Zero Memory Widgets
LIRIS Research Report 20030311

THIERRY EXCOFFIER
Université Claude Bernard, Lyon 1

March 11, 2003

Abstract

Widget libraries have now been developped and used for years. In all these libraries, widget instances require computer memory. But this memory is not really required, We have implemented a widget library to prove that it is possible to use zero bytes of memory per widget. In such a library, there is no widget reference, so widget programming is easier even in a low level language such as C. Programs are more robust because they do not use pointers, make no memory management and do not translate data between application and widget. To set the attributes of a widget, it is not possible to use the widget's pointer so a current state is used as in OpenGL. Most classic widgets were integrated into the library, and it should be possible to integrate widgets of all kinds without any problem.

Contents

1 Introduction

This article shows how to use zero memory widgets (ZMW). Beside the fact that they do not use memory in the user program nor in the windowing system (X11 or Windows), ZMW are easy to use because there is: no pointer or reference on the widget, no concept of widget creation or destruction, no memory leak, no function to get or set widget data, no callback or event-handling functions.

In many applications, the widget tree reflects the application's data. Hence both data and structures are duplicated. It is a burden for the programmer to synchronize these structures. This synchronization consumes computer memory and time. With ZMW, this problem disappears, as the widget tree is virtually created from the application's data.

ZMW library is as powerful as any other widget library in terms of functionalities. But it uses much more computer time for static GUIs than with classic widget libraries, because data is computed and not stored. For highly dynamic GUIs, *e.g.*, a tearable contextual menu hierarchy, ZMW library should be more efficient than classic widget libraries, because of the absence of memory management.

After some words on classic widget libraries, we explain how a ZMW library can work. Examples are given to highlight several functionalities such as widget composition, event handling, attribute setting, class creation. The last part presents usage constraints of a ZMW library and the conclusion.

2 The classic widget libraries

Widget library history is complex, only some samples are taken from this history in the X11 world. More information may be found in papers on human-computer interaction history [Mye98] or software tools [Mye95].

It is difficult to create widgets using only the X11 basic functions contained in the Xlib. So the *Intrinsic Library* [ON92] was created in order to make the widget creation process easier for the programmer. The most famous widget libraries based upon the *Intrinsic Library* are *Athena* [Fla92] and *Motif* [HFB93]. As X11, the *Intrinsic Library* is a complex piece of software because it is very generic. An interesting fact is that windows may contain a patchwork of widgets created with the *Intrinsic Library*, *e.g.*, *Athena* and *Motif* widgets.

The *Intrinsic Library* is too powerful, and hence rather complex to use. Many attempts have been made to make widgets libraries more usable by the programmers, so they can easily create new widget classes or compose GUIs. Some use a low-level programming language like C, for example the GTK [GMtGt02] library. Others rely heavily on an object-oriented language like C++, for example Amulet [MMM$^+$97] and Qt [Dal02].

In all those widget libraries, instances are created and the user manipulates references in order to modify the widgets. This is done to optimize the time to update the screen.

To receive events, the user must provide an event handler, the parameters of which are defined by the widget library. Usually, one of the event handler's parameters is a pointer to a user-defined data. If the user-defined data is an integer or a data structure, there is no problem. But if two integers are needed, a new data structure must be created, to only serve as a parameter to the event handler. An exception is Amulet, because in this library event handlers are objects and not functions, so there is no problem with parameters.

For all these reasons, widgets are fairly difficult to program. Many people use GUI builders as Glade or wxWindows. With such builders, only handlers have to be written. But GUI builders may only be used for static GUIs, in which the widget tree is predefined and does not change when an application is running. So GUI builders do not simplify programming applications where the widget trees depend upon application data.

User interface description languages are an alternative to GUI builders. The GUI description can be generated on the fly, so the interface may be dynamic.

There are other widget libraries that do not use references. The most used one is the tandem HTML/JavaScript [ECM99]. The user defines the graphical interface and how events are managed, without using references most of the time. This approach seems to be very successful, as it is now used in most web pages. There is some other interface language definition using XML/JavaScript such as Entity, or XUL in Mozilla.

These approaches, mostly without references make programming easier, but they have drawbacks: the user needs to learn a new language, and the implementation is complex because two languages (HTML/JavaScript) are required to display and interact with the web page.

Our ZMW library is based upon these ideas, but replaces HTML/JavaScript with a classic programming language.

3 How to use a Zero Memory Widget library

The following examples show how it is possible to create an easy-to-use widget library without: intermediate language, preprocessing, widget references, pointers, computer memory.

The examples are in the C language but could be easily ported to any language. The snapshots are taken from the ZMW library prototype.

The GDK library is used by ZMW as an interface to the windowing system.

3.1 How it works

The user provides a function whose execution traverses the widget tree. The action performed on the widget depends upon a current state: it may be either computing widget size, drawing on the window or analyzing an event.

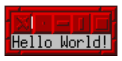
This table shows a parallel between classic widget libraries and ZMW libraries.

	Classic widget library	**Zero memory widget library**
Class	A class or a constructor	A function
Creation	Constructor call	*Not applicable*
Destruction	Destructor call	*Not applicable*
Instance	A data structure	A call to the function
Display	Widget display-method call	Some calls to the function
Event	Widget event-method call	Some calls to the function
Size change	Recompute widgets size	Nothing to do

The users of the 3D graphic libraries PHIGS [ISO97a] and OpenGL [WND+99] will recognize this opposition. In PHIGS, the library stores a 3D model to optimize visualization. When the application updates the library's 3D model, the display is updated by the library without any user intervention. With OpenGL, by default, the library does not store data. When the application updates its 3D model, it is required to redraw the whole 3D model to update the display. PHIGS has always been powerful but OpenGL is easier to use.

3.2 "Hello World" program

This program displays a label in a window.

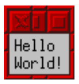

Hello World!

```
#include "zmw.h"

void hello_world(void)
{
  static GdkWindow *w = NULL ;

  ZMW(zmw_window(&w))
    {
      zmw_text("Hello␣World!") ;
    }
}

int main(int argc, char *argv[])
{
  zmw_run(argc, argv, hello_world) ;
  return 0 ;
}
```

`hello_world` is a C function, but for ZMW, it is a widget class. A function call is a widget instance. In this example, `hello_world` is the root of the widget hierarchy. `hello_world` is called by `zmw_run` to compute size, draw on the screen and receive events. The memory footprint of the program is unchanged by the `hello_world` call.

3.3 Containers or how to layout widgets

With a classic widget library, containers may be defined, as in GTK, by using a function to modify the children list or, as in *Motif*, by specifying the parent of the created widget. With ZMW both ways are impossible because there is no widget data structure, so there is no memory, hence no pointers. The solution is, as in HTML, to create a block of widgets. The parameter of `ZMW` specifies the layout and the graphic aspect of the block's contents.

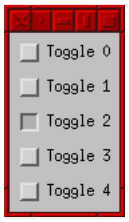

Hello
World!

```
void two_widgets(void)
{
  ZMW(zmw_box_vertical())
    {
      zmw_text("Hello") ;
      zmw_text("World!") ;
    }
}
```

3.4 The data required are stored on the user's side

With a classic widget library, data is copied to the widget and must be retrieved from the widget, so many functions are required to handle data. With ZMW, data is used as provided by the program, without any memory allocation or copy. There is no function to get or set the data.

```
void toggle_list (void)
{
  static int toggle_state [5] = {0, 0, 1, 0, 0} ;
  char toggle_name[100] ;
  int i ;

  ZMW(zmw_box_vertical())
    {
      for(i=0; i<5; i++)
        {
          sprintf(toggle_name, "Toggle␣%d", i) ;
          zmw_toggle_with_label(&toggle_state[i ], toggle_name) ;
        }
    }
}
```



The argument of the toggle widget is the integer it displays and modifies. The toggle state is stored in the application's data structure and nowhere else.

A side-effect of this method is that two toggles may be synchronized by giving them the same integer to modify.

A drawback of this method is that in some cases, a piece of data is required by the widget, but not by the application. For example, the cursor position in an editable text widget or a window pointer as in the `hello_world` program. For such cases, a resource system could be implemented to simplify user programs, this will be mentioned later in the paper.

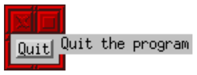
3.5 Event handling

In classic widget libraries, event handlers are connected to the widgets. As these event handlers require a fixed set of parameters, they are cumbersome to use. Another problem is that they hamper program linearity because event-handling functions should be written even if they are used only once and are as short as one line of code. The following example is written with the GTK library and creates a *quit* button.

```
void handler_destroy(GtkWidget *widget, gpointer data)
{
  printf("END\n") ;
  exit (0) ;
}
...
  button = gtk_button_new_with_label("Quit");
  gtk_signal_connect(GTK_OBJECT(button), "clicked",
                     GTK_SIGNAL_FUNC(handler_destroy), NULL);
```

With ZMW, such problems vanish. The action is defined after the widget, it is executed if the widget receives the expected event. In the following example, the function `zmw_activated` returns true if a mouse button had been pressed and released on the widget, and `zmw_tip_visible` returns true if the cursor was still and over the widget for a short time.

```
zmw_button("Quit") ;
if ( exit_allowed && zmw_activated() )
   {
     printf("END\n") ;
     exit (0) ;
   }
if ( zmw_tip_visible () )
   {
     ZMW( zmw_window_popup(&window_tip) )
       {
         zmw_text("Quit␣the␣program") ;
       }
   }
```



The button may be activated by clicking the mouse, or if the button has the focus, by pressing a key.

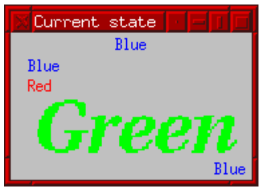
If `exit_allowed` is false, the button is disabled and so displayed grayed. It is a side-effect of the C language boolean evaluation method: if `exit_allowed` is false, the function `zmw_activated()` is not called, so the ZMW library knows that the button is not sensitive. In other languages, two “`if`” should be required.

The `Quit the program` message is a balloon tip. The tip is activated when the cursor stays still for a while. The example opens a popup window, but it is possible to open a normal window or even to insert the tip’s message after the “Quit” button in the current window.

3.6 Current state

As there is no widget pointer or reference, using a state is an easy and powerful way to give graphic attributes to widgets. There are functions to modify the state contents as the alignment, the color, the font. When a widget is defined, its attributes are those defined in the current state. This programming method has many advantages: the functions have less parameters and global changes are easier.

```
ZMW(zmw_box_vertical())
  {
    zmw_horizontal_alignment(ZMW_CENTER) ;
    zmw_foreground(ZMW_BLUE) ;
    zmw_text("Blue") ;
    ZMW(zmw_box_vertical())
      {
        zmw_horizontal_alignment(ZMW_LEFT) ;
        zmw_text("Blue") ;
        zmw_foreground(ZMW_RED) ;
        zmw_text("Red") ;
        zmw_foreground(ZMW_GREEN) ;
        zmw_font("-*-utopia-bold-*-*-*-*-400-*-*-*-*-*-1") ;
        zmw_text("Green") ;
      }
    zmw_horizontal_alignment(ZMW_RIGHT) ;
    zmw_text("Blue") ;
  }
```



As for 3D graphic trees (VRML [ISO97b] for example), attributes are saved when entering a node and restored when leaving it. So when an attribute is modified inside a node, its value is not modified outside the subtree. In the example, the last text color is blue and not green.

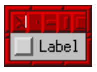
The ZMW library requires a stack of current states to handle the attributes of the widgets. The memory required depends on the size of the state stack, which is the depth of the widget tree.

In the *Fresco* widget library [LP93] there are widgets applying 3D transformation to widget trees. From the user's point of view, it seems there is a current 3D transformation but it does not really exist in the library. In fact, each widget has a 3D transformation as attribute, and the widget is displayed with this attribute. If a widget applying a 3D transformation is modified, its modification is propagated wherever required to update the 3D transformation attributes.

3.7 Widget class creation by composition

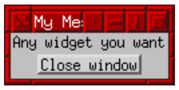
Composing existing widgets is an easy way to create new widget classes. The following example is the composition of a toggle and a label. The toggle is the small square and the label is the text to its right. The user can change the toggle state by clicking on the rectangle containing both widgets.

```
void toggle_with_label(int *value, const char *label)
{
  ZMW(zmw_box_horizontal())
    {
      zmw_toggle(value) ;
      zmw_text(label) ;
    }
  if ( zmw_activated() )
    {
      *value = 1 - *value ;
    }
}
```



In some cases, a user widget tree must be inserted into the widget composition. For example a viewport with scrollbar may scroll any widget provided by the user. The following code defines the widget class to display a message; the message can be any widget tree, the message window has a title and a close button. A boolean specifies if the message is visible.

```
/* Widget class: message */

void zmw_message(GdkWindow **w, Zmw_Boolean *visible,
                 const char *title, const char *button_name)
{
  ZMW_EXTERNAL_RESTART ; /* An external widget will be used */
  if ( * visible )
    ZMW(zmw_window(w, title))
    {
      ZMW(zmw_box_vertical())
        {
          ZMW_EXTERNAL ; /* The external widget */
          zmw_horizontal_alignment(Zmw_False) ;
          zmw_horizontal_expand(Zmw_False) ;
          zmw_button(button_name) ; /* Button to close the message */
          if ( zmw_activated() )
            * visible = Zmw_False ;
        }
    }
  ZMW_EXTERNAL_STOP ; /* An external widget has been used */
}

/* This code fragment displays the text widget "Any widget you want"
   in the message window */

  static GdkWindow *message_window=NULL ;
  static Zmw_Boolean visible = Zmw_False ;
  ...
  ZMW(zmw_message(&message_window, &visible, "My␣Message", "Close␣window"))
    {
      zmw_text("Any␣widget␣you␣want") ;
    }
```



3.8 Base widget class creation

The creation of a new type of widget is easy because there is neither memory allocation nor complex data structures to initialize, such as in the *Intrinsic Library*. The example below shows the definition of the horizontal box widget.

```
void zmw_box_horizontal(void)
{
  switch( zmw_action() )
    {
      case Zmw_Compute_Required_Size:
        zmw_box_horizontal_compute_required_size() ;
        break ;
      case Zmw_Compute_Children_Allocated_Size_And_Pre_Drawing:
      case Zmw_Compute_Children_Allocated_Size:
        zmw_box_horizontal_compute_children_allocated_size() ;
        break ;
      case Zmw_Init:
      case Zmw_Post_Drawing:
      case Zmw_Event:
        break ;
    }
}
```

Modifying existing widget classes is straightforward. The following example is the code of an enhanced box widget that draws a background.

```
void zmw_my_box_horizontal(void)
{
   if ( zmw_action() == Zmw_Compute_Children_Allocated_Size_And_Pre_Drawing )
      {
         zmw_my_box_draw_background() ;
      }
   zmw_box_horizontal() ;
}
```

4 Our implementation

In our library, the *action* to perform when a widget is evaluated can be to compute its required size, to display the widget, to dispatch the input event, to search accelerators, to display widget information in HTML, etc. The *action* is a function, so the users may add their owns *action*.

Most of the *actions* require several steps. For example, to display a box, these *sub-actions* must be evaluated:

- compute the required size of each child;
- compute the required size of the box;
- using the allocated box size, computes the children's allocated size;
- display the box background;
- display each children;
- do some post drawing.

The *sub-actions* are those required to define a new widget class as in the two previous code fragments.

The heart of the algorithm is the loop performing the *sub-actions* on the widget. This loop is hidden in the `ZMW` C language macro. When the program contains the following code fragment :

```
ZMW(zmw_horizontal_box())
   {
   ...
   }
```

It is translated as :

```
for(zmw_init_widget() ;
    zmw_horizontal_box(),(*ZMW_ACTION)() ;
    zmw_state_pop())
   {
    ...
   }
```

In the C language, the `for` construct contains three parts:

1. The initialisation: `zmw_init_widget` performs some initializations before "entering" in the widget. Its main purpose is to set the loop counter to 0.
2. The end of loop test: The following functions use the loop counter to determine their behavior.

- `zmw_horizontal_box()` perform the sub-action required by the widget, in this case an horizontal box.
- `(*ZMW_ACTION)()` This function computes the scheduling of the sub-actions in order to perform the required action. In all the cases, this function will increment the loop counter. When all the sub-actions are done, it stops the loop. If the loop continues, it pushes the current state and initializes the new state, in order to perform the action on the children defined in the loop body.

  The C implementation uses a function pointer stored in the current state. It is useful in order to change the action while traversing the widget tree. For example, on the first loop the *draw* action will become a *compute required size* action for the children.

3. The "Increment": `zmw_state_pop()` restores the current state. This insures that the loop body will always start with the same current state.

The following table contains the execution trace of the drawing of a window containing a box containing two labels A and B. To draw the window contents, it is necessary to compute sizes, so the action is set to *compute_required_size*.

Widget name	Sub-action	Action	Loop counter
Window	Initialize	draw	0
Box	Initialize	required_size	0
Label A	Initialize	required_size	0
Label A	Required_Size	required_size	1
Label B	Initialize	required_size	0
Label B	Required_Size	required_size	1
Box	Required_Size	required_size	1
Window	Allocated_Size	draw	1
Box	Initialize	draw	0
Label A	Initialize	required_size	0
Label A	Required_Size	required_size	1
Label B	Initialize	required_size	0
Label B	Required_Size	required_size	1
Box	Allocated_Size	draw	1
Label A	Initialize	draw	0
Label A	Allocated_Size	draw	1
Label A	Post_Drawing	draw	2
Label B	Initialize	draw	0
Label B	Allocated_Size	draw	1
Label B	Post_Drawing	draw	2
Box	Post_Drawing	draw	2
Window	Post_Drawing	draw	2

The *pre-drawing* is done when computing *allocated_size* sub-action. As the required size is not stored in the widget, it must be computed each time it is needed. In this trace we can see that the label required sizes are computed twice.

5 Constraints of a Zero Memory Widget library

The programming interface to the ZMW is minimal. The library has no information about widgets, which may lead to minor problems.

5.1 Program state modification

The function defining a widget must verify the following property in order to be used successfully by ZMW:

Two successive calls to a widget must do exactly the same thing. The only exception is when a widget receives an event, in which case the user program may change its state.

This is required because the function will be called at least twice. The first time to compute the widget size and the second one, to draw the widget. If the program state changes between these two calls, the library does not know about the change and the widget will be displayed incorrectly.

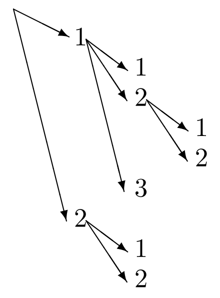
5.2 Widget naming

Widget naming is required to reference widgets. Referencing a widget is common, to indicate, for example:

- the deepest pull-down menu displayed, if there is one;
- the widget with the focus;
- the widget being dragged.

The simplest way to obtain widget references is to name them as in all widget libraries. If the user does not name the widgets, a default naming scheme is used:

Widget tree



Widget names

1
1/1
1/2
1/2/1
1/2/1
1/3
2
2/1
2/2

With a static GUI, the default naming scheme works, but with a highly dynamic GUI, some problems arise. For example in the following code, the number of buttons depends on the value of the variable `display_A`, and the variable is modified while the window is visible.

```
ZMW(zmw_box_vertical())
  {
    if ( display_A )
      zwm_button("A") ;
    zwm_button("B") ;
  }
```

The default naming scheme will not work in the following sequence of events, in which the focused widget changes without user action :

1. The `display_A` variable is false.
2. The user clicks on “button B”, the button takes the focus, so the name of the focused widget is `.../1`
3. The `display_A` variable becomes true.

4. The “button A” name is `.../1` so “button A” has the focus.
5. The “button B” name is `.../2` so it is no more focused.

There is no automatic solution to this problem, because the library cannot distinguish between widgets. The programmer must set the widget’s name in the case of highly dynamic GUIs.

```
ZMW(zmw_box_vertical())
  {
     if ( display_A )
        zwm_button("A") ;
     zmw_name("B") ;
     zmw_button("B") ;
  }
```

With this program, “B” is always named `.../B` and “A” is always named `.../1` so there is no confusion.

5.3 Seldom used widget data

In most widget classes, some widget data is not useful to the programmer. The window pointer (`GdkWindow` typed in the examples) is never used. The cursor position in an editable text is rarely used. The program would be more readable if those widget data were invisible to the programmer; so the widget must contain some data when the application does not provide storage space.

The way to add these data in a no-memory widget library is to store the data as a resource. As in classic widget libraries, a resource is a triple: (*widget name, attribute, value).* The X11 resource `xterm.vt100.background: black` means that the attribute `background` of the widget `xterm.vt100` is set to `black`.

A ZMW library could use the same system to store the values that are not useful to the user. The resources could be used as in other systems to allow the user to configure the application’s appearance.

6 Conclusion

This paper shows that with ZMW it is possible to make widget programming easy even in a language without a garbage collector such as C.

ZMW have another advantage: once the library API is defined, the widget implementation may be swapped on the fly with no overhead, because there is no widget instance. So it is straightforward to create themable widgets.

The examples given run with the library prototype. Readers interested in the way to make a ZMW library may download the prototype from :
`http://www710.univ-lyon1.fr/~exco/ZMW/`

The implementation allows hierarchical tearable menus, drag and drop, scrollbars, viewports, selection handles, tips, notebook, accelerators, pictures, animations and a minimal file browser.

The rendering is accelerated using a geometry cache. In the future OpenGL will be used in place of GDK to display widget because its current graphical state is more adapted to a ZMW library.

ZMW are now possible because current computers are powerful enough to display all the widgets in a very short amount of time. A lot of CPU time will be wasted if a ZMW library is used to create a static GUI, in the other hand, it makes application coding simple.

References

[Dal02] Matthias Kalle Dalheimer. *Programming with Qt, 2nd Edition. Writing Portable GUI applications on Unix and Win32.* O'Reilly, 2002.

[ECM99] ECMA. Standard ecma-262: Ecmascript language specification. ftp://ftp.ecma.ch/ecma-st/Ecma-262.pdf, 1999.

[Fla92] David Flanagan, editor. *Volume 5: X Toolkit Intrinsics Reference Manual.* O'Reilly, 1992.

[GMtGt02] Tony Gale, Ian Main, and the GTK team. Gtk+ 2.0 tutorial. http://www.gtk.org/tutorial/, 2002.

[HFB93] Dan Heller, Paula Ferguson, and David Brennan. *Volume 6A: Motif Programming Manual.* O'Reilly, 1993.

[ISO97a] ISO. Programmer's hierarchical interactive graphics system (PHIGS) – part 1: Functional description, 1997.

[ISO97b] ISO. The virtual reality modeling language – part 1: Functional specification and utf-8 encoding. http://www.web3d.org/fs_specifications.htm, 1997.

[LP93] M. Linton and C. Price. Building distributed user interfaces with fresco. In *Proceedings of the Seventh X Technical Conference, Boston, Massachusetts*, pages 77–87, January 1993.

[MMM$^+$97] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, June 1997.

[Mye95] Brad A. Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction*, 2(1):54–103, March 1995.

[Mye98] Brad A. Myers. A brief history of human-computer interaction technology. *ACM Transactions on Computer-Human Interaction*, 5(2):44–54, March 1998.

[ON92] Tim O'Reilly and Adrian Nye. *Volume 4M: X Toolkit, Intrinsics Programming Manual, Motif Edition.* O'Reilly, 1992.

[WND$^+$99] Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, and OpenGL Architecture Review Board. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Addison-Wesley, 1999.