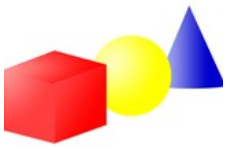
The OWLNext Journal



Number: 1 **Jun, 2009**

Hello OWLNext users!

This is the first issue of the OWLNext journal. The idea that motivated is to provide a better way to communicate to the users how to get the most of OWL & OWLNext framework, how to know the latest technologies and to provide a better way to reach more users in a easy way.

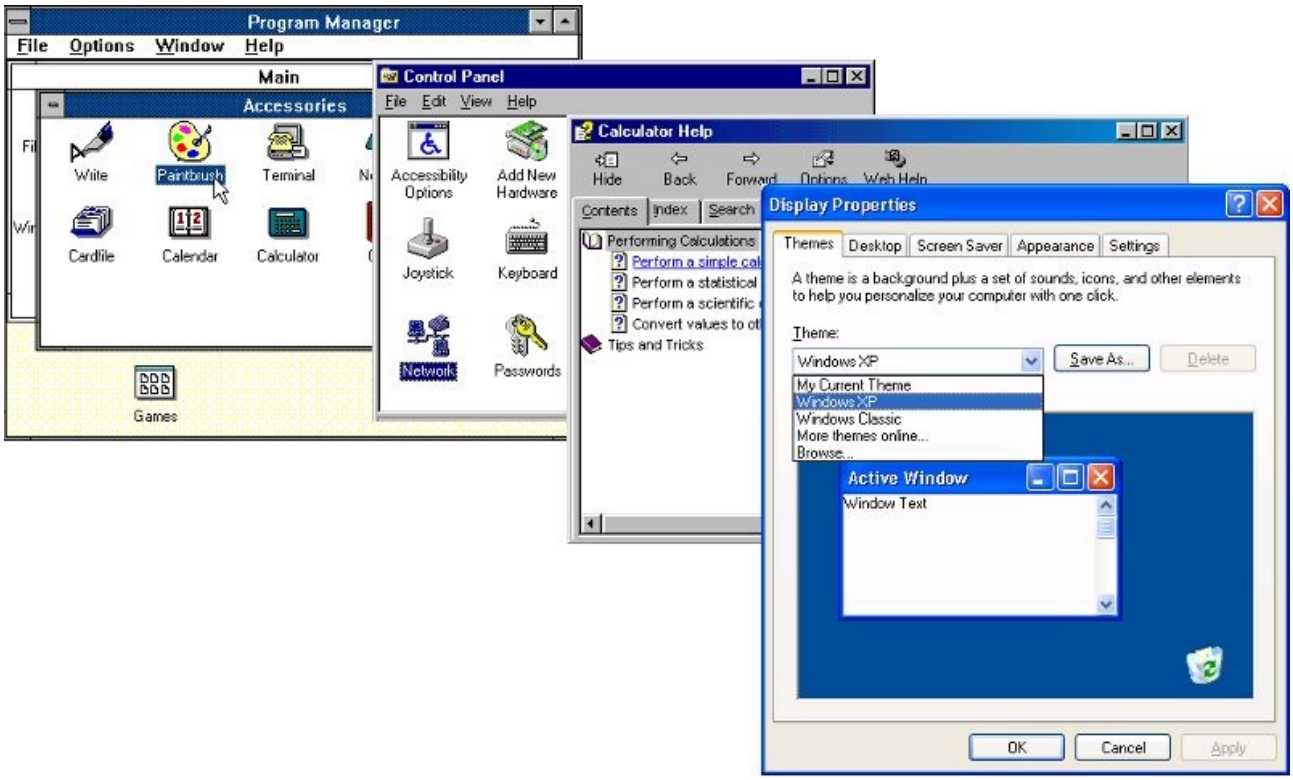
In this first issue we will introduce a cool technology to provide a custom, candy and sexy good looking frame for your applications.

Across several issues, I expect to review more and more state-of-the-art technologies, stay tuned and enjoy.

Sincerely yours
Sebastian Ledesma

Drawing in the Non-Client area: TCoolFrame.

Since the beginning of Windows era (and actually since the beginning of Mac era), all Windows applications used to look pretty the same: square and plain blue.



But there is a way to customize this. Windows provide well documented messages (really? we´ll see...) that inform to the application about the events relating to the 'Non-client' area. The 'Non-client' area is, as his name indicates, the area that not conforms the 'client area' (that's where your application works by showing menus, controls and so on). Basically is composed by the caption bar, the frame's edges, the system and others buttons (maximize, minimize, close, etc.).

The good thing about the non-client area, it´s that you don´t need to care about it. All the drawing and behavior is provided by Windows itself. The bad thing is that in nowadays you need all the resources available s to build cool interfaces that attracts the user.

Take a look of applications like Microsoft Messenger, Windows Media Player or Google Chrome. All these applications provides interfaces that distinguish them from normal applications.

Well this feature is very feasible using OWL & OWLNext, let´s see.

The basic message that our application must respond is the **WM_NCPAINT.** As you suppose, it´s send when Windows informs you the need to paint the frame and edges of your window. Tipically you ignore this message, and it passes across the library until reachs the DefWindowProc. So our first mission is to catch this message. We define the **EvNCPaint(...)** function, ask for the full windows area by calling **GetWindowRect()**, create a **TWindowDC** object and call the appropiated GDI functions. That´s it. Or so.

After showing our Da-Vinci abilities we will notice that the system buttons (maximize box, minimize, etc.) are covered by our drawing, but when we pass over there they show again.

So a first solution working is to have a window without system buttons. By setting the appropriated windows attributes we avoid this shammy situation.

But why stop here? We can also draw our system buttons, a candy ones that express what be can do, and at the same time interact with Windows so it recognizes them but don´t redraw over.

The solution is to catch **WM_NCHITTEST.** By attending this message with the function **EvNCHitTest(...)**, we inform to Windows over what part of the Non-Client are the mouse it is. Each interesting part is called 'HitTest'. This allows to provide a custom quantity and location of system buttons, you receive the coordinates of the mouse and the only thing you need to do is to respond over what 'hitTest' is the mouse: HTMAXBUTTON, HTMINBUTTON, etc. Now when you place the mouse over any of the custom-drawed standard box, just a seconds later a Windows tooltip will show the name, ie: 'Maximize', 'Minimize', 'Close', etc.

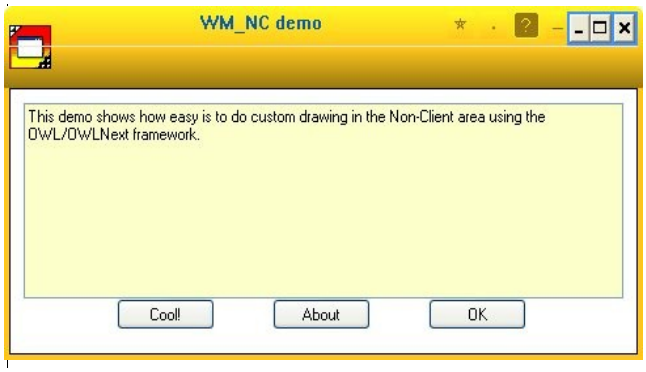
Windows is still drawing the classic buttons when we click with the mouse over a system button, even when they are placed at different positions (our cutoms vs the classic ones). So to close the circle we will respond to **WM_NCLBUTTUNDOWN** and **WM_NCLBUTTONUP.** The **EvNCLButtonDown(...)** function is designed to respond the first message. It not only receives the mouse position but also over what 'hittest' the mouse is on. The second one receives the same parameteres, and usually in this function we take the appropriated action for the button pressed.

So it's almost done, a final refinement it's to trace the **WM_NCMOUSEMOVE** so we can draw a button pressed or not, depending if we entered into their area or we leave it.

Additionally other refinement is to detect if we draw our frame as 'activated' or no. The **EvNCActivate** function respond to **WM_NCACTIVATE** and keeps internal state of the active/inactive value.

So, as nothing can stop us, now we go beyond and want to use a custom height for the caption bar. How to do it? Just use **EvNCCalcSize** and adjust the client area by setting a new top for the client area. To make easy the things we just update the client top position with the difference to the standard caption height.

We are ready. At least that's was what I was thinking. A rebel drawing was hard diyng. Using the excellent WinSight I analyzed the message saga and noticed a mysterious WM_0xAE. As Winsight was born in 1991, and last updated in 1996 it seemed the case of a new message. *But MSDN didn't admit its existence.*



Google's people faced a similar situation when creating Chrome, but they didn't found a way to replicate the problem. I just noticed that every time I've pressed the help button the rebel drawing appeared after clicking over the window. Killing the help button wasn't an acceptable solution.

More Google searching, and more evidence that we are facing a X-file case. The mysterious message is an undocumented Windows message. Those who discovered called it ***WM_NCUAHCAPTION***, and has a brother which is named ***WM_NCUAHDRAWFAME***. Apparently both were born with Windows XP and are related to the 'XP-themes' support. Actually I wasn't able to found exactly which parameters they carry, but if you don't attend them, they will get the DefWindowProc and Windows will proceed with a partial standard drawing that will sluggish your nice drawing.

Now we really are done. I've added a set of cool functions for convenience of the developer, and reserved something for enhancing later. The source code should be compatible with the current version of OWLNext and with OWL 5.0x, and the binary runs in every version of Win32 (from Win95 to Windows 7), it should also be compilable with Win16 but obviously I didn't go that far.

You can download the source code trough any of the affiliated OWLNext sites.

Grow and procreate■