VisualAge® C++ Professional for AIX®

IBM® Open Class™: 2D Graphics

Version 5.0

Note!
Before using this information and the product it supports, be sure to read the general information under "Notices" on page v.

Edition Notice

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Contents

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

AIX
AS/400
DB2
CICS
C Set ++
IBM
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architechture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries licensed exclusively through X/Open Company Limited.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered tradmark of Ziff Communicatoins Company and is used by IBM Corporation under license

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.

Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

Chapter 1. 2D Graphics Overview

The 2D Graphics classes let you create, manipulate, and render two-dimensional graphics. You do not need to know linear algebra or any other advanced mathematics to effectively use these classes. All advanced mathematics is implemented for you. You can extend most of these classes to add your own functionality by deriving from the abstract base classes, implementing the pure virtual functions, and adding whatever behavior you need.

The 2D Graphics Framework classes can be used to perform the following:

- Creation of graphical user interface (GUI) elements and decorations.
- Manipulation of pixel-based images, screen captures and so on.
- Creation of complex 2D graphics composed of groups of simple graphics used for modeling.

The 2D Graphics Framework is grouped into the following main categories:

- Attribute and Bundle
- Geometry and Graphics
- Modeling
- Rendering
- Transformation

Attribute and Bundle Classes
The bundle classes allow you to selectively change drawing attributes from those set for the current graphic port. The bundle classes derive from IAttributeState, an abstract base class. The Open Class Library provides you with the IGrafBundle class that let you modify drawing attributes such as the following:

Attribute	Description
fill	The color that spreads over the interior of the geometry.
frame	The color and the line that outlines the border of the geometry.
transfer mode	The style by which your application draws a fill, a frame, or an image onto the screen.

You also have access to other specialized bundle classes like IFillBundle, IFrameBundle, and IFillAndFrameBundle. These classes provide less functionality than IGrafBundle in order to simplify and streamline your application.

Geometry and Graphics Classes
The 2D Graphics Framework provides geometry classes and corresponding *graphics classes*. Both of these classes provide functionality for different areas of graphics programming:

- Graphics classes are derived from IMGraphic. They can own a geometry class and can have functions to transform and otherwise manipulate the geometry.
- Geometry classes have no common parent class. They have functions that let you manipulate the geometry, but are not intended to have classes derived from them.

The geometry classes let you describe geometric shapes and images. For example, you use the geometric classes to specify the end points of a line, the vertices of a polygon, or the color table for an image.

The graphics classes let you manipulate your geometric shapes and images with the high-level functionality of the IMGraphic class. The graphics classes let you do the following:

- Access attribute and bundle data
- Render your shapes and images to a graphics port (draw your graphic onto the screen)
- Detect whether the graphic has been selected by the user (hit detection)
- Transform the graphic (rotations, scaling, translations)
- Determine bounding rectangles (the smallest rectangular area your graphic can occupy)
- Combine several graphics into a group for common operations

Generally, you would create an object with a geometry class that describes your shape or image. To use this geometry class object in your application, you would create an object with a corresponding graphics class, using that geometry class object as an argument.

For example, you can create an IGLine2D object to describe a line segment by specifying its two end points. You can then create an ILine object, using your IGLine2D object as an argument. With this ILine object you can render your line, determine whether your line intersects a specific rectangle, or change one of your line's attributes.

The geometry classes serve as the foundation for creating graphics. Use these classes to describe pure geometric shapes. The geometry classes provide many ways to construct a shape because you construct the geometry first, and then use it to construct the graphic. The geometry classes are grouped into the following categories:

Category	**Description**
Computational geometry	Computational geometry classes, including points and collections of points, are used for 2D geometric calculations, and as building blocks for other 2D geometry classes.
Path geometry	Path geometry classes describe shapes of an unenclosed path. These shapes cannot be filled. Some geometric shapes, such as point, line, and non-enclosing polyline and curves, are of path geometry.
Area geometry	Area geometry classes describe shapes of an enclosed path. These shapes can be filled. Enclosing geometric shapes such as rectangle, circle, ellipse, polygon, and enclosing curves (loops) are of area geometry.

Category	Description
Special area geometry	Special area geometry classes, used primarily for clipping, make area geometries by concatenating different geometric shapes. These classes let you perform constructive area geometric operations through boolean operations such as union, intersection, and subtraction.
Image geometry	Image geometry classes handle and manipulate 2D images (bitmaps). These classes convert basic geometry from coordinate units to pixels. Two-dimensional attribute bundles have special properties for determining the color of an image (transfer modes) and for specifying filters to control the appearance of an image when it has been enlarged or shrunk.

For immediate mode rendering, a user can create geometry and graphics attributes and render directly to the graphics rendering/display system, the rendering pipeline, without using the modeling classes. However, IMGraphic modeling classes are more convenient for retained mode rendering where a user needs to attach attributes and transformation functions to a geometry.

Modeling classes
Modeling classes associate geometries with graphic attributes, and transformations for simple graphics and hierarchical collections of graphics. IMGraphic is the abstract class of IGraphicGroup and its related classes. Use these classes to create complex graphics models (scenes), and extend these classes to enable graphics metadata storage. The virtual draw function lets you render the geometry of a drawable graphic with attributes and transforms applied.

Rendering classes
The rendering pipeline, controlled through IGrafPort, performs imaging operations. You associate a graphic and its state information, including its attribute state, transformation matrix, and clipping state, with a grafport. The grafport renders the graphic from this state information.

IGrafPort is associated with IGrafDevice. For performance reasons, IGrafDevice is at the level of abstraction of platform/device-specific code. The concrete subclass of IGrafDevice is platform/device specific. IGrafDevice encapsulates the non-portable code: the device-dependent layer is completely hidden from the client.

Transformation classes
Transformation classes, specifically IGrafMatrix, define transformation matrices for manipulating graphics geometry and modeling. Transformation functions let you rotate, translate, scale, or apply a general transformation to the geometry without using matrix multiplication.

Useful information
See also IGraphicText in the text display classes to create and manipulate text as a graphic.

See also the 2DCombo sample in the code samples directory.

The Open Class graphics classes use the following naming conventions to help you distinguish graphics classes from geometry classes:

- Graphics class names begin with the letter "I" and followed by the name of the graphic (for example, ILine).
- Geometry class names begin with the letters "IG" and followed by the name of the geometry (for example, IGLine2D).

This guide describes the classes and data types common to the graphics classes. Member functions throughout the framework use some terms to indicate a particular function's role in storage management. These terms are provided here for your convenience while reading this guide.

Functions that allocate and take responsibility for storage, or return pointers to instances, have special names and abide by the following guidelines:

Convention	Description
Create	When you see "Create" in the name, the function makes and returns a new instance that the caller must delete.
Copy	When you see "Copy" in the name, the function copies an existing instance and the caller must delete that copy.
Adopt	When you see "Adopt" in the name, the function accepts an instance and is responsible for deleting it. Adopt functions that cannot have "Adopt" in their names (for example, constructors) prefix it to the name of the formal parameter.
Orphan	When you see "Orphan" in the name, the function transfers responsibility for deleting an instance to the caller.
Get	When you see "Get" in the name, the function returns a pointer to an instance, and the caller is not responsible for deleting the instance.

Additional reading

For background information on graphics programming, refer to a dictionary of graphics terms or to the following books on graphics programming:

- Foley, James; vanDam, Andries; Feiner, Steven; Hughes, John. *Computer Graphics*. Reading, Massachusetts: Addison-Wesley Publishing Company, 1990.
- Bartels, Richard; Beatty, John; Barsky, Brian. *An Introduction to Splines for use in Computer Graphics & Geometric Modeling*. Los Altos, California: Morgan Kaufmann Publishers, Inc., 1987.
- Farin, Gerald. *Curves and Surfaces for Computer Aided Design*. Boston, Massachusetts: Academic Press, 1990.
- Wyszecki, G³nter. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. John Wiley & Sons, Inc., 1982.

RELATED CONCEPTS

2D Attributes and Bundles
2D Geometry

2D Modeling
2D Rendering Pipeline
2D Transformations

2D Graphics Features

The 2D Graphics Framework provides a full range of capabilities for high-quality 2D graphics:

IEEE Double-Precision (64-bit) Floating Point Arithmetic
Arithmetic calculations are performed with 64-bit floating point data types. This gives you much finer resolution and greater accuracy.

Graphics and Geometry Classes
The framework includes a full range of extensible 2D graphics classes. The graphics functionality is separated (put in different classes) from the geometry shapes to give you more flexibility. You can use graphics and geometry classes together, or use the geometry classes by themselves.

NURBS
The 2D curve geometries are defined by Non-Uniform Rational B-Splines (NURBS). Non-uniform means that the parameterization of the curve can be changed to allow kinks, gaps, and smooth joins. Rational means that rational polynomials are used to allow exact representations of circles, ellipses, and other conic sections. B-Spline stands for Basis spline, which is a spline curve very similar to the BÚzier curve, but provides more local control and continuity control to give you better curve fitting and modeling.

Customizable Architecture
The application-programming interfaces are clean so that you can easily extend or customize only those classes that require additional functionality and leave all other classes as they are.

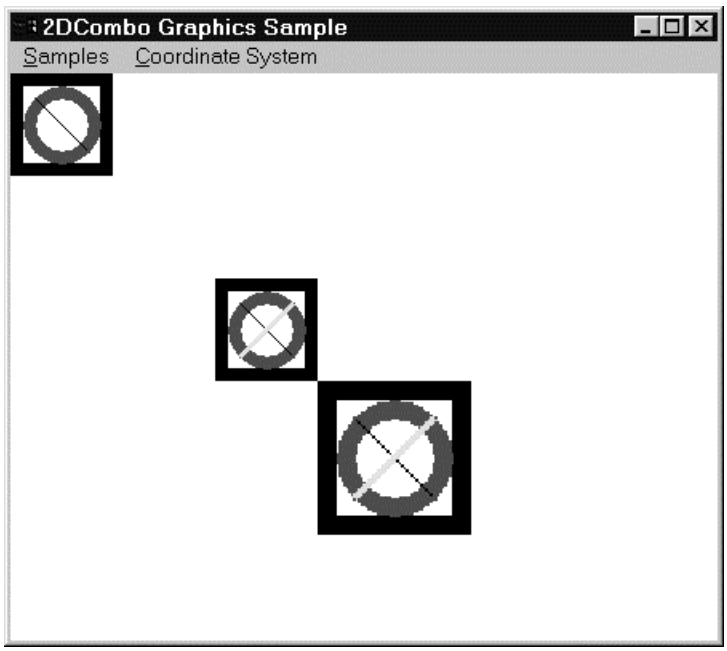
RELATED CONCEPTS

2D Graphics Overview

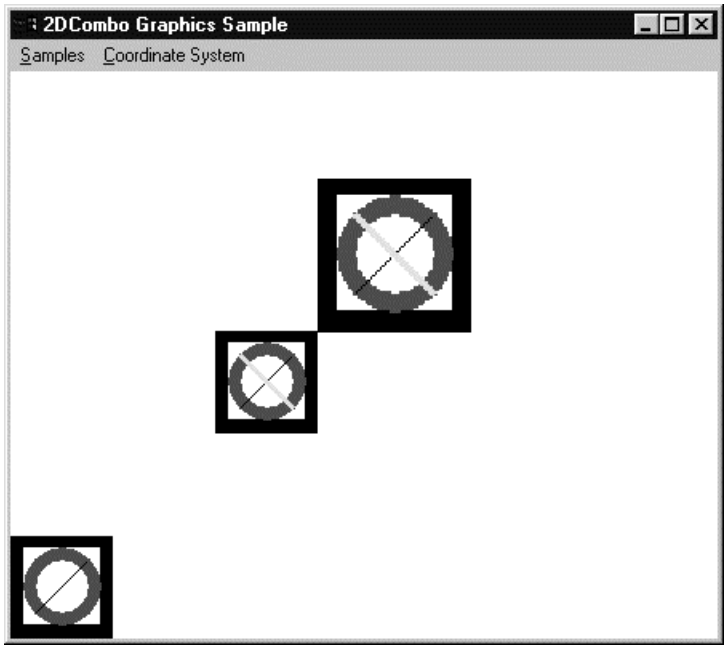
2D Coordinate System and Data Types

2D Coordinate System and Unit Resolution
The 2D geometry classes are specified in x and y coordinate units. The default 2D coordinate system positions the origin (0,0) in the upper-left corner of the view. The 2D Graphics Framework uses this coordinate system as a default because a typical document page starts at the top and extends downward. This is referred to as the World Coordinate space throughout this guide.

The following image taken from the 2D Combo Sample draws several images with the upper-left coordinate system:

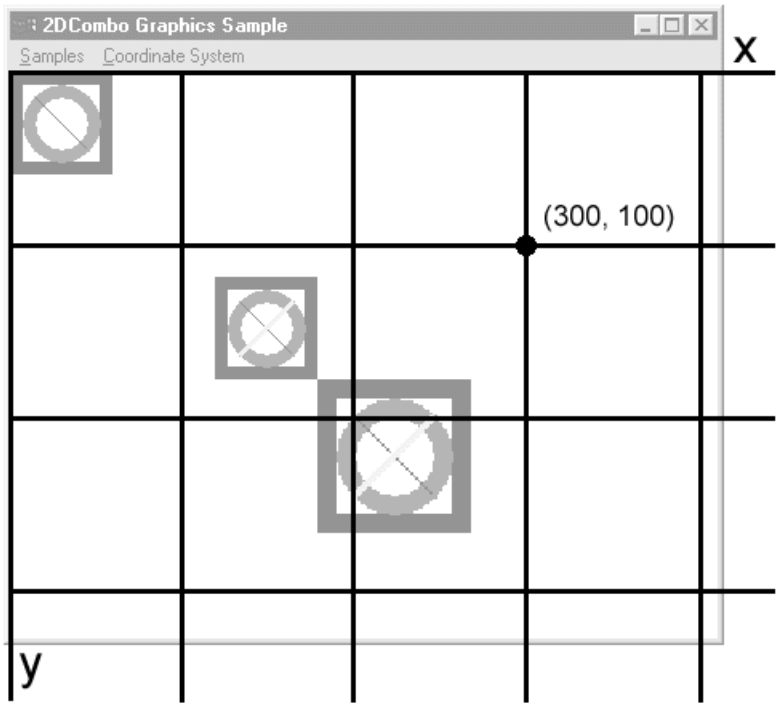


You can use a lower-left coordinate system that positions the origin at the lower-left corner of the view instead. The following image, from the 2D Combo Sample, draws the same images with the lower-left coordinate system:



You can find the 2D Combo Sample in the samples/ioc/graph2d/2dcombo directory.

Positive coordinate units along the x-axis lie to the right of the origin and positive coordinate units along the y-axis lie below the origin. The grid lines represent where points in coordinate units lie. For example, the point (300, 100) is 300 pixels in the positive x direction and 100 pixels in the positive y direction:



On a display device, the unit of any coordinate or dimension is assumed to be one pixel. The display size of a pixel depends on the *dpi* (dots per inch) characteristics of a particular device.

On a printing device, this coordinate unit is defined as a *point* (1/72 of an inch). A point is the unit of measurement used by printers, graphic designers, and type setters. It is common and convenient for desktop publishing systems to assume that one pixel equals one point.

For example, on a 96dpi display device, a 96-by-96 pixel square appears as a 1-by-1 square on the screen. When printed, the resulting square on paper is (about) 1.33-by-1.33 inches in size.

The enumerated type *EOrientation* defined in ICoordinateSystem describes the coordinate systems supported by the Open Class Library:

- The *kOriginUpperLeft* orientation positions the origin in the **upper-left** corner of the view.
- The *kOriginLowerLeft* orientation positions the origin in the **lower-left** corner of the view.

Use IExtendedRootGrafPort to define the coordinate system you wish to use.

Numeric and Arithmetic Data Types

2D geometry is based on IEEE double-precision floating point numerical values. The system has the following benefits and reduces the risk of numeric errors:

- Precision
- Speed: Many RISC processors and math co-processors work directly with double precision.
- Stability: Double precision is much more immune to round-off, under and overflow problems than fixed or single precision.

The following data types define the variables used in 2D calculations.

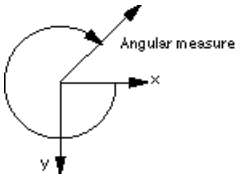
- GCoordinate
- GDegrees
- GParametric
- GIntensity

GCoordinate
typedef double GCoordinate;

GCoordinate expresses values defined or measured in 2D coordinate units such as positions or lengths. It maps directly to pixels on the screen and logical 1/72 inch on printers. Variables of type GCoordinate are used in all of the geometry classes to allow for smooth curves. GCoordinate provides sufficient range and precision for large documents, complex curves, and matrix operations.

GDegrees
typedef double GDegrees;

GDegrees measures angles in degrees and is used when an angle needs to be specified or returned. Angles start with 0 degrees at the +x-axis, and they increase from 0 degrees to 360 degrees going clockwise in 2D (because the +y-axis points down). Angles greater than 360 degrees are interpreted as modulo 360 degrees. Negative angles are added to a multiple of 360 degrees.



GParametric
typedef double GParametric;

GParametric represents the parameter of a parametric curve, loop, or surface. The parametric curve or loop is represented by the function P(u) that returns point P on the curve for a particular value of the parameter u. P(0) returns the point at the start of the curve, and P(1) returns the point at the end of the curve

GIntensity
typedef float GIntensity;

GIntensity is a 32-bit data type that represents a range between 0 and 1.0 that is used in most device color spaces to represent color intensity. This data type is found in the IBasicColor class.

RELATED CONCEPTS

2D Graphics Overview
2D Graphics Features

Chapter 2. 2D Attributes and Bundles

2D Graphics attributes and bundles classes provide primitive attributes, such as paint, pen, and image transfer mode. It also provides IGrafBundle which is the collection class for combinations of these primitive attributes used to render a 2D graphic geometry. Attributes and bundles classes are mainly used for assigning and collecting attributes for 2D graphics.

Attributes are the characteristics that describe how geometries are rendered. In a graphics system, you cannot render geometry without specifying the attributes because the system has no information about how to display the geometry. You can set the attributes for a rendering device without actually rendering any geometry. You can represent that set by a collection of attribute values. IAttributeState and its subclasses are designed to fill this role.

The number of attributes required by an application depends on the purpose. The 2D Graphics Framework provides a mechanism, a bundle, to collect attributes together for rendering. The IGrafBundle class collects together all attributes needed to render a geometry.

A graphic object can have an IGrafBundle or another subclass of IAttributeState. The 2D Graphics Framework provides some commonly used IAttributeState classes.

Attribute	**Description**
Drawing operations	Determine whether the attribute bundle is for fill, frame, or fill and frame.
Paints	Fill paint: Specifies the color of the fill for a 2D graphic. Frame paint: Specifies the color of the frame for a 2D graphic. Fill and frame paint: Specifies the color and pattern of the fill or frame for a 2D graphic.
Transfer modes	Fill transfer modes: (All 2D graphics drawn into a 2D image.) Provides algorithms to determine the color of a fill pixel when a 2D graphic fill pixel of one color overlays a 2D graphic or image pixel of another color. Frame transfer modes: (All 2D graphics drawn into a 2D image.) Provides algorithms to determine the color of a frame pixel when a 2D graphic frame pixel of one color overlays a 2D graphic or image pixel of another color. Image transfer modes: (2D images only.) Provides algorithms to determine the color of a 2D image pixel when a pixel of one color overlays a pixel of another color.
Frame styles	Pen styles: Specifies the pattern of the frame. Cap styles: Assigns a flush, square, or round shape that is added to the start or end of the frame for a line geometry to create a finished effect. Joint styles: Assigns a filler for the gap that is created in the frame by a sharp joint in a curve, using the standard bevel, round, and mitre frame joints.
Image sampling	Provides algorithms for sampling an image prior to drawing when it has been enlarged (enlarging the pixel information) or shrunk (compressing the pixel information). Algorithms make tradeoffs between quality and drawing speed.

RELATED CONCEPTS

Attribute and Bundle Classes
Color and Pattern Attributes
Frame Styles
"Transfer Modes" on page 17
"Image Sampling" on page 19
"Attribute Class Summary" on page 21

Attribute and Bundle Classes

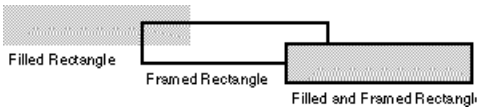
IAttributeState
IAttributeState is an abstract base class that defines drawing attributes for 2D graphics. IAttributeState has pure virtual data accessor functions to retrieve information about the attributes and drawing operation for the bundle. The drawing operation determines whether the bundle provides fills, frames, or fills and frames.

IGrafBundle
IGrafBundle is a collection of attributes that contains imaging information for the graphic rendering device. IGrafBundle is derived from IAttributeState and can include:

- Fill and frame attributes
- Image attributes

Fill and Frame Attributes
The *fill* is a color that spreads over the interior of the geometry, and the *frame* is the color and line that outlines the border of the geometry.

You can specify in a IGrafBundle object the *drawing operation*. The drawing operation tells the rendering device to fill the geometry, frame the geometry, or fill and frame the geometry, as shown in the figure:



The most frequently used attributes are fill paint, frame paint, pen, and draw operations.

You can specify the fill color and the frame color with the IBaseColor class. The IBaseColor class allows you to define a color in a variety of color spaces, including RGB, HSV, and CIE-XYZ.

You can specify the fill paint and the frame paint with the IPaint class. The IPaint class allows you to define the color, the image pattern, and the mask pattern (like cross-hatching) for a source primitive. (You can construct from an IPaint object an IBaseColor object.)

You can specify the attributes of the *pen* used to draw the frame. These attributes include the following:

pen type
The *pen type* describes how the pen draws the frame. These types include solid, dotted, dashed, and invisible.

pen width
The *pen width* controls the width of the frame. A pen width greater than one is always a solid pen type.

end cap style
The *end cap style* controls the shape of the unattached end of an unenclosed frame. These shapes include flush, round, and square. The pen width must be greater than one for this style to have any effect.

joining style
The *joining style* controls the shape formed by two intersecting lines. These shapes include bevel, roind, and miter. The pen width must be greater than one for this style to have any effect.

You can specify the fill *transfer mode* and frame transfer mode with the IColorTransferMode class. The IColorTransferMode class allows you to specify how your application draws a color on the screen.

Image Attributes
IGrafBundle lets you change the transfer mode and the *sampling control* of an image. The sampling control is the filtering method that controls the appearance of a rendered image when it has been interpolated (enlarged) or decimated (shrunk).

The image transfer mode and the image sampling control apply for the geometry of both IGImage and the IGImage set in IPaint for image pattern fill.

Special Purpose Attribute Bundles
While IGrafBundle provides more functionality than the special purpose attribute bundles by allowing you to specify transfer mode and image sampling settings for 2D image graphics, special purpose bundles are still provided. These classes are easy to construct and use. The supported special purpose attribute bundle classes are IFillBundle, IFrameBundle, and IFillAndFrameBundle.

IFillBundle
The IFillBundle class contains imaging information for 2D graphic objects of the solid-filled, no frame variety. The color is applied to the fill only, and one constructor lets you pass a transfer mode that is applied to the fill color.

IFrameBundle
The IFrameBundle class contains imaging information for 2D graphic objects of the non-filled, framed (with any size of frame) variety. These constructors have default pen width and pen balance values. The color is applied to the frame only and one constructor lets you pass a transfer mode that is applied to the frame color.

IFillAndFrameBundle
The IFillAndFrameBundle contains imaging information for 2D graphic objects of the solid-filled, framed (with any size of frame) variety. These constructors have default pen width and pen balance values. The colors are applied to the fill and frame, and one constructor lets you pass two transfer modes. One transfer mode is applied to the fill color and the other is applied to the frame color.

RELATED CONCEPTS

2D Attributes and Bundles
Color and Pattern Attributes
Frame Styles

"Transfer Modes" on page 17
"Image Sampling" on page 19
"Attribute Class Summary" on page 21

Color and Pattern Attributes

Color and Pattern Attributes classes specify color appearances, and allow you to create or modify existing patterns.

IPaint
The IPaint class specifies the color appearance of the geometry source primitive. The 2D Graphics Framework differs from traditional graphics systems in that it does not use a mechanism for foreground color, foreground pattern, background color, or background pattern. Instead, you specify a fill and frame color appearance by using a combination of IPaint objects and a transfer mode attribute.

The IPaint class is a concrete class that encapsulates the color and image pattern you can set. If you do not specify a color or image pattern, IPaint provides a default value of no image and of a solid pattern.

One of the constructors of the IPaint class can take the following three parameters:

- A reference of a color of type IBaseColor.
- A mask pattern of type IMaskPattern.
- a *pattern phase* parameter of type IGPoint2D. This parameter defines the offset for tiling the pattern. The default value is the origin of the object.

You may also create an IPaint object with an image pattern of type IGImage, and a pattern phase.

Each IPaint object contains an object of IBaseColor. IPaint also has the image pattern which allows for the specification of a repeating image as the filled or framed appearance. Each IPaint object contains a pointer to an IGImage object.

WIN

If the IGImage is larger than 8x8 pixels, only the upper left corner of the image is used.

To extend the capability of IPaint, you can derive IPaint to add attributes that allow you to express, for example, gradients, fills, or 2D shaders.

IBaseColor
IBaseColor is a concrete class that encapsulates all possible color spaces including RGB, HSV, and CIE-XYZ. It has the default representation of compact RGB color (where each element of red, green, and blue is represented by an 8-bit unsigned number). The IBaseColor class is as follows:

IBaseColor uses two data types to define color ranges and intensities:

- GIntensity
- CharIntensity

GIntensity is of type float. CharIntensity is an unsigned character.

The enumerated type EPredefinedColor defines some commonly used colors.

The default constructor sets the color to black. The Open Class Library provides you with the following ways to create IBaseColor objects:

- Using a predefined color. Create your object with the enumerated type EPredefinedColor.
- Constructing a gray by specifying the intensity of the gray and its opacity.
- Using the compact RGB color space. Create your object with the red, green, and blue components of the color, and the opacity.

The precision of the RGB color space is 8 bit by default. The base class implementation only supports the compact RGB color. You can derive a class from IBaseColor to represent your colors in a different precision or a different color space.

Your derived class constructor must at least initialize the base class by calling the base class constructor that takes RGB color as the parameter as follows:

```
IBaseColor(
    CharIntensity red,
    CharIntensity green,
    CharIntensity blue,
    CharIntensity opacity = 255);
```

You can also do this by calling the setters in the base class. Otherwise the default compact RGB color support does not represent any specific color other than the default black. In that case, the color specified in your derived class is wrong because it differs from what appears in the base class. Your derived class can override all virtual functions in the base class, thus overloading the default compact RGB color support.

For conversion to device-independent universal CIE-XYZ color space, your derived class should implement the following conversion functions that provide access for color conversion and color matching capabilities:

```
virtual void elementsInXYZColorSpace(
    GIntensity& X,
    GIntensity& Y,
    GIntensity& Z) const;
virtual void setElementsInXYZColorSpace(
    GIntensity X,
    GIntensity Y,
    GIntensity Z) const;
```

GIntensity is used here for precision calculation. It is defined to be a single-precision floating-point number.

IMaskPattern

IMaskPattern is an 8-by-8 image pattern of 1-bit color depth (black = transparent, white = solid). You can create the pattern or use one provided by IMaskPattern, returned by the following functions:

- solid()
- diagonalDown()
- crossPattern()
- diagonalCross()
- diagonalUp()

- horizontal()
- vertical()

When creating an image pattern, the pre-defined IMaskPattern provides a transparent background and uses the IPaint color specified to draw the foreground. However, for a custom IMaskPattern, the background is opaque and the background color will always be the default color (black). The foreground color for the custom IMaskPattern should still be the IPaint color. In short, a user-defined IMaskPattern will always have a black background, and only a pre-defined IMaskPattern will have a transparent background.

RELATED CONCEPTS

2D Attributes and Bundles
Attribute and Bundle Classes
Frame Styles
"Transfer Modes" on page 17
"Image Sampling" on page 19
"Attribute Class Summary" on page 21

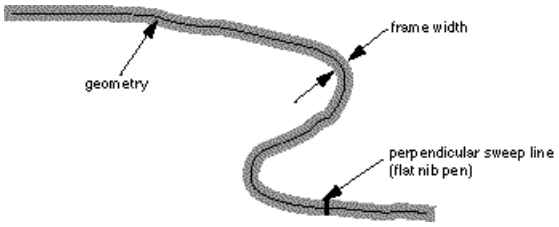
Frame Styles

The frame of the geometry is drawn using the following attributes: IPaint, IColorTransferMode, IPen, IJoint, and ICap. IPaint and IColorTransferMode apply to both the frame and the inside of the geometry while IPen, IJoint, and ICap define the frame of the geometry.

This section describes the classes that define frame styles of the geometry.

IPen
IPen is a concrete class that describes how to frame the edge of a geometry. IPen defines a pen by a pen width, pen balance, and pen type. By default, it is a centered-solid pen.

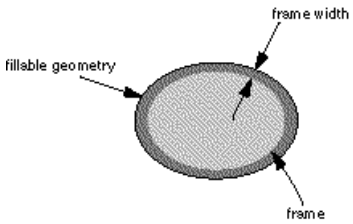
Pen width
Width is a floating-point value that represents the device-independent width of the pen in coordinate units. A centered-frame pen of a width can be thought of as using a pen with a flat nib of the specified pen width, and drawing a curve with the nib perpendicular to the direction of the curve with the center of the nib on the curve. In the figure, the "perpendicular sweep line" is the flat nib:



Balance
The enumerated type EPenBalance includes the following values:
- kCenterFrame
- kInsetFrame

For kCenteredFrame, the pen is centered over the border of the geometry for path and area-enclosing geometries.

For kInsetFrame, the pen fits exactly along the inside edge of a filled area-enclosing geometry as shown in the figure:



As a result, the size of the graphic drawn will be the same whether or not it has a frame.

The kInsetFrame value is useful for specifying the outside edge of a frame with pixel perfection rules. The kInsetFrame value describes a pen (of the pen width specified in the base class) on the inside of the edge of the geometry. The kInsetFrame value is only used for geometries that can be filled. When a geometry object to frame is not fillable, the pen behaves as a center framed IPen. An IPen kInsetFrame applied to IGLine2D, IGPolyline2D, and IGCurve2D always behaves as a center framed IPen.

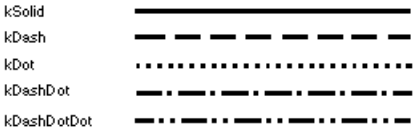
Pen Types
The enumerated type EPenType defines the values of different pen styles:

- kHairline
- kSolid
- kDashed
- kDot
- kDashDot
- kDashDotDot
- kInvisible
- kUnknownPen

The simplest pen style is kHairline. The kHairline style means the device renders a geometry frame using the fastest possible means. End caps, joints, and transfer modes are not applied to an IPen kHairline. Use the kHairline style only when speed is more important than accuracy. The only constraint placed on a hairline pen is that it must render within 1 pixel from the edge of a geometry.

If you want to ensure that a line will look the same across all platforms, use the kHairline style. The appearance of lines with a thicknesses greater than one pixel vary depending on the underlying platform support.

The following figure shows some of the pen styles available:



The layout of the patterns of these pens is not predictable because the underlying platform support is not consistent.

WIN

OS/2

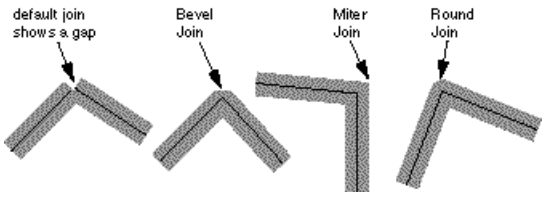
The kDashed, kDot, kDashDot, and kDashDotDot pen styles work only when pen width is one or less.

IJoint
The joint style controls the rendering of sharp points. Every sharp point causes a gap to occur between the edges of the frame. The IJoint class determines how to fill the gap of a sharp joint.

The enumerated type EJointType includes the standard joint styles as follows:

- kBevel
- kMiterLimit
- kRound

The following figure shows the standard joint styles:



The bevel joint style (kBevel) simply closes the gap between one edge and the next with a triangle.

The miter joint style (kMiterLimit) takes the two pens, and continues running in the original directions until the outside edges meet. The miter joint style has limitations. As the angle of the joint becomes greater, the limitation of what the joint can become when rendered is not predictable. On Windows® platforms, it is dependent on the GDI device drivers.

The round joint style (kRound) puts a conic section in the gap such that tangent continuity is maintained.

ICap
ICap places a cap at the start, at the end, or at an interior discontinuity (C0 discontinuity) of the frame of a linear geometry (IGLine2D, IGPolyline2D, IGCurve2D). Caps are not used on area geometries (IGRect2D, IGEllipse2D, IGPolygon2D, IGLoop2D, IGArea).

The enumerated type ECapType defines the supported cap styles:

- kFlush: A flush end cap does nothing.
- kSquare
- kRound
- kUnknownCap

The figure shows the supported cap styles:

A flush end cap (kFlush) does nothing.

A square cap (kSquare) extends the curve in a straight line by half the pen width.

A round cap (kRound) puts a semi-circle at the start or end of the curve.

RELATED CONCEPTS

2D Attributes and Bundles
Attribute and Bundle Classes
Color and Pattern Attributes
"Transfer Modes"
"Image Sampling" on page 19
"Attribute Class Summary" on page 21

Transfer Modes

A transfer mode is the specification of an interaction that takes place on the color of the primitive being drawn, on the current color at the destination location, or between both of these colors.

Transfer modes are algorithms that determine the resulting color when pixels of one color (source color) overlay pixels of another color (destination color) when 2D graphics are drawn. There is a wide range of transfer modes so that you can blend colors to achieve different visual effects when you work with 2D images.

One IGrafBundle can own a transfer mode for a fill, frame, and image. The transfer mode adopted for fill and frame applies to the 2D graphic that owns that bundle only when that graphic is drawn to a 2D image. The transfer mode adopted for an image applies to a 2D image when it is drawn to a view or when one 2D image is drawn on top of another 2D image.

Most transfer modes operate within a specific color space, which makes transfer modes device independent. When no transfer mode is adopted, the source color is completely copied over the destination color.

For information on mode type definitions, see texts on Windows graphics systems.

IColorTransferMode
The color transfer mode applies for color paint of fill and frame for the geometry to be rendered. In the case of rendering a geometry of IGImage or the IGImage set in the IPaint for image pattern fill, the color transfer mode implies the color is applied using such mode to the area the IGImage is rendered.

The default behavior of IColorTransferMode is source-replaces-destination or source copy mode.

The enumerated type EColorMode describes a set of supported color transfer modes:

- kSourceCopy
- kInvertSource
- kDestinationCopy
- kInvertDestination
- kOR
- kAND
- kXOR
- kInvertedSourceAND
- kInvertedSourceOR
- kInvertedDestinationAND
- kInvertedDestinationOR
- kInvertedAND
- kInvertedOR
- kInvertedXOR
- kONES
- kZEROS

IImageTransferMode
The image transfer mode applies for the geometry of IGImage. The default behavior of IImageTransferMode is source-replaces-destination or source copy.

The enumerated type EImageMode describes the supported set of image transfer modes:

- kSrcCopy
- kSrcORDst
- kSrcANDDst
- kSrcXORDst
- kSrcANDInvertDst
- kInvertSrc
- kInvertResultOfSrcORDst
- kSrcANDPattern
- kInvertSrcORDst
- kPatternCopy
- kPatternORInvertSrcORDst
- kPatternXORDst
- kInvertDst
- kAllZeros
- kAllOnes
- kOCHalfTone

RELATED CONCEPTS

2D Attributes and Bundles
Attribute and Bundle Classes
Color and Pattern Attributes
Frame Styles
"Image Sampling" on page 19
"Attribute Class Summary" on page 21

Image Sampling

Image sampling is the process of investigating the status of an image prior to drawing it when the image is either:

- Interpolated (enlarged), increasing the number of pixels during fractional translates and scales that result in stretching
- Decimated (shrunk), reducing the number of pixels during scale operations that result in shrinking

The image sampling control applies for the geometry of IGImage only. When a rendered image undergoes a transformation composed of something other than an integral translation, it is necessary to resample the image. When an image is drawn on one device and then drawn on another device of a different resolution, it is interpolated or decimated as appropriate to maintain the same size on both devices. Interpolation and decimation also occur during rotate, scale, perspective, and translate operations on the image graphic. Both interpolation and decimation can be used during a perspective transformation.

The IImageSamplingControl class allows you to control the filtering method employed during interpolation and decimation.

The enumerated type ESamplingMode provides several stock filtering methods:

- kBlackOnWhite
- kWhiteOnBlack
- kColorOnColor
- kHalfTone

WIN

These modes map directly into corresponding StretchBltMode for the GDI StretchBlt function.

RELATED CONCEPTS

2D Attributes and Bundles
Attribute and Bundle Classes
Color and Pattern Attributes
Frame Styles
"Transfer Modes" on page 17
"Attribute Class Summary" on page 21

Use Bundles in the Modeling Hierarchy

A graphic object hierarchy is traversed from the root to the leaves with the graphics state passed from parent to child through inheritance. Bundles are inherited through the modeling hierarchy by concatenating the object bundle with the state of the current IGrafPort.

Bundle inheritance uses the following rules:

- If an object does not have a bundle, the bundle is inherited. Otherwise it uses its bundle.
- If a bundle does not have an attribute, the attribute is inherited. Otherwise it uses the bundle's attribute.

There are two reasons for bundle inheritance: defaulting and saving storage.

Defaulting specifies that the parent fill in the bundle information. For instance, if the color of a graphic in a group is inherited, defaulting specifies that the subgroup graphic be filled in by the group's color.

If all frame paints are black, the black paint needs to be specified only in the root, thus saving storage.

RELATED CONCEPTS

2D Attributes and Bundles

RELATED TASKS

"Use Polymorphic Attributes in IGrafBundle"

Use Polymorphic Attributes in IGrafBundle

For ease of use, IGrafBundle has an API of copy semantics for setting standard attributes by passing the references of instances of attributes as parameters. The attribute object is copied and stored in the IGrafBundle object. Every standard attribute is extensible by subclassing. However, the attribute class must provide the typeid and polymorphic GraphicCopy::copy support to avoid "slicing" subclass of an attribute. Slicing the subclass of an attribute means that subclass specific information is lost because the subclass object is copied using a base class copy function which only has access to the base class data.

For example, the IPen object should have typeid and copy functions that the subclass IStyledPen overrides. Whenever an IPen object is referenced in IGrafBundle, if the source is an IStyledPen, IGrafBundle detects if the source is an IPen type by using the typeid support and calls the copy function IStyledPen stored in the resultant pointer instead of a copy of the concrete IPen object. This is true for all IAttributeState subclasses and for complex attribute classes such as IPaint.

In addition to subclassing one of the attribute classes in order to use polymorphic attributes, there are classes subclassed from IMStreamable available for such purpose. These classes use the typeid and polymorphic copy functions from IMStreamable. The classes subclassed from IMStreamable are:

- IPaint
- IBaseColor
- IPen
- IColorTransferMode
- IImageTransferMode
- IImageSamplingControl
- IJoint, and ICap

You can also use polymorphic attributes without using IMStreamable. However, you must subclass one of the attribute classes.

RELATED CONCEPTS

2D Attributes and Bundles

RELATED TASKS

"Use Bundles in the Modeling Hierarchy" on page 19

Attribute Class Summary

The table summarizes the attribute and bundle classes:

Class	Description
IAttributeState	An abstract base class that defines drawing attributes for graphics.
IGrafBundle	A collection of attributes that contains imaging information for the graphic rendering device.
IFillBundle	A derived class of IAttributeState that contains imaging information for a solid-filled, no-frame 2D graphic object. This is a special purpose bundle.
IFrameBundle	A derived class of IAttributeState that contains imaging information for a non-filled, framed 2D graphic object. This is a special purpose bundle.
IFillAndFrameBundle	A derived class of IAttributeState that contains imaging information for a filled, framed 2D graphic object. This is a special purpose bundle.
IPaint	A concrete class that defines the color and image pattern for a source primitive.
IBaseColor	A concrete class that encapsulates all possible color spaces including RGB, HSV, and CIE-XYZ. Implements compact RGBA Model (24-bit color + 8-bit alpha channel).
IMaskPattern	A class that defines an 8-by-8 image pattern of 1-bit color depth used in IPaint.
IPen	A class that provides most functionality for 2D pens. This class defines the frame style.
ICap	A class for implementing caps, the style in which to finish open lines and curves. This class defines the frame style.
IJoint	A class for implementing frame joints. This class defines the frame style.
IColorTransferMode	A class that describes the behavior of how source color, for color paint of fill and frame, is drawn at the destination.
IImageTransferMode	A class that describes the behavior of how a source image (IGImage) is drawn at the destination.
IImageSamplingControl	A class to specify a filtering method to control the appearance of the rendered image when it has been interpolated or decimated.

RELATED CONCEPTS

2D Attributes and Bundles
Attribute and Bundle Classes
Color and Pattern Attributes

Frame Styles
"Transfer Modes" on page 17
"Image Sampling" on page 19

Chapter 3. 2D Geometry

The 2D Graphics framework provides geometry classes and corresponding IMGraphic-derived graphics classes. The classes described here are the basic geometric forms that the 2D Graphics framework knows how to render, the "atoms" for building more sophisticated graphics.

Geometry Classes
The 2D Graphics framework geometry classes fall into the following general categories:

Computational geometry
Computational geometry classes are basic definitions for doing geometric computations. Computational geometry is used for calculations. It is not rendered directly except by IGRect2D which is filled and framed like other area-enclosing geometry. Use these classes for constructing other types of geometry.

Path geometry
Path geometry classes are primitives that divide space. Path geometry includes lines, polylines and curves. They may be framed, but not filled.

Area geometry
Area geometry classes are primitives enclosing space. They may be framed and filled.

Special area geometry
Special area geometry classes concatenate different geometric shapes to make new area geometries. One would generally use these area geometries to create clipping areas.

Image geometry
Image geometry classes handle and manipulate images.

Path geometry classes describe open shapes that need not be filled. Area geometry classes however describe shapes of enclosed path that can be filled. Some geometric shapes such as point, line and non-enclosing polyline and non-enclosing curves are of path geometry while enclosing geometric shapes such as rectangle, circle, ellipse, polygon and enclosing curves (loops) can be both path and area geometry.

Graphic Classes
The table summarizes the graphic classes, derived from IMGraphic, and their corresponding geometry classes:

Graphic class	Geometry class
ILine	IGLine2D: A line segment formed by 2 points.
ICurve	IGCurve2D: A general purpose approximating spline curves used for arc and other quadratic curves.
IPolyline	IGPolyline2D: A set of connected line segments.
ILoop	IGLoop2D: A closed curve.

Graphic class	Geometry class
IEllipse	IGEllipse2D: Circles and ellipses.
IPolygon	IGPolygon2D: A polygon with an arbitrary number of 2D vertices.
IArea	IGArea: A set of geometries enclosing areas combined by boolean operations.
IImage	IGImage: A pixel-based image.

The 2D graphics classes whose names specify a geometry (ILine, ICurve, IEllipse, and so on) are derived from IMGraphic.

The IMGraphic-derived graphics classes are wrappers for their referenced geometry classes. The graphics classes have functions that let you manipulate the geometry from within the graphic without accessing the geometry functions directly. The advantage of the wrapper architecture is that the 2D graphics classes bring together the functionality of the geometry classes and the high-level functionality of the IMGraphic class.

All IMGraphic-derived classes use a drawing port (IGrafPort) to render the graphic; they own an attribute bundle (IGrafBundle) and know the transformation matrix.

RELATED CONCEPTS

"Computational Geometry Classes"
2D Graphic Transformations Overview
"Point Arrays" on page 26
"Path and Area Geometry" on page 27
"Special Geometry" on page 30
Geometry Class Summary

Computational Geometry Classes

Computational geometry classes consist of basic definitions for completing geometric computations. Computational geometry is used for calculations. It is not rendered directly except IGRect2D, which is filled and framed like other area-enclosing geometry. Use these classes for constructing other types of geometry.

Points
The IGPoint2D class defines (x, y) coordinates that represent a point or vector on the 2D coordinate plane. The IGRPoint2D class defines (x, y, w) coordinate values that represent a homogeneous (rational) point or vector on the 2D coordinate plane. The IGPoint2D and IGRPoint2D member functions are summarized here:

IGPoint2D:

- crossProduct
- dotProduct
- interpolate
- normalize
- setPolarCoordinates
- setVectorLength
- setVectorAngle

- vectorLength
- vectorAngle

IGRPoint2D:

- divW
- dropW

IGPoint2D
IGPoint2D defines points on the coordinate plane. Some functions also treat IGPoint2Ds as vectors. IGPoint2Ds are used as the basic building blocks for most of the geometric primitives, and as a computational type for many types of geometric calculations.

This is a primitive type, and is not intended to be subclassed. IGPoint2Ds are treated much like numbers, with the typical operators (+, -, *, /) defined for them and acting component-wise on the fX and fY variables.

For example,

```
IGPoint2D a, b, c;
c = a + b;
```

Is equivalent to:

```
IGPoint2D a, b, c;
c.fX = a.fX + b.fX;
c.fY = a.fY + b.fY;
```

The complete list of operators defined for points is:

+, -, *, /, +=, -=, *=, /=, ==, !=, []

A numeric value can be supplied with the * and / operators to work as a scalar multiply; (for example, it is treated as a point with both fX and fY set to the value). The [ ] operator indexes the components (p[0] returns fX, p[1] returns fY). IGPoint2D's member variables, fX and fY are public, and designed to be directly accessed.

Mathematics makes distinction between points and vectors. Because their representation is the same, the 2D Graphics framework uses IGPoint2D in both roles.

IGRPoint2D
IGRPoint2D defines rational points containing a third coordinate, fW. They are most commonly used for defining rational or homogeneous coordinates in IGCurve2D. Curves use the rational coordinates to accurately represent conic sections (curves, arcs, ellipses, and so on). Rational points are also occasionally useful for situations where you need to keep track of a "perspective" term. Typically, you do not deal directly with IGRPoint2Ds, because the curve and transformation implementation manage the mathematical details.

Unlike IGPoint2D, the only operators defined for IGRPoint2D are ==, != and []. The [] operator indexes the fX, fY and fW components (p[0] returns fX, p[1] returns fY, and p[2] returns fW).

You can construct IGRPoint2Ds directly from IGPoint2Ds. In this case the value of the rational component fW is set to 1.0. Conversion from IGRPoint2Ds to IGPoint2Ds must be done explicitly via the divW and dropW methods.

IGRect2D
The IGRect2D class defines an axis-aligned rectangle defined by four coordinates, the top, left, bottom and right. Like IGPoint2D, IGRect2D is a "primitive" type, and not intended to be subclassed or extended. Also like IGPoint2D, the member variables of IGRect2D are public for ease of access. An IGRect2D is renderable by the IGrafPort as both a filled and a framed object.

When IGrafPort renders a filled and framed rectangle, the frame occupies the exterior boundary of the rectangle and the filled area occupies the entire rectangle except for the right and the lower exterior boundaries.

For example, the frame for an IGRect2D with the dimensions (0,0) - (5,5) consists of four lines running from (0,0) to (5,0) to (5,5) to (0,5) to (0,0). The filled area for the rectangle occupies all the elements from (0,0) to (4,0), (0,1) to (4,1), down to and including (0,4) to (4,4). Thus, the filled area overlaps the upper and left frame lines, but not the lower and right frame lines.

An IGRect2D is considered to be "empty" if fRight <= fLeft or fBottom <= fTop. Empty IGRect2Ds do not fill, and the extendTo ignores them. To find out if a rectangle is empty, call the isEmpty function.

RELATED CONCEPTS

2D Geometry
"Point Arrays"
"Path and Area Geometry" on page 27
"Special Geometry" on page 30
Geometry Class Summary

Point Arrays

IGPoint2DArray handles arrays of IGPoint2Ds and is the basis for IGPolyline2D and IGPolygon2D. IGRPoint2DArray handles arrays of IGRPoint2Ds and is the basis for IGCurve2D and IGLoop2D.

Both point array classes inherit from IRawArray. This class provides a low level substitute for traditional C arrays created with malloc and free. Along with the functionality of IRawArray, the point array classes let you do the following:

- Create a bounding rectangle that encompasses all of the points in the array
- Determine the number of points in the array
- Retrieve a point (or class IGPoint2D or IGRPoint2D) at the specified index
- Set a point at the specified index

IGPoint2DArrays are used to store and manipulate linear arrays of points (IGPoint2D and IGRPoint2D). They are designed to be efficient, compact and simple. These arrays are subclasses of IRawArray templates instantiated with the appropriate points. IRawArray does not call the constructor for each element of an array, thus improving performance.

You can create rational IGRPoint2DArray from an IGPoint2DArray (with all the w's defaulting to 1.0). The reverse is not true, however, because an explicit projection is required.

RELATED CONCEPTS

2D Geometry
"Computational Geometry Classes" on page 24
"Path and Area Geometry"
"Special Geometry" on page 30
Geometry Class Summary

Path and Area Geometry

Path and area geometry contains primitives that divide space. Path geometry includes lines, polylines and curves. They may be framed, but not filled.

IGline2D
The IGLine2D class defines line segments that are defined by two endpoints with an implied direction that moves from the start point to the endpoint.

IGCurve2D
The IGCurve2D class is the general curve primitive in the 2D Graphics framework. Use this to implement free-form curves, arcs, and path geometry containing discontinuities. The IGCurve2D class can represent the geometry of all other primitives (lines, polygons, ellipses).

The curve classes let you implement free-form curves, arcs, and path geometry that contain discontinuities (breaks or kinks in the smoothness of the curve) and that can be joined to form long, continuous curves. IGLoop2D derives from IGCurve2D. IGCurve2D and IGLoop2D can represent the geometry of all other 2D geometry classes.

Curves are defined by Non-Uniform Rational B-Splines (NURBs). Non-uniform means that the parameterization of the curve can be changed to allow kinks, gaps, and smooth joins. Rational means that rational polynomials are used to allow exact representations of circles, ellipses, and other conic sections. B-Spline stands for Basis spline, which is a spline curve very similar to the BÚzier curve, but provides more local control and continuity to give you better curve fitting and modeling.

IGCurve2D has derived classes, listed in the table, that you can use to instantiate a IGCurve2D. These classes serve to clarify different types of arcs that have similar arguments.

Class	Description
IGConicEndCenterEnd	This class creates a conic curve from a start point, endpoint, center point, and a Boolean value. The points create an elliptical shape. The conic curve tangents are perpendicular to the vectors between p0 and centerPoint, and centerPoint and p2. When the vectors are equal, a circular arc is generated that is less than 180 degrees.

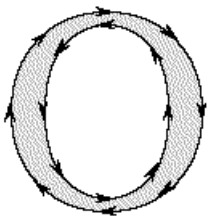
Class	Description
IGCardinalSpline	This class creates a parametric curve that interpolates a set of control points passed to it in the array. The fit is accomplished by interpolating the middle two control points and finding the tangent at those points that are parallel to the line between the previous control point and the next control point. The tension parameter controls how tightly the curve interpolates the control polyline. The tension parameter values range from 0.0 to 1.0 where 0.0 is very tight and 1.0 is very loose. The number of points in the array must be a multiple of four.
IGArcThrough3Points	This class creates an arc that passes through three points.
IGHermiteSpline	This class generates a parametric spline by interpolating between the two endpoints passed to it in the array and their tangents. The number of points in the array must be a multiple of four.
IGTensionSpline	This class creates a parametric spline from an array of points that are interpolated by tension and bias values. The tension controls how close the curve is to the control polyline. The number of points in the array must be a multiple of four.

IGLoop2D

IGLoop2D is a subclass of IGCurve2D that draws filled shapes. All of the functions supplied for IGCurve2D also apply for IGLoop2D. IGPie2D, a subclass of IGLoop2D, creates a pie-shaped wedge.

Loops are closed curves for drawing filled curves. Loops derive from curves. IGCurve2D and IGLoop2D can represent the geometry of all other 2D geometry classes because they have constructors that accept any 2D geometry. To fill a loop, use one of the fill algorithms described in “Fill Loops and Polygons” on page 29.

Discontinuities (breaks) in the curves of a loop are closed when the loop is drawn. For hit detection, the direction of each loop segment is used to determine whether a point is inside or outside the loop. For example, the letter “O” can be created from two concentric circles with the inner circle running in the opposite parametric direction from the outer circle. This way, points that lie between the two circles are found to be on the letter.



IGEllipse2D
The ellipse classes let you implement elliptical shapes that are inscribed inside a bounding rectangle. The bounding rectangle can be passed as the following:

- An IGRect2D.
- An IGEllipse2D.
- Dimensions for a center and radius.
- Dimensions for a center, height, and width.

The enclosed ellipse touches the midpoint of each edge of the bounding rectangle. When the height and width of the bounding rectangle are equal, the enclosed ellipse is a perfect circle. IEllipse2D has an internal matrix for transforming the ellipse without altering the original points of the ellipse. When you call a transformation function on an ellipse, the values are concatenated to the matrix.

IGPolygon2D
A polygon is a set of points that are connected by straight line segments and closed automatically. Polygons derive from polylines. You should not specify the same starting and ending points to close the polygon because the last point gets connected to the first point automatically. Polygons are closed line segments for drawing filled shapes. The polygons use the fill algorithms described in "Fill Loops and Polygons" for filling the polygon.

RELATED CONCEPTS

2D Geometry
"Computational Geometry Classes" on page 24
"Point Arrays" on page 26
"Special Geometry" on page 30
Geometry Class Summary

RELATED TASKS

"Fill Loops and Polygons"

Fill Loops and Polygons

Loops and polygons have a Boolean EOFill flag to determine the fill algorithm. A fill algorithm is necessary for loops and polygons because when a polygon or a loop is added to an area, the fill algorithm indicated by the fill flag affects the shape of the area. If portions of the interior of the geometry are not filled, those portions are cut out when the geometry is added to the area. Since only a geometry is passed into an area, the fill flag property belongs to the geometry.

The 2D Graphics framework fills loops and polygons with two different functions controlled by the EOFill flag. If the EOFill flag is true, then the polygon is filled with an Even-Odd rule; otherwise a non-zero rule is used.

The figures show the different fill results:

On Windows GDI device, the EOFill is mapped to ALTERNATE fill mode and the non-zero is mapped to WINDING fill mode.

RELATED CONCEPTS

2D Geometry

RELATED TASKS

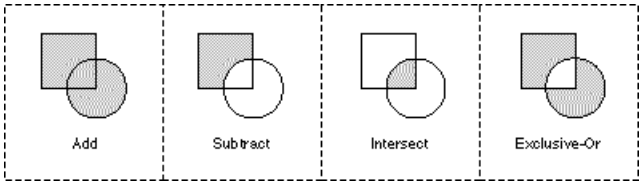
"Troubleshoot General Protection Faults" on page 31

Special Geometry

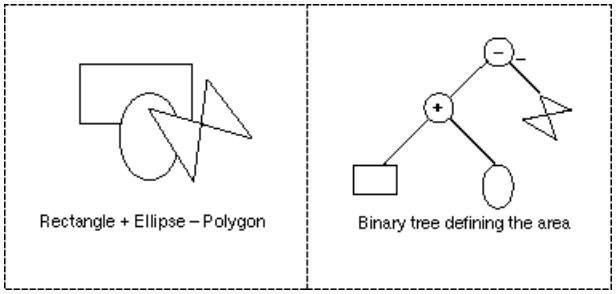
Special area geometry classes, used primarily for clipping, make area geometries by concatenating different geometric shapes.

IGArea
The IGArea class is a device-independent specification of an arbitrarily-shaped area. An IGArea object (henceforth referred to as an area) is defined by performing binary constructive area geometry (CAG) operations on other area-defining geometries, such as rectangles, ellipses, and polygons.

The following figure shows the operations. The area resulting from each operation is the shaded area:



You can form a binary tree to describe the content of any area. For example, the following area is defined to be the rectangle plus the ellipse minus the polygon.



When the area above is filled, only the highlighted outline is filled. Similarly, when it is framed, the frame is drawn around the boundary of the filled area.

Special Image: IGImage
IGImage is a special geometry. Even though it is a raster image, it does have geometric properties such as bounds and is transformable (by re-sampling if needed).

An image is a device-independent, pixel-based graphic that lets you work in pixels. You create an image by drawing graphics into it. The graphics are converted from coordinate units to pixels.

IGImage also encapsulates image data. Image data is normally stored in different file formats. These formats include bitmap (BMP), GIF, PCX, and TIFF.

RELATED CONCEPTS

2D Geometry
"Computational Geometry Classes" on page 24
"Point Arrays" on page 26
"Path and Area Geometry" on page 27
Geometry Class Summary

Troubleshoot General Protection Faults

For the classes IGPoint2DArray, IGCurve2D, IGLoop2D, IGPolyline2D, IGPolygon2D, and IGTensionSpline, you can receive general protection faults for very large numbers of points. These classes use IRawArray. Because IRawArray does not call constructors for the space it allocates, it does not initialize objects with vtables properly initialized. This can result in errors.

IRawArray does not check whether allocation is successful. Before constructing a new array, you should check if there is enough memory and use a reasonable size.

RELATED CONCEPTS

2D Geometry

RELATED TASKS

"Fill Loops and Polygons" on page 29

Geometry Class Summary

This is the architecture summary for 2D geometry classes.

2D Graphic Classes

Class	Description
ILine	Wrapper for the geometry class IGLine2D.
IPolyline	Wrapper for the geometry class IGPolyline2D.
ICurve	Wrapper for the geometry class IGCurve2D.
IEllipse	Wrapper for the geometry class IGEllipse2D.
ILoop	Wrapper for the geometry class IGLoop2D.
IPolygon	Wrapper for the geometry class IGPolygon2D.
IArea	Wrapper for the geometry class IGArea.
IImage	Wrapper for the geometry class IGImage.
IGraphicText	A specialized text string for creating and manipulating local text as a graphic.

2D Geometry Classes

	Class	Description
	IGLine2D	A line segment defined by a starting point and an ending point.
	IGPolyline2D	A set of points connected by straight lines.
	IGCurve2D	Free-form curves, arcs, and path geometry.
	IGConicEndCenterEnd	A lightweight class derived from IGCurve2D to disambiguate the constructors for arcs having similar arguments. It creates a curve as a conic with tangents perpendicular to ab and bc and centered about the second point.
	IGArcThrough3Points	A lightweight class derived from IGCurve2D to disambiguate the constructors for arcs having similar arguments. It creates a curve as a circular arc that passes through the three given points.
	IGCardinalSpline	A lightweight class derived from IGCurve2D to disambiguate the constructors for cubic splines having similar arguments. It creates a curve as a cubic matrix spline, where the curve interpolates the control points.
	IGHermiteSpline	A lightweight class derived from IGCurve2D to disambiguate the constructors for cubic splines having similar arguments. It creates a curve as a cubic matrix spline, where the curve is defined by point/tangent pairs.

	Class	Description
	IGTensionSpline	A lightweight class derived from IGCurve2D to disambiguate the constructors for cubic splines having similar arguments. It creates a curve as a cubic matrix spline, where the curve is defined by points, and interpolation is controlled by bias and tension values.
	IBaseCurve	Templatized base class for IGCurve2D.
	IGEllipse2D	Elliptical shapes including circles.
	IGLoop2D	A closed curve that can be filled as well as framed.
	IGPie2D	A subclass of IGLoop2D that constructs a pie-shaped wedge.
	IGPolygon2D	A closed polyline that can be filled as well as framed.
	IGArea	An arbitrary shape created from Boolean operations on one or more 2D geometries.
	IGImage	A pixel-based graphic.

2D Graphics Computational Geometry Classes

	Class	Description
	IGPoint2D	Represents a 2D point.
	IGRPoint2D	Represents rational (homogeneous) 2D points.
	IGPoint2DArray	Represents an extensible array of 2D points used in cases where an array or list of points is needed.
	IGRPoint2DArray	Represents an extensible array of 2D rational (homogeneous) points used in cases where an array or list of points is needed.

	Class	Description
	IRawArray	Template class that provides a low level substitute for traditional C arrays created with malloc and free. IRawArray stores only primitive types and objects with no virtual functions.
	IGRect2D	Used for 2D geometric calculations and as a basic building block for other 2D geometry classes. This is the only underlying 2D geometry class that can be sent directly to the drawing port for rendering. It is used to construct ellipses and polygons.

RELATED CONCEPTS

2D Geometry
Computational Geometry Classes
Point Arrays
Path and Area Geometry
Special Geometry

Chapter 4. 2D Modeling

The 2D Graphics framework provides two distinct ways of rendering geometries on a device:

- Draw the geometry directly to the device. The class IGrafPort supports a well-defined, fixed set of 2D geometries through a set of overloaded draw functions. This approach does not associate attributes and transformation matrices with geometry. This makes it suitable for immediate mode rendering only.
- Draw the geometry via a higher-level abstraction called IMGraphic. This is a retained-mode approach to rendering graphical primitives.

2D Modeling classes also provide a convenient way to associate geometry with attributes and transforms for rendering and storage.

IMGraphic is the abstract base class for representing the 2D primitives of the 2D Graphics framework. It is a higher-level manifestation of graphical objects which can be held in a collection, transformed, and rendered to a graphic device through an IGrafPort.

IMGraphic can be considered as a utility class for applications to hold geometry-related data that includes the geometry definition, its IGrafBundle (set of graphical attributes defining the representation of the geometry), and a set of transformation functions. You can derive from default IMGraphic objects and store other information specified.

Each IMGraphic object provides the following functionality:

- Holds a set of its own attributes and transformations.
- Provides streaming capability (with some restrictions on some of its subclasses).
- Provides inquiry functions as a mechanism for direct manipulation of IMGraphic objects such as hit detection.
- Provides extensibility through subclassing.
- Allows, through the IGraphicGroup subclass, creation of hierarchies of IMGraphic objects. IGraphicGroup creates a collection of IMGraphic objects forming a group. Each of the IMGraphic objects can be an IGraphicGroup itself, so one can create a hierarchy of objects.

Create your own IMDrawable and IMGraphic derived class to create complex graphics such as an automobile or human figure. A complex graphic is actually a combination of the geometry classes. The drawing port can render it because it knows all of the geometry classes.

RELATED CONCEPTS

"Graphics Class Architecture" on page 36
"IMGraphic Models" on page 36
"IGraphic Hierarchies and Groups" on page 40
"IGraphic Hierarchy and Group Iteration" on page 43
Modeling Examples
Modeling Class Summary

Graphics Class Architecture

The 2D graphic classes derive from IMDrawable. IMDrawable specifies the minimal protocol for drawing and streaming a 2D graphic. IMGraphic is an abstract base class that derives from IMDrawable:

IMDrawable provides the protocol for drawing a 2D graphic. IMGraphic provides the protocol common to all 2D graphics obtaining attribute information which it uses at rendering time, transforming the graphic, and finding the bounds of the graphic.

All IMGraphic derived classes use a drawing port (IGrafPort) to render the graphic; own an attribute bundle (IGrafBundle); and know the transformation matrix for a graphic (IGrafMatrix).

All IMGraphic derived classes know how to draw themselves. The graphics provide the geometry, attribute, and transformation information to the drawing port when the draw function of the graphic is called. This is a pure virtual function for IMGraphic, and all IMGraphic subclasses have to override this functionality.

IMGraphic Class Architecture
The 2D graphic classes whose names specify a geometry, such as ILine, ICurve, IEllipse, and so on, are derived from IMGraphic. A graphics class has public functions specific to that graphic that allow the graphic to interact with its referenced geometry.

The graphics classes are wrappers for their referenced geometry classes. For example, the graphics class ILine is a wrapper class for the geometry class IGLine2D. The graphics classes have functions that let you manipulate the geometry from within the graphic without accessing the geometry functions directly. The advantage to the wrapper architecture is that the 2D graphics classes bring together the functionality of the geometry classes and the high-level functionality of the IMGraphic class.

All IMGraphic-derived classes use a drawing port (IGrafPort) to render the graphic; own an attribute bundle (IGrafBundle); and know the transformation matrix for itself.

RELATED CONCEPTS

2D Modeling
"IMGraphic Models"
"IGraphic Hierarchies and Groups" on page 40
"IGraphic Hierarchy and Group Iteration" on page 43
Modeling Examples
Modeling Class Summary

IMGraphic Models

This section provides information on IMGraphic subclasses, attributes and bundles, transformations and bounds.

IMGraphic Subclasses
To provide application specific behavior, you can create concrete IMGraphic subclasses. The following are the two most common ways to derive classes from IMGraphic:

- Geometry manipulation (also termed Simple IMGraphics). This mechanism defines objects that have specific geometry associated with them. The base IMGraphic protocol defines the ways you can draw and manipulate them. The supported subclasses correspond to the well defined 2D geometry set. For most 2D graphics geometry classes, for example, IGLine2D, a corresponding IMGraphic counterpart class, ILine, exists that supports object-specific needs such as rendering, hit detection, and so on. These subclasses act as wrapper classes for the geometry and thus have a "has a" relationship with their geometry counterpart. You can also derive from IMGraphic to encapsulate more than one geometric primitive.
- IMGraphic manipulation (also termed Complex IMGraphics). This mechanism allows you to build a hierarchy of graphics to manipulate as a single graphic or group of graphics.

In addition to these two approaches, you can derive classes to create application specific subclasses, building application specific behavior into each of the subclasses. This is the most flexible approach. Use it to make classes more efficient for your specific applications.

NOTE: All concrete subclasses of IMGraphic must define, at minimum, the following pure virtual member functions:

```
virtual void draw(IGrafPort&) const = o;
virtual void transformBy(const IGrafMatrix&) = 0;
virtual IGRect2D geometricBounds() const = 0;
```

In addition to these mandatory member functions, subclasses should override some of the base implementations for performance gains, for example, translateBy, rotateBy, scaleBy, and so on.

IMGraphic Attribute Bundles
The properties that determine how a 2D graphic is drawn, such as fill color, frame color, and pen width, are called attributes. There are a number of classes for creating attribute bundles. Attribute bundles are instantiated and passed to a graphic to define the attributes for a particular graphic. If a graphic does not have an attribute bundle, the default graphic state of the root drawing port defines the graphic's attributes. See 2D Attributes and Bundles for a detailed discussion of attributes.

The graphic bundle is heavyweight, so it is expensive to copy. Use adoptBundle() and orphanBundle() semantics when changing and modifying the IMGraphic bundle. These semantics limit applications to create objects on the heap only. Once adopted, the IMGraphic object takes full responsibility to properly destroy the bundle when the IMGraphic object is destroyed.

To modify an attribute of an IMGraphic object:

1. Orphan the bundle.
2. Change the attribute.
3. Have the IMGraphic adopt the bundle again.

This process keeps the IMGraphic base class simple and avoids duplicating IGrafBundle functions.

The IMGraphic base class provides default implementations for all bundle related member functions. Subclasses need not override this functionality, unless they have an attribute-based cache which needs to be invalidated or updated whenever the bundle is adopted and orphaned. For example, the result of looseFitBounds, when cached, needs to be invalidated (or reevaluated) when the attributes change. For performance reasons, subclasses should use lazy evaluation for rebuilding the caches.

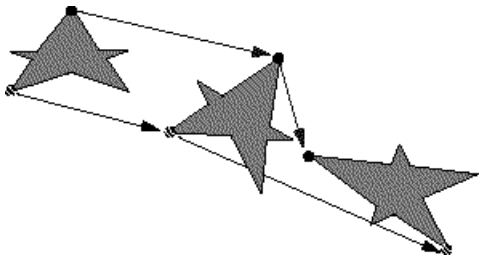
IMGraphic has functions to adopt, browse, and orphan bundles.

IMGraphic Transformation
Transformation is the process of altering every point in a graphic to resize (scale), reorient (rotate), or move (translate) the graphic.

Transformations can alter an IMGraphic's shape (by scaling and perspective transformation) and position (by rotating and moving). The transformation methods allow applications to change an existing IMGraphic's shape and location without having to recreate the IMGraphic.

IMGraphic transformation functions, translateBy, rotateBy, and scaleBy, provided for convenience and performance, let you transform a 2D graphic without using a matrix. You supply the values, and functions' default implementations create a temporary matrix and call the transformBy function, which takes an instance of IGrafMatrix as a parameter. Subclasses should override the default implementations to optimize for specific geometry and usage. Transformations change the points of the 2D geometry directly (except in the case of IEllipse) without copying the original points. IGrafMatrix is a 3-by-3 matrix for performing algebraic operations on a set of coordinate points (regular points, rational points, or vectors) that define a 2D graphic. IGrafMatrix is described at length in "2D Graphics Transformations."

The figure displays a star undergoing various transformations:



Subclasses of IMGraphic can implement these transformations in two ways:

- IMGraphics that own an IGrafMatrix (such as IGraphicGroup) concatenate all transformations with this local matrix. The rendering pipeline uses the resulting concatenated matrix as the overriding matrix at the time of rendering the IMGraphic onto the specified IGrafPort. The local matrix is often referred to as the modeling matrix for the IMGraphic.
- Other IMGraphic subclasses, which are light weight and do not own a matrix, apply the transform directly to the geometry they own, thus actually changing the geometry. Transformation methods of all supported IMGraphic subclasses are operations without undo capability. If your application must implement an undo capability, save a copy of the IMGraphic prior to transforming it.

All IMGraphic subclasses are “closed” to arbitrary transformations. An IGPolygon2D, when transformed by an arbitrary transformation, is still remains an IGPolygon2D. However certain 2D graphics geometries do not possess this closure property. For example, an IGRect2D when transformed by a perspective matrix is no longer a rectangle, and has no meaning of either width or height. The original specification of the rectangle (top left and the bottom right corner) are not sufficient to describe the transformed version of the rectangle. (IMGraphic does not support IGRect for precisely this reason, however an IGRect2D can be represented by an IGPolygon2D.) All IMGraphic subclasses must be closed to arbitrary transformations.

All transformations applied are relative; you cannot “un-transform” a transformed IMGraphic by passing an identity matrix to the IMGraphic member function transformBy(). If your application must be capable of undoing the last transformation, transform the IMGraphic object by the inverse of the matrix used for the relative transformations. For example, undo transformBy(matrix) by the call transformBy (matrix.invert()) or rotateBy(30.0) by rotateBy(-30.0). The matrix must be invertible and round-off errors can occur.

IMGraphic Bounds

All 2D graphics classes have bounds. The bounds of a graphic are defined by the smallest IGRect2D that encloses the geometry of the graphic.

The IMGraphic inquiry member functions allow you to inquire about certain specific properties of the graphics object without modifying the object itself. Any graphical representation of a geometry has at least two different types of bounds:

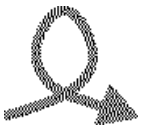
- Loose fit bounds
- Geometric bounds

The most common use of the inquiry functions is to detect whether a graphic is selected (hit or picked). The geometricBounds and looseFitBounds functions are fast and suitable when speed is more important than accuracy.

The 2D Graphics framework uses both the supported bounds inquiry member functions frequently, so the functions must be fast and efficient. Your concrete subclasses should cache these bounds if computing them is expensive.

These methods do not follow the strict guidelines of bit wise consts and are labeled as abstract const methods. Because they are abstract consts, they are unsafe when stored either in shared memory or in ROM (specially used for servers).

This figure displays the IMGraphic used in discussions of bounds mechanisms:

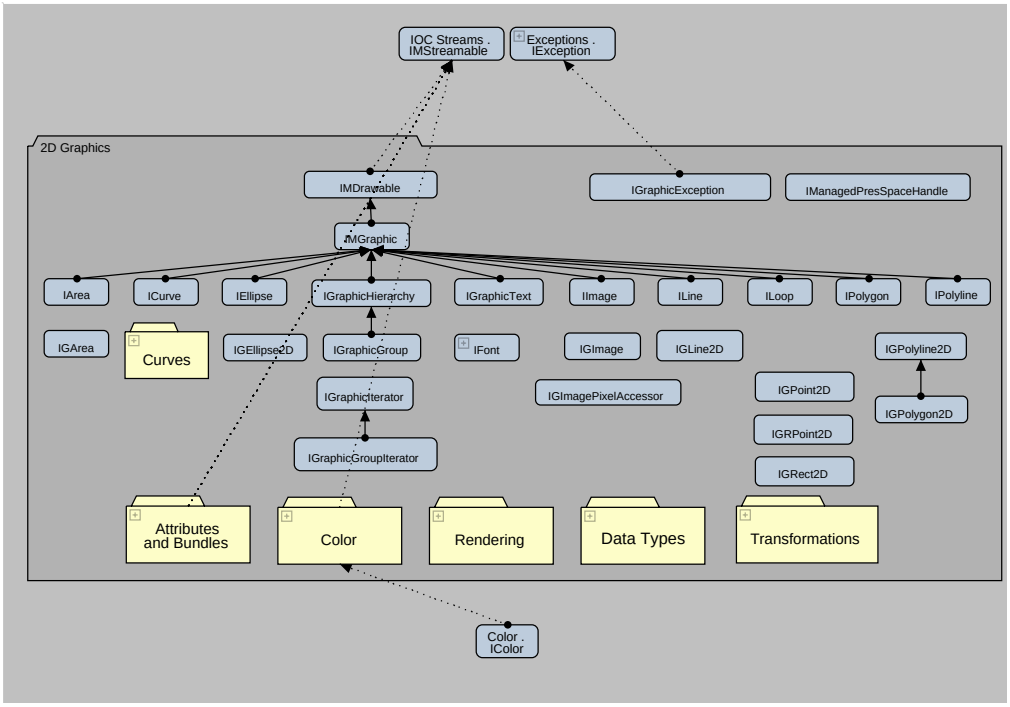


RELATED CONCEPTS

2D Modeling
“Graphics Class Architecture” on page 36
“IGraphic Hierarchies and Groups” on page 40
“IGraphic Hierarchy and Group Iteration” on page 43

IGraphic Hierarchies and Groups

The class IGraphicGroup derives from IGraphicHierarchy, which derives from IMGraphic:



IGraphicGroup provides a concrete implementation of iterating its children. The IGraphicIterator created iterates only one level. If you need to iterate more than one level deep, create iterators on subsequent IGraphicGroups.

IGraphicHierarchy
IGraphicHierarchy is an abstract base class that defines a hierarchy protocol for graphics that are grouped together into a graphic conglomerate.

You can create groups or hierarchies to descend from the base class IGraphicHierarchy for making available the iterator polymorphically.

The 2D Graphics framework provides a concrete subclass of IGraphicHierarchy, IGraphicGroup, which allows creation of trees of graphics to allow manipulation of a conglomerate group.

IGraphicGroup
IGraphicGroup is a concrete class, derived from IGraphicHierarchy, that implements a group of graphics as a tree structure. It allows you to combine 2D graphics into a unit that is treated as a whole. IGraphicGroup inherits directly from IMGraphic and thus each of the nodes owns its own IGrafBundle and can thus affect its own side of the hierarchy.

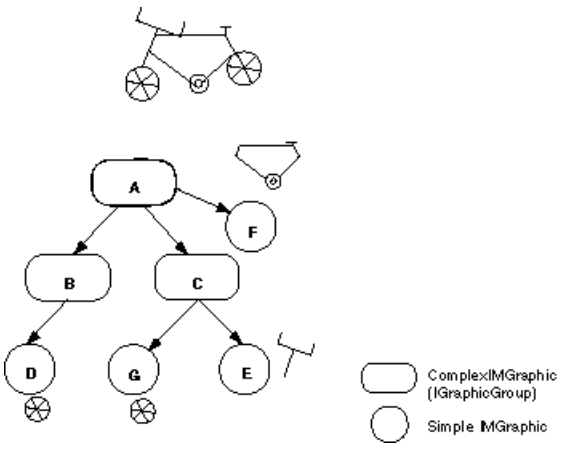
IGraphicGroup, as an IMGraphic subclass, references other IMGraphic objects. Although all manipulative behavior of complex IMGraphics is similar to simple IMGraphics, unlike the previous types, these objects do not completely encapsulate IMGraphic objects they refer to. IGraphicGroup descends from the abstract base class IGraphicHierarchy which makes available polymorphically the methods to create iterators for traversing groups.

IGraphicGroup does not provide support for DAGs (Directed Acyclic Graphics).

IGraphicGroup does not allow its children to have more than one parent in a team. (IGraphicGroup does no checks for this you must check for multiple references).

The destructor of IGraphicGroup destroys itself and all its children.

IGraphicGroup Relationships
IGraphicGroup creates a collection of IMGraphic objects forming a group. Each of the IMGraphic objects can be an IGraphicGroup itself, so one can create a hierarchy of objects. The figure shows an example of a hierarchy created by IGraphicGroup to create the image of a simplified bicycle:



In the figure, the elements have the following relationships:

- A, B and C are IGraphicGroups.
- D, E, F and G are different simple IMGraphics encapsulating more than one geometry.
- A has references to the following elements:
  - B, the rear wheel
  - C, the front wheel
  - F, the body of the bike
- B refers to D.
- C refers to the IMGraphics G and E.

Groups B and C control the transformations associated to the rear and the front wheel respectively. The two wheel are represented by the primitive geometries D and G. E represents the handle-bar of the bike. Thus moving node C moves both the front wheel and the handle-bar. Moving node A moves the entire bike.

While applying transformation matrix to the children at the time of rendering, the group creates a temporary IGrafPort object and concatenates its matrix with that

stored in the IGrafPort. This new IGrafPort renders its children and is destroyed once the child is completely rendered. The IGrafPort objects are created on the stack.

IGraphicGroup Attribute and Transformation Hierarchy
Each IGraphicGroup, if it so chooses, defines its own attributes and transformation. By default the attribute bundle is NIL and the transformation matrix is set to identity. As IGraphicGroup is a complex IMGraphic, it has references to other IMGraphics, its children. By definition, each of the child must inherit the attribute traits and transformations of its parent. However, since each child can be multiply referenced, it inherits these by concatenating the parents information, without modifying its own at the time of rendering. The concatenation of these attributes is achieved at the time of the draw call.

Both the attribute and the matrix are concatenated with the IGrafPort object which is passed as a parameter to the draw call.

The IGraphicGroup constructors let you instantiate a group with an attribute bundle. There is a default constructor.

```
IGraphicGroup(IGrafBundle* adoptedBundle);
```

In addition, each graphic adopted into a group can have its own attribute bundle. The attribute bundle of the graphic overrides the attribute bundle for the group into which the graphic is adopted.

In the bicycle example described in IGraphicGroup relationships, attributes and transformations of object A (body of bike) are concatenated with the IGrafPort object passed to A (as a parameter to the member function draw) and a new IGrafPort object, APortObject, is created on the stack. APortObject is passed to object C which concatenates its state and creates a new port object, CPortObject. This new CPortObject is passed to object E to be rendered. Object E concatenates its state with CPortObject and renders itself using this new state.

Matrix concatenation is simple post concatenation of the parent's matrix with that of the child. Thus, the matrix used to render object E is:

(E's Matrix) * (C's Matrix) * (A's Matrix)

In the wagon example, described in Complex IMGraphic, the wagon group has no attribute bundle so it has a default bundle. When the wagon body is adopted into the wagon group, it has a bundle that specifies fill and frame attributes that override the default bundle of the wagon group. When the wagon wheels are adopted into the wagon group, they also have attribute bundles that specify fill and frame attributes that override the default bundle of the wagon group. Because the wagon body and wagon wheel groups are at the same level in the hierarchy, their bundles have no affect on each other.

To transform the entire group, apply a transformation to the top-most graphic in the hierarchy. To transform only a part of the entire group, apply a transformation to the appropriate nested group. You can, of course, always transform any graphic before you adopt it into its group.

In the wagon example, a translation is applied to the wagon group to make the wagon body and four wheels move a specified distance. A rotation is applied to each wheel group so that the wheels rotate in addition to the translation.

The transformation matrix of the parent is post-concatenated with the matrix of the child. For example, the transformation and rotation matrix for the wagon wheels is found by multiplying the wagon group transformation matrix by the wagon wheel rotation matrix as follows:

((wagon wheel rotation matrix) * (wagon group translation matrix))

RELATED CONCEPTS

2D Modeling
Graphics Class Architecture
IMGraphic Models
IGraphic Hierarchy and Group Iteration
Modeling Examples
Modeling Class Summary

IGraphic Hierarchy and Group Iteration

Since IGraphicGroup allows creation of hierarchies, support for iterating the hierarchy has been built in this base class to be available polymorphically.

IGraphicIterator
This function is pure virtual in the abstract base class IGraphicHierarchy and all subclasses must provide an implementation. Subclasses who do not wish to expose their children may return an empty iterator when this member function is invoked.

```
IGraphicIterator* createGraphicIterator() const = 0;
```

This function creates an IGraphicIterator which iterates through the first level of a hierarchy. In the bicycle example, the iterator created for IGraphicGroup, A iterates over B, C and F. To iterate further, you must create iterators for both B and C. All subclasses creating hierarchies must provide a concrete implementation.

IGraphicIterator is an active iterator that facilitates the iteration over the children of an IGraphicHierarchy. IGraphicIterator functions include:

```
const IMGraphic *IGraphicIterator::first();
const IMGraphic *IGraphicIterator::next();
const IMGraphic *IGraphicIterator::last();
```

The createIterator function is called by the IGraphicGroup::draw function. The returned iterator covers one level of the group. The entire group is covered because the IGraphicGroup::draw functions are called recursively.

```
IGraphicIterator IGraphicHierarchy::createIterator();
```

IGraphicIterator has first, next, and last functions for traversing the group. The order that graphics are placed into the group is important because it affects the drawing order. This order is determined by the adoptFirst and adoptLast functions of the IGraphicGroup.

2D graphics are added to a group with the adoptFirst and adoptLast functions. Function adoptFirst puts the adopted graphic into the first position in the group; adoptLast puts the adopted graphic into the last position in the group. You nest groups by adopting a IGraphicGroup instance, which is derived from IMGraphic. There is no limit to how deep you can nest groups, but the same graphic cannot be adopted by more than one group.

```
IGraphicGroup::adoptFirst( MGraphic* graphic );
IGraphicGroup::adoptLast( MGraphic* graphic );
```

IGraphicGroupIterator
IGraphicGroupIterator provides functions to traverse and manage a hierarchical tree of graphic objects.

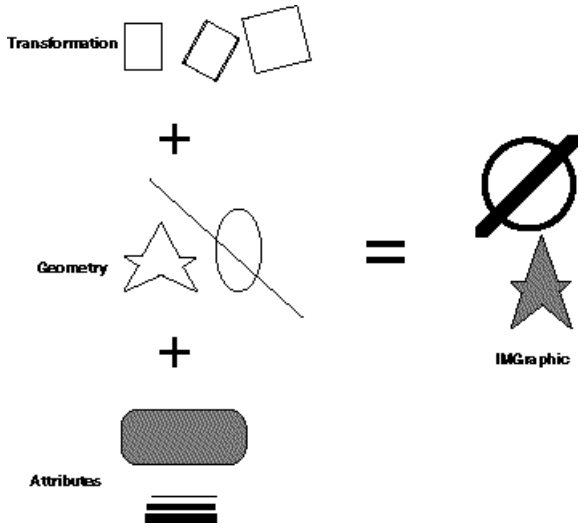
RELATED CONCEPTS

2D Modeling
"Graphics Class Architecture" on page 36
"IMGraphic Models" on page 36
"IGraphic Hierarchies and Groups" on page 40
Modeling Examples
Modeling Class Summary

Modeling Examples

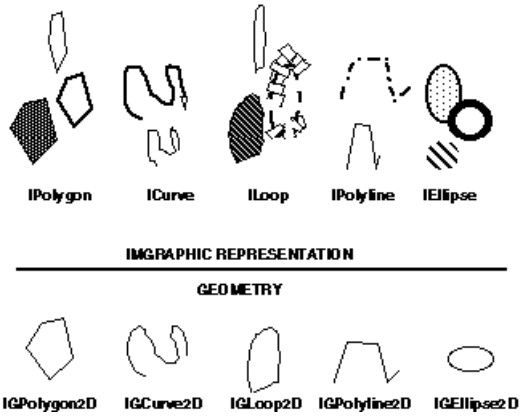
This section provides examples of IMGraphic subclasses and conglomerate groups of graphics.

IMGraphic Basic Components
The figure shows the basic components of IMGraphic:



IMGraphic and Corresponding Geometries
If your application needs to hold geometry-related data along with its graphical representation, use IMGraphic and its subclasses. You can also implement your own retained mode implementations similar to IMGraphic or extend IMGraphic for their particular application.

The figure shows various IMGraphics and their corresponding geometries.



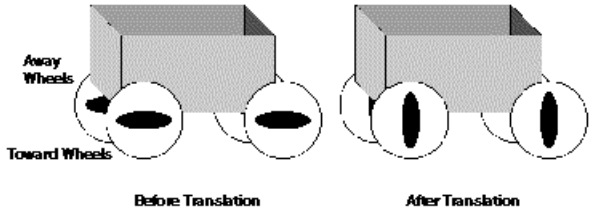
The 2D Graphics framework supports other concrete IMGraphic subclasses:

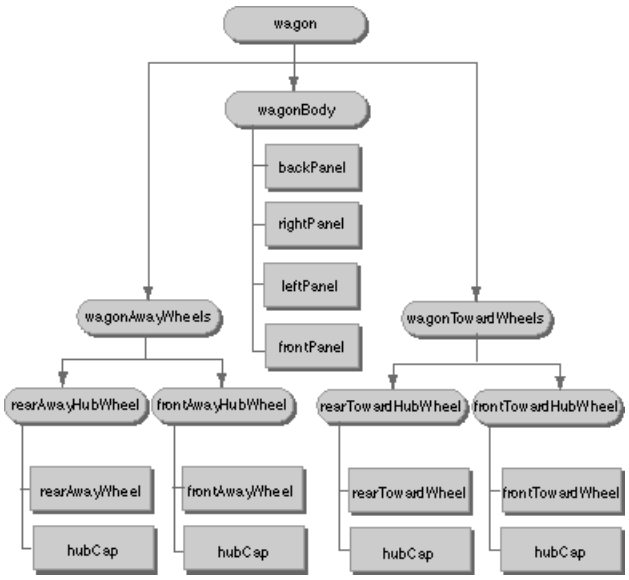
- IGraphicGroup
- IImage
- IArea

User-defined IMGraphic Subclasses
The figure shows simple and complex user-defined subclasses of IMGraphic.

Complex IMGraphic
A complex 2D graphic consists of a number of interrelated shapes. A wagon has a body, four wheels, and a handle. When the wagon moves, the wheels rotate. A 2D graphic group lets you create a complex 2D graphic such as a wagon that you can hit test, transform, and draw as a hierarchical unit.

In the wagon shown in the figure, the body consists of four panels and rests on four wheels. When the wagon moves, the wheels move too, but also rotate. The 2DDocs graphics sample located in the samples/ioc/graph2d/2ddocs directory draws this wagon:





RELATED CONCEPTS

2D Modeling
"Graphics Class Architecture" on page 36
"IMGraphic Models" on page 36
"IGraphic Hierarchies and Groups" on page 40
"IGraphic Hierarchy and Group Iteration" on page 43
Modeling Class Summary

Draw with IMDrawable and IMGraphic

The class IMDrawable, the abstract base class for IMGraphic, abstracts the common drawing behavior of the class. This class is useful if you are interested only in the draw function (as for printing) and do not want to provide overloaded functionality for primitives.

IMDrawable has a pure virtual draw function that is implemented in the IMGraphic derived classes. All IMGraphics draw onto the IGrafPort passed to IMGraphic as a parameter. Besides the state information encapsulated by the IGrafPort , the IMGraphic object itself contains all other information needed to render the graphic. This information includes the geometry, attribute bundle and any transformation information. This member function is an abstract const, so that subclasses of IMGraphic can cache expensive drawing information such as bounds. Concrete subclasses must implement the draw member function:

```
virtual void draw(IGrafPort& port) const = 0
```

All IMGraphics draw synchronously and do not handle simple updating or animating requirements. If you need this functionality, you must create subclasses providing such behavior.

When drawing primitives as a collection, such as in a list of IMDrawable objects, the drawing sequence is the same as it would be when draw calls are made on the specified IGrafPort. Thus drawing a polygon and an ellipse renders differently depending upon the order in which they are rendered.

The IGrafPort you pass to this member function is a passive iterator which is acted upon by the IMGraphic object to which it is passed. The IMGraphic object elaborates itself to the IGrafPort in terms of the 2D primitive set. IGrafPort does not have to be a rendering port.

RELATED CONCEPTS

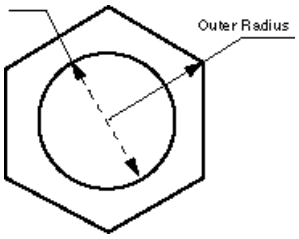
2D Modeling

RELATED TASKS

Create a Graphics Class
"Use IMGraphic Constructors" on page 48
"Use IMGraphic Destructors" on page 49
"Work with Bundles" on page 50
"Manipulate IMGraphic Objects" on page 50
"Compute Bounds" on page 53
"Implement IGraphicGroup" on page 55

Create a Graphics Class

The following sample class derived from IMGraphic creates a special 2D primitive to define the top view of a bolt. This class stores a transformation matrix for a local coordinate system. (This example does not take into account performance and efficiency.)



The figure shows the results of the class defining the top view of the bolt. (Dimensioning and labels are not part of IMGraphic.)

```
class IBoltTop : public IMGraphic
{
public:
IBoltTop(GCoord BoltDiameter, GCoord outerRadius, IGPoint2D center);
IBoltTop(const IBoltTop&);
IBoltTop& operator= (const IBoltTop&);
virtual void draw(IGrafPort&) const;
virtual IGPoint2D alignmentBasePoint() const;
virtual IGRect2D geometricBounds() const;
virtual void transformBy(const IGrafMatrix& matrix);
private:
IBoltTop(); // For streaming purposes only.
IGrafMatrix fMatrix;
IGPolygon2D fPolygon; // This is the outer polygon
IGEllipse2D fCircle; // This is the inner circle
void computePolygon(GCoord outerRad, int numOfSides);
};
IBoltTop::IBoltTop()
{
}
IBoltTop::IBoltTop(GCoord boltDia, GCoord outerDia, IGPoint2D center)
: fCircle(boltDia, center)
// calculate the hexagon polygon from these parameters
```

```
// The side of the polygon = outerDiameter / 2.0
IGPoint2DArray polygonPoints(6);
IGPoint2D tmpPoint;
for (unsigned long i = 0, theta = 0.0; i < 6; i ++, theta += kPi/6)
tmpPoint.fX = center.fX + outerDia * sin(theta);
tmpPoint.fY = center.fY + outerDia * cos(theta);
polygonPoints.setPoint(i, tmpPoint);
}
}
void IBoltTop::draw(IGrafPort &port) const
{
/*
* draw the geometry with the grafGrafBundle and the matrix
* associated with this primitive
*/
port.draw(fPolygon, fGrafBundle, fMatrix);
port.draw(fCircle, fGrafBundle, fMatrix);
}
IGPoint2D IBoltTop::alignmentBasePoint() const
{
// The alignment point is the center of the circle.
IGPoint2D point;
point.x = fCircle.centerX();
point.y = fCircle.centerY();
return point;
}
IGRect2D IBoltTop::geometricBounds() const
{
// Get bounds of the polygon
// pass the bounds to the bundle for altering.
bounds = fPolygon.bounds();
}
void IBoltTop::transformBy(const IGrafMatrix& matrix)|
{
fMatrix.concatWith(matrix);
}
```

RELATED CONCEPTS

2D Modeling

RELATED TASKS

"Draw with IMDrawable and IMGraphic" on page 46
"Use IMGraphic Constructors"
"Use IMGraphic Destructors" on page 49
"Work with Bundles" on page 50
"Manipulate IMGraphic Objects" on page 50
"Compute Bounds" on page 53
"Implement IGraphicGroup" on page 55

Use IMGraphic Constructors

The IMGraphic constructors are protected and are used only when you derive from IMGraphic to create a custom 2D graphic.

IMGraphic is an abstract base class that cannot be instantiated. In the following descriptions, creating IMGraphic actually means creating an instance of one of the IMGraphic subclasses.

The default constructor creates an instance of IMGraphic with a NIL IGrafBundle:

IMGraphic()

The copy constructor does a complete copy of the source IMGraphic. The base class makes a copy of the IGrafBundle:

IMGraphic(const IMGraphic& source)

This constructor creates an IMGraphic adopting the parameter adoptBundle. Once the bundle is adopted, let go of all the aliases to the bundle:

IMGraphic(const IGrafBundle* adoptBundle)

The default assignment operator does a complete copy of the source IMGraphic into the destination and clones the source IMGraphic bundle. The behavior of the assignment operator is similar to that of the copy constructor:

IMGraphic& operator= (const IMGraphic& source)

RELATED CONCEPTS

2D Modeling

RELATED TASKS

"Draw with IMDrawable and IMGraphic" on page 46
Create a Graphics Class
"Use IMGraphic Destructors"
"Work with Bundles" on page 50
"Manipulate IMGraphic Objects" on page 50
"Compute Bounds" on page 53
"Implement IGraphicGroup" on page 55

Use IMGraphic Destructors

The base implementation deletes the instance of the IGrafBundle owned by an IMGraphic:

virtual ~IMGraphic()

See also adoptBundle(IGrafBundle*) and orphanBundle() member functions.

RELATED CONCEPTS

2D Modeling

RELATED TASKS

"Draw with IMDrawable and IMGraphic" on page 46
Create a Graphics Class
"Use IMGraphic Constructors" on page 48
"Work with Bundles" on page 50
"Manipulate IMGraphic Objects" on page 50
"Compute Bounds" on page 53
"Implement IGraphicGroup" on page 55

Work with Bundles

The following section provides tasks for adopting, orphaning and getting bundles for IMGraphic.

Adopt Bundles for IMGraphic
To attach ownership of a bundle to the graphic, a graphic calls the adoptBundle function. A graphic can have ownership of only one bundle at a time. If the graphic already owns a bundle, the existing bundle is deleted and the new bundle is adopted. (As pointers are passed, it is important for the clients not to keep references to the bundle passed as the parameter.) The bundle is deleted when the graphic that adopted it is deleted:

```
virtual void adoptBundle(IGrafBundle *bundle)
```

Orphan Bundles for IMGraphic
A bundle should not be modified while it is under adoption. The bundle must be orphaned, modified, and then readopted when you want to change its attributes. The orphanBundle function orphans a bundle by returning ownership of the bundle to the caller. The caller is then responsible for deleting the bundle and changing the bundle attributes:

```
virtual IGrafBundle* orphanBundle()
```

When orphaned, the IMGraphic bundle is set to NIL, and when drawn, the IMGraphic uses the default mechanism of attributes/bundles, or its parents bundle as the case may be.

Get Bundles for IMGraphic
The inquiry function, bundle, returns a pointer to a constant bundle that cannot be changed. Use this function to make inquiries about the attributes present in the bundle. This function provides an alias to the bundle stored in the IMGraphic object.

```
virtual const IGrafBundle* bundle() const
```

RELATED CONCEPTS

2D Modeling

RELATED TASKS

"Draw with IMDrawable and IMGraphic" on page 46
Create a Graphics Class
"Use IMGraphic Constructors" on page 48
"Use IMGraphic Destructors" on page 49
"Manipulate IMGraphic Objects"
"Compute Bounds" on page 53
"Implement IGraphicGroup" on page 55

Manipulate IMGraphic Objects

This section provides tasks for manipulating IMGraphic objects.

Transform IMGraphic Objects
IMGraphic::transformBy is a pure virtual function that transforms a graphic through the given IGrafMatrix.

```
virtual void transformBy(const IGrafMatrix& matrix) = 0
```

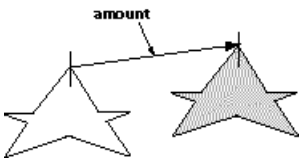
All concrete subclasses of IMGraphic must define this member function. You can create a local matrix or directly transform the geometry.

Subclasses that own an IGrafMatrix for manipulation must post multiply the parameter matrix with the local matrix for proper effect.

Move IMGraphic Objects
IMGraphic::translateBy moves the graphic by the amount specified in x and y coordinates relative to its current position (added to each point in the geometry). This is the default implementation. Your subclasses should override the default implementation to optimize for specific geometry and usage. You can use this method with baseAlignmentPoint() to move the geometry's base alignment point and the geometry.

```
virtual void translateBy(const IGPoint2D&)
```

The figure displays a star moved by the value of amount:



Rotate IMGraphic Objects
IMGraphic::rotateBy rotates the graphic clockwise around a center of rotation by the specified angle where:

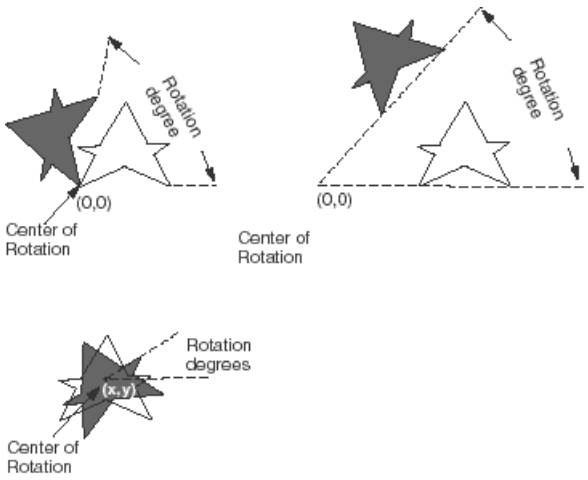
- angle is the amount of the rotation in degrees. Positive values rotate in a clockwise direction.
- centerOfRotation determines the point around which the graphic is rotated. A center of rotation other than the center of the graphic can rotate the graphic beyond the visible area of the view. The default center of rotation is IGPoint::kOrigin (the upper-left corner of the view).

The following is an example of a call to rotateBy:

```
virtual void rotateBy(
    const GDegrees,
    const IGPoint2D& centerOfRotation = IGPoint2D::kOrigin);
```

The result of a call to rotateBy() is the same as creating a rotation IGrafMatrix and passing it to transformBy(). This is the default implementation. Your subclasses should override the default implementation to optimize for specific geometry and usage.

The figure shows the results of rotating the star about different center of rotations:



Scale IMGraphic Objects

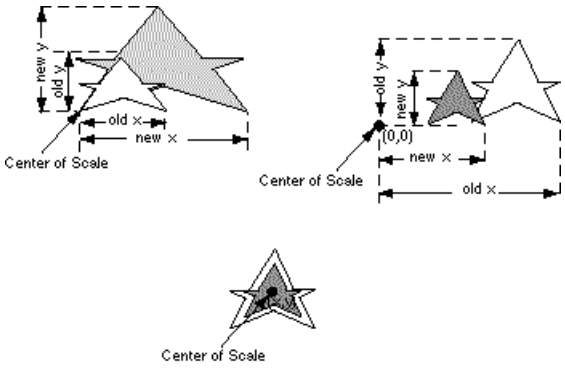
IMGraphic::scaleBy changes the size of a graphic by multiplying the graphic's coordinates by a constant value. It scales the graphic about the centerOfScale by a factor, where:

- factor determines the amount of scaling in the x and y directions for uniform (the x and y values are the same) and non-uniform (the x and y values are different) scaling.
- centerOfScale determines the point around which the graphic is scaled. A center of scale other than the center of the graphic can scale the graphic beyond the visible area of the view.

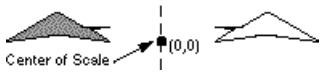
Similar to rotateBy() and translateBy(), the result of this call is the same as creating a scaling matrix and passing it to transformBy(). This is the default implementation. Your subclasses should override this default implementation to optimize for specific geometry and usage.

```
virtual void scaleBy(
    const IGPoint2D& factor,
    const IGPoint2D& centerOfScale = IGPoint2D::kOrigin);
```

In the figure, the X coordinate of the parameter amount is (new x/ old x) and the Y coordinate is (new y/ old y). In case of uniform scaling both the X and the Y coordinate are the same. The figure also shows scaling about different centers of scale;



The effect of negative scale factors is the same as mirroring. Scaling by -1.0 in the x direction is similar to mirroring about the y-axis.A negative scale factor in the y direction is similar to mirroring about the x-axis. The figure shows the effects of scaling an asymmetric star by (-1.0, 1.0).



Do not scale by 0.0. Scaling by 0.0 (in either X or Y) shrinks the graphic to zero and cannot be undone.

RELATED CONCEPTS

2D Modeling

RELATED TASKS

"Draw with IMDrawable and IMGraphic" on page 46
Create a Graphics Class
"Use IMGraphic Constructors" on page 48
"Use IMGraphic Destructors" on page 49
"Work with Bundles" on page 50
"Compute Bounds"
"Implement IGraphicGroup" on page 55

Compute Bounds

Provided in this section are tasks for computing the geometric bounds of a graphic.

Compute Bounds Using looseFitBounds
The looseFitBounds member function takes the drawing port as a parameter and returns an IGRect2D that encloses the entire graphic, taking into account the pen information. The function estimates the geometric bounds by calling the member function IMGraphic::geometricBounds(). It alters them through IGrafBundle::alterBounds() if the graphic has a non-NIL attribute bundle. LooseFitBounds returns a bounding rectangle that is either the same or larger than the geometricBounds bounding rectangle. Use this member function to speed up quick rejects for hit testing and quick redrawing.

This is a virtual function and all concrete subclasses of IMGraphic must provide an implementation:

```
virtual IGRect2D looseFitBounds(const IGrafPort* port = 0) const;
```

Only subclasses that need to cache the data need to override the member function. Use the port if you are computing looseFitBounds for that particular port rather than the default port.

The figure shows loose fit bounds of an IMGraphic curve:


looseFitBounds

Because the implementation for looseFitBounds calls bounds on the geometry first, a small amount of inaccuracy can occur. The returned IGRect2D might enclose extra points beyond the graphic and its bundle information. The rectangle is greater than or equal to the pixels touched when you render the IMGraphic.

The function estimates bounds by taking the bounds of the convex hull of the geometry and altering the bounds by the attributes used to render the IMGraphic. The convex hull for a 2D curve is the convex polygon formed by the control points of the curve such that it completely encloses the curve. The convex hull for a simple geometry is the geometry itself. For example, the pen width of the frame or the end caps for the geometry changes the bounds.

Infinite miters can cause problems in estimating bounds. The length of the miter is a function of the angle of the joint. As the angle of the joint approaches zero degrees, the miter length approaches infinity. In such cases IMGraphic returns the miter limit instead of infinity. Infinite geometries return infinite bounds.

Compute Bounds Using geometricBounds
The geometricBounds member function represents the pure geometric bounds of the graphic. This function returns the IGRect2D for the rectangular bounds of the image without the influence of IGrafBundle. It computes the bounds in a method similar to loose fit bounds (by calculating the convex hull) without the effects of attributes. Implementation of this member function generally entails computing the bounds of the geometry held by the IMGraphic. This member function is pure virtual function.

```
virtual IGRect2D geometricBounds() const = 0;
```

The figure shows the rectangular geometric bounds of an IMGraphic curve:


geometricBounds

RELATED CONCEPTS
2D Modeling

RELATED TASKS
"Draw with IMDrawable and IMGraphic" on page 46
Create a Graphics Class
"Use IMGraphic Constructors" on page 48
"Use IMGraphic Destructors" on page 49

"Work with Bundles" on page 50
"Manipulate IMGraphic Objects" on page 50
"Implement IGraphicGroup"

Implement IGraphicGroup

IGraphicGroup is implemented as a simple sequence list of graphic objects. It offers functions for inserting and removing items from the list, much like the functionality of the underlying collection classes. The more complex methods use the IGraphicIterator facility:

These methods include the following:

- void adoptFirst (IMGraphic *)
- void adoptLast (IMGraphic *)
- virtual void orphan (const IMGraphic *)
- IMGraphic* orphanFirst()
- IMGraphic* orphanLast()

IGraphicGroup has adopt and orphan semantics clearly sending the message that an IGraphicGroup owns all its children and the caller should not hold on to pointers outside this class. Member function adoptLast adopts the IMGraphic and puts it at the end of the collection while adoptFirst puts it in the beginning of the collection.

An instance of IGraphicGroupIterator is created by calling member function createIterator on an IMGraphic.

Find the Bounds of an IGraphicGroup
IGraphicGroup inherits the looseFitBounds and geometricBounds functions from IMGraphic described in "2D Graphics Framework." When you call one of these functions on a group, the appropriate bounds are returned for the entire group.

If you want to find the bounds on a nested group or graphic within the group, you have to use an iterator to locate the graphic or nested group that has been selected. To do this, call the IGraphicGroup::createIterator function that returns a IGraphicIterator instance. IGraphicIterator is an abstract base class, and by default createIterator creates a shallow iterator that iterates one level of the group.

RELATED CONCEPTS

2D Modeling

RELATED TASKS

"Draw with IMDrawable and IMGraphic" on page 46
Create a Graphics Class
"Use IMGraphic Constructors" on page 48
"Use IMGraphic Destructors" on page 49
"Work with Bundles" on page 50
"Manipulate IMGraphic Objects" on page 50
"Compute Bounds" on page 53

Modeling Class Summary

The classes used for modeling are listed in the table below:

Class	Description
IMDrawable	An abstract base class, from which the 2D graphics classes descend, that allows a graphic object to draw itself.
IMGraphic	An abstract base class that provides protocols for constructing and deleting a 2D graphic, obtaining attribute information, transforming the graphic, and finding the bounds of the graphic.
ILine	Wrapper for the geometry class IGLine2D.
IPolyline	Wrapper for the geometry class IGPolyline2D.
ICurve	Wrapper for the geometry class IGCurve2D.
IEllipse	Wrapper for the geometry class IGEllipse2D.
ILoop	Wrapper for the geometry class IGLoop2D.
IPolygon	Wrapper for the geometry class IGPolygon2D.
IArea	Wrapper for the geometry class IGArea2D.
IImage	Wrapper for the geometry class IGImage2D.
IGraphicHierarchy	An abstract base class that defines a hierarchy protocol for grouped graphics.
IGraphicIterator	An abstract base class that allows iteration of a graphics hierarchy.
IGraphicGroup	A concrete class, derived from IGraphicHierarchy, that implements a group of graphics as a tree structure. This class for combines 2D graphics into a unit that is treated as a whole.
IGraphicGroupIterator	A class that provides functions to traverse and manage a hierarchical tree of graphic objects.

RELATED CONCEPTS

2D Modeling
"Graphics Class Architecture" on page 36
"IMGraphic Models" on page 36
"IGraphic Hierarchies and Groups" on page 40
"IGraphic Hierarchy and Group Iteration" on page 43
Modeling Examples

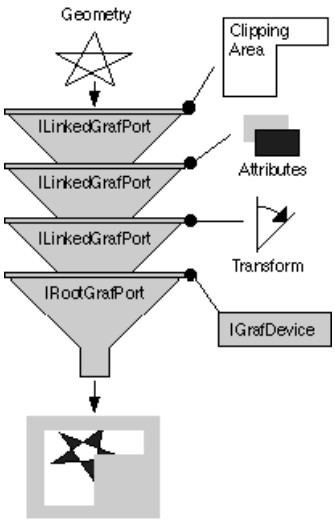
Chapter 5. 2D Rendering Pipeline

The rendering pipeline model for the 2D Graphics framework is the IGrafPort class. Through this class, you can associate graphics attribute states, transformation matrix, and clipping areas as the state for a given port. You can render geometry by using IGrafPort with the state applied along the rendering pipeline and the states will remain stored in the port.

At the root IGrafPort pipeline or chain, an IGrafDevice deals with all the aspects of rendering geometry with respect to the state set.

The IGrafPort is the component for constructing a hierarchy of graphics states. The IGrafPort can be constructed by specifying a link to the parent IGrafPort and some state information. The state information does not have to be the complete information required to render the geometry, because the chain of IGrafPorts eventually provides enough information to render the geometry. In the event of missing state information, the mechanism inherits the values from its parent. IRootGrafPort has the default values for all attributes.

The figure provides an overview of the rendering pipeline:



The 2D Graphics framework provides two distinct ways of rendering geometries on a device.

- Draw the geometry directly to the device. The class IGrafPort supports a well-defined, fixed set of 2D geometries through a set of overloaded draw functions. This approach does not associate attributes and transformation matrices with geometry. This makes it suitable for immediate mode rendering only.
- Draw the geometry through a higher-level abstraction called IMGraphic. This is a retained-mode approach to rendering graphical primitives.

Immediate and retained mode
In pure immediate mode rendering, the application objects are rendered without keeping any record of the information (geometry, attributes, hierarchies) used to

render them. In retained mode, the record of all objects, their hierarchies, and other related information, is stored for rendering.

RELATED CONCEPTS

2D Graphics Rendering Pipeline Overview
"GrafPorts"
"Special GrafPorts in Open Class" on page 60
Rendering Pipeline Class Summary

RELATED REFERENCES

IGrafPort
IMGraphic

GrafPorts

The IGrafPort class is a convenient container of the graphic state that is associated with the undefined native rendering device, such as the GDI interface for Windows. Its most important function is that it supplies a set of non-virtual draw calls that make it convenient for drawing geometries.

For convenience, the IGrafPort class provides 2D draw calls.

The large number of disjoint calls (as opposed to a polymorphic single call) enforces the rule that only a small, well-defined set of basic 2D geometries is supported by graphics devices.

IGrafPort also provides 2D draw calls that take an additional bundle and model matrix for convenience in certain specialized conditions. The draw calls use the graphics state set by subclasses of IGrafPort. ILinkedGrafPort is a very efficient way to set up hierarchical graphics states. You can make these states shared among the appropriate geometry objects to be extended. The graphic state includes attribute state (bundle), clipping area, and transformation matrices.

There are two kinds of transform matrices in rendering:

- The view matrix describes the view coordinate space. Every geometry and every attribute associated with geometry is subject to transform when rendered.
- The model matrix is the matrix describes the model coordinate space. Only the geometry is subject to transform, not the attributes.

Member functions allow you to access the graphic state.

The 2D Graphics framework provides two distinct ways of rendering geometries on a device.

- Draw the geometry directly to the device. The class IGrafPort supports a well-defined, fixed set of 2D geometries through a set of overloaded draw functions. This approach does not associate attributes and transformation matrices with geometry. This makes it suitable for immediate mode rendering only.
- Draw the geometry through a higher-level abstraction called IMGraphic. This is a retained-mode approach to rendering graphical primitives.

Note: Immediate and retained mode: In pure immediate mode rendering, the application objects are rendered without keeping any record of the information

(geometry, attributes, hierarchies) used to render them, while in retained mode, the record of all objects, their hierarchies, and other related information is stored for rendering.

IGrafPort subclasses
A subclass of IGrafPort inherits the set of convenient draw functions in the base class.

IBaseRootGrafPort
The IBaseRootGrafPort class contains an IRootGrafState object, which contains default values for every possible substate that is needed for rendering a geometry. It has a default graphic state setting.

IBaseRootGrafPort uses World Coordinate units to draw all the graphics. One unit is equivalent to one point (1/72 of an inch). Graphics that draw to IBaseRootGrafPort are resolution independent.

ILinkedGrafPort
An ILinkedGrafPort can associate itself with one of the three types of graphic state information. These three types of graphic state are attributes, transformation matrix, and clipping information. The attributes are represented by an IAttributeState object. IAttributeState is an abstract base class of IGrafBundle, IFillBundle, and so on. The transformation matrix is represented by an IGrafMatrix object. The clipping information is represented by an IGArea object.

With this class, one can construct a hierarchy of "linked" ports each representing a different state. All state information are kept hierarchically so it is easy for one to change some states without reconstructing the hierarchy.

IStatefulGrafPort (special class)
The IStatefulGrafPort class is similar to IBaseRootGrafPort because it contains an IRootGrafState object, which has a full set of graphic states that is needed for rendering a geometry. It also has a pointer to an IGrafDevice object. The difference from the IBaseRootGrafPort is that you can set the value of the graphic states individually, and therefore it provides greater flexibility. This gives you a simpler API, but you have the responsibility of determining which of the states needs to be set and structuring the graphic state information. This is not as efficient as ILinkedGrafPort because the way hierarchical state linking can be shared among draw callers.

Important: This class is provided for experimental use and is subject to change. Use it for special case implementation only.

For any primitives that the native platform does not support, a subclass of IGrafDevice called IFrameBuffer can perform appropriate rendering in software to form the raster image. Along the way of rendering, the order of complexity of a high-level geometry with the specific attributes, transform, and clipping states can be reduced to a simpler geometry successively, until either a native operation can be carried out for such a geometry, or the resulting image can be constructed at the end. For example, a circle defined by a center and radius is to be rendered with a gray pattern fill. If the native rendering engine does not support gray fill pattern for conics but does for polygons, the circle is discretized to a polygon and the native polygon with gray fill pattern is performed.

GrafPort Subclass Notes
It may be that there will be a case that an image is created by loading from an

images file and is drawn to both IBaseRootGrafPort and IRootGrafPort. The size of the image displayed in IBaseRootGrafPort will be bigger than the size of the image in IRootGrafPort. But if the image is used as a fill pattern to fill graphics in both IBaseRootGrafPort and IRootGrafPort, the images in both GrafPorts will be the same size.

RELATED CONCEPTS

2D Rendering Pipeline
"Special GrafPorts in Open Class"
Rendering Pipeline Class Summary

RELATED TASKS

"Draw a Simple IMDrawable IPolygon" on page 62
Access Graphic State

Special GrafPorts in Open Class

For the 2D Graphics framework, the default coordinate system positions the origin (0,0) in the upper-left corner of the view. (This coordinate system was called "left-handed" in previous releases of this class. The alternate coordinate system, with the origin located in the lower-left corner of the view, was called "right-handed.") Dual coordinate systems and multiple mapping modes are supported by using IExtendedRootGrafPort which is a subclass of IBaseRootGrafPort. For example:

```
EDeviceCoordinateSystem coordSystem;
if (ICoordinateSystem::applicationOrientation() == ICoordinateSystem::originLowerLeft)
{
  // traditional Cartesian coordinate system (used in OS/2 GPI)
  coordSystem = ICoordinateSystem::kOriginLowerLeft;
}
else
{
  coordSystem = ICoordinateSystem::kOriginUpperLeft;
}
// IExtendedRootGrafPort with either
// left-hand or right-hand coordinates
IExtendedRootGrafPort port(presSpace, coordSystem);
```

The enum type EDeviceMappingMode defines the modes of device mapping: kPixel, kLowMetric, kHighMetric, kLowEnglish, kHighEnglish, kTwips.

By default, IExtendedRootGrafPort and IBaseRootGrafPort are pixel-based. All graphics on screen, except text, are drawn in pixels when an IExtendedRootGrafPort or IBaseRootGrafPort is used. When using either of these GrafPorts, text is always drawn in World Coordinate units.

RELATED CONCEPTS

2D Rendering Pipeline
"GrafPorts" on page 58
Rendering Pipeline Class Summary

Render 2D Graphics

Use IMDrawable

In order to support IMDrawable hierarchies, the parent's state must be passed into the IMDrawable::draw() call so that the child can concatenate its state with the parent's:

```
void draw (IGrafPort& parentPort)
```

An IMDrawable typically concatenates its local state (for example, bundle, matrix) with that of the parent port via one or more ILinkedGrafPort classes. It then calls one or more draw functions on the IGrafPort.

IMDrawable subclasses that need more specialized behavior have to define extra semantics to support this behavior.

RELATED CONCEPTS

2D Rendering Pipeline

RELATED TASKS

Access Graphic State
Draw Directly to the Device
"Draw a Simple IMDrawable IPolygon" on page 62
"Draw More Complex Forms" on page 63
"Draw to the Screen" on page 63

Access Graphic State

Member functions allow you to access graphic state:

```
enum EMatrixKind{kView, kModel}
const IGrafMatrix* matrix(EMatrixKind) const
const IGArea* clipArea() const
const IAttributeState* attributes() const
```

This function returns the device type as a string. For a given IGrafPort, you can determine what kind of device the IGrafPort encapsulates:

```
const IString* portType() const
```

The constructors are protected because IGrafPort is an abstract base class:

```
protected: IGrafPort()
```

The functions useState and doneWithState are to be used for concurrency model of IGrafPort implementation. The subclass can override these methods to implement multi-thread safe IGrafPort:

```
virtual void useState() const = 0
virtual void doneWithState() const = 0
```

RELATED CONCEPTS

2D Rendering Pipeline

RELATED TASKS

"Use IMDrawable" on page 61
Draw Directly to the Device
"Draw a Simple IMDrawable IPolygon"
"Draw More Complex Forms" on page 63
"Draw to the Screen" on page 63

Draw Directly to the Device

The following pseudocode is an example of how an application can use this approach to create a red line with the 2D Graphics framework:

```
{
  // ... create a displayPort - an instance of IGrafPort...
  // Creates a line
  IGLine2D line(IGPoint2D(0.0, 0.0), IGPoint2D(1.0, 1.0));
  // Creates a red color bundle
  IGrafBundle redColor(IBasicColor(1.0, 0.0, 0.0));
  //Render the line on to the GrafPort
  displayPort->draw(line, redColor);
}
```

RELATED CONCEPTS

2D Rendering Pipeline

RELATED TASKS

"Use IMDrawable" on page 61
Access Graphic State
"Draw a Simple IMDrawable IPolygon"
"Draw More Complex Forms" on page 63
"Draw to the Screen" on page 63

RELATED REFERENCES

IGLine2D
IGPoint2D
IGrafBundle
IGrafPort

Draw a Simple IMDrawable IPolygon

Simple IMDrawables such as IPolygon need only specify the attributes contained in the IGrafBundle class (draw operation, fill and frame paints, and so on). The draw call of IPolygon is shown below.

```
IPolygon::draw (IGrafPort& parent) const
{
  ILinkedAttributePort bundlePort (&parent, fBundle);
  bundlePort.draw (fPolygon);
}
```

Since IPolygon makes only one IGrafPort::draw call, it can use the more convenient form of the draw call that takes the overriding bundle parameter:

```
IPolygon::draw (const IGrafPort& parent) const
{
  parent.draw (fPolygon, *fBundle);
}
```

RELATED CONCEPTS

2D Rendering Pipeline

RELATED TASKS

"Use IMDrawable" on page 61
Access Graphic State
Draw Directly to the Device
"Draw More Complex Forms"
"Draw to the Screen"

Draw More Complex Forms

To draw a graphic that has its own clip area, you can define your own class and its draw function as follows:

```
IGraphicWithClipArea::draw (IGrafPort& parent) const
{
  ILinkedClipPort clipPort (&parent, fClipArea);
  clipPort.draw (fPolygon);
  clipPort.draw (fEllipse);
}
```

The graphic consists of a polygon and an ellipse.

To draw an IMDrawable that requires concatenation of multiple objects, for example, an IGrafBundle object and a model matrix, you can define your own class and its draw function as follows:

```
IGraphicWithBundleAndMatrix::draw (IGrafPort& parent) const
{
  ILinkedMatrixPort matrixPort (&parent, fMatrix);
  ILinkedBundlePort bundlePort (&matrixPort, &fBundle);
  bundlePort.draw (fPolygon);
  bundlePort.draw (fEllipse);
}
```

To create a graphic with more than one bundle (multiple linked ports that use the same parent port), you can define your own class and its draw function as follows:

```
IGraphicWithBundlesAndMatrix::draw (IGrafPort& parent) const
{
  ILinkedMatrixPort matrixPort (&parent, fMatrix);
  ILinkedBundlePort redBundlePort (&matrixPort, fRedBundle);
  ILinkedBundlePort greenBundlePort (&matrixPort,
                                     fGreenBundle);
  redBundlePort.draw (fPolygon);
  greenBundlePort.draw (fEllipse);
}
```

RELATED CONCEPTS

2D Rendering Pipeline

RELATED TASKS

"Use IMDrawable" on page 61
Access Graphic State
Draw Directly to the Device
"Draw a Simple IMDrawable IPolygon" on page 62
"Draw to the Screen"

Draw to the Screen

To draw directly to the screen, use the following:

```
// Drawing geometries directly to the screen in different colors
// This GrafPort contains a device cache and default values for
// all graphic state
IRootGrafPort rootPort (&screenDevice);
rootPort.draw (myRect, redBundle);
rootPort.draw (myCurve, greenBundle, curveMatrix);
rootPort.draw (myArea);
```

Draw to the Screen with Clip Area

To draw directly to the screen using nested clip areas and specified coordinate systems, use the following:

```
// Drawing geometries to the screen using a different
// coordinate system and clip area
// This GrafPort contains a device cache and default values
// for all graphic state
IRootGrafPort rootPort (&screenDevice);
// This GrafPort overrides the clip area; rootPort is its parent
ILinkedClipPort clippedPort (&rootPort, myClipArea);
// clipped to the area; bundle overridden
clippedPort.draw (myRect, yellowBundle);
// unclipped; bundle & matrix overridden
rootPort.draw (myLoop, weirdBundle, someMatrix);
// clipped; using rootPort's bundle & matrix
clippedPort.draw (myEllipse);
// You can draw IMDrawables, too!
myGraphic.draw (rootPort); // unclipped
myOtherGraphic.draw (clippedPort); // clipped
```

RELATED CONCEPTS

2D Rendering Pipeline

RELATED TASKS

"Use IMDrawable" on page 61
Access Graphic State
Draw Directly to the Device
"Draw a Simple IMDrawable IPolygon" on page 62
"Draw More Complex Forms" on page 63

Rendering Pipeline Class Summary

The table summarizes the 2D Graphics framework rendering classes:

Class	Description
IGrafPort	An abstract base class that provides device-independent drawing protocols. The drawing port channels that information to a graphics device.
IBaseRootGrafPort	A subclass of IGrafPort that designates the top port of a port hierarchy.
IExtendedRootGrafPort	A special class that defines the coordinate system and mapping mode of the graphic device.
ILinkedGrafPort	A class that provides basic functionality for linked ports.
IStatefulGrafPort	A special class derived from IGrafPort that provides the ability to set the value of the graphic states individually.

RELATED CONCEPTS

2D Rendering Pipeline
"GrafPorts" on page 58
"Special GrafPorts in Open Class" on page 60

Chapter 6. 2D Transformations

IGrafMatrix implements the full set of 2D linear transformations including pseudo-perspective. Most users of graphic transforms use IGrafMatrix as ready-made classes. In addition to implementing transformations, IGrafMatrix implements a number of matrix utilities including concatenation, inversion, computing the determinant and transposing the matrix. IGrafMatrix provides the following:

- **Matrix transformations**: identity, translation, scaling, rotation, affine, and perspective.
- **Matrix elements**: concatenation, mathematical operations, and perspective mapping.

You can also transform graphics using the IMGraphic transformation functions described in "2D Graphics Framework."

The 2D Graphics framework uses transformations to transform points, rational points, and vectors. A vector has magnitude and direction but not location. Vectors can only be scaled, rotated, or sheared from the origin. Points and rational points have location, and they can be translated. They have an extra homogeneous coordinate "w". The extra coordinate allows affine transformations such as translation, and rotation around an arbitrary point. Points have an implied w == 1.0.

IGrafMatrix provides the following features:

- Batch transformations.
- Queries, to see if the pipeline can take advantage of fast rendering algorithms.
- Accelerators to speed up typical rendering transformation.
- Caching.

IGrafMatrix is not a general math package. It does not solve sets of linear equations or eigenvalue problems, and it does not have arbitrary dimension. It is designed for the kinds of linear transformations (where the coordinate space is modified linearly) needed by graphics, geometries, and the rendering pipeline.

Useful Information
For general discussions of points, rational points, and vectors, see "Mathematical Elements for Computer Graphics," section 2-15 "Translation and Homogeneous Coordinates."

For information on perspective mapping, see Paul Heckbert's UC Berkeley Masters Thesis: "Fundamentals of Texture Mapping and Image Warping," Appendix A.2 "Inferring Projective Mappings from 2-D Quadrilaterals."

RELATED CONCEPTS

"Transformation Matrices" on page 68
"Matrix Transformations" on page 69
"Matrix Concatenation" on page 71
"Perspective Map Transformations" on page 72
"Matrix Elements" on page 74
"Transformations Class Summary" on page 75

Transformation Matrices

IGrafMatrix defines a 2D transformation matrix. A 2D transformation matrix is an array of numbers with three rows and three columns for performing algebraic operations on a set of homogeneous coordinate points (regular points, rational points, or vectors) that define a 2D graphic.

IGrafMatrix transforms points in homogeneous space using rational points represented by the IGRPoint class. In homogeneous space, the third IGRPoint coordinate (fW) allows translation, rotation around an arbitrary point, scaling around an arbitrary point, perspective transformations, and shear transformations in addition to rotating and scaling around the origin through matrix multiplication. IGPoints have an implied fW of 1.0. Scaling a homogeneous matrix by a scalar does not change how it transforms an IGPoint; however, scaling a homogenous matrix by a scalar does change how it transforms an IGRPoint because an IGRPoint exists in homogeneous space.

IGrafMatrix implements the full set of 2-D linear affine transformations (translate, rotate, and scale) and supports concatenation (multiplying one matrix by another matrix). You can derive from IGrafMatrix to define your own transformation matrix implementation. The following table lists the IGrafMatrix member functions. All functions have default implementations that you do not need to override.

Constructors
IGrafMatrix functions can be called directly. The default constructor creates an identity matrix. The other constructors create rotate, translate, and scale matrices. In most cases, you can avoid having to specify or know about the numbers in the matrix. One of the constructors takes a specified array of numbers; and the matrix, element, and setElement functions provide access to the matrix.

IGrafMatrix Inquiry Functions
The following table describes the IGrafMatrix inquiry functions:

Member function	Action
matrix	Fills in a 9-element array of GCoordinates.
rotate	Returns true iff (if and only if) the matrix is a rotation matrix. If the matrix is a rotation matrix, rotate fills in the angle between 0 and 360 degrees, and the center of rotation.
type	Returns iff the matrix is identity, translate, scale around a point, rotate around a point, affine, or perspective.
isIdentity	Returns true if the matrix is identity.
isTranslate	Returns true iff the matrix only translates points.
isAffine	Returns true iff perspectiveX==0, perspectiveY==0 and homogeneous==1. A rectangle passed through an affine matrix becomes a parallelogram.
isRectilinear	Returns true iff a rectangle passed through the matrix is still a rectangle.

IGrafMatrix Mathematical Operations
The following table describes the IGrafMatrix mathematical operations:

Member function	Action
invert	Allows a matrix to reverse the effect it has on points. For example, if a transform moves a point from (0,0) to (1,1), the inverse transformation will move a point from (1,1) back to (0,0). If the transform cannot be inverted, throws an exception and does not change the destination.
untransformPoint	Is a shortcut method to invert on the fly
determinant	Gets the determinant. If the determinant is 0, a matrix is not invertible.
makeAdjoint	Converts the matrix to its "Classical Adjoint."
transpose	Swaps the rows and columns of a matrix.
normalize	Divides the matrix by the homogeneous value. Normalize raises an eUndefinedTransform ITransformException if the homogeneous value is 0.0.

RELATED CONCEPTS

2D Transformations
"Matrix Transformations"
"Matrix Concatenation" on page 71
"Perspective Map Transformations" on page 72
"Matrix Elements" on page 74
"Transformations Class Summary" on page 75

Matrix Transformations

IGrafMatrix has a set of calls to support the common graphics transformations: identity, translate, scale and rotate. An empty constructor creates an identity matrix. These calls allow you to:

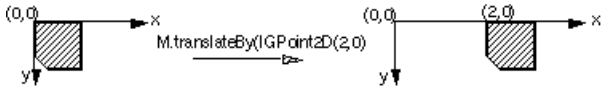
- Construct translate, scale and rotate matrices.
- Apply a translate, scale, or rotate to the matrix (relative transformation). Relative transformations are shortcuts for concatenating the matrix with a translate, scale or rotate matrix.
- Set a matrix to one of these transformations identity, translate, scale and rotate (absolute transformation).

To set up a translate, rotate, or scaling matrix, call the translateBy, rotateBy, and scaleBy functions. To set up a matrix that will, for example, first scale and then rotate the graphic, call the scaleBy and rotateBy functions in sequence. The matrix stores the cumulative transformation so you can apply one matrix to various graphics instead of scaling and rotating each graphic explicitly.

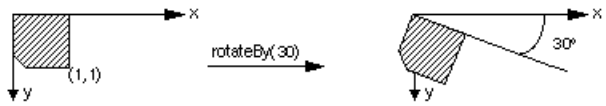
The setToRotate, setToScale, and setToTranslate functions set the matrix to rotate, scale, or translate, respectively. These functions cannot be called in sequence like rotateBy, scaleBy, and translateBy. If you call them in sequence, only the last call has any effect.

Identity
An identity transformation does nothing when applied to a set of points and vectors. It sets the destination to the source when you call transformPoint(s) or unTransformPoint.

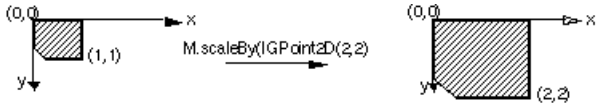
Translation
For translation, a delta point specifies the amount an object moves along the x and y axis. Function translateBy(IGPoint2D(2,0)) moves an object from the origin along the x axis by 2 units.



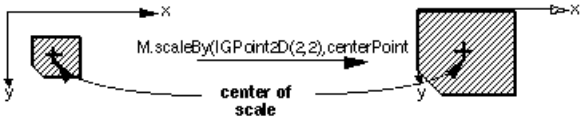
Rotation
The angle of rotation is in degrees and is measured from the positive x axis to the positive y axis. Rotation occurs around a point. The default rotation is about the origin.



Scaling
A point specifies scaling factors for x and y. A scale factor with x==y is a uniform scale. If a scale factor is greater than one, it enlarges the object. If the value is between zero and one, it shrinks the object. A scale factor that is less than zero mirrors an object.



Scaling is relative to the center of scale. That is, a point at the center doesn't change. Change increases the further a point is from the center. The default center of scale is the origin: (0,0).



RELATED CONCEPTS

2D Transformations
"Transformation Matrices" on page 68
"Matrix Concatenation" on page 71
"Perspective Map Transformations" on page 72
"Matrix Elements" on page 74
"Transformations Class Summary" on page 75

Matrix Concatenation

Matrices can be concatenated with other matrices. Concatenation is equivalent to sticking two matrices together such that a point first passes through one matrix and then the next during a transformation. The functions translateBy, scaleBy, and rotateBy are just concatenation shortcuts for translate, scale, and rotate matrices.

Think of concatenation as putting a transformation into a "list" of transformations. Later when transformPoint is called, the point is passed through each transform in order. For linear matrices, the "list" is compressed into a single matrix.

Concatenation is not commutative, that is if A and B are matrices, then A times B is not necessarily equal to B times A. For example, rotation matrix times a scale matrix does not necessarily give the same result as a scale matrix times a rotation matrix even though the component matrices are identical. This means it matters where the new transformation is placed in the "list". For this reason the 2D Graphics framework defines the notion of pre-multiplication and post-multiplication. Every function has a corresponding pre-multiply method with a prefix "pre" in front of it. The function that does not have the prefix has an implicit meaning of post-multiplication. The preRotateBy, preScaleBy, and preTranslateBy functions pre-multiply the specified transformation matrix by whatever values exist in the parameter list.

To create a sequence of arbitrary transformations rather than just translation, scaling, and rotation, use the concatWith function. Function concatWith concatenates two transformation matrices so that during the transformation each point passes first through one matrix and then the next. Function concatWith places the specified transformation matrix after the existing matrix.

For speed, the two concatenated matrices are formed into one matrix internally.

You can use the preConcatWith function to place the specified transformation matrix before the existing matrix.

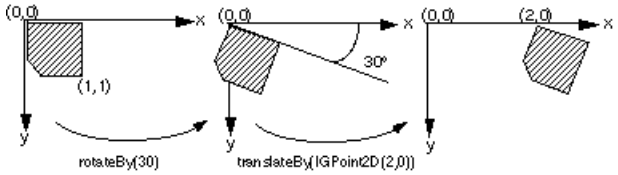
RELATED CONCEPTS

2D Transformations
"Transformation Matrices" on page 68
"Matrix Transformations" on page 69
"Perspective Map Transformations" on page 72
"Matrix Elements" on page 74
"Transformations Class Summary" on page 75

Concatenate Matrices

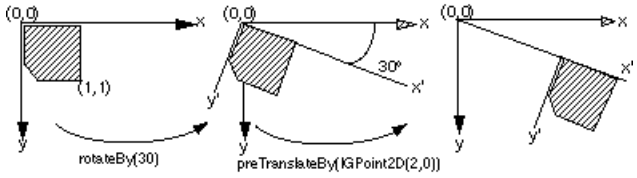
Complex transformations are often constructed with a series of transformations. It is important to understand that the two types of multiplication, pre-multiply and post-multiply, if applied to a sequence of transformations, usually give two different results. The two examples below demonstrate the process of transforming an object with a concatenated matrix. The first example shows that matrix is post-rotated and then post-translated.

```
matrix.setToIdentity();
matrix.rotateBy(30);
matrix.translateBy(IGPoint2D(2,0));
```



The same transformation operations are rotated and then pre-translated.

```
matrix.setToIdentity();
matrix.rotateBy(30);
matrix.preTranslateBy(IGPoint2D(2,0));
```



Notice that the resulting images are very different for the two examples. Post-multiplication is used to manipulate an object in a fixed space. Pre-multiplication is used to manipulate the space itself (such as the windowing system).

RELATED CONCEPTS

2D Transformations

RELATED TASKS

"Use Perspective Mapping" on page 73"Access Matrix Elements" on page 75

Perspective Map Transformations

You can define a transformation matrix by specifying what a convex quadrilateral (four-sided polygon) looks like before and after the transformation. This technique utilizes a perspective map. A perspective map defines a linear transformation from a source convex quadrilateral to a destination quadrilateral.

You can do the following:

- Use the IGrafMatrix::setToPerspectiveMap function to create a perspective map to define all of the standard transformations
- Define a translation matrix by specifying a source polygon and a destination polygon that is the same as the source, but in a different (x, y) position
- Define a scaling matrix by making the destination polygon larger or smaller than the source
- Define a rotation matrix by rotating the destination polygon.

Function setToPerspectiveMap sets the matrix to a perspective mapping from a source polygon to a destination polygon. The source and destination polygons must be convex, and they must have four vertices (quadrilateral). It creates the following perspective mapping:

fromQuadrilateral [0] —> toQuadrilateral [0]
fromQuadrilateral [1] —> toQuadrilateral [1]

fromQuadrilateral [2] —> toQuadrilateral [2]
fromQuadrilateral [3] —> toQuadrilateral [3]

You can use a perspective map to define all of the standard transformations:

- Moving the quadrilateral defines translation.
- Making the quadrilateral larger or smaller defines scale.
- Rotating the quadrilateral defines rotation.

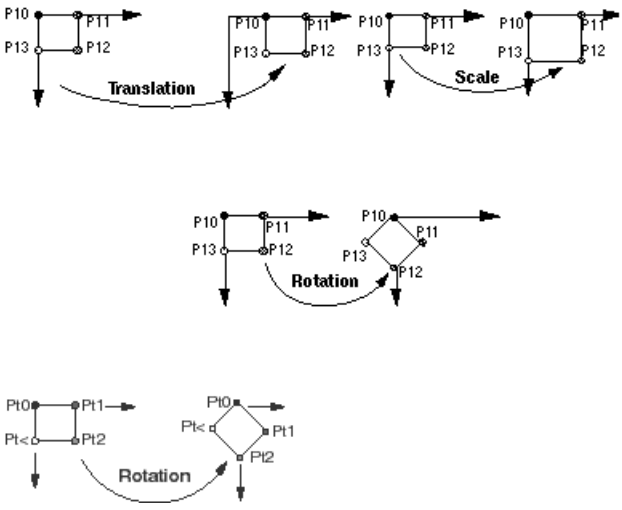
You can describe affine and perspective transformations. You can even easily describe shear transformations.

RELATED CONCEPTS

2D Transformations
"Transformation Matrices" on page 68
"Matrix Transformations" on page 69
"Matrix Concatenation" on page 71
"Matrix Elements" on page 74
"Transformations Class Summary" on page 75

Use Perspective Mapping

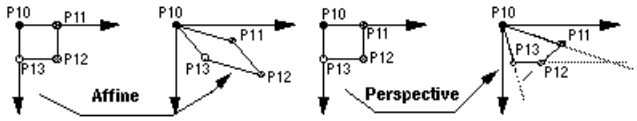
Standard Transformations
The figures show the standard transformations:



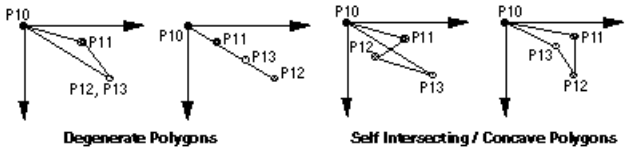
Affine and Perspective Transformations
Perspective mapping is particularly useful to describe arbitrary affine transformations and perspective transformations. Any concatenated sequence of translation, scaling, and rotation produces a result in which parallel lines are preserved, but angles and line segment lengths may or may not be preserved. If the source and destination polygons are parallelograms, the resulting matrix is affine. If the sides are not parallel, the resulting matrix is perspective with vanishing point at the point of intersection.

The real power of this method is in the fact that you can use it to describe arbitrary affine transformations and perspective transformations. If the source and

destination are parallelograms, the resulting matrix is affine. If the sides are not parallel, the matrix is a perspective matrix, and the point of intersection is the "vanishing point"



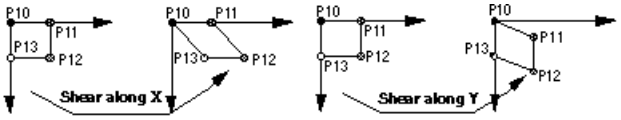
Function setToPerspectiveMap raises an exception if the source or destination polygons are invalid. It raises an exception if the source polygon is degenerate (two vertices are co-linear or equal).



Perspective transform can be represented by IGrafMatrix object in this implementation but cannot be applied when rendering geometries because the hardware support for such transform is not available.

Shearing Transformation
A shear transformation is an affine transformation where the x coordinate of every point is changed in proportion to its y coordinate. Vectors cannot be translated; they can only be sheared.

All matrix elements are of enumerated type EMatrixIndex and can be accessed with the element and setElement functions. You can use the setElement function to set up a shearing matrix. Once you set up the shear matrix, you can call setToPerspective to create a perspective transformation. The figure shows perspective mapping for a shear transformation:



RELATED CONCEPTS

2D Transformations

RELATED TASKS

"Concatenate Matrices" on page 71
"Use Perspective Mapping" on page 73"Access Matrix Elements" on page 75

Matrix Elements

A linear matrix is defined by the coefficients in a set of linear equations. For the two-dimensional case the matrix looks like this:

```
                | scaleX     shearY     perspectiveX |
(x,y,w) *       | shearX     scaleY     perspectiveY |  = (x',y',w')
                | translateX translateY homogeneous  |
```

Where:

```
x' = x*scaleX + y*shearX + w*translateX
y' = x*shearY + y*scaleY + w*translateY
w' = x*perspectiveX + y*perspectiveY + w*homogeneous
```

Matrix Elements are intended for advanced users only.

RELATED CONCEPTS

2D Transformations
"Transformation Matrices" on page 68
"Matrix Transformations" on page 69
"Matrix Concatenation" on page 71
"Perspective Map Transformations" on page 72
"Transformations Class Summary"

Access Matrix Elements

You can get access to matrix elements in a number of ways:

- Create an array of nine GCoordinates. The enumeration EMatrixIndex addresses the elements in the array, then construct an IGrafMatrix with this array, or you can call matrix to fill in the array, then
- Construct the matrix directly with the nine elements.
- Get and set individual elements with element and setElement functions. They allow you to logically access the "elements" of the array without forcing the matrix to actually store the information.

The code below shows the setElement function and the declaration for EMatrixIndex. Applying the matrix to a rational point (x, y, w) results in a rational point (x′, y′, w′) using matrix multiplication:

```
setElement( EMatrixIndex index, GCoordinate& element );
EMatrixIndex{
              kScaleX = 0,     kSheary = 1,     kPerspectiveX = 2,
              kShearX = 3,     kScaly = 4,      kPerspectivey = 5,
              kTranslateX = 6, kTranslatey = 7, kHomogeneous = 8
};
```

RELATED CONCEPTS

2D Transformations

RELATED TASKS

"Concatenate Matrices" on page 71
"Use Perspective Mapping" on page 73

Transformations Class Summary

This table summarizes the 2D Graphics Framework transformation classes.

Class	Description
IGrafMatrix	Base classes that provide transformation matrices to move, rotate, and translate graphics and geometries.

Class	Description
IGQuadrilateral	Class that specifies the four-sided polygon used to define perspective and affine mapping.

RELATED CONCEPTS

2D Transformations
"Transformation Matrices" on page 68
"Matrix Transformations" on page 69
"Matrix Concatenation" on page 71
"Perspective Map Transformations" on page 72
"Matrix Elements" on page 74