Sunrise Boxer

{Structures}

The diSessa Family Foundation
October 2021
www.Computational-Literacy.com

What is this document?

"Boxer Structures" is a high-level overview of all the elements of Boxer, how they work, and what they are typically used for. It explains, in a general way, the nature of boxes, how to use them, and the various different *kinds of boxes*. It explains both standard and unusual features of the Boxer language and user interface, including the (unusual) way that Boxer files work, and the many features for changing the basic interactive properties of the system (the "reconstructible interface"). It briefly treats different paradigms of programming in Boxer, including "actor-oriented" and "activation-oriented" programming.

"Boxer Structures" should be especially well-suited for:

- relatively sophisticated programmers who are new to Boxer and want to get an overview of all that's there.
- documentation for non-beginners of how Boxer and its different elements and features work, to help them along with projects—e.g., "how do "triggers work?" or "how does the "build" command work to create complex Boxer objects?"
- a place to discover new features when you feel you are ready to expand your Boxer horizons.

It is **NOT**:

- an introductory tutorial to instruct you in getting started working with Boxer. (You will find a few simple tutorials in the Docs & Demos that are distributed with Boxer. Docs & Demos also has minimal documents, "survival guides," to help you get started.)
- documentation of the details of the interface.
- a listing and explanation of all Boxer commands. (See the on-line or "hard-copy" Boxer Manual, again in Docs & Demos.)

BOXER STRUCTURES [*]

[*] © Andrea A. diSessa, 1994 – 2021.

INTRODUCTION: WHAT IS BOXER?

This document explains the basic organization, structures and mechanisms of Boxer. Boxer is fundamentally a place to build and use computational tools. With it, for example, you might construct and/or use:

- A hypertext document that contains text, pictures, simulations or other computational objects, all organized in a well-structured, easy-to-peruse non-linear form. Such documents could constitute “microworlds” intended to support free exploration and learning about a particular topic.
- An application or tool of the usual sort, say a “draw” program, a program to help you design bridges and understand the stress and strain in them, or a personal notebook or organizer.
- A family of tools designed to work together to perform a class of tasks. Users of Boxer have constructed tool kits to help teachers teach and students explore basic number facts and elementary number theory, Euclidean (and non-Euclidean!) geometry, Newtonian physics, the analysis of astronomical images, and the structure and nomenclature of organic molecules. Boxer is especially good at allowing you to combine and personalize tools.
- The organization and interactive properties of your own daily work environment or a work environment targeted to a particular project or task. Boxer allows you an unusual amount of flexibility to organize what you see on the screen and how you interact with it. There are facilities to interact with the Internet and transparently maintain remote Boxer facilities as part of your own world.

It is not a bad place to start by thinking about Boxer as a super text processor. You can type, cut, paste and copy text and images into Boxer essentially whenever and wherever you want. But Boxer is enhanced compared to ordinary text processors in two critical ways. First, among the “texts” you can type into Boxer are programming commands that you can point to and execute in order to accomplish things. Boxer is a complete programming language and environment. The second critical extension of text processing is in structuring. Among the “characters” you can type in Boxer are various kinds of boxes. You can put anything you want inside a box, including more text and boxes. In fact, the organization of any Boxer environment is essentially nothing more than text and boxes, possibly inside more text and boxes. Some boxes may be images (which are themselves boxes that are contained in the midst of other boxes and text).

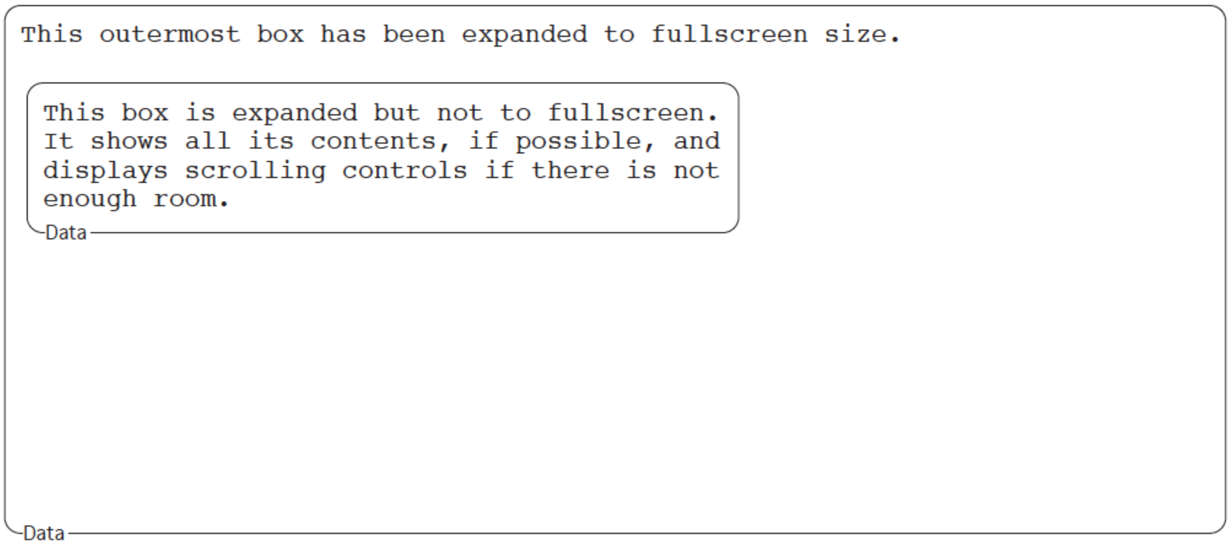
From the outside, a box behaves like a big text character, and it may be cut, copied and pasted alone or with other text and boxes. From the inside, a box may be a world of its own with its own organization and interactive properties.

Here we will not explain the Boxer editor and interface details of how you make, edit and interact with text and boxes. Consult “interface” documentation for that. Nor will we explain the hundreds of programming commands that define Boxer as a language. Consult the Command Manual for that. Instead, we look at the basic static and dynamic forms that constitute the core of Boxer structure.

GENERIC BOX STRUCTURE

Sizes

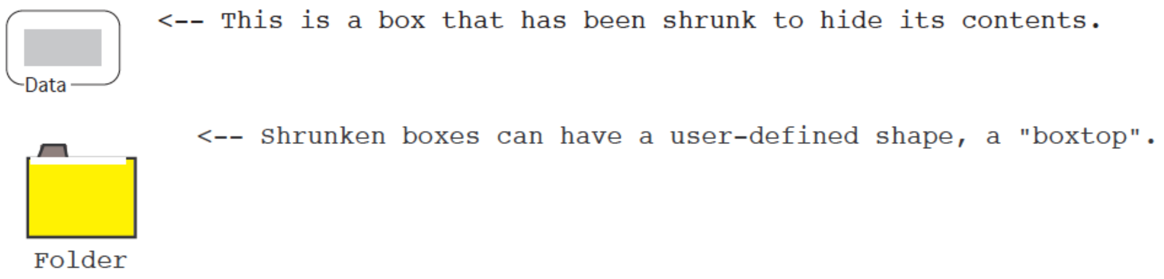
All types of boxes (individually explained below) share some basic characteristics. First, you can put anything you want, text and more boxes, inside a box—as much or as little as you want for your own purposes (although it is generally best to keep boxes as small as possible while still doing their jobs). Boxes generally expand to contain as much as you put into them, and they begin to scroll when you've put more in than will fit on the screen. In addition to their regular expanding/shrinking presentation, boxes can be expanded to fill the whole screen. You would do this generally for one of two reasons: You might expand a box to full screen to enter a new working environment, free of surrounding distractions. Or you may need temporarily to zoom in on a particular box that is too big or complex to be seen clearly in its full context.



```
This outermost box has been expanded to fullscreen size.

This box is expanded but not to fullscreen.
It shows all its contents, if possible, and
displays scrolling controls if there is not
enough room.
```

Figure 1. Normal (expanded) and fullscreen boxes.

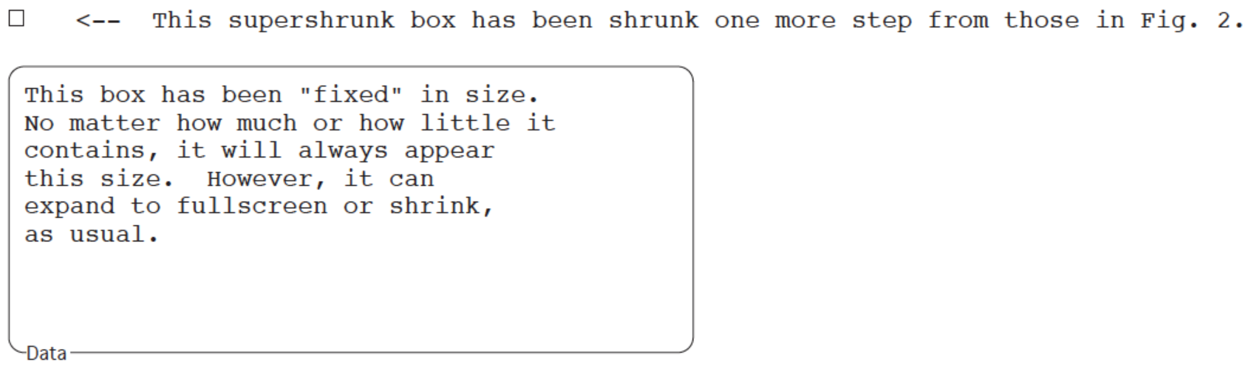
You may also shrink a box down to a small gray region (or to an icon if you have taken the trouble to define a "boxtop" for the box). Generally you will do this to hide the contents of a box you are no longer interested in, or to obtain more free "screen real estate" for other boxes you are more interested in for the moment. There are commands (FULLSCREEN-BOX, etc.) to expand and shrink boxes from programs.



```
<-- This is a box that has been shrunk to hide its contents.

<-- Shrunken boxes can have a user-defined shape, a "boxtop".
```

Figure 2. Options for a shrunken box.

There are two less frequently used box sizes. A shrunken box or icon may be shrunk one more step to a "super shrunk" size, about as big as a text character. This is handy when you have many, many boxes that you want on the screen at the same time, or when you want essentially to hide a box, putting it temporarily out of your current focus of attention. Finally, you can grab the lower right corner of a box and drag it to any new size, which then is fixed until you resize or simply poke the corner again to release. Graphics boxes are ordinarily fixed size since it would be distracting to have them expanding and shrinking as you draw in them. Any other box may be fixed also—many times for the same reason, to avoid distracting size changes of a box whose contents are frequently changing.

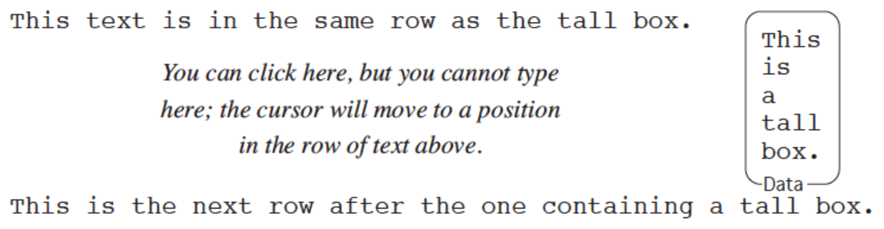


```
□   <--  This supershrunk box has been shrunk one more step from those in Fig. 2.

This box has been "fixed" in size.
No matter how much or how little it
contains, it will always appear
this size.  However, it can
expand to fullscreen or shrink,
as usual.
```

Figure 3. Supershrunk and fixed-size boxes.

Structure

Boxes' contents generally are strictly logically ordered: A box, like text, contains a vertical sequence of lines (called rows) each of which is a sequence of words or boxes. This is very unlike the behavior of windows. Boxes may not overlap, and must be placed at a particular position in a particular row of another box. The reason for this is that everything in Boxer is computationally accessible, and therefore you must know unambiguously which box another box is part of, and which row and item number it is in that box. Having your whole world computationally organized turns out to be extremely powerful. If you want things organized more freely, with overlaps and ambiguities, use a graphics box, as explained below. A consequence of having this strict organizational principle is that if you have a row consisting of text and one tall box, you will not be able to place any new text or box in the space below the text and above the next row. On the other hand, that space counts as part of the row, so clicking in it counts as clicking in that row.



```
This text is in the same row as the tall box.   This
                                                is
                                                a
                                                tall
                                                box.
This is the next row after the one containing a tall box.
```

You can click here, but you cannot type here; the cursor will move to a position in the row of text above.

Figure 4. "Rows" in Boxer.

Closets

Every box has a hidden place into which you may put things you want in the box but not generally visible. The closet is really just an invisible first row that you can make visible when you want to see it. (A closet appears generally containing a box called "drawer," as in a "drawer inside a closet." But you may delete this box, copy and paste in more drawers, or put anything else you want on the closet row.) A set of informatively named drawers might provide excellent partitioning of the resources available inside a microworld.

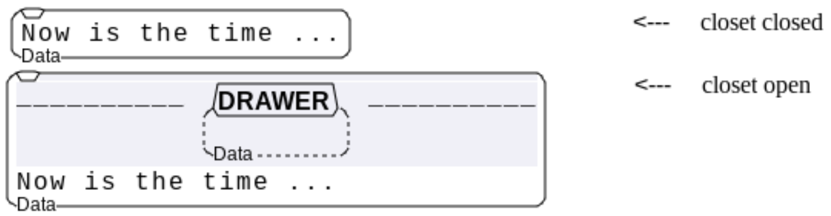


Figure 5. The closet row of a box may be hidden or shown.

Closets generally behave as any other part of the box, although they are not counted as part of the box's contents for some purposes. Special features of closets will be noted below. The closet in the top level box of your Boxer world is special in that it contains a "preferences" command that allows you to set attributes of your Boxer environment. (Alternatively, use the Preferences selection of the Edit menu.)

Naming

Any box may be given a name, which appears in a name-tab at the top of a box. Names must be a single word. If you leave spaces in a name tab, Boxer creates a name with underscores instead of spaces: If you have a box named "my box," you will have to refer to it as "my_box." Boxer is indifferent to case in names: ANDY is the same as andy or Andy. Boxer will not let you give two boxes in the same place the same name.

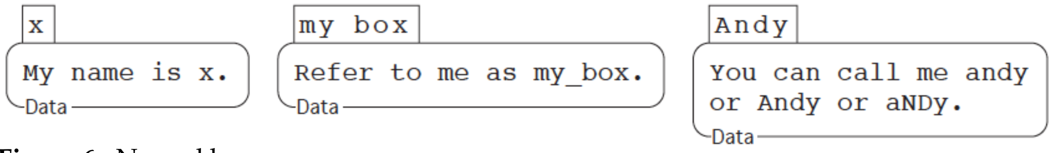


Figure 6. Named boxes.

Named boxes may be referred to by name in any program or in any row you execute directly. In fact, once you have named a box, you must use its name to refer to it in any text that is intended to be executed. A named box is treated only as a definition within executed text; it will not itself execute. As a consequence, you may place named sub-procedures or local variables essentially any convenient place in a procedure. Named boxes are a good place to put comments since they won't interfere with the surrounding procedure. See "comment" in Figure 7 below.

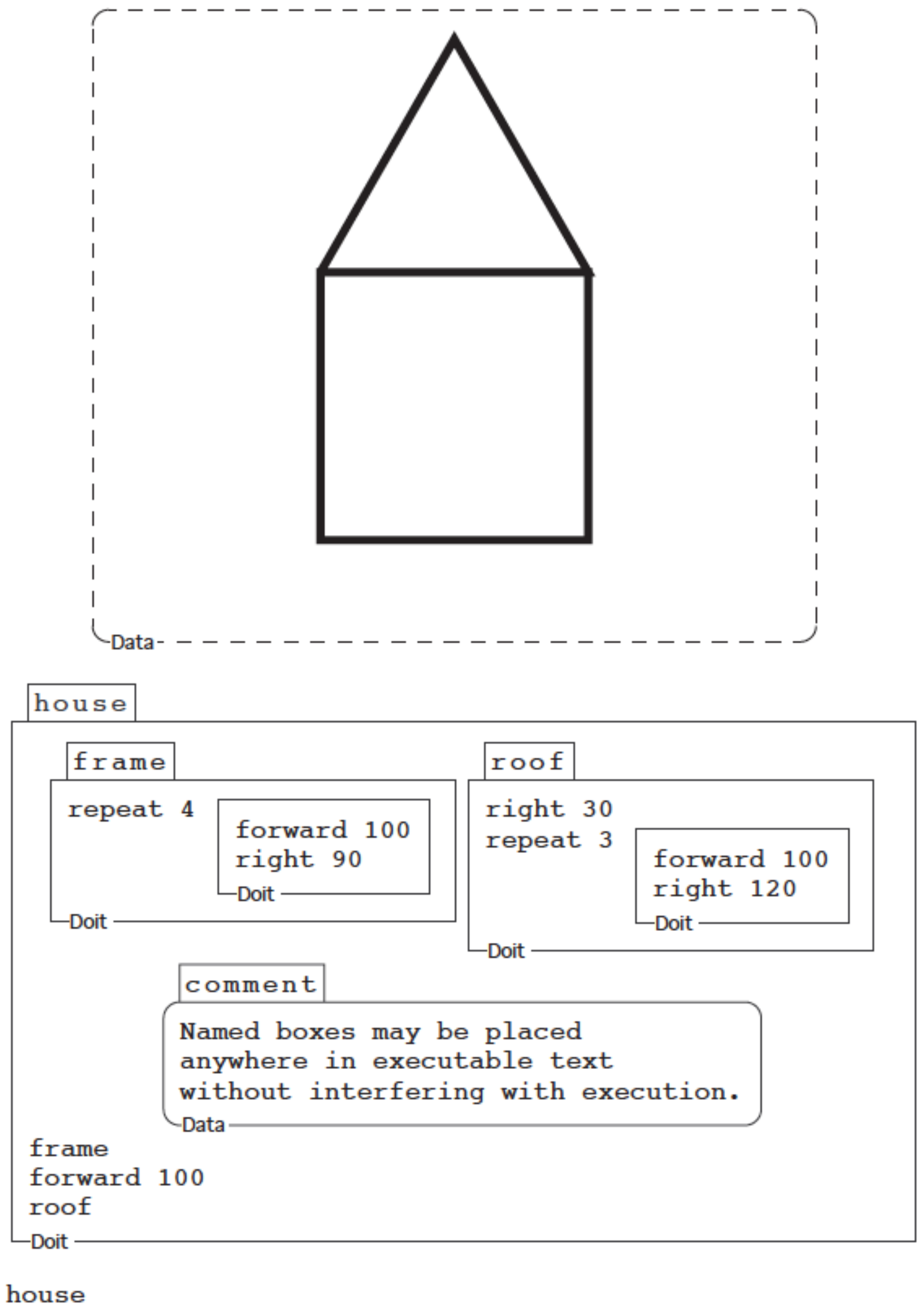
Use BUILD to create named boxes during program execution. See DEFINE in "Boxer extensions" in the tool-box (demo folder), or see BUILD in the on-line Command Manual, Data Manipulation chapter (section on data construction).

Properties

Every box has a set of properties that may be adjusted using the Box Properties selection of the Box menu or with right-press (Mac control mouse-press) on either of the top corners of a box. Properties include options for the box's image when shrunk (boxtop), automatic expanding (or shrinking) of box on entry (or exit), and file or URL links.

DOIT BOXES

Doit boxes are Boxer's procedures, things that are intended to be "done" or executed. They can be described as "procedures" or "programs." Doit boxes have square corners and, unless Boxer preferences are set otherwise, a type identifier in the bottom border.



```
repeat 4
    forward 100
    right 90
```

```
right 30
repeat 3
    forward 100
    right 120
```

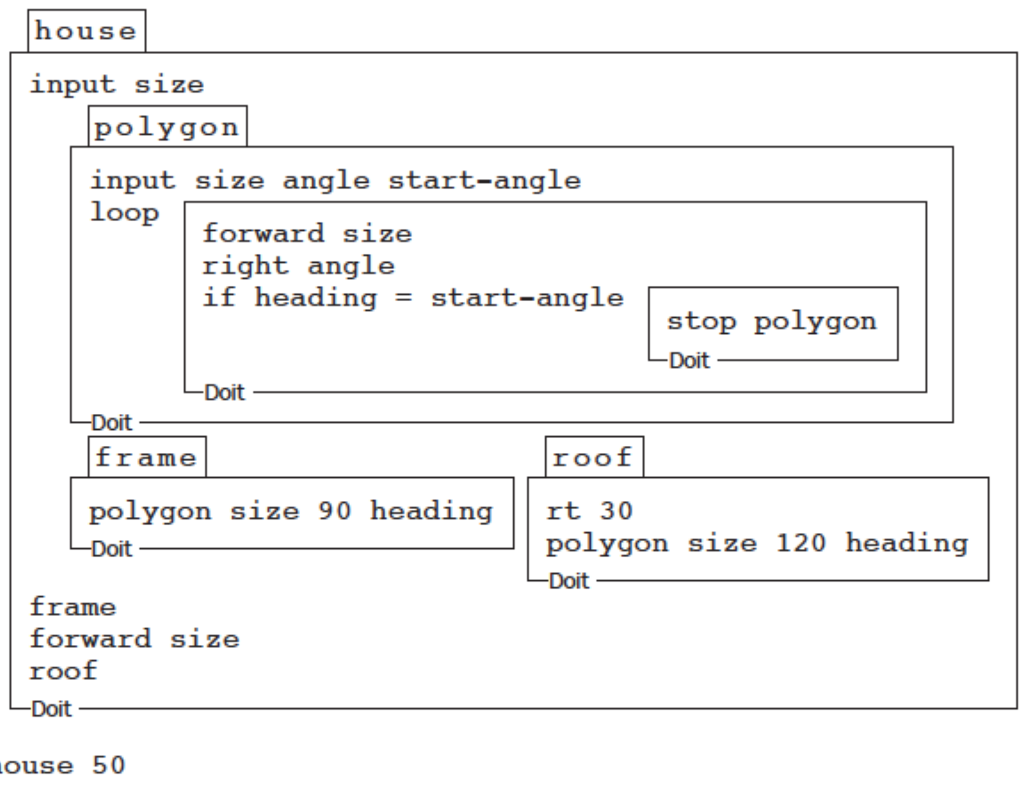
```
Named boxes may be placed
anywhere in executable text
without interfering with execution.
```

```
frame
forward 100
roof
```

```
house
```

Figure 7. Procedures (doit boxes) defining a "house".

Doit boxes may have any number of inputs. This is accomplished by typing “input” and listing names for all the inputs on the first line of a doit box. Thereafter, whenever you use the doit box (procedure) you must provide an input for each input name.



```
house
input size
  polygon
  input size angle start-angle
  loop
      forward size
      right angle
      if heading = start-angle
          stop polygon
          Doit
      Doit
  Doit
  frame
  polygon size 90 heading
  Doit
  roof
  rt 30
  polygon size 120 heading
  Doit
frame
forward size
roof
Doit

house 50
```

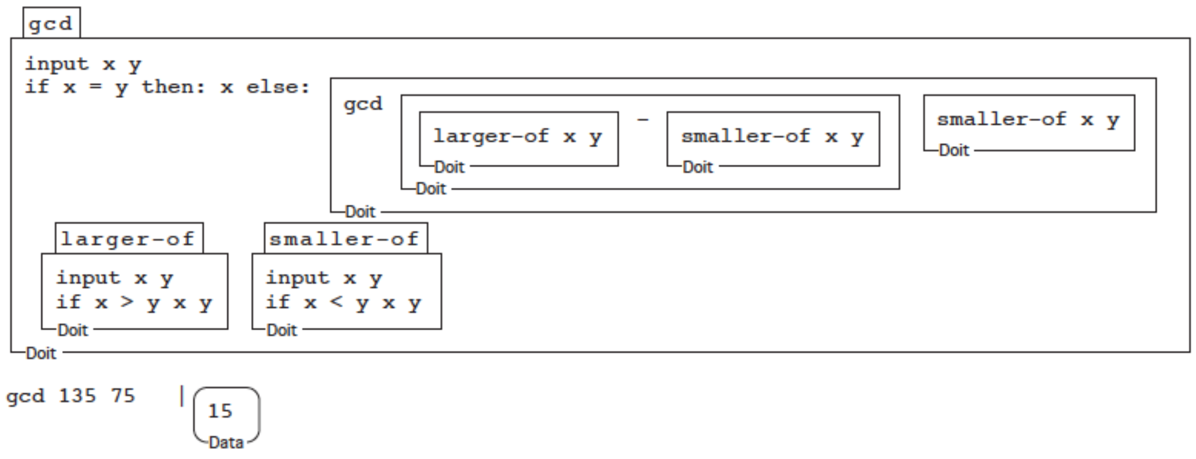
Figure 8. Procedures HOUSE and POLYGON have inputs.

The execution of a doit box may be terminated at any point in the text before the end (where it would ordinarily terminate) by using the STOP command. STOP terminates the execution of the doit box of which it is a part. Or, given a procedure name as input (e.g., STOP POLYGON; see Figure 8), STOP stops the named procedure. To avoid ambiguity about which box should be stopped in cases of nested procedures/doit boxes, it may be best to specify the procedure to stop.

Returned Values (Outputs)

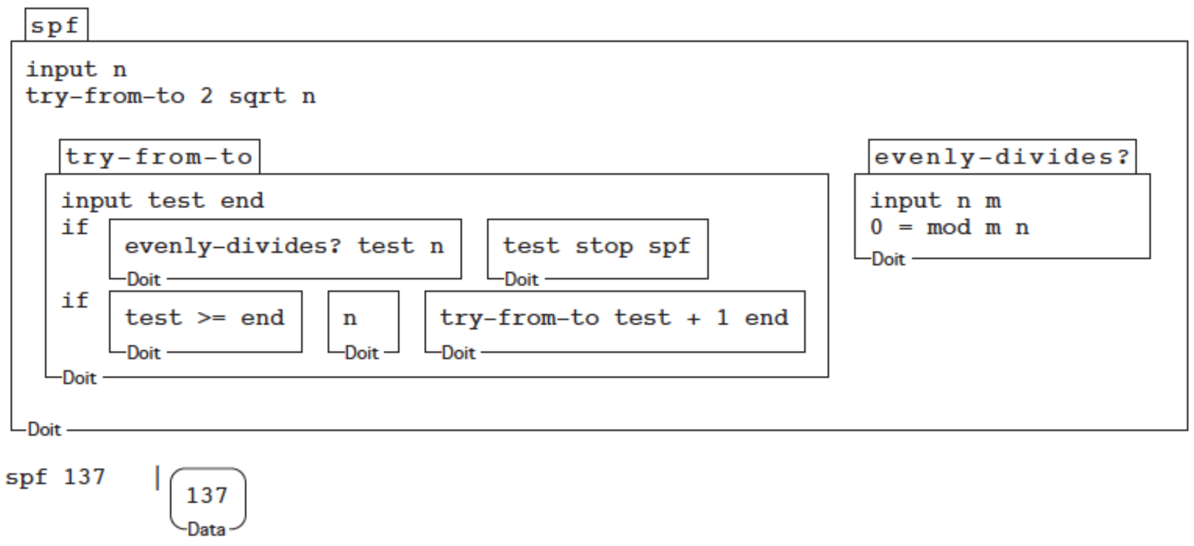
If the last expression executed in a procedure (doit box) has a value, that value will be returned as the value of the procedure. “The last expression executed” may literally be the last expression in the box, or the one just before a STOP command. Again, named boxes do not count in execution; hence they may be placed at the end of a doit box without affecting returned values. Using STOP with an input just after an expression that returns a value can be a powerful way to stop or return values to high-level procedures from low-level ones. (This is sometimes called “catch and throw”; a lower-level procedure “throws” a value up to a higher level one.) A typical expression returning a value might be an arithmetic expression, 3 + 5, or a turtle command that returns a value, like POSITION. A typical command that does not return a value might be a turtle command, FORWARD 100. Values of expressions returning a value, but which are not the final expression in a procedure, are simply lost.

If a value is returned from a command executed at top level, then that value appears on the same line as the command, but separated by a vertical bar, |.



```
gcd
input x y
if x = y then: x else:
  gcd  larger-of x y  -  smaller-of x y    smaller-of x y

larger-of
input x y
if x > y x y

smaller-of
input x y
if x < y x y

gcd 135 75   | 15
```

Figure 9. A greatest common divisor (GCD) procedure returns a value. If X = Y in GCD, then X is the last thing executed, and hence it is returned as the value of GCD. Otherwise, the value of another call to GCD is returned.



```
spf
input n
try-from-to 2 sqrt n

try-from-to
input test end
if  evenly-divides? test n   test stop spf
if  test >= end   n   try-from-to test + 1 end

evenly-divides?
input n m
0 = mod m n

spf 137   | 137
```

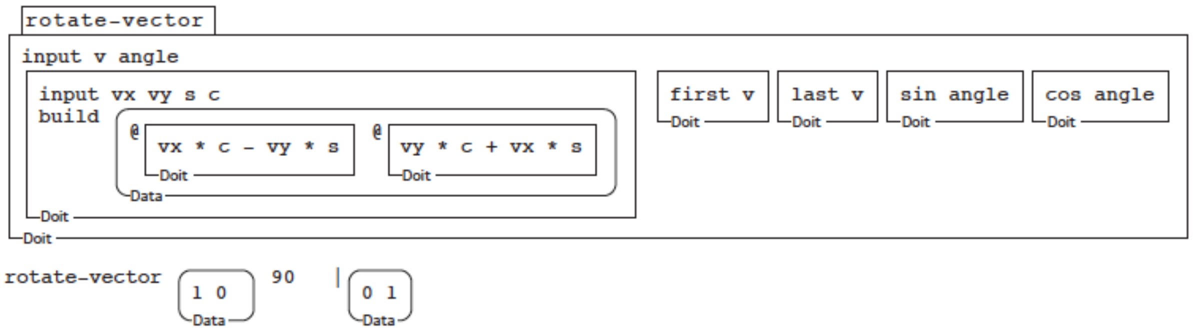
Figure 10. Smallest prime factor, SPF, sequentially tries candidate factors, starting at 2. If TEST evenly divides N in TRY-FROM-TO, TEST is returned as the last thing executed before SPF is stopped. Either the *then* or the *else* clause of the second IF will be the last steps executed in TRY-FROM-TO. So they don't need a STOP.

Uses of Doit Boxes

- Defining procedures: A doit box with a name defines a procedure that may be used anywhere in the box in which it appears, or in any sub-box. (See scoping section, below, for details and exceptions.)

- Local procedures: A named doit box may conveniently be placed inside another doit box to define a local sub-procedure. It is an excellent technique to structure code in this way, putting sub-procedures exactly where they are needed. See Figures 8, 9 and 10.

• Unnamed-procedures: A procedure that is only used in one place does not need a name. Just put the doit box in the line of code where it is needed (followed by any inputs that are necessary). If you need to compute the values of a number of local variables for temporary use, an unnamed doit box with an input for each needed local variable may be just the ticket.



```
rotate-vector
input v angle
  input vx vy s c
  build @ vx * c - vy * s  @ vy * c + vx * s
                                                first v   last v   sin angle   cos angle

rotate-vector  1 0   90   | 0 1
```

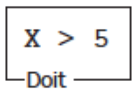
Figure 11. An unnamed procedure take four inputs—VX, VY, S, and C—to set up a local environment for computing. The BUILD command constructs compound data by replacing expressions preceded by @ with their values.

• Grouping: It is frequently both convenient and informative to group a set of commands together in a doit box to help a reader (or yourself) understand a piece of code. For example, although the Boxer REPEAT command can accept any list of commands, it is helpful to use a doit box in order to provide visual clarity, e.g. by placing commands in the REPEATed statement on separate lines. See REPEAT in Figure 7.

• Parsing: Boxer has rules to determine how things should be grouped in compound expressions, for example, whether 2 * 3 + 5 means 2 * (3 + 5) or (2 * 3) + 5. However, it is frequently useful to use explicit markings to show parsing. Parentheses are not meaningful in Boxer, but you can use doit boxes for this purpose. Boxes have the advantage of (1) being visually clearer, (2) you can't have an "unbalanced" box the way you may accidentally use only one parenthesis, and (3) you can expand or shrink a box to aid perusing code. Some Boxer programmers box every expression to eliminate all ambiguity and for other reasons (below).

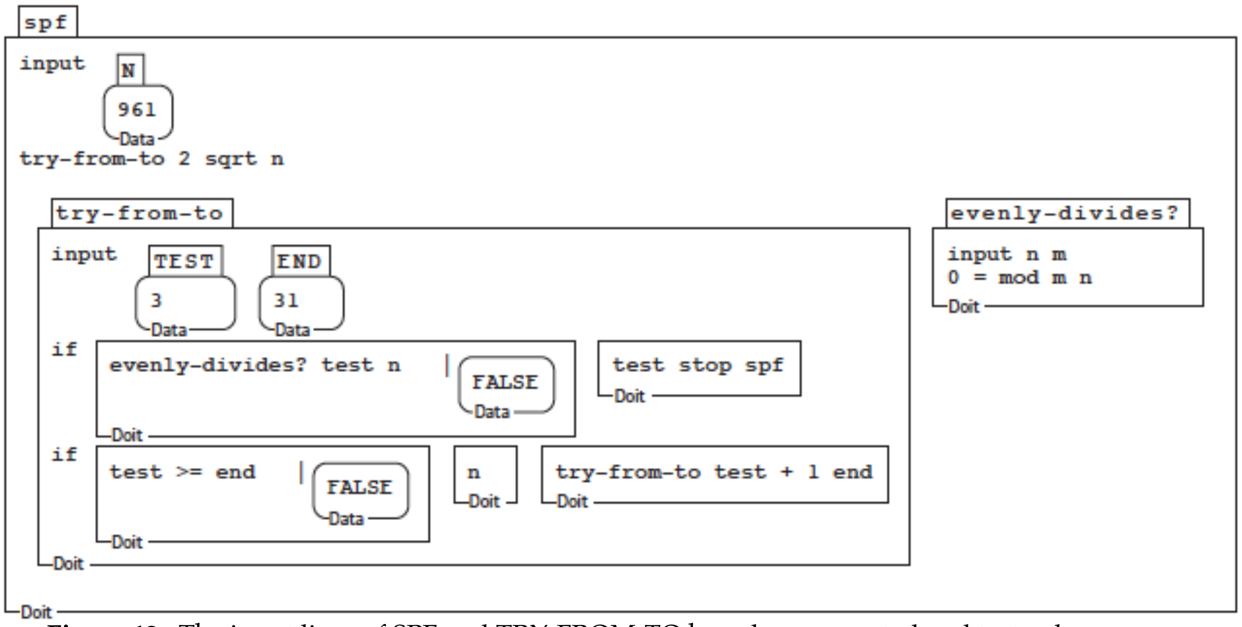
Note that, in general, you should enclose the *then* and *else* clauses of IF in doit boxes. Boxer generally expects this help to parse what you intend. The exception is when *then* or *else* clauses are single words, such as a variable, in which case enclosing doit boxes are not needed.

• Stepping: In Boxer, you can point to any line and execute it (e.g., with a double click) to see what it does or what value may be returned. If you box expressions, it makes it much easier to check pieces of a line by executing them. For example, while it is proper to write an expression "IF X > 5", if you write it with a boxed predicate,



```
X > 5
Doit
```

then you can simply point to and execute X > 5 to see whether true or false is returned. (Note that the | marker that separates an expression from its value acts as a comment marker; hence you needn't clean up stepped code by removing returned values.) As a convenience, you may execute the input line of a procedure during stepping, and the input names will be replaced with named boxes in which to place test values for stepping the body of the procedure. Again, there is no need to clean up these test values for normal execution. See Figure 12.



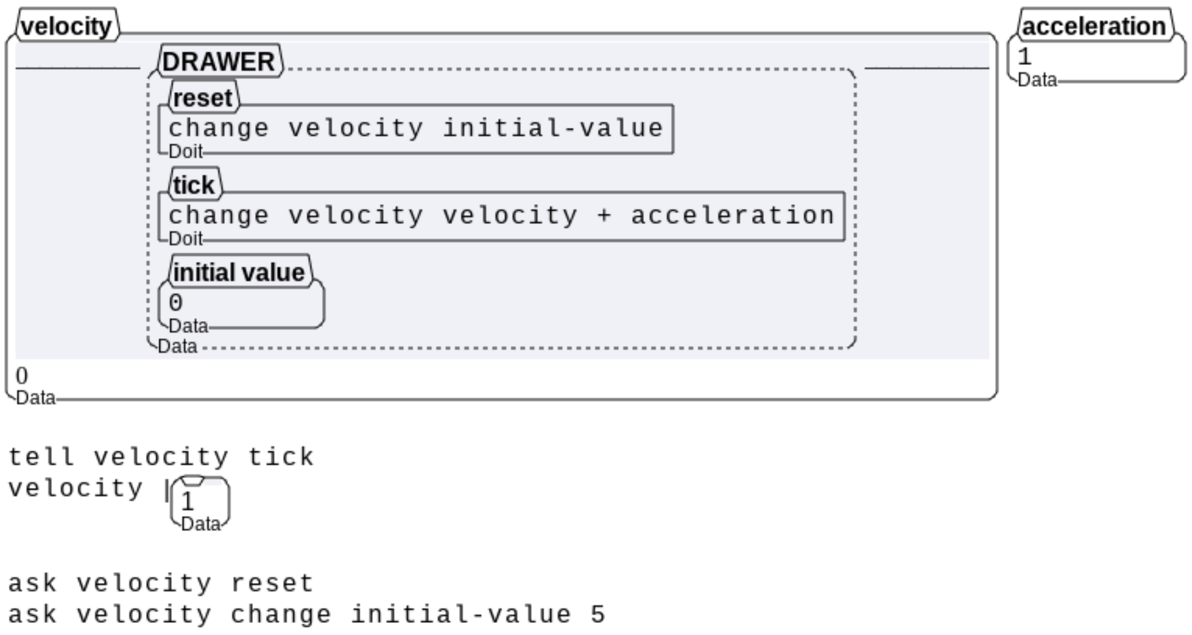
```
spf
input  N
       961
       Data
try-from-to 2 sqrt n

  try-from-to                                                evenly-divides?
  input  TEST   END                                          input n m
         3      31                                           0 = mod m n
         Data   Data                                         Doit
  if  evenly-divides? test n  | FALSE      test stop spf
                                Data       Doit
      Doit
  if  test >= end  | FALSE      n       try-from-to test + 1 end
                     Data       Doit    Doit
      Doit
  Doit
Doit
```

Figure 12. The input lines of SPF and TRY-FROM-TO have been executed and test values filled in. Then the predicates in each IF have been tested by executing them.

DATA BOXES

Data boxes define "information" in Boxer. In contrast to doit boxes, they are inert and, except when explicitly directed to, Boxer never "looks inside" a data box. When executed, data boxes provide a copy of themselves as information. Data boxes also make good containers, places in which to define microworlds or other sub-environments of Boxer. All data is passed into procedures (inputs) in data boxes and returned (outputs) the same way. (There are a small number of "for convenience" exceptions to this. You need not type numbers into Boxer inside data boxes, although Boxer will pass them around internally and return them as values inside data boxes. See also "flavored inputs" below.) Data boxes have rounded corners and, unless preferences are set otherwise, a label in the bottom border.

Data boxes may contain as much "stuff" (text, pictures, procedures, microworlds, etc.) as you like. This is a great source of flexibility and power, as examples below will show. The most common use of a data box is to provide a copy of its full contents (e.g., by executing its name), or to allow you to replace its full contents by some new contents. The command CHANGE changes a box's contents, as in CHANGE <*the name of a* *variable> <new contents>*. CHANGE does not change the closet of a data box. This allows you to keep many properties of a box the same—e.g., its boxtop (icon), some basic available information and procedures—while changing the rest of it. A variable may know what kind of data it is supposed to contain by placing a "type" variable in its closet, and it may contain generic procedures to operate on itself (e.g., a "reset" or "compute-value" procedure). Boxer has a great variety of ways of asking for pieces of a complex data box, or changing those pieces. Consult the Command Manual for details.



```
change velocity initial-value
change velocity velocity + acceleration
0
0
```

```
tell velocity tick
velocity | 1
           Data

ask velocity reset
ask velocity change initial-value 5
```

Figure 13. A "variable," VELOCITY, has been enhanced with objects in its closet to update (TICK) and RESET its value. VELOCITY can then respond to messages, but still returns values as an ordinary variable.

Uses of Data Boxes

• Variables: A named data box is a variable. It provides a copy of itself when its name is executed, and CHANGE can be used to set it. As with doit boxes, named data boxes may be global to a part of Boxer, or they may be placed as local variables inside procedures.

• Complex data: Since a data box may contain any combination of other Boxer structures, a great variety of data types may be created. Boxer provides primitive commands to access and change parts of data boxes: (1) as a list of items, (2) as a sequence of rows or columns, or (3) as an array with two indices (row and column). Beyond that, you can place named boxes inside a data box to develop more particular data structures. A data base, for example, is represented in Boxer typically as a list of unnamed data boxes (entries) each of which contain the same set of named data boxes (fields). See Figure 14. Consult the Command Manual for ways to access named or unnamed parts of data boxes (e.g., ASK, or "dot notation" explained in the special characters section below).

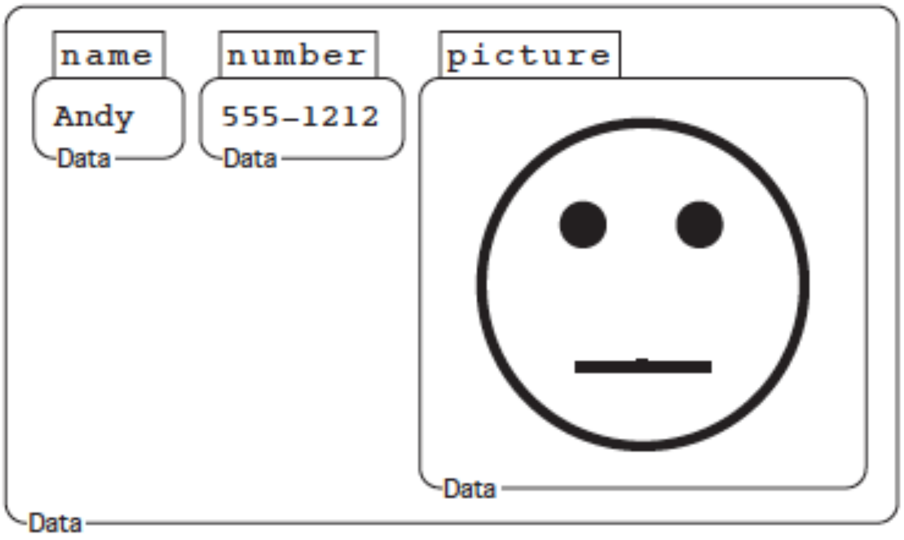


Figure 14. An entry in a database is simply a set of named boxes inside a box.

- Environments and microworlds: Most people put their individual work environments or microworlds in data boxes. (So executing one of these will provide a copy of itself as an alternative to using the copy action from Boxer's text processing capabilities.) Within a data box used as an environment a typical structure is to place (1) a graphics box for graphical input and output, (2) a data box with a list of commands to serve as a menu and (3) the set of procedures and variables on the bottom or top of the environment that you are building. See Figure 15. As you finish constructing your environment, procedures and variables will disappear to the closet or inside superprocedures, and they be replaced by a "read-me" data box with text or other means of documentation. Textual menus may be replaced with buttons, or other graphical interface objects.

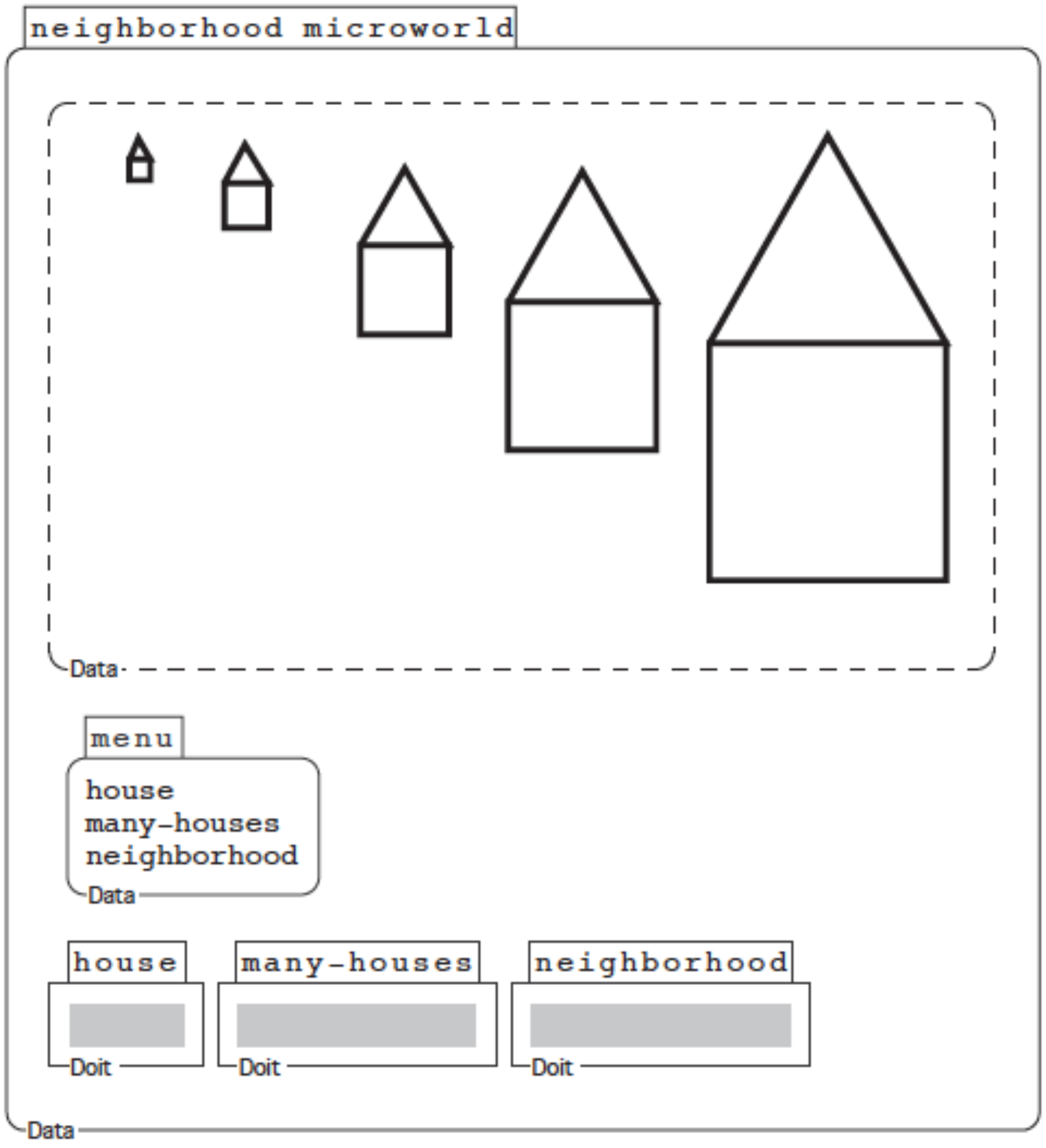
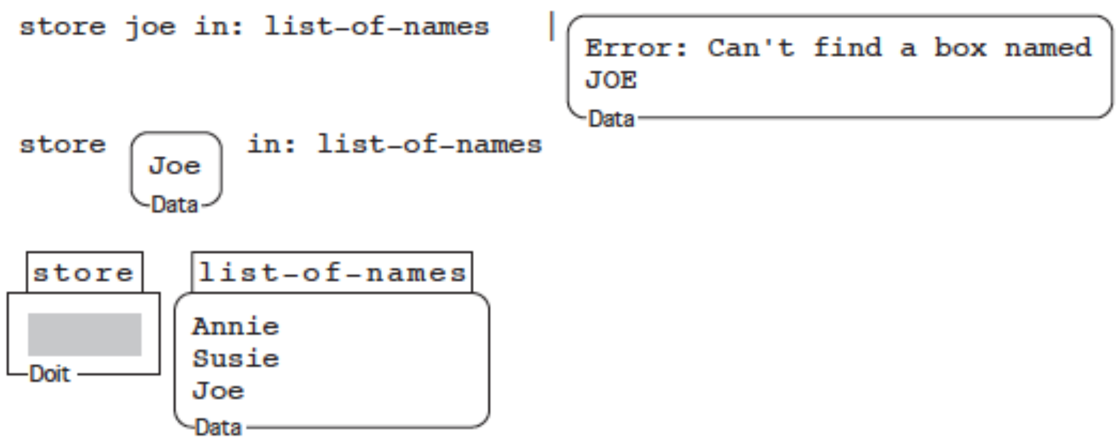


Figure 15. A typical structure during development of a microworld.

- Input and Output: Other than incremental drawing in a turtle box, changing a data box that happens to be visible on the screen is the most frequent means of visual output for programs running in Boxer. Boxer regularly and automatically updates the visual display to show current values, even if a program is running. Editing variables may also provide input for a procedure execution. This is less handy during the running of a procedure, although there are ways to allow editing during procedure execution. See EDIT-BOX, INPUT? and HANDLE-INPUT in the Command Manual.

- Quoting: Like most languages, Boxer interprets words in the text of its procedures as primitive commands, or as references to named variables or procedures. So, if you want to pass a literal word as input to a procedure, you need to "quote" it. Surrounding the word with a data box serves to quote it in Boxer; quote marks, per se, have no special meaning. See "flavored inputs" below, and the paragraphs just before it, for exceptions. Figure 16 has an example.



```
store joe in: list-of-names

Error: Can't find a box named
JOE

store Joe in: list-of-names

store

list-of-names
Annie
Susie
Joe
```

Figure 16. A data box “quotes” words and other text.

- Actors: Since a data box can contain arbitrary data and procedures, it can be used as an “actor.” The Boxer command TELL (synonym, ASK) can be used to send messages to these actors and to get information (returned values) from them. See Figure 13, the Programming Quick Start tutorial and also the Command Manual for details.

- Transparent boxes: Ordinarily, named objects like procedures and variables are accessible only inside the box in which they are defined. This is the right restriction to keep parts of different microworlds from becoming confused with each other. It is also appropriate to keep an actor’s personal data and procedures local. However, sometimes one might want to package a set of procedures or variables in a single box as a tool kit to place in an environment, adding all those names and capabilities to the environment. Transparent data boxes do this trick of letting names “leak out.” Note that supplied boxes in closets (“drawers”) are transparent, as are graphics boxes by default (see below). Transparent boxes are also excellent to group data and procedures of a complex environment into meaningful clusters, e.g., different drawers in a box’s closet. Transparent boxes usually revert to being non-transparent when copied because it is dangerous to have too many transparent boxes around. The Box menu contains an item to “export names” (turn a box transparent) or to “contain names” (make it non-transparent). [Shortcuts—Macintosh: command-e; Windows: control-e = ctrl-e) toggles the transparency of a box.]

GRAPHICS BOXES

Graphics boxes serve two primary purposes in Boxer. First, they are places where pictures appear and can be constructed. In this regard, graphics boxes are the fields in which sprites (the Boxer equivalent of Logo turtles) move around, draw lines, stamp shapes and text (see Figure 7 and below). Second, graphics boxes are designed to be the places in which you modify and extend the usual “text-editor” Boxer interface to arbitrary mouse and keyboard interactions that might be necessary or desirable for some applications. Graphics boxes have the label “data.” If you cut a graphic from another application and then paste it into Boxer , it appears as a graphics box.

It is important to remember, however, that graphics boxes are first class data in Boxer—they may be named, passed as inputs and returned as outputs to procedures, and they may constitute parts of more complex data objects.

Graphics boxes, unlike other boxes, have two visible presentations. First, they may appear in their standard graphical form. But they may also be “flipped” to reveal the ordinary Boxer box-and-text form in which to inspect or create any code or data that help define the properties of the graphical presentation. In the flipped form, the usual Boxer text editing mouse and keyboard interaction prevails. Among the prominent inhabitants of the flipped side of a graphics box are the textual form of sprites. Flipping may be accomplished by mousing the lower-left corner of a graphics box.

Layers

The graphical form of graphics boxes actually consists of three layers. On top are the shapes of any sprites that might exist in the box. Sprites may cover each other if they overlap. The order of insertion into the graphics box determines which sprite is on top. You can adjust the “depth” order of sprites with commands like push-backward, pull-forward, push-to-back, pull-to-front, as are commonly available in layered graphics editors.

Beneath the sprite layer is the graphics layer proper. This contains any of the lines drawn, shapes or text stamped by sprites.

On the bottom is the background graphics layer. Typical ways these three layers may be affected are listed below.

Sprite Layer - Add or delete sprites from the graphics box (generally done in flipped textual presentation). The various standard APPEND, DELETE or CHANGE commands can also add, delete or change the sprite-contents of a graphics box. Move sprites around with turtle commands. Change the shapes or sizes of sprites. Ask sprites to hide or show themselves.

Graphics Layer - Have a sprite draw as it moves. Have a sprite stamp its own shape, a set of built-in special shapes (circles, etc.), some text, or any arbitrary shape, which would be the image in another graphics box. You can clear the graphics layer all at once with a CLEARSCREEN (CS) command, or set it with a CHANGE-GRAPHICS command.

Background - You may set the background to any uniform color (SET-BACKGROUND), or clear it (CLEAR-BACKGROUND). In addition, the FREEZE command will press any current graphics-layer picture into the background where they will not be cleared by graphics CLEARSCREEN commands.

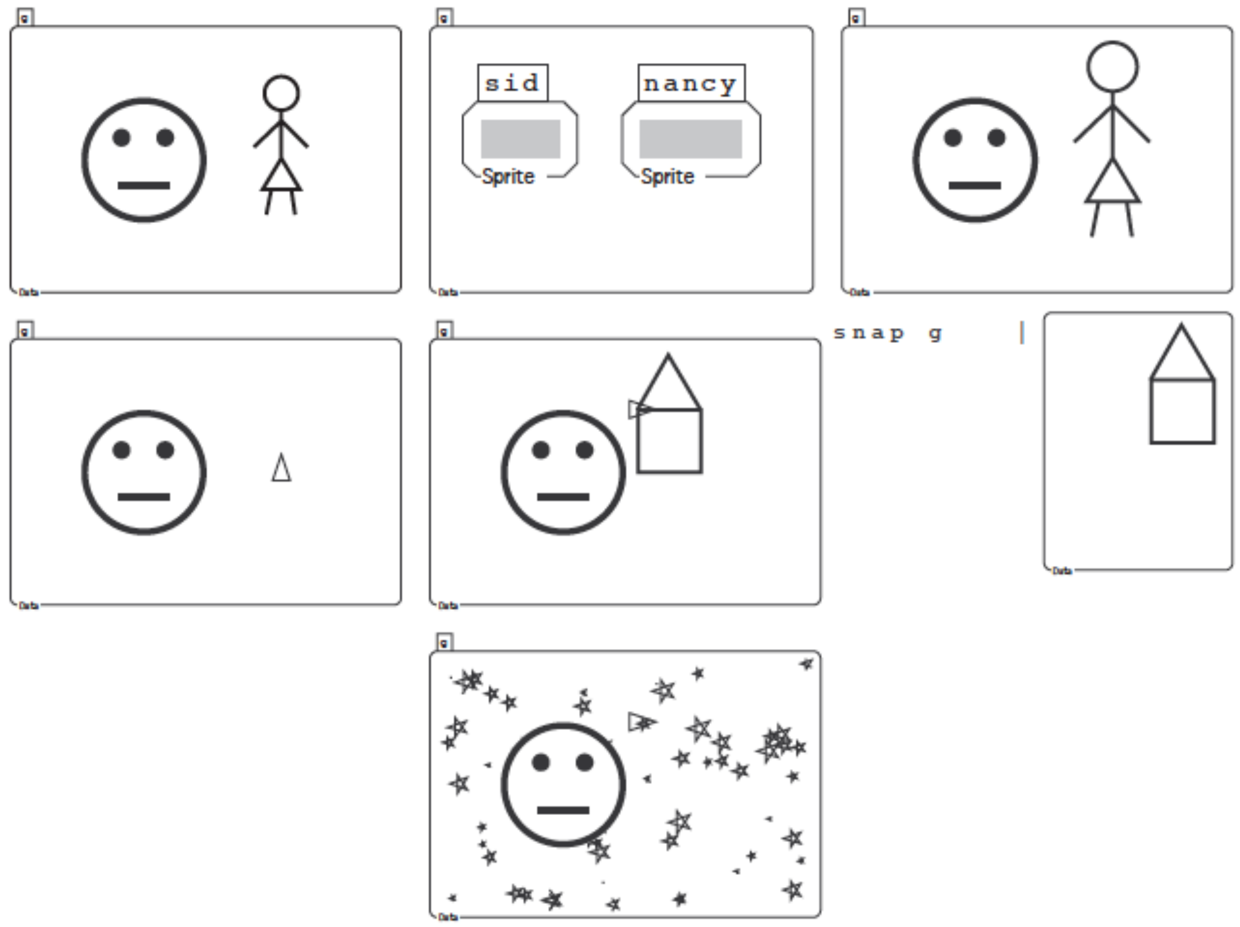


Figure 17. Left to right, top to bottom:
(a) Two sprites in their graphical presentation.
(b) The two sprites after the graphics box is "flipped" to the textual side.
(c) ASK NANCY SET-SPRITE-SIZE 1.7 (grow larger).
(d) ASK NANCY SETSHAPE TURTLE-SHAPE.
(e) ASK NANCY HOUSE (Nancy draws a house).
(f) SNAP G requests a copy of the graphics in a graphics box. (Sprites are not captured.)
(g) CHANGE-GRAPHICS G STARS (replaces current graphics with a new one).

Information from a Graphics Box

Graphics boxes are copied in full when accessed by name or when executed directly. The SNAP command returns only the graphics, without sprites or other internal structure. SNIP allows you to select a rectangular part of the graphics. The color of graphics, or of the background, at any point in a graphics box may be requested.

Ordinarily most graphics box commands that treat them as drawing fields [e.g., CLEAR, SET-BACKGROUND, and drawing mode commands (clip at boundary, or wrap)] need to be used with ASK <*graphics box*> <*action*>, unless you are inside the graphics box. However, a special form of a graphics box, a turtle box, is provided so that ASK is not necessary for most graphics commands that are addressed to either the box or to the sprite provided as part of a turtle box. (Turtle boxes are simply a transparent graphics box containing a transparent sprite.) As a default, graphics boxes are created in transparent form so the names of any internal sprites are "leaked out," and thus sprites needn't be addressed through their containing graphics box.

How to redefine mouse interaction in a graphics box will be explained later, in the section on Reconstructible Interface.

Uses for Graphics Boxes

- Pictures: Graphics boxes contain all graphical data in Boxer.

- Drawing field: Using the capabilities of sprites, one can incrementally draw pictures in a graphics box.

- Interactive field, container of interactive objects: Graphics boxes make good buttons, button bars, or more complex interactive IO fields. Some Boxer applications may present themselves only in a graphics box with interactive subcomponents (like sprite-constructed pull-down menus) and/or fields where one uses the mouse in a direct manipulation manner to configure or build with graphical entities.

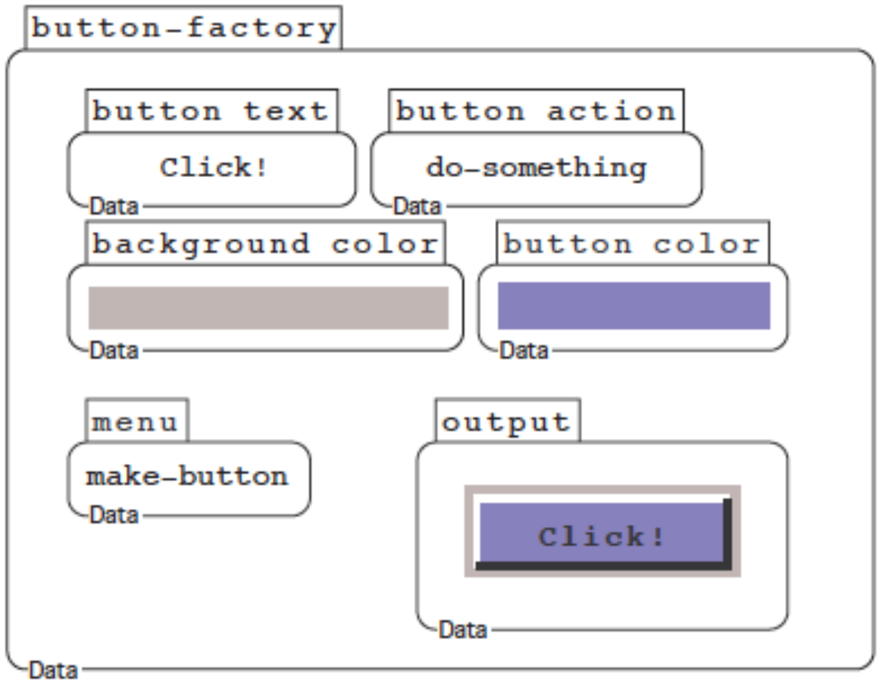


Figure 18. This "button factory" generates buttons (in the output box) that may be cut and pasted anywhere to activate any action with a click. This illustrates that interactive graphical objects in Boxer are first-class, can be returned by a program, but are also a normal cut-and-paste parts of the Boxer universe. Colors in Boxer are also graphics boxes whose editable and CHANGEable flip side contains red-green-blue percentages that specify the color. An optional fourth component in a color box specifies transparency, from 0 (completely transparent) to 100 (opaque).

- Interactive data objects: Since graphics boxes are first class and their contents may be changed with CHANGE, one may add new data types to Boxer that have their own graphical interface. So, for example, we can add graphical vectors to Boxer by building them in a graphics box. With reconstructible interface capabilities (below) one may allow users to create a new vector with a keystroke, and change the vector's visual presentation (say, an arrow) with the mouse. On the flip side of the graphics box, one may put the working representation, say, coordinate numbers. When CHANGE is used to change such coordinates, a trigger (below) changes the graphical presentation. Some local state and capabilities (say, a scale variable or mouse-activated DRAG-THE-VECTOR procedure) may be kept in the vector's closet. Finally, new commands may be added to Boxer that take vectors as inputs (e.g., a MOVE command that moves a sprite) and/or produce vectors as outputs (e.g., a vector addition command).

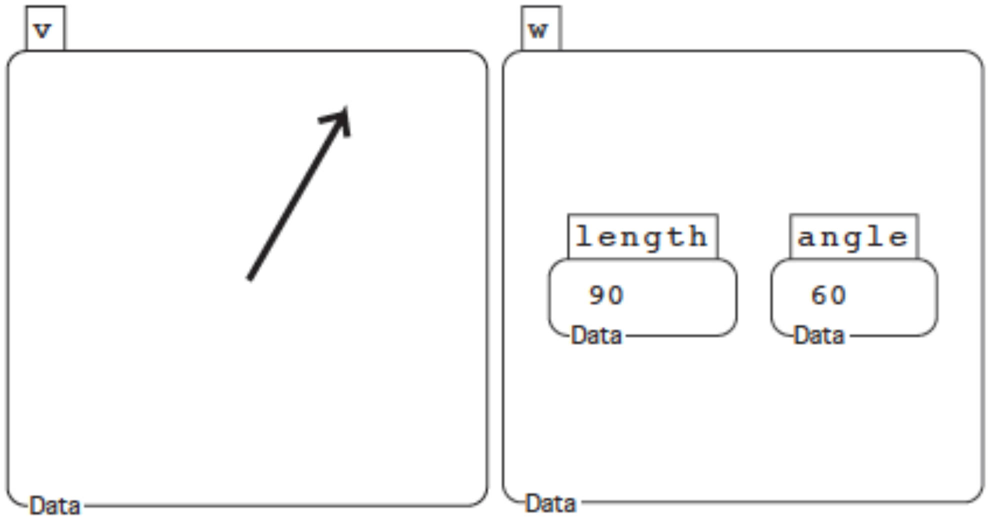


Figure 19. Objects like these vectors are constructible in Boxer. The graphics presentation is directly manipulable with the mouse. The "flipped" presentation shows internal representation. Vectors may be inputs or outputs to Boxer procedures.

- Region detection for sprites. One may query the graphics or background under a sprite for its color (COLOR-UNDER), hence determine something about where the sprite is located.

SPRITES

Sprites are mobile, interactive graphical objects in Boxer similar to Logo turtles. They have many built-in capabilities and properties, such as a pen that may be up or down, and that draws when it is down and the sprite moves. Sprites can stamp shapes, including their own, and type text. They have shape, size, position and heading sub-variables. Sprites can be queried about whether they are touching other sprites. See the Command Manual for a full listing of sprite properties and how they work. You may add data or procedures to customize the capabilities of a sprite.

Sprites may exist inside or outside of graphics boxes, but they will only be *graphically* visible from the graphical presentation of a graphics box in which they are placed. Ordinarily sprites must be addressed by name using ASK (TELL), e.g. ASK JOE FORWARD 20. Inside a sprite, you may execute commands for that sprite without ASK. Also, if you create a turtle box, the sprite that comes with it will be transparent, which means sprite commands will automatically get to the sprite (and properties like SHAPE and HEADING will also find their way out) without need to use ASK.

Subsprites:

Sprites may be placed inside other sprites, in which case they become part of that sprite. That means, if a sprite moves or is changed in size, any subsprites will move and expand or shrink right along. If a sprite is hidden, ordinarily subsprites will also be hidden. Technically speaking, a subsprite's position, heading and size (etc.) properties are all *relative to* its supersprite's. That is, the supersprite establishes the frame of reference for subsprites. Subsprites make excellent parts of more complex graphical objects, like parts of a face or the knob of a slider.

Sprites are first class data, so they may be passed into or returned as values of procedures. Unlike other data objects, sprites provide a port to themselves, rather than a copy, when executed. This is because, for example, when you use the name of a sprite, you likely mean to get access to that sprite (e.g., for use with ASK or TELL) rather than to have a copy of it. If you mean to have a copy—a new sprite—COPY will provide it.

Ordinarily sprites are created in lightweight (or "diet") form. That is, to save memory and shield users from some complexity, only some of the sprites' properties (position and heading) will appear in a sprite. The command SHOW-SPRITE-PROPERTIES (which appears in a diet sprite's closet) will bring them all into view. Individual properties will come into view when Boxer detects you need them (e.g., when you CHANGE an attribute directly). You may delete any properties you don't want to see; they will still work correctly.

Sprites will not appear in the graphics presentation of a graphics box if they are "insulated" by being placed inside *another* box inside the graphics box (unless their container is transparent).

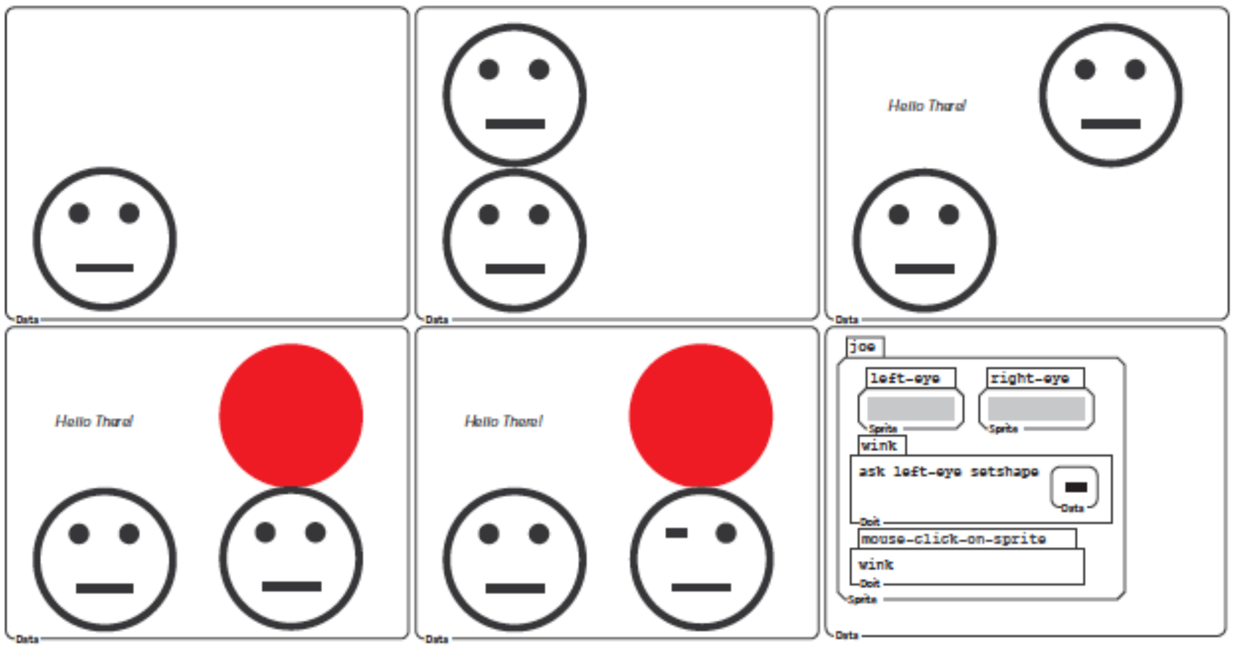
How to define sprites' reactions to mouse clicks will be treated below in the section on Reconstructible Interface.

Uses of Sprites

- Active parts of a picture: Sprites may be animals, people, clouds, trees, or any other part of a picture that you need to move or change (e.g., shape or size) independent of the rest of the picture.

- Drawing instrument: Sprites may draw or stamp to add to a picture in a graphics box.

- Interactive objects: Sprites are the right materials out of which to build clickable and/or movable objects in the field of a graphics box.



```
ask left-eye setshape
```

```
wink
```

Figure 20. Left to right, top to bottom:
(a) A sprite.
(b) ASK JOE STAMP-SELF HOP-UP.
(c) ASK JOE TYPE MESSAGE HOP-RIGHT.
(d) ASK JOE SET-PEN-COLOR RED STAMP-CIRCLE 50 HOP-DOWN.
(e) ASK JOE WINK.
(f) On the insides of Joe are a subsprite for each eye, a WINK procedure, and a mouse-click procedure that makes him respond to the mouse.

PORTS

Ports are views of another box (called the target of the port) from some other place in Boxer. In general, ports behave identically to their target, e.g., if you have both on screen at the same time, you may edit either and both change instantly. Ports are Boxer's only method of breaking the strict hierarchical organization of boxes within boxes; with ports you can have access to some remote part of Boxer from anywhere you like, and different objects may share parts with ports. The target of a port may only be a box; it cannot be a region of a box or another port.

One can make a port as one makes other boxes (key stroke or menu item), after which one must use the mouse to point out the target. Or, the PORT-TO <*box*> command will return a port to <*box*>. (However, if <*box*> is a doit box, port-to will execute it and expect a box in return to port to.) Ports may be named, expanded and shrunk independent of their targets. A port to a named box behaves like an unnamed box unless it is itself given a name.

Ports are "sticky" in the sense that when executed, they return a new port to the target. Ports may be passed as inputs to and returned as outputs from procedures. Stickiness also means that when Boxer data selection commands take a part of a port, they try to maintain as much contact as possible with the target. This may result in ports to parts of original target. Thus, FIRST of a port to a box containing a series of boxes will return a box containing a port to the first box in the original target.

Ports to doit boxes execute just like the target, except that any names in the doit box refer to names accessible from the target, not those accessible from the place of execution. In contrast, names in a regular doit box refer to objects accessible *in the place of execution*, not the place the procedure is defined.

RETARGET is the command that changes the target of a port, just like CHANGE changes the contents of a data box.

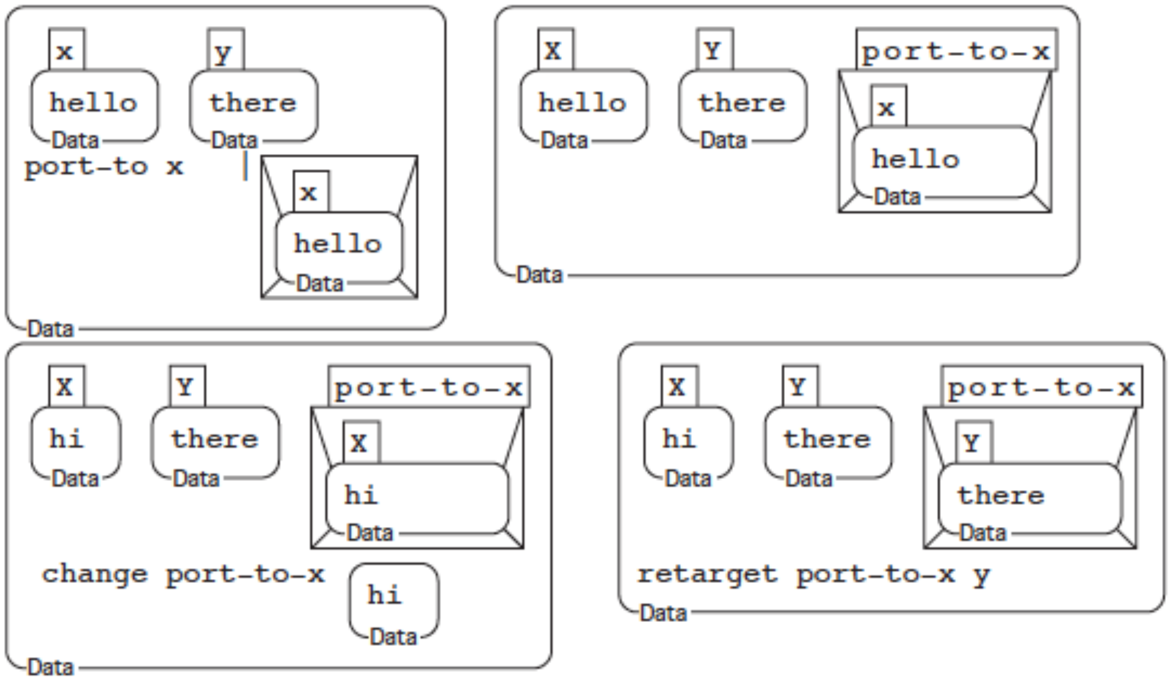


Figure 21. Left to right, top to bottom:
(a) PORT-TO returns a port to a box.
(b) Ports can, themselves, be named.
(c) Changing a port also changes its target, and vice versa.
(d) Ports may be retargeted to another box.

Uses of Ports

- An interface mechanism: Ports can make normally invisible parts of Boxer, such as a variable buried deep in a complex procedure, available for inspection and manipulation. You might want to port to a sprite, or to a whole graphics box, so that you can watch and edit in the ordinary way while at the same time observing the graphical presentation of the graphics box. If possible, Boxer provides you with a port to the offending code in an error message so you can change it directly in the port without having first to wander around to find the problem code.

- Hypertext links: Ports provide instant access to boxes, which may be considered nodes in a hypertext document. An interface command (key stroke or menu item) can "zoom" you to the target of a port, "traversing the link" to the context where the target is actually located.

- Shared data: You can create shared parts of two different objects with ports. For example in a database, Aunt Tilly and Uncle Bill might have the same phone number. If Uncle Bill's phone number in his entry in the database is a port to Aunt Tilly's number, whenever you change one number, both will be changed.

- Object access: Many times in Boxer you may want access to some particular data object rather than to have a copy of it, as the usual "information providing" behavior of data boxes. For example, you almost always want to talk to a particular sprite or data box used as an actor rather than talking to a copy of those boxes. Ports are frequently exactly the structures to use to obtain "object access." So, a list of objects to talk to may be a box containing *ports to* those objects. Or a port that you RETARGET may constitute a variable object to talk to or change. (Using names, instead of ports, to refer to objects to talk to may be inconvenient, for example, if you are creating sprites or other objects on the fly. Or using names may just be inelegant if a port to the object is really what you want.)

- Programming without names: To ensure there are no name mix-ups, you may want to use ports to variables or procedures instead of names.

- "Lexical scoping": The names in a procedure ordinarily refer to objects accessible at the point a procedure is executed. If, instead, you want the names to refer to objects accessible *from the location of a procedure's definition*, use a port to the procedure rather than its name. [Alternatively, name a port to the procedure (e.g., next to the procedure) rather than naming procedure itself]. Lexical scoping is less subject to mistakes of name confusion, since the place from which names are looked up is fixed, unlike the many different places a procedure may be executed.

THE COPY AND EXECUTE MODEL

In order to understand how Boxer executes a complex procedure, it helps to have a model of what is going on inside Boxer. Whenever a procedure is called by name, you can imagine that Boxer copies the definition of the procedure into the position the name occupied, and then Boxer continues execution in its normal left to right, top to bottom sequence. Any inputs are executed as the first thing a procedure does, and the values of these are placed as local variables in the procedure. Any returned values are left in place of a procedure when it is finished executing. This activity is called "copy and execute."

Since procedures in general call other procedures, the copy and execute model can build complex nested boxes within boxes, with local variables perhaps at each level. Older Boxers could show you the copy and execute model in simple cases as an optional kind of "doit" key press or menu selection. We are planning to re-implement it.

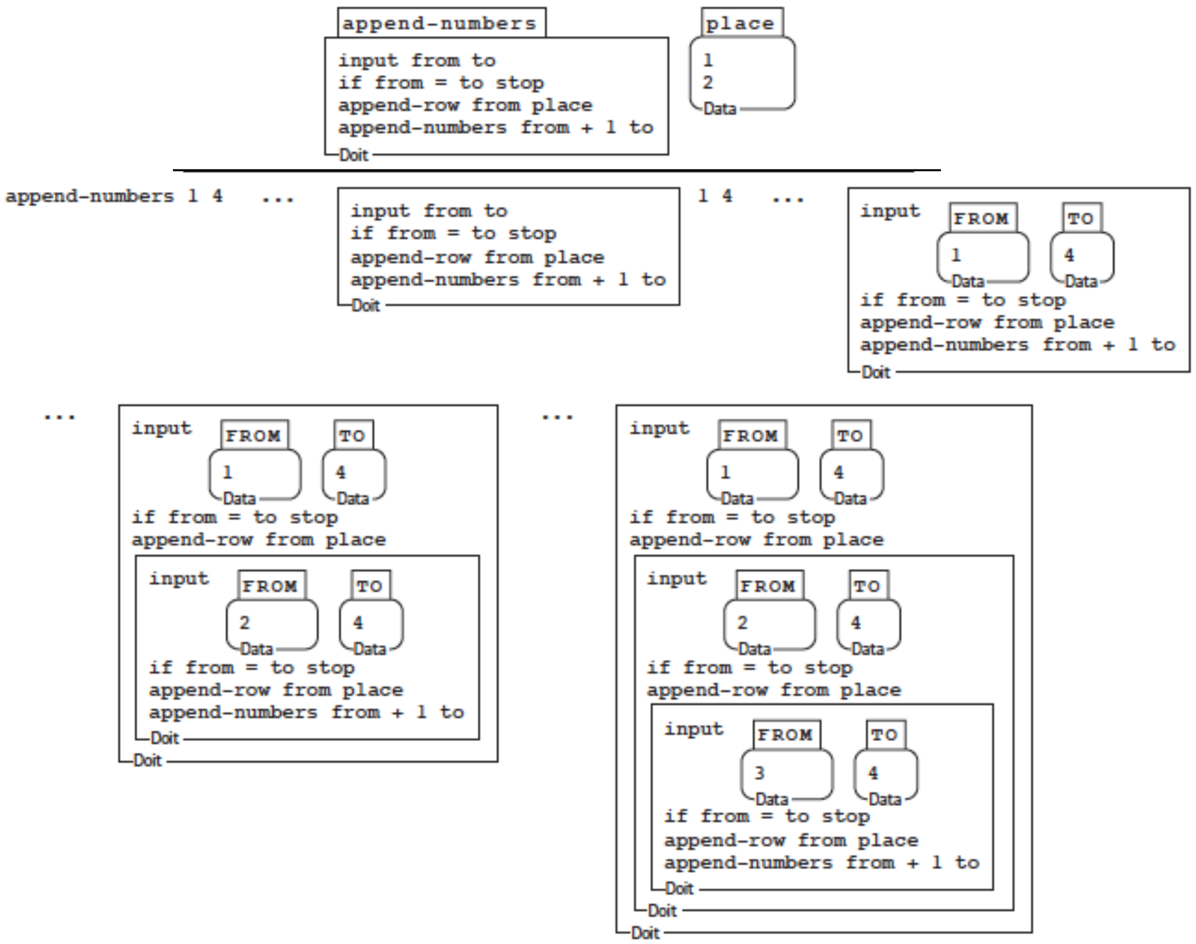
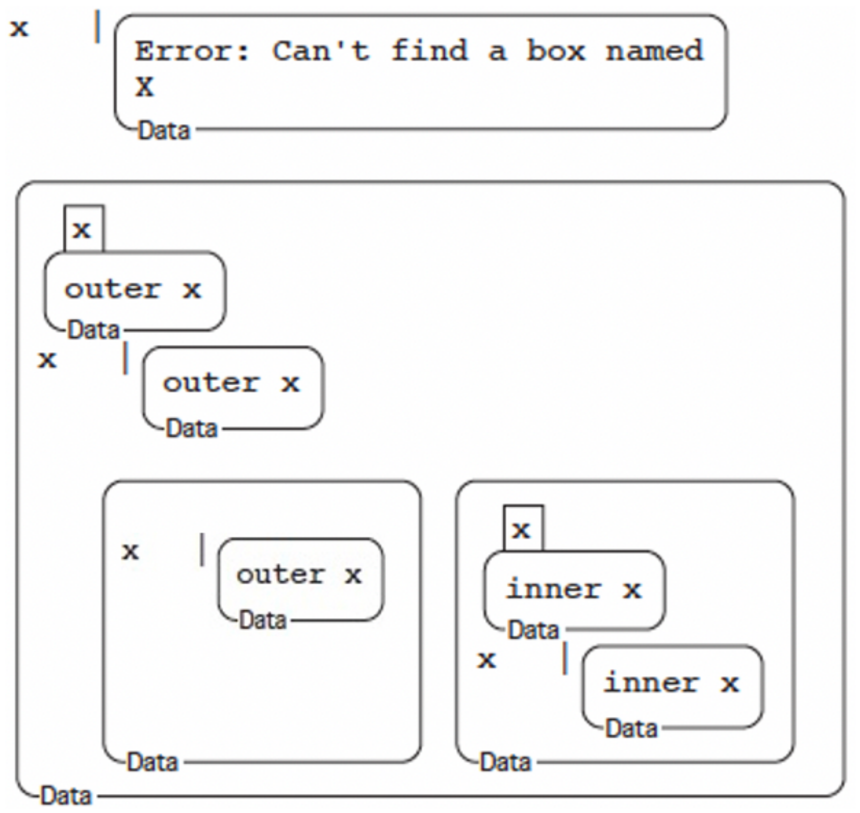


Figure 22. A procedure definition (top row) and snapshots from the “copy and execute” model of its execution. Snapshots are separated by “…”.

Scoping

The way that Boxer determines what data or procedure you are referring to when you use a name is called “scoping” (in the sense of defining the scope or range of a name’s effectiveness). Boxer’s rules are very simple. To find the object a name refers to, Boxer looks in the current box, including its closet. If that fails, Boxer checks the box that contains the current box, and so on. Primitives conceptually exist in the top-level box. Because Boxer looks “upward and outward,” named boxes closer to a given name use will be found and used before more distant boxes or primitives having the same name. This is called “shadowing.” A more local name box shadows a more global one, so the latter cannot be seen. You can ask Boxer to list the names of boxes in some box (NAMES), whether a name is accessible at all (NAME?), whether it is accessible but not primitive (LOCAL-NAME?), whether it is both accessible (i.e., *not* shadowed) and primitive (TOP-LEVEL-NAME?), or whether it exists in the present box (NAME-IN-BOX?).



```
Error: Can't find a box named
X
```

Figure 23. Scoping: Boxer looks locally, then "upward" in the box hierarchy to find names.

The copy and execute model together with the scoping rules above determine the way that names refer to boxes within the execution of a complex procedure, which may involve inputs and other local variables and procedures. For example, inputs of a certain name will always be used in a procedure even if there is a primitive by the same name. This method is call "dynamic scoping" because name access is determined by the dynamic context of procedure/subprocedure, not just by the "static" context where a procedure is defined.

Parsing and Syntactic Issues

The copy and execute model does not explain exactly how Boxer determines what the inputs of a procedure are in an expression that starts with the name of the procedure. For simple expressions, Boxer will usually do what you expect. For more complex expressions, there are many ambiguities. It is impossible to develop a simple set of rules that are both easy to learn and compatible with familiar conventions such as precedence in arithmetic expressions, e.g., a + b * c = a + (b * c), not (a + b) * c.

For this reason, it is usually more productive to write unambiguous expressions rather than to learn Boxer's detailed rules. The following rules of thumb are helpful.

(1) In general, avoid putting more than one command on a single line. This avoids having the second command and/or its inputs interpreted, somehow, as an input of the first. [For example, ASK (TELL) grabs everything to the end of the line it appears in as the message. So placing an unrelated command at the end of a line containing an ASK generally just won't work properly.]

(2) Grouping complex inputs in doit boxes clarifies how an expression should be parsed. As we pointed out earlier, this also allows easier checking by running parts of a procedure (e.g., executing an input by itself to see the value returned), and it also helps you peruse code by expanding or shrinking parts at need. See Figure 12 for an example.

Besides parsing, there are two syntactic issues worth mentioning. The first is that some commands expect inputs in a certain format. IFS, for example, expects a data box with a predicate (an expression that returns either true or false) and an action on each line. Although we've made such commands as few as possible, and designed their input forms to be as simple as possible, there is really no alternative to either learning the form or keeping documentation for it handy. Consult the Command Manual.

Finally, we mentioned that data in Boxer essentially always should appear in a data box. This means that names that do not represent doit boxes for immediate execution nor data boxes for immediate copying should be placed in a data box to "quote" them and keep Boxer from executing them. For example, DELETE JOE should really have JOE in a data box, for this expression does not mean to copy or execute JOE.

On the other hand, experience shows this is frequently both inconvenient and confusing for novices. Hence, we've provided mechanisms, called "flavored inputs", that avoid the need for quoting. The general rule is that if a command would ordinarily need a quoted input, but (1) the command is likely used by relative novices, and (2) it will rarely be used in the case the input needs to be executed, then you may use the non-quoted name in place of enclosing it in a data box. NAME-HELP, which provides information about names of primitives or of user-defined boxes, is a prototypical example. NAME-HELP SPRITE will tell you something about all objects that have SPRITE in their names. Check ambiguous cases for using unboxed inputs in the Command Manual.

FLAVORED INPUTS

To increase Boxer's flexibility in the hands of advanced users, especially to allow them to develop materials for beginners that make minimal demands in understanding Boxer's structure, we have included a number of variations on standard input protocol. Ordinarily, Boxer expects to execute what it finds as an input to a procedure, and it expects to find a data box as the resulting value. To get useful variations of this behavior, you may use the adjectives PORT-TO, DATAFY, or BOX-REST in the input line immediately before the input name that you want to behave differently.

port-to (e.g., "input port-to x"): The port-to flavor of input is the most powerful and important alternative to the standard flavor. It makes a port to the data box provided as input, instead of providing a copy for the internal use of the procedure. Consult Figure 24. For example, if you have a procedure with input line "INPUT PORT-TO VARIABLE" then if the procedure is called on a data box, X, the procedure will have a port to X (called VARIABLE) to use internally. Thus, if you "CHANGE VARIABLE ...", X will actually be changed. This makes VARIABLE a transparent pseudonym for X if you want to ASK or modify X. The primitive CHANGE itself uses a port flavor for its first input.

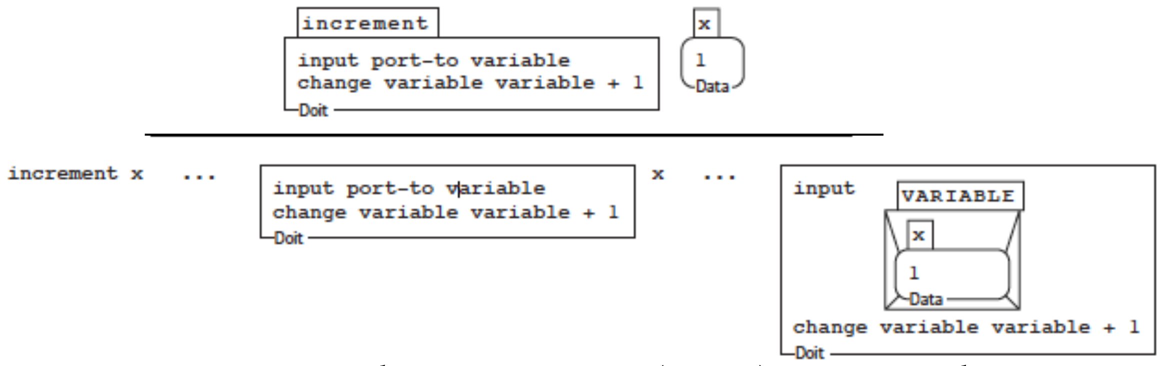


Figure 24. A procedure with a port flavored input (top row) and some snapshots from the "copy and execute" model of its execution.

Port-to flavor is not very useful when a procedure is used as input. Boxer will in that case execute the procedure, and then try to make a port to the result. But if the procedure returns a port of its own accord, the input would be a port, even without use of port flavor (via the stickiness property of ports). On the other hand, if the procedure returns a data box, then that box will itself be a copy of another box or a brand new box created by a primitive; there is little use in porting to it since, at that point, the box doesn't exist in any place in Boxer that will continue to exist after procedure execution stops. This is not illegal, but Boxer ordinarily warns you to suggest you have written a peculiar program.

datafy (e.g., "input datafy x"): Datafy inputs do not execute anything, but they take exactly one word or box and wrap a data box around it to create the input variable. Their main use is to allow one to input a word or doit box into a procedure without having to take the trouble (or to remember to take the trouble) of quoting the input by enclosing the word or doit box in a data box. See Figure 25.

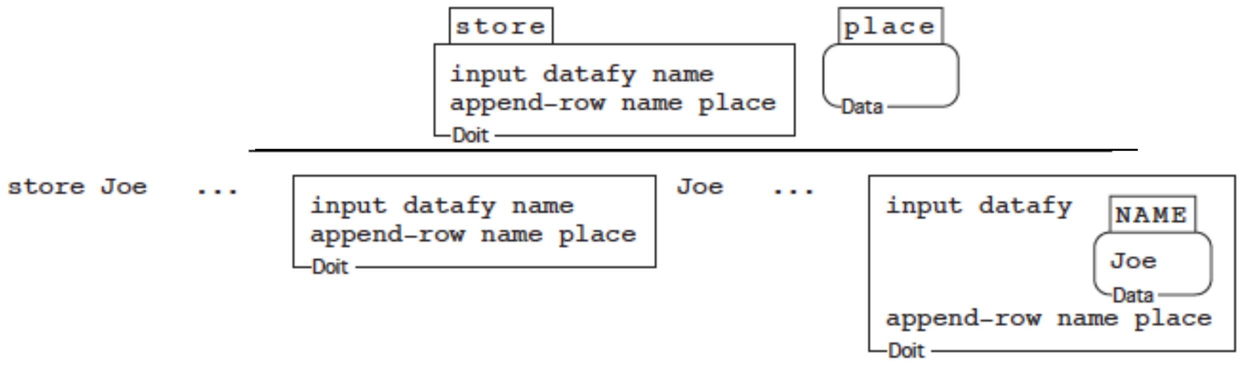


Figure 25. A procedure with a data flavored input (top row) and some snapshots from the "copy and execute" model of its execution.

box-rest (e.g., "input box-rest x"): Box-rest is similar to datafy, except it takes the entire remaining part of the input line and places it in a data box. It is useful when you need to look at the full input to decide what to do, for example, when you want to simulate a procedure that takes a variable number of inputs. A variable number of inputs is otherwise not possible in Boxer. Because box-rest gobbles the rest of the line, it only makes sense as the last or only input to a procedure. See Figure 26.

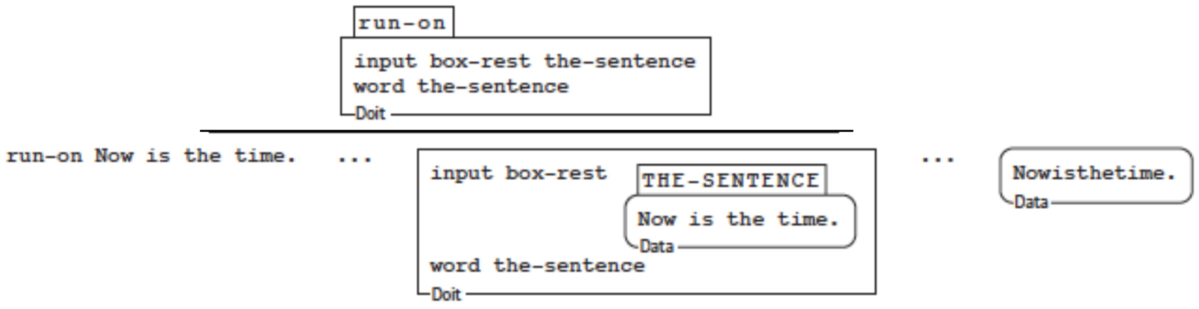


Figure 26. A procedure with a box-rest flavored input and some snapshots from the "copy and execute" model of its execution. (WORD joins a sequence of words into one.)

SPECIAL CHARACTERS

Boxer reserves some characters for special purposes, and hence these will behave differently, for example, as parts of names.

. Periods are used in Boxer to separate the names in a chained sequence of names. Hence X.Y.Z refers to the Z found in the Y box that is found in the box named X. Only the last of these named boxes (here, Z) may be a doit box.

: Colons are used in Boxer for comments within a command line; words ending in a colon will be ignored by Boxer when it executes a line of code. They are most useful to label inputs, e.g., FORWARD STEPS: 100. If you forget the names of inputs to a procedure, Boxer can provide them. The input prompt is under the HELP menu, or see the interface part of the Command Manual for keystroke activation.

; Largely for compatibility with other languages, Boxer uses ";" as a comment character. Hence Boxer ignores what appears after a semicolon on a line when executing code.

| The vertical bar will appear in Boxer when you execute a line and a value is returned. It separates the command executed from the value returned. If the command is executed again, any new value will replace the older one after |. | acts as a comment character in the midst of code, so you don't need to clean up when you test the returned values of part of a more complex procedure.

@ At-sign is a way to have Boxer compute parts of a line before executing it. The copy and execute model for @ is that, when Boxer finds itself executing a line containing @, it first executes all the expressions preceded with @, leaving their (unboxed) returned values in place in the line. Then Boxer executes the line thus created. See Figure 27. This is equivalent to RUN BUILD <*a data box containing the full line in which @ occurs*>. See BUILD in the Command Manual. At-sign is, in fact, a convenient extension of its use in BUILD (see Figure 11).

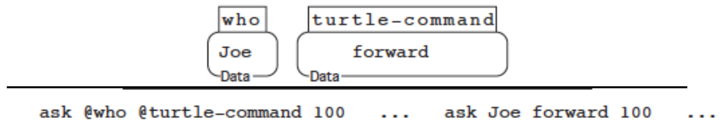


Figure 27. Snapshots from the "copy and execute" model for @.

The most common uses of @ are:

• To compute parts of a line of code before executing it, as in Figure 27.

• To "undo" the effects of a datafy input. So (consult Figure 28), if F has a datafy flavored input, F NAME will not execute NAME, but pass "NAME" in a data box into F. F @NAME, in contrast, will execute NAME, unbox the result, and then F (working as usual) will get as input variable that object—typically a word— in a data box. If NAME is a data box containing "Harvey," F @NAME will result in F having a data box containing "Harvey" as input.

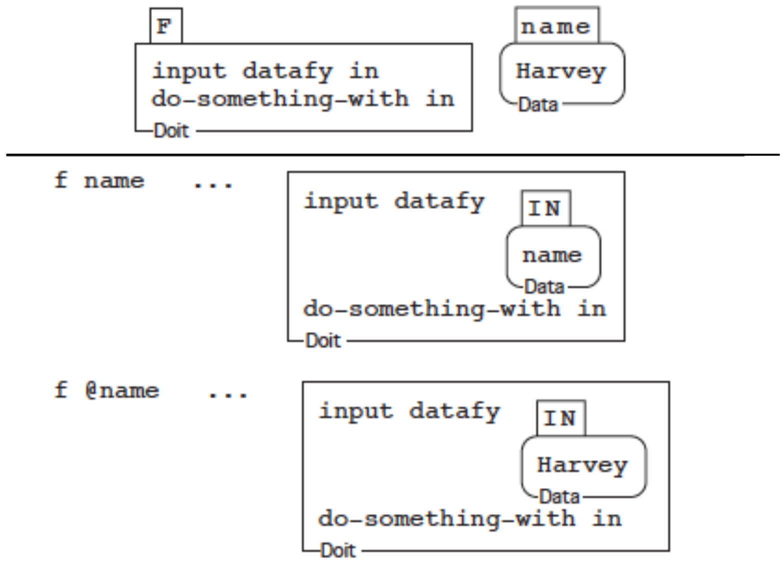
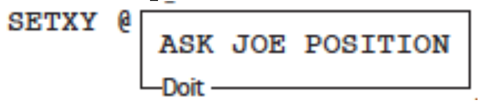


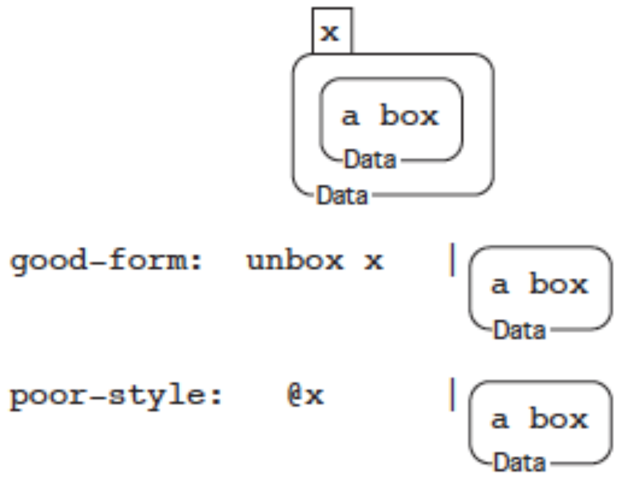
Figure 28. A procedure with data flavored input (top row): contrasting a normal call (second row) with use of @ to "undo" the data flavor (last row).

• To unbox a box that contains multiple items for a procedure that needs multiple inputs. For example, POSITION returns a box containing the x and y coordinates of a sprite. SETXY needs two inputs, so you might execute SETXY @POSITION. A more realistic use would be to tell a sprite to set her x and y coordinates to those of another sprite as in:



• As a replacement for the command UNBOX (which removes one data box wrapper from a piece of data, *provided* that data contains exactly one data box).

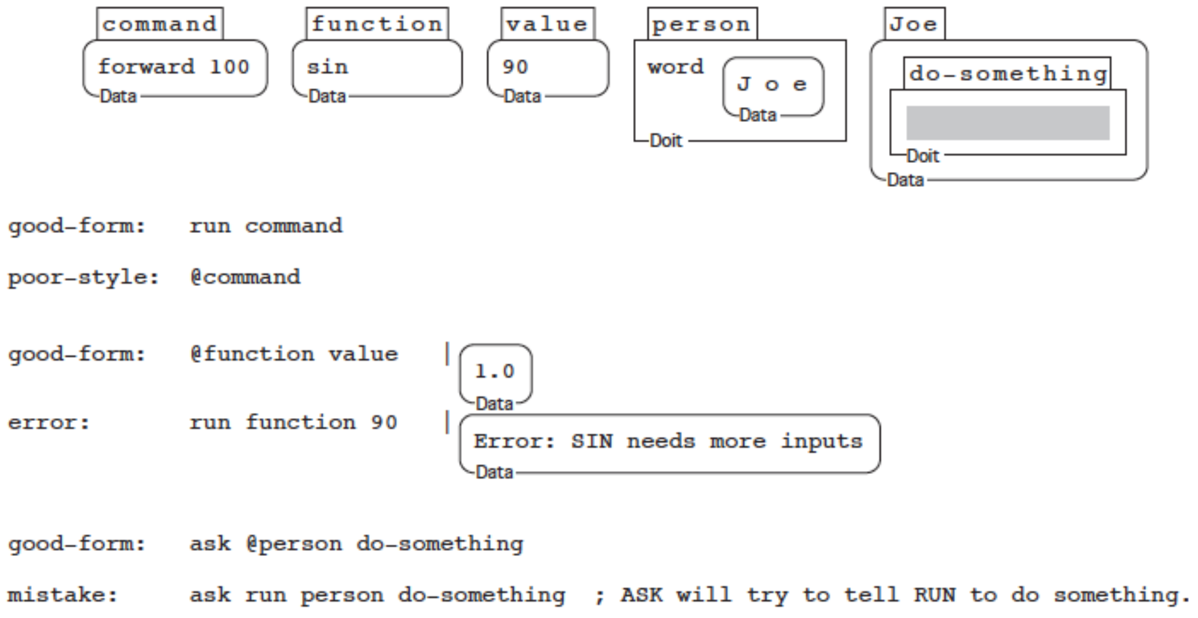
This usage is generally to be avoided stylistically, as @ is more powerful than UNBOX (it causes two executions—the execution of the item it precedes and then the execution of the resulting line). Such complexity, if you don't really need it, can lead to difficult-to-track-down bugs.



```
good-form:  unbox x   | a box
poor-style:   @x      | a box
```

Figure 29. Unbox and @ appear to work similarly, but @ involves two executions. Here, the second execution just copies the data box resulting from the first.

- As a replacement for RUN under some circumstances. (RUN executes a data box as if it were a doit box.) In particular, @ is useful when you want to unbox the result of a RUN, or when you need to have the unboxed result of an execution interpreted in the context of other items. For example, if the variable PROCEDURE contains the name of a procedure (say, F) that needs an input, RUN PROCEDURE X will result in an error. @PROCEDURE X will, in contrast, execute F with input X. See Figure 30 for examples.

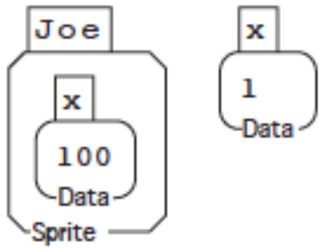


```
good-form:    run command

poor-style:   @command

good-form:    @function value    | 1.0

error:        run function 90    | Error: SIN needs more inputs

good-form:    ask @person do-something

mistake:      ask run person do-something  ; ASK will try to tell RUN to do something.
```

Figure 30. Some contrasting uses of @ and RUN.

! Within the input to a BUILD, an exclamation point before a name or doit box denotes “evaluate this expression.” Unlike @ in this usage, ! does not unbox the result, which is therefore always a data box (including graphics and sprites). See the Command Manual documentation for BUILD.

^ The caret is a special symbol designed to use with ASK (TELL). For example, ASK JOE FORWARD X has JOE use *his* X, while ASK JOE FORWARD ^X has JOE use *the X available where ASK appears*. In more detail: ASK <*who*> <*what*> ordinarily takes the entire message <*what*> literally and executes it in the place <*who*>. This means that any procedure or variable names in the message refer to those procedures accessible from the position of <*who*>. If you want to use procedures or variables from the place that ASK is executed, you must precede those names with ^, meaning “the local one” or “from here.” The most frequent use is in a procedure with an input, say X, that you want to pass on to a sprite, say, JOE. Hence you would say ASK JOE FORWARD ^X to avoid JOE using his X, or some other X that is meaningful inside JOE. Figure 31 illustrates.



```
long-line:  ask joe forward x

short-line: ask joe forward ^x
```

Figure 31. Using ^ to send a variable that is accessible at the location of ASK to the remote place of execution (the place ASKed).

TRIGGERS

Boxer has a trigger mechanism that allows actions to be taken automatically whenever certain events occur. Specific events that can trigger actions might be: when a user (a) enters a box (i.e., places the typing cursor in it), (b) exits a box (moving the typing cursor out of it) or (c) when a box is modified (either by direct editing or with any of commands like CHANGE). On each such event, one can arrange for a designated Boxer procedure to be set in action. To set a trigger, all you need to do is put a box with a special name, ending in -TRIGGER, in the box on which the trigger is to be set. The -TRIGGER box will usually be most conveniently placed in the closet of the box to be effected. Trigger names are “magic” (reserved for special meaning) like other Boxer interface hooks (see next section).

Triggers are local; they don’t scope in the way usual doit or data box names do. So if you set a trigger on a box, it will *not* be set on all contained boxes. There is no provided mechanism especially to turn triggers off and on. But you can do this with a number of other Boxer structures. For example, you may use a “flag” variable, which is true or false, and which triggers check to see if they should carry out some action. Finally, the name tab counts as part of a box for entry and exit triggers.

entry-trigger: A box named entry-trigger will be executed whenever the typing cursor enters the box in which it is defined. A typical use might be to start some process whenever a user enters a microworld or some part of a microworld. The process might run some animation, offer the user some help, or it might be a "monitor" program that interprets the user's actions specially in this box. Entry triggers are run only when you enter the main area of a box, not when you jump into a subbox.

exit-trigger: A box named exit-trigger will be executed whenever the user moves the typing cursor out of the box in which it is defined. A typical use might be to "clean up" after a user's use of a box, or to notify other parts of an environment about what a user has done in a certain box. Exit triggers are triggered even if you are in a subbox of the triggered box and click directly outside the triggered box.

modified-trigger: A box named modified-trigger will be executed whenever a user changes a part of a box, or when any Boxer function that directly changes a box is executed (as in CHANGE, CHANGE-ITEM, APPEND-ROW, etc.). While a user is editing a box, the modified trigger will be postponed until the user exits the box that is being modified. Like exit triggers, exiting a box from a subbox still activates the trigger. Modified triggers are powerful and useful. With them you may:

- cause a box to behave like a cell of a spread sheet, initiating an update process whenever the user makes a change.
- have a "monitor" that checks that a user's edit of a box is of the appropriate form. If it is not of the appropriate form, the procedure may fix the user's input or return to a prior valid value.
- enforce that different parts of an environment are changed together. For example, a variable "temperature" might have a modified trigger to change the graphical image of thermometer. Boxer's own sprite properties keep the textual representation of those properties and graphical representation of a sprite in synch with modified triggers.
- build a simple debugging tool that prints out some part of the state of a program whenever a certain variable is changed.

THE RECONSTRUCTIBLE INTERFACE

Boxer is designed to allow users a wide variety of ways to extend or modify the provided mouse and keyboard interface. These are richest in graphics boxes, which are intended to be the locus of specialized "poke and drag" interfaces for particular applications. However, one can also redefine mouse and keyboard actions in the context of non-graphics boxes.

The protocol for all these changes and extensions is the same. Whenever the user presses a key or clicks a mouse button, Boxer executes a specially named "interface message" command, such as a-key, mouse-click, mouse-double-click, etc. If the user has not defined a procedure or variable by that name, Boxer carries out the standard associated action or gives a warning if there is no default action. If there is a user-defined box with the appropriate name, that is executed instead. Thus, knowing how to change the Boxer interface comes down essentially to knowing the list of "magic names" that serve as interface messages.

Redefining Keys

The magic suffix -key in the name of a box redefines the action that Boxer takes when you press a key. Thus, a procedure may be named a-key or x-key, and it will be executed instead of the ordinary text editor "insert character" action whenever that key is executed within the scope of the -key procedure. Since Boxer does not distinguish case in names of boxes, the special prefix CAPITAL- should be attached for capital letters, e.g., capital-a-key. Control keys operate in the same way, e.g., Macintosh: command-a-key, option-b-key; Windows: ctrl-a-key, alt-b-key. The actions taken on pressing special keys, like RETURN-KEY and DELETE-KEY, also may be redefined. The same holds for function keys like F1-key, etc. A value returned from a -key execution (e.g., the -key box might be a data box) will be placed where the typing cursor is located when the particular key is pressed; ∣ is not used in this case.

Since it may become difficult to type in a box whose keys are being redefined, a special provision is made to temporarily suspend key/mouse interface redefinitions. A menu item, "Disable Key/Mouse Redefinitions", gives one the usual key and mouse bindings, ignoring any redefinitions. "Enable Key/Mouse Redefinitions" then turns the new definitions on again.

Uses of Redefining Keys:

- Add or change a function key: Frequently an application may need quick access to a set of functions, for example, a key to reset a simulation. Or you may bind new data types that you have defined in Boxer (e.g., vectors) to keys for easy user access to them.

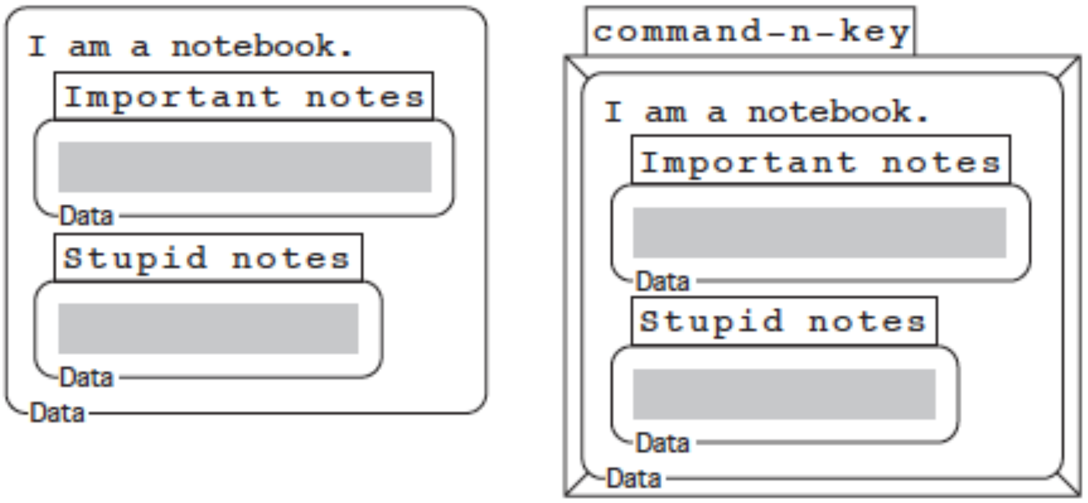


Figure 32. Pressing command-n will execute command-n-key and bring up a port to the notebook anywhere you wish. (A Windows version should use ctrl-n-key instead of command-n-key.) After editing the port, just delete it; the notebook box, itself, will not be deleted.

- Templates: Naming a variable with a –key name makes that variable instantly appear on pressing the specified key. This is a good way to make templates available, say, for entries in a database. If you want to edit a variable and have changes remain in the original (see Figure 32), bind the key to a port to the variable. This is a simple way to create a notebook that is available instantly on demand; deleting the port that is created by a key press leaves any changes in the original variable.

- Real-time, key-stroke controllers: Sometimes it is useful to have key strokes instantly cause actions, say in a video game or in a simplified turtle drawing system for a pre-schooler. Redefining keys in a particular box is an excellent way to do this.

Suggestions for redefining keys:
In general, it is better to redefine function keys, especially ones that are not ordinarily needed by Boxer users—rather than alphabetic keys—so as not to interfere with normal keyboard operation of the system. Keep other keystroke redefinitions isolated in boxes used mainly as places in which "instant" actions are intended to occur. Use Boxer INPUT? and HANDLE-INPUT commands to have keystroke redefinitions active while another program is executing, e.g. for real-time control. See the Command Manual for a fuller list of magic names, details and examples of methods of using key redefinition.

Graphics Box Mouse Redefinition

The magic names MOUSE-CLICK-ON-GRAPHICS and MOUSE-DOUBLE-CLICK-ON-GRAPHICS are executed when the mouse is clicked in a graphics box but not over a sprite. Prefixes may be added (Macintosh: COMMAND- or OPTION-; Windows CTRL- or ALT-).

These commands are in effect only when the graphics box is in its graphical (not textual) presentation. There may be default actions taken on some of these clicks, unless redefined by the user.

-CLICK-ON-GRAPHICS commands get executed in the graphics box to which the mouse is pointing when a click or press is given. So different graphics boxes may easily have different actions. On the other hand, normal rules of scoping apply, so graphics mouse definitions in a certain box will—unless there are local graphics mouse definitions in individual graphics boxes—affect all graphics boxes inside the box in which the definitions are made.

Clicks can initiate extended actions, and, for example, those enacted procedures may check the state of the mouse button(s) to distinguish a click (MOUSE-BUTTONS = 0, i.e., the button was released) from a press (MOUSE-BUTTONS > 0). Similarly, the instigated action may check the state of the mouse buttons to know when to terminate (e.g., a drag operation). INPUT? and HANDLE-INPUT may be used in a procedure in order to have mouse clicks active while some other process is running. See the Command Manual for details and examples.

Sprite Mouse Redefinition

Redefining actions when a sprite is clicked works identically to graphics mouse redefinition, except (1) the magic names use "SPRITE" instead of "GRAPHICS" and (2) the corresponding interface message is executed inside the clicked sprite.

MOUSE-CLICK-ON-SPRITE   MOUSE-DOUBLE-CLICK-ON-SPRITE

Command prefixes (COMMAND-, CTRL-, etc.) may be added.

Boxer may have default actions when sprite clicks have not been redefined. For example, MOUSE-CLICK-ON-SPRITE, by default, activates FOLLOW-MOUSE. (The primitive FOLLOW-MOUSE causes a sprite to track the mouse until the mouse button is released.) Again, by putting the definitions in appropriate places, you can change the behavior of all sprites in your world, only those in a certain graphics box (put definitions in that graphics box; make it not transparent), or individual sprites (redefinitions in individual sprites). Similarly, you may check the state of the mouse with MOUSE-BUTTONS to distinguish a click from a press, and to terminate an extended action instigated by a click or press.

Commands like MOUSE-POSITION return the coordinate position of the mouse in a graphics box. With them you can write your own specialized follow-mouse command, determine what region the user is pointing to in a graphics box, or track the mouse so as to change size or other properties of sprites or other graphics objects. INPUT? and HANDLE-INPUT can enable mouse clicks to operate while a program is running.

Redefining Editor Mouse Clicks and Hot Spot Actions

MOUSE-CLICK and MOUSE-DOUBLE-CLICK names (or with optional prefixes— Macintosh: OPTION- or COMMAND-; Windows: ALT- or CTRL-) will redefine mouse actions within the box in which they are defined. Redefining mouse clicks is in general not a good thing to do with such important gestures in Boxer, except deliberately to restrict a user's options (e.g., to prohibit execution by disabling MOUSE-DOUBLE-CLICK).

Each of the "hot spots" on a box, the four corners, the name tab and the type label, have magic names associated with them to allow redefining mouse clicks there.

MOUSE-CLICK-ON-TOP-LEFT (default action = shrink)
MOUSE-CLICK-ON-TOP-RIGHT (default action = expand)
MOUSE-CLICK-ON-BOTTOM-LEFT (default action = graphics flip)
MOUSE-CLICK-ON-BOTTOM-RIGHT (default action = resize)

MOUSE-CLICK-ON-NAME
MOUSE-CLICK-ON-TYPE

(DOUBLE-CLICK may be used in place of CLICK.)

Redefining these may be useful to restrict user access as well as modify or extend it. For example, defining MOUSE-CLICK-ON-BOTTOM-RIGHT as an empty procedure effectively keeps a user from changing the size of a fixed-size box. A handy convention is to have MOUSE-DOUBLE-CLICK-ON-NAME defined to provide help or information on that box, say, by setting a "help" or "instructions" variable that is visible to the user. Consult the Boxer Tool Box for a model.

Redefining Shrunken Box Icons (Boxtops)

You may redefine the visual presentation of a shrunken box to any graphic (icon). Place a graphics box containing your icon inside the box to be altered, and give the graphics box the magic name BOXTOP. The clickable buttons in Figure 18 and other cut-and-pasteable, interactive objects can be made with boxtops and redefined mouse actions. Boxtops can be dynamically altered with CHANGE-GRAPHICS BOXTOP .... Or you can just execute turtle commands in a turtle box that has been named "boxtop". In addition, there are some built in boxtop options available as Box Properties options (Box menu >> Boxtops).

FILE BOXES

Boxes that have been read in from a disk file, or that have been written out as a file, appear with double-thick borders to distinguish them from regular boxes. These boxes "remember" the disk file to which they are associated. Executing a Save from the menu, or keystroke shortcut (Macintosh: command-s; Windows: ctrl-s) while your cursor is anywhere within a file box will save that file box to disk. Boxer highlights the box it is saving. Save Box As File forces the box that the typing cursor is in to become a file box. In addition to menu and keystroke file commands, Boxer has regular commands to SAVE and OPEN files. Consult the on-line or hard-copy Command Manual. Boxer can save and read plain text files (use the Export dialog).

File boxes may be nested inside one another. Saving from within an interior file box saves only that file box, not ones that contain it or ones inside it. The title bar of the Boxer window (Mac version only, as of 3-03) shows which box will be saved. If a file box that contains other file boxes is saved and then read back in, the subfile boxes appear with a Boxer file icon, or with the box's boxtop if there is one. The sub-file boxes will be read from disk only when clicked or used in another way.

You can set an AUTOLOAD property of file boxes to make them automatically read themselves in when a file box containing them is read in. File properties are saved with the superior box, so, for example, a file can be AUTOLOAD in one place and non-AUTOLOAD in another. You can also declare a box READ ONLY, which will prevent a Save from easily overwriting the old file. Finally, you may convert a file box back to a regular, non-file box or convert it to a net box (below). All of these functions are available in Box Properties (Box menu, or accessible from the menu that pops up when you press control-mouse (right-press) on either of the top box corners). If you change a box's file properties, *you must then save the superior* file box in order for properties like AUTOLOAD to work.

If the top-level box is saved as a file, it becomes a "world box." Starting Boxer by double-clicking on a world box reinstates the box as the top-level box, unlike the usual case where a file appears necessarily as a subbox of the Boxer top-level.

Uses of File Boxes

Aside from the usual uses of files, file boxes allow you to create large workspaces containing many subfiles, e.g., for a particular long-term project. The subfiles are generally read in only when they are used. This is handy also in microworlds that have parts that are used only relatively rarely or that need to be saved separately from the rest of the microworld.

Transparent, READ ONLY, AUTOLOAD file boxes makes excellent tool boxes. For example, a teacher may provide students such a box of tools. When students save their own work in a box containing the tool box, the tool box itself will not be saved. But when that student box is read in again, the tools will be there for the student. Any changes the teacher saves to the tool box will automatically be available to students. And READ ONLY will prevent students from accidentally overwriting the teacher's tool box if they happen to try to save while inspecting the tools.

NETWORKING AND MAIL

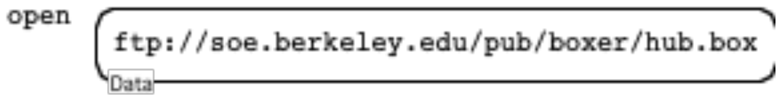
> *As of October 2021, networking and mail are unsupported during a re-design and reimplementing phase. Consult newer sources of information.* ***The information below is out of date until re-design and reimplementation are complete.*** *It remains here only so that users can get an idea of what services will be available in the future.*

If you have a connection to the Internet, you can use Boxer's capability to browse and collect Boxer resources (and directories and plain text files) that are located on remote machines. To do this, Boxer has special *network boxes*, which, like file boxes, have double-thick borders. Net and file boxes are nearly identical, except net boxes read in over the network rather than from a local file. Net boxes maintain a virtual connection to remote data. They "remember" the network location of the information and fetch it when you try to use it. So, for example, when you save net boxes within a file and read the file back in, the net boxes will appear black and shrunken (or with a boxtop icon, if one is defined) and initially be empty. A click or any other use action, such as using a net box as an input to a procedure, will trigger filling via the network. Thereafter, you can use net boxes just like ordinary data boxes. You may also save a net box back to its network location, provided you have the appropriate password.

Three kinds of objects can be connected to net boxes. Most common, (1) any Boxer file on a remote, Internet accessible machine can become a permanent (virtual) presence in any of your Boxer worlds. If those remote files contain further net boxes, you can browse the whole connected portion of the Boxer universe. In addition to Boxer files, net boxes can connect to (2) plain text files or (3) directories (folders). Directories appear also as data boxes that contain named subboxes, which are files or subdirectories.

If you want to keep the contents of a net box as part of your world (say, to customize it), you can force the conversion of a net box to an ordinary data box. You do this by using the "links" portion of Box Properties [one the Box menu, or the popup menu that appears when you right click on either of the top two corners of a box].

To create a net box from scratch, you need to know the "address" (URL of the file or directory to which you want to connect, and any login names and passwords necessary). Then use the Boxer OPEN command with the file address as input. See the Command Manual for details. The following will connect you to the Boxer hub of the universe.



```
open  ftp://soe.berkeley.edu/pub/boxer/hub.box
```

Alternatively, you can convert a file or regular box to a net box with Box Properties, links section, and enter a URL. Box Properties also allows you to inspect the URL for a net box, or the file location for a file box.

Suggestions:
In making boxes for others to link to, make sure you have a "browsable layer": that is, a set of small boxes that do not take long to read in but give a person browsing enough information—like file size and a brief description—so that s/he can make an intelligent decision about whether it is worth waiting for the net box to read in. Leave information on what kind of feedback, if any, you are interested in and appropriate e-mail addresses. Keep binary and other non-Boxer useful files out of any directories that you make accessible. Use a ".box" suffix on Boxer files to help ensure they are read in properly. See the on-line Command Manual for details.

We intend to coordinate interesting boxes to browse from our hub. Send URLs and information on how you imagine your box should be categorized to: **<to be determined>**. We solicit especially (1) descriptions of Boxer projects around the world, (2) solicitations for Boxer subcommunities and network projects, (3) general tools and utilities you feel people may like to have, (4) ideas on excellent things to do with Boxer, including "great hacks," (5) materials for learning various subject matter with Boxer, (6) examples of especially interesting student work.

Mail

Inactive as of October 2021

You can use the Boxer MAIL command to send e-mail. Before you send mail, you must enter your local host mail server into Boxer Preferences (under the Edit menu).

LINKS TO EXTERAL FILES

You may create a Boxer icon (a "black box") at the position of your cursor in Boxer that is linked to any external (non-Boxer) file on your machine. Use the File pulldown menu, "Link to File" option. Double clicking on such a box activates the file just as if it were double-clicked from the finder. A single click allows you to choose from (1) activating the external file, as above, (2) opening suitable files (such as a plain text file) in Boxer, or (3) adjusting the links from the icon. In the future (as of October 2021) we hope to integrate these links into Boxer mailing capability so that, for example, a linked-to file will be sent as an attachment.

EXTENSIONS

Boxer extensions are special files that add extra features to Boxer. Boxer automatically loads any extensions placed in a folder called “Extensions” when it starts up. The Extensions folder must be placed in the same folder as the Boxer application. Extensions may be placed in a folder (for example, “Extensions(off)”) when you don't want them loaded at startup.