Grasp GUI on PC

Technical Report 96-11

Xiaofeng Cheng

Committee:

Dr. James Cross (Major Professor)
Dr. Kai Chang
Dr. Jennifer McManis

October 21, 1996
Abstract

The GRASP/Ada/C (Graphical Representation of Algorithms, Structures and Processes) has been successfully implemented in the X-Window environment and is getting more and more attention from software engineers around the world. It has become necessary to migrate it to PC’s Windows environment. With the emerging popularity of MFC (Microsoft Foundational Classes) classes and their strong OO nature, it is natural to implement Grasp PC version with Microsoft Visual C++.

Another reason Visual C++ was chosen over the traditional standard Windows API (Application Programmer’s Interface) is the extensive use of message-to-method mapping, which eliminates the "switch" statement and lets developers spend less time communicating with Windows and more time developing the application’s code.

The GUI (Graphical User Interface) of GRASP is the first step to migrate to Windows environment. After this step, the GRASP team will concentrate on the event handlers and the migration of the underlying C code.

The Classes

There are seven related classes to support the GUI functionality:

1. CGraspApp: The window application class
2. CGraspDoc: The class that handles the applications data
3. CMainFrame: The main window
4. CChildFrame: The code/CSD (Control Structure Diagram) frame window
5. CGraspView: The code/CSD view
6. CCpgWnd: The CPG (Complexity Profile Graph) frame window
7. CCpgView: The CPG view

The interrelationship of the class objects:
Figure 1: The relationship between the class objects

- CGraspApp class has one instance in this project, whose behavior determines the application
- CGraspApp object has one CMainFrame object and 2 CMultiDocTemplate objects, one for CSD, the other for CPG.
- CMultiDocTemplate object knows its document, frame and view class, so that they can be created dynamically.
- The view classes (both CGraspView and CCpgView) know their document, and vise versa. This property enable them to communicate at run time.

Initial code skeleton:

With the help of Microsoft Visual C++ 4.0, it is very convenient to get the initial code:

Figure 2.1 -- Choose new at the file menu, the dialog window will show up. Choose Project Workspace
Figure 2.2 -- choose AppWizard, and type project name Grasp

Figure 2.3 -- First step using AppWizard: Document type and Language
Figure 2.4 -- Database support

Figure 2.5 -- OLE support
Figure 2.6 -- Customize the application features

Figure 2.7 -- Other options
Figure 2.8 -- Customize the classes

Figure 2.9 -- The summery of the project features, classes and files

Upon finish, AppWizard will generate the Following classes:
Class CGraspApp in files:
- Grasp.h: This is the main header file for the application. It includes other project specific headers (including Resource.h) and declares the CGraspApp application class.
- Grasp.cpp: This is the main application source file that contains the application class CGraspApp. It also declares the one and only CWinApp class object, whose behavior determines the Grasp application.

Class CMainFrame in files MainFrm.h and MainFrm.cpp:
- These files contain the frame class CMainFrame, which is derived from CMDIFrameWnd and controls all MDI frame features.

Class CGraspDoc in files GraspDoc.h and GraspDoc.cpp
- These files contain the CGraspDoc class. It is necessary to edit these files to add special document data and to implement file saving and loading (via CGraspDoc::Serialize).

Class CGraspView in files GraspView.h and GraspView.cpp
- It is the code/CSD view of the document, a subclass of CViewClass

Class CChildFrame in files ChildFrm.h and ChildFrm.cpp
- a subclass of CMDIChildWnd, will be used to contain the code/CSD view

AppWizard also generate other files:

- StdAfx.h, StdAfx.cpp -- These files are used to build a precompiled header (PCH) file named Grasp.pch and a precompiled types file named StdAfx.obj.
- Resource.h -- This is the standard header file, which defines new resource IDs. Microsoft Developer Studio reads and updates this file according to the resource editor results.
- Grasp.rc -- This is a listing of all of the Microsoft Windows resources that the program uses. It includes the icons, bitmaps, and cursors that are stored in the RES subdirectory. This file can be directly edited in Microsoft Developer Studio.
- Grasp.reg -- This is an example .REG file that shows you the kind of registration settings the framework will set for you. You can use this as a .REG file to go along with your application or just delete it and rely on the default RegisterShellFileTypes registration.
- Grasp.clw -- This file contains information used by ClassWizard to edit existing classes or add new classes. ClassWizard also uses this file to store information needed to create and edit message maps and dialog data maps and to create prototype member functions.
- res\GraspDoc.ico -- This is an icon file, which is used as the icon for MDI child windows for the CGraspDoc class. This icon is included by the main resource file Grasp.rc.
- res\Grasp.ico -- This is an icon file, which is used as the application’s icon. This icon is included by the main resource file Grasp.rc.
- res\Grasp.rc2 -- This file contains resources that are not edited by Microsoft Developer Studio. You should place all resources not editable by the resource editor in this file.

Resources:
Figure 3.0 -- The resource view let you view and edit the resources

Figure 3.1 -- IDR_GRASPTYPE

Figure 3.2 -- IDR_MAINFRAME
First the menu resource for IDR_GRASPTYPE (R stands for Resource) was edited to look like the GRASP Main Menu. Then the resource for IDR_MAINFRAME was edited according to the GRASP startup menu. And another menu resource IDR_CPGTYPE for the CPG menu was added. Finally a resource IDR_MENU1 was added to represent the GRASP startup menu, and it will be used as the resource of floating point menu.

Since the CPG and CSD windows have control bars, they needed to be added as dialog resources: IDD_CPGWND and IDD_GRASP(D stands for dialog). The window style need to be Child because they exist only inside their parent window. The static controls and button controls and sliders were added to the window, and the proper styles and appropriate resource ID’s were assigned.

Create Objects and hook them up

- Create the new classes:
1. Create the CCpgWnd class: with the help of Class Wizard, the code was generated for class CCpgWnd, which is a child class of CMDIChildWnd.

2. Create the CCpgView class with the help of Class Wizard, it was implemented as a subclass of CView.

**Hook up the CPG window:**

1. In the CGraspApp class, a member variable m_pTempCPG of type (CMultiDocTemplate *) was added. Here "m" stands for member, "p" stands for pointer. Using this name conversion, it is not so easy to be trapped into type errors. CMultiDocTemplate class is a MFC class used to connect document, its views and frame windows.

2. Inside the method CGraspApp::InitialInstance, the following initialization statement were added:

   ```
   m_pTempCPG = new CMultiDocTemplate(
      IDR_CPGTYPE,
      RUNTIME_CLASS(CGraspDoc),
      RUNTIME_CLASS(CCpgWnd), // standard MDI childframe
      RUNTIME_CLASS(CCpgView)); // create main MDI Frame window
   ```

Therefore m_pTempCPG hooks classes CGraspDoc, CCpgWnd, CCpgView together with the menu resource IDR_CPGTYPE. When an object of CCpgWnd class is constructed in the future, it knows its document class and view class and menu resources.

3. Since the m_pTempCPG was "new"ed, it has to be "delete"ed. The method ExitInstance was modified to take care of this.

   ```
   int CGraspApp::ExitInstance()
   {
      // TODO: Add your specialized code here and/or call the base class
      delete m_pTempCPG;
      return CWinApp::ExitInstance();
   }
   ```

4. Use Class Wizard, a event handler CMainFrame::OnCpgGenerate() was added to handle the menu item event ID_CPG_GENERATECPG. Here is the code:

   ```
   void CMainFrame::OnCpgGenerate()
   {
      // TODO: Add your command handler code here
      CMDIChildWnd* pActiveChild=MDIGetActive(); //Get the code window
      CDocument* pDocument; // the document
      if (pActiveChild == NULL ||
          (pDocument=pActiveChild->GetActiveDocument())==NULL)
      {
         TRACE0("Warning: No active document for Generate_CPG command\n");
         AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
         return;
      }
      // Get the CPG template
      CDocTemplate* pTemplate=((CGraspApp*) AfxGetApp())->m_pTempCPG;
      ASSERT_VALID(pTemplate);  
      // using the template to create the frame window, so that
      // the new frame window is connected to the document.
      CFrameWnd *pFrame=pTemplate->CreateNewFrame(pDocument,pActiveChild)
      if (pFrame==NULL)
      {
      }
   ```
TRACE0("Warning: failed to create new frame\n");
AfxMessageBox(AFX_IDP_COMMAND_FAILURE);
return;
}

// make the frame visible
pTemplate->InitialUpdateFrame(pFrame,pDocument);

● Implement the visual appearance:
  1. Since the toolbar feature is not needed, the toolbar generating code block in
     CMainFrame::OnCreate() was commented out

  2. MFC MDI document has a style that only the active window’s menu will show up. A
     floating menu is necessary which is identical to the system menu. Clicking the right mouse
     button will pop up the menu. The menu resource IDR_MENU1 was used for this purpose,
     and here is the code:

     void CMainFrame::OnContextMenu(CWnd* pWnd, CPoint point)
     {
       // TODO: Add your message handler code here
       CMenu menu;
       menu.LoadMenu(IDR_MENU1);
       menu.GetSubMenu(0)->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTBUTTON,
       point.x, point.y, this);
       // menu.TrackPopupMenu(TPM_RIGHTBUTTON, point.x, point.y, this);
     }

  3. The CSD/code control bar should be created inside CChildFrame objects. A member
     variable m_wndDialogBar of class CDialogBar was added, and the member function
     OnCreate was modified:

     int CChildFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
     {
       if (CMDIClass::OnCreate(lpCreateStruct) == -1)
       return -1;

       // TODO: Add your specialized creation code here
       if (!m_wndDialogBar.Create(this, IDD_GRASP, CBRS_TOP|CBRS_BOTTOM,
       9000))
       {
         TRACE("Failed to create toolbar\n");
         return -1;       // fail to create
       }
       return 0;
     }

  4. The same was done to CCpgWnd class:

     int CCpgWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
     {
       if (CMDIClass::OnCreate(lpCreateStruct) == -1)
       return -1;

       // TODO: Add your specialized creation code here
if (!m_wndDialogBar.Create(this, IDD_CPGWND, CBRS_TOP|CBRS_BOTTOM, 
9000))
{
    TRACE("Failed to create DialogBar\n");
    return -1;    // fail to create
}
return 0;

• Make the CGraspView an editable view class
  1. A member variable m_rich of CRichEditControl need to be added, and the method OnCreate need to be modifed to take care of the creation of m_rich:

int CGraspView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    CRect rect(0,0,0,0);
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // TODO: Add your specialized creation code here
    m_rich.Create(ES_AUTOVSCROLL | ES_MULTILINE | ES_WANTRETURN | 
        WS_CHILD | WS_VSCROLL, rect, this, 1);

    return 0;
}

  2. Member variable m_rich need to be resizable with the view, the code is:

void CGraspView::OnSize(UINT nType, int cx, int cy)
{
    CRect rect;
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
    GetClientRect(rect);
    m_rich.SetWindowPos(&wndTop,0,0,rect.right-rect.left,
        rect.bottom-rect.top, SWP_SHOWWINDOW);
}

• Read and write Ascii file from and to permanent storage
  1. A member variable m_bText of class CString was added and was made protected because it is not desirable for other classes to manipulate it directly.
  2. A reader function GetText() is added to retrieve the reference of m_bText.
  3. To take care of the file i/o, MFC provided an CArchive object, which will handle binary data. Since our file is in ascii format, the storing and loading code has to be rewritten to handle one character a time:

void CGraspDoc::Serialize(CArchive& ar)
{
    BYTE buf;

    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
}
int iLength = m_bText.GetLength();
for (int i=0; i < iLength; i++) {
    buf=m_bText[i];
    ar << buf;
}
else
{
    // TODO: add loading code here
    while (1)
    {
        try {
            ar >> buf;
        }
        catch (CArchiveException *e)
        {
            if (e->m_cause != CArchiveException::endOfFile)
            {
                TRACE0("Unknown exception loading file!\n");
                throw;
            } else
            {
                TRACE0("End of file reached...\n");
                e->Delete();
                break;
            }
        }
        m_bText += buf;
    }
}

Variable buf was declared as type BYTE instead of char because CArchive class doesn’t have an operation "<<" or ">>" on char, but it does have operation "<<" and ">>" on BYTE. Since BYTE and char have the same size of storage, buf can to be converted to BYTE before being Serialized.

Communication between document and code/CSE view:
1. 2 help member functions were written to help either update from document to view or from view to document:

   ```c
   void CGraspView::UpdateToDoc()
   {
       CGraspDoc* pDoc = GetDocument();
       ASSERT_VALID(pDoc);

       // TODO: add draw code for native data here
       m_rich.GetWindowText(pDoc->GetText());
       m_rich.SetModify(FALSE);
       pDoc->SetModifiedFlag(TRUE);
   }
   
   void CGraspView::UpdateFromDoc()
   {
       CGraspDoc* pDoc = GetDocument();
   ```
ASSERT_VALID(pDoc);

// TODO: add draw code for native data here
m_rich.SetWindowText(pDoc->GetText());

2. The methods OnInitialUpdate and OnUpdate() are modified like this

void CGraspView::OnInitialUpdate()
{
    UpdateFromDoc();
    CView::OnInitialUpdate();

    // TODO: Add your specialized code here and/or call the base class
}

void CGraspView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    // TODO: Add your specialized code here and/or call the base class
    UpdateFromDoc();
    CView::OnUpdate(pSender, lHint, pHint);
}

3. When do you update TO the document? Here it is: (When the window is closed)

void CChildFrame::OnClose()
{
    // TODO: Add your message handler code here and/or call default
    CView* view=GetActiveView();
    if (view != NULL)
        ((CGraspView *)view)->UpdateToDoc();
    CMDIChildWnd::OnClose();
}

4. The actually saving happens when you send a "Save" command. The virtual function
OnSaveDocument was written to update data from view to document:

BOOL CGraspDoc::OnSaveDocument(LPCTSTR lpszPathName)
{
    // TODO: Add your specialized code here and/or call the base class
    POSITION pos=GetFirstViewPosition();
    CView *view;
    CRuntimeClass*prt;

    // You can do a run time class check here;
    while (pos != NULL) {
        view=GetNextView(pos);
        prt = view->GetRuntimeClass();
        if ( strcmp( prt->m_lpszClassName, "CGraspView" ) == 0 )
            ((CGraspView *)view)->UpdateToDoc();
    }

    return CDocument::OnSaveDocument(lpszPathName);
}
Some other issues:

1. To handle the CSD/code window’s menu command "Exit this Window", a event handler OnExitwin() was added:

   ```cpp
tvoid CChildFrame::OnExitwin()
   { // TODO: Add your command handler code here
      OnClose();
   }
```

2. Since the GRASP might have multiple windows but only one menu, I added a window menu in both view to take care of window arranging and choosing functions. i.e. "Tile", "Arrange Icons", and a list of windows.

3. To keep track of most recent used files, a menu item was added with menu ID "ID_FILE_MRU_1" under the "File" menu. This will take care of the tracking of most recent used files. The number of tracking was set to 4 when the AppWizard was run.

---

The results

The testing results are as follows. The application gives the expected appearance and does file loading and saving properly.

![Initial appearance when Grasp application starts](image)

Figure 4.1: Initial appearance when Grasp application starts
Figure 4.2: Close the CSD window and the menu looks like this

Figure 4.3: Choose New CSD... from the File menu and you can edit the CSD window

```c
main() {
    int i=5;
    int j=6;

    printf("please enter a number...");
```
Figure 4.4, 4.5: You can generate the CPG window, and choose the window arrangement
Figure 4.6: Windows are arranged in tiles

```c
main() {
    int i=5;
    int j=6;

    printf("please enter a number...");
}
```

Figure 4.7: When CSD window is active, the menu is different

```c
main() {
    int i=5;
    int j=6;

    printf("please enter a number...");
    cout << endl;
}
Figure 4.8: The file menu looks like this, notice the most recent used files.

Figure 4.9: Exit the CSD window and only the CPG window is left. The document is not closed yet.
Figure 4.10: Maximize the CPG window. When you close the last window (CPG window), a dialog shows up because the document has not been saved yet.

Figure 4.11: You can choose the file name in the Save As dialog box.
Figure 4.12: Because Grasp2 exists, confirmation is needed

Figure 4.13: This is what looks like after closing the last window, same as Figure 4.2

Opening files takes the same procedure.

Appendices:

1) Tools used:
   - GRASP/Ada/C X-Window version: This is the starting point from which the GUI on PC was
designed.

- AppWizard: A tool Microsoft Visual C++ provided to quickly generate initial code. Because every window program has the almost same code structure, AppWizard happily generated this code for you.

- MFC library: They form a Dynamically Linked Library (you can use them statically, of course) to ease the difficulties of Windows programming and to enforce the Object Oriented nature of your program.

- Class Wizard: It is also a code generator plus more. It can provide code skeleton for any class member functions according to user specification. It can also write message map (event hanler prototype) for the users. Because AppWizard code hides many internal details from the user, Class Wizard knows how to initialize data and when it is necessary to call the base class function.

- Resource Editor: It is a powerful WYSIWYG(What You See Is What You Get) resource editor and can be used to edit icon, bitmap, menu, string table, dialog resources, and all output will be stored in resource.h (for ID’s) and Grasp.rc.

2) The Code:

- They can be accessed under directory Grasp

3) References:


- Paul Perry, Chris Corry, ..., "Using Visual C++ 2", Que Corp., 1994