

# 12

## **DRAWING WITH QUICKDRAW** *Includes Demonstration Program QuickDraw*

### **Mathematical Foundations of QuickDraw**

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

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

### **The Coordinate Plane**

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

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

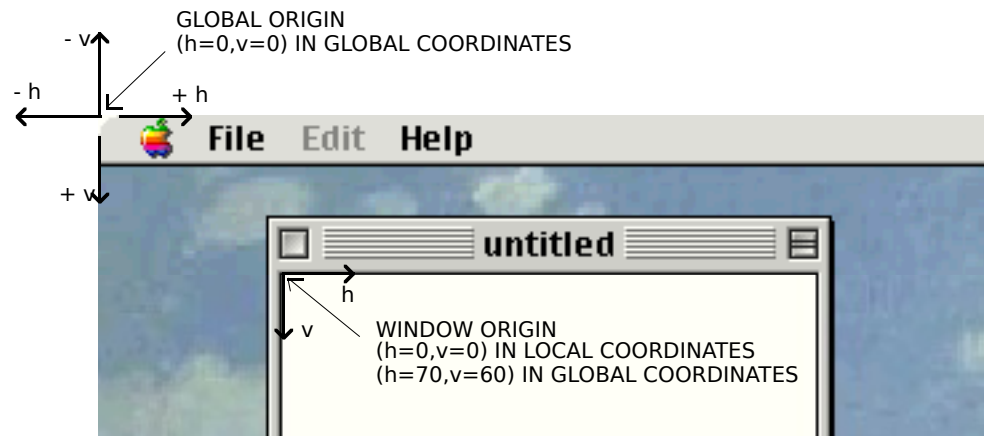


FIG 1 - LOCAL AND GLOBAL COORDINATE SYSTEMS

## Points

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.

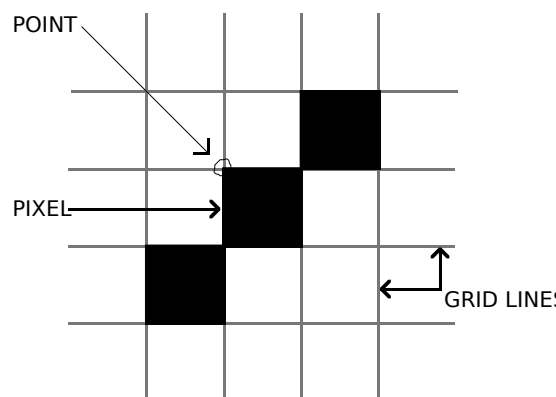


FIG 2 - POINTS AND PIXELS

The data type for points is Point:

```
Point = RECORD
CASE INTEGER OF
0: (
v:    INTEGER;
h:    INTEGER;
);
1: (
vh:   ARRAY [0..1] OF INTEGER;
);
END;

PointPtr = ^Point;
```

## 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:

```
Rect = RECORD
CASE INTEGER OF
```

```

0: (
  top:      INTEGER;
  left:     INTEGER;
  bottom:   INTEGER;
  right:    INTEGER;
);
1: (
  topLeft:  Point;
  botRight: Point;
);
END;

RectPtr = ^Rect;

```

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.

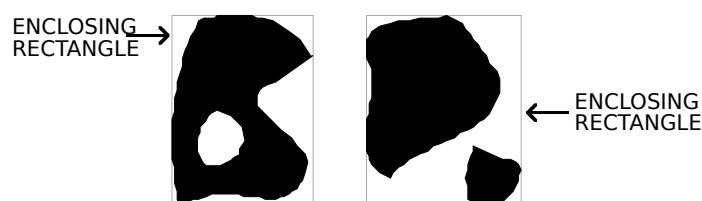


FIG 3 - TWO REGIONS

The data type for regions is Region:

```

Region = RECORD
  rgnSize:      UInt16;      { Size in bytes. }
  rgnBBox:      Rect;        { Enclosing rectangle. }
END;

RegionPtr = ^Region;
RgnPtr = ^Region;
RgnHandle = ^RgnPtr;

```

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

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

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

---

### ***The Graphics Pen***

---

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

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

### Getting and Setting the Pen State

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

Function	Description
<code>GetPenState</code>	Returns, in a <code>PenState</code> record, the graphics pen's current location, size, transfer mode, and pattern.
<code>SetPenState</code>	Using information supplied by a <code>PenState</code> record, sets the graphics pen's location, size, transfer mode, and pattern.
<code>PenNormal</code>	Resets the pen size, transfer mode, and pattern to the state initialised when the colour graphics port was 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 `fgColor` field. The colour represented by the pixel value in the `fgColor` field is the colour actually used as the foreground colour.

If your application uses the Palette Manager, you may also use the Palette Manager function `PmForeColor` to set the foreground colour.

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

#### Background Colour

The function `RGBBackColor` is used to assign a *requested* background colour to the `rgbBkColor` field of the colour graphics port. The pixel value determined by the Color Manager to represent the closest available match for the device is stored in the `bkColor` field. The colour represented by the pixel value in the `bkColor` field is the colour actually used as the background colour.

If your application uses the Palette Manager, you may also use the Palette Manager function `PmBackColor` to set the background colour.

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

## **Pixel Patterns and Bit Patterns**

---

### **Pixel Patterns**

---

If you wish to draw or paint with a colour pattern, rather than the colour in the colour graphics port's fgColor field, you can assign a pixel pattern to the colour graphics port's pnPixPat field using the function PenPixPat. (Initially, the pixel pattern in the colour graphics port's pnPixPat field is all-"black". When you assign a non-all-"black" pattern, the pattern in the pnPixPat field overrides the foreground colour.)

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

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

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

### **Bit Patterns**

---

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

```
PenPat(qd.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 white as follows:

```
BackPat(qd.white);
```

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

The PenPat and BackPat functions may also be used to assign other bit patterns to the pnPixPat and bkPixPat fields of the colour graphics port.

## **Transfer Modes**

---

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

- **Boolean Pattern Mode.** Boolean pattern modes apply to line drawing, framing, painting, erasing, and filling operations.

- **Boolean Source Mode.** Boolean source modes apply to text drawing and copying operations.
- **Arithmetic Source Mode.** Arithmetic source modes apply to drawing (including text drawing), painting, and copying operations.

### **Boolean Pattern Modes**

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

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

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

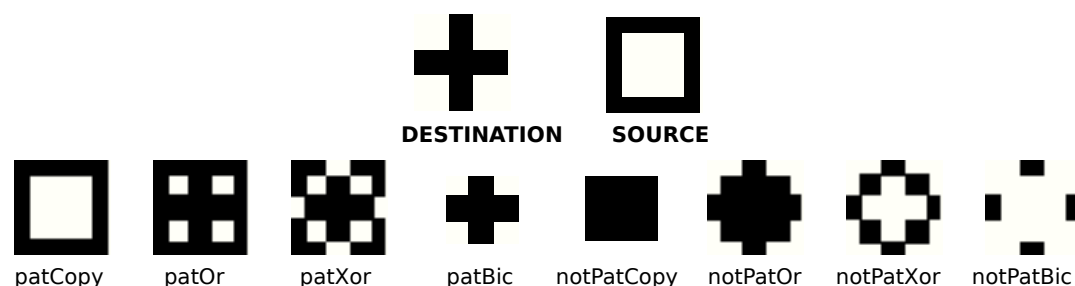


FIG 4 - EFFECT OF PATTERN MODES

### **Boolean Source Modes**

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

The relevant constants are `srcCopy`, `srcOr`, `srcXor`, `srcBic`, `notSrcCopy`, `notSrcOr`, `notSrcXor`, and `notSrcBic`. In general, for pixel images, you will probably want to use the `srcCopy` mode (which causes the destination pixels to be overwritten completely) or one of the arithmetic source modes.

## **Arithmetic Source Modes**

---

Arithmetic source modes may be assigned to both the `pnMode` and `txMode` fields of the colour graphics port, and may be passed as parameters in QuickDraw functions for copying pixel images.

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

<b>Constant</b>	<b>Value</b>	<b>Description</b>
<code>blend</code>	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 <code>srcCopy</code> mode.
<code>addPin</code>	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 <code>srcBic</code> mode.
<code>addOver</code>	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. If the destination is a bitmap or 1-bit pixel image, revert to <code>srcXor</code> mode.
<code>subPin</code>	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 <code>srcOr</code> mode.
<code>transparent</code>	36	Replace the source and destination pixel with the source pixel if the source pixel is not equal to the background colour.
<code>addMax</code>	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 <code>srcBic</code> mode.
<code>subOver</code>	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 <code>srcXor</code> mode.
<code>adMin</code>	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 <code>srcOr</code> 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:

<b>Function</b>	<b>Description</b>
<code>MoveTo</code>	Moves the graphics pen location to the specified location, in local coordinates.
<code>Move</code>	Moves the graphics pen a specified distance from its current location.
<code>LineTo</code>	Draws a line from the current pen location to the specified location, in local coordinates.
<code>Line</code>	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,

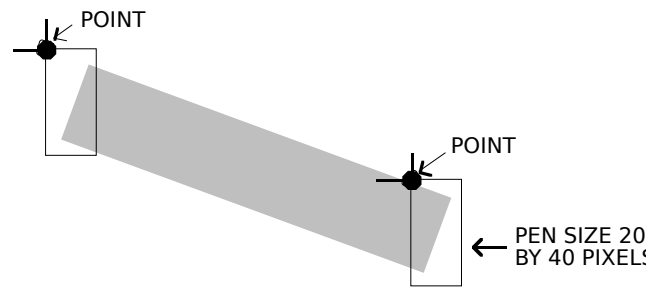


FIG 5 - A LINE DRAWN BY LineTo OR Lir

## 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 is unaffected. Framed shapes are drawn using the current graphics pen size, foreground colour or pen pixel/bit pattern, and pen pattern mode.

Functions for drawing framed shapes are as follows:

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

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

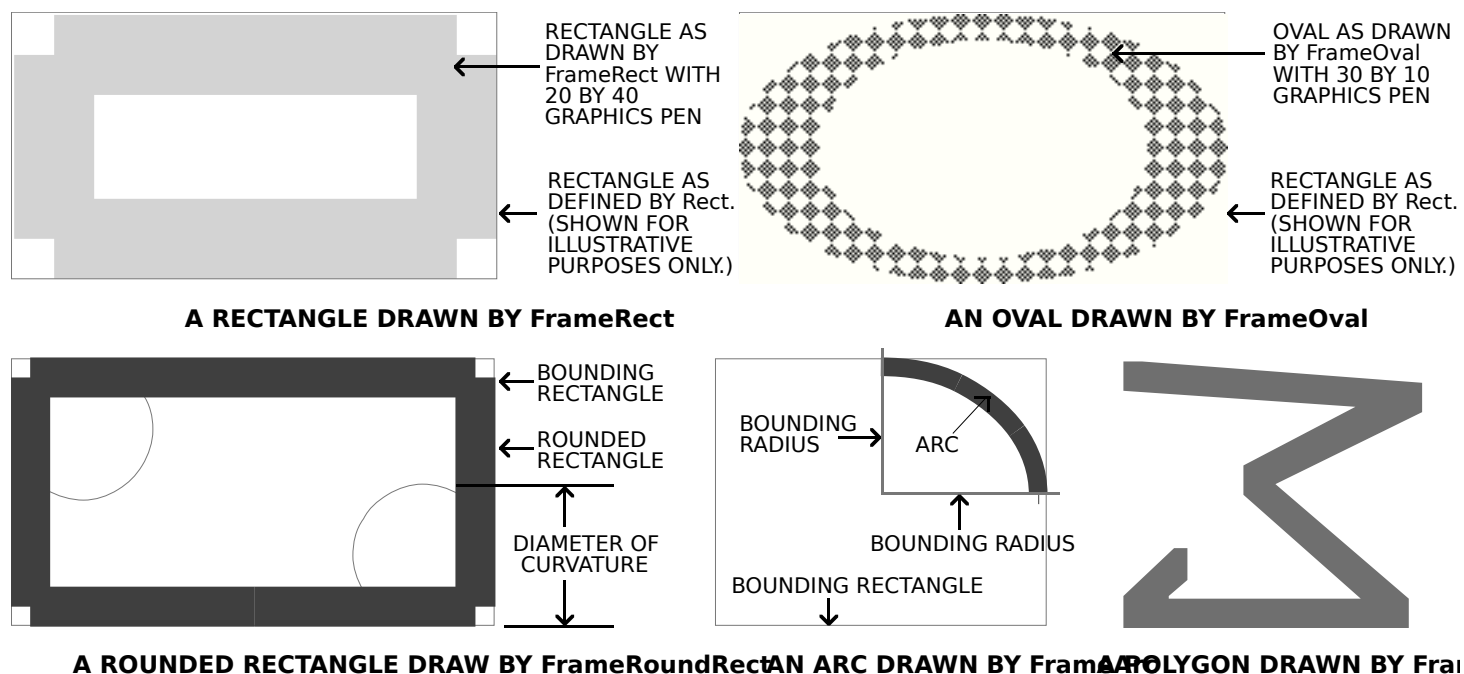


FIG 6 - FRAMED SHAPES DRAWN WITH QUICKDRAW FRAMED SHAPE DRAWING FUNCTION



## **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 `FrameRegion`.

## **Drawing Painted and Filled Shapes**

Painting a shape fills both its outline and its interior with the current foreground colour or graphics pen pixel/bit pattern (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:

<b>Function</b>	<b>Description</b>
<code>PaintRect</code>	Fills a rectangle with the current foreground colour or graphics pen pixel/ bit pattern.
<code>PaintOval</code>	Fills an oval with the current foreground colour or graphics pen pixel/ bit pattern.
<code>PaintRoundRect</code>	Fills a round rectangle with the current foreground colour or graphics pen pixel/ bit pattern.
<code>PaintArc</code>	Fills a wedge with the current foreground colour or graphics pen pixel/ bit pattern.
<code>PaintPoly</code>	Fills a polygon with the current foreground colour or graphics pen pixel/ bit pattern.
<code>PaintRgn</code>	Fills a region with the current foreground colour or graphics pen pixel/ bit pattern.
<code>FillRect</code>	Fills a rectangle with a specified bit pattern.
<code>FillCRect</code>	Fills a rectangle with a specified pixel pattern.

FillOval	Fills an oval with a specified bit pattern.
FillCOval	Fills an oval with a specified pixel pattern.
FillRoundRect	Fills a round rectangle with a specified bit pattern.
FillCRoundRect	Fills a round rectangle with a specified pixel pattern.
FillArc	Fills a wedge of an oval with a specified bit pattern.
FillCArc	Fills a wedge of an oval with a specified pixel pattern.
FillPoly	Fills a polygon with a specified bit pattern.
FillCPoly	Fills a polygon with a specified pixel pattern.
FillRgn	Fills a region with a specified bit pattern.
FillCRgn	Fills a region with a specified pixel pattern.

## Wedges

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

## Painted and Filled Polygons and Regions

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

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

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

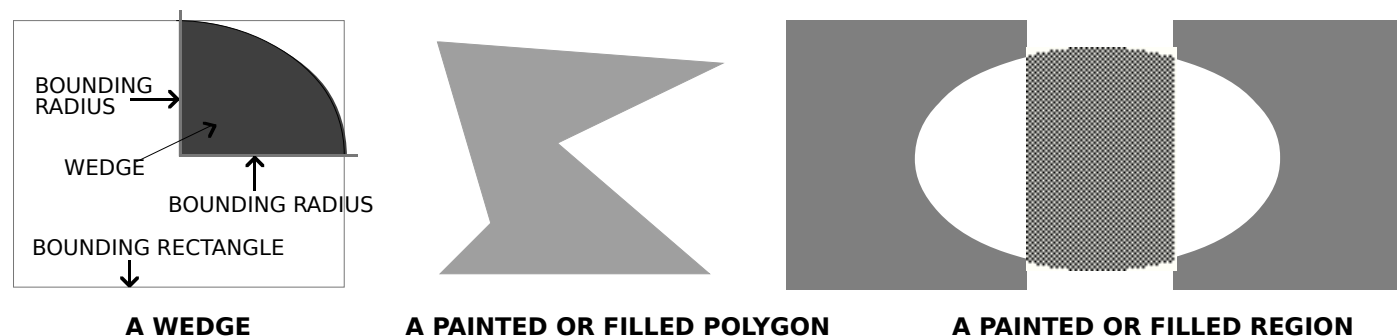


FIG 7 - A WEDGE, A PAINTED OR FILLED POLYGON, AND A PAINTED OR FILLED REGION

## Erasing and Inverting Shapes

Erasing a shape fills both its outline and its interior with the background colour or background pixel/bit pattern (that is, the pattern in the `bkPixPat` field of the colour graphics port). Inverting a shape simply inverts all the pixels in the shape; for example, all black pixels become white, and vice versa.

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

## Functions for Erasing and Inverting Shapes

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

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

## Drawing Pictures

Your application can record a sequence of QuickDraw drawing operations in a **picture** and play its image back later. 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 `SetFont`. `SetFont` takes a single parameter, of type `SIInt16`, 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 `GetFNum`.<sup>1</sup> For example, the following sets the current font to Palatino:

```
fontNum : integer;

GetFNum('Palatino', fontNum);
SetFont(fontNum);
```

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

## Setting and Modifying the Text Style

---

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

```
TextFace([bold]);           { Set to bold. }
TextFace([bold] + [italic=]); { Set to bold and italic. }
TextFace([thePort^.txFace] + [bold]); { Add bold to existing. }
TextFace([thePort^.txFace] - [bold]); { Remove bold. }
TextFace([normal]);       { Set to plain. }
```

## Setting the Font Size

---

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

## Changing the Width of Characters

---

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

## Transfer Mode

---

The transfer mode initially assigned to the `txMode` field of the colour graphics port is the Boolean source mode `srcOr`. This mode causes the colour of the glyph<sup>2</sup> to be determined by the foreground colour and the drawn glyph to completely overwrite the existing pixels. (In this mode only those bits which make up the actual glyph are drawn.)

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

## Copying Pixel Images Between Graphics Ports

---

QuickDraw provides the following three primary image-processing functions:

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

<sup>2</sup> A glyph is the visual representation of a character.

- Specify a transfer mode.
- Modify the image by passing it through a mask.

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

## Coercion of CGrafPtr Data Type to GrafPtr Data Type

The `CopyBits`, `CopyMask`, and `CopyDeepMask` functions date from the era of black-and-white Macintoshes, which is why they expect a pointer to a bitmap, not a pixel map, in their source and destination parameters.

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

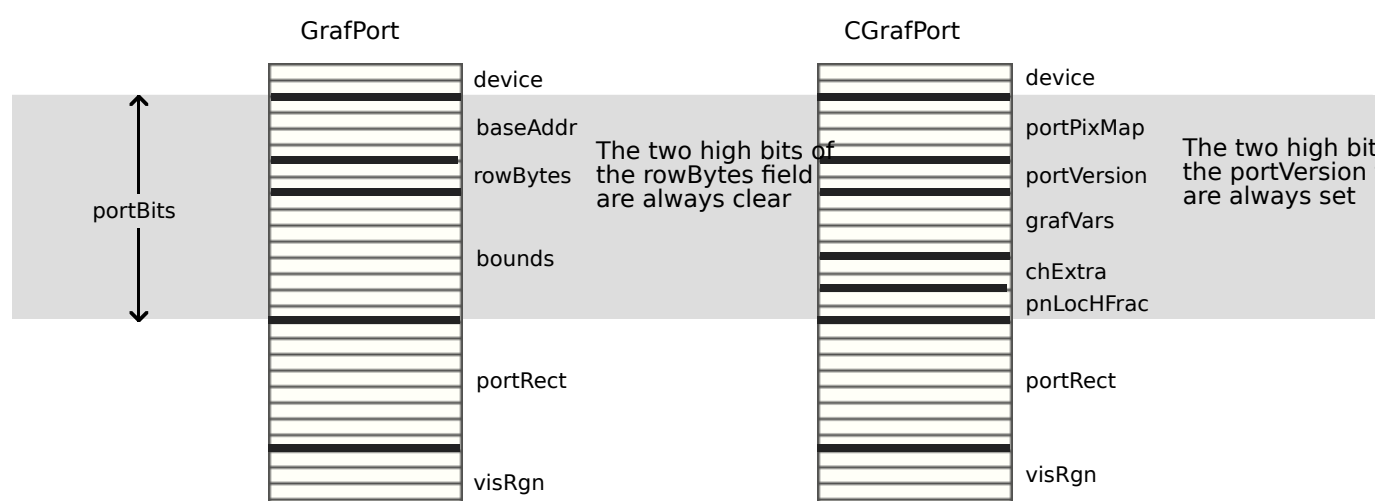


FIG 9 - FIRST 27 BYTES OF GrafPort AND CGrafPort STRUCTURES

When you use `CopyBits`, `CopyMask`, and `CopyDeepMask` to copy pixel images between colour graphics ports, you must coerce each port's `CGrafPtr` data type to a `GrafPtr` data type, dereference the `portBits` fields of each and then pass these "bitmaps" in the source and destination parameters. For example, if your application copies a pixel image from a colour graphics port called, say, `myColourPort`, you could specify `GrafPtr(myColourPort)^.portBits` in the source parameter.

All this works because:

- In a `CGrafPort` structure, the two high bits of the `portVersion` field are always set. These bits in a `GrafPort` structure are the two high bits in the `portBits.rowBytes` field, which are always clear.
- By looking at these bits, `CopyBits`, `CopyMask`, and `CopyDeepMask` can establish that you have passed the functions a handle to a pixel map rather than the base address of a bitmap.

## Using Masks

With `CopyMask` and `CopyDeepMask`, you supply a pixel map to act as the copying mask. The values of pixels in the mask act as weights that proportionally select between source and destination pixel values.

On indexed devices, pixel images are always copied using the colour table of the source `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 PixMap 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, CopyMask, or CopyDeepMask. (That said, setting foreground and background colours to something other than black or white can achieve some interesting colouration effects.)

## **Dithering**

---

You can use **dithering** with CopyBits and CopyDeepMask. Dithering is a technique used by these functions to mix existing colours together to create the illusion of a third colour that may be unavailable on an indexed device, and to improve images that you shrink when copying them from a direct device to an indexed device.

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

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

```
ditherCopy = 64    { Add to source mode for dithering. }
```

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

## ***Manipulating Rectangles and Regions***

---

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

For example, you could define a rectangle to bound an oval and then frame the oval. You could then use `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:

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

### ***Manipulating Regions***

---

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

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

### ***Manipulating Polygons***

---

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

## Scaling Shapes and Regions Within the Same Graphics Port

---

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

## Highlighting

---

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

The **system highlight colour**, which can be changed by the user at the Highlight Color item in the Color pane of the Appearance control panel, is stored in a low memory global represented by the symbolic name `HiliteRGB`. It can be retrieved using `LMGetHiliteRGB`. You can override the default colour using the function `HiliteColor`. The current colour is copied to the `rgbHiliteColor` field of the `GrafVars` structure, a handle to which is stored in the `grafVars` field of the colour graphics port structure.

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

`Color QuickDraw` resets the highlight bit after performing each drawing operation, so your application should always clear the highlight bit immediately before calling `InvertRgn` (or, indeed, any of the other drawing or image-copying function that uses the `patXor` or `srcXor` transfer modes.) The highlight mode can be retrieved and set using `LMGetHiliteMode` and `LMSetHiliteMode`, and `BitClr` may be used to clear the highlight bit:

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

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

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

## Drawing Other Graphics Entities

---

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

- Cursors, which are 16-by-16 pixel images which map the user's movements of the mouse to relative locations on the screen.
- Icons, which are images that an object, concept, or message. Icons are stored as resources.

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



## **Saving and Restoring the Colour Graphics Port Drawing State**

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

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

Since the introduction of the Appearance Manager, it has also become necessary to save and restore the pen pixel/bit pattern and background pixel/bit pattern in functions that call the Appearance Manager functions `SetThemeBackground`, `SetThemePen`, and/or `SetThemeWindowBackground`. Recall from Chapter 6 — The Appearance Manager that constants of type `ThemeBrush` are passed in the `inBrush` parameter of these Appearance Manager functions and that the value in the `inBrush` parameter can represent either a colour or a pattern depending on the current appearance. 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` or `bkPixPat` field.

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

### **Mac OS 8.5 (Appearance Manager Version 1.1) and Later**

Appearance Manager 1.1, which was introduced with Mac OS 8.5, introduced the following functions for saving, restoring, and normalising the graphics port drawing state:

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

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

### **Mac OS 8.1 (Appearance Manager Version 1.0.3) and Earlier**

To accommodate versions of the Appearance Manager earlier than Version 1.1, you could establish a data structure as shown in the following example, and provide applications-defined functions for saving the drawing state to, and restoring it from, the fields of such a structure:

```
DrawingState = record
    penLocSizeModePat : PenState;           { Pen location, size, mode, pattern. }
    requestedForeColour : RGBColor;         { rgbFgColor field of colour graphics port. }
    requestedBackColour : RGBColor;         { rgbBkColor field of colour graphics port. }
    textTransferMode : SInt16;              { txMode field of colour graphics port. }
    penPixelPattern : PixPatHandle;         { pnPixPat field of colour graphics port. }
    backPixelPattern : PixPatHandle;        { bkPixPat field of colour graphics port. }
    penBitPattern : Pattern;                 { If pen pixel pattern is a bit pattern. }
    backBitPattern : Pattern;                { If background pixel pattern is a bit pattern. }
end;
```

Your function for normalising the drawing environment should:

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

## ***Main QuickDraw Constants, Data Types and Functions***

---

### ***Constants***

---

#### ***Boolean Pattern Modes***

<code>patCopy</code>	= 8
<code>patOr</code>	= 9
<code>patXor</code>	= 10
<code>patBic</code>	= 11
<code>notPatCopy</code>	= 12
<code>notPatOr</code>	= 13
<code>notPatXor</code>	= 14
<code>notPatBic</code>	= 15

#### ***Boolean Source Modes***

<code>srcCopy</code>	= 0
<code>srcOr</code>	= 1
<code>srcXor</code>	= 2
<code>srcBic</code>	= 3
<code>notSrcCopy</code>	= 4
<code>notSrcOr</code>	= 5
<code>notSrcXor</code>	= 6
<code>notSrcBic</code>	= 7
<code>ditherCopy</code>	= 64

#### ***Arithmetic Transfer Modes***

<code>blend</code>	= 32
<code>addPin</code>	= 33
<code>addOver</code>	= 34
<code>subPin</code>	= 35
<code>transparent</code>	= 36
<code>addMax</code>	= 37
<code>subOver</code>	= 38
<code>adMin</code>	= 39
<code>ditherCopy</code>	= 64

#### ***Highlighting***

<code>hilite</code>	= 50
<code>hiliteBit</code>	= 7
<code>pHiliteBit</code>	= 0

#### ***Pattern List Resource ID for Pattern Resources in the System File***

<code>sysPatListID</code>	= 0
---------------------------	-----

### ***Data Types***

---

`PixelType` = `SInt8`;

#### ***Point***

`Point` = RECORD  
CASE INTEGER OF  
0: (

```

    v:    INTEGER;
    h:    INTEGER;
  );
  1: (
    vh:   ARRAY [0..1] OF INTEGER;
  );
END;
```

```
PointPtr = ^Point;
```

### **Rect**

```

Rect = RECORD
  CASE INTEGER OF
    0: (
      top:    INTEGER;
      left:   INTEGER;
      bottom: INTEGER;
      right:  INTEGER;
    );
    1: (
      topLeft: Point;
      botRight: Point;
    );
  );
END;
```

```
RectPtr = ^Rect;
```

### **Region**

```

Region = RECORD
  rgnSize:  UInt16;
  rgnBBox:  Rect;
END;
```

```

RegionPtr = ^Region;
RgnPtr = ^Region;
RgnHandle = ^RgnPtr;
```

### **Polygon**

```

Polygon = RECORD
  polySize:          INTEGER;
  polyBBox:          Rect;
  polyPoints:        ARRAY [0..0] OF Point;
END;
```

```

PolygonPtr = ^Polygon;
PolyPtr = ^Polygon;
PolyHandle = ^PolyPtr;
```

### **PenState**

```

PenStatePtr = ^PenState;
PenState = RECORD
  pnLoc:    Point;
  pnSize:   Point;
  pnMode:   INTEGER;
  pnPat:    Pattern;
END;
```

## **Functions**

---

### **Initialising QuickDraw**

```
PROCEDURE InitGraf(globalPtr: UNIV Ptr);
```

### **Managing the Graphics Pen**

```

PROCEDURE HidePen;
PROCEDURE ShowPen;
PROCEDURE GetPen(VAR pt: Point);
PROCEDURE GetPenState(VAR pnState: PenState);
PROCEDURE SetPenState({CONST}VAR pnState: PenState);
PROCEDURE PenSize(width: INTEGER; height: INTEGER);
PROCEDURE PenMode(mode: INTEGER);
```

PROCEDURE PenNormal;

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

```
PROCEDURE RGBForeColor({CONST}VAR color: RGBColor);
PROCEDURE RGBBackColor({CONST}VAR color: RGBColor);
PROCEDURE GetForeColor(VAR color: RGBColor);
PROCEDURE GetBackColor(VAR color: RGBColor);
PROCEDURE GetCPixel(h: INTEGER; v: INTEGER; VAR cPix: RGBColor);
PROCEDURE SetCPixel(h: INTEGER; v: INTEGER; {CONST}VAR cPix: RGBColor);
```

### **Creating and Disposing of Pixel Patterns**

```
FUNCTION GetPixPat(patID: INTEGER): PixPatHandle;
FUNCTION NewPixPat: PixPatHandle;
PROCEDURE CopyPixPat(srcPP: PixPatHandle; dstPP: PixPatHandle);
PROCEDURE MakeRGBPat(pp: PixPatHandle; {CONST}VAR myColor: RGBColor);
PROCEDURE DisposePixPat(pp: PixPatHandle);
```

### **Getting Pattern Resources**

```
FUNCTION GetPattern(patternID: INTEGER): PatHandle;
PROCEDURE GetIndPattern(VAR thePat: Pattern; patternListID: INTEGER; index: INTEGER);
```

### **Changing the Pen and Background Pixel Pattern and Bit Pattern**

```
PROCEDURE BackPixPat(pp: PixPatHandle);
PROCEDURE PenPixPat(pp: PixPatHandle);
PROCEDURE BackPat({CONST}VAR pat: Pattern);
PROCEDURE PenPat({CONST}VAR pat: Pattern);
```

### **Drawing Lines**

```
PROCEDURE MoveTo(h: INTEGER; v: INTEGER);
PROCEDURE Move(dh: INTEGER; dv: INTEGER);
PROCEDURE LineTo(h: INTEGER; v: INTEGER);
PROCEDURE Line(dh: INTEGER; dv: INTEGER);
```

### **Drawing Rectangles**

```
PROCEDURE FrameRect({CONST}VAR r: Rect);
PROCEDURE PaintRect({CONST}VAR r: Rect);
PROCEDURE FillRect({CONST}VAR r: Rect; {CONST}VAR pat: Pattern);
PROCEDURE FillCRect({CONST}VAR r: Rect; pp: PixPatHandle);
PROCEDURE InvertRect({CONST}VAR r: Rect);
PROCEDURE EraseRect({CONST}VAR r: Rect);
```

### **Drawing Rounded Rectangles**

```
PROCEDURE FrameRoundRect({CONST}VAR r: Rect; ovalWidth: INTEGER; ovalHeight: INTEGER);
PROCEDURE PaintRoundRect({CONST}VAR r: Rect; ovalWidth: INTEGER; ovalHeight: INTEGER);
PROCEDURE FillRoundRect({CONST}VAR r: Rect; ovalWidth: INTEGER; ovalHeight: INTEGER;
    {CONST}VAR pat: Pattern);
PROCEDURE FillCRoundRect({CONST}VAR r: Rect; ovalWidth: INTEGER; ovalHeight: INTEGER;
    pp: PixPatHandle);
PROCEDURE InvertRoundRect({CONST}VAR r: Rect; ovalWidth: INTEGER; ovalHeight: INTEGER);
PROCEDURE EraseRoundRect({CONST}VAR r: Rect; ovalWidth: INTEGER; ovalHeight: INTEGER);
```

### **Drawing Ovals**

```
PROCEDURE FrameOval({CONST}VAR r: Rect);
PROCEDURE PaintOval({CONST}VAR r: Rect);
PROCEDURE FillOval({CONST}VAR r: Rect; {CONST}VAR pat: Pattern);
PROCEDURE FillCOval({CONST}VAR r: Rect; pp: PixPatHandle);
PROCEDURE InvertOval({CONST}VAR r: Rect);
PROCEDURE EraseOval({CONST}VAR r: Rect);
```

### **Drawing Arcs and Wedges**

```
PROCEDURE FrameArc({CONST}VAR r: Rect; startAngle: INTEGER; arcAngle: INTEGER);
PROCEDURE PaintArc({CONST}VAR r: Rect; startAngle: INTEGER; arcAngle: INTEGER);
PROCEDURE FillArc({CONST}VAR r: Rect; startAngle: INTEGER; arcAngle: INTEGER;
    {CONST}VAR pat: Pattern);
PROCEDURE FillCArc({CONST}VAR r: Rect; startAngle: INTEGER; arcAngle: INTEGER);
```

```

        pp: PixPatHandle);
PROCEDURE InvertArc({CONST}VAR r: Rect; startAngle: INTEGER; arcAngle: INTEGER);
PROCEDURE EraseArc({CONST}VAR r: Rect; startAngle: INTEGER; arcAngle: INTEGER);

```

### ***Drawing and Painting Polygons***

```

PROCEDURE FramePoly(poly: PolyHandle);
PROCEDURE PaintPoly(poly: PolyHandle);
PROCEDURE FillPoly(poly: PolyHandle; {CONST}VAR pat: Pattern);
PROCEDURE FillCPoly(poly: PolyHandle; pp: PixPatHandle);
PROCEDURE InvertPoly(poly: PolyHandle);
PROCEDURE ErasePoly(poly: PolyHandle);

```

### ***Drawing Regions***

```

PROCEDURE FrameRgn(rgn: RgnHandle);
PROCEDURE PaintRgn(rgn: RgnHandle);
PROCEDURE FillRgn(rgn: RgnHandle; {CONST}VAR pat: Pattern);
PROCEDURE FillCRgn(rgn: RgnHandle; pp: PixPatHandle);
PROCEDURE InvertRgn(rgn: RgnHandle);
PROCEDURE EraseRgn(rgn: RgnHandle);

```

### ***Setting Text Characteristics***

```

PROCEDURE TextFont(font: INTEGER);
PROCEDURE TextFace(face: StyleParameter);
PROCEDURE TextMode(mode: INTEGER);
PROCEDURE TextSize(size: INTEGER);
PROCEDURE SpaceExtra(extra: Fixed);
PROCEDURE GetFontInfo(VAR info: FontInfo);

```

### ***Drawing and Measuring Text***

```

PROCEDURE DrawChar(ch: CharParameter);
PROCEDURE DrawString(s: Str255);
PROCEDURE DrawText(textBuf: UNIV Ptr; firstByte: INTEGER; byteCount: INTEGER);
FUNCTION CharWidth(ch: CharParameter): INTEGER;
FUNCTION StringWidth(s: Str255): INTEGER;

```

### ***Copying Images***

```

PROCEDURE CopyBits({CONST}VAR srcBits: BitMap; {CONST}VAR dstBits: BitMap;
    {CONST}VAR srcRect: Rect; {CONST}VAR dstRect: Rect; mode: INTEGER;
    maskRgn: RgnHandle);
PROCEDURE CopyMask({CONST}VAR srcBits: BitMap; {CONST}VAR maskBits: BitMap;
    {CONST}VAR dstBits: BitMap; {CONST}VAR srcRect: Rect;
    {CONST}VAR maskRect: Rect; {CONST}VAR dstRect: Rect);
PROCEDURE CopyDeepMask({CONST}VAR srcBits: BitMap; {CONST}VAR maskBits: BitMap;
    {CONST}VAR dstBits: BitMap; {CONST}VAR srcRect: Rect;
    {CONST}VAR maskRect: Rect; {CONST}VAR dstRect: Rect; mode: INTEGER;
    maskRgn: RgnHandle);

```

### ***Getting and Setting the Highlight Colour and HighLight Mode***

```

PROCEDURE HiliteColor({CONST}VAR color: RGBColor);
PROCEDURE LMGetHiliteRGB(VAR hiliteRGBValue: RGBColor);
PROCEDURE LMSetHiliteRGB({CONST}VAR hiliteRGBValue: RGBColor);
FUNCTION GetCTable(ctID: INTEGER): CTabHandle;
PROCEDURE DisposeCTable(cTable: CTabHandle);

```

### ***Creating and Managing Polygons***

```

FUNCTION OpenPoly: PolyHandle;
PROCEDURE ClosePoly;
PROCEDURE KillPoly(poly: PolyHandle);
PROCEDURE OffsetPoly(poly: PolyHandle; dh: INTEGER; dv: INTEGER);

```

### ***Creating and Managing Rectangles***

```

PROCEDURE SetRect(VAR r: Rect; left: INTEGER; top: INTEGER; right: INTEGER; bottom: INTEGER);
PROCEDURE OffsetRect(VAR r: Rect; dh: INTEGER; dv: INTEGER);
PROCEDURE InsetRect(VAR r: Rect; dh: INTEGER; dv: INTEGER);
FUNCTION SectRect({CONST}VAR src1: Rect; {CONST}VAR src2: Rect; VAR dstRect: Rect): BOOLEAN;
PROCEDURE UnionRect({CONST}VAR src1: Rect; {CONST}VAR src2: Rect; VAR dstRect: Rect);

```

## Version 2.1

```
FUNCTION PtInRect(pt: Point; {CONST}VAR r: Rect): BOOLEAN;
PROCEDURE Pt2Rect(pt1: Point; pt2: Point; VAR dstRect: Rect);
PROCEDURE PtToAngle({CONST}VAR r: Rect; pt: Point; VAR angle: INTEGER);
FUNCTION EqualRect({CONST}VAR rect1: Rect; {CONST}VAR rect2: Rect): BOOLEAN;
FUNCTION EmptyRect({CONST}VAR r: Rect): BOOLEAN;
```

### **Creating and Managing Regions**

```
FUNCTION NewRgn: RgnHandle;
PROCEDURE OpenRgn;
PROCEDURE CloseRgn(dstRgn: RgnHandle);
PROCEDURE DisposeRgn(rgn: RgnHandle);
PROCEDURE CopyRgn(srcRgn: RgnHandle; dstRgn: RgnHandle);
PROCEDURE SetEmptyRgn(rgn: RgnHandle);
PROCEDURE SetRectRgn(rgn: RgnHandle; left: INTEGER; top: INTEGER; right: INTEGER;
    bottom: INTEGER);
PROCEDURE RectRgn(rgn: RgnHandle; {CONST}VAR r: Rect);
PROCEDURE OffsetRgn(rgn: RgnHandle; dh: INTEGER; dv: INTEGER);
PROCEDURE InsetRgn(rgn: RgnHandle; dh: INTEGER; dv: INTEGER);
PROCEDURE SectRgn(srcRgnA: RgnHandle; srcRgnB: RgnHandle; dstRgn: RgnHandle);
PROCEDURE UnionRgn(srcRgnA: RgnHandle; srcRgnB: RgnHandle; dstRgn: RgnHandle);
PROCEDURE DiffRgn(srcRgnA: RgnHandle; srcRgnB: RgnHandle; dstRgn: RgnHandle);
PROCEDURE XorRgn(srcRgnA: RgnHandle; srcRgnB: RgnHandle; dstRgn: RgnHandle);
FUNCTION RectInRgn({CONST}VAR r: Rect; rgn: RgnHandle): BOOLEAN;
FUNCTION PtInRgn(pt: Point; rgn: RgnHandle): BOOLEAN;
FUNCTION EqualRgn(rgnA: RgnHandle; rgnB: RgnHandle): BOOLEAN;
FUNCTION EmptyRgn(rgn: RgnHandle): BOOLEAN;
FUNCTION BitMapToRegion(region: RgnHandle; {CONST}VAR bMap: BitMap): OSErr;
```

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

```
PROCEDURE ScalePt(VAR pt: Point; {CONST}VAR srcRect: Rect; {CONST}VAR dstRect: Rect);
PROCEDURE MapPt(VAR pt: Point; {CONST}VAR srcRect: Rect; {CONST}VAR dstRect: Rect);
PROCEDURE MapRect(VAR r: Rect; {CONST}VAR srcRect: Rect; {CONST}VAR dstRect: Rect);
PROCEDURE MapRgn(rgn: RgnHandle; {CONST}VAR srcRect: Rect; {CONST}VAR dstRect: Rect);
PROCEDURE MapPoly(poly: PolyHandle; {CONST}VAR srcRect: Rect; {CONST}VAR dstRect: Rect);
```

### **Determining Whether QuickDraw has Finished Drawing**

```
FUNCTION QDDone(port: GrafPtr): BOOLEAN;
```

### **Retrieving Color QuickDraw Result Codes**

```
FUNCTION QDError: INTEGER;
```

### **Managing Port Rectangles and Clipping Regions**

```
PROCEDURE ScrollRect({CONST}VAR r: Rect; dh: INTEGER; dv: INTEGER; updateRgn: RgnHandle);
PROCEDURE SetOrigin(h: INTEGER; v: INTEGER);
PROCEDURE PortSize(width: INTEGER; height: INTEGER);
PROCEDURE MovePortTo(leftGlobal: INTEGER; topGlobal: INTEGER);
PROCEDURE SetClip(rgn: RgnHandle);
PROCEDURE GetClip(rgn: RgnHandle);
PROCEDURE ClipRect({CONST}VAR r: Rect);
```

### **Manipulating Points in Colour Graphics Ports**

```
PROCEDURE LocalToGlobal(VAR pt: Point);
PROCEDURE GlobalToLocal(VAR pt: Point);
PROCEDURE AddPt(src: Point; VAR dst: Point);
PROCEDURE SubPt(src: Point; VAR dst: Point);
PROCEDURE SetPt(VAR pt: Point; h: INTEGER; v: INTEGER);
FUNCTION EqualPt(pt1: Point; pt2: Point): BOOLEAN;
FUNCTION GetPixel(h: INTEGER; v: INTEGER): BOOLEAN;
```

## **Relevant Appearance Manager Data Types and Functions**

The following data types and functions are available only in Mac OS 8.5 (Appearance Manager 1.1) or later.

### **Data Types**

```
ThemeDrawingState = ^LONGINT;
```

## Functions

```
FUNCTION NormalizeThemeDrawingState: OSStatus;
FUNCTION GetThemeDrawingState(VAR outState: ThemeDrawingState): OSStatus;
FUNCTION SetThemeDrawingState(inState: ThemeDrawingState; inDisposeNow: BOOLEAN): OSStatus;
FUNCTION DisposeThemeDrawingState(inState: ThemeDrawingState): OSStatus;
```

## Demonstration Program

---

```
{
// QuickDrawDemo.p
//
// This program:
//
// • Opens a window in which the results of various QuickDraw drawing operations are
//   displayed. Individual line and text drawing, framing, painting, filling, erasing,
//   inverting, 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 minimum, the program contains no functions for
// updating the window or for responding to activate and operating system events.
//
// The program utilises the following resources:
//
// • 'WIND' resources for the main window, and a small window used for the CopyBits
//   demonstration (purgeable) (initially visible).
//
// • An 'MBAR' resource and associated 'MENU' resources (preload, non-purgeable).
//
// • Two 'ICON' resources (purgeable) used for the boolean source modes demonstration
//
// • Two 'PICT' resources (purgeable) used in the arithmetic source modes demonstration
//
// • 'STR#' resources (purgeable) containing strings used in the source modes and text
//   drawing demonstrations.
//
// • Three 'ppat' resources (purgeable), two of which are used in various drawing,
//   framing, painting, filling, and erasing demonstrations. The third is used in the
//   drawing with mouse demonstration.
//
// • A 'SIZE' resource with with is32BitCompatible flag set.
// }
}

program QuickDrawDemo;

//
// .....
// ..... interfaces
//
uses
    { Universal Interfaces. }
    Appearance, Devices, Fonts, GestaltEqu, LowMem, Menus, Processes, QuickdrawText, Sound,
    TextUtils, ToolUtils;

//
// ..... constants
//

const
    rMenubar = 128;
    mApple = 128;
    iAbout = 1;
    mFile = 129;
    iQuit = 11;
    mDemonstration = 131;
    iLine = 1;
    iFrameAndPaint = 2;
    iFillEraseInvert = 3;
    iPolygonRegion = 4;
```

## Version 2.1

```
iText = 5;
iScrolling = 6;
iBooleanSourceModes = 7;
iArithmeticSourceModes = 8;
iHighlighting = 9;
iDrawWithMouse = 10;
iDrawingState = 11;
rWindow = 128;
rPixelPattern1 = 128;
rPixelPattern2 = 129;
rPixelPattern3 = 130;
rDestinationIcon = 128;
rSourceIcon = 129;
rFontsStringList = 128;
rBooleanStringList = 129;
rArithmeticStringList = 130;
rPicture = 128;
MAXLONG = $7FFFFFFF;

//
..... type declarations

type
DrawingState = record
  penLocSizeModePat : PenState;
  requestedForeColour : RGBColor;
  requestedBackColour : RGBColor;
  textTransferMode : SInt16;
  penPixelPattern : PixPatHandle;
  backPixelPattern : PixPatHandle;
  penBitPattern : Pattern;
  backBitPattern : Pattern;
end;

//
..... global variables

var
gMacOS85Present : Boolean;
gDone : boolean;
gWindowPtr : WindowPtr;
gDrawWithMouseActivated : boolean;
gPixelDepth : SInt16;
gIsColourDevice : boolean;
gWhiteColour : RGBColor;
gBlackColour : RGBColor;
gRedColour : RGBColor;
gYellowColour : RGBColor;
gGreenColour : RGBColor;
gBlueColour : RGBColor;

// ..... main
program block variables

osError : OSErr;
response : SInt32;
mainMenuBarHdl : Handle;
mainMenuHdl : MenuHandle;
mainEvent : EventRecord;
mainErr : OSErr;

//
..... routine prototypes

procedure DoInitManagers; forward;
procedure DoEvents({const} var theEvent : EventRecord); forward;
procedure DoDemonstrationMenu(menuItem : SInt16); forward;
procedure DoLines; forward;
procedure DoFrameAndPaint; forward;
procedure DoFillEraseInvert; forward;
procedure DoPolygonAndRegion; forward;
procedure DoScrolling; forward;
procedure DoText; forward;
procedure DoBooleanSourceModes; forward;
procedure DoArithmeticSourceModes; forward;
```





```

    case menuID of
      mApple: begin
        if (menuItem = iAbout) then
          begin
            SysBeep(10);
          end
        else begin
          GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
          daDriverRefNum := OpenDeskAcc(itemName);
          end;
        end;

      mFile: begin
        if (menuItem = iQuit) then
          begin
            gDone := true;
          end;
        end;

      mDemonstration: begin
        DoDemonstrationMenu(menuItem);
      end;

      otherwise begin
        end;

    end;
    { of case statement }

    HiliteMenu(0);
  end;

  inDrag: begin
    DragWindow(windowPtr, theEvent.where, qd.screenBits.bounds);
  end;

  inContent: begin
    if (windowPtr <> FrontWindow) then
      begin
        SelectWindow(windowPtr);
      end
    else begin
      if gDrawWithMouseActivated then
        begin
          DoDrawWithMouse;
        end;
      end;
    end;

    otherwise begin
      end;

  end;
  { of case statement }
end;
end;

updateEvt: begin
  windowPtr := WindowPtr(theEvent.message);
  BeginUpdate(windowPtr);
  EndUpdate(windowPtr);
end;

otherwise begin
  end;

end;
{ of case statement }
end;
{ of procedure DoEvents }

//  DoDemonstrationMenu

procedure DoDemonstrationMenu(menuItem : SInt16);
begin
  gDrawWithMouseActivated := false;

  case menuItem of

```

```

iLine: begin
  DoLines;
end;

iFrameAndPaint: begin
  DoFrameAndPaint;
end;

iFillEraseInvert: begin
  DoFillEraseInvert;
end;

iPolygonRegion: begin
  DoPolygonAndRegion;
end;

iText: begin
  DoText;
end;

iScrolling: begin
  DoScrolling;
end;

iBooleanSourceModes: begin
  DoBooleanSourceModes;
end;

iArithmeticSourceModes: begin
  DoArithmeticSourceModes;
end;


iHighlighting: begin
  DoHighlighting;
end;

iDrawWithMouse: begin
  SetWTitle(gWindowPtr, 'Drawing with the mouse');
  RGBBackColor(gWhiteColour);
  FillRect(gWindowPtr^.portRect, qd.white);
  gDrawWithMouseActivated := true;
end;

iDrawingState: begin
  DoDrawingState;
end;

otherwise begin
  end;

  { of case statement }
end;
  { of procedure DoDemonstrationMenu }

//  DoLines

procedure DoLines;
var
  oldClipRgn : RgnHandle;
  newClipRect : Rect;
  a, b, c, top, left, bottom, right : SInt16;
  theColour : RGBColor;
  finalTicks : UInt32;
  systemPattern : Pattern;
  pixpatHdl : PixPatHandle;

begin
  PenNormal;

  RGBBackColor(gBlueColour);
  FillRect(gWindowPtr^.portRect, qd.white);

  newClipRect := gWindowPtr^.portRect;
  InsetRect(newClipRect, 10, 10);
  oldClipRgn := NewRgn;
  GetClip(oldClipRgn);
  ClipRect(newClipRect);

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

```

```

SetWTitle(gWindowPtr, 'Drawing lines with colours');
RGBBackColor(gWhiteColour);
FillRect(gWindowPtr^.portRect, qd.white);

for a := 1 to 59 do
begin
  b := DoRandomNumber(0, gWindowPtr^.portRect.right - gWindowPtr^.portRect.left);
  c := DoRandomNumber(0, gWindowPtr^.portRect.right - gWindowPtr^.portRect.left);

  theColour.red := DoRandomNumber(0, 65535);
  theColour.green := DoRandomNumber(0, 65535);
  theColour.blue := DoRandomNumber(0, 65535);
  RGBForeColor(theColour);

  PenSize(a * 2, 1);

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

  Delay(2, finalTicks);
end;

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

SetWTitle(gWindowPtr, 'Click mouse for more lines!');
while not Button do
begin
end;
SetWTitle(gWindowPtr, 'Drawing lines with system-supplied bit patterns');
FillRect(gWindowPtr^.portRect, qd.white);
c := 0;

for a := 1 to 38 do
begin
  b := DoRandomNumber(0, gWindowPtr^.portRect.bottom - gWindowPtr^.portRect.top);
  c := DoRandomNumber(0, gWindowPtr^.portRect.bottom - gWindowPtr^.portRect.top);

  theColour.red := DoRandomNumber(0, 32767);
  theColour.green := DoRandomNumber(0, 32767);
  theColour.blue := DoRandomNumber(0, 32767);
  RGBForeColor(theColour);

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

  PenSize(1, a * 2);

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

  Delay(5, finalTicks);
end;

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

SetWTitle(gWindowPtr, 'Click mouse for more lines!');
while not Button do
begin
end;
SetWTitle(gWindowPtr, 'Drawing lines with a pixel pattern');
FillRect(gWindowPtr^.portRect, qd.white);

pixpatHdl := GetPixPat(rPixelPattern1);
if (pixpatHdl = nil) then
begin
  ExitToShell;
end;
PenPixPat(pixpatHdl);

for a := 1 to 59 do
begin
  b := DoRandomNumber(0, gWindowPtr^.portRect.right - gWindowPtr^.portRect.left);
  c := DoRandomNumber(0, gWindowPtr^.portRect.right - gWindowPtr^.portRect.left);

  PenSize(a * 2, 1);

  MoveTo(b, gWindowPtr^.portRect.top);

```

```

    LineTo(c, gWindowPtr^.portRect.bottom);

    Delay(5, finalTicks);
    end;

    DisposePixPat(pixpatHdl);

    SetClip(oldClipRgn);
    DisposeRgn(oldClipRgn);

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

    SetWTitle(gWindowPtr, 'Click mouse for more lines');
    while not Button do
        begin
            end;
    SetWTitle(gWindowPtr, 'Drawing lines using pattern mode patXor');


    RGBBackColor(gRedColour);
    FillRect(gWindowPtr^.portRect, qd.white);

    PenSize(1, 1);
    PenPat(qd.black);
    PenMode(patXor);

    left := gWindowPtr^.portRect.left + 10;
    top := gWindowPtr^.portRect.top + 10;
    right := gWindowPtr^.portRect.right - 10;
    bottom := gWindowPtr^.portRect.bottom - 10;

    b := right;
    for a := left to right + 1 do
        begin
            MoveTo(a, top);
            LineTo(b, bottom);
            b := b - 1;
        end;

    a := bottom;
    for b := top to bottom + 1 do
        begin
            MoveTo(left, a);
            LineTo(right, b);
            a := a - 1;
        end;
    end;
    { of procedure DoLines }

//  DoFrameAndPaint

procedure DoFrameAndPaint;
var
    a : SInt16;
    theRect : Rect;
    finalTicks : UInt32;
    systemPattern : Pattern;
    pixpatHdl : PixPatHandle;

begin
    PenNormal;
    PenSize(30, 20);

    for a := 0 to 2 do
        begin
            RGBBackColor(gWhiteColour);
            FillRect(gWindowPtr^.portRect, qd.white);

            //
            ..... preparation

            if (a = 0) then
                begin
                    SetWTitle(gWindowPtr, 'Framing and painting with a colour');

                    RGBForeColor(gRedColour);           // set foreground colour to red
                    end
                else if (a = 1) then
                    begin

```

## Version 2.1

```
SetWTitle(gWindowPtr, 'Framing and painting with a bit pattern');

RGBForeColor(gBlueColour);           // set foreground colour to blue
RGBBackColor(gYellowColour);        // set foreground colour to yellow
GetIndPattern(systemPattern, sysPatListID, 16); // get bit pattern for pen
PenPat(systemPattern);               // set pen bit pattern
end
else if (a = 2) then
begin
SetWTitle(gWindowPtr, 'Framing and painting with a pixel pattern');

pixpatHdl := GetPixPat(rPixelPattern1); // get pixel pattern for pen
if (pixpatHdl = nil) then
begin
ExitToShell;
end;
PenPixPat(pixpatHdl);                // set pen pixel pattern
end;

//
```

.....  
framing and painting

```
SetRect(theRect, 30, 32, 151, 191);
FrameRect(theRect);                 // FrameRect
MoveTo(30, 29);
DrawString('FrameRect');
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
FrameRoundRect(theRect, 30, 50);    // FrameRoundRect
MoveTo(170, 29);
DrawString('FrameRoundRect');
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
FrameOval(theRect);                 // FrameOval
MoveTo(310, 29);
DrawString('FrameOval');
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
FrameArc(theRect, 330, 300);        // FrameArc
MoveTo(450, 29);
DrawString('FrameArc');
Delay(30, finalTicks);

OffsetRect(theRect, -420, 186);
PaintRect(theRect);                 // PaintRect
MoveTo(30, 214);
DrawString('PaintRect');
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
PaintRoundRect(theRect, 30, 50);    // PaintRoundRect
MoveTo(170, 214);
DrawString('PaintRoundRect');
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
PaintOval(theRect);                 // PaintOval
MoveTo(310, 214);
DrawString('PaintOval');
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
PaintArc(theRect, 330, 300);        // PaintArc
MoveTo(450, 214);
DrawString('PaintArc');
Delay(30, finalTicks);

if (a < 2) then
begin
SetWTitle(gWindowPtr, 'Click mouse for more!');
while not Button do
begin
end;
end;
end;
```



## Version 2.1

```
SetRect(theRect, 30, 32, 151, 191);
MoveTo(30, 29);
if (a < 2) then
  begin
    FillRect(theRect, fillPat);           // FillRect
    DrawString('FillRect');
  end
else if (a = 2) then
  begin
    FillCRect(theRect, fillPixpatHdl);   // FillCRect
    DrawString('FillCRect');
  end
else if (a = 3) then
  begin
    InvertRect(theRect);                 // InvertRect
    DrawString('InvertRect');
  end;
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
MoveTo(170, 29);
if (a < 2) then
  begin
    FillRoundRect(theRect, 30, 50, fillPat); // FillRoundRect
    DrawString('FillRoundRect');
  end
else if (a = 2) then
  begin
    FillCRoundRect(theRect, 30, 50, fillPixpatHdl); // FillCRoundRect
    DrawString('FillCRoundRect');
  end
else if (a = 3) then
  begin
    InvertRoundRect(theRect, 30, 50);     // InvertRoundRect
    DrawString('InvertRoundRect');
  end;
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
MoveTo(310, 29);
if (a < 2) then
  begin
    FillOval(theRect, fillPat);           // FillOval
    DrawString('FillOval');
  end
else if (a = 2) then
  begin
    FillCOval(theRect, fillPixpatHdl);    // FillCOval
    DrawString('FillCOval');
  end
else if (a = 3) then
  begin
    InvertOval(theRect);                 // InvertOval
    DrawString('InvertOval');
  end;
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
MoveTo(450, 29);
if (a < 2) then
  begin
    FillArc(theRect, 330, 300, fillPat);  // FillArc
    DrawString('FillArc');
  end
else if (a = 2) then
  begin
    FillCArc(theRect, 330, 300, fillPixpatHdl); // FillCArc
    DrawString('FillCArc');
  end
else if (a = 3) then
  begin
    InvertArc(theRect, 330, 300);        // InvertArc
    DrawString('InvertArc');
  end;
Delay(30, finalTicks);

OffsetRect(theRect, -420, 186);
MoveTo(30, 214);
if (a < 3) then
  begin
```



```

    EraseRect(theRect);           // EraseRect
    DrawString('EraseRect');
    end
else begin
    InvertRect(theRect);         // InvertRect
    DrawString('InvertRect');
    end;
Delay(30, finalTicks);


OffsetRect(theRect, 140, 0);
MoveTo(170, 214);
if (a < 3) then
    begin
        EraseRoundRect(theRect, 30, 50);           // EraseRoundRect
        DrawString('EraseRoundRect');
        end
    else begin
        InvertRoundRect(theRect, 30, 50);          // InvertRoundRect
        DrawString('InvertRoundRect');
        end;
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
MoveTo(310, 214);
if (a < 3) then
    begin
        EraseOval(theRect);                         // EraseOval
        DrawString('EraseOval');
        end
    else begin
        InvertOval(theRect);                         // InvertOval
        DrawString('InvertOval');
        end;
Delay(30, finalTicks);

OffsetRect(theRect, 140, 0);
MoveTo(450, 214);
if (a < 3) then
    begin
        EraseArc(theRect, 330, 300);                // EraseArc
        DrawString('EraseArc');
        end
    else begin
        InvertArc(theRect, 330, 300);                // InvertArc
        DrawString('InvertArc');
        end;
Delay(30, finalTicks);

if (a < 3) then
    begin
        SetWTitle(gWindowPtr, 'Click mouse for more!');
        while not Button do
            begin
                end;
            end;
        end;
    end;

DisposePixPat(fillPixpatHdl);
DisposePixPat(backPixpatHdl);
end;
{ of procedure DoFillEraseInvert }

//  DoPolygonAndRegion

procedure DoPolygonAndRegion;
var
    backPat : Pattern;
    fillPixpatHdl : PixPatHandle;
    polygonHdl : PolyHandle;
    regionHdl : RgnHandle;
    finalTicks : UInt32;
    theRect : Rect;

begin
    SetWTitle(gWindowPtr, 'Framing, painting, filling, and erasing polygons and regions');

    RGBBackColor(gWhiteColour);
    FillRect(gWindowPtr^.portRect, qd.white);

```

## Version 2.1

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

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

fillPixpatHdl := GetPixPat(rPixelPattern2); // get pixel pattern for fill functions)
if (fillPixpatHdl = nil) then
  begin
    ExitToShell;
  end;
RGBForeColor(gRedColour); // set red colour for foreground
RGBBackColor(gYellowColour); // set yellow colour for background
PenNormal;

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

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

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

FramePoly(polygonHdl); // FramePoly
MoveTo(30, 29);
DrawString('FramePoly (colour)');
Delay(30, finalTicks);

OffsetPoly(polygonHdl, 140, 0);
PaintPoly(polygonHdl); // PaintPoly
MoveTo(170, 29);
DrawString('PaintPoly (colour)');
Delay(30, finalTicks);

OffsetPoly(polygonHdl, 140, 0);
FillCPoly(polygonHdl, fillPixpatHdl); // FillCPoly
MoveTo(310, 29);
DrawString('FillCPoly (pixel pattern)');
Delay(30, finalTicks);

OffsetPoly(polygonHdl, 140, 0);
ErasePoly(polygonHdl); // ErasePoly
MoveTo(450, 29);
DrawString('ErasePoly (bit pattern)');
Delay(30, finalTicks);

FrameRgn(regionHdl); // FrameRgn
MoveTo(30, 214);
DrawString('FrameRgn (colour)');
Delay(30, finalTicks);

OffsetRgn(regionHdl, 140, 0);
PaintRgn(regionHdl); // PaintRgn
MoveTo(170, 214);
DrawString('PaintRgn (colour)');
Delay(30, finalTicks);

OffsetRgn(regionHdl, 140, 0);
FillCRgn(regionHdl, fillPixpatHdl); // FillCRgn
MoveTo(310, 214);
DrawString('FillCRgn (pixel pattern)');
Delay(30, finalTicks);

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



## Version 2.1

```
        end;

    //
    ..... set text size

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

    // ..... 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 div 2), a * 46 - 10);
    DrawString(textString);


    Delay(30, finalTicks);
    end;

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

    GetFNum('Geneva', fontNum);
    TextFont(fontNum);
    TextSize(10);
    TextFace([]);

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

    SetRect(theRect, gWindowPtr^.portRect.left + 5, gWindowPtr^.portRect.bottom - 55,
            gWindowPtr^.portRect.left + 138, gWindowPtr^.portRect.bottom - 5);
    EraseRect(theRect);
    InsetRect(theRect, 5, 5);
    GetIndString(textString, rFontsStringList, 9);
    RGBForeColor(gWhiteColour);
    TETextBox(@textString[1], SInt16(textString[0]), theRect, teFlushLeft);
    end;
    { of procedure DoText }

//  DoScrolling

procedure DoScrolling;
var
    pixpat1Hdl, pixpat2Hdl : PixPatHandle;
    theRect : Rect;
    oldClipHdl, regionAHdl, regionBHdl, regionCHdl, scrollRegionHdl : RgnHandle;
    a : SInt16;

begin
    SetWTitle(gWindowPtr, 'Scrolling pixels');

    RGBBackColor(gWhiteColour);
    FillRect(gWindowPtr^.portRect, qd.white);

    pixpat1Hdl := GetPixPat(rPixelPattern1);
    if (pixpat1Hdl = nil) then
        begin
            ExitToShell;
        end;
    PenPixPat(pixpat1Hdl);
    PenSize(50, 0);
    SetRect(theRect, 30, 30, 286, 371);
    FrameRect(theRect);
    SetRect(theRect, 315, 30, 571, 371);
    FillCRect(theRect, pixpat1Hdl);

    pixpat2Hdl := GetPixPat(rPixelPattern2);
    if (pixpat2Hdl = nil) then
        begin
            ExitToShell;
        end;
    BackPixPat(pixpat2Hdl);

    regionAHdl := NewRgn;
    regionBHdl := NewRgn;
```



```

SetRect(theRect, 304, a * 191 + 30, 368, a * 191 + 94);
PlotIcon(theRect, sourceIconHdl);
MoveTo(304, a * 191 + 27);
DrawString('Source');
end;

```

```

RGBForeColor(gBlackColour);
RGBBackColor(gWhiteColour);

```

```

for a := 0 to 1 do
begin
if (a = 1) then
begin
RGBForeColor(gYellowColour);
RGBBackColor(gRedColour);
end;

for b := 0 to 7 do
begin
SetRect(theRect, b * 69 + 28, a * 191 + 121, b * 69 + 92, a * 191 + 185);
PlotIcon(theRect, destIconHdl);
end;
end;

```

```

RGBForeColor(gBlackColour);
RGBBackColor(gWhiteColour);

```

```

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

```

```

for a := 0 to 1 do
begin
if (a = 1) then
begin
RGBForeColor(gYellowColour);
RGBBackColor(gRedColour);
end;

for b := 0 to 7 do
begin
Delay(30, finalTicks);
SetRect(theRect, b * 69 + 28, a * 191 + 121, b * 69 + 92, a * 191 + 185);
CopyBits(sourceIconMap, qd.thePort^.portBits, sourceIconMap.bounds, theRect,
b, nil);
GetIndString(sourceString, rBooleanStringList, b + 1);
MoveTo(b * 69 + 28, a * 191 + 118);
DrawString(sourceString);
end;
end;

```

```

HUnlock(sourceIconHdl);
end;
{ of procedure DoBooleanSourceModes }

```

```
//  DoArithmeticSourceModes
```

```

procedure DoArithmeticSourceModes;
var
sourceHdl, destinationHdl : PicHandle;
sourceRect, destRect : Rect;
a, b, arithmeticMode : SInt16;
modeString : Str255;
finalTicks : UInt32;

begin
arithmeticMode := 32;
SetWTitle(gWindowPtr, 'CopyBits with arithmetic source modes');

RGBForeColor(gBlackColour);
RGBBackColor(gWhiteColour);
FillRect(gWindowPtr^.portRect, qd.white);

sourceHdl := GetPicture(rPicture);
if (sourceHdl = nil) then
begin
ExitToShell;
end;

```

```

SetRect(sourceRect, 44, 21, 201, 133);
HNoPurge(Handle(sourceHdl));
DrawPicture(sourceHdl, sourceRect);
HPurge(Handle(sourceHdl));
MoveTo(44, 19);
DrawString('SOURCE IMAGE');

destinationHdl := GetPicture(rPicture + 1);
if (destinationHdl = nil) then
begin
ExitToShell;
end;

HNoPurge(Handle(destinationHdl));
for a := 44 to 403 by 179 do
begin
for b := 21 to 274 by 126 do
begin
if ((a <> 44) or (b <> 21)) then
begin
SetRect(destRect, a, b, a+157, b+112);
DrawPicture(destinationHdl, destRect);
end;
end;
end;
HPurge(Handle(destinationHdl));

for a := 44 to 403 by 179 do
begin
for b := 21 to 274 by 126 do
begin
if ((a <> 44) or (b <> 21)) then
begin
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, nil);

arithmeticMode := arithmeticMode + 1;
end;
end;
end;

ReleaseResource(Handle(sourceHdl));
ReleaseResource(Handle(destinationHdl));
end;
{ of procedure DoArithmeticSourceModes }

//  DoHighlighting

procedure DoHighlighting;
var
oldHighlightColour : RGBColor;
a : SInt16;
theRect : Rect;
hiliteVal : SInt8;
finalTicks : UInt32;

begin
SetWTitle(gWindowPtr, 'Highlighting');

RGBForeColor(gBlackColour);
RGBBackColor(gWhiteColour);
FillRect(gWindowPtr^.portRect, qd.white);

LMGetHiliteRGB(oldHighlightColour);

for a := 0 to 2 do
begin
MoveTo(50, a * 100 + 60);
DrawString('Clearing the highlight bit and calling InvertRect. ');
Delay(60, finalTicks);
SetRect(theRect, 44, a * 100 + 44, 557, a * 100 + 104);

```

## Version 2.1

```
hiliteVal := LMGetHiliteMode;
BitClr(@hiliteVal, pHiliteBit);
LMSetHiliteMode(hiliteVal);

if (a = 1) then
  begin
    HiliteColor(gYellowColour);
  end
else if (a = 2) then
  begin
    HiliteColor(gGreenColour);
  end;

InvertRect(theRect);

MoveTo(50, a * 100 + 75);
Delay(60, finalTicks);
DrawString('Click mouse to unhighlight. ');
DrawString('(Note: The call to InvertRect reset the highlight bit ...');

while not Button do
  begin
    end;

MoveTo(45, a * 100 + 90);
DrawString('... so we clear the highlight bit again before calling InvertRect.>');
Delay(60, finalTicks);

LMSetHiliteMode(hiliteVal);

InvertRect(theRect);
end;

HiliteColor(oldHighlightColour);

Delay(60, finalTicks);
MoveTo(50, 350);
DrawString('Original highlight colour has been reset.>');
end;
  { of procedure DoHighlighting }

//  DoDrawWithMouse

procedure DoDrawWithMouse;
var
  pixpatHdl : PixPatHandle;
  initialMouse, previousMouse, currentMouse : Point;
  drawRect : Rect;
  randomNumber : UInt16;
  theColour : RGBColor;

begin
  RGBBackColor(gWhiteColour);
  FillRect(gWindowPtr^.portRect, qd.white);

  pixpatHdl := GetPixPat(rPixelPattern3);
  if (pixpatHdl = nil) then
    begin
      ExitToShell;
    end;
  PenPixPat(pixpatHdl);
  PenSize(1, 1);
  PenMode(patXor);

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

  GetMouse(previousMouse);

  while StillDown do
    begin
      GetMouse(currentMouse);

      if ((currentMouse.v <> previousMouse.v) or (currentMouse.h <> previousMouse.h)) then
        begin
          FrameRect(drawRect);
        end;
    end;
end;
```





## Version 2.1

```
begin
  ignored := NormalizeThemeDrawingState;
end
else begin
  DoNormalizeDrawingState;
end;
{$elsec}
DoNormalizeDrawingState;
{$endc}

DoDrawingStateProof(0);
Delay(120, finalTicks);

{$ifc TARGET_CPU_PPC}
if (gMacOS85Present = true) then
  begin
    ignored := GetThemeDrawingState(themeDrawingState);
  end
else begin
  DoGetDrawingState(portDrawingState);
end;
{$elsec}
DoGetDrawingState(portDrawingState);
{$endc}

theRect := gWindowPtr^.portRect;
theRect.right := theRect.right - 300;

theErr := SetThemeBackground(kThemeListViewBackgroundBrush, gPixelDepth, glsColourDevice);
EraseRect(theRect);

theRect.left := theRect.left + 150;

theErr := SetThemeBackground(kThemeListViewSortColumnBackgroundBrush, gPixelDepth, glsColourDevice);
EraseRect(theRect);

theErr := SetThemePen(kThemeListViewSeparatorBrush, gPixelDepth, glsColourDevice);

theRect.left := theRect.left - 150;
for a := theRect.top to theRect.bottom by 18 do
  begin
    MoveTo(theRect.left, a);
    LineTo(theRect.right - 1, a);
  end;

Delay(120, finalTicks);

DoDrawingStateProof(1);

Delay(120, finalTicks);

{$ifc TARGET_CPU_PPC}
if (gMacOS85Present = true) then
  begin
    ignored := SetThemeDrawingState(themeDrawingState,true);
  end
else begin
  DoSetDrawingState(portDrawingState);
end;
{$elsec}
DoSetDrawingState(portDrawingState);
{$endc}

DoDrawingStateProof(2);
end;
{ of procedure DoDrawingState }

//  DoGetDrawingState

procedure DoGetDrawingState(var portDrawingState : DrawingState);
var
  currentPort : GrafPtr;

begin
  GetPort(currentPort);

  GetPenState(portDrawingState.penLocSizeModePat);
  GetForeColor(portDrawingState.requestedForeColour);
  GetBackColor(portDrawingState.requestedBackColour);
  portDrawingState.textTransferMode := currentPort^.txMode;
```







## **types**

---

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

## **Global Variables**

---

`gMacOS85Present` will be set to true if Mac OS 8.5 (Appearance Manager 1.1) or later is present. `gDone` will be set to true when the user selects Quit from the File menu, thus causing program termination. `gWindowPtr` will be assigned the pointer to the main window's colour graphics port. `gDrawWithMouseActivated` will be set to true when the Draw With Mouse item is chosen from the Demonstration menu, and to false when other items are chosen.

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

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

## **DoEvents**

---

Within the `mouseDown` case, at the `inContent` case, if the `mouseDown` is within the content region of the window when it is the front window and `gDrawWithMouseActivated` is true, the application-defined routine `DoDrawWithMouse` is called.

## **DoDemonstrationMenu**

---

`DoDemonstrationMenu` branches according to the user's choices in the Demonstration menu. In all but the `iDrawWithMouse` case, the only action taken is to call the relevant application-defined routine.

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

## **DoLines**

---

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

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

The next block sets the window's clipping region to a rectangle 10 pixels inside the port rectangle. The first two lines define such a rectangle. The next two lines save the current clipping region for later restoration. The call to `ClipRect` establishes the new clipping region, in effect assigning it to the colour graphics port's `clipRgn` field.

### **Lines Drawn With Foreground Colour And Black Pen Pattern**

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

Within the for loop, random numbers between 0 and the width of the port rectangle are assigned to two variables which will be used to specify the starting and finishing horizontal coordinates for each of 60 drawn lines. The fields of an `RGBColor` variable are also assigned random values, this time between 0 and 65535 (the maximum possible value for a `UInt16`). The call to `RGBColor` assigns this random colour as the requested foreground colour. The pen width is increased by two pixels. Finally, the call to `MoveTo` moves the pen to the random horizontal location at the top of the port rectangle, and the call to `LineTo` draws a line to the random horizontal location at the bottom of the port rectangle. The line drawing is clipped to the current clipping region.

### **Lines Drawn With System-Supplied Bit Patterns**

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

### **Lines Drawn With A Pixel Pattern**

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

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

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

### ***Lines Drawn With Pattern Mode patXor***

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

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

### ***DoFrameAndPaint***

---

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

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

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

#### ***Preparation***

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

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

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

#### ***Framing and Painting***

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

Before `DoFrameAndPaint` exits, `DisposePixPat` is called to free the memory obtained by the `GetPixPat` call.

### ***DoFillEraseInvert***

---

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

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

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

#### ***Preparation***

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

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

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

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

## **Filling, Erasing, and Inverting**

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

Before `DoFillEraseInvert` exits, `DisposePixPat` is called twice to free the memory obtained by the two `GetPixPat` calls.

## **DoPolygonAndRegion**

---

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

### **Preparation**

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

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

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

### **Framing, Painting, Filling, And Erasing**

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

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

## **DoText**

---

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

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

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

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

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

## **DoScrolling**

---

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

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

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

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



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

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

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

Before `DoScrolling` exits, the saved clipping region is restored and all the memory obtained by the `GetPixPat` and `NewRgn` calls is freed.

## **DoBooleanSourceModes**

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

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

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

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

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

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

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

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

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

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

The last line unlocks the icon image data.

## **DoArithmeticSourceModes**

`DoArithmeticSourceModes` demonstrates the effects of the arithmetic source modes.

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

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

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

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

Before DoArithmeticSourceModes exits, ReleaseResource is called twice to free the memory obtained by the GetPicture calls.

## **DoHighlighting**

DoHighlighting demonstrates highlighting, first with the colour set by the user in the Colour pane of the Appearance control panel, and then with two colours set by the program.

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

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

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

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

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

## **DoDrawWithMouse**

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

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

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

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

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

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

- FrameRect is called to draw the framed rectangle.
- If the current mouse horizontal coordinate is greater than or equal to the initial horizontal mouse coordinate, the current mouse horizontal coordinate is assigned to the right field of the rectangle.
- If the current mouse vertical coordinate is greater than or equal to the initial vertical mouse coordinate, the current mouse vertical coordinate is assigned to the bottom field of the rectangle.
- If the current mouse horizontal coordinate is less than or equal to the initial horizontal mouse coordinate, the current mouse horizontal coordinate is assigned to the left field of the rectangle.
- If the current mouse vertical coordinate is less than or equal to the initial vertical mouse coordinate, the current mouse vertical coordinate is assigned to the top field of the rectangle.
- FrameRect is called again with the newly defined rectangle passed in.

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

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

When the mouse button is released:

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

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

## ***doDrawingState***

---

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

Note that the call to NormalizeThemeDrawingState or doNormalizeDrawingState is included in this function for demonstration purposes only. Ordinarily, these would be called (if required) at other points in an application. If the target is the PowerPC target: If Mac OS 8.5 is present, NormalizeThemeDrawingState is called; otherwise the application-defined function doNormalizeDrawingState is called. If the target is the 68K target, the application-defined function doNormalizeDrawingState is called.

The call to GetThemeDrawingState or doGetDrawingState saves the drawing state prior to the calls to the Appearance Manager functions SetThemeBackground and SetThemePen, which, depending on the current appearance, will change either the colour or the pattern in the relevant fields of the colour graphics port. If the target is the PowerPC target: If Mac OS 8.5 is present, GetThemeDrawingState is called; otherwise the application-defined function doGetDrawingState is called. If the target is the 68K target, the application-defined function doGetDrawingState is called.

The call to SetThemeDrawingState or doSetDrawingState restores the saved drawing state.

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

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

## ***doGetDrawingState***

---

doGetDrawingState saves the current drawing state to a variable of type drawingState.

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 drawingState structure. GetForeColor and GetBackColor save the current foreground and background colours. The next line saves the text source mode.

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

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

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

## ***doSetDrawingState***

---

doSetDrawingState restores the saved drawing state.

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 penPixelFormat field of the drawingState 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 backPixelFormat field of the drawingState structure does not contain NULL, the background pixel pattern is restored, otherwise the background bit pattern is restored.

## ***doNormalizeDrawingState***

---

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

## ***doDrawingStateProof***

---

doDrawingStateProof is called by doDrawingState to draw rectangles in the right half of the window in the pen and background colours and patterns as they were after the call to NormalizeThemeDrawingState/doNormalizeDrawingState,

after the calls to the Appearance Manager functions, and after the call to SetThemeDrawingState/doSetDrawingState. Note that the colour or pattern in which the second pair of rectangles is drawn will depend on the current appearance.

### ***DoGetDepthAndDevice and doRandomNumber***

---

DoGetDepthAndDevice and DoRandomNumber are incidental to the demonstration.

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

### ***main program block***

---

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

Note that error handling in this block, as in other areas of the program, is somewhat rudimentary in that the program simply terminates.