

MacDraft: WORKING PAPER DISTRIBUTED FOR COMMENTS ONLY

Opening The Hood Of A Word Processor

by Alan Kay*

Abstract

In the near past a user had to be a "techie" just to get an operating system or word processor to do something. The Macintosh revolution with WYSIWYG user interfaces has provided a vehicle for everyone to use the personal computer as an amplifier of their reach. But as one's reach is amplified, so too one's vision. Now that the users can grasp much more of the computer material, they can also better see what they need. Many of these new needs are for ideosyncratic restructuring of their tools to suit new situations as they come up. Not only is "vertical software" a wave of the future, but to do most of it one's self will be the crest.

*Hayn Hirsh assisted with the design. Bill Swartout of ISI/USC was the prime kibitzer. Valuable insights were supplied by Bill Atkinson, Steve Capps, Dan Ingalls, Larry Tesler, and Bud Tribble.

Apple Computer Corporation

Macintosh Division, Bandley Road, Cupertino, CA

Opening The Hood Of A Word Processor

by Alan Kay

Personal computers imply personal software. Word processors, the "universal tool", should be the most ideosyncratic. But a canned system with many users is the least likely to fit the wide diversity of needs. Even parameters and style sheets help only a little -- they resemble the half-hearted attempts at "customizable" financial packages before VISICALC came along. As users, we expect to make our own pictures with a MacPaint or a MacDraw, our own financial planners with a MacProject or MacPlan, our own documents with a MacWrite. As we become more sophisticated, our ideas about what our word processor, our graphics system should do for us diverge more and more from those of the initial designers. We now want to edit our tools as we have previously edited our documents.

With paper, pen, and a xerox machine, doctors and dentists can design and make just the forms that are needed for their business. But they won't look as nice as they would like -- nor will they be able to compute any of the relationships implied by the forms. A "kit" system like 1-2-3 will help with the relationships but doesn't give enough flexibility in

layout or list processing. The growing vertical software market will soon offer these professionals a package that will help them do some of these tasks, but it is likely that some of the flexibility of a 1-2-3 will not be furnished. What is needed are ways to build applications that first, do useful things for the user; second, surreptitiously teach the user deeper ideas about how the application is structured, and third, let the user "open the hood" and have a high probability of success at doing customizations that the applications programmers didn't specifically anticipate.

At first glance, a word processor seems like a difficult target. To build one from scratch, even in the highest current languages, requires more design expertise than most novice users possess. It appears that the novice must instead try to *modify* a system designed and built by someone else -- a little easier than "from scratch" construction, but fraught with the difficulties of penetrating another's style, terminology, and sheer bulk of code.

But even a simple, or simplified, system will not be enough. Imagine our user with a nice WYSIWYG word processor writing away in the midst of a document, the frustration building: "Damn it! I'm sick and tired of the way this thing allocates space. I want less between words and more after punctuations." After checking all the pulldowns for the fiftieth time and finding no feature that proports to supply parameters to the layout routines, our user swallows nervously and decides to "crack the hood".

If his "car" is a recent model he is likely to be shocked. When one opens the hood of a modern car, it can be hard even to find the typical parts of an engine, such as the carbureter and distributor, let alone make any changes without an oscilloscope! What our user needs is more like a Model T, with points that can be set by hand with a dime.

And even if we can make the under the hood layout as simple as a Model T, it is very likely that most users will still not have enough short-term memory to make sense of it, unless it surprises them by looking just like a mechanism with which they are quite familier. But, it can't look like

BASIC, for example, because the program would be much too large and unwieldy. The other two tools that a novice would be likely to have experienced are a spreadsheet package and a graphics system. Current versions of these aren't powerful enough to help, but what if we cheated by designing our own and supplied them with the word processor as integrated tools?

This "everything is built from things you already know" approach is similar in spirit to Sketchpad of more than 20 years past, Smalltalk of more than 10 years past, and the recent Framework system of Ashton-Tate.

For its range of application, Sketchpad probably had the highest power to simplicity ratio. Smalltalk would be next, and then Framework. Smalltalk is the most general of these systems -- everything in it is an object, made from a class object, able to send and receive messages and to remember in terms of objects. For the users we are contemplating, though, Smalltalk permits too many styles of programming and has a bit too atomic view of the world.

Sketchpad has only one style, that of causing desired relationships between variables to happen. Unfortunately, it is all too easy to give a perfectly reasonable relationship that Sketchpad's solver will not be able to deal with or will take forever to settle down.

Framework, a 1984 release for micros, is one of the most interesting designs since VISICALC. It has more of the flavor of the old PARC Smalltalk than the other so-called integrated systems. Also, aficionados will see resemblances with the even older NLS hypertext system of Englebart (at SRI in the 1960's). The user interface is "Apple Modern" with multiple overlapping windows (Framework calls them "frames") and a row of pull down menus across the top. User interaction is also familiar with a "select first" then "invoke operation/message" style. (Needless to say, some of the joys of this style are completely lost on the mouseless IBM PC, which requires all the many modes and escapes that we thought were a thing of the past!)

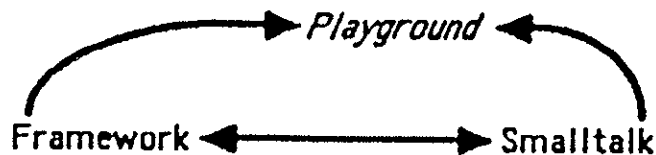
Frames can be nested indefinitely, but not all frames can contain other frames, and not all objects can be frames (integers, strings, etc., are second class citizens). A tree-like path name suffices to identify objects down to the lowest (character) level. Each frame has a function written in the VISICALC-like language FRED that is responsible for the image in the frame. Thus, a Framework spreadsheet is just a square array of frames; a Framework graph is just a blank frame whose FRED program invokes the @displaygraph function, with arguments that dip into other frames for values. All of these functions can be modified by the user and new kinds of simple frames can be built from scratch by the user. "Macros" gathered from user interactions can also be used to refer to and control elements.

Two other intangible but very important attributes of Framework are that all interactions are extremely fast, almost instant, and that the context-sensitive Help is the best (and fastest) I've seen.

On the other hand, there are many parts of the system that cannot be viewed -- the coordinates and dimensions of the frames, and the word processor, for example. Data-dependant control only works within a frame. Something special must be done by the user to get a graph whose values depend on a spreadsheet to re-eval when the spreadsheet changes.

In short, Framework is rather complicated for what it can do. In many ways there are more things to learn about it than for all of Smalltalk -- and Smalltalk could do so very much more. On the other hand, Framework on a Mac or LISA with a mouse should be an extremely attractive system since so many functional capabilities were done so well.

Our new system wants to be somewhere between Framework and Smalltalk but simpler -- more *buoyant*-- and more fun. Hence:



Approach

The users of *Playground* will get their first experience (as they do with LISA or Framework) by creating documents that are a combination of text, pictures, data-driven graphics, and information that has been retrieved, filtered, and calculated.

The three new tools, PlayWrite, PlayDraw, and PlayCalc, would be at the same scale of feature as their predecessors on the Mac. For example, PlayWrite, as presented to the novice, wouldn't have to handle multicolumn layouts. They differ in that they all work on the same document and can be nested within each other (as in Larry Tesler's old Galley Editor). For example, a PlayDraw or PlayCalc image is one of the paragraph objects of PlayWrite; a PlayWrite text paragraph can be one of the contents of a PlayDraw rectangle or PlayCalc cell.

The approach in getting users to tailor sophisticated software is to first get them to edit drawings, spreadsheet calculations, and text as they have done in the past. Then show them how spreadsheet tactics can be used to make custom controllable graphics objects. Finally, when they want to crack the hood, reveal that the word processor is just another data-driven graphics program -- similar to the ones they have been making but with a little more mechanism.

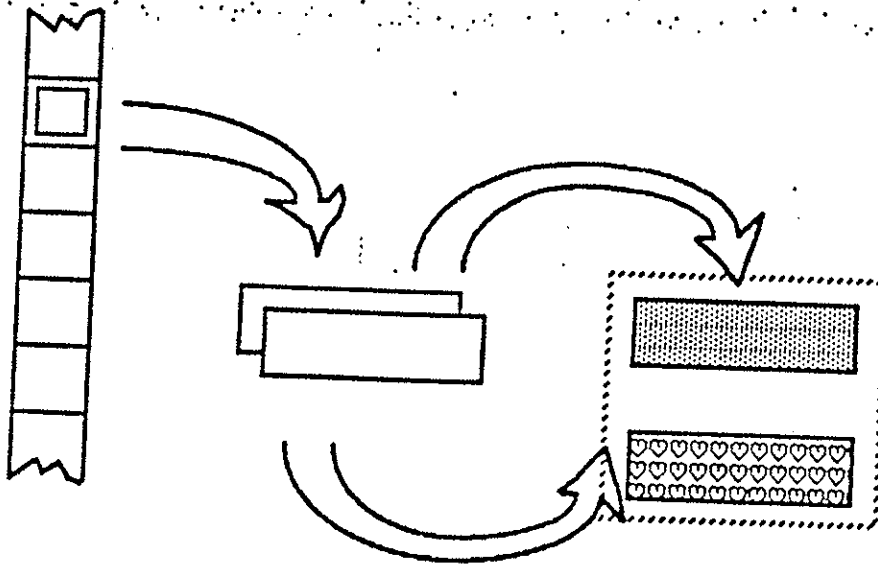


What follows is an attempt to uncover a simple but powerful way to think about objects and a *style* of user interaction that will lead to our goal. Syntax and formats may and will vary from example to example -- it's just too difficult to be consistent at this stage. I would expect, for example that the eventual syntax for code would wish to resemble the English "situation - action - consequence" layout found in good tool manuals. I have not addressed this issue at all at this point. Suggested contents for menus, pulldowns, or even the use of pulldowns for certain purposes have just been pulled out of thin

air. I have stayed strictly with the data-driven "retrieval style" programming that forces a local relation between a value and its rule.

Getting Started

It will not be too surprising to learn that the user's perception of rectangles, viewing, pointing, and help will be the key to this experiment. Accordingly, the introductory session should involve some tailoring even in the first few minutes. The screen will look a bit like ApplePaint/Draw. On the left will be a visible menu of "drawing symbols" with among other things a rectangle entry. As with AppleDraw, we get the user to make some changes to a sample drawing which include drawing rectangles, going to the fill menu to choose a fill pattern, moving things around, using Duplicate, and simple grouping.



Pointing into a group selects the most interior object that can be dealt with. This is often a *gap* between objects to permit insertions. Dragging and multiple clicks expand the selection to enclosing groups. Pointing at the edge of a group selects all it encloses. There is a *grid* that underlies the positioning of objects and helps line them up as "before-after" or "over-under" relationships. Insertion happens along one of these axes unless overridden. Features, as mentioned below, also "stick" along the grid.

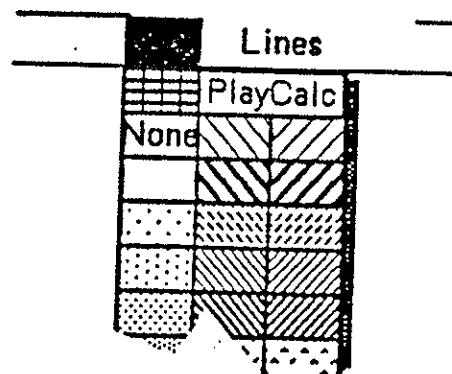
A rectangle can be typed into. The user will discover that the width of the rectangle will stay fixed causing the text to wrap around, but that

the height will stretch to accommodate the text. All the usual modeless editing features obtain including boldface, italics, etc. A pulldown called Appearance Rules has a large range of possibilities. Attributes are "sticky to the left" -- that is, if one inserts just to the right of a bolded font, or one that is smaller, etc., then the new characters will pick up the attributes to their immediate left.

A return will terminate the typing into the rectangle and will start another one lined up directly below. Features are "sticky up" -- the new rectangle will have all the features of the one above it, such as margins, justification, framing, and so forth. The two rectangles are grouped automatically.

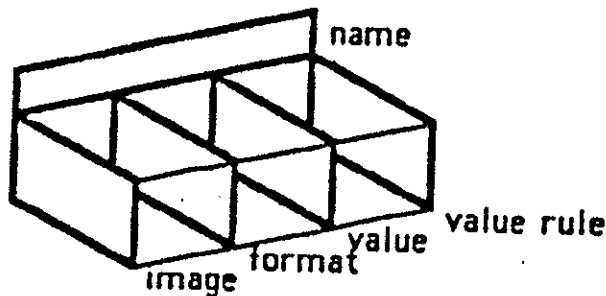
The user is next asked to draw a larger rectangle and consult the fill menu. We get the user to notice that one of the fill patterns is called PlayCalc and has the typical grid shaped icon.

	A	B	C	D
1				
2				
3				
4				
5				
6				
7				
8				
9				



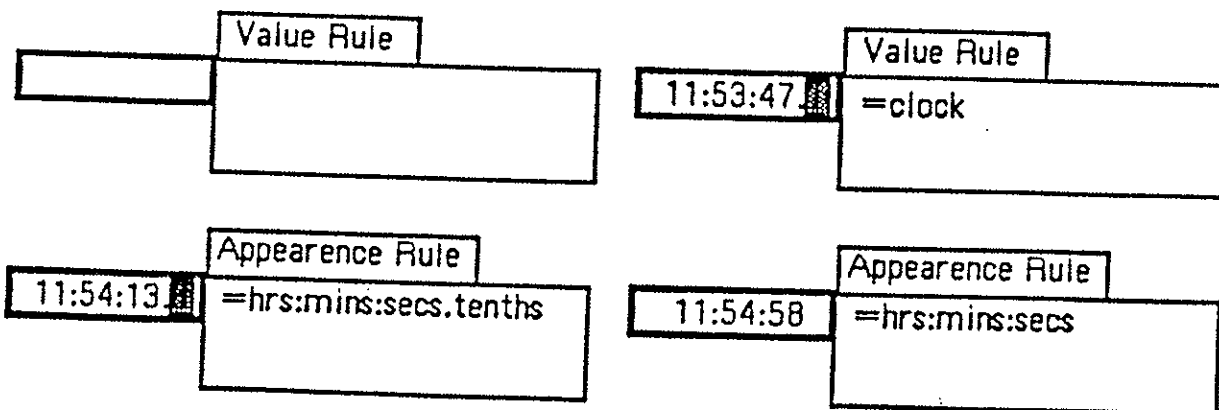
The rectangle is thus "filled" with a spreadsheet -- it becomes a *group* of spreadsheet cells. Pointing into the spreadsheet rectangle puts a selection down to the text level as it did with the text paragraph. We get the user to do several typical spreadsheet edits to solidify its perception as a familiar object. This would include putting in numbers in several

cells, a Value Rule to sum and subtract to get a total, and an Appearance Rule to cause the display to show two decimal places and a dollar sign.



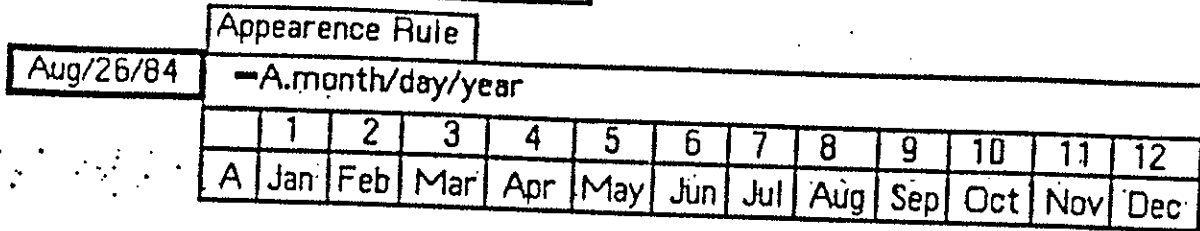
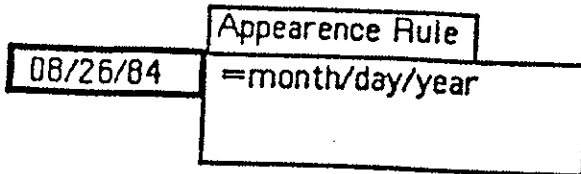
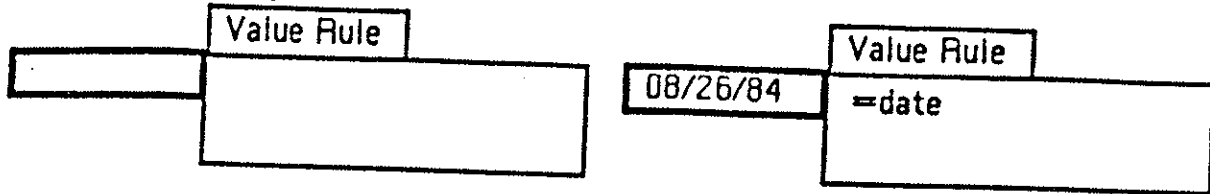
Spreadsheet-like Rectangular Cell

Next we show the user that an isolated rectangle can act like one of the spreadsheet cells. This is done in two ways. The Duplicate command that was previously encountered is applied to an individual cell in the just made spreadsheet. The newly created cell can be dragged out of the spreadsheet onto the paper where it can function all by itself. A good test for this would be to get the user to give its value rule a function to retrieve the system clock. Initially this would show an image resolved to (say) tenths of a second zipping along. The user would be encouraged to edit the appearance rule to only show the time down to the second.



The second way is to draw a rectangle and use the Fill command to paint a cell into it. A good value-rule for this one would be to retrieve the system date and change the appearance to show the months spelled out. The possibilities for nesting and hiding and the familiar use of ranges in spreadsheets will be seen in a new light. We note in passing that the

value rule of the rectangle is to be a cell, the *value rule* of the cell is to be a date.



Since the calendar is a rectangle and a text character is a rectangle and a text word is a rectangle, we might suspect that the newly made calendar could be inserted into a paragraph of text.

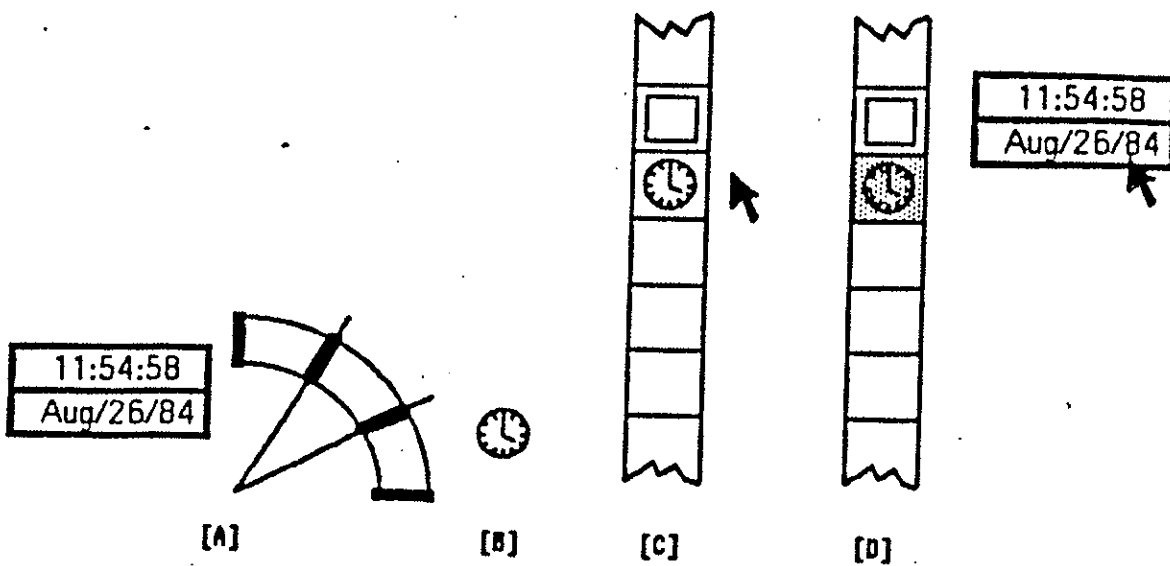
This is a test of using a rectangle like a paragraph. We see that wrap around extends the rectangle at the bottom but that the margins are kept. Different fonts bold face, *italic*, and so forth seem to work.

Oh good, a return makes a new thing that also works like a paragraph.

I can even have one that centers the text for me.

I wonder if I can paste in the date rectangle I just made Aug/26/84

Finally the two spreadsheet cells can be grouped.



The user has just made a desk accessory and tailored it. Now it would be nice to include this newly made tool in the menu of desktop resources. In [A] the user has decided to make a clock icon to stand for the time tool. One quadrant with guidelines is constructed. Duplication, flipping, and reduction complete the icon in [B]. The icon is included in the menu (which just turns out to be a spreadsheet in disguise!) in [C]. Finally, the new desk accessory is tested by mousing the icon which gives rise to the clock/calender in [D]. This style of interaction resembles more the current AppleDeskTop in which icons can be made, moved around, and invoked, than it does current AppleTools, most (all?) of which have fixed and uneditable menus of resources.

The user has now got the idea that every rectangle acts like a spreadsheet cell, but that there is considerably more flexibility available without additional complexity. In fact, the number of concepts that have to be learned ^{is} are far less than if a draw system and a spreadsheet system had to be approached as separate ideas. This way of teaching will work as long as (1) the total number of concepts doesn't get too large, and (2) the concepts remain concretely tied to the document creation metaphor.

Spreadsheets And Data Bases Are The Same

A *group* of spreadsheet cells, or anything else for that matter, is a *relation* of *fields*. Instances of a group act like similar records. A

relation of relations is what data base people call a *join*. Retrieval is usually accomplished by giving a cell a value-rule that is the name of the relation and some filter restrictions on values for certain cells that comprise it. The result is gathered into a group that can be used for further filtering. Thus every aggregate can be also be thought of as dynamically retrieving its members as opposed to a statically set up grouping. The user only has to remember one idea: every place there is something is only because there is a value-rule that has retrieved it there.

For example, electronic mail is a grouping of objects whose value-rule is:

=self joined-to	mail	date	name	...
	new	alan		

In other words, the group is whatever was there plus anything new that turns up with the user's name in the name field. Most users would prefer to receive more than just what is specifically addressed to them:

=self joined-to	mail	date	name	...	message
		new	*alan		*alan@Mac

That is, add new objects that either have the user's name in the name field, or mention the user or Macintosh in the public part of the message.

A simple calender can be set up by scheduling a retrieval when some condition is met in the future:

=if Date=Sept 6, 1984 and Time= 2:45 then	flash	You have 15 minutes to get to MacFutures Seminar!
--	-------	---

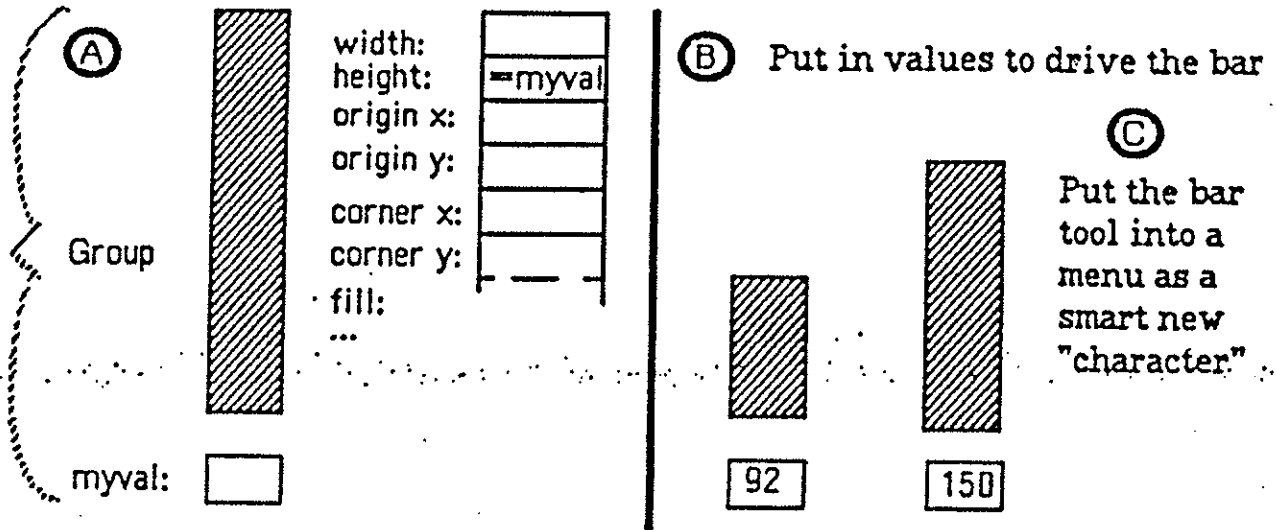
Of course, one could imagine that a real calender entry would actually generate the value-rule for this cell from having been given the knowledge that the meeting is at 3:00 each Thursday.

Rectangles Are A Spreadsheet

Most of the time the user will just use the existing tools -- and be quite happy in the bargain with the way they all work on the same document.

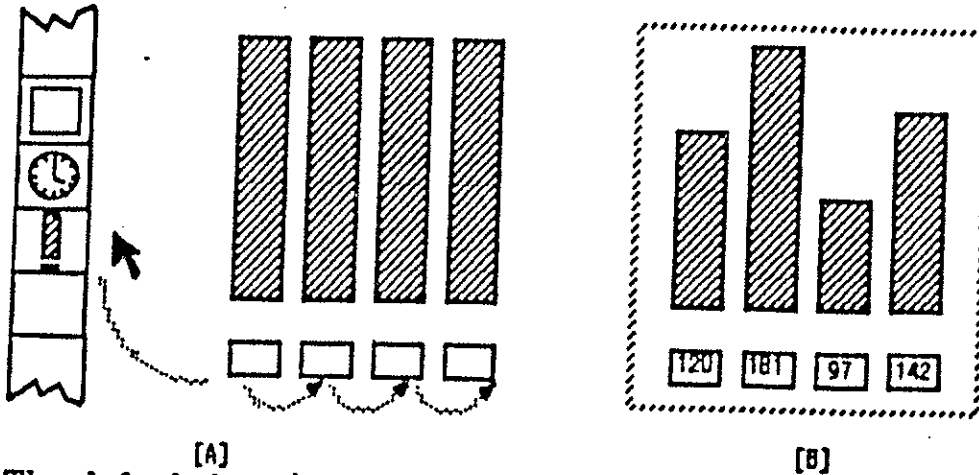
supplied tools as though they had made them, is a bar-chart tool. The only new information needed to get started is that the attributes of any object can be displayed. They show up as a spreadsheet also making it easy to add new ways to determine values.

Stage One: Making the height of a bar depend on a value



Here we take a vanilla rectangle of PlayDraw and combine it with a PlayCalc cell which we name "myval". All of the "properties" of the rectangle are data driven so a simple bar can be constructed just by making the rectangle's height depend on myval. We note in passing that there must have been (are) other value rules for driving the height of a rectangle -- those that are obtained from the size changing "birdies" for example. Just how alternatives are specified without requiring the user to make "dangerous" edits to already working code will be discussed later. For now, we can think of every user edit as a kind of subclassing: "I want something just like that, except...".

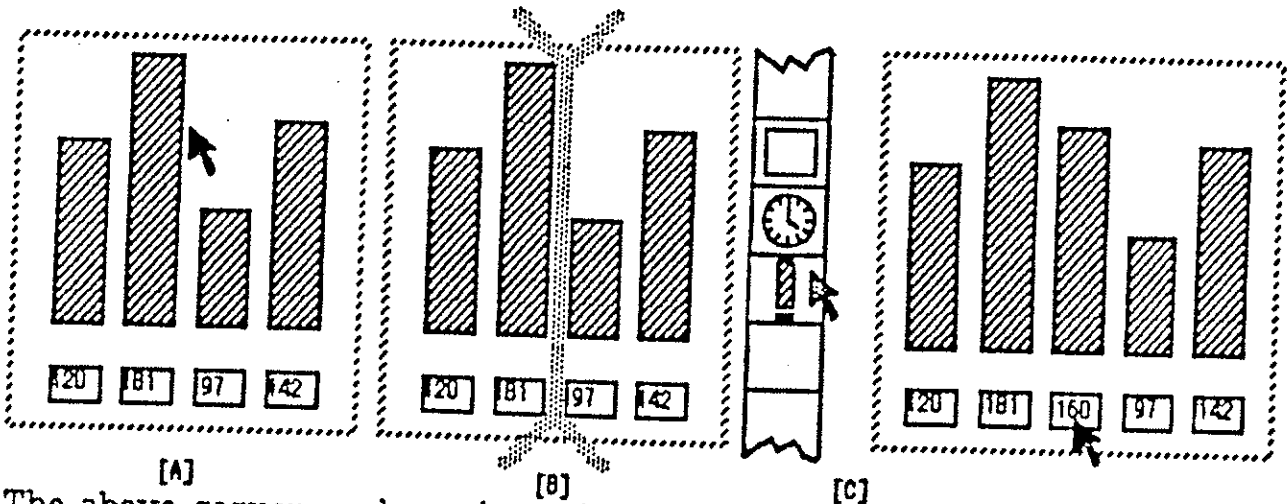
Stage Two: Making a bar chart with a predetermined number of bars



we could think of every box as of "class size" and then implicit grouping is along line and P. This could be made possible to choose another container type as a default.

The default location for new items to appear is lined up to the right as though they were text characters. In [A], the menu button has been pushed four times to get the lineup of four bars. The implicit grouping and sticky to the left characteristics previously discovered for text also obtain here. Pointing into a value box permits a number to be entered that drives the height of the bars. If one points between any two objects, a text-like cursor shows where the seam is. This brings us to:

Stage Three: Editing new bars into a group of bars



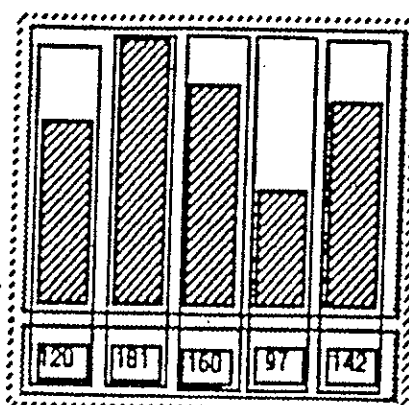
The above sequence shows inserting a new object into a sequence of objects. The two parts of [C] show: (1) a new bar being gotten from the menu to go where the cursor has been positioned in [B], and (2) the value cell being edited to set the new bar to its height. It is clear that the cursor object must be interacting with the before-after relations in the

*an was ment
is would
the because
really is a
of clear*

bar sequence group and the grid, to cause room to be made for the new bar rather than have the new bar just plop on top of the old ones.

The next stage is to examine the group we have just made. From the recent experience with the single bar, we can expect that the attributes will be some kind of spreadsheet.

Stage Four: Examining the attributes of a group



Value Rules	1	2	3	4	5
width:					
height:	=myval.self	=myval.self	=myval.self	=myval.self	=myval.self
origin x:					
origin y:					
corner x:					
corner y:					
fill:					
...					
myval	120	181	160	97	142

would work for clear too

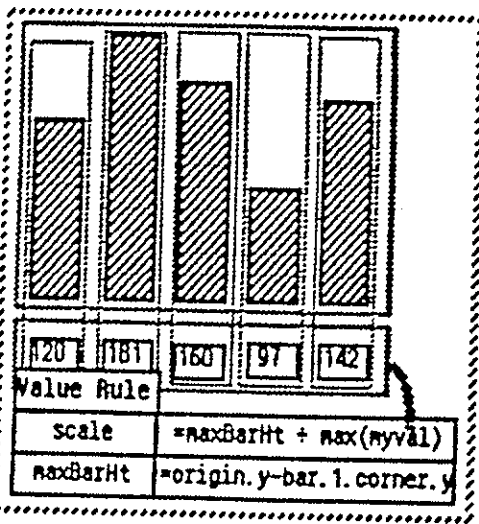
The idea is that the group unifies the composite sheet along shared attributes. Note if the sheet were transposed, it would resemble a relational database -- as indeed it is. In fact it is really a kind of join between the myval rectangles and the patterned rectangles. The greyed rectangles show different kinds of groupings/joins that can be discovered. The sideways ones have to do with matching properties or types, the vertical ones happen because we forced the initial grouping between the two rectangles to make the first bar.

It's clear that many aspects of this need better thinking. For example, I really should have shown the other attributes of the myval rectangles (origin, corner, etc.). What is really shown is just the myval fill rule. These are familiar self-description problems to Smalltalkers. They are called *role* problems and they have to do with the difference between the character on the stage and the actor who is wearing the costume (e.g. they are likely to have different names and ages, and perhaps even different gender!). Please help work this out. Until then, the house of cards still stands, and I shall continue.

We have gotten this far simply by combining two rectangles (one for the bar and the other to hold the value); the current barchart was produced almost automatically by duplication and sticky grouping. To get the bars to normalize and scale, we must add two more spreadsheet cells to the bar group. Note that this brings to attention another collection of things missing in the group spreadsheet (for version two of this paper, of course!). The missing items are group attributes for at least the major enclosing group. What we actually showed should be subgroupings and delineated that way in the composite chart.

Going back to the picture, we widen the enclosing rectangle to make room for our new cells, religiously certain that the grouper will do the right thing.

Stage Five: Getting the Bar Graph to scale and normalize



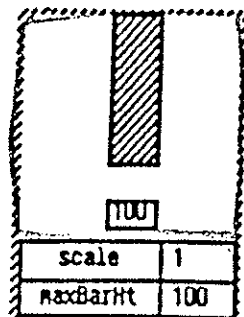
Value Rules	1	2	...
Height	=myval.self*scale	=myval.self*scale	etc.

*we are
or don't
they that
is in P
ready parents
& labels
we are (has just made)
no contour,
with
l).*

Most of the symbols should not have to be remembered and typed. Pointing ala spreadsheets should supply the names: myval (pointing at the entire group), origin (pointing at the enclosing group's upper left hand corner), bar.1.corner (pointing at a bottom corner of bar 1), etc. It's possible that, in certain cases, it would be better to do away with symbols altogether and just use graphics links (an example between myval and scale).

The sticky features of groups allow this specialized bar chart to be reduced to a "proto Bar Chart" of just one bar that can be entered into the

menu as a general tool. Here's what it would look like after the Attribute View has been turned off.



Of course, the group level cells could also be suppressed for this "costume-level" view. More questions arise. It is clear that the replicable part of this tool is the *bar*, not the scale and *maxBarHt*. We don't want more of them when we concatenate bars. This is reminiscent of the distinction in spreadsheets between absolute and relative references. Also, this has a bit of an unholy "vitalist" flavor to it, in that the atom carries the "life force" of the cell. But perhaps this is a quibble.

One could just as easily think of the singleton as a cell with all the genetic information for the "tissue". *or that, ala APL, the singleton is actually (also) a vector of one element*

*between
will
be
about*

*not
clear*

Exercises Left To The Reader

Spreadsheets are quite weak when dealing with groups. Languages like *APL* and *LISP* are quite strong. The tool that has just been made will accept bars as though they were characters. Bars can be rearranged and removed but the ones that remain will always normalize to the size of the enclosing group. There are several interesting things still to be done.

We should be able to link an external spreadsheet with the myvals. This amounts to conforming one array with another. *APL* automatically adjusts to the larger of the two. We would hope that the myval group (which starts off as a singleton) would do the same.



Finally, we need to have some way to have two-way retrievals without having to invoke a flimsy constraint mechanism. The bar allows us to grab the filled rectangle and drag it, but its height will instantly snap back to the value in myval. If the rules in myval were something like:

$$A \leftarrow 3 * B$$

*! is like a
constant*

Analogy



M ... 029244

- if bar.self changes then bar.self.height
- if haveMouse then Getnumber
- if neither then oldVal



(with a better syntax) then the right thing would happen. The first rule would be fired off by a change in bar.self.height which would make the conditional part redundant and it could be eliminated. If the old value is compared against the new (as in some spreadsheets) and they are the same, then the circularity could be terminated. ~~Go for it!~~

The Model T

Now we get to heart of this gripping story: "It is a stormy night in 1902. From an overturned buggy in a ditch a drenched Henry Ford stares angrily at his horse, then lifts his gaze musingly to the heavens . . ." Well, actually -- it would be too much to expect that most of the users who easily mastered the tools of Playground would have the interest, design ability, and perseverance to built their own word processor. We, after all, have given them a fine one that already works. Yet there will be plenty of features they will want that we didn't anticipate. Now is the time to see if they will be able to change and add to *Playground's* tools.

First, I want to sketch how the rare user would build a word processor from the materials supplied and the knowledge gained from making things in *Playground*. Even though most won't, it is this structuring, analogous to making clocks, bar charts, and simulators, that will make up the fruitful user illusion. In other words, no matter what Ferrari-like code we really have running down there, we'll cover up that 12 cylinder double-down-draft engine with the apparel of a Model T when the user opens the hood.

To a first approximation, text characters are just rectangles that enclose small funny shaped pictures. And, in fact, it doesn't take long with a reasonable drawing system to make some.

$$\text{height} = \text{self.nyval} * \text{scale}$$

undefined & does fix any val of height

~~height =~~

$$\text{height} = \text{base} + (\text{self.nyval} * \text{scale})$$

$$\text{nyval} = \frac{\text{if have mouse then (mouse.y - origin)} \div \text{scale}}$$

$$\text{height} = \text{if have mouse then mouse.y}$$

mouse
ords



We can well imagine that the spreadsheet calculation for width might include something like: $\text{Width} = \text{if bold then stdWidth} * 1.2$, etc. Of course, size, boldness, slantness, italicness, etc., aren't really affine transforms. But to the "first approximation" that we are working to, affine transforms work quite well.

Each of the characters would be transferred to an extension of the tools menu. We already saw that the tools menu was overlapped with the keyboard. Now we can stick our own characters into the keyboard map. The "align to the right" and "sticky to the left" properties of the graphics system will get characters to be in a line as they are typed. A boldface attribute, for example, will be merged from one character to the next.

Unfinished Stuff Ahead

This is a sketch of how the word processor has been structured by the applications programmer.

A *galley* is a rectangle that has a top, left and right margins, and contains a gridding made up of *lines*

A *paragraph* is a chain of rectangle groupings. The rectangles are character-boxes containing nonspaces separated by rectangles containing spaces. Insertions into this structure preserve the chain because character boxes look sideways to coalesce themselves into the proper non-space-box or space-box. One of the attributes of these larger boxes is width, which is obtained as a spreadsheet-like sum of the characters that have declared themselves to be members of the box.

A *textis* a chain of paragraphs.

A *documentis* a text laid into a galley.

We get an idea of why this is simple enough to be understood by looking at a non-space-box's rule for position. Its top left vertical (origin *y*) value is the same as its before box in the chain if it can also fit within the right margin, otherwise, it has to take its origin *y* value from the *line* below the one its predecessor is in. The horizontal (origin *x*) value is either the left margin or takes its value from the before box. That's it. In computerese:

fit = before.corner.x + width ≤ right margin
 origin y = if fit then before.line.y else before.line.below.y
 origin x = if fit then before.x else left margin

Notice that referring to before ^{corner} forces evaluation back to the left through the paragraph until a word is found that has no before. It would have a rule that got its *x* and *y* from paragraph level info such as the paragraph owned indent value plus left margin. Then values would propagate along the chain to the right as words found their place in the right order.

Also, we haven't told the line objects to do anything. They capture the word objects by retrieving the word-boxes that have *y* values that match their own.

*holog
 hick
 how to combine
 with
 dynamic
 programming
 constraints*

$\text{nonspaces} = \text{"}.origin.y = y \text{ and " is nonspace}$
 $\text{totalspaces} = \text{"}.origin.y = y \text{ and " is space}$
 $\text{lastspace} = \text{" in spaces and "}.after.y \neq y$
 $\text{justifyspaces} = \text{spaces} - \text{lastspace}$
 $\text{spaceleft} = (\text{right margin} - \text{left margin}) - \text{sum}(\text{nonspaces.width})$

The lastspace cell finds (if there is one) the space box that dangles at the end of the line. The justifyspaces box contains the spaces whose widths must be adjusted if right justification is desired.

$\text{spacewidth} = \text{spaceleft} \div \text{count}(\text{justifyspaces})$

Each spacebox will have a width rule:

$\text{width} = \text{if justify then line.spacewidth else default}$

Notice that we have a race condition. We can't get spacewidth until we've settled the question of which boxes are in the line. That has to be done with default widths even though the justify flag may be on. The spacewidth rule should actually default until it can evaluate the count. It is clear that this is not at all serious and easily fixed.

The scheme of the editor is now clear. Editing is done by inserting between before-after relations of character boxes, as with any graphic edit. The character boxes group themselves by common properties into space-boxes and nonspace-boxes. The boxes find where they are in the galley by asking backwards until an anchor is found, then streaming forward trying to place themselves after the box before them if they can fit within the margin, otherwise going to the left margin of the next line. The lines in the galley are grouped above-below and contain the allowable vertical grid values. They sum up the white space left to be used by the space boxes to determine width if justification is turned on.

Getting The Space After A Period To Be Bigger

We notice that the application programmer has not anticipated that a user might wish to change the allocation of white space rules to get more

after a period. The lines do the calculation for total white space and for an equal distribution of it to the space-boxes.

The user actually will have to think about what to do. On his side, the editing scheme is quite familiar. There are only 10 rules to scrutinize, they are quite clear in their intent, and there can be no others that directly change the values in question.

But there are still many choices. And it is just in this area of design and choice that the user is weak. Should he try to get the characters to group in yet another way? Should he get the line object to find the period/space combinations and calculate for them separately? Right now all the spaces in a line are going after the spacewidth value concurrently. How can the ones after the period be headed off?

First, our user would soon note that spaceleft is the critical resource. If the number of spaces in a line are 12, and two of them have periods or commas, then what needs to be calculated is:

$$\text{spacewidth} = \text{spaceleft} \div (\text{count}(\text{justifyspaces}) + \text{count}(\text{specialspaces}))$$

The specialspaces then must know to take up twice spacewidth. So we need to add a way to find specialspaces. Since a space has to know in order to get the correct value from a line, we'll let it figure it out.

specialspace = before contains "." or ~~"!~~" or "?" or ";" or " ;"
width = if specialspace then 2*spacewidth else spacewidth

The line object just has to extract specialspaces from justifyspaces.

specialspaces = * in justifyspaces and * is specialspace

Again, this is not quite right because it ignores the case in which only the default width would be small enough to get the words on the line. The calculation we are making would reduce the width available on the vanilla spaces to less than they are allowed. The prudent user would turn off his feature in that case. The curious user would use a different

default width. The industrious user would either iterate, or examine the initial line membership relations further.

In any case, there is no question that this is easier than recursive return on investment models!

--==!!==--

References

Zloof The "by example" papers

Ingalls Smalltalk-76

A Smalltalk Operating Plan

Tesler The Smalltalk User Interface

Galley Editor

Kay The Two Novice Programming Papers

Computer Software (Sci Am.)

Smallthoughts about Smalltalk

Data As A Base

Atkinson Roladex

Ashton-Tate Framework

Sutherland Sketchpad

Bricklin Infotech paper on visicalc

PROLOG

What Else?