GUI

From OSDev Wiki

A **Graphical User Interface** or **GUI** uses graphics in combination with text to display output. GUIs usually require a mouse to be able to take input and to be used to their fullest extent.

Contents

Requirements

Coding a GUI takes lots of time, knowledge, and patience. The OS has to have a graphics driver and a mouse driver at the very least, so it can check for mouse clicks on areas like buttons on the screen. Like with most other things, coding a GUI in real mode would probably be easier than in protected mode because of the availability of BIOS interrupt calls.

You need to know how to get into a graphics mode and plot pixels, lines, and shapes on the screen for starters. You also need to know about the signals that PS/2 mice send when they are moved and buttons are pressed and held down, and how to implement a driver for handling all that stuff.

Techniques

There are many ways to write a GUI, the following technique is a simple, quick and dirty way to create a generic GUI for your OS. The tutorial assumes the kernel is running in protected mode and the video resolution has been set using VBE in a linear frame buffer and multi-tasking is already enabled in your kernel. The kernel should probe the BIOS for the VBE Mode Info Block (see Getting VBE Mode Info), which returns information about the established video resolution mode.

The Screen module creates a Z-Buffer for off screen writing and exposes methods used by other modules to write to the buffer.

```
void InitScreen();
void PutPixel(int X, int Y, int RGB);
void PutRect(int X, int Y, int Width, int Height, int RGB);
void PutLine(int X1, int Y1, int X2, int Y2, int RGB);
```

Somewhere in real mode, the VESA BIOS Extentions were called to set the resolution (int 0x10 function 0x4f02) and the mode information was retrieved from VBE (int 0x10 function 0x4f01). The mode info is stored in a VbeModeInfo struct as outlined in the VBE documentation and here. This tutorial assumes a flat/linear frame buffer is selected (bit 14 of BX register is set when calling function 0x4f01). This will map the video memory to the processors addess space making it accessible by the kernel (if the segment desciptors are setup right) This way no video page swapping is needed, just one giant memcpy.

During initialization, a Z-Buffer is allocated large enough to hold all the pixels and a thread is kicked off to refresh the screen. The refresh proc just copies the data from the z-buffer to the video memory in a continuous loop:

```
void InitScreen(){
    ScreenBufferSize = VbeModeInfoBlock.XResolution * VbeModeInfoBlock.\
                                                    * (VbeModeInfoBlock.
    ScreenZBuffer = alloc(ScreenBufferSize);
    CreateThread(ScreenRefreshProc, NULL);
}

void ScreenRefreshProc(){
   for (;;){
       PaintDesktop(); //paints the desktop and all windows
       PaintCursor(); //finally paints the cursor so it's on top
#if WAIT_FOR_VERTICAL_RETRACE
       while ((inportb(0x3DA) & 0x08));
       while (!(inportb(0x3DA) & 0x08));
#endif
       memcpy(VbeModeInfoBlock.VideoBaseAddr, ScreenZBuffer, ScreenBuffe
 }
}
```

Writing to RAM is generally faster than writing to video memory and many areas of the Z-buffer will be repeatidly over written during a single screen refresh operation, making a z-buffer neccessary.

Now a function to plot pixels to the z-buffer. This will be used by other methods instead of writing directly to video memory:

```
void PutPixel(int x,int y, int color){
   //do not write memory outside the screen buffer, check parameters agc
   if (x<0 || x>VbeModeInfoBlock.XResolution || y<0 || y>VbeModeInfoBloc
   if (x) x = (x*(VbeModeInfoBlock.BitsPerPixel>>3));
   if (y) y = (y*VbeModeInfoBlock.BytesPerScanLine);
   register char * cTemp;
   cTemp = &g_ScreenBuffer[x+y];
   cTemp[0] = color & 0xff;
   cTemp[1] = (color>>8) & 0xff;
   cTemp[2] = (color>>16) & 0xff;
}
```

PutLine() and PutRect() a should check the boundaries of the X and Y parameters to ensure no pixels are plotted outside the allocated memory area of the z-buffer. This prevents memory corruption and the pixels wont be visible on the display anyway. I'll leave the PutRect and PutLine implementation up to you or may come back and fill it in later.

In the RefreshScreenProc() method you'll notice the PaintDesktop() call. This GUI method works by painting every object from the bottom up. First the desktop background is painted then each window is painted on top of it (or one another depending on the z-order of the windows and their positions):

```
void PaintDesktop(){
   //fill the background of the desktop
   PutRect(0,0,Screen.Width, Screen.Height, 0xc0c0c0c0);
   //now tell every child window to paint itself:
   for (int i=0 ; i<ChildWindows.size() ; i++){
       ChildWindows[i].Paint();
   }
}
```

Each child window has it's own Paint() method as well and subsequently calls each of it's child window's Paint() method:

```
void Window::Paint(){
    //paint a navy blue window
    PutRect(this->Left, this->Top, this->Width, this->Height, rgbNavy);
    //put a small red square in the top right hand corner of the window
    PutRect(this->Left + this->Width - 5, this->Top, 5, 5, rgbRed);
    //put the title bar text at the top of the window
    Put8x8String(this->Left+1, this->Top+1, this->Title, rgbBlack);
    for (int i=0 ; i<ChildWindows.size() ; i++){
       ChildWindows[i].Paint();
   }
}
```

After the Paint() method of all the windows have been called, the RefreshScreenProc calls PaintCursor():

```
void PaintCursor(){
   //just make a white box at the cursor position:
   PutRect(Mouse.X, Mouse.Y, 5, 5, rgbWhite);
}
```

TODO

Theres a lot that can be done to improve the quality and performance. Using this bottom up approach means that each area of the screen could potentially be overwritten several times if multiple windows are stacked on top of one another. There is a lot of overhead re-rendering the same windows for every refresh itteration. While this technique has a low memory requirement, performance can be greatly improved by pre-rendering windows in a seperate buffer and copying the contents to the screen buffer. Each Window would have it's own screen buffer where rendering is performed only when neccessary (creation, resize, etc). A similar technique is used here (http://www.osdever.net/tutorials/view/gui-development) . In this (http://www.osdever.net/tutorials/view/gui-development) example each window has it's own canvas and the contents are copied to the screen buffer during refresh operations and rendering is performed only once. The downside to this technique is the additional memory requirements.

Some suggestions and things to watch out for:

- All drawing should be performed on the z-buffer. Drawing to the video memory will get overwritten by the ScreenRefreshProc.
- You will need some way to keep track of child windows and to setup their z-order (display order), so top level windows get drawn last, bottom level windows get drawn first.
- Make more calls in the window's Draw() method to make better looking windows. Add a border and some 3D effects.
- Pixel plotting differs depending on resolution and color depth. You'll want fast pixel plotting methods for each resolution and color depth.
- See Drawing In Protected Mode for writing strings using fixed width fonts.
- See Simple Scalable Fonts for a simple scalable font technique.
- Use Bresenham's line drawing algorithms for best performance and visual appeal.
- PutRect and PutLine should have their own pixel plotting (calling PutPixel on a large rect will slow rendering WAY down).
- There are several ways to handle mouse and keyboard input. None are terribly difficult. Perhaps the easiest is for each Window to capture each mouse and keyboard event and determine whether or not to process them. Another way is for the mouse and keyboard modules to determine which windows to send messages or events to. These generally are implementation specific requirements.
- Use an optimized version of memcpy. Otherwise, on emulators like Bochs, screen updates will be painfully slow. (See: Optimized memory functions)

The new Alternative: UEFI

Instead of using VBE or real mode BIOS calls, you can use the (U)EFI methods, provided that you make your OS run on (U)EFI and not on old clunky BIOS.

See Also

Articles

- Text UI
- Getting VBE Mode Info
- Drawing In Protected Mode
- GUI Development (http://osdever.net/tutorials/GUI_tut.php)

Threads

- Alpha Blending

Categories: Video | Graphical UI
