Multiplexing multiple tools into one in a tooltip

devblogs.microsoft.com/oldnewthing/20060628-05

June 28, 2006

Raymond Chen

The tooltip control lets you set multiple "tools" (regions of the owner window) for it to monitor. This is very convenient when the number of tools is manageably small and they don't move around much. For example, the toolbar control creates a tool for each button. But if you have hundreds or thousands of screen elements with tooltips, creating a tool for each one can be quite a lot of work, especially if the items move around a lot. For example, the listview control does not create a separate tool for each listview item, since a listview can have thousands of items, and scrolling the view results in the items moving around. Updating the tool information whenever the listview control scrolls would be extremely slow, and the work would be out of proportion to the benefit. (Updating thousands of tools on the off chance the user hovers over one of them doesn't really sit well on the cost/benefit scale.)

Instead of creating a tool for each item, you can instead multiplex all the tools into one, updating that one tool dynamically to be the one corresponding to the element the user is currently interacting with. We'll start with a fresh scratch program and create a few items which we want to give tooltips for.

```
int g_cItems = 10;
int g_cyItem = 20;
int g_cxItem = 200;
BOOL
GetItemRect(int iItem, RECT *prc)
{
 SetRect(prc, 0, g_cyItem * iItem,
         g_cxItem, g_cyItem * (iItem + 1));
 return iItem >= 0 && iItem < g_cItems;
}
int
ItemHitTest(int x, int y)
{
 if (x < 0 || x > g_cxItem) return -1;
 if (y < 0 || y > g_cItems * g_cyItem) return -1;
 return y / g_cyItem;
}
void
PaintContent(HWND hwnd, PAINTSTRUCT *pps)
{
 COLORREF clrSave = GetBkColor(pps->hdc);
 for (int iItem = 0; iItem < g_cItems; iItem++) {
  RECT rc;
  GetItemRect(iItem, &rc);
  COLORREF clr = RGB((iItem & 1) ? 0x7F : 0,
                     (iItem & 2) ? 0x7F : 0,
                     (iItem & 4) ? 0x7F : 0);
  if (iItem & 8) clr *= 2;
  SetBkColor(pps->hdc, clr);
  ExtTextOut(pps->hdc, rc.left, rc.top,
             ETO_OPAQUE, &rc, TEXT(""), 0, NULL);
 }
 SetBkColor(pps->hdc, clrSave);
}
```

We merely paint a few colored bands. To make things more interesting, you can add scroll bars. I leave you to deal with that yourself, since it would be distracting from the point here, although it would also make the sample a bit more realistic.

Next, we create a tooltip control and instead of creating a tool for each element, we create only one. For starters, it’s an empty tool with no rectangle. The `g_iItemTip` variable tells us which item this tooltip is standing in for at any particular moment; we use `-1` as a sentinel indicating that the tooltip is not active.

```
HWND g_hwndTT;
int g_iItemTip;
BOOL
OnCreate(HWND hwnd, LPCREATESTRUCT lpcs)
{
 g_hwndTT = CreateWindowEx(WS_EX_TRANSPARENT, TOOLTIPS_CLASS, NULL,
                           TTS_NOPREFIX,
                           0, 0, 0, 0,
                           hwnd, NULL, g_hinst, NULL);
 if (!g_hwndTT) return FALSE;
 g_iItemTip = -1;
 TOOLINFO ti = { sizeof(ti) };
 ti.uFlags = TTF_TRANSPARENT;
 ti.hwnd = hwnd;
 ti.uId = 0;
 ti.lpszText = TEXT("Placeholder tooltip");
 SetRectEmpty(&ti.rect);
 SendMessage(g_hwndTT, TTM_ADDTOOL, 0, (LPARAM)&ti);
 return TRUE;
}
```

You may have noticed that we do not use the `TTF_SUBCLASS` flag in our tool. We'll see why later.

The single tool for the tooltip covers our entire client rectangle. We maintain this property as the window resizes.

```
void
OnSize(HWND hwnd, UINT state, int cx, int cy)
{
 TOOLINFO ti = { sizeof(ti) };
 ti.hwnd = hwnd;
 ti.uId = 0;
 GetClientRect(hwnd, &ti.rect);
 SendMessage(g_hwndTT, TTM_NEWTOOLRECT, 0, (LPARAM)&ti);
}
```

We need to keep the `g_iItemTip` up to date so we know which item our tooltip is standing for at any particular moment. That is done by the `UpdateTooltip` function:

```
void
UpdateTooltip(int x, int y)
{
 int iItemOld = g_iItemTip;
 g_iItemTip = ItemHitTest(x, y);
 if (iItemOld != g_iItemTip) {
   SendMessage(g_hwndTT, TTM_POP, 0, 0);
 }
}
```

To update the tooltip, we check whether the mouse is over the same item as it was last time. If not, then we update our "Which item is under the mouse now?" variable and pop the old bubble (if any). And we always relay the message to the tooltip so it can do its tooltip thing. This function also explains why we did not use the `TTF_SUBCLASS` flag when we created our tool: We need to do some processing before the tooltip. If we had allowed the tooltip to subclass, then it would process the mouse message first, which means that our `TTM_POP` would have popped the new updated tooltip instead of the stale old tooltip.

This `UpdateTooltip` function is very important. It must be called any time the mouse may be hovering over a different item. This could be because the mouse moved or because the items under the mouse changed positions. I don't have any scrolling in this example, but if I did, then you would see a call to `UpdateTooltip` whenever we updated the scroll origin point because the act of scrolling may have moved the item that was under the mouse. (Failing to maintain mouse state after a scrolling operation is a common programming oversight.) Furthermore, if items were added or deleted dynamically, then a call to `UpdateTooltip` would have to be made once an item was added or deleted because the added or deleted item might be the one under the mouse.

The easy one to take care of is the mouse motion:

```
void
RelayEvent(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
 UpdateTooltip(GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam));
 MSG msg;
 msg.hwnd = hwnd;
 msg.message = uiMsg;
 msg.wParam = wParam;
 msg.lParam = lParam;
 SendMessage(g_hwndTT, TTM_RELAYEVENT, 0, (LPARAM)&msg);
}
LRESULT CALLBACK
WndProc(HWND hwnd, UINT uiMsg, WPARAM wParam, LPARAM lParam)
{
 if ((uiMsg >= WM_MOUSEFIRST && uiMsg <= WM_MOUSELAST) ||
     uiMsg == WM_NCMOUSEMOVE) {
  RelayEvent(hwnd, uiMsg, wParam, lParam);
 }
 switch (uiMsg) {
  ... as before ...
}
```

If we get a mouse message, then the `RelayEvent` message updates our tooltip state and then relays the message to the tooltip. See the discussion above for the importance of doing this in the right order.

You can run the program now. Observe that the program acts as if each colored band has its own tooltip, even though there is really only one tooltip that we keep recycling.

We’re still not done. The tooltip text is the same for each item, which is unrealistic for a real program. We’ll address this next time.

Raymond Chen

Follow