MINOΣ—System Integration

Bernd Paysan

August 29, 1998

Abstract

This paper presents the integration of MINOΣ—a toolkit for rapid application development of graphical user interfaces (GUI RAD) in Forth—into the system. Two main areas of interrest are data base query via SQL and integration of OpenGL. Other improvements of MINOΣ are also covered. This text blatantly contains parts that were already presented last time and on the last Forth Tagung in Moers.

1 Introduction

Last year, I presented MINOΣ here. It could just create a simple calculator. Since then many improvements have happend.

But step by step:

1.1 What's MINOΣ?

MINOΣ is the answer to the question "Is there something like Visual BASIC (Microsoft) or Delphi (Imprise) in Forth?" Basically, these GUI RADs contain two components: a library with a wide variety of elements for a graphical user interface; e.g. windows, buttons, edit–controls, drawing areas, etc.; and an editor to combine the elements 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 necessarily 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 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 projects, 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 one 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. The Windows port of MINOΣ looks more like the "modern Forth" Claus Vogt portrayed in `de.comp.lang.forth`: it shows random general protection faults. Well, just like any other Windows program.

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 its 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, scalers, and text fields 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. I plan to split up further objects, especially toggle buttons.

3 News

3.1 New Platforms

MINOΣ runs now under Windows 95/NT. Jens Wilke sponsored a MSDN[1]-CD, and since MINOΣ relies only on some few X functions and

[1] Micro$oft Documents Nothing

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

does all the rest itself, it shouldn't have been too difficult. Unfortunately, deadline pressure and mismanagement at Microsoft efficiently prevents a quality control that's worth the name[2]. Furthermore, X has 10 years advance of maturity than Win32.

With one sentence: while you fight your own bugs under X, you fight against bugs in the system, and imprecise and partly incorrect documentation. Since I don't use Windows much (except at work, as expensive X Terminal), there is no big pressure to work around the bugs.

The whole porting took about a month. This at least shows that the concept of MINOΣ is pretty portable. Smaller programs don't pose too much difficulties, more demanding like Theseus has problems (crashes under Windows 95, and has serious problems under NT). OpenGL-Widgets don't work at all, since Windows prevents me drawing OpenGL operations into a bitmap (though the documentation says it should work).

3.2 New Widgets: OpenGL Canvas

Apart from some small additions, there is now an OpenGL widget (`glcanvas`). Similarly to the `canvas` widget you can execute painting operations, in this case OpenGL operations.

OpenGL is a language to describe 3D elements, created from Silicon Graphics derived from a formerly proprietary language (Iris GL). Although Microsoft tried to invent their own "standard" (Direct3D), OpenGL is widely accepted as cross-platform standard. Even some Windows games use an OpenGL subset instead of Direct3D, that is supported for the popularly Voodoo 3D cards (GLIDE).

Both under Windows and under Linux, OpenGL libraries are freely available (oglfix for old Windows 95 versions, included in newer versions, and MESA as well for Linux and Windows).

From a guiding example, I will show how you program an OpenGL widget using MINOΣ (see Figure 3).

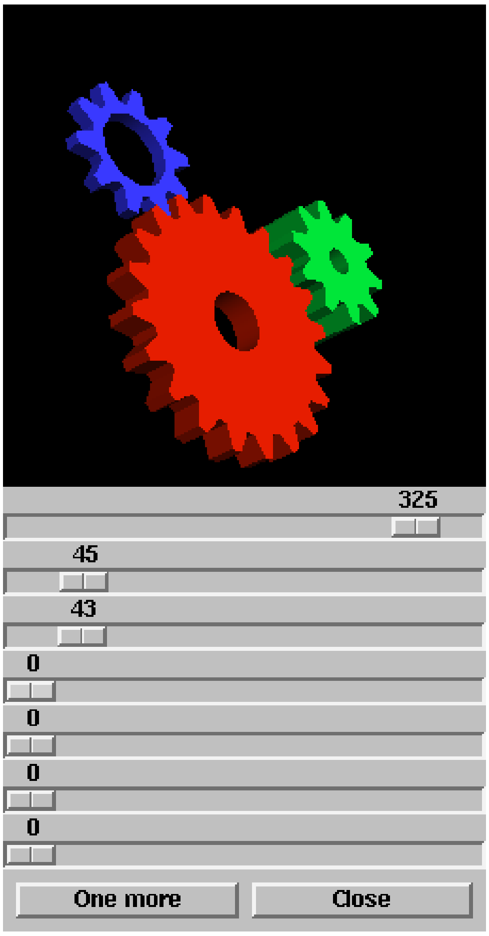

Figure 3: GL-Widget "gears"

[2] In fact there are only very few companies that can't get a demo running which doesn't crash when given to the CEO. Windows 98 is after Windows NT 4.0 the second Windows personally blue screened by Bill Gates in public.

But first a look at the code. Since MINOΣ doesn't yet provide special comfort, I first define a cylinder coordinate system:

```
: r,phi ( r angle -- x y )
  fsincos f>r fover f* fswap fr> f*
  f>fs f>fs ;
```

OpenGL expects the parameter on the data stack (C calling conventions), therefore `f>fs` converts the values to "single float" (on the data stack). I store the front view of a cogwheel's tooth in an array:

```
: array  Create cells allot
  DOES> swap cells + ;

$E array points

: >points ( r0 r1 r2 angle -- )
    f>r
    2 fpick f>fs 0         $0 points 2!
    fover   f>fs 0         $2 points 2!
    fdup fr@ r,phi         $4 points 2!
    fr@ f2* r,phi          $6 points 2!
    fdup fr@ 3 fm* r,phi   $8 points 2!
    fr@ 4 fm* r,phi        $A points 2!
    fr> 4 fm* r,phi        $C points 2! ;
```

As next step the whole tooth is created. First, front and back surfaces are described as "triangle fan". OpenGL is a state machine, just like Postscript you create paths, but 3D now. Therefore OpenGL and Forth fit together quite well. A triangle fan starts with one point, you define a second point, and from the third point, each point describes a triangle consisting of the starting point, the current point and the previous point. The shading model here is "flat", thus there are rough edges.

```
: tooth ( teeth r0 r1 r2 rw -- )
    GL_FLAT glShadeModel drop
    f2/ f>fs >r pi fm/ f2/ >points r>
    \ front and back side
    #1 0 0 glNormal3f drop
    GL_TRIANGLE_FAN glBegin drop
         $0 $C DO
             dup I points 2@
             glVertex3f drop  -2 +LOOP
    glEnd drop fsneg
    [ !-1 f>fs ] Literal 0 0
                 glNormal3f drop
    GL_TRIANGLE_FAN glBegin drop
        $E $0 DO  dup I points 2@
        glVertex3f drop   2 +LOOP
    glEnd drop fsneg
```

I create the outside as "quad strip". Here you create rectangles using a zig zag pattern, going from front to bag and left to right over the rectangles. Remember: all OpenGL surfaces must be oriented clockwise.

```
\ outer side
    GL_QUAD_STRIP glBegin drop
    $C $2 DO
        fsneg dup I points 2@
                    glVertex3f drop
        fsneg dup I points 2@
                    glVertex3f drop
        I 3 and 0= IF
            0 #1 0
        ELSE
            0 I 2+ points 2@ fs>f
            I points 2@ fs>f f-
            fs>f fs>f f- f>fs f>fs swap
        THEN
        glNormal3f drop
    2 +LOOP
    glEnd drop
```

The inner side should look round and smooth, and therefore the colors need interpolation. The shading model therefore is "smooth". I use a quad strip here, too, but since there aren't flat planes, additional normal vectors are required. These are valid for both left and right borders, and point right into the center.

```
\ inner side
    GL_SMOOTH glShadeModel drop
    GL_QUAD_STRIP glBegin drop fsneg
        0 $0 points 2@ swap fsneg
        swap fsneg glNormal3f drop
        fsneg dup $0 points 2@
                     glVertex3f drop
        fsneg dup $0 points 2@
                     glVertex3f drop
        0 $C points 2@ swap fsneg
        swap fsneg glNormal3f drop
        fsneg dup $C points 2@
                     glVertex3f drop
        fsneg dup $C points 2@
                     glVertex3f drop
    glEnd 2drop ;
```

OpenGL doesn't require to repeat complex computations necessary for the creation of a 3D surface each time anew. There are "display lists", and they remember coordinates quite well. You can draw these lists with the current coordinate transformations, and store them on the display server (when OpenGL is used over the net, i.e. with an X server with GLX extensions). Just like a metafile for Windows or MacOS, you only change the drawing mode, and all OpenGL instructions are compiled to that list.

```
: create-tooth ( teeth r0 r1 r2 rw -- n )
    1 glGenLists
    GL_COMPILE over glNewList drop
```

```
    swap tooth glEndList drop ;
```

I use such a tooth to create the complete cogwheel. The current transformation matrix rotates by $360/n$ degrees, a tooth draws itself by calling the display list, and rotation continues. Transformation matrixes have their own stack, and push/pop operations. I compile the whole cogwheel into a display list, too.

```
: cogwheel ( tooth teeth -- )
    glPushMatrix drop
    0 ?DO
        #1 0 0 !&360 i' fm/
        f>fs glRotatef drop
        dup glCallList drop
    LOOP drop
    glPopMatrix drop ;

: create-cogwheel ( list teeth -- cogwheel )
    1 glGenLists
    >r GL_COMPILE r@ glNewList drop
    cogwheel  glEndList drop r> ;
```

Now we need just a few colors and the position of the light source, then we are ready to start.

```
Create .pos   !&5  f>fs , !&5  f>fs ,
              !&10 f>fs , !0   f>fs ,
Create .red   !&.8 f>fs , !&.1 f>fs ,
              !&0  f>fs , !1   f>fs ,
Create .green !&0  f>fs , !&.8 f>fs ,
              !&.2 f>fs , !1   f>fs ,
Create .blue  !&.2 f>fs , !&.2 f>fs ,
              !&1  f>fs , !1   f>fs ,

Create textures 0 , 0 , 0 ,
```

The picture should contain three cogwheels. So first, we initialize OpenGL, and create the three different cogwheels.

```
: create-gears ( -- cogwheel0 cogwheel1
cogwheel2 )
    .pos GL_POSITION GL_LIGHT0
    glLightfv drop

    GL_CULL_FACE
    GL_LIGHTING
    GL_LIGHT0
    GL_DEPTH_TEST
    GL_NORMALIZE
    5 0 DO  glEnable drop  LOOP

    &20 !&.2  !&.73 !&.87 !&.2
      create-tooth    &20 create-cogwheel
    &10 !&.10 !&.33 !&.47 !&.4
      create-tooth    &10 create-cogwheel
    &10 !&.26 !&.33 !&.47 !&.1
      create-tooth    &10 create-cogwheel ;
```

The cogwheels should rotate, depending on the number of teeth, and time. Certainly independent of the frame rate.

```
: rotation ( teeth -- fn )
    &86400 swap / timer@ * &360 um*
    d>f !$.00000001 f* ;
```

Since the cogwheels are stored as list, without color and position, they must be colored and positioned.

```
: call-cogwheel ( n r+ tx ty tz color -- )
    GL_AMBIENT_AND_DIFFUSE GL_FRONT
    glMaterialfv drop
    glPushMatrix drop
    f>fs f>fs f>fs glTranslatef drop
    >r #1 0 0 r> glRotatef drop
    glCallList drop
    glPopMatrix drop ;
```

The final drawing routine gets some parameters: all those you can set with the sliders, the display lists, and the `glcanvas` object. I set projection and drawing mode, clear the pixmap, and perform all the rotations. Finally I draw the cogwheels, and that's it.

```
: draw-cogwheel ( o g0 g1 g2 alx aly alz pitch
bend roll zoom -- )
 { g0 g1 g2 alx aly alz alp alb alr zoom |
    glcanvas with
        h @ w @ 0 0 glViewport drop

        GL_PROJECTION glMatrixMode drop
        glLoadIdentity drop

        !&60 f>fd !&5 f>fd
        w @ h @ >
        IF
            w @ s>f h @ fm/
            !1 f>fd !-1 f>fd
            fdup f>fd fnegate f>fd
        ELSE
            h @ s>f w @ fm/
            fdup f>fd fnegate f>fd
            !1 f>fd !-1 f>fd
        THEN
        glFrustum drop

        GL_MODELVIEW glMatrixMode drop
        glLoadIdentity drop
        zoom 100 + negate s>f
        !0.08 f* f>fs 0 0 glTranslatef
        drop
        GL_COLOR_BUFFER_BIT
        GL_DEPTH_BUFFER_BIT
        or glClear drop

        0 0 #1 alx s>f f>fs glRotatef drop
        0 #1 0 aly s>f f>fs glRotatef drop
        #1 0 0 alz s>f f>fs glRotatef drop
        0 0 #1 alp s>f f>fs glRotatef drop
        0 #1 0 alb s>f f>fs glRotatef drop
        #1 0 0 alr s>f f>fs glRotatef drop

        !&9  -&5 rotation f+ f>fs >r
        !-&9 -&5 rotation f+ f>fs >r
        !0  &10 rotation f+ f>fs >r
```

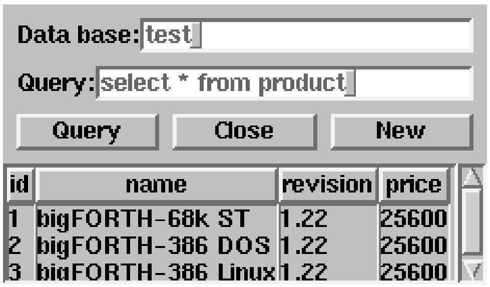

Figure 4: SQL query dialog

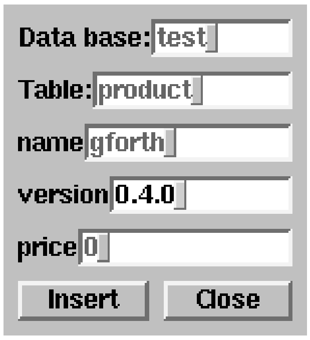

Figure 5: SQL input form

```
      !-&.6  !-&.4 !0 g0 r>
             .red   call-cogwheel
      !&.62  !-&.4 !0 g1 r>
             .green call-cogwheel
      !-&.6  !&.84 !0 g2 r>
             .blue  call-cogwheel
endwith } ;
```

3.2.1 Future

I'm not fully satisfied with the OpenGL interface. An abstract interface, some sort of "turtle graphics", but 3D, is what I have in mind. Especially when using textures, automatisms are useful. It should be possible to create 3D objects and perform other operations aside from the display operations. Collision detection or inverse kinematic comes in mind.

3.3 Data Bases: an SQL[3] Interface

Data base interfaces are a very typical application for GUI designers. Input form creation, and queries require only some few GUI elements. Since data bases depend very much on the customer's wishes, you must provide simple tools to create input and query forms. Figure 4 shows a very simple data base query.

Certainly I haven't written a SQL data base. There are quite a few SQL data bases for Linux, commercials like Adabas-D, and recently announced, Informix, Oracly, Sybase, and rumored DB/2. And free ones like mySQL, mSQL and PostgreSQL. MINOΣ uses a SQL class to interface the (unfortunately not standardized) access to the data bases. I wrote that class using the PostgreSQL library, other libraries including ODBC could be written to replace this class.

But let's look at the details first. We need a query function. The field `db` contains the name of the data base, the field `query` the query string. The results are presented in a viewport. Since Theseus[4] doesn't provide names for viewports yet, but names it automatically, I help me with an internal detail of Theseus, to get to the viewport.

```
: do-query
  db get database new >r
  querys get r@ database with exec endwith
  (vviewport-00) self
  r@ database with entry-box endwith
  (vviewport-00) with assign resized endwith
  r> with dispose endwith ;
```

The query itself is quite simple. First, create a connection to the data base. A data base object holds the state of this connection. I send the query string right to the data base. Then I transform the result into a table created from MINOΣ widgets, and put them into the viewport. Finally, I close the connection to the data base.

Now you must insert data into the data base. Assume we have a table `product`, and must create an input form. Figure5 shows such an input form. Similar to the query form, we create the input fields first, and then need a word, that performs the insertion.

```
: do-insert
  db get database new >r
  s" max(id)" r@ database with select endwith
  table get r@ database with from ) endwith
  0 0 r@ database with tuple@ s>number clear
endwith
```

[3] Structured Query Language

[4] the GUI editor of MINOΣ

Figure 6: MIDI Player

```
  table get r@ database with insert( endwith
  drop 1+ r@ database with int, endwith
  #name get r@ database with string, endwith
  #version get r@ database with string,
endwith
  #price get drop r@ database with int,
endwith
  r@ database with ) endwith
  r> with dispose endwith ;
```

First, the user should not need to select the ID of the entry. We ask the data base for the maximal ID. This number, incremented by one, is the new ID. The data base object provides methods that simplify the creation of a query string. We use them here for both the SELECT query as for the INSERT query.

Note that this is not a perfect transaction. A multi-user data base might have two users query for the max ID at the same time (returning the same result), which would result in two entries with the same ID. But proper data base use is another topic.

3.4 MIDI Interface

Modern systems don't just provide colorful surfaces, sound is also quite important. Writing a simple application like a MIDI player with MINOS is a matter of a few keystrokes (see Figure 6).

So we desing a dialog box with a text widget and four buttons. We define two variables

```
midi-player ptr player
cell var midi-path
```

in the Variable section. Furthermore, we need a word that creates the player, just in case it is not there:

```
: ?player ( -- )
  player self 0= IF
     midi-player new bind player THEN
  filename get player file  ;
```

Furthermore, the player has to be destroyed, when the dialog is closed:

```
: dispose ( -- )
  midi-path @ IF  midi-path [ also memory ]
HandleOff [ previous ] THEN
  super dispose ;
```

So, let's define the actions bound to the buttons:

First, we need a file to load from; that's the most complicated part.

```
^^ S[ filename assign >r 2dup midi-path  r>
?player ]S
screen self file-selector new >r
s" MIDI" s" " midi-path @
IF  midi-path $@  ELSE  S" *.mid"  THEN
r> file-selector with assign
dpy xywh 2/ nip 1 swap resize map endwith
```

Then, we need a button to start playing:

```
?player player start
```

Further, a button to stop playing

```
?player player stop
```

And finally, a button to close the whole dialog

```
close
```

That all looks quite simple, and it is. You just must call the dialog "`midi`", Theseus then inserts the loading functions for the MIDI player class.

4 Future

MINOΣ now has a sufficient set of features. Theseus stabilized lately, but still doesn't support all of the features. It would be nice to create 3D objects more user-friendly (point&click or at least import from other programs). Data base input forms can be partly created automatically, since you can ask the data base about the tables' structure. On the other side, it is possible to create tables and relations pointing and clicking.

But most important: MINOΣ needs documentation, documentation, documentation.

I want to point out that MINOΣ is available under the General Public License (GPL). Since it is a library, and part of an incremental compiler, modifications (and applications are modifications!) may only given away under GPL, or as separate code (source only then). For those who don't like that, there is a commercial license, which includes printed documentation. This will take some time, since the documentation isn't written yet.

You can find MINOΣ on the web. Jens Wilke donated space on his web server, the URL is `http://www.jwdt.com/~paysan/bigforth.html`