12

DRAWING WITH QUICKDRAW

Includes Demonstration Program QuickDraw

Mathematical Foundations of QuickDraw

QuickDraw defines the following mathematical constructs which are widely used in its functions and data types:

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

The Coordinate Plane

QuickDraw maintains a **global coordinate** system for the entire potential drawing space. The screen on which QuickDraw displays images represents a small part of a large global coordinate plane. The global coordinate plane is bounded by the limits of QuickDraw coordinates, which range from -32768 to 32767. The (0,0) origin point of the global coordinate plane is assigned to the upper-left corner of the screen. From there, 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 (ordinarily labelled v) and a horizontal coordinate (ordinarily labelled h).

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

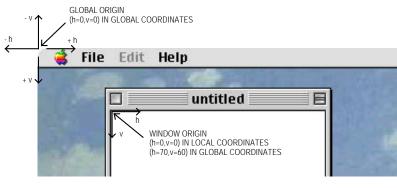


FIG 1 - LOCAL AND GLOBAL COORDINATE SYSTEMS

The intersection of (imaginary) horizontal and vertical grid lines on the coordinate plane marks a **point**. There is a distinction between points on the coordinate grid and **pixels** (the dots which make up the visible image on the screen). 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.

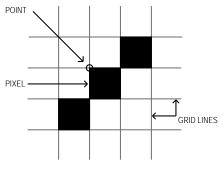


FIG 2 - POINTS AND PIXELS

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

Rectangles are used to define active areas on the screen, to assign coordinate systems to graphics entities, and to specify the sizes and locations for various graphics operations. Rectangles, like points, are mathematical entities which have no direct representation on the screen. Just as points are infinitely small, the borders of the rectangle are infinitely thin.

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 an **empty rectangle**, that is, one that contains no data.

Regions

One of QuickDraw's most powerful features is to work with regions of arbitrary size, shape and complexity. 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.

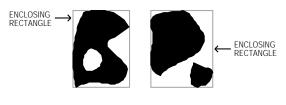


FIG 3 - TWO REGIONS

The data type for regions is Region:

```
struct Region
{
    unsigned short rgnSize; // Size in bytes.
    Rect rgnBBox; // Enclosing rectangle.
    ... // More data if region is not rectangular.
};
typedef struct Region Region;
typedef Region *RgnPtr;
typedef RgnPtr *RgnHandle;
```

The rgnSize field contains the size, in bytes, of the region. The maximum size is 64 KB.

The rgnBBox field is a rectangle which completely encloses the region. The simplest region is a rectangle. In this case, the rgnBBox field defines the entire region, and there is no optional region data. For rectangular regions (or empty regions), the rgnSize field contains 10. The data for more complex regions is stored in a proprietary format.

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 colour 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 pnLoc field of the colour 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 width and height (in pixels) stored in the pnSize field of the colour 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 colour 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 pnVis field of 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

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, siz and pattern.	
PenNormal	Resets the pen size, transfer mode, and pattern to the state initialised when the colour graphics port was opened.

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

Foreground and Background Colour

Foreground Colour

The function RCBForecol or is used to assign a *requested* foreground colour to the rgbFgCol or field of the colour graphics port. The pixel value determined by the Color Manager to represent the closest available match for the device is stored in the fgCol or field. The colour represented by the pixel value in the fgCol or field is the colour actually used as the foreground colour.

If your application uses the Palette Manager, 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 assign a requested background colour to the rgbBkColor field of the colour graphics port. The pixel value determined by the Color Manager to represent the closest available match for the device is stored in the bkColor field. The colour represented by the pixel value in the bkColor field is the colour actually used as the background colour.

If your application uses the Palette Manager, 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 in the colour graphics port's fgCol or field, you can assign a pixel pattern to the colour graphics port's pnPixPat field using the function PenPixPat. (Initially, the pixel pattern in the colour graphics port's pnPixPat field is all-"black". When you assign a non-all-"black" pattern, the pattern in the pnPixPat field 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 PenPixPat to assign the pattern to the pnPixPat field.

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 assign a pixel pattern to the colour graphics port's bkPixPat field using the function BackPixPat. (Initially, the pixel pattern in the colour graphics port's bkPixPat field is all-"white". When you assign a non-all-"white" pattern, the pattern in the bkPixPat field 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 to the pnPixPat field by calling PenPat and passing in the bit pattern contained in the QuickDraw global variable black as follows:

PenPat(&qd. bl ack);

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

BackPat(&qd.white);

When you use the PenPat and BackPat functions, QuickDraw constructs a pixel pattern equivalent to the bit pattern, The colour 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 assign other bit patterns to the pnPixPat and bkPixPat fields of the colour 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

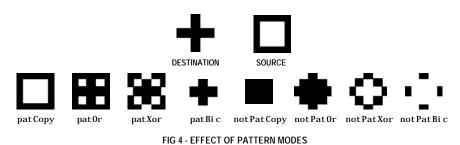
Pattern modes may be assigned to the pnMode field of the colour graphics port using the PenMode function. The modes are represented by eight constants, each of which relates to a specific Boolean operation (COPY, OR, XOR, and BIC (for bit clear) and their inverse variants.

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.	
pat0r	Apply foreground colour.	Leave alone.	
patXor	Invert.	Leave alone.	
patBi c	Apply background colour.	Leave alone.	
notPatCopy	Apply background colour.	Apply foreground colour.	
notPatOr	Leave alone.	Force black.	
notPatXor	Leave alone.	Invert.	

notPatBi c	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 assigned to the pnMode field of the colour graphics port.



Boolean Source Modes

Boolean source modes may be assigned to the t_{xMode} field of the colour graphics port using the function $T_{extMode}$, 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. In general, for pixel images, you will probably want to use the srcCopy mode (which causes the destination pixels to be overwritten completely) or one of the arithmetic source modes.

Arithmetic Source Modes

Arithmetic source modes may be assigned to both the pnMode and txMode fields of the colour 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. Although rarely used by applications, arithmetic transfer modes produce predictable results on indexed devices because they work with RGB colours rather than with colour table indexes. The arithmetic source modes and their effects in a colour environment are as follows:

Constant	Value	Description	
bl end	32	Replace destination pixel with a blend of the source and destination pixel colours. If the destination is a bitmap or 1-bit pixel image, revert to srcCopy mode.	
addPi n	33	Replace destination pixel with the sum of the source and destination pixel colours up to a maximum allowable value. If the destination is a bitmap or 1-bit pixel image, revert to srcBi c mode.	
value of the red, green or blue component exceeds 65,536, then subtract		Replace destination pixel 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. If the destination is a bitmap or 1-bit pixel image, revert to srcXor mode.	
		Replace destination pixel with the difference of the source and destination pixel colours, but not less than a minimum allowable value. If the destination is a bitmap or 1-bit pixel image, revert to src0r mode.	
transparent	36	Replace the source and destination pixel with the source pixel if the source pixel is not equa the background colour.	
		Compare the source and destination pixels, and replace the destination pixel with the colour containing the greater saturation of each of the RGB components. If the destination is a bitmap or 1-bit pixel image, revert to srcBic mode.	
the value of the red, green or blue is less than 0, add the negative result to $65,536$. If t		Replace destination pixel 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. If the destination is a bitmap or 1-bit pixel image, revert to srcXor mode.	
adMi n	39	Compare the source and destination pixels, and replace the destination pixel with the colour containing the lesser saturation of each of the RGB components. If the destination is a bitmap or 1-bit pixel image, revert to src0r mode.	

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.	
Li neTo	Draws a line from the current pen location to the specified location, in local coordinates.	
Li ne	Draws a line a specified distance from the graphics pen's current location.	

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

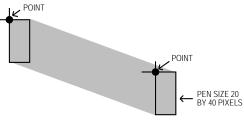


FIG 5 - A LINE DRAWN BY LineTo OR Line

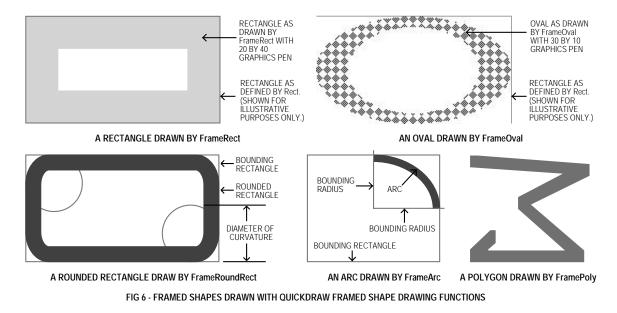
Functions for Drawing Framed Shapes

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

Function	Description
FrameRect	Draws a rectangle, the position and size of which are defined by a $_{ m Rect}$ structure.
Frame0val	Draws an oval, the position and size of which are determined by a bounding rectangle defined by a ${\tt Rect}~$ 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 oval Width and oval Height 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.

Functions for drawing framed shapes are as follows:

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.



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 $0_{penPoly}$ and then call Li_{neTo} a number of times to create lines from the first vertex to the second, from the second vertex to the third, and so on. 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 FrameRegi on.

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 (that is, the pattern in the pnPixPat field of the colour graphics port). 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.

Function	Description			
PaintRect	Fills a rectangle with the current foreground colour or graphics pen pixel/ bit pattern.			
Pai nt0val	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.			
Pai nt Pol y	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.			
Fill0val	Fills an oval with a specified bit pattern.			
FillCOval	Fills an oval with a specified pixel pattern.			
Fill RoundRect 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.			
Fill CArc Fills a wedge of an oval with a specified pixel pattern.				
FillPoly	Fills a polygon with a specified bit pattern.			
Fill CPol y 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.			

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

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.

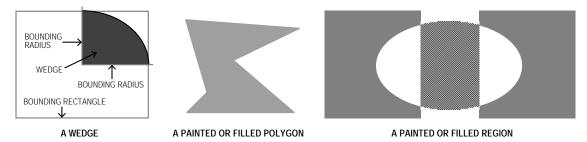


FIG 7 - A WEDGE, A PAINTED OR FILLED POLYGON, AND A PAINTED OR FILLED REGION

Erasing and Inverting Shapes

Erasing a shape fills both its outline and its interior with the background colour or background pixel/bit pattern (that is, the pattern in the bkPixPat field of the colour graphics port). 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.	
Erase0val 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.	
ErasePol y	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.	
Invert0val	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. Pictures provide a form of graphic data exchange: one program can draw something that was defined in another program, with great flexibility and without having to know any details about what is being drawn. 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 - Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

Drawing Text

Text is just another form of graphics, as is evidenced by the colour graphics port text-related fields txFont, txFace, txSize, txMode, and spExtra. QuickDraw functions are available for changing the values in these fields.

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.

Fonts are resources, and the font family ID is a resource ID. You can get the font family ID using Get FNum¹ For example, the following sets the current font to Palatino:

short 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 of usage are as follows:

```
TextFace(bold);// Set to bold.TextFace(bold | italic);// Set to bold and italic.)TextFace(thePort->txFace | bold);// Add bold to existing.TextFace(thePort->txFace &~ 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 application-defined text-justification functions.

Transfer Mode

The transfer mode initially assigned to the t_{xMode} field of the colour graphics port is the Boolean source mode src0r. This mode causes the colour of the glyph² 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 src0r 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.

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.

 $^{^2}$ A glyph is the visual representation of a character.

Copying Pixel Images Between Graphics Ports

QuickDraw provides the following three primary image-processing functions:

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

Coercion of CGrafPtr Data Type to GrafPtr Data Type

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, not a pixel map, in their source and destination parameters.

Fig 9 shows the relative locations of the first four bytes of the portBits field in a graphics port and the portPixMap field in a colour graphics port. The portBits field is actually a structure of type BitMap, and the first four bytes of that structure (baseAddr) are a pointer to a bit image. portPixMap is a handle to a PixMap structure, the first four bytes of which are a pointer to the pixel map's image data.

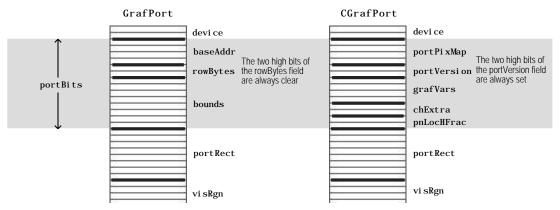


FIG 9 - FIRST 27 BYTES OF GrafPort AND CGrafPort STRUCTURES

When you use CopyBits, CopyMask, and CopyDeepMask to copy pixel images between colour graphics ports, you must coerce each port's CGrafPtr data type to a GrafPtr data type, dereference the portBits fields of each and then pass these "bitmaps" in the source and destination parameters. For example, if

your application copies a pixel image from a colour graphics port called, say, myColourPort, you could specify (GrafPtr) myColourPort->portBits in the source parameter.

All this works because:

- In a CGrafPort structure, the two high bits of the portVersion field are always set. These bits in a GrafPort structure are the two high bits in the portBits.rowBytes field, which are always clear.
- By looking at these bits, 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 copying mask. The values of pixels in the mask act as weights that proportionally select between source and destination pixel values.

On indexed devices, pixel images are always copied using the colour table of the source Pi xMap structure for source colour information, and using the colour table of the current GDevice structure for destination colour information. The colour table attached to the destination Pi xMap is ignored.

When the Pi xMap structure for the mask is 1 bit deep, it has the same effect as a bitmap mask, that is, a black bit in the mask means that the destination pixel will take the colour of the source pixel and a white bit in the mask means that the destination pixel is to retain its current colour. When masks have Pi xMap structures with pixel depths greater than 1, Colour QuickDraw takes a weighted average between the colours in the source and destination Pi xMap structures. Within each pixel, the calculation is done in RGB colour, on a colour component basis. As an example, a red mask (that is, one with high values for the red components of all pixels) filters out red values coming from the source pixel image.

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 used by these functions to mix existing colours together to create the illusion of a third colour that may be unavailable on an indexed device, and to improve images that you shrink when copying them from a direct device to an indexed device.

If you specify a destination rectangle that is smaller than the source rectangle when using CopyBits, CopyMask, CopyDeepMask on a direct device, Color QuickDraw automatically uses an averaging technique to produce the destination pixels, maintaining high-quality images when shrinking them. On indexed devices, Color QuickDraw averages these pixels only when you explicitly specify dithering.

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.

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 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.)

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 handle. 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 <code>OffsetRect</code> 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 <code>Uni onRgn</code> 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.		
Equal Rect	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.		
PtToAngl e	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.
Equal Rgn	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).
Uni onRgn	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 Color 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 to the rgbHiliteColor field of the GrafVars structure, a handle to which is stored in the grafVars field of the colour graphics port structure.

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.

Color QuickDraw resets the highlight bit after performing each drawing operation, so your application should 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, which are 16-by-16 pixel images which map the user's movements of the mouse to relative locations on the screen.
- Icons, which are images that an object, concept, or message. Icons are stored as resources.

Cursors and Icons are addressed at Chapter 13 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.)

Saving and Restoring the Drawing Environment

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 colour 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 theme. If it is a colour, that colour will be assigned to the relevant field of the graphics port structure, that is, the rgbFgCol or or rgbBkCol or field. If it is a pattern, that pattern will be assigned to the relevant field of the colour graphics port structure, that is, the pnPixPat or bkPixPat field.

Accordingly, in the era of the Appearance Manager, you should furnish an application which calls SetThemeBackground, SetThemePen, and/or SetThemeWindowBackground with functions for saving and restoring the *complete* drawing environment and, if required, a function for re-initialising that environment. With regard to the former two functions, you could establish a data structure as shown in the following example, and provide applications-defined functions for saving the drawing environment to, and restoring it from, the fields of such a structure:

struct drawingEnvironment

	PenState	penLocSi zeModePat;	11	Pen location, size, mode, pattern.
	RGBCol or	requestedForeCol our;	11	rgbFgColor field of colour graphics port.
	RGBCol or	requestedBackCol our;	11	rgbBkColor field of colour graphics port.
	SInt16	<pre>textTransferMode;</pre>	11	txMode field of colour graphics port.
	Pi xPat Handl e	penPi xel Pattern;	11	pnPixPat field of colour graphics port.
	Pi xPat Handl e	backPi xel Pattern;	11	bkPixPat field of colour graphics port.
	Pattern	penBitPattern;	11	If pen pixel pattern is a bit pattern.
	Pattern	backBitPattern;	11	If background pixel pattern is a bit pattern.
3.				•

typedef struct drawingEnvironment drawingEnvironment;

Your function for re-initialising the drawing environment should:

- Call PenNormal to initialise the pen location, size, mode, and pattern.
- Call RGBForeColor and RGBBackColor to set the foreground and background colours to, respectively, black and white.
- Call Text Mode with the Boolean source mode src0r.
- Call BakPat with the pattern in the QuickDraw global variable white.

Constants

Boolean Pattern Modes

patCopy	=	8
pat0r	=	9
patXor	=	10
patBic	=	11
notPatCopy	=	12
notPatOr	=	13
notPatXor	=	14
notPatBi c	=	15

Boolean Source Modes

srcCopy	= 0
src0r	= 1
srcXor	= 2
srcBi c	= 3
notSrcCopy	= 4
notSrc0r	= 5
notSrcXor	= 6
notSrcBi c	= 7
ditherCopy	= 64
10	

Arithmetic Transfer Modes

bl end	=	32
addPi n	=	33
add0ver	=	34
subPi n	=	35
transparent	=	36
addMax	=	37
sub0ver	=	38
adMi n	=	39
ditherCopy	=	64

Highlighting

hilite	=	50
hiliteBit	=	7
pHiliteBit	=	0

Pattern List Resource ID for Pattern Resources in the System File

sysPatListID = 0

Data Types

typedef unsigned char PixelType;

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

```
struct Region
{
    short rgnSize;
    Rect rgnBBox;
};
```

typedef struct Region Region; typedef Region *RgnPtr, **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

Initialising QuickDraw

void InitGraf(void *globalPtr);

Managing the Graphics Pen

voi d voi d	Hi dePen(voi d); ShowPen(voi d);
voi d	GetPen(Point *pt);
voi d	<pre>GetPenState(PenState *pnState);</pre>
voi d	<pre>SetPenState(const PenState *pnState);</pre>
voi d	<pre>PenSize(short width, short height);</pre>
voi d	<pre>PenMode(short mode);</pre>
voi d	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

```
PixPatHandleGetPixPat(short patID);PixPatHandleNewPixPat(void);voidCopyPixPat(PixPatHandle srcPP, PixPatHandle dstPP);
```

voi d	<pre>MakeRGBPat(PixPatHandle pp, const RGBColor *myColor);</pre>
voi d	DisposePixPat(PixPatHandle pp);

Getting Pattern Resources

PatHandl e	<pre>GetPattern(short patternID);</pre>	
voi d	GetIndPattern(Pattern *thePat, short patternListID, short index));

Changing the Pen and BackGround Pixel Pattern and Bit Pattern

voi d	<pre>BackPixPat(PixPatHandle pp);</pre>
voi d	<pre>PenPixPat(PixPatHandle pp);</pre>
voi d	<pre>BackPat(const Pattern *pat);</pre>
voi d	<pre>PenPat(const Pattern *pat);</pre>

Drawing Lines

voi d	<pre>MoveTo(short h, short v);</pre>
voi d	Move(short dh, short dv);
voi d	<pre>LineTo(short h, short v);</pre>
voi d	Line(short dh, short dv);

Drawing Rectangles

voi d	<pre>FrameRect(const Rect *r);</pre>
voi d	<pre>PaintRect(const Rect *r);</pre>
voi d	<pre>FillRect(const Rect *r,ConstPatternParam pat);</pre>
voi d	FillCRect(const Rect *r, PixPatHandle pp);
voi d	<pre>InvertRect(const Rect *r);</pre>
voi d	<pre>EraseRect(const Rect *r);</pre>

Drawing Rounded Rectangles

voi d	FrameRoundRect(const Rect *r,short ovalWidth,short ovalHeight);
voi d	PaintRoundRect(const Rect *r,short ovalWidth,short ovalHeight);
voi d	FillRoundRect(const Rect *r, short ovalWidth, short ovalHeight, const Pattern *pat;
voi d	FillCRoundRect(const Rect *r,short ovalWidth,short ovalHeight,PixPatHandle pp);
voi d	InvertRoundRect(const Rect *r, short ovalWidth, short ovalHeight);
voi d	EraseRoundRect(const_Rect_*r,short_ovalWidth,short_ovalHeight);

Drawing Ovals

voi d	<pre>FrameOval(const Rect *r);</pre>
voi d	<pre>PaintOval(const Rect *r);</pre>
voi d	<pre>FillOval(const Rect *r, const Pattern *pat);</pre>
voi d	<pre>FillCOval(const Rect *r, PixPatHandle pp);</pre>
voi d	<pre>InvertOval(const Rect *r);</pre>
voi d	<pre>EraseOval(const Rect *r);</pre>

Drawing Arcs and Wedges

voi d	FrameArc(const Rect *r,short startAngle,short arcAngle);
voi d	PaintArc(const Rect *r, short startAngle, short arcAngle);
voi d	FillArc(const Rect *r, short startAngle, short arcAngle, const Pattern *pat);
voi d	FillCArc(const Rect *r, short startAngle, short arcAngle, PixPatHandle pp);
voi d	InvertArc(const Rect *r,short startAngle,short arcAngle);
voi d	<pre>EraseArc(const Rect *r, short startAngle, short arcAngle);</pre>

Drawing and Painting Polygons

voi d	<pre>FramePoly(PolyHandle poly);</pre>
voi d	PaintPoly(PolyHandle poly);
voi d	<pre>FillPoly(PolyHandle poly, const Pattern *pat);</pre>
voi d	<pre>FillCPoly(PolyHandle poly, PixPatHandle pp);</pre>
voi d	InvertPoly(PolyHandle poly);
voi d	ErasePoly(PolyHandle poly);

Drawing Regions

voi d	FrameRgn(RgnHandle	rgn);	
voi d	PaintRgn(RgnHandle	rgn);	
voi d	FillCRgn(RgnHandle	rgn, PixPatHandle	pp);
voi d	EraseRgn(RgnHandle	rgn);	

void InvertRgn(RgnHandle rgn); void FillRgn(RgnHandle rgn, const Pattern *pat);

Setting Text Characteristics

voi d	<pre>TextFont(short font);</pre>
voi d	<pre>TextFace(short face);</pre>
voi d	<pre>TextMode(short mode);</pre>
voi d	<pre>TextSize(short size);</pre>
voi d	<pre>SpaceExtra(Fixed extra);</pre>
voi d	<pre>GetFontInfo(FontInfo *info);</pre>

Drawing and Measuring Text

voi d	DrawChar(short ch);
voi d	<pre>DrawString(ConstStr255Param s);</pre>
voi d	<pre>DrawText(const void *textBuf, short firstByte, short byteCount);</pre>
short	CharWidth(short ch);
short	<pre>StringWidth(ConstStr255Param s);</pre>

Copying Images

voi d	CopyBits(const BitMap *srcBits,const BitMap *dstBits,const Rect *srcRect,
	const Rect *dstRect,short mode,RgnHandle maskRgn);
voi d	CopyMask(const BitMap *srcBits,const BitMap *maskBits,const BitMap *dstBits,
	<pre>const Rect *srcRect, const Rect *maskRect, const Rect *dstRect);</pre>
voi d	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

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

Creating and Disposing of Colour Tables

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

Creating and Managing Polygons

Pol yHandl e	OpenPoly(void);
void	ClosePoly(void);
voi d	KillPoly(PolyHandle poly);
voi d	OffsetPoly(PolyHandle poly, short dh, short dv);

Creating and Managing Rectangles

voi d	<pre>SetRect(Rect *r, short left, short top, short right, short bottom);</pre>
voi d	OffsetRect(Rect *r, short dh, short dv);
voi d	<pre>InsetRect(Rect *r, short dh, short dv);</pre>
Boolean	<pre>SectRect(const Rect *src1, const Rect *src2, Rect *dstRect);</pre>
voi d	UnionRect(const Rect *src1, const Rect *src2, Rect *dstRect);
Bool ean	PtInRect(Point pt, const Rect *r);
voi d	Pt2Rect(Point pt1, Point pt2, Rect *dstRect);
voi d	PtToAngle(const Rect *r, Point pt, short *angle);
Bool ean	EqualRect(const Rect *rect1, const Rect *rect2);
Boolean	<pre>EmptyRect(const Rect *r);</pre>

Creating and Managing Regions

RgnHandl e	NewRgn(void);
voi d	OpenRgn(void);
voi d	CloseRgn(RgnHandle dstRgn);
voi d	DisposeRgn(RgnHandle rgn);
voi d	CopyRgn(RgnHandle srcRgn, RgnHandle dstRgn);
voi d	SetEmptyRgn(RgnHandle rgn);
voi d	SetRectRgn(RgnHandle rgn, short left, short top, short right, short bottom);
voi d	RectRgn(RgnHandle rgn, const Rect *r);
voi d	OffsetRgn(RgnHandle_rgn, short_dh, short_dv);

voi d	InsetRgn(RgnHandle rgn, short dh, short dv);
voi d	SectRgn(RgnHandle srcRgnA, RgnHandle srcRgnB, RgnHandle dstRgn);
voi d	UnionRgn(RgnHandle srcRgnA, RgnHandle srcRgnB, RgnHandle dstRgn);
voi d	DiffRgn(RgnHandle srcRgnA, RgnHandle srcRgnB, RgnHandle dstRgn);
voi d	XorRgn(RgnHandle srcRgnA, RgnHandle srcRgnB, RgnHandle dstRgn);
Bool ean	PtInRgn(Point pt, RgnHandle rgn);
Bool ean	RectInRgn(const Rect *r, RgnHandle rgn);
Bool ean	Equal Rgn(RgnHandle rgnA, RgnHandle rgnB);
Bool ean	<pre>EmptyRgn(RgnHandle rgn);</pre>
0SErr	BitMapToRegion(RgnHandle region, const BitMap *bMap);

Scaling and Mapping Points, Rectangles, Polygons, and Regions

voi d	ScalePt(Point *pt,const Rect *srcRect,const Rect *dstRect);
voi d	MapPt(Point *pt,const Rect *srcRect,const Rect *dstRect);
voi d	MapRect(Rect *r, const Rect *srcRect, const Rect *dstRect);
voi d	MapRgn(RgnHandle rgn,const Rect *srcRect,const Rect *dstRect);
voi d	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

voi d	<pre>ScrollRect(const Rect *r, short dh, short dv, RgnHandle updateRgn);</pre>
voi d	SetOrigin(short h, short v);
voi d	PortSize(short width, short height);
voi d	MovePortTo(short leftGlobal,short topGlobal);
voi d	GetClip(RgnHandle rgn);
voi d	SetClip(RgnHandle rgn);
voi d	ClipRect(const Rect *r);

Manipulating Points in Colour Graphics Ports

voi d	GlobalToLocal(Point *pt);
voi d	LocalToGlobal(Point *pt);
voi d	AddPt(Point src, Point *dst);
voi d	<pre>SubPt(Point *src, Point *dst);</pre>
voi d	<pre>SetPt(Point *pt, short h, short v);</pre>
Bool ean	<pre>EqualPt(Point pt1, Point pt2);</pre>
Bool ean	GetPixel(short h, short v);

Demonstration Program

```
// **********
// QuickDraw.c
// *********
                                                           ******
//
// This program:
11
     Opens a window in which the results of various QuickDraw drawing operations are
11 .
11
      displayed. Individual line and text drawing, framing, painting, filling, erasing,
11
      inverting, and copying operations are chosen from a Demonstration pull-down menu.
11
11 •
     Quits when the user chooses Quit from the File menu.
11
// 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.
11
// The program utilises the following resources:
11
11 .
     'WIND' resources for the main window, and a small window used for the CopyBits
//
      demonstration (purgeable) (initially visible).
11
11 .
     An 'MBAR' resource and associated 'MENU' resources (preload, non-purgeable).
11
     Two 'ICON' resources (purgeable) used for the boolean source modes demonstration.
11 .
```

// • Two 'PICT' resources (purgeable) used in the arithmetic source modes demonstration. 11 // • 'STR#' resources (purgeable) containing strings used in the source modes and text 11 drawing demonstrations. 11 11 . 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. 11 11 11 // • A 'SIZE' resource with with is32BitCompatible flag set. 11 // _____includes #include <Appearance. h> #include <Devices.h> #include <Fonts.h> #include <LowMem. h> #include <Menus. h> #include <Processes.h> #include <Sound. h> #include <TextUtils.h> #include <ToolUtils.h> // _____ defines 128 #define rMenubar #define mApple 128 #define iAbout 1 #define mFile 129 #define iQuit 11 #define mDemonstration 131 #define iLine 1 #define iFrameAndPaint
#define iFillEraseInvert 2 3 #define iPolygonRegion 4 #define iText
#define iScrolling 5 6 #define iBooleanSourceModes 7 #define iArithmeticSourceModes 8
#define iHighlighting 9 #define iDrawWithMouse 10 #define iDrawingEnvironment 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 0x7FFFFFFF #define MAXLONG // typedefs struct drawingEnvironment { penLocSi zeModePat; PenState requestedForeColour: **RGBColor** requestedBackColour; **RGBColor** SInt16 textTransferMode; PixPatHandle penPixelPattern; PixPatHandle backPixelPattern; penBitPattern; Pattern backBitPattern; Pattern **}**: typedef struct drawingEnvironment drawingEnvironment; // Boolean gDone; WindowPtr gWindowPtr;

```
Bool ean
         gDrawWithMouseActivated;
SInt16
         gPi xel Depth;
         gIsColourDevice = false;
Bool ean
RGBColor gBlackColour = { 0xFFFF, 0xFFFF, 0xFFFF };
RGBColor gBlackColour = { 0x0000, 0x0000, 0x0000 };
RGBColor gRedColour
                       = \{ 0xAAAA, 0x0000, 0x0000 \};
RGBColorgkedcolour= {0xAAAA, 0x0000, 0x0000 };RGBColorgYellowColour= {0xFFFF, 0xCCCC, 0x0000 };RGBColorgGreenColour= {0x0000, 0x9999, 0x0000 };RGBColorgBlueColour= {0x6666, 0x6666, 0x9999 };
// ..... function prototypes
voi d
                               (void);
       main
voi d
       doInitManagers
                               (void);
voi d
       doEvents
                               (EventRecord *);
                               (SInt16);
voi d
       doDemonstrationMenu
       doLi nes
                               (void);
voi d
       doFrameAndPaint
                               (void);
voi d
       doFillEraseInvert
voi d
                               (void);
       doPol ygonAndRegi on
                               (void);
voi d
       doScrolling
voi d
                               (void):
voi d
       doText
                               (void);
       doBool eanSourceModes
                               (void);
voi d
       doArithmeticSourceModes
voi d
                               (void);
       doHi ghl i ght i ng
voi d
                               (void);
voi d
       doDrawWithMouse
                               (void);
       doDrawi ngEnvi ronment
voi d
                               (void):
       doGetDrawi ngEnvi ronment
                               (drawingEnvironment *);
voi d
       doSetDrawingEnvironment (drawingEnvironment *);
voi d
       doInitDrawingEnvironment (void);
voi d
       doDrawingEnvironmentProof (SInt16);
voi d
       doGetDepthAndDevice (void);
voi d
UInt16 doRandomNumber
                               (UInt16, UInt16);
void main(void)
{
  Handl e
             menubarHdl:
  MenuHandle menuHdl;
  EventRecord eventStructure;
  Boolean
           gotEvent;
  doInitManagers();
  // ..... cause the Appearance-compliant menu bar definition function to be called directly
  RegisterAppearanceClient();
  // ..... see random number generator
  GetDateTime((UInt32 *) (&qd. randSeed));
  // ..... set up menu bar and menus
  if(!(menubarHdl = GetNewMBar(rMenubar)))
    ExitToShell();
  SetMenuBar(menubarHdl);
  DrawMenuBar();
  if(!(menuHdl = GetMenuHandle(mApple)))
    ExitToShell();
  el se
    AppendResMenu(menuHdl, 'DRVR');
  // .....open window
  if(!(gWindowPtr = GetNewCWindow(rWindow, NULL, (WindowPtr)-1)))
    Exi tToShell();
  SetPort(gWindowPtr);
  TextSize(10);
```

```
^{\prime\prime} ..... get pixel depth and whether colour device for certain Appearance functions
```

doGetDepthAndDevice();

```
// .....
                                                        ..... eventLoop
                    .....
 gDone = false;
 while(!gDone)
 {
   gotEvent = WaitNextEvent(everyEvent, &eventStructure, MAXLONG, NULL);
   if(gotEvent)
     doEvents(&eventStructure);
 }
}
void doInitManagers(void)
{
  MaxAppl Zone();
 MoreMasters();
 InitGraf(&qd.thePort);
 InitFonts();
 InitWindows();
 InitMenus();
 TEI nit();
 InitDialogs(NULL);
 InitCursor();
 FlushEvents(everyEvent, 0);
}
void doEvents(EventRecord *eventStrucPtr)
{
 SInt8
          charCode;
 SInt 32
          menuChoice;
          menuID, menuItem;
 SInt16
 SInt16
         partCode;
 WindowPtr windowPtr;
 Str255
         itemName:
 SInt16
          daDriverRefNum;
  switch(eventStrucPtr->what)
 {
   case kevDown:
   case autoKey:
     charCode = eventStrucPtr->message & charCodeMask;
     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, &windowPtr))
     {
       switch(partCode)
       {
        case inMenuBar:
          menuChoice = MenuSelect(eventStrucPtr->where);
          menuID = HiWord(menuChoice);
          menuItem = LoWord(menuChoice);
          if(menuID == 0)
            return;
          switch(menuID)
          {
            case mApple:
```

```
if(menuItem == iAbout)
                 SysBeep(10);
               else
               {
                 GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
                 daDriverRefNum = OpenDeskAcc(itemName);
               break;
             case mFile:
               if(menuItem == iQuit)
                 gDone = true;
               break:
             case mDemonstration:
               doDemonstrationMenu(menuItem);
               break;
           HiliteMenu(0);
           break;
         case inDrag:
           DragWindow(windowPtr, eventStrucPtr->where, &qd. screenBits. bounds);
           break;
         case inContent:
           if(windowPtr != FrontWindow())
             Sel ectWindow(windowPtr);
           else
             if(gDrawWithMouseActivated)
               doDrawWithMouse();
           break;
       }
     }
     break;
    case updateEvt:
     windowPtr = (WindowPtr) eventStrucPtr->message;
      BeginUpdate(windowPtr);
     EndUpdate(windowPtr);
      break;
  }
}
  11
voi d
     doDemonstrationMenu(SInt16 menuItem)
{
  gDrawWithMouseActivated = false;
  switch(menuItem)
  {
    case iLine:
      doLines();
     break:
    case iFrameAndPaint:
      doFrameAndPaint();
     break;
    case iFillEraseInvert:
      doFillEraseInvert();
     break;
    case iPolygonRegion:
     doPol ygonAndRegi on();
     break;
    case iText:
      doText();
     break;
    case iScrolling:
      doScrolling();
      break;
```

```
case iBooleanSourceModes:
      doBooleanSourceModes();
      break:
    case iArithmeticSourceModes:
      doArithmeticSourceModes();
      break:
    case iHighlighting:
      doHighlighting();
      break;
    case iDrawWithMouse:
      SetWTitle(gWindowPtr, "\pDrawing with the mouse");
      RGBBackColor(&gWhiteColour);
      FillRect(&gWindowPtr->portRect, &qd. white);
      gDrawWithMouseActivated = true;
      break:
    case iDrawingEnvironment:
      doDrawi ngEnvi ronment();
      break:
 }
}
  void doLines(void)
{
  RgnHandl e
                ol dCl i pRgn;
                newClipRect;
  Rect
  SInt16
                a, b, c, top, left, bottom, right;
  RGBCol or
                theColour:
  UInt32
               finalTicks;
                systemPattern;
  Pattern
  PixPatHandle pixpatHdl;
  PenNormal();
  RGBBackColor(&gBlueColour);
  FillRect(&gWindowPtr->portRect, &qd. white);
  newClipRect = gWindowPtr->portRect;
  InsetRect(&newClipRect, 10, 10);
  oldClipRgn = NewRgn();
  GetClip(oldClipRgn);
  ClipRect(&newClipRect);
             ..... lines drawn with foreground colour and black pen pattern
  // .....
  SetWTitle(gWindowPtr, "\pDrawing lines with colours");
  RGBBackColor(&gWhiteColour);
  FillRect(&gWindowPtr->portRect, &qd. white);
  for(a=1; a<60; a++)
  {
   b = doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.left);
c = doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.left);
                   = doRandomNumber(0, 65535);
    theColour.red
    theColour.green = doRandomNumber(0, 65535);
    theColour.blue = doRandomNumber(0, 65535);
    RGBForeColor(&theColour);
    PenSize(a * 2, 1);
    MoveTo(b, gWindowPtr->portRect.top);
    LineTo(c, gWindowPtr->portRect.bottom);
    Delay(2, &finalTicks);
  }
  11 ..
                             ..... lines drawn with system-supplied bit patterns
  SetWTitle(gWindowPtr, "\pClick mouse for more lines");
  while(!Button()) ;
```

```
SetWTitle(gWindowPtr, "\pDrawing lines with system-supplied bit patterns");
FillRect(&gWindowPtr->portRect, &qd. white);
c = 0:
for(a=1; a<39; a++)
{
  b = doRandomNumber(0, gWindowPtr->portRect.bottom - gWindowPtr->portRect.top);
c = doRandomNumber(0, gWindowPtr->portRect.bottom - gWindowPtr->portRect.top);
  theColour.red
                  = doRandomNumber(0, 32767);
  theColour.green = doRandomNumber(0, 32767);
  theColour.blue = doRandomNumber(0, 32767);
  RGBForeCol or (&theCol our);
  GetIndPattern(&systemPattern, sysPatListID, a);
  PenPat(&systemPattern);
  PenSize(1, a * 2);
  MoveTo(gWindowPtr->portRect.left,b);
  LineTo(gWindowPtr->portRect.right,c);
  Del ay(5, &final Ticks);
}
// ..... lines drawn with a pixel pattern
SetWTitle(gWindowPtr, "\pClick mouse for more lines");
while(!Button()) ;
SetWTitle(gWindowPtr, "\pDrawing lines with a pixel pattern");
FillRect(&gWindowPtr->portRect, &qd. white);
if(!(pixpatHdl = GetPixPat(rPixelPattern1)))
  ExitToShell();
PenPixPat(pixpatHdl);
for(a=1; a<60; a++)
{
  b = doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.left);
c = doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.left);
  PenSize(a * 2, 1);
  MoveTo(b, gWindowPtr->portRect.top);
  LineTo(c, gWindowPtr->portRect.bottom);
  Del ay(5, &final Ticks);
}
Di sposePi xPat(pi xpatHdl);
SetClip(oldClipRgn);
Di sposeRgn(ol dCl i pRgn);
                                                .....lines drawn with pattern mode patXor
11
SetWTitle(gWindowPtr, "\pClick mouse for more lines");
while(!Button())
SetWTitle(gWindowPtr, "\pDrawing lines using pattern mode patXor");
RGBBackCol or (&gRedCol our);
FillRect(&gWindowPtr->portRect, &qd. white);
PenSi ze(1, 1);
PenPat(&qd. bl ack);
PenMode(patXor);
        = gWindowPtr->portRect.left + 10;
left
        = gWindowPtr->portRect.top + 10;
top
right = gWindowPtr->portRect.right - 10;
bottom = gWindowPtr->portRect.bottom - 10;
for(a=left, b=right; a<right+1; a++, b--)</pre>
{
  MoveTo(a, top);
  LineTo(b, bottom);
```

```
}
  for(a=bottom, b=top; b<bottom+1; a--, b++)</pre>
  {
    MoveTo(left, a);
   LineTo(right, b);
  }
}
   11
void doFrameAndPaint(void)
  SInt16
               a:
               theRect:
  Rect
  UInt32
               finalTicks;
  Pattern
               systemPattern;
 PixPatHandle pixpatHdl;
  PenNormal();
  PenSi ze(30, 20);
  for(a=0; a<3; a++)
  {
   RGBBackColor(&gWhiteColour);
   FillRect(&gWindowPtr->portRect, &qd. white);
    // .....
                                                                ..... preparation
   if(a == 0)
    {
     SetWTitle(gWindowPtr, "\pFraming and painting with a colour");
     RGBForeCol or (&gRedCol our);
                                                          // set foreground colour to red
   }
   else if (a == 1)
    {
     SetWTitle(gWindowPtr, "\pFraming and painting with a bit pattern");
     RGBForeCol or (&gBl ueCol our);
                                                       // set foreground colour to blue
     RGBBackColor(&gYellowColour);
                                                      // set foreground colour to yellow
                                                             // get bit pattern for pen
// set pen bit pattern
     GetIndPattern(&systemPattern, sysPatListID, 16);
     PenPat(&systemPattern);
   }
   else if (a == 2)
    {
     SetWTitle(gWindowPtr, "\pFraming and painting with a pixel pattern");
                                                         // get pixel pattern for pen
     if(!(pixpatHdl = GetPixPat(rPixelPattern1)))
       ExitToShell();
     PenPi xPat(pi xpatHdl);
                                                               // set pen pixel pattern
   }
    // ..... framing and painting
   SetRect(&theRect, 30, 32, 151, 191);
    FrameRect(&theRect);
                                                                           // FrameRect
   MoveTo(30, 29);
    DrawString("\pFrameRect");
   Del ay(30, &fi nal Ti cks);
   OffsetRect(&theRect, 140, 0);
   FrameRoundRect(&theRect, 30, 50);
                                                                      // FrameRoundRect
    MoveTo(170, 29);
    DrawString("\pFrameRoundRect");
   Delay(30, &final Ticks);
   OffsetRect(&theRect, 140, 0);
    FrameOval(&theRect);
                                                                           // FrameOval
    MoveTo(310, 29);
    DrawString("\pFrameOval");
   Del ay(30, &fi nal Ti cks);
   OffsetRect(&theRect, 140, 0);
    FrameArc(&theRect, 330, 300);
                                                                            // FrameArc
    MoveTo(450, 29);
```

```
DrawString("\pFrameArc");
    Del ay(30, &final Ticks);
    OffsetRect(&theRect, -420, 186);
    PaintRect(&theRect);
                                                                             // PaintRect
    MoveTo(30, 214);
    DrawString("\pPaintRect");
Delay(30, &finalTicks);
    OffsetRect(&theRect, 140, 0);
    PaintRoundRect(&theRect, 30, 50);
                                                                        // PaintRoundRect
    MoveTo(170, 214);
    DrawString("\pPaintRoundRect");
    Del ay(30, &final Ticks);
    OffsetRect(&theRect, 140, 0);
    PaintOval(&theRect);
                                                                             // PaintOval
    MoveTo(310.214):
    DrawString("\pPaint0val");
Del ay(30, &final Ticks);
    OffsetRect(&theRect, 140, 0);
    PaintArc(&theRect, 330, 300);
                                                                              // PaintArc
    MoveTo(450, 214);
   DrawString("\pPaintArc");
Delay(30, &finalTicks);
   if(a < 2)
    {
      SetWTitle(gWindowPtr, "\pClick mouse for more");
      while(!Button()) ;
   }
  }
  Di sposePi xPat(pi xpatHdl);
  void doFillEraseInvert(void)
  SInt16
               a:
               fillPat, backPat;
  Pattern
  PixPatHandle fillPixpatHdl, backPixpatHdl;
  Rect
               theRect;
  UInt32
               finalTicks;
  PenNormal():
  PenSize(30, 20);
  for (a=0; a<4; a++)
  {
   if(a < 3)
    {
      RGBBackColor(&gWhiteColour);
      FillRect(&gWindowPtr->portRect, &qd. white);
    }
    // .....
                                          ..... preparation
    if(a == 0)
    {
      SetWTitle(gWindowPtr, "\pFilling and erasing with colours");
      RGBForeCol or ( &gBl ueCol our) ;
                                                        // set blue colour for foreground
      RGBBackColor(&gRedColour);
                                                        // set red colour for background
      GetIndPattern(&fillPat, sysPatListID, 1); // get black bit pattern for fill functions
      BackPat(&qd.white);
                                                 // set white bit pattern for background
    }
    else if(a == 1)
    {
      SetWTitle(gWindowPtr, "\pFilling and erasing with bit patterns");
      RGBForeCol or (&gBl ueCol our);
                                                        // 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(gWindowPtr, "\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(gWindowPtr, "\pInverting");
  BackPat(&qd. white);
  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");
Delay(30, &finalTicks);
OffsetRect(&theRect, 140, 0);
MoveTo(170, 29);
if(a < 2)
{
                                                                       // FillRoundRect
  FillRoundRect(&theRect, 30, 50, &fillPat);
  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");
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");
3
else if (a == 3)
```

```
{
  InvertOval(&theRect);
                                                                              // InvertOval
  DrawString("\pInvert0val");
}
Del ay(30, &final Ticks);
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");
Del ay(30, &fi nal Ti cks);
OffsetRect(&theRect, -420, 186);
MoveTo(30, 214);
if(a < 3)
{
                                                                               // EraseRect
  EraseRect(&theRect);
  DrawString("\pEraseRect");
}
else
{
  InvertRect(&theRect);
                                                                              // InvertRect
  DrawString("\pInvertRect");
}
Del ay(30, &final Ticks);
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");
Del ay(30, &final Ticks);
OffsetRect(&theRect, 140, 0);
MoveTo(310, 214);
if(a < 3)
{
  EraseOval(&theRect);
                                                                               // EraseOval
  DrawString("\pEraseOval");
}
else
{
                                                                              // InvertOval
  InvertOval(&theRect);
  DrawString("\pInvert0val");
}
Del ay(30, &final Ticks);
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");
   Delay(30, &finalTicks);
   if(a < 3)
   {
     SetWTitle(gWindowPtr, "\pClick mouse for more");
     while(!Button()) ;
   }
 }
  Di sposePi xPat(fillPi xpatHdl);
  Di sposePi xPat(backPi xpatHdl);
}
void doPolygonAndRegion(void)
{
  Pattern
               backPat;
  PixPatHandle fillPixpatHdl;
  PolyHandl e
               pol ygonHdl ;
               regionHdl;
  RgnHandl e
  UInt 32
               finalTicks;
               theRect;
  Rect
  SetWTitle(gWindowPtr, "\pFraming, painting, filling, and erasing polygons and regions");
  RGBBackColor(&gWhiteColour);
  FillRect(&gWindowPtr->portRect, &qd. white);
  // .....
                                                                ..... preparation
                                                      // get bit pattern for background
// set bit pattern for background
  GetIndPattern(&backPat, sysPatListID, 17);
  BackPat(&backPat);
  if(!(fillPixpatHdl = GetPixPat(rPixelPattern2)))// get pixel pattern for fill functions
    ExitToShell();
  RGBForeColor(&gRedColour);
                                                        // set red colour for foreground
                                                     // set yellow colour for background
  RGBBackColor(&gYellowColour);
  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
  FramePoly(polygonHdl);
  MoveTo(30, 29);
 DrawString("\pFramePoly (colour)");
Delay(30, &finalTicks);
  OffsetPoly(polygonHdl, 140, 0);
  PaintPoly(polygonHdl);
                                                                            // PaintPoly
  MoveTo(170, 29);
  DrawString("\pPaintPoly (colour)");
  Del ay(30, &final Ticks);
```

```
OffsetPoly(polygonHdl, 140, 0);
  FillCPoly(polygonHdl, fillPixpatHdl);
                                                                                // FillCPoly
  MoveTo(310, 29);
  DrawString("\pFillCPoly (pixel pattern)");
  Del ay(30, &fi nal Ti cks);
  OffsetPoly(polygonHdl, 140, 0);
  ErasePoly(polygonHdl);
                                                                                // ErasePoly
  MoveTo(450, 29);
  DrawString("\pErasePoly (bit pattern)");
  Del ay(30, &fi nal Ti cks);
  FrameRgn(regionHdl);
                                                                                 // FrameRgn
  MoveTo(30, 214);
  DrawString("\pFrameRgn (colour)");
Delay(30, &finalTicks);
  OffsetRgn(regionHdl, 140, 0);
  PaintRgn(regionHdl);
                                                                                 // PaintRgn
  MoveTo(170, 214);
  DrawString("\pPaintRgn (colour)");
Del ay(30, &final Ticks);
  OffsetRgn(regionHdl, 140, 0);
  FillCRgn(regionHdl, fillPixpatHdl);
                                                                                 // FillCRgn
  MoveTo(310, 214);
  DrawString("\pFillCRgn (pixel pattern)");
  Del ay (30, &fi nal Ti cks);
  OffsetRgn(regionHdl, 140, 0);
  EraseRgn(regionHdl);
                                                                                 // EraseRgn
  MoveTo(450, 214):
  DrawString("\pEraseRgn (bit pattern)");
Delay(30, &finalTicks);
  KillPoly(polygonHdl);
  Di sposeRgn(regi onHdl);
  Di sposePi xPat(fillPi xpatHdl);
  BackPat(&qd. white);
void doText(void)
  SInt16
            windowCentre, a, fontNum, stringWidth;
            theRect:
  Rect
  Str255
            textString;
  UInt32
            final Ticks;
  RGBBackColor(&gWhiteColour);
  FillRect(&gWindowPtr->portRect, &qd. white);
  SetWTitle(gWindowPtr, "\pDrawing text with default source mode (src0r)");
  windowCentre = (gWindowPtr->portRect.right - gWindowPtr->portRect.left) / 2;
  SetRect(&theRect, windowCentre, gWindowPtr->portRect.top, gWindowPtr->portRect.right,
          gWindowPtr->portRect.bottom);
  RGBBackCol or (&gBl ueCol our);
  FillRect(&theRect, &qd. white);
  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);
      RGBForeCol or (&gRedCol our);
    }
    else if(a == 2)
      TextFace(bold);
    else if(a == 3)
```

}

{

```
{
     GetFNum("\pTimes", &fontNum);
     TextFont(fontNum);
     TextFace(italic);
     RGBForeCol or (&gYellowCol our);
   }
   else if (a == 4)
     TextFace(underline);
   else if (a == 5)
   {
     GetFNum("\pHelvetica", &fontNum);
     TextFont(fontNum);
     TextFace(outline);
     RGBForeCol or (&gGreenCol our);
   3
   else if (a == 6)
     TextFace(shadow);
   else if (a == 7)
   {
     GetFNum("\pChicago", &fontNum);
     TextFont(fontNum);
     TextFace(condense);
     RGBForeColor(&gBlackColour);
   }
   else if(a == 8)
   {
     TextFace(extend);
   }
   // .....
                   if(a < 7)
     TextSize(a * 2 + 15);
   el se
     TextSize(12);
   \prime\prime means 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);
   Delay(30, &finalTicks);
 }
                       ..... reset to Geneva 10pt normal
  11
  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,gWindowPtr->portRect.left + 5,gWindowPtr->portRect.bottom - 55,
         gWindowPtr->portRect.left + 138, gWindowPtr->portRect.bottom - 5);
 EraseRect(&theRect);
 InsetRect(&theRect, 5, 5);
 GetIndString(textString, rFontsStringList, 9);
 RGBForeColor(&gWhiteColour);
 TETextBox(&textString[1], textString[0], &theRect, teFlushLeft);
void doScrolling(void)
 PixPatHandle pixpat1Hdl, pixpat2Hdl;
 Rect
              theRect:
  RgnHandl e
              oldClipHdl, regionAHdl, regionBHdl, regionCHdl, scrollRegionHdl;
 SInt16
              a:
 SetWTitle(gWindowPtr, "\pScrolling pixels");
  RGBBackColor(&gWhiteColour);
```

}

{

```
if(!(pixpat1Hdl = GetPixPat(rPixelPattern1)))
    ExitToShell();
  PenPi xPat(pi xpat1Hdl);
  PenSi ze(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);
  Uni onRgn(regi onAHdl, regi onBHdl, regi onCHdl);
  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);
    theRect.top ++;
  }
  SetRect(&theRect, 80, 30, 571, 371);
  BackPixPat(pixpat1Hdl);
  for(a=0; a<371; a++)
  {
    ScrollRect(&theRect, 0, -1, scrollRegionHdl);
    theRect.bottom --;
  }
  SetClip(oldClipHdl);
  Di sposePi xPat(pi xpat1Hdl);
  Di sposePi xPat(pi xpat2Hdl);
  Di sposeRgn(ol dCl i pHdl);
  Di sposeRgn(regi onAHdl);
  Di sposeRgn(regi onBHdl);
  Di sposeRgn(regi onCHdl);
  Di sposeRgn(scrollRegionHdl);
  voi d
    doBool eanSourceModes(void)
 Rect
          theRect:
  Handle destIconHdl, sourceIconHdl;
  SInt16 a, b;
 UInt32 finalTicks;
  BitMap
         sourceI conMap;
         sourceString;
  Str255
  SetWTitle(gWindowPtr, "\pBoolean source modes");
  RGBForeCol or (&gBl ackCol our);
  RGBBackCol or (&gGreenCol our);
  FillRect(&gWindowPtr->portRect, &qd. white);
  SetRect(&theRect, gWindowPtr->portRect.left, gWindowPtr->portRect.top,
          gWindowPtr->portRect.right, (gWindowPtr->portRect.bottom ·
```

FillRect(&(gWindowPtr->portRect), &qd. white);

3

ł

```
gWindowPtr->portRect.top) / 2);
  RGBBackColor(&gWhiteColour);
  FillRect(&theRect, &qd. white);
  destIconHdl = GetIcon(rDestinationIcon);
  sourceIconHdl = GetIcon(rSourceIcon);
  for(a=0; a<2; a++)
  {
    if(a == 1)
    {
      RGBForeColor(&gYellowColour);
      RGBBackCol or (&gRedCol our);
    }
    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);
      RGBBackCol or (&gRedCol our);
    }
    for(b=0; b<8; b++)
    {
      SetRect(&theRect, b * 69 + 28, a * 191 + 121, b * 69 + 92, a * 191 + 185);
      PlotIcon(&theRect, destIconHdl);
    }
  }
  RGBForeColor(&gBlackColour);
  RGBBackColor(&gWhiteColour);
  HLock(sourceIconHdl);
  sourceIconMap.baseAddr = *sourceIconHdl;
  sourceIconMap.rowBytes = 4;
  SetRect (&sourceI conMap. bounds, 0, 0, 32, 32);
  for(a=0; a<2; a++)
  {
    if(a == 1)
    {
      RGBForeColor(&gYellowColour);
      RGBBackCol or (&gRedCol our);
    }
    for(b=0; b<8; b++)
    {
       \begin{array}{l} Del ay(30,\,\&fi\,nal\,Ti\,cks)\,;\\ SetRect(\&theRect,\,b\ *\ 69\ +\ 28,\,a\ *\ 191\ +\ 121,\,b\ *\ 69\ +\ 92,\,a\ *\ 191\ +\ 185)\,; \end{array} 
      CopyBits(&sourceIconMap, &qd. thePort->portBits, &sourceIconMap. bounds, &theRect,
                b, NULL);
      GetIndString(sourceString, rBooleanStringList, b + 1);
      MoveTo(b * \overline{69} + 28, a * \overline{191} + 118);
      DrawString(sourceString);
    }
  }
  HUnlock(sourceIconHdl);
}
```

```
void doArithmeticSourceModes(void)
{
  PicHandle sourceHdl, destinationHdl;
           sourceRect, destRect;
  Rect
           a, b, arithmeticMode = 32;
  SInt16
  Str255
           modeString;
  UInt32
           finalTicks;
  SetWTitle(gWindowPtr, "\pCopyBits with arithmetic source modes");
  RGBForeCol or (&gBl ackCol our);
  RGBBackColor(&gWhiteColour);
  FillRect(&(gWindowPtr->portRect), &qd. white);
  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);
  for(a=44; a<403; a+=179)
  {
    for (b=21; b<274; b+=126)
    {
      if(a == 44 \&\& b == 21)
      continue;
     Del ay(60, &final Ticks);
      GetIndString(modeString, rArithmeticStringList, arithmeticMode - 31);
      MoveTo(a, b - 2);
      DrawString(modeString);
      SetRect(&destRect, a, b, a+157, b+112);
      CopyBits(&((GrafPtr) gWindowPtr)->portBits,
              &((GrafPtr) gWindowPtr)->portBits,
              &sourceRect, &destRect,
              arithmeticMode + ditherCopy, NULL);
      arithmeticMode ++;
   }
  }
  ReleaseResource((Handle) sourceHdl);
  ReleaseResource((Handle) destinationHdl);
}
void doHighlighting(void)
{
  RGBColor oldHighlightColour;
  SInt16
           a:
           theRect;
  Rect
  UInt8
           hiliteVal;
  UInt32
           finalTicks;
```

```
SetWTitle(gWindowPtr, "\pHighlighting");
  RGBForeColor(&gBlackColour);
  RGBBackColor(&gWhiteColour);
  FillRect(&(gWindowPtr->portRect), &qd. white);
  LMGetHiliteRGB(&oldHighlightColour);
  for(a=0; a<3; a++)
  {
   MoveTo(50, a * 100 + 60);
DrawString("\pClearing the highlight bit and calling InvertRect.");
   Del ay(60, &fi nal Ti cks);
   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);
   MoveTo(50, a * 100 + 75);
   Delay(60, &finalTicks);
   DrawString("\pClick mouse to unhighlight. ");
   DrawString("\p(Note: The call to InvertRect reset the highlight bit ...");
   while(!Button()) ;
   MoveTo(45, a * 100 + 90);
   DrawString("\p... so we clear the highlight bit again before calling InvertRect.)");
   Delay(60, &finalTicks);
   LMSetHiliteMode(hiliteVal);
   InvertRect(&theRect);
  }
  HiliteColor(&oldHighlightColour);
  Del ay(60, &final Ticks);
 MoveTo(50, 350);
 DrawString("\pOriginal highlight colour has been reset.");
}
void doDrawWithMouse(void)
{
  PixPatHandle pixpatHdl;
  Point
               initial Mouse, previous Mouse, current Mouse;
  Rect
               drawRect:
  UInt16
               randomNumber:
  RGBCol or
               theColour;
  RGBBackColor(&gWhiteColour);
  FillRect(&gWindowPtr->portRect, &qd. white);
  if(!(pixpatHdl = GetPixPat(rPixelPattern3)))
   ExitToShell();
  PenPi xPat(pi xpatHdl);
  PenSize(1, 1);
  PenMode(patXor);
  GetMouse(&initialMouse);
  drawRect.left = drawRect.right = initialMouse.h;
  drawRect.top = drawRect.bottom = initialMouse.v;
  GetMouse(&previousMouse);
  while(StillDown())
```

{ GetMouse(¤tMouse);

```
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)</pre>
        drawRect.left = currentMouse.h;
      if(currentMouse.v <= initialMouse.v)</pre>
        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(&qd. black);
  BackPixPat(pixpatHdl);
  randomNumber = doRandomNumber(0, 4);
  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(&qd. white);
void doDrawingEnvironment(void)
{
  drawingEnvironment environment;
                      theRect:
  Rect
  SInt16
                      a:
  UInt32
                      finalTicks;
  RGBBackCol or (&gBl ueCol our);
  FillRect(&gWindowPtr->portRect, &qd. white);
  SetWTitle(gWindowPtr, "\pSaving and restoring the drawing environment");
  doInitDrawingEnvironment();
  doDrawingEnvironmentProof(0);
  Delay(120, &finalTicks);
  doGetDrawingEnvironment(&environment);
  theRect = gWindowPtr->portRect;
  theRect.right -= 300;
  SetThemeBackground(kThemeListViewBackgroundBrush, gPixelDepth, gIsColourDevice);
  EraseRect(&theRect);
```

```
theRect.left += 150:
```

}

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

SetThemePen(kThemeListViewSeparatorBrush, gPixelDepth, gIsColourDevice);

```
theRect.left -= 150:
 for(a=theRect.top;a<=theRect.bottom;a+=18)</pre>
 {
   MoveTo(theRect.left, a);
   LineTo(theRect.right - 1, a);
 }
 Delay(120, &finalTicks);
 doDrawingEnvironmentProof(1);
 Delay(120, &finalTicks);
 doSetDrawingEnvironment(&environment);
 doDrawingEnvironmentProof(2);
}
                    // **************
void doGetDrawingEnvironment(drawingEnvironment *environment)
 GrafPtr currentPort;
 GetPort(&currentPort);
 GetPenState(&environment->penLocSizeModePat);
 GetForeColor(&environment->requestedForeColour);
 GetBackColor(&environment->requestedBackColour);
 environment->textTransferMode = currentPort->txMode;
 environment->penPixelPattern = NULL;
 environment->backPixelPattern = NULL;
 if((**((CGrafPtr) currentPort)->pnPixPat).patType != 0)
   environment->penPixelPattern = ((CGrafPtr) currentPort)->pnPixPat;
 if((**((CGrafPtr) currentPort)->bkPixPat).patType != 0)
   environment->backPixelPattern = ((CGrafPtr) currentPort)->bkPixPat;
 else
   environment->backBitPattern =
                          *(PatPtr) (*(**((CGrafPtr) currentPort)->bkPixPat).patData);
}
  11
void doSetDrawingEnvironment(drawingEnvironment *environment)
{
 GrafPtr currentPort;
 GetPort(&currentPort);
 SetPenState(&environment->penLocSizeModePat);
 RGBForeColor(&environment->requestedForeColour);
 RGBBackCol or(&environment->requestedBackCol our);
 TextMode(environment->textTransferMode);
 if(environment->penPixelPattern)
   PenPi xPat (envi ronment - >penPi xel Pattern);
 if(environment->backPixelPattern)
   BackPixPat(environment->backPixelPattern);
 else
   BackPat(&environment->backBitPattern);
}
void doInitDrawingEnvironment()
 PenNormal();
 RGBForeColor(&gBlackColour);
 RGBBackColor(&gWhiteColour);
```

```
TextMode(src0r);
 BackPat(&qd. white);
  void doDrawingEnvironmentProof(SInt16 phase)
ł
        theRect;
 Rect
 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 saved drawing environment");
 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);
}
void doGetDepthAndDevice(void)
{
 GDHandle deviceHdl;
 deviceHdl = LMGetMainDevice();
 gPixelDepth = (*(*deviceHdl)->gdPMap)->pixelSize;
 if(BitTst(&(*deviceHdl)->gdFlags, gdDevType))
   gIsColourDevice = true;
3
UInt16 doRandomNumber(UInt16 minimum, UInt16 maximum)
{
 UInt16 randomNumber;
 SInt32 range, t;
 randomNumber = Random();
 range = maximum - minimum;
 t = (randomNumber * range) / 65535;
 return(t + minimum);
}
// **********
```

Demonstration Program 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.

#define

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.

#typedef

A variable of type drawingEnvironment will be used in the function doDrawingEnvironment to save and restore the drawing environment on either side of calls to the Appearance Manager functions SetThemeBackground and SetThemePen. (These functions change the pen and background colours or patterns.)

Global Variables

gDone will be set to true when the user selects Quit from the File menu, thus causing program termination. gWindowPtr will be assigned the pointer to the main window's colour graphics port. gDrawWithMouseActivated will be set to true when the Draw With Mouse item is chosen from the Demonstration menu, and to false when other items are chosen.

gPixelDepth will be assigned the pixel depth of the main device. gIsColourDevice will be assigned true if the graphics device is a colour device and false if it is a monochrome device. The values in these two variables are required by the Appearance Manager functions SetThemeBackground and SetThemePen.

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 application-defined functions in the demonstration. The call to GetDateTime 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. Calling GetDateTime is one way to seed the generator. 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 application-defined 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 application-defined 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 colour 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 two 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, in effect assigning it to the colour graphics port's clipRgn field.

Lines Drawn With Foreground Colour And Black Pen Pattern

After the window title is set, FillRect is called with the white pattern while 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 obtained by the GetPixPat call.

At this point, the clipping region saved at the start of the function is restored, and all of the memory obtained 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, ${\tt DisposePixPat}$ is called to free the memory obtained 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 colour 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 colour 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 obtained 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 systemsupplied 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 obtained by OpenPoly, DisposeRgn is called to free all the memory obtained by NewRgn, DisposePixPat is

called to free all the memory obtained 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 obtained 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 record. That way, the CopyBits routine can be used to copy the bitmap into the colour 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 record, 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 obtained by the GetPicture calls.

doHighlighting

doHighlighting demonstrates highlighting, first with the colour set by the user in the Colour pane of the Appearance control panel, 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(&qd.gray).)

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 position.

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.

doDrawingEnvironment

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

The call to doInitDrawingEnvironment is included in this function for demonstration purposes only. Ordinarily, it would be called (if required) at other points in an application.

The call to doGetDrawingEnvironment saves the drawing environment prior to the calls to the Appearance Manager functions SetThemeBackground and SetThemePen, which, depending on the current theme, will change either the colour or the pattern in the relevant fields of the colour graphics port.

The call to doSetDrawingEnvironment restores the saved drawing environment.

The intervening code simply draws an Appearance-compliant list view in the left half of the window.

The calls to doDrawingEnvironmentProof 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 doInitDrawingEnvironment, after the calls to the Appearance Manager functions, and after the call to doSetDrawingEnvironment.

doGetDrawingEnvironment

doGetDrawingEnvironment saves the current drawing environment to a variable of type drawingEnvironment.

The call to GetPort assigns a pointer to the current port to the variable currentPort.

GetPenState saves the current pen location, size, pattern mode, and pattern to the penLocSizeModePat field of the drawingEnvironment structure. GetForeColor and GetBackColor save the current foreground and background colours. The next line saves the text source mode.

At the next two lines, the fields of the drawingEnvironment structure relating to the pen and background pixel patterns are initialised to NULL.

The patType field of the PixPat structure whose handle resides in the pnPixPat field of the colour graphics port will contain 0 if the pattern is a bit pattern and 1 if it is a pixel pattern. If it is a pixel pattern, the handle to the PixPat structure is saved.

If the patType field of the PixPat structure whose handle resides in the bkPixPat field of the colour graphics port indicates that the pattern is a pixel pattern, the handle to the PixPat structure is saved. If the pattern is a bit pattern, a pointer to the pattern data is saved.

doSetDrawingEnvironment

doSetDrawingEnvironment restores the saved drawing environment.

The first four calls restore the saved pen location, size, pattern mode, and bit pattern, the requested foreground and background colours, and the text source mode.

If the penPixelPattern field of the drawingEnvironment structure does not contain NULL, the pen pixel pattern is restored, overriding the effect of the pattern aspect of the previous call to SetPenState.

If the backPixelPattern field of the drawingEnvironment structure does not contain NULL, the background pixel pattern is restored, otherwise the background bit pattern is restored.

doInitDrawingEnvironment

doInitDrawingEnvironment shows how you might initialise the drawing environment. It sets the pen size to (1, 1), the pen pattern mode to patCopy, and the pen pattern to black. It also sets the foreground and background colours to black and white respectively, the text source mode to srcOr and the background pattern to white.

doDrawingEnvironmentProof

doDrawingEnvironmentProof is called by doDrawingEnvironment 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 doInitDrawingEnvironment, after the calls to the Appearance Manager functions, and after the call to doSetDrawingEnvironment. Note that the colour or pattern in which the second pair of rectangles is drawn will depend on the current theme.

doGetDepthAndDevice and doRandomNumber

doGetDepthAndDevice and doRandomNumber are incidental to the demonstration.

doGetDepthAndDevice gets the pixel depth of the main device, and whether the device is a colour device or a monochrome device, for the Appearance Manager functions SetThemeBackground and SetThemePen. doRandomNumber returns a random number between the specified minimum value and the specified maximum value minus one.