## 1 <br> 2

## 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; l| Vertical coordinate.
    short h; ll 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 I eft;
    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; |l Size in bytes.
    Rect rgnBBox; | Enclosingrectangle.
};
typedef struct Region Region;
typedef Region *RgnPtr;
typedef RgnPtr *RgnHandle;
```

Thergnsize field contains the size, in bytes, of the region. The maximum size is 64 KB .
The $\operatorname{rgnBBox}$ 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 Get pen 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

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

| Get PenState | Returns, in a PenState structure, the graphics pen's current location, size, transfer mode, and pattern. |
| :--- | :--- |
| Set PenState | Using information supplied by a PenSt at e structure, sets the graphics pen's location, size, transfer mode, <br> and pattern. |
| PenNormal | Resets the pen size, transfer mode, and pattern to the state initialised when the colour graphics port was <br> opened. |

## Foreground and Background Colour

## Foreground Colour

The function RGBForecolor is used to assign a requested foreground colour to the rgbFgColor 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 f Color field. The colour represented by the pixel value in the f g col 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 $b \mathrm{kcol}$ or field. The colour represented by the pixel value in the bkcol or 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 scroll Rect 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 fg Color 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'spnpixpat 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 thepnPixpat 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 variableblack as follows:
PenPat( \&qd.black);

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 whit e 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 |
| pat Apply foreground colour. | Apply background colour. |  |
| pat Or | Apply foreground colour. | Leave alone. |
| pat Xor | Invert. | Leave alone. |
| pat Bic | Apply background colour. | Leave alone. |
| not Pat Copy | Apply background colour. | Apply foreground colour. |
| notPat Or | Leave alone. | Force black. |
| not Pat Xor | Leave alone. | Invert. |


| notPat Bic | 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 $t \times M o d e$ field of the colour graphics port using the function Text Mode, 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 $\mathrm{t} \times \mathrm{Mode}$ 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 |
| :---: | :---: | :---: |
| blend | 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. |
| addPin | 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 sic Bic mode. |
| addOver | 34 | 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. <br>  |
| subPin | 35 | 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 sricor mode. |
| transparent | 36 | Replace the source and destination pixel with the source pixel if the source pixel is not equal to the background colour. |
| addMax | 37 | 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. |
| subover | 38 | 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. |
| adMin | 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 srcor mode. |

## Drawing Lines and Framed Shapes

## Functions for Drawing Lines

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

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

| Moveto | Moves the graphics pen location to the specified location, in local coordinates. |
| :--- | :--- |
| Move | Moves the graphics pen a specified distance from its current location. |
| Lineto | Draws a line from the current pen location to the specified location, in local coordinates. |
| Line | Draws a line a specified distance from the graphics pen's current location. |

Fig 5 shows a line drawn 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.

Functions for drawing framed shapes are as follows:
Function Description

| FrameRect | Draws a rectangle, the position and size of which are defined by a Rect structure. |
| :--- | :--- |
| Frameoval | Draws an oval, the position and size of which are determined by a bounding rectangle <br> defined by a Rect structure. |
| FrameRound Rect | Draws a rounded rectangle, the position and size of which are determined by a bounding <br> rectangle defined by a Rect structure. Curvature of the corners is defined by oval widt h and <br> oval height parameters. |
| FrameArc | Draws an arc, the position and size of which are determined by a bounding rectangle <br> defined by a Rect structure. Starting point and arc extent are determined by st art Angle and <br> arcAngle parameters. |
| FramePoly | Draws a polygon by "playing back" all the line drawing calls that define it. |
| FrameRgn | Draws an outline around a specified region. The line is drawn just inside the region. |

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


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

## Framed Polygons and Regions

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

## Framed Polygons

To define a polygon you must first call openpoly and then call LineTo 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 Fr a meRegion.

## 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.

## Functions for Painting and Filling Shapes

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

| Function | Description |
| :---: | :---: |
| PaintRect | Fills a rectangle with the current foreground colour or graphics pen pixel/ bit pattern. |
| Paint0val | Fills an oval with the current foreground colour or graphics pen pixel/ bit pattern. |
| PaintRoundRect | Fills a round rectangle with the current foreground colour or graphics pen pixel/ bit pattern. |
| PaintArc | Fills a wedge with the current foreground colour or graphics pen pixel/ bit pattern. |
| Paintpoly | Fills a polygon with the current foreground colour or graphics pen pixel/ bit pattern. |
| PaintRgn | Fills a region with the current foreground colour or graphics pen pixel/ bit pattern. |
| Fill Rect | Fills a rectangle with a specified bit pattern. |
| $\underline{\text { FillCRect }}$ | Fills a rectangle with a specified pixel pattern. |
| Filloval | Fills an oval with a specified bit pattern. |
| FillCOval | Fills an oval with a specified pixel pattern. |
| Fill RoundRect | Fills a round rectangle with a specified bit pattern. |
| Fill Croundrect | Fills a round rectangle with a specified pixel pattern. |
| fillarc | Fills a wedge of an oval with a specified bit pattern. |
| FillCArc | Fills a wedge of an oval with a specified pixel pattern. |
| Fill Poly | Fills a polygon with a specified bit pattern. |
| FillCPoly | Fills a polygon with a specified pixel pattern. |
| Fill Rgn | Fills a region with a specified bit pattern. |
| FillCRgn | Fills a region with a specified pixel pattern. |

## Wedges

The wedges drawn by Paintarc, fillarc, andfillCArc 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 Paint Rgn. Note that, where two regions overlap, the additional area is added to the region and the overlap is removed from the 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. |
| Eraseoval | Fills an oval with the current background colour or pixel/ bit pattern. |
| EraseRoundRect | Fills a round rectangle with the current background colour or pixel/ bit pattern. |
| EraseArc | Fills a wedge with the current background colour or pixel/ bit pattern. |
| Erasefoly | Fills a polygon with the current background colour or pixel/ bit pattern. |
| EraseRgn | Fills a region with the current background colour or pixel/ bit pattern. |
| InvertRect | Inverts all the pixels in a rectangle. |
| Invertoval | Inverts all the pixels in an oval. |
| InvertRoundRect | Inverts all the pixels in a round rectangle. |
| Invertarc | Inverts all the pixels in a wedge. |
| Invertpoly | Inverts all the pixels in a polygon. |
| InvertRgn | Inverts all the pixels in a region. |

## Drawing Pictures

Your application can record a sequence of QuickDraw drawing operations in a picture and play its image back later. 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 $\operatorname{sint} 16$, 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 fnum. ${ }^{1}$ For example, the following sets the current font to Palatino:
short font Num;
Get FNum(" 1 pPal atino", \&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, andextend. Some examples of usage are as follows:

TextFace(bold); $\quad$ II Set to bold
TextFace(bold italic); $\quad \|$ Set to bold and italic.)
TextFace(thePort->txFace| bold); $\quad \mid \quad$ Add bold to existing.
TextFace(thePort->txFace $\& \sim$ bold); $\quad \mid l$ Remove bold.
TextFace(normal); $\quad$ /l 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 textjustification functions.

## Transfer Mode

The transfer mode initially assigned to the $\mathrm{t} \mathrm{m}_{\mathrm{m}}$ de field of the colour graphics port is the Boolean source modesrcor. This mode causes the colour of the glyph ${ }^{2}$ to be determined by the foreground colour and the drawn glyph to completely overwrite the existing pixels. (In this mode only those bits which make up the actual glyph are drawn.)

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

[^0]
## 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.
- CopyDepmask, 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 port bits 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 port bits 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, my col our Port, you could specify (Grafptr) myColourport->portBits in the source parameter.

All this works because:

- In a CGrafport structure, the two high bits of the port version 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 pixmap 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 $\mathrm{Pi} \times \mathrm{Map}$ is ignored.

When the PixMap 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 PixMap structures with pixel depths greater than 1, Colour QuickDraw takes a weighted average between the colours in the source and destination Pixmap 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, Copy Mask, 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, Copy Mask, 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.
```


## Copying From Offscreen Graphics Ports

To gracefully display complex images, your application should construct the image in an offscreen graphics world and then use copyBits to transfer the image to the onscreen graphics port. (Offscreen graphics worlds are addressed at Chapter 13 - Offscreen Graphics Worlds, Pictures, Cursors, and Icons.)

## Scrolling Pixels in the Port Rectangle

You can use scroll Rect to scroll the pixels in the port rectangle. Scroll Rect 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 offsetrect to move the oval's bounding rectangle downwards. Using the offset bounding rectangle, you could frame a second, connected oval to form a figure eight with the first oval. You could then use that shape to help define a region. You could create a second region, and then use UnionRgn to create a region from the union of the two.

## Manipulating Rectangles

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

| Function | Description |
| :--- | :--- |
| EmptyRect | Determine whether a rectangle is an empty rectangle. |
| EqualRect | Determine whether two rectangles are equal. |
| InsetRect | Shrinks or expands a rectangle. |
| OffetRect | Moves a rectangle. |
| PtInRect | Determines whether a pixel is enclosed in a rectangle. |
| PttaAgle | Calculates the angle from the middle of a rectangle to a point. |
| PthRect | Determines the smallest rectangle that encloses two points. |
| SectRect | Determines whether two rectangles intersect. |
| UnionRect | Calculates the smallest rectangle that encloses two rectangles. |

## Manipulating Regions

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

| Function | Description |
| :--- | :--- |
| CopyRgn | Makes a copy of a region. |
| DiffRgn | Subtracts one region from another. |
| EmptyRgn | Determines whether a region is empty. |
| EqualRgn | Determines whether two regions have identical sizes, shapes, and locations. |
| InsetRgn | Shrinks or expands a region. |
| Offget gn | Moves a region. |
| PtInRgn | Determines whether a pixel is within a region. |
| RectIngn | Determines whether a rectangle intersects a region. |
| RectRgn | Changes the structure of an existing region to that of a rectangle (using a Rect $).$ |
| SectRgn | Calculates the intersection of two regions. |


| SetEmptyRgn | Sets a region to empty. |
| :--- | :--- |
| SetRectRgn | Changes the structure of an existing region to that of a rectangle (using coordinates). |
| UnionRgn | Calculates the union of two regions. |
| XorRgn | Calculates the difference between the union and the intersection of two regions. |

## Manipulating Polygons

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

## Scaling Shapes and Regions Within the Same Graphics Port

To scale shapes and regions within the same graphics port, you can use the functions Scalept, Mappt, MapRect, MapRgn, and Mappoly.

## Highlighting

Highlighting is used when selecting and deselecting objects such as text or graphics. TextEdit, for example, uses highlighting to indicate selected text. If the current highlight colour is, for example, blue, TextEdit draws the selected text, then uses Invertign 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 graf vars 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। invertign (or, indeed, any of the other drawing or image-copying function that uses the pat xor or srcxor transfer modes.) The highlight mode can be retrieved and set using LMGethilitemode and LMSethilitemode, andBitClr may be used to clear the highlight bit:

```
Ulnt8 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 Text Mode:

```
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 Set ThemeBackground, Set ThemePen, and/or Set ThemewindowBackground. Recall from Chapter 6 - The Appearance Manager that constants of type Theme Brush 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 rgbFgColor or rgbbkcolor 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 orbkpixpat field.

Accordingly, in the era of the Appearance Manager, you should furnish an application which calls Set ThemeBackground, Set Themepen, and/or Set Themewindowbackground 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 penLocSizeModePat; || Pen |ocation, size, mode, pattern.
    RGBColor requestedForeColour; l| rgbFgColor field of colour graphics port.
    RGBColor requestedBackColour; || rgbBkColor field of colour graphics port.
    SIntl6 textTransferMode; l| txMode field of colour graphics port
    PixpatHandle penPixelPattern; |l pnPixPat field of colour graphics port.
    PixpatHandle backPixelPattern; |l bkPixPat field of colour graphics port.
    Pattern penBitPattern; ll lf pen pixel pattern is a bit pattern.
    Pattern backBitPattern; ll lf background pixel pattern is a bit pattern
};
typedef struct drawingEnvironment drawingEnvironment;
```

Your function for re-initialising the drawing environment should:

- Callpennormal 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.
- Calltextmode with the Boolean source modesrcor.
- Call bakpat with the pattern in the QuickDraw global variablewhite.


## Main QuickDraw Constants, Data Types and Functions

## Constants

## Boolean Pattern Modes

| patCopy | $=8$ |
| :--- | :--- |
| patOr | $=9$ |
| patXor | $=10$ |
| patBic | $=11$ |
| notPatCopy | $=12$ |
| notPatOr | $=13$ |
| notPatXor | $=14$ |
| notPatBic | $=15$ |

## Boolean Source Modes

| srcCopy | $=0$ |
| :--- | :--- |
| srcOr | $=1$ |
| srcXor | $=2$ |
| srcBic | $=3$ |
| notSrcCopy | $=4$ |
| notSrcOr | $=5$ |
| notSrcXor | $=6$ |
| notSrcBicc | $=7$ |
| ditherCopy | $=64$ |

## Arithmetic Transfer Modes

| blend | $=32$ |
| :--- | :--- |
| addPin | $=33$ |
| addOver | $=34$ |
| subPin | $=35$ |
| transparent | $=36$ |
| addMax | $=37$ |
| subOver | $=38$ |
| adMin | $=39$ |
| 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 Pixel Type;
```


## Point

```
struct Point
{
    short v
    short h;
};
typedef struct Point Point;
typedef Point *PointPtr.
```

Rect

```
struct Rect
{
\begin{tabular}{ll} 
short & top; \\
short & Ieft;
\end{tabular}
```

```
    short bottom;
    short right;
};
typedef struct Rect Rect;
typedef Rect *RectPtr;
```


## Region

```
struct Region
{
    short rgnSize;
};
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

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

## Getting and Setting Foreground, Background , and Pixel Colour

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


## Creating and Disposing of Pixel Patterns

```
PixPatHandle GetPixPat(short patID);
PixPatHandle NewPixPat(void);
void CopyPixPat(PixPatHandle srcPP,PixPatHandle dstPP)
```

```
void MakeRGBPat(PixPatHandle pp,const RGBColor *myColor);
void DisposePixPat(PixPatHandle pp);
```


## Getting Pattern Resources

```
PatHandle GetPattern(short patternlD);
void GetIndPattern(Pattern*thePat, short patternlistlD, short index);
```


## Changing the Pen and BackGround Pixel Pattern and Bit Pattern

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


## Drawing Lines

| void | MoveTo(short h, short v); ; |
| :--- | :--- |
| void | Move(short dh, short dv); |
| void | LineTo(short h, short v); |
| void | Line(short dh, short dv); |

## Drawing Rectangles

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


## Drawing Rounded Rectangles

void FrameRoundRect(const Rect *r, short oval Width, short oval Height)
void PaintRoundRect(const Rect *r, short oval Width, short oval Height) ;
void Fill RoundRect(const Rect *r, short oval Width, short oval Height, const pattern *pat;
void FillCRoundRect(const Rect *r, short oval Width, short oval Height, pixpat Handle pp);
void InvertRoundRect(const Rect *r, short oval Width, short oval Height);
void EraseRoundRect(const Rect *r, short oval Width, short oval Height);

## Drawing Ovals

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


## Drawing Arcs and Wedges

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


## Drawing and Painting Polygons

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


## Drawing Regions

```
void FrameRgn(RgnHandle rgn);
void PaintRgn(RgnHandle rgn);
void FillCRgn(RgnHandle rgn, PixPatHandle pp);
void EraseRgn(RgnHandle rgn);
```

```
void InvertRgn(RgnHandle rgn);
void Fi||Rgn(RgnHandle rgn, const Pattern *pat);
```


## Setting Text Characteristics

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


## Drawing and Measuring Text

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


## Copying Images

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


## Getting and Setting the Highlight Colour and HighLight Mode

```
void HiliteColor(const RGBColor *color);
void LMGetHiliteRGB(RGBColor *hiliteRGBValue);
void LMSetHiliteRGB(const RGBColor *hiliteRGBValue);
Ulnt8 LMGetHiliteMode(void);
void LMSetHiliteMode(UInt8 value);
```


## Creating and Disposing of Colour Tables

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


## Creating and Managing Polygons

| PolyHandle | OpenPoly(void); |
| :--- | :--- |
| voidd | ClosePoly(void); |
| void | KillPoly(PolyHandle poly); |
| void | OffsetPoly(PolyHandle poly, short dh, short dv) ; |

## Creating and Managing Rectangles

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


## Creating and Managing Regions

```
RgnHandle NewRgn(void);
void OpenRgn(void);
void CloseRgn(RgnHandle dstRgn):
void DisposeRgn(RgnHandlergn);
void CopyRgn(RgnHandle srcRgn, RgnHandle dstRgn);
void SetEmptyRgn(RgnHandle rgn);
void SetRectRgn(RgnHandle rgn, short left, short top, short right, short bottom);
void RectRgn(RgnHandle rgn,const Rect *r):
void OffsetRgn(RgnHandlergn,short dh, short dv);
```

```
void I nsetRgn(RgnHandle rgn, short dh, short dv);
void SectRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
void UnionRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
void DiffRgn(RgnHandle srcRgnA, RgnHandle srcRgnB,RgnHandle dstRgn);
void XorRgn(RgnHandle srcRgnA,RgnHandle srcRgnB,RgnHandle dstRgn);
Boolean PtInRgn(Point pt, RgnHandlergn);
Boolean RectInRgn(const Rect *r,RgnHandle rgn);
Boolean EqualRgn(RgnHandle rgnA,RgnHandle rgnB);
Boolean EmptyRgn(RgnHandle rgn);
OSErr BitMapToRegion(RgnHandle region,const BitMap *bMap);
```


## Scaling and Mapping Points, Rectangles, Polygons, and Regions



## Determining Whether QuickDraw has Finished Drawing

Boolean QDDone(GrafPtr port);

## Retrieving Color QuickDraw Result Codes

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


## Managing Port Rectangles and Clipping Regions

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


## Manipulating Points in Colour Graphics Ports

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


## Demonstration Program

```
*******************************************************************************************
QuickDraw.c
*************************************************************************************************
1/
This program:
- Opens a window in which the results of various QuickDraw drawing operations are
    displayed. Individual line and text drawing, framing, painting, filling, erasing,
    i nverting, and copying operations are chosen from a Demonstration pull - down menu.
- Quits when the user chooses Quit from the File menu.
To keep the non-QuickDraw code to a mi nimum, the program contains no functions for
updating the window or for responding to activate and operating system events
The program utilises the following resources:
    'WIND' resources for the main window, and a small window used for the CopyBits
    demonstration (purgeable) (initially visible).
    An 'MBAR' resource and associated 'MENU' resources (preload, non-purgeable).
    Two 'ICON' resources (purgeable) used for the boolean source modes demonstration.
```



```
Boolean gDrawWithMouseActivated;
SIntl6 gPixelDepth;
Boolean glsColourDevice= false;
RGBColor gWhiteColour = { OXFFFF, OXFFFF, OXFFFF };
RGBColor gBlackColour ={ 0x0000, 0\times0000, 0\times0000 }
RGBCOIOr gRedColour ={ 0xAAAA, 0x0000,, 0x0000 };
RGBColor gYellowColour= { 0xFFFF, OxCCCC, 0x0000 };
RGBColor gGreenColour ={0\times0000, 0x9999, 0\times0000 };
RGBColor gBlueColour= ={0\times6666,0\times6666,0\times9999};
1/
```

$\qquad$
void main (void):
void dolnitManagers (void);
void doEvents (EventRecord *)
void doDemonstrationMenu (SInt16);
void dolines (void);
void doFrameAndPaint (void);
void dofillEraselnvert (void);
void doPolygonAndRegion (void);
void doscrolling (void);
void doText (void);
void dobooleanSourcemodes (void);
void doArithmeticSourceModes (void);
void doHighlighting (void);
void doDrawWithMouse (void)
void doDrawingEnvironment (void)
void dogetDrawingEnvironment (drawingenvironment *);
void doSetDrawingEnvironment (drawingEnvironment *);
void dolnitDrawingenvironment (void);
void doDrawingEnvironmentProof (SInt16);
void doGetDepthAndDevice (void);
UInt16 doRandomNumber (Ulnt16, UInt16):
|| ********************************************************************************* main
void main(void)
{
Handle menubarHdl;
MenuHandle menuHdl;
EventRecord eventStructure;
Boolean gotEvent;
| |

```
\(\qquad\)
```

                                    initialise managers
    dolnitManagers();
    || ...... cause the Appearance-compliant menu bar definition function to be called direct|y
    RegisterAppearanceClient();
    ```

```

    GetDateTime((U) nt 32 *) (&qd.randSeed));
    ```

```

    if(!(menubarHd| = GetNewMBar(rMenubar)))
        ExitToShell();
    Set MenuBar(menubar Hdl);
    DrawMenuBar();
    if(!(menuHdl = GetMenuHandle(mApple)))
        ExitToShell();
    else
        AppendResMenu(menuHdl,' DRVR');
    ```

```

    if(!(gWindowPtr = GetNewCWindow(rWindow,NULL,(WindowPtr)-1)))
        ExitToShel|();
    SetPort(gWindowPtr);
    TextSize(10);
    |/ ........................ get pixel depth and whether colour device for certain Appearance functions
    ```
```

    doGetDepthAndDevice();
    ```

```

    gDone = false;
    while(!gDone)
    {
        gotEvent = Wait NextEvent(everyEvent, &eventStructure,MAXLONG,NULL);
        if(gotEvent)
            doEvents(&eventStructure);
    }
    }
||************************************************************************** dolnitmanagers
void dolnitManagers(void)
{
MaxApplZone();
MoreMasters();
InitGraf(\&qd.thePort);
InitFonts();
Init Windows();
I nitMenus();
TEInit()
InitDialogs(NULL);
InitCursor();
FI ushEvents(everyEvent,0);
}
||******************************************************************************* doEvents
void doEvents(EventRecord *eventStrucPtr)
{
SInt8 charCode;
SInt32 menuChoice;
SInt16 menulD, menultem;
SInt16 partCode;
Wi ndowPtr windowPtr;
Str255 itemName;
SInt16 daDriverRefNum;
switch(eventStrucPtr - >what)
{
case keyDown:
case autokey:
charCode = eventStrucPtr.>message \& charCodeMask;
if((eventStrucPtr->modifiers \& cmdKey) != 0)
{
menuChoice = MenuEvent(event StrucPtr):
menulD = Hi Word(menuChoice);
menultem = LoWord(menuChoice);
if(menulD == mFile \&\& menultem == iQuit)
gDone = true;
}
break;
case mouseDown:
if(partCode= FindWindow(eventStrucPtr->where, \&windowPtr))
{
switch(partCode)
{
case inMenuBar:
menuChoice= MenuSelect(eventStrucPtr->where);
menulD = Hi Word(menuChoice);
menultem= LoWord(menuChoice);
if(menulD == 0)
return;
switch(menulD)
{
case mApple:

```
```

                    if(menultem== iAbout)
                        SysBeep(10):
                    else
                    {
                            Get MenultemText(Get MenuHandle(mApple), menultem, itemName);
                    daDriverRefNum= OpenDeskAcc(itemName);
                    }
                    break;
                    case mFile
                            if(menultem== iQuit)
                        gDone = true;
                    break;
                    case mDemonstration:
                    doDemonstrationMenu(menultem);
                        break;
                }
                    HiliteMenu(0);
                    break;
                case inDrag:
                    DragWindow(windowPtr, eventStrucPtr - >where, &qd.screenBits.bounds);
                    break;
                case inContent:
                    if(windowPtr != Front Wi ndow())
                    Select Window(windowPtr);
                    else
                    if(gDrawWithmouseActivated)
                        doDrawWithMouse();
                    break;
                }
        }
        break;
        case updateEvt:
        wi ndowPtr = (WindowPtr) eventStrucPtr->message;
        BeginUpdate(windowPtr);
        EndUpdate(wi ndowPtr);
        break;
    }
    }
||********************************************************************** dodemonstrationMenu
void doDemonstrationMenu(SInt 16 menultem)
{
gDrawWithMouseActivated = false;
switch(menultem)
{
case iLine.
dolines();
break;
case i FrameAndPaint:
doFrameAndPaint();
break;
case i FillEraselnvert:
doFillEraselnvert();
break;
case i PolygonRegion:
doPolygonAndRegion();
break;
case iText:
doText();
break;
case iScrolling:
doscrolling();
break;

```
```

        case i BooleansourceModes:
            doBool eanSourceModes();
            break;
        case iArithmeticSourceModes:
        doArithmet icSourceModes();
        break;
        case i Highlighting:
        doHighlighting();
        break;
        case i DrawWithMouse
        Set WTit|e(gWindowPtr,"\pDrawing with the mouse");
        RGBBackColor(&gWhiteColour);
        Fi||Rect(&gWindowPtr->portRect,&qd.white);
        gDrawWithMouseActivated= true;
        break;
        case i DrawingEnvironment:
        doDrawingenvironment();
        break;
    }
}
||****************************************************************************** dolines
void dolines(void)
{
RgnHandle oldClipRgn;
Rect newClipRect
SInt16 a, b, c, top, left, bottom, right;
RGBColor theColour;
Ulnt32 finalTicks;
Pattern systemPattern;
PixPatHandle pixpatHdl;
PenNormal();
RGBBackColor(\&gBlueColour);
Fi||Rect(\&gWi ndowPtr->portRect,\&qd. white);
newClipRect=gWindowPtr->portRect;
| nsetRect(\&newClipRect,10,10);
oldClipRgn=NewRgn();
GetClip(oldClipRgn);
ClipRect(\&newClipRect);
|
Set WTit|e(gWindowPtr,"\pDrawing lines with colours");
RGBBackColor(\&gWhiteColour);
Fi||Rect(\&gWi ndowPtr>>portRect,\&qd. white);
for(a=1;a<60;a++)
{
b= doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.left);
c= doRandomNumber(0,gWi ndowPtr->portRect.right - gWindowPtr->portRect.|eft);
theColour.red= doRandomNumber(0,65535);
theColour.green= doRandomNumber(0,65535);
theColour.blue= doRandomNumber(0,65535);
RGBForeColor(\&theColour);
PenSize(a* 2,1);
MoveTo(b,gWindowPtr->portRect.top);
LineTo(c,gWi ndowPtr->portRect.bottom);
Delay(2,\&final Ticks);
}
11

```
\(\qquad\)
```

                | ines drawn with system-supplied bit patterns
    Set WTit|e(gWindowPtr,"\pClick mouse for more |ines");
    while(!Button()) ;
    ```
```

Set WTitle(gWindowPtr,"\pDrawing lines with system-supplied bit patterns");
Fil|Rect(\&gWindowPtr->portRect,\&qd.white);
c = 0
for(a=1;a<39;a+t)
{
b = doRandomNumber(0,gWindowPtr->portRect.bottom - gWindowPtr->portRect.top);
c= doRandomNumber(O,gWindowPtr->portRect.bottom - gWindowPtr->portRect.top);
theColour.red= doRandomNumber(0,32767);
theColour.green = doRandomNumber(0,32767);
theColour.blue = doRandomNumber(0, 32767);
RGBForeColor(\&theColour);
GetIndPattern(\&systemPattern, sysPatList।D,a);
PenPat(\&systemPattern);
PenSize(1, a * 2);
MoveTo(gWindowPtr - >port Rect.left,b);
LineTo(gWindowPtr->portRect.right,c);
Delay(5,\&finalTicks);
}
1/
Set WTitle(gWindowPtr,"\pClick mouse for more I ines");
while(!Button()) ;
Set WTit| (gWindowPtr,"\pDrawing lines with a pixel pattern");
Fil|Rect(\&gWindowPtr->portRect,\&qd.white);
if(!(pixpatHdl=GetPixPat(rPixelPatternl)))
ExitToShell();
PenPixPat(pixpatHdI);
for(a=1;a<60;a++)
{
b = doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.I eft);
c= doRandomNumber(0,gWindowPtr->portRect.right - gWindowPtr->portRect.Ieft);
PenSize(a * 2,1);
MoveTo(b,gWi ndowPtr->portRect.top);
LineTo(c,gWindowPtr.>portRect.bottom);
Delay(5,\&finalTicks);
}
DisposePixPat(pixpatHdI);
SetClip(oldClipRgn);
DisposeRgn(oldClipRgn);

```

```

SetWTitle(gWindowPtr,"\pClick mouse for more lines");
while(! Button()) ;
Set WTitI (g WindowPtr,"\pDrawing I ines using pattern mode pat Xor");
RGBBackColor(\&gRedColour);
Fil|Rect(\&gWindowPtr->portRect,\&qd.white);

```
```

PenSize(1,1);

```
PenSize(1,1);
PenPat(&qd.black);
PenPat(&qd.black);
PenMode(patXor);
PenMode(patXor);
I eft = gWindowPtr.>portRect.|eft + 10;
top = gWindowPtr->portRect.top + 10;
right = gWindowPtr->portRect.right - 10;
bottom = gWindowPtr->portRect.bottom - 10;
for(a=| eft,b=right;a<right+1;a+t,b - )
{
    MoveTo(a,top);
    LineTo(b,bottom);
```

```
}
    for(a=bottom,b=top;b<bottom+1;a\cdots,b+t)
    {
        MoveTo(left,a);
        LineTo(right,b);
    }
}
|| ************************************************************************* doframeAndPaint
void doFrameAndPaint(void)
{
    SInt16 a;
    Rect therect;
    Ulnt32 finalTicks;
    Pattern systemPattern;
    PixpatHandle pixpatHdl;
    PenNormal();
    PenSize(30,20);
    for(a=0;a<3;a++)
    {
        RGBBackColor(&gWhiteColour);
        Fil|Rect(&gWindowPtr->port Rect,&qd.white);
```



```
        if(a== 0)
        {
        Set WTitle(gWindowPtr,"\pFraming and painting with a colour");
        RGBForeColor(&gRedColour); | | set foreground colour to red
        }
        else if(a == 1)
        {
            Set WTitIe(gWindowPtr,"\pFraming and painting with a bit pattern");
            RGBForeColor(&gBIueColour); Il set foreground colour to blue
            RGBBackColor(&gYellowColour); |l set foreground colour to yellow
            GetIndPattern(&systemPattern,sysPatListID,16); |l get bit pattern for pen
            PenPat(&systemPattern); ll set pen bit pattern
        }
        else if (a== 2)
        {
            Set WTitle(gWindowPtr,"\pFraming and painting with a pixel pattern");
            if(!(pixpatHdl= GetPixPat(rPixelPattern1)))
                ExitToShel।();
            PenPixPat(pixpatHdI); l| set pen pixel pattern
        }
    l/
```

$\qquad$

``` framing and painting
    SetRect(&theRect, 30,32,151,191);
    FrameRect(&theRect); // FrameRect
    MoveTo(30,29);
    DrawString("\pFrameRect");
    Delay( 30,&finalTicks);
    OffsetRect(&therect,140,0);
    FrameRoundRect(&theRect, 30,50); | | FrameRoundRect
    MoveTo(170,29);
    DrawString("\pFrameRoundRect");
    Delay(30,&finalTicks);
    OffsetRect(&theRect,140,0);
    FrameOval(&therect); I/ FrameOval
    MoveTo(310,29);
    DrawString("\pFrameOval");
    Delay(30,&finalTicks);
    OffsetRect(&theRect, 140,0);
    FrameArc(&theRect,330,300); I/ FrameArc
    MoveTo(450, 29);
```

```
        Drawstring("\pFrameArc");
        Delay(30,&finalTicks);
        OffsetRect(&theRect,-420,186);
        PaintRect(&theRect); /| PaintRect
        MoveTo(30,214);
        DrawString("\pPaintRect");
        Delay(30,&finalTicks);
        Off setRect(&theRect, 140,0):
        PaintRoundRect(&theRect, 30,50); /| PaintRoundRect
        MoveTo(170,214);
        DrawString("\pPaintRoundRect");
        Delay(30,&finalTicks);
        Of fsetRect(&theRect,140,0);
        PaintOval(&theRect); |l PaintOval
        MoveTo(310,214);
        DrawString("\pPaintOval");
        Delay( 30, &final Ticks);
        Of fsetRect(&theRect,140,0):
        PaintArc(&theRect, 330,300);
        |/ PaintArc
        MoveTo(450,214)
        DrawString("\pPaintArc");
        Delay(30,&finalTicks);
        if(a<2)
        {
        Set WTitI (gWindowPtr,"\pClick mouse for more");
        while(!Button()) ;
        }
    }
    DisposePixPat(pixpatHdl);
}
||*************************************************************************** dofillEraselnvert
void dofillEraselnvert(void)
{
    SInt16 a;
    Pattern fil|Pat, backPat;
    PixPatHandle fi||PixpatHdl, backPixpatHdl.
    Rect therect;
    UInt32 finalTicks;
    PenNormal();
    PenSize(30,20);
    for(a=0;a<4;at+)
    {
        if(a < 3)
        {
            RGBBackColor(&gWhiteColour);
            Fi||Rect(&gWi ndowPtr->portRect,&qd.white);
        }
```



```
        if(a == 0)
        {
            SetWTit|e(gWindowPtr,"\pFilling and erasing with colours");
        RGBForeColor(&gBlueColour): I| set blue colour for foreground
        RGBBackColor(&gRedColour); I| set red colour for background
        GetIndPattern(&fi||Pat, sysPatList|D,1); || get black bit pattern for fi|| functions
        BackPat(&qd.white); || set white bit pattern for background
        }
        else if(a == 1)
        {
        Set WTitIe(gWindowPtr,"\pFilling and erasing with bit patterns");
```

RGBForeColor ( \&gBlueColour) ; RGBBackColor ( \&gYellowColour) ; GetIndPattern(\&fillPat, sysPatListID, 37);

I/ set blue colour for foreground || set yellow colour for background /। get bit pattern for fill functions

```
    GetIndPattern(&backPat, sysPatListID,19); Il get bit pattern for background
    BackPat(&backPat)
}
else if (a == 2)
{
    Set WTitI (g WindowPtr, "\pFilIing and erasing with pixel patterns");
    if(!(fil|PixpatHd|=GetPixPat(rPixe|Patternl))) || get pixel patt - fil| functions
        ExitToShell();
    if(!(backPixpatHdl = GetPixPat(rPixelPattern2))) | get pixel pattern - background
        ExitToShell();
    BackPixPat(backPixpatHdI); I| set pixel pattern - background
}
else if(a == 3)
{
    Set WTitIe(gWindowPtr,"\plnverting");
    BackPat(&qd.white);
    SetRect(&theRect, 30,15,570,29);
    EraseRect(&theRect);
    SetRect(&theRect, 30,200,570,214);
    EraseRect(&theRect);
}
| |
```

$\qquad$

```
                        fi|ling, erasing, and inverting
```

```
SetRect(&theRect, 30,32,151,191);
```

SetRect(\&theRect, 30,32,151,191);
MoveTo(30,29):
MoveTo(30,29):
if(a<2)
if(a<2)
{
{
Fil|Rect(\&theRect,\&fi||Pat); || Fi||Rect
Fil|Rect(\&theRect,\&fi||Pat); || Fi||Rect
DrawString("\pFil|Rect");
DrawString("\pFil|Rect");
}
}
else if(a == 2)
else if(a == 2)
{
{
Fi|ICRect(\&theRect,fi|IPixpatHd|): || Fi||CRect
Fi|ICRect(\&theRect,fi|IPixpatHd|): || Fi||CRect
DrawString("\pFi||CRect");
DrawString("\pFi||CRect");
}
}
else if(a == 3)
else if(a == 3)
{
{
InvertRect(\&theRect); |/| InvertRect
InvertRect(\&theRect); |/| InvertRect
DrawString("\plnvertRect");
DrawString("\plnvertRect");
}
Delay( 30,\&final Ticks);
Delay( 30,\&final Ticks);
OffsetRect(\&therect, 140,0);
OffsetRect(\&therect, 140,0);
MoveTo(170,29);
MoveTo(170,29);
if(a<2)
if(a<2)
{
{
Fi||RoundRect(\&theRect, 30,50,\&fi||Pat); || Fi||RoundRect
Fi||RoundRect(\&theRect, 30,50,\&fi||Pat); || Fi||RoundRect
DrawString("\pFi||RoundRect");
DrawString("\pFi||RoundRect");
}
}
else if(a == 2)
else if(a == 2)
{
{
Fi||CRoundRect(\&theRect, 30,50,fi||PixpatHd|); || Fi||CRoundRect
Fi||CRoundRect(\&theRect, 30,50,fi||PixpatHd|); || Fi||CRoundRect
DrawString("\pFi||CRoundRect");
DrawString("\pFi||CRoundRect");
}
}
else if(a == 3)
else if(a == 3)
{
{
I nvertRoundRect(\&theRect, 30,50); l| InvertRoundRect
I nvertRoundRect(\&theRect, 30,50); l| InvertRoundRect
DrawString("\plnvertRoundRect");
DrawString("\plnvertRoundRect");
}
}
Delay(30,\&finalTicks);
Delay(30,\&finalTicks);
Of fsetRect(\&theRect, 140,0);
Of fsetRect(\&theRect, 140,0);
MoveTo(310,29);
MoveTo(310,29);
if(a < 2)
if(a < 2)
{
{
Fi||Oval(\&theRect,\&fi||Pat); || Fi||Oval
Fi||Oval(\&theRect,\&fi||Pat); || Fi||Oval
DrawString("\pFil|Oval");
DrawString("\pFil|Oval");
}
}
else if(a == 2)
else if(a == 2)
{
{
Fi|ICOval(\&theRect,fi||PixpatHd|); |l Fil|COval
Fi|ICOval(\&theRect,fi||PixpatHd|); |l Fil|COval
DrawString("\pFi||COval");
DrawString("\pFi||COval");
}
}
else if(a == 3)

```
else if(a == 3)
```

\{
InvertOval(\&theRect); || |nvertoval
Drawstring("\plnvertOval");
\}
Delay (30, \&final Ticks);
OffsetRect ( \&theRect, 140, 0) ;
MoveTo(450, 29);
if(a<2)
\{

```
Fi||Arc(&theRect, 3 30, 300,&fil|Pat);
                                    || Fi||Arc
```

    DrawString("|pFil|Arc");
    \}
else if(a = = 2)
\{
FillCArc(\&therect, 330,300,fil|PixpatHd); || FillCArc
Drawstring("\pFil|CArc");
\}
elseif(a == 3)
\{
InvertArc(\&theRect, 330, 300); || |nvertarc
DrawString(" (pInvertArc");
\}
Delay(30, \&final Ticks);
Of fsetRect (\&theRect,-420,186);
MoveTo(30, 214);
if(a<3)
\{
EraseRect (\&theRect) ; I/ Eraserect
DrawString(" $\operatorname{DPEraseRect")\text {;}}$
\}
else
\{
InvertRect (\&theRect); I/ InvertRect
DrawString("\plnvertRect");
\}
Delay ( 30 , \&final Ticks);
Of f setrect (\&therect, 140, 0) ;
MoveTo(170, 214);
if $(a<3)$
\{
EraseRoundRect (\&theRect, 30, 50); / EraseroundRect
DrawString(") pEraseRoundRect");
\}
else
\{
I nvertRoundRect (\&theRect, 30, 50); |l |nvertRoundRect
Drawstring(") plnvertRoundRect");
\}
Delay (30, \&final Ticks);
Offsetrect ( \&therect, 140, 0) ;
MoveTo(310, 214);
if(a<3)
\{
EraseOval(\&theRect); I/ EraseOval
Drawstring("|pEraseOval") ;
\}
else
\{
InvertOval(\&theRect); II Invertoval
Drawstring("|plnvertoval");
\}
Delay ( 30 , \&final Ticks);
Of fsetrect ( \& theRect, 140, 0) ;
MoveTo(450, 214);
if(a<3)
\{
EraseArc(\&theRect, 330,300); $\quad$ // EraseArc
Drawstring(") pEraseArc");
\}
else

```
        {
            InvertArc(&theRect,330,300); || |nvertArc
            DrawString("\plnvertArc");
        }
        Delay( 30,&final Ticks);
        if(a<3)
        {
            Set WTitI (g Wi ndowPtr,"\pClick mouse for more");
            while(!Button()) ;
        }
    }
    DisposePixPat(fil|PixpatHdI);
    Di sposePixPat(backPixpatHdI);
}
||************************************************************************* dopolygonAndRegian
void doPolygonAndRegion(void)
{
    Pattern backPat;
    PixpatHandle fil|PixpatHdl;
    PolyHandle polygonHdl;
    RgnHandle regionHdl;
    Ulnt32 finalTicks;
    Rect theRect;
    Set WTit|e(gWindowPtr,"\pFraming, painting, filling, and erasing polygons and regions");
    RGBBackColor(&gWhiteColour);
    Fi|IRect(&gWi ndowPtr->portRect, &qd.white);
    1/
```

$\qquad$

```
                        preparation
    GetIndPattern(&backPat, sysPatList।D,17); | get bit pattern for background
    BackPat(&backPat); ll set bit pattern for background
    if(!(fil|PixpatHdl=GetPixPat(rPixelPattern2)))|l get pixel pattern for fill functions
        ExitToShell();
    RGBForeCol or(&gRedColour); | set red colour for foreground
    RGBBackColor(&gYellowColour); |l set yellow colour for background
    PenNormal();
    polygonHdl = OpenPoly(); ll 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(); I/ define region
    OpenRgn();
    SetRect(&t heRect, 30, 218,151,279);
    FrameRect(&theRect);
    SetRect(&theRect, 30, 316,151,377);
    FrameRect(&theRect);
    SetRect(&theRect,39,248,142,341);
    FrameOval(&theRect);
    CloseRgn(regionHdl);
    11
```

$\qquad$

``` framing, painting, filling, and erasing
    FramePoly(polygonHdl); I/ FramePoly
    MoveTo(30,29);
    DrawString("\pFramePoly (colour)");
    Delay(30,&finalTicks);
    OffsetPoly(polygonHdl, 140,0);
    PaintPoly(polygonHdl); ll PaintPoly
    MoveTo(170,29);
    DrawString("\pPaintPoly (colour)");
    Delay(30,&finalTicks);
```

```
    OffsetPoly(polygonHdl, 140,0);
    Fil|CPoly(polygonHdI,filIPixpatHdl); ll Fil|CPoly
    MoveTo(310,29);
    DrawString("\pFi|ICPoly (pixel pattern)");
    Delay(30,&finalTicks);
    OffsetPoly(polygonHdl, 140,0);
    ErasePoly(polygonHdl); |/ ErasePoly
    MoveTo(450,29)
    DrawString("\pErasePoly (bit pattern)");
    Delay(30,&finalTicks);
    FrameRgn(regionHdl); |/ FrameRgn
    MoveTo(30,214);
    DrawString("\pFrameRgn (colour)");
    Delay(30,&finalTicks);
    OffsetRgn(regionHdl, 140,0);
    PaintRgn(regionHdl); ll PaintRgn
    MoveTo(170,214);
    DrawString("\pPaintRgn (colour)");
    Delay(30,&finalTicks);
    OffsetRgn(regionHdl, 140,0)
    Fi||CRgn(regionHdl,fil|PixpatHd|); || Fi||CRgn
    MoveTo(310,214);
    DrawString("\pFi||CRgn (pixel pattern)");
    Delay(30,&finalTicks);
    Of f setRgn(regionHdl, 140,0);
    EraseRgn(regionHdl); I/ EraseRgn
    MoveTo(450,214);
    DrawString("\pEraseRgn (bit pattern)");
    Delay(30,&finalTicks);
    KilIPoly(polygonHdI);
    Di sposeRgn(regionHdl);
    DisposePixPat(fil|PixpatHdl);
    BackPat(&qd.white);
}
||************************************************************************************* doText
void doText(void)
{
    SIntl6 windowCentre, a, fontNum, stringWidth;
    Rect therect:
    Str255 textString;
    UInt32 finalTicks;
    RGBBackColor(&gWhiteColour);
    Fi||Rect(&gWi ndowPtr->portRect,&qd.white);
    SetWTitle(gWindowPtr,"\pDrawing text with default source mode (srcor)");
    windowCentre=(gWindowPtr->portRect.right - gWindowPtr->portRect.|eft) | 2;
    SetRect(&theRect,windowCentre,gWi ndowPtr->portRect.top,gWindowPtr->portRect.right,
                gWi ndowPtr->portRect.bottom);
    RGBBackColor(&gBI ueColour);
    Fil|Rect(&theRect,&qd.white);
    for(a=1;a<9;a++)
    {
        I| ......................................................................arious text fonts, text sty|es, and foreground colours
    if(a == 1)
    {
        Get FNum("\pGeneva", &fontNum);
        TextFont(font Num);
        TextFace(normal).
        RGBForeColor(&gRedColour);
    }
    else if(a == 2)
        TextFace(bold);
    else if(a == 3)
```

```
    {
        Get FNum("\pTimes",&font Num);
        TextFont(font Num);
        TextFace(italic);
        RGBForeColor(&gYellowColour);
    }
    else if(a == 4)
    TextFace(underline);
    else if(a == 5)
    {
        GetFNum("\pHelvetica",&fontNum);
        TextFont(font Num);
        TextFace(outline);
        RGBForeColor(&gGreenColour);
    }
    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< l)
        TextSize(a * 2 + 15);
    else
        TextSize(12);
    |/ .............. get a string and draw it in the set font, style, size, and foreground colour
        GetIndString(textString,rFontsStringList,a);
        stringWidth= StringWidth(textString);
        MoveTo(windowCentre - (stringWidth ( 2),a * 46 - 10);
        DrawString(textString);
        Delay(30,&finalTicks);
    }
    11
```

$\qquad$

``` reset to Geneva 10 pt normal
    GetFNum("\pGeneva", &fontNum);
    TextFont(fontNum) ;
    TextSize(10);
    TextFace(normal);
    |/ .................. erase a rectangle, get a string, and use TETextBox to draw it |eft justified
    SetRect(&theRect,gWindowPtr - >portRect.|eft + 5,gWindowPtr->portRect.bottom . 55,
        gWindowPtr->portRect.Ieft + 138,gWindowPtr->portRect.bottom - 5);
    EraseRect(&theRect);
    InsetRect(&theRect,5,5);
    GetIndString(textstring,rFontsStringList, 9);
    RGBForeColor(&gWhiteColour);
    TETextBox(&textString[1],textString[0], &theRect,teFIushLeft);
}
||********************************************************************************* doScrolling
void doscrolling(void)
{
    PixpatHandle pixpat1Hdl, pixpat2Hdl;
    Rect therect;
    RgnHandle oldClipHdl, regionAHdl, regionBHdl, regionCHdl, scrol| RegionHdl;
    SInt16 a;
    SetWTitle(gWindowPtr,"\pScrolling pixels");
    RGBBackColor(&gWhiteColour);
```

```
    Fi||Rect(&(gWindowPtr->portRect),&qd.white);
    if(!(pixpat1Hd| = GetPixPat(rPixe|Pattern1)))
        ExitToShell();
    PenPixPat(pixpat1HdI);
    PenSize(50,0);
    SetRect(&theRect, 30,30,286,371);
    FrameRect(&theRect);
    SetRect(&therect, 315,30,571,371);
    Fil|CRect(&therect, pixpatlHdI);
    if(!(pixpat2HdI = GetPixPat(rPixelPattern2)))
        ExitToShell();
    BackPixPat(pixpat2HdI);
    regionAHdl = NewRgn();
    regionBHdl = NewRgn();
    regionCHdl = NewRgn();
    SetRect(&therect, 80, 30, 236,371);
    RectRgn(regionAHdl, &theRect);
    SetRect(&theRect,315,30,571,371);
    RectRgn(regionBHdl, &t heRect);
    Uni onRgn(regionAHdI, regionBHdI, regionCHdI);
    oldClipHdl = NewRgn();
    GetClip(oldClipHdl);
    SetClip(regionCHdl);
    SetRect(&theRect, 80, 30,571,371);
    scrol|RegionHdl = NewRgn();
    for(a=0;a<371;a++)
    {
        Scrol|Rect(&therect,0,1, scrol|RegionHdl);
        theRect.top ++;
    }
    SetRect(&theRect, 80, 30,571,371);
    BackPixPat(pixpat1HdI);
    for(a=0;a<371;a++)
    {
        Scrol|Rect(&theRect, 0, - 1, scrol|RegionHd|);
        theRect.bottom ..;
    }
    SetClip(oldClipHdI);
    DisposePixPat(pixpat1HdI);
    DisposePixPat(pixpat2HdI);
    DisposeRgn(oldClipHdl);
    DisposeRgn(regi onAHdl);
    DisposeRgn(regionBHdl);
    Di sposeRgn(regionCHdl);
    DisposeRgn(scrol|RegionHdl);
|| *********************************************************************** do Bool eanSourceModes
void dobooleanSourceModes(void)
{
    Rect theRect;
    Handle destIconHdl, sourcelconHdl;
    SInt16 a, b;
    Ulnt32 finalTicks;
    BitMap sourcelconMap;
    Str255 sourceString;
    SetWTit|e(gWindowPtr,"\pBoolean source modes");
    RGBForeColor(&gBlackColour);
    RGBBackColor(&gGreenColour);
    Fil|Rect(&gWindowPtr->portRect,&qd.white);
    SetRect(&theRect,gWi ndowPtr - >portRect.I eft,gWindowPtr->portRect.top,
        gWindowPtr->portRect.right,(gWindowPtr->portRect.bottom .
```

\}

```
        gWi ndowPtr->portRect.top) / 2);
    RGBBackColor(&gWhiteColour):
    Fi||Rect(&theRect,&qd.white);
    destIconHdI= GetIcon(rDestinationIcon);
    sourcelconHdl= Getlcon(rSourcelcon);
    for(a=0;a<2;a++)
    {
    if(a== 1)
    {
        RGBForeColor(&gYellowColour);
        RGBBackColor(&gRedColour);
    }
    SetRect(&theRect, 235,a* * 191 + 30,299,a * 191 + 94);
    P|ot|con(&theRect,dest|conHd|);
    MoveTo(235,a*191 + 27);
    DrawString("\pDestination");
    SetRect(&theRect,304,a*191 + 30,368,a * 191 + 94);
    P|ot|con(&theRect, sourcelconHdl);
    MoveTo(304,a*191 + 27);
    DrawString("\pSource");
}
RGBForeColor(&gB|ackColour);
RGBBackColor(&gWhiteColour);
    for(a=0;a<2;at+)
{
    if(a == 1)
    {
        RGBForeColor(&gYellowColour);
        RGBBackColor(&gRedColour);
    }
    for(b=0;b<8;b++)
    {
        SetRect(&theRect,b * 69 + 28,a * 191 + 121,b * 69 + 92,a * 191 + 185);
        PlotIcon(&theRect,destIconHdI);
    }
}
RGBForecolor(&gBlackColour);
RGBBackColor(&gWhiteColour);
HLock(sourcelconHdl);
    sourcelconMap.baseAddr = *sourcelconHdl;
    sourcelconMap.rowBytes = 4;
    SetRect(&sourcelconMap.bounds, 0,0,32,32);
    for(a=0;a<2;a++)
    {
    if(a == 1)
    {
        RGBForeColor(&gYellowColour);
        RGBBackColor(&gRedColour);
    }
    for(b=0;b<8;b++)
    {
        Delay(30,&finalTicks);
        SetRect(&theRect,b * 69 + 28,a * 191 + 121,b * 69 + 92,a * 191 + 185);
        CopyBits(&sourcelconMap,&qd.theport->portBits, &sourcelconMap.bounds,&therect,
                b,NULL);
        GetIndString(sourcestring,rBooleanstringlist,b + 1);
        MoveTo(b * 69 + 28,a * 191 + 118);
        DrawString(sourcestring);
    }
}
HUnlock(sourcelconHdl);
}
||********************************************************************** doArithmeticSourceModes
```

```
void doArithmeticsourceModes(void)
{
    PicHandle sourceHdl, destinationHdl;
    Rect sourceRect, destRect;
    SInt16 a, b, arithmeticModee= 32;
    Str255 modestring;
    Ulnt32 finalTicks;
    SetWTitI (gWindowPtr,"\pCopyBits with arithmetic source modes");
    RGBForeColor(&gBlackColour);
    RGBBackColor(&gWhiteColour);
    Fi||Rect(&(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 |MAGE");
    if(!(destinationHdl = GetPicture(rPicture + 1)))
        ExitToShell();
    HNoPurge((Handle) destinationHdl);
    for(a=44;a<403;at=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;
            Delay(60,&finalTicks);
            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);
}
|| ****************************************************************************** doHighlighting
void doHighlighting(void)
\0I
    RGBColor oldHighlightColour;
    SIntl6 a;
    Rect therect;
    Ulnt8 hiliteVal;
    UInt32 finalTicks;
```

```
    Set WTitIe(gWindowPtr,"\pHighlighting");
    RGBForeColor(&gBlackColour);
    RGBBackColor(&gWhiteColour)
    Fi|IRect(&(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.");
        Delay(60,&finalTicks);
        SetRect(&theRect,44,a * 100 + 44,557,a * 100 + 104);
        hiliteVal = LMGetHiliteMode();
        BitClr(&hiliteVal, pHiliteBit);
        LMSetHiliteMode(hiliteVal);
        if(a == 1)
        HiliteColor(&gYellowColour):
        else if(a== 2)
            HiliteColor(&gGreenColour);
        InvertRect(&theRect);
        MoveTo(50,a * 100 + 75);
        Del ay(60,&finalTicks);
        DrawString("\pClick mouse to unhighlight. ");
        DrawString("\p(Note: The cal| 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);
        I nvertRect(&theRect);
    }
    HiliteColor(&oldHighlightColour);
    Delay(60,&finalTicks);
    MoveTo(50,350);
    DrawString("\pOriginal highlight colour has been reset.");
}
|| ************************************************************************ doDrawWithMouse
void doDrawWithMouse(void)
{
    PixPatHandle pixpatHdl;
    Point initial Mouse, previousMouse, currentMouse;
    Rect drawRect;
    Ulnt16 randomNumber:
    RGBColor theColour;
    RGBBackColor(&gWhiteColour);
    Fi||Rect(&gWi ndowPtr->portRect, &qd.white);
    if(!(pixpatHdl=GetPixPat(rPixelPattern3)))
        ExitToShell();
    PenPixPat(pixpatHdl);
    Pensize(1,1);
    PenMode(pat Xor);
    GetMouse(&initial Mouse);
    drawRect.|eft = drawRect.right = initial Mouse.h;
    drawRect.top = drawRect.bottom= initial Mouse.v;
    GetMouse(&previousMouse);
    while(Stil|Down())
```

\{
Get Mouse(\&current Mouse):

```
    if(currentMouse.v != previousMouse.v || currentMouse.h != previousMouse.h)
    {
        FrameRect(&drawRect);
        i f(currentMouse.h >= initialMouse.h)
            drawRect.right = currentMouse.h;
        i f(currentMouse.v>= initi al Mouse.v)
            drawRect.bottom=currentMouse.v;
        i f(currentMouse.h s= i niti al Mouse.h)
            drawRect.I eft = currentMouse.h;
        i f(currentMouse.v <= initi al Mouse.v)
            drawRect.top=currentMouse.v;
        FrameRect(&drawRect);
        }
        previous Mouse.v=current Mouse.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(pat Copy);
    PenPat(&qd.black);
    BackPixPat(pixpatHdl);
    randomNumber= doRandomNumber(0,4);
    if(randomNumber== 0)
        PaintRect(&drawRect) ;
    else if(randomNumber== 1)
        Er a seRoundRect(&dr awRect,50,50) ;
    else if(randomNumber== 2)
        PaintOval(&drawRect);
    else if(randomNumber== 3)
        PaintArc(&drawRect,0, doRandomNumber(0, 360));
    BackPat(&qd.white);
}
| | ****************************************************************** doDramingmnvironment
void doDrawingenvironment(void)
{
    drawingEnvironment environment;
    Rect theRect;
    SInt16 a;
    Ulnt32 finalTicks;
    RGBBackColor(&gB|ueColour);
    Fi||Rect(&gWi ndowPtr->portRect,&qd.white);
    SetWTit|e(gWindowPtr,"\pSaving and restoring the drawing environment");
    dolnitDrawingenvironment();
    doDrawingEnvironmentProof(0);
    Delay(120,&final Ticks);
    doGetDrawingEnvironment(&environment);
    theRect = gWindowPtr->portRect;
    theRect.right = = 300;
    Set ThemeBackground(kThemeListViewBackgroundBrush,gPixel Depth,glsColourDevice);
    EraseRect(&theRect);
    theRect.left += 150;
```

```
    Set ThemeBackground(kThemeListViewSort ColumnBackgroundBrush,gPixel Depth,glsColourDevice);
    EraseRect(&theRect);
    Set ThemePen(kThemeList ViewSeparatorBrush,gPi xel Depth,glsColourDevice);
    theRect.I eft - = 150;
    for(a=theRect.top;a<=theRect.bottom;a+=18)
    {
        MoveTo(theRect.left,a);
        LineTo(theRect.right - 1,a);
    }
    Delay(120,&finalTicks);
    doDrawingEnvironmentProof(1);
    Delay(120,&finalTicks);
    doSetDrawingEnvironment(&environment);
    doDrawingEnvironmentProof(2);
}
|| ***************************************************************** dogetDramingEnvironment
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)
    else
        environment->backBitPattern =
                            *(PatPtr) (*(**((CGrafPtr) currentPort)->bkPixPat).patData);
}
||********************************************************************* doset DramingEnvironment
void doSetDrawingEnvironment(drawingEnvironment *environment)
{
    GrafPtr currentPort;
    GetPort(&currentPort);
    SetPenState(&environment - >penLocSizeModePat);
    RGBForeColor(&environment.>requestedForeColour);
    RGBBackColor(&environment - >requestedBackColour);
    TextMode(environment->textTransferMode);
    if(environment - >penPixel Pattern)
        PenPixpat(environment - >penPixelPattern);
    if(environment->backPixel Pattern)
        BackPixPat(environment - >backPixelPattern);
    else
        BackPat(&environment - >backBitPattern);
}
||******************************************************************* dolnit DrawingEnvironment
void dolnitDrawingenvironment()
{
    PenNormal();
    RGBForeColor(&gBlackColour);
    RGBBackColor(&gWhiteColour);
```

```
    TextMode(srcor);
    BackPat(&qd.white)
}
|| *************************************************************** doDrawingEnvironment Proof
void doDrawingEnvironmentProof(SIntl6 phase)
{
    Rect theRect;
    MoveTo(324,phase * 117 + 41);
    if(phase == 0)
        DrawString("\pBefore calls to SetThemePen and Set ThemeBackground");
    else if(phase == 1)
        DrawString("\pAfter cal|s to Set ThemePen and Set ThemeBackground");
    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);
}
|| *********************************************************************** doGetDepthAndDevice
void doGetDepthAndDevice(void)
{
    GDHandle deviceHdl;
    deviceHdl = LMGetMainDevice();
    gPixelDepth = (*(*deviceHdI)
    if(BitTst(&(*deviceHdl)->gdFIags,gdDevType))
        glsColourDevice= true;
}
||************************************************************************* doRandomNumber
UInt16 doRandomNumber(UIntl6 mi nimum, UI nt 16 maximum)
{
    UInt16 randomNumber;
    SInt32 range, t;
    randomNumber = Random();
    range = maximum - minimum;
    t = (randomNumber * range) | 65535;
    return(t + mi nimum);
}
||***********************************************************************************************
```


## 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

[^1]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. gDraw WithmouseActivated 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.
gPixel Depth will be assigned the pixel depth of the main device. glsColourDevice 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 Set ThemeBackground and Set ThemePen.

The fields of the RGBColor global variables are as igned 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 handi ing i n main(), as i nother 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 gDrawithmouseActivated 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 i Draw withmouse case, the only action taken is tocall the relevant applicationdefined function.

Note that the global variable gDrawithMouseActivated is set to false at function entry, and is set to true within the i DrawWithmouse 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 pat Xor. 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, Fill Rect 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 o 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 Ul nt 16). 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 areloaded by the call to GetIndPattern and are drawn from the 38 patterns in the 'PAT\#' resource in the System file with resource ID syspatistld (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 as igns 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 pat xor 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 $\begin{aligned} & \text { wo lines, the pen pattern and mode are set to the defaults and the pen size is }\end{aligned}$ 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 Ioop, RGBForeColor is called to set the requested foreground colour to red.

The second time around the Ioop, RGBForeColor and RGBBackColor are called to set the requested foreground and background colours to, respectively, blue and yellow, Getlndpattern 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 patternto the pen.

## Framing and Painting

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

Before doframeAndPaint exits, Disposepixpat is called to free the memory obtained by the GetPixpat call
dofilleraselnvert 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 I oop, 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 I oop, RGBForeColor and RGBBackColor are called to set the requested
foreground and background colours to, respectively, blue and yellow. In addition,
Get| ndPattern is called twice, once to assign a bit pattern to a Pattern variable which will
be passed as the second parameter i n calls to fill Rect, 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 | oop, 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 |oop, 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.l
```


## Filling, Erasing, and Inverting

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

Before dofilleraselnvert 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 OffetRgn 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 Iocation 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 loopexits, the text font, size and style are returned to Geneva lopt 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.

I n 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 Scroll Rect function. Laterally, this extends from the left inside of the framed rectangle to the right hand side of the filled rectangle. The call to NewRgnthencreates the empty region required by the Scroll Rect 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 Scroll Rect. Scroll Rect fills the "vacated" areas with the background pattern.

Between the for loops, the rectangle used by Scroll Rect 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 plotlcon 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 plotlconcalls 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 Ioop 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 Copy Bits 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 as igned 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 Set Rect 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 t Mode (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 fill Rect clears the window to white.

The first call to GetPicture Ioads a 'PICT' resource into a Picture structure. (Since the ' PICT' resource is purgeable, it is made non-purgeable immediately it is retrieved, used i mmediately, and immediately made purgeable again. $\quad$ The call to Drawicture 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 Iocations, 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 t Mode (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 memp 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 LMGet HilitergB.
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 iflelse 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 lnvert Rect resets the highlight bit. Accordingly, when the user clicks the mouse button, the highlight bit is cleared once again before Invert Rect 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

doDraw WithMouse demonstrates the use of the mouse to define bounding rectangles for QuickDraw shape drawing functions. It al so 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.)
doDraw Withmouse is called when a mousedown occurs in the window while it is the front window, provided that the global variable gDrawithMouseActivated 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 pat $x$ or (Note: For a black-and-white "rubber band", replace the PenPixpat call with PenPat(\&qd.gray).)

The call to Get Mouse 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 Get Mouse as igns 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 filcker 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 Iocation, 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 dolnit Drawingenvironment 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 doget DrawingEnvironment saves the drawing environment prior to the calls to the Appearance Manager functions Set ThemeBackground and Set ThemePen, 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 doSet DrawingEnvironment 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 doDrawingEnvironment Proof 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 dolnit Drawingenvironment, after the calls to the Appearance Manager functions, and after the call to doSet Drawingenvironment

## 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 Get BackColor
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 O if the pattern is a bit pattern and l 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 Pi xPat 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

doset Drawingenvironment 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 penPixel Pattern 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

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

```
doDrawingEnvi ronmentProof is cal|ed 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 dolnitDrawingEnvironment, 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 Set ThemeBackground
and SetThemePen. doRandomNumber returns a random number between the specified mi ni mum value
and the specified maximum value mi nus one
```


[^0]:    ${ }^{1}$ If you know the font family ID, you can get its name by calling the Font Manager's Get $\mathrm{Font} \mathrm{Name} \mathrm{function}$. family ID or the font name, you can use the Resource Manager's Get IndResource function followed by the Get ResInfo function to determine the names and IDs of all available fonts.
    ${ }^{2}$ A glyph is the visual representation of a character.

[^1]:    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 Set ThemeBackground and Set ThemePen. (These functions change the pen and background colours or patterns.)

