# Wimp Topics - Displaying Text in Windows

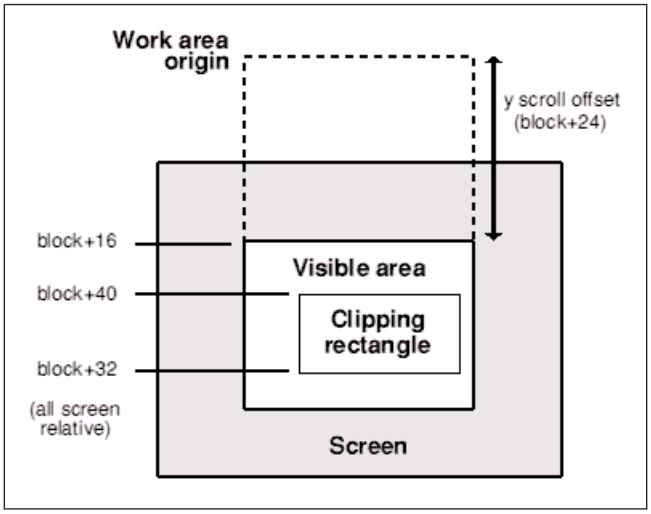
Alan Wrigley describes some useful redraw techniques to help you with

Most programmers tackling the Wimp for the first time design windows in which information can be displayed using icons. Often this is quite adequate, since many programs present information that can be neatly packaged up into discrete units - the current status of an operation, co-ordinates, and so on. But sooner or later there comes a need to handle text of a more complex and unpredictable nature - the contents of a help file, say, or a list of items produced by a database search. You may even need to write a program which edits or processes text in some way.

For situations such as these, and indeed for many other purposes such as drawing graphics, the Wimp provides the means for you to draw directly to your window without the use of icons. The basic techniques of window redraw were covered in Mastering the Wimp (RISC User 3:3, 3:4 and 4:4), and in the book Wimp Programming for All. It is assumed that you are familiar with these basic techniques, but it is worth repeating the main elements of the redraw process here.

Firstly, the window in question must have bit 4 of the window flags unset (which indicates that it wishes to receive redraw requests). The Wimp will then return poll reason code 1 to the task whenever any section of the window needs redrawing. This will happen when the window is first opened, or when a part of it which was previously hidden becomes visible (for example if it is scrolled or moved from behind another window).

Redraw must be carried out using a well-defined process within a redraw loop. You must not write to the screen outside a proper redraw loop. The loop is entered after calling Wimp_RedrawWindow , which returns the information you need to carry out the redraw. The Wimp tells you the co-ordinates of the current visible area of the window, the scroll bar positions, and the co-ordinates of the specific rectangle that needs to be redrawn (known as the clipping rectangle). You must perform the loop repeatedly, redrawing the requested areas, until no more rectangles are returned. As an example, imagine that another window is dragged diagonally across yours. Each time the window moves a measurable distance, it will expose an L-shaped section of your window. The Wimp will send you a redraw request, and will return two consecutive rectangles which together will cover the newly exposed area. You then redraw both these rectangles before passing control back to Wimp_Poll .




The relationship between the window's work area and the screen co-ordinates returned by "Wimp_RedrawWindow"

When the window to be redrawn contains lines of text, the redraw process can be broken down into three stages - calculating the co-ordinates of the area to be redrawn, deciding which section of text relates to those co-ordinates, and actually putting the text on the screen. Each of these stages will be described in turn.

## CLIPPING RECTANGLE CO-ORDINATES

For small amounts of text you can get away with ignoring the clipping rectangle and just redrawing the whole of the work area at each redraw request. Provided you are in a proper redraw loop, it is always quite safe to draw more than is required since anything outside the clipping rectangle will be ignored anyway.

For larger amounts this approach will result in an unacceptable time delay, particularly when the window is scrolled down, and this will increase the further away you are from the work area origin. In these circumstances you must take note of the clipping rectangle and only redraw the parts that are needed.

Where text is concerned, this is often not as difficult as it sounds. You can safely ignore the x co-ordinate of the rectangle, and just draw complete lines of text each time, since the overhead involved in drawing part of a line unnecessarily is offset by the fact that you don t need to calculate where on the line to start and end. This means that you only need to take the y co-ordinates into account; and since the text in most cases will have been displayed in equally-spaced lines, it is relatively easy to work out on which lines the clipping rectangle begins and ends, using the values returned by the Wimp.

To do this, you first need to be aware that all the co-ordinates returned by the Wimp are relative to the screen, and not to the window s work area. However, your program only knows where the text is positioned in relation to the window, not the screen, so you must perform some calculations to translate between the two. The notional screen co-ordinates of the work area origin can be found by subtracting the vertical scroll position from the top of the visible area. I say notional because its position may actually be way off the top of the screen if the window has been scrolled (see Figure 1). Now you can subtract from this the maximum y co-ordinate of the clipping rectangle to get its top relative to the work area, and similarly the minimum y co-ordinate to get its bottom. Finally divide these by the line spacing in OS units to find out the start and end lines of the section that needs to be redrawn. The following procedure performs these actions:

```
1000 DEF PROCredraw
1010 SYS "Wimp_GetRectangle",,block% TO more%
1020 WHILE more%
1030  xorigin%=block%!4-block%!20
1040  yorigin%=block%!16-block%!24
1050  maxyclip%=yorigin%-block%!40
1060  minyclip%=yorigin%-block%!32
1070  topline%=maxyclip% DIV 32
1080  bottomline%=1+minyclip% DIV 32
1090  REM redraw code here
1130  SYS "Wimp_GetRectangle",,block% TO more%
1140 ENDWHILE
1150 ENDPROC
```

This code should be easy to understand as it follows the process outlined above. First of all we get the work area origin in xorigin% and yorigin%. Then we get the clipping rectangle which lies between maxyclip% and minyclip% in work area co-ordinates. Finally we translate this rectangle into line numbers, stored in topline% and bottomline%. You will notice that we have assumed a line spacing of 32 OS units, which is the same as that used by Edit. If you want more spacious-looking text you could use 36 or 40 instead (though if you are writing in Assembler you will find 32 better as you can do your division using shift instructions). We have also incremented the bottom line by 1 to ensure that the whole of the line gets redrawn.

This example assumes that the text will start at the very edges of the window. In practice it looks neater to leave a small gap, say 4 OS units, at both the top and the left-hand edge. This can be catered for automatically by offsetting the origin by that amount in both directions:

```
1030 xorigin%=block%!4-block%!20+4
1040 yorigin%=block%!16-block%!24-4
```

## MATCHING TEXT TO CO-ORDINATES

So much for the redraw theory, but how do you match up the line numbers to the text you actually want to display? This depends on how you store the text, and can be done in several ways. For example, you could hold all the text in a string array, say text$(), with each element of the array containing one line of text. For the purposes of redraw, this is the simplest solution since you can then just write the text from text$(topline%) to text$(bottomline%) without further ado. The text could be put into the array in the first place by reading DATA statements, or by reading from a file (though if the text doesn t already have line terminators in the right places, you must calculate their positions for yourself so that the correct number of characters is included in each line in the array).

For large chunks of text, or text whose content may change during execution of the program, the array approach is less convenient, since

one small alteration to the text may require the contents of some or all of the array elements to be changed around. An alternative method is to store the text directly in memory, and to keep an array of pointers to the start of each line. If the text is altered, the pointers can be updated without having to alter all the array elements. There are various ways of allocating a chunk of memory in which to store the text - these were described in Technical Queries (RISC User 5:4).

## WRITING TEXT TO THE WINDOW

The process of actually getting the text into the window is quite straightforward. We will stick to the system font for the purposes of this article - outline fonts are a bit more tricky since you obviously have to know the point size before you can position text in the window. The text is output in exactly the same way as you would write to the screen in VDU 5 mode in a non-multi-tasking program, using the same VDU and SWI calls, and specifying the actual screen co-ordinates. However, since it is being done within a redraw loop, the Wimp makes sure that only your window is written to, and ignores any co-ordinates that lie elsewhere. Before writing any line of text, the graphics cursor must be positioned. Having converted from screen co-ordinates to work area co-ordinates in order to calculate the correct text line, the top of the line must be converted back to screen co-ordinates to find the y position for the cursor (we can't just use the value returned for the clipping rectangle since that may not fall on a line boundary). The x position of the cursor will be the same as the work area origin x co-ordinate, since we are drawing complete lines each time. So the code to display the text would look something like this, assuming the array method of storage:

```
1090 FOR i%=topline% TO bottomline%
1100 MOVE xorigin%,yorigin%-i%*32
1110 PRINT text$(i%)
1120 NEXT
```

## WINDOW SCROLLING

When scrolling a text display, it usually looks neater if the top of the window coincides exactly with a line of text. This can easily be achieved by adjusting the scroll bar whenever a request to open the window is received. When the redraw request is then issued, the scroll bar will reflect the adjusted setting and your redraw code will automatically put the text in the right place. If the text is spaced at 32 OS units, all you have to do is ensure that the scroll bar is a multiple of 32 before calling Wimp_OpenWindow :

```
block%!24=block%!24 AND &FFFFFFE0
SYS "Wimp_OpenWindow",,block%
```

## SETTING THE EXTENT

One problem you often face is how to make sure that the vertical extent of the window is the right size for the text you want to display. If it's too short, some of the text will never be seen, while if it's too long, there will be a large and unnecessary amount of space at the bottom. The solution to this is to set the window extent as soon as the length of the text is known (i.e. when loaded from a file or read from DATA statements), and to reset it if ever the length of the text changes. This is quite easily done with a call to Wimp_SetExtent . For example, if the text is 100 lines long with a line spacing of 32, then allowing for the 4-unit offset at the top, you would need an extent of 3204. Assuming that you have made the window the full width of a mode 12 screen (1280 units), the call would look like this:

```
!block%=0:block%!4=3204
block%!8=1280:block%!12=0
SYS "Wimp_SetExtent",whandle%,block%
```

## MARKING TEXT

Finally I want to look at the situation where you may want to mark a line or lines of text. For example, in Edit you can use Select and Adjust, or a drag operation, to mark a section of text, which is shown by inverting the colours. This particular process is quite complex, but a simpler situation would be where you might want to mark a single line of text when the user clicks over it, perhaps to indicate which item has been chosen from a search list. The procedure we will adopt is to find out the co-ordinates of the click, match those to a line of text, set a variable (mark%) to show the selected line, and then to force a redraw of that line with a different background colour (mid-grey). To do the latter, all we need to add to the redraw code is the following:

```
1092 IF i%=mark% THEN
1094 SYS "Wimp_SetColour",3
1096 RECTANGLE FILL xorigin%,yorigin%-i%
*32,1280,-32
1098 SYS "Wimp_SetColour",7
1099 ENDIF
```

This month's magazine disc contains an application called Window Writer which demonstrates the principles covered in this