Manual for Object Graphics Library 3.0

Julian Smart

September 1998

Contents

1 Introduction

Object Graphics Library (OGL) is a C++ library supporting the creation and manipulation of simple and complex graphic images on a canvas.

It can be found in the directory `utils/ogl/src` in the wxWindows distribution. The file `ogl.h` must be included to make use of the library.

Please see *OGL overview* (p. 63) for a general description how the object library works. For details, please see the *class reference* (p. 4).

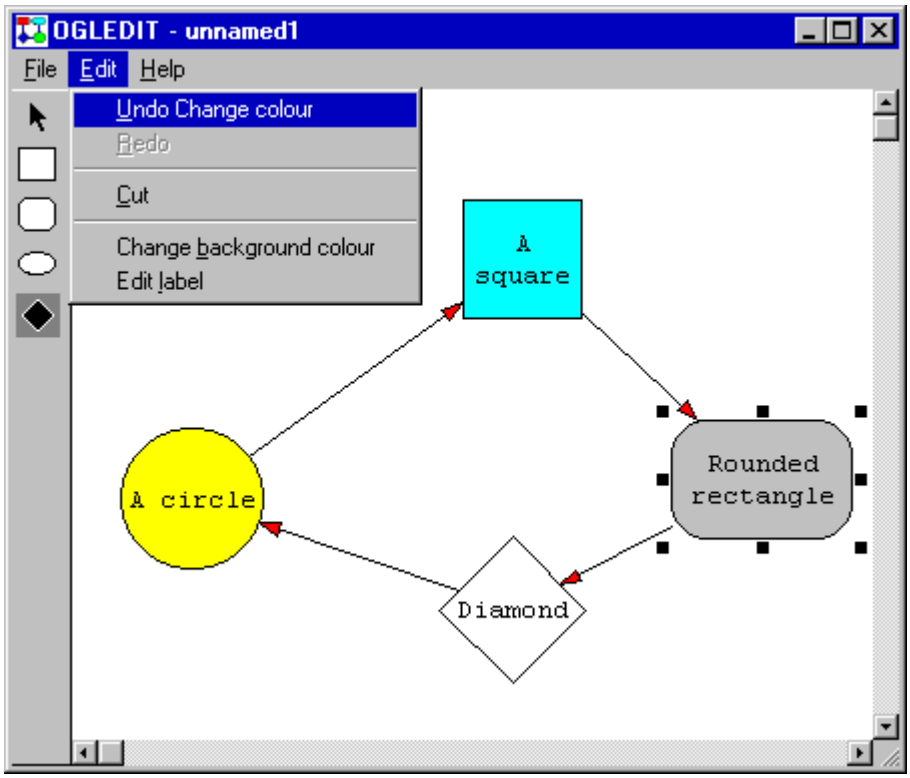
1.1 File structure

These are the files that comprise the OGL library.

basic.h Header for basic objects such as wxShape and wxRectangleShape.
basic.cpp Basic objects implementation (1).
basic2.cpp Basic objects implementation (2).
bmpshape.h wxBitmapShape class header.
bmpshape.cpp wxBitmapShape implementation.
canvas.h wxShapeCanvas class header.
canvas.cpp wxShapeCanvas class implementation.
composit.h Composite object class header.
composit.cpp Composite object class implementation.
constrnt.h Constraint classes header.
constrnt.cpp Constraint classes implementation.
divided.h Divided object class header.
divided.cpp Divided object class implementation.
drawn.h Drawn (metafile) object class header.
drawn.cpp Drawn (metafile) object class implementation.
graphics.h Main include file.
lines.h wxLineShape class header.
lines.cpp wxLineShape class implementation.
misc.h Miscellaneous graphics functions header.
misc.cpp Miscellaneous graphics functions implementation.
ogldiag.h wxDiagram class header.
ogldiag.cpp wxDiagram implementation.
mfutils.h Metafile utilities header.
mfutils.cpp Metafile utilities implementation.

2 OGLEdit: a sample OGL application

OGLEdit is a sample OGL application that allows the user to draw, edit, save and load a few shapes. It should clarify aspects of OGL usage, and can act as a template for similar applications. OGLEdit can be found in `samples/ogledit` in the OGL distribution.



The wxWindows document/view model has been used in OGL, to reduce the amount of housekeeping logic required to get it up and running. OGLEdit also provides a demonstration of the Undo/Redo capability supported by the document/view classes, and how a typical application might implement this feature.

2.1 OGLEdit files

OGLEdit comprises the following source files.

- doc.h, doc.cpp: MyDiagram, DiagramDocument, DiagramCommand, MyEvtHandler classes related to diagram functionality and documents.
- view.h, view.cpp: MyCanvas, DiagramView classes related to visualisation of the diagram.
- ogledit.h, ogledit.cpp: MyFrame, MyApp classes related to the overall application.
- palette.h, palette.cpp: EditorToolPalette implementing the shape palette.

2.2 How OGLEdit works

OGLEdit defines a DiagramDocument class, each of instance of which holds a MyDiagram member which itself contains the shapes.

In order to implement specific mouse behaviour for shapes, a class MyEvtHandler is defined which is 'plugged into' each shape when it is created, instead of overriding each shape class individually. This event handler class also holds a label string.

The DiagramCommand class is the key to implementing Undo/Redo. Each instance of DiagramCommand stores enough information about an operation (create, delete, change colour etc.) to allow it to carry out (or undo) its command. In DiagramView::OnMenuCommand, when the user initiates the command, a new DiagramCommand instance is created which is then sent to the document's command processor (see wxWindows manual for more information about doc/view and command processing).

Apart from menu commands, another way commands are initiated is by the user left-clicking on the canvas or right-dragging on a node. MyCanvas::OnLeftClick in view.cpp shows how the appropriate wxClassInfo is passed to a DiagramCommand, to allow DiagramCommand::Do to create a new shape given the wxClassInfo.

The MyEvtHandler right-drag methods in doc.cpp implement drawing a line between two shapes, detecting where the right mouse button was released and looking for a second shape. Again, a new DiagramCommand instance is created and passed to the command processor to carry out the command.

DiagramCommand::Do and DiagramCommand::Undo embody much of the interesting interaction with the OGL library. A complication of note when implementing undo is the problem of deleting a node shape which has one or more arcs attached to it. If you delete the node, the arc(s) should be deleted too. But multiple arc deletion represents more information that can be incorporated in the existing DiagramCommand scheme. OGLEdit copes with this by treating each arc deletion as a separate command, and sending Cut commands recursively, providing an undo path. Undoing such a Cut will only undo one command at a time - not a one to one correspondence with the original command - but it's a reasonable compromise and preserves Do/Undo whilst keeping our DiagramCommand class simple.

2.3 Possible enhancements

OGLEdit is very simplistic and does not employ the more advanced features of OGL, such as:

- attachment points (arcs are drawn to particular points on a shape)
- metafile and bitmaps shapes
- divided rectangles
- composite shapes, and constraints
- creating labels in shape regions
- arc labels (OGL has support for three movable labels per arc)
- spline and multiple-segment line arcs
- adding annotations to node and arc shapes
- line-straightening (supported by OGL) and alignment (not supported directly by OGL)

These could be added to OGLEdit, at the risk of making it a less useful example for beginners.

3 Class reference

These are the main OGL classes.

3.1 wxOGLConstraint

See also *wxCompositeShape overview* (p. 64)

An wxOGLConstraint object helps specify how child shapes are laid out with respect to siblings and parents.

Derived from

wxObject

See also

wxCompositeShape (p. 17)

3.1.1 wxOGLConstraint::wxOGLConstraint

wxOGLConstraint()

Default constructor.

wxOGLConstraint(**int** *type*, **wxShape ***_constraining_, **wxList&** *constrained*)

Constructor.

Parameters

constraining
The shape which is used as the reference for positioning the *constrained* objects.

constrained
Contains a list of wxShapes which are to be constrained (with respect to *constraining*) using *type*.

type
Can be one of:

- **gyCONSTRAINT_CENTRED_VERTICALLY**: the Y co-ordinates of the centres of the bounding boxes of the constrained objects and the constraining object will be the same
- **gyCONSTRAINT_CENTRED_HORIZONTALLY**: the X co-ordinates of the centres of the bounding boxes of the constrained objects and the constraining object will be the same
- **gyCONSTRAINT_CENTRED_BOTH**: the co-ordinates of the centres of the bounding boxes of the constrained objects and the constraining object will be the same

- **gyCONSTRAINT_LEFT_OF**: the X co-ordinates of the right hand vertical edges of the bounding boxes of the constrained objects will be less than the X co-ordinate of the left hand vertical edge of the bounding box of the constraining object
- **gyCONSTRAINT_RIGHT_OF**: the X co-ordinates of the left hand vertical edges of the bounding boxes of the constrained objects will be greater than the X co-ordinate of the right hand vertical edge of the bounding box of the constraining object
- **gyCONSTRAINT_ABOVE**: the Y co-ordinates of the bottom horizontal edges of the bounding boxes of the constrained objects will be less than the Y co-ordinate of the top horizontal edge of the bounding box of the constraining object
- **gyCONSTRAINT_BELOW**: the Y co-ordinates of the top horizontal edges of the bounding boxes of the constrained objects will be greater than the X co-ordinate of the bottom horizontal edge of the bounding box of the constraining object
- **gyCONSTRAINT_ALIGNED_TOP**: the Y co-ordinates of the top horizontal edges of the bounding boxes of the constrained objects will be the same as the Y co-ordinate of the top horizontal edge of the bounding box of the constraining object
- **gyCONSTRAINT_ALIGNED_BOTTOM**: the Y co-ordinates of the bottom horizontal edges of the bounding boxes of the constrained objects will be the same as the Y co-ordinate of the bottom horizontal edge of the bounding box of the constraining object
- **gyCONSTRAINT_ALIGNED_LEFT**: the X co-ordinates of the left hand vertical edges of the bounding boxes of the constrained objects will be the same as the X co-ordinate of the left hand vertical edge of the bounding box of the constraining object
- **gyCONSTRAINT_ALIGNED_RIGHT**: the X co-ordinates of the right hand vertical edges of the bounding boxes of the constrained objects will be the same as the X co-ordinate of the right hand vertical edge of the bounding box of the constraining object
- **gyCONSTRAINT_MIDALIGNED_TOP**: the Y co-ordinates of the centres of the bounding boxes of the constrained objects will be the same as the Y co-ordinate of the top horizontal edge of the bounding box of the constraining object
- **gyCONSTRAINT_MIDALIGNED_BOTTOM**: the Y co-ordinates of the centres of the bounding boxes of the constrained objects will be the same as the Y co-ordinate of the bottom horizontal edge of the bounding box of the constraining object
- **gyCONSTRAINT_MIDALIGNED_LEFT**: the X co-ordinates of the centres of the bounding boxes of the constrained objects will be the same as the X co-ordinate of the left hand vertical edge of the bounding box of the constraining object
- **gyCONSTRAINT_MIDALIGNED_RIGHT**: the X co-ordinates of the centres of the bounding boxes of the constrained objects will be the same as the X co-ordinate of the right hand vertical edge of the bounding box of the constraining object

3.1.2 wxOGLConstraint::~wxOGLConstraint

~wxOGLConstraint()

Destructor.

3.1.3 wxOGLConstraint::Equals

bool Equals(**double** *x*, **double** *y*)

Returns TRUE if *x* and *y* are approximately equal (for the purposes of evaluating the constraint).

3.1.4 wxOGLConstraint::Evaluate

bool Evaluate()

Evaluates this constraint, returning TRUE if anything changed.

3.1.5 wxOGLConstraint::SetSpacing

void SetSpacing(**double** *x*, **double** *y*)

Sets the horizontal and vertical spacing for the constraint.

3.2 wxBitmapShape

Draws a bitmap (non-resizable).

Derived from

wxRectangleShape (p. 33)

3.2.1 wxBitmapShape::wxBitmapShape

wxBitmapShape()

Constructor.

3.2.2 wxBitmapShape::~wxBitmapShape

~wxBitmapShape()

Destructor.

3.2.3 wxBitmapShape::GetBitmap

wxBitmap& GetBitmap() **const**

Returns a reference to the bitmap associated with this shape.

3.2.4 wxBitmapShape::GetFilename

wxString GetFilename() **const**

Returns the bitmap filename.

3.2.5 wxBitmapShape::SetBitmap

void SetBitmap(**const wxBitmap&** *bitmap*)

Sets the bitmap associated with this shape. You can delete the bitmap from the calling application, since reference counting will take care of holding on to the internal bitmap data.

3.2.6 wxBitmapShape::SetFilename

void SetFilename(**const wxString&** *filename*)

Sets the bitmap filename.

3.3 wxDiagram

Encapsulates an entire diagram, with methods for reading/writing and drawing. A diagram has an associated wxShapeCanvas.

Derived from

wxObject

See also

wxShapeCanvas (p. 51)

3.3.1 wxDiagram::wxDiagram

wxDiagram()

Constructor.

3.3.2 wxDiagram::~wxDiagram

~wxDiagram()

Destructor.

3.3.3 wxDiagram::AddShape

void AddShape(**wxShape***shape*, **wxShape** **addAfter = NULL*)

Adds a shape to the diagram. If *addAfter* is non-NULL, the shape will be added after this one.

3.3.4 wxDiagram::Clear

void Clear(**wxDC&** *dc*)

Clears the specified device context.

3.3.5 wxDiagram::DeleteAllShapes

void DeletesAllShapes()

Removes and deletes all shapes in the diagram.

3.3.6 wxDiagram::DrawOutline

void DrawOutline(**wxDC&** *dc*, **double** *x1*, **double** *y1*, **double** *x2*, **double** *y2*)

Draws an outline rectangle on the current device context.

3.3.7 wxDiagram::FindShape

wxShape* FindShape(**long** *id*) **const**

Returns the shape for the given identifier.

3.3.8 wxDiagram::GetCanvas

wxShapeCanvas* GetCanvas() **const**

Returns the shape canvas associated with this diagram.

3.3.9 wxDiagram::GetCount

int GetCount() **const**

Returns the number of shapes in the diagram.

3.3.10 wxDiagram::GetGridSpacing

double GetGridSpacing() **const**

Returns the grid spacing.

3.3.11 wxDiagram::GetMouseTolerance

int GetMouseTolerance()

Returns the tolerance within which a mouse move is ignored.

3.3.12 wxDiagram::GetShapeList

wxList* GetShapeList() **const**

Returns a pointer to the internal shape list.

3.3.13 wxDiagram::GetQuickEditMode

bool GetQuickEditMode() **const**

Returns quick edit mode.

3.3.14 wxDiagram::GetSnapToGrid

bool GetSnapToGrid() **const**

Returns snap-to-grid mode.

3.3.15 wxDiagram::InsertShape

void InsertShape(**wxShape ***shape*)

Inserts a shape at the front of the shape list.

3.3.16 wxDiagram::LoadFile

bool LoadFile(**const wxString&** *filename*)

Loads the diagram from a file.

3.3.17 wxDiagram::OnDatabaseLoad

void OnDatabaseLoad(**wxExprDatabase&** *database*)

Called just after the nodes and lines have been read from the wxExprDatabase. You may override this; the default member does nothing.

3.3.18 wxDiagram::OnDatabaseSave

void OnDatabaseSave(**wxExprDatabase&** *database*)

Called just after the nodes and lines have been written to the wxExprDatabase. You may override this; the default member does nothing.

3.3.19 wxDiagram::OnHeaderLoad

bool OnHeaderLoad(**wxExprDatabase&** *database*, **wxExpr&** *expr*)

Called to allow the 'diagram' header object to be read. The default member reads no further information. You may wish to override this to read version information, author name, etc.

3.3.20 wxDiagram::OnHeaderSave

bool OnHeaderSave(**wxExprDatabase&** *database*, **wxExpr&** *expr*)

Called to allow instantiation of the 'diagram' header object. The default member writes no further information. You may wish to override this to include version information, author name, etc.

3.3.21 wxDiagram::OnShapeLoad

bool OnShapeLoad(**wxExprDatabase&** *database*, **wxShape&** *shape*, **wxExpr&** *expr*)

Called to read the shape from the *expr*. You may override this, but call this function first. The default member calls ReadAttributes for the shape.

3.3.22 wxDiagram::OnShapeSave

bool OnShapeSave(**wxExprDatabase&** *database*, **wxShape&** *shape*, **wxExpr&** *expr*)

Called to save the shape to the *expr* and *database*. You may override this, but call this function first. The default member calls WriteAttributes for the shape, appends the shape to the database, and of the shape is a composite, recursively calls OnShapeSave for its children.

3.3.23 wxDiagram::ReadContainerGeometry

void ReadContainerGeometry(**wxExprDatabase&** *database*)

Reads container geometry from a wxExprDatabase, linking up nodes which are part of a composite. You probably won't need to redefine this.

3.3.24 wxDiagram::ReadLines

void ReadLines(**wxExprDatabase&** *database*)

Reads lines from a wxExprDatabase. You probably won't need to redefine this.

3.3.25 wxDiagram::ReadNodes

void ReadNodes(**wxExprDatabase&** *database*)

Reads nodes from a wxExprDatabase. You probably won't need to redefine this.

3.3.26 wxDiagram::RecentreAll

void RecentreAll(**wxDC&** *dc*)

Make sure all text that should be centred, is centred.

3.3.27 wxDiagram::Redraw

void Redraw(**wxDC&** *dc*)

Draws the shapes in the diagram on the specified device context.

3.3.28 wxDiagram::RemoveAllShapes

void RemoveAllShapes()

Removes all shapes from the diagram but does not delete the shapes.

3.3.29 wxDiagram::RemoveShape

void RemoveShape(**wxShape*** *shape*)

Removes the shape from the diagram (non-recursively) but does not delete it.

3.3.30 wxDiagram::SaveFile

bool SaveFile(**const wxString&** *filename*)

Saves the diagram in a file.

3.3.31 wxDiagram::SetCanvas

void SetCanvas(**wxShapeCanvas*** *canvas*)

Sets the canvas associated with this diagram.

3.3.32 wxDiagram::SetGridSpacing

void SetGridSpacing(**double** *spacing*)

Sets the grid spacing. The default is 5.

3.3.33 wxDiagram::SetMouseTolerance

void SetMouseTolerance(**int** *tolerance*)

Sets the tolerance within which a mouse move is ignored. The default is 3 pixels.

3.3.34 wxDiagram::SetQuickEditMode

void SetQuickEditMode(**bool** *mode*)

Sets quick-edit-mode on or off. In this mode, refreshes are minimized, but the diagram may need manual refreshing occasionally.

3.3.35 wxDiagram::SetSnapToGrid

void SetSnapToGrid(**bool** *snap*)

Sets snap-to-grid mode on or off. The default is on.

3.3.36 wxDiagram::ShowAll

void ShowAll(**bool** *show*)

Calls Show for each shape in the diagram.

3.3.37 wxDiagram::Snap

void Snap(**double ***x*, **double ***y*)

'Snaps' the coordinate to the nearest grid position, if snap-to-grid is on.

3.4 wxDrawnShape

Draws a pseduo-metafile shape, which can be loaded from a simple Windows metafile.

wxDrawnShape allows you to specify a different shape for each of four orientations (North, West, South and East). It also provides a set of drawing functions for programmatic drawing of a shape, so that during construction of the shape you can draw into it as if it were a device context.

Derived from

wxRectangleShape (p. 33)

See also *wxRectangleShape* (p. 33).

3.4.1 wxDrawnShape::wxDrawnShape

wxDrawnShape()

Constructor.

3.4.2 wxDrawnShape::~wxDrawnShape

~wxDrawnShape()

Destructor.

3.4.3 wxDrawnShape::CalculateSize

void CalculateSize()

Calculates the wxDrawnShape size from the current metafile. Call this after you have drawn into the shape.

3.4.4 wxDrawnShape::DestroyClippingRect

void DestroyClippingRect()

Destroys the clipping rectangle. See also *wxDrawnShape::SetClippingRect* (p. 15).

3.4.5 wxDrawnShape::DrawArc

void DrawArc(**const wxPoint&** *centrePoint*, **const wxPoint&** *startPoint*, **const wxPoint&** *endPoint*)

Draws an arc (see wxWindows documentation for details).

3.4.6 wxDrawnShape::DrawAtAngle

void DrawAtAngle(**int** *angle*)

Sets the metafile for the given orientation, which can be one of:

- oglDRAWN_ANGLE_0
- oglDRAWN_ANGLE_90
- oglDRAWN_ANGLE_180
- oglDRAWN_ANGLE_270

See also *wxDrawnShape::GetAngle* (p. 15).

3.4.7 wxDrawnShape::DrawEllipticArc

void DrawEllipticArc(**const wxRect&** *rect*, **double** *startAngle*, **double** *endAngle*)

Draws an elliptic arc (see wxWindows documentation for details).

3.4.8 wxDrawnShape::DrawLine

void DrawLine(**const wxPoint&** *point1*, **const wxPoint&** *point2*)

Draws a line from *point1* to *point2*.

3.4.9 wxDrawnShape::DrawLines

void DrawLines(**int** *n*, **wxPoint&** *points[]*)

Draws *n* lines.

3.4.10 wxDrawnShape::DrawPoint

void DrawPoint(**const wxPoint&** *point*)

Draws a point.

3.4.11 wxDrawnShape::DrawPolygon

void DrawPolygon(**int** *n*, **wxPoint&** *points[]*, **int** *flags = 0*)

Draws a polygon. *flags* can be one or more of **oglMETAFLAGS_OUTLINE** (use this polygon for the drag outline) and **oglMETAFLAGS_ATTACHMENTS** (use the vertices of this polygon for attachments).

3.4.12 wxDrawnShape::DrawRectangle

void DrawRectangle(**const wxRect&** *rect*)

Draws a rectangle.

3.4.13 wxDrawnShape::DrawRoundedRectangle

void DrawRoundedRectangle(**const wxRect&** *rect*, **double** *radius*)

Draws a rounded rectangle. *radius* is the corner radius. If *radius* is negative, it expresses the radius as a proportion of the smallest dimension of the rectangle.

3.4.14 wxDrawnShape::DrawSpline

void DrawSpline(**int** *n*, **wxPoint&** *points[]*)

Draws a spline curve.

3.4.15 wxDrawnShape::DrawText

void DrawText(**const wxString&** *text*, **const wxPoint&** *point*)

Draws text at the given point.

3.4.16 wxDrawnShape::GetAngle

int GetAngle() **const**

Returns the current orientation, which can be one of:

- oglDRAWN_ANGLE_0
- oglDRAWN_ANGLE_90
- oglDRAWN_ANGLE_180
- oglDRAWN_ANGLE_270

See also *wxDrawnShape::DrawAtAngle* (p. 13).

3.4.17 wxDrawnShape::GetMetaFile

wxPseudoMetaFile& GetMetaFile() **const**

Returns a reference to the internal 'pseudo-metafile'.

3.4.18 wxDrawnShape::GetRotation

double GetRotation() **const**

Returns the current rotation of the shape in radians.

3.4.19 wxDrawnShape::LoadFromMetaFile

bool LoadFromMetaFile(**const wxString&** *filename*)

Loads a (very simple) Windows metafile, created for example by Top Draw, the Windows shareware graphics package.

3.4.20 wxDrawnShape::Rotate

void Rotate(**double** *x*, **double** *y*, **double** *theta*)

Rotate about the given axis by the given amount in radians.

3.4.21 wxDrawnShape::SetClippingRect

void SetClippingRect(**const wxRect&** *rect*)

Sets the clipping rectangle. See also *wxDrawnShape::DestroyClippingRect* (p. 13).

3.4.22 wxDrawnShape::SetDrawnBackgroundColour

void SetDrawnBackgroundColour(**const wxColour&** *colour*)

Sets the current background colour for the current metafile.

3.4.23 wxDrawnShape::SetDrawnBackgroundMode

void SetDrawnBackgroundMode(**int** *mode*)

Sets the current background mode for the current metafile.

3.4.24 wxDrawnShape::SetDrawnBrush

void SetDrawnBrush(**wxPen*** *pen*, **bool** *isOutline = FALSE*)

Sets the pen for this metafile. If *isOutline* is TRUE, this pen is taken to indicate the outline (and if the outline pen is changed for the whole shape, the pen will be replaced with the outline pen).

3.4.25 wxDrawnShape::SetDrawnFont

void SetDrawnFont(**wxFont*** *font*)

Sets the current font for the current metafile.

3.4.26 wxDrawnShape::SetDrawnPen

void SetDrawnPen(**wxPen*** *pen*, **bool** *isOutline = FALSE*)

Sets the pen for this metafile. If *isOutline* is TRUE, this pen is taken to indicate the outline (and if the outline pen is changed for the whole shape, the pen will be replaced with the outline pen).

3.4.27 wxDrawnShape::SetDrawnTextColour

void SetDrawnTextColour(**const wxColour&** *colour*)

Sets the current text colour for the current metafile.

3.4.28 wxDrawnShape::Scale

void Scale(**double** *sx*, **double** *sy*)

Scales the shape by the given amount.

3.4.29 wxDrawnShape::SetSaveToFile

void SetSaveToFile(**bool** *save*)

If *save* is TRUE, the image will be saved along with the shape's other attributes. The reason why this might not be desirable is that if there are many shapes with the same image, it would be more efficient for the application to save one copy, and not duplicate the information for every

shape. The default is TRUE.

3.4.30 wxDrawnShape::Translate

void Translate(**double** *x*, **double** *y*)

Translates the shape by the given amount.

3.5 wxCircleShape

An wxEllipseShape whose width and height are the same.

Derived from

wxEllipseShape (p. 25).

3.5.1 wxCircleShape::wxCircleShape

wxCircleShape(**double** *width = 0.0*)

Constructor.

3.5.2 wxCircleShape::~wxCircleShape

~wxCircleShape()

Destructor.

3.6 wxCompositeShape

This is an object with a list of child objects, and a list of size and positioning constraints between the children.

Derived from

wxRectangleShape (p. 33)

See also

wxCompositeShape overview (p. 64)

3.6.1 wxCompositeShape::wxCompositeShape

wxCompositeShape()

Constructor.

3.6.2 wxCompositeShape::~wxCompositeShape

~wxCompositeShape()

Destructor.

3.6.3 wxCompositeShape::AddChild

void AddChild(**wxShape ***child*, **wxShape ***addAfter = NULL*)

Adds a child shape to the composite. If *addAfter* is non-NULL, the shape will be added after this shape.

3.6.4 wxCompositeShape::AddConstraint

wxOGLConstraint * AddConstraint(**wxOGLConstraint ***constraint*)

wxOGLConstraint * AddConstraint(**int** *type*, **wxShape ***constraining*, **wxList&***constrained*)

wxOGLConstraint * AddConstraint(**int** *type*, **wxShape ***constraining*, **wxShape ***constrained*)

Adds a constraint to the composite.

3.6.5 wxCompositeShape::CalculateSize

void CalculateSize()

Calculates the size and position of the composite based on child sizes and positions.

3.6.6 wxCompositeShape::ContainsDivision

bool FindContainerImage(**wxDivisionShape ***division*)

Returns TRUE if *division* is a descendant of this container.

3.6.7 wxCompositeShape::DeleteConstraint

void DeleteConstraint(**wxOGLConstraint ***constraint*)

Deletes constraint from composite.

3.6.8 wxCompositeShape::DeleteConstraintsInvolvingChild

void DeleteConstraintsInvolvingChild(**wxShape ***child*)

This function deletes constraints which mention the given child. Used when deleting a child from the composite.

3.6.9 wxCompositeShape::FindConstraint

wxOGLConstraint * FindConstraint(**long** *id*, **wxCompositeShape ****_actualComposite_)

Finds the constraint with the given id, also returning the actual composite the constraint was in, in case that composite was a descendant of this composite.

3.6.10 wxCompositeShape::FindContainerImage

wxShape * FindContainerImage()

Finds the image used to visualize a container. This is any child of the composite that is not in the divisions list.

3.6.11 wxCompositeShape::GetConstraints

wxList& GetConstraints() **const**

Returns a reference to the list of constraints.

3.6.12 wxCompositeShape::GetDivisions

wxList& GetDivisions() **const**

Returns a reference to the list of divisions.

3.6.13 wxCompositeShape::MakeContainer

void MakeContainer()

Makes this composite into a container by creating one child wxDivisionShape.

3.6.14 wxCompositeShape::OnCreateDivision

wxDivisionShape * OnCreateDivision()

Called when a new division shape is required. Can be overriden to allow an application to use a different class of division.

3.6.15 wxCompositeShape::Recompute

bool Recompute()

Recomputes any constraints associated with the object. If FALSE is returned, the constraints could not be satisfied (there was an inconsistency).

3.6.16 wxCompositeShape::RemoveChild

void RemoveChild(**wxShape ***_child_)

Removes the child from the composite and any constraint relationships, but does not delete the child.

3.7 wxDividedShape

A wxDividedShape is a rectangle with a number of vertical divisions. Each division may have its text formatted with independent characteristics, and the size of each division relative to the whole image may be specified.

Derived from

wxRectangleShape (p. 33)

See also

wxDividedShape overview (p. 63)

3.7.1 wxDividedShape::wxDividedShape

wxDividedShape(**double** *width = 0.0*, **double** *height = 0.0*)

Constructor.

3.7.2 wxDividedShape::~wxDividedShape

~wxDividedShape()

Destructor.

3.7.3 wxDividedShape::EditRegions

void EditRegions()

Edit the region colours and styles.

3.7.4 wxDividedShape::SetRegionSizes

void SetRegionSizes()

Set all region sizes according to proportions and this object total size.

3.8 wxDivisionShape

A division shape is like a composite in that it can contain further objects, but is used exclusively to divide another shape into regions, or divisions. A wxDivisionShape is never free-standing.

Derived from

wxCompositeShape (p. 17)

See also

wxCompositeShape overview (p. 64)

3.8.1 wxDivisionShape::wxDivisionShape

wxDivisionShape()

Constructor.

3.8.2 wxDivisionShape::~wxDivisionShape

~wxDivisionShape()

Destructor.

3.8.3 wxDivisionShape::AdjustBottom

void AdjustBottom(**double** *bottom*, **bool** *test*)

Adjust a side, returning FALSE if it's not physically possible to adjust it to this point.

3.8.4 wxDivisionShape::AdjustLeft

void AdjustLeft(**double** *left*, **bool** *test*)

Adjust a side, returning FALSE if it's not physically possible to adjust it to this point.

3.8.5 wxDivisionShape::AdjustRight

void AdjustRight(**double** *right*, **bool** *test*)

Adjust a side, returning FALSE if it's not physically possible to adjust it to this point.

3.8.6 wxDivisionShape::AdjustTop

void AdjustTop(**double** *top*, **bool** *test*)

Adjust a side, returning FALSE if it's not physically possible to adjust it to this point.

3.8.7 wxDivisionShape::Divide

void Divide(**int** *direction*)

Divide this division into two further divisions, horizontally (*direction* is wxHORIZONTAL) or vertically (*direction* is wxVERTICAL).

3.8.8 wxDivisionShape::EditEdge

void EditEdge(**int** *side*)

Interactively edit style of left or top side.

3.8.9 wxDivisionShape::GetBottomSide

wxDivisionShape * GetBottomSide()

Returns a pointer to the division on the bottom side of this division.

3.8.10 wxDivisionShape::GetHandleSide

int GetHandleSide()

Returns the side which the handle appears on (DIVISION_SIDE_LEFT or DIVISION_SIDE_TOP).

3.8.11 wxDivisionShape::GetLeftSide

wxDivisionShape * GetLeftSide()

Returns a pointer to the division on the left side of this division.

3.8.12 wxDivisionShape::GetLeftSideColour

wxString GetLeftSideColour()

Returns a pointer to the colour used for drawing the left side of the division.

3.8.13 wxDivisionShape::GetLeftSidePen

wxPen * GetLeftSidePen()

Returns a pointer to the pen used for drawing the left side of the division.

3.8.14 wxDivisionShape::GetRightSide

wxDivisionShape * GetRightSide()

Returns a pointer to the division on the right side of this division.

3.8.15 wxDivisionShape::GetTopSide

wxDivisionShape * GetTopSide()

Returns a pointer to the division on the top side of this division.

3.8.16 wxDivisionShape::GetTopSideColour

wxString GetTopSideColour()

Returns a pointer to the colour used for drawing the top side of the division.

3.8.17 wxDivisionShape::GetTopSidePen

wxPen * GetTopSidePen()

Returns a pointer to the pen used for drawing the left side of the division.

3.8.18 wxDivisionShape::ResizeAdjoining

void ResizeAdjoining(**int** *side*, **double** *newPos*, **bool** *test*)

Resize adjoining divisions at the given side. If *test* is TRUE, just see whether it's possible for each adjoining region, returning FALSE if it's not.

side can be one of:

- DIVISION_SIDE_NONE
- DIVISION_SIDE_LEFT
- DIVISION_SIDE_TOP
- DIVISION_SIDE_RIGHT
- DIVISION_SIDE_BOTTOM

3.8.19 wxDivisionShape::PopupMenu

void PopupMenu(**double** *x*, **double** *y*)

Popup the division menu.

3.8.20 wxDivisionShape::SetBottomSide

void SetBottomSide(**wxDivisionShape ***shape*)

Set the pointer to the division on the bottom side of this division.

3.8.21 wxDivisionShape::SetHandleSide

int SetHandleSide()

Sets the side which the handle appears on (DIVISION_SIDE_LEFT or DIVISION_SIDE_TOP).

3.8.22 wxDivisionShape::SetLeftSide

void SetLeftSide(**wxDivisionShape ***shape*)

Set the pointer to the division on the left side of this division.

3.8.23 wxDivisionShape::SetLeftSideColour

void SetLeftSideColour(**const wxString&** *colour*)

Sets the colour for drawing the left side of the division.

3.8.24 wxDivisionShape::SetLeftSidePen

void SetLeftSidePen(**wxPen ***pen*)

Sets the pen for drawing the left side of the division.

3.8.25 wxDivisionShape::SetRightSide

void SetRightSide(**wxDivisionShape ***shape*)

Set the pointer to the division on the right side of this division.

3.8.26 wxDivisionShape::SetTopSide

void SetTopSide(**wxDivisionShape ***shape*)

Set the pointer to the division on the top side of this division.

3.8.27 wxDivisionShape::SetTopSideColour

void SetTopSideColour(**const wxString&** *colour*)

Sets the colour for drawing the top side of the division.

3.8.28 wxDivisionShape::SetTopSidePen

void SetTopSidePen(**wxPen ***pen*)

Sets the pen for drawing the top side of the division.

3.9 wxEllipseShape

The wxEllipseShape behaves similarly to the wxRectangleShape but is elliptical.

Derived from

wxShape (p. 33)

3.9.1 wxEllipseShape::wxEllipseShape

wxEllipseShape(**double** *width = 0.0*, **double** *height = 0.0*)

Constructor.

3.9.2 wxEllipseShape::~wxEllipseShape

~wxEllipseShape()

Destructor.

3.10 wxLineShape

A wxLineShape may be attached to two nodes; it may be segmented, in which case a control point is drawn for each joint.

A wxLineShape may have arrows at the beginning, end and centre.

Derived from

wxShape (p. 33)

3.10.1 wxLineShape::wxLineShape

wxLineShape()

Constructor.

Usually you will call *wxLineShape::MakeLineControlPoints* (p. 30) to specify the number of segments in the line.

3.10.2 wxLineShape::~wxLineShape

~wxLineShape()

Destructor.

3.10.3 wxLineShape::AddArrow

void AddArrow(**WXTYPE** *type*, **bool** *end = ARROW_POSITION_END*, **double** *arrowSize = 10.0*, **double** *xOffset = 0.0*, **const wxString&** *name = ""*, **wxPseudoMetaFile ***_mf = NULL_, **long** *arrowId = -1*)

Adds an arrow (or annotation) to the line.

type may currently be one of:

ARROW_HOLLOW_CIRCLE Hollow circle.
ARROW_FILLED_CIRCLE Filled circle.
ARROW_ARROW Conventional arrowhead.
ARROW_SINGLE_OBLIQUE Single oblique stroke.
ARROW_DOUBLE_OBLIQUE Double oblique stroke.
ARROW_DOUBLE_METAFILE Custom arrowhead.

end may currently be one of:

ARROW_POSITION_END Arrow appears at the end.
ARROW_POSITION_START Arrow appears at the start.

arrowSize specifies the length of the arrow.

xOffset specifies the offset from the end of the line.

name specifies a name for the arrow.

mf can be a wxPseduoMetaFile, perhaps loaded from a simple Windows metafile.

arrowId is the id for the arrow.

3.10.4 wxLineShape::AddArrowOrdered

void AddArrowOrdered(**wxArrowHead ***_arrow_, **wxList&** *referenceList*, **int** *end*)

Add an arrowhead in the position indicated by the reference list of arrowheads, which contains all legal arrowheads for this line, in the correct order. E.g.

```
Reference list:      a b c d e
Current line list:   a d
```

Add c, then line list is: a c d.

If no legal arrowhead position, return FALSE. Assume reference list is for one end only, since it potentially defines the ordering for any one of the 3 positions. So we don't check the reference list for arrowhead position.

3.10.5 wxLineShape::ClearArrow

bool ClearArrow(**const wxString&** *name*)

Delete the arrow with the given name.

3.10.6 wxLineShape::ClearArrowsAtPosition

void ClearArrowsAtPosition(**int** *position = -1*)

Delete the arrows at the specified position, or at any position if *position* is -1.

3.10.7 wxLineShape::DrawArrow

void DrawArrow(**ArrowHead ***_arrow_, **double** *xOffset*, **bool** *proportionalOffset*)

Draws the given arrowhead (or annotation).

3.10.8 wxLineShape::DeleteArrowHead

bool DeleteArrowHead(**long** *arrowId*)

bool DeleteArrowHead(**int** *position*, **const wxString&** *name*)

Delete arrowhead by id or position and name.

3.10.9 wxLineShape::DeleteLineControlPoint

bool DeleteLineControlPoint()

Deletes an arbitary point on the line.

3.10.10 wxLineShape::DrawArrows

void DrawArrows(**wxDC&** *dc*)

Draws all arrows.

3.10.11 wxLineShape::DrawRegion

void DrawRegion(**wxDC&** *dc*, **wxShapeRegion ***_region_, **double** *x*, **double** *y*)

Format one region at this position.

3.10.12 wxLineShape::EraseRegion

void EraseRegion(**wxDC&** *dc*, **wxShapeRegion ***_region_, **double** *x*, **double** *y*)

Format one region at this position.

3.10.13 wxLineShape::FindArrowHead

wxArrowHead * FindArrowHead(**long** *arrowId*)

wxArrowHead * FindArrowHead(**int** *position*, **const wxString&** *name*)

Find arrowhead by id or position and name.

3.10.14 wxLineShape::FindLineEndPoints

void FindLineEndPoints(**double ***fromX*, **double ***fromY*, **double ***toX*, **double ***toY*)

Finds the x, y points at the two ends of the line. This function can be used by e.g. line-routing routines to get the actual points on the two node images where the lines will be drawn to/from.

3.10.15 wxLineShape::FindLinePosition

int FindLinePosition(**double** *x*, **double** *y*)

Find which position we're talking about at this x, y. Returns ARROW_POSITION_START, ARROW_POSITION_MIDDLE, ARROW_POSITION_END.

3.10.16 wxLineShape::FindMinimumWidth

double FindMinimumWidth()

Finds the horizontal width for drawing a line with arrows in minimum space. Assume arrows at end only.

3.10.17 wxLineShape::FindNth

void FindNth(**wxShape ***image*, **int ***nth*, **int ***noArcs*, **bool** *incoming*)

Finds the position of the line on the given object. Specify whether incoming or outgoing lines are being considered with *incoming*.

3.10.18 wxLineShape::GetAttachmentFrom

int GetAttachmentFrom() **const**

Returns the attachment point on the 'from' node.

3.10.19 wxLineShape::GetAttachmentTo

int GetAttachmentTo() **const**

Returns the attachment point on the 'to' node.

3.10.20 wxLineShape::GetEnds

void GetEnds(**double ****x1*, **double ****y1*, **double ****x2*, **double ****y2*)

Gets the visible endpoints of the lines for drawing between two objects.

3.10.21 wxLineShape::GetFrom

wxShape * GetFrom() **const**

Gets the 'from' object.

3.10.22 wxLineShape::GetLabelPosition

void GetLabelPosition(**int** *position*, **double ****x*, **double ****y*)

Get the reference point for a label. Region x and y are offsets from this. position is 0 (middle), 1 (start), 2 (end).

3.10.23 wxLineShape::GetNextControlPoint

wxPoint * GetNextControlPoint(**wxShape ****shape*)

Find the next control point in the line after the start/end point, depending on whether the shape is at the start or end.

3.10.24 wxLineShape::GetTo

wxShape * GetTo()

Gets the 'to' object.

3.10.25 wxLineShape::Initialise

void Initialise()

Initialises the line object.

3.10.26 wxLineShape::InsertLineControlPoint

void InsertLineControlPoint()

Inserts a control point at an arbitrary position.

3.10.27 wxLineShape::IsEnd

bool IsEnd(**wxShape ***_shape_)

Returns TRUE if _shape_ is at the end of the line.

3.10.28 wxLineShape::IsSpline

bool IsSpline()

Returns TRUE if a spline is drawn through the control points, and FALSE otherwise.

3.10.29 wxLineShape::MakeLineControlPoints

void MakeLineControlPoints(**int** _n_)

Make a given number of control points (minimum of two).

3.10.30 wxLineShape::OnMoveLink

void OnMoveLink(**wxDC&** _dc_, **bool** _moveControlPoints = TRUE_)

Called when a connected object has moved, to move the link to correct position.

3.10.31 wxLineShape::SetAttachmentFrom

void SetAttachmentTo(**int** _fromAttach_)

Sets the 'from' shape attachment.

3.10.32 wxLineShape::SetAttachments

void SetAttachments(**int** _fromAttach_, **int** _toAttach_)

Specifies which object attachment points should be used at each end of the line.

3.10.33 wxLineShape::SetAttachmentTo

void SetAttachmentTo(**int** _toAttach_)

Sets the 'to' shape attachment.

3.10.34 wxLineShape::SetEnds

void SetEnds(**double** _x1_, **double** _y1_, **double** _x2_, **double** _y2_)

Sets the end positions of the line.

3.10.35 wxLineShape::SetFrom

void SetFrom(**wxShape ***_object_)

Sets the 'from' object for the line.

3.10.36 wxLineShape::SetIgnoreOffsets

void SetIgnoreOffsets(**bool** _ignore_)

Tells the shape whether to ignore offsets from the end of the line when drawing.

3.10.37 wxLineShape::SetSpline

void SetSpline(**bool** _spline_)

Specifies whether a spline is to be drawn through the control points (TRUE), or a line (FALSE).

3.10.38 wxLineShape::SetTo

void SetTo(**wxShape ***_object_)

Sets the 'to' object for the line.

3.10.39 wxLineShape::Straighten

void Straighten(**wxDC*** _dc = NULL_)

Straighten verticals and horizontals. _dc_ is optional.

3.10.40 wxLineShape::Unlink

void Unlink()

Unlinks the line from the nodes at either end.

3.11 wxPolygonShape

A wxPolygonShape's shape is defined by a number of points passed to the object's constructor. It can be used to create new shapes such as diamonds and triangles.

Derived from

wxShape (p. 33)

3.11.1 wxPolygonShape::wxPolygonShape

wxPolygonShape(void)

Constructor. Call *wxPolygonShape::Create* (p. 32) to specify the polygon's vertices.

3.11.2 wxPolygonShape::~wxPolygonShape

~wxPolygonShape()

Destructor.

3.11.3 wxPolygonShape::Create

void Create(**wxList*** *points*)

Takes a list of wxRealPoints; each point is an *offset* from the centre. The polygon's destructor will delete these points, so do not delete them yourself.

3.11.4 wxPolygonShape::AddPolygonPoint

void AddPolygonPoint(**int** *pos = 0*)

Add a control point after the given point.

3.11.5 wxPolygonShape::CalculatePolygonCentre

void CalculatePolygonCentre()

Recalculates the centre of the polygon.

3.11.6 wxPolygonShape::DeletePolygonPoint

void DeletePolygonPoint(**int** *pos = 0*)

Deletes a control point.

3.11.7 wxPolygonShape::GetPoints

wxList * GetPoints()

Returns a pointer to the internal list of polygon vertices (wxRealPoints).

3.11.8 wxPolygonShape::UpdateOriginalPoints

void UpdateOriginalPoints()

If we've changed the shape, must make the original points match the working points with this function.

3.12 wxRectangleShape

The wxRectangleShape has rounded or square corners.

Derived from

wxShape (p. 33)

3.12.1 wxRectangleShape::wxRectangleShape

wxRectangleShape(**double** *width = 0.0*, **double** *height = 0.0*)

Constructor.

3.12.2 wxRectangleShape::~wxRectangleShape

~wxRectangleShape()

Destructor.

3.12.3 wxRectangleShape::SetCornerRadius

void SetCornerRadius(**double** *radius*)

Sets the radius of the rectangle's rounded corners. If the radius is zero, a non-rounded rectangle will be drawn. If the radius is negative, the value is the proportion of the smaller dimension of the rectangle.

3.13 wxPseudoMetaFile

A simple metafile-like class which can load data from a Windows metafile on all platforms.

Derived from

wxObject

3.14 wxShape

The wxShape is the top-level, abstract object that all other objects are derived from. All common functionality is represented by wxShape's members, and overriden members that appear in derived classes and have behaviour as documented for wxShape, are not documented separately.

Derived from

wxShapeEvtHandler (p. 56)

3.14.1 wxShape::wxShape

wxShape(**wxShapeCanvas*** *canvas = NULL*)

Constructs a new wxShape.

3.14.2 wxShape::~wxShape

~wxShape()

Destructor.

3.14.3 wxShape::AddLine

void AddLine(**wxLineShape*** *line*, **wxShape*** *other*, **int** *attachFrom = 0*, **int** *attachTo = 0*, **int** *positionFrom = -1*, **int** *positionTo = -1*)

Adds a line between the specified canvas shapes, at the specified attachment points.

The position in the list of lines at each end can also be specified, so that the line will be drawn at a particular point on its attachment point.

3.14.4 wxShape::AddRegion

void AddRegion(**wxShapeRegion*** *region*)

Adds a region to the shape.

3.14.5 wxShape::AddText

void AddText(**const wxString&** *string*)

Adds a line of text to the shape's default text region.

3.14.6 wxShape::AddToCanvas

void AddToCanvas(**wxShapeCanvas*** *theCanvas*, **wxShape*** *addAfter=NULL*)

Adds the shape to the canvas's shape list. If *addAfter* is non-NULL, will add the shape after this one.

3.14.7 wxShape::AncestorSelected

bool AncestorSelected() **const**

TRUE if the shape's ancestor is currently selected.

3.14.8 wxShape::ApplyAttachmentOrdering

void ApplyAttachmentOrdering(**wxList&** *linesToSort*)

Applies the line ordering in *linesToSort* to the shape, to reorder the way lines are attached.

3.14.9 wxShape::AssignNewIds

void AssignNewIds()

Assigns new ids to this image and its children.

3.14.10 wxShape::Attach

void Attach(**wxShapeCanvas*** *can*)

Sets the shape's internal canvas pointer to point to the given canvas.

3.14.11 wxShape::AttachmentIsValid

bool AttachmentIsValid(**int** *attachment*) **const**

Returns TRUE if *attachment* is a valid attachment point.

3.14.12 wxShape::AttachmentSortTest

bool AttachmentSortTest(**int** *attachment*, **const wxRealPoint&** *pt1*, **const wxRealPoint&** *pt2*) **const**

Returns TRUE if *pt1* is less than or equal to *pt2*, in the sense that one point comes before another on an edge of the shape. *attachment* is the attachment point (side) in question.

This function is used in *wxShape::MoveLineToNewAttachment* (p. 45) to determine the new line ordering.

3.14.13 wxShape::CalcSimpleAttachment

wxRealPoint CalcSimpleAttachment(**const wxRealPoint&** *pt1*, **const wxRealPoint&** *pt2*, **int** *nth*, **int** *noArcs*, **wxLineShape*** *line*)

Assuming the attachment lies along a vertical or horizontal line, calculates the position on that point.

Parameters

pt1

The first point of the line repesenting the edge of the shape.

pt2
The second point of the line representing the edge of the shape.

nth
The position on the edge (for example there may be 6 lines at this attachment point, and this may be the 2nd line.

noArcs
The number of lines at this edge.

line
The line shape.

Remarks

This function expects the line to be either vertical or horizontal, and determines which.

3.14.14 wxShape::CalculateSize

void CalculateSize()

Called to calculate the shape's size if dependent on children sizes.

3.14.15 wxShape::ClearAttachments

void ClearAttachments()

Clears internal custom attachment point shapes (of class wxAttachmentPoint).

3.14.16 wxShape::ClearRegions

void ClearRegions()

Clears the wxShapeRegions from the shape.

3.14.17 wxShape::ClearText

void ClearText(**int** *regionId = 0*)

Clears the text from the specified text region.

3.14.18 wxShape::Constrain

bool Constrain()

Calculates the shape's constraints (if any). Applicable only to wxCompositeShape, does nothing if the shape is of a different class.

3.14.19 wxShape::Copy

void Copy(**wxShape&** *copy*)

Copy this shape into *copy*. Every derived class must have one of these, and each Copy implementation must call the derived class's implementation to ensure everything is copied. See also *wxShape::CreateNewCopy* (p. 37).

3.14.20 wxShape::CreateNewCopy

wxShape* CreateNewCopy(**bool** *resetMapping = TRUE*, **bool** *recompute = TRUE*)

Creates and returns a new copy of this shape (calling *wxShape::Copy* (p. 37)). Do not override this function.

This function should always be used to create a new copy, since it must do special processing for copying constraints associated with constraints.

If *resetMapping* is TRUE, a mapping table used for complex shapes is reset; this may not be desirable if the shape being copied is a child of a composite (and so the mapping table is in use).

If *recompute* is TRUE, *wxShape::Recompute* (p. 46) is called for the new shape.

Remarks

This function uses the wxWindows dynamic object creation system to create a new shape of the same type as 'this', before calling Copy.

If the event handler for this shape is not the same as the shape itself, the event handler is also copied using *wxShapeEvtHandler::CreateNewCopy* (p. 57).

3.14.21 wxShape::DeleteControlPoints

void DeleteControlPoints()

Deletes the control points (or handles) for the shape. Does not redraw the shape.

3.14.22 wxShape::Detach

void Detach()

Disassociates the shape from its canvas by setting the internal shape canvas pointer to NULL.

3.14.23 wxShape::Draggable

bool Draggable()

TRUE if the shape may be dragged by the user.

3.14.24 wxShape::Draw

void Draw(**wxDC&** *dc*)

Draws the whole shape and any lines attached to it.

Do not override this function: override OnDraw, which is called by this function.

3.14.25 wxShape::DrawContents

void DrawContents(**wxDC&** *dc*)

Draws the internal graphic of the shape (such as text).

Do not override this function: override OnDrawContents, which is called by this function.

3.14.26 wxShape::DrawLinks

void DrawLinks(**wxDC&** *dc*, **int** *attachment = -1*)

Draws any lines linked to this shape.

3.14.27 wxShape::Erase

void Erase(**wxDC&** *dc*)

Erases the shape, but does not repair damage caused to other shapes.

3.14.28 wxShape::EraseContents

void EraseContents(**wxDC&** *dc*)

Erases the shape contents, that is, the area within the shape's minimum bounding box.

3.14.29 wxShape::EraseLinks

void EraseLinks(**wxDC&** *dc*, **int** *attachment = -1*)

Erases links attached to this shape, but does not repair damage caused to other shapes.

3.14.30 wxShape::FindRegion

wxShape * FindRegion(**const wxString&** *regionName*, **int ***regionId*)

Finds the actual image ('this' if non-composite) and region id for the given region name.

3.14.31 wxShape::FindRegionNames

void FindRegionNames(**wxStringList&** *list*)

Finds all region names for this image (composite or simple). Supply an empty string list.

3.14.32 wxShape::Flash

void Flash()

Flashes the shape.

3.14.33 wxShape::FormatText

void FormatText(**const wxString&** *s*, **int** *i = 0*)

Reformats the given text region; defaults to formatting the default region.

3.14.34 wxShape::GetAttachmentMode

bool GetAttachmentMode() **const**

Returns the attachment mode, which is TRUE if attachments are used, FALSE otherwise (in which case lines will be drawn as if to the centre of the shape). See *wxShape::SetAttachmentMode* (p. 47).

3.14.35 wxShape::GetAttachmentPosition

bool GetAttachmentPosition(**int** *attachment*, **double*** *x*, **double*** *y*, **int** *nth = 0*, **int** *noArcs = 1*, **wxLineShape*** *line = NULL*)

Gets the position at which the given attachment point should be drawn.

If *attachment* isn't found among the attachment points of the shape, returns FALSE.

3.14.36 wxShape::GetBoundingBoxMax

void GetBoundingBoxMax(**double ***width*, **double ***height*)

Gets the maximum bounding box for the shape, taking into account external features such as shadows.

3.14.37 wxShape::GetBoundingBoxMin

void GetBoundingBoxMin(**double ***width*, **double ***height*)

Gets the minimum bounding box for the shape, that defines the area available for drawing the contents (such as text).

3.14.38 wxShape::GetBrush

wxBrush* GetBrush() **const**

Returns the brush used for filling the shape.

3.14.39 wxShape::GetCanvas

wxShapeCanvas* GetCanvas() **const**

Gets the internal canvas pointer.

3.14.40 wxShape::GetCentreResize

bool GetCentreResize() **const**

Returns TRUE if the shape is to be resized from the centre (the centre stands still), or FALSE if from the corner or side being dragged (the other corner or side stands still).

3.14.41 wxShape::GetChildren

wxList& GetChildren() **const**

Returns a reference to the list of children for this shape.

3.14.42 wxShape::GetClientData

wxObject* GetClientData()

Gets the client data associated with the shape (NULL if there is none).

3.14.43 wxShape::GetDisableLabel

bool GetDisableLabel() **const**

Returns TRUE if the default region will not be shown, FALSE otherwise.

3.14.44 wxShape::GetEventHandler

wxShapeEvtHandler* GetEventHandler() **const**

Returns the event handler for this shape.

3.14.45 wxShape::GetFixedHeight

bool GetFixedHeight() **const**

Returns TRUE if the shape cannot be resized in the vertical plane.

3.14.46 wxShape::GetFixedSize

void GetFixedSize(**bool *** *x*, **bool *** *y*)

Returns flags indicating whether the shape is of fixed size in either direction.

3.14.47 wxShape::GetFixedWidth

bool GetFixedWidth() **const**

Returns TRUE if the shape cannot be resized in the horizontal plane.

3.14.48 wxShape::GetFont

wxFont* GetFont(**int** *regionId = 0*) **const**

Gets the font for the specified text region.

3.14.49 wxShape::GetFunctor

wxString GetFunctor() **const**

Gets a string representing the type of the shape, to be used when writing out shape descriptions to a file. This is overridden by each derived shape class to provide an appropriate type string. By default, "node_image" is used for non-line shapes, and "arc_image" for lines.

3.14.50 wxShape::GetId

long GetId() **const**

Returns the integer identifier for this shape.

3.14.51 wxShape::GetLinePosition

int GetLinePosition(**wxLineShape*** *line*)

Gets the zero-based position of *line* in the list of lines for this shape.

3.14.52 wxShape::GetLines

wxList& GetLines() **const**

Returns a reference to the list of lines connected to this shape.

3.14.53 wxShape::GetMaintainAspectRatio

bool GetMaintainAspectRatio() **const**

If returns TRUE, resizing the shape will not change the aspect ratio (width and height will be in the original proportion).

3.14.54 wxShape::GetNumberOfAttachments

int GetNumberOfAttachments() **const**

Gets the number of attachment points for this shape.

3.14.55 wxShape::GetNumberOfTextRegions

int GetNumberOfTextRegions() **const**

Gets the number of text regions for this shape.

3.14.56 wxShape::GetParent

wxShape * GetParent() **const**

Returns the parent of this shape, if it is part of a composite.

3.14.57 wxShape::GetPen

wxPen* GetPen() **const**

Returns the pen used for drawing the shape's outline.

3.14.58 wxShape::GetPerimeterPoint

bool GetPerimeterPoint(**double** *x1*, **double** *y1*, **double** *x2*, **double** *y2*, **double *** *x3*, **double *** *y3*)

Gets the point at which the line from (x1, y1) to (x2, y2) hits the shape. Returns TRUE if the line hits the perimeter.

3.14.59 wxShape::GetRegionId

int GetRegionId(**const wxString&** *name*)

Gets the region's identifier by name. This is *not* unique for within an entire composite, but is unique for the image.

3.14.60 wxShape::GetRegionName

wxString GetRegionName(**int** *regionId = 0*)

Gets the region's name. A region's name can be used to uniquely determine a region within an entire composite image hierarchy. See also *wxShape::SetRegionName* (p. 49).

3.14.61 wxShape::GetRegions

wxList& GetRegions()

Returns the list of wxShapeRegions.

3.14.62 wxShape::GetRotation

double GetRotatation() **const**

Returns the angle of rotation in radians.

3.14.63 wxShape::GetSensitivityFilter

void GetSensitivityFilter() **const**

Returns the sensitivity filter, a bitlist of values. See *wxShape::SetSensitivityFilter* (p. 49).

3.14.64 wxShape::GetShadowMode

int SetShadowMode() **const**

Returns the shadow mode. See *wxShape::SetShadowMode* (p. 50).

3.14.65 wxShape::GetSpaceAttachments

bool GetSpaceAttachments() **const**

Indicates whether lines should be spaced out evenly at the point they touch the node (TRUE), or whether they should join at a single point (FALSE).

3.14.66 wxShape::GetTextColour

wxString GetTextColour(**int** *regionId = 0*) **const**

Gets the colour for the specified text region.

3.14.67 wxShape::GetTopAncestor

wxShape * GetTopAncestor() **const**

Returns the top-most ancestor of this shape (the root of the composite).

3.14.68 wxShape::GetX

double GetX() **const**

Gets the x position of the centre of the shape.

3.14.69 wxShape::GetY

double GetY() **const**

Gets the y position of the centre of the shape.

3.14.70 wxShape::HitTest

bool HitTest(**double** *x*, **double** *y*, **int*** *attachment*, **double*** *distance*)

Given a point on a canvas, returns TRUE if the point was on the shape, and returns the nearest attachment point and distance from the given point and target.

3.14.71 wxShape::Insert

void InsertInCanvas(**wxShapeCanvas*** *canvas*)

Inserts the shape at the front of the shape list of *canvas*.

3.14.72 wxShape::IsHighlighted

bool IsHighlighted() **const**

Returns TRUE if the shape is highlighted. Shape highlighting is unimplemented.

3.14.73 wxShape::IsShown

bool IsShown() **const**

Returns TRUE if the shape is in a visible state, FALSE otherwise. Note that this has nothing to do with whether the window is hidden or the shape has scrolled off the canvas; it refers to the internal visibility flag.

3.14.74 wxShape::MakeControlPoints

void MakeControlPoints()

Make a list of control points (draggable handles) appropriate to the shape.

3.14.75 wxShape::MakeMandatoryControlPoints

void MakeMandatoryControlPoints()

Make the mandatory control points. For example, the control point on a dividing line should appear even if the divided rectangle shape's handles should not appear (because it is the child of a composite, and children are not resizable).

3.14.76 wxShape::Move

void Move(**wxDC&** *dc*, **double** *x1*, **double** *y1*, **bool** *display = TRUE*)

Move the shape to the given position, redrawing if *display* is TRUE.

3.14.77 wxShape::MoveLineToNewAttachment

void MoveLineToNewAttachment(**wxDC&** *dc*, **wxLineShape*** *toMove*, **double** *x*, **double** *y*)

Move the given line (which must already be attached to the shape) to a different attachment point on the shape, or a different order on the same attachment.

Cals *wxShape::AttachmentSortTest* (p. 35) and then *wxShapeEvtHandler::OnChangeAttachment* (p. 58).

3.14.78 wxShape::MoveLinks

void MoveLinks(**wxDC&** *dc*)

Redraw all the lines attached to the shape.

3.14.79 wxShape::NameRegions

void NameRegions(**const wxString&** *parentName = ""*)

Make unique names for all the regions in a shape or composite shape.

3.14.80 wxShape::Rotate

void Rotate(**double** *x*, **double** *y*, **double** *theta*)

Rotate about the given axis by the given amount in radians (does nothing for most shapes). But even non-rotating shapes should record their notional rotation in case it's important (e.g. in dog-leg code).

3.14.81 wxShape::ReadConstraints

void ReadConstraints(**wxExpr ***clause*, **wxExprDatabase ***database*)

If the shape is a composite, it may have constraints that need to be read in in a separate pass.

3.14.82 wxShape::ReadAttributes

void ReadAttributes(**wxExpr*** *clause*)

Reads the attributes (data member values) from the given expression.

3.14.83 wxShape::ReadRegions

void ReadRegions(**wxExpr *** *clause*)

Reads in the regions.

3.14.84 wxShape::Recentre

void Recentre()

Does recentring (or other formatting) for all the text regions for this shape.

3.14.85 wxShape::RemoveFromCanvas

void RemoveFromCanvas(**wxShapeCanvas*** *canvas*)

Removes the shape from the canvas.

3.14.86 wxShape::ResetControlPoints

void ResetControlPoints()

Resets the positions of the control points (for instance when the shape's shape has changed).

3.14.87 wxShape::ResetMandatoryControlPoints

void ResetMandatoryControlPoints()

Reset the mandatory control points. For example, the control point on a dividing line should appear even if the divided rectangle shape's handles should not appear (because it is the child of a composite, and children are not resizable).

3.14.88 wxShape::Recompute

bool Recompute()

Recomputes any constraints associated with the shape (normally applicable to wxCompositeShapes only, but harmless for other classes of shape).

3.14.89 wxShape::RemoveLine

void RemoveLine(**wxLineShape*** *line*)

Removes the given line from the shape's list of attached lines.

3.14.90 wxShape::Select

void Select(**bool** *select = TRUE*)

Selects or deselects the given shape, drawing or erasing control points (handles) as necessary.

3.14.91 wxShape::Selected

bool Selected() **const**

TRUE if the shape is currently selected.

3.14.92 wxShape::SetAttachmentMode

void SetAttachmentMode(**bool** *flag*)

Sets the attachment mode to TRUE or FALSE. If TRUE, attachment points will be significant when drawing lines to and from this shape. If FALSE, lines will be drawn as if to the centre of the shape.

3.14.93 wxShape::SetBrush

void SetBrush(**wxBrush ***brush*)

Sets the brush for filling the shape's shape.

3.14.94 wxShape::SetCanvas

void SetCanvas(**wxShapeCanvas*** *theCanvas*)

Identical to *wxShape::Attach* (p. 47).

3.14.95 wxShape::SetCentreResize

void SetCentreResize(**bool** *cr*)

Specify whether the shape is to be resized from the centre (the centre stands still) or from the corner or side being dragged (the other corner or side stands still).

3.14.96 wxShape::SetClientData

void SetClientData(**wxObject ***_clientData_)

Sets the client data.

3.14.97 wxShape::SetDefaultRegionSize

void SetDefaultRegionSize()

Set the default region to be consistent with the shape size.

3.14.98 wxShape::SetDisableLabel

void SetDisableLabel(**bool** _flag_)

Set _flag_ to TRUE to stop the default region being shown, FALSE otherwise.

3.14.99 wxShape::SetDraggable

void SetDraggable(**bool** _drag_, **bool** _recursive = FALSE_)

Sets the shape to be draggable or not draggable.

3.14.100 wxShape::SetDrawHandles

void SetDrawHandles(**bool** _drawH_)

Sets the _drawHandles_ flag for this shape and all descendants. If _drawH_ is TRUE (the default), any handles (control points) will be drawn. Otherwise, the handles will not be drawn.

3.14.101 wxShape::SetEventHandler

void GetEventHandler(**wxShapeEvtHandler ***_handler_)

Sets the event handler for this shape.

3.14.102 wxShape::SetFixedSize

void SetFixedSize(**bool** _x_, **bool** _y_)

Sets the shape to be of the given, fixed size.

3.14.103 wxShape::SetFont

void SetFont(**wxFont ***_font_, **int** _regionId = 0_)

Sets the font for the specified text region.

3.14.104 wxShape::SetFormatMode

void SetFormatMode(**int** *mode*, **int** *regionId = 0*)

Sets the format mode of the default text region. The argument can be a bit list of the following:

FORMAT_NONE No formatting.
FORMAT_CENTRE_HORIZ Horizontal centring.
FORMAT_CENTRE_VERT Vertical centring.

3.14.105 wxShape::SetHighlight

void SetHighlight(**bool** *hi*, **bool** *recurse = FALSE*)

Sets the highlight for a shape. Shape highlighting is unimplemented.

3.14.106 wxShape::SetId

void SetId(**long** *id*)

Set the integer identifier for this shape.

3.14.107 wxShape::SetMaintainAspectRatio

void SetMaintainAspectRatio(**bool** *flag*)

If the argument is TRUE, tells the shape that resizes should not change the aspect ratio (width and height should be in the original proportion).

3.14.108 wxShape::SetPen

void SetPen(**wxPen ***_pen_)

Sets the pen for drawing the shape's outline.

3.14.109 wxShape::SetRegionName

void SetRegionName(**const wxString&** *name*, **int** *regionId = 0*)

Sets the name for this region. The name for a region is unique within the scope of the whole composite, whereas a region id is unique only for a single image.

3.14.110 wxShape::SetSensitivityFilter

void SetSensitivityFilter(**int** *sens=OP_ALL*, **bool** *recursive = FALSE*)

Sets the shape to be sensitive or insensitive to specific mouse operations.

sens is a bitlist of the following:

- OP_CLICK_LEFT
- OP_CLICK_RIGHT
- OP_DRAG_LEFT
- OP_DRAG_RIGHT
- OP_ALL (equivalent to a combination of all the above).

3.14.111 wxShape::SetShadowMode

void SetShadowMode(**int** *mode*, **bool** *redraw = FALSE*)

Sets the shadow mode (whether a shadow is drawn or not). *mode* can be one of the following:

SHADOW_NONE No shadow (the default).
SHADOW_LEFT Shadow on the left side.
SHADOW_RIGHT Shadow on the right side.

3.14.112 wxShape::SetSize

void SetSize(**double** *x*, **double** *y*, **bool** *recursive = TRUE*)

Sets the shape's size.

3.14.113 wxShape::SetSpaceAttachments

void SetSpaceAttachments(**bool** *sp*)

Indicate whether lines should be spaced out evenly at the point they touch the node (TRUE), or whether they should join at a single point (FALSE).

3.14.114 wxShape::SetTextColour

void SetTextColour(**const wxString&** *colour*, **int** *regionId = 0*)

Sets the colour for the specified text region.

3.14.115 wxShape::SetX

void SetX(**double** *x*)

Sets the *x* position of the shape.

3.14.116 wxShape::SetX

void SetY(**double** *y*)

Sets the *y* position of the shape.

3.14.117 wxShape::SpaceAttachments

void SpaceAttachments(**bool** *sp*)

Sets the spacing mode: if TRUE, lines at the same attachment point will be spaced evenly across that side of the shape. If false, all lines at the same attachment point will emanate from the same point.

3.14.118 wxShape::Show

void Show(**bool** *show*)

Sets a flag indicating whether the shape should be drawn.

3.14.119 wxShape::Unlink

void Unlink()

If the shape is a line, unlinks the nodes attached to the shape, removing itself from the list of lines for each of the 'to' and 'from' nodes.

3.14.120 wxShape::WriteAttributes

void WriteAttributes(**wxExpr ***_clause_)

Writes the shape's attributes (data member values) into the given expression.

3.14.121 wxShape::WriteRegions

void WriteRegions(**wxExpr ***_clause_)

Writes the regions.

3.15 wxShapeCanvas

A canvas for drawing diagrams on.

Derived from

wxScrolledWindow

See also

wxDiagram (p. 7)

3.15.1 wxShapeCanvas::wxShapeCanvas

wxShapeCanvas(**wxWindow*** *parent = NULL*, **wxWindowID** *id = -1*, **const wxPoint&** *pos = wxDefaultPosition*, **const wxSize&** *size = wxDefaultSize*, **long** *style = wxBORDER*)

Constructor.

3.15.2 wxShapeCanvas::~wxShapeCanvas

~wxShapeCanvas()

Destructor.

3.15.3 wxShapeCanvas::AddShape

void AddShape(**wxShape ***shape*, **wxShape ***addAfter = NULL*)

Adds a shape to the diagram. If *addAfter* is non-NULL, the shape will be added after this one.

3.15.4 wxShapeCanvas::FindShape

wxShape * FindShape(**double** *x1*, **double** *y*, **int ***attachment*, **wxClassInfo ***info = NULL*, **wxShape ***notImage = NULL*)

Find a shape under this mouse click. Returns the shape (or NULL), and the nearest attachment point.

If *info* is non-NULL, a shape whose class which is a descendant of the desired class is found.

If *notImage* is non-NULL, shapes which are descendants of *notImage* are ignored.

3.15.5 wxShapeCanvas::FindFirstSensitiveShape

wxShape * FindFirstSensitiveShape(**double** *x1*, **double** *y*, **int ***attachment*, **int** *op*)

Finds the first sensitive shape whose sensitivity filter matches *op*, working up the hierarchy of composites until one (or none) is found.

3.15.6 wxShapeCanvas::GetDiagram

wxDiagram* GetDiagram() **const**

Returns the canvas associated with this diagram.

3.15.7 wxShapeCanvas::GetGridSpacing

double GetGridSpacing() **const**

Returns the grid spacing.

3.15.8 wxShapeCanvas::GetMouseTolerance

int GetMouseTolerance() **const**

Returns the tolerance within which a mouse move is ignored.

3.15.9 wxShapeCanvas::GetShapeList

wxList* GetShapeList() **const**

Returns a pointer to the internal shape list.

3.15.10 wxShapeCanvas::GetQuickEditMode

bool GetQuickEditMode() **const**

Returns quick edit mode for the associated diagram.

3.15.11 wxShapeCanvas::InsertShape

void InsertShape(**wxShape*** *shape*)

Inserts a shape at the front of the shape list.

3.15.12 wxShapeCanvas::OnBeginDragLeft

void OnBeginDragLeft(**double** *x*, **double** *y*, **int** *keys = 0*)

Called when the start of a left-button drag event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

See also *wxShapeCanvas::OnDragLeft* (p. 54), *wxShapeCanvas::OnEndDragLeft* (p. 54).

3.15.13 wxShapeCanvas::OnBeginDragRight

void OnBeginDragRight(**double** *x*, **double** *y*, **int** *keys = 0*)

Called when the start of a right-button drag event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

See also *wxShapeCanvas::OnDragRight* (p. 54), *wxShapeCanvas::OnEndDragRight* (p. 54).

3.15.14 wxShapeCanvas::OnEndDragLeft

void OnEndDragLeft(**double** *x*, **double** *y*, **int** *keys = 0*)

Called when the end of a left-button drag event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

See also *wxShapeCanvas::OnDragLeft* (p. 54), *wxShapeCanvas::OnBeginDragLeft* (p. 53).

3.15.15 wxShapeCanvas::OnEndDragRight

void OnEndDragRight(**double** *x*, **double** *y*, **int** *keys = 0*)

Called when the end of a right-button drag event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

See also *wxShapeCanvas::OnDragRight* (p. 54), *wxShapeCanvas::OnBeginDragRight* (p. 53).

3.15.16 wxShapeCanvas::OnDragLeft

void OnDragLeft(**bool** *draw*, **double** *x*, **double** *y*, **int** *keys = 0*)

Called when a left-button drag event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

draw is alternately TRUE and FALSE, to assist drawing and erasing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

See also *wxShapeCanvas::OnBeginDragLeft* (p. 53), *wxShapeCanvas::OnEndDragLeft* (p. 54).

3.15.17 wxShapeCanvas::OnDragRight

void OnDragRight(**bool** *draw*, **double** *x*, **double** *y*, **int** *keys = 0*)

Called when a right-button drag event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

draw is alternately TRUE and FALSE, to assist drawing and erasing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

See also *wxShapeCanvas::OnBeginDragRight* (p. 53), *wxShapeCanvas::OnEndDragRight* (p. 54).

3.15.18 wxShapeCanvas::OnLeftClick

void OnLeftClick(**double** *x*, **double** *y*, **int** *keys = 0*)

Called when a left click event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

3.15.19 wxShapeCanvas::OnRightClick

void OnRightClick(**double** *x*, **double** *y*, **int** *keys = 0*)

Called when a right click event on the canvas background is detected by OnEvent. You may override this member; by default it does nothing.

keys is a bit list of the following:

- KEY_SHIFT
- KEY_CTRL

3.15.20 wxShapeCanvas::Redraw

void Redraw()

Calls wxDiagram::Redraw.

3.15.21 wxShapeCanvas::RemoveShape

void RemoveShape(**wxShape ***_shape_)

Calls wxDiagram::RemoveShape.

3.15.22 wxShapeCanvas::SetDiagram

void SetDiagram(**wxDiagram ***_diagram_)

Sets the diagram associated with this diagram.

3.15.23 wxShapeCanvas::Snap

void Snap(**double ***_x_, **double ***_y_)

Calls wxDiagram::Snap.

3.16 wxShapeEvtHandler

wxShapeEvtHandler is a class from which wxShape (and therefore all shape classes) are derived. A wxShape also contains a pointer to its current wxShapeEvtHandler. Event handlers can be swapped in and out, altering the behaviour of a shape. This allows, for example, a range of behaviours to be redefined in one class, rather than requiring each shape class to be subclassed.

Derived from

wxObject

3.16.1 wxShapeEvtHandler::m_handlerShape

wxShape* m_handlerShape

Pointer to the shape associated with this handler.

3.16.2 wxShapeEvtHandler::m_previousHandler

wxShapeEvtHandler* m_previousHandler

Pointer to the previous handler.

3.16.3 wxShapeEvtHandler::wxShapeEvtHandler

void wxShapeEvtHandler(**wxShapeEvtHandler ***_previous = NULL_, **wxShape ***_shape = NULL_)

Constructs a new event handler.

3.16.4 wxShapeEvtHandler::~wxShapeEvtHandler

void ~wxShapeEvtHandler()

Destructor.

3.16.5 wxShapeEvtHandler::CopyData

void CopyData(**wxShapeEvtHandler&** *handler*)

A virtual function to copy the data from this object to *handler*. Override if you derive from wxShapeEvtHandler and have data to copy.

3.16.6 wxShapeEvtHandler::CreateNewCopy

wxShapeEvtHandler* CreateNewCopy()

Creates a new event handler object of the same class as this object, and then calls *wxShapeEvtHandler::CopyData* (p. 57).

3.16.7 wxShapeEvtHandler::GetPreviousHandler

wxShapeEvtHandler* GetPreviousHandler() **const**

Returns the previous handler.

3.16.8 wxShapeEvtHandler::GetShape

wxShape* GetShape() **const**

Returns the shape associated with this handler.

3.16.9 wxShapeEvtHandler::OnBeginDragLeft

void OnBeginDragLeft(**double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when the user is beginning to drag using the left mouse button.

3.16.10 wxShapeEvtHandler::OnBeginDragRight

void OnBeginDragRight(**double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when the user is beginning to drag using the right mouse button.

3.16.11 wxShapeEvtHandler::OnBeginSize

void OnBeginSize(**double** *width*, **double** *height*)

Called when a shape starts to be resized.

3.16.12 wxShapeEvtHandler::OnChangeAttachment

void OnChangeAttachment(**int** *attachment*, **wxLineShape*** *line*, **wxList&** *ordering*)

Override this to prevent or intercept line reordering. wxShape's implementation of this function calls *wxShape::ApplyAttachmentOrdering* (p. 35) to apply the new ordering.

3.16.13 wxShapeEvtHandler::OnDragLeft

void OnDragLeft(**bool** *draw*, **double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called twice when the shape is being dragged, once to allow erasing the old image, and again to allow drawing at the new position.

3.16.14 wxShapeEvtHandler::OnDragRight

void OnDragRight(**bool** *draw*, **double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called twice when the shape is being dragged, once to allow erasing the old image, and again to allow drawing at the new position.

3.16.15 wxShapeEvtHandler::OnDraw

void OnDraw(**wxDC&** *dc*)

Defined for each class to draw the main graphic, but not the contents.

3.16.16 wxShapeEvtHandler::OnDrawContents

void OnDrawContents(**wxDC&** *dc*)

Defined for each class to draw the contents of the shape, such as text.

3.16.17 wxShapeEvtHandler::OnDrawControlPoints

void OnDrawControlPoints(**wxDC&** *dc*)

Called when the shape's control points (handles) should be drawn.

3.16.18 wxShapeEvtHandler::OnDrawOutline

void OnDrawOutline(**wxDC&** *dc*)

Called when the outline of the shape should be drawn.

3.16.19 wxShapeEvtHandler::OnEndDragLeft

void OnEndDragLeft(**double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when the user is stopping dragging using the left mouse button.

3.16.20 wxShapeEvtHandler::OnEndDragRight

void OnEndDragRight(**double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when the user is stopping dragging using the right mouse button.

3.16.21 wxShapeEvtHandler::OnEndSize

void OnEndSize(**double** *width*, **double** *height*)

Called after a shape is resized.

3.16.22 wxShapeEvtHandler::OnErase

void OnErase(**wxDC&** *dc*)

Called when the whole shape should be erased.

3.16.23 wxShapeEvtHandler::OnEraseContents

void OnEraseContents(**wxDC&** *dc*)

Called when the contents should be erased.

3.16.24 wxShapeEvtHandler::OnEraseControlPoints

void OnEraseControlPoints(**wxDC&** *dc*)

Called when the shape's control points (handles) should be erased.

3.16.25 wxShapeEvtHandler::OnHighlight

void OnHighlight(**wxDC&** *dc*)

Called when the shape should be highlighted.

3.16.26 wxShapeEvtHandler::OnLeftClick

void OnLeftClick(**double** *x*, **double** *y*, **int** *keys =0*, **int** *attachment = 0*)

Called when the shape receives a left mouse click event.

3.16.27 wxShapeEvtHandler::OnMoveLink

void OnMoveLink(**wxDC&** *dc*, **bool** *moveControlPoints=TRUE*)

Called when the line attached to an shape need to be repositioned, because the shape has moved.

3.16.28 wxShapeEvtHandler::OnMoveLinks

void OnMoveLinks(**wxDC&** *dc*)

Called when the lines attached to an shape need to be repositioned, because the shape has moved.

3.16.29 wxShapeEvtHandler::OnMovePost

bool OnMovePost(**wxDC&** *dc*, **double** *x*, **double** *y*, **double** *oldX*, **double** *oldY*, **bool** *display = TRUE*)

Called just after the shape receives a move request.

3.16.30 wxShapeEvtHandler::OnMovePre

bool OnMovePre(**wxDC&** *dc*, **double** *x*, **double** *y*, **double** *oldX*, **double** *oldY*, **bool** *display = TRUE*)

Called just before the shape receives a move request. Returning TRUE allows the move to be processed; returning FALSE vetoes the move.

3.16.31 wxShapeEvtHandler::OnRightClick

void OnRightClick(**double** *x*, **double** *y*, **int** *keys = 0*, **int** *attachment = 0*)

Called when the shape receives a mouse mouse click event.

3.16.32 wxShapeEvtHandler::OnSize

void OnSize(**double** *x*, **double** *y*)

Called when the shape receives a resize request.

3.16.33 wxShapeEvtHandler::OnSizingBeginDragLeft

void OnSizingBeginDragLeft(**wxControlPoint*** *pt*, **double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when a sizing drag is beginning.

3.16.34 wxShapeEvtHandler::OnSizingDragLeft

void OnSizingDragLeft(**wxControlPoint*** *pt*, **bool** *draw*, **double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when a sizing drag is occurring.

3.16.35 wxShapeEvtHandler::OnSizingEndDragLeft

void OnSizingEndDragLeft(**wxControlPoint*** *pt*, **double** *x*, **double** *y*, **int** *keys=0*, **int** *attachment = 0*)

Called when a sizing drag is ending.

3.16.36 wxShapeEvtHandler::SetPreviousHandler

void SetPreviousHandler(**wxShapeEvtHandler*** *handler*)

Sets the previous handler.

3.16.37 wxShapeEvtHandler::SetShape

void SetShape(**wxShape*** *shape*)

Sets the shape for this handler.

3.17 wxTextShape

As wxRectangleShape, but only the text is displayed.

Derived from

wxRectangleShape (p. 33)

3.17.1 wxTextShape::wxTextShape

void wxTextShape(**double** *width = 0.0*, **double** *height = 0.0*)

Constructor.

3.17.2 wxTextShape::~wxTextShape

void ~wxTextShape()

Destructor.

3.18 Functions

These are the OGL functions.

3.18.1 ::wxOGLInitialize

void wxOGLInitialize() Initializes OGL.

3.18.2 ::wxOGLCleanUp

void wxOGLCleanUp() Cleans up OGL.

4 Topic overviews

The following sections describe particular topics.

4.1 OGL overview

wxShapeCanvas (p. 51), derived from **wxCanvas**, is the drawing area for a number of *wxShape* (p. 33) instances. Everything drawn on a wxShapeCanvas is derived from wxShape, which provides virtual member functions for redrawing, creating and destroying resize/selection 'handles', movement and erasing behaviour, mouse click behaviour, calculating the bounding box of the shape, linking nodes with arcs, and so on.

The way a client application copes with 'damage' to the canvas is to erase (white out) anything should no longer be displayed, redraw the shape, and then redraw everything on the canvas to repair any damage. If quick edit mode is on for the canvas, the complete should be omitted by OGL and the application.

Selection handles (called control points in the code) are implemented as wxRectangleShapes.

Events are passed to shapes by the canvas in a high-level form, for example **OnLeftClick**,**OnBeginDragLeft**, **OnDragLeft**, **OnEndDragLeft**. The canvas decides what is a click and what is a drag, whether it is on a shape or the canvas itself, and (by interrogating the shape) which attachment point the click is associated with.

In order to provide event-handling flexibility, each shapes has an 'event handler' associated with it, which by default is the shape itself (all shapes derive from wxShapeEvtHandler). An application can modify the event-handling behaviour simply by plugging a new event handler into the shape. This can avoid the need for multiple inheritance when new properties and behaviour are required for a number of different shape classes: instead of overriding each class, one new event handler class can be defined and used for all existing shape classes.

A range of shapes have been predefined in the library, including rectangles, ellipses, polygons. A client application can derive from these shapes and/or derive entirely new shapes from wxShape.

Instances of a class called *wxDiagram* (p. 7) organise collections of shapes, providing default file input and output behaviour.

4.2 wxDividedShape overview

Classes: *wxDividedShape* (p. 20)

A wxDividedShape is a rectangle with a number of vertical divisions. Each division may have its text formatted with independent characteristics, and the size of each division relative to the whole image may be specified.

Once a wxDividedShape has been created, the user may move the divisions with the mouse. By pressing Ctrl while right-clicking, the region attributes can be edited.

Here are examples of creating wxDividedShape objects:

```
/*
 * Divided rectangle with 3 regions
 *
```

```
 */

wxDividedShape *dividedRect = new wxDividedShape(50, 60);

wxShapeRegion *region = new wxShapeRegion;
region->SetProportions(0.0, 0.25);
dividedRect->AddRegion(region);

region = new wxShapeRegion;
region->SetProportions(0.0, 0.5);
dividedRect->AddRegion(region);

region = new wxShapeRegion;
region->SetProportions(0.0, 0.25);
dividedRect->AddRegion(region);

dividedRect->SetSize(50, 60); // Allow it to calculate region sizes
dividedRect->SetPen(wxBLACK_PEN);
dividedRect->SetBrush(wxWHITE_BRUSH);
dividedRect->Show(TRUE);
dividedRect->NameRegions();

/*
 * Divided rectangle with 3 regions, rounded
 *
 */

wxDividedShape *dividedRect3 = new wxDividedShape(50, 60);
dividedRect3->SetCornerRadius(-0.4);

region = new wxShapeRegion;
region->SetProportions(0.0, 0.25);
dividedRect3->AddRegion(region);

region = new wxShapeRegion;
region->SetProportions(0.0, 0.5);
dividedRect3->AddRegion(region);

region = new wxShapeRegion;
region->SetProportions(0.0, 0.25);
dividedRect3->AddRegion(region);

dividedRect3->SetSize(50, 60); // Allow it to calculate region sizes
dividedRect3->SetPen(wxBLACK_PEN);
dividedRect3->SetBrush(wxWHITE_BRUSH);
dividedRect3->Show(TRUE);
dividedRect3->NameRegions();
```

4.3 wxCompositeShape overview

Classes: *wxCompositeShape* (p. 17), *wxOGLConstraint* (p. 4)

The wxCompositeShape allows fairly complex shapes to be created, and maintains a set of constraints which specify the layout and proportions of child shapes.

Add child shapes to a wxCompositeShape using *AddChild* (p. 18), and add constraints using *AddConstraint* (p. 18).

After children and shapes have been added, call *Recompute* (p. 19) which will return TRUE is the constraints could be satisfied, FALSE otherwise. If constraints have been correctly and consistently specified, this call will succeed.

If there is more than one child, constraints must be specified: OGL cannot calculate the size and position of children otherwise. Don't assume that children will simply move relative to the parent without the use of constraints.

To specify a constraint, you need three things:

1. a constraint type, such as gyCONSTRAINT_CENTRED_VERTICALLY;
2. a reference shape, with respect to which other shapes are going to be positioned - the *constraining* shape;
3. a list of one or more shapes to be constrained: the *constrained* shapes.

The constraining shape can be either the parent of the constrained shapes, or a sibling. The constrained shapes must all be siblings of each other.

For an exhaustive list and description of the available constraint types, see the *wxOGLConstraint constructor* (p. 4). Note that most constraints operate in one dimension only (vertically or horizontally), so you will usually need to specify constraints in pairs.

You can set the spacing between constraining and constrained shapes by calling *wxOGLConstraint::SetSpacing* (p. 6).

Finally, a wxCompositeShape can have *divisions*, which are special child shapes of class wxDivisionShape (not to be confused with wxDividedShape). The purpose of this is to allow the composite to be divided into user-adjustable regions (divisions) into which other shapes can be dropped dynamically, given suitable application code. Divisons allow the child shapes to have an identity of their own - they can be manipulated independently of their container - but to behave as if they are contained with the division, moving with the parent shape. Divisions boundaries can themselves be moved using the mouse.

To create an initial division, call *wxCompositeShape::MakeContainer* (p. 19). Make further divisions by calling *wxDivisionShape::Divide* (p. 22).

5 Bugs

These are the known bugs.

- In the OGLEdit sample, .dia files are output double-spaced due to an unidentified bug in the way a stream is converted to a file.

6 Change log

Version 3.0, September 8th 1998

- Version for wxWindows 2.0.
- Various enhancements especially to wxDrawnShape (multiple metafiles, for different orientations).
- More ability to override functions e.g. OnSizeDragLeft, so events can be intercepted for Do/Undo.

Version 2.0, June 1st 1996

- First publicly released version.

7 Index

—G—

—H—

—I—

—L—

—M—

—N—

—O—

—P—

—R—

—S—

—T—

—U—

—W—