MINOΣ — Visual bigFORTH

Bernd Paysan

August 26, 1997

Abstract

MINOΣ is a toolkit for rapid development of graphical user interfaces (GUIs) in Forth. MINOΣ comprises a widget library (called MINOΣ, too), and a graphical editor to master MINOΣ, therefore it's called Theseus. This paper gives an overview over the widget classes. An example is used to show how to create dialogs with the help of Theseus.

1 Introduction

1.1 What's Visual?

The wish to have a "Visual Forth" as counterpart to Visual BASIC (Microsoft) and Delphi (Borland) was well heard at the Forth–Tagung in 1996, and even before. Even C++ and new languages as Java have something similar, but Forth doesn't.

Mostly, these programming systems are integrated development environments[1] completed by a form painting program. This form painting program makes use of a library containing a variety of elements for a graphical user interface; e.g. windows, buttons, edit–controls, drawing areas, etc.. These elements can be combined with the mouse by drag&drop or click&point actions. Missing code then is inserted to add actions when buttons are pressed.

Typical applications are often related to data base access. Therefore, many of these systems already contain a data base engine or at least a standardized interface to a data base, such as ODBC.

Another aspect are complex components. With some of these toolkits, you can create a web browser with some mouse clicks and a few keystrokes. However, these components hide their details, a shrink wrapped web browser application is not necessesarily worse.

The interactivity of these tools usually is not very high. You create your form, write your actions as code and compile it more (Delphi) or less (Visual Age for C++) fast. Trying it usually isn't possible before the compiler run.

1.2 Why Visual?

It isn't really necessary to brush graphical user interfaces together, as it isn't to edit texts WYSIWYG. Many typesetting functions are more semantically than visual, e.g. a text is a headline or emphasized instead of written in bold 18 point Garamond or 11 point Roman italics. All this is true for user interfaces, to some extend much more. It's not the programmer that decides which font and size to use for the UI — that's up to the user. As is color of buttons and texts.

Also to layout individual widgets, more abstraction than defining position, width and height makes sense. Typically buttons are arranged horizontally or vertically, perhaps with a bit distance between them. The size of buttons must follow the containing strings, and should conform to aesthetics (e.g. each button in a row has the same width).

Such an abstract model, related to TEX's boxes&glues, programs quite good even without a visual editor. The programmer isn't responsible for "typesetting" the buttons and boxes. This approach is quite usual in Unix. Motif and Tcl/Tk use neighborhood relations, Interviews uses boxes&glues. I decided for boxes&glues, since it's a fast and intuitive solution, although

[1] That's what we had with Forth ever since

it needs more objects to get the same result.

These concepts contradict somehow with a graphical editing process, since the editors I know don't provide abstract concepts ("place left of an object" or "place in a row"), but positions.

1.3 Visual Forth?

One point makes me think: the packets that allow real visual form programming have many years of programming invested in. Microsoft, Borland, and IBM may hire hundreds of programmers just for one such project. This man–power isn't available for any Forth project. But stop:

- Forth claims that good programmers can work much more efficient with Forth
- A team of 300 (wo)men blocks itself. If the boss partitions the work, the programmers need to document functions, and to read documents from other programmer related to other functions and must understand them, or ask questions to figure things out. Everybody knows that documenting takes much longer than writing the code, and explaining is even worse. Thus at a certain project complexity level, no time is left for the programming task; all time is used to specify planned functions and read the specification from other programmers. Or the programmers just chat before the door holes of the much too small and noisy cubicles.
- A good programmer reportedly works 20 times as fast as a bad, even though he can't type in more key strokes per time. The resulting program is either up to 20 times shorter or has 20 times less bugs (or both) — with more functionality at the same time. Teamwork however prevents good programmers from work, since they are frustrated by bad programmers surrounding them, from their inability to produce required information in time; and the bad programmers are frustrated by the good ones, which makes them even worse.
- Therefore, even in large project, the real work is (or should be) done by a small "core team". Then the Dilbert rule applies: what can be done with two people, can be done with on at half of the costs.

Furthermore, bigFORTH–DOS already contains a "Text-GUI", without graphical editor, but with an abstract boxes&glue concept, which, as claimed above, hinders the use of such an editor.

Finally I wanted to get rid of DOS, and port bigFORTH to a real operating system (Linux). In contrast to Windows and OS/2, user interface and screen access are separated there. Drawing on the screen uses the X Window System (short X), the actual user interface is implemented in a library. This is the reason, why there is no common interface, but a lot of different libraries, such as Athena Widgets, Motif, Tcl/Tk, xforms, Qt, gtk, and others. The "look and feel" from Motif-like buttons is quite common, even Windows and MacOS resemble it.

All these libraries have disadvantages. The Athena Widgets are hopelessly outdated. Motif is commercial, even if a free clone (Lesstif) is in creation. It's slow and a memory hog. Tcl/Tk consumes less memory, but it's *even* slower. How do you explain your users that drawing a window takes seconds, while Quake renders animated 3D-graphic on the same machine? Qt is fast, but it's written in C++ and doesn't have a foreign language interface now. gtk, the GIMP toolkit, has more foreign language interfaces, and it's free, but it wasn't available until recently.

Therefore I decided to port the widget classes from bigFORTH–DOS to X, and write an editor for it. Such classes written in Forth naturally fit in an development environment an are — from the Forth point of view — easier to maintain. There are not such many widget libraries in C, because it's a task written in an afternoon, but because the available didn't fit the requests, and a modification looked desperate.

1.4 The Name — Why MINOΣ?

"Visual XXX" is an all day's name, and it's too much of a microsoftism for me. "Forth" is a no-word, especially since the future market consists of one billion Chinese, and for them four is a number of unluck (because "se" (four) sounds much like "se" (death)). However, even Borland doesn't call their system "Visual TurboPascal", but "Delphi".

Greek is good, anyway, since this library relates to the boxes&glues model of $\TeX$, which is pronounced greek, too. Compared with Motif, the library is quite copact (MINimal), and since it's mainly for Linux, the phonetic distance is small. . . I pronounce it greek: "menoz".

1.5 Port to Windows

I ported MINOΣ to Windows 95/NT, on the demand of some potential users. It doesn't run near as stable as under Linux/X, since there are a hideous number of subtle bugs in Windows, and I don't have the time to work around all of them. Drawing polygons doesn't work as well as on X, and all the bugs that are in the memory drawing device can drive me nuts.

2 Widget Classes: Display, Widget, Actor

The principle of the class hierarchy was fixed with the given library for DOS. This library distinguishes between widgets ("window gadgets") and displays. Displays are widgets that also can paint, such as windows, viewports, backing stores and double buffers. They are responsible for translating the abstract interface to the actual graphic library, and for event handling (mouse clicks, key strokes, redraws, etc.).

The widgets themselves are divided into boxes (horizontal and vertical), buttons, toggles, labels, icons, text input fields, sliders, scalers, canvas. . . alltogether currently 88 classes.

Originally, all the actions that are invoked at clicks where simple Forth words. It has shown that this wasn't suitable. Objects manipulate data representations, and it's useful to have the action tied to the data. Therefore, the actions now are translated using "action" objects. E.g. a toggle button may set a variable to "on" or "off", and retrieve it's state from the variable. Or some radio buttons change the number in a variable. Therefore a number of different action classes provides interfaces of object actions for simple things to complex things as showing tool tips. This solves the problem of varying reactions on events with simple means, without making the default path more complicated.

One further class is related to displays: the resources. This class contains screen specific data, such as display, screen, font, colors, color-map, cursors, and the graphic context.

A class hierarchy comprises a common interface, thus methods and variables, which are understood by all subclasses. The main elements of the widget protocol (Figure 1) and displays (Figure 2) are presented here.

Derived classes certainly have additional variables, object pointers, and eventually additional methods.

The display class is derived from the widget class. Therefore it understands all messages of a widget class. Some displays as viewports, backing store, and double buffer can be used as normal widgets as part of a dialog or a window.

2.1 Composed Objects

More complex objects such as sliders and scalers are composed out of simpler objects (especially glues). This was inspired by gtk, which composes even simple objects. I implemented sliders and scalers as one object before, and the result was quite lengthy code, difficult to debug. The composed objects require only half of the code, and where written in one day. Composed objects take more memory at run-time, and are presumed to redraw slightly slower.

2.2 The Complete Class Hierarchy

The class hierarchy states also the memory size of the object (for variables) and the size of the method table (per class). Indentation shows subclassing.

2.2.1 Actions

The available actions concentrate on toggle and radio buttons. These buttons have two distinct states — set or reset. The action may set a flag (`toggle-var`), store a number to a variable (`toggle-var`), or actions at set and reset (`toggle`), or to query and change

Method	Purpose
PARENT	points to the parent object
WIDGETS	points to the next object
DPY	the display of this widget
INIT	initializes the object
DISPOSE	deletes the object
HGLUE	horizontal glue
VGLUE	vertical glue
XINC	horizontal size increment
YINC	vertical size increment
XYWH	bounding box
RESIZE	changes size
REPOS	changes position
RESIZED	recomputes size
!RESIZED	more detailed recomputation
CLOSE	closes the window
DRAW	draws itself
ASSIGN	assigns a new contents
CLICKED	click event handling
KEYED	keystroke handling
INSIDE?	is this point inside the object?
HANDLE-KEY?	does it handle keystrokes?
FOCUS	object got focus
DEFOCUS	object looses focus
SHOW	the object is visible
HIDE	the object is invisible
MOVED	pointer over the object
LEAVE	pointer leaves object
DELETE	remove object from list
APPEND	add object to list
SHOW-YOU	object should show itself
FIRST-ACTIVE	set active object to the first
NEXT-ACTIVE	next object becomes active
PREV-ACTIVE	previous object becomes active

Figure 1: Widget messages

Method	Purpose
XRC	resource
LINE	line between two points
TEXT	paint text
IMAGE	draw pixmap
BOX	draw rectangle
MASK	paint icon
FILL	fill polygon
STROKE	draw polygon outline
DRAWER	call drawing routine
DRAWABLE	resources for drawing
SYNC	end update
MAP	map window
UNMAP	unmap window
MOUSE	mouse position
SCREENPOS	screen position of display
TRANS	coordinate transformation
TRANS'	reverse transformation
TRANSBACK	transformation to GET–WIN
GET–DPY	get outer display
GET–WIN	get containing window
SET–FONT	set font
SET–COLOR	set color
SET–CURSOR	set mouse cursor
TXY!	set tile offset
CLIP–RECT	set clipping rectangle
GET–EVENT	get event
HANDLE–EVENT	handle events
SCHEDULE–EVENT	schedule events
CHILD–MOVED	distributes mouse moves
CLICK	wait for mouse click
CLICK?	query mouse click
MOVED?	query mouse move
MOVED!	set mouse as moved
SHOW–ME	show object at (x,y)
SCROLL	scroll to (x,y)
CLIPX	horizontal clipping
CLIPY	vertical clipping
GEOMETRY	resize in object coordinates
>EXPOSED	wait until visible

Figure 2: Display messages

(`toggle-state`). Slider and scaler (with maximum position and step width) are handled similar to `toggle-state`.

```
ACTOR                    12  80
  SCALE-VAR              20  80
    SCALE-DO             24  80
    SLIDER-VAR           24  80
      SLIDER-DO          28  80
  SIMPLE                 16  80
    DRAG                 16  80
    REP                  16  80
    DRAWER               16  80
  TOGGLE-STATE           20  80
    SCALE-ACT            24  80
      SLIDER-ACT         28  80
  TOGGLE-VAR             16  80
    TOGGLE-NUM           20  80
  TOGGLE                 24  80
```

2.2.2 X–Resource

This object contains server related data like fonts, graphic context, colors, and similar.

```
XRESOURCE                48  92
```

2.2.3 Combined Widgets

These widgets contain other widgets and compute their arrangement. The letters stay for:

H horizontal

V vertical

A one active element, navigation with TAB

R radio buttons

T tabbed box — all non-glue objects have equal size

Further, there are combined widgets like sliders, scalers, boxes which contain a viewport and the corresponding sliders, as well as boxes that can be resized by the user using a `hsizer` or `vsizer`. Furthermore, some boxes set the stepping width during resize and sliding.

Beside being partitioned in different classes, boxes contain attributes. Their size can be fixed to a minimum, both horizontal as vertical. A separating space can be inserted between each element, the box may have a shadow, and made invisible. This allows to create the popular card files. This could also be used to hide commands that are currently not available.

```
COMBINED                 68 224
  VBOX                   68 224
    SLIDERVIEW           76 224
      ASLIDERVIEW        76 224
    VRBOX                68 224
    VTBOX                68 224
      VRTBOX             68 224
      VATBOX             68 224
        VARTBOX          68 224
    VABOX                68 224
      VASBOX             72 228
      VRESIZE            68 224
      MODAL              76 224
      VARBOX             68 224
  HBOX                   68 224
    HRBOX                68 224
    HTBOX                68 224
      HRTBOX             68 224
      HATBOX             68 224
        HARTBOX          68 224
    HABOX                68 224
      HASBOX             72 228
      HRESIZE            68 224
      HARBOX             68 224
```

2.2.4 Buttons and Labels

Active components are available in many flavors, with and without icon, as button, toggle button, to open menus ...

```
GADGET                   28 188
  WIDGET                 36 196
    BOXCHAR              48 200
      BUTTON             52 200
        ALERTBUTTON      60 200
        MENU-ENTRY       52 200
          EDIMENU-ENTRY  56 200
          MENU-TITLE     60 208
            INFO-MENU    68 212
            SUB-MENU     60 208
        ICON-BUTTON      56 200
          BIG-ICON       56 200
          ICON-BUT       56 200
        LBUTTON          52 200
          FILE-WIDGET    76 200
          TEXT-LABEL     52 200
            MENU-LABEL   52 200
```

```
    TOGGLECHAR              52 208
      FLIPICON              56 208
      TOGGLEICON            60 208
      TBUTTON               56 208
        TICONBUTTON         64 208
        TOPINDEX            56 208
        TOGGLEBUTTON        60 208
        FLIPBUTTON          56 208
        RBUTTON             56 208
    TRIBUTTON               48 200
      SLIDETRI              48 200
  ICON                      40 196
  ICON-PIXMAP               44 200
```

2.2.5 Text Fields

Text fields allow to enter texts and numbers (with syntax checking)

```
TEXTFIELD                   64 212
  INFOTEXTFIELD             68 212
    INFONUMBERFIELD         68 212
  NUMBERFIELD               64 212
```

2.2.6 Slider and Resizer

Slider position the interior of viewports; scaler are useful to enter numbers (in a given range). Resizer change the size of a `hasbox` or a `vasbox`, dragging the border into the desired direction.

```
    HSLIDER                 68 228
      HSCALER               80 232
      HSLIDER0              68 228
    VSLIDER                 68 228
      VSCALER               80 232
      VSLIDER0              68 228
  HRTSIZER                  52 208
    HXRTSIZER               52 208
    HMRTSIZER               52 208
      HSIZER                52 208
  VRTSIZER                  52 208
    VXRTSIZER               52 208
    VMRTSIZER               52 208
      VSIZER                52 208
```

2.2.7 Glues

Glues are expandable objects. Inserted at the right place, they allow a decent layout. E.g. placing two glues left and right of an object centers the object. Sliders transform the total and the visible size of a viewport into glue values, easing computation of slider width and position. One special glue is the canvas, which allows drawing into it. It understands some sort of turtle graphic.

```
(NIL                        36 196
ARULE                       48 196
GLUE                        52 196
  RULE                      56 196
    CANVAS                 108 272
    MFILL                   56 196
    MSKIP                   56 196
      SSKIP                 64 196
    VRULE                   56 196
    HRULE                   56 196
  VGLUE                     52 196
  HGLUE                     52 196
```

2.2.8 Terminal and Editor

Terminal and screen editors also are available as elementary components.

```
TERMINAL                   104 276
  SCREDIT                  144 300
```

2.2.9 Displays

These are windows, viewports (display only a section), double buffer (for flicker free drawing), menu frames... Furthermore, standard dialogs like the file selector are derived from the window class.

```
DISPLAYS                   136 356
  WINDOW                   148 368
    FILE-SELECTOR          176 372
    TERMWIN                148 368
    MENU-WINDOW            148 368
    FRAME                  152 368
      MENU-FRAME           152 368
  (NILSCREEN               136 356
  BACKING                  160 360
    VIEWPORT               216 372
      SCRVIEWPORT          216 372
      HVIEWPORT            224 372
      VVIEWPORT            224 372
    DOUBLEBUFFER           160 360
```

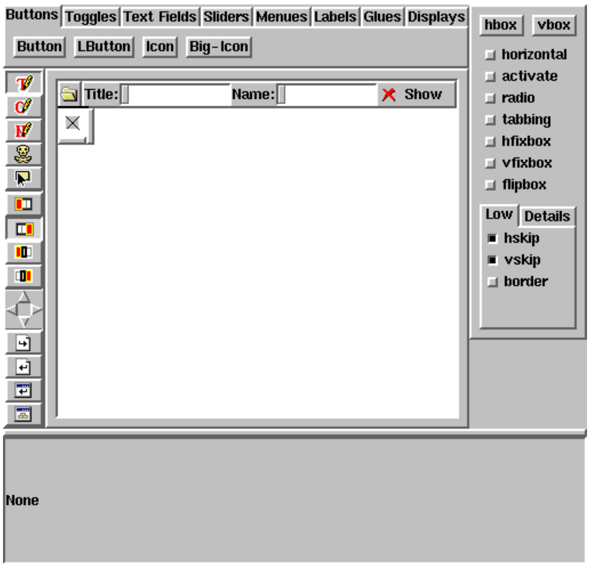

Figure 3: Theseus after starting it

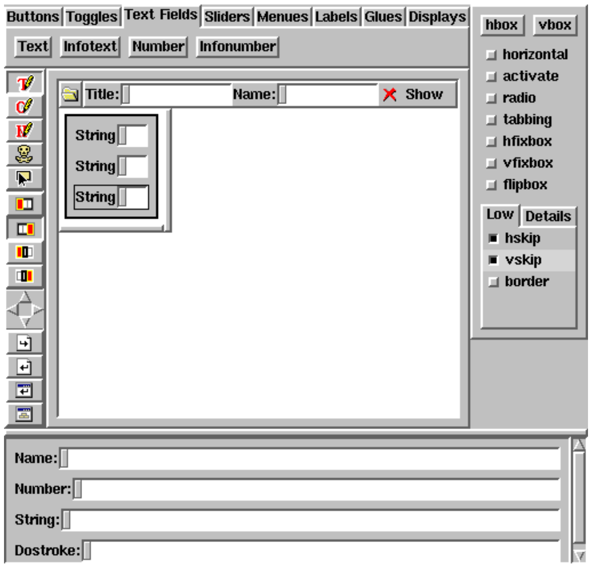

Figure 4: Input and output fields

3 Theseus — the GUI editor

How do you edit such a user interface? Formatting buttons and text fields is done by the system, therefore not the task of the programmer, which only has to fix the logical arrangement.

The project therefore is hierarchically arranged. The topmost hierarchy are the dialog windows. These windows understand two additional methods, `open` and `modal-open` which allows to create both non-modal and modal dialogs. The user then creates a framework of horizontal and vertical boxes inside the dialog. These boxes are filled with contents and glues then.

Two small examples will show how to use Theseus. The first creates a small calculator operating on integers. Figure 3 shows the editor at the project start.

Input fields and the result field should appear one beneath the other, therefore a `vbox` is created, and three `infonumberfields` inside it. This step is shown in Figure 4.

Beneath the two input fields the operation buttons should be arranged one aside each other. A horizontal box (`hbox`) does the job, with four buttons in it. A small distance between each field and each button would be nice, too. Figure 5 shows the state after these operations.

Now these objects need a useful text. Therefore you click each object (in edit mode), and type the text. The result is shown in Figure 6.

To reference the input field, each one must have an internal name. Choose name mode, click to the fields and enter the name (`a#`, `b#`, and `c#`). Now you can insert code, i.e. for the operation `A+B`. Corresponding to the example in Figure7, the other code is inserted, too.

The code looks as follows:

```
a# get b# get d+ r# assign
a# get b# get d- r# assign
a# get b# get d* r# assign
a# get b# get drop ud/mod r# assign drop
```

But stop! Maybe it's useful to take the result and copy it to one of the input buttons for reuse. Thus two additional buttons are required, and to make it nice, all buttons should have the same size (with "tabbing" box style). The window must have a title, and a name; to have it shown after startup, click on the "Show" button, too. The result is sown in Figure 8.

The additional code looks like this:

```
r# get a# assign
r# get b# assign
```

Now you can try the result by pressing the "run" icon. Theseus generates the code and

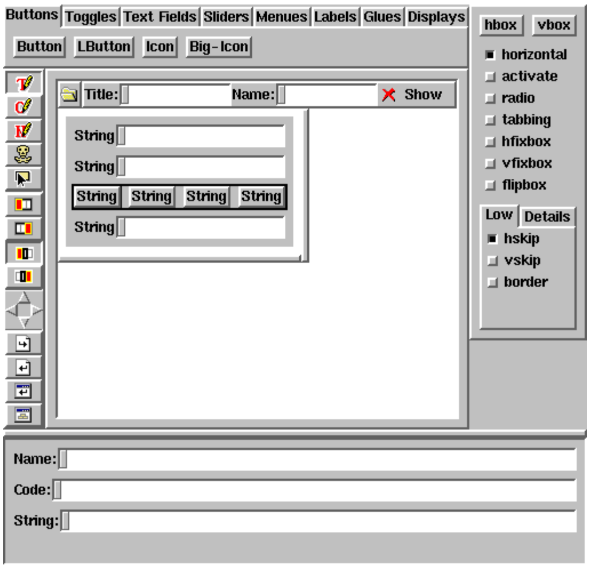

Figure 5: Buttons for computation

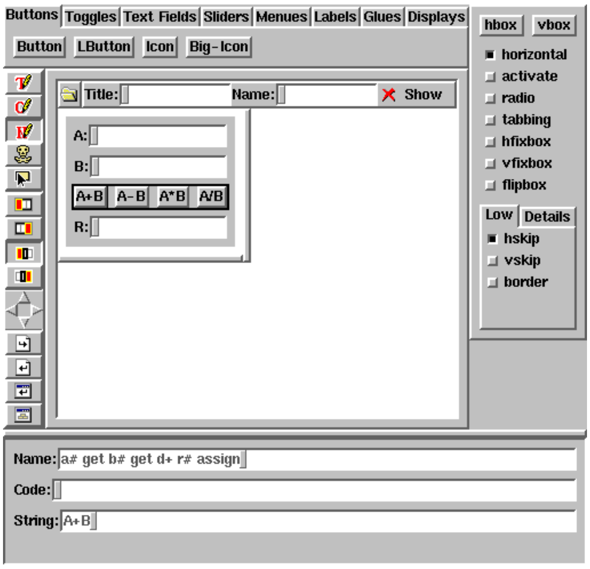

Figure 7: Code

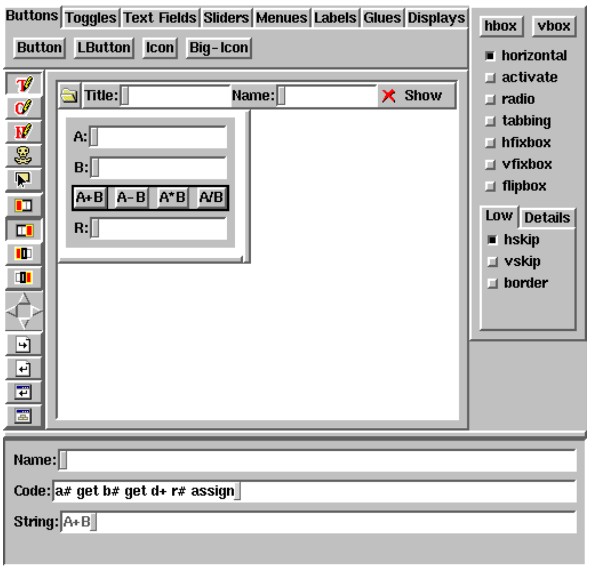

Figure 6: Texts

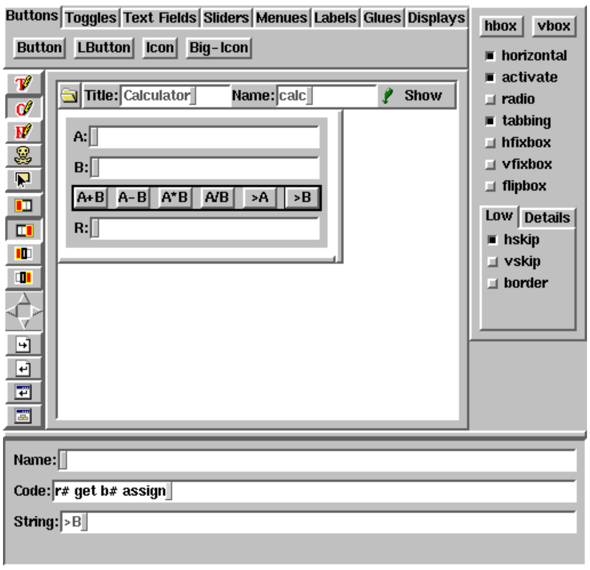

Figure 8: More buttons, more code

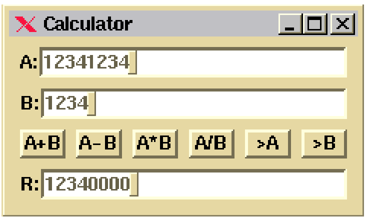

Figure 9: The calculator

starts a new invocation of bigFORTH which compiles it and starts the application. Figure 9 shows the final window.

3.1 Automatically Generated Code

Theseus generates Forth code from these buttons. It derives a class from `window`, which will contain the dialog. All objects (except boxes) get a name (generated automatically, if none exists) and an object pointer to access them. The code for this example project looks as shown in Figure 10. This code is also MINOΣ' internal data format.

4 Outlook and Conclusion

MINOΣ has a lot of features that haven't been explained here. Theseus isn't finished yet, but it can compose most of the things you need. It isn't as interactive as I wish it (especially it can't run application code from within the editor yet); and debugging can be improved. It also lacks documentation, and tons of good examples.

To be even more competitive, MINOΣ would need more complex classes, such as a WYSIWYG text editor, a web browser (both could be identical), OpenGL drawing areas, an ODBC or SQL interface to data bases, image export and import, and more. The web browser should work as online help system,which yet has nothing but a name yet ("Ariadne").

To get all these things done while I can only work part-time on MINOΣ, I decided to give MINOΣ on Linux away for free, if it's used according to the rules of the GNU public license (GPL), so other people can join the effort. For commercial users and users of MINOΣ for Windows, the usual commercial license is available.

```
\ automatic generated code
\ do not edit
windows also forth

window class calc
public:
  early open
  early modal-open
  infonumberfield ptr a#
  infonumberfield ptr b#
  | button ptr (button-00)
  | button ptr (button-01)
  | button ptr (button-02)
  | button ptr (button-03)
  | button ptr (button-04)
  | button ptr (button-05)
  infonumberfield ptr r#
how:
  : open       screen self new >o map o> ;
  : modal-open screen self new >o map stop o> ;
class;

calc implements
  : init  super init ^ { ^^ | ( [dumpstart] )
        &0. ]N s" A:" ^ infonumberfield new dup ^^ with bind a# endwith
        &0. ]N s" B:" ^ infonumberfield new dup ^^ with bind b# endwith
          ^^ S[ a# get b# get d+ r# assign ]S s" A+B" ^ button new dup ^^ with bind
(button-00) endwith
          ^^ S[ a# get b# get d- r# assign ]S s" A-B" ^ button new dup ^^ with bind
(button-01) endwith
          ^^ S[ a# get b# get d* r# assign ]S s" A*B" ^ button new dup ^^ with
bind (button-02) endwith
          ^^ S[ a# get b# get drop ud/mod r# assign drop ]S s" A/B" ^ button new dup
^^ with bind (button-03) endwith
          ^^ S[ r# get a# assign ]S s" >A" ^ button new dup ^^ with bind (button-04) endwith

          ^^ S[ r# get b# assign ]S s" >B" ^ button new dup ^^ with bind (button-05) endwith

        6 ^ hatbox new 1 hskips
        &0. ]N s" R:" ^ infonumberfield new dup ^^ with bind r# endwith
      4 ^ vabox new panel
    ( [dumpend] ) } 1 0 ^ modal new 0 hskips 0 vskips s" Calculator" assign ;
class;

script? [IF]
: main
  calc open
  &1 0 ?DO  stop  LOOP ; main
bye [THEN]
```

Figure 10: Automatically generated code