12

DRAWING WITH QUICKDRAW

Demonstration Program: QuickDraw

Introduction

As stated at Chapter 11, QuickDraw is a collection of system software routines that your application uses to perform imaging operations, that is, the construction and display of graphical information for display on output devices such as screens and printers.

The Coordinate Plane, Points, Rectangles, and Regions

The following mathematical constructs are widely used in QuickDraw's functions and data types:

- The coordinate plane.
- The point.
- The rectangle.
- The region.

The Coordinate Plane

A Macintosh screen (or screens) represents part of a global coordinate plane bounded by the limits of QuickDraw coordinates (-32768 to 32767). The (0,0) origin point of this global coordinate plane is at the upper-left corner of the main screen. From the upper-left coordinate of the main screen, coordinate values decrease to the left and up and increase to the right and down. Any pixel on the screen can be specified by a vertical coordinate and a horizontal coordinate.

In addition to the global coordinate system, QuickDraw maintains a **local coordinate system** for every window's graphics port. The relationship between local and global coordinates is shown at Fig 1.

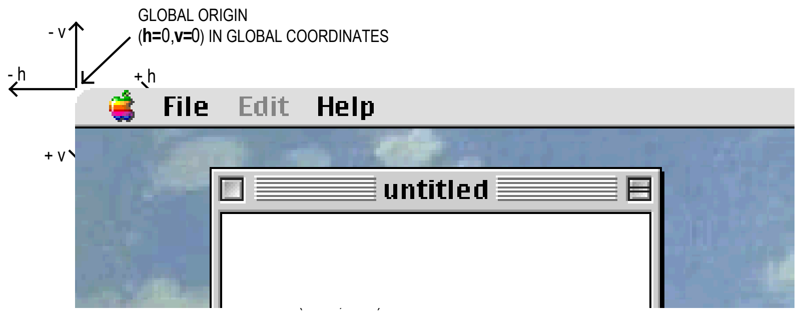


FIG 1 - LOCAL AND GLOBAL COORDINATE SYSTEMS

Points

The location on the coordinate plane where imaginary horizontal and vertical grid lines intersect is called a **point**. Points themselves are dimensionless whereas a pixel is not. As shown at Fig 2, a pixel "hangs" down and to the right of the point by which it is addressed. A pixel thus lies between the infinitely thin lines of the coordinate grid.

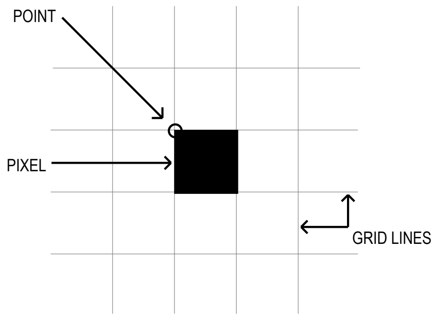


FIG 2 - A POINT AND A PIXEL

The data type for points is `Point`:

```
struct Point
{
  short v;  // Vertical coordinate.
  short h;  // Horizontal coordinate.
};
typedef struct Point Point;
typedef Point *PointPtr;
```

Rectangles

A **rectangle**, whose borders are infinitely thin in the same way that a point is infinitely small, is used to define an area on the screen.

The data type for rectangles is `Rect`:

```
struct Rect
{
  short top;
  short left;
  short bottom;
  short right;
};
typedef struct Rect Rect;
typedef Rect *RectPtr;
```

If the bottom coordinate of a rectangle is equal to or less than the top, or the right coordinate is less than the left, the rectangle is said to be an **empty rectangle**.

Regions

A **region** is an arbitrary area, or set of areas, the outline of which is one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate ones, and can even have holes in the middle. In the examples at Fig 3, the region on the left has a hole and the one on the right consists of two unconnected areas.

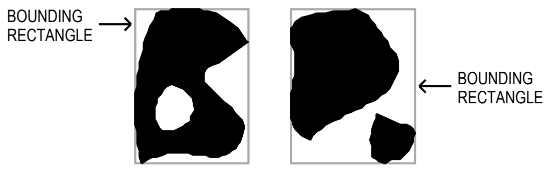


FIG 3 - TWO REGIONS

Region Objects and Accessor Functions

QuickDraw stores information about regions in opaque data structures called **region objects**. The data type `RgnHandle` is defined as a reference to a region object:

```
typedef struct OpaqueRgnHandle* RgnHandle;
```

One accessor function is provided to access the information in region objects:

Function	***Description***
`GetRegionBounds`	Get the region's bounding rectangle.

For a region which is a rectangle, the rectangle returned by `GetRegionBounds` defines the entire region. The data for more complex regions is stored in the region object in a proprietary format. The function `IsRegionRectangular` may be used to determine whether a specific region is rectangular.

The Graphics Pen, Foreground and Background Colours, Pixel Patterns and Bit Patterns, and Transfer Modes

The Graphics Pen

The metaphorical graphics pen used for drawing lines and shapes in a graphics port is rectangular in shape and its size (that is, its height and width) is measured in pixels. Whenever you draw into a graphics port, the characteristics of the graphics pen determine how the drawing looks. Those characteristics are as follows:

- Pen **location**, which is specified in local coordinates stored in the graphics port. The functions `Move` and `MoveTo` are used to move the pen to a specified location, and the function `GetPen` gets the pen's current location.
- Pen **size**, which is specified by the a width and a height (in pixels) stored in the graphics port. The pen's default size is one-by-one pixel; however, `PenSize` can be used to change the size and shape up to a 32,767-by-32767 pixel square. Note that, if either the width or height is set to 0, the pen does not draw.
- Pen **colour**, that is, the graphics port's foreground colour.
- Pen **pattern**, which defines the pattern that the pen draws with.
- Pen **transfer mode**, a Boolean or arithmetic operation which determines how QuickDraw transfers the pen pattern to the pixel map during drawing operations.
- Pen **visibility**, which is specified by an integer stored in the graphics port, indicating whether drawing operations will actually appear. For example, for 0 or negative values, the pen draws with "invisible ink". The functions `ShowPen` and `HidePen` are used to change pen visibility.

Getting and Setting the Pen State

The following functions are used to get and set the current pen state:

Function	*Description*
`GetPenState`	Returns, in a `PenState` structure, the graphics pen's current location, size, transfer mode, and pattern.
`SetPenState`	Using information supplied by a `PenState` structure, sets the graphics pen's location, size, transfer mode, and pattern.
`PenNormal`	Resets the pen size, transfer mode, and pattern to the state initialised when the graphics port was opened.

Foreground and Background Colour

Foreground Colour

The function `RGBForecolor` is used to set the foreground colour in the graphics port. You may also use the Palette Manager function `PmForeColor` to set the foreground colour.

The foreground colour is used by the graphics pen for **drawing** lines, framed shapes, and text. The foreground colour is also used by QuickDraw shape **painting** functions.

Background Colour

The function `RGBBackColor` is used to set the background colour in the graphics port. You may also use the Palette Manager function `PmBackColor` to set the background colour.

The background colour is used by QuickDraw **erasing** functions, and is also used by the `ScrollRect` function to replace the scrolled pixels.

Pixel Patterns and Bit Patterns

Pixel Patterns

If you wish to draw or paint with a colour *pattern* rather than the colour set by `RGBForecolor`, you can set the pen pixel pattern in the graphics port using `SetPortPenPixPat` or `PenPixPat`. (Initially, the pen pixel pattern in the graphics port is all-"black". When you set a non-all-"black" pattern, the pen pattern in the graphics port overrides the foreground colour.)

You define a pixel pattern in a `'ppat'` resource. To retrieve the pixel pattern stored in the `'ppat'` resource, you use the `GetPixPat` function. The handle to a `pixPat` data structure returned by `GetPixPat` may then be used in a call to `SetPortPenPixPat` or `PenPixPat` to set the pixel pattern.

Similarly, if you wish to erase with a pixel pattern rather than the background colour, or replace the pixels scrolled by `ScrollRect` with a pixel pattern rather than the background colour, you can set the background pixel pattern in the graphics port using `SetPortBackPixPat` or `BackPixPat`. (Initially, the background pixel pattern in the graphics port is all-"white". When you assign a non-all-"white" pattern, the background pattern in the graphics port overrides the background colour)

In addition to drawing, painting and erasing functions, QuickDraw includes shape **filling** functions, which may be used to fill a specified shape using a specified pixel pattern. A handle to a `pixPat` data structure is passed in the `thePPat` parameter of these functions.

Bit Patterns

After drawing or painting with a pixel pattern, you can return to drawing or painting with the foreground colour by simply restoring the default all-"black" pattern by calling `PenPat` and passing in the bit pattern contained in the QuickDraw global variable `black` as follows:

```
Pattern blackPattern;

PenPat(GetQDGlobalsBlack(&blackPattern));
```

After erasing with a pixel pattern, you can return to erasing with the background colour by simply restoring the default all-"white" pattern by calling `BackPat` and passing in the bit pattern contained in the QuickDraw global variable `white` as follows:

```
Pattern whitePattern;

BackPat(GetQDGlobalsWhite(&whitePattern));
```

When you use the `PenPat` and `BackPat` functions, QuickDraw constructs a pixel pattern equivalent to the bit pattern, The graphics port's current foreground colour is used for the "black" bits in the bit pattern, and the background colour is used for the "white" bits.

The `PenPat` and `BackPat` functions may also be used to set other bit patterns in the graphics port.

Transfer Modes

The term **transfer mode** may be considered as a generic term encompassing three different transfer mode types. Each has to do with the way source pixels interact with destination pixels during drawing, painting, erasing, filling, and **copying** operations. The three types of transfer mode are as follows:

- ***Boolean Pattern Mode***. Boolean pattern modes apply to line drawing, framing, painting, erasing, and filling operations.
- ***Boolean Source Mode.*** Boolean source modes apply to text drawing and copying operations.
- ***Arithmetic Source Mode.*** Arithmetic source modes apply to drawing (including text drawing), painting, and copying operations.

Boolean Pattern Modes

Pattern modes may be set as pen transfer modes in the graphics port using the `PenMode` function. The modes are represented by eight constants, each of which relates to a specific Boolean operation (COPY, OR, XOR, and BIC (for bit clear)) and their inverse variants.

The effects of these modes are best explained assuming a 1-bit (black-and-white) environment in which the foreground colour is black and the background colour is white. The following lists the pattern modes and describes the effect of source pixels on destination pixels in such an environment.

Pattern Mode	***Action On Destination Pixel***	
	If source pixel is black	***If source pixel is white***
`patCopy`	Apply foreground colour.	Apply background colour.
`patOr`	Apply foreground colour.	Leave alone.
`patXor`	Invert.	Leave alone.
`patBic`	Apply background colour.	Leave alone.
`notPatCopy`	Apply background colour.	Apply foreground colour.
`notPatOr`	Leave alone.	Force black.
`notPatXor`	Leave alone.	Invert.
`notPatBic`	Leave alone.	Apply background colour.

These effects are illustrated at Fig 4. Note particularly that `patCopy` causes the destination pixels to be completely over-written. `patCopy` is the transfer mode initially set in the graphics port.

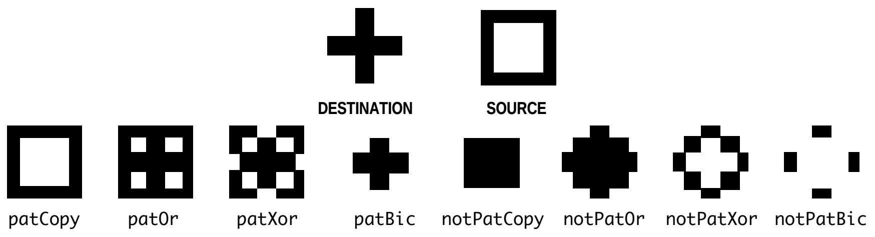


FIG 4 - EFFECT OF PATTERN MODES

Boolean Source Modes

Boolean source modes may be set as text in the graphics port using the function `TextMode`, and may be passed as parameters in QuickDraw functions for copying pixel images. The Boolean source modes are the equivalent in text drawing and copying to the Boolean pattern mode used for non-text drawing, painting, filling, and erasing operations.

The relevant constants are `srcCopy`, `srcOr`, `srcXor`, `srcBic`, `notSrcCopy`, `notSrcOr`, `notSrcXor`, and `notSrcBic`. The additional non-standard mode `grayishTextOr` is useful for drawing text in deactivated or disabled user interface objects. (This mode is considered non-standard because it is not stored in pictures and printing with it is undefined.)

Arithmetic Source Modes

Arithmetic source modes may be set in the graphics port, and may be passed as parameters in QuickDraw functions for copying pixel images.

Arithmetic source modes perform arithmetic operations on the values of the red, green and blue components of the source and destination pixels. Because they work with RGB colours rather than colour table indexes, arithmetic transfer modes produce predictable results on indexed devices. The arithmetic source modes and their effects in a colour environment are as follows:

Constant	***Value***	***Description***
`blend`	32	Destination pixel is replaced with a blend of the source and destination pixel colours. Revert to `srcCopy` mode if the destination is a bitmap or 1-bit pixel image.
`addPin`	33	Destination pixel is replaced with the sum of the source and destination pixel colours up to a maximum allowable value. Revert to `srcBic` mode if the destination is a bitmap or 1-bit pixel image.
`addOver`	34	Destination pixel is replaced with the sum of the source and destination pixel colours, but if the value of the red, green or blue component exceeds 65,536, then subtract 65,536 from that value. Revert to `srcXor` mode if the destination is a bitmap or 1-bit pixel image.
`subPin`	35	Destination pixel is replaced with the difference of the source and destination pixel colours, but not less than a minimum allowable value. Revert to `srcOr` mode if the destination is a bitmap or 1-bit pixel image.
`transparent`	36	Source and destination pixel are replaced with the source pixel if the source pixel is not equal to the background colour.
`addMax`	37	Destination pixel is replaced with the colour containing the greater saturation of each of the RGB components of the source and destination pixels. Revert to `srcBic` mode if the destination is a bitmap or 1-bit pixel image.
`subOver`	38	Destination pixel is replaced with the difference of the source and destination pixel colours, but if the value of the red, green or blue is less than 0, add the negative result to 65,536. Revert to `srcXor` mode if the destination is a bitmap or 1-bit pixel image.
`adMin`	39	Destination pixel is replaced with the colour containing the lesser saturation of each of the RGB components of the source and destination pixels. Revert to `srcOr` mode if the destination is a bitmap or 1-bit pixel image.

Drawing Lines and Framed Shapes

Functions for Drawing Lines

You can move the graphics pen to a specified location, and you can draw lines from that location. Lines are drawn using the current graphics pen size, foreground colour or pen pixel/bit pattern, and pen pattern mode.

Functions for moving the graphics pen and drawing lines are as follows:

Function	***Description***
`MoveTo`	Moves the graphics pen location to the specified location, in local coordinates.
`Move`	Moves the graphics pen a specified distance from its current location.
`LineTo`	Draws a line from the current pen location to the specified location, in local coordinates.
`Line`	Draws a line a specified distance from the graphics pen's current location.

Fig 5 shows a line drawn with a pen of size 20-by-40 pixels. Note that the pen "hangs" below and to the right of the defining points,

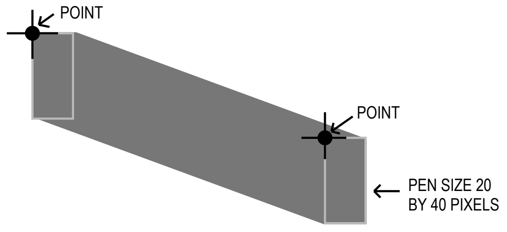


FIG 5 - A LINE DRAWN BY LineTo OR Line

Functions for Drawing Framed Shapes

Framing a shape draws its outline only, using the current pen size, foreground colour or pen pixel/bit pattern, and pen pattern mode. The pixels in the interior of the shape are unaffected. Framed shapes are drawn using the current graphics pen size, foreground colour or pen pixel/bit pattern, and pen pattern mode.

Functions for drawing framed shapes are as follows:

Function	***Description***
`FrameRect`	Draws a rectangle, the position and size of which are defined by a `Rect` structure.
`FrameOval`	Draws an oval, the position and size of which are determined by a bounding rectangle defined by a `Rect` structure.
`FrameRoundRect`	Draws a rounded rectangle, the position and size of which are determined by a bounding rectangle defined by a `Rect` structure. Curvature of the corners is defined by `ovalWidth` and `ovalHeight` parameters.
`FrameArc`	Draws an arc, the position and size of which are determined by a bounding rectangle defined by a `Rect` structure. Starting point and arc extent are determined by `startAngle` and `arcAngle` parameters.
`FramePoly`	Draws a polygon by "playing back" all the line drawing calls that define it.
`FrameRgn`	Draws an outline around a specified region. The line is drawn just inside the region.

Fig 6 shows various framed shapes drawn with various graphics pen sizes and bit patterns. Note that the bounding rectangles completely enclose the shapes they bound, that is, no pixels extend outside the infinitely thin lines of the bounding rectangle. Note also that the arc is a portion of the circumference of an oval bounded by a pair or radii joining at the oval's centre.

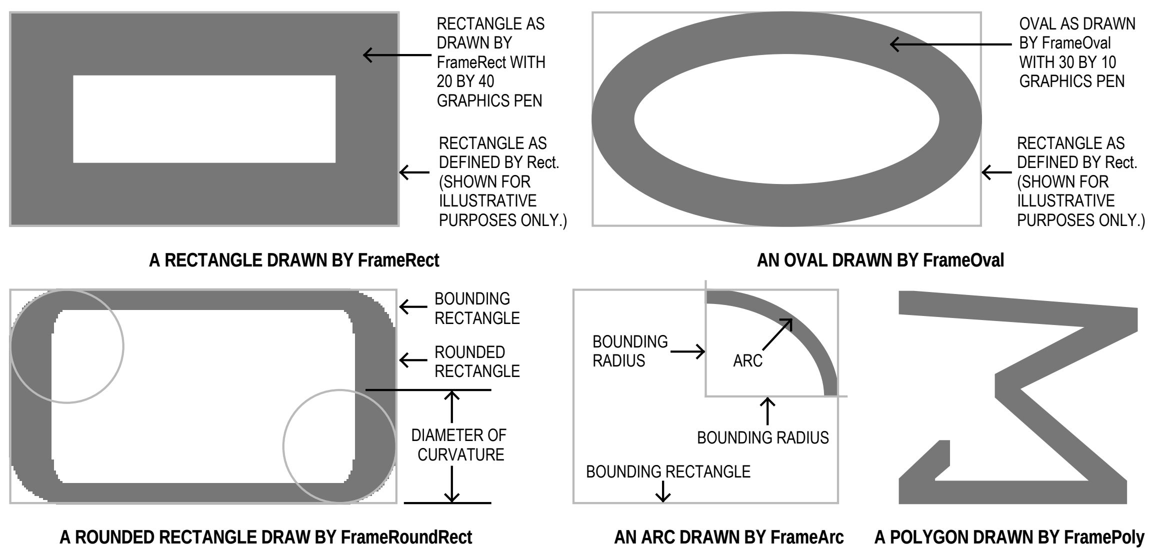

RECTANGLE AS DRAWN BY FrameRect WITH 20 BY 40 GRAPHICS PEN
RECTANGLE AS DEFINED BY Rect. (SHOWN FOR ILLUSTRATIVE PURPOSES ONLY.)
OVAL AS DRAWN BY FrameOval WITH 30 BY 10 GRAPHICS PEN
RECTANGLE AS DEFINED BY Rect. (SHOWN FOR ILLUSTRATIVE PURPOSES ONLY.)
A RECTANGLE DRAWN BY FrameRect
AN OVAL DRAWN BY FrameOval
BOUNDING RECTANGLE
ROUNDED RECTANGLE
DIAMETER OF CURVATURE
BOUNDING RADIUS
ARC
BOUNDING RADIUS
BOUNDING RECTANGLE
A ROUNDED RECTANGLE DRAW BY FrameRoundRect
AN ARC DRAWN BY FrameArc
A POLYGON DRAWN BY FramePoly

FIG 6 - FRAMED SHAPES DRAWN WITH QUICKDRAW FRAMED SHAPE DRAWING FUNCTIONS

Framed Polygons and Regions

Framed polygons and regions require that you call several functions to create and draw them. You begin by calling a function that collects drawing commands into a definition for the object. You then use drawing functions to define the object before calling a function which signals the end of the object definition. Finally, you use a function which draws the newly-defined object.

Framed Polygons

To define a polygon you must first call `OpenPoly`. You then call `LineTo` a number of times to create lines from the first vertex to the second, from the second vertex to the third, etc. You then call `ClosePoly`, which completes the definition process. After defining a polygon in this way, you can draw the polygon, as a framed polygon, using `FramePoly`.

Note that, in the framed polygon at Fig 5, the final defining line from the last vertex back to the first vertex was not drawn during the definition process. Note also that, as in all line drawing, `FramePoly` hangs the pen down and to the right of the infinitely thin lines that define the polygon.

Framed Regions

To define a region, you can use any set of lines or shapes, including other regions, so long as the region's outline consists of one or more closed loops. First, however, you must call `NewRgn` and `OpenRgn`. You then use line, shape, or region drawing commands to define the region. When you have finished collecting commands to define the outline of the region, you call `CloseRgn`. You can then draw the framed region using `FrameRegion`.

Drawing Painted and Filled Shapes

Painting a shape fills both its outline and its interior with the current foreground colour or graphics pen pixel/bit pattern. Filling a shape fills both its outline and its interior with a pixel pattern or bit pattern passed in a parameter of the QuickDraw shape filling functions.

Transfer Mode. Painting operations utilise the current graphics pen pattern mode. In filling operations, the transfer mode is invariably the pattern mode `patCopy`, meaning that the destination pixels are always completely overwritten.

Functions for Painting and Filling Shapes

The following lists the available functions for painting and filling shapes:

Function	Description
`PaintRect`	Fills a rectangle with the current foreground colour or graphics pen pixel/ bit pattern.
`PaintOval`	Fills an oval with the current foreground colour or graphics pen pixel/ bit pattern.
`PaintRoundRect`	Fills a round rectangle with the current foreground colour or graphics pen pixel/ bit pattern.
`PaintArc`	Fills a wedge with the current foreground colour or graphics pen pixel/ bit pattern.
`PaintPoly`	Fills a polygon with the current foreground colour or graphics pen pixel/ bit pattern.
`PaintRgn`	Fills a region with the current foreground colour or graphics pen pixel/ bit pattern.
`FillRect`	Fills a rectangle with a specified bit pattern.
`FillCRect`	Fills a rectangle with a specified pixel pattern.
`FillOval`	Fills an oval with a specified bit pattern.
`FillCOval`	Fills an oval with a specified pixel pattern.
`FillRoundRect`	Fills a round rectangle with a specified bit pattern.
`FillCRoundRect`	Fills a round rectangle with a specified pixel pattern.
`FillArc`	Fills a wedge of an oval with a specified bit pattern.
`FillCArc`	Fills a wedge of an oval with a specified pixel pattern.
`FillPoly`	Fills a polygon with a specified bit pattern.
`FillCPoly`	Fills a polygon with a specified pixel pattern.
`FillRgn`	Fills a region with a specified bit pattern.
`FillCRgn`	Fills a region with a specified pixel pattern.

Wedges

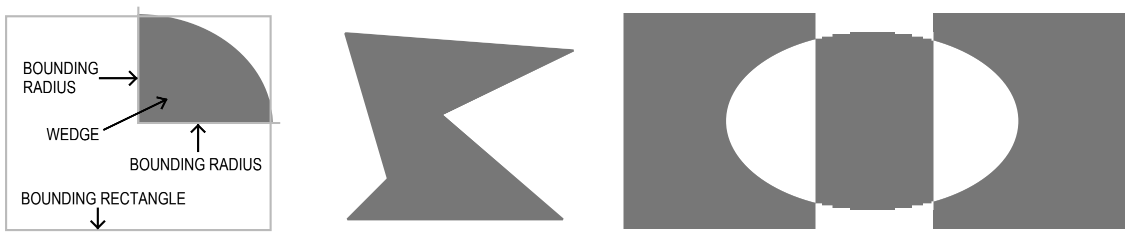
The **wedges** drawn by `PaintArc`, `FillArc`, and `FillCArc` are pie-shaped segments of an oval bounded by a pair of radii joining at the oval's centre. A wedge includes part of the oval's interior. Like the framed arc, wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii. Fig 7 shows a wedge.

Painted and Filled Polygons and Regions

The general procedure for drawing painted and filled polygons and regions is the same as described for their framed counterparts, above.

Fig 7 shows the polygon as defined for the framed polygon at Fig 6, but this time drawn with one of the polygon painting or filling functions. Note that, although the final defining line from the last vertex back to the first vertex was not drawn, the painting and filling functions complete the polygon (whereas `FramePoly` did not).

Fig 7 also shows a region comprising two rectangles and an overlapping oval, drawn using `PaintRgn`. Note that, where two regions overlap, the additional area is added to the region and the overlap is removed from the region.



A WEDGE　　A PAINTED OR FILLED POLYGON　　A PAINTED OR FILLED REGION

FIG 7 - A WEDGE, A PAINTED OR FILLED POLYGON, AND A PAINTED OR FILLED REGION

Erasing and Inverting Shapes

Erasing a shape fills both its outline and its interior with the background colour or background pixel/bit pattern. Inverting a shape simply inverts all the pixels in the shape; for example, all black pixels become white, and vice versa.

Transfer Mode. In erasing operations, the transfer mode is invariably the pattern mode `patCopy`, meaning that the destination pixels are always completely overwritten.

Functions for Erasing and Inverting Shapes

The following list the available functions for painting and filling shapes:

Function	Description
`EraseRect`	Fills a rectangle with the current background colour or pixel/ bit pattern.
`EraseOval`	Fills an oval with the current background colour or pixel/ bit pattern.
`EraseRoundRect`	Fills a round rectangle with the current background colour or pixel/ bit pattern.
`EraseArc`	Fills a wedge with the current background colour or pixel/ bit pattern.
`ErasePoly`	Fills a polygon with the current background colour or pixel/ bit pattern.
`EraseRgn`	Fills a region with the current background colour or pixel/ bit pattern.
`InvertRect`	Inverts all the pixels in a rectangle.
`InvertOval`	Inverts all the pixels in an oval.
`InvertRoundRect`	Inverts all the pixels in a round rectangle.
`InvertArc`	Inverts all the pixels in a wedge.
`InvertPoly`	Inverts all the pixels in a polygon.
`InvertRgn`	Inverts all the pixels in a region.

Drawing Pictures

Your application can record a sequence of QuickDraw drawing operations in a **picture** and play its image back later. Fig 8 shows an example of a simple picture containing a filled rectangle, a filled oval, and some text.

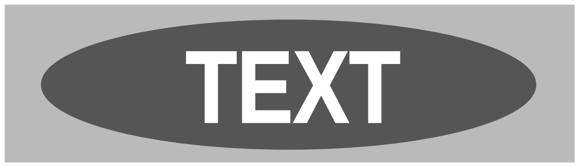


FIG 8 - A SIMPLE QUICKDRAW PICTURE

The subject of pictures is addressed in more detail at Chapter 13.

Drawing Text

Setting the Font

The font used to draw text in a graphics port may be set using the function `TextFont`. `TextFont` takes a single parameter, of type `SInt16`, which may be either a predefined constant or a **font family ID** number. Although predefined constants remain in the header file Fonts.h, their use is now discouraged by Apple.

You can get the font family ID using `GetFNum`.[1] For example, the following sets the current font to Palatino:

[1] If you know the font family ID, you can get its name by calling the Font Manager's `GetFontName` function. If you do not know either the font family ID or the font name, you can use the Resource Manager's `GetIndResource` function followed by the `GetResInfo` function to determine the names and IDs of all available fonts.

```
SInt16 fontNum;

GetFNum("\pPalatino",&fontNum);
TextFont(fontNum);
```

Setting and Modifying the Text Style

You use the function `TextFace` to change the text style, using any combination of the constants `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, and `extend`. Some examples are as follows:

```
TextFace(bold);                              // Set to bold.
TextFace(bold | italic);                     // Set to bold and italic.)
TextFace(GetPortTextFace(thePort) | bold);   // Add bold to existing.
TextFace(GetPortTextFace(thePort) &~ bold);  // Remove bold.
TextFace(normal);                            // Set to plain.
```

Setting the Font Size

You use the function `TextSize` to change the font size in typographical **points**. A point is approximately 1/72 inch.

Changing the Width of Characters

Widening and narrowing space and non-space characters lets you meet special formatting requirements. You use `SpaceExtra` to specify the extra pixels to be added to or subtracted from the standard width of the space character. `SpaceExtra` is ordinarily used in text-justification functions.

Transfer Mode

The transfer mode initially set in the graphics port is the Boolean source mode `srcOr`. This mode causes the colour of the glyph[2] to be determined by the foreground colour and the drawn glyph to completely overwrite the existing pixels. (In this mode only those bits which make up the actual glyph are drawn.)

You should generally use either `srcOr` or `srcBic` when drawing text, because all other transfer modes draw the character's background as well as the glyph itself. This can result in the clipping of characters by adjacent characters.

Copying Pixel Images Between Graphics Ports

QuickDraw provides the following three functions for copying pixel images between graphics ports:

- `CopyBits`, which copies a pixel image to another graphics port, optionally allowing you to:
    - Resize the image.
    - Modify the image with transfer modes.
    - Clip the image to a region.
- `CopyMask`, which copies a pixel image to another graphics port, allowing you to:
    - Resize the image.
    - Modify the image by passing it through a mask.
- `CopyDeepMask`, which combines the effects of `CopyBits` and `CopyMask`, optionally allowing you to:
    - Resize the image.
    - Clip the image to a region.
    - Specify a transfer mode.

[2] A glyph is the visual representation of a character.

- Modify the image by passing it through a mask.

The mask used by `CopyMask` and `CopyDeepMask` may be another pixel map whose pixels indicate proportionate weights of the colours for the source and destination pixels.

The `CopyBits`, `CopyMask`, and `CopyDeepMask` functions date from the era of black-and-white Macintoshes, which is why they expect a pointer to a bitmap in their source and destination parameters. Thus, when you are copying pixel maps using these functions, you must cast the address of the handle to the pixel map to a pointer to a bitmap. By looking at certain information in the graphics port object, `CopyBits`, `CopyMask`, and `CopyDeepMask` can establish that you have passed the functions a handle to a pixel map rather than the base address of a bitmap.

Using Masks

With `CopyMask` and `CopyDeepMask`, you supply a pixel map to act as the mask. The mask's pixels proportionally select between source and destination pixel values.

In the case of masks that are 1 bit deep, black bits in the mask cause the destination pixel to take the colour of the source pixel and white bits cause the destination pixel to retain its current colour. In the case of masks with pixel depths greater than 1, Colour QuickDraw takes a weighted average between source and destination colours. For example, a blue mask (that is, one with high values for the blue components of all pixels) filters out blue values coming from the source.

Transfer Modes

`CopyBits` and `CopyDeepMask` both allow you to specify the transfer mode, which can be either a Boolean source mode or an arithmetic source mode.

The Importance of Foreground and Background Colour

Applying a foreground colour other than black or a background colour other than white to the pixel can produce an unexpected result. For consistent results, you should set the foreground colour to black and the background colour to white before using `CopyBits`, `CopyMask`, or `CopyDeepMask`. (That said, setting foreground and background colours to something other than black or white can achieve some interesting colouration effects.)

Dithering

You can use **dithering** with `CopyBits` and `CopyDeepMask`. Dithering is a technique involving the mixing of existing colours to create the illusion of a third colour. This is most useful for images displayed on indexed devices, which can only display a limited number of colours at any one time.

You can add dithering to any transfer mode by adding the following constant to the transfer mode:

```
ditherCopy = 64      // Add to source mode for dithering.
```

Copying From Offscreen Graphics Ports

To gracefully display complex images, your application should construct the image in an **offscreen graphics world** and then use `CopyBits` to transfer the image to the onscreen graphics port. (Offscreen graphics worlds are addressed at Chapter 13.)

Scrolling Pixels in the Port Rectangle

You can use `ScrollRect` to scroll the pixels in the port rectangle. `ScrollRect` takes four parameters: the rectangle to scroll, a horizontal distance to scroll, a vertical distance to scroll, and a region reference. `ScrollRect` is a special form of `CopyBits` which copies bits enclosed by a rectangle and stores them within that same rectangle. The vacated area is filled with the current background colour or pixel/bit pattern.

Manipulating Rectangles and Regions

QuickDraw provides many functions for manipulating rectangles and regions. You can use the functions which manipulate rectangles to manipulate any shape based on a rectangle, that is, rounded rectangles, ovals , arcs, and wedges.

For example, you could define a rectangle to bound an oval and then frame the oval. You could then use `OffsetRect` to move the oval's bounding rectangle downwards. Using the offset bounding rectangle, you could frame a second, connected oval to form a figure eight with the first oval. You could then use that shape to help define a region. You could create a second region, and then use `UnionRgn` to create a region from the union of the two.

Manipulating Rectangles

The following summarises the functions for manipulating, and performing calculations on, rectangles:

Function	***Description***
`EmptyRect`	Determine whether a rectangle is an empty rectangle.
`EqualRect`	Determine whether two rectangles are equal.
`InsetRect`	Shrinks or expands a rectangle.
`OffsetRect`	Moves a rectangle.
`PtInRect`	Determines whether a pixel is enclosed in a rectangle.
`PtToAngle`	Calculates the angle from the middle of a rectangle to a point.
`Pt2Rect`	Determines the smallest rectangle that encloses two points.
`SectRect`	Determines whether two rectangles intersect.
`UnionRect`	Calculates the smallest rectangle that encloses two rectangles.

Manipulating Regions

The following summarises the functions for manipulating, and performing calculations on, regions:

Function	***Description***
`CopyRgn`	Makes a copy of a region.
`DiffRgn`	Subtracts one region from another.
`EmptyRgn`	Determines whether a region is empty.
`EqualRgn`	Determines whether two regions have identical sizes, shapes, and locations.
`InsetRgn`	Shrinks or expands a region.
`OffsetRgn`	Moves a region.
`PtInRgn`	Determines whether a pixel is within a region.
`RectInRgn`	Determines whether a rectangle intersects a region.
`RectRgn`	Changes the structure of an existing region to that of a rectangle (using a Rect).
`SectRgn`	Calculates the intersection of two regions.
`SetEmptyRgn`	Sets a region to empty.
`SetRectRgn`	Changes the structure of an existing region to that of a rectangle (using coordinates).
`UnionRgn`	Calculates the union of two regions.
`XorRgn`	Calculates the difference between the union and the intersection of two regions.

Manipulating Polygons

You can use `OffSetPoly` to move a polygon; however, QuickDraw provides no other functions for manipulating polygons.

Scaling Shapes and Regions Within the Same Graphics Port

To scale shapes and regions within the same graphics port, you can use the functions `ScalePt`, `MapPt`, `MapRect`, `MapRgn`, and `MapPoly`.

Highlighting

Highlighting is used when selecting and deselecting objects such as text or graphics. TextEdit, for example, uses highlighting to indicate selected text. If the current highlight colour is, for example, blue, TextEdit draws the selected text, then uses `InvertRgn` to produce a blue background for the text.

The **system highlight colour**, which can be changed by the user at the **Highlight Color** item in the **Appearance** pane of the Appearance control panel, is stored in a low memory global represented by the symbolic name `HiliteRGB`. It can be retrieved using `LMGetHiliteRGB`. You can override the default colour using the function `HiliteColor`. The current colour is copied the graphics port object, and may be retrieved from there using the function `GetPortHiliteColor`.

Color QuickDraw implements highlighting by replacing the background colour with the highlight colour. Another low memory global, represented by the symbolic name `HiliteMode`, contains a byte which represents the current highlight mode. One bit in that byte, represented by the constant `pHiliteBit`, is used to toggle the background and highlight colours.

Because Color QuickDraw resets the highlight bit after performing each drawing operation, your application must always clear the highlight bit immediately before calling `InvertRgn` (or, indeed, any of the other drawing or image-copying function that uses the `patXor` or `srcXor` transfer modes.)

The highlight mode can be retrieved and set using `LMGetHiliteMode` and `LMSetHiliteMode`, and `BitClr` may be used to clear the highlight bit:

```
UInt8 hiliteMode;
...
hiliteMode = LMGetHiliteMode();
BitClr(&hiliteMode,pHiliteBit);
LMSetHiliteMode(hiliteMode);
```

Another way to use highlighting is to add this constant to the transfer mode you pass in calls to the functions `PenMode`, `CopyBits`, `CopyDeepMask` and `TextMode`:

```
hilite = 50  // Add to source or pattern mode for highlighting.
```

Drawing Other Graphics Entities

In addition to drawing lines, rectangles, rounded rectangles, ovals, arcs, wedges, polygons and regions, and text, you can also use QuickDraw to draw the following:

· Cursors.

· Icons.

Cursors and Icons are addressed at Chapter 13.

Saving and Restoring the Graphics Port Drawing State

As stated above, the functions `GetPenState` and `SetPenState` are used to save and restore the graphics pen's location, size, transfer mode, and pattern, and `PenNormal` is used to initialise the pen's size, transfer mode, and pattern.

Typically, an application calls `GetPenState` at the beginning of a function that changes the pen's location, size, transfer mode, and/or pattern and restores the saved state to the pen on exit from that function. Depending on its requirements, an application might also save and restore the graphics port's foreground and background colours, and the text transfer mode, in the same way.

Since the introduction of the Appearance Manager, it has also become necessary to save and restore the pen pixel/bit pattern and background pixel/bit pattern in functions that call the Appearance Manager functions `SetThemeBackground`, `SetThemePen`, and/or `SetThemeWindowBackground`. Recall from Chapter 6 — The Appearance Manager that constants of type `ThemeBrush` are passed in the `inBrush` parameter of these

Appearance Manager functions and that the value in the `inBrush` parameter can represent either a colour or a pattern depending on the current appearance.

Accordingly, in the era of the Appearance Manager, applications which call `SetThemeBackground` and/or `SetThemePen` will need to take measures to save and restore the *complete* graphics port drawing state and, if required, normalise that state.

The following functions are used for saving, restoring, and normalising the graphics port drawing state:

Function	***Description***
`GetThemeDrawingState`	Obtains the drawing state of the current graphics port.
`SetThemeDrawingState`	Sets the drawing state of the current graphics port.
`NormalizeThemeDrawingState`	Sets the current graphics port to the default drawing state.
`DisposeThemeDrawingState`	Releases the memory associated with a reference to a graphics port's drawing state. (Note that this memory may also be released by passing true in the `inDisposeNow` parameter of `SetThemeDrawingState`.)

Information about the current state of the graphics port is stored in a structure of type `ThemeDrawingState`. This is a private data structure.

Main QuickDraw Constants, Data Types and Functions

Constants

Boolean Pattern Modes

```
patCopy      = 8
patOr        = 9
patXor       = 10
patBic       = 11
notPatCopy   = 12
notPatOr     = 13
notPatXor    = 14
notPatBic    = 15
```

Boolean Source Modes

```
srcCopy      = 0
srcOr        = 1
srcXor       = 2
srcBic       = 3
notSrcCopy   = 4
notSrcOr     = 5
notSrcXor    = 6
notSrcBic    = 7
```

Arithmetic Transfer Modes

```
blend        = 32
addPin       = 33
addOver      = 34
subPin       = 35
transparent  = 36
addMax       = 37
subOver      = 38
adMin        = 39
```

Add Dithering to Transfer Modes

```
ditherCopy   = 64
```

Highlighting

```
hilite       = 50
hiliteBit    = 7
pHiliteBit   = 0
```

Pattern List Resource ID for Pattern Resources in the System File

```
sysPatListID = 0
```

Data Types

Point

```
struct Point
{
  short   v;
  short   h;
};

typedef struct Point Point;
typedef Point *PointPtr;
```

Rect

```
struct Rect
{
  short   top;
  short   left;
  short   bottom;
  short   right;
};

typedef struct Rect Rect;
typedef Rect *RectPtr;
```

Region

```
typedef struct OpaqueRgnHandle *RgnHandle;
```

Polygon

```
struct Polygon
{
  short   polySize;
  Rect    polyBBox;
  Point   polyPoints[1];
};

typedef struct Polygon Polygon;
typedef Polygon *PolyPtr, **PolyHandle;
```

PenState

```
struct PenState
{
  Point   pnLoc;
  Point   pnSize;
  short   pnMode;
  Pattern pnPat;
};

typedef struct PenState PenState;
```

Functions

Managing the Graphics Pen

```
void          HidePen(void);
void          ShowPen(void);
void          GetPen(Point *pt);
void          GetPenState(PenState *pnState);
void          SetPenState(const PenState *pnState);
void          PenSize(short width,short height);
void          PenMode(short mode);
void          PenNormal(void);
```

Getting and Setting Foreground, Background , and Pixel Colour

```
void          RGBForeColor(const RGBColor *color);
void          RGBBackColor(const RGBColor *color);
void          GetForeColor(RGBColor *color);
void          GetBackColor(RGBColor *color);
void          GetCPixel(short h,short v,RGBColor *cPix);
void          SetCPixel(short h,short v,const RGBColor *cPix);
```

Creating and Disposing of Pixel Patterns

```
PixPatHandle  GetPixPat(short patID);
PixPatHandle  NewPixPat(void);
void          CopyPixPat(PixPatHandle srcPP,PixPatHandle dstPP);
void          MakeRGBPat(PixPatHandle pp,const RGBColor *myColor);
void          DisposePixPat(PixPatHandle pp);
```

Getting Pattern Resources

```
PatHandle     GetPattern(short patternID);
void          GetIndPattern(Pattern *thePat,short patternListID,short index);
```

Changing the Pen and BackGround Pixel Pattern and Bit Pattern

```
void          BackPixPat(PixPatHandle pp);
void          PenPixPat(PixPatHandle pp);
void          BackPat(const Pattern *pat);
void          PenPat(const Pattern *pat);
```

Drawing Lines

```
void          MoveTo(short h,short v);
void          Move(short dh,short dv);
void          LineTo(short h,short v);
void          Line(short dh,short dv);
```

Drawing Rectangles

```
void          FrameRect(const Rect *r);
void          PaintRect(const Rect *r);
void          FillRect(const Rect *r,ConstPatternParam pat);
void          FillCRect(const Rect *r,PixPatHandle pp);
void          InvertRect(const Rect *r);
void          EraseRect(const Rect *r);
```

Drawing Rounded Rectangles

```
void          FrameRoundRect(const Rect *r,short ovalWidth,short ovalHeight);
void          PaintRoundRect(const Rect *r,short ovalWidth,short ovalHeight);
void          FillRoundRect(const Rect *r,short ovalWidth,short ovalHeight,const Pattern *pat);
void          FillCRoundRect(const Rect *r,short ovalWidth,short ovalHeight,PixPatHandle pp);
void          InvertRoundRect(const Rect *r,short ovalWidth,short ovalHeight);
void          EraseRoundRect(const Rect *r,short ovalWidth,short ovalHeight);
```

Drawing Ovals

```
void          FrameOval(const Rect *r);
void          PaintOval(const Rect *r);
void          FillOval(const Rect *r,const Pattern *pat);
void          FillCOval(const Rect *r,PixPatHandle pp);
void          InvertOval(const Rect *r);
void          EraseOval(const Rect *r);
```

Drawing Arcs and Wedges

```
void          FrameArc(const Rect *r,short startAngle,short arcAngle);
void          PaintArc(const Rect *r,short startAngle,short arcAngle);
void          FillArc(const Rect *r,short startAngle,short arcAngle,const Pattern *pat);
void          FillCArc(const Rect *r,short startAngle,short arcAngle,PixPatHandle pp);
void          InvertArc(const Rect *r,short startAngle,short arcAngle);
void          EraseArc(const Rect *r,short startAngle,short arcAngle);
```

Drawing and Painting Polygons

```
void          FramePoly(PolyHandle poly);
void          PaintPoly(PolyHandle poly);
void          FillPoly(PolyHandle poly,const Pattern *pat);
void          FillCPoly(PolyHandle poly,PixPatHandle pp);
void          InvertPoly(PolyHandle poly);
void          ErasePoly(PolyHandle poly);
```

Drawing Regions

```
void          FrameRgn(RgnHandle rgn);
void          PaintRgn(RgnHandle rgn);
void          FillCRgn(RgnHandle rgn,PixPatHandle pp);
void          EraseRgn(RgnHandle rgn);
void          InvertRgn(RgnHandle rgn);
void          FillRgn(RgnHandle rgn, const Pattern *pat);
```

Setting Text Characteristics

```
void          TextFont(short font);
void          TextFace(short face);
void          TextMode(short mode);
void          TextSize(short size);
void          SpaceExtra(Fixed extra);
void          GetFontInfo(FontInfo *info);
```

Drawing and Measuring Text

```
void          DrawChar(short ch);
void          DrawString(ConstStr255Param s);
void          DrawText(const void *textBuf,short firstByte,short byteCount);
short         CharWidth(short ch);
short         StringWidth(ConstStr255Param s);
```

Copying Images

```
void          CopyBits(const BitMap *srcBits,const BitMap *dstBits,const Rect *srcRect,
              const Rect *dstRect,short mode,RgnHandle maskRgn);
void          CopyMask(const BitMap *srcBits,const BitMap *maskBits,const BitMap *dstBits,
              const Rect *srcRect,const Rect *maskRect,const Rect *dstRect);
void          CopyDeepMask(const BitMap *srcBits,const BitMap *maskBits,const BitMap *dstBits,
              const Rect *srcRect,const Rect *maskRect,const Rect *dstRect,short mode,
              RgnHandle maskRgn);
```

Getting and Setting the Highlight Colour and HighLight Mode

```
void          HiliteColor(const RGBColor *color);
void          LMGetHiliteRGB(RGBColor *hiliteRGBValue);
void          LMSetHiliteRGB(const RGBColor *hiliteRGBValue);
UInt8         LMGetHiliteMode(void);
void          LMSetHiliteMode(UInt8 value);
```

Creating and Disposing of Colour Tables

```
CtabHandle    GetCTable(short ctID);
void          DisposeCTable(CTabHandle cTable);
```

Creating and Managing Polygons

```
PolyHandle    OpenPoly(void);
void          ClosePoly(void);
void          KillPoly(PolyHandle poly);
void          OffsetPoly(PolyHandle poly,short dh,short dv);
```

Creating and Managing Rectangles

```
void          SetRect(Rect *r,short left,short top,short right,short bottom);
void          OffsetRect(Rect *r,short dh,short dv);
void          InsetRect(Rect *r,short dh,short dv);
Boolean       SectRect(const Rect *src1,const Rect *src2,Rect *dstRect);
void          UnionRect(const Rect *src1,const Rect *src2,Rect *dstRect);
Boolean       PtInRect(Point pt,const Rect *r);
void          Pt2Rect(Point pt1,Point pt2,Rect *dstRect);
void          PtToAngle(const Rect *r,Point pt,short *angle);
Boolean       EqualRect(const Rect *rect1,const Rect *rect2);
Boolean       EmptyRect(const Rect *r);
```

Creating and Managing Regions

```
RgnHandle     NewRgn(void);
void          OpenRgn(void);
void          CloseRgn(RgnHandle dstRgn);
void          DisposeRgn(RgnHandle rgn);
void          CopyRgn(RgnHandle srcRgn,RgnHandle dstRgn);
void          SetEmptyRgn(RgnHandle rgn);
void          SetRectRgn(RgnHandle rgn,short left,short top,short right,short bottom);
void          RectRgn(RgnHandle rgn,const Rect *r);
void          OffsetRgn(RgnHandle rgn,short dh,short dv);
void          InsetRgn(RgnHandle rgn,short dh,short dv);
void          SectRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
void          UnionRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
void          DiffRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
void          XorRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
Boolean       PtInRgn(Point pt,RgnHandle rgn);
Boolean       RectInRgn(const Rect *r,RgnHandle rgn);
Boolean       EqualRgn(RgnHandle rgnA,RgnHandle rgnB);
Boolean       EmptyRgn(RgnHandle rgn);
OSErr         BitMapToRegion(RgnHandle region,const BitMap *bMap);
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

```
void          ScalePt(Point *pt,const Rect *srcRect,const Rect *dstRect);
void          MapPt(Point *pt,const Rect *srcRect,const Rect *dstRect);
void          MapRect(Rect *r,const Rect *srcRect,const Rect *dstRect);
void          MapRgn(RgnHandle rgn,const Rect *srcRect,const Rect *dstRect);
void          MapPoly(PolyHandle poly,const Rect *srcRect,const Rect *dstRect);
```

Determining Whether QuickDraw has Finished Drawing

```
Boolean       QDDone(GrafPtr port);
```

Retrieving Color QuickDraw Result Codes

```
short         QDError(void);
```

Managing Port Rectangles and Clipping Regions

```
void          ScrollRect(const Rect *r,short dh,short dv,RgnHandle updateRgn);
void          SetOrigin(short h,short v);
void          PortSize(short width,short height);
void          MovePortTo(short leftGlobal,short topGlobal);
void          GetClip(RgnHandle rgn);
void          SetClip(RgnHandle rgn);
void          ClipRect(const Rect *r);
```

Manipulating Points in Graphics Ports

```
void          GlobalToLocal(Point *pt);
void          LocalToGlobal(Point *pt);
void          AddPt(Point src,Point *dst);
void          SubPt(Point *src,Point *dst);
void          SetPt(Point *pt,short h,short v);
Boolean       EqualPt(Point pt1,Point pt2);
Boolean       GetPixel(short h,short v);
```

Relevant Appearance Manager Data Types and Functions

Data Types

```
typedef struct OpaqueThemeDrawingState *ThemeDrawingState;
```

Functions

```
OSStatus      NormalizeThemeDrawingState(void);
OSStatus      GetThemeDrawingState(ThemeDrawingState *outState);
OSStatus      SetThemeDrawingState(ThemeDrawingState inState,Boolean inDisposeNow);
OSStatus      DisposeThemeDrawingState(ThemeDrawingState inState);
```

Demonstration Program QuickDraw Listing

```
// *******************************************************************************************
// QuickDraw.c                                                            CLASSIC EVENT MODEL
// *******************************************************************************************
//
// This program opens a window in which the results of various QuickDraw drawing operations
// are displayed.  Individual line and text drawing, framing, painting, filling, erasing,
// inverting, copying, etc., operations are chosen from a Demonstration pull-down menu.
//
// To keep the non-QuickDraw code to a minimum, the program contains no functions for
// updating the window or for responding to activate and operating system events.
//
// The program utilises the following resources:
//
// •  A 'plst' resource.
//
// •  'WIND' resources for the main window, and a small window used for the CopyBits
//    demonstration (purgeable) (initially visible).
//
// •  An 'MBAR' resource and associated 'MENU' resources (preload, non-purgeable).
//
// •  Two 'ICON' resources (purgeable) used for the boolean source modes demonstration.
//
// •  Two 'PICT' resources (purgeable) used in the arithmetic source modes demonstration.
//
// •  'STR#' resources (purgeable) containing strings used in the source modes and text
//    drawing demonstrations.
//
// •  Three 'ppat' resources (purgeable), two of which are used in various drawing,
//    framing, painting, filling, and erasing demonstrations.  The third is used in the
//    drawing with mouse demonstration.
//
// •  A 'SIZE' resource with the acceptSuspendResumeEvents, canBackground,
//    doesActivateOnFGSwitch, and isHighLevelEventAware flags set.
//
// *******************************************************************************************

// .................................................................................. includes

#include <Carbon.h>

// ................................................................................... defines

#define rMenubar                128
#define mAppleApplication       128
#define  iAbout                 1
#define mFile                   129
#define  iQuit                  12
#define mDemonstration          131
#define  iLine                  1
#define  iFrameAndPaint         2
#define  iFillEraseInvert       3
#define  iPolygonRegion         4
#define  iText                  5
#define  iScrolling             6
#define  iBooleanSourceModes    7
#define  iArithmeticSourceModes 8
#define  iHighlighting          9
#define  iDrawWithMouse         10
#define  iDrawingState          11
#define rWindow                 128
#define rPixelPattern1          128
#define rPixelPattern2          129
#define rPixelPattern3          130
#define rDestinationIcon        128
#define rSourceIcon             129
```

```
#define rFontsStringList        128
#define rBooleanStringList      129
#define rArithmeticStringList   130
#define rPicture                128
#define MAX_UINT32              0xFFFFFFFF

// .................................................................................. global variables

Boolean   gRunningOnX     = false;
Boolean   gDone;
WindowRef gWindowRef;
Boolean   gDrawWithMouseActivated;
SInt16    gPixelDepth;
Boolean   gIsColourDevice = false;
RGBColor  gWhiteColour    = { 0xFFFF, 0xFFFF, 0xFFFF };
RGBColor  gBlackColour    = { 0x0000, 0x0000, 0x0000 };
RGBColor  gRedColour      = { 0xAAAA, 0x0000, 0x0000 };
RGBColor  gYellowColour   = { 0xFFFF, 0xCCCC, 0x0000 };
RGBColor  gGreenColour    = { 0x0000, 0x9999, 0x0000 };
RGBColor  gBlueColour     = { 0x6666, 0x6666, 0x9999 };

// ............................................................................... function prototypes

void    main                    (void);
void    doPreliminaries         (void);
OSErr   quitAppEventHandler     (AppleEvent *,AppleEvent *,SInt32);
void    doEvents                (EventRecord *);
void    doDemonstrationMenu     (MenuItemIndex);
void    doLines                 (void);
void    doFrameAndPaint         (void);
void    doFillEraseInvert       (void);
void    doPolygonAndRegion      (void);
void    doScrolling             (void);
void    doText                  (void);
void    doBooleanSourceModes    (void);
void    doArithmeticSourceModes (void);
void    doHighlighting          (void);
void    doDrawWithMouse         (void);
void    doDrawingState          (void);
void    doDrawingStateProof     (SInt16);
void    doGetDepthAndDevice     (void);
UInt16  doRandomNumber          (UInt16,UInt16);

// ************************************************************************************** main

void  main(void)
{
  UInt32        seconds;
  MenuBarHandle menubarHdl;
  SInt32        response;
  MenuRef       menuRef;
  EventRecord   eventStructure;
  Boolean       gotEvent;

  // .................................................................................. do prelimiaries

  doPreliminaries();

  // ......................................................................... seed random number generator

  GetDateTime(&seconds);
  SetQDGlobalsRandomSeed(seconds);

  // ............................................................................ set up menu bar and menus

  menubarHdl = GetNewMBar(rMenubar);
  if(menubarHdl == NULL)
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();

  Gestalt(gestaltMenuMgrAttr,&response);
  if(response & gestaltMenuMgrAquaLayoutMask)
  {
    menuRef = GetMenuRef(mFile);
    if(menuRef != NULL)
    {
      DeleteMenuItem(menuRef,iQuit);
      DeleteMenuItem(menuRef,iQuit - 1);
      DisableMenuItem(menuRef,0);
    }

    gRunningOnX = true;
  }

  // ............................................................................ open window

  if(!(gWindowRef = GetNewCWindow(rWindow,NULL,(WindowRef)-1)))
    ExitToShell();

  SetPortWindowPort(gWindowRef);
  UseThemeFont(kThemeSmallSystemFont,smSystemScript);

  // ....... get pixel depth and whether colour device for certain Appearance Manager functions

  doGetDepthAndDevice();

  // ............................................................................ eventLoop

  gDone = false;

  while(!gDone)
  {
    gotEvent = WaitNextEvent(everyEvent,&eventStructure,MAX_UINT32,NULL);
    if(gotEvent)
      doEvents(&eventStructure);
  }
}

// *************************************************************************** doPreliminaries

void  doPreliminaries(void)
{
  OSErr osError;

  MoreMasterPointers(32);
  InitCursor();
  FlushEvents(everyEvent,0);

  osError = AEInstallEventHandler(kCoreEventClass,kAEQuitApplication,
                          NewAEEventHandlerUPP((AEEventHandlerProcPtr) quitAppEventHandler),
                          0L,false);
  if(osError != noErr)
    ExitToShell();
}

// **************************************************************************** doQuitAppEvent

OSErr  quitAppEventHandler(AppleEvent *appEvent,AppleEvent *reply,SInt32 handlerRefcon)
{
  OSErr    osError;
  DescType returnedType;
  Size     actualSize;

  osError = AEGetAttributePtr(appEvent,keyMissedKeywordAttr,typeWildCard,&returnedType,NULL,0,
                            &actualSize);

  if(osError == errAEDescNotFound)
  {
    gDone = true;
    osError = noErr;
  }
  else if(osError == noErr)
    osError = errAEParamMissed;

  return osError;
}

// ********************************************************************************** doEvents

void  doEvents(EventRecord *eventStrucPtr)
{
  SInt32          menuChoice;
  MenuID          menuID;
  MenuItemIndex   menuItem;
  WindowPartCode  partCode;
  WindowRef       windowRef;

  switch(eventStrucPtr->what)
  {
    case kHighLevelEvent:
      AEProcessAppleEvent(eventStrucPtr);
      break;

    case keyDown:
      if((eventStrucPtr->modifiers & cmdKey) != 0)
      {
        menuChoice = MenuEvent(eventStrucPtr);
        menuID = HiWord(menuChoice);
        menuItem = LoWord(menuChoice);
        if(menuID == mFile && menuItem  == iQuit)
          gDone = true;
      }
      break;

    case mouseDown:
      if(partCode = FindWindow(eventStrucPtr->where,&windowRef))
      {
        switch(partCode)
        {
          case inMenuBar:
            menuChoice = MenuSelect(eventStrucPtr->where);
            menuID = HiWord(menuChoice);
            menuItem = LoWord(menuChoice);

            if(menuID == 0)
              return;

            switch(menuID)
            {
              case mAppleApplication:
                if(menuItem == iAbout)
                  SysBeep(10);
                break;

              case mFile:
                if(menuItem == iQuit)
                  gDone = true;
                break;

              case mDemonstration:
                doDemonstrationMenu(menuItem);
                break;
            }
            break;

          case inDrag:
            DragWindow(windowRef,eventStrucPtr->where,NULL);
            break;

          case inContent:
            if(windowRef != FrontWindow())
              SelectWindow(windowRef);
            else
              if(gDrawWithMouseActivated)
                doDrawWithMouse();
            break;
        }
      }
      break;

    case updateEvt:
      windowRef = (WindowRef) eventStrucPtr->message;
      BeginUpdate(windowRef);
      EndUpdate(windowRef);
      break;
  }
}

// ******************************************************************** doDemonstrationMenu

void  doDemonstrationMenu(MenuItemIndex menuItem)
{
  Rect    portRect;
  Pattern whitePattern;

  gDrawWithMouseActivated = false;

  switch(menuItem)
  {
    case iLine:
      doLines();
      break;

    case iFrameAndPaint:
      doFrameAndPaint();
      break;

    case iFillEraseInvert:
      doFillEraseInvert();
      break;

    case iPolygonRegion:
      doPolygonAndRegion();
      break;

    case iText:
      doText();
      break;

    case iScrolling:
      doScrolling();
      break;

    case iBooleanSourceModes:
      doBooleanSourceModes();
      break;

    case iArithmeticSourceModes:
      doArithmeticSourceModes();
      break;

    case iHighlighting:
      doHighlighting();
      break;
    case iDrawWithMouse:
      SetWTitle(gWindowRef,"\pDrawing with the mouse");
      RGBBackColor(&gWhiteColour);
      GetWindowPortBounds(gWindowRef,&portRect);
      FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));
      gDrawWithMouseActivated = true;
      break;

    case iDrawingState:
      doDrawingState();
      break;
  }

  HiliteMenu(0);
}

// ********************************************************************************** doLines

void  doLines(void)
{
  Rect         portRect, newClipRect;
  Pattern      whitePattern, systemPattern, blackPattern;
  RgnHandle    oldClipRgn;
  SInt16       a, b, c;
  RGBColor     theColour;
  UInt32       finalTicks;
  PixPatHandle pixpatHdl;

  PenNormal();

  RGBBackColor(&gBlueColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

  newClipRect = portRect;
  InsetRect(&newClipRect,10,10);
  oldClipRgn = NewRgn();
  GetClip(oldClipRgn);
  ClipRect(&newClipRect);

  // .................................................................................. lines drawn with foreground colour and black pen pattern

  SetWTitle(gWindowRef,"\pDrawing lines with colours");
  RGBBackColor(&gWhiteColour);
  FillRect(&portRect,&whitePattern);

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  for(a=1;a<60;a++)
  {
    b = doRandomNumber(0,portRect.right - portRect.left);
    c = doRandomNumber(0,portRect.right - portRect.left);

    theColour.red   = doRandomNumber(0,65535);
    theColour.green = doRandomNumber(0,65535);
    theColour.blue  = doRandomNumber(0,65535);
    RGBForeColor(&theColour);

    PenSize(a * 2,1);

    MoveTo(b,portRect.top);
    LineTo(c,portRect.bottom);

    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(2,&finalTicks);
  }

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);

  // ........................................................ lines drawn with system-supplied bit patterns

  SetWTitle(gWindowRef,"\pClick mouse for more lines");
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  while(!Button()) ;
  SetWTitle(gWindowRef,"\pDrawing lines with system-supplied bit patterns");
  FillRect(&portRect,&whitePattern);

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  for(a=1;a<39;a++)
  {
    b = doRandomNumber(0,portRect.bottom - portRect.top);
    c = doRandomNumber(0,portRect.bottom - portRect.top);

    theColour.red   = doRandomNumber(0,32767);
    theColour.green = doRandomNumber(0,32767);
    theColour.blue  = doRandomNumber(0,32767);
    RGBForeColor(&theColour);

    GetIndPattern(&systemPattern,sysPatListID,a);
    PenPat(&systemPattern);

    PenSize(1, a * 2);

    MoveTo(portRect.left,b);
    LineTo(portRect.right,c);

    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(5,&finalTicks);
  }

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);

  // ....................................................................... lines drawn with a pixel pattern

  SetWTitle(gWindowRef,"\pClick mouse for more lines");
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  while(!Button()) ;
  SetWTitle(gWindowRef,"\pDrawing lines with a pixel pattern");
  FillRect(&portRect,&whitePattern);

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  if(!(pixpatHdl = GetPixPat(rPixelPattern1)))
    ExitToShell();
  PenPixPat(pixpatHdl);


  for(a=1;a<60;a++)
  {
    b = doRandomNumber(0,portRect.right - portRect.left);
    c = doRandomNumber(0,portRect.right - portRect.left);

    PenSize(a * 2,1);

    MoveTo(b,portRect.top);
    LineTo(c,portRect.bottom);

    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(5,&finalTicks);
  }

  DisposePixPat(pixpatHdl);
```

```
  SetClip(oldClipRgn);
  DisposeRgn(oldClipRgn);

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);

  // .................................................................... lines drawn with pattern mode patXor

  SetWTitle(gWindowRef,"\pClick mouse for more lines");
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  while(!Button()) ;
  SetWTitle(gWindowRef,"\pDrawing lines using pattern mode patXor");

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  RGBBackColor(&gRedColour);
  FillRect(&portRect,&whitePattern);

  PenSize(1,1);
  PenPat(GetQDGlobalsBlack(&blackPattern));
  PenMode(patXor);

  InsetRect(&portRect,10,10);

  for(a = portRect.left,b = portRect.right;a < portRect.right + 1;a++,b--)
  {
    MoveTo(a,portRect.top);
    LineTo(b,portRect.bottom);

    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  }

  for(a = portRect.bottom,b = portRect.top;b < portRect.bottom + 1;a--,b++)
  {
    MoveTo(portRect.left,a);
    LineTo(portRect.right,b);

    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  }

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// *************************************************************************** doFrameAndPaint

void  doFrameAndPaint(void)
{
  SInt16       a;
  Rect         portRect, theRect;
  Pattern      whitePattern;
  UInt32       finalTicks;
  Pattern      systemPattern;
  PixPatHandle pixpatHdl;

  PenNormal();
  PenSize(30,20);

  for(a=0;a<3;a++)
  {
    RGBBackColor(&gWhiteColour);
    GetWindowPortBounds(gWindowRef,&portRect);
    FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

    if(!gRunningOnX)
      SetThemeCursor(kThemeWatchCursor);
```

```
// ........................................................................................ preparation

if(a == 0)
{
  SetWTitle(gWindowRef,"\pFraming and painting with a colour");
  RGBForeColor(&gRedColour);                              // set foreground colour to red
}
else if(a == 1)
{
  SetWTitle(gWindowRef,"\pFraming and painting with a bit pattern");

  RGBForeColor(&gBlueColour);                            // set foreground colour to blue
  RGBBackColor(&gYellowColour);                        // set foreground colour to yellow
  GetIndPattern(&systemPattern,sysPatListID,16);                // get bit pattern for pen
  PenPat(&systemPattern);                                           // set pen bit pattern
}
else if (a == 2)
{
  SetWTitle(gWindowRef,"\pFraming and painting with a pixel pattern");

  if(!(pixpatHdl = GetPixPat(rPixelPattern1)))               // get pixel pattern for pen
    ExitToShell();
  PenPixPat(pixpatHdl);                                           // set pen pixel pattern
}

// ................................................................................ framing and painting

SetRect(&theRect,30,32,151,191);
FrameRect(&theRect);                                                          // FrameRect
MoveTo(30,29);
DrawString("\pFrameRect");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,140,0);
FrameRoundRect(&theRect,30,50);                                          // FrameRoundRect
MoveTo(170,29);
DrawString("\pFrameRoundRect");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,140,0);
FrameOval(&theRect);                                                          // FrameOval
MoveTo(310,29);
DrawString("\pFrameOval");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,140,0);
FrameArc(&theRect,330,300);                                                    // FrameArc
MoveTo(450,29);
DrawString("\pFrameArc");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,-420,186);
PaintRect(&theRect);                                                          // PaintRect
MoveTo(30,214);
DrawString("\pPaintRect");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,140,0);
PaintRoundRect(&theRect,30,50);                                          // PaintRoundRect
MoveTo(170,214);
DrawString("\pPaintRoundRect");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);
```

```
    OffsetRect(&theRect,140,0);
    PaintOval(&theRect);                                                          // PaintOval
    MoveTo(310,214);
    DrawString("\pPaintOval");
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);

    OffsetRect(&theRect,140,0);
    PaintArc(&theRect,330,300);                                                    // PaintArc
    MoveTo(450,214);
    DrawString("\pPaintArc");
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);

    if(!gRunningOnX)
      SetThemeCursor(kThemeArrowCursor);

    if(a < 2)
    {
      SetWTitle(gWindowRef,"\pClick mouse for more");
      QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
      while(!Button()) ;
    }
  }

  DisposePixPat(pixpatHdl);
}

// ********************************************************************** doFillEraseInvert

void  doFillEraseInvert(void)
{
  SInt16       a;
  Rect         portRect, theRect;
  Pattern      whitePattern, fillPat, backPat;
  PixPatHandle fillPixpatHdl, backPixpatHdl;
  UInt32       finalTicks;

  PenNormal();
  PenSize(30,20);

  for(a=0;a<4;a++)
  {
    if(a < 3)
    {
      RGBBackColor(&gWhiteColour);
      GetWindowPortBounds(gWindowRef,&portRect);
      FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));
    }

    if(!gRunningOnX)
      SetThemeCursor(kThemeWatchCursor);

    // ..................................................................... preparation

    if(a == 0)
    {
      SetWTitle(gWindowRef,"\pFilling and erasing with colours");

      RGBForeColor(&gBlueColour);                             // set blue colour for foreground
      RGBBackColor(&gRedColour);                               // set red colour for background
      GetIndPattern(&fillPat,sysPatListID,1);      // get black bit pattern for fill functions
      BackPat(&whitePattern);                           // set white bit pattern for background
    }
    else if(a == 1)
    {
      SetWTitle(gWindowRef,"\pFilling and erasing with bit patterns");

      RGBForeColor(&gBlueColour);                             // set blue colour for foreground
      RGBBackColor(&gYellowColour);                          // set yellow colour for background
      GetIndPattern(&fillPat,sysPatListID,37);            // get bit pattern for fill functions
      GetIndPattern(&backPat,sysPatListID,19);                // get bit pattern for background
      BackPat(&backPat);                                      // set bit pattern for background
    }
    else if (a == 2)
    {
      SetWTitle(gWindowRef,"\pFilling and erasing with pixel patterns");

      if(!(fillPixpatHdl = GetPixPat(rPixelPattern1)))      // get pixel patt - fill functions
        ExitToShell();
      if(!(backPixpatHdl = GetPixPat(rPixelPattern2)))       // get pixel pattern - background
        ExitToShell();
      BackPixPat(backPixpatHdl);                              // set pixel pattern - background
    }
    else if(a == 3)
    {
      SetWTitle(gWindowRef,"\pInverting");

      BackPat(&whitePattern);
      SetRect(&theRect,30,15,570,29);
      EraseRect(&theRect);
      SetRect(&theRect,30,200,570,214);
      EraseRect(&theRect);
    }

    // ....................................................... filling, erasing, and inverting

    SetRect(&theRect,30,32,151,191);
    MoveTo(30,29);
    if(a < 2)
    {
      FillRect(&theRect,&fillPat);                                                // FillRect
      DrawString("\pFillRect");
    }
    else if(a == 2)
    {
      FillCRect(&theRect,fillPixpatHdl);                                         // FillCRect
      DrawString("\pFillCRect");
    }
    else if(a == 3)
    {
      InvertRect(&theRect);                                                     // InvertRect
      DrawString("\pInvertRect");
    }
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);

    OffsetRect(&theRect,140,0);
    MoveTo(170,29);
    if(a < 2)
    {
      FillRoundRect(&theRect,30,50,&fillPat);                               // FillRoundRect
      DrawString("\pFillRoundRect");
    }
    else if(a == 2)
    {
      FillCRoundRect(&theRect,30,50,fillPixpatHdl);                        // FillCRoundRect
      DrawString("\pFillCRoundRect");
    }
    else if(a == 3)
    {
      InvertRoundRect(&theRect,30,50);                                    // InvertRoundRect
      DrawString("\pInvertRoundRect");
    }
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);

    OffsetRect(&theRect,140,0);
```

```
MoveTo(310,29);
if(a < 2)
{
  FillOval(&theRect,&fillPat);                                          // FillOval
  DrawString("\pFillOval");
}
else if(a == 2)
{
  FillCOval(&theRect,fillPixpatHdl);                                    // FillCOval
  DrawString("\pFillCOval");
}
else if(a == 3)
{
  InvertOval(&theRect);                                                // InvertOval
  DrawString("\pInvertOval");
}
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,140,0);
MoveTo(450,29);
if(a < 2)
{
  FillArc(&theRect,330,300,&fillPat);                                     // FillArc
  DrawString("\pFillArc");
}
else if(a == 2)
{
  FillCArc(&theRect,330,300,fillPixpatHdl);                              // FillCArc
  DrawString("\pFillCArc");
}
else if(a == 3)
{
  InvertArc(&theRect,330,300);                                          // InvertArc
  DrawString("\pInvertArc");
}
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,-420,186);
MoveTo(30,214);
if(a < 3)
{
  EraseRect(&theRect);                                                  // EraseRect
  DrawString("\pEraseRect");
}
else
{
  InvertRect(&theRect);                                                // InvertRect
  DrawString("\pInvertRect");
}
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRect(&theRect,140,0);
MoveTo(170,214);
if(a < 3)
{
  EraseRoundRect(&theRect,30,50);                                  // EraseRoundRect
  DrawString("\pEraseRoundRect");
}
else
{
  InvertRoundRect(&theRect,30,50);                                // InvertRoundRect
  DrawString("\pInvertRoundRect");
}
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);
```

```
    OffsetRect(&theRect,140,0);
    MoveTo(310,214);
    if(a < 3)
    {
      EraseOval(&theRect);                                                   // EraseOval
      DrawString("\pEraseOval");
    }
    else
    {
      InvertOval(&theRect);                                                 // InvertOval
      DrawString("\pInvertOval");
    }
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);

    OffsetRect(&theRect,140,0);
    MoveTo(450,214);
    if(a < 3)
    {
      EraseArc(&theRect,330,300);                                             // EraseArc
      DrawString("\pEraseArc");
    }
    else
    {
      InvertArc(&theRect,330,300);                                           // InvertArc
      DrawString("\pInvertArc");
    }
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);

    if(!gRunningOnX)
      SetThemeCursor(kThemeArrowCursor);

    if(a < 3)
    {
      SetWTitle(gWindowRef,"\pClick mouse for more");
      QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
      while(!Button()) ;
    }
  }

  DisposePixPat(fillPixpatHdl);
  DisposePixPat(backPixpatHdl);
}

// ********************************************************************** doPolygonAndRegion

void  doPolygonAndRegion(void)
{
  Rect         portRect, theRect;
  Pattern      whitePattern, backPat;
  PixPatHandle fillPixpatHdl;
  PolyHandle   polygonHdl;
  RgnHandle    regionHdl;
  UInt32       finalTicks;

  SetWTitle(gWindowRef,"\pFraming, painting, filling, and erasing polygons and regions");

  RGBBackColor(&gWhiteColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  // ........................................................................ preparation

  GetIndPattern(&backPat,sysPatListID,17);                    // get bit pattern for background
  BackPat(&backPat);                                          // set bit pattern for background
```

```
if(!(fillPixpatHdl = GetPixPat(rPixelPattern2)))     // get pixel pattern for fill functions
  ExitToShell();
RGBForeColor(&gRedColour);                                  // set red colour for foreground
RGBBackColor(&gYellowColour);                           // set yellow colour for background
PenNormal();

polygonHdl = OpenPoly();                                                      // define polygon
MoveTo(30,32);
LineTo(151,32);
LineTo(96,103);
LineTo(151,134);
LineTo(151,191);
LineTo(30,191);
LineTo(66,75);
ClosePoly();

regionHdl = NewRgn();                                                          // define region
OpenRgn();
SetRect(&theRect,30,218,151,279);
FrameRect(&theRect);
SetRect(&theRect,30,316,151,377);
FrameRect(&theRect);
SetRect(&theRect,39,248,142,341);
FrameOval(&theRect);
CloseRgn(regionHdl);

// ................................................ framing, painting, filling, and erasing

FramePoly(polygonHdl);                                                             // FramePoly
MoveTo(30,29);
DrawString("\pFramePoly (colour)");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetPoly(polygonHdl,140,0);
PaintPoly(polygonHdl);                                                             // PaintPoly
MoveTo(170,29);
DrawString("\pPaintPoly (colour)");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetPoly(polygonHdl,140,0);
FillCPoly(polygonHdl,fillPixpatHdl);                                               // FillCPoly
MoveTo(310,29);
DrawString("\pFillCPoly (pixel pattern)");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetPoly(polygonHdl,140,0);
ErasePoly(polygonHdl);                                                             // ErasePoly
MoveTo(450,29);
DrawString("\pErasePoly (bit pattern)");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

FrameRgn(regionHdl);                                                                // FrameRgn
MoveTo(30,214);
DrawString("\pFrameRgn (colour)");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRgn(regionHdl,140,0);
PaintRgn(regionHdl);                                                                // PaintRgn
MoveTo(170,214);
DrawString("\pPaintRgn (colour)");
QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
Delay(30,&finalTicks);

OffsetRgn(regionHdl,140,0);
```

```
  FillCRgn(regionHdl,fillPixpatHdl);                                            // FillCRgn
  MoveTo(310,214);
  DrawString("\pFillCRgn (pixel pattern)");
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  Delay(30,&finalTicks);

  OffsetRgn(regionHdl,140,0);
  EraseRgn(regionHdl);                                                          // EraseRgn
  MoveTo(450,214);
  DrawString("\pEraseRgn (bit pattern)");
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  Delay(30,&finalTicks);

  KillPoly(polygonHdl);
  DisposeRgn(regionHdl);
  DisposePixPat(fillPixpatHdl);
  BackPat(&whitePattern);

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// ********************************************************************************** doText

void  doText(void)
{
  Rect    portRect, theRect;
  Pattern whitePattern;
  SInt16  windowCentre, a, fontNum, stringWidth;
  Str255  textString;
  UInt32  finalTicks;

  RGBBackColor(&gWhiteColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

  SetWTitle(gWindowRef,"\pDrawing text with default source mode (srcOr)");

  windowCentre = (portRect.right - portRect.left) / 2;
  SetRect(&theRect,windowCentre,portRect.top,portRect.right,portRect.bottom);
  RGBBackColor(&gBlueColour);
  FillRect(&theRect,&whitePattern);

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  for(a=1;a<9;a++)
  {
    // .............................................. set various text fonts, text styles, and foreground colours

    if(a == 1)
    {
      GetFNum("\pGeneva",&fontNum);
      TextFont(fontNum);
      TextFace(normal);
      RGBForeColor(&gRedColour);
    }
    else if(a == 2)
      TextFace(bold);
    else if(a == 3)
    {
      GetFNum("\pTimes",&fontNum);
      TextFont(fontNum);
      TextFace(italic);
      RGBForeColor(&gYellowColour);
    }
    else if(a == 4)
      TextFace(underline);
    else if(a == 5)
    {
      GetFNum("\pHelvetica",&fontNum);
      TextFont(fontNum);
      TextFace(normal);
      RGBForeColor(&gGreenColour);
    }
    else if(a == 6)
      TextFace(bold + italic);
    else if(a == 7)
    {
      GetFNum("\pCharcoal",&fontNum);
      TextFont(fontNum);
      TextFace(condense);
      RGBForeColor(&gBlackColour);
    }
    else if(a == 8)
    {
      TextFace(extend);
    }

    // ........................................................................ set text size

    if(a < 7)
      TextSize(a * 2 + 15);
    else
      TextSize(12);

    // .......... get a string and draw it in the set font, style, size, and foreground colour

    GetIndString(textString,rFontsStringList,a);
    stringWidth = StringWidth(textString);
    MoveTo(windowCentre - (stringWidth / 2),a * 46 - 10);
    DrawString(textString);

    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(30,&finalTicks);
  }

  // ............................................................ reset to Geneva 10pt normal

  GetFNum("\pGeneva",&fontNum);
  TextFont(fontNum);
  TextSize(10);
  TextFace(normal);

  // ............ erase a rectangle, get a string, and use TETextBox to draw it left justified

  SetRect(&theRect,portRect.left + 5,portRect.bottom - 55,portRect.left + 118,
          portRect.bottom - 5);
  EraseRect(&theRect);
  InsetRect(&theRect,5,5);
  GetIndString(textString,rFontsStringList,9);
  RGBForeColor(&gWhiteColour);
  TETextBox(&textString[1],textString[0],&theRect,teFlushLeft);

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// ******************************************************************************** doScrolling

void  doScrolling(void)
{
  Rect         portRect, theRect;
  Pattern      whitePattern;
  PixPatHandle pixpat1Hdl, pixpat2Hdl;
  RgnHandle    oldClipHdl, regionAHdl, regionBHdl, regionCHdl, scrollRegionHdl;
  SInt16       a;
  UInt32       finalTicks;
```

```
SetWTitle(gWindowRef,"\pScrolling pixels");

RGBBackColor(&gWhiteColour);
GetWindowPortBounds(gWindowRef,&portRect);
FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

if(!gRunningOnX)
  SetThemeCursor(kThemeWatchCursor);

if(!(pixpat1Hdl = GetPixPat(rPixelPattern1)))
  ExitToShell();
PenPixPat(pixpat1Hdl);
PenSize(50,0);
SetRect(&theRect,30,30,286,371);
FrameRect(&theRect);
SetRect(&theRect,315,30,571,371);
FillCRect(&theRect,pixpat1Hdl);

if(!(pixpat2Hdl = GetPixPat(rPixelPattern2)))
  ExitToShell();
BackPixPat(pixpat2Hdl);

regionAHdl = NewRgn();
regionBHdl = NewRgn();
regionCHdl = NewRgn();
SetRect(&theRect,80,30,236,371);
RectRgn(regionAHdl,&theRect);
SetRect(&theRect,315,30,571,371);
RectRgn(regionBHdl,&theRect);
UnionRgn(regionAHdl,regionBHdl,regionCHdl);

oldClipHdl = NewRgn();
GetClip(oldClipHdl);
SetClip(regionCHdl);

SetRect(&theRect,80,30,571,371);

scrollRegionHdl = NewRgn();

for(a=0;a<371;a++)
{
  ScrollRect(&theRect,0,1,scrollRegionHdl);
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  theRect.top ++;
  Delay(1,&finalTicks);
}

SetRect(&theRect,80,30,571,371);
BackPixPat(pixpat1Hdl);

for(a=0;a<371;a++)
{
  ScrollRect(&theRect,0,-1,scrollRegionHdl);
  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  theRect.bottom --;
  Delay(1,&finalTicks);
}

SetClip(oldClipHdl);

DisposePixPat(pixpat1Hdl);
DisposePixPat(pixpat2Hdl);
DisposeRgn(oldClipHdl);
DisposeRgn(regionAHdl);
DisposeRgn(regionBHdl);
DisposeRgn(regionCHdl);
DisposeRgn(scrollRegionHdl);
```

```
  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// ********************************************************************** doBooleanSourceModes

void  doBooleanSourceModes(void)
{
  Rect         portRect, theRect;
  Pattern      whitePattern;
  Handle       destIconHdl, sourceIconHdl;
  SInt16       a, b;
  UInt32       finalTicks;
  BitMap       sourceIconMap;
  Str255       sourceString;
  PixMapHandle destinationPixMapHdl;

  SetWTitle(gWindowRef,"\pBoolean source modes");

  RGBForeColor(&gBlackColour);
  RGBBackColor(&gGreenColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));
  SetRect(&theRect,portRect.left,portRect.top,portRect.right,
          (portRect.bottom - portRect.top) / 2);
  RGBBackColor(&gWhiteColour);
  FillRect(&theRect,&whitePattern);

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  destIconHdl = GetIcon(rDestinationIcon);
  sourceIconHdl = GetIcon(rSourceIcon);

  for(a=0;a<2;a++)
  {
    if(a == 1)
    {
      RGBForeColor(&gYellowColour);
      RGBBackColor(&gRedColour);
    }

    SetRect(&theRect,235,a * 191 + 30,299,a * 191 + 94);
    PlotIcon(&theRect,destIconHdl);
    MoveTo(235,a * 191 + 27);
    DrawString("\pDestination");

    SetRect(&theRect,304,a * 191 + 30,368,a * 191 + 94);
    PlotIcon(&theRect,sourceIconHdl);
    MoveTo(304,a * 191 + 27);
    DrawString("\pSource");
  }

  RGBForeColor(&gBlackColour);
  RGBBackColor(&gWhiteColour);

  for(a=0;a<2;a++)
  {
    if(a == 1)
    {
      RGBForeColor(&gYellowColour);
      RGBBackColor(&gRedColour);
    }

    for(b=0;b<8;b++)
    {
      SetRect(&theRect,b * 69 + 28,a * 191 + 121,b * 69 + 92,a * 191 + 185);
      PlotIcon(&theRect,destIconHdl);
    }
  }

  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);

  RGBForeColor(&gBlackColour);
  RGBBackColor(&gWhiteColour);

  HLock(sourceIconHdl);
  sourceIconMap.baseAddr = *sourceIconHdl;
  sourceIconMap.rowBytes = 4;
  SetRect(&sourceIconMap.bounds,0,0,32,32);

  destinationPixMapHdl = GetPortPixMap(GetQDGlobalsThePort());

  for(a=0;a<2;a++)
  {
    if(a == 1)
    {
      RGBForeColor(&gYellowColour);
      RGBBackColor(&gRedColour);
    }

    for(b=0;b<8;b++)
    {
      Delay(30,&finalTicks);
      SetRect(&theRect,b * 69 + 28,a * 191 + 121,b * 69 + 92,a * 191 + 185);
      CopyBits(&sourceIconMap,
               (BitMap *) *destinationPixMapHdl,
               &sourceIconMap.bounds,
               &theRect,
               b,NULL);
      GetIndString(sourceString,rBooleanStringList,b + 1);
      MoveTo(b * 69 + 28,a * 191 + 118);
      DrawString(sourceString);

      QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    }
  }

  HUnlock(sourceIconHdl);

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// ******************************************************************* doArithmeticSourceModes

void  doArithmeticSourceModes(void)
{
  Rect         portRect, sourceRect, destRect;
  Pattern      whitePattern;
  PicHandle    sourceHdl, destinationHdl;
  SInt16       a, b, arithmeticMode = 32;
  PixMapHandle currentPixMapHdl;
  Str255       modeString;
  UInt32       finalTicks;

  SetWTitle(gWindowRef,"\pCopyBits with arithmetic source modes");

  RGBForeColor(&gBlackColour);
  RGBBackColor(&gWhiteColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  if(!(sourceHdl = GetPicture(rPicture)))
    ExitToShell();
  SetRect(&sourceRect,44,21,201,133);
  HNoPurge((Handle) sourceHdl);
  DrawPicture(sourceHdl,&sourceRect);
  HPurge((Handle) sourceHdl);
  MoveTo(44,19);
  DrawString("\pSOURCE IMAGE");

  if(!(destinationHdl = GetPicture(rPicture + 1)))
    ExitToShell();
  HNoPurge((Handle) destinationHdl);
  for(a=44;a<403;a+=179)
  {
    for(b=21;b<274;b+=126)
    {
      if(a == 44 && b == 21)
       continue;
      SetRect(&destRect,a,b,a+157,b+112);
      DrawPicture(destinationHdl,&destRect);
    }
  }
  HPurge((Handle) destinationHdl);

  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);

  currentPixMapHdl = GetPortPixMap(GetWindowPort(gWindowRef));

  for(a=44;a<403;a+=179)
  {
    for(b=21;b<274;b+=126)
    {
      if(a == 44 && b == 21)
       continue;

      Delay(60,&finalTicks);

      GetIndString(modeString,rArithmeticStringList,arithmeticMode - 31);
      MoveTo(a,b - 2);
      DrawString(modeString);

      SetRect(&destRect,a,b,a+157,b+112);

      CopyBits((BitMap *) *currentPixMapHdl,
               (BitMap *) *currentPixMapHdl,
               &sourceRect,&destRect,
               arithmeticMode + ditherCopy,NULL);

      arithmeticMode ++;

      QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    }

  }

  ReleaseResource((Handle) sourceHdl);
  ReleaseResource((Handle) destinationHdl);

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// ************************************************************************** doHighlighting

void  doHighlighting(void)
{
  Rect     portRect, theRect;
  Pattern  whitePattern;
  RGBColor oldHighlightColour;
  SInt16   a;
  UInt8    hiliteVal;
  UInt32   finalTicks;

  SetWTitle(gWindowRef,"\pHighlighting");

  RGBForeColor(&gBlackColour);
  RGBBackColor(&gWhiteColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

  LMGetHiliteRGB(&oldHighlightColour);

  for(a=0;a<3;a++)
  {
    MoveTo(50,a * 100 + 60);
    DrawString("\pClearing the highlight bit and calling InvertRect.");
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(60,&finalTicks);
    SetRect(&theRect,44,a * 100 + 44,557,a * 100 + 104);

    hiliteVal = LMGetHiliteMode();
    BitClr(&hiliteVal,pHiliteBit);
    LMSetHiliteMode(hiliteVal);

    if(a == 1)
      HiliteColor(&gYellowColour);
    else if(a == 2)
      HiliteColor(&gGreenColour);

    InvertRect(&theRect);
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);

    MoveTo(50,a * 100 + 75);
    Delay(60,&finalTicks);
    DrawString("\pClick mouse to unhighlight.  ");
    DrawString("\p(Note:  The call to InvertRect reset the highlight bit ...");
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);

    while(!Button()) ;

    MoveTo(45,a * 100 + 90);
    DrawString("\p... so we clear the highlight bit again before calling InvertRect.)");
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
    Delay(60,&finalTicks);

    LMSetHiliteMode(hiliteVal);

    InvertRect(&theRect);
    QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
  }

  HiliteColor(&oldHighlightColour);

  Delay(60,&finalTicks);
  MoveTo(50,350);
  DrawString("\pOriginal highlight colour has been reset.");

  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
}

// *************************************************************************** doDrawWithMouse

void  doDrawWithMouse(void)
{
  Rect         portRect, drawRect;
  Pattern      whitePattern, blackPattern;
  PixPatHandle pixpatHdl;
  Point        initialMouse, previousMouse, currentMouse;
  UInt16       randomNumber;
  RGBColor     theColour;
  RGBBackColor(&gWhiteColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));

  if(!(pixpatHdl = GetPixPat(rPixelPattern3)))
    ExitToShell();
  PenPixPat(pixpatHdl);
  PenSize(1,1);
  PenMode(patXor);

  GetMouse(&initialMouse);
  drawRect.left = drawRect.right  = initialMouse.h;
  drawRect.top  = drawRect.bottom = initialMouse.v;

  GetMouse(&previousMouse);

  while(StillDown())
  {
    GetMouse(&currentMouse);

    if(currentMouse.v != previousMouse.v || currentMouse.h != previousMouse.h)
    {
      FrameRect(&drawRect);

      if(currentMouse.h >= initialMouse.h)
        drawRect.right = currentMouse.h;
      if(currentMouse.v >= initialMouse.v)
        drawRect.bottom = currentMouse.v;
      if(currentMouse.h <= initialMouse.h)
        drawRect.left = currentMouse.h;
      if(currentMouse.v <= initialMouse.v)
        drawRect.top = currentMouse.v;

      FrameRect(&drawRect);
    }

    previousMouse.v = currentMouse.v;
    previousMouse.h = currentMouse.h;
  }

  FrameRect(&drawRect);

  theColour.red   = doRandomNumber(0,65535);
  theColour.green = doRandomNumber(0,65535);
  theColour.blue  = doRandomNumber(0,65535);
  RGBForeColor(&theColour);

  PenMode(patCopy);
  PenPat(GetQDGlobalsBlack(&blackPattern));
  BackPixPat(pixpatHdl);

  randomNumber = doRandomNumber(0,3);

  if(randomNumber == 0)
    PaintRect(&drawRect);
  else if(randomNumber == 1)
    EraseRoundRect(&drawRect,50,50);
  else if(randomNumber == 2)
    PaintOval(&drawRect);
  else if(randomNumber == 3)
    PaintArc(&drawRect,0,doRandomNumber(0,360));

  BackPat(&whitePattern);
}

// *************************************************************************** doDrawingState

void  doDrawingState(void)
```

```
{
  Rect              portRect, theRect;
  Pattern           whitePattern;
  ThemeDrawingState themeDrawingState;
  SInt16            a;
  UInt32            finalTicks;

  RGBBackColor(&gBlueColour);
  GetWindowPortBounds(gWindowRef,&portRect);
  FillRect(&portRect,GetQDGlobalsWhite(&whitePattern));
  SetWTitle(gWindowRef,"\pSaving and restoring the graphics port drawing state");

  if(!gRunningOnX)
    SetThemeCursor(kThemeWatchCursor);

  NormalizeThemeDrawingState();

  doDrawingStateProof(0);
  Delay(120,&finalTicks);

  GetThemeDrawingState(&themeDrawingState);

  theRect = portRect;
  theRect.right -= 300;

  SetThemeBackground(kThemeBrushListViewBackground,gPixelDepth,gIsColourDevice);
  EraseRect(&theRect);

  theRect.left += 150;

  SetThemeBackground(kThemeBrushListViewSortColumnBackground,gPixelDepth,gIsColourDevice);
  EraseRect(&theRect);

  SetThemePen(kThemeBrushListViewSeparator,gPixelDepth,gIsColourDevice);

  theRect.left -= 150;
  for(a=theRect.top;a<=theRect.bottom;a+=18)
  {
    MoveTo(theRect.left,a);
    LineTo(theRect.right - 1,a);
  }

  Delay(120,&finalTicks);
  doDrawingStateProof(1);
  Delay(120,&finalTicks);

  SetThemeDrawingState(themeDrawingState,true);

  doDrawingStateProof(2);

  if(!gRunningOnX)
    SetThemeCursor(kThemeArrowCursor);
}

// ********************************************************************** doDrawingStateProof

void  doDrawingStateProof(SInt16 phase)
{
  Rect theRect;

  MoveTo(324,phase * 117 + 41);
  if(phase == 0)
    DrawString("\pBefore calls to SetThemePen and SetThemeBackground");
  else if(phase == 1)
    DrawString("\pAfter calls to SetThemePen and SetThemeBackground");
  else if(phase == 2)
    DrawString("\pAfter restoration of graphics port drawing state");

  MoveTo(324,phase * 117 + 54);
  DrawString("\pPen pattern/colour");
  MoveTo(462,phase * 117 + 54);
  DrawString("\pBackgrd pattern/colour");

  SetRect(&theRect,324,phase * 117 + 58,438,phase * 117 + 132);
  PaintRect(&theRect);
  SetRect(&theRect,462,phase * 117 + 58,576,phase * 117 + 132);
  EraseRect(&theRect);

  QDFlushPortBuffer(GetWindowPort(FrontWindow()),NULL);
}

// ******************************************************************** doGetDepthAndDevice

void doGetDepthAndDevice(void)
{
  GDHandle deviceHdl;

  deviceHdl = GetMainDevice();
  gPixelDepth = (*(*deviceHdl)->gdPMap)->pixelSize;
  if(((1 << gdDevType) & (*deviceHdl)->gdFlags) != 0)
    gIsColourDevice = true;
}

// ************************************************************************* doRandomNumber

UInt16  doRandomNumber(UInt16 minimum,UInt16 maximum)
{
  UInt16 randomNumber;
  SInt32 range, t;

  randomNumber = Random();
  range = maximum - minimum + 1;
  t = (randomNumber * range) / 65536;
  return (t + minimum);
}

// *******************************************************************************************
```

Demonstration Program QuickDraw Comments

When this program is run, the user should choose items from the Demonstration menu and click the mouse button when instructed to do so by the advisory text in the window's title bar.

defines

In addition to the usual constants relating to menus and the window, constants are established for pixel pattern, icon, string list, and picture resource IDs.

Global Variables

The fields of the RGBColor global variables are assigned values representing the colours described by the variable names.

main

Random numbers are used by various functions in the demonstration. The call to SetQDGlobalsRandomSeed seeds the random number generator. randSeed is a QuickDraw global variable which holds the seed value for the random number generator. Unless randSeed is modified, the same sequence of numbers will be generated each time the program is run. The parameter to the GetDateTime call receives the number of seconds since midnight, January 1, 1904, a value that is bound to be different each time the program is run.

Note that error handling in main(), as in other areas of the program, is somewhat rudimentary in that the program simply terminates.

doEvents

Within the mouseDown case, at the inContent case, if the mouseDown is within the content region of the window when it is the front window and gDrawWithMouseActivated is true, the function doDrawWithMouse is called.

doDemonstrationMenu

doDemonstrationMenu switches according to the user's choices in the Demonstration menu. In all but the iDrawWithMouse case, the only action taken is to call the relevant function.

Note that the global variable gDrawWithMouseActivated is set to false at function entry, and is set to true within the iDrawWithMouse case (which executes if the user chooses the Draw With Mouse item). Also note that the window's background is filled with the white colour, using the white pattern, within this case.

doLines

doLines demonstrates line drawing using colours, bit patterns, pixel patterns, and with the Boolean pattern mode patXor. doLines also demonstrates modifying the graphics port's clipping region so as to clip drawing to that modified region.

The first line sets the graphics pen's size, pattern, and pattern mode to the defaults. The next three lines fill the window's content area with blue.

The next block sets the window's clipping region to a rectangle 10 pixels inside the port rectangle. The first two lines define such a rectangle. The next two lines save the current clipping region for later restoration. The call to ClipRect establishes the new clipping region by setting it in the graphics port object.

Lines Drawn With Foreground Colour And Black Pen Pattern

After the window title is set, FillRect is called with the white pattern with the background colour is set to white. This fill is clipped to the current clipping region, which is a rectangle 10 pixels inside the port rectangle.

Within the for loop, random numbers between 0 and the width of the port rectangle are assigned to two variables which will be used to specify the starting and finishing horizontal coordinates for each of 60 drawn lines. The fields of an RGBColor variable are also assigned random values, this time between 0 and 65534 (one less than the maximum possible value for a UInt16). The call to RGBColor assigns this random colour as the requested foreground colour. The pen width is increased by two pixels. Finally, the call to MoveTo moves the pen to the random horizontal location at the top of the port rectangle, and the call to LineTo draws a line to the random horizontal location at the bottom of the port rectangle. The line drawing is clipped to the current clipping region.

Lines Drawn With System-Supplied Bit Patterns

This line drawing operation is similar to the previous one except that a system-supplied bit pattern is assigned to the graphics pen and the lines are drawn from left to right rather than top to bottom. The bit patterns are loaded by the call to GetIndPattern and are drawn from the 38 patterns in the 'PAT#' resource in the System file with resource ID sysPatListID (0). The call to PenPat assigns the specified bit pattern to the graphics pen. In this operation, the height of the pen, rather than the width, is increased by two each time around the for loop.

Lines Drawn With A Pixel Pattern

In this line drawing operation, before the for loop is entered, GetPixPat is called to allocate a PixPat structure and initialise it with information from the specified 'ppat' resource. The call to PenPixPat then assigns this pixel pattern to the graphics pen.

After the last line is drawn, DisposePixPat is called to free the memory allocated by the GetPixPat call.

At this point, the clipping region saved at the start of the function is restored, and all of the memory allocated by the NewRgn call is freed.

Lines Drawn With Pattern Mode patXor

This block demonstrates a well-known but nonetheless exotic capability of the humble line when it operates in the pattern mode patXor.

The content area is filled with red, following which the pen size and pen pattern are set to the defaults. The call to PenMode sets the pen mode to patXor The next four lines assign values to four variables which will be used to ensure that the starting and ending locations of each drawn line will be ten pixels inside the port rectangle. The for loops, proceeding clockwise, draw lines from points 10 pixels inside the periphery of the port rectangle through the centre of the rectangle to points on the opposite side of the rectangle. The effect of patXor on any destination pixel is to invert it. For example, assuming a white background and black pen colour, any white pixel in the path of the drawn lines will be turned black and any black pixel will be turned white. This produces a pattern known as a moire (watered silk) pattern.

doFrameAndPaint

doFrameAndPaint demonstrates the use of QuickDraw's framing and painting functions with the exception of those relating to polygons and regions.

At the first two lines, the pen pattern and mode are set to the defaults and the pen size is set to 30 pixels wide and 20 pixels high.

The for loop is traversed three times, once for framing and painting with a colour, once for framing and painting with a bit pattern, and once for framing and painting with a pixel pattern. The first action is to fill the port rectangle with the colour white using the white pattern.

Preparation

The first time around the loop, RGBForeColor is called to set the requested foreground colour to red.

The second time around the loop, RGBForeColor and RGBBackColor are called to set the requested foreground and background colours to, respectively, blue and yellow, GetIndPattern loads one of the system-supplied bit patterns, and PenPat makes that pattern the pen's current bit pattern.

The third time around the loop, a call to GetPixPat loads a 'ppat' resource, creating a new PixPat structure, and a call to PenPixPat assigns that pixel pattern to the pen.

Framing and Painting

In this section, SetRect is used to assign the coordinates of a rectangle to the fields of a Rect structure, and OffsetRect is used to move the rectangle horizontally and vertically between the calls to the various framing and painting functions.

Before doFrameAndPaint exits, DisposePixPat is called to free the memory allocated by the GetPixPat call.

doFillEraseInvert

doFillEraseInvert demonstrates the use of QuickDraw's filling, erasing, and inverting functions with the exception of those relating to polygons and regions.

At the first two lines, the pen pattern and mode are set to the defaults and the pen size is set to 30 pixels wide and 20 pixels high.

The for loop is traversed four times, once for filling and erasing with colours, once for filling and erasing with bit patterns, once for filling and erasing with a pixel patterns, and once for inverting. The first action, on the first three passes only, is to fill the port rectangle with the colour white using the white pattern.

Preparation

The first time around the loop, RGBForeColor and RGBBackColor are called to set the requested foreground and background colours to, respectively, blue and red. In addition, the calls to GetIndPattern and BackPat set the background pattern to black.

The second time around the loop, RGBForeColor and RGBBackColor are called to set the requested foreground and background colours to, respectively, blue and yellow. In addition, GetIndPattern is called twice, once to assign a bit pattern to a Pattern variable which will be passed as the second parameter in calls to FillRect, FillOval, etc., and once, in conjunction with BackPat, to assign a bit pattern to the graphics port's bkPixPat field.

The third time around the loop, GetPixPat is called twice, once to assign a pixel pattern to a the variable which will be passed as the second parameter in calls to FillCRect, FillCOval, etc., and once, in conjunction with BackPixPat, to assign a pixel pattern to the graphics port's bkPixPat field.

The fourth time around the loop, and preparatory to calls to the erasing functions, the call to BackPat sets the background pattern to white. (The calls to SetRect and EraseRect simply erase the existing text in the window.)

Filling, Erasing, and Inverting

In this section, SetRect is used to assign the coordinates of a rectangle to the fields of a Rect structure, and OffsetRect is used to move the rectangle horizontally and vertically between the calls to the various filling, erasing, and inverting functions.

Before doFillEraseInvert exits, DisposePixPat is called twice to free the memory allocated by the two GetPixPat calls.

doPolygonAndRegion

doPolygonAndRegion demonstrates defining a polygon and a region and the use of some of QuickDraw's polygon and region framing, painting, filling, and erasing functions.

Preparation

The calls to GetIndPattern and BackPat set the background pattern to one on the system-supplied bit patterns. The call to GetPixPat gets the pixel pattern to be used by the filling functions. The calls to RGBForeColor and RGBBackColor set the requested foreground and background colours. PenNormal sets the pen's size, pattern mode, and pattern to the defaults.

The OpenPoly call initiates the recording of the polygon definition, the MoveTo and LineTo calls define the polygon, and ClosePoly stops the recording. Note that, in this demonstration, the last vertex is not joined to the first vertex.

The NewRgn call allocates memory for a new region and a region pointer, initialises the contents of the region and make it an empty rectangle. OpenRgn initiates the recording of a region shape. The next seven lines create a region definition comprising two rectangles and an overlapping oval. CloseRgn terminates the recording.

Framing, Painting, Filling, And Erasing

In this section, OffsetPoly and OffsetRgn are used to move the polygon and region horizontally between the calls to the framing, filling, and erasing functions. OffsetPoly modifies the polygon's definition. OffsetRgn adjusts the coordinates of the region.

Before doPolygonAndRegion exits, KillPoly is called to free all the memory allocated by OpenPoly, DisposeRgn is called to free all the memory allocated by NewRgn, DisposePixPat is called to free all the memory allocated by GetPixPat, and the background pattern is set to white.

doText

doText draws text in various fonts, sizes and styles. In addition, the last block demonstrates drawing justified text within a specified rectangle using the TextEdit function TETextBox.

Prior to the for loop, the variable windowCentre is assigned a value which represents a location midway across the port rectangle, and the right half of the content area is filled with blue.

Within the first section of the for loop, the text font is changed using GetFNum and TextFont, the text style is changed using TextFace, and the foreground colour is changed. At the last two sections within the loop, the text size is changed using TextSize, a string is retrieved from a 'STR#' resource, the width of the string in pixels is determined, and the string is drawn centred laterally in the window.

After the loop exits, the text font, size and style are returned to Geneva 10pt plain.

At the final block, a small rectangle is defined at the bottom left of the content area. Because the current background colour is blue, the call to EraseRect erases the rectangle in that colour. The rectangle is then inset by five pixels all round. A string is then loaded from a 'STR#' resource and the foreground colour is set to white. Finally, TETextBox is called to draw the text within the specified rectangle with left justification. (Other available justification constants are teFlushRight and teCenter.)

doScrolling

doScrolling demonstrates scrolling pixels within a specified rectangle, with the operation clipped to a region comprising two unconnected rectangular areas.

The first call to GetPixPat loads a 'ppat' resource. The call to PenPixPat assigns that pixel pattern to the pen, which is then made 50 pixels wide and zero pixels high. A framed rectangle is then drawn in the left half of the window. (Note that, because the pen height is set to zero, the two sides of the rectangle will be drawn but not the top and bottom.) A filled rectangle is then drawn in the right side of the window using the same pixel pattern.

In the next block, another 'ppat' resource is retrieved. The call to BackPixPat makes this pixel pattern the background pixel pattern.

The next block creates a region comprising two separate rectangles, the first one coincident with the "inside" of the framed rectangle and the second one coincident with the whole of the filled rectangle). The current clipping region is then saved and the newly created region is established as the current clipping region.

The following call to SetRect defines a rectangle for the first parameter of the ScrollRect function. Laterally, this extends from the left inside of the framed rectangle to the right hand side of the filled rectangle. The call to NewRgn then creates the empty region required by the ScrollRect calls.

In the first for loop, the pixels within the clipping region within the specified rectangle are scrolled downwards, the top of the rectangle being incremented downwards between calls to ScrollRect. ScrollRect fills the "vacated" areas with the background pattern .

Between the for loops, the rectangle used by ScrollRect is redefined and the background pixel pattern is changed to the pixel pattern used to draw the original rectangles.   The scrolling operation is then repeated, this time in an upwards direction.

Before doScrolling exits, the saved clipping region is restored and all the memory allocated by the GetPixPat and NewRgn calls is freed.

doBooleanSourceModes

doBooleanSourceModes demonstrates the effects of the Boolean source modes in both black-and-white and colour.

The first block fills the content area with green and then fills the top half of the content area with white. This block leaves the foreground colour black and the background colour white.

The next block loads two 32 bit by 32 bit 'ICON' resources. One icon contains the image of a cross and the other contains the image of a square.

The first for loop calls PlotIcon four times, twice to draw the icons in the white area at the top of the window, and twice to draw them in the green area at the bottom of the window. The rectangle passed in the first parameter of the PlotIcon calls expands the icon to 64 pixels by 64 pixels. The calls to RGBForeColor and RGBBackColor cause the icons in the green area to be drawn using a foreground colour of yellow and a background colour of red.

The foreground and background colours are reset to black and white before the second for loop is entered.

The second for loop draws the cross icon eight times across the bottom of the white half of the window. The foreground and background colours are then changed to yellow and red before this process is repeated across the bottom of the green area of the window.

The foreground and background colours are again reset to black and white.

As a preamble to what is to come, note that there is no special data type for an icon. It is simply 128 bytes of bit data arranged as 32 rows of 4 bytes per row. All that is available is a handle to that 128 bytes of data. The intention is to cause the 128 bytes of data which constitutes the square icon to be regarded as bitmap data pointed to by the baseAddr field of a BitMap structure. That way, the CopyBits routine can be used to copy the bitmap into the graphics port.

Because CopyBits is one of those functions which can move memory around, the first action is to lock the icon data in the heap. The address of the square icon image data is then assigned to the baseAddr field of a BitMap structure, the rowBytes field is assigned the value 4, and the bounds field is assigned a rectangle defining the normal icon size.

The final for loop calls CopyBits to copy the bit image into the graphics port sixteen times, overdrawing the previously drawn cross icons. The call to SetRect within the inner for loop defines the expanded destination rectangle which governs the size at which the image will be drawn. This rectangle is passed in the destRect parameter of the CopyBits call. Note that, in the CopyBits call, the value passed in the tMode (transfer mode) parameter is incremented each time through the loop so that the square image overdraws the cross image once in each of the eight available Boolean source modes. The three lines following the CopyBits call retrieve the appropriate string containing the relevant source mode from the specified 'STR#' resource and draw this string above each copied image.

The last line unlocks the icon image data.

doArithmeticSourceModes

doArithmeticSourceModes demonstrates the effects of the arithmetic source modes.

Since CopyBits will be called, the foreground and background colours are set to black and white respectively. The call to FillRect clears the window to white.

The first call to GetPicture loads a 'PICT' resource into a Picture structure. (Since the 'PICT' resource is purgeable, it is made non-purgeable immediately it is retrieved, used immediately, and immediately made purgeable again.) The call to DrawPicture draws the picture in the top left of the window, where it is labelled as the source image.

The second call to GetPicture loads another 'PICT' resource which will be used as the destination image. The first for loop draws this picture in the window at eight separate locations, these locations being determined by the rectangle passed in the first parameter of the DrawPicture calls.

The last for loop is traversed once for each of the eight arithmetic source modes. CopyBits is called eight times to overdraw the destination images with the source image. Note that the value in the tMode (transfer mode) parameter of the CopyBits call is incremented each time around the loop. Note also that, each time around the loop, a new string is retrieved from a 'STR#' resource and drawn above the destination image.

Before doArithmeticSourceModes exits, ReleaseResource is called twice to free the memory allocated by the GetPicture calls.

doHighlighting

doHighlighting demonstrates highlighting, first with the colour set by the user in the Appearance pane of the Appearance control panel (Mac OS 8/9) or in System Preferences (Mac OS X), and then with two colours set by the program.

Firstly, the highlight colour set by the user is saved via a call to LMGetHiliteRGB.

The for loop is traversed three times. On the second and third traverses, the highlight colour is changed.

Within the for loop, a copy of the value at the low memory global HiliteMode is retrieved using LMGetHiliteMode, BitClr is called to clear the highlight bit, and LMSetHiliteMode is called to set to low memory global to this new value. At the if/else block, the highlight colour is changed if this is the second or third time around the loop. With the highlight bit cleared, InvertRect is called to invert a specified rectangle.

Note that the call to InvertRect resets the highlight bit. Accordingly, when the user clicks the mouse button, the highlight bit is cleared once again before InvertRect is called once again. This second call restores the colour in the specified rectangle to the background colour.

Before the doHighLighting function returns, it sets the highlight colour to the saved highlight colour.

doDrawWithMouse

doDrawWithMouse demonstrates the use of the mouse to define bounding rectangles for QuickDraw shape drawing functions. It also demonstrates the implementation of the "rubber band" rectangle commonly used to provide visual feedback to the user as he drags the mouse during such operations. (While the mouse button remains down, the "rubber-band" rectangle is continually erased and redrawn as the mouse is moved. It is erased when the mouse button is released.)

doDrawWithMouse is called when a mouse-down occurs in the window while it is the front window, provided that the global variable gDrawWithMouseActivated is set to true.

The call to GetPixPat loads a 'ppat' resource containing a small 8 pixel by 8 pixel pattern. This pixel pattern is assigned to the pen by the call to PenPixPat. The call to PenSize makes the pen size one pixel high by one pixel wide. The pen pattern mode is then set to patXOr. (Note: For a black-and-white "rubber band", replace the PenPixPat call with:

```
  PenPat(GetQDGlobalsGray(&grayPattern));
```

The call to GetMouse saves the initial mouse location to a Point variable. The contents of the fields of this variable will remain unchanged. Those coordinates are also used to initialise the left and top fields of the Rect variable drawRect.

The next call to GetMouse assigns the initial location of the mouse to another Point variable. The contents of the fields of this variable will continually change as the mouse is dragged.

The while loop continues to execute while the mouse button remains down. Within the loop, the current mouse location is retrieved and compared with the previous mouse location (the first if statement). If the mouse has moved:

- FrameRect is called to draw the framed rectangle.

- If the current mouse horizontal coordinate is greater than or equal to the initial horizontal mouse coordinate, the current mouse horizontal coordinate is assigned to the right field of the rectangle.

- If the current mouse vertical coordinate is greater than or equal to the initial vertical mouse coordinate, the current mouse vertical coordinate is assigned to the bottom field of the rectangle.

- If the current mouse horizontal coordinate is less than or equal to the initial horizontal mouse coordinate, the current mouse horizontal coordinate is assigned to the left field of the rectangle.

- If the current mouse vertical coordinate is less than or equal to the initial vertical mouse coordinate, the current mouse vertical coordinate is assigned to the top field of the rectangle.

- FrameRect is called again with the newly defined rectangle passed in.

Because the drawing mode is patXor, the first call to FrameRect erases the old rectangle. Because FrameRect is only called if the mouse has moved, the flicker which would otherwise occur when the mouse is stationary is avoided.

Below the if block, and preparatory to the next comparison of current and previous mouse location, the current mouse location becomes the previous mouse location.

When the mouse button is released:

- The final call to FrameRect erases the final "rubber-band" rectangle.

- The foreground colour is set to a random colour, the pen pattern mode is set to patCopy, the pen pattern is set to black, and the background pixel pattern is set to that previously used to draw the "rubber band".

- The rectangle as at mouse button release is used in calls to QuickDraw painting and erasing functions to draw rectangles, round rectangles, ovals, and arcs. Just which function is called depends on the value returned by the call to doRandomNumber.

- The background pattern is set to white.

doDrawingState

doDrawingState is similar to the function doDrawListView in the demonstration program Appearance, the difference being that, in doDrawingState, the drawing state is saved at entry and restored at exit.

Note that the call to NormalizeThemeDrawingState or is included in this function for demonstration purposes only. Ordinarily, this function would be called (if required) at other points in an application.

The call to GetThemeDrawingState saves the drawing state prior to the calls to the Appearance Manager functions SetThemeBackground and SetThemePen, which will change either the colour or the pattern settings in the graphics port.

The call to SetThemeDrawingState restores the saved drawing state.

The intervening code simply draws a Mac OS 8/9-type list view in the left half of the window.

The calls to doDrawingStateProof are also for demonstration purposes only. As will be seen, this function simply draws rectangles in the right half of the window in the pen and background colours and patterns as they were after the call to NormalizeThemeDrawingState, after the calls to the Appearance Manager functions, and after the call to SetThemeDrawingState.

doDrawingStateProof

doDrawingStateProof is called by doDrawingState to draw rectangles in the right half of the window in the pen and background colours and patterns as they were after the call to NormalizeThemeDrawingState, after the calls to the Appearance Manager functions, and after the call to SetThemeDrawingState.

doRandomNumber

doRandomNumber are incidental to the demonstration.

The use of the QuickDraw random number generator is quite adequate for the purposes of this demonstration. However, a professional programmer would not regard it as measuring up to the minimal standards of a "serious" random number generator. (See the article on random number generators at http://www.mactech.com/articles/mactech/Vol.08/08.03/RandomNumbers/index.html.)