Con l'utilizzo della nuova interfaccia di Windows 95 i nostri programmi possono finalmente rinnovare il proprio "look" e sfruttare nuove funzionalità

Taskbar e icone in Windows 95

di Alessandro Scotti

Tra le numerose innovazioni offerte da Windows 95 e Windows NT 3.51 con il nuovo shell, una delle più importanti è sicuramente la TaskBar, ovvero la finestra di sistema che contiene l'ormai celebre pulsante Start (o Avvio) e la lista delle finestre attive. All'interno della taskbar si trova anche una zona che contiene di solito l'ora corrente, e talvolta altre icone e informazioni. Questa zona, chiamata "tray notification area" (TNA), è disponibile anche alle applicazioni che necessitano di una icona sempre visibile e che possa essere raggiunta rapidamente: come vedremo, usarla ed aggiungere un tocco in più ai nostri programmi è tutt'altro che difficile.

Come funziona

Un esempio di utilizzo della TNA si può avere lanciando Resource Meter, QuickRes oppure stampando un documento: in tutti i casi appare una piccola icona vicino all'orologio di sistema. La differenza con le normali icone associate ad un programma sta nel fatto che una icona inserita nella TNA può reagire all'input dell'utente: per convenzione un click del pulsante destro provoca la comparsa di un menu, mentre un doppio click del pulsante sinistro esegue un'azione di default (indicata in neretto nel menu). Questo funzionamento non a caso ricorda da vicino quello del menu di sistema di un programma, ma se dal punto di vista dell'utente il meccanismo è pressoché identico, ciò che accade "dietro le quinte" è ben diverso. Nel caso della TNA infatti, i menu non sono gestiti automaticamente e non esiste un menu di default: tutte le azioni sono a carico dell'applicazione che controlla l'icona. Quest'ultima in pratica si limita a comunicare all'applicazione che al suo interno è stato premuto il tasto destro del mouse, ed il programma reagisce presentando all'utente un menu. Naturalmente nulla vieta al programma di effettuare il reboot immediato del sistema... ma di solito si preferisce attenersi a quella che è l'interfaccia standard di Windows 95.

Trattandosi di Windows, non c'è da stupirsi se l'icona comunica con l'applicazione inviando messaggi. L'icona però non è una finestra e non possiede quindi una funzione WndProc preposta alla gestione degli eventi, perciò questi devono essere inviati ad una "vera" finestra. Niente paura, praticamente tutti i programmi Windows dispongono di una finestra principale e se ce ne fosse bisogno è sempre possibile crearne una invisibile. Rimane tuttavia il problema di quali messaggi inviare: se l'icona spedisse un semplice WM_RBUTTONDOWN in seguito ad un click del pulsante destro, la funzione WndProc che lo riceve potrebbe scambiarlo non a torto per un evento generato nella propria client area. Inoltre la stessa WndProc potrebbe voler gestire più di una icona, perché porsi dei limiti? La soluzione adottata dai progettisti è tale che al programmatore viene lasciata la massima libertà: sarà infatti l'applicazione a decidere per ogni icona il valore del messaggio inviato ed un identificativo che, in caso si voglia usare lo stesso messaggio per diverse icone, permetta
di riconoscere l'icona che lo ha generato.

Le API

L'accesso alla "tray notification area" avviene tramite una sola funzione contenuta nella libreria di sistema SHELL32 (entrata numero 248) e dichiarata come segue:

```
BOOL APIENTRY Shell_NotifyIcon(
    DWORD dwMessage,
    PNOTIFYICONDATA lpData);
```

Il parametro dwMessage specifica quale azione deve essere effettuata dalla funzione, ovvero se una icona deve essere inserita (NIM_ADD), modificata (NIM_MODIFY) o rimossa dalla taskbar (NIM_DELETE). Tutti i dati necessari all'operazione da eseguire si trovano in una struttura di tipo NOTIFYICONDATA, vedi figura 1, a cui punta il secondo parametro lpData. La funzione restituisce FALSE se c'è stato un errore, oppure TRUE (un qualsiasi valore diverso da zero) in caso di successo.

```
typedef struct {
    DWORD  cbSize;           // Dimensione della struttura (byte)
    HWND   hWnd;             // Handle della finestra di gestione
    UINT   uID;              // Identificativo dell'icona
    UINT   uFlags;           // Flag (costanti NIF_xxx)
    UINT   uCallbackMessage; // Messaggio di notifica
    HICON  hIcon;            // Handle dell'icona
    CHAR   szTip[64];        // Testo del tooltip
} NOTIFYICONDATA, *PNOTIFYICONDATA;

// Messaggi da inviare tramite Shell_NotifyIcon per
// inserire, modificare o rimuovere una icona dalla TNA
#define NIM_ADD         0x00000000
#define NIM_MODIFY      0x00000001
#define NIM_DELETE      0x00000002

// Flag
#define NIF_MESSAGE     0x00000001
#define NIF_ICON        0x00000002
#define NIF_TIP         0x00000004
```

Figura 1 - Definizione della struttura NOTIFYICONDATA (versione ANSI) e delle costanti associate

Non è sempre necessario riempire tutti i campi: gli unici tre sempre richiesti sono cbSize, che contiene la dimensione della struttura, hWnd, che specifica l'handle della finestra che gestisce l'icona e riceve i messaggi inviati dalla taskbar, e uID, che identifica l'icona con un valore numerico. I restanti campi della struttura sono opzionali e controllati dal campo uFlags: per esempio se il bit NIF_ICON di uFlags è settato, il campo hIcon è valido e contiene l'handle di una icona 16x16 da inserire nella TNA. Analogamente se il bit NIF_TIP è settato il campo szTip contiene il testo del tooltip associato all'icona e gestito direttamente dalla taskbar. Questo sistema è utile soprattutto per le operazioni di modifica, dato che evita di dover specificare ogni volta anche i campi che rimarrano invariati.

Una volta inserita, l'icona appare subito a sinistra dell'orologio (se abilitato con l'apposita opzione nella configurazione della taskbar) o in fondo a destra, ed eventuali icone già presenti vengono spostate per far posto alla nuova. La posizione assoluta dell'icona cambia inoltre a seconda della posizione e delle dimensioni della taskbar, ed in genere non è nota all'applicazione. Del resto questa informazione è abbastanza inutile, benché probabilmente si possa escogitare qualche trucco per ottenerla.

Come si diceva prima, è l'applicazione che decide quale messaggio vuole ricevere dall'icona, riempiendo il campo uCallbackMessage della struttura e settando il bit NIF_MESSAGE di uFlags (è anche possibile, sebbene meno utile, non specificare alcun messaggio e creare così una icona "passiva"). La taskbar invia questo messaggio alla finestra specificata nel campo hWnd ogni volta che una azione del mouse interessa l'area nella quale si trova l'icona. L'identificativo di quest'ultima, cioè il campo uID della struttura NOTIFYICONDATA, viene copiato nel parametro wParam, mentre lParam contiene l'evento che ha generato il messaggio, per esempio WM_MOUSEMOVE o WM_RBUTTONDOWN.
Per esaurimento dei parametri disponibili le coordinate del mouse non vengono trasmesse, ma è comunque possibile ricavarle attraverso le API GetMessagePos o GetCursorPos. E' facile vedere come lo scopo dei progettisti della TNA sia pienamente raggiunto: dei quattro parametri di WndProc ben tre (hWnd, wMsg e wParam) sono stati decisi dall'applicazione, e questo livello di controllo si traduce spesso in codice efficiente e di facile manutenzione.

Data la necessità di passare a Shell_NotifyIcon un puntatore ad una struttura, è di solito abbastanza scomodo invocare questa funzione direttamente. Considerando anche il numero limitato di azioni che è possibile intraprendere, l'approccio migliore consiste probabilmente nel realizzare tre diverse funzioni per aggiungere, modificare e rimuovere icone dalla taskbar. Il codice necessario è molto semplice, ma i benefici in termini di chiarezza e semplicità d'uso sono notevoli. Volendo, è possibile racchiudere queste funzioni in una DLL o un una apposita libreria, ma in alcuni casi si può risparmiare il passaggio di qualche parametro inserendole nell'applicazione stessa ed utilizzando costanti o variabili globali. La scelta, che dipende dalla situazione o dai gusti personali, è lasciata al lettore. Nel programma di esempio, le funzioni tbarAddIcon, tbarDelIcon e tbarModifyIcon si trovano all'interno dell'applicazione ed il valore di uCallbackMessage è stabilito a priori.

Un altro utile accorgimento consiste nel separare la gestione dei messaggi che riguardano le icone in una funzione apposita. In questo modo non si appesantisce troppo la funzione WndProc che piuttosto di frequente nei programmi C consiste in una gigantesca istruzione switch di difficile lettura.

Perché usare la TNA

La tray notification area serve soprattutto a quei programmi che eseguono del lavoro "in background", o che comunque non necessitano di una finestra sempre attiva. In alcuni casi è infatti preferibile rinunciare ad una vera e propria finestra, perché quest'ultima anche se minimizzata andrà a finire nella taskbar, magari provocando una riduzione nella dimensione dei pulsanti associati alle finestre attive o altri effetti collaterali che è possibile evitare. Un valido esempio è rappresentato dal Print Manager, che in fase di stampa inserisce una piccola icona nella TNA e al termine la rimuove e si chiude automaticamente. L'utente ha ancora la possibilità di attivare una finestra di gestione con un click sull'icona, ma nella maggior parte dei casi ciò non è necessario: si ha dunque il vantaggio minimizzare l'impatto dell'applicazione sul desktop senza alcuna perdita di funzionalità.

In qualsiasi momento l'applicazione può modificare il contenuto dell'icona o il testo del tooltip (o addirittura il valore del messaggio ad essa associato). Queste caratteristiche devono essere tenute presenti e utilizzate il più possibile, soprattutto dalle applicazioni che non hanno una finestra principale. Come al solito conviene fare riferimento a qualche programma "originale" ed esaminarne il metodo di interfaccia verso l'utente. Il Resource Meter controlla costantemente la percentuale di risorse utilizzate dal sistema ed usa una icona per la presentazione dei dati. L'icona rappresenta un contenitore che si svuota man mano che le risorse disponibili diminuiscono e benché non fornisca informazioni dettagliate rappresenta una valida sintesi della situazione. Spostando il mouse sull'icona e tenendolo fermo per qualche decimo di secondo, compare un tooltip nel quale si trovano informazioni più precise, come "GDI: 94%" o "System: 92%". Infine con un doppio click sull'icona viene presentata una finestra nella quale i dati sulle risorse vengono visualizzati graficamente ed in modo completo. Questo triplice livello di presentazione dei dati è probabilmente uno degli usi tipici per i quali è stata realizzata la TNA, e sicuramente uno degli approcci da considerare quando si scrivono programmi analoghi.

Un ulteriore considerazione da fare è che in Windows 95 non è più concesso alle applicazioni di ridisegnare dinamicamente la propria icona. Il metodo usato in Windows 3.x, consistente in pratica nel registrare la classe principale senza icona e nello gestire poi i messaggi WM_PAINTICON e WM_ICONERASEBKGND, non funziona più. In effetti, Windows 95 non si prende nemmeno il disturbo di inviare i suddetti messaggi ai programmi marcati dal linker con versione 4.0 o superiore. Se necessario, rimane comunque possibile cambiare l'icona nella taskbar mediante la funzione SetClassWord (SetClassLong in Win32), ma questo metodo si usa abbastanza raramente, sebbene abbia il vantaggio di funzionare anche nelle "vecchie" versioni di Windows. In tutti i casi, purtroppo, la riduzione delle dimensioni dell'icona a 16x16 pixel non permette di realizzare disegni particolarmente elaborati, ma del resto la taskbar deve occupare il minor spazio possibile per non ridurre troppo l'area di lavoro del desktop.

Nell'occhio del... ciclope

Una realizzazione pratica di quanto visto finora si trova nel programma Cyclops, che segue i movimenti del mouse attraverso una icona inserita nella taskbar. Trattandosi di una applicazione a 32 bit scritta per sfruttare le nuove API disponibili, sarà bene soffermarsi su alcune peculiarità che rendono questo tipo di programmazione leggermente diversa da quella in ambiente Windows a 16 bit.

Il secondo parametro che Windows passa alla funzione WinMain di un programma, chiamato di solito hPrevInstance o hPrevInst, rappresenta l'handle dell'ultima istanza del programma, o NULL se al momento del lancio non ci sono altre istanze in esecuzione. La maggior parte dei programmi Windows, per evitare che vengano avviate più istanze della stessa applicazione, controlla appunto il valore di hPrevInstance ed esce subito se questo è diverso da NULL (o perlomeno tralascia l'inizializzazione: registrare di nuovo la classe per la finestra porterebbe ad un errore). In Win32 però il valore di hPrevInstance è sempre NULL, e un controllo del tipo precedente fa sì che l'applicazione continui tranquillamente la propria esecuzione. Questo comportamento è lecito, perché dal momento che un programma a 32 bit viene eseguito nel proprio address space vengono eliminati possibili problemi. Per fare un esempio, ogni istanza della stessa applicazione potrà chiamare con successo RegisterClass anche usando gli stessi parametri. Proprio a causa dei diversi address space, il problema delle istanze multiple deve essere affrontato in modo differente, ovvero utilizzando una risorsa che sia accessibile a tutti i processi. Nella maggior parte dei casi può essere sufficiente un mutex, ovvero un oggetto che protegge da accessi simultanei una risorsa condivisa (mutex sta per mutual exclusive [object]), ma per il mio programma ho scelto di utilizzare un semaforo, che ha il vantaggio di poter limitare il numero di istanze contemporaneamente in esecuzione ad un valore qualsiasi. La prima istanza crea il semaforo e gli assegna un nome, possibilmente non troppo banale, ed un valore iniziale. Opportunamente la funzione CreateSemaphore non fallisce se il semaforo esiste già, ma restituisce l'handle del semaforo esistente: in questo modo le successive istanze del programma accedono tutte allo stesso semaforo ed ogni accesso, effettuato con la funzione WaitForSingleObject, decrementa il valore associato al semaforo. Quando questo valore si azzera il semaforo diventa "rosso", WaitForSingleObject fallisce ed il programma termina immediatamente.

```
#define MAX_INSTANCES  2

int PASCAL WinMain( ... )
{
    HANDLE      hSem;
    DWORD       dwWait;

    hSem = CreateSemaphore( NULL,   // Crea il semaforo (o lo apre
        MAX_INSTANCES,              // se ne esiste già uno con lo
        MAX_INSTANCES,              // stesso nome)
        "MyAppControlSemaphore" );
    if( hSem == NULL )              // Operazione fallita,
        return( 0 );                // meglio terminare il programma
    dwWait = WaitForSingleObject( hSem, 0L );
    if( dwWait != WAIT_OBJECT_0 ) { // Non è possibile accedere al
        return( 0 );                // semaforo: fine programma
    ...
    // In uscita, aggiorniamo il numero di istanze
    // associato al semaforo
    ReleaseSemaphore( hSem, 1, NULL );
    return( ... );
}
```

Figura 2 - Esempio di utilizzo dei semafori per limitare il numero di istanze di un programma a 32 bit

Come d'obbligo per una applicazione Windows 95, è disponibile un menu che viene attivato con un click del pulsante destro sull'icona. In Win32 per mostrare a video un menu popup non è più sufficiente chiamare semplicemente TrackPopupMenu, perché se non ci si ricorda anche di portare in primo piano l'applicazione mediante la funzione SetForegroundWindow si va incontro ad effetti collaterali orribili a vedersi. La più comune SetActiveWindow non raggiunge lo stesso risultato, dato che per così dire ha un funzionamento locale al processo che la chiama, mentre il nostro scopo è quello di estendere l'effetto a tutto il desktop. Per vedere cosa accade quando manca la chiamata a SetForegroundWindow basta rimuovere la riga corrispondente dal programma, ricompilare e mandarlo in esecuzione: click destro sull'icona del programma, click sinistro sulla taskbar ed ecco che il menu scompare sotto la taskbar!
Una volta che il menu popup funziona correttamente, è buona regola stabilire una voce di default (evidenziata in neretto con la funzione SetMenuDefaultItem) la cui azione possa essere attivata con un doppio click sull'icona. Nella maggior parte delle applicazioni TNA-oriented l'effetto è solitamente quello di attivare una finestra vera e propria che consenta di interagire con il programma, ma naturalmente si possono avere diversi funzionamenti a seconda del tipo di applicazione.

Un'ultima nota va infine spesa per la funzione LoadImage usata per caricare le icone 16x16 da inserire nella taskbar. Windows95 fa largo uso di icone 16x16 (le cosiddette small icons), e queste ultime vengono spesso ottenute riducendo le originali icone 32x32. Chiamare la funzione LoadIcon per icone "piccole" provoca comunque l'applicazione dell'algoritmo di stretching e anti-aliasing standard, e anche se alla fine la dimensione dell'icona non è cambiata il contenuto può essere ben diverso dall'originale! A questo proposito è bene anche fare in modo che l'icona principale di un programma contenga al suo interno due versioni della stessa immagine, una di dimensioni 32x32 ed una 16x16. In questo modo l'Explorer non dovrà utilizzare il precedente algoritmo e l'aspetto dell'icona rimarrà sotto controllo in tutte le modalità di visualizzazione (inoltre la gestione di entrambi i tipi di icona è una delle numerose condizioni necessarie per ottenere il "Windows 95 logo" per un programma).

TNA e programmi a 16 bit

Tutte le funzioni descritte in precedenza sono a 32 bit, e quindi non facilmente accessibili da parte di programmi a 16 bit. Passeggiare con un debugger nelle librerie di sistema, però, porta ogni tanto alla scoperta di qualche interessante funzionalità: nel nostro caso l'esame della funzione Shell_NotifyIcon dimostra che è possibile utilizzare la TNA anche in programmi a 16 bit ed in modo sicuro.
In ambiente Win32, uno dei meccanismi messi a disposizione dal sistema operativo per lo scambio dei dati tra processi consiste nell'invio di un particolare messaggio, WM_COPYDATA. Seppur con qualche limitazione, questo messaggio permette ad una applicazione di inviare un blocco di dati ad un processo eseguito in un altro address space, senza dover ricorrere a meccanismi più potenti ma anche molto più complessi. Ora la funzione Shell_NotifyIcon deve in qualche modo parlare con la WndProc che gestisce la finestra della taskbar, e non è del tutto sorprendente che venga usato il messaggio in questione, anche perché il nome del primo parametro (dwMessage) non si presta a molte interpretazioni. In effetti, la precedente funzione non fa altro che riempire una struttura che differisce dalla NOTIFYICONDATA standard solo per l'aggiunta di due nuovi campi: nel primo viene posto un valore fisso (34753423h, utilizzato per convalidare la struttura), mentre nel secondo viene semplicemente salvato il messaggio, ovvero il primo argomento della funzione. Tutti gli altri valori vengono copiati dalla struttura originale e la nuova struttura viene infine spedita alla taskbar attraverso un messaggio WM_COPYDATA.
Rimane però un ultimo problema: qual'è l'handle della taskbar? Facile, basta chiamare la funzione FindWindow specificando il nome della classe della finestra di cui si vuole ricevere l'handle, ovvero "Shell_TrayWnd". A questo punto qualsiasi programma a 16 bit può accedere tranquillamente alla TNA, riempiendo correttamente la struttura estesa ed inviando un messaggio WM_COPYDATA alla taskbar. Il file TRAY.PAS mostra una possibile realizzazione in Turbo Pascal (o Delphi), ma la traduzione in C non dovrebbe presentare alcuna difficoltà.

```
typedef struct {
    DWORD dwTag;           // Campo di controllo (0x34753423)
    DWORD dwMessage;       // Operazione da effettuare (NIM_xxx)
    NOTIFYICONDATA stNID;  // Struttura NOTIFYICONDATA standard
} NOTIFYICONDATA_PRIVATE;
```

Figura 3 - La struttura non documentata che Shell_NotifyIcon invia alla taskbar con il messaggio WM_COPYDATA

Questo meccanismo, come del resto accade quando non c'è documentazione ufficiale, andrebbe usato con cautela perché potrebbe cambiare in futuro. Tuttavia esiste già un'applicazione Microsoft che ne fa uso (QuickRes, uno dei PowerToys disponibili gratuitamente sul sito FTP o WWW di Microsoft che probabilmente proprio a causa di questo motivo è stato aggiornato dopo la release ufficiale di Windows 95), perciò è ragionevolmente lecito attendersi che le cose rimangano come stanno ancora per qualche tempo. In ogni caso non dovrebbero esserci complicazioni di nessun tipo sulle versioni di Windows che non supportano la nuova interfaccia: se FindWindow non trova la finestra o se la finestra sbagliata riceve un messaggio WM_COPYDATA che non conosce, ci sono buone probabilità che non accada nulla di irreparabile, anzi che non accada proprio nulla!

Conclusioni

Dato il rapido diffondersi di Windows 95, prima o poi dovremo fare tutti i conti con la programmazione a 32 bit e con nuove API e controlli. Fortunatamente il passaggio non deve essere necessariamente traumatico: iniziando ad affrontare piccoli problemi, quali la gestione di una icona nella TNA, è possibile avvicinarsi a questo "nuovo" ambiente in modo graduale ed anche divertente. Nel nostro caso inoltre, poter lavorare a 16 bit consente di utilizzare strumenti di sviluppo già esistenti (Delphi, per esempio) e magari di effettuare il porting a 32 bit quando saranno disponibili strumenti analoghi più aggiornati. Microsoft comunque sta spingendo molto per fare in modo che la maggior parte delle nuove applicazioni sia scritta a 32 bit, in modo da funzionare anche su Windows NT, e se per il momento c'è ancora Windows 95 a fare da raccordo con il "vecchio" mondo a 16 bit, a mio avviso lo studio dell'architettura Win32 e della nuova interfaccia utente può essere un buon investimento fin da ora.

Nota: questo articolo è stato pubblicato sul numero 44 di *Computer Programming* (Febbraio 1996).

I sorgenti si possono scaricare dal sito www.ascotti.org (http://www.ascotti.org/programming/zip/cyclops.zip).