Introduction
Overview
New architecture
Upgrading

How to...
Build the class library
Add CGraph to a project
Create a simple graph
Get and set properties
Set array properties
Use property enumerators
Save and restore property settings
Handle events
Print a graph (AFX applications)

Reference
Files you need to distribute

Whatever your Windows development environment, Graphics Server provides you with the software components necessary to take advantage of our graphing engine. In addition to its traditional DLL API, Graphics Server includes a VBX, 16- and 32-bit OCXs, 16- and 32-bit property DLLs, and the CGraph C++ class library (16- and 32-bit compatible).

Thanks to the flexibility of C++ you are in a position to use most of these components. So which one should you use? We recommend the CGraph class library over the other interfaces because it gives the C++ programmer an optimal combination of power and ease-of-use.

With CGraph, you control your graphs using a simple property syntax very similar to what you would use if you were programming with the Graph control. For example, to draw a horizontal, 3D bar graph of the values 1 through 5, you need only these lines of code:

```
m_Graph.NumSets = 1;
m_Graph.NumPoints = 5;
m_Graph.GraphType = CGraph::bar3d;
m_Graph.GraphStyle = CGraph::horizontal;
for ( int i = 0; i < 5; i++)
    m_Graph.GraphData[i] = i + 1;
m_Graph.DrawMode = CGraph::draw
```

CGraph shares the same set of properties as the VBX and the OCXs. For a full description of graph properties, see the manual for the *Graph Control*.

The version of CGraph in this release is a departure from the one issued with 4.0x. In the previous release, we supplied five component libraries in source form, which you then had to compile to match your application. It was also highly dependent on MFC.

This version reduces the complexity of the earlier release by several orders of magnitude. Equally important, it reduces dependencies on MFC and other model and compiler-dependent factors. Much of the code formerly supplied has been embodied in property-based support DLLs, GSPROP16.DLL and GSPROP32.DLL. What remains is a "shell" providing a class wrapper around these DLLs.

The new approach offers several important advantages:

- A simplified structure makes CGraph easier to build and use.

  You now build only one library rather than five. Graphs are controlled using a properties and events identical to the VBX and OCX interface.

- Event handling has been significantly enhanced.

  Immediately after an event, you can now read a new set of properties that contain event parameters such as HitPoint and HitSet, making event-handling as easy to implement as it is in Visual Basic.

- Support for more compilers.

  The new version can be compiled and used with either Microsoft Visual C++ or Borland C++. It can be modified for use with other compilers.

- Reduced dependency on application frameworks.

  CGraph is no longer derived from a base window class (CWnd in Microsoft Visual C++, TWindow in Borland C++). It is now a free-standing class. The new version relies on MFC/OWL only for string, point, and size classes.

For those committed to maintain applications using the old CGraph, an updated version will be supplied on request. However, all new users of CGraph are strongly advised to use the version provided here.

CGraph is now a free-standing class. It is no longer derived from a base window class (CWnd in MS VC++, TWindow in Borland C++). This is primarily in order to make it easier to build and more adaptable for use with compilers other than VC++.

Apart from changing how the library is built, the new architecture changes some aspects of how CGraph objects are implemented in your code.

Window handle

CGraph has a public member, m_hWnd, which contains the handle of its associated window--the one created when you call CGraph's Create function. CGraph's default value--what you get when you refer simply to the unqualified name of a CGraph object--is an HWND of value m_hWnd.

If, in the past, you have used CWnd functions to manipulate a CGraph object, you will have to replace them with direct calls to the equivalent Windows function.

For example,

```
MoveWindow(  0, 0, cx, cy);
```

becomes

```
::MoveWindow( m_Graph, 0, 0, cx, cy, FALSE);
```

Enabling the toolbar

One of the significant benefits from basing CGraph on GSPROPxx.DLL is that you no longer have to take any special action to make CGraph's in-graph toolbar enabled in an MFC-based application. Previously you were obliged to respond to the OnCmdMsg event; otherwise the toolbar buttons were grayed-out. Now you just set the Toolbar property to On.

Access to the low-level API

CGraph version 4.0x included a set of member functions that gave access to the low-level Graphics Server API. In order to provide a measure of backward compatibility, those functions are implemented in CGraph version 4.5x. However, they are no longer documented and may not be included in future releases of CGraph.

Before you can use CGraph in a project, you must compile the library file. Your Graphics Server installation includes projects that you can use for this purpose:

..\Graphcls\Source\Borland

16-bit	CGRAPH.IDE
32-bit	CGRAPH32.IDE

..\Graphcls\Source\Msvc

16-bit	CGRAPH.MAK
32-bit	CGRAPH32.MAK

When the library is compiled, the header files GSW.H and GSPROP.H must be available to the preprocessor. These files are installed in the directory ..\C\Lib. Make sure you have them in your INCLUDE path.

Remember to build the library using the same compiler options as your application:

- Memory model (applies only to 16-bit compilers)
- Application vs DLL (applies only to 16-bit compilers)
- Release vs Debug mode
- Static vs DLL foundation classes

We recommend you use LARGE model.

To use the CGraph class in your application, you must edit your project and its files as follows:

- Put an #include for CGRAPH.H in your source.

  You can find this file in the directory ..\GRAPHCLS.

- Make CGRAPH.H, CGPROP.H, and CGARRAY.H available to the preprocessor by placing them on your INCLUDE path.

  CGRAPH.H further includes CGPROP.H and CGARRAY.H, so all three must be available to the preprocessor. You can find them in the directory ..\GRAPHCLS.

- Make CGRAPH.LIB, GSPROPxx.LIB, and GSWDLLxx.LIB available to the linker by placing them on your LIB path.

  You can find GSPROPxx.LIB and GSWDLLxx.LIB in the directory ..\C\Lib. For instructions on how to build CGRAPH.LIB, see Building the class library.

Working example applications can be found in ..\GRAPHCLS \SAMPLES \BORLAND and ..\ GRAPHCLS \SAMPLES \MSVC.

To create a graph....

- Include a CGraph object in one of your application's view classes.

```
CGraph m_Graph;
```

- Create a window for the graph by calling CGraph's Create function in your view's OnCreate event.

```
m_Graph.Create( NULL, WS_CHILD | WS_VISIBLE, rect, m_hWnd, 1);
```

- Determine the attributes of the graph by setting its properties.

```
CRect rect;
GetClientRect( &rect);

m_Graph.Create( NULL, WS_CHILD | WS_VISIBLE, rect, m_hWnd, 1);
m_Graph.RunTime = TRUE;
m_Graph.Toolbar = CGraph::runTime;
m_Graph.Hot = CGraph::on;
m_Graph.SDKMouse = CGraph::on;
m_Graph.GraphData[0][0] = 1;
m_Graph.GraphData[0][1] = 2;
m_Graph.GraphData[0][2] = 3;
m_Graph.GraphData[0][3] = 4;
m_Graph.GraphData[0][4] = 5;
m_Graph.GraphTitle = "Hello Graph";
m_Graph.GraphType = CGraph::bar2d;
m_Graph.DrawMode = CGraph::blit;
```

CGraph's properties are integers, reals, or strings (scalar and one- or two-dimensional arrays). They are implemented as classes in CGraph such that when you refer to them their values are set/obtained from GSPROPxx.DLL.

Technically, we have achieved this by overloading their assignment and cast operators. Our method imposes a few restrictions on the use of CGraph properties in C++ expressions, since the more sophisticated operators are not implemented.

For example, rather than use

```
m_Graph.GraphData[0][0]++;
```

you must use

```
m_Graph.GraphData[0][0] = m_GraphData[0][0] + 1;
```

For a complete description of graph properties and settings, see the manual for the *Graph Control* or the control's help, GRAPHPPD.HLP.

Array properties play an important part in the specification of a graph. As the name suggests, an array property is a collection of identically typed property elements. You refer to the elements collectively by their array property name, and select them individually by means of a numeric array property index. Both one and two-dimensional array properties are in use. The best examples are the data sets, labels and legends that the graph displays.

All the array properties have zero-based indexes. Valid indexes range from zero to one less than the number of elements, in either dimension. This may seem self-evident, but beware of it if you have come from programming the Graph control in Visual Basic. In Visual Basic some of the array property indexes are one-based.

Two two-dimensional array properties, called GraphData and XPosData, specify the Y and, optionally, the X values portrayed in the graph. A third two-dimensional array property, called ExtraData, optionally provides additional information pertinent to certain types of graph. In Graph control parlance, the primary dimensions of these arrays represent the different data sets of the graph and the secondary dimensions represent the different points.

To give you a picture of the organization of the data in the two-dimensional array properties, this is how an array for a graph of sales and costs for a three-monthly period would appear if you defined and initialized it in standard C++:

```
const int NumSets = 2;
const int NumPoints = 3;
double GraphData [NumSets][NumPoints] = {
    /* month                1       2       3 */
    /* sales data set */    100,    150,    250,
    /* costs data set */     75,     75,     80
};
```

The array property classes provide operator member functions to simplify the tasks of dimensioning and indexing the array property members.

The overloaded function call operator sets the dimensions of an array property. The syntax depends on whether the array property has one or two dimensions. For example:

```
graph.GraphData(2,12);
graph.LabelText(12);
```

The first example dimensions the GraphData array to contain two arrays of twelve elements each. The dimensions usually reflect the number of data sets and the number of points in the graph. The second example dimensions the LabelText array to contain twelve elements.

The overloaded index operator indexes a given element of an array property. The operators support one and two-dimensional array indexing. Array indexes range from zero to one less than the number of elements. The index operator returns a reference for the given element of the array. Then the overloaded assignment operator provides a natural syntax for getting and setting the value of the element. Indexed array property elements can appear in left and right-hand side expressions. For example:

```
CGraph graph;
...
graph.GraphData[0][0] = 100;
graph.GraphData[0][0] = 2 * graph.GraphData[0][0];
ASSERT(graph.GraphData[0][0] == 200);
...
graph.LabelText[5] = "June";
graph.LabelText[5] = graph.LabelText[5] + " 1993";
ASSERT(graph.LabelText[5] == "June 1993");
```

The array property classes also provide overloaded assignment operators for setting the contents of an entire array in one operation. One form of assignment sets all the elements of the array to the same value. For example:

```
CGraph graph;
...
graph.GraphData(2,3);
graph.GraphData = 100;

#ifdef _DEBUG
for ( UINT s = 0; s < 2; ++s ) {
    for ( UINT e = 0; e < 3; ++e ) {
        ASSERT(graph.GraphData[s][e] == 100);
    }
}
#endif
```

Another form sets the elements of the array by copying them from an external array. For example:

```
CGraph graph;
double GraphData [2][3] = {
    /* point             1       2       3 */
    /* data set 1 */    100,    150,    250,
    /* data set 2 */     75,     75,     80
};
...
graph.GraphData(2,3);
graph.GraphData = &GraphData[0][0];
#ifdef _DEBUG
for ( UINT s = 0; s < 2; ++s ) {
    for ( UINT e = 0; e < 3; ++e ) {
        ASSERT(graph.GraphData[s][e] == GraphData[s][e]);
    }
}
#endif
```

The operators do not provide for array assignments in right-hand side expressions.

Most operator member functions return a reference either to themselves or to a temporary object of another property class, allowing compound operations to be performed. For example:

```
graph.GraphData(2,12) = 0;
```

In this example a single statement dimensions and initializes the GraphData array property.

There is a close relationship between the dimensions of the data array properties and the non-array members NumSets and NumPoints. NumSets and NumPoints alone determine the number of data sets and the number of points that appear in a graph. They also control how many elements you may store in the data array properties. For the GraphData. XPosData, and ExtraData arrays the maximum number of elements is the product of NumSets and NumPoints. For the remaining data array properties it is the maximum of NumSets and NumPoints.

The dimensions you give to any of the data array properties do not determine the appearance of the graph, nor do they override any settings imposed by NumSets and NumPoints on the number of elements you can store in a given data array. For this reason, it is important you set NumSets and NumPoints to their proper values before attempting to set the dimensions and store values in the data array properties.

The following example shows a correct approach to setting up the data arrays for a graph:

```
CGraph graph;
const int NumSets = 2;
const int NumPoints = 3;
double GraphData [NumSets][NumPoints] = {
    /* month                1       2       3 */
    /* sales data set */    100,    150,    250,
    /* costs data set */     75lo    75,     80
};
LPCSTR LegendText [NumSets] = { "Sales", "Costs" };
LPCSTR LabelText [NumPoints] = { "Jan", "Feb", "Mar" };
...
graph.NumSets = NumSets;
graph.NumPoints = NumPoints;
graph.GraphData(NumSets, NumPoints) = &GraphData[0][0];
graph.LegendText(NumSets) = LegendText;
graph.LabelText(NumPoints) = LabelText;
```

Some numeric properties require you to supply one of a pre-determined range of values. Such a property is referred to as an *enumerated property*. The C++ language defines *enumerators* as the way of specifying names for a known and reasonably limited set of values that you can assign to a given type of variable.

The Graph class library defines enumerators for most, if not all, of the enumerated properties of a Graph control. Proper use of the enumerators will help to clarify the meaning of enumerated property settings in your application code. For example, consider the following code, which shows two ways to specify the properties of a scatter graph. Both make the same property settings. The first uses the enumerators while the second uses just numeric constants.

```
CGraph graph;
...
graph.GraphType = CGraph::scatter;
graph.GraphStyle = CGraph::scatterCurve | CGraph::scatterSymbols;
graph.SymbolSize = 125;
graph.ThickLines = CGraph::on;
graph.CurveType = CGraph::polynomial;
graph.CurveOrder = 1;
graph.CurveSteps = 50;
graph.DrawMode = CGraph::draw;
...
graph.GraphType = 9;
graph.GraphStyle = 1 | 2;
graph.SymbolSize = 125;
graph.ThickLines = 1;
graph.CurveType = 0;
graph.CurveOrder = 1;
graph.CurveSteps = 50;
graph.DrawMode = 2;
```

As you can see, the use of property enumerators in the first example serves to make the intention of the code much clearer.

The CGraph class provides graph template files as a mechanism for saving a graph's property settings at design time and restoring them at run time, or for saving changes made during one run of a program and restoring them at the next.

Graph template files can store all property settings for a graph, including data arrays. A single template file can store settings for multiple graphs.

By default, template files take the file extension .GSP. They are in ASCII text format and are structured exactly like an INI file. Each section of a template is headed by the name of a graph enclosed by brackets. Below the name is a list of property names and their settings:

```
[My Graph Name]
ColorData=32
GraphCaption=Box-whisker graph
GraphStyle=2
GraphType=17
LabelText=Mon~Tue~Wed~Thu~Fri
NumPoints=5
NumSets=7
...
```

If you store data in a template, you should keep in mind that the size of the file will grow in proportion to the number of points in a graph. There is a limit to how much data a single template can store--the product of NumPoints and NumSets cannot exceed 500. When this limit is reached, no further graph information will be saved in the file. You can overcome this limitation either by choosing not to store data in the template or by storing only one graph definition per file.

Template-related properties

Before you save or restore properties in a template file, you need to specify which file to use. You do this by setting the GraphFile property, which accepts a string containing a file name. The setting can also include a path to the file; if it does not, the current working directory is assumed.

Sections within a template file are referenced by the GraphName property. If the GraphName property is empty when you save a graph definition, Graphics Server will use the default name Graph.

Graph definitions are saved and restored by means of the DrawMode property, which has three settings for this purpose:

Setting	Action
9	Restores properties of a previously saved graph.
10	Saves the current property settings to a graph template file. Only the properties defining the appearance of the graph, including any text, are saved.
11	Saves the current property settings, including data (ErrorBar, ExtraData, GraphData, OverlayData, XPosData and ZPosData).

If you have enabled events by setting the Hot or SDKMouse properties, your application will be sent messages notifying it of these events:

```
enum {
        pressEvent = WM_USER + 1,
        trackEvent,
        pressEvent,
        hitEvent,
        hotHitEvent
    };
```

By default event notification messages will be sent to CGraph's parent window. The message numbers will be in the range WM_USER + 1 to WM_USER + 5.

CGraph contains a set of event parameter properties through which you can obtain information about an event immeditely after it occurs.

Event	Property	Reports
press/track	HitX	X coordinate
press/track	HitY	Y coordinate
press/track	HitDataX	X coordinate (your data units)
press/track	HitDataY	Y coordinate (your data units)
press	HitStatus	Mouse status
hit	HitRegion	Region number
hotHit	HitPoint	Point number
hotHit	HitSet	Set number
hotHit	HotButton	Mouse button status

Example

Below is an extract from the sample application, CGDEM.

```
BEGIN_MESSAGE_MAP(CCgdemView, CView)
        //{{AFX_MSG_MAP(CCgdemView)
        ON_WM_CREATE()
        ON_WM_SIZE()
        ON_MESSAGE( CGraph::hotHitEvent, OnHotHit)
        ON_MESSAGE( CGraph::pressEvent, OnPress)
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

LRESULT CCgdemView::OnHotHit(WPARAM wParam, LPARAM lParam)
{
    char message[20];
    sprintf( message, "Point %d", (int) m_Graph.HitPoint);
    MessageBox( message);
    return 0;
}

LRESULT CCgdemView::OnPress(WPARAM wParam, LPARAM lParam)
{
    char message[40];
    sprintf( message, "DataX = %f, DataY = %f", (double) m_Graph.HitDataX, (double)
   m_Graph.HitDataY);
    MessageBox( message);
```

```
    return 0;
}
```

Overview
Basic printing
Controlling print size and position
Selecting color or monochrome

In the Microsoft application framework (AFX), the printing of a form is a function of the view class. The member function in question comes from the CView base class and is named OnPrint. AFX calls OnPrint in response to a print command from the menu or toolbar.

The default implementation of OnPrint prints by calling the standard OnDraw member function and passing it the printer device context. One aim of this scheme is that OnDraw can concentrate on rendering a logical image of the form, not on the specific target device, which may be the normal view window, a print preview window or the printer.

An OnDraw member function can be found in every view class. The default OnDraw in your view class should look something like this:

```
void CPrintInfoView::OnDraw(CDC* pDC)
{
  CPrintInfoDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);
}
```

This is where you will want to add code to control printing for a GGraph member of your view. (For information on adding a CGraph member function , see Creating a simple graph.)

Printing is done by setting CGraph's DrawMode property. Although DrawMode is technically a graph property, its role is like that played by a member function. Setting DrawMode to the enumerated constant CGraph::print is like calling a member function of the object to print the graph.

To enable printing, set the DrawMode property of the CGraph member object somewhere in the OnDraw member of the view. Remember, though, that OnDraw is called to display your view on-screen as well as print it. You only want to set the graph to print when the view is printing, and you can do this by making it conditional on the IsPrinting member of the device context object. For example:

```
void CPrintInfoView::OnDraw(CDC* pDC)
{
  CPrintInfoDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  if ( pDC->IsPrinting() ) {
    m_Graph.DrawMode = CGraph::print;
  }
}
```

Simply setting DrawMode to print usually isn't sufficient. The graph appears on a separate page from other printed output, and at a default size and aspect ratio. The reason the graph is printed separately is that it is printing on a local device context, rather than the one the view is using, and Windows always arranges that output to different device contexts is kept separate, even if the eventual output device is the same.

What is needed is to tell the graph to print on the same device context as the rest of the view. You can do this through the PrintInfo property.

PrintInfo is an array property that controls both the device context and the size and position for printing. The elements of the PrintInfo array have the following meanings:

Index	Description
0	The handle of the printing device context in non-AFX applications. Set this element to zero and see elements 12 and 13 for how to pass a pointer to an MFC device context object.
1	The X position of the top left corner of the graph as it appears on the page, expressed in the scale units of the printer.
2	The Y position of the top left corner of the graph as it appears on the page, expressed in the scale units of the printer.
3	The width of the graph as it appears on the page, expressed in the scale units of the printer.
4	The height of the graph as it appears on the page, expressed in the scale units of the printer.
5	The X coordinate of the top left corner of the page.
6	The Y coordinate of the top left corner of the page.
7	The width of the page in scale units.
8	The height of the page in scale units.
9	Selects preservation of graph aspect ratio on printing.
10	Selects portrait or landscape orientation. Not used in this context.
11	Selects printing at full page-size. Not used in this context.
12	The low-order word of the pointer to the MFC device context object. This element is only for use in AFX applications.
13	The high-order word of the pointer to the MFC device context object. This element is only for use in AFX applications.

To control the size and position of the graph on a printed page, set values in the PrintInfo array before setting DrawMode to print the graph.

```
void CPrintInfoView::OnDraw(CDC* pDC)
{
  CPrintInfoDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  if ( pDC->IsPrinting() ) {
  #ifdef WIN32
    m_Graph.PrintInfo[0] = 0;
    m_Graph.PrintInfo[12] = LOWORD( pDC );
    m_Graph.PrintInfo[13] = HIWORD( pDC );
  #else /* ndef WIN32 */
    m_Graph.PrintInfo[0] = (int) pDC->m_hDC;
```

```
    m_Graph.PrintInfo[12] = 0;
    m_Graph.PrintInfo[13] = 0;
  #endif /* ndef WIN32 */

    /* set the page size in mm */
    m_Graph.PrintInfo[5] = 0;
    m_Graph.PrintInfo[6] = 0;
    m_Graph.PrintInfo[7] = pDC->GetDeviceCaps( HORZSIZE );
    m_Graph.PrintInfo[8] = pDC->GetDeviceCaps( VERTSIZE );

    m_Graph.DrawMode = CGraph::print;
  }
}
```

The graph now prints and previews in the same way as the rest of the view. By default the graph appears at the top of the page at a similar size and aspect ratio to how it appears on-screen. You can modify this behavior with some of the other elements of PrintInfo. Setting the position and size of the graph on the page is the job of elements 1 to 4. For example, to print the graph 102 mm wide by 76 mm high at the bottom right corner of the page.

```
void CPrintInfoView::OnDraw(CDC* pDC)
{
  CPrintInfoDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  if ( pDC->IsPrinting() ) {
    /* set the printing device context */
  #ifdef WIN32
    m_Graph.PrintInfo[0] = 0;
    m_Graph.PrintInfo[12] = LOWORD( pDC );
    m_Graph.PrintInfo[13] = HIWORD( pDC );
  #else /* ndef WIN32 */
    m_Graph.PrintInfo[0] = (int) pDC->m_hDC;
    m_Graph.PrintInfo[12] = 0;
    m_Graph.PrintInfo[13] = 0;
  #endif /* ndef WIN32 */

    /* set the page size in mm */
    m_Graph.PrintInfo[5] = 0;
    m_Graph.PrintInfo[6] = 0;
    m_Graph.PrintInfo[7] = pDC->GetDeviceCaps( HORZSIZE );
    m_Graph.PrintInfo[8] = pDC->GetDeviceCaps( VERTSIZE );

    /* set the size and position of the graph on the page */
    m_Graph.PrintInfo[3] = 102;
    m_Graph.PrintInfo[4] = 76;
    m_Graph.PrintInfo[1] = m_Graph.PrintInfo[7] - m_Graph.PrintInfo[3];
    m_Graph.PrintInfo[2] = m_Graph.PrintInfo[8] - m_Graph.PrintInfo[4];

    m_Graph.DrawMode = CGraph::print;
  }
}
```

For additional information on the options for sizing and positioning the graph on the page, refer to the *Graph Control* reference manual.

The graph size and position above is given in mm because those are the scale units given for the printer. Change the scale units if you want to use different units for the graph. For example, to print the graph 4 inches wide by 3 inches high instead, expressed in 0.01 inch (LOENGLISH) units, set the page size in the same units.

```
/* set the page size in LOENGLISH */
m_Graph.PrintInfo[5] = 0;
m_Graph.PrintInfo[6] = 0;
m_Graph.PrintInfo[7] = MulDiv( pDC->GetDeviceCaps( HORZSIZE ), 1000, 254 );
m_Graph.PrintInfo[8] = MulDiv( pDC->GetDeviceCaps( VERTSIZE ), 1000, 254 );

/* set the size and position of the graph on the page */
m_Graph.PrintInfo[3] = 400;
m_Graph.PrintInfo[4] = 300;
m_Graph.PrintInfo[1] = 0;
m_Graph.PrintInfo[2] = 0;
```

A third property that can be used to control printing is PrintStyle. This property enables you to say if you want the graph printed in color or monochrome and also if you want a border around the graph on the page. Again, set PrintStyle before setting DrawMode to print the graph. For example:

```
m_Graph.PrintStyle = CGraph::mono | CGraph::border;
m_Graph.DrawMode = CGraph::print;
```

When you distribute an application that uses Graphics Server, you must include several supporting files. Which supporting files you distribute depends on whether your application is 16- or 32-bit.

With 16-bit applications distribute

GSW16.EXE	Graphing engine
GSWDLL16.DLL	Kernel API DLL
GSWAG16.DLL	AutoGraph DLL
GSPROP16.DLL	Property DLL

With 32-bit applications distribute

GSW32.EXE	Graphing engine
GSWDLL32.DLL	Kernel API DLL
GSWAG32.DLL	AutoGraph DLL
GSPROP32.DLL	Property DLL