GUI Development

by Brandon F.

Why use this document?

The purpose of this tutorial is to try to explain how to create a simple Graphical User Interface Program for use in a DOS environment or for use in a home-brew type of Operating System. I was actually asked to write this tutorial, which will some day be posted on the internet.

Requirements for this tutorial

For this tutorial on GUI developement, I HIGHLY recommend reading the following:

- 256-color Graphics Tutorials (Specifically, read the BITMAPs and DOUBLEBUFFER sections).
- C/C++ Programming Tutorials (If you know C / C++, MAKE SURE you know Linked lists and Binary Trees).

You will also need:

- A C/C++ Compiler (I use BorlandC v3.0, although DJGPP could definitely do it).
- A VGA compatible display adapter and monitor (At least 64KBytes video RAM for mode 0x13).
- A 286 based PC (That's the minimum that BorlandC allows), with 2MBytes of RAM.

OS STEP: Make sure you can handle a GUI

In order for you to correctly run a GUI using these methods, you must have included in your Operating System the following items: A memory manager of some sort, and (if you run in protected mode) a V86 handler so you can call real mode interrupts. Your memory manager MUST include a malloc( ) or equivalent, realloc( ) or equivalent, and free( ) or equivalent. We use binary trees to store the window and control lists, and therefore we need to dynamically allocate memory. A V86 handler is needed to be implemented in order to make real mode BIOS calls. These calls are ONLY used to set the graphics mode. If you wish, you may skip the V86 handler if you have set mode equivalent functions that change the video registers. You can call real mode interrupts without the V86 handler if your Operating System is a real mode-type OS.

GUI STEP: Set up your GUI environment

This is the step that you start coding your GUI. You will need to make a graphics library that uses bitmaps and double buffers. Example routines are posted here:

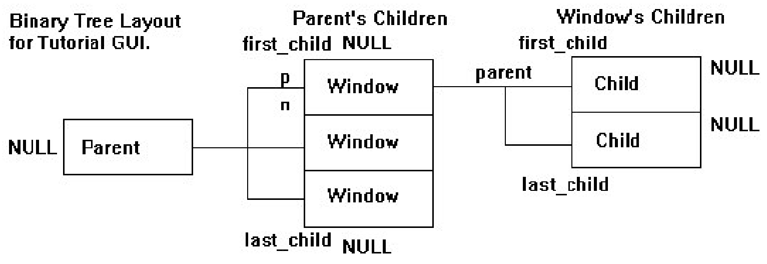
```
unsigned char *VGA = (unsigned char *)0xA0000000L;
unsigned char *dbl_buffer;

typedef struct tagBITMAP              /* the structure for a bitmap. */
{
    unsigned int width;
    unsigned int height;
    unsigned char *data;
} BITMAP;

typedef struct tagRECT
{
    long x1;
    long y1;
    long x2;
    long y2;
} RECT;

void init_dbl_buffer(void)
{
    dbl_buffer = (unsigned char *) malloc (SCREEN_WIDTH * SCREEN_HEIGHT);
    if (dbl_buffer == NULL)
    {
        printf("Not enough memory for double buffer.\n");
        getch();
        exit(1);
    }
}

void update_screen(void)
{
    #ifdef VERTICAL_RETRACE
      while ((inportb(0x3DA) & 0x08));
      while (!(inportb(0x3DA) & 0x08));
    #endif
    memcpy(VGA, dbl_buffer, (unsigned int)(SCREEN_WIDTH * SCREEN_HEIGHT));
}

void setpixel (BITMAP *bmp, int x, int y, unsigned char color)
{
    bmp->data[y * bmp->width + x];
}

/* Draws a filled in rectangle IN A BITMAP. To fill a full bitmap call as drawrect (&bmp, 0, 0, bmp.width, bmp.height, color);*/
void drawrect(BITMAP *bmp, unsigned short x, unsigned short y, unsigned short x2, unsigned short y2, unsigned char color)
```

```
{
    unsigned short tx, ty;
    for (ty = y; ty < y2; ty++)
        for (tx = x; tx < x2; tx++)
            setpixel (bmp, tx, ty, color);
}

void draw_bitmap_old(BITMAP *bmp, int x, int y)
{
    int j;
    unsigned int screen_offset = (y << 8) + (y << 6) + x;
    unsigned int bitmap_offset = 0;

    for(j = 0; j < bmp->height; j++)
    {
        memcpy(&dbl_buffer[screen_offset], &bmp->data[bitmap_offset], bmp->width);

        bitmap_offset += bmp->width;
        screen_offset += SCREEN_WIDTH;
    }
}

void main()
{
    unsigned char key;
    do
    {
        key = 0;
        if (kbhit()) key = getch();

        /* You must clear the double buffer every time to avoid evil messes (go ahead and try without this, you will see) */
        memset (dbl_buffer, 0, SCREEN_WIDTH * SCREEN_HEIGHT);

        /* DRAW ALL BITMAPS AND DO GUI CODE HERE */

        /* Draws the double buffer */
        update_screen();

    } while (key != 27);        /* keep going until escape */
}
```

Now, how does this code work? I first set up a pointer to the video memory at address 0xA0000000. If you are running in a protected mode OS (except for one that emulates DOS), you must set up this variable as 0xA0000. The BITMAP and RECT structures should be fairly straightforward: RECT defines an area on the screen ([x1,y1][x2,y2]) and BITMAP defines a bitmap in memory. The way it works is you use the width of the bitmap to find out how much is on a single horizontal line in the bitmap. You loop this through from 0 until height to draw. Do not forget to set the screen offset (see in the draw_bitmap functions). To make the GUI or program(you can use the above code for nearly ANY graphical program) run smoothly, you can do something called double buffering. This means that you allocate an area that is the size of the screen in memory (use malloc or equivalent) and draw directly to it instead of to the video memory. When finished drawing, you write the doublebuffer to the video memory and you image is shown. Please note, that you need to wait for what is called a "Vertical Retrace". This is the last phase that the video card goes through when finishing a screen update. When this is finished, then you draw the doublebuffer to the screen and the screen will not flicker. It is alot faster for the computer to draw to system memory rather than make an I/O request everytime you want to write to the screen.

GUI STEP: GUI Theory - binary trees

Binary tree structures are critical to my GUI method developement. The structure is like so: A parent window which supports child windows. The basic window structure should look like so:



Or as see in code, every box from above looks like:

```
struct WINDLIST
{
    RECT position;
    unsigned long handle;
    unsigned char *caption;
    unsigned long flags;
    unsigned char needs_repaint;

    BITMAP wbmp;

    struct WINDLIST *prev, *next;
    struct WINDLIST *first_child, *last_child;
    struct WINDLIST *parent;
};
```

The RECT position contains the X and Y coordinates as well as X2 and Y2 coordinates. Just like if you call a drawbox function you need to give x1, y1, x2, y2... As a rule of safety, DO NOT make x2 or y2 less than x1 or y1. The parent pointer from above structure points to the window structure below it and so on... so wnd.parent will give you it's parent... most likely the desktop, unless you decide to implement MDI forms(I have not done so yet). To get the top most window in the chain, you would go parent->first_child. This window will be drawn last on the screen... and the lowest zordered window will be drawn first (parent->last_child). You can also access the next and previous windows... This is how you would cycle through the windows for drawing: Call once like this: "repaint_children(0);". This will draw the parent window(window 0), and then cycle through all the children and their children too. If you change a window (like change the titlebar color), then change the "needs_redraw" variable to 1. When you call repaint children, it will redraw it's bitmap.

```
static void repaint_children(unsigned long parent)
{
    struct WINDLIST *wnd, *child;
    if (parent >= wm_num_windows)
        return;

    wnd = wm_handles[parent];
    if (wnd == NULL)
        return;

    if (wnd->needs_repaint)
    {
        windowborder(&wnd->wbmp, 0, 0, wnd->wbmp.width, wnd->wbmp.height);
        wnd->needs_repaint = 0;
    }

    draw_bitmap(&wnd->wbmp, wnd->position.x1, wnd->position.y1);
    for (child = wnd->first_child; child != NULL; child = child->next)
        repaint_children(child->handle);
}
```

Now, you will notice that the windows all have bitmaps. You need to fill in the bitmap structure for each window upon it's creation. This means that wnd->wbmp.width = wnd->position.x2 - wnd->position.x1 and so on, as well as allocate the bitmap's data field (This is where all the window's viewable stuff is drawn). If the window's bitmap is not allocated correctly, DO NOT ALLOW IT TO DRAW as it will crash your GUI and possibly your whole machine. Instead, you should shut the GUI down and say there is no more memory. You may notice the "wm_handles" variable as well. Declare it as struct WINDLIST **wm_handles. It's an array of pointers to windows. Also declare a handle counter "wm_num_handles" as a long. Absolutely nasty... Here's how you work it:

On GUI init, you must initialize the list of windows, wm_handles, as well as create it's first window... like a desktop window:

```
    wm_handles = malloc(sizeof(struct WINDLIST *));
    wm_handles[0] = &wm_system_parent_window;
    wm_num_windows = 1;
```

On window creation(createwin function) you must resize the wm_handle variable to accommodate more windows:

```
    struct WINDLIST *wnd;

    wnd = malloc(sizeof(*wnd));
    wm_handles = realloc(wm_handles, sizeof(struct WINDLIST *) * (wm_num_windows + 1));
    wm_handles[wm_num_windows] = wnd;

    memset(wnd, 0, sizeof(*wnd));
    wnd->handle = wm_num_windows++;

    /* set window variables here... Fill them ALL IN */
```

To move a window to the front of a list, you must FIRST unlink the window from the list by pointing the next and previous windows to eachother:

```
    /* Remove window from the parent's list of children */
    if (wnd->prev != NULL)
        wnd->prev->next = wnd->next;
    if (wnd->next != NULL)
        wnd->next->prev = wnd->prev;
    if (wnd == wnd->parent->first_child)
        wnd->parent->first_child = wnd->next;
    if (wnd == wnd->parent->last_child)
        wnd->parent->last_child = wnd->prev;
```

...and THEN add it to the end of the list, by modifying the last window in the chain so that it points to "this" window (wnd). Change wnd so that it points to the previous last window:

```
    /* Add window to end of parent's list of children */
    wnd->prev = wnd->parent->last_child;
    wnd->next = NULL;
    if (wnd->parent->last_child != NULL)
        wnd->parent->last_child->next = wnd;
    wnd->parent->last_child = wnd;
    if (wnd->parent->first_child == NULL)
        wnd->parent->first_child = wnd;
```

The last difficult part of the GUI list manipulation is checking what window the point (x, y) is in. Checking the top most window first, and then the next top most, etc... returning when the first window with the point (x, y) is found. You may notice that it calls itself as well. It calls itself with the children windows so that they may be checked too:

```
struct WINDLIST *inwhatwin(struct WINDLIST *parent, int x, int y)
{
    struct WINDLIST *child, *hit;

    for (child = parent->last_child; child != NULL; child = child->prev)
    {
        hit = inwhatwin(child, x, y);
        if (hit != NULL) return hit;
    }

    if (pt_inrect(parent->position, x, y)) return parent;
    return NULL;
}
```

Using this basic information, you should have the basic frameworkings of a very simple GUI, using simple filled boxes as windows. All you have to do is create simple wrapper code to get mouse or keyboard input to move the windows. Simply move the x1, y1, x2, y2 coordinates based on the cursor coords...

GUI STEP: BEYOND THIS GUI

Now with this basic GUI, you can add things like control support(buttons, textboxes, labels, you get the idea), and menu support. I added control support in about 2-3 hours of coding and being tired. Resizing is not difficult, you just need to compare the window's width and height with that of it's bitmap's, and then resize the bitmap accordingly, and set it's "needs_repaint" to 1. That way, the newly resized window will be resized and your machine will not die from bad color values. For menus, I am still experimenting with them. They are going to be a real pain(I think). Here's a hint that I found out: You need a stucture for a menu and a structure for each menu selection. An example is "file" is a menu, but "new", "save", "open", and "exit" are selections. Each selection should be given the opportunity(give it a pointer to a menu) to drop it's own menu. This creates something like a "start->programs->accessories" type of thing for you windows users. Textboxes will also be a pain, they need a text variable that can be resized dynamically, every KByte or so, and you need a BITMAP for it's control (visible).