Christophe PICHAUD
Lead Software Architect chez Infeeny
christophep@cpixxi.com
www.windowscpp.com

Windows : faire un menu Démarrer en C++

Dans cet article, nous allons réaliser une application Windows qui ressemble à un menu Démarrer. Il s'agit de lister les applications et de les afficher pour permettre à l'utilisateur de les lancer.

Pourquoi une telle idée me direz-vous ? Parce que selon moi, le menu Démarrer de Windows 10 est moche. Et puis connaissant les API de dessin GDI+, je me suis imaginé un menu Démarrer avec des rectangles multi-color.

Préambule sur le développement graphique Windows

Vous allez me dire OK, mais pour développer sous Windows on fait comment ? Ce n'est pas dur, on va utiliser un Framework graphique que Microsoft utilise : *Microsoft Foundation Classes* (MFC). Pour ceux qui pensent que les MFC sont ringards, je leur évoquerais juste DevExpress, SyncFusion, Telerik et leur dirait que ces Frameworks NET sont équivalents. On fait des graphiques, du Ribbon, des grids et des applications qui ont un look détonnant. Oui les MFC ont 25 ans car c'est le début du C++ chez Microsoft. La première fois que les API Windows étaient encapsulées dans des classes C++. A titre de comparaison, QT est né à peu près à la même époque et QT a toujours le vent en poupe…

Architecture logicielle

Nous allons réaliser une application SDI avec une fenêtre principale et une Dockable Pane encrée à gauche qui présente les applications sous forme d'arbre (TreeView). 1

Pour récupérer la liste des applications et des groupes, on va utiliser une astuce : itérer dans le répertoire « C:\ProgramData\Microsoft\Windows\Start Menu\Programs ». 2

Dans ce répertoire, on a tout ce qu'il faut. Pour chaque élément, on a :

- Un nom
- Un lien
- Une icône

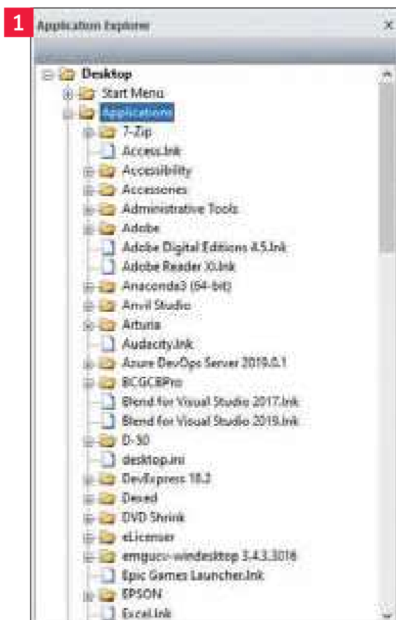
En ce qui concerne l'affichage des applications, on va utiliser différents panneaux (vues):

- Une vue pour le détail d'un lien
- Une vue pour un affichage à l'ancienne
- Une vue de type affichage « moderne » via GDI+ 3

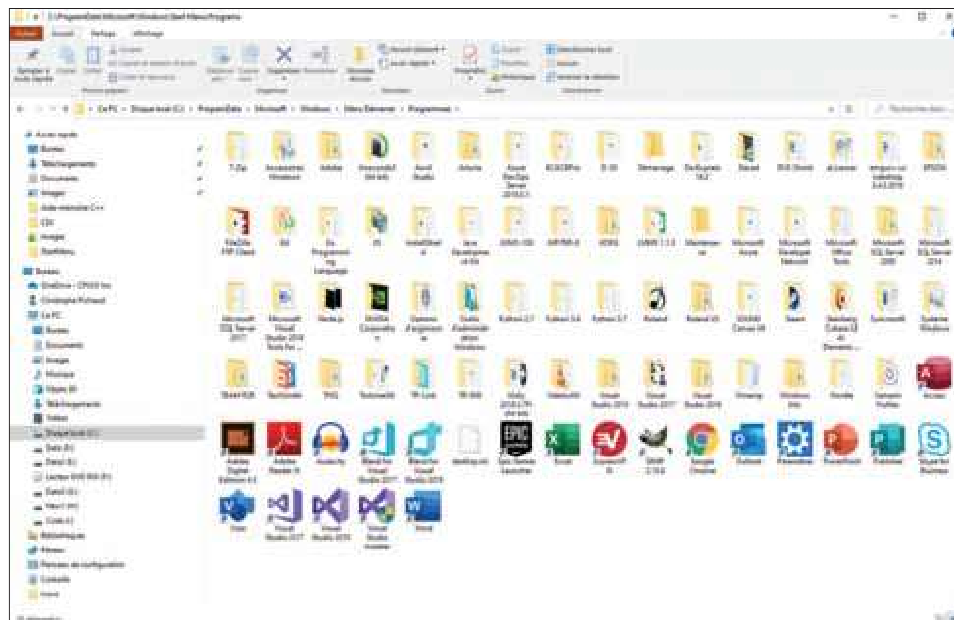
Communication Applicative

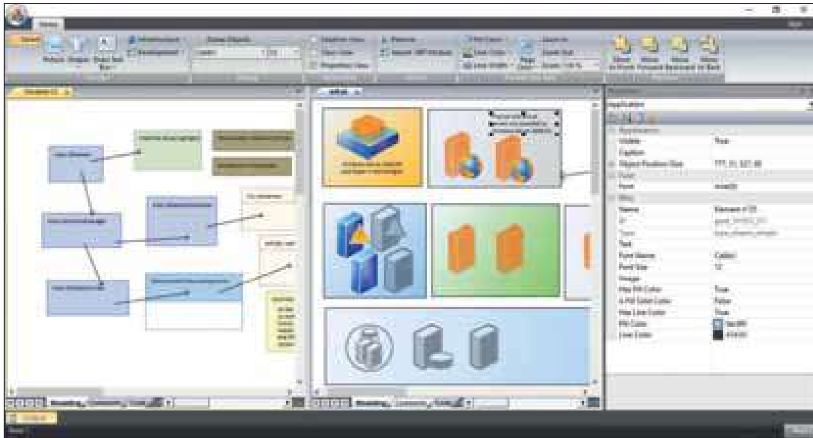
Pour arriver à un résultat final comme celui-ci, il y a des règles à respecter. D'abord, il y a une infrastructure de communication interne

3



1


2


4

à l'application. Je m'explique. C'est une application C++ ; il y a donc des classes et il faut instancier des objets pour que tout fonctionne. Comment ça marche ? L'application contient les classes suivantes :

Classe	Hérite de	Description
CMyDesktopApp	CWinAppEx	C'est la classe instanciée en globale
CMyDesktopDoc	CDocument	Classe de data à charger ou sauvegarder
CMyDesktopViewEx	CFormView	Vue affichée en premier
CStartMenuView	CFormView	Vue à l'ancienne
CStartMenuViewEx	CScrollView	Vue graphique Moderne
CMainFrame	CFrameWndEx	Menu, Ribbon & co
CApplicationViewBar	CDockablePane	Panneau de gauche (TreeView)

La classe Application et la cascade d'appel du Framework

Comment se lance l'application ? Dans MyDesktop.cpp, il y a une déclaration globale :

```
// The one and only CMyDesktopApp object
CMyDesktopApp theApp;
```

C'est comme ça que tout démarre. C'est le point d'entrée d'une application MFC. A partir de là, les actions suivantes sont enclenchées :

- CMyDesktopApp ::InitInstance
  - Création du triplet <CMainFrame, CMyDesktopDoc, CMydesktopViewEx> comme canvas d'affichage principal
  - Création du Ribbon
- CMainFrame ::OnCreate
  - Création du Ribbon
  - Création du DockablePane avec le TreeView
- CMyDesktopViewEx ::OnInitialUpdate

La symbiose des MFC consiste à créer une interaction entre un MainFrame (la partie visible de l'application, Ribbon, menu, toolbar, dockable panes), un document et une vue principale via cette séquence d'appel dans InitInstance de la classe dérivée de CWInAppEx:

```
// Register the application's document templates.  Document templates
//  serve as the connection between documents, frame windows and views
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
    IDR_MAINFRAME,
    RUNTIME_CLASS(CMyDesktopDoc),
    RUNTIME_CLASS(CMainFrame),      // main SDI frame window
    RUNTIME_CLASS(CMyDesktopViewEx));
  //RUNTIME_CLASS(CMyDesktopView));
if (!pDocTemplate)
    return FALSE;
AddDocTemplate(pDocTemplate);
```

C'est la base des MFC pour le modèle Document/Vue.

Abstraction des MFC

Pour avoir un code lisible, il faut dépasser le cadre des MFC. En effet, le but principal n'est pas de mixer son code au milieu des MFC. C'est trop facile mais ce n'est pas élégant. Ce qu'il faut, c'est y mettre une couche d'abstraction supplémentaire. Dans le cas de cette application, j'y ai introduit la notion de Manager. Un manager est un objet de haut niveau qui connaît tout le monde. Pourquoi une telle idée me direz-vous ? Pour isoler le code et pouvoir le réutiliser.

Réutilisation d'un canevas de dessin

Pour pouvoir afficher des éléments graphiques dans une vue (CScrollView), j'ai réutilisé l'infrastructure de mon projet UltraFluid Modeler… Voici à quoi ça ressemble : 4

Dans ce projet, un élément graphique :
- Possède des propriétés
- Est sélectionnable, déplaçable, resizable
- Peut passer devant, derrière
- On peut zoomer ou pas

Bref, c'est tout ce dont j'ai besoin, pourquoi le recoder si je peux réutiliser ???

La réalité est un peu plus douloureuse car on tire plusieurs classes mais ce n'est pas grave comparé aux fonctionnalités importées. Voici la liste des classes importées :

Classe	Hérite de	Description
CElement	CObject	C'est un objet graphique
CElementContainer	CObject	C'est une collection sérialisable d'objets
CElementManager	CObject	C'est le Handler global à réutiliser
CDrawingContext		C'est le contexte graphique GDI+

L'assemblage en LEGO

Comment on fait pour merger des classes C++ de plusieurs projets pour que cela marche ? C'est la question à 1 million de dollars ! Il faut se mettre à la place du compilateur. Je dois connaître tous les éléments (token) donc tout doit être déclaré : énumérations, structures, variables globales, types, classes dépendantes, etc.

Le Manager

La classe CElementManager est un handler globale pour une classe dérivée de CScrollView. Une fois que la classe passée en paramètre est bien une classe CStartMenuViewEx, tout est cablé ! Il faut juste créer des éléments graphiques de type CElement et les ajouter au membre m_objects du manager.

La routine principale de l'application est la suivante :

```
GetManager()->LoadStartMenu(this);
```

Au démarrage de l'application, la vue CMyDesktopViewEx va créer la liste des applications et remplir le TreeView du Doackable Pane de gauche via :

```
CString strPath = _T("C:\\ProgramData\\Microsoft\\Windows\\Start Menu\\Programs");
  CFileManager::SearchDrive(_T("*.*"), strPath, true, false,
pMainFrame->m_wndApplicationView._hAppz);
  pMainFrame->m_wndApplicationView.m_wndFileView.Expand(
pMainFrame->m_wndApplicationView._hAppz, TVE_EXPAND);
```

Voici le code de SearchDrive :

Code complet sur programmez.com & github

Ajout de l'élément graphique

Pour afficher un élément graphique nouveau (un élément de menu démarrer), il faut le définir :

Code complet sur programmez.com & github

Enrichissement de CElementManager

Pour afficher des éléments graphiques, il faut convertir le vector de ApplicationLink en objet CStartMenuElement. C'est le rôle de la méthode LoadStartMenu de la classe CElementManager :

Code complet sur programmez.com & github

L'appel à la méthode Invalidate() provoque l'appel à redessiner la vue, donc c'est un appel à OnDraw.

```
void CStartMenuViewEx::OnDraw(CDC* pDC)
{
  CMyDesktopDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
  if (!pDoc)
    return;

  GetManager()->DrawEx(this, pDC);
}
```

Cela se traduit par un appel indirect à Draw une fois que le contexte graphique Windows CDC est obtenu par CElementManager :: Draw(pView, pDrawDC); dans :

```
void CElementManager::Draw(CStartMenuViewEx* pView, CDC* pDC)
{
  // Initialize GDI+ graphics context
  Graphics graphics(pDC->m_hDC);
  // just like that
  //graphics.ScaleTransform(0.75f, 0.75f);
  graphics.ScaleTransform(m_fZoomFactor, m_fZoomFactor);

  // TODO: add draw code for native data here
  for (vector<std::shared_ptr<CElement>>::const_iterator i = GetObjects().begin();
i != GetObjects().end();
i++)
  {
    std::shared_ptr<CElement> pElement = *i;
    pElement->m_pView = pView;

    // Construct the graphic context for each element
    CDrawingContext ctxt(pElement);
    ctxt.m_pGraphics = &graphics;

    //pElement->Draw(pView, pDC);
    pElement->Draw(ctxt);

    if (pView != NULL && pView->m_bActive &&
!pDC->IsPrinting() && IsSelected(pElement))
      pElement->DrawTracker(ctxt, TrackerState::selected);
  }
}
```

Vous allez me dire, OK pour le dessin mais comment on lance les applications ? Il suffit de faire bouton droit, Start.. Il y a un appel à ShellExecute du lien : rien de compliqué.

Conclusion

La programmation graphique Windows permet de mettre en œuvre toute la puissance de la programmation orientée objet (OOP) et c'est ludique. GDI+ permet toutes les créations graphiques possibles et ce de manière confortable et simple à utiliser. Le code de l'application est disponible sur :
https://github.com/ChristophePichaud/MyDesktop