Using the MNS_DRAGDROP style: Menu rearrangement

devblogs.microsoft.com/oldnewthing/20111230-00

December 30, 2011

Raymond Chen

In order to do drag-drop rearrangement of menus, you need four things, most of which we already know how to do.

1. Dragging an item out of a menu. Check.
2. Dropping an item into a menu. Check.
3. Connecting the drag with the drop.
4. Rearranging menu items in response to the operation.

Let's do step 4 first, just to mix things up. And since this is just a demonstration rather than production code, I'm only going to support string menu items of up to 255 characters in length.

```
BOOL MoveMenuItem(HMENU hmenu, UINT uPosFrom, UINT uPosInsertAfter)
{
 BOOL fRc = FALSE;
 TCHAR sz[256];
 if (GetMenuString(hmenu, uPosFrom, sz, 256, MF_BYPOSITION) &&
     InsertMenu(hmenu, uPosInsertAfter, MF_BYPOSITION,
                GetMenuItemID(hmenu, uPosFrom), sz)) {
  if (uPosFrom > uPosInsertAfter) uPosFrom++;
  fRc = DeleteMenu(hmenu, uPosFrom, MF_BYPOSITION);
 }
 return fRc;
}
```

One thing you might not have noticed is that I inserted the copy before deleting the original. That way, we don't get stuck in the situation where we deleted the original, then the reinsertion fails, and now we've lost the menu item. (We can still get stuck if the deletion of the original fails, but the hope is that that is much more unlikely than the failure of an insertion.)

Okay, the next part is connecting the drag with the drop. To do that, I'll need some helper COM objects. But first, I'm going to introduce something that I should have introduced earlier: Objects that do nothing! (Just like our scratch program, they start out doing nothing, and then we'll modify them to do something.)

```
// dummy data object
class CEmptyDataObject : public IDataObject
{
public:
  // IUnknown
  STDMETHODIMP QueryInterface(REFIID riid, void **ppvObj)
  {
    IUnknown *punk = NULL;
    if (riid == IID_IUnknown) {
      punk = static_cast<IUnknown*>(this);
    } else if (riid == IID_IDataObject) {
      punk = static_cast<IDataObject*>(this);
    }
    *ppvObj = punk;
    if (punk) {
      punk->AddRef();
      return S_OK;
    } else {
      return E_NOINTERFACE;
    }
  }
  STDMETHODIMP_(ULONG) AddRef()
  {
    return ++m_cRef;
  }
  STDMETHODIMP_(ULONG) Release()
  {
    ULONG cRef = --m_cRef;
    if (cRef == 0) delete this;
    return cRef;
  }
  // IDataObject
  STDMETHODIMP GetData(FORMATETC *pfe, STGMEDIUM *pmed)
  {
    ZeroMemory(pmed, sizeof(*pmed));
    return DV_E_FORMATETC;
  }
  STDMETHODIMP GetDataHere(FORMATETC *pfe, STGMEDIUM *pmed)
  {
    return E_NOTIMPL;
  }
  STDMETHODIMP QueryGetData(FORMATETC *pfe)
  {
    return DV_E_FORMATETC;
  }
  STDMETHODIMP GetCanonicalFormatEtc(FORMATETC *pfeIn,
                                     FORMATETC *pfeOut)
  {
    *pfeOut = *pfeIn;
    pfeOut->ptd = NULL;
    return DATA_S_SAMEFORMATETC;
  }
```

```
  STDMETHODIMP SetData(FORMATETC *pfe, STGMEDIUM *pmed,
                       BOOL fRelease)
  {
    return E_NOTIMPL;
  }
  STDMETHODIMP EnumFormatEtc(DWORD dwDirection,
                             LPENUMFORMATETC *ppefe)
  {
  *ppefe = NULL;
  return E_NOTIMPL;
  }
  STDMETHODIMP DAdvise(FORMATETC *pfe, DWORD grfAdv,
                    IAdviseSink *pAdvSink, DWORD *pdwConnection)
  {
    *pdwConnection = 0;
    return OLE_E_ADVISENOTSUPPORTED;
  }
  STDMETHODIMP DUnadvise(DWORD dwConnection)
  {
    return OLE_E_ADVISENOTSUPPORTED;
  }
  STDMETHODIMP EnumDAdvise(LPENUMSTATDATA *ppefe)
  {
    *ppefe = NULL;
    return OLE_E_ADVISENOTSUPPORTED;
  }
  CEmptyDataObject() : m_cRef(1) { }
  virtual ~CEmptyDataObject() { }
private:
  ULONG m_cRef;
};
```

The `CEmptyDataObject` is simply a data object that contains no data. And here's an equally uninteresting `CEmptyDropTarget`:

```
class CEmptyDropTarget : public IDropTarget
{
public:
  // IUnknown
  STDMETHODIMP QueryInterface(REFIID riid, void **ppvObj)
  {
    IUnknown *punk = NULL;
    if (riid == IID_IUnknown) {
      punk = static_cast<IUnknown*>(this);
    } else if (riid == IID_IDropTarget) {
      punk = static_cast<IDropTarget*>(this);
    }
    *ppvObj = punk;
    if (punk) {
      punk->AddRef();
      return S_OK;
    } else {
      return E_NOINTERFACE;
    }
  }
  STDMETHODIMP_(ULONG) AddRef()
  {
    return ++m_cRef;
  }
  STDMETHODIMP_(ULONG) Release()
  {
    ULONG cRef = --m_cRef;
    if (cRef == 0) delete this;
    return cRef;
  }
  // IDropTarget
  STDMETHODIMP DragEnter(IDataObject *pdto, DWORD grfKeyState,
                         POINTL pt, DWORD *pdwEffect)
  {
    *pdwEffect = DROPEFFECT_NONE;
    return E_NOTIMPL;
  }
  STDMETHODIMP DragOver(DWORD grfKeyState, POINTL pt, DWORD *pdwEffect)
  {
    *pdwEffect = DROPEFFECT_NONE;
    return E_NOTIMPL;
  }
  STDMETHODIMP DragLeave()
  {
    return E_NOTIMPL;
  }
  STDMETHODIMP Drop(IDataObject *pdto, DWORD grfKeyState,
                    POINTL pt, DWORD *pdwEffect)
  {
    *pdwEffect = DROPEFFECT_NONE;
    return E_NOTIMPL;
  }
```

```
  CEmptyDropTarget() : m_cRef(1) { }
  virtual ~CEmptyDropTarget() { }
private:
  ULONG m_cRef;
};
```

Okay, now back to item 3: Connecting the drag with the drop. Your initial reaction might be to create a new clipboard format called, say, `DraggedMenuItem` which takes the form of a `TYMED_HGLOBAL` consisting of a struct like

```
struct MENUANDITEM
{
 HMENU hmenu;
 UINT uItem;
};
```

But once you do that, you already have a problem: What happens if this item is dragged out of a 32-bit process and dropped into a 64-bit process? The size of `HMENU` is different between the two processes, so the 32-bit and 64-bit `MENUANDITEM` structures are not compatible. This is an example of how you need to be aware of inter-process communications scenarios when developing persistence formats. In this case, we are passing a pointer-sized object between processes. Although most people think of a persistence format as a file format, here's a case where a persistence format takes the form of an in-memory storage format.

You might decide to solve this problem by tweaking the structure to accommodate 32-bit and 64-bit Windows:

```
struct MENUANDITEM
{
 __int64 i64Menu;
 UINT uItem;
 void SetMenu(HMENU hmenu) { i64Menu = (INT_PTR)hmenu; }
 HMENU GetMenu() const { return (HMENU)(INT_PTR)i64Menu; }
};
```

But there's an easier way out: Since we only want to support drag/drop menu editing from within the same menu (we don't care about dragging an item from one menu to another menu), the drag source and drop target reside in the same process, so all we need to do is verify the data object's identity, and if it's our data object, we can consult side data to determine what is being dragged.

Okay, so let's start with a fresh scratch program, and paste in the following:

- `CDropSource`
- `CEmptyDataObject`
- `CEmptyDropTarget`

- `MoveMenuItem`
- HANDLE_WM_MENUDRAG
- HANDLE_WM_MENUGETOBJECT

Okay, enough shopping. Now to teach our drop target how to recognize that the data object being dropped on it is our own:

```
class CMenuDataObject : public CEmptyDataObject
{
public:
  CMenuDataObject(HMENU hmenu, UINT uPos)
    : m_hmenu(hmenu), m_uPos(uPos) { }
public:
  const HMENU m_hmenu;
  const UINT m_uPos;
};
CMenuDataObject *g_pdtoDrag;
```

Our special data object when dragging a menu item merely carries around the menu and item so we can find it later. The magical bit is that we also keep track of the object being dragged. (Exercise: Since this is a demo program, the object is just a global variable. What is the correct way of keeping track of `g_pdtoDrag`?)

Now we get to teach our drop target to recognize `CMenuDataObject` and only `CMenuDataObject`:

```
class CMenuDropTarget : public CEmptyDropTarget
{
public:
  // IDropTarget
  STDMETHODIMP DragEnter(IDataObject *pdto, DWORD grfKeyState,
                         POINTL pt, DWORD *pdwEffect);
  STDMETHODIMP DragOver(DWORD grfKeyState, POINTL pt, DWORD *pdwEffect);
  STDMETHODIMP DragLeave();
  STDMETHODIMP Drop(IDataObject *pdto, DWORD grfKeyState,
                    POINTL pt, DWORD *pdwEffect);
  CMenuDropTarget(HMENU hmenu, UINT uPos)
    : m_hmenu(hmenu), m_uPos(uPos), m_uPosDrag(uPosNone) { }
  void Reset() { m_uPosDrag = uPosNone; }
private:
  static const UINT uPosNone = 0xFFFFFFFF;
private:
  HMENU m_hmenu;   // menu being dropped on
  UINT m_uPos;     // menu item being dropped on
  UINT m_uPosDrag; // menu item being dragged, if from the same menu
                   // else uPosNone
};
HRESULT CMenuDropTarget::DragEnter(
    IDataObject *pdto, DWORD grfKeyState, POINTL pt, DWORD *pdwEffect)
{
  Reset();
  IUnknown *punk;
  if (SUCCEEDED(pdto->QueryInterface(IID_PPV_ARGS(&punk)))) {
    punk->Release();
  }
  if (punk == g_pdtoDrag && g_pdtoDrag->m_hmenu == m_hmenu) {
    m_uPosDrag = g_pdtoDrag->m_uPos;
  } else {
    m_uPosDrag = uPosNone;
  }
  return DragOver(grfKeyState, pt, pdwEffect);
}
```

The job of `CMenuDropTarget::DragEnter` is to determine whether the item being dragged is a menu item from the same menu. We detect that the object being dragged is `g_pdtoDrag` by first querying for the canonical unknown, to remove any layers of wrapping COM may have placed around the object. We compare this against `g_pdtoDrag`, which is a bit of a cheat; more properly we should call `g_pdtoDrag->QueryInterface` to get the canonical unknown for `g_pdtoDrag`, but we can cheat because we know that `CMenuData-Object` is singly-derived from `IUnknown` and that it does not support aggregation (and therefore it is its own canonical unknown). (Exercise: Why is it okay to use `punk` after releasing it?)

Anyway, if the item is confirmed to be our item after all, then we copy the menu item position so we can move it on the drop.

```
HRESULT CMenuDropTarget::DragOver(
    DWORD grfKeyState, POINTL pt, DWORD *pdwEffect)
{
  if (m_uPosDrag == uPosNone) {
    *pdwEffect = DROPEFFECT_NONE;
  } else {
    *pdwEffect &= DROPEFFECT_MOVE;
  }
  return S_OK;
}
HRESULT CMenuDropTarget::DragLeave()
{
  Reset();
  return S_OK;
}
```

The `DragOver` and `DragLeave` methods are largely uninteresting. `DragOver` just gives appropriate feedback, and `DragLeave` forgets about the data object that is no longer being dragged over us. The real excitement is in the `Drop` method.

```
HRESULT CMenuDropTarget::Drop(
    IDataObject *pdto, DWORD grfKeyState, POINTL pt, DWORD *pdwEffect)
{
  DragEnter(pdto, grfKeyState, pt, pdwEffect);
  if (*pdwEffect & DROPEFFECT_MOVE) {
     MoveMenuItem(m_hmenu, m_uPosDrag, m_uPos);
  }
  return S_OK;
}
```

When the drop happens, we move the menu item. Kind of anticlimactic, isn't it.

Okay, at this point the `WM_MENUDRAG` and `WM_MENUGETOBJECT` handlers are old hat:

```
LRESULT OnMenuDrag(HWND hwnd, UINT uPos, HMENU hmenu)
{
 LRESULT lres = MND_CONTINUE;
 if (g_pdtoDrag == NULL && hmenu == GetSubMenu(GetMenu(hwnd), 0)) {
  g_pdtoDrag = new(std::nothrow) CMenuDataObject(hmenu, uPos);
  if (g_pdtoDrag) {
   IDropSource *pds = new(std::nothrow) CDropSource();
   if (pds) {
    DWORD dwEffect;
    DoDragDrop(g_pdtoDrag, pds, DROPEFFECT_MOVE, &dwEffect);
    pds->Release();
   }
   g_pdtoDrag->Release();
   g_pdtoDrag = NULL;
  }
 }
 return lres;
}
LRESULT OnMenuGetObject(HWND hwnd, MENUGETOBJECTINFO *pmgoi)
{
 HRESULT hr = E_NOTIMPL;
 if (pmgoi->hmenu == GetSubMenu(GetMenu(hwnd), 0) &&
     (pmgoi->dwFlags & (MNGOF_BOTTOMGAP | MNGOF_TOPGAP))) {
  IDropTarget *pdt = new(std::nothrow)
      CMenuDropTarget(pmgoi->hmenu, pmgoi->uPos);
  if (pdt) {
   hr = pdt->QueryInterface(*(IID*)pmgoi->riid, &pmgoi->pvObj);
   pdt->Release();
  }
 }
 return SUCCEEDED(hr) ? MNGO_NOERROR : MNGO_NOINTERFACE;
}
    HANDLE_MSG(hwnd, WM_MENUDRAG, OnMenuDrag);
    HANDLE_MSG(hwnd, WM_MENUGETOBJECT, OnMenuGetObject);
// and change CoInitialize and CoUninitialize
// to OleInitialize and OleUninitialize, respectively
```

There is a tricky part in `OnMenuGetObject`, namely that we only return a drop target if the mouse is *between* items, because it is only when you are between items that you are actually rearranging.

And there you have it, some menu drag/drop stuff. It was a lot of typing (mostly for those dummy objects), but not a lot of work.

Raymond Chen

Follow