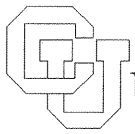
NoPumpG: Creating Interactive Grahpics With Spreadsheet Machinery

Clayton Lewis

CU-CS-372-87



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* Supported by NSF NYI #CCR-9357740, ONR #N00014-96-1-0720, and a Packard Fellowship in Science and Engineering from the David and Lucile Packard Foundation.

NoPumpG: Creating Interactive Grahpics with Spreadsheet Machinery

Clayton Lewis

CS-CU-372-87 August 1987

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

This research was supported by the Air Force Human Resources Laboratory and the Institute of Cognitive Science.

NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery

Clayton Lewis
Department of Computer Science and
Institute of Cognitive Science
Campus Box 430
University of Colorado
Boulder CO 80309
(303) 492 6657

ABSTRACT.

The spreadsheet has made computing power widely accessible to nonprogrammers. By adding a small number of new concepts to the basic spreadsheet framework NoPumpG makes it possible to create interactive graphics, including animation, of the sort seen in physics and geometry tutorial demonstrations, without programming. While not as powerful as some other systems of this kind, NoPumpG appears to offer a favorable balance between power and conceptual simplicity.

ACKNOWLEDGEMENTS.

My ideas about spreadsheets, and the desirability of extending them, were strongly influenced by Robert Balzer, Gerhard Fischer, Thomas Green, Donald Norman, and Gary Olson, who participated in a workshop on new approaches to programming in Boulder in 1986. The discussion here owes much to their insights. Andreas Lemke, David Kieras, Jakob Nielsen and Beth Richards have made useful suggestions about the work. I thank the Air Force Human Resources Laboratory and the Institute of Cognitive Science for financial support.

INTRODUCTION.

Programming is difficult to learn, slow, and effortful. Many people who need and want computer support beyond that provided by canned applications are unwilling or unable to invest in programming skills. Those who do program find progress slow and many potentially attractive applications are never built. Where can we look for ways to provide the flexible control of computing that programming provides, at much less cost?

The participants in a recent workshop on new approaches to programming (Lewis and Olson, in press) cited the spreadsheet as a success case worthy of attention: spreadsheets are given major credit for the microcomputer revolution, by permitting large numbers of nonprogrammers (indeed non-computer-users) to develop and use their own customized applications. Horowitz and Munson (1984), writing for software engineers, note that the spreadsheet has brought many new users to computers and produced a large productivity gain.

The success of the spreadsheet has brought with it a number of extensions. Graphical

output from spreadsheets, in the form of business graphics, is common. Extensions to provide access to database functions are provided in a number of systems. The ASP system (Piersol 1986) is a spreadsheet which can operate on any Smalltalk data object, including bit images. Van Emden, Ohki, and Takeuchi (1986) describe a spreadsheet interface to logic programming. Lewis (1985) describes a spreadsheet capable of manipulating approximate quantities and relationships.

The present paper describes extensions designed to permit a user to create, control, and manipulate graphical objects using a spreadsheet framework. It aims to capitalize on the empirically-proven success of the basic spreadsheet concept to bring the creation of interactive graphics within the reach of nonprogrammers, and to reduce the time and effort required to produce interactive graphics in traditional programming approaches.

The design of these extensions has been guided by the following principles, intended to reduce the conceptual overhead in using the resulting tool, while maximizing its flexibility.

Add as little as possible to the basic spreadsheet paradigm. The market shows that people can understand and use spreadsheets.

Provide low-level primitives from which larger structures can be built. Graphical toolkits, such as Pinball Construction Set (Budge 1983), provide easy access only to specialized graphical parts which can be used only (in this case) to build pinball machines.

Strive to support a wide range of sample applications. The examples presented below are drawn from a number of different domains, including geometry and physics. Other possible applications, such as a simple flight simulator, have influenced the design. Features have been added to the design when needed to support a sample application, provided the features appear to be of general value and to be easy to understand.

Don't try to do everything. The original spreadsheet represents a successful compromise between simplicity and power, delivering much less power than traditional programming but with much greater simplicity. In Gerhard Fischer's phrase, the "subjective computability" of spreadsheets is often greater than that of a traditional programming language, even if its "objective computability" is limited, because people can understand what they are doing. In extending the spreadsheet into new domains some problems may prove resistant to any sufficiently simple approach, and should not be attempted.

The extensions are embodied in a prototype system, called NoPumpG. While the interactions supported by the prototype are quite general in character, the practical applications to which they are most suited are tutorial demonstrations that permit students to view and manipulate physical or geometrical systems. A number of further extensions are contemplated that could produce a tool with other

applications, such as rapid development of user interfaces.

The presentation here will proceed as follows. First I describe the essential features of the spreadsheet idea that provides the foundation for NoPumpG, and propose an analysis of what makes the spreadsheet idea work. In extending the spreadsheet most features, though not all, that figure in this analysis are preserved. Against this background I describe the single most important of the spreadsheet extensions in NoPumpG: the linkage between spreadsheet cells and graphical entities. I next describe the NoPumpG prototype which implements this key extension and a number of ancillary extensions. Each extension is then illustrated by examples constructed using the prototype. I then compare NoPumpG with other systems with similar capabilities, and consider limitations of and possible enhancements to the design.

WHAT ARE SPREADSHEETS, AND WHY ARE THEY GOOD?

For our purposes a spreadsheet is a collection of cells, each of which can contain a value and a formula. A formula refers to one or more other cells, and specifies how to compute a value using the values of those cells as arguments. This computed value becomes the value of the cell containing the formula. The value of a cell that has no formula can be entered or edited by the user. When the value of any cell changes, the values of any cells containing formulae referring to that cell are recomputed. The values of all cells that have values are always displayed.

This simple scheme has a number of profound advantages. First, the two-dimensional layout normally used for cells presents a familiar, concrete image which can easily be related to paper spreadsheets that are in common use in many tasks. This advantage is not exploited in the NoPumpG design, because paper spreadsheets are not used in controlling interactive graphics.

The automatic propagation of changes permits the user to state dependencies among quantities without being concerned with the manner in which these dependencies are enforced. The user does not have to detect when changes occur, decide what to do about them, or specify the order in which actions in response to changes will be taken, all of which are essential concerns in conventional programming.

Since the spreadsheet user does not need to specify how changes are processed, many of the concepts and mechanisms of procedural programming languages are not needed. Users do not have to use and understand flow of control, parameter passing, recursion, and similar difficult notions.

The fact that values are always visible (or can be made visible by simple operations like scrolling), and that values entered by the user can be modified by the user at any time, eliminates the need to pump data back and forth across an opaque barrier separating user from program. Consider a program in a traditional language that adds a series of numbers. Specifying the core dependencies between result and data is quite easy, but much more must be written: input code is needed to pump the data

in, and output code is needed to pump the result out. Once data are in they are inaccessible: if you want to be able to edit your input numbers without reentering all of them you will need a more complex program than a casual user is likely to write.

None of this pumping code, to permit you to enter your numbers, see the result, and edit the numbers if desired, is needed in a spreadsheet. The spreadsheet implements a form of inter-referential i/o (Draper 1986) in which objects are shared between user and system and can be seen and operated on by both. Avoidance of pumping code is a key objective in NoPumpG, and contributes most of the name of the system (the 'G' is for 'Graphics'.)

Direct visibility and modifiability of data, combined with automatic updating, enable spreadsheets to provide excellent immediate feedback to users on the effects of changes to data or to formulae. In conventional programming reexecution of a program, after a change to program or data, is often slow. Even in interpreted systems overt user actions are required to make the reexecution happen. NoPumpG exploits and extends the spreadsheet's strength in this regard to provide fast, continuous, graphical feedback for user actions.

Thomas Green uses the term 'viscosity' to describe the difficulty or ease with which one part of a large structure can be modified. In conventional programming viscosity is often high: changing the kind of value a variable holds in one place requires changes in other places, including perhaps its declaration; changing a loop heading requires changes to the body (if an index variable is eliminated, for example); adding a new procedure may interfere with access to existing procedures (in a block structured language), and so forth. Spreadsheets have low viscosity.

A final advantage of spreadsheets over most conventional programming approaches is in the treatment of aggregate operations like 'sum' or 'average'. In most procedural languages since Fortran such operations must be described by the programmer as a sequence of scalar operations, such as adding an individual datum to a running total. This translation is unintuitive for novices and wastes time for experienced users. In spreadsheets such operations can be specified directly as applying to a group of cells. Such aggregate operations are not included in NoPumpG; they have not been needed in the applications considered so far.

EXTENDING THE SPREADSHEET TO CONTROL GRAPHICS.

How can the characteristic strengths of the spreadsheet be extended to specify and control interactive graphics? The key innovation is a bidirectional linkage between certain cells in the spreadsheet, called control cells, and graphical entities. Consider a line segment as an example of a graphical object. The four coordinates specifying its end points are held in four control cells that are created when the line is created. If the value of any of these cells is changed, the line is automatically redrawn accordingly. If the line is moved, by a mouse movement, the values of the corresponding control cells will be updated automatically to reflect the new coordinates.

This picture is oversimplified in one respect. If a formula is placed in a control cell, it is evaluated as usual in a spreadsheet, and the line is redrawn as just decribed. But if an attempt is now made to move the line using the mouse, the coordinate corresponding to this control cell will not change, since it is determined by its formula, and the line will only move in such a way as to leave that coordinate unchanged.

This simple extension to the spreadsheet paradigm produces a simple but expressive tool for building graphical interactions, as illustrated below.

The NoPumpG prototype.

This extension, and a few ancillary extensions described below, have been implemented in a prototype system for the Apple Macintosh using Turbo Pascal. This prototype was used to produce all of the examples presented below. The prototype uses a generally Macintosh-like interaction style to permit the user to carry out the following operations.

Create and modify cells. Examples of cells can be seen in Figure 1. Each cell has a name, an optional formula, and a value. When a cell is created the user provides a name. Its initial value is zero and it contains no formula. The user can enter a new value by clicking on the old value, provided the cell contains no formula, and typing a new value. Formulae are entered by clicking on the formula portion of a cell, selecting an operator from a pull-down menu, and clicking on one or more other cells to act as the operands for this operator. A variety of arithmetic and trigonometric operators are provided, together with some special operators described below.

Since cells can quickly clutter the screen a simple means is provided to conceal and redisplay cells. Cells and graphical objects coexist in the same screen area, so the user is permitted drag a cell to any desired location on the screen. Thus unlike cells in typical spreadsheets, NoPumpG cells are not locked into a grid arrangement.

Create graphical objects. The prototype provides line segments, small bitmaps, called 'sketches', created using a simple bitmap editor, and fixed text strings. When any of these is created control cells are created for it and automatically displayed. These cells can be modified in the same manner as user-created cells. A line comes with four control cells, one for each coordinate of each end. Sketches and text strings come with two control cells, controlling the horizontal and vertical position of the object. Additional control cells for these objects that determine their visibility are described later.

Move graphical objects. The ends of a line, the middle of a sketch, and the beginning of a text string are mouse sensitive. Placing the mouse cursor on any of these points, and holding the mouse button down while moving the mouse, will send a request to the object to move to follow the mouse. This request will be honored to the extent

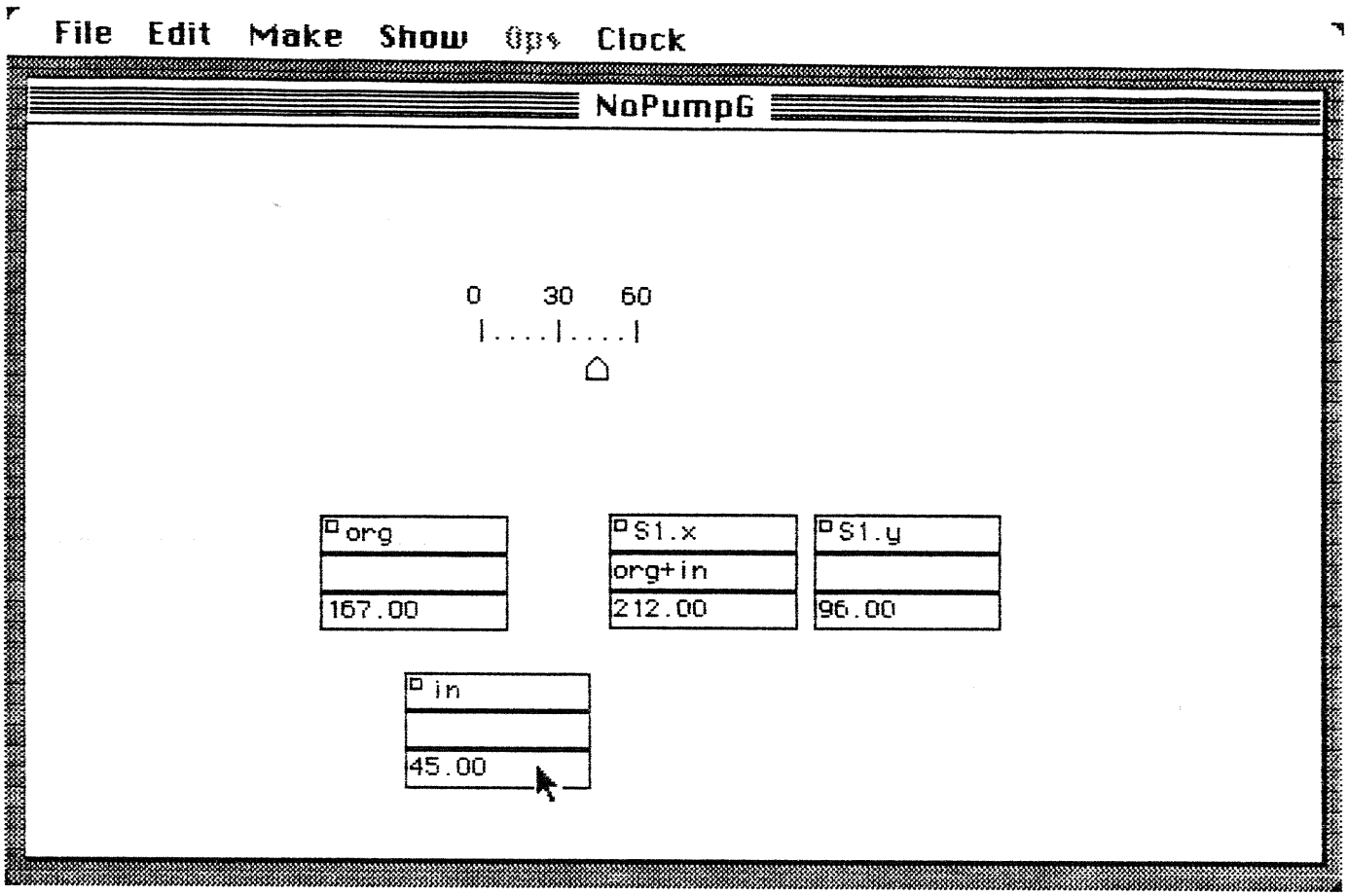


Figure 1: A simple output device. The pointer moves horizontally to a position determined by the cell IN. Normally these cells would not be left exposed to view.

that it does not conflict with formulae placed in the control cells for the point. If a point has a formula in its X control cell, but none in its Y control cell, it moves vertically but not horizontally when dragged. If an object is moved its control cells are automatically updated.

Additional features of the prototype permit the user to delete objects, control the visibility of graphical objects, and export and import objects. Where relevant these additional features will be described below.

The implementation of the prototype follows a message-passing discipline, though it is not written in an object-oriented language. Graphical objects respond to mouse actions by sending update requests to their control cells. A cell will update itself if it does not contain a formula. When any cell changes its value it notifies any other cells or graphical objects that depend on it; these objects then reevaluate themselves. Thus a line or other graphical object redraws itself when notified by any of its control cells. A cell containing a formula evaluates the formula and replaces its old value with the new result when notified by any of the cells that are arguments in its formula.

Even though no special effort has been devoted to tuning the implementation, this simple computational mechanism produces quite good performance on the Macintosh. Responses to user actions are immediate, though sometimes jerky for more complex structures. Animation, obtained as described below, varies in smoothness with the number of moving or changing objects and the speed of their motions. A simulation of an orbiting satellite, which permits the user to vary the initial position and velocity of the satellite, produces an adequate depiction of the motion as long as the speed of the satellite is not too great, and not too many cells that must be updated as the satellite moves, such as those holding the X and Y components of gravitational force, are kept visible on the screen.

Graphical output.

Figure 1 shows how graphical output from the spreadsheet can be accomplished in NoPumpG. The pointer is a sketch, and the cells S1.X and S1.Y contain, and control, its position. Since it is intended that the pointer move only horizontally the cell S1.Y contains no formula (to protect it from inadvertent dragging it could be given the dummy formula 'const'.) S1.X, which controls the horizontal position of the pointer, is given a formula which makes it the sum of the cell IN, where the value to be displayed is placed, and the cell ORG, which sets the origin of the pointer's motion. Thus if IN is zero the pointer will appear at X coordinate ORG. The tick marks and scale are fixed text strings dragged into position; the control cells for them have been concealed. The value of IN could be modified by editing, or by placing a formula in IN that calculates a value in terms of any other cells in the spreadsheet. Whatever the value of IN is, the pointer will appear in the corresponding position, because of the linkage between the control cells and the sketch.

In this and most of the other examples to be shown, cells are left exposed on the screen so as to show some of the formulae used. In practice, only cells whose values are of interest to the user, or for which the user might wish to supply new values, would be left visible. Thus in most applications of an output device like this one none of these cells would be seen.

Figure 2 shows a more elaborate output device. The mercury is formed from three lines tied together by formulae in their control cells. The cells L2.Y2 and L3.Y2 are shown: the formula in the latter indicates that its value will always be just a copy of the value of L2.Y2. Thus as line L2, forming the top of the mercury, moves up, the upper end of line L3, forming the right side of the mercury, moves up with it. The formula in L2.Y2 is similar to that used to compute the position of the pointer in Figure 1. As the cell IN is changed the value of L2.Y2, which controls the position of the top of the mercury, changes accordingly (the Y coordinate increases down the screen.) Other formulae in cells not shown keep the pieces of the mercury tied together as required.

The treatment of the parts of the mercury illustrates another effect of the linkage between control cells and graphical objects: it is possible to set up constraints between objects. By placing formulae in control cells that refer to other control cells, graphical objects can be attached to one another, as shown here, or constrained to move in concert. Parts of the same object can be constrained in the same manner. The sides of the mercury are kept vertical by making the X coordinate of one end equal to the X coordinate of the other.

The same principles can be used to produce other kinds of output devices. A needle dial can be constructed by fixing one end of a line segment and computing the coordinates of the other end as trigonometric functions of the input cell.

Graphical input.

Since control cells and graphical objects are linked bidirectionally, graphical input to the spreadsheet can be contrived as easily as output. Figure 3 shows the indicator of Figure 1 modified to act as a slide controller. Since cell S1.X now has no formula in it the pointer can be dragged horizontally. To keep the pointer from straying vertically, cell S1.Y is given the dummy formula 'const'. The formula in the cell OUT dictates that it will contain as its value the distance between the pointer and ORG, wherever the pointer is dragged.

Coordinated input and output.

Figure 4 shows an example drawn from Borning (1981). The outer lines in the first panel form an arbitrary quadrilateral, while the inner lines connect the midpoints of adjacent sides. The inner lines always form a parallelogram, not matter how the quadrilateral is distorted. The second panel shows that this is so even when the quadrilateral is made concave; this display shows the result of dragging one of the vertices of the figure into a new position.

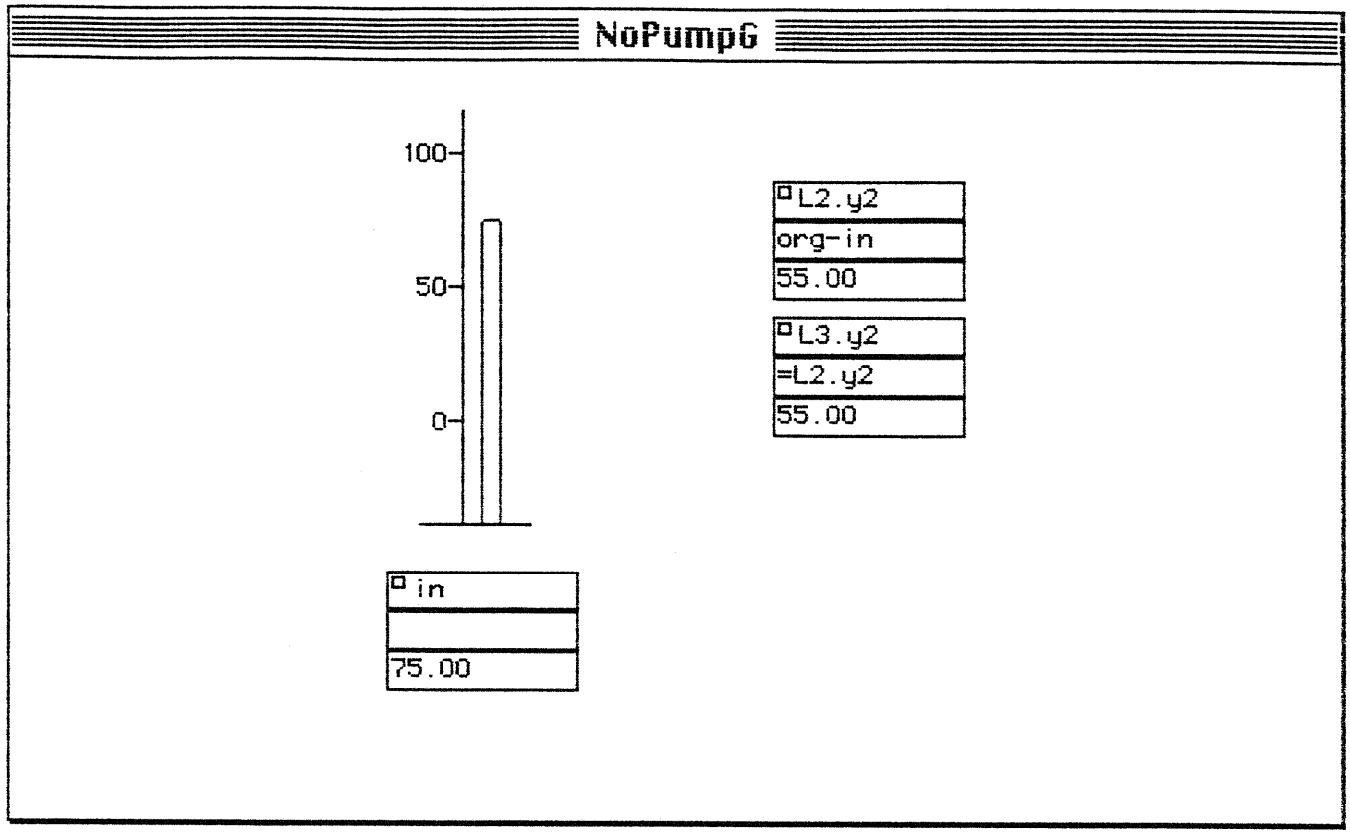


Figure 2: A more elaborate output device. The cells shown on the right show how the height of the line forming the top of the mercury, L2, depends on the value in IN, and how the top of line L3, forming the right side of the mercury, is tied to L2.

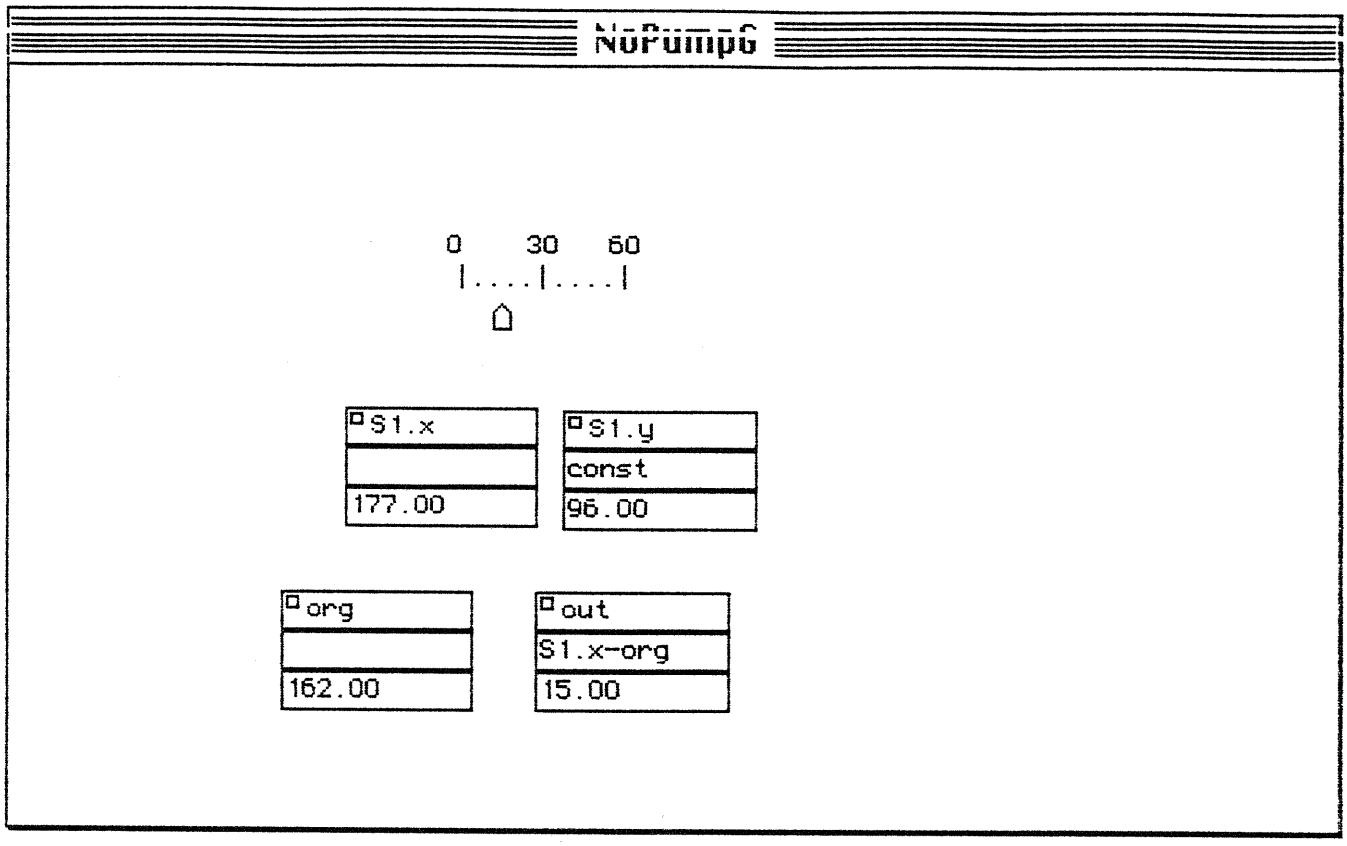
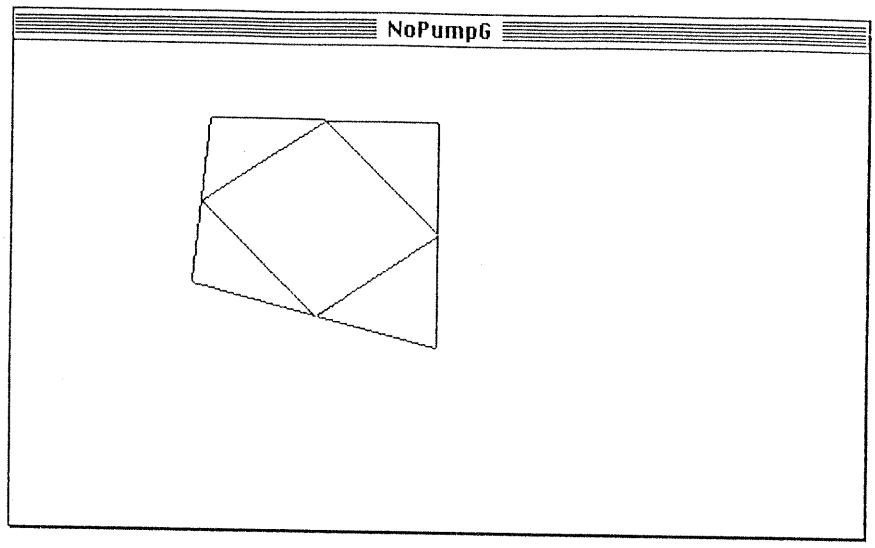


Figure 3: A simple slide controller. Since the control cell for the X coordinate of the pointer, S1.X, contains no formula, the pointer can be dragged horizontally. The cell OUT is updated automatically to reflect the position of the pointer.



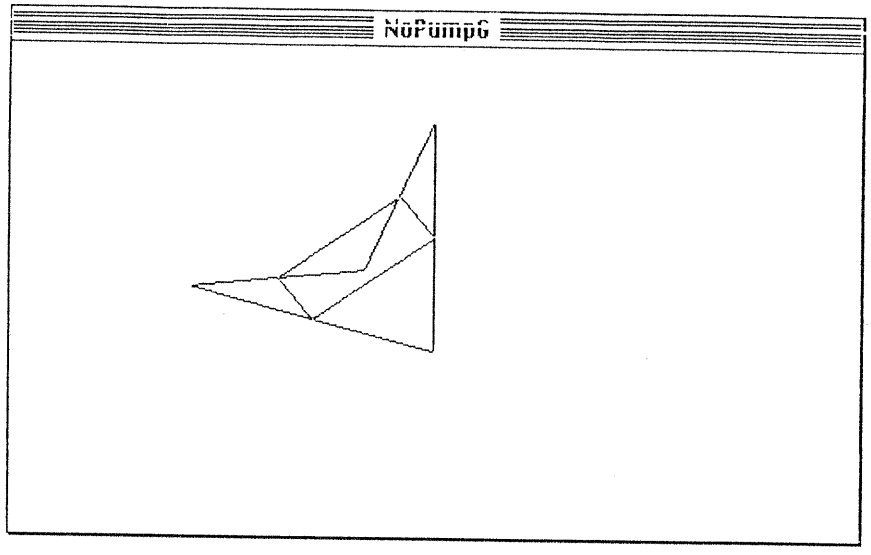


Figure 4: Two views of a quadrilateral with the midpoints of its sides connected. The midpoint connectors are updated as the vertices of the figure are dragged, and always form a parallelogram.

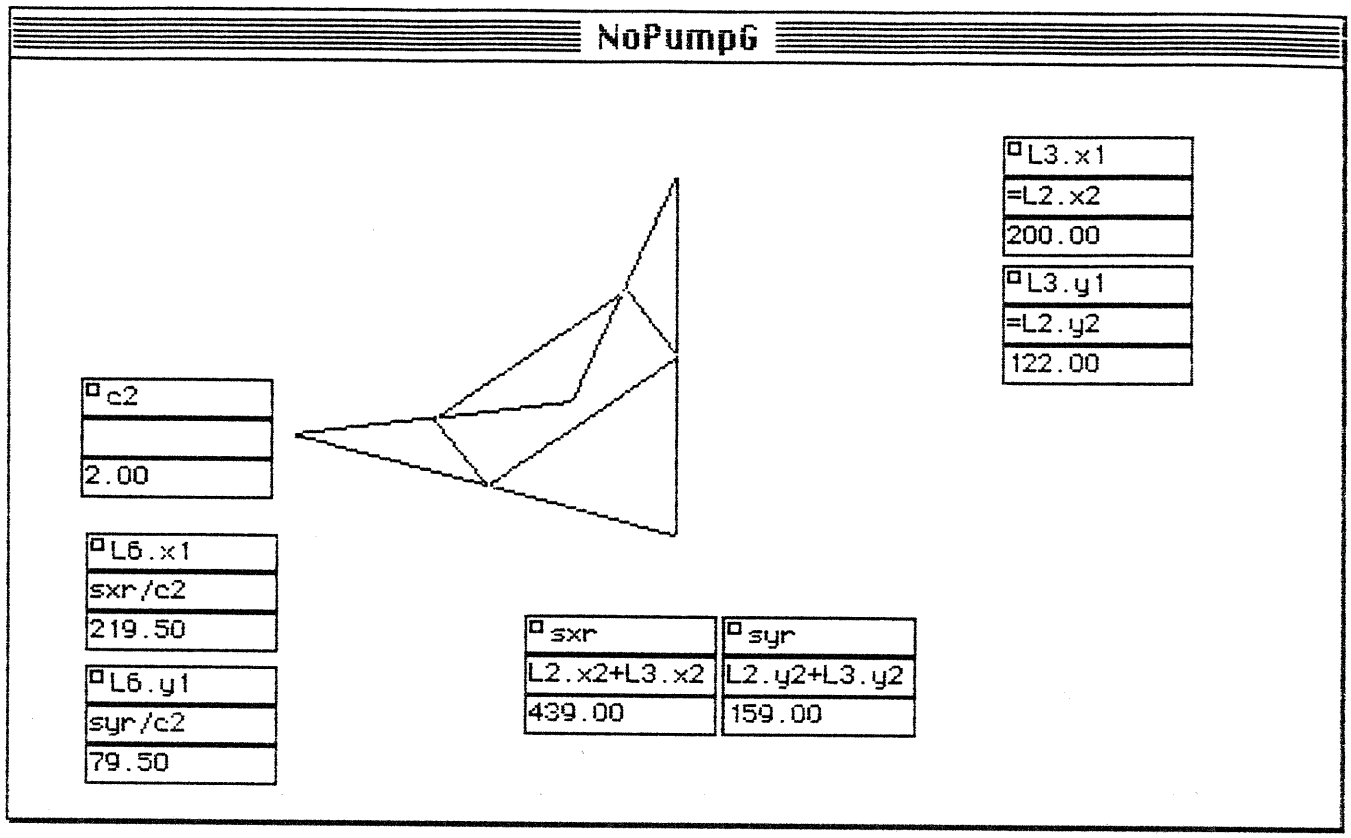


Figure 5: Some of the cells used to build the structure in Figure 4. The cells in the upper right illustrate how the lines forming the quadrilateral are tied together at the corners. The cells in lower center show how the sums of the coordinates of the ends of the sides are computed, and the cells in the lower left show how these sums are used to compute the coordinates of the ends of the midpoint connectors.

This example is easy to construct in NoPumpG. The four lines of the quadrilateral are connected by placing formulae like those shown in the upper right portion of Figure 5 in the cells controlling one end of each line. Cells like those shown in the lower portion of the figure contain formulae that compute the sums of the end coordinates of each line. Dividing these sums by two produces the coordinates of each midpoint, and formulae to this effect are simply inserted into the control cells of four new lines, as illustrated in the left-hand portion of the figure. The unconstrained line ends of the original quadrilateral can now be dragged at will, and the coordinates of the midpoint connectors will be continuously recomputed, and those lines redrawn, accordingly.

This example shows how a dynamic display, responding immediately to graphical manipulations, can be constructed by using the linkage between control cells and graphical objects in both directions within the same structure. Changes to the position of the quadrilateral, controlled in the graphical domain, cause recomputation within the spreadsheet, which in turn causes the midpoint connectors to be displayed in new positions.

Animation by time-dependent positions.

Because graphical objects are redisplayed in response to changes in their control cells, and the control cells can be given formulae which depend on arbitrary computations elsewhere in the spreadsheet, animation can be obtained simply by permitting formulae which are time-dependent. This is accomplished in NoPumpG by including a CLOCK cell in the spreadsheet, whose value is automatically updated by the system. Means are provided for stopping and starting the clock as desired. If a formula in a control cell depends directly or indirectly on the clock, the corresponding graphical object will move.

Figure 6 shows two snapshots of a simple example of this form of animation. The cells controlling the position of the buttefly sketch, S1.X and S1.Y, contain formulae which depend on the clock. The advance of the clock between the two panels of the figure has caused the sketch to move down and to the right.

This form of animation can be used with more complex structures, since any objects whose positions are related to an object that is tied to the clock will also be updated automatically. For example, it would be easy to tie one vertex of the quadrilateral in Figure 4 to the clock. As it moved (in whatever manner was specified) the parallelogram would be automatically adjusted.

Note that this kind of animation is accomplished in NoPumpG without any further extensions to the spreadsheet framework. Once a cell containing a clock is provided animation becomes possible without requiring users to master any additional concepts: the basic spreadsheet evaluation mechanism, together with the linkage between cells and graphical objects, does the work.

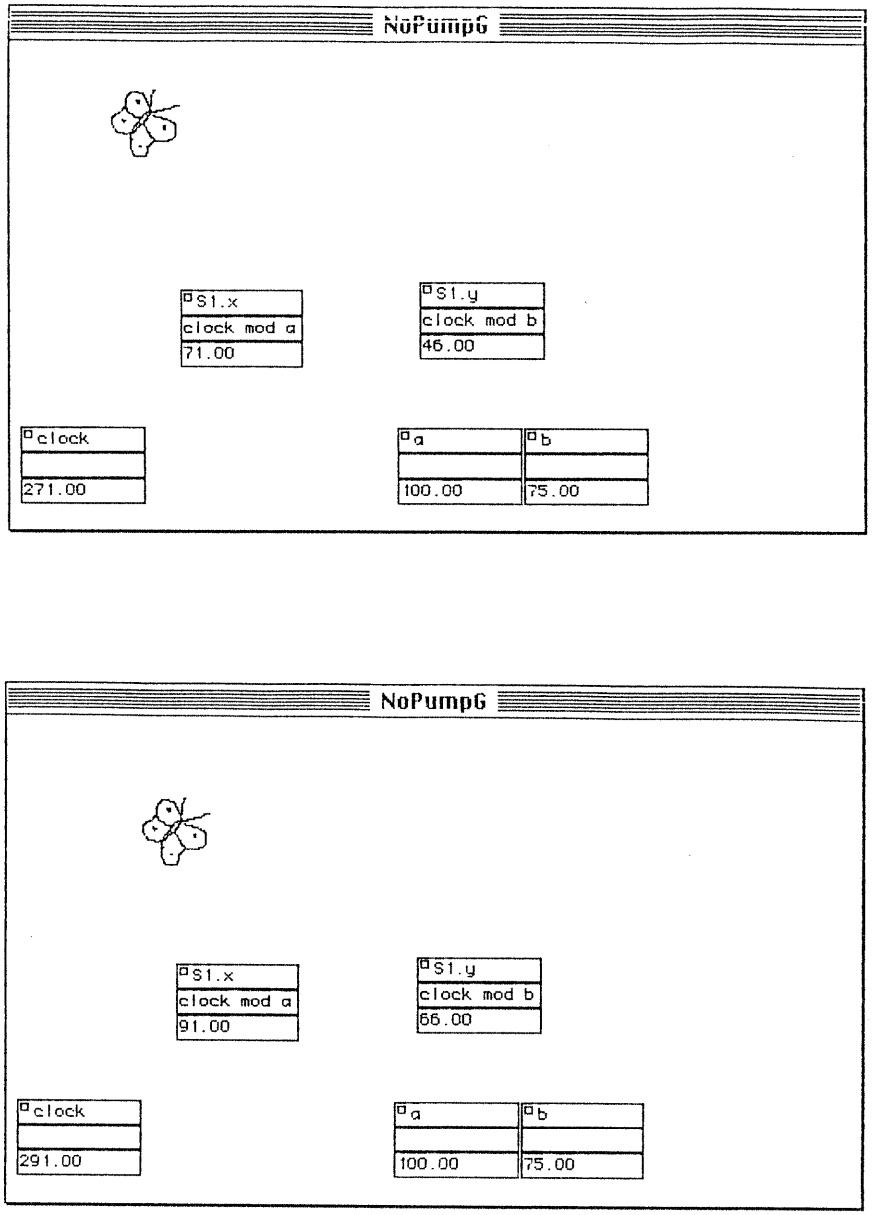


Figure 6: Animation by time-dependent position. As the clock advances between the two panels the values of the control cells for the position of the butterfly, S1.X and S1.Y, are updated, causing the butterfly to move down and to the right. The Y coordinate increases downwards.

Integration.

In many physical systems it is not possible to specify the positions of objects as simple functions of time, because these functions are determined dynamically. Consider an object which is to move with a velocity determined by a user-adjustable control. Since the settings of the control and how they change are not known in advance it is not possible to express the position of the object as a simple function of time.

One approach to this problem would be to express the new position of the object at an instant as the sum of its old position and its current velocity, scaled in an appropriate way. But placing a formula of this kind in a spreadsheet leads to an immediate loop: the position depends directly on itself.

To deal with this problem NoPumpG provides an integral operator. If F is a cell placing the formula $\int F\,dt$ in another cell C produces the following behavior. When the clock is started, and has the value zero, the value of C is set to zero. When the clock is updated the value of C is augmented by the product of the time since the last update and the current value of F. Even though F is an argument in the formula for C, C is not updated when F changes but only when the clock is updated.

This integral operator makes it easy to specify the motion of objects in a physically reasonable way. Figure 7 shows a demonstration of the motion of a mass suspended by an elastic cord. The position of the mass is expressed in a physically meaningful manner as the sum of an intitial position and its integrated velocity, where the velocity is obtained by integrating the acceleration, and the acceleration is the quotient of the mass and the total force on the weight. The total force is the sum of a gravitational force and a force produced by the spring, and so on. The parameters of the model, including the mass and the spring constant, can be varied, even while it is running, simply by placing new values in the associated cells. The demonstration can easily be elaborated by providing graphical controllers, like that shown in Figure 3, with which to vary these parameters.

Animation by visibility control.

While time-varying positions can accomplish animation adequately in many situations, some motions are too complex to be conveniently described in this manner. Figure 8 shows two views of a butterfly in flight. Describing the transition between these views in terms of the motions of smaller graphical objects would be tedious, and would tax the performance of the system.

NoPumpG deals with this problem by permitting different views of an object to be displayed cyclically in the same location, producing animation by the succession of images. Doing this requires only one new feature: additional control cells that determine whether a graphical object appears on the screen or not. Figure 9 shows these cells and their use in animating the butterfly. In the first panel the control cell S1.V, which controls the visibility of sketch 1, has the value 1, while S2.V, which

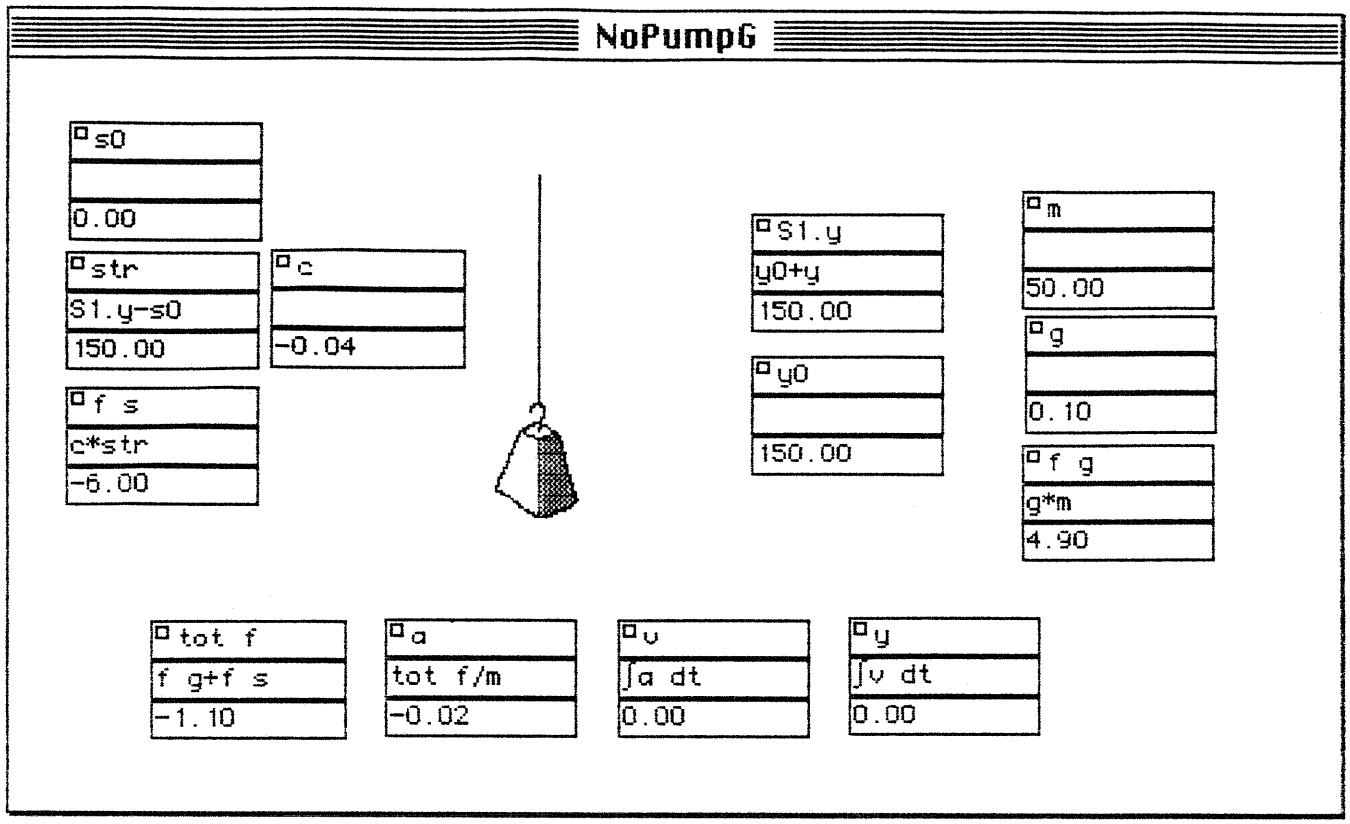


Figure 7: A simulation of a mass suspended from an elastic cord. The formulae in the cells show how the position of the mass is specified from basic physical principles, with the force due to the cord calculated on the left, the force due to gravity on the far right, and the change in position due to these forces at the bottom. The control cell S1.Y, which determines the vertical position of the mass, contains a value obtained by adding the change in position to an initial height of the mass. When the clock runs the mass oscillates up and down because of the time integrals in some of the formulae.

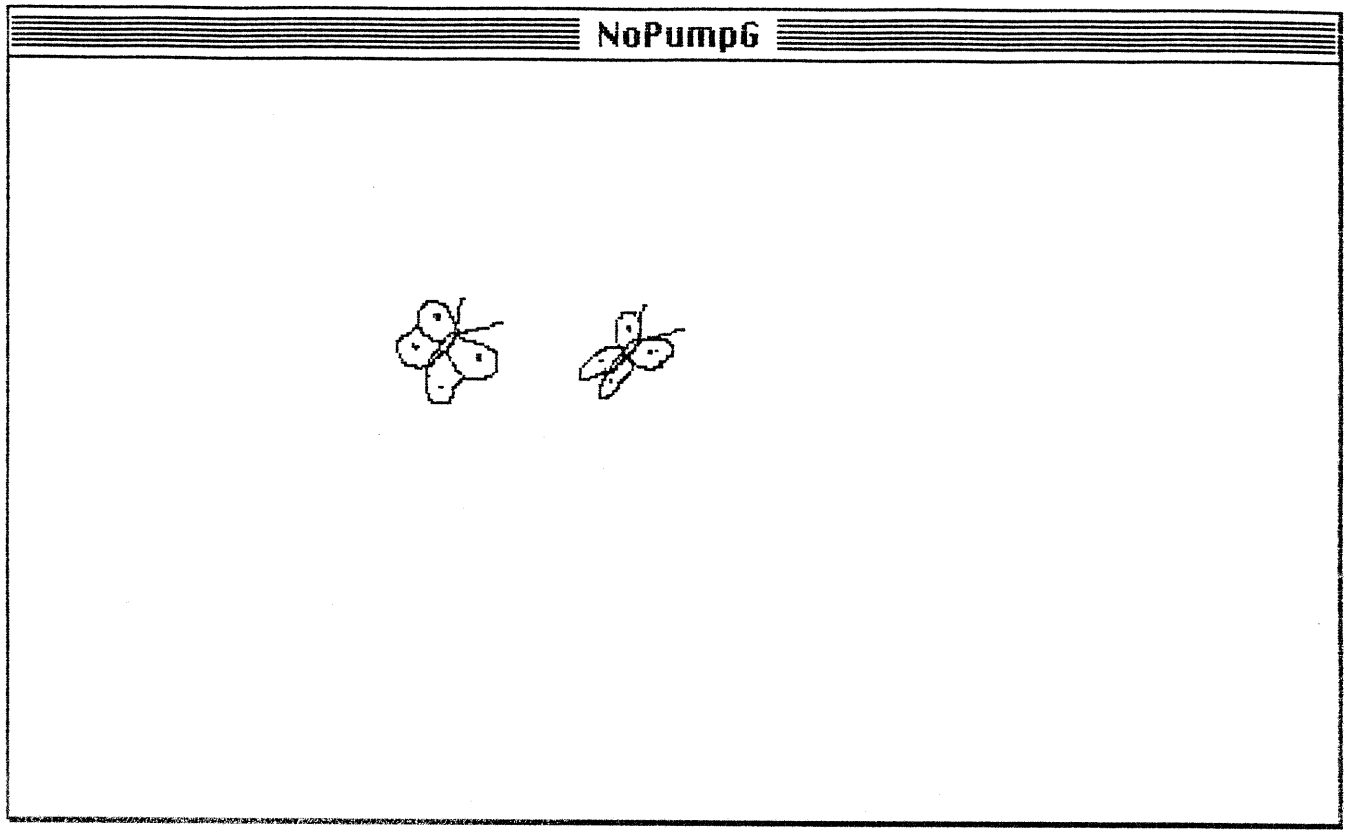


Figure 8: Two views of a butterfly in flight.

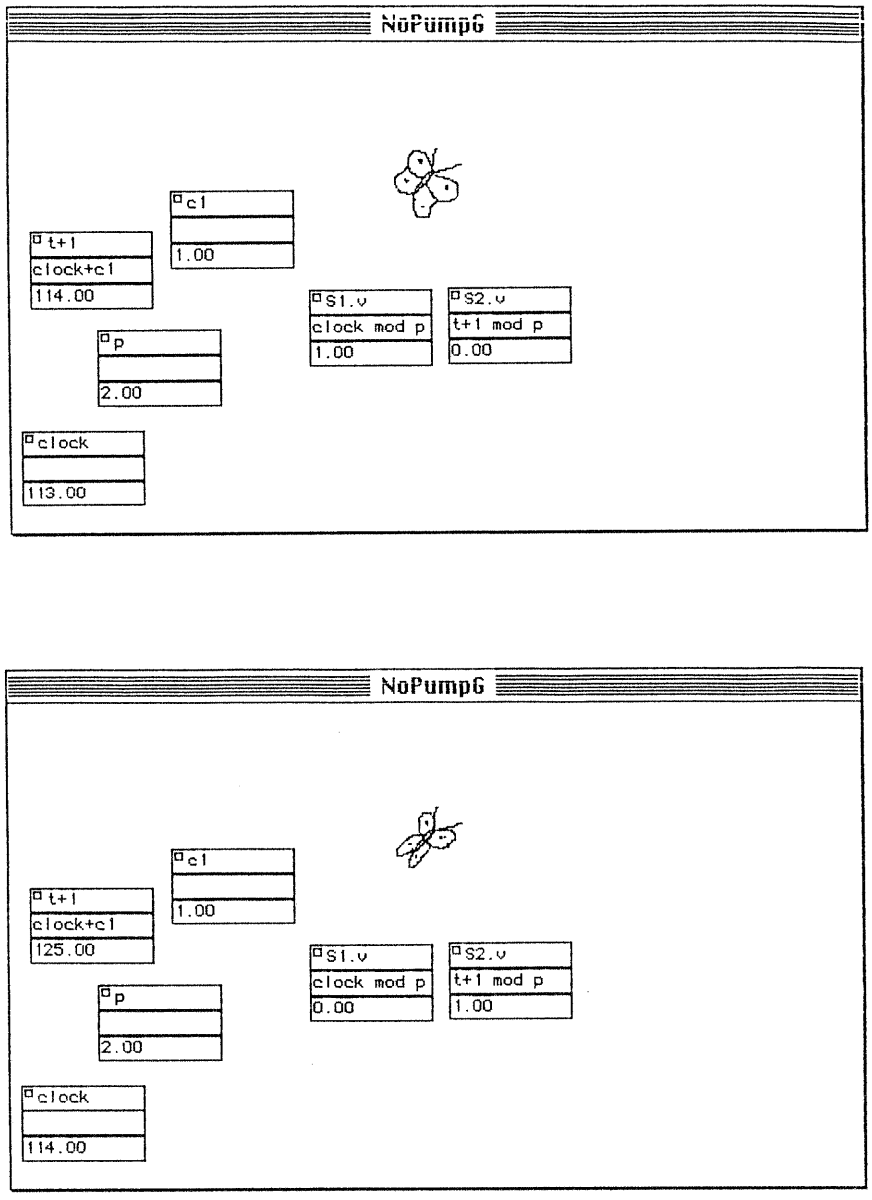


Figure 9: Showing how animation can be obtained by alternating the views shown in Figure 8. The cells S1.V and S2.V control the visibility of the two views. The formulae in these cells are tied to the clock in such a way that they are displayed alternately as the clock advances.

source changes more recently. If the cell is a control cell (see below) the lc op takes one cell as an argument. The value of the lc will be either the value dictated by the position of the associated graphical object, as for an ordinary control cell, or the value of the cell selected as the argument, whichever has been updated more recently. If lc is used in an ordinary cell it takes two cells as arguments and its value is the value of whichever of these has been updated more recently. Lc can be used to get some of the effects produced by true contraints. For example, to tie two control cells together so that they are equal but either can be updated by dragging, place an lc in each with the other as argument. This is shown in the demo "**or lines**", where it is used to keep the ends of two movable lines aligned. The **if** op takes three arguments. The first argument is tested. If it is positive the value is the second argument; otherwise it is the third argument. The **?=** op has value 1 if its operands are equal and zero otherwise.

4 Lines. To **create** a line select Line from the New menu. A line comes with 5 cells, called **control cells**, four of which hold the coordinates of its ends, while the fifth determines whether the line is visible. They have names indicating what coordinate they control. The ends can be **dragged**. If one of the control cells for the line has a formula in it that coordinate cannot be changed by dragging but will be determined by the formula. For example, if you use the eq op to make the second y coordinate of a line equal to the first y coordinate, the line will be constrained to be horizontal. The first end can be dragged in any direction, but the first end can only be dragged horizontally.

5 Text. To **create** a piece of text select Text from the New menu. The cursor will change to a text cursor, inviting you to click a location on the screen where you wish to type. Type in the desired text. You can backspace to correct mistakes. Typing is terminated whenever you make your next mouse selection (hitting Return or Enter has no effect.) You get three **control cells** with your text, two of which contain (and control) the coordinates of the beginning of the text string. You can use these to move the text, tie it to another object, etc. Text can be **dragged** if the mouse is placed near the beginning of the text string.

5 Sketches. A sketch is a little picture created by editing a bitmap. To **create** one select Sketch from the New menu, and click the location on the screen where you wish the sketch to appear. An **editing window** will appear on the screen, showing a grid, each cell of which represents a bit in the sketch. To permit you to make a series of related sketches the grid will be initialized to the last sketch you made (or loaded from disk.) Click the mouse to flip a bit on or off. Dragging will flip a series of bits on or off depending on what happened to the cell in which the dragging started. That is, if you wish to turn a series of bits on by dragging you must begin the drag in a cell that is off, and vice versa. A box to the right of the editting grid shows you what your picture will look like on the screen. When you are satisfied, click the go away box on the editing window. Sketches have two **control cells** which can be used to move them, tie them to other objects, and so forth, and a third which controls whether the sketch is visible. Sketches can be **dragged** if the mouse is placed near the point corresponding to the center of the editting grid.

6 Pens. A pen object, which is shown as a "V" on the screen, can leave a visible trace when it moves. It has x and y control cells, and a third which controls whether the pen is down: a positive value in this cell means that the pen is down and will leave a trace; other values mean pen is up and no trace will be left. The pen can be dragged, for freehand drawing, or it can be made to move by dependence on the clock or other cells whose value is changes. The traces left by pens can be erased by selecting **erase ink** on the **Edit** menu.

7 Clock. There is a clock cell provided by the system. Use the Clock menu to start, reset, and stop it. You can obtain animation by using the clock as an argument in a formula that computes a coordinate of a line or other object. The demo "**bird**" shows this.

The clock is implicitly used by the ∫ **operator**. When the clock runs, the cell with the ∫ operator is updated by adding the product of the clock time since last update and the current value of the argument cell. This implements an approximation to the time integral of the argument cell. For example, if you create a cell containing the velocity of an object (possibly controlled by a graphical controller) the ∫ op can be used to calculate the position of the object. This is shown in the demo "**igtst**".

8 Visibility control cells. Besides animating things by giving them time-dependent positions, you can devise frame-sequence animation by using the **visibility control cells** of a group a related sketches. To do this you put all the sketches at the same position, but arrange things so that only one is visible at a time in sequence. The demo "**butterfly**" shows this: there are two pictures of the butterfly shown alternately.

Lines and text as well as sketches have these visibility control cells. In each case the associated object is displayed only when the visibility control cell is positive.

9 Deleting objects. Cells can be hidden using their go-away box, and other objects can be hidden using their visibility control cells. But you may want to get rid of something entirely, especially if you intend to export your work, as discussed below. **Delete** on the **Edit** menu lets you click on an object to delete it. The object will refuse to go away if it has something else depending on it or on one of its associated control cells, or if it is the clock.

Note: The way dependencies are determined is crude; you won't be allowed to delete something if its control cells depend on each other. You can deal with this by editing the formulae in these cells to remove the dependency.

Note: Deleted objects will still show up in the **Show** menu, but will not be enabled. Try to ignore them.

10 Getting rid of everything. Selecting **New** on the **File** menu flushes all work so you can start afresh.

11 Exporting and importing work. The current contents of the system can be stored on disk by selecting **Save as** from the **File** menu, or by responding yes when you are asked whether you want to save what you have before quitting. Work can be reloaded by selecting **Open**, which will cause what you load to **replace** what you have, or **Merge**, which will **add** what you load to what you have. The dialog is rocky; someday it will be more Mac-like.

You can **Merge** in the same material as many times as you wish. Thus you can save a useful subassembly and then merge it into later work as needed.

Nothing is done about making names unique, so if you do merge in several copies be prepared to find that you have several objects with the same name.

12 Stopping work. Choose **Quit** from the **File** menu. You'll be given a chance to save what you've done on the way out.

13 Housekeeping. When new cells are created, or hidden cells revealed, they may overlay other cells, creating what looks like garbage. You can usually sort things out by moving one of the cells off of the others. To minimize this effect new cells are created in an area at the bottom left of the screen in such a way that you can create five cells before getting overlap. It is a good idea to move cells out of this area, or hide them, as soon as convenient. A similar approach is used for lines, so it is a good idea to drag a new line away, or otherwise change its position, before creating too many other ones.

It may also happen that you find it impossible to drag an object that overlaps others. What is happening is that your mouse click is being handled by some object that itself won't move. If you are really stuck you may have to edit some control cells to break the impasse.

controls sketch 2, is zero. Objects are visible just when their visibility control cells are positive, so sketch 1 is shown and sketch 2 is not. In the second panel the clock has advanced, and the contents of the visibility control cells has changed so that sketch 1 is hidden and sketch 2 is shown. This alternation continues as the clock runs. The position control cells for the two sketches have the same values, so the sketches alternate in the same position on the screen.

Concurrent processes.

As the clock advances, any objects whose positions depend on the clock are updated automatically by the system. Simultaneously, dragging by the user is detected and any resulting changes in the display are made. As a result no special techniques are needed to model situations in which several activities, including user input, are going forward concurrently.

Handling bidirectional relationships in a simple evaluation model.

The machinery described so far has an important limitation most easily seen by considering a position control cell for an object. If this cell has a formula it can change the position of the object, but the object cannot be dragged by the user to provide a new value for the cell. If the cell has no formula the object can be dragged but the cell cannot change the position of the object. Thus a single graphical object can provide input to the spreadsheet, or output from it, but not both.

An example drawn from Borning (1981) illustrates the problem. Suppose we wish to illustrate the relationship between Celsius and Fahrenheit temperatures. A natural way to do this would be to provide two thermometers, like that shown in Figure 2, but arranged so that we can drag either mercury column to a new level, and have the other column move to the equivalent level on the other scale. We cannot accomplish this with the features so far described, because the level of the mercury is either determined by a formula, in which case we cannot drag it, or draggable, in which case it cannot be adjusted by the spreadsheet in response to changes in the other thermometer.

Borning's ThingLab solves this problem by using a more powerful kind of constraint than that supported by the basic spreadsheet machinery. ThingLab constraints are specifications of relationships among quantities or objects, together with mechanisms for modifying these entities so as to satisfy those relationships. These modification methods can change any of the entities in the constraint. In the thermometer example the mercury columns are constrained to be equal, but in contrast to what happens in a simple NoPumpG model this constraint would be satisfied by moving the other column whenever the user modifies one column.

NoPumpG deals with this problem with one added feature, which does not require any modification of the simple spreadsheet evaluation mechanism. The LC operator, standing for 'last changed', works as follows. If LC is used in a formula in an ordinary cell it takes two other cells as arguments. The value of the formula is the

value of whichever argument has more recently changed. If placed in a control cell it takes just one cell as argument, with the results of dragging the associated object playing the role of the second argument. Thus a control cell with an LC operator in its formula will respond to dragging as if it contained no formula, but will also respond to changes in its argument cell.

Figure 10 shows how the temperature conversion problem is dealt with using the LC operator. The formulae in the cells show how L2.Y2, which controls the position of the top of the mercury in the thermometer on the left, can be set either by dragging or by a temperature computed (ultimately) from the position of the mercury on the right. Similar arrangements provide for the mercury on the right to move when the mercury on the left is dragged, or to be dragged itself.

Abstraction.

In building structures of any complexity, abstraction, the ability to identify and reuse substructures, is essential. Subroutines in conventional programming languages are a good example of abstraction, as are class definitions in object-oriented programming. NoPumpG supports abstraction through its export and import facilities. At any time the content of a model being built in NoPumpG can be stored. Any stored model can then be retrieved and merged with the current model. By repeating this merge operation many copies of a stored model can be incorporated in the current model.

The thermometers in Figure 10 where built in this way starting with the thermometer in Figure 2. It was not necessary to repeat the work of tying the pieces of the mercury together, or building the scale.

This abstraction facility serves NoPumpG's commitment to minimizing the proliferation of concepts. The idea of copying is familiar and concrete, and does not require such additional concepts as the distinction between classes and instances, as may be required in other approaches.

On the other hand, not all repeated operations can be abstracted using just copying. Suppose one is building a structure in which a large number of text strings are to be tied to a like number of sketches, in pairs. For one pair this is done by placing appropriate formulae in the control cells of the text string. There is no way to use copying to avoid the necessity of performing this operation manually for each pair.

A further limitation of simple copying is that changes to a substructure are not automatically reflected wherever the substructure is used. If an improvement were made to the thermometer in Figure 2 corresponding changes would have to be made to each copy of it in Figure 10. In more complex abstraction schemes this problem is dealt with by separating the definitions of objects from instances of the definition. Changing the definition of the thermometer would change any instances subsequently created from it. The example in Figure 10 could be handled by reinstantiating the definition each time the model is set up for execution. Since this

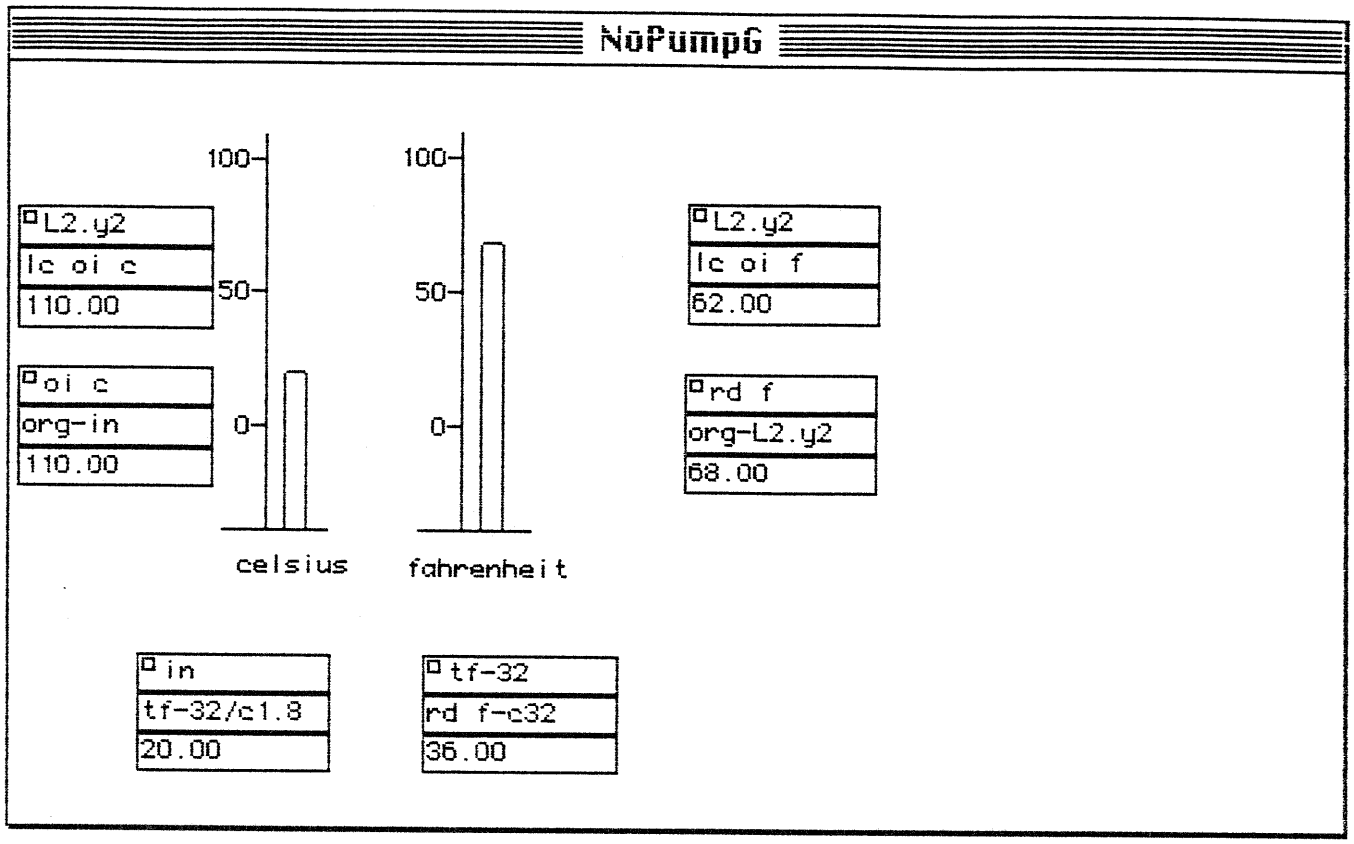


Figure 10: Coupled thermometers. When either mercury column is dragged the other registers the corresponding temperature on its scale. The LC operators in the two cells L2.Y2, which control the heights of the mercury, permit these heights to be controlled either by dragging or by a calculation driven by the height of the other column. The calculation that drives the Celsius thermometer can be traced in the cells shown. The Fahrenheit mercury has been dragged to an absolute height of 62, which corresponds to a scale reading of 68°, as shown in the cell RD F. This Fahrenheit scale value is converted to Celsius in the cells TF-32 and IN. The resulting Celsius value of 20° is converted to an absolute height of 110 in the cell OI C, and this value determines the height of the Celsius mercury in the cell L2.Y2 at upper left. If the Celsius mercury were to be dragged this value of L2.Y2 would be replaced by the result of the drag, and a similar computation, not shown, would adjust the Fahrenheit mercury.

cannot be done using simple copying NoPumpG pays a price for using a conceptually simple abstraction scheme.

COMPARISON WITH OTHER APPROACHES.

How does NoPumpG, with its spreadsheet foundations, compare with other systems with similar application goals? I group other systems according to their central idea, or at least the central idea that seems most important in comparing their treatment of graphics with NoPumpG's. I will emphasize the way in which a programmer must specify and control interactive graphics in each system, with secondary attention to the interface provided to an ultimate end user.

Procedural languages.

CMU Tutor (Sherwood 1985) produces graphics by executing procedural commands embedded in a language designed to support tutorial interactions. Procedural languages do not normally support creation and modification of graphics by direct manipulation, but CMU Tutor implements an elegant linkage between programs and their output that provides some of the benefits of direct manipulation. The program instructions that draw graphics contain specifications of points in the graphics output area. If one of these point specifications is selected for editing, and then the mouse is clicked somewhere in the graphics area, the clicked point replaces the previously specified point. The same machinery can be used to add new point specifications to a program, so the code that produces a drawing can be produced largely by clicking points in the output area.

While this innovation gives it some important advantages over other procedural graphics system, CMU Tutor is still a procedural language. Users must organize the presentation of material, and interactions with the user, in procedural form.

The Boxer system (diSessa, 1985; diSessa and Abelson, 1986) provides an innovative and economical conceptual framework in which diverse operations including text editing and programming can be performed. As in CMU Tutor, the expressive medium for graphics in Boxer is a procedural language, but some important facilities are added. The Boxer language supports moving entities called sprites which have speed and heading attributes. Since sprites move autonomously according to their heading and speed attributes, animations involving multiple objects moving at once can be obtained in a natural way.

Programming by Rehearsal (Finzer and Gould, 1984) is a simplified object-oriented programming system in which objects (called performers, in a theatre metaphor for programming) are controlled by messages (called cues). While message-passing permits a more convenient style of programming for many problems than standard procedural languages, the language is still procedural at bottom in that the response of an object to a message is specified as a procedure. The programmer must specify the order in which actions occur in response to a message.

Programming by Rehearsal includes an elegant method for easing the burden of this procedural programming on people not accustomed to it. Rather than writing out code to specify how an performer should respond to a cue, the programmer can put the system in an observation mode and then demonstrate manually how the performer should respond. The system captures the actions in the demonstration and writes the corresponding code automatically.

Simple procedural approaches are ill suited to describing highly dynamic situations in which one or more objects are in motion and user actions or other operations must be catered for simultaneously. The difficulty is that the programmer must indicate how control is to be distributed among several activities, such as moving an object or watching for user input. Boxer deals with this problem with its sprite mechanism, which makes it possible to specify the behavior of different sprites and leave it up to the system to manage them. Programming by Rehearsal uses a form of multiprocessing in which user actions can spawn new processes. In both cases new concepts are needed to adapt the procedural framework to handle concurrent activities; no new concepts are needed in NoPumpG's spreadsheet framework.

Constraint systems.

Borning's ThingLab (1981) permits graphical interactions to be programmed without specifying procedures directly. While ThingLab is implemented in an object-oriented, and hence procedural, framework, once a suitable collection of primitive objects has been defined a programmer works with objects whose behavior is specified in terms of constraints that they and their parts obey, rather than in terms of responses to messages. As mentioned earlier, constraints in ThingLab are relationships that the system will enforce as objects are moved or changed, together with methods for enforcing it. For example, the midpoint of a line segement is constrained to lie on the segment equidistant from its ends, no matter how the segment is moved or stretched.

ThingLab's primitives include objects whose behavior cannot be replicated in NoPumpG. For example, elements are provided which can be graphically connected on the screen by direct manipulation. These can be used to build dataflow diagrams or models of electronic circuits.

Borning (1986) describes extensions to the original ThingLab to permit constraints to be specified graphically, rather than by writing SmallTalk code, as in the original. In some applications this method works in a manner similar to NoPumpG. For example, the midpoint constraint can be expressed by showing how the coordinates of the midpoint can be computed from the coordinates of the ends of the segment, much as in NoPumpG. However, constraints in ThingLab are ordinarily bidirectional, in that changes to either of two mutually constrained objects affect the other, not unidirectional as in NoPumpG. Having defined the midpoint of a segment one could move the midpoint, and the system would respond by changing the segment in some way so as to maintain the constraint between midpoint and segment. In NoPumpG the natural way of defining midpoint would result in a

midpoint that could not be moved but would always be determined by the current position of the line segment. The bidirectional constraint could be constructed using the LC operator, but this would be considerably more complicated than the unidirectional one.

Duisberg (1986) describes how constraints involving time can be used to produce animation. The principle is the same as employed in NoPumpG: if the position of an object is made to depend on time the object will move. This avoids the perplexities of specifying concurrent activities procedurally. The embodiment of this idea is more sophisticated, and complex, in Duisberg's system than in NoPumpG, however. A class of specialized objects, called responses, is provided; these generate streams of screen update events which the implementation interleaves to produce smooth animation.

Together with these extensions, ThingLab offers much more functionality than NoPumpG. This difference in functionality represents an expected tradeoff between power and complexity. Some of ThingLab's greater power comes from additional primitives, constructed at the SmallTalk implementation level. Analogous extensions to NoPumpG would be possible (and probably desirable), but would increase the complexity of the system.

ThingLab's notion of constraint is a powerful one, but more complex than the simple notion of computation that NoPumpG inherits from the spreadsheet. To specify constraints users must write SmallTalk code, or, if their problem is within the scope of the graphical method for building constraints, must deal with some more sophisticated notions than are involved in NoPumpG, such as anchoring an entity to keep it from being changed during constraint satisfaction, or merging two objects to form a new one that is subject to the constraints specified in each original object.

Smith's Alternate Reality Kit or ARK (1986, 1987) provides a constraint-like mechanism for controlling the movement of simulated physical objects. Objects called 'interactors' are provided that, when turned on, produce influences on and interactions among other objects. For example, a 'gravity' interactor produces attractive forces between objects; users can turn off this interactor, or modify it, to see the effect of various forms of gravitational forces on the movement of objects.

ARK is implemented in and provides access to SmallTalk, and uses a message-passing model of computation in addition to the constraint-like facility provided by interactors. Means are provided to simplify this side of the system, so that users can drive simulation models using simple pushbutton and slide controls.

It is not clear what kind of programming is required to create new objects, interactors, and the like in ARK, so a direct comparison with NoPumpG is difficult. It does appear that some features of ARK would be difficult to provide in NoPumpG. For example, a mouse-controlled hand is used in ARK to pick up and move objects. The hand imparts velocity to the object, so that it is possible to throw objects across

the screen using the mouse. It is conceivable that a somewhat similar effect could be contrived in NoPumpG but it would be complicated and unnatural: there is no apparent way to determine just when an object starts and stops being dragged.

On the other hand, some of the objectives of ARK can be achieved in NoPumpG with its simpler conceptual framework. It is not difficult to set up gravitational attractions between objects, for example, and permit a user to experiment with different force laws; the same approach used in the example in Figure 7 works quite well. But one objective of ARK is to make physical laws concrete by representing them as visible objects, the interactors, that can be placed in a space or removed. It would be difficult to bundle NoPumpG's alternate gravity specifications in such a neat way.

Dataflow.

Hookup (Levitt 1986) is a system primarily intended for control of a music synthesizer, that incidentally supports interactive graphics with about the same level of conceptual complexity as NoPumpG. Hookup allows the user to build a dataflow diagram whose nodes are drawn from a collection of primitives that include slide controls, clocks, sprites, x-y graphs, storage devices, and others. By interconnecting these nodes by direct manipulation the user can specify (for example) that the X coordinate of a sprite will be the value of a clock, and that its Y coordinate will be provided by a slide control. An auxiliary input to a sprite selects one of a number of prepared views to be displayed at the sprite's current position, so animation by alternation of views is easy.

As it stands, Hookup does not provide primitives at as low a level as NoPumpG, so less flexibility is provided. For example, there is no line segment primitive, so building animated linkages is not possible. On the other hand, Hookup provides a variety of specialized nodes whose function cannot easily be replicated in NoPumpG, including push buttons, which produce transient signals, and nodes which respond to such signals by starting clocks, by storing or deleting data, and the like. There appears to be no reason why Hookup could be extended to provide the level of primitives that NoPumpG offers, while retaining these additional features.

If Hookup were extended in this way, the two systems would offer interesting competing metaphors. Hookup's dataflow metaphor, in which nodes communicate by sending values along paths, in the much the way electronic components do, might or might not be as readily comprehensible to nontechnical users as NoPumpG's spreadsheet model. On the other hand, the spreadsheet model may or may not be extensible to deal with events like button-presses in as natural a way as the dataflow model.

LIMITATIONS AND EXTENSIONS.

As I have said, NoPumpG aims to provide broad functionality while adding as little

as possible to the conceptual framework of the spreadsheet. It is inevitable that the resulting design, like the spreadsheet itself, will solve some problems poorly or not at all. In this section I will take stock of NoPumpG's deficiencies, and consider what might be done to remedy them. In some cases, to be considered first, it appears that simple extensions to the implementation, not requiring much in the way of new conceptual overhead, should suffice. In other cases it is not clear what should be done, or whether any sufficiently comprehensible extension is possible.

Some simple extensions.

Additional graphical primitives. Some form of parametrized curve should be provided in the same manner as line segments. Provision could easily made to permit such curves to have indefinitely many parameters (and hence control cells.)

Pens. In many problems it would be useful to show a visible trace of the motion of an object, or to display a function graphically. Pens could be provided as primitives, with control cells for position and for pen-up or pen-down. Means would be needed to erase the traces produced by the pens.

Text i/o. NoPumpG's current text objects are fixed and so cannot be used to permit the user to enter text or to display text computed from other information. Enhanced text objects could be provided whose value would be held in a control cell. The text would be editable, if this cell contained no formula, or would be computed if the cell contained a formula. Appropriate operations on text, such as concatenation, would be needed.

Graphical operations. Tying the ends of two line together by placing formulae in control cells is workable but tedious. A purely graphical shortcut could be provided by which pointing at two points would cause formulae to be placed in the control cells for the coordinates of the first point (say) that copied the values of the corresponding cells for the second point. Similar graphical operations could be provided that would place formulae in such a way as to locate one point in a given position relative to another, or to fix a point in place by placing the 'const' operator in its control cells.

Access to the mouse. If NoPumpG were to be useful as a tool for building user interfaces it would need to support a much wider range of mouse actions. As it is, the mouse can be accessed only when the user chooses to drag an object: actions like clicking a button, or simply moving the mouse without depressing the button, cannot be handled. It would be possible to provide cells which always contain the position of the mouse, and the state of the mouse button. This would make possible more flexible use of the mouse, though using such low-level information would be tedious. Determining whether the mouse was on a particular object when the button was pressed, for example, would be possible but complicated. Higher-level support for some common operations would be desirable. For example, one might associate a new control cell with a sketch that would toggle between one and zero when the mouse was clicked on the sketch.

More difficult issues.

Aggregates. NoPumpG has not inherited from spreadsheets the desirable ability to specify operations like 'total' or 'average' on groups of cells. Although this could not be done in just the usual way, since NoPumpG's cells are not placed in a grid of rows and columns, means could be provided to allow the user to select a group of cells as an argument.

There are deeper issues in handling aggregates for which the basic spreadsheet mechanism provides no solution, however. How would one cause a new cell to be added to an aggregate at runtime (for example, when reading data from a file)? How would one refer to such a cell once it was created? Some spreadsheets provide a macro language that can deal with these matters, but as discussed further below these languages add a great deal of procedural machinery to the basic spreadsheet paradigm. It is possible that such problems should be avoided by permitting aggregates of data to appear as values of single cells, rather than as scalar values situated in a group of cells, but such an approach may be difficult to understand.

Andreas Lemke has pointed out that some of these same issues arise in NoPumpG in connection with graphical objects. How could one create a new line segment dynamically? How would one refer to its control cells? One could imagine an operator with suitable arguments that would build a new line or other object, with specified position and with specified formulae in its control cells, and another that would permit new formulae to be constructed and inserted in specified cells. But this machinery would involve many concepts alien to the basic spreadsheet framework.

Relative copying of formulae. Spreadsheets provide a very useful shortcut by permitting formulae to be copied from cell to cell in such a way that the arguments in the formula are adjusted to suit the context into which the copy is placed. This exploits the fact that cells are placed in a regular array, so that the relative position of a formula and any cell to which it refers is well defined. If a formula refers to a cell one place to the right and two up, a copy of it will refer to the cell in the same relative position to the copy. This relative copying saves a great deal of work if (as often happens) several groups of cells are related computationally in the same way, for example, if several rows of a table are to be totalled at the end of each row.

NoPumpG cells are not placed in a grid, so this facility cannot be provided in a straightforward way. It might be possible to permit the user to group cells into local grid arrangements on the screen, and permit relative copying between cells so grouped.

Collisions. Many problems involve detecting and responding to collisions between objects, such as a ball and a wall, or two simulated vehicles. Such interactions can be handled in NoPumpG in simple cases, but not easily. There are two aspects of the problem. First, determining when a collision occurs involves tedious computation

with coordinates. Second, taking action when a collision is detected is a special case of handling an event, to be discussed more generally below.

The first problem, determining when a collision occurs, could be given some high-level support, but the most useful feasible form of such support is not clear. In principle every primitive could have a control cell whose value would be one if the object overlapped any other object and zero otherwise. This would be computationally expensive, and would still leave it up to the user to determine what object had been hit. Another approach would be to provide a specialized primitive, a bumper, with controls cells indicating whether a collision is occurring and an angle representing the orientation of the contour being struck. This would make it easier to produce some common interactions, like bounces. Distinguishing barriers from background scenery would require further machinery.

Events, more generally. Suppose a problem requires a change in behavior when a button is pressed, or a collision occurs, that persists after the button is released or the collision ends. In the simplest spreadsheet model such behavior is impossible: formulae can have different values when the button is down from when it is up, assuming the state of the button is reflected in some cell, but when the button goes back up these formulae will revert to their former values.

NoPumpG already has two ways to get around this apparent restriction. First, suppose some cell contains the integral of the button state cell, and suppose the button state is zero when the button is up and one when it is down. If the button is pressed the integral will become positive, and stay positive even after the button is released. Thus formulae which depend on this integral can change their output permanently when the button is pressed.

NoPumpG provides an if-then-else operator which can be used to produce this desired latching behavior in a more straightforward way, as pointed out by David Kieras. This operator takes three arguments. It examines the value of its first argument, and chooses the value of its second or third argument as its value depending on whether the value of the first argument is positive or not. Suppose the cell BUTTON_BEEN_PRESSED contains the formula IF BUTTON_STATE THEN 1 ELSE BUTTON_BEEN_PRESSED. Whatever the initial value of this cell, it will become 1 and stay 1 when the button is pressed. The apparent loop in this formula is harmless because BUTTON_BEEN_PRESSED only propagates an update notice to itself when its value actually changes, which happens only once when the button is pressed. More elaborate formulae, involving more cells, can be used to perform more complex operations like storing the value of the clock whenever an event occurs.

These features of NoPumpG make it possible to handle events, without added concepts in the language, but the treatment is unnatural and clumsy. A version of Hookup's approach, in which specialized primitives are provided which respond to event signals, might be preferable, despite some added conceptual overhead. For example, a SNAPSHOT operator with two arguments could be devised whose value

would be the value of its first argument at the last time at which its second argument changed from nonpositive to positive. The button latch would just be SNAPSHOT 1 BUTTON_STATE, and a device to note the time at which an event occurred would be SNAPSHOT CLOCK EVENT.

Connectable components. While the NoPumpG user can construct objects out of cells, lines, and other primitives, and then interconnect these objects by placing appropriate formulae in the cells, there is no graphical way to accomplish this interconnection. The kind of dataflow diagram that one can construct in ThingLab or Hookup can be built statically in NoPumpG but cannot be built or modified dynamically without manually manipulating formulae.

A start could be made towards providing the desired functionality by supplying new primitives called (say) 'connect-in' and 'connect-out'. A connect-out object would behave just like an ordinary cell, internally, but would appear as a distinctive icon on the screen. A connect-in object would have an associated cell to contain its value, which would be the value of a connect-out object to which the connect-in object is connected by a line segment on the screen. By moving line segments around different connections of connect-in and connect-out objects could be made. The treatment of connect-in objects that are not connected to any connect-out object would have to be determined. One approach would be to give each connect-in object a default value that it assumes whenever it is not connected.

While this facility would be a step forward, and would permit simple dataflow diagrams to be built dynamically in NoPumpG, it is not adequate for all situations. Some interconnections of objects require several cells to be linked between the same two objects, which would require several connections to be placed manually in this scheme. It appears that a more general solution would require the ability to encapsulate and place under simple graphical control arbitrary actions, including placing numbers of connections and entering new formulae. I consider this general issue next.

More powerful abstraction. As mentioned above, NoPumpG's sole abstraction mechanism permits the user to export objects and reimport copies of them. This allows the internal structure of objects, like the thermometer in Fig 2, to be specified once and used many times. However, it does not permit the work of interconnecting objects to be specified once and used repeatedly, nor does it permit operations like interconnection to be placed under the control of simple user actions.

One approach to this problem would be to incorporate a procedural language in NoPumpG in which operations like formula entry could be specified. A device like that used in Programming by Rehearsal might be used to shield the user from detailed knowledge of this language.

Even with this shielding, such an approach would represent a major departure from NoPumpG's philosophy of hewing as close as possible to the spreadsheet computational model. While spreadsheets often provide a macro language which is

in this spirit, such a language does not use the basic spreadsheet execution model, and involves the same difficult issues of control flow, use of variables, and the like, as standard programming languages. For example, users of the macro facility in Microsoft's EXCEL spreadsheet (Microsoft, 1985) may encounter these notions: dereferencing, goto, input, and return (with and without a returned value).

Another direction to explore, which might require less of a departure from the spreadsheet model, or at least less traffic with procedural language, would be the use of analogy. It might be possible to permit the user to indicate two groups of objects and ask the system to modify the second so that it resembled the first. If two objects in the first group were connected in a particular way, as indicated by formulae in the two objects referring to parts of the other, the system would place analogous formulae in corresponding objects in the second group. This can be seen as an extension of the idea of relative copying discussed above. How correspondences between objects in the two groups should be established is not clear; one approach would be to rely on relative position, as suggested for relative copying.

Even if this abstraction-by-analogy is workable there remains the problem of placing such operations under the control of simple user actions, like clicking an icon with the mouse. If objects were provided which could take as their value designated groups of other objects, an operator could be devised which would take as arguments two such groups and an object which would produce an event signal (a change of value from nonpositive to positive, as suggested earlier). When the event signal was received the operator would adjust the second group by analogy with the first.

It remains to be seen whether a design along these lines is possible. If it is, it would then have to be determined whether the conceptual complexity thus introduced would pay its way, or whether its inclusion would increase only what is in principle possible in NoPumpG rather than what users actually do.

CONCLUSION.

NoPumpG builds on the basic concepts of the spreadsheet to provide flexible support for interactive graphics and animation. The main added concept is that of a bidirectional linkage between graphical objects and cells in the spreadsheet. Additional features, the presence of a clock in a spreadsheet cell, cells whose values control the visibility of graphical objects, an integral operator, and an operator which permits a cell to reflect values from more than one source, increase the power of NoPumpG without adding many new concepts. The resulting system is simpler than, but also less powerful than, other systems with similar goals. Like the spreadsheet itself, NoPumpG aims to provide broad, but not unlimited, functionality, and natural interaction, on a very simple conceptual foundation.

This design goal is open to challenge. It may be that in the long run users will benefit more from systems that are less minimal conceptually, and provide a good setting in which to learn about procedural concepts like recursion, or about complex data structures, or about message passing, or constraints. But two arguments can be made

against this. First, market success demonstrates unequivocally that people at large have embraced the spreadsheet despite its limited power and flexibility. In practice, spreadsheets deliver more computing power than apparently more powerful frameworks that are more difficult to understand and use. Second, our understanding of computation generally is too immature to permit us to make reliable judgements about how people "should" think about computation. Any of today's programming paradigms may seem inappropriate in the future, as more and more disposable computational power, possibly organized into massively parallel systems, becomes commonplace.

REFERENCES.

Borning, A. (1981) The programming language aspects of ThingLab, a constraint oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, **3**, 353-387.

Borning, A. (1986) Defining constraints graphically. In *Proc. CHI'86 Human Factors in Computing Systems*, New York: ACM, 137-143.

Budge, B. (1983) *Pinball construction set* (Computer program). San Mateo, CA: Electronic Arts.

diSessa, A.A. (1985) A principled design for an integrated computational environment. *Human-Computer Interaction*, **1**, 1-47.

diSessa, A.A. and Abelson, H. (1986) Boxer: A reconstructible computational medium. *Communications of the ACM*, **29**, 859-868.

Draper, S.W. (1986) Display managers as the basis for user-machine communication. In D.A. Norman and S.W. Draper (Eds.) *User Centered System Design: New perspectives on human-computer interaction*. Hillsdale, NJ: Erlbaum, 339-352.

Duisberg, R.A. (1986) Animated graphical interfaces. In *Proc. CHI'86 Human Factors in Computing Systems*, New York: ACM, 131-136.

Finzer, W. and Gould, L. (1984) Programming by rehearsal. *Byte*, **9**,187-210.

Horowitz, E. and Munson, J.B. (1984) An expansive view of reusable software. *IEEE Transactions on Software Engineering*, **SE-10**, 477-487.

Levitt, D. (1986) Hook Up: An iconic, real-time data-flow language for entertainment. Unpublished technical note, MIT Media Lab.

Lewis, C.H. (1985) Extending the spreadsheet interface to handle approximate quantities and relationships. In *Proc. CHI'85 Human Factors in Computing Systems*, New York: ACM, 55-59.

Lewis, C.H. and Olson, G. (in press) Can psychology lower the barriers to programming? To appear in *Proceedings of the Second Workshop on Empirical Studies of Programmers*, Ablex.

Microsoft (1985) *Microsoft EXCEL: Arrays, Functions, and Macros*. Bellevue WA: Microsoft.

Piersol, K.W. (1986) Object oriented spreadsheets: The Analytic Spreadsheet Package. *OOPSLA '86 Proceedings*, ACM, 385-390.

Sherwood, B.A. (1985) An integrated authoring environment. *Proc. IBM Academic Information Systems University AEP Conference.*

Smith, R.B. (1986) The alternate reality kit. In *Proceedings of 1986 IEEE Workshop on Visual Languages*, Washington, DC: IEEE, 99-106.

Smith, R.B. (1987) Experiences with the alternate reality kit: An example of the tension between literalism and magic. In *Proc. CHI'87 Human Factors in Computing Systems*, New York: ACM, 61-67.

van Emden, M.H., Ohki, M., and Takeuchi, A. (1986) Spreadsheets with incremental queries as a user interface for logic programming. *New Generation Computing*, **4**, 287-304.

USING THE NOPUMPG PROTOTYPE

Clayton Lewis
Department of Computer Science and
Institute of Cognitive Science
Campus Box 430
Boulder CO 80309 USA.

A technical report on NoPumpG, of which this is an Appendix, is available from the author.

The NoPumpG project has been supported by the Air Force Human Resources Laboratory.

Comments, suggestions, and reports of related work are eagerly solicited. Please pass the software on to any interested people. Every conceivable disclaimer applies to the software: it is a very rough draft!

1 Getting started. Double click the NoPumpG icon. An information screen will appear. Press RETURN to begin work. Select **Merge** from the **File** menu to bring in a demo from the disk. Most of the demos do something when you start the clock, using the **Clock** menu.

2 Cells. To **create** a cell, select cell from the New menu. You will be prompted for a name. Cells have three fields: name, formula, and value, displayed as a stack of boxes. To **move** a cell drag in the name field. To edit a **formula** (or put one in where there is none) click on the formula field. This will enable the Ops menu. Select an operator (see below). If an argument is needed the cursor will change to a cross. Click on the cell you want for an argument. If more arguments are needed the cursor will stay as a cross until you click on enough other cells. Note that the prototype does not support nesting of operators. To build up formulae with more operators you will have to create additional cells. Also, constants cannot be entered as parts of formulae. You must create a cell and put the desired number in it. To edit the **value** of a cell click on the value field. You will be prompted for a new value. To **hide** a cell click on its little go-away box in the name field. To **bring a hidden cell back** select its name on the **Show** menu.

3 Operators. Most of the operators are straightforward. Here are descriptions of the odd ones. The **eq** op just indicates that this cell's value will be the same as that of the argument cell you click. The **const** operator locks in the current value of the cell. The value cannot be changed, by editing or by moving an associated line, until the const operator is removed. The **null** op puts in an empty formula. The $\int$ op approximates the time integral of its argument cell; see discussion of clock below. The **lc** operator is a feature which permits a cell to be updated in either of two ways. The name stands for "last changed", meaning that the cell takes its value from whichever input