give the command again.

Model Slide brings up the model slide for this document that contains decorative frames, page numbers, and a title box. Features of the model slide are shown, but are not editable, on the actual slides. They are drawn on the model in the same way they would be on an ordinary slide.

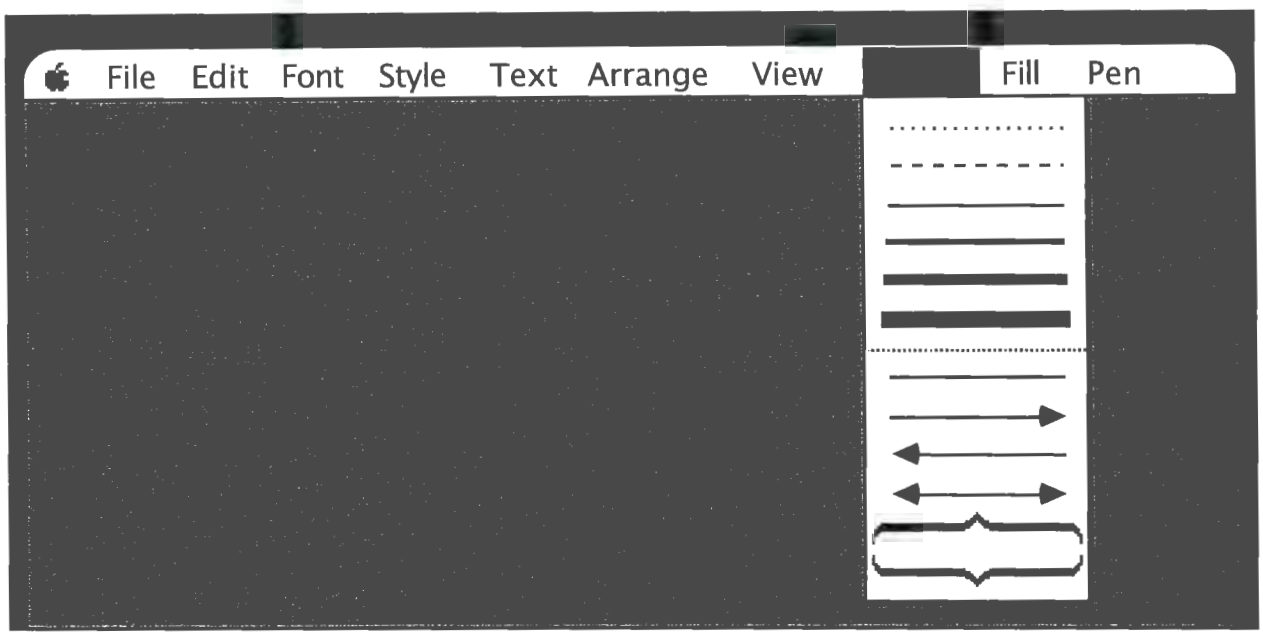
The model slide contains an indestructable text box containing the word "Title". You can change the characteristics of the box to suit your taste—position, typeface, etc. An empty copy of this box will appear on every new slide you create. The text you type into it, if any, will be the title for that slide.

The slide number is simply a box containing the text "\#". When a slide is printed, the two characters are replaced by the slide number. The font, etc. is determined by the font of the "\#".

When viewing the model slide, the Model Slide command shows a checkmark. You can return to view the image you had before selecting Model Slide by selecting it again. You can also return by pressing either the forward or backward buttons in the window. Finally, you can go to a specific slide by entering it s number in the window box, or using the Titles or Slide Sorter views.

Model Page serves a purpose analogous to model slide, but allows the printed page to be decorated with page number, running heads, etc. It applies only on the printed pages and its image is not superimposed on the on-screen slides. Its purpose is not to apply to the slide images printed full size but to the condensed printed formats where added page info may be useful.

The **Find** command searches all the slides in the presentation for text. It is similar to the search commands in word processors, but there is no "replace" option. The dialog box does, however, offer a choice of whether to search notes or not. The dialog disappears from the screen if the search is successful.



The **Lines**, **Fill**, and **Pen** menus have the same purpose as those in MacDraw, but there are a number of differences. The lines menu does not offer an invisible line, but it does offer two types of dotted lines. [Note: lines styles available will depend partly upon features offered in Microsoft Windows.] In addition to the arrow types, a line can also be turned into a brace, a very handy type of line in text slides.

The **Fill** menu chooses the fill for boxes, but it is primarily for choosing colors rather than patterns. On monochrome screens, the colors will appear as patterns. It additionally offers a small choice of cross-hatching patterns and none (transparent). The default is none.

Pen determines the color of box edges, lines, and text. The pen menu can be applied to a text selection or graphics selection. If color is not in use, only solid black and solid white may be available, but this also depends on Microsoft Windows' graphics.

Austin on Tables

This note is to record (not to say promote) my opinions and suggestions on the construction of tables with our new application, *Presenter*. My favorite proposal, as you already know, is that we ignore the problem. This is always the cheapest and simplest solution to any problem, and its chances for success are often underestimated. And it has that wondrous property of not introducing any *new* problems!

In building tables with MacDraw, the only irritating difficulty I have found is the proper location of text within the cells of the table. This is automatically taken care of in *Presenter*, so I would be satisfied to construct my own tables with the primitives available. As an example, I used MacDraw to create a table, shown below, from the Intel slides. It was fairly obvious how to proceed, and it took only a few minutes. The hard part was getting the text centered in the boxes.

MCS®-96 FAMILY OF PRODUCTS

Options		68 pin	48 pin
Digital I/O	ROMless	8096	8094
	ROM	8396	8394
	EPROM	8796	8794
Analog & Digital I/O	ROMless	8097	8095
	ROM	8397	8395
	EPROM	8797	8795

Well, so much for that. Most users are not so ascetic as I in these matters, and I know the majority here think we need more specialized tools to build tables the "right" way. What could such tools accomplish? Here's my list:

- It should be easy to insert, delete, or move columns and rows. With only the primitive tools, rows are just multiple selections. They can be cut, pasted, and dragged around to accomplish the desired result, but you don't get any special help in selecting, etc.

- It should be easy to change row height or column width, or resize the entire table. This isn't too bad with the primitive tools either, but, again, you don't get any special help in selecting, etc.

- We want variable ruling. Some lines might be heavier than others; some may be absent altogether. With the primitive tools, this is accomplished by simply drawing the ruling where appropriate and leaving it out elsewhere. This is easy, but it does get into some confusion with overlapping objects. It also makes the previous two operations harder because there are extra lines littering the field when you're trying to work on the boxes.

- When entering text in the table cells, you would like to be able to jump from cell to cell without having to use the mouse. The tab key would be an appropriate method. We have no

support for this with the existing specification, but something could probably be kludged up. The main reason I haven't already done so is that the problem is really wider. We also want to be able to cut and paste the text of multiple cells in one operation. This is not only to move things around in the table easily, but also to exchange data with other applications—spreadsheets.

- We would like to occasionally use full text boxes within the table cells. This is no problem working at the primitive level, but could prove more problematic in a higher level table concept: The multi-box cut-and-paste of text above gets complicated because strange things happen if text from a text box does not land in a text box when pasted. Should "text-boxness" be a property of the text that is carried along on the clip board? And what about allowing some space around the text? Text boxes don't provide margins—they aren't needed in other contexts. Is another mechanism needed?

- We might like to have the minimum size of a row or column determined by the maximum size of the text in any one of its boxes. In addition, the size could grow (and shrink?) automatically as text was entered that exceeded the size of the box. This would make it a lot easier to make unplanned changes, although the correct way to grow and shrink can only be guessed, and this sort of side-effect has often proved a double edged sword.

What do I see as potential for the table object as a full-fledged *Presenter* concept? It could make rows and columns easier to select by providing a shorthand device. Deleting and inserting whole rows and columns would be more automatic, i.e. the table would automatically open up to make space or close up to remove space. Ruling would be less messy because automatic adjustments of the table would also adjust the ruling. And last, but I think most significant, all the text in the table could be connected in a way that would allow multi-box cutting and pasting as I discussed above.

As far as disadvantages, I see an added concept that could have complexity comparable to a text box, but be of considerably less utility. This is a tricky judgement to make, but the lesson is to make any table mechanisms as simple as possible. And by simple, I don't mean elegant. I mean bonehead simple—the kind of stuff you don't need a manual for.

I really don't think it's possible to get all the table features we might want and still keep the mechanism proportionate to its role within *Presenter*. What kind of compromise can we live with? With fear in my heart, I tender the following:

Tables

Creating tables. Any box can be converted into a table by means of a menu command called "Table". The command subdivides the original box into cells approximately 1 1/4 inches wide by 3/8 inch high. It then presents a dialog box in which to specify the number of columns and rows. Initial values depend on how many standard cells fit in the original box. The dialog also offers a check box to make all rulings visible. Invisible ruling shows as dotted lines when the table tool is in use (see below.) The Table command can be given again to change the number of rows or columns.

A table acts somewhat like a big box composed of smaller label boxes. The overall table is selectable as an ordinary graphics object and can be moved or resized as usual. The cell boxes contain text in the same way a label does, but they will not automatically expand to accommodate

new typing.

Selecting cells. In a table, text can be selected within a single cell box, or the entire text of any rectangular group of boxes can be selected. You select cell text with the I-beam cursor as you would in a label box. Edit as usual, changing font, etc. Keyboard commands (cursor keys?) can be used to jump the selection to neighboring boxes. If you begin a selection in one cell and drag to a neighboring cell, a block of cells is selected—or rather the text contents of a block of cells is selected. Any rectangular block of cells can selected this way. Their selection is shown by inverting the color of the entire cell. (A shortcut might be nice for selecting entire rows or columns, but I'm not sure.)

Cut and paste. Selected cells can have text commands applied and can be cut and pasted. Cutting (or copying) text concatenates the text from the selected boxes, in row-major order, saving the cell boundaries as tab characters. Pasting such text into a table selection places it into the selected cells the same way. If there are fewer cells selected than there are tab characters in the clipboard, the extra is not pasted in. Pasting into non-tables works in the usual way with tab characters remaining in the text. If the cells are pasted into a spreadsheet, they are all transferred in their original relationship.

Table tool. To construct attractive tables, the ruling lines can be adjusted with a special selection tool. They can be selected either singly or in groups by shift-clicking or marquee. Once selected, they can be colored with the Pen menu or have their width changed with the Lines menu. Chosing a color of none makes them invisible except when the table tool is in use.

Changing a table. A selected line, or group of lines running the same direction, can be dragged within the table to change the dimensions of the rows and columns. Dragging the lines at the edges of the table changes the size of the overall table. A row or column disappears entirely (contents too?) if one of the lines is dragged onto its neighbor or if the Clear command is given. Selecting a line and giving the Duplicate command creates a new row or column, below or to the right of the line. The Distribute command can be used to evenly space a group of lines in the table.

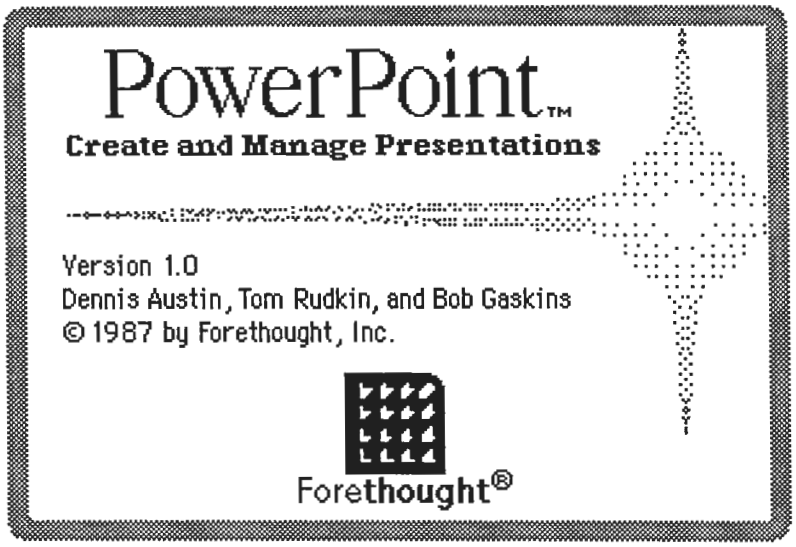
That's it. You were probably expecting more: There is no provision for making odd tables like the example at the beginning of this note. There is no provision for having text boxes in the table. These things are possible by combining tables with other primitives, or just building a do-it-youself table out of primitive boxes, but there is no automation. There is also no automatic grow, shrink, or space redistribution.

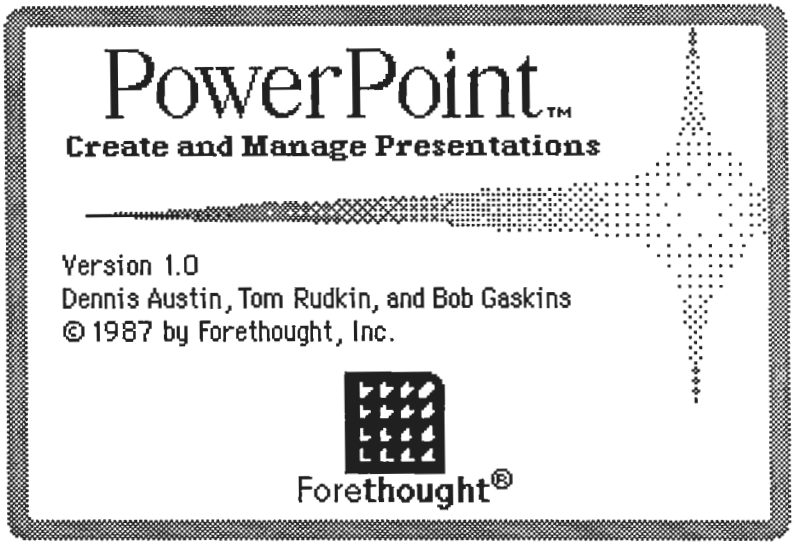
I think what's there is pretty helpful, though. It covers a large range of real tables, and, most important to me, it treats the contents of the table in a useful way that also integrates well with spreadsheets.

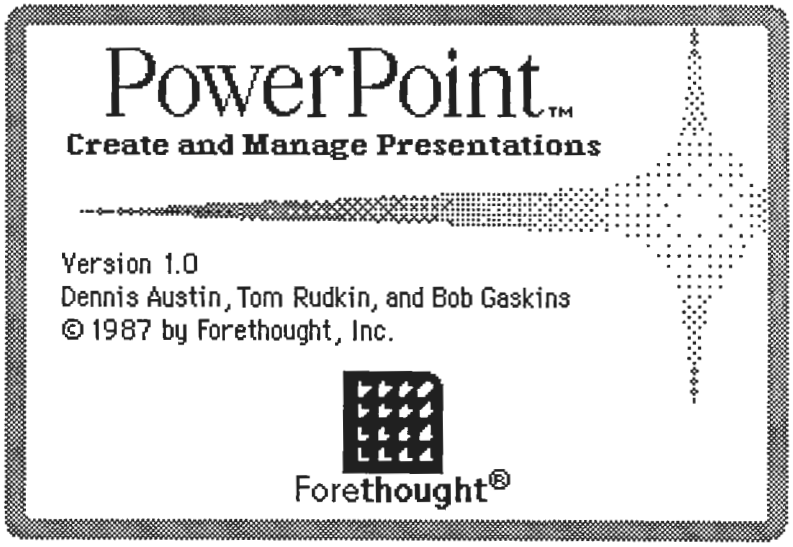
Dennis Austin
July 19, 1985

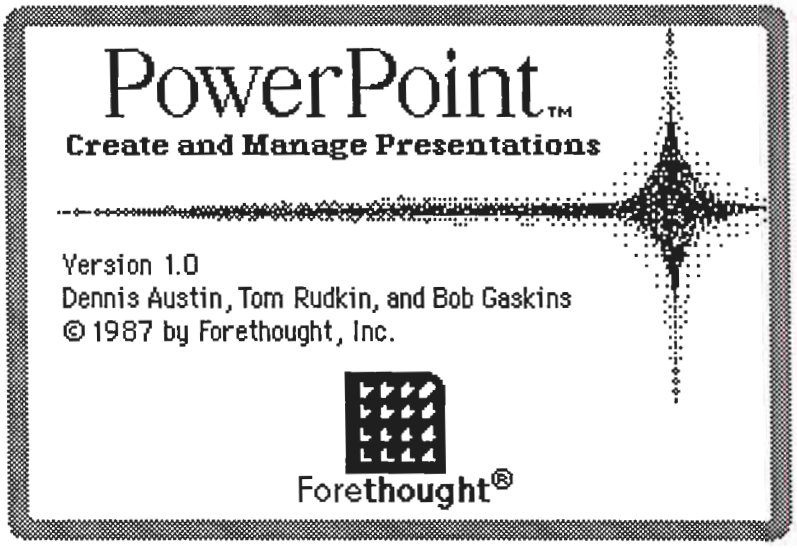
Updated
August 14,1985

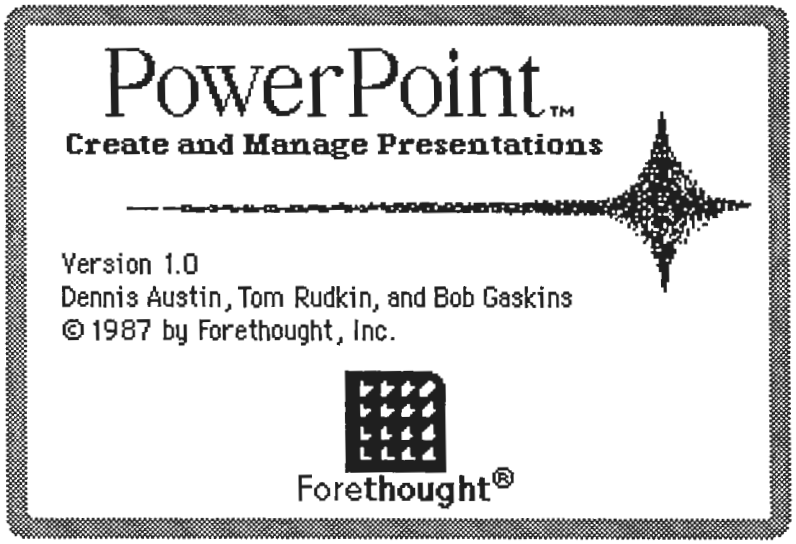
Final

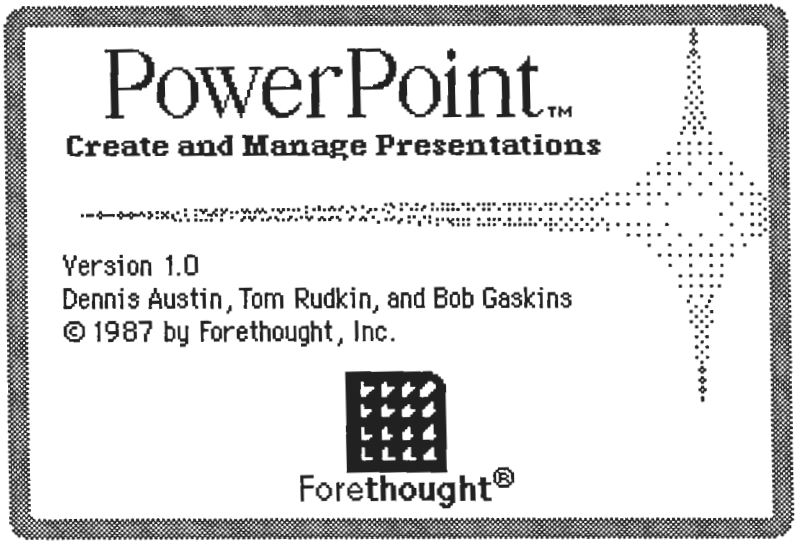

PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®

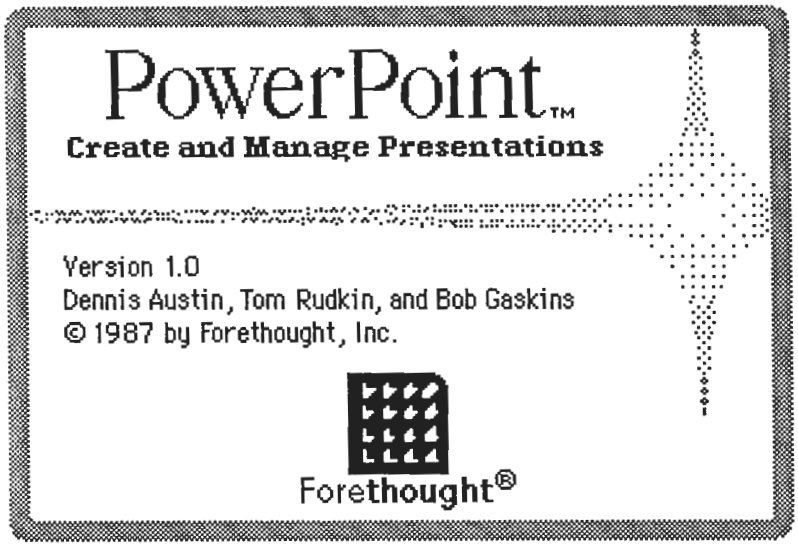

PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®

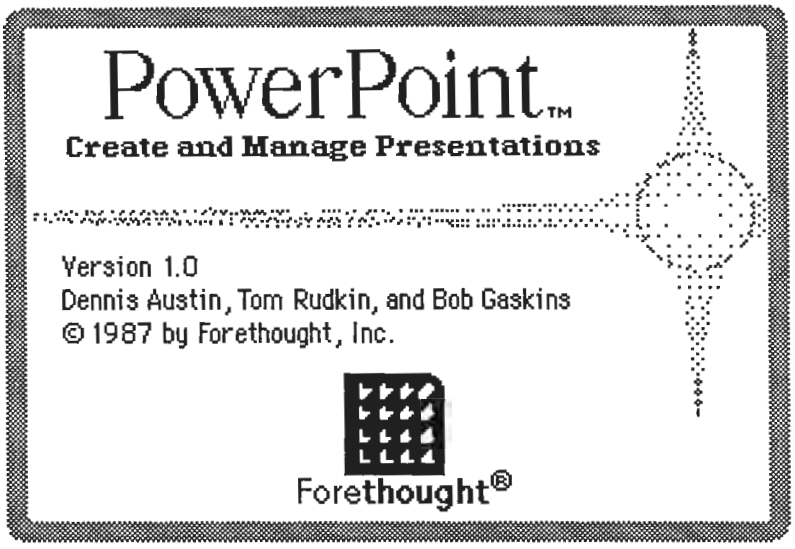

PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®

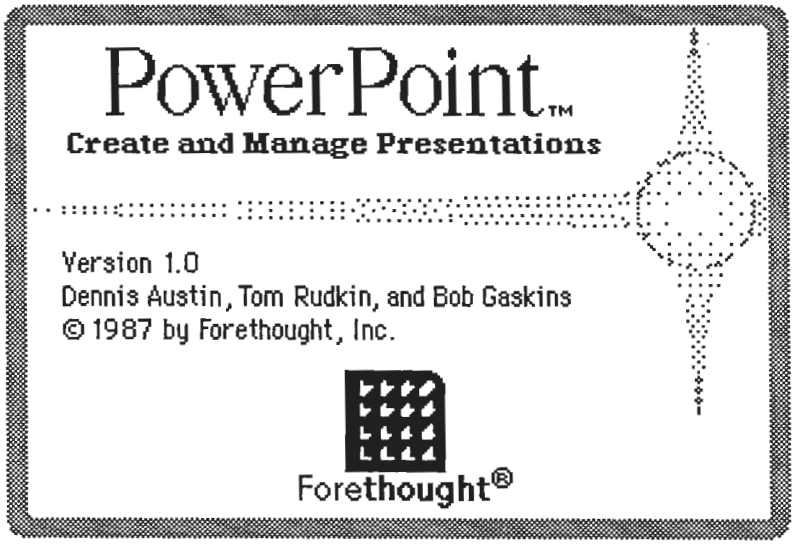

PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®


PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®


PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®


PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®


PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®


PowerPoint™
Create and Manage Presentations
Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.
Forethought®

PowerPoint™

Create and Manage Presentations

Version 1.00
Dennis Austin, Tom Rudkin, and Bob Gaskins
Copyright © 1987 by Forethought, Inc.



Forethought®

PowerPoint™

Version 1.0
© 1987 Forethought, Inc.

by Dennis Austin, Tom Rudkin, and Bob Gaskins



Forethought®

PowerPoint™

Create and Manage Presentations

Version 1.0
Dennis Austin, Tom Rudkin, and Bob Gaskins
© 1987 by Forethought, Inc.



Forethought®

Text Scaling

May 2, 1986
updated July 10, 1986
update November 23, 1986

Presenter changes the scale of a slide by recalculating the position and size of each object at the requested scale. (The actual calculations are performed by routines in the utility module.) Original full-scale sizes are retained so that unchanged objects will not lose any precision of placement. Objects edited in reduced scale, of course, can only be sized and located at reduced precision. This design concentrates scaling overhead at the change-scale command and avoids the overhead on screen drawing.

For text, however, things are not quite so simple. CoreEdit draws text at its actual font size and the scaling must take place in the QuickDraw hooks. The central problem is that CoreEdit must also *measure* the text in order to decide the line breaks and do selection. Since we want the line breaks to occur at the same place in all scales, measurement must be done at full scale and full accuracy.

When text is drawn at reduced scale, however, the Font Manager, performs major league magic to make the text look good on the screen. It does not simply scale the character bitmaps but obtains other fonts, computes fractional character spacings, and performs other tricks unknown. The result is nice looking text with measurements that may not be quite at the right scale. Resolving the differences is difficult because the basic text module, CoreEdit, was not written to handle scaling and it is difficult to make serious changes to its internals.

The May 2 Solution

After several schemes have proven deficient, I have finally settled on one that performs very accurately, but only on Laser Fonts and only with Mac+ ROMs. This combination is required because the new ROM contains a new Font Manager with support for fractionally spaced fonts and the new Laser Fonts contain the fractional spacing information. Since Postscript engines scale fonts exactly, the new Font Manager tries to immitate that exactness on screen—without sacrificing appearance— by ingenious use of letter spacing. The same software that does this handles our scaling perfectly.

CoreEdit works completely at full scale. All drawing, including characters, erasures, insertion points, and selections, are scaled through the QuickDraw hooks. The exact mechanism of the scaling is complicated by the fact that CoreEdit normally draws characters into an off-screen bitmap and then BLTs them onto the screen. This means

that Presenter must have fairly intimate knowledge of CoreEdit's workings.

Every entry into the Etext module (at least all those that may call CoreEdit) begin by setting up a clip region and installing the QuickDraw scaling hooks. Every corresponding exit undoes that setup.

Only one change was necessary within CoreEdit itself. As an optimization, before drawing a line of text CoreEdit checks to see that part of the line is actually within both the visible region and the clipping region. If it isn't, it doesn't bother to draw the text. Because CoreEdit may be working at a different scale from the screen, the check doesn't work properly. I disabled the check. (As of July, only disabled if scaling.)

July Improvements

A great deal of additional selection accuracy, especially for non-fractional fonts and old ROMs, has been obtained by informing CoreEdit when scaling is in effect and having it signal the type of measurement it is doing on text. The signalling is received by the scaling version of StdTxMeas, SclTxMeas, declared in EText.

CoreEdit sometimes measures text for on-screen purposes, such as when determining what character the mouse points to, and sometimes for full-scale purposes, such as when determining line breaks. For full-scale measurements, StdTxMeas does all the work necessary so SclTxMeas simply passes the call on. For on-screen measurements, however, SclTxMeas needs to give full-scale equivalents of a scaled font. It calls StdTxMeas to do a scaled measurement, and then unscales the result itself before returning.

CoreEdit actually has to measure each line twice when scaling is in effect. Once to determine the line break, and again to determine the screen width of the line —which may be wider than the margins! This all takes place in the ReDsp routine.

CoreEdit signals SclTxMeas through a shared global variable, Scaling. It is in the ApplScratch area used by CoreEdit. This covert communication has sometimes been called "banging on the walls". When scaling is in effect, EText always sets the byte to TRUE. This indicates to CoreEdit that lines must be measured twice. Whenever CoreEdit does text measurement for line breaks, it turns the byte FALSE. (It actually sets it FALSE around certain calls to DtrmFit, made from within ReDsp). SclTxMeas checks the byte and performs appropriately.

There is no way to solve the problem that characters at the end of lines get cut off—that is a pure font scaling problem that can only be corrected by using fractionally spaced

fonts. It should not be a serious problem except when using widened styles (bold, italic, outline, or shadow) with the old ROMs.

Vertical waits till November

Eventually we noticed that the total height of multi-line text erred significantly in scaled views. The problem was solved analogously to the horizontal case. Whenever CoreEdit calls GetFontInfo, it checks to see if Scaling is true. If so, it turns it off temporarily and calls GetFontInfo again. This second call reports figures that are linearly scaled to full view.

The ascent and descent of the screen font must be used for editing to work properly, but the leading can be reduced to make the total line height match the the height computed with Scaling false. This works quite well even if the leading becomes negative, although editing can get ugly if its less than about negative one. Negative four is set as a limit.

Presenter's Use of CoreEdit

November 23, 1986

CoreEdit is the basic text editing package from MacWrite, written by Encore Systems. It is available for licensing from Apple for a modest fee. Presenter uses CoreEdit to do all it's text editing, but many changes were necessary before it would behave properly in a graphics-based program. This memo documents those changes.

Notes on Using CoreEdit

CoreEdit comes with excellent documentation that is, unfortunately, not quite complete. I have learned a lot by reading the source and performing experiments. Some of these insights are listed below for future reference.

1. Measure too narrow: If the measure of a paragraph is so narrow that CoreEdit is unable to put even a single character on a line, it will loop adding more and more lines to the lines array until it exhausts heap storage. In Presenter it is easy an quick to make such a skinny box. I have put a patch in CoreEdit to correct this problem.

2. Tab stops to the right of the right margin: CoreEdit seems to ingore these and they cause no problem.

3. Ruler measurement system: Margins, tabs, etc. are measured in absolute terms (i.e. in GrafPort coordinates) and are not given with respect to the edit rectangle as I originally guessed.

4. CatchUp: CoreEdit includes an undocumented routine called CatchUp. The idea is apparently that CEInsertChar will only adjust the line that is actually being typed. That avoids having the updating of subsequent lines degrade typing response during blitz periods. Catchup will update the remainder of the paragraph if necessary. Etext calls catchup in its EIdle routine, guaranteeing that so long as the program is busy processing new characters it will not update other lines.

5. Returned rectangles: Most CoreEdit calls return a rectangle that "exactly fits" the text of the paragraph. What it really returns is the edit rectangle with its bottom adjusted to the size of the paragraph. Even this is not always true: CatchUp, if there is no catching up to do, returns the edit rectangle with no adjustments made. The biggest problem is that there is no indication whether the last line in fact has any characters on it. If the paragraph ends with a return, it is one line taller and that line must be taken

out as far as positioning the next paragraph.

Etext keeps the edit rectangle in the edit record up to date after all calls. That avoids the CatchUp problem. The edit rectangle stored includes the empty last line (if any), but, the positioning of the next paragraph is determined by finding the bottom of the last line in the LOS array. I can only consider this a hack, but its impact is minimal.

6. Return characters: Paragraphs may or may not end in a return character. MacWrite stores a return when it is typed, so every paragraph ends with a return except the last one. In Presenter, Etext throughs away all return characters in text boxes so there is never a return at the end. In label text, however, returns are inserted whereever they are typed.

7. Selections across paragraphs: Etext keeps track of the paragraph numbers where the selection begins and ends. Each paragraph in the range has its selected character range stored in its edit record. Each record contains a lot of other information about selection too, however, and its not entirely clear how to maintain it in a multi-paragraph selection. The scheme I have finally settled on, after considerable experimentation, is to leave all paragraphs in the selection in mouse-down state. That way shift-click extension can be implemented by simply resuming a drag that was already in-prgress. To make this work, however, CESetSelRng had to be modified to leave the edit record in mouse-down, drag-forward, character-selecting mode. This instead of undefined mouse state.

8. Scaling: Text scaling is implemented by substituting scaling procedures in the QuickDraw low-level hooks. the scaling method used, however, does all measuring at the reduced scale and is unfortunately subject to rounding errors. In order to get accurate line breaks, the details of scaling have gotten quite complex and are the subject of a separate document.

9. Dynamic sizing of text objects: We have borderless text objects that automatically resize themselves to fit the text they contain. This is accomplished by creating large edit rectangles for the duration of an edit session and then resizing the object afterwards, but it could not be done with the original CoreEdit because when it works in a large edit rectangle it insists on erasing lots of the rectangle and messing up the screen.

10. Edit Records: The Pascal declaration of the edit record included with the CoreEdit package is erroneous. It is missing the field firstLine (thus throwing off subsequent offsets) and has the fields MxDscnt and MxAscnt in the wrong order.

11. CEChngStyle: The documented interface to CEChngStyle does not tell the

whole story. The Style parameter is actually passed as a word, of course, the low order byte of which contains the style bits to be turned on. The actual assembly routine, however, also looks at the high-order byte and interprets it as the set of styles to be turned off. (Well, actually, a style is only turned off if it appears in both sets—don't ask me why.) This can be used from Pascal by declaring the parameter to be a record type that contains two fields, both of type Style.

12. CEGetRangeInfo: The documentation claims that, if the font is not consistent in the selected range, then -1 is returned as the font number. Similarly, if the size is not consistent, -1 is returned as the size. The font works as documented, but the size returned on an inconsistent selection is 255 instead of -1; CoreEdit keeps size in a byte variable...

Patches to CoreEdit

Many changes have been made to CoreEdit to accommodate Presenter's needs. Every effort has been made to minimize changes because it is possible that an improved version of CoreEdit may become available in the future and we would want to be able to move our changes to any new version. All changes are marked in the CoreEdit source.

1. As already mentioned above, a patch was placed in ReDsp that keeps CoreEdit from going into an infinite loop when it can't fit any characters on a line. Display simply stops if it finds a line on which it could display no characters (including a CR).

2. In FmtStuf, CoreEdit tries to optimize display by checking whether the line to be displayed falls within both the clipping rectangle and visible region of the grafport. This check doesn't work in presenter because the grafport rectangles are scaled and CoreEdit is working with unscaled rectangles. I had to disable the optimization. (But it is only disabled if scaling is in effect; see use of DecTabChr in change #4).

2.b. CoreEdit had a bug in the optimization mentioned above that prevented it from working properly with negative coordinates. I have repaired it.

3. Also in FmtStuf, at the end of PrintLine, CoreEdit likes to erase below and to the right and left of the line being drawn. We want text to draw transparently on the screen. EDraw resets the CoreEdit global UseBuffer before calling CEDraw so all drawing goes directly to the screen instead of to the off-screen buffer. It is drawn with mode ScrOr, the default for text, so all I had to do was eliminate these extra erasures to have the text put up transparently. (Note: drawing during edit is <u>not</u> transparent because the off-screen buffer is used there.)

At the end of ReDsp (the routine), CoreEdit has an EraseRect calls that are intended to clear up any text below the paragraph that has been deleted from the current image. These have been defeated unless UseBuffer is on, i.e. we are actually editing, not just displaying. During EDraw calls, the edit rectangles are only one pixel high, so these erases were not a problem on the screen, but they did show up in drawing to a picture.

4. In order to give the best results in scaling of text (see separate memo) CoreEdit signals the txMeas QDProc whether it is measuring text for purposes of determining line breaks or for purposes of displaying the selection highlightling or click point. The idea is to increase selection accuracy, particularly for the old Font Manager. When measuring for line breaks, no scaling takes place. For selection measurements, however, the measurements are done at scale and then unscaled before returning. Since the measurement is done at real screen scale, highlighting is accurate.

In FmtStuf, the ReDsp routine has been modified to signal that measurements are about to be done for the line break determination. The Grossly Covert channel that is used is a new global Booleane in the CoreEdit globals at ApplScratch. If scaling is in effect Etext sets it TRUE before calling CoreEdit. Normally it's FALSE. When DtrmFit is called for line break determination, the byte is inverted. This allows the scaling routine to perform appropriately. ReDsp must actually call DtrmFit twice for each line —once to get the line break and again to measure the on-screen width (and height) of the line. The second call is skipped if EText has not indicated that scaling is in effect.

4b. As it happens, CoreEdit was using 14 bytes of data at ApplScratch. ApplScratch is only 12 bytes long, so the GhostWindow feature of the window manager could not be used because CoreEdit overwrote its global. To make room for the the new global described in (4), we had to move the head and tail pointers for the backspace buffer. They are now out in the heap with the buffer itself. The buffer heap object has been renamed CoreMore because it contains a number of data objects.

5. A new routine has been added: CEChngAll(font, size, stylePack, sRect). It is necessary to permit font changes from the font menu to be executed in one go rather than redrawing after each step—font, size, and style. Internally, in file Misc.text, the DoChng in-line routine was changed into a real subroutine that is now called by all the format-changing entry points including CEChngAll. The code is a bit sleazy but I was most interested in keeping physical changes to a minimum. Special note: The stylePack in CEChngAll is more straightforward than the one in CEChngStyle. There are simply styles to turn off and styles to turn on, not matching bits to turn off.

The DoChng routine was also discovered to be not setting the returned text rectangle properly in the original code, so that was fixed too.

6. CoreEdit normally does a lot of erasing to keep stray bits off the screen. Since text can appear in a graphics context in Presenter, we have eliminated as much erasing as possible. (This causes a problem when a paragraph grows into space formerly occupied by a succeeding paragraph, but it is compensated by code in EText (in RippleDown).) The changes are in the ReDsp routine (in redsp.text) and PrintLine (in fmtstuf.text).

Two new variables were added at the end of the edit record: Leftest and Ritest. These are the leftmost and rightmost boundaries of the text in the paragraph. They are calculated by ReDsp, which sets Leftest to the minimum value of all the StrtPx values in LOS and Ritest to the maximum of all the EndPx values. They are calculated from scratch only on a complete redisplay. In a partial redisplay Leftest can only be decreased and Ritest can only increase; i.e. the paragraph can only grow wider.

Erasures are handled on a per-line basis in PrintLine. It used to erase from the line edges (StrtPx & EndPx) out to the edges of the edit rectangle. In the new code, it erases only out to Leftest and Ritest. In other words, it only needs to erase out to limits of the width of the paragraph. (Actually it will erase out to RDSLeft and RDSRight if they are further out than Leftest and Ritest. See below.)

New values for Leftest and Ritest are being calculated in ReDsp at the same time the old values are being used in PrintLine. For that reason, two new variables were also added to the locals for ReDsp: RDSLeft and RDSRight. ReDsp calculates the new values in these fields while the old ones are being used in PrintLine. When the redisplay is finished, Leftest and Ritest are updated from RDSLeft and RDSRight.

7. When a selection crosses a line boundary, CoreEdit used to highlight all the way out to the margin. When text is used as expanding labels, there is no handy margin to extend to. Consequently, in InvSelect, the code is changed to extend highlighting out to the edges defined by Leftest and Ritest in the edit record. (see number 6)

8. InvSelect used to confine the insertion point and selection hilighting to the edit rectangle. Because of scaling, text may sometimes lie outside the edit rectangle, so these restrictions have been lifted.

9. To make it easier to work on CoreEdit, procedure names readable by MacsBug were added to all routines callable from Pascal. Most of these work through macros in CE.text, but some are scattered through the code.

10. The various changes that have been made have had miscellaneous side effects: short branches have been been changed to long, Leftest and Ritest are used in some

places where margins had been used, and so on. I have tried to leave traces in the comments. In two cases, actual bugs were found in CoreEdit. (1) Redsp: In calculating the position of the first character for centered text, a logical shift was being used to divide by two. (2) PrintLine: In checking the line against the visRgn and clipRgn, there was a MOVE.L instruction that should have moved only a word. It was clobbering the adjacent field in the RDS variables.

11. The ReDsp routine checks to see if the last line of a paragraph is empty, and, if so, erases it along with any other lines below that have been deleted since the last call to ReDsp. It had a BGE instruction instead of a BGT, however, so the code would never work for a paragraph that ended in a CR.

12. CESetSelRng was changed to leave the edit record as if the selection had been made by the mouse, dragging forward in character-selecting mode. This allows Etext to assume the current selection is already to do shift-extension simply by resuming a drag that has already begun.

13. CoreEdit had a number of memory management problems. It called _DrawText without locking the text object. In rare circumstances this could cause garbage to be drawn when the text moved during the trap call. Now the text object is locked for the entire duration of ReDsp.

CoreEdit also performed lock and unlock by directly modifying bits in the master pointer. Since this is a no-no for future machines, these were changed to HLock and HUnlock.

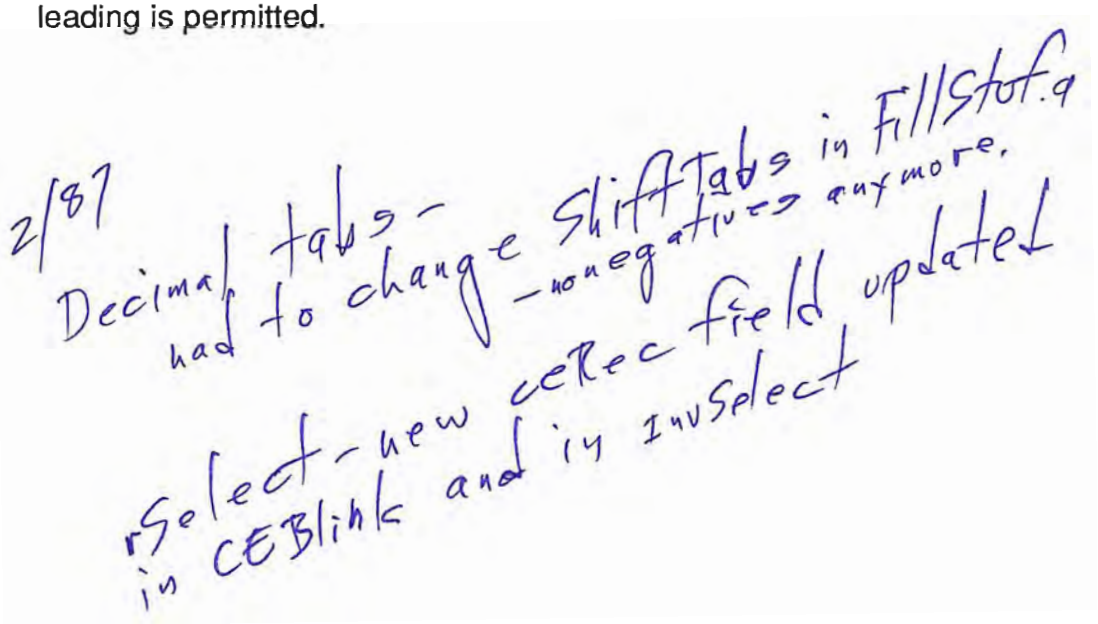
Finally, there is a routine that computes the available memory by looping the the heap on its own examining the data structures, locked and purgeable bits, etc. This also won't work in the future and so was replaced by a call to PurgeSpace. This routine is only available in the new ROM, so the old code is left in place for old ROMs.

14. The Ruler record used by CoreEdit contains a field specifying line spacing. As originally written, it was an integer specifying the number of half-lines of *additional* spacing. 0 was single spacing, 1 was one-and-a-half spacing, 2 was double spacing, etc. To meet Presenter's requirements, CoreEdit was changed to expect this field to be spacing specified in tenths of a line. 10 is single spacing, 15 is one-and-a-half spacing, and 20 is double spacing. Numbers smaller than 10 are permitted, but do not always look very good during edits.

15. CoreEdit adds a constant to the end of each line to make sure it displays italics properly. This kludge works okay for small fonts, but for big fonts the constant would be huge. In our version, an amount equal to half of the maximum line ascent is used. A

small constant is also allowed on the left for a few fonts that display slightly to the left of their character position. These allowances are also included in the values of "leftest" and "ritest" so EText knows how many pixels CoreEdit is using.

16. The scaling scheme (see separate memo) was also applied to veritcal scaling. CoreEdit asks for FontInfo twice; once with the Scaling Boolean on and once with it off. The leading value of the on-screen (scaling=TRUE) case is reduced sufficiently to make the total line height match the full-scale line height (scaling=FALSE). Negative leading is permitted.



Text Scaling

May 2, 1986

Presenter changes the scale of a slide by recalculating the position and size of each object at the requested scale. (The actual calculations are performed by routines in the utility module.) Original full-scale sizes are retained so that unchanged objects will not lose any precision of placement. Objects edited in reduced scale, of course, can only be sized and located at reduced precision. This design concentrates scaling overhead at the change-scale command and avoids the overhead on screen drawing.

For text, however, things are not quite so simple. CoreEdit draws text at its actual font size and the scaling must take place in the QuickDraw hooks. The central problem is that CoreEdit must also *measure* the text in order to decide the line breaks and do selection. Since we want the line breaks to occur at the same place in all scales, measurement must be done at full scale and full accuracy.

When text is drawn at reduced scale, however, the Font Manager, performs major league magic to make the text look good on the screen. It does not simply scale the character bitmaps but obtains other fonts, computes fractional character spacings, and performs other tricks unknown. The result is nice looking text with measurements that may not be quite at the right scale. Resolving the differences is difficult because the basic text module, CoreEdit, was not written to handle scaling and I am unwilling to make any serious changes to its internals.

The Current Solution

After several schemes have proven deficient, I have finally settled on one that performs very accurately, but only on Laser Fonts and only with Mac+ ROMs. This combination is required because the new ROM contains a new Font Manager with support for fractionally spaced fonts and the new Laser Fonts contain the fractional spacing information. Since Postscript engines scale fonts exactly, the new Font Manager tries to immitate that exactness on screen—without sacrificing appearance— by ingenious use of letter spacing. The same software that does this handles our scaling perfectly.

CoreEdit works completely at full scale. All drawing, including characters, erasures, insertion points, and selections, are scaled through the QuickDraw hooks. The exact mechanism of the scaling is complicated by the fact that CoreEdit normally draws characters into an off-screen bitmap and then BLTs them onto the screen. This means that Presenter must have fairly intimate knowledge of CoreEdit's workings.

Every entry into the Etext module (at least all those that may call CoreEdit) begin by setting up a clip region and installing the QuickDraw scaling hooks. Every corresponding exit undoes that setup.

Only one change was necessary within CoreEdit itself. As an optimization, before drawing a line of text CoreEdit checks to see that part of the line is actually within both the visible region and the clipping region. If it isn't, it doesn't bother to draw the text. Because CoreEdit may be working at a different scale from the screen, the check doesn't work properly. I disabled the check.

Future Improvements

It may be possible to improve measurement accuracy on non-fractional fonts by arranging some cooperation between the scaling QuickDraw hooks and CoreEdit. If the scaling routines knew when CoreEdit was measuring for the purpose of selection, as opposed to the purpose of line break determination, it do the measurement at the reduced scale and then magnify it back up for CoreEdit. That would make the selection much more accurate. There is no way to solve the problem that characters at the end of lines get cut off—that is a pure font scaling problem that can only be corrected by using fractionally spaced fonts.

Ginger: Thanks very much for sending me the Font Manager draft for the new ROMs. It is an amazing piece of work. I have FractEnable set now and it works better, but I still have some problems that I can't figure out.

I'm using an 18 pt font and scaling it (with QD procs) to 2/3. In other words I want a 12 pt font to be drawn on screen, but I want the measurements to match a scaled 18 pt font. With Helvetica, this works pretty well. You can see the effects of the fractional spacing doing its job. Selection is often off by a pixel or two, but with the fractional spacing this seems unavoidable.

But with Geneva, the measurements of the font come out much different than the actual drawn spacing, resulting in selection being off by 10-20 pixels over the length of a line. I realize that Geneva does not have the fractional spacing tables in the FOND resource, but that doesn't explain to me why the drawing is so far off from the scaled measurements. Incidentally, if I switch to 1/2 scale I get nicely matched text and measurements in either font.

Although the fonts being drawn looked good, I thought it might help if I tried the new ScaleDisable feature. Unfortunately, I found that if I call SetFScaleDisable (or even set the byte with the debugger) I get an ugly scaled font on the screen. The setting almost seems to be the reverse of the spec. To confuse matters more, when I switch to 1/2 scaling I get a nice looking 9 pt font no matter how the global is set!

Can you shed any light on what's going on?

-- Dennis Austin

Text in Presenter

April 2, 1986
updated July 30, 1986
updated Oct. 24, 1986
updated November 17, 1986

Each box-type object displayed on a slide can contain text. The object record contains the number of paragraphs, a handle for the text, and a ruler number to indicate the formatting. At the lowest level in Presenter, all text display and editing is handled by the CoreEdit module licensed from Apple. So as not to encumber Slide with the complexities of CoreEdit, the actual supervising of CoreEdit is handled by an in-between module, Etext.

Etext interface

In the realm of data, Etext exports only one explicit interface, the paragraph record (ParasRec) and paragraph record array (Paras). Slide also knows the structure of the paragraph itself, but only implicitly.

If a box is a simple label, the text handle stored in the object record is the handle of the paragraph. If the object is a text box, then the handle is for a paragraph record array. Each entry in the array, that is each paragraph record, contains the handle of the paragraph involved plus a level number for that paragraph. the level number is the outline indention level on the ruler.

The internal structure of a paragraph, or even paragraph record, is largely immaterial to Slide, but it does need to know the format in order to pack and upack slides from the presentation (in SOpenSlide and SCloseSlide). In presentation form, the text of a slide is stored in a single heap object. All paragraphs are concatenated into one long string, each paragraph preceded by a word giving its level. The paragraphs are concatenated in the order of the object they lie within, so no links are needed. Within the string, on paragraph is laid out:

level | text bytes | the text | format bytes | the formats

The level, number of bytes of text, and number of bytes in the formats, are each one word. The text and the formats are as long as indicated by their respective lengths.

This format must be known not only for opening and closing slides, but also for features such as *Find* and *Check Spelling*.

From Slide, Etext is called in two ways. The most straightforward is EDraw, which draws the text content of a box. It takes the text handle, the number of paragraphs, the ruler number, the text rectangle as parameters and draws the text in the current Grafport. It draws transparently over any other objects.

The other use of Etext is for actual editing of the text. There is a procedure OpenEdit, taking parameters similar to EDraw, that establishes a text as in-edit. When Slide opens an edit session, it establishes the selection as SELTEXT. Many other calls can then be made to actually edit the text. When a selection is made outside the text (or even when the slide is deactivated) the edit session is ended (CloseEdit) and an updated text handle is returned to Slide.

Etext is also called by Cmd to effect various editing commands. Cmd changes fonts, styles, alignment, and indent level. By use of the procedure EMenuInfo it determines the font, style, and alignment of the current selection for displaying checkmarks on the menus.

The Edit Rectangle

CoreEdit bases its editing on an edit rectangle, but it is rather free about erasing slightly outside the rectangle. It also shows insertion points outside the rectangle. For that reason, Etext originally clipped all displays to the edit rectangle. This seemed unsatisfactory, but the only solution was within CoreEdit itself. Eventually, CoreEdit was modified to eliminate most of the uneeded erasures. Now, the text box rectangle used is a slightly inset version of the object's rectangle, calculated using the constants HMARG and VMARG. For label text, the edit rectangle is enlarged to allow the text to expand arbitrarily during editing.

Global State

Etext keeps a great deal of global state about an open edit session. Much of it is in global variables, but CoreEdit keeps lots more in the heap. Because of the global variables, Etext can only handle one edit at a time. Two different windows cannot have open edits simultaneously. Because of this, the edit session is closed when a slide is deactivated. Before closing, Slide calls EGetSel to remember the text selection parameters (paragraph numbers and offsets of beginning and end) and stuffs them in the slide context record. When the slide is reactivated, a new edit session is opened

and the selection is restored with ESetSel.

Another consideration is that text objects other than the one being edited may need to be drawn because of scrolling, etc. For that reason EDraw does not rely on any of the global variables used by the rest of Etext. It is the only routine that does not implicitly apply to the current edit session.

Cut & Paste, Undo

Editing of the text is largely handled within CoreEdit. Etext is responsible primarily for the maintaining the paragraph array for text boxes and for splitting and joining paragraphs appropriately—whatever that means. Etext takes the view that when an entire paragraph is selected, then its beginning and end are included and if copied will be pasted as an entire paragraph. Otherwise any selection within a paragraph is taken to be just the text.

A more interesting aspect of editing is Undo. From Etext's point of view, any undo operation can be broken down into two parts: deleting some text (possibly none), and inserting some text in its place (possibly none). Etext keeps an undo buffer that contains cleared text and the selection that it should replace in case of an undo. During an undo, the text selection simply exchanges places with the text in the undo buffer.

To perform a copy, the copied text is only stored in memory once and handles appear both in the scrap and in the undo buffer. If an undo is actually performed, of course, the text is actually duplicated and the scrap is separated.

Rulers

The term ruler is used two ways within the Etext. The first usage is to refer the text box rulers used by presenter. These are often termed Prulers since they are presenter rulers. CoreEdit has its own idea of what a ruler is, though, and so there is occasional confusion. When an edit session is opened (or a text box drawn) Etext obtains the Pruler for the box from the Ruler module. It then converts the Pruler into five CoreEdit rulers—one for each possible indentation level. The handles for these are kept in an array for use as needed. At the conclusion of an edit session the rulers are disposed of.

The Ruler unit defines the data type PRuler and performs two general functions. First, it manages the presentation ruler pool. Rulers are stored in a dynamic array in the heap.

Only the ruler numbers are kept in object records. The Ruler unit can retrieve the actual ruler given its number, change an existing ruler, add a new ruler, etc. The ruler pool being used is passed to Ruler in a call on RActivate.

Ruler numbers are actually encode rather than being a direct index into the ruler array. The number divided by four is the actual index and the remainder is the alignment value (left, centered, right, justified). This avoids duplication of rulers that differ only in alignment. The PRuler record contains an alignment field, however, and this is where alignment is stored outside of the ruler table. Within the table, the alignment field is used as a reference counter.

The second function of Ruler is to handle the on-screen display and manipulation of rulers. A routine is provided to draw a ruler. Slide calls this routine and changes the selection to SELRULER. Mouse clicks are then passed to Ruler for processing. RClick tracks mouse movements and updates the indention markers. It returns a status to Slide to tell whether the click changed the ruler, in which case the text needs to be redrawn, was outside the ruler, or had no effect .

When a ruler is showing for a text box, Slide highlights the box with a dotted border just like the one used for borderless text selections. The border disappears when a slide is deactivated, but the ruler is not erased. When the slide is reactivated, RActivate is used to reestablish the context and slide rehighlights the box.

Fore**thought**, Inc.

from the Mac of ***Dennis Austin*** February 10, 1987

To: The Presenter User's Group

Subject: "Size to Text"

This is to explain the way the Size to Text menu item works. It ain't simple.

Like the other items in its menu group, Size to Text applies only to box objects, not lines. It is a check mark attribute that can be turned on or off on individual objects or on groups in the usual way.

If there is no selection, the menu shows the default setting using a diamond if it is turned on. Again like other items in its menu group, the default setting applies only to new diagram objects, not to labels or text boxes. The default for labels and text boxes is always on.

For a label that has the size-to-text attribute, the size cannot be changed with the resize handles. They have no affect. The size is determined by the text which it contains and the weight of the frame, if any, and its degree of roundness. The size is recomputed whenever any of these change.

For a text box that has size-to-text attribute, the sides and top can be changed by dragging resize handles but the bottom is always adjusted to fit the text (allowing for the width of the frame, if any.) Turning the attribute off leaves the text box to be sized completely manually.

A newly drawn text box or diagram object that does not yet contain any text is not resized to fit the text until some text is first typed into it. For a diagram object, the text will be initially vertically centered, as though it were a normal manually sized box. When the text is unselected, the box will be reshaped to fit.

Gotchas:

When the size of objects is determined by the text within them, the text stays put and the object is resized around it. This means that objects are easily thrown off-grid by any change: text changes, line weight changes, framing on/off.

Objects that have frames, fill, etc. are not resized every time they are drawn because they would always have to be drawn twice to be correct. Consequently, if you open a presentation with different fonts than were available at its creation, framed objects that are sized-to-text may not have accurate frames.

Forethought, Inc.

from the Mac of ***Dennis Austin*** December 19, 1986

To: Tony, Dave, Tom

Subject: Reading More files into Presenter

This is a rough specification for an interface that Bear River could use to read More files into presenter. The translation from More outline to Presenter slides is specified separately.

Bear River would write a Pascal unit called Input. It has one routine, IRead, that constructs a sequence of slides in the active presentation by reading a More File (or any other file type that it can make sense of.)

IRead uses Standard File to select the file to input. It gets the document handle as a parameter. It then uses the existing Slide unit procedures to construct new slides:

```
function SNewSlide (aView: WinViews; {slide/notes/handout}
                    var aSlide: SlideRec; {the slide created}
                    aEnv: DrawEnv; {the environment}
                    aScale: integer; {viewing scale}
                    aSRect: Rect; {slide boundary rectangle}
                    aMasterSlp: Ptr {its (open) master slide}
                    ): Ptr; {returned (opaque) SlPtr}
```

Allocate a new slide and initialize the slide record for it. The slide is created open (unpacked) but it has been flushed into the given slide record. The slide does not become active. Its context pointer is returned. If aMasterSlp is NIL, a default title object is created; otherwise a title object is copied from aMasterSlp.

```
function SOpenSlide(aView: WinViews; { slide/notes/handout }
                    aSlide: SlideRec; {the slide to open}
                    aEnv: DrawEnv; {the environment}
                    aScale: integer; {viewing scale}
                    aSRect: Rect; {slide boundary rectangle}
                    aMasterSlp: Ptr {its (open) master slide}
                    ): Ptr; {returned (opaque) SlPtr}
```

Unpack this "slide" so that it can be activated or drawn. Return a slide context pointer. The slide does not become the currently active one.

```
procedure SFlushChanges(slpParam: Ptr; var slide: SlideRec;
                    var edited: boolean);
```

The slpParam is a slide context pointer. If this slide has been edited since it was opened, or since the most recent call to SFlushChanges, if any, then set 'edited' to TRUE and update the packed format; otherwise set edited FALSE.

```
procedure SCloseSlide(var slpParam: Ptr);
```

The slpParam is a slide context pointer. Discard the upacked representation and slide context, but do not update the packed representation. Set slpParam to NIL.

```
procedure SDisposeSlide(var slide: SlideRec; var env:
                        DrawEnv);
```
Delete the representation of this slide.

Input then adds contents to each new slide with the following proposed new routines:

```
procedure SSetTitle(slp: Ptr; s: Str255);
```
This changes the text of the title to the argument. The title remains centered and retains the font of the master title. The title will be one line unless the argument string contains return characters.

```
procedure SNewTextBox(slp: Ptr; location: Rect;
                      paras: Handle; var cnt: integer);
```
Construct a new text box at the given location. The text box has the attributes of the default text box of the presentation. The box contains the paragraphs in the `paras` argument. It's actual type is `ParasHandle` exported from EText, but that type is not available in the Slide interface. If not all of the paragraphs fit inside the text box rectangle, the extra paragraphs are deleted. The number actually used is returned in `cnt`.

The ParasHandle must be constructed by building CoreEdit paragraphs. A new routine will be added to Etext to facilitate this:

```
function EParHndl(s: Str255; afont, asize: integer;
                  astyle: Style): Handle;
```
The anonymous handle returned is, in fact, a CoreEdit `ParHndl` that is suitable for placement in a `Paras` array.

Page Setup in PowerPoint

Unlike most programs, PowerPoint displays two dialog boxes in response to the Page Setup command. The top dialog box is provided by the printer resource in your system file. It allows you to select a paper size and perhaps a reduction (or enlargement) percentage. The LaserWriter permits any percentage from 25% (one quarter size) to 400% (quadruple size). The ImageWriter allows only 50% (half size) and 100% (full size). Other printers may not support reduction at all.

The top dialog box also allows you to select the orientation of the slide on the page; the slide can be printed either normally or sideways on the paper you select. **Note:** handouts and notes pages are always printed in the *opposite* orientation from that selected for slides. If the slides print sideways the notes will always print normal on the page; if slides print normal on the paper, the notes will come out sideways. With most printers and most paper sizes, normal printing results in tall ("portraiit") slides, and sideways printing results in wide ("landscape") slides. Sometimes, however, this situation will be reversed (e.g. Computer Paper on wide carriage ImageWriters).

The secondary dialog box, unique to PowerPoint, is fully revealed after you have pressed OK on the top dialog. (If you want to review the choices you made in the top dialog box, you can get it back by clicking the Page Setup button here.) This dialog shows you the slide size that has resulted from your choices of paper, orientation, and reduction or enlargement. It then permits you to adjust the dimensions of the slides that will be printed, so long as the slide fits on the chosen paper.

You can have PowerPoint select the slide dimensions for you, subject to shape restrictions, or you can type your own custom size limited only by the maximum printable area of the paper (according to the printer you are using.) If you choose Slide Shape (3:4), PowerPoint will compute the biggest rectangle of 3 to 4 aspect ratio that will fit on the page. It displays that size in inches (or centimeters if you are using a metric system.) It does a similar computation if you click on 35mm Shape (2:3). If you click on the bottom button, the size is changed to the full printable area. You can then type custom (smaller) dimensions if you like.

No matter what size is computed or typed, the slide will always be printed centered on the paper. The rectangles on the right of the dialog shows how the current slide dimensions are fit on the paper. The outermost rectangle represents the page boundary. Within that, a second rectangle delimits the printable area of that page. Within that, a gray rectangle shows the slide area chosen.

Say, for example, you choose US Letter (8.5 x 11) as your paper, 100% for reduction, and sideways for the orientation. On the LaserWriter, US Letter paper has a maximum printable area of 7.666 by 10.111. This is the size that PowerPoint will display if you click on the bottom radio button. If you instead choose Slide Shape (3:4), PowerPoint computes dimensions of 7.5 tall by 10 wide--good round numbers for this size paper. Now type in some custom numbers, say 6 tall by 4 wide. Observe how the display rectangles change to demonstrate your setup.

Now consider the affect of magnification on the slide dimensions. Let's say you chose 50% reduction in the top dialog. When computing slide dimensions, PowerPoint will always give you the biggest size that will fit on the page *after* any reduction or enlargement. If you chose Slide Shape (3:4) when the reduction is 50%, the dimensions will now be 15 tall by 20 wide. That is because a slide of this size will still fit on the paper after it has been reduced 50%.

If you have a custom size chosen, it will not be changed as a result of a change in magnification—so long as it still fits on the paper. Say you had a custom size of 7.5 by 10 (typed by you, not selected by PowerPoint). Changing the reduction to 50% will not affect your slide dimensions because those dimensions will still fit on the page after the reduction. But now let us suppose you

change the magnification to 200%, or double size. The custom size you have entered cannot be maintained because doubling it will no longer fit on the page. PowerPoint handles this by showing you the maximum size that *will* fit, given this enlargement.

Enlargement and reduction are very powerful features of the LaserWriter, but it is easy to get mixed up trying to imagine what happens with different choices. One rule that will help you remember is that PowerPoint will never allow a slide dimension to be bigger than would fit on the paper *after* any reduction or enlargement. A second rule is that the (3:4) and (2:3) options instruct PowerPoint to compute the *biggest* slide that will fit on the page (again *after* reduction or enlargement).

—Dennis
2/25/87

Presenter Notes Pages

To implement notes pages and the master handout page in Presenter, we will let the Slide modules deal with a single notes page and the handout page in much the same manner as they deal with a slide. The "active slide" will in fact be a slide, a notes page, or the handout page (or NIL), and a "slide context" for an "open slide" can be a slide, a notes page, or a handout page. In addition, we will change the way the drawing environment is managed within Presenter.

Drawing Environment

In unit `ShareDefs`, we change the record `DrawEnv` so that it only contains handles—all the data that changes but which must be shared between all slides and notes of a presentation (defaults, whether guides are showing, etc.) are moved into a separate object with its handle in `DrawEnv`. This way, the environment can be shared simply by passing this record of handles around; the data in the objects referenced by the handles is shared without having multiple copies. However, we remove the master slide context pointer, slide scale, and slide rectangle from the environment, since these apply to slides only and not to notes; instead, we keep separate values for these in the document record.

In `ShareDefs`, define the following new types:

```
EnvHandle = ^EnvPtr;
EnvPtr    = ^EnvRec;
EnvRec    = record
   dOpaque:    boolean;        {all of current DrawEnv beginning at  }
   ...                         {  dOpaque; excludes sRect and scale}
   guidePt:    Point;
   end;
```

and modify type `DrawEnv`:

```
DrawEnv   = record
   pics:       Handle;
   rulers:     Handle;
   fonts:      Handle;
   tools:      Handle;
   colors:     Handle;         {???}
   envH        EnvHandle;
   end;
```

In `DocData`, add the following fields to `DocRec`:

```
   slideScale: integer;        {current env.scale}
   notesScale: integer;        {scale in which to view notes handout pages}
```

```
masterSlide: Ptr;          {current env.masterSlp}
masterNotes: Ptr;          {context ptr for master notes}
slideRect:   Rect;         {current env.sRect}
pageRect:    Rect;         {printable paper size}
```

The printable paper size is obtained from the printing unit, and is used in place of `slideRect` for notes pages and the master handout page.

Of course, all parts of Presenter that use the environment currently must be modified to take its new structure into account. Primarily, this means that fields of the environment other than the pictures, rulers, fonts, and tools are now either indirectly referenced through the `envH` handle or, in the cases of scale, master slide context pointer and slide rectangle, obtained from a different source. Also, it is no longer necessary to call `SGetEnv` and to copy the environment from the active slide into the document record before deactivating a slide, because now the single shared copy of each environment object is always up to date; and it's no longer necessary to update the master slide context's environment whenever the active slide context's environment is changed (rescaled, guides shown or hidden, etc.) for the same reason.

Scrap

In `ShareDefs`, add the following field to `ScrapRec`, to make up for the fact that slide rectangle is no longer in the environment:

```
slideRect:   Rect;         {current env.sRect in ScrapRec}
```

The scrap has to be handled differently. In `Cmd`, when the environment of the current document is copied to the scrap, its slide rectangle must also be placed in the scrap (since it's no longer part of the environment).

When the scrap is extracted (when closing or saving the presentation to which the scrap currently "belongs"—see `CheckExtractScrap` in `Cmd`), the object `scrap.env.envH^^` must be copied to a new object at `newEnv.envH^^`, and the new object must be disposed when the scrap is eventually disposed. (Note: if the data in `envH` aren't needed in order to paste or post slides or objects from the scrap, then we could just set `scrap.env.envH` to `NIL` during Cut or Copy and ignore it during extract and dispose.)

Whenever a picture is created of a slide on the scrap (`PPostScrap`, and `PPaste` if pasting into a slide "as picture"), the master slide is always opened, used to draw the slide, and closed (currently, `scrap.env.masterSlp` is used if it exists, and it must be handled specially).

As for the scale, it does not need to be stored on the scrap at all, since the slides or objects on the scrap will either be drawn at full scale (if they are drawn as a picture) or pasted into a new slide or presentation at the scale of the target.

Sorters

In unit `Sorter`, we will entirely eliminate the record `SorterContext` and the pointer and handle

types for it, and instead use the DocRec. (This wasn't originally possible, since DocRec used to be defined in Pres and hence wasn't visible to Sorter; but now it's defined in DocData and can be used by Sorter.) Add the following fields, currently in SorterContext, to DocRec:

```
isInsertPoint:  boolean;
selection:   integer;
sFrame:      Rect;
...
sHGap:       integer;
```

and delete the field sorterH from DocRec.

The variables and arguments in Sorter currently of type SorterHandle (or just Handle in the interface) are changed to be of type DocHandle. All the other fields of SorterContext are already in DocRec, and Sorter will just use them (but not modify them). The routines ZOpenView and ZCloseView no longer allocate and dispose heap storage. ZOpenView takes a DocHandle as an argument rather than returning one as its result, and no longer has arguments slides, view, scale, or environ; it still must set the selection information and find the center of the selected slide. The routine ZSetNumSlides is deleted, since Sorter obtains number of slides from the document record.

In Pres, the sorter is "always open", even when not on view. Therefore, PGrow must always call ZGrow, in order to relayout the slide sorter, but PGrow ignores the *var* arguments set by ZGrow unless the view is a sorter. This also means that ZOpenView no longer has to layout the slide sorter, since it's been kept up to date even when the view was not a sorter.

Notes Pages

In unit SlideList, the record type SListEntry for linking together slides in a list will be defined as follows (only the last field is new):

```
SListEntry = record
    nextSlide:   SListHandle;    {next & prev link SListEntries in}
    prevSlide:   SListHandle;    { doubly-linked list}
    selected:    boolean;        {is slide selected in sorter?}
    title:       StringHandle;   {title of this slide}
    image:       Handle;         {handle to bit image of miniature slide; handle
                                   or master ptr is NIL if image not computed}
    slide:       SlideRec;       {slide record, used by Slide modules}
    notes:       SlideRec;       {notes for the slide, used by Slide modules}
end;
```

When opening a slide *or* notes page, Pres calls SOpenSlide, as now. This happens when changing view, and when changing from one slide or notes page to another (e.g. via the slider). (See **Master Handout Page** below for how handout pages are integrated into this.) However, its arguments are changed somewhat:

```
function SOpenSlide(slideType: WinViews; slide: SlideRec; envNow: DrawEnv;
                    scale: integer; sRect: Rect; masterSlp: Ptr;
                    slidePic: PicHandle): Ptr;
```

- `slideType` is one of the window views `V_SLIDE` or `V_NOTES` to indicate whether a slide or a notes page is being opened. (When opening a master slide or master notes page, use `V_SLIDE` or `V_NOTES`, not `V_SLIDEMASTER` or `V_NOTESMASTER`).

- `Slide` is now the `SlideRec` for either a slide or a notes page.

- `Scale`, `sRect`, and `masterSlp` used to be in the environment, but now they vary depending on whether it's a slide or notes page; `Pres` gets the scale, slide rectangle and master context pointer appropriate for the slide or notes out of the `DocRec`, as described above. `MasterSlp` may be `NIL`, meaning that this *is* the master slide or notes page being opened. See **Drawing Environment** above.

- `SlidePic` is `NIL` when opening a slide, but is a picture of the slide when opening a notes page. `Pres` has to compute the picture out of the corresponding (closed) slide before opening the notes page, much as it already has to do for slide sorter and for posting a picture of a slide to the desk scrap. `SOpenSlide` has to add the picture to the picture pool (found in the environment), and `SCloseSlide` deletes it from the pool, so that **Cut** and **Copy** work correctly. The picture "belongs to" the Slide modules after this operation, so `Pres` can neither dispose it nor use it afterwards. It is permissible for `slidePic` to be `NIL` even when opening a notes page, in which case no slide picture is drawn.

- EnvNow is the drawing environment, as modified in **Drawing Environment** above.

- As now, the function returns a slide context pointer for the opened slide or notes page.

When creating a new slide *or* notes page, `Pres` calls `SNewSlide` as now, but calls it twice, first for the slide and then for the notes page. This happens when the user selects the **New Slide** command, and when creating the first slide and the master slide of a new presentation. Its arguments are slightly changed:

```
function SNewSlide(slideType: WinViews; var slide: SlideRec; envNow: DrawEnv;
                   scale: integer; sRect: Rect; masterSlp: Ptr;
                   slidePic: PicHandle): Ptr;
```

The changes are the same as for `SOpenSlide`.

When deleting a slide permanently, `Pres` calls `SDisposeSlide` as now, except it must be called twice, for both the slide and the notes page. This happens, e.g., when the undo state is disposed after a **Cut**).

When copying a slide, `Pres` calls `SCopySlide` as now, except it must be called for both the slide

and the notes page. This happens when a slide is copied to the scrap or pasted from the scrap and when a new presentation is cloned from an existing one (master slide and notes only).

When extracting a slide from its environment, Pres calls SExtractScrap as now, except it must be called for both the slide and the notes page. This happens whenever the presentation from which the scrap was cut or copied is saved or closed, and when it is pasted into a presentation different from the one from which it was cut or copied.

When a presentation is created (**New...**) or opened (**Open**), both the master slide and master notes page are created or opened. They are kept open as long as the presentation is open, and slide context pointers to them are stored in 2 new fields in a DocRec.

Whenever a slide or notes page (including master) is currently on view, Pres stores its slide context pointer in the docSlp field of the DocRec; at present that field is only used when viewing the current slide. The slideNum (current slide number) and docSlideH (handle to current slide's SListEntry) fields will continue to have their current meaning, except that "current slide" means both slide and notes page.

Whenever the user wishes to view the master slide, Pres activates the master slide context, and deactivates it when the view changes to something else; this is no change from the current scheme.

Whenever the user wishes to view the master notes page, Pres first closes the master notes slide context pointer, creates a (current) picture of the master slide, and reopens the master notes page with this new picture. It then activates the master notes slide context pointer, and deactivates it when the view changes to something else.

Master Handout Page

To implement the master handout page,we simply extend the changes described above for notes pages. In unit DocData, we add the following fields to DocRec:

```
handoutPage: SlideRec;
handoutPic:  PicHandle;
```

HandoutPage holds the same information about the (single) master handout page as the slide and notes fields of an SListEntry do for a slide or notes page. HandoutPic is a picture of the slide frames to be drawn as the "background" of the handout page to show where slides will go when printed with this handout master; they show where both two-per-page and six-per-page slides will go.

When opening or creating the master notes page, Pres calls the new SOpenSlide or SNewSlide described above, with the following differences:

- slideType is V_HANDOUTPAGE to indicate a master handout page is being opened.
- Slide is the SlideRec for the handout page, namely handoutPage from the DocRec.

- `Scale` and `sRect` are `notesScale` and `pageRect`, respectively, from the `DocRec`.
- `MasterSlp` is `NIL`, since this *is* the master handout page being opened.
- `SlidePic` is `handoutPic` from the `DocRec`. `SOpenSlide` does *not* have to add the picture to the picture pool, since the master handout page is never cut or copied. It is permissible for `slidePic` to be `NIL`, in which case no slide frames are drawn.

When a presentation is created, the master handout page is created; however, it is immediately closed, since it only needs to be open when it is the current view and during printing. When it is the current view, `Pres` opens and activates it and stores its slide context pointer in the `docSlp` field of the `DocRec`, as for slides and notes; `Pres` deactivates and closes this context pointer when the view changes to something else.

When a presentation is closed, `Pres` calls `SDisposeSlide` to dispose the master handout page's storage.

When the user clones an existing presentation, `Pres` calls `SCopySlide` and `SExtractScrap` to copy the master handout page and extract the copy's picture and ruler references from the existing presentation.

To obtain the picture of slide borders for the master handout page, `Cmd` calls a routine in the printing unit when the presentation is created or opened, and again whenever the user runs Page Setup. `Cmd` is responsible for placing the new picture handle into the `DocRec`, and for disposing the previous picture handle, if any, first.

Units of Presenter

Dennis Austin
December 18, 1985

This note describes the basic internal structure of Presenter in terms of Pascal units. All units are kept in files by the same name except for the main program, kept it the file *prog.*

Presenter (prog)

Presenter is the program unit for Presenter and contains the main event loop. It serves as the window controller and top level event dispatcher.

As window controller, it manages document windows, the toolbox window, a writeln debugging window, and desk accessories. It handles the Apple menu. It handles the basics of New, Open, Close, and Quit commands in order to implement document windows. It also handles the debug menu commands to show/hide the window and toggle debug mode. It implements dragging and resizing for all windows, but for the document windows it even does scroll bar resizing, drawing, and highlighting.

Window management is a complex job because the toolbox window is always kept frontmost when a document window is active. The implementation of this scheme is described in another document, *The toolbox window in Presenter.*

As event dispatcher, Presenter does the following:

- Desk accessories receive all events that are targetted for them.
- The debug window receives events targetted for it. The debug window is implemented by three procedures within Presenter, but they are in fact only an interface to the Apple unit WritelnWindow which does most of the work.
- Events for the toolbox window are passed to the ToolsWin unit.
- Events for document windows go to the DocWin unit.
- Menu commands that are not handled directly by Presenter (Desk Accessories, New, Open, Close, and Debug) are passed on to the Command unit.

Events are not passed to a central parser in ToolsWin and DocWin, but rather to specific procedures for each type of event. These units are somewhat like TextEdit in this respect; they have procedures for update, activate, click, etc.

For each document window, Presenter remembers a document handle that it gets from DocWin when the window is created. It supplies the handle to DocWin whenever the call does not refer to th current document window. (DocWin remembers the handle for the current window.) In this way, DocWin doesn't need to use or save any information in the window record.

Finally, Presenter relays the result of tools window clicks to the document window. When a click is sent to ToolsWin, Presenter gets a return code that it then passes to DocWin to notify it that the toolbox has been clicked. DocWin can can call ToolsWin directly to either get and set the current slide number or get and set the current tool.

ToolsWin

The tools window is simple. It displays a picture resource for the standard tools. One of the tools is shown inverted to indicate the current choice. At the top is a short, masked scroll bar that is used for changing slides. Below the bar is TextEdit area where the current slide number is displayed.

ToolsWin responds to only two events: update and click. If the click is in the slide number area, the entire number is selected and events are captured until (a) a return is typed (indicating that a new slide number has been entered), or (b) the mouse is clicked outside of the slide number area (indicating that no new number will be entered). Click returns a value that tells whether the click had no effect, changed the slide number, or selected a different tool.

To DocWin, ToolsWin provides the ability to get/set the current slide number, get/set the current tool, and to add or delete a custom tool. Custom tools are not yet implemented.

DocWin

DocWin is the central unit controlling the document window contents. Its basic role is the implementation of scrolling. It handles the scroll bar events and keeps track of the current origin of the slide rectangle with respect to the window. It controls the window contents by invoking the Pres and Slide units to draw the graphics and to process other events for the window. Pres and Slide ignore scrolling completely and function

within the coordinate system and clipping region already set up by DocWin.

The main program always notifies DocWin of the result of a click in the tools window. DocWin is then responsible for changing slides or changing the current drawing tool.

Pres

Pres implements the concept of a *presentation*. A presentation is the data structure built to represent an open Presenter document. It exports the record type DocRec that is the basic repository for information about the presentation. It contains the head of the slide list, the number of slides, the current slide, the current viewing mode, the drawing environment (grid, etc.), and even such information is the scroll bar handles and the file associated with the document.

It exports procedures that create, open, save, and close presentations. These largely entail creation and maintainence the slide list, which is a linked list of records representing the individual slides. It also activates and deactivates presentations and keeps track of the current presentation.

But Pres also has another purpose. As proprietor of presentations, it is also responsible for implementing the sorter views. To do this it exports procedures to handle events for the sorter view and also procedures to implement standard editing of the slide list.

Slide

The Slide unit implements the concept of a slide. It exports the record SlideRec which contains the basic representation of a slide. SlideRec has a handle for an object array and a text string that hold the bulk of the actual slide data. It also exports the type Obj Rec, the representation of a single object on a slide.

Slide can draw a slide in the current port and handles all events intended for a slide. It thus implements drawing, selecting, dragging and resizing of objects. It also implements editing of the object list through a TextEdit-like interface. Before a slide can be displayed, however it must be *open*. Opening a slide unpacks the information stored in the slide record and its associated data structures so that it can be accessed quickly. Opening (or creating) a slide returns a handle (a pointer actually) that can be used in subsequent calls to identify the slide. Most calls, however, implicitly refer to the currently active slide. There are procedures to activate and deactivate slides, but a slide cannot be activated until it is open.

BFile

August 11, 1986

The BFile unit processes files of a special type called B Files. These files consist of a header record plus a number of data blocks of varied size. Macintosh files are limited to 2^{23} bytes, but there is no other limit on the size of each block. The number of blocks may be limited by the implementation. The unit caller is responsible for finding, opening, and closing the underlying data files, and for pre-flighting of writes if needed.

If an I/O error occurs the BFile unit presents a dialog box analyzing the error. Usually a file error must result in cancellation of the operation in progress, but sometimes the dialog can offer a choice of Canceling the operation or Ignoring the error. If an error is ignored, the data block is assumed to be invalid. We may allow a slide to be deleted if it is bad. An exported variable, `BError`, indicates the status of the last call. If true, cancellation has been requested.

While a B File is open, it is identified by an addressing context. A handle (FileHandle) to that context is returned by routines that open it, and context is destroyed (and the handle disposed) by the close routine. Read and write accesses require the FileHandle as a parameter.

Routines

```
function BOpen(refnum: integer; var header: Handle): FileHandle;
```

Read the file and return its header record. Establish the addressing context and return it as the function result. If the file is not a bfile, then the returned FileHandle (and header) is NIL. Each call to BOpen establishes a new addressing context.

```
function BPreRead(refnum: integer; var header: Handle): FileHandle;
```

This operation is semantically equivalent to `BOpen`, but it also reads all blocks into memory as efficiently as possible.

```
function BRead(f: FileHandle; bn: integer): Handle;
```

Read block `bn` of file `f` into the heap and return the block's handle. If the block is

already in memory, just return its handle. Returned handle is marked not purgeable. Returns NIL if block number is NILBLOCK, or is out of range. If the block was read with an ignored I/O error, BError is set but a valid handle is returned.

```
function BNew(refnum, headerSize, blocks: integer): FileHandle;
```

Discard any current contents of the file `refnum`. Establish and return an addressing context. Does a test write to the file to make sure its writable, and sets BError accordingly.

`HeaderSize` is the number of bytes to be reserved for the header record.

`Blocks` is the number of blocks that the new version of the file is expected to contain. The file access will be most efficient if it is accurate (or at least not too small), but zero is a perfectly acceptable guess; in fact, if the correct number is unknown, zero is probably the best guess.

```
function BPreWrite(f: FileHandle; data: Handle): integer;
```

Add a new block of data to the file. Return the new block number. The block is mapped into the file, but is not actually written to disk. If the data handle is NIL, the block number NILBLOCK is returned. Never sets BError.

```
function BWrite(f: FileHandle; data: Handle): integer;
```

Write a block of data to the file and return its block number. If the handle was previously read from or written to this file, its existing block number is unchanged. (NOT IMPLEMENTED)

```
procedure BFlushFile (f: FileHandle; header: Handle);
```

Write all blocks out to the file. This is the most efficient way to write out the entire file, and assures that the file contains no unsused space due to fragmentation. If a block has been purged but is not already on disk it is written as a zero-length block.

```
procedure BWriteHeader(f: FileHandle; header: Handle);
```

Write a header record to the file. The header object must be no larger than the number

of bytes reserved in the FReWrite call or it will be truncated. (Also writes the internal block table and other info to the file.) (NOT IMPLEMENTED)

```
procedure BClose(f: FileHandle; dispose: Boolean);
```

Discard the file adrressing context. If `dispose` is true, all blocks from the file that are currently in memory are disposed.

```
function BSize(f: FileHandle):  longint;
```

Return the size of the file in bytes. This is the sum of all block sizes plus header and block table. It should be the same as the end-of-file marker if the file is on disk.

Usage

BFiles can be used in various ways. At their most elaborate, they can manage a file as backing storage for the heap. They are probably not the best for this purpose, but they are general enough. In Presenter, BFiles are intended to be accessed as follows:

Open command: Pre-read the file to get all blocks in memory. The header record contains the block number of the document record. Walk the document tree in pre-order from there, using BRead to substitute handles for block numbers in the appropriate records.

Save command: Use BNew to create an output file. Walk the document tree in post-order, pre-writing each block and replacing handles with block numbers.

See if the disk contains sufficient free space to flush the file. If not, and if this is not a "Save as", see if deleting the input file would make enough space and if so, do it.

If there is now enough space, flush the file, write the header, and close it. If no errors were reported, and if this was not a "Save as", and if the input was not already deleted to make space, close the input file and delete it.

If the window is not to be closed perform the open command tree walk above to replace the block numbers with handles.

Cut & Paste

September 18, 1986

Presenter's internal clipboard is represented by a record variable in the Cmd unit. Its type is declared in Utility so that it can be used in Pres, Slide, and Etext during cut and paste operations. The principle components of the clipboard are in two fields: a type code and a single long word. Usually, the long word is a handle that leads to an arbitrarily complex structure in the heap. This table lists all the possibilities:

Code	Meaning	Long Word	Where pastable
SCRAPNONE	Disposed	not used	no pastable
SCRAPSLIST	Slide List	SListHandle	Sorter
SCRAPSREC	Slide *	^SlideRec	slide/object
SCRAPOLIST	Object List	OListPtr	slide/object
SCRAPTEXT	Text	CharsHandle	object/text
SCRAPCETEXT	CE Text	ParHndl	object/text
SCRAPPARAS	Paragraphs	ParasHandle	Tbox obj/Tbox sel/other?
SCRAPDESK	desk scrap	not used	slide/object

* Pres never places a slide on the clipboard in this form; it places a slide list. When a slide list (of one slide) is pasted into a *slide*, however, Pres must convert it to this form for passing to the Slide unit.

Desk Scrap

When the type on the internal clipboard is a desk scrap, the clipboard data is from another application or a desk accessory. This is the situation when Presenter first starts up. Once a cut or copy command is given in Presenter, the system clipboard is discarded and the type changes to one of the internal types. The type never returns to desk scrap unless new data is placed on the system clipboard by a desk accessory or another application under Switcher.

When all Presenter windows are deactivated—either to activate a desk accessory or to change to another application with Switcher—Cmd calls Pres (which may, in turn, call Slide) to post the internal clipboard data to the desk scrap. Upon re-activating a Presenter window, Cmd checks to see if the desk scrap has been changed. If not, it leaves the internal clipboard type and data unchanged. If it has changed, it calls Pres (which may, in turn, call Slide) to discard the internal clipboard and then sets the type to desk scrap.

Presenter posts only types TEXT and PICT (and possibly MWRT) to the desk scrap. If the internal type is slide list, Pres will post a picture of the first slide as PICT and its title

as TEXT. Other slides in the list are not posted. [This isn't too exciting, but it's probably satisfactory.] If the internal type is CEtext or paragraphs, Etext will post TEXT and MWRT (and possibly PICT). If the internal type is an object list, Slide will post the objects as a PICT.

When the type is desk scrap and a paste command is given, the scrap manager is called directly by Pres or Slide to get its scrap data. Slide normally prefers PICT type from the desk scrap (accepting TEXT if that's all there is), but demands TEXT if the current selection is a text selection. Slide may also accept MWRT (MacWrite) format if implementation time permits. The sorter view does not accept any data from the desk scrap.

Note: Presenter does not put its internal types on the system clipboard. This means that you cannot cut some slides, quit Presenter, start it up again with a new presentation, and paste them as anything other than a picture. Internal clipboard types only exist during one incarnation of Presenter.

Interfaces

In utility we put the type definitions

```
LongOpaque = LongInt;          {A 4-byte quantity whose type
                                is unknown.}

{Clipboard:     name            what stored          the data}

ScrapType =  (SCRAPNONE        {nothing             not used}
             SCRAPSLIST,       {A slide list        SListHandle}
             SCRAPSREC,        {A slide             ^SlideRec}
             SCRAPOLIST,       {Object list         OListPtr}
             SCRAPTEXT,        {Plain text          CharsHandle}
             SCRAPCETEXT,      {CoreEdit text       ParHndl}
             SCRAPPARAS,       {Presenter paragraphs ParasHandle}
             SCRAPDESK);       {Use desk scrap      not used}

{Internal Clipboard record}
ScrapRec = record
             typ: ScrapType;   {type of data on the clipboard}
             data: LongOpaque; {handle/pointer/whatever leading
                                to the data}
             envh: EnvHandle;  {of source presentation}
             temp: Boolean;    {envh is a creation of Cmd}
             auxi: integer;    {extra info used by slide}
             auxl: LongOpaque; {extra info used by slide}
  end;
```

Pres and Slide provide routines as follows:

`procedure XCut(var scrap: ScrapRec);`
Cut the current selection on to the clipboard.

`procedure XCopy(var scrap: ScrapRec);`
Duplicate the current selection and place it on the clipboard.

`procedure XDispScrap(var scrap: ScrapRec);`
Dispose of the contents of the clip board. Set its type to none. Pres disposes temp environment contents

`procedure XPaste(var scrap: ScrapRec; asPict: Boolean);`
Paste the contents of the clipboard, if possible, according to the selection. The asPict parameter instructs Slide to paste the scrap as a picture in a box.

`procedure XPostScrap(var scrap: ScrapRec);`
Post the data to the desk scrap.

`procedure XExtractScrap(var scrap: ScrapRec; envh: EnvHandle);`
Extract the scrap data from its current environment . ~~Create a new environment.~~ Pres initializes contents
See section below.

slide only?

where X is either S or P. Cmd always calls the P version of the routine; if Pres cannot complete the call because the view is NORMAL or MASTER, or because it does not recognize the scrap type in PostScrap or DispScrap, it passes the call on to Slide.

Screen updates for cut or paste are handled by those routines, possibly by calling InvalRect. For sorter-level cut and paste, Pres may need to call Sorter for this purpose.

Scrap Environment

In the case of objects, and indirectly in the case of slides, the data on the clipboard contains links to per-presentation data. Currently these are the picture pool and the ruler pool. So long as the objects or slides are pasted back into the presentation from which they came, this presents no problem. If they are pasted into a different presentation, however, the accompanying pictures and rulers must be copied to the destination document and the linkages updated.

It is for this reason that the scrap record contains an environment handle. The environment record contains handles to the picture and ruler pools and will also contain any similar aggregations in the future.

Slide transfers objects between environments whenever necessary. SPaste, for example, examines the environment on the clipboard, and if it is not the same as the

current slide's environment, the objects are transferred. The routine SCopySlide, used by Pres in cut/paste operations, will also move the slide's objects to a new environment if necessary. Finally, the routine SExtractScrap isolates the scrap from its environment when that environment (presentation) is being closed, or saved.

If the clipboard contains data from a presentation that is being closed, Cmd creates a temporary environment (one not belonging to any presentation) and calls PExtractScrap to move the scrap into the temporary environment. When the scrap is disposed, Cmd must also dispose the temporary environment.

SExtractScrap is called by PExtractScrap. If the type is object or slide the objects are copied to the new environment passed as a parameter. For slide list scraps, Pres converts each one into a slide and calls SExtractScrap for each.

Presenter Data Structures

January 13, 1986

Document Context Record

A document record is stored in the heap for every open document. The types DocRec, DocPtr, and DocHandle are exported by Pres. Pres also creates and releases the document records as the documents are created, opened, and closed. The document data is shared by Pres and DocWin, and the handles are ultimately kept, by Prog, in the "reference constant" field of the corresponding window record.

Actually the document record is not shared data. Only the current mode (slide or sorter) is shared. Most of the fields in DocRec are used only internally to Pres. Two are used only by its client, DocWin. (DocWin does not need to store much information about a document. It uses only the scroll bar control handles.) In a stricter modularization, the document record could be private to Pres. DocWin could store the document handle in its own data structure, say a document window record, along with the scoll bar handles only it needs. I like this setup, but I did not adopt it because I wanted to avoid any extra units of heap allocation. No sense allocating two records for every document when one would do.

Pres is responsible for maintaing the state of a presentation as a whole, and the key to that state is the document context record. Simple information is stored there directly, and the rest is contained in the drawing environment record, the slide list, and the current slide context—all reached from the document record.

Drawing Environment Record

The environment is kept in a subrecord of the document record. It contains the per-document information that is needed by Slide in order to perform slide editing. It includes includes the settings for various options such as the grid, and also the handles for the data pools used across the entire presentation. There are three such pools: pictures, rulers, and custom tools.

The environment record is shared by Slide, Pres, and Cmd. It is declared in Slide. It is initialized (and saved) by Pres and updated by Slide.

The Slide List

Each slide in the presentation is represented by a slide record. SlideRec is exported from Slide. Each slide record contains handles for two arrays containing a compact representation of the slide data. The slide's objects are stored in one array, and the

slide's text is in another. New slide records are allocated, initialized, and updated by Slide. Pres uses the slide record only to save and retrieve the slide data from disk.

Pres does, however, manage the slide records in a list representing all slides in a presentation. The slide records form subrecords in the private record SListEntry, the remainder of which is the list linkage. The slide list is built by opening or creating a presentation. It is traversed to change slides or to edit the presentation. The list is neither long nor intensively manipulated, so its entries are relocatable and linked with handles.

Slide Context Record

In an open presentation, one slide is displayed in the editing window. To display a slide it must first be unpacked from its compact representation. When Slide opens a slide, it returns a pointer to a slide context record to its caller. This record is analogous to a document context record for the presentation as a whole and exists only while the slide is open. It contains (or points to), in a convenient form, all the information needed to display and edit a slide. It is a private data structure in Slide, although pointers to slide contexts are stored by callers.

When a slide is opened, its representation is expanded from compact text array and object array whose handles are kept in the slide record. The objects are arranged in a linked list so that they may be easily edited. Each object's text is allocated its own heap storage so it can be used by TextEdit or CoreEdit. The object list entries are non-relocatable so pointers can be used for linkage.

The slide context record records the head and tail of the object list, the text edit record in use with this slide, and the current selection. In order to have all data necessary for editing available from a single source, a copy of the drawing environment record is also stored here. It should not be different from the one in the document record.

The Object List

The OListEntry record is also a private data structure of Slide. It contains the linkage pointers and an object record. The object record is a public data structure and occurs both in object arrays (for the compact representation) and in OListEntry's (for the open-slide representation.) Object records are accessed by Cmd to make changes to individual objects: line style, patterns, etc.

There are four types of objects: lines, rectangles, text boxes, and tables. Round cornered rectangles are considered a variant of ordinary rectangles and one can be converted into the other. Roundedness is proportional to the shorter side.

The size and shape of an object is stored as a rectangle. For line objects, the line extends from the topLeft point to the botRight point, but the rectangle may be empty—the two points do not necessarily form a legitimate rectangle.

The object record contains a handle for any text or picture that it contains. When a slide is closed, this field is used to store a long integer instead of a handle. For text, it contains the offset of the beginning of its text in the slide's text array. For pictures, it contains the picture number in the presentations picture pool.

Selection

The selection is stored in essential form in a slide context record. There are six different types of selection:

None -	No selection in the slide
Text -	The selection is in the text of one of the objects
Line -	The selection is a line-type object
Box -	The selection is a non-line object
Multi -	More than one object is selected
Ruler -	A ruler is being edited

For single-object selections, the slide context contains a pointer to the selected object. For multiple selections, a bit in each OListEntry tells whether or not it is part of the selection. For display purposes, an array of rectangles (move bars and resize handles) is also saved which tells Slide how to highlight the selection. This is only maintained for the currently active presentation, however. It is recomputed from the basic selection data whenever a window is activated.

Slide Internals

February 17, 1986

The Object Record

Each object displayed on a slide is represented by an object record. The object record contains all the permanent attributes of the object. These include the following:

`otype: ObjType`

There are four different types of object:

`LINEOBJ`	Any type of line.
`RECTOBJ`	Rectangles or round rectangles.
`TBOXOBJ`	Text boxes.
`TABLEOBJ`	Tables.

`orect: Rect;`

Every object is located by the two points given in its `orect` field. For all but lines (non-lines are referred to as box objects), `orect` is a true rectangle describing its size and location in the slide. A box object is drawn competely within orect.

For lines, the two points topLeft and botRight are the two ends of the line, but there is no requirement that the points form a legitimate rectangle. Pt2Rect must be used to find the rectangle that contains the line, and this will, of course, yield empty rectangles for perfectly horizontal or vertical lines. Because of pen width, the line always extends outside of its containing rectangle.

For objects in the current slide, the `orect` field is scaled to the current view. All on-screen calculations are thus made at current scale.

`iline: Byte;`

This field identifies the type of line used in the object. It is not used for objects designated as "solid". Its primary function is to determine the pen width used for drawing the object. This is returned by the function `LinePix`. The value `NOLINE` is used for solid and invisible objects to keep the edge from being drawn. Line objects cannot use `NOLINE`.

`ipat: Byte;`

The pattern is used either for drawing lines, or if the object is solid, to fill it. The pattern is typically `IPATBLACK`. `IPATNONE` is used for invisible objects. The actual pattern variable is found by indexing the array `fillPats`.

`corner: Byte;`

For rectangles, corner is set to zero. To round a rectangle, its value is increased.

The diameter of the corner arc is found from the function `CornerDiameter`. The field is ignored for lines.

```
iruler:    Byte;
```
This indexes the ruler array to find the ruler specifications for a text box. The ruler array is a per-document structure and can be found from the current drawing environment record.

```
otextLen: Integer;
otextH:   Handle;
```
These are the objects text. Usage to be defined.

The Object List

When a slide is active, its object records are stored in a doubly linked display list called the O List. Although Slide exports the format of the object record, O List entries are private structures in slide. Each list entry contains an object record plus the following transitory data:

```
nextObj:  OListPtr;
prevObj:  OListPtr;
```
These are the links on the object list. They are pointers, not handles. The first and last objects in the list have NIL links in the appropriate direction. There is always at least one object in the list because the title object cannot be deleted. The first and last entries in the list can be found from slide context record.

```
selected: Boolean;
```
This Boolean is used to mark a selected object in a multiple selection. It is not used in single selections.

```
fRect:     Rect;
```
This is an unscaled copy of the object's `orect` field. The unscaled copy is stored here so that errors are not introduced through the scaling process.

The Slide Context Record

A slide context record hold the general state of an open slide. It is created and returned by OpenSlide or NewSlide. It is a private data structure, but anonymous pointers to slide contexts are held by other modules.

```
titleObj: OListPtr;
lastObj:  OListPtr;
```

These are the first and last objects in the O List.

```
sel:        SelType;
selObj:     OListPtr;
```

The slide may have a selection in it indicated by `sel`. If the selection is a single object, `selObj` points to the object.

```
env:        DrawEnv;
```

The drawing environment from the document record is copied here for use by Slide. *This* is the up to date copy. When the slide is closed, it is passed back to the document record.

```
oedit:      Boolean;
tedit:      Boolean;
```

These indicate whether the objects have been edited or the text has been edited. They are for optimization in closing the slide.

```
teh:        TEHandle;
```

This is the TextEdit record used during slide editing.

Global State

Most of the per-slide state is stored in the slide context record, so most of the global state is is either permanent (like certain fill patterns) or very transitory (like undo state).

```
fullRect: Rect;
picRect:  Rect;
```

These two rectangles represent the slide area being drawn. They are per-document data. fullRect represents the printable page area as determined by chosen printer and the Page Setup command. picRect represents the same area as scaled to the current view.

```
curSlp:     SlPtr;
```

This is the slide context pointer for the currently active slide.

```
curSel:     SelType;
sela:       array[SelPart] of Rect;
```

State for the current selection. (See section on selections.)

```
undo...
```

A host of variables representing the current undo state. (See section on undo.)

Selection

A selection has a type, stored in a global variable but also in the slide context record, and additional information depending on the type. For single object selections and text selections, another field in the context record points to the selected object. For multiple object selections, Booleans in the O List entries indicate whether they are included or not. These Booleans should be maintained false except when actually in use.

The current selection can be one of the following types:

`SELNONE`	no objects selection
`SELTEXT`	text is selected; object is `selObj`, details in the TextEdit record
`SELLINE`	a line is selected; object is `selObj`;
`SELBOX`	a box type object is selected; object is is `selObj`;
`SELMULTI`	multiple objects are selected; use `selected` Booleans
`SELRULER`	to be defined

Highlighting of a box object can be stored in the global variable `sela`. It is an array of rectangles representing the move bars and resize handles for an object. `sela` is also used for lines, but only the two handles for top left and bottom right are needed. The procedure HiliteSelection will toggle the highlighting on or off (highlighting is exclusive or-ed onto the screen).

The variable `sela` is also used to determine when a mouse-down is within a handle or a move bar.

The move bars shown around lines are very difficult to draw with a simple algorithm. Consequently a complex algorithm is used that draws dotted lines paralleling the main line. To get nice dots and nice parallels, extensive case analysis is used. Its pretty ugly.

Selecting

A selection is made either by `ScanByPoint` for simple clicks or `ScanByRect` for selection boxes. Selecting by point is simple for box objects, but for lines requires more careful analysis. The function `NearLine` measures how close a point is to a line. To be on a line, a point must be within its containing rectangle and must be on the slope. Calculations are made using the toolbox fixed arithmetic for speed. For lines that are almost perfectly vertical or horizontal, the slope calculations break down. For these cases it is sufficient to be within the (expanded) containing rectangle.

The Command Interface

For commands like line style, pattern, etc. the command unit performs changes directly on object records. The procedure `GetSelObj` makes available pointers to actual object records in the object list and allows `Cmd` to change the objects. Updating the screen will redraw the modified objects.

Object pointers are returned one at a time by passing in an index value starting with zero. Each call automatically increments the index passed in. If there is more than one object selected, higher indexes will return more objects.

Dragging

When there is a mouse-down in the slide, it may be for selection, moving, resizing, drawing, or table adjustment. Table adjustment is not yet designed, but the other functions are fully implemented excepting grid and guide considerations. Selecting is simply a matter of finding which object is closest to the selected coordinates. The other operations all entail some type of *dragging* in which mouse motion is followed by ghost objects on the slide.

Objects that are being dragged are called ghost objects because they do not actually exist on the slide until the mouse button is released. Their image is formed by exclusive-oring onto the screen rather than direct drawing. For moving and resizing, the object is drawn in a special ghosting pattern. The pattern is designed so that any figure can be ghosted exactly on top of itself without disappearing. Black would not do, for example, because any black figure would disappear when a black pattern is exclusive-ored over it. Black is used for drawing, however, since superposition is not an issue in that case.

Different strategies are used for single object dragging and multiple object dragging. Although the multi-object moving strategy would be fast enough for single object moving, it is not adequate for drawing and resizing where the object is being redrawn mostly on top of itself.

Multiple object dragging is drawn by the procedure `MultiGhost`. It offsets all objects in a multiple selection and them draws them in ghost form. For each mouse motion, the ghost objects are redrawn in their current locations in order to erase them. Then `MultiGhost` is called again to draw them at their new location. It can take a long time to draw all the selected objects, so the mouse position is updated again before drawing the second time.

Single object dragging is accomplished with three procedures:

`DragSetup` - This establishes the figure to be dragged: its type, location, line style, and pattern. These data are saved in "own" variables for use by the other two procedures. `DragSetup` also draws the initial ghost image.

`DragFigure` - Taking only a new location rectangle as a parameter, erases the figure at its old location and redraws it at the new location (and with a new shape if the rectangle so indicates.) The own variable for location is updated for the next call. Ideally, DragFigure could be executed only during the vertical blanking interval for the screen. That way the ghost can be drawn with no discernible flickering, although the flickering can actually strengthen the "ghostliness". It is at least important that the drawing not accidentally mis-synchonize with the blanking interval in such a way that it appears not be drawn at all while the mouse is moving at certain speeds. To avoid that problem, one clock tick is always allowed before erasing an image.

`DragComplete` - Erases the ghost object at its last position (and shape).

These three routines are used in three separate drag loops. One is in DragSelection, one in ResizeSelection, and one in DrawNewObject. The loops are straightforward except for three conditions.

Bounds Rectangle

Before beginning any drag operation, a bounds rectangle must be computed in order to confine the drag to the slide image area. While dragging, `PinReal` is used to keep the mouse location within the bounds. `PinReal` serves the same purpose as the toolbox `PinRect` but does not have the off-by-one bug at the rectangle boundary.

For moving and drawing objects, the bounds are straightforward to compute. For resizing, however, the analysis is sufficiently complex to require a subroutine to itself. The case for lines is simple, but for box objects the bounds depend both on which handle is being pulled and on whether the option key is being used (opposite corner fixed or center fixed) for a total of eight different cases.

Shift Key Constraint

The shift key is used to restrict motion in various ways depending on the operation. The procedure Constrain converts a mouse position to its correspondingly constrained point. It uses an original constrained point and a type of constraint to do the mapping.

There are the following types of constraints:

ORTHO	—New point must be in horizontal or vertical line with original
DIAG	—New point must be on a 45° angle with original
ORTHODIAG	—Either ORTHO or DIAG
PROP	—Like DIAG but slope is determined from an own variable
ORTHOPROP	—Either ORTHO or DIAG

In the PROP (proportional) case, the slope own variable must be setup in advance by calling the procedure SetupProportion.

Option Key Centering

Most of the complexity related to the option key is entailed in computing the bounds rectangle. Other than that, it involves only the adjustment of the entire object rectangle rather than just a single corner while dragging.

Undo

Undo is handled by two procedures. SaveForUndo discards any existing undo state and establishes a new state. It must be passed an operation type of

UNDONONE	—next change will not be undoable
UNDOMOD	—the selection is about to be modified
UNDODEL	—the selection will be deleted
UNDOINS	—an insertion will follow

SaveForUndo saves a copy of the currently selected object, or the entire O List if there is a multiple selection. It also translates the operation type into a code that includes not only the operation but the type of selection. The code is also capable of specifying the inverse of the operation so that an undo command itself can be undone.

For graphics, the undo state is stored in four variables:

undoObj	—a pointer to a copy of an entry in the O List
undoSel	—a pointer into the O List
undoTitleObj	—the head of a copy of the O List
undoState	—the type of undo state in effect

Not all of the variables are used for any particular state. These are the possible cases, depending on value of undoState:

UNONE	—no undo state saved
UOBJ	—undoObj is an unmodified copy of undoSelObj
UDELOBJ	—undoObj has been deleted from the O List
UXDELOBJ	—undoObj has been reinserted; copy is at undoSelObj
ULIST	—undoTitleObj is the head of a copy of the O List
UDELLIST	—like ULIST, but objects in selection have been deleted in O List
UXDELLIST	—an undone UDELLIST; undoTitleObj has been swapped with O List
UINS	—before the insertion, undoSelObj was the last object in the O List
UXINS	—insertion has been undone; undoObj heads list of undone insertions

Since all information for an undo is saved within Slide, the Undo procedure itself needs no parameters. It performs the undo, and then changes the undo state so that it can be undone (one of the X states).

Sample Pres

7 items | 247K in disk | 144K available


Data Base Pub Demo


enter Dem

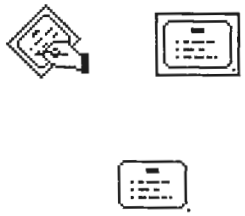

Training Demo

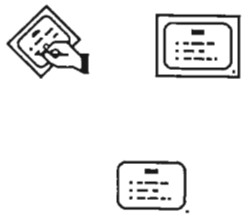

bug


Darrell Pres Final


Bd Report 19 Nov/D


Presenter





Text

April 2, 1986

Each box-type object displayed on a slide can contain text. The object record contains the number of paragraphs, a handle for the text, and a ruler number to indicate the formatting. At the lowest level in Presenter, all text display and editing is handled by the CoreEdit module licensed from Apple. So as not encumber Slide with the complexities of CoreEdit, the actual supervising of CoreEdit is handled by an in-between module, Etext.

Etext interface

In the realm of data, Etext exports only one explicit interface, the paragraph record (ParasRec) and paragraph record array (Paras). Slide also knows the structure of the paragraph itself, but only implicitly.

If a box contains only one paragraph, the text handle stored in the object record is the handle of the paragraph. If there is more than one, it is the handle of a paragraph record array. Each entry in the array, that is each paragraph record, contains the handle of the paragraph involved plus a level number for that paragraph. the level number is the outline indention level on the ruler. Note that in single-paragraph boxes, the level of the paragraph must always be zero.

The internal structure of a paragraph, or even paragraph record, is largely immaterial to Slide, but it does need to know the format in order to pack and upack slides from the presentation (in SOpenSlide and SCloseSlide). In presentation form, the text of a slide is stored in a single heap object. All paragraphs are concatenated into one long string, each paragraph preceded by a word giving its level. The paragraphs are concatenated in the order of the object they lie within, so no links are needed. Within the string, on paragraph is laid out:

level | text bytes | the text | format bytes | the formats

The level, number of bytes of text, and number of bytes in the formats, are each one word. The text and the formats are as long as indicated by their respective lengths. This format must be known not only for opening and closing slides, but also for features such as *Find* and *Check Spelling*.

From Slide, Etext is called in two ways. The most straightforward is EDraw, which draws the text context of a box. It takes the text handle, the number of paragraphs, the ruler number, the text rectangle as parameters and draws the text in the current Grafport. It is supposed to draw transparently over any other objects, but this feature is not yet implemented in CoreEdit.

The other use of Etext is for actual editing of the text. There is a procedure OpenEdit, taking parameters similar to EDraw, that establishes a text as in-edit. When Slide opens an edit session, it establishes the selection as SELTEXT. Many other calls can then be made to actually edit the text. When a selection is made outside the text (or even when the slide is deactivated) the edit session is ended and an updated text handle is returned to Slide.

Etext is also called by Cmd to effect various editing commands. Cmd changes fonts, styles, alignment, and indent level. By use of the procedure EMenuInfo it determines the font, style, and alignment of the current selection for displaying checkmarks on the menus.

The Edit Rectangle

CoreEdit bases its editing on an edit rectangle, but it is rather free about erasing slightly outside the rectangle. It also shows insertion points outside the rectangle. For that reason, Etext currently clips all displays to the edit rectangle. Although this seems unsatisfactory, the only solution lies within CoreEdit itself. The rectangle used is a slightly inset version of the object's rectangle. OpenEdit is only called from one place within Slide, so the standard inset is performed there.

Global State

Etext keeps a great deal of global state about an open edit session. Much of it is in global variables, but CoreEdit keeps lots more in the heap. Because of the global variables, Etext can only handle one edit at a time. Two different windows cannot have open edits simultaneously! Because of this, the edit session is closed when a slide is deactivated. Before closing, Slide calls EGetSel to remember the text selection parameters (paragraph numbers and offsets of beginning and end) and stuffs them in the slide context record. When the slide is reactivated, a new edit session is opened and the selection is restored with ESetSel.

Another consideration is that text objects other than the one being edited may need to be drawn because of scrolling, etc. For that reason EDraw does not rely on any of the

global variables used by the rest of Etext. It is the only routine that does not implicitly apply to the current edit session.

Notes on Using CoreEdit

CoreEdit comes with excellent documentation that is, unfortunately, not quite complete. I have learned a lot by reading the source and performing experiments. Some of these insights are listed below for future reference.

1. Measure too narrow: If the measure of a paragraph is so narrow that CoreEdit is unable to put even a single character on a line, it will loop adding more and more lines to the lines array until it exhausts heap storage. In Presenter it is easy an quick to make such a skinny box. As a quick fix, I have put a patch in CoreEdit to limit a single paragraph to 32 text lines. A better solution could probably be had.

2. Tab stops to the right of the right margin: CoreEdit seems to ingore these and they cause no problem.

3. Ruler measurement system: Margins, tabs, etc. are measured in absolute terms (i.e. in GrafPort coordinates) and are not given with respect to the edit rectangle as I originally guessed.

4. CatchUp: CoreEdit includes an undocumented routine called CatchUp. The idea is apparently that CEInsertChar will only adjust the line that is actually being typed. That avoids having the updating of subsequent lines degrade typing response during blitz periods. Catchup will update the remainder of the paragraph if necessary. Etext calls catchup in its EIdle routine, guaranteeing that so long as the program is busy processing new characters it will not update other lines.

5. Returned rectangles: Most CoreEdit calls return a rectangle that "exactly fits" the text of the paragraph. What it really returns is the edit rectangle with its bottom adjusted to the size of the paragraph. Even this is not always true: CatchUp, if there is no catching up to do, returns the edit rectangle with no adjustments made. The biggest problem is that there is no indication whether the last line in fact has any characters on it. If the paragraph ends with a return, it is one line taller and that line must be taken out as far as positioning the next paragraph.

Etext keeps the edit rectangle in the edit record up to date after all calls. That avoids the CatchUp problem. The edit rectangle stored includes the empty last line (if any), but, the positioning of the next paragraph is determined by finding the bottom of the last line in the LOS array. I can only consider this a hack, but its impact is minimal.

6. Return characters: Paragraphs may or may not end in a return character. When a return is typed, it is entered in the paragraph and then the paragraph is split after the return to create a new paragraph. The last paragraph in a box may not contain a return. If it is selected and pasted into the middle of another paragraph, the paste will not cause a paragraph split.

7. Selections across paragraphs: Etext keeps track of the paragraph numbers where the selection begins and ends. Each paragraph in the range has its selected character range stored in its edit record. Each record contains a lot of other information about selection too, however, and its not entirely clear how to maintain it in a multi-paragraph selection. Etext does not really keep the mouse-down Boolean, the last char the mouse was over, the direction of the drag, word selection mode, etc. up to date. In fact I don't know what needs to be up to date and how to do it, so shift-clicking, for example, doesn't really work right in spite of a great deal of effort in the EClick routine.

8. Scaling: Text scaling is implemented by substituting scaling procedures in the Quick Draw low-level hooks. the scaling method used, however, does all measuring at the reduced scale and is unfortunately subject to rounding errors. Scaling will be reimplemented in the future to be done entirely in the Quick Draw hooks. That will allow measurements to be made at full scale and avoid rounding errors. Because this change is yet to be made, scaling of rulers has been left entirely unimplemented.

9. Dynamic sizing of text objects: Someday, borderless text objects should automatically resize themselves to fit the text they contain. This can be accomplished by creating large edit rectangles for the duration of an edit session and then resizing the object afterwards, but it is not practical because when CoreEdit works in a large edit rectangle it insists on erasing lots of it and messing up the screen.

10. Edit Records: The Pascal declaration of the edit record included with the CoreEdit package is erroneous. It is missing the field firstLine (thus throwing off subsequent offsets) and has the fields MxDscnt and MxAscnt in the wrong order.

11. CEChngStyle: The documented interface to CEChngStyle does not tell the whole story. The Style parameter is actually passed as a word, of course, the low order byte of which contains the style bits to be turned on. The actual assembly routine, however, also looks at the high-order byte and interprets it as the set of styles to be turned off. The right way to package this information from Pascal, however, is not really clear.

Rulers

The term ruler is used two ways within the Etext. The first usage is to refer the text box rulers used by presenter. These are often termed Prulers since they are presenter rulers. CoreEdit has its own idea of what a ruler is, though, and so there is occasional confusion. When an edit session is opened (or a text box drawn) Etext obtains the Pruler for the box from the Ruler module. It then converts the Pruler into five CoreEdit rulers—one for each possible indentation level. The handles for these are kept in an array for use as needed. At the conclusion of an edit session the rulers are disposed of.

The Ruler unit defines the data type PRuler and performs two general functions. First, it manages the presentation ruler pool. Rulers are stored in a dynamic array in the heap. Only the ruler numbers are kept in object records. The Ruler unit can retrieve the actual ruler given its number, change an existing ruler, add a new ruler, etc. The ruler pool being used is passed to Ruler in a call on RActivate.

Ruler numbers are actually encode rather than being a direct index into the ruler array. The number divided by four is the actual index and the remainder is the alignment value (left, centered, right, justified). This avoids duplication of rulers that differ only in alignment. The PRuler record contains an alignment field, however, and this is where alignment is stored outside of the ruler table. Within the table, the alignment field is used as a reference counter.

The second function of Ruler is to handle the on-screen display and manipulation of rulers. A routine is provided to draw a ruler. Slide calls this routine and changes the selection to SELRULER. Mouse clicks are then passed to Ruler for processing. RClick tracks mouse movements and updates the indention markers. It returns a status to Slide to tell whether the click changed the ruler, in which case the text needs to be redrawn, was outside the ruler, or had no effect .

When a ruler is showing for a text box, Slide highlights the box with a dotted border just like the one used for borderless text selections. The border disappears when a slide is deactivated, but the ruler is not erased. When the slide is reactivated, RActivate is used to reestablish the context and slide rehighlights the box.

The Toolbox Window in Presenter

Presenter allows several documents, or presentations, to be opened and viewed simultaneously by displaying them in separate document windows in the same way that MacDraw, Microsoft Word, and other products do. Macintosh conventions call for overlapping windows with the top one considered *active*. The active window is specially highlighted in its menu bar and is the window affected by menu commands and keyboard input. In presenter, this means you can work on one document at a time, but switch documents by simply activating a different window.

Presenter has one additional window, the toolbox, that does not obey Macintosh conventions because it always floats in front of all the document windows—even the active one. In a sense, it is always active when one of the document windows is active. It can be overlaid by a system window (desk accessory), but returns to the foreground as soon as a document window is activated.

Implementing the toolbox window requires Presenter to perform some tasks itself which would otherwise be handled by the system routines. This note describes that implementation.

The Active Window

Macintosh allows you to follow conventions by using routines like SelectWindow and FrontWindow to keep track of the active window. Presenter is unable to use these routines, however, because its active window is *not* the front window. Its toolbox is the front window and the active one is just behind it.

To accomplish this arrangement, Presenter must keep track of the active window itself. It maintains the variable curWindow for that purpose. When curWindow is NIL, it means that no document window is active—either none is open or a system window has the floor.

Activate Events

An application receives an activate event when one of its windows is becoming active or inactive. Presenter assumes that all such events are transitions between document windows and system windows. Transitions between different document windows are handled specially (see below.)

When a document is made active, presenter must enable its menus, display any current selection in the window, and bring the tools window to the front. The system has already highlighted the window and moved it to the front. Moving the tools to the front does not affect the fact that the document window is the "real" active one, but it does create a non-standard situation— the window returned by FrontWindow is *not* the active window. Consequently, Presenter has to set the variable curWindow to the window receiving the activate message.

Deactivate events are only received by the front window because that is the window that by the usual convention is active. Since presenter never allows any of its document windows to appear in front, it always receives deactivate events for the tools window. It knows which of its windows is the real active window, however, so that is the one that it deactivates. It sets curWindow to NIL. Menu items not suitable for system windows are disabled, the selection is removed, and, because the system coudn't do it, the highlighting is removed from the window.

Select Window

When a MouseDown event occurs in a non-active window, convention calls for activation of that window by calling SelectWindow. If a system window is active at the time of the click, that is exactly what presenter does. That generates an appropriate deactivate event for the system window and and activate for Presenter. As described above, the activate brings the document window to the front, highlights it, and then brings the tools to the front of the whole pile. (Note: when a system window is clicked, Presenter does not handle it at all because it's a system event. The desk accessory code will call SelectWindow.)

If the click is simply a change from one active document window to another, however, we want to switch windows without shuffling and redrawing the tools window. That is accomplished by Presenter's own routine, CustomSelectWindow. It performs as follows:

1. Deactivate curWindow by hiding the selection and unhighlighting its title bar.

2. Figure out what region of the chosen window will need to be updated when it is moved to the front. Store this in a temporary region variable.

3. Use SendBehind to move the chosen window to just behind the tools, which are, of course, in front of everything.

4. Call PaintOne on the window, passing it the region calculated in step 2. That

blanks the area to be redrawn and places it in the update region for the next update event.

5. Call CalcVis to recompute the window's visible region for the update. (SendBehind does not do this automatically.)

6. Set curWindow to the new window, highlight its title bar, and show its selection.

Dragging Windows

Unfortunately, the convenient procedure DragWindow does not allow its caller to determine whether the dragged window is to be activated or not. Instead, it inquires about the state of the command key and, if it is not down, calls SelectWindow on the moved window. That generates all sorts of ugly and unecessary window shuffling, so Presenter only uses it when a sytem window is the active one. (Note: if the window being dragged is a sytem window, Presenter does not handle it at all because it's a system event. In that case the desk accessory will call DragWindow.)

Presenter implements its own version in the procedure CustomDragWindow. It works in the same way as DragWindow, but calls CustomSelectWindow instead of SelectWindow. To implement this it performs the following:

1. Set the grafport to a special global-coordinate port maintained for this purpose.

2. Copy the chosen windows strucRgn to a temporary region variable (the same one used by CustomSelectWindow.)

3. Call DragGrayRgn to follow the mouse movements of the window dragging and return the amount of relative motion.

4. Use MoveWindow to move the window to a new location determined from the result of step 3.

5. If the command key is not down and the window being dragged is a document window, call CustomSelectWindow.

Summary

Presenter's special window situation is accommodated by two different mechanisms. First, activate events are handled specially so that the tools remain in front whenever a

Forethought, Inc.

from the Mac of ***Dennis Austin*** December 22, 1986

To: Tony, Dave, Tom, Bob

Subject: Reading More files into Presenter

This is a rough specification for an interface that Bear River could use to read More files into Presenter. The translation from More outline to Presenter slides is specified separately.

Bear River would write a Pascal unit called Input. It has one routine, IRead, that constructs a sequence of slides in the active presentation by reading a More File (or any other file type that it can make sense of.)

IRead uses Standard File to select the file to input. It gets the document handle as a parameter. It then uses the existing Pres unit procedure to construct new slides:

```
function PNewSlide(pres: DocHandle);
```
This routine will need to be modified to return a pointer to the created slide. The pointer is used in two new calls below. (Actually this is more complicated because PNewSlide doesn't do exactly what's necessary. Probably need a new routine.)

IRead then adds contents to each new slide with the following proposed new routines:

```
procedure SSetTitle(slp: Ptr; s: Str255);
```
This changes the text of the title to the argument. The title remains centered and retains the font of the master title. The title will be one line unless the argument string contains return characters.

```
procedure SNewTextBox(slp: Ptr; location: Rect; paras: Handle);
```
Construct a new text box at the given location. The text box has the attributes of the default text box of the presentation. The box contains the paragraphs in the `paras` argument. It's actual type is `ParasHandle` exported from EText, but that type is not available in the Slide interface. It is a handle to an array of `ParasRec` records, defined:

```
ParasRec = record
               align: ALEFT..AJUST; {not yet used}
               level: 0..4; {indent level}
               parH: Handle; {CoreEdit paragraph}
           end;
```

The `ParasRec` must be constructed by building CoreEdit paragraphs. A new routine will be added to Etext to facilitate this:

```
function EParHndl(s: Str255; afont, asize: integer;
                  astyle: Style): Handle;
```

PowerPoint™ on AppleShare™

Tom Rudkin
Forethought, Inc.
February 10, 1987

This note describes Forethought's intention with respect to running PowerPoint on the AppleShare network. However, because technical documentation from Apple about AppleShare is still very incomplete, it is possible that we will actually implement something different from this in version 1.0 of PowerPoint. This note also explains how the current version (1.0D8, dated January 19, 1987) works with AppleShare.

Our general philosophy is that any given file (presentation) can be opened by any number of readers simultaneously, or by a single reader/writer; it may never be opened by two or more different users unless all of them have read-only access. Furthermore, any number of users can execute a single copy of the PowerPoint application simultaneously (whether they are legally permitted to so is a different issue, not addressed by this note).

The Current Version of PowerPoint

In version 1.0D8 of PowerPoint, there is no shared access either to the application file or to data files. When one user is executing a particular copy of the application on a network server, any other user trying to execute it will see the message, "The application is already open or is missing" from the Finder. When one user has a particular presentation file on a network server open in a window in his workstation, the file will not even be visible to any other user in the file list in the standard Open dialog.

There is a simple way for you to modify version 1.0D8 so that a single copy of the application can be shared from multiple workstations, if you have the Apple utility program ResEdit:

- execute the program ResEdit
- select (do not open) the PowerPoint application file
- choose "Get Info" from the File menu
- check the checkbox labeled "Cached"
- choose "Quit" from the File menu
- click the "Yes" button when asked if you want to save PowerPoint

Our Intentions for the Final Product

We foresee at least three important usage paradigms for presentation files:

1. A user is working alone on a presentation and, in general, he does not wish to share any access to it with other users.

   He should place the file in a "private" folder; that is, one with no access privileges to other users. PowerPoint will allow the user to open the presentation and to save changes over it, but no other user will be able to open it or save over it.

2. Several users need to read and modify a presentation.

   They should place the file in a folder to which each of them has See Files and Make Changes privileges. As soon as one user opens the file (except "Open an Untitled Copy"—see below), no other user will be able to open it at all until the first user closes it; thus multi-user access is serial, not simultaneous.

3. Several users wish to share a "library" of standard presentations, from which they copy formats, parts of slides, etc.

   The "librarian" should place the presentations in a folder to which he alone has Make Changes privilege, but to which other users have See Files privilege. Whenever any of the other users opens the file, he gets a read-only copy of it, and the file is closed as soon as it is read; if he tries to save it, he sees an alert saying he does not have write privilege to the file, with a button on the alert that leads to a "Save as" dialog so that he may save it in another file. If the librarian opens it, he gets read-write access to it, and until he closes it no other user can open it; if the librarian happens to try to open it just at the moment that another user is opening a read-only copy, he sees an alert saying that, while he ordinarily is permitted read-write access to the file, he is temporarily denied write access; he may choose either to procede with read-only access or not open the file at all at this time.

In the Open command, we will use a "file filter" to eliminate from consideration any file that is already open on that user's own workstation (i.e. he has it open in one of his windows); however, any file that is open only from some other workstation is available to him in the file list in the standard Open dialog. (This in particular may change, when we receive more technical information from Apple.) This means that a user may see a file in the Open dialog which he nevertheless is not currently permitted to open; we believe this is better than having files appear in the list on some occassions and not on others, depending only on whether another user already has the file open for writing. (The standard Open dialog does not ever show files in a folder to which you do not have See Files privileges.)

PowerPoint's Open command presents the option of opening "an untitled copy" of

a presentation, versus opening the presentation itself. We describe both of these cases in more detail below.

When opening an untitled copy, we will attempt to open the file for read-only access. If this succeeds, we will read the file into memory, close the file, and title the window "Untitled". Our intention is that that should fail if any other user currently has the file open for writing, but otherwise it should succeed. Note that, once the Open is complete, other users (or indeed the same user) may open the file for either read-only or read-write access, because this user no longer has it open.

When opening a file in the "ordinary" way (not untitled), we will attempt to open the file for read-write access first. If this fails, but if we decide that we could open it for read-only access (in particular, if no other user already has the file open for writing), then we will do one of two things. If this user does not have Make Changes privilege to the file's folder, or the file is locked or is on a locked disk—i.e. if the user would not normally be able to write the file—then we will open the file for read-only access, read the file into memory, and close the file, but we will use the file's name as the window's title (unlike opening an untitled copy). If he has Make Changes privilege to the file's folder, but another user currently has the file open for read access, then we will give an alert offering to give this user a read-only copy or to cancel the Open; if he requests the read-only copy, we will proceed as in the former case. In both cases, if the user attempts to save the file, we will give an alert explaining why he cannot save the file and offering a button leading to the normal "Save as" dialog.



JAN 1 8 1987

Registration of File Types and Application Signatures—Confirmation

Date 1/8/87

Company Name:	Forethought, Inc.
Technical Contact:	Dennis Austin
Application Name:	(Please provide us with this information)
Confirmed Signatures:	PPNT
Confirmed Types:	SLDS

Dear Developer:

Thank you for your request for registration of File Types and Application Signatures. This letter serves as confirmation; please keep it. Treat them well and they'll give you years of good service.

_X_ Your original request is attached, you have been assigned the signatures and types listed above.

___ Your original request is attached, with the choices which couldn't be registered circled. These choices were already in use, so please submit an alternate selection.

___ Your request needs clarification; please resubmit your choices along with the equivalent hexadecimal values. Your original request is attached, with questionable choices circled.

If we've misunderstood and listed your request incorrectly, send us a letter bomb as soon as possible so that we may destroy our records that show the incorrect listing.

Thank you,

Shirley McCoy
Apple Computer, Inc.
Developer Technical Support

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
408 996-1010
TLX 171-576

Registration of File Types and Application Signatures

Apple Developer Technical Support

When you ship your Macintosh application, you must register your application's signature, and the types of any files your application creates, with Apple Developer Technical Support.

The Finder Interface chapter of *Inside Macintosh* contains information about how these signatures and file types are used. Additional information about setting up this information in your application is contained in Macintosh Technical Note #48. It should be noted that Apple reserves all signatures and file types containing entirely lower-case letters.

These file types and signatures must be unique, because the Finder uses them to associate your application's documents with your application and its icons. If two applications use the same type or signature, the Finder may display the wrong icon, or launch the wrong application. That's why it's important to register these types with Apple Developer Technical Support. We'll call you if there's a conflict, and a written confirmation will be sent when your information is registered.

We're currently revising our database, and would like to hear from *all* developers who are currently shipping applications, *even if you've previously registered your types and signatures!* If you are shipping more than one application, please copy this form and send us one copy for each application. Thank you.

Send to: Apple Developer Technical Support, Attn: Application Information
20525 Mariani Avenue, M/S 3-T; Cupertino, CA 95014

Developer name: Forethought, Inc.
Address: 250 Sobrante Way
Sunnyvale CA 94086

RECEIVED
DEC 7 1986

Technical Contact: Dennis Austin
Phone: (408) 737-7070

RECEIVED
JAN 2 1987

Product name: not yet named

Please print the following information carefully. If any non-printing characters are used, please include hexadecimal values (for example, " S T R $20 ").

Signatures: PRES[NO] PPNT[OK] SLMK

second choices - ignore if others are available

Custom file types you create: (Don't include standard types like TEXT or PICT)
PRED[NO] SLDS[OK] SLD1#

Custom resource type or scrap type that you create (we will *not* arbitrate these, as conflicts are not usually a problem. Don't include standard types):

Registration of File Types and Application Signatures

Apple Developer Technical Support

When you ship your Macintosh application, you must register your application's signature, and the types of any files your application creates, with Apple Developer Technical Support.

The Finder Interface chapter of *Inside Macintosh* contains information about how these signatures and file types are used. Additional information about setting up this information in your application is contained in Macintosh Technical Note #48. It should be noted that Apple reserves all signatures and file types containing entirely lower-case letters.

These file types and signatures must be unique, because the Finder uses them to associate your application's documents with your application and its icons. If two applications use the same type or signature, the Finder may display the wrong icon, or launch the wrong application. That's why it's important to register these types with Apple Developer Technical Support. We'll call you if there's a conflict, and a written confirmation will be sent when your information is registered.

We're currently revising our database, and would like to hear from *all* developers who are currently shipping applications, *even if you've previously registered your types and signatures!* If you are shipping more than one application, please copy this form and send us one copy for each application. Thank you.

Send to: Apple Developer Technical Support, Attn: Application Information
20525 Mariani Avenue, M/S 3-T; Cupertino, CA 95014

Developer name: _____
Address: _____
_____

Technical Contact: _____
Phone: _____

Product name: _____

Please print the following information carefully. If any non-printing characters are used, please include hexadecimal values (for example, " S T R $20 ").

Signatures:

____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____

Custom file types you create: (Don't include standard types like TEXT or PICT)

____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____

Custom resource type or scrap type that you create (we will *not* arbitrate these, as conflicts are not usually a problem. Don't include standard types):

____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____ ____



DEC 19 1986

Registration of File Types and Application Signatures—Confirmation

Date 12/18/86

Company Name: Forethought, Inc.

Technical Contact: Dennis Austin

Application Name: **(please provide us with this information as soon as you have decided)**

Confirmed Signatures:

Confirmed Types:

Dear Developer:

Thank you for your request for registration of File Types and Application Signatures. This letter serves as confirmation; please keep it. Treat them well and they'll give you years of good service.

___ Your original request is attached, you have been assigned the signatures and types listed above.

_X_ Your original request is attached, with the choices which couldn't be registered circled. These choices were already in use, so please submit an alternate selection.

___ Your request needs clarification; please resubmit your choices along with the equivalent hexadecimal values. Your original request is attached, with questionable choices circled.

If we've misunderstood and listed your request incorrectly, send us a letter bomb as soon as possible so that we may destroy our records that show the incorrect listing.

Thank you,

Shirley McCoy
Apple Computer, Inc.
Developer Technical Support

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, California 95014
408 996-1010
TLX 171-576

Registration of File Types and Application Signatures

Apple Developer Technical Support

When you ship your Macintosh application, you must register your application's signature, and the types of any files your application creates, with Apple Developer Technical Support.

The Finder Interface chapter of *Inside Macintosh* contains information about how these signatures and file types are used. Additional information about setting up this information in your application is contained in Macintosh Technical Note #48. It should be noted that Apple reserves all signatures and file types containing entirely lower-case letters.

These file types and signatures must be unique, because the Finder uses them to associate your application's documents with your application and its icons. If two applications use the same type or signature, the Finder may display the wrong icon, or launch the wrong application. That's why it's important to register these types with Apple Developer Technical Support. We'll call you if there's a conflict, and a written confirmation will be sent when your information is registered.

We're currently revising our database, and would like to hear from *all* developers who are currently shipping applications, *even if you've previously registered your types and signatures!* If you are shipping more than one application, please copy this form and send us one copy for each application. Thank you.

Send to: Apple Developer Technical Support, Attn: Application Information
20525 Mariani Avenue, M/S 3-T; Cupertino, CA 95014

Developer name: Forethought, Inc.
Address: 250 Sobrante Way
Sunnyvale CA 94086

RECEIVED
DEC 7 1986

Technical Contact: Dennis Austin
Phone: (408) 737-7070

Product name: not yet named

Please print the following information carefully. If any non-printing characters are used, please include hexadecimal values (for example, " S T R $20 ").

Signatures: PRES PPNT SLMK _____ ← second choices - ignore if others are available

Custom file types you create: (Don't include standard types like TEXT or PICT)
PRED SLDS SLID _____

Custom resource type or scrap type that you create (we will *not* arbitrate these, as conflicts are not usually a problem. Don't include standard types):

_____ _____ _____ _____

Boston MacWorld Expo — Trip Report

Dennis Austin
August 1985

The show. The Expo was held at Boston's Bayside Exposition Center, about five miles south of town, in a distinctly non-high tech area. That may have reduced local traffic at the show, but the crowd still seemed near capacity. In addition, an impressive proportion of show-goers visiting the Forthought booth were serious business users. In spite of the high quality attendance, however, we didn't transact much business. Judging from the remarks of people in nearby booths, other companies had similar experience.

The booth. Among the large island booths, I think we had the worst location in the hall by a good margin. Fortunately, the show is small enough that most visitors probably went by every booth and we didn't lose too much from our location. We just didn't look very prominent.

The design of the booth is good for an inexpensive setup. Where union rules permit, we could probably save time in the future by doing some of the setup and teardown ourselves. All that is needed is a wrench and a Philips screwdriver. During the show, it would be nice to have an extra chair in the center, a stool at the counter, and a shelf in the demo stations. Placing the demo stations on a carpet-covered step, as done at the Apple booth, would be helpful for relieving the leg and back pain associated with long hours of demonstrations. We should consider it at the next redesign.

Nashoba. Without these guys, we would have been dead. It takes eight or more people to operate a 20x30 booth. They did a great job of demonstrating, and I think they enjoyed it and learned a lot. Our relationship with Nashoba couldn't be better right now. Let's try to keep it up. We also need to extend thanks to Jeff Martin for doing some demo duty.

Peggy Baily. Peggy did an A^+ job of arranging the show, setting up the booth, and tearing it down. The show administration, laborers, Apple, and everybody else she came in contact with were instantly her friends and Forethought's servants. There really weren't any problems at all.

Marketing. Talking to real people in the Macintosh market (proto-customers) led me to a couple of conclusions about our marketing. I am well aware of the fact that I know zip about marketing, and I'm on really soft ground here. Nevertheless, I've got an opinion so here it is.

Naming. First, we have a serious problem with naming. FileMaker, by itself, is a pretty poor name. It isn't offensive, but it doesn't stick in your head easily. That alone makes it a bad choice, but when you have another product named Factfinder, its just plain dumb. The confusion never ends.

And the naming problem doesn't end there. We have two other names, MACWARE and Forethought, to confuse people. Our ads say MACWARE, and our booth says MACWARE, but if you want to find our company—say in a telephone book, in an index-to-advertisers, in an index to booth locations, or in a mail-order software catalog—you have to know the name Forethought! At least partly for this reason, we have lousy brand recognition.

Its really a shame that our ads, our service, and our products are so good and our naming is all screwed up.

Factfinder. On a more positive marketing note, I think we have a promising new strategy for Factfinder. The problem in selling Factfinder is getting across its purpose. "A free-form filing system" may be an applicable technical description, but when you say it to potential customers, their eyes just glaze over. The only impression they vaguely form is that it's some kind of data base product.

When I demonstrated Factfinder, I emphasized its utility in doing research, organizing notes, and in writing papers. Time and again, my audiences would come back with "Oh, I see! But how is it different from Think Tank?" I would explain that Factfinder is really an alternative and complementary approach to "idea processing": random access to notes rather than hierarchical access. People were usually impressed and many said that keyword access would be more useful for them. The important thing, though, is *they understood what the product was for*. Thank you, Living Videotext.

I think this line of thought has a lot of potential. Not only does it suggest a way to market Factfinder, it implies new possiblities for product development. I discussed the benefits of a combined Factfinder/Outliner product with a number of people and they were invariably excited about the possibilities.

Austin on Tables

This note is to record (not to say promote) my opinions and suggestions on the construction of tables with our new application, *Presenter*. My favorite proposal, as you already know, is that we ignore the problem. This is always the cheapest and simplest solution to any problem, and its chances for success are often underestimated. And it has that wondrous property of not introducing any *new* problems!

In building tables with MacDraw, the only irritating difficulty I have found is the proper location of text within the cells of the table. This is automatically taken care of in *Presenter*, so I would be satisfied to construct my own tables with the primitives available. As an example, I used MacDraw to create a table, shown below, from the Intel slides. It was fairly obvious how to proceed, and it took only a few minutes. The hard part was getting the text centered in the boxes.

MCS®-96 FAMILY OF PRODUCTS

Options		68 pin	48 pin
Digital I/O	ROMless	8096	8094
	ROM	8396	8394
	EPROM	8796	8794
Analog & Digital I/O	ROMless	8097	8095
	ROM	8397	8395
	EPROM	8797	8795

Well, so much for that. Most users are not so ascetic as I in these matters, and I know the majority here think we need more specialized tools to build tables the "right" way. What could such tools accomplish? Here's my list:

- It should be easy to insert, delete, or move columns and rows. With only the primitive tools, rows are just multiple selections. They can be cut, pasted, and dragged around to accomplish the desired result, but you don't get any special help in selecting, etc.

- It should be easy to change row height or column width, or resize the entire table. This isn't too bad with the primitive tools either, but, again, you don't get any special help in selecting, etc.

- We want variable ruling. Some lines might be heavier than others; some may be absent altogether. With the primitive tools, this is accomplished by simply drawing the ruling where appropriate and leaving it out elsewhere. This is easy, but it does get into some confusion with overlapping objects. It also makes the previous two operations harder because there are extra lines littering the field when you're trying to work on the boxes.

- When entering text in the table cells, you would like to be able to jump from cell to cell without having to use the mouse. The tab key would be an appropriate method. We have no

support for this with the existing specification, but something could probably be kludged up. The main reason I haven't already done so is that the problem is really wider. We also want to be able to cut and paste the text of multiple cells in one operation. This is not only to move things around in the table easily, but also to exchange data with other applications—spreadsheets.

- We would like to occasionally use full text boxes within the table cells. This is no problem working at the primitive level, but could prove more problematic in a higher level table concept: The multi-box cut-and-paste of text above gets complicated because strange things happen if text from a text box does not land in a text box when pasted. Should "text-boxness" be a property of the text that is carried along on the clip board? And what about allowing some space around the text? Text boxes don't provide margins—they aren't needed in other contexts. Is another mechanism needed?

- We might like to have the minimum size of a row or column determined by the maximum size of the text in any one of its boxes. In addition, the size could grow (and shrink?) automatically as text was entered that exceeded the size of the box. This would make it a lot easier to make unplanned changes, although the correct way to grow and shrink can only be guessed, and this sort of side-effect has often proved a double edged sword.

What do I see as potential for the table object as a full-fledged *Presenter* concept? It could make rows and columns easier to select by providing a shorthand device. Deleting and inserting whole rows and columns would be more automatic, i.e. the table would automatically open up to make space or close up to remove space. Ruling would be less messy because automatic adjustments of the table would also adjust the ruling. And last, but I think most significant, all the text in the table could be connected in a way that would allow multi-box cutting and pasting as I discussed above.

As far as disadvantages, I see an added concept that could have complexity comparable to a text box, but be of considerably less utility. This is a tricky judgement to make, but the lesson is to make any table mechanisms as simple as possible. And by simple, I don't mean elegant. I mean bonehead simple—the kind of stuff you don't need a manual for.

I really don't think it's possible to get all the table features we might want and still keep the mechanism proportionate to its role within *Presenter*. What kind of compromise can we live with? With fear in my heart, I tender the following:

Tables

Creating tables. Any box can be converted into a table by means of a menu command called "Table". The command subdivides the original box into cells approximately 1 1/4 inches wide by 3/8 inch high. It then presents a dialog box in which to specify the number of columns and rows. Initial values depend on how many standard cells fit in the original box. The dialog also offers a check box to make all rulings visible. Invisible ruling shows as dotted lines when the table tool is in use (see below.) The Table command can be given again to change the number of rows or columns.

A table acts somewhat like a big box composed of smaller label boxes. The overall table is selectable as an ordinary graphics object and can be moved or resized as usual. The cell boxes contain text in the same way a label does, but they will not automatically expand to accommodate

create with table tool

new typing.

Selecting cells. In a table, text can be selected within a single cell box, or the entire text of any rectangular group of boxes can be selected. You select cell text with the I-beam cursor as you would in a label box. Edit as usual, changing font, etc. Keyboard commands (cursor keys?) can be used to jump the selection to neighboring boxes. If you begin a selection in one cell and drag to a neighboring cell, a block of cells is selected—or rather the text contents of a block of cells is selected. Any rectangular block of cells can selected this way. Their selection is shown by inverting the color of the entire cell. (A shortcut might be nice for selecting entire rows or columns, but I'm not sure.)

Cut and paste. Selected cells can have text commands applied and can be cut and pasted. Cutting (or copying) text concatenates the text from the selected boxes, in row-major order, saving the cell boundaries as tab characters. Pasting such text into a table selection places it into the selected cells the same way. If there are fewer cells selected than there are tab characters in the clipboard, the extra is not pasted in. Pasting into non-tables works in the usual way with tab characters remaining in the text. If the cells are pasted into a spreadsheet, they are all transferred in their original relationship.

Table tool. To construct attractive tables, the ruling lines can be adjusted with a special selection tool. They can be selected either singly or in groups by shift-clicking or marquee. Once selected, they can be colored with the Pen menu or have their width changed with the Lines menu. Chosing a color of none makes them invisible except when the table tool is in use.

Changing a table. A selected line, or group of lines running the same direction, can be dragged within the table to change the dimensions of the rows and columns. Dragging the lines at the edges of the table changes the size of the overall table. A row or column disappears entirely (contents too?) if one of the lines is dragged onto its neighbor or if the Clear command is given. Selecting a line and giving the Duplicate command creates a new row or column, below or to the right of the line. The Distribute command can be used to evenly space a group of lines in the table.

That's it. You were probably expecting more: There is no provision for making odd tables like the example at the beginning of this note. There is no provision for having text boxes in the table. These things are possible by combining tables with other primitives, or just building a do-it-youself table out of primitive boxes, but there is no automation. There is also no automatic grow, shrink, or space redistribution.

I think what's there is pretty helpful, though. It covers a large range of real tables, and, most important to me, it treats the contents of the table in a useful way that also integrates well with spreadsheets.

Dennis Austin
July 19, 1985

Updated
August 14,1985

Fore**thought**, Inc.

from the Mac of ***Dennis Austin*** August 7,1985

To: Whom it may concern

Subject: System disks

To make it easier for us all to have similar Macintosh system disks, I have placed a few prototype disks in the box beside the LaserWriter. In order to have an ample and unvarying selection of fonts, and to avoid rebooting any more than necessary, I have taken to using a single system disk and keeping my applications on the disk with their data. If there is more than one data disk, I make another copy of the application. If you would like to try this mode of operating, you can copy one of the system disks I've made.

There are two system disks, one for the LaserWriter and one for the Imagewriter. They have all the Apple desk accessories, including a scrapbook with the Forthought logos, etc. The Laser system has the the LaserWriter printing resource and the complete set of Laser fonts (plus a few others, including Cairo and Taliesen). The other has the Imagewriter printing resource and no Laser fonts, but has all the other fonts.

Both include a copy of Switcher 4.0 that has been dragged out to the desktop so you can start it without opening the disk window. Both also have a special startup screen that identifies the disk.

You can customize your own version of these disks by throwing away the startup screen or the switcher, adding a small application, or changing the list of fonts and desk accessories. For this last, I have also left a disk that contains the Apple Font/DA Mover and files with a complete set of fonts and DA's.

There are a few catches, however:

1) The scrapbook is located on the same disk as the currently running application, **not** necessarily the system disk. You may be surprised to see an empty scrapbook sometimes. Copy the scrapbook file from the system folder to your application disk if you want to use it in that application.

2) Clipboard files are also sometimes created on the application disk. Usually the clipboard is in memory, however, so there are not too many problems with it. With Switcher, there doesn't seem to be any problem.

3) If you want to use copy-protected software (yuch!), you'll have to use a backup program like Copy II Mac to make your data/application disks. We have several copies of Copy II Mac around, although we haven't paid for it...

If you have any questions, simply dial 800 MACWARE.

The Foundation Document Editor

An Assessment — May 1985

Dennis Austin

Foundation documents are a sequence of paragraphs, pictures, tables, and other components formatted together for high quality presentation. All of the components can be edited within Foundation. No cut-and-paste between applications is necessary to produce a final document. Tables, for example are prepared in place like an embedded spread sheet; formulas can be revised and new data entered while surrounding text and graphics remain on-screen and active. The editor is designed to take advantage of a bitmapped graphics screen capable of displaying the diverse data objects. Documents are shown, as far as possible, just as they will appear in their final printed form.

The Foundation editor was originally written for a proprietary graphics and windowing package. In the six months since I joined Forethought, we have converted it to use Microsoft Windows. For me it has been primarily an educational effort, since I have had to learn a great deal about both Microsoft Windows (in three drastically different versions) and Foundation. These notes convey my personal assessment of Foundation in both design and implementation. I do not discuss Microsoft Windows here, but I must at least note that it has impressed me very favorably.

The Windows version of the editor runs well enough for simple demonstrations of its integration, but is far from being even a basis for a product. Data fields and tables are not currently functional. Text and bitmap graphics can be entered and edited, but both are primitive demonstration versions. The graphics, in particular, is suitable only for showing the degree of integration achieved.

ARCHITECTURE

Foundation can be divided into five architectural elements.

Virtual Memory and Support. This is a global heap managed as paged segments of arbitrary size. The in-memory pages must remain within the data segment, as short addresses are used for access. Virtual addresses are double words containing a file number and offset. Thus virtual addresses can describe data stored in actual files as well as virtual memory (scratch files or real memory).

Core Editor. This maintains the state of a document, implementing such operations as delete, copy, and insert. A document is represented as a sequence of "objects" of diverse types (characters, bitmaps, data cells, etc.) grouped into regions (paragraphs, graphics areas, tables, sections, etc.) Regions can be nested, allowing a paragraph to appear within in a picture, for example.

The document is maintained using piece tables in which an object takes the place of the more traditional single character. To handle large documents with lots of pieces, the table is organized as a two-level B-tree where the lower level is composed of virtual memory pages and the upper level remains resident. There are two piece tables, one for objects and one for regions.

Class managers. The various types of objects and regions are called classes. Operations on objects of a class are always performed by the class manager. The paragraph class handles paragraph regions, the text class handles character objects, and so forth. Each manager defines a number of standard operations by building a class record containing function pointers. Whenever the editor performs an operation on an object, it's class is used to find the appropriate manager to call. Classes are identified by registered class numbers.

The current Foundation software contains managers for the following classes:

- paragraph region
- text (character objects)
- picture region
- bitmap objects
- window region (actually the display manager)

Numeric data classes are implemented for the original version of Foundation, but have not been converted to the Microsoft Windows version.

Command Interpreter and Commands. This is the outer loop of the editor. It reads mouse and keyboard input, determines the correct response, and engages the editor to perform the appropriate actions.

Trace/Debug. Although not strictly part of the Foundation editor, the ftest program is a application in its own right. It is run at the same time as the editor, where it receives and logs trace messages, displays them in a window, and can accept debugging commands from the keyboard that analyze the state of the editor.

The above descriptions are admittedly terse, leaving out many key aspects of the overall design. As an example, the editor is designed to handle more than one editing window, possibly for different documents, at the same time. Many such omitted details are reflected in the implementation, even though they are not apparent, or working, in the demonstration. To gain some appreciation of the relative sizes of the parts, I have compiled an appendix listing source files grouped roughly by function. The number of lines in each file is shown.

At the highest level, I like this architecture. Most of the concepts have been tried and proven in other systems and their proper combination seems a good prospect for success. At a more detailed level, however, some problems lurk.

The core editor has an intrinsically difficult task, because it must deal efficiently both with objects as small as single characters, typically occurring in long runs, and objects as large as

bitmaps that occur only rarely. This kind of design demand usually leads to sub-optimal results, although it is not clear whether this is the case in Foundation.

The interface to class managers, perhaps the most difficult and important interface in the design, is vague and haphazard. At least it is still unclear to me. It works for the current classes, but a thorough redesign would be a wise investment.

There are other interface problems resulting from a lack of strong design direction. Schizophrenia is one: the virtual memory system has multiple interfaces, as does the piece table mechanism. There is also little shielding of low-level design decisions from module clients. Callers of the virtual memory system, for example, know—and sometimes need to know—the structure of a virtual address.

USER INTERFACE

Foundation's user interface is still largely undetermined. The editor is considered base technology for products yet to come, and the proper top-level interface has not been designed. Some choices had to be made for demonstration purposes, of course. The Windows version tries to maintain the flavor of the Macintosh while taking advantage of some of the unique features of Windows.

The bitmap editor, for example, uses a pop-up toolkit. This would not be practical on the Macintosh because only one window at a time is active and you bounce back and forth between the drawing and the toolkit.

The text editor uses a typical Mac-like interface, but also provides keyboard commands. The existing keyboard commands are borrowed from Unix visual editor, but that is only a default decision awaiting further refinement.

MODULARITY

Although the source text is broken into files by general function, Foundation is not well modularized. There are no clearly defined boundaries and interfaces, and most of the interfaces are very broad.

Foundation is written in the C language. Instead of the C practice of using header files to define and document module interfaces, most exported function names are grouped into automatically generated header files that are included by many source files. The largest of these declares 160 external functions and 25 structure types as well as many simple types and constants.

Altogether, Foundation has about 350 external functions defined in 20,000 lines of C files. That means an external is declared approximately every 60 lines, a fairly dense average. The number of externals is not so important as their confinement, but, unfortunately, Foundation is not organized in a way that restricts access or arranges modules in layers.

The number of static variables is relatively small, but, for debugging reasons, most of them are defined in a single source file, globals.c. That makes it unclear which variables belong to what module and what variables, if any, are truly global.

STYLE

Foundation is written using some traditional C style, such as the indentation convention, and some non-traditional C style, such as the naming convention. The naming convention is a variant of a system (sometimes called "Hungarian") due to Charles Simonyi. Names are divided into parts by beginning each part with a capital letter. The parts are chosen and arranged in a special way that indicates the kind and data type of the object.

The naming system has many advantages when used with ingenuity. However, I find the particular names chosen in Foundation too often ignore the semantics of an object in favor of its type. This makes it difficult for a person unfamiliar with the code to fathom its meaning.

The remaining aspects of the style follow the lead of the naming convention. To me, the code has a flavor that seems to come from a system builder who is undaunted by complexity. It is tight, regular, and thorough, but written almost as if the algorithms were to be hidden from the casual reader. Fortunately, Peter has relieved this situation by adding a lot of program comments.

PORTABILITY

The C language is available for most microcomputers. Although there are variations in the implementations, Foundation does not rely on any C features beyond the most basic. It uses almost no C library routines, so is largely independent of the run-time system. It does use the Microsoft `far` key word to handle the difficulties of the 8086 world, but these could simply disappear in a 68000 implementation, for example.

The files basic.h and tune.h allow Foundation to be readily adapted to different environments by setting up various macro definitions. The current version, based on Microsoft Windows, confines its Windows interface to only a few files that could be replaced in another environment. The dependence on Windows is minimal.

At the same time, since Windows should eventually be available on many different machines, the Windows version of Foundation will be available for all of those machines almost automatically. Windows hides all device dependence from its client programs.

COMPLETENESS

The core editor and virtual memory systems are not prototype software, but complete programs. Many details have been thoroughly worked out in the design and appear in the code even if they

have not been debugged, or even used, in the demonstration system. The history and "undo" mechanism is an example of this. It is a complicated system, however, and many aspects of the design need re-evaluation. There are also, undoubtedly, many remaining bugs.

The text editing features are fairly complete, but are not really successful as implemented. They are not particularly well matched to the underlying Windows primitives, and need to be much faster.

The picture editing features are demonstration quality only. I have too little knowledge of the data and computation facilities to judge their state.

ROBUSTNESS

There are two aspects to robustness: tolerance of unusual circumstances and tolerance of errors. Foundation is generally excellent in anticipation of pathological conditions. It is written so as not to degrade unreasonably with large files, large numbers of fonts, etc. Internal tables are typically limited only by memory space. Here, however, is a hard limit. When your out of memory, your out of it. There is also no recovery from I/O or system errors. Neither of these would be terribly difficult to handle, though.

Foundation also includes a number of checks to help detect programming errors. In debug mode, for example, virtual memory pages are check-summed to catch pages that have been modified but not so marked.

Notwithstanding these efforts, Foundation is still a large program without any firewalls to separate modules. It also runs on an unprotected machine. For these reasons, bugs typically crash the entire machine and can be excruciatingly difficult to find. Investigation of problems often requires a deep understanding of the entire system's structure if it is to succeed.

DOCUMENTATION

There is no specification or design documentation for Foundation. There are quite a few memos describing individual aspects of the system, but they are difficult to read and far from complete. It is hard for a new programmer to learn the software.

For those who already understand the structure, however, there is a good reference manual that is automatically generated from program commentary. Every exported function is annotated with a comment, and the comments are extracted to form a sorted list of all functions, their arguments, and their descriptions. Data types, particularly record structures, are documented in the same way. Unfortunately, the manual can give little indication of how the functions form interfaces since Foundation is not actually modular.

FUNDAMENTALS

I am personally interested in composite documents, having worked on the same concept at Gavilan for two years before joining Forethought. Gavilan's system also displayed formatted documents on a bit mapped screen. Both architectures are "object-oriented" with object class managers in charge of the details of the various elements of a document. Both use a memory management system to make optimal use of main store and hide the details of secondary storage.

These are complex systems based on technology that is still developing. It is easy to underestimate the difficulty of their successful completion. First there is the central problem afflicting all large software projects: achieving a design that allows the program to be decomposed into intellectually manageable units, and then upholding the quality of the that design with coding techniques. Neither Gavilan nor Forethought has met this challenge particularly well, although for different reasons. Gavilan had good design, but inadequate tools. Forethought has better tools, but weak design discipline.

In addition to this classic, I see two special considerations for composite document editors. First there is the graphic user interface. What factors create a good user interface in a direct drive graphic environment? Now that software is appearing in quantity for the Macintosh, we're starting to see enough variety to distinguish the capable from the clumsy. But user interface design is still pure art and is likely to remain so for a few more years. As more direct drive programs come to market, expectations of excellence will rise.

For composite documents, the difficulty of designing a good user interface is compounded because of the diversity of data objects and therefore of editing metaphors. It is, in effect, a word processor, a spreadsheet, a graphics program, and who-knows-what-else combined. Each of these areas must have an appropriate interface and they must be sensible and intuitive when taken as a whole. This is essentially the problem faced by the recent integrated business products, which, on the whole, have flopped.

The second problem is internal integration of various types of document components. The notion of object-oriented programming is a natural basis for the solution, since the user actually visualizes the components as objects within the document and sends them messages by poking them with the mouse and selecting commands to act upon them. The trick, however, is specifying the messages that object classes should respond to.

One could choose the message level of a windows management system like that in the Macintosh or Microsoft Windows. There, messages are events like paint requests, mouse clicks, and menu selections. This is adequate, but has no unifying qualities that guide a class manager its relationship to the document as a whole. High level messages have the opposite problem. There is a limited number of messages chosen to reflect the role of an object within a document, but it can be hard to anticipate all the types of communication that may become necessary—especially if the universe of object types is not known in advance.

In what direction will we find the solutions? The advancing state-of-the-art is leading us. Some questions of the user interface have now been all but settled with the conventions adopted

by the Macintosh world. Hints on a good interface to class managers may also come from Macintosh components like CoreEdit and from Microsoft Windows conceptual building blocks like child windows.

RECOMMENDATIONS

The Foundation editor has potential as a basis for other products, but it I think it needs a great deal of work. I would suggest the following:

Reconsideration of the memory management package to simplify its interface, hide its implementation details, and take better advantage of the Windows memory manager.

Simplify the interface to the core editor and strictly separate the levels of regions and objects. The display manager should work in terms of regions, both in interactions with the core editor and with the display process. The region managers, in turn, work in terms of objects, if appropriate. Some regions may not need a general-purpose object interface.

Design new object manager and region manager interfaces. The interfaces should be extremely narrow and well defined. Consider the utility of the Windows child window mechanism as part of the interface.

Design new region and object managers. These should be designed to have very little dependence on the editor software. It should be possible to employ them in other software contexts.

In addition to these items, of course, is the design and development of the product to be based on Foundation. A full function composite document editor is certainly not the right product. From a technical point of view, it is difficult to do well. From a marketing point of view, integrated products are simply not hot prospects. Even the eagerly anticipated Lotus Jazz is already taking criticism for being too complex, and it is not even released.

To create a more modest special-purpose product out of Foundation, we could proceed on a more modest budget. Still, there is a lot to be done. A development team for completing the job in a year or so should include:

Architect: Interface designer and specifier, design consultant.

Support person: do memory management, document structure and representation.

Graphics programmer: design and build graphics region manager.

Text expert: design and build text region manager.

Product builder: adapts underlying support to form final product.

The project is challenging and fascinating, and has great long-term potential because the Foundation software would be suitable for building additional products with lower development costs. Unfortunately, to me, this does not sound like an appropriate course for the "new" Forethought. Our all-too-finite resources do not accommodate such a long-term investment. This is a discouraging situation, but we must recognize it. We already recognize that our first internal product must be fairly simple. I think we also need a more economical approach to its development.

Foundation Source Files

General support and virtual memory

```
  36 tune.h
  67 vm.h
  89 ch.h
 148 macdefs.h
 166 globals.c
 176 da.c
 263 ios.c
 293 support.c
 377 basic.h
 395 alloc.c
1252 vm.c
3262
```

Text display and edit

```
  11 ln.h
  52 cltext.h
  54 ratio.h
  62 dif.h
  63 clpar.h
  82 dev.h
  92 fmtline.h
 108 attpar.c
 165 cltext.c
 171 selpar.c
 173 dev.c
 347 dif.c
 366 ln.c
 415 fmtline.c
 421 cltextft.c
 444 devwin.c
 602 font.h
 890 clpar.c
4518
```

Pictures

```
 207 sg.h
 213 sg.c
 460 clpix.c
 660 clpict.c
 713 pixwin.c
2253
```

Data fields

```
  24 clfield.h
  24 flcvt.h
 220 flcvt.c
 228 clfield.c
 229 parser.h
 337 clrow.h
 366 eval.c
 487 rowutil.c
 736 clrow.c
 916 parser.c
3567
```

Core Editor

```
   8 ed.h
  85 cehist.h
 513 cehist.c
 726 cecp.c
 847 pt.c
 951 edext.h
1150 eddcl.h
1219 ce.c
5499
```

Control, commands, and basic display

```
  37 cmddefs.h
 242 cm.h
  86 cmdext.h
  69 cmdft.c
 605 cmd.c
 607 cmd2.c
 128 cmd3.c
   3 init.h
  14 prng.h
  27 pixwin.h
  33 icl.h
  60 initext.h
 100 clpict.h
 116 winint2.c
 137 scext.h
 148 chtype.h
 179 rl.c
 188 minigraf.h
 214 chcm.h
 239 prims.h
```

```
  247 clscreen.c
  327 screen.h
  333 clft.c
  428 kbd.h
  437 cl.c
  441 re.c
  589 clww.c
  631 winint.c
 1209 doc.c
 7874
```

Ftest debugger

```
   11 ftest.h
   22 epm.h
   93 msgs.c
  112 tk.h
  222 pxft.c
  253 fterror.c
  335 test.c
  432 tk.c
  503 printf.c
  633 px.c
 1404 ftest.c
 4020
```

Tabulation

Dennis Austin

Originally, the tabulator on a typewriter was a simple device for quickly positioning the carriage in order to type data in columns. Now, though, the advent of word processors has made tab stops more than a typing convenience. They are needed, for example, with proportionally spaced fonts, to precisely align text on the page. They have taken on new characteristics for the entry of right-aligned or centered columns. Most important, the positioning of the tab stop and the text that is aligned with it is a *stored feature of the document*. That means that stops can be moved, added, or deleted during later editing to change the position of columns.

The ability to change tab stops after typing, and have the text realign itself automatically, is one of the amazing advances of word processors over typing. It is, however, usually a confusing topic to typists and sometimes painful even to experienced editors.

I think much of the confusion stems from the notion of invisible tab "characters" that are inserted in the text each time the tab key is pressed, and then reinterpreted whenever the text is reformatted. The purpose of the tab key is to get to a certain position, *not* to skip a certain number of tab stops. The following paragraphs outline a proposal designed to simplify and amplify the usefulness of tabs in Forethought documents .

Tables

Our first improvement in tabulation is a semantic distinction between tables and running text. Tables are characterized by the need to operate on rows or columns rather than strings of characters. They do not use the features of word fill or justification associated with text in paragraphs. And lastly, they need elaborate facilities for centering, aligning at decimal points, and the like.

Forethought tables are completely different entities from running text. When a table appears in a document, the positioning of its columns, their alignment, format, and so forth, are properties of the table. They are unrelated to tab stops set up as a continuing property of the surrounding text.

Formatting of tables is another subject unto itself and I mention it here only to remove its realm from the duties of ordinary tabs. Tabs in running text will serve only to begin and left-align text at positions between the margins. As such, their role is rather minor in comparison to that of tabs in a more traditional word processor.

How Tabs Work

When a tab is inserted in text, it means that white space is to be inserted so that the following character will lie at a certain position. The position is determined by a particular tab stop, not the "next" tab stop. We say a tab is bound to a particular tab stop. If you move the stop, the text is realigned accordingly. Moving, adding, and deleting other tab stops does not affect text aligned to the first.

When a tab is typed, each press of the key positions the insertion point to the next tab stop to its right. If there are no more stops, the cursor does not move. Regardless of the number of key presses, only one tab is stored in the document. It is bound to the stop where text finally resumes on the line.

When you move a stop on the tab ruler, all tabs bound to it change their white space and reformat the text accordingly. Changing the position of a stop only affects tabs that are bound to that stop. Adding of new stops does not affect the text format at all. Deleting a stop demands that any tabs bound to it must either be deleted or rebound. (Deletion seems to be the more obvious alternative, but I am not certain which behavior is best. Perhaps a choice must be offered.)

Because the tab key moves the insertion point rightward, all tabs originally show as white space. After later editing, however, the stop to which a tab was bound may lie to its left on the line. When this happens, the tab remains in effect (in case the situation is subsequently changed), but does not display any white space at all. (Another alternative from Peter Bishop: a line break is inserted so that the text after the tab begins at the correct postion but on the next line.)

For editing purposes, a tab acts as a character. It is selected like any other character. When text is inserted from a move or copy, tabs are bound as though they were being typed at their new location: to the next stop to their right.

Tabs are rebound by typing with the insertion point positioned just after the tab's white space. Pressing the tab key then moves the insertion point and rebinds the tab to the next stop rightward. Pressing reverse tab (shift-tab, generally) will rebind the tab to the next stop leftward. If this shrinks the tab to nothing, the tab is deleted. Backspacing over a tab also deletes it.

Tab Rulers

Tab stops are set on a tab ruler. The ruler is a graphical device similar to the rulers used by MacWrite, LisaWrite, and others. The ruler contains other format settings as we find them appropriate. A command allows the ruler to be displayed for modification at any time.

To create local tab contexts, a command inserts a new ruler, a copy of the global one. The context ends by inserting an End-Local-Format marker. This is another graphical device similar to a Page-Break marker. It reinstates the global ruler. Local contexts can be nested arbitrarily.

Absolute Tabs

Peter Bishop has suggested the idea of tabs that are not bound to any stop, but simply to an absolute position on the line. When the tab is typed, the position is determined by a combination of tabs, spaces, or backspaces. Perhaps it could be chosen by pointing with the mouse and clicking in a special way. At any rate, a tab so typed would not be affected by the tab ruler at any time. This is an interesting idea that we can investigate in the future.

Evaluation

The design suggested above is a substantial departure from conventional practice and will undoubtedly be more difficult to implement. It will need prototyping and evaluation before we can adopt it without reservation. I expect, however, that it can eventually eliminate many of the sticky problems that arise with conventional tabs.

Dennis Austin

Notes on Document Models

Dennis Austin

I have written these notes to clarify my opinions about document structure for myself, as well as describe them for you. My motivation is a growing dissatisfaction with the graphic user interface to word processors. I am impressed by the document creation capabilities of File Maker and I am groping for the right way to bring those capabilities to general documents.

At this point I think experimentation with simple user interfaces should be the next step in development. I am not sure when the experimentation could begin, however, and I also have little idea how to adapt the core editor to handle the type of document I describe. The main issue is that the core editor does not treat a document as fundamentally pictorial. This does not necessarily diminish its utility, however, since even in my view a document still has many sequential aspects.

A Diatribe on Sequentiality

A text files is composed of a sequence of characters. Exactly how the characters might be formatted into lines and the lines placed on a page -- turned into actual text, in other words -- is up to the program using the file. The only rule is that if character A precedes character B in the sequence, A will appear either above or to the left of B on the page.

Text editors display and edit text according to some simple rules of line formatting. Lines end with new-line characters, for example. The mapping from screen image to sequential text file is almost transparent. To slide a word over on a line, for example, you insert spaces or tabs in front of it. This model meshes nicely with the I/O devices generally used for editing and printing; they are strictly sequential character devices too.

When batch document formatters appeared, they used text files, prepared with a text editor, as input. They naturally adopted a sequential model of the document, building it essentially by placing one character after another on a page, according to what by now are quite complex rules. (Concepts like macros and diversions represent departures from strict sequentiality.)

Finally, word processors completed the evolutionary chain, again relying on the sequential model. Now the editor and formatter are combined, bu moving things around on a page (aside from running heads, etc.) is still largely a matter of inserting or deleting spaces and tabs.

Semantically, of course, text is a sequential thing. Sequences of characters form words, sequences of words form sentences, sequences of sentences form paragraphs, etc. And continuous text is normally displayed as a long sequence of characters divided into lines, paragraphs, and sections. Some documents, like novels, contain almost nothing but text of this sort. (They still

have running heads, title pages, etc.)

Most of what we read, however, is not like this. We read advertising, magazines, newspapers, catalogs, financial statements, manuals, directories, and more. To a greater or lesser degree, these documents demand graphic arts design in addition to simple sequential text placement. Preparing them comprises both writing "copy" and composing pages.

Graphics First

As word processors become more sophisticated, and their users less so, the sequential document model becomes more and more cumbersome. While pages do contain sequential text, page layout is graphic. Only a a graphic interface can give the direct control that is appropriate.

At the outermost level of design, then, a document editor should be a graphics editor. Its job is to create pages by placing and sizing the various elements of the document, both relative to the page and relative to one another.

Elements are the building blocks of a document. They are often simply placed with respect to a page, but they can also be made to overlap and determine their position and/or appearance from the position or contents of other elements. Elements have common properties such as:

- Shape and Size (line, rectangle, circle, etc.)
- Edges (none or thickness and brush type)
- Background (transparent or brush type)
- Type (contents)

Elements can be purely graphical, or they can have contents that they display according to their own rules. This can be text, a picture, or information extracted, or computed, from a data base, etc. This is the source of the semantic contents of the document. Most important, the contents can be contained in the document itself, or may be derived from an outside file or program.

Preparation of a document proceeds, perhaps even simultaneously during the editing process, on two fronts: construction of elements and placement of elements (page layout).

Text Elements

As the historical pre-eminence of the sequential document model may suggest, most documents prepared on computers are primarily text. It is with text that computers are of greatest service, precisely because of its
sequential semantics, but two-dimensional display. Changes to text are usually made for semantic purposes, and the computer automates the translation into a new two-dimensional

DENNIS,
I FOUND THIS
IN A PILE. IS IT
HISTORICALLY THE INVENTION
OF TEXT BOXES — EXACTLY ONE
YEAR AGO?

Notes on document models

Dennis Austin

I have written these notes to clarify my opinions about document structure for myself, as well as describe them for you. My motivation is a growing dissatisfaction with the graphic user interface to word processors. I am impressed by the document-creation capabilities of File Maker and I am groping for the right way to bring those capabilities to general documents.

At this point I think experimentation with simple user interfaces should be the next step in development. I am not sure when the experimentation could begin, however, and I also have little idea how to adapt the core editor to handle the type of document I describe. The main issue is that the core editor does not treat a document as fundamentally pictorial. This does not necessarily diminish its utility, however, since even in my view a document still has many sequential aspects.

A Diatribe on Sequentiality

A text files is composed of a sequence of characters. Exactly how the characters might be formatted into lines and the lines placed on a page -- turned into actual text, in other words-- is up to the program using the file. The only rule is that if character A precedes character B in the sequence, A will appear either above or to the left of B on the page.

Text editors display and edit text according to some simple rules of line formatting. Lines end with new-line characters, for example. The mapping from screen image to sequential text file is almost transparent. To slide a word over on a line, for example, you insert spaces or tabs in front of it. This model meshes nicely with the I/O devices generally used for editing and printing; they are strictly sequential character devices too.

When batch document formatters appeared, they used text files, prepared with a text editor, as input. They naturally adopted a sequential model of the document, building it essentially by placing one character after another on a page, according to what by now are quite complex rules. (Concepts like macros and diversions represent departures from strict sequentiality.)

Finally, word processors completed the evolutionary chain, again relying on the sequential model. Now the editor and formatter are combined, but moving things around on a page

We still want the user to edit the semantics, not the layout. But we definitely have the full range of graphics input tools with which to work.

(aside from running heads, etc.) is still largely a matter of inserting or deleting spaces and tabs.

Semantically, of course, text is a sequential thing. Sequences of characters form words, sequences of words form sentences, sequences of sentences form paragraphs, etc. And continuous text is normally displayed as a long sequence of characters divided into lines, paragraphs, and sections. Some documents, like novels, contain almost nothing but text of this sort. (They still have running heads, title pages, etc.)

Most of what we read, however, is not like this. We read advertising, magazines, newspapers, catalogs, financial statements, manuals, directories, and more. To a greater or lesser degree, these documents demand graphic arts design in addition to simple sequential text placement. Preparing them comprises both writing "copy" and composing pages.

Graphics First

As word processors become more sophisticated, and their users less so, the sequential document model becomes more and more cumbersome. While pages do contain sequential text, page layout is graphic. Only a a graphic interface can give the direct control that is appropriate.

doesn't quite follow

At the outermost level of design, then, a document editor should be a graphics editor. Its job is to create pages by placing and sizing the various elements of the document, both relative to the page and relative to one another.

Elements are the building blocks of a document. They are often simply placed with respect to a page, but they can also be made to overlap and determine their position and/or appearance from the position or contents of other elements. Elements have common properties such as:

- Shape and Size (line, rectangle, circle, etc.)
- Edges (none or thickness and ~~brush~~ pattern type)
- Background (transparent or ~~brush~~ pattern type)
- Type (contents)

Elements can be purely graphical, or they can have contents that they display according to their own rules. This can be text, a picture, or information extracted, or computed, from a data base, etc. This is the source of the semantic contents of the document. Most important, the contents can be contained in the document itself, or may be derived from an outside file or program.

Placement of pictures:

A)
1) near text
2) at a particular point on page (or on even/odd page pair) absolute or reference
3) between specific paragraphs
4) fancy rules (alternating top + bottom)
→ how do we specify this?

B) by page layout

Preparation of a document proceeds, perhaps even simultaneously during the editing process, on two fronts: construction of elements and placement of elements (page layout).

Text Elements

As the historical pre-eminence of the sequential document model may suggest, most documents prepared on computers are primarily text. It is with text that computers are of greatest service, precisely because of its sequential semantics, but two-dimensional display. Changes to text are usually made for semantic purposes, and the computer automates the translation into a new two-dimensional display, essentially taking over the once difficult job of typesetting.

From this we can expect that the most useful type of element will be text. In fact, the success or failure of a graphical approach to document preparation is likely to rest on the characteristics of text elements. The graphical outer shell must not work against the sequential nature of text.

The basic text box can be compared to a MacWrite page. Its width is adjustable and the text formats to fit. Text may overflow into a subsequent box automatically. Unlike a MacWrite page, features like tabs and centering are relative to the box. Character properties may change, but larger-scale properties like line spacing, justification, and paragraph indent stay constant within a box.

Boxes can be split in two or rejoined. When insertions are made, boxes cay automatically grow vertically, pushing other movable elements onto subsequent pages. An immovable element would force a new page to be inserted before it, thus maintaining its position.

The user interface for box creation, splitting, etc. is of crucial importance. There will be tradeoffs between simplicity, generality, and speed of use. The key to success will be different user interfaces for different purposes. Each of our products can use the text elements in different ways depending upon the type of application they cater to.

Internally, our programs should be structured to permit elements of the various types to function independent of their context. This creates a high-level toolkit for the building of our applications in the same way that windows, controls, etc. create a low-level toolkit.

this is why our first goal is to try to make it unnecessary to edit the layout.

NO

Whenever editing of the layout is necessary, we must try to obtain the semantics of the edit so that additional layout editing after the document editing is not necessary.

Discussion on 3/4/85

The window formatter really is a page layout program. We need the ability to specify page layout using graphics editor.

Glue between rectangles

(text rectangles have a given width (controls wrapping) and a variable height (# of line))

(text rectangles may need a ~~glue~~ real width as well)

(Specify glue by laying out 2 different sizes

→ second size determined by system. Movement of rectangles on second is constrained by glue settings.

Page layout uses field objects to specify contents. Need a way of specifying continuation boxes for a field.

Part of the document environment is the current page layout (First page of section or document may use a new page layout - this is how we do headers + footers + columns + new pages)

Fore**thought**, Inc.

from the Mac of ***Dennis Austin*** July 8, 1986

To: Macintosh Users

Subject: Incrementitus ad nauseam

The good news is that there has been no new release of software from Apple, although we all know this won't last forever. The bad news is that I made some kind of mistake in building the master disks for the last release (System 3.2, etc.) I don't know how it happened, but there is something a little wrong with the Helvetica fonts on those disks. You can get some strange letter spacing, or at least not-standard letter spacing, when using various styles. There may be some other problems with it too. This will probably not affect you so long as you never use your documents with a system disk that has a correct Helvetica on it, but the fastidious among you may want to get everything right.

Note: This is not an everybody-upgrade-now deal. Do what you want.

Converting

I have made new master disks that are functional equivalents of the ones I made when 3.2 was released. They are in the box in my office for you to copy as needed. They are the same as the last versions except for dropping Athens 18, New York 10, and Courier 9. This gives about 10k more free space on the disks which was badly needed. If you need a little more space, you can get about 11k more by deleting things from the Scrapbook. I also dropped FKEY 9 (screen saver) because some people had trouble with it.

For those with custom system files, I can do a special quicky upgrade using Apples resource editor, ResEdit. It is the fastest way to get the right Helvetica, but some say ResEdit should not be trusted with valuable disks. All I can say is that I updated my own system with it and had no problems.

For the record, the current versions are still:

System File	3.2
Finder	5.3
ImageWriter	2.3
LaserWriter	3.1
Hard Disk 20	1.1
Font/DA Mover	3.2
Chooser DA	2.3

Forethought, Inc.

from the Mac of ***Dennis Austin*** May 29, 1986

To: Macintosh Users

Subject: New System Software

For the sheer joy of stirring up trouble, I've decided that it would be fun for all of us to switch to new and incompatible versions of the Macintosh system software! We're not really the first ones on the bandwagon since this is basically the stuff that came out with the Macintosh Plus in January. So far we have been unable to use it because the 3Com software for the 3Server couldn't handle it. 3Com has finally fixed their problems, but the delay was probably well taken since Apple has made slight revisions twice since January. Another small revision is expected within a month, but it should not be incompatible with the current stuff.

You may be thinking, "Well, I'll just wait until everybody else has switched and see how they get along." Unfortunately, the new LaserWriter software is incompatible with the old so if we don't all switch at the same time we'll constantly be powering the LaserWriter off and on to reinitialize it every time somebody uses a different version. I propose we all change Tuesday June 3rd.

For those of you that use more-or-less standard Forethought system disks, everything is real easy. I have created new master disks, described below, which you can copy to get up to date. For those who like to customize their system disks, things will be tougher. The only good way to proceed is to copy one of the standard system disks and then customize it to match your old one. (If you can live without your custom version for a while, upgrading will be much easier with the upcoming revision from Apple. They will provide an installer program that can update your custom system without changing any of its custom features.)

The Master Disks

Laser System

This is the basic Forethought System disk. It has version 3.1.1 of the System software and version 5.2 of the Finder. It has the standard Macintosh Plus desk accessories, plus three others that I have added on a lark. It also contains a few custom function keys. See below for more on both of these.

It includes the 3.0 LaserWriter software and a set of fonts aimed at LaserWriter printing with a few others for variety and compatiblity:

Helvetica	9,10,12,14,18,24	Laser use
Times	9,10,12,14,18,24	Laser use
Courier	9,10,12,14,18,24	Laser use
Symbol	9,12	Laser use; more sizes are available
Geneva	9,10,12,18,20,24	Most common ImageWriter font
Athens	18	For variety

Venice	14	For variety
Monaco	9,12	The standard monospaced font
Cairo	18	Pictures
Taliesin	18	Pictures
Chicago	12	System font

Proof System

This is identical to the Laser System disk except that instead of containing the LaserWriter and LaserPrep resources in the system folder it has an ImageWriter resource. It is intended for use by people preparing documents intended for the LaserWriter, but on a machine that is not attached to the LaserWriter. It permits proof-printing on an ImageWriter instead.

3Com System

This is identical to the Laser System disk but also contains a copy of the EtherMac program from 3Com. It is EtherMac 1.1. Earlier versions will not work with this new Apple software. Copy this disk if you use volumes on the 3Server. You can then modify your copy to automatically log in, etc. Remember to update any system folder on your network volume.

IW System

This is like the Laser System, but it includes the ImageWriter resource instead of the LaserWriter. It has a set of fonts more suitable for use on the ImageWriter. It does include some laser fonts in limited sizes just for compatibility.

Helvetica	9,12	Laser use
Times	9,12	Laser use
Courier	9,12	Laser use
Geneva	9,10,12,14,18,20,24	Most common ImageWriter font
New York	9,10,12,14,18,20,24,36	Times-like font for the ImageWriter
Athens	18	For rationalists
London	18	Old English
Los Angeles	12,24	Laid back
San Franciso	18	For anonymous threats
Toronto	9,12	Courier-like, but proportional spacing
Venice	14	For calligraphers
Monaco	9,12	The standard monospaced font
Cairo	18	Pictures
Taliesin	18	Pictures
Chicago	12	System font

Other Disks

I have also updated some master disks containing MacWrite, MacPaint, MacDraw, and MacTerminal as well as various utilities, fonts, and desk accessories in their current versions. See me for more information.

Desk Accessories

The new system files contain three desk accessories that I have added to the standard Apple set. I have found them useful and thought that you might, too. If you don't want them, or you need more space on your system disk, you can remove them with the Font/DA Mover program. This progam has been re-released with the new system software, though, so be sure to get version 3.1 from me.

FileFinder

This is an accessory sold directly by Nashoba Systems, the developer of FileMaker. Its purpose is to allow you to copy, delete, or rename files without quitting your application to return to the Finder. Have you ever wanted to save a file, but there wasn't enough room? With this accessory you can look at the disk and delete or copy something elsewhere to make the space you need. It also allows you to find out how much space is available and read GetInfo comments, etc.

Multi-Scrap

This one works like the old familiar scrapbook, but it allows you to pick the scrapbook file you want to use. You can even create new scrapbooks and organize your scraps as you like. Unlike the Apple scrapbook, it also gives the option of viewing a scrap at full size or reducing it to fit in the window as needed. (While running, it adds its own menu to the menu bar.)

Windows

When you select Windows it adds a new menu to your menu bar. On it are listed all the windows you have open. You can bring a particular window to the top by selecting it from the menu. Another menu command allows you to close (remove) the Windows menu. This can be handy if you use a lot of folders in the Finder.

Function Keys

Holding down the combination Command-Shift and typing a number is referred to as selecting a function key, or more usually, an FKEY. The numbers 1-4 are supplied by Apple and are described in the Macintosh manual, although it does not refer to them as FKEYs. The master system disks have had the standard FKEY 3 changed a bit, plus they have four new ones installed. These cost you only 3k of storage in your system file.

3:

Copy the screen to a MacPaint document. The only difference from the standard Apple FKEY 3 is that the MacPaint picture has a nice sharp black edge on it and has the rounded corners you see on the screen. It is also centered on the page instead of being in the upper left corner. MacPaint documents are named Screen A, Screen B, etc.

5:

Copies a portion of the current screen to the clipboard. When you select FKEY 5, the cursor changes to a little ⌜-shape. Point it at the upper left corner of a rectangular area you wish to copy. Press the mouse button (cursor changes to a ⌟-shape) and drag to the lower right corner. Release the button and the area you marked will be placed on the clipboard ready for you to paste into a document or scrapbook.

Warning: Make sure you drag from upper left to lower right. The cursor shape changes in such a way that it's easy to remember, but if you do it wrong, I think it crashes.

8:

This just changes the cursor to appear twice its normal size. It's a novelty that is sometimes useful in demos. Select FKEY 8 again to change it back.

9:

FKEY 9 puts the Mac to sleep temporarily. It blacks out the screen and starts a white apple bouncing around on it. The idea is to keep your screen from burning in an image. Press the mouse button to wake up again.

0:

This opens a window on the screen that displays the names of the FKEYs that are installed in the system you are using. This is the number you need to remember and it will tell you what the others are! Unfortunately, the standard Apple ones don't have names, so you won't learn much about them.

Forethought, Inc.

from the Mac of ***Dennis Austin***

February 20, 1985

To: EtherMac users

Subject: 3Server Backups

Files stored on our 3Server disk are normally backed up to tape automatically, requiring human intervention only to physically change tapes. I am the usual human, but since I'm not very reliable, others of you may occasionally be stuck with EtherMac out of service because I forgot to change tapes, or because a tape ran out before I expected. This note explains our tape scheme so you can change tapes yourself if you need to.

Every Monday I put in a tape labeled TFx. Monday night (actually 3:00AM Tuesday) the server dumps its entire disk to that tape. It never needs a second tape for this backup. If I forget to put the tape in, though, Tuesday morning the little LCD display says "insert tape" and EtherMac won't work until you do and the backup is finished —about 45 minutes.

There are actually four tapes: TF1, TF2, TF3, and TF4. They are used in rotation so that we always have full backups for the last four weeks. Whenever I put one in, I note it on the calendar above the 3Server so I know which one was used when. If you need to put one in, simply use the next one in numerical order.

Every Tuesday I put in a tape labeled Ix. It is used for the incremental backups run every night except Monday. Incremental tapes backup only those volumes changed since the previous incremental.

There used to be two incremental tapes, I1 and I2, that were used in rotation like the TF tapes. I write them on the calendar too. Nowadays, though, one incremental tape isn't enough to make it through a full week's incremental backups. When it runs out (about Friday) the LCD announces "insert next tape" and you are stuck.

This is solved by having four tapes: I1a, I1b, I2a, and I2b. The "a" tape is the one I put in on Tuesday. If it runs out, put in the corresponding "b" tape.

That's all you need to know. If I'm available, feel free to find me instead of puzzling through these instructions. But now you're equipped for emergencies.