Chapter 10

Data Visualization

CAD/CAM have revolutionized much of the engineering design and manufacturing processes in the world and eliminated some traditional disciplines. For example, engineering drawing was one of the core curriculums for many engineering majors prior to the birth of computer-aided design and drafting systems. In today's world, many engineering students in universities do not learn manual drafting techniques; instead they learn how to use CAD systems to design components and generate engineering drawings from computerized models. The days of using protractors and compasses to create drawings are gone for good.

If asked to describe what a CAD system is in a few words, I would say it is essentially a collection of sophisticated tools that are exposed to users via graphical user interface (GUI) for creating, visualizing, and manipulating data. Visualization is essential to CAD systems because it allows designers and engineers to collaborate and to accurately see what the product is really going to look like throughout the entire design process. Without the data visualization capability, a CAD system is not that much different from a pocket calculator.

Data visualization is mainly studied in computer graphics. It is beyond the scope of this book to discuss graphics programming in detail. Instead, we will focus on how to develop a simple Windows-based application that calls our least squares approximation APIs and displays the end result on the screen.

10.1 Overview of windows application

In the early days, programmers depended on Win32 API to develop Windows-based 32-bit applications. Although Win32 API gives developers maximum control, it is difficult to master. A basic action such as opening a window or adding some text, a few buttons, or other controls can require lengthy coding. It can take months or even years to learn how to master event handlers and all other techniques necessary to build a full-featured application. In the early 1990s, Microsoft released the *Microsoft Foundation Class (MFC) Library* in C++ that wraps big portions of the Windows Win32 API in C++ classes. The MFC library brings Windows programming down

to the average programmer. It uses an object-oriented model that eliminates much of the tedium and exacting detail of the Win32 API, yet it still offers most of the power needed to create full-featured Windows programs. MFC uses a model-view-controller pattern to separate programs into more manageable pieces. The Visual C++ resource editors, MFC AppWizard and ClassWizard significantly reduce the time needed to write code that is specific to a particular application. For example, the resource editor creates a header file that contains assigned values for `#define` constants. AppWizard generates skeleton code for an entire application, and ClassWizard generates prototypes and function bodies for message handlers.

In 1995, Sun Microsystems, a company best known for its high-end UNIX workstations, released the Java programming language. Sun describes Java as a "*simple, object-oriented, distributed, interpreted, and dynamic language.*" It is true that Java is simple to learn, as it removed or streamlined the area where programmers have had difficulty or that have been the largest source of bugs. For example, we will not find pointers in Java, nor will we find overloaded operators. To an experienced C/C++ programmer, these omissions may be difficult to get used to, but to beginners or programmers who have worked in other languages, they make the Java language far easier to learn. Furthermore, memory management in Java is automatic, which avoids memory leaking that is common to all levels of programmers. Java applications are typically compiled to *bytecode* (`.class` file) that can run on any *Java virtual machine (JVM)* regardless of computer architecture. For these reasons, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 10 million users.

Microsoft saw the challenge from Java and started the development of the .NET Framework in the late 1990s, originally under the name of *Next Generation Windows Services*. By late 2000, the first beta versions of .NET 1.0 were released. It includes a large library and provides language interoperability (each language can use code written in other languages) across several programming languages. Programs written for the .NET Framework execute in a software environment (as contrasted to a hardware environment), known as the *Common Language Runtime (CLR)*, an application virtual machine that provides services such as security, memory management, and exception handling. The class library and the CLR together constitute the .NET Framework. With .Net Framework, applications are usually written in *managed code* that has its execution managed by the .NET Framework Common Language Runtime. Managed code may be implemented in $C^{\#}$, J#, Microsoft Visual Basic .NET, or managed C++. All of these languages share a unified set of class libraries and can be encoded into an *Intermediate Language (IL)*. A runtime-aware compiler compiles the IL into native executable code within a managed execution environment that ensures type safety, array bound and index checking, exception handling, and garbage collection. It is said that, by using managed code and compiling in this managed execution environment, developers can avoid many typical programming mistakes that lead to security holes and unstable applications. Also, many unproductive programming tasks are automatically taken care of, such as type safety checking, memory management, and destruction of unneeded objects.

The MFC library is implemented with unmanaged C++ and, hence, is not directly compatible with .Net Framework. *Windows Forms* (or *WinForms* for short) is the name given to the graphical application programming interface included as a part of Microsoft .NET Framework, providing access to native Microsoft Windows interface elements by wrapping the extant Windows API in managed code. Windows Forms is for creating Microsoft Windows applications on the .NET Framework. This framework provides a modern, object-oriented, extensible set of classes. With Windows Forms, developers are able to create a rich client application that can access a wide variety of data sources and provide data-display and data-editing facilities using Windows Forms controls.

Windows Presentation Foundation (WPF) is a graphics platform included in Microsoft .NET Framework 3.0 and later versions. It is a next-generation presentation system for building Windows client applications with visually stunning user experiences. We can create a wide range of both stand-alone and browser-hosted applications that incorporate documents, media, 2D and 3D graphics, and animations. In essence, WPF has the following advantages:

- WPF offers additional programming enhancements for Windows client application development. One obvious enhancement is the ability to develop an application using both *markup* and *code-behind*. The *Extensible Application Markup Language (XAML)* (markup) is usually used to implement the appearance of an application while managed programming languages (code-behind) are used to implement its behavior. This separation of appearance and behavior makes the development of windows-based applications more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior.

- XAML opens up a world of possibilities for collaboration, because many graphics design tools understand the XAML format.

- WPF changes how we do graphics programming. It integrates three basic Windows elements – text, controls, and graphics – into a single programming model and puts these three elements in the same element tree in the same manner. Without WPF, developing a graphics application would involve a number of different technologies, ranging from GDI/GDI+ for 2D graphics to Direct3D or OpenGL for 3D graphics. WPF, on the other hand, is designed as a single model for graphics application development, providing seamless integration between such services within an application. Similar constructs can be used for creating animations, data binding, and 3D models.

- When creating a UI, developers arrange their controls by location and size to form a layout. A key requirement of any layout is to adapt to changes in window size and display settings. Rather than forcing developers to write the code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for application developers. The cornerstone of the layout system is relative positioning, which increases the ability to adapt to changing window and display conditions. WPF manages a layout using different containers. Each container has its own layout logic – some stack elements together, others arrange them in a grid of invisible cells, and so on. WPF includes several layout controls:

Name	Description
StackPanel	Places elements in a horizontal or vertical stack. This layout container is typically used for small sections of a larger, more complex window.
WrapPanel	Places elements in a series of wrapped lines. In horizontal orientation, the WrapPanel lays items out in a row from left to right and then onto subsequent lines. In vertical orientation, the WrapPanel lays out items in a top-to-bottom column and then uses additional columns to fit the remaining items.
DockPanel	Aligns elements against an entire edge of the container.
Grid	Arranges elements in rows and columns according to an invisible table. This is one of the most flexible and commonly-used layout containers.
UniformGrid	Places elements in an invisible table but forces all cells to have the same size. This layout container is used infrequently.
Canvas	Allows elements to be positioned absolutely using fixed coordinates. This layout container is the most similar to traditional Windows Forms, but it doesn't provide anchoring or docking features. As a result, it's an unsuitable choice for a resizable window unless you're willing to do a fair bit of work.

Enough has been said about WPF. Two simple examples will be given to illustrate how XAML is used to design the appearance of applications. The first example is to create a browser-based application that draws and stacks three shapes in a grid. Save the following code as `DrawShapes.xaml` in any folder:

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowTitle="Draw sahpes"
  Title="Draw shapes">
  <Grid>
    <StackPanel>
      <TextBlock Text="Ellipse:" Margin="5"/>
      <Ellipse Width="150" Height="70"  Fill="yellow" Stroke="Black"/>

      <TextBlock Text="Rectangle without fillet:" Margin="5"/>
      <Rectangle Width="150" Height="70" Fill="red" Stroke="Black"/>

      <TextBlock Text="Rectangle with fillet of radius 20:"Margin="5"/>
      <Rectangle Width="150" Height="70" RadiusX="20" RadiusY="20"
                 Fill="blue" Stroke="Black"/>
    </StackPanel>
  </Grid>
```

```
</Page>
```

Double click `DrawShapes.xaml`, and Microsoft's Internet Explorer will display the specified shapes as follows:

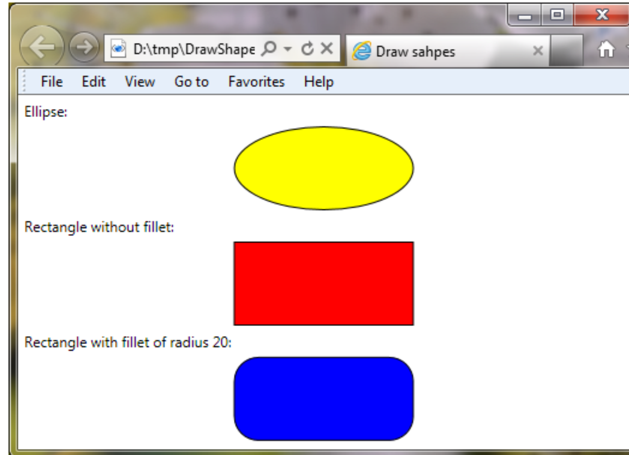


Figure 10.1: Draw simple shapes via XAML

This is a simple markup-only, browser-based application since no $C^{\#}$or .Net VB code has been involved. Our second example is creating a `Canvas` inside the `Grid` with the background of rich-looking gradient color (from light-blue to blue). A `DrawLine` button is also placed inside the `Grid` at the top-right corner with (20,20) margin measured from the top-right corner. Save the following markup code as `DrawLine.xaml` in any folder.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   WindowTitle="Button in mixed layout" Height="400" Width="600">
  <Grid x:Name="myGrid">
   <Canvas x:Name="myCanvas" ClipToBounds="True">
   <Canvas.Background>
      <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <LinearGradientBrush.GradientStops>
      <GradientStop Color="LightBlue" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
      </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>
   </Canvas.Background>
   </Canvas>
   <Button Content="DrawLine" Height="23" HorizontalAlignment="Right"
                   Margin="0,20,20,0" Name="button1"
                   VerticalAlignment="Top" Width="75"/>
  </Grid>
</Page>
```

Double click DrawLine.xaml, and Microsoft's Internet Explorer will display the browser-based application as follows:

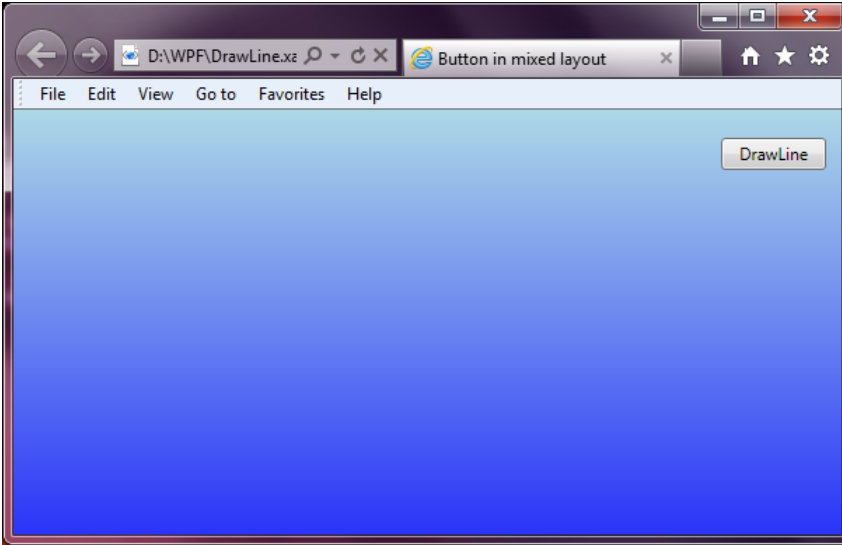


Figure 10.2: Create a browser-based application via XAML

Clicking the DrawLine button does nothing because no "code-behind" has been implemented to handle the button clicking event. We will not implement the event handler for this case because WPF and WinForm are part of .Net Framework that works with managed code such as C#, Microsoft Visual Basic .NET, or managed C++.

Throughout this book, all of our algorithms have been implemented in conventional C/C++ (or unmanaged C/C++ code in Microsoft's term) to make them workable for any platform. Consequently, we will not be able to call these APIs from WPF and WinForm without considerable effort. For this reason, MFC is chosen to implement our Windows-based application that displays the best fit line, circle, and arc via GDI interface.

10.2 Create MFC application template

The MFC library uses an object-oriented model that eliminates much of the tedium and exacting detail of the Win32 API, yet it still offers most of the power needed to create full-featured Windows programs. As a matter of fact, most commercially available windows-based CAD systems have been developed using the MFC library. This section is designed to familiarize readers with the basics of MFC library through creation of an MFC application template with customized GUIs.

So far, all algorithms have been implemented as console applications. For a console application, the only absolute requirement is a function named main. The operating

system calls `main` when we run the program, and from that point on, we implement our application and can use any programming structure we want. If an application program needs to get user keystrokes, for example, it calls an appropriate I/O function such as `getchar`. When the Windows operating system launches a program, it calls the program's `WinMain` function. An essential difference between a console program and a program written for Windows is that a console program calls the operating system to get user input, but a Windows-based program processes user input via *messages* from the operating system. Messages in Windows are strictly defined and apply to all programs. For example, a `WM_CREATE` message is sent when a window is being created; a `WM_LBUTTONDOWN` message is sent when the user presses the left mouse button. All messages have two 32-bit parameters that convey information such as cursor coordinates, key code, and so forth. Windows sends `WM_COMMAND` messages to the appropriate window in response to user menu choices, dialog button clicks, and so on. Command message parameters vary depending on the window's menu layout.

Message mapping in MFC provides an efficient way to direct Windows messages to an appropriate C++ object instance. In the implementation (.cpp) file that defines the member functions for our class, start the message map with the `BEGIN_MESSAGE_MAP` macro, then add macro entries for each of our message-handler functions, and complete the message map with the `END_MESSAGE_MAP` macro. Although message-map macros are important, we generally won't have to use them directly. As will be seen later, this is because the `Properties` window automatically creates message-map entries in our source files when we use it to associate message-handling functions with messages. Any time we want to edit or add a message-map entry, we can use the `Properties` window to achieve so.

Besides the difference in handling user input, console and windows-based applications handle output/display also differently. Many console programs write directly to the video memory and the printer port. The disadvantage of this technique is the need to supply driver software for every video board and every printer model. Windows introduced a layer of abstraction called the *Graphic Device Interface (GDI)*. Windows provides the video and printer drivers, so the application program does not need to know the type of video card and printer attached to the system. Instead of addressing the hardware, the application program calls GDI functions that reference a data structure called *device content (DC)*. Windows maps the device content structure to a physical device and issues the appropriate input/output instruction. In essence, applications based on the Microsoft Win32 API do not access graphics hardware directly. Instead, GDI interacts with device drivers on behalf of the applications.

Having discussed the major differences between a console and windows-based application, it is time for us to create an MFC application template. Let's start Visual Studio 2010 and click `File->New->Project` to bring up the `New Project` dialog box. Select `Visual C++->MFC->MFC Application` in the `Templates` pane to open the wizard. Define the application settings such as path and project name using the MFC Application Wizard as shown in figure 10.3. Click `OK` to bring up an application wizard. Let's select `Single document` for application type and `MFC standard` for project style as illustrated in figure 10.4.

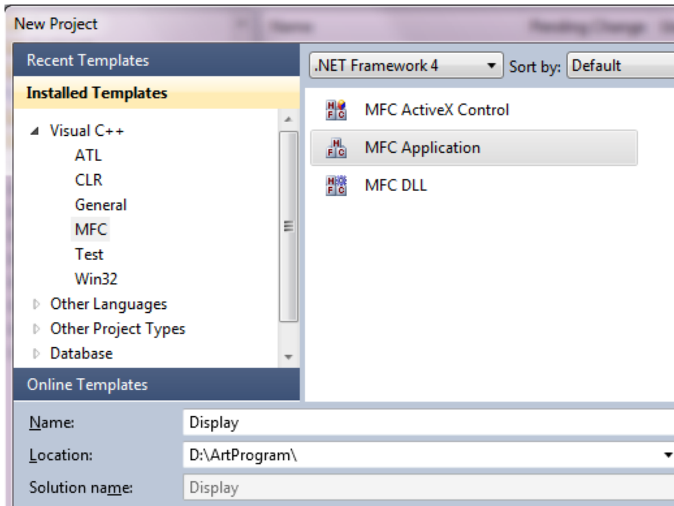


Figure 10.3: Create a project for MFC application

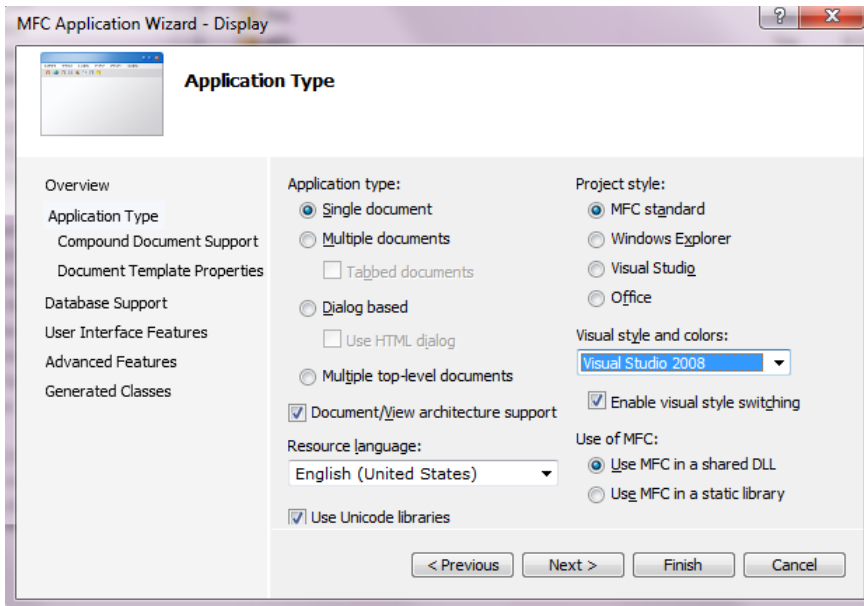


Figure 10.4: Create a single document MFC application

MFC makes it easy to work with both single-document interface (SDI) and multiple-document interface (MDI) applications. SDI applications allow only one open document frame window at a time. MDI applications allow multiple document frame windows to be open in the same instance of an application. An MDI application has a window within which multiple MDI child windows, which are frame windows themselves, can be opened, each containing a separate document. In some applications, the child windows

can be of different types, such as chart windows and spreadsheet windows. In that case, the menu bar can change as MDI child windows of different types are activated. In our case, SDI application is good enough.

Repeatedly click `Next` until we reach `Advanced Features`. Then, deselect every checkbox and click `Finish` as shown in figure 10.5.

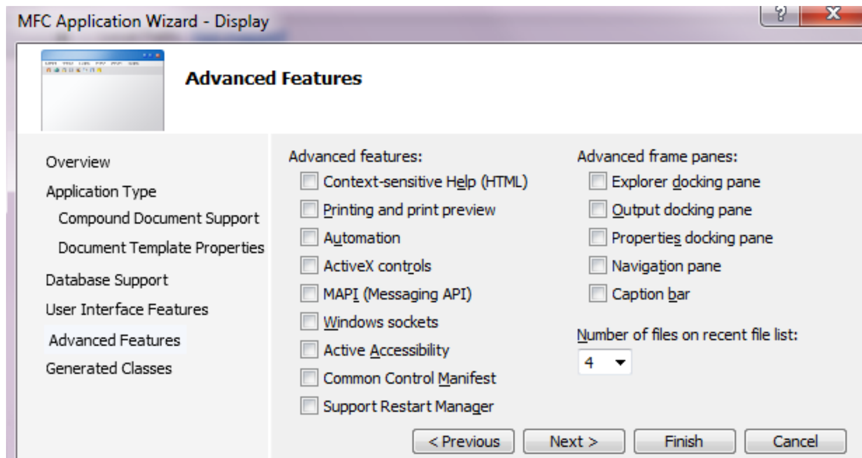


Figure 10.5: Deselect all advanced features

Five `*.cpp` source files will be added into the project as indicated in figure 10.6.

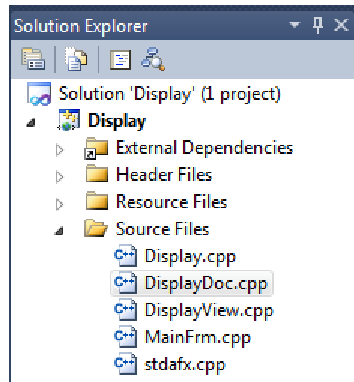


Figure 10.6: Source files created by wizards

Among these five source files, we are mainly interested in `DisplayView.cpp` in which event handling, message mapping, and drawing activities will be implemented. At this stage, we can, without any coding, compile and press `F5` to run the project, which brings up a familiar windows document application with `File`, `Edit`, `view`, and `Help` entries on the menu-bar and a few icons on the toolbar as shown below.

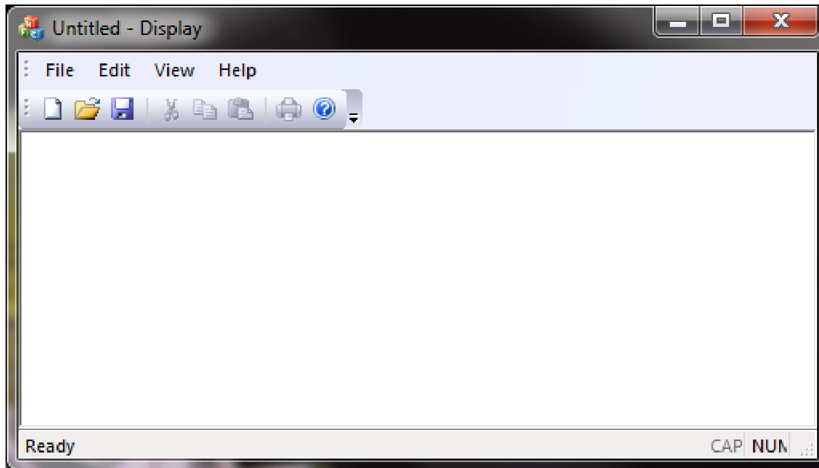


Figure 10.7: Appearance of MFC application

As seen above, MFC Application Wizard has eased the burden for us to design and add basic features to a new MFC application. The framework defines the skeleton of an application and supplies standard user-interface implementations that can be placed onto the skeleton. As a programmer, our job is to fill in the rest of the skeleton, which are those things that are specific to our application.

We now look how to add the following three customized icons to the toolbar:

```
BestLine,
BestCircle,
BestArc.
```

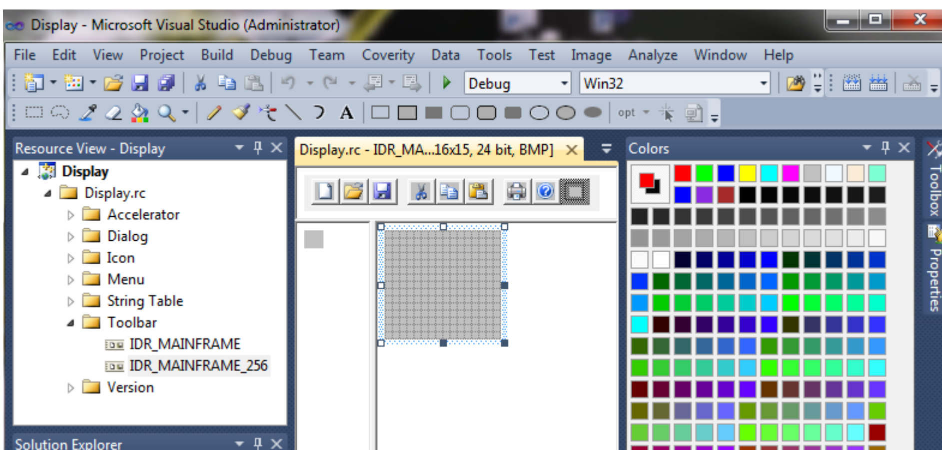


Figure 10.8: Invoke Toolbar utility

These three icons are GUIs for users to interact with the application to create the best

fit line, circle, or arc. With Microsoft Visual Studio IDE, we can use the resource editor to create and edit menus, dialog boxes, custom controls, accelerator keys, bitmaps, icons, cursors, strings, and version resources. To see how it works, click `View->Resource View` on the project's menu-bar to bring up `Resource View` and place it on the left pane as shown in figure 10.9. Expand the `Toolbar` folder and double click `IDR_MAINFRAME_256` to bring up the toolbar recourse file `Display.rc`. Click the gray, rectangle box after the question mark icon on the toolbar (referring to figure 10.8). Then select a proper tool on the `Image editor` toolbar to create three bitmap images to represent the best fit line, circle, and arc icons as shown in figure 10.9. It should be pointed out that software companies usually use more advanced image editing tools to create the rich and nice-looking bitmap images for toolbar icons.

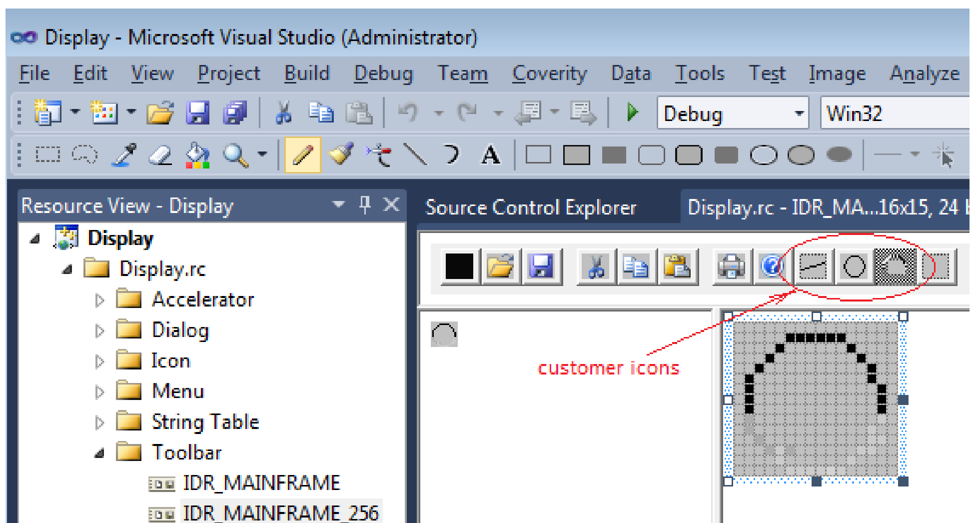


Figure 10.9: Design customized command icons

Save, compile, and run the project. We now should see three new icons on the toolbar (screenshot is not provided). Stop the application so that we can continue our implementation. If `Display.rc` is not active in the `Toolbar` editor window, double click `IDR_MAINFRAME_256` from `Resource View` pane to make it active. Then, double click the best fit line icon to bring up `Properties` window of the `Toolbar Editor` as shown in figure 10.10. Type `ID_BestLine` for `ID` name and "*Best Fit Line: Click the left mouse button to generate points, then the right button to create best fit line*" for `Prompt` (referring to figure 10.10). The description for this command will be displayed on the `status bar`. We similarly name `ID_BestCircle` and `ID_BestArc` for the other two icons and provide informative prompt text strings.

Save, compile, and run the project. When the application is running, moving the mouse cursor to each icon triggers display of text string on the `status bar` at the bottom of the application document. The prompt string helps users take required actions.

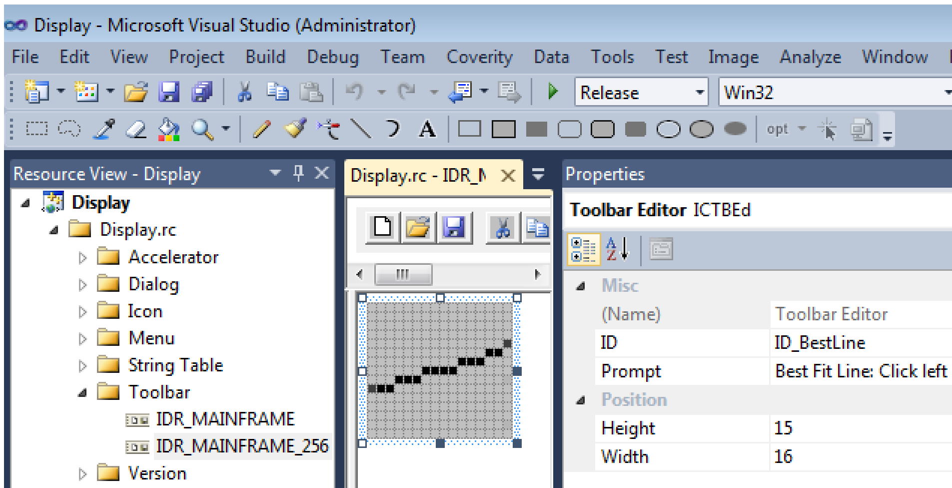


Figure 10.10: Add prompt text string

We are done with GUI design. It is time to explore how MFC `Class Wizard` assists us to add a message handler (a member function that handles Windows messages) to a class and map Windows messages to the message handler. On the project menu-bar, click `Project->Class Wizard`. In the popup `MFC Class Wizard` dialog box, select `CDisplayView` for `Class name` and browse down to find `ID_BestLine` in `Object IDs`. Then, click the `Add Handler` button to add `COMMAND` and also `UPPDATE_COMMAND_UI` as illustrated in figure 10.11.

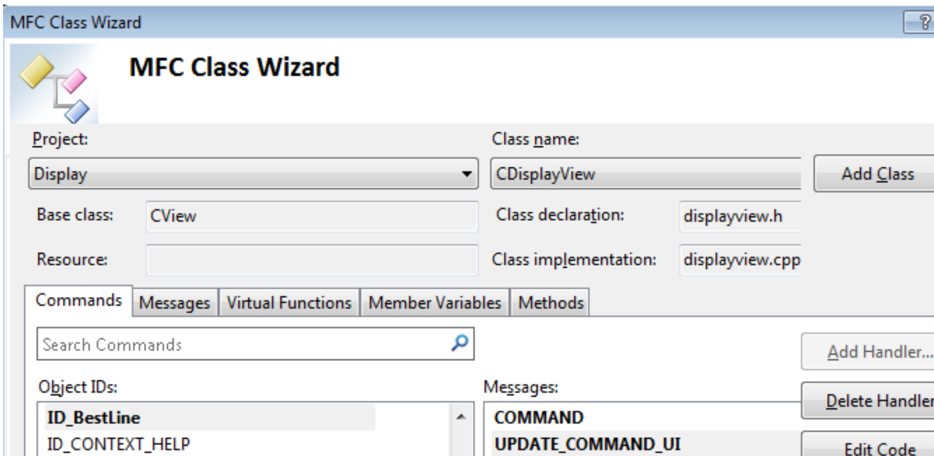


Figure 10.11: Add command ID

Click the `Apply` button, which will add the required declarations to the header file `DisplayView.h`, the corresponding entries to `Message Map`, and the following stub member function bodies to the class's implementation file `DisplayView.cpp`:

```
void CDisplayView::OnBestline()
{
   // TODO: Add your command handler code here
}


void CDisplayView::OnUpdateBestline(CCmdUI *pCmdUI)
{
   // TODO: Add your command update UI handler code here
}
```

Based on the above skeleton code bodies, we can implement or edit our message handler code later. We need to repeat similar processes for `ID_BestCircle` and `ID_BestArc`.

We also need to add the left and right mouse-button click entries and methods to `DisplayView.cpp`. On the project menu-bar, click `Project->Class Wizard`. In the popup `MFC Class Wizard` dialog box, select `Messages` and browse down to find `WM_LBUTTONDOWN` and click `Add Handler` (referring to figure 10.11). Next, browse down to find `WM_RBUTTONUP` and click `Add Handler`.

We are done with the design and implementation of application template. Let's open `DisplayView.cpp` to see what MFC `Class Wizard` has done to this `.cpp` file. We will immediately notice that all macro entries for our message-handler functions have been added to the message map as seen below:

```
BEGIN_MESSAGE_MAP(CDisplayView, CView)
  ON_WM_RBUTTONUP()
  ON_COMMAND(ID_BestLine, &CDisplayView::OnBestline)
  ON_UPDATE_COMMAND_UI(ID_BestLine, &CDisplayView::OnUpdateBestline)
  ON_WM_LBUTTONDOWN()
  ON_COMMAND(ID_BestArc, &CDisplayView::OnBestarc)
  ON_UPDATE_COMMAND_UI(ID_BestArc, &CDisplayView::OnUpdateBestarc)
  ON_COMMAND(ID_BestCircle, &CDisplayView::OnBestcircle)
  ON_UPDATE_COMMAND_UI(ID_BestCircle,&CDisplayView::OnUpdateBestcircle)
END_MESSAGE_MAP()
```

Scrolling down the source file, we will also notice that all message-handler stub member function bodies have been created and are waiting for us to add our code to handle specific events. For example, the left mouse button down message-handler function body is created as

```
void CDisplayView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // TODO: Add your message handler code here and/or call default
}
```

In the next section we shall do the actual implementation for message mapping, event handling, and drawing activities.

10.3 Implementation of event handlers

In the previous section, we showed how MFC helped us build a windows-based application with customized GUIs. In this section, we will do the actual implementation to handle a specific event such as `BestFitLine` button clicking, `left mouse button down`, `right mouse button up`, etc. The main tasks are outlined below:

1. Make `BestLine` an active icon by default. When users click a different icon, disable the active one and enable the clicked one.

2. Click the left mouse button to generate points and store them in STL `vector`.

3. Draw a tiny circle at each clicked place to indicate where the point is. This is accomplished by calling the GDI function via an MFC wrapper API.

It helps to understand the implementation with the above tasks in mind. Prior to the actual implementation, however, we have to include `MyHeader.h` in this project so that STL `vector`, `GPosition2d`, and other structures are exposed. This may be done by adding `#include "MyHeader.h"` in each `.cpp` file. Alternatively, we may simply add it in `stdafx.h` since this standard system header file is included in each `.cpp` file. We then need to add the path `D:\ArtProgram\include` in the field of `Addtional Include Directories` in project's configuration properties settings (referring to Figure 4.8 in Chapter 4) so the compiler knows where to look for `MyHeader.h`. We can now start our implementation. Open `DisplayView.h` to add the following member variables:

```
public:
   double          m_scale;
   CRect           m_clientRect;
   int             m_fitOption;
   bool            m_bDrawCurve;
   vector<GPosition2d> m_gPoints;
```

The variable `m_clientRect` stores the size of the active document in pixels and is used to determine a transformation between the *device* and *world* coordinates. The device coordinate is given in pixels, and the origin is at the top-left corner with the y-direction pointing downward (the so-called *left-hand coordinate system*). The world coordinate often refers to the coordinate system used by applications and is measured in either metric or imperial units. Its origin can be anywhere but is often at the bottom-left corner with the x-axis pointing rightward and the y-axis pointing upward (the so-called *right-hand coordinate system*). The variable `m_fitOption` takes three values: 1, 2, and 3 corresponding to the best fit line, circle, and arc. The variable `m_bDrawCurve` indicates whether it is ready to draw a curve. Finally, `m_gPoints` collects all points generated by repeatedly clicking the left mouse button. Open `DisplayView.cpp` and initialize the variables inside the constructor as follows:

```
CDisplayView::CDisplayView()
{
   m_scale = 1.0;
   m_clientRect.left = 0;
   m_clientRect.top = 0;
```

```
   m_clientRect.bottom = 400;
   m_clientRect.right = 800;
   m_fitOption = 1;
   m_bDrawCurve = false;
}
```

The constructor is called when the executable is invoked. At this time, the application window (or the client window) has not been brought up. Therefore, the exact window size is unknown, and `m_clientRect` is initialized by some reasonable guessing values. It will be correctly reset when the application window shows up.

When the `BestLine` icon is clicked, the `OnBestline` method will be called. Inside this method, we initialize the variables as follows:

```
void CDisplayView::OnBestline()
{
   // TODO: Add your command handler code here
   m_fitOption = 1;
   m_bDrawCurve = false;
   m_gPoints.clear();
   Invalidate();
}
```

A call to `Invalidate` sends out a `WM_PAINT` message, which, in turn, triggers the `paint` method to refresh and repaint the document. Since we have cleaned `m_gPoints` and set `m_bDrawCurve` to false, the effect of `Invalidate` is to erase what has been drawn previously in the screen. This is important because users need a fresh screen to experiment with the best fit line when they click the `BestLine` icon.

Next, we add implementation to

```
void CDisplayView::OnUpdateBestline(CCmdUI *pCmdUI)
{
   // TODO: Add your command update UI handler code here
   if (m_fitOption == 1)
      pCmdUI->SetCheck(1);
   else
      pCmdUI->SetCheck(0);
}
```

This method is called whenever a toolbar icon is clicked or a screen is resized. It is responsible to set the `BestLine` icon active or inactive, depending on the value of `m_fitOption`. We need to add similar implementations for `BestCircle` and `BestArc`:

```
void CDisplayView::OnBestcircle()
{
   // TODO: Add your command handler code here
    m_fitOption = 2;
    m_bDrawCurve = false;
    m_gPoints.clear();
```

```
    Invalidate();
}

void CDisplayView::OnUpdateBestcircle(CCmdUI *pCmdUI)
{
   // TODO: Add your command update UI handler code here
   if (m_fitOption == 2)
      pCmdUI->SetCheck(1);
   else
      pCmdUI->SetCheck(0);
}

void CDisplayView::OnBestarc()
{
   // TODO: Add your command handler code here
   m_fitOption = 3;
   m_bDrawCurve = false;
   m_gPoints.clear();
   Invalidate();
}

void CDisplayView::OnUpdateBestarc(CCmdUI *pCmdUI)
{
   // TODO: Add your command update UI handler code here
   if (m_fitOption == 3)
      pCmdUI->SetCheck(1);
   else
      pCmdUI->SetCheck(0);
}
```

As was said at the beginning of this section, data points are generated by clicking the left mouse button. The point sending to `OnLButtonDown` specifies the x- and y-coordinate of the cursor. These coordinates are always relative to the upper-left corner of the window (i.e., the device coordinates). Therefore, conversions between the device and world coordinates are required. If both the device and world coordinate use the pixel unit, the conversion is simply:

$$x_{\mathrm{wd}} = x_{\mathrm{dc}},$$
$$y_{\mathrm{wd}} = y_{\mathrm{dc_max}} - y_{\mathrm{dc}},$$

which transforms the origin from top-left to bottom-left corner and from a left-hand to right-hand coordinate system. If the world coordinate is measured in a different unit, then the ratio of a pixel on the client window to the corresponding distance on the world coordinate has to be determined. By taking this ratio, known as *scaling factor* or *scale*, into consideration, the conversion formula is

$$x_{\mathrm{wd}} = x_{\mathrm{dc}}/\mathrm{scale},$$
$$y_{\mathrm{wd}} = (y_{\mathrm{dc_max}} - y_{\mathrm{dc}})/\mathrm{scale}.$$

For simplicity, we set `m_scale=1` for now in our implementation:

```
void CDisplayView::DCToWorld(CPoint &cPoint, GPosition2d &gPoint)
{
   gPoint.x = cPoint.x / m_scale;
   gPoint.y = (m_clientRect.bottom - cPoint.y ) / m_scale;
}

void CDisplayView::WorldToDC(GPosition2d &gPoint, CPoint &cPoint)
{
   cPoint.x = (int)(gPoint.x * m_scale);
   cPoint.y = (int)(m_clientRect.bottom - gPoint.y * m_scale);
}
```

Since the above two methods are customized functions, we have to declare their signatures inside the class of `CDisplayView` in `DisplayView.h`:

```
public:
   void DCToWorld(CPoint &cPoint, GPosition2d &gPoint);
   void WorldToDC(GPosition2d &gPoint, CPoint &cPoint);
```

We now implement the `OnLButtonDown` method that handles the left-mouse-button clicking event. The basic tasks are:

- Convert the mouse position to the world coordinate when the left button is down.
- Add the converted mouse position to `m_gPoints`.
- Send a `WM_PAINT` message via a call to `Invalidate` so that circles can be drawn at collected mouse positions.

Accordingly, the implementation is:

```
void CDisplayView::OnLButtonDown(UINT nFlags, CPoint point)
{
   // TODO: Add your message handler code here and/or call default
   if (!m_bDrawCurve)
   {
      GPosition2d gPoint;
      DCToWorld(point, gPoint);
      m_gPoints.push_back(gPoint);
      Invalidate();
   }
}
```

`Invalidate` sends a message to the `CDisplayView::OnDraw` method that overrides the virtual function `CView::OnDraw`. It is called by the framework to perform screen display, printing, and print preview. In our case, `CDisplayView::OnDraw` will draw tiny red circles on the screen to indicate mouse clicking positions.

```
void CDisplayView::OnDraw(CDC* pDC)
{
   // TODO: add draw code for native data here
   int      i;
   CPoint   cPoint;
   CPen     redPen;

   GetClientRect(m_clientRect); // Reset client window size

   // Draw collected points in red circles.
   // Pen style is solid and width is 2 pixels.
   redPen.CreatePen(PS_SOLID, 2, RGB(225, 0, 0));
   pDC->SelectObject(&redPen);
   for (i=0; i<(int)m_gPoints.size(); i++)
   {
      WorldToDC(m_gPoints[i], cPoint);

      // Draws an ellipse specified by a bounding Rectangle:
      pDC->Ellipse(CRect(cPoint.x-3, cPoint.y-3,
                         cPoint.x+3, cPoint.y+3));
   }
}
```

The class of a device-context (CDC) object provides member functions for working with a device context, such as a display or printer, as well as members for working with a display context associated with the client area of a window. All drawing is done through the member functions of a CDC object. The class provides member functions for device-context operations, working with type-safe graphics device interface (GDI) object selection and drawing tools to draw simple shapes such as lines, rectangles, ellipses, polygons, and so on. Explanations of these graphics methods can be readily found on the Microsoft Developer Network (MSDN) website. For example, googling "CDC ellipse MSND" will lead us to the MSDN website with detailed explanations of two signatures of `CDC::Ellipse`. The MSDN website has been the important reference resource for professional programmers who develop Windows applications.

Save and compile the project. Press `F5` to start windows application. When the application window shows up, we can click the left mouse button. With each click, a red circle will be drawn instantly.

10.4 Draw line, circle, and arc

Data points are generated by repeatedly clicking the left mouse button while moving the mouse around. When enough points have been collected (the number of points should be greater than 2), we need to send out the `WM_PAINT` message to tell `CDisplayView::OnDraw` to draw the best fit curve. This is done by clicking the right mouse button. The implementation of `OnRButtonUp` method is:

```
void CDisplayView::OnRButtonUp(UINT /* nFlags */, CPoint point)
{
   if (m_gPoints.size() > 2)
   {
      m_bDrawCurve = true;
      Invalidate();
   }
}
```

Before implementing `CDisplayView::OnDraw` to draw the best fit curve, we need to add our dynamic linking library to the project so that the least squares approximation APIs can be called. Let's create a new folder `Lib` under `Display` project in the `Solution Explore` pane (referring to section 4.2 for detail). Highlight `Lib` and click the right mouse button to add `FunctDLL_d.lib` and `FunctDLL_r.lib`. It is important to exclude `FunctDLL_r.lib` from building a `Debug` code and `FunctDLL_d.lib` from `Release` mode (referring to figure 4.14).

We now consider how to draw a curve. In computer graphics and computer-aided geometric design, curves are often represented parametrically. For example, a vector-valued parametric line is given by

$$\boldsymbol{r}(t) = \boldsymbol{p} + t\boldsymbol{v},$$

where $\boldsymbol{p}$ is an arbitrary point on the line, $\boldsymbol{v}$ is the direction vector, and $t \in (-\infty, +\infty)$ is the arc length parameter. For any given t, evaluation of the above equation yields a unique point on the line. For the convenience of programming, we may define a data structure to store a finite line as

```
typedef struct{
   GPosition2d   m_point; // arbitrary point on the line
   GUnitVector2d m_dirV;  // direction vector
   double        m_start; // start parameter of finite line
   double        m_end;   // end parameter of finite line
} GLine2d;
```

Similarly, a vector-valued parametric ellipse is given by

$$\boldsymbol{r}(t) = \boldsymbol{c} + \cos(t)\,\boldsymbol{a} + \sin(t)\,\boldsymbol{b},$$

where $\boldsymbol{c}$ is the center of the ellipse, $\boldsymbol{a}$ and $\boldsymbol{b}$ are the semi-major and semi-minor axes of the ellipse, and $t \in [0, 2\pi]$ is the angular parameter in radian. When the lengths of $\boldsymbol{a}$ and $\boldsymbol{b}$ are the same, an ellipse degenerates to a circle. Accordingly, we may use the following data structure to store both an ellipse and a circle:

```
typedef struct{
   GPosition2d  m_center;    // center of ellipse
   GVector2d    m_majorAxis; // semi-major axis
   double       m_minorMajorRatio;
} GEllipse2d;
```

Since a semi-minor axis can be derived from the semi-major axis and the ratio of the lengths of minor and major axes, the above representation saves memory usage by omitting the semi-minor axis.

We can go on and on discussing parametric representations of all kinds of curves and surfaces, associated structures for these curves and surfaces, and implementation of supporting algorithms that include point-stroking, transformation, evaluation, etc. Since not every reader is interested in curve and surface modeling if he or she does not want to work in gaming and CAD/CAM industries, we will not go into detail about parametric representation of a line, circle, arc, B-spline curve, etc. Besides, covering such information needs a book, not a section. Interested readers may refer to my online book *Java Graphics Programming* at www.infogoaround.org or any published computer graphics and geometric modeling book.

If we had defined data structures to support parametric representations of all curves and implemented a generic API to stroke points on these curves, the display of a parametric curve would simply be rendered by drawing a bunch of lines that connect the obtained stroking points. Omission of such information makes our implementation of `Display` application relatively tedious.

We first consider how to draw a line. Given the data points, `apiBestFitLine` returns a point `pt` on the best fit line and the line direction `dirV`. Such information is only good to define an infinite line. Addtional information is required to draw a finite line on the screen. Referring to the following figure, the dot-product of displacement vector $(\boldsymbol{p}_i - \boldsymbol{p})$ and the direction unit vector `dirV` yields the signed distance from $\boldsymbol{p}$ to the projection point $\boldsymbol{p}_{\text{proj}}$.

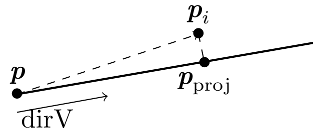


Figure 10.12: Project a point onto a line

Denoting this dot-product by parameter t, we can loop through each point to compute the corresponding parameter

$$t_i = (\boldsymbol{p}_i - \boldsymbol{p}) \cdot \boldsymbol{v},$$

where $\boldsymbol{v}$ is the direction unit vector. By recording the smallest and largest parameters: t_{min} and t_{max}, we can compute the start and end points of the best fit line as follows:

$$\boldsymbol{p}_{\text{start}} = \boldsymbol{p} + t_{\text{min}}\boldsymbol{v} \quad \text{and} \quad \boldsymbol{p}_{\text{end}} = \boldsymbol{p} + t_{\text{max}}\boldsymbol{v}.$$

The start and end points are sufficient for us to draw a finite line that best fits the collected points.

Drawing a circle is straightforward. However, it requires several special functions to properly draw a circular arc – the trimmed circle. We start with the implementation of an API that computes an angle of a circle from the given point.

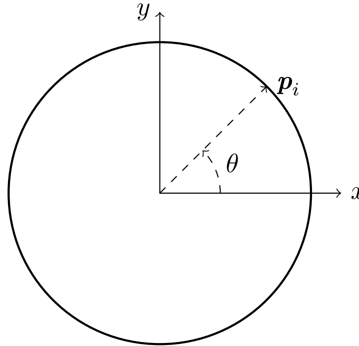


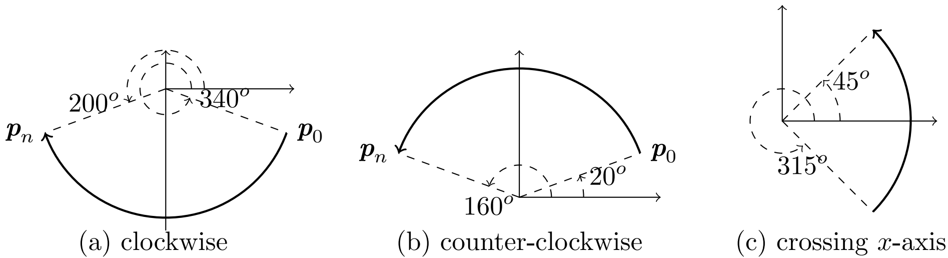
Figure 10.13: Evaluation of a circle

Referring to figure 10.13, an angle, which is the parameter of parametric circle, is *always* measured counter-clockwise from the major axis (i.e., the x-axis in the above figure) to avoid getting a negative parameter. It can be computed in the following code list:

```
/*---------------------------------------------------------------
API Name
   apiGetAngleFromPoint

Description
   It computes the angular parameter of a circle from a given point

Signature
   double apiGetAngleFromPoint(GPosition2d &center, GPosition2d& pt)

   INPUT:
      center      center of the circle
      pt          point that is either on or near the circle
   OUTPUT:
      returns the angular parameter associated with the given point

History

   Y.M. Li     03/10/2013 : Creation date
---------------------------------------------------------------*/
#include "MyHeader.h"

double apiGetAngleFromPoint(GPosition2d &center, GPosition2d& pt)
{
   double      dx, dy, angle;
   dx = pt.x - center.x;
```

```
   dy = pt.y - center.y;
   angle = atan2(dy, dx);
   if (angle < 0)
      angle += 2.0 * GPI;
   return angle;
}
```

With this API, we can determine the start and end angle of the arc that corresponds to the first and last clicked point. Besides the start and end angle, we also need to know the orientation of an arc that indicates how the point on the arc flows when moving from $\boldsymbol{p}_0$ towards $\boldsymbol{p}_n$. Therefore, an arc can have a clockwise or counter-clockwise orientation as shown below. If the orientation is messed up, the arc will be drawn incorrectly. It should be noted that, to avoid getting negative angular parameter, the angle parameter of an arc is always measured counter-clockwise from the major axis as illustrated below. This counter-clockwise measurement of angle parameter has nothing to do with the orientation of the arc.



(a) clockwise (b) counter-clockwise (c) crossing x-axis

Figure 10.14: Different orientation of arc

Though tricky, it is possible to determine the correct orientation of the fitting arc. We first compute the starting and ending angles (i.e., θ_0 and θ_n) that correspond to $\boldsymbol{p}_0$ and $\boldsymbol{p}_n$. We then take a middle point in an array of collected mouse points and compute the corresponding angle θ. If the fitting arc is counter-clockwise oriented as illustrated in figure 10.14 (b), we have $\theta_0 < \theta < \theta_n$ because parametric angle is always measured counter-clockwise from the major axis. If $\theta_n < \theta < \theta_0$ (referring to figure 10.14 (a)), the fitting arc is clockwise oriented. It should be noted that, when an arc crosses the major axis as shown in (c), the parametric interval is broken into two: $[315^o, 360^o]$ and $[0^o, 45^0]$. Because of these configurations, it is not very straightforward to implement an algorithm that detects whether the given arc parameter is in the parametric interval. We purposely break the implementation into several `if-else` blocks to make the code more understandable:

```
/*-----------------------------------------------------------------
API Name
   apiIsParInRange

Description
   It checks if the given angle is in the valid angular parametric
   interval. Note: angles are always measured counter clockwise
```

```
   regardless arc orientation.

Signature
   bool apiIsParInRange(double u_beg, double u_end,
                        bool bCounterClock, double u)

   INPUT:
      u_beg          starting parameter
      u_end          ending parameter
      bCounterClock  true if it is counter-clock oriented
      u              angular parameter to be checked
   OUTPUT:
      returns true if it is in the interval

History

   Y.M. Li     03/10/2013 : Creation date
-----------------------------------------------------------------*/
#include "MyHeader.h"

bool apiIsParInRange(double u_beg, double u_end,
                     bool bCounterClock, double u)
{
   double   twoPI, tol;
   bool     isInside = false, bCrossAxis = false;

   tol = 1.0e-8;
   twoPI = 2.0 * GPI;
   if (bCounterClock)
   {
      if (u_end < u_beg)
         bCrossAxis = true;
   }
   else if (u_end > u_beg)
   {
      bCrossAxis = true;
   }

   if (!bCrossAxis)
   {
      if (bCounterClock)
      {
         if (u > u_beg - tol && u < u_end + tol)
            isInside = true;
      }
      else
      {
```

```
         if (u > u_end - tol && u < u_beg+ tol)
            isInside = true;
      }
   }
   else
   {
      if (bCounterClock)
      {
         if (u > u_beg- tol && u < twoPI + tol ||
             u > 0 && u < u_end + tol)
             isInside = true;
      }
      else
      {
         if (u < u_beg + tol && u > 0 ||
             u > u_end - tol && u < twoPI + tol)
             isInside = true;
      }
   }

   return isInside;
}
```

In apiStrokeCircle, the number of stroking points is determined based on the full sweep angle 2π. For the case of arc (i.e., the trimmed circle), the sweep angle is no longer 2π and has to be determined properly. The computation of the sweep angle is not simply the difference of start and end angles. Referring to figure 10.14, the sweep angle for (c) is 90° and is determined by taking into consideration that the arc crosses the major-axis (i.e., the x-axis).

```
/*----------------------------------------------------------------
API Name
   apiGetSweepAngle

Description
   It computes the sweep angle of an arc based on the given
   radius, start and end angles, as well as orientation.

Signature
   double apiGetSweepAngle(double radius, double startAngle,
                           double endAngle, bool bCounterClock)

   INPUT:
      radius         radius of arc
      startAngle
      endAngle
      bCounterClock  true if it is counter-clock oriented
   OUTPUT:
```

```
      returns the sweep angle

History

   Y.M. Li     03/10/2013 : Creation date
----------------------------------------------------------------*/
#include "MyHeader.h"

double apiGetSweepAngle(double radius, double startAngle,
                     double endAngle, bool bCounterClock)
{
   double   tol, sweepAngle, twoPI = 2.0 * GPI;

   tol = 1.0e-5 / radius;
   if (fabs(fabs(endAngle - startAngle)-twoPI) < tol)
   {
      sweepAngle = 2.0 * GPI;
   }
   else
   {
      if (bCounterClock)
      {
         if (startAngle < endAngle)
            sweepAngle = endAngle - startAngle;
         else
            sweepAngle = twoPI - (startAngle - endAngle); //Cross axis
      }
      else
      {
         if (startAngle > endAngle)
            sweepAngle = startAngle - endAngle;
         else
            sweepAngle = twoPI - (endAngle - startAngle); //Cross axis
      }
   }
   return sweepAngle;
}
```

The method to stroke a circular arc is the same as the one used to stroke a circle except that the sweep angle is not 2π and has to be determined by calling the above API. Because of the similarity of the two APIs, let us copy `apiStrokeCircle.cpp` and save it as `apiStrokeArc.cpp`. We then modify it as follows to stroke a circular arc:

```
/*---------------------------------------------------------------
API Name
   apiStrokeArc

Description
```

```
   It strokes an arc with respect to the chord-height tolerance.

Signature
   void apiStrokeArc(GPosition2d &center, double radius, double tol
              double startAngle, double endAngle, bool bCounterClock,
              std::vector<GPosition2d>& pStrkPts,  apiError &rc)

    INPUT:
      center         center of circle
      radius         radius of circle
      startAngle
      endAngle
      bCounterClock  true if arc orientation is counter clockwise
      tol            chord-height tolerance
    OUTPUT:
      pStrkPts       STL vector that stores stroking points
      apiError       error code: api_OK if no error

History

   Y.M. Li     03/01/2013 : Creation date
---------------------------------------------------------------*/
#include "MyHeader.h"

void apiStrokeArc(GPosition2d &center, double radius, double tol
              double startAngle, double endAngle, bool bCounterClock,
              std::vector<GPosition2d>& pStrkPts,  apiError &rc)
{
   int         i, nStrkPts;
   double      sweepAngle, delta, sin_delta, cos_delta;

   // Initialize data and determine the number of stroking points
   rc = api_OK;
   delta = 2.0 * acos(1.0 - tol / radius);
   sweepAngle = apiGetSweepAngle(radius, startAngle,
                           endAngle, bCounterClock);
   nStrkPts = 2 + (int)(sweepAngle / delta);
   delta = sweepAngle / (nStrkPts - 1);
   if (!bCounterClock)
      delta = -delta;
   pStrkPts.resize(nStrkPts);

   // Loop to compute stroking points on the arc centered at (0,0)
   sin_delta = sin(delta);
   cos_delta = cos(delta);
   pStrkPts[0].x = radius * cos(startAngle);
   pStrkPts[0].y = radius * sin(startAngle);
```

```
    for (i=1; i<nStrkPts; i++)
    {
         pStrkPts[i].x = pStrkPts[i-1].x * cos_delta -
                         pStrkPts[i-1].y * sin_delta;
         pStrkPts[i].y = pStrkPts[i-1].y * cos_delta +
                         pStrkPts[i-1].x * sin_delta;
    }

    // Translate the points based on the given center
    for (i=0; i<nStrkPts; i++)
    {
       pStrkPts[i].x += center.x;
       pStrkPts[i].y += center.y;
    }
}
```

It should be pointed out that it is possible to combine two APIs, `apiStrokeCircle` and `apiStrokeArc` into a single API to stroke both the circle and arc with `startAngle`, `endAngle`, and `bCounterClock` being the optional input arguments. However, we keep them separated to make each API more readable.

All the functions we just implemented need to be added to our dynamic-link library and exported properly so that they can be accessed by our windows-based application. We have implemented all necessary supporting functions and are ready to modify `CDisplayView::OnDraw` to render the line, circle, and arc. With added comments, the implementation is mostly self-explanatory and descriptions of GDI functions (e.g., `pDC->SelectObject`, `pDC->Ellipse`, `pDC->LineTo`, etc.) can be readily found on the MSDN website:

```
void CDisplayView::OnDraw(CDC* pDC)
{
   // TODO: add draw code for native data here
   int      i;
   CPoint   cPoint;
   CPen     blackPen, redPen;

   GetClientRect(m_clientRect);

   // Pen style is solid and width is 1 pixel for
   // the black pen and 2 pixels for the red pen:
   blackPen.CreatePen(PS_SOLID, 1, RGB(0, 0, 0));
   redPen.CreatePen(PS_SOLID, 2, RGB(225, 0, 0));

   // Draw collected points with red pen:
   pDC->SelectObject(&redPen);
   for (i=0; i<(int)m_gPoints.size(); i++)
   {
      WorldToDC(m_gPoints[i], cPoint);
```

```
      // Draws a circle specified by a bounding Rectangle:
      pDC->Ellipse(CRect(cPoint.x-3, cPoint.y-3,
                         cPoint.x+3, cPoint.y+3));
   }

   if (m_bDrawCurve && m_gPoints.size() > 2)
   {
      // Draw line, circle, or arc
      apiError    rc = api_OK;
      double      tol = 0.5 / m_scale;
      GPosition2d pt, pt_min, pt_max;
      GVector2d   dirV;

      // Draw line, circle, or arc with black pen:
      pDC->SelectObject(&blackPen);

      if (m_fitOption == 1)
      {
         double      dx, dy, d, dmin = 1.0e+10, dmax = 0.0;
         apiBestFitLine(m_gPoints, pt, dirV, rc);
         if (rc != api_OK)
            return;

         // Compute the start and end point of the finite line:
         dx = m_gPoints[0].x - pt.x;
         dy = m_gPoints[0].y - pt.y;
         dmin = dmax = dx * dirV.x + dy *dirV.y;
         for (i=1; i<(int)m_gPoints.size(); i++)
         {
            dx = m_gPoints[i].x - pt.x;
            dy = m_gPoints[i].y - pt.y;
            d = dx * dirV.x + dy *dirV.y;
            if (d > dmax)
               dmax = d;
            else if (d <dmin)
               dmin = d;
         }
         pt_min.x = pt.x + dmin * dirV.x;
         pt_min.y = pt.y + dmin * dirV.y;
         WorldToDC(pt_min, cPoint);
         pDC->MoveTo(cPoint.x, cPoint.y);

         pt_max.x = pt.x + dmax * dirV.x;
         pt_max.y = pt.y + dmax * dirV.y;
         WorldToDC(pt_max, cPoint);
         pDC->LineTo(cPoint.x, cPoint.y);
```

```
    }
    else if (m_fitOption == 2)
    {
       double   dRadius;
       vector<GPosition2d> pStrkPts;

       apiBestFitCircle(m_gPoints, pt, dRadius, rc);
       if (rc != api_OK)
          return;
       apiStrokeCircle(pt, dRadius, tol, pStrkPts, rc);
       if (rc != api_OK)
          return;

       WorldToDC(pStrkPts[0], cPoint);
       pDC->MoveTo(cPoint.x, cPoint.y);
       for (i=1; i<(int)pStrkPts.size(); i++)
       {
          WorldToDC(pStrkPts[i], cPoint);
          pDC->LineTo(cPoint.x, cPoint.y);
       }
    }
    else if (m_fitOption == 3)
    {
       int      n;
       double   dRadius, startAngle, endAngle, angle;
       bool     bCounterClock = true;
       vector<GPosition2d> pStrkPts;

       apiBestFitArc(m_gPoints, pt, dRadius, rc);
       if (rc != api_OK)
          return;

       // Determine start/end angle and orientation
       n = m_gPoints.size();
       startAngle = apiGetAngleFromPoint(pt, m_gPoints[0]);
       endAngle = apiGetAngleFromPoint(pt, m_gPoints[n-1]);
       angle = apiGetAngleFromPoint(pt, m_gPoints[n/2]);
       if (!apiIsParInRange(startAngle,endAngle,bCounterClock,angle))
          bCounterClock = false;

       apiStrokeArc(pt, dRadius, startAngle, endAngle,
                    bCounterClock, tol, pStrkPts, rc);
       if (rc != api_OK)
          return;

       WorldToDC(pStrkPts[0], cPoint);
       pDC->MoveTo(cPoint.x, cPoint.y);
```

```
        for (i=1; i<(int)pStrkPts.size(); i++)
        {
           WorldToDC(pStrkPts[i], cPoint);
           pDC->LineTo(cPoint.x, cPoint.y);
        }
      }
   }
}
```

It is noted that the tolerance used for stroking a circle or an arc is defined as `0.5/m_scale`. If both the world and device coordinate units are pixel, then `m_scale=1` and 0.5 stands for "half pixel". Otherwise, the tolerance for the world coordinate is the half pixel divided by the scaling factor.

We are done with all required implementations! Save and compile this project. The application can be run by either pressing `F5`, double clicking `Display.exe`, or invoking the executable from the command line. By clicking one of three customized icons on the tool bar, we can experiment with the corresponding fitting method and visualize the result. Figure 10.15 illustrates the result of best fit arc. By design, the best fit arc interpolates the start and end point and approximates the remaining points in a least squares sense.

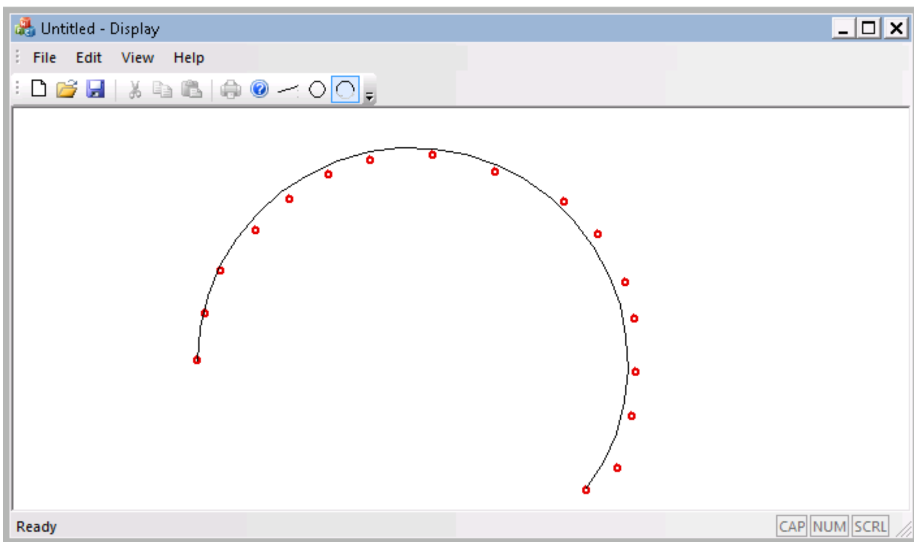


Figure 10.15: Best fit arc

10.5 Display user data

The windows-based application we just created is good for "show-off" or casual demos but is of little practical use. In real world applications, we usually want to display our

own data sets, not some randomly-clicked point cloud. Recall how our main drivers for console applications were designed: each driver opens a specific input file defined via the command-line arguments, reads input data through I/O methods, and calls an associated function to run a test case. For a windows-based application, an input file is usually browsed and selected via the Explorer-style `File Open` dialog box (referring to figure 10.16). By using Explorer-style common dialogs, we give users a consistent and comfortable experience across different programs. We will thus discuss in this section how to add the `File Open` dialog box into our application so that we can browse a file and open it for display. As will be seen later, this seemingly difficult task is readily accomplished with MFS `ClassWizard`. With a slight modification, the code discussed in this section can also be used for a familiar `File Save As` operation.

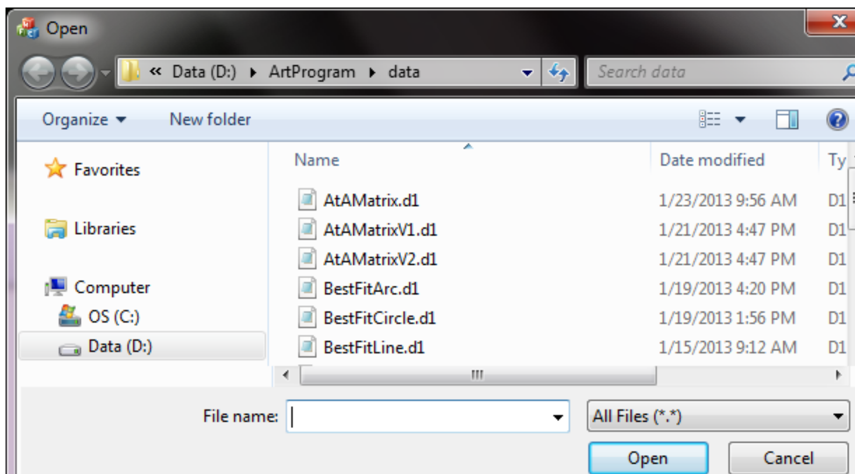


Figure 10.16: File Open dialog box

Let us start our `Display` application from the Visual Studio environment by pressing `F5` in debug mode. When the application window shows up, click `File->Open` to bring up a `File Open` dialog as shown in figure 10.16. This is MFC's default appearance of the `File Open` dialog box. Do not open any file or close or cancel the dialog box. Instead, go back to Visual Studio and click from main menu bar `Debug->Break All` to halt the execution and break into the debugger. At the `Call Stack` window, find `CWinApp::OnFileOpen()` and double click it, which leads us to the following method:

```
void CWinApp::OnFileOpen()
{
   ENSURE(m_pDocManager != NULL);
   m_pDocManager->OnFileOpen();
}
```

`CWinApp` is a base class from which we derive a windows application object, and the member function `CWinApp::OnFileOpen` in the base class can be overridden by derived classes. The purpose of this section is to override this method with our own implementation of `OnFileOpen` in `CDisplayView` class.

Stop the application. On the project menu-bar, click `Project->Class Wizard`. In the popup `MFC Class Wizard` dialog box, select `CDisplayView` for `Class name` and browse down to find `ID_FILE_OPEN` (referring to Figure 10.11); highlight it and click the `Add Handlers` button. Consequently, the following macro entry will be added to message map:

```
ON_COMMAND(ID_FILE_OPEN, &CDisplayView::OnFileOpen)
```

The `CDisplayView::OnFileOpen` method body will also be created. Let us open `DisplayView.cpp` and add the following code to override the default implementation of the `OnFileOpen` method in `CWinApp` class:

```
void CDisplayView::OnFileOpen()
{
   // TODO: Add your command handler code here
   CFileDialog fileDlg(TRUE);

   if (fileDlg.DoModal() == IDOK)
   {
      ;
   }
}
```

An instance of `CFileDialog` is created by its constructor with one parameter `TRUE`, indicating to create a `File Open` dialog box. If it is set to `FALSE`, the `File Save As` dialog box will be constructed. Save the change, compile the project, and press `F5` to start the program. When the application window shows up, click `File->Open` to bring up the `File Open` dialog box that looks similar to Figure 10.16. When browsing a folder as we usually do with Windows Explorer, we notice that files of different types are all shown in the file list box. Our purpose is to select specific types of data files (e.g., files with these extensions `.txt`, `.d*`, `.o*`, and `.n*`) so that they can be rendered on the screen. We thus naturally ask whether MFC provides filters for the `File Open` dialog. The answer is "YES." As indicated below, the `CFileDialog` constructor takes several optional (or default) parameters to control how the `File Open` dialog box behaves.

```
explicit CFileDialog(
   BOOL bOpenFileDialog,
   LPCTSTR lpszDefExt = NULL,
   LPCTSTR lpszFileName = NULL,
   DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
   LPCTSTR lpszFilter = NULL,
   CWnd* pParentWnd = NULL,
   DWORD dwSize = 0
);
```

The descriptions of these optional parameters are

bOpenFileDialog

Set to TRUE to construct a File Open dialog box or FALSE to construct a File Save As dialog box.

lpszDefExt

The default filename extension. If the user does not include an extension in the Filename edit box, the extension specified by `lpszDefExt` is automatically appended to the filename. If this parameter is `NULL`, no file extension is appended.

lpszFileName

The initial filename that appears in the filename edit box. If `NULL`, no filename initially appears.

dwFlags

A combination of one or more flags that allows you to customize the dialog box. The default bitwise value indicates "Hides the Read Only check box" and "Causes the Save As dialog box to generate a message box if the selected file already exists. The user must confirm whether to overwrite the file."

lpszFilter

A series of string pairs that specifies filters you can apply to the file. If you specify file filters, only the selected files will appear in the Files list box.

pParentWnd

A pointer to the file dialog-box object's parent or owner window.

dwSize

The size of the `OPENFILENAME` structure. This value is dependent on the operating system version. The default size of 0 means that the MFC code will determine the proper dialog box size to use based on the operating system version on which the program is run.

To control what types of files appear in the File list box, we need only the optional parameter `szFilters`. By the rule of optional parameters, however, we need to specify all default arguments that appear in front of `szFilters` as:

```
void CDisplayView::OnFileOpen()
{
   // TODO: Add your command handler code here
    TCHAR szFilters[]=
      _T("My Files (*.txt;*.d*;*.o*;*.n*)|*.txt;*.d*;*.o*;*.n*|")
      _T("All Files (*.*)|*.*||");
   CFileDialog fileDlg(TRUE, NULL, NULL, OFN_EXPLORER, szFilters);

   if (fileDlg.DoModal() == IDOK)
   {
      ;
   }
}
```

The `OFN_EXPLORER` flag tells the system to use the specified template to create a dialog box that is a child of the default Explorer-style dialog box. Two sets of string pairs are used as file filters. The first pair ensures that only those files whose suffixes are

```
.txt; *.d*; *.o*; *.n*
```

will be shown in the File list box. The second pair ensures that all files will be shown on the File list box. When executing the program, we can select either `My Files` or `All Files (*.*)` filter option as indicated below.

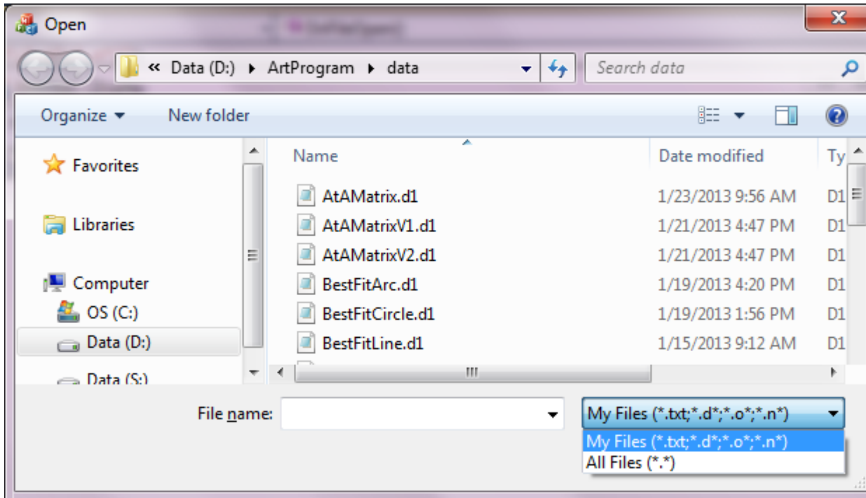


Figure 10.17: File open dialog with filter

After an instance of `CFileDialog` has been constructed, call the `DoModal` member function to display the above dialog box. The user can either enter the path and file name or browse the folder to find the desired file. `DoModal` returns a flag, indicating whether the user has selected the `OK (IDOK)` or the `Cancel (IDCANCEL)` button. If `DoModal` returns `IDOK`, we can use `CFileDialog`'s public member function `GetPathName` to obtain the selected filename and its path and then use such information to open the input file to read input data. Let's expand the implementation of `CDisplayView::OnFileOpen` to achieve this goal:

```
void CDisplayView::OnFileOpen()
{
   // TODO: Add your command handler code here
   TCHAR szFilters[]=
      _T("My Files (*.txt;*.d*;*.o*;*.n*)|*.txt;*.d*;*.o*;*.n*|")
      _T("All Files (*.*)|*.*||");
   CFileDialog fileDlg(TRUE, NULL, NULL, OFN_EXPLORER, szFilters);

   if (fileDlg.DoModal() == IDOK)
   {
      FILE      *infile=NULL;
      CString   pathName = fileDlg.GetPathName();

      _wfopen_s(&infile, pathName, _T("r"));
```

```
        if (infile)
        {
            int       i, n;
            m_gPoints.clear();
            fscanf_s(infile, "%d", &n);
            m_gPoints.resize(n);
            for (i=0; i<n; i++)
            {
                fscanf_s(infile,"%lf %lf",&m_gPoints[i].x,&m_gPoints[i].y);
            }
            fclose(infile);
        }

        m_bDrawCurve = TRUE;
        OnInitialUpdate(); // Triger to draw curves
    }
}
```

We use `_wfopen_s` instead of `fopen_s` because `pathName` is stored in `CSting` rather than `char` string. Save the changes, compile the project, and run the program. By browsing and opening, for example, `D:\ArtProgram\data\BestFitLine.d1`, we expect the data points and the best fit line to be drawn properly on the screen. It is, however, not the case due to the mismatch of two coordinate systems. Referring to the `DCToWorld` and `WorldToDC` methods implemented in the previous section, we assumed that both the world and device coordinate systems use a pixel as a unit. Furthermore, data points were generated by clicking the left mouse button while moving the cursor around the client windows, which indicates that all points are visible. The data points stored in a text file usually do not use a pixel as a unit, and also the world coordinate system may not necessarily overlap with the device coordinate system as shown below.

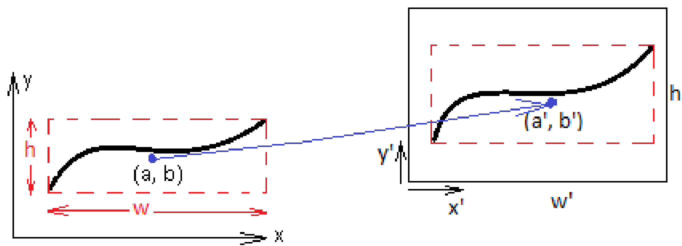


Figure 10.18: Mapping between world and screen coordinate systems

Therefore, two transformations are usually required to move a curve in the (x, y) to (x', y') coordinate system:

1. Scaling transformation: since two coordinate systems use different units, a scaling factor needs to be determined. Assume that the available draw area in the (x', y') coordinate system is $w' \times h'$. Our purpose is to maximally fit the image in the (x, y) coordinate system to the drawing area defined by $w' \times h'$. Assume the

maximum width and height of the image in (x, y) coordinate system is (w, h). Then,

$$\text{xScale} = w'/w \quad \text{and} \quad \text{yScale} = h'/h.$$

To leave some margin in the drawing area, we take

```
0.8 * min(xSvale, yScale)
```

as the scaling factor.

2. Translation: Compute the center of the image in (x, y) system and the center of the drawing area in the (x', y') coordinate system. Denoting two centers by (a, b) and (a', b'), the translation vector is defined as

$$\boldsymbol{v} = (a' - a, b' - b).$$

With the above two transformations in mind, we can now complete the implementation of `OnFileOpen` as follows:

```
void CDisplayView::OnFileOpen()
{
   // TODO: Add your command handler code here
   TCHAR szFilters[]=
      _T("My Files (*.txt;*.d*;*.o*;*.n*)|*.txt;*.d*;*.o*;*.n*|")
      _T("All Files (*.*)|*.*||");
   CFileDialog fileDlg(TRUE, NULL, NULL, OFN_EXPLORER, szFilters);

   if (fileDlg.DoModal() == IDOK)
   {
      FILE     *infile=NULL;
      CString  pathName = fileDlg.GetPathName();

      _wfopen_s(&infile, pathName, _T("r"));
      if (infile)
      {
         int      i, n;
         LONG     dx, dy;
         double   xmin, xmax, ymin, ymax, xScale, yScale;
         GPosition2d pt_mid_scrn, pt_mid_wd;
         CPoint      cPt;

         m_gPoints.clear();
         fscanf_s(infile, "%d", &n);
         m_gPoints.resize(n);
         for (i=0; i<n; i++)
         {
            fscanf_s(infile, "%lf %lf", &m_gPoints[i].x,
                                        &m_gPoints[i].y);
         }
```

```
        fclose(infile);

        // Determine the scaling factor
        xmin = xmax = m_gPoints[0].x;
        ymin = ymax = m_gPoints[0].y;
        for (i=1; i<n; i++)
        {
           if (xmin > m_gPoints[i].x)
              xmin = m_gPoints[i].x;
           else if (xmax < m_gPoints[i].x)
              xmax = m_gPoints[i].x;

           if (ymin > m_gPoints[i].y)
              ymin = m_gPoints[i].y;
           else if (ymax < m_gPoints[i].y)
              ymax = m_gPoints[i].y;
        }

        dx = m_clientRect.right - m_clientRect.left;
        dy = m_clientRect.bottom - m_clientRect.top;
        xScale = dx / (xmax - xmin);
        yScale = dy / (ymax - ymin);
        m_scale = 0.8 * min(xScale, yScale);

        cPt.x = m_clientRect.left + dx / 2;
        cPt.y = m_clientRect.top + dy / 2;
        DCToWorld(cPt, pt_mid_scrn);
        pt_mid_wd.x = xmin + 0.5 * (xmax - xmin);
        pt_mid_wd.y = ymin + 0.5 * (ymax - ymin);

        m_translation.x = pt_mid_scrn.x - pt_mid_wd.x;
        m_translation.y = pt_mid_scrn.y - pt_mid_wd.y;
        m_bDrawCurve = TRUE;
        OnInitialUpdate(); // Triger to draw curves
      }
   }
}
```

We also need to modify the `WorldToDC` method to take the translation into consideration:

```
void CDisplayView::WorldToDC(GPosition2d &gPoint, CPoint &cPoint)
{
   GPosition2d pt;
   pt.x = gPoint.x + m_translation.x;
   pt.y = gPoint.y + m_translation.y;
   cPoint.x = (int)(pt.x * m_scale);
   cPoint.y = (int)(m_clientRect.bottom - pt.y * m_scale);
}
```

We are now ready to play with the windows-based application `Display`. When the `BestLine` icon is active, we can either click mouse buttons to generate a best fit line or open an existing data file (e.g., `BestFitLine.d2`) to display the data points as tiny red circles and the best fit line in black color as shown in figure 10.19. Similarly, when the `BestCircle` or `BestArc` icon is active, we can open corresponding data files to visualize the input and output results.

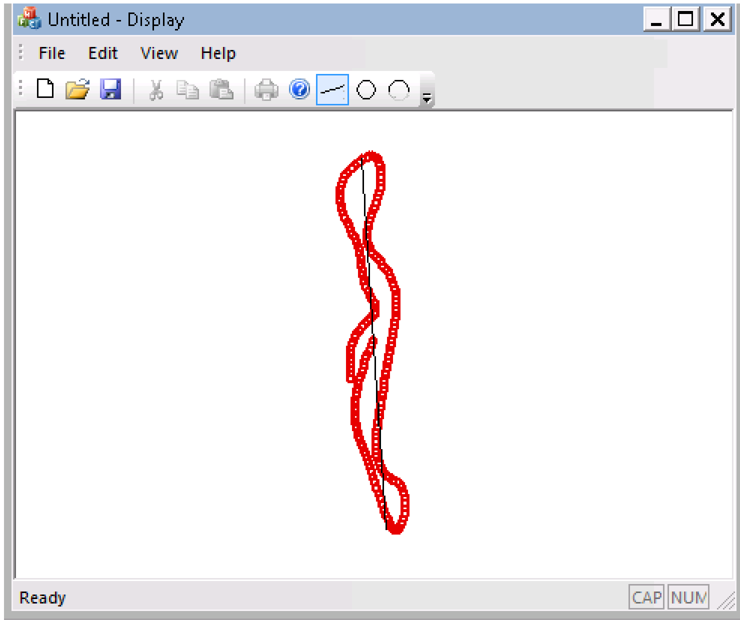


Figure 10.19: Best fit line to customer data points

10.6 Summary

We have seen how the MFC Application Wizard and Class Wizard simplify our work in developing a windows-based application. Though a very simple application program, it does provides hands-on experience in Windows programming and gives a glimpse of how sophisticated CAD systems are developed. Most GUIs in windows-based CAD systems have historically been built with native Win32 APIs and MFC. Therefore, the hands-on experience and MFC knowledge gained in this chapter will certainly help readers not only shorten the learning curve when working in a software company that develops windows-based applications, but also develop their own windows-based applications.

Coordinate transformations play an essential role in data visualization. Geometric 2D and 3D transformations used in computer graphics are mappings from one coordinate system to the other. They play a central role in model construction and visualization. For example, tools such as rotating, zooming, and mirroring an image found in most CAD systems are all based on geometric transformations. In this chapter, we studied only limited 2D transformations that are necessary to map images in the

world coordinate to the device coordinate (e.g., the client window of our `Display` application). General discussions about 2D and 3D transformations can be found in most computer graphics books. Readers can also find such information in my online book, *Java Graphics Programming*, at www.infogoaround.org. After becoming comfortable with coordinate transformations, readers can expand this Display tool to include commonly-used commands such as `zoom-in`, `drag`, `rotate`, etc. as shown below. This is the tool I developed at work to visualize and manipulate geometric entities saved in the internal format known only by the internal math functions.

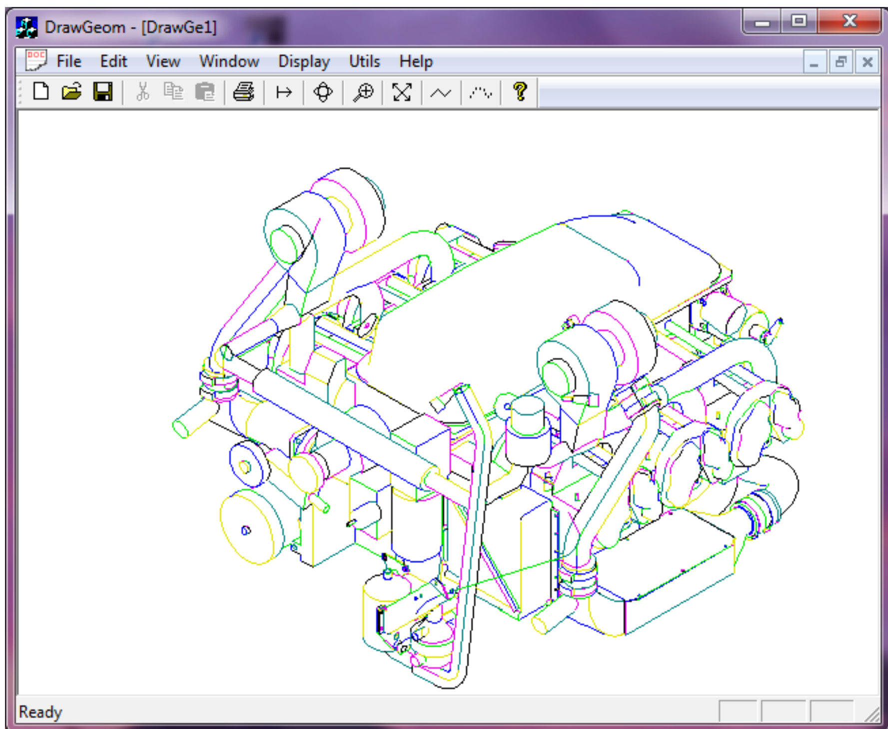


Figure 10.20: Display tool with drag, rotate, zoom-in, and fit commands.

It should be pointed out that the transformations we used in this chapter did not take into consideration the so-called *pixel aspect ratio*, which is the ratio between the width and height of each pixel. We assumed that most modern computers have squared pixels. If this is not the case, the aspect ratio may be critical when users try to draw a figure with symmetric shape such as circles and squares.

Microsoft is constantly evolving its core technologies. To keep up the pace, developers in software industry are constantly learning and renewing their knowledge and skills. Win32 and MFC have traditionally been the primary choice for windows-based applications developed with unmanaged C/C++. With the introduction of Microsoft .Net framework,

developers are implementing applications in managed code such as C#and .Net VB. For this reason, Windows Forms and Windows Presentation Foundation are gaining popularity in developing windows-based applications.

In our example, display of line, circle, and arc is rendered via Microsoft's Graphics Device Interface. The GDI provides functions for drawing points, lines, rectangles, polygons, ellipses, bitmaps, and text. For 3D rendering, Microsoft DirectX is preferred over GDI as its *High Level Shading Language (HLSL)* is capable of doing vertex, geometry, and pixel (or fragment) shading. DirectX has also a well-defined set of graphics processing unit (GPU) functionality.