10 Version 1.2 (Frozen)

BASIC QUICKDRAW

Includes Demonstration Program BasicQuickDrawPascal

Mathematical Foundations of QuickDraw

QuickDraw defines the following mathematical constructs which are widely used in its routines 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 your 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).

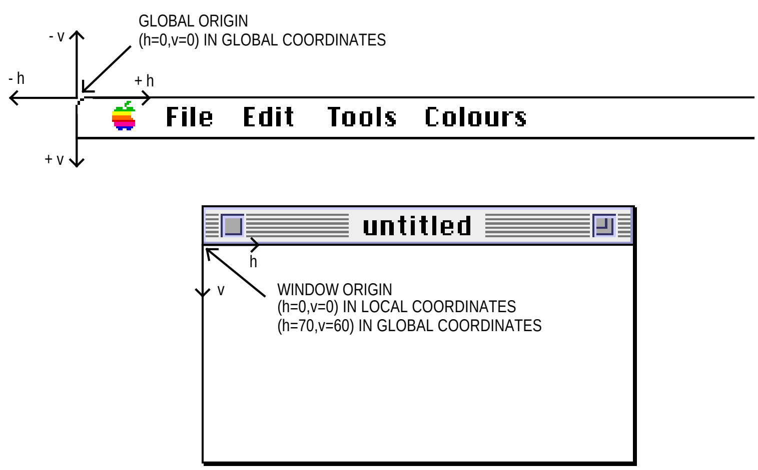

FIG 1 - LOCAL AND GLOBAL COORDINATE SYSTEMS

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.

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.

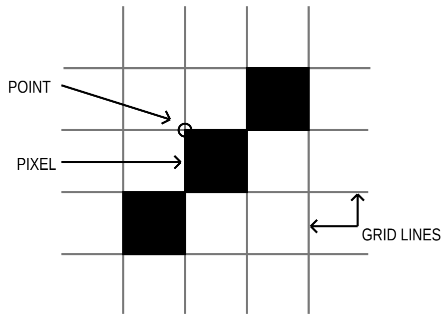

FIG 2 - POINTS AND PIXELS

The data type for points is `Point`:

```
type
Point = record
case integer of
0: (
  v:integer;  {vertical coordinate.}
  h:integer;    {horizontal coordinate.}
   );
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`:

```
type
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.

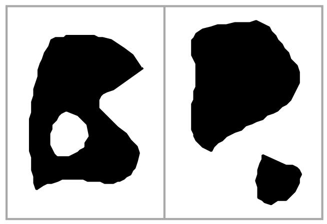

FIG 3 - TWO REGIONS

The data type for regions is `Region`:

```
type
Region = record
  rgnSize:  integer;  {size in bytes}
  rgnBBox:  Rect;      {enclosing rectangle}
  …                    {More data if region is not rectangular.}
  end;

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

The `regionSize` field contains the size, in bytes, of the region. The maximum size is 32 KB when using Basic QuickDraw (64 KB when using Color QuickDraw). 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.

Black and White Drawing: The Basic Graphics Port

The `GrafPort` Structure

Basic QuickDraw performs its operations in a graphics port based on a data structure of type `GrafPort`:

```
GrafPort = record
  device:      integer;    {Device-specific information. (0 = screen.)}
  portBits:    BitMap;     {BitMap.}
  portRect:    Rect;       {Port Rectangle.}
  visRgn:      RgnHandle;  {Visible region.}
  clipRgn:     RgnHandle;  {Clipping region.}
  bkPat:       Pattern;    {Background pattern.}
  fillPat:     Pattern;    {Fill pattern.}
  pnLoc:       Point;      {Pen location.}
  pnSize:      Point;      {Pen size.}
  pnMode:      integer;    {Pen mode.}
  pnPat:       Pattern;    {Pen pattern.}
  pnVis:       integer;    {Pen visibility.}
  txFont:      integer;    {Font number for text.}
  txFace:      Style;      {Text's font style.}
  txMode:      integer;    {Transfer mode for text.}
  txSize:      integer;    {Font size for text.}
  spExtra:     Fixed;      {Spacing for full justification.}
```

```
    fgColor:     longint;    {Foreground colour.}
    bkColor:     longint;    {Background colour.}
    colrBit:     integer;    {Colour bit}
    patStretch:  integer;    {(Used internally.)}
    picSave:     Handle;     {Picture being saved. (Used internally.)}
    rgnSave:     Handle;     {Region being saved. (Used internally.)}
    polySave:    Handle;     {Polygon being saved. (Used internally.)}
    grafProcs:   QDProcsPtr; {Low-level drawing routines.}
    end;
  GrafPtr = ^GrafPort;
  WindowPtr = GrafPtr;
```

Field Descriptions

`portBits` The `portBits` field of a black-and-white graphics port contains the **bitmap**, a data structure of type `bitMap` which defines a black-and-white physical image in terms of the QuickDraw coordinate plane. The `bitMap` data type is as follows:

```
type
BitMap = record
    baseAddr:  Ptr;      {Pointer to bit image.}
    rowBytes:  integer;  {Row width.}
    bounds:    Rect;     {Boundary rectangle.}
    end;

BitMapPtr = ^BitMap;
BitMapHandle = ^BitMapPtr;
```

The `baseAddr` field contains a pointer to the beginning of the **bit image**.[1] A bit image is a collection of bits in memory that form a grid. Fig 4 illustrates a bit image, which can be visualised as a matrix of rows and columns of bits with each row containing the same number of bytes. A bit image can be any length that is a multiple of the row's width in bytes.

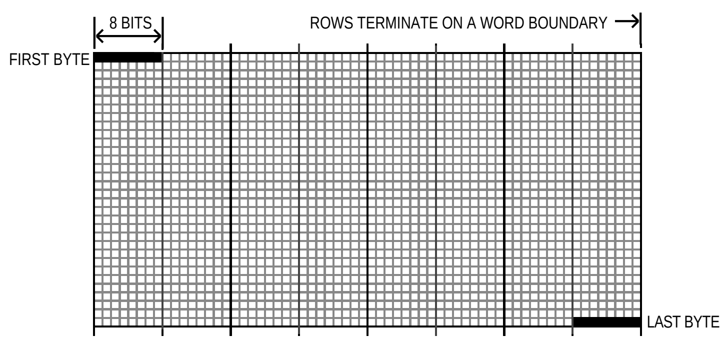

FIG 4 - A BIT IMAGE

The screen itself is one large visible bit image. On a Macintosh Classic, for example, the screen is a 342-by-512 bit image, with a row width of 64 bytes. These 21,888 bytes of memory are displayed as a matrix of 175,104 pixels on the screen. Each bit corresponds to one screen pixel. If a bit's value is 0, its screen pixel is white; if the bit's value is 1, the screen pixel is black.

The `rowBytes` field contains the width of a row in bytes. A bitmap must always begin on a word boundary and contain an integral number of words in each row.

The `bounds` field is the bitmap's **boundary rectangle**. The boundary rectangle serves two purposes. Its first purpose is to link the local coordinates system of a graphics port to QuickDraw's global coordinate system (see Fig 5).

[1]There can be several `bitMap`s pointing to the same bit image, each imposing its own coordinate system on it.

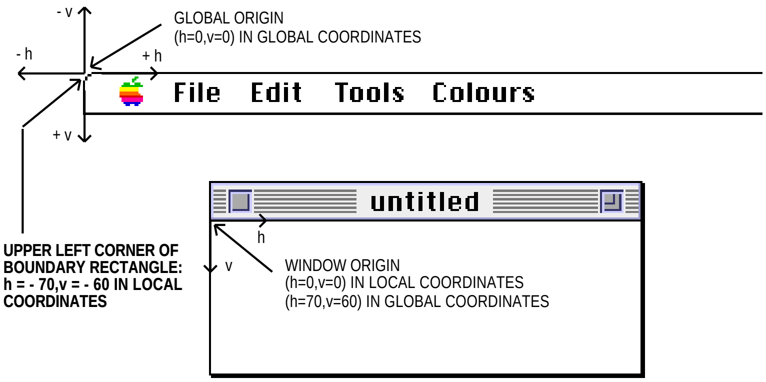

FIG 5 - LOCAL AND GLOBAL COORDINATE SYSTEMS AND THE BOUNDARY RECTANGLE

The boundary rectangle's second purpose is to define the area of an image into which QuickDraw can draw.

`portRect`

The `portRect` field denotes the port rectangle that defines a subset of the bitmap to be used for drawing. All drawing done by your application occurs inside the port rectangle. As previously explained, the boundary rectangle defines the local coordinate system used by the port rectangle. The port rectangle usually falls within the boundary rectangle, but it is not required to do so.

`visRgn`

The `visRgn` field designates the visible region of the graphics port. The visible region is the region of the graphics port that is actually visible on screen, and is manipulated by the Window Manager. For example, if the user moves one window in front of another, the Window Manager logically removes the area of overlap from the visible region of the window at the back. When you draw into the back window, whatever is being drawn is clipped to the visible region so that it does not run over into the front window.

`clipRgn`

The `clipRgn` field specifies the graphics port's **clipping region**, which you can use to limit drawing to any region within the port rectangle. The initial clipping region is an arbitrarily large rectangle covering the entire coordinate plane. You can set the clipping region to any arbitrary region.

`bkPat`
`fillPat`

The `bkPat` and `fillPat` fields of a GrafPort record contain patterns used by certain QuickDraw routines. The `bkPat` field contains the background pattern used when an area is erased or when bits are scrolled out of it. When asked to fill an area with a specified pattern, QuickDraw stores the given pattern in the `fillPat` field and then calls a low-level drawing routine which uses the pattern stored in that field.

`PnLoc`
`pnSize`
`pnMode`
`pnPat`
`pnVis`

The `PnLoc`, `pnSize`, `pnMode`, `pnPat`, and `pnVis` fields of a graphics port relate to the graphics pen. Each graphics port has one, and only one, such pen, which is used for drawing lines, shapes and text. The pen has four characteristics: a location, a size (height and width), a drawing mode, and a drawing pattern.

`txFont`
`txFace`
`txMode`
`txSize`
`spExtra`

The `txFont`, `txFace`, `txMode`, `txSize`, and `spExtra` fields of a graphics port determine how text is drawn, that is, the typeface, font style, font size and how they are placed in a bit image. QuickDraw can draw characters as quickly and easily as it draws lines and shapes. Text is drawn with the baseline positioned at the pen location.

`fgColor`
`bkColor`
`colorBit`

The `fgColor`, `bkColor`, and `colorBit` fields contain values for drawing in the eight-colour system available with basic QuickDraw. (On a colour screen, you can draw with these eight colours even when you are using a basic graphics port.)

The fgColor field contains the graphics port foreground colour (the default is black) and bkColor contains its background colour (the default is white). You can use ForeColor and BackColor to change these fields. The colorBit field tells the colour imaging software which plane of the colour picture to draw into.

Note that these colours are recorded when drawing into a QuickDraw picture[2] (so that the picture can be reconstructed using the specified colours) but they cannot be stored in a bitmap.

More on The Boundary Rectangle, Port Rectangle, Visible Region and Clipping Region

All drawing in a graphics port occurs in the intersection of the boundary rectangle and the port rectangle and, within that intersection, all drawing is cropped to the graphics port's visible region and its clipping region. Fig 6 illustrates the relationship between these rectangles and regions.

As shown at Fig 6, QuickDraw assigns the entire screen as the boundary rectangle of window A. The boundary rectangle shares the same local coordinate system as the port rectangle of window A. The upper-left corner (that is, the window origin) of this port rectangle has a horizontal coordinate of 0 and a vertical coordinate of 0, whereas the upper-left corner for window A's boundary rectangle has a horizontal coordinate of -60 and a vertical coordinate of -40. The clipping region shown has been set by the program, using SetClip, to exclude the scroll bar areas of Window B. This ensures that any drawing in Window B will not over-write the scroll bars.

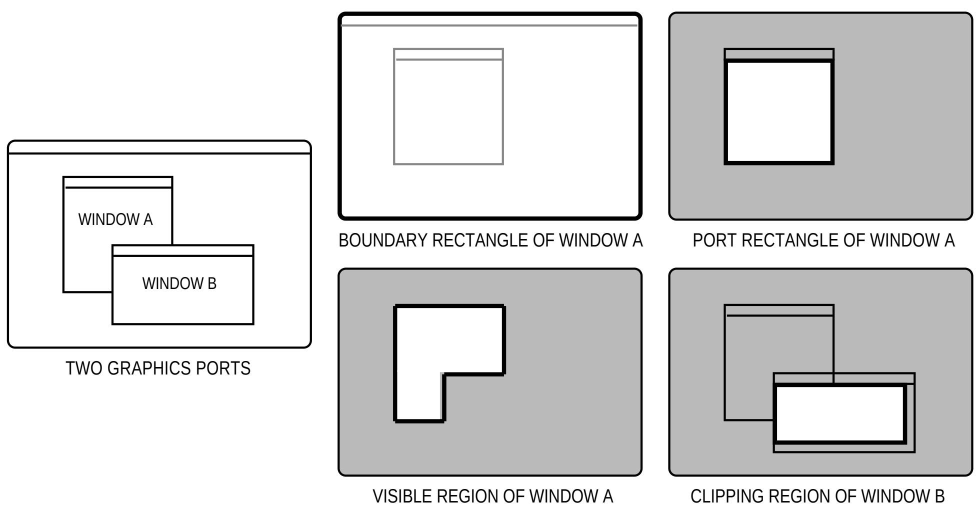

FIG 6 - BOUNDARY RECTANGLE, PORT RECTANGLE, VISIBLE REGION AND CLIPPING REGION

Drawing in Basic Graphics Ports

The QuickDraw routines described in the following operate in both a basic graphics port and a colour graphics port. Many of these routines have additional capabilities when performed in the more sophisticated colour environment provided by Color QuickDraw. However, if your application does not use colour, or uses only a few colours, you may find it unnecessary to create the Color QuickDraw environment.

[2]See Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

The Graphics Pen

The metaphorical graphics pen used for drawing in the graphics port is rectangular in shape and its size (that is, its height and width) is measured in pixels. 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-32 767 pixel square. Note that, if either the width or height is set to 0, the pen does not draw.

Graphics Pen Characteristics

Whenever you draw into a graphics port, the characteristics of the graphics pen determine how the drawing looks. Those characteristics are:

- Pen **location**, specified in local coordinates stored in the `pnLoc` field of the graphics port.
- Pen **size**, specified by the width and height (in pixels) stored in the `pnSize` field of the graphics port.
- Pen **pattern**, which defines, in effect, the "ink" that the pen draws with, and which is stored in the `pnPat` field of the graphics port. The pen pattern, which can range from solid black to intricate patterns, is defined in a **bit pattern**.
- **Pattern mode** (also called **transfer mode**), which specifies how the pen pattern interacts with white or any existing drawing that the pattern overlays, and which is stored in the `pnMode` field of the graphics port.
- Pen **visibility**, 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 following QuickDraw routines relate to the graphics pen:

Routine	Description
`MoveTo` `Move`	Change the pen's location. The graphics pen can be located anywhere on the local coordinate plane of the graphics port.
`GetPen`	Determine the pen's current location.
`PenPat`	Change the pen's bit pattern (see below).
`PenMode`	Change the pen's pattern mode. (A pattern mode determines how the pen's bit pattern interacts with the existing bit image according to one of eight Boolean operations.)
`GetPenState`	Determine the size, location, pattern and pattern mode of the graphics pen. Returns a `PenState` record.
`SetPenState`	Restore the size, location, pattern and pattern mode retrieved by `GetPenState` after temporarily changing those characteristics.

Bit Patterns

As previously stated, one characteristic of the graphics pen is the pen pattern, which is defined in a bit-pattern. A bit-pattern is a 64-pixel image, organised as an 8-by-8 pixel square, which defines a repeating design. The patterns defined in a bit pattern are usually black and white, although any two of basic QuickDraw's eight colours can be used on a colour screen. Bit patterns are defined in data structures of type `Pattern`.

Patterns were originally defined as:

```
   Pattern = packed array [0..7] of 0..255;
```

With the introduction of the Universal Headers, the definition was changed to:

```
 Pattern = record
   pat:    packed array [0..7] of SInt8;
 end;
```

The old array definition of Pattern would cause 68000 based CPU's to crash in certain circumstances. The new definition may require changes to older source code in order to compile.

You can use bit patterns to draw lines and shapes. So that adjacent areas of the same pattern form a continuous coordinated pattern, all patterns are drawn relative to the origin of the graphics port.

Five bit patterns are predefined as QuickDraw global variables (see Fig 7). The pattern `white` is the default pattern for graphics ports.

FIG 7 - RECTANGLES DRAWN USING THE FIVE BIT PATTERNS PREDEFINED AS GLOBAL VARIABLES

Other Bit Patterns

You can create your own bit patterns in your program code, but it is usually simpler and more convenient to store them in resources of type `'PAT '` or `'PAT#'`. You can use `GetPattern` and `GetIndPattern` to access bit patterns stored as system resources.

The five predefined patterns are available not only through the global variables provided by QuickDraw but also as system resources stored in the system resource file. A total of 38 bit patterns, including the five basic patterns, are stored in the system resource file. Some are shown at Fig 8

FIG 8 - RECTANGLES DRAWN USING OTHER BIT PATTERNS IN THE SYSTEM RESOURCE FILE

Boolean Transfer Modes With 1-Bit Pixels

Another characteristic of the graphics pen is the transfer mode. **Boolean transfer modes**, which apply to the one-bit pixels in the black-and-white drawing environment, describe an interaction between the pixels that your application draws and the pixels that are already in the destination bitmap.

Note that these modes apply to the process of copying bits from one graphics port to another as well as drawing with the graphics pen. Black-and-white drawing thus uses two types of Boolean transfer modes:

- **Pattern Modes**. Pattern modes apply to drawing with the graphics pen. The `penMode` field of a graphics port stores the pattern mode for the graphics pen.
- **Source Modes**. You use the source modes when using `CopyBits` (see below) to copy a bit image from one graphics port to another, and also when drawing text. (The source mode for text is stored in the `textMode` field of graphics port.

For both pattern and source modes, there are four Boolean operations: COPY, OR, XOR, and BIC (for bit clear). Each of these operations has an inverse variant in which the pattern or source is inverted before the transfer, so in fact there are eight operations in all. These eight operations have names defined as constants. Those constants, and the effects of the transfer modes they represent on a one-bit destination pixels, are as follows:

Pattern Mode	Source Mode	Action On Destination Pixel	
		If pattern or source pixel is black	If pattern or source pixel is white
patCopy	srcCopy	Force black	Force white
notPatCopy	notSrcCopy	Force white	Force black
patOr	srcOr	Force black	Leave alone
notPatOr	notSrcOr	Leave alone	Force black
patXor	srcXor	Invert	Leave alone
notPatXor	notSrcXor	Leave alone	Invert
patBic	srcBic	Force white	Leave alone
notPatBic	notSrcBic	Leave alone	Force white

Adding Dithering to Source Modes

You can add dithering to any source mode by adding the following constant, or the value it represents, to the source mode:

```
ditherCopy = 64
```

Dithering primarily applies to colour environments, where it may be used to create additional (pseudo) colours on indexed devices. Dithering also improves images that you shrink while copying them from one graphics port to another, or that you copy from a direct pixel device to an indexed device. In the black-and-white environment, using dithering when shrinking 1-bit images between basic graphics ports can produce much better representations of the original images.

Drawing Lines, Rectangles, Ovals, Arcs and Wedges

By starting at a particular position and moving the graphics pen, you can use QuickDraw routines to define and directly draw a number of graphics shapes using the size and pattern of the graphics pen. The following describes how various graphics shapes are drawn with the graphics pen.

Lines

Using QuickDraw routines, you can draw lines onscreen using the size, pattern and pattern mode of the graphics pen for the current graphics port. A **line** is defined by two points: the current location of the graphics pen and its destination. The pen "hangs" below and to the right of the defining points, as shown at Fig 9.

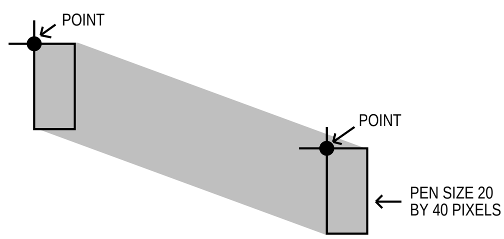

FIG 9 - A LINE DRAWN WITH A BIT PATTERN

Rectangles

To give a rectangle a shape that can be drawn on the screen, you must use QuickDraw rectangle drawing routines, all of which take a `Rect` as a parameter. All drawing by these routines is contained within the rectangle defined by the `Rect` parameter. Fig 10 shows a rectangle drawn with the QuickDraw routine `FrameRect` using the same graphics pen used at Fig 9. (Note that the black line representing the rectangle defined in the `Rect` parameter used by `FrameRect` is shown for illustrative purposes only.)

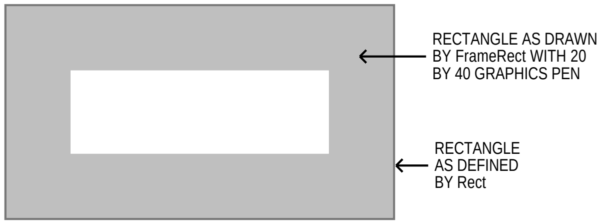

FIG 10 - A RECTANGLE DRAWN BY THE FrameRect PROCEDURE

Bounding Rectangles

You use rectangles known as **bounding rectangles** to define the outermost limits of other shapes, such as rounded rectangles, ovals, arcs, and wedges. Bounding rectangles completely enclose the shapes they bound, that is, no pixels extend outside the infinitely thin lines of the bounding rectangle.

Rounded Rectangles

A **rounded rectangle** is a rectangle with rounded corners. The figure is defined by a bounding rectangle, along with the width and height of the ovals forming the corners (called the **diameters of curvature**).

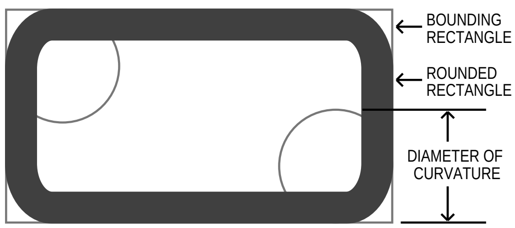

FIG 11 - A ROUNDED RECTANGLE

The corner width and corner height are limited to the width and height of the rectangle itself. If they are longer, the rounded rectangle becomes an oval. Fig 11 shows a rounded rectangle drawn with the QuickDraw routine `FrameRoundRect`.

Ovals, Arcs and Wedges

Ovals. An **oval** is a circular or elliptical shape defined by the bounding rectangle that encloses it.

Arcs and Wedges. An **arc** is a portion of the circumference of an oval bounded by a pair or radii joining at the oval's centre. An arc does not include the bounding radii or any part of the oval's interior. A **wedge** is a pie-shaped segment of an oval bounded by a pair of radii joining at the oval's centre. A wedge includes part of the oval's interior. Arcs and 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 12 shows an arc (drawn using the QuickDraw routine `FrameArc`) and a wedge (drawn using the QuickDraw routine `PaintArc`).

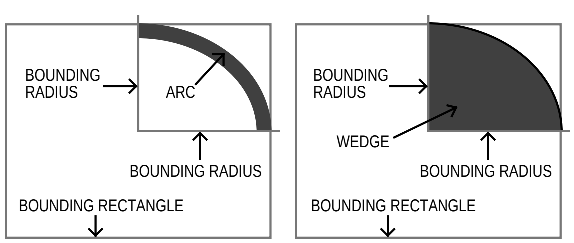

FIG 12 - AN ARC AND A WEDGE

Drawing Polygons, Regions and Pictures

Three types of graphics objects — polygons, regions and pictures — require you to call several routines to create and draw them. You begin by calling a routine that collects drawing commands into a definition for the object. You then use a series of drawing routines to define the object before calling a routine which signals the end of the object definition. Finally, you use a routine which draws your newly-defined object.

Polygons

You use lines to define a polygon. First, however, you must 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 using one of the framing, painting, filling, erasing or inverting routines for polygons (see below).

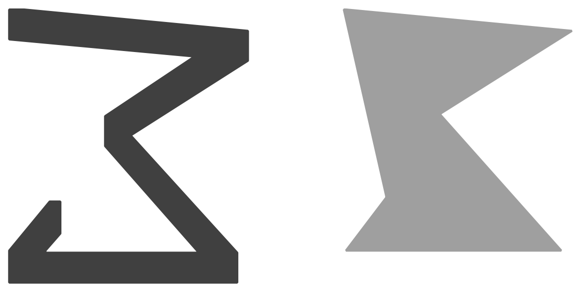

FIG 13 - DRAWING A POLYGON

Fig 13 shows the same polygon drawn with `FramePoly` (on the left) and `FillPoly` (on the right). In this particular polygon, the final defining line from the last vertex back to the first vertex was not drawn. In this situation, `FillPoly`, in effect, completes the polygon, whereas `FramePoly` does not. Note also that, as in line drawing, `FramePoly` hangs the pen down and to the right of the infinitely thin lines that define the polygon.

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 region using one of the framing, painting, filling, erasing or inverting routines for regions (see below).

Fig 14 shows a region comprising two rectangles and an overlapping oval, drawn using `PaintRgn`. Note that, where two figures overlap, the additional area is added to the region and the overlap is removed from the region.

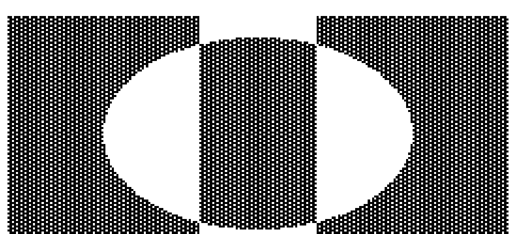

FIG 14 - DRAWING A REGION

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 15 shows an example of a simple picture containing a rectangle and an oval.

FIG 15 - A SIMPLE QUICKDRAW PICTURE

The subject of pictures is addressed in more detail at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.

Routines for Drawing Lines

You specify where to begin drawing a line by using `MoveTo` or `Move` to place the graphics pen at some point in the window's local coordinate system. You then call `LineTo` or `Line` to draw the line from there to another point. `MoveTo` and `LineTo` require you to specify a point in the local coordinate system of the current graphics port. `Move` and `Line` require relative horizontal and vertical distances.

Routines for Drawing Shapes — Framing, Painting, Filling, Erasing, and Inverting

QuickDraw routines for drawing shapes may be divided into five groups as follows:

- **Framing.** Framing a shape draws its outline only, using the current pen size, pen pattern, and pattern mode. The interior of the shape in unaffected.
- **Painting and Filling.** Painting a shape fills both its outline and its interior with the current pen pattern. Filling a shape fills both its outline and its interior with the pattern specified in the `fillPat` field of the basic graphics port.
- **Erasing.** Erasing a shape fills both its outline and its interior with the current background pattern, that is, the pattern specified in the `bkPat` field of the basic graphics port
- **Inverting.** Inverting a shape reverses the colours of all pixels within its boundary. On a black-and-white monitor, all the black pixels become white and vice versa.

The following lists the available framing, painting, filling and erasing routines:

Frame	Paint & Fill	Erase	Invert	Shape Drawn/Erased/Inverted
`FrameRect`	`PaintRect` `FillRect`	`EraseRect`	`InvertRect`	A rectangle. Position and size are defined by a `Rect` structure.
`FrameOval`	`PaintOval` `FillOval`	`EraseOval`	`InvertOval`	An oval. Position and size are determined by a bounding rectangle specified by a `Rect` structure.
`FrameRoundRect`	`PaintRoundRect` `FillRoundRect`	`EraseRoundRect`	`InvertRoundRect`	A rounded rectangle. Position and size are determined by a bounding rectangle specified by a `Rect` structure. Curvature of the corners is defined by `ovalWidth` and `ovalHeight` parameters.
`FrameArc`	`PaintArc` `FillArc`	`EraseArc`	`InvertArc`	An arc. Position and size are determined by a bounding rectangle specified by a `Rect` structure. Starting point and arc extent are determined by `startAngle` and `arcAngle` parameters.

`FramePoly`	`PaintPoly` `FillPoly`	`ErasePoly`	`InvertPoly`	A polygon. Draws the polygon by "playing back" all the line drawing calls that define it.
`FrameRgn`	`PaintRgn` `FillRgn`	`EraseRgn`	`InvertRgn`	As defined by the specified region.

Drawing Text

On the Macintosh, text is just another form of graphics, as is evidenced by the basic graphics port text-related fields `txFont`, `txFace`, `TxSize`, `txMode`, and `spExtra`. QuickDraw routines 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 `TextFont`. `TextFont` takes a single parameter, of type `short`, which may be either a predefined constant or a **font family ID** number. The predefined constants[3] are as follows:

```
systemFont = 0  { System font (Chicago).  Used to draw text in menus, dialog boxes,}
                {etc.  The Chicago font family ID is 0.}
applFont   = 1  {Default application font (Geneva). Suggested default font for use by}
                {applications which do not support user selection of fonts.}
newYork    = 2
geneva     = 3
monaco     = 4
venice     = 5
london     = 6
athens     = 7
sanFran    = 8
toronto    = 9
cairo      = 11
losAngeles = 12
times      = 20
helvetica  = 21
courier    = 22
symbol     = 23
mobile     = 24
```

For fonts not represented by these predefined constants, if you know the font name, you can get the font family ID[4] using `GetFNum`.[5] For example, the following sets the current font to Palatino:

```
fontNum: integer;

GetFNum("Palatino", fontNum);
TextFont(fontNum);
```

Note that the system font and the application font have **special font designators**. The system font's special font designator is 0 and the application font's special font designator is 1. These special designators are not actual font family (resource) ID numbers and cannot be used as such in Resource Manager calls; however, they can be used in place of the font family ID in the `txFont` field of the graphics port and in text-related calls that take a font family ID. The system maps the special designators to the actual font family IDs.

Do not use the font family ID of 0 to specify the Chicago font because the ID can vary on localised systems. To specify the Chicago font, follow the same procedure as in the example for Palatino, above.

[3] The predefined constants should be used with caution, since most of the fonts they represent have become obsolete.

[4] Fonts are resources, and the font family ID is a resource ID.

[5] If you know the font family ID, you can get its name by calling the Font Manager's `GetFontName` procedure. 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 `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 `TextSize` to change the font size in typographical **points**. A point is approximately 1/72 inch, which is very close to the size of a screen pixel.

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 routines.

Specifying the Transfer Mode

The transfer mode may be set using `TextMode`. By default, the transfer mode is set to `srcOr`, which causes drawn text to overlay the existing graphics. This mode produces the best results for drawing text because it writes only those bits which make up the actual glyph.[6]

While all of the transfer modes apply to the drawing of text, you should generally use either `srcOr` or `srcBic` when drawing text, because all other transfer modes can result in the clipping of glyphs by adjacent glyphs.

The `grayishTextOr` Text Transfer Mode. The non-standard text drawing transfer mode`grayishTextOr` is useful for displaying disabled user interface items.[7] This mode produces a dithered black and white glyph on a black and white destination device.

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 (usually 32-by-32 or 16-by-16 pixels) which represents an object, concept, or message. Icons are stored as resources.

Cursors and Icons are addressed at Chapter 12 — Offscreen Graphics Worlds, Pictures, Cursors, and Icons.)

Manipulating Rectangles and Regions

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

---

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

[7]The `grayishTextOr` mode is considered non-standard because it is not stored in pictures and printing with it is undefined.

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 routines for manipulating, and performing calculations on, rectangles:

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

Manipulating Regions

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

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

Manipulating Polygons

Note that, while you can use `OffSetPoly` to move a polygon, QuickDraw provides no other routines for calculating or 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 routines `ScalePt`, `MapPt`, `MapRect`, `MapRgn`, and `MapPoly`.

Copying Bits Between Graphics Ports

QuickDraw provides the following three primary image-processing routines:

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

When copying images between basic graphics ports using `CopyBits`, you specify a source bitmap and a destination bitmap. If you specify different sized source and destination rectangles, `CopyBits` scales the source image to fit the destination. The manner by which `CopyBits` transfers the bits between bitmaps depends on the source mode that you specify in the `CopyBits` call.

To copy only certain bits from a bitmap, you can use `CopyMask`, which is a specialised variant of `CopyBits`. `CopyMask` transfers bits only where the corresponding bits of another bit image, which serves as a mask, are set to 1 (that is, black). Note that `CopyMask`, unlike `CopyDeepMask`, does not allow scaling or resizing.

Use of Offscreen Graphics Worlds

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 12 — 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.

Main Basic QuickDraw Constants, Data Types and Routines

Constants

Basic QuickDraw Colours

```
whiteColor      = 30
blackColor      = 33
yellowColor     = 69
magentaColor    = 137
redColor        = 205
cyanColor       = 273
greenColor      = 341
blueColor       = 409
```

Pattern Modes

```
patCopy         = 8
patOr           = 9
patXor          = 10
```

```
patBic          = 11
notPatCopy      = 12
notPatOr        = 13
notPatXor       = 14
notPatBic       = 15
```

Source Modes

```
srcCopy         = 0
srcOr           = 1
srcXor          = 2
srcBic          = 3
notSrcCopy      = 4
notSrcOr        = 5
notSrcXor       = 6
notSrcBic       = 7
ditherCopy      = 64
```

Special Text Transfer Mode

```
grayishTextOr = 49;
```

Pattern List Resource ID for Patterns in the System File

```
sysPatListID= 0;
```

Data Types

Pattern

```
Pattern = record
  pat:     packed array [0..7] of SInt8;
  end;

PatPtr = ^Pattern;
PatHandle = ^PatPtr;
```

Patterns were originally defined as:

```
Pattern = packed array [0..7] of 0..255;
```

The old array definition of Pattern would cause 68000 based CPU's to crash in certain circumstances. The new definition may require changes to older source code in order to compile.

Point

```
Point = record
  CASE integer of
  0: (
    v:  integer;   {vertical coordinate.}
    h:  integer;   {horizontal coordinate.}
     );
  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:   integer;    {size in bytes}
  rgnBBox:   Rect;       {enclosing rectangle}
  …          {More data if region is not rectangular.}
  end;

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

GrafPort

```
GrafPort = record
  device:                  integer;  {Device-specific information. (0 = screen.)}
  portBits:                BitMap;   {BitMap.}
  portRect:                Rect;   {Port Rectangle.}
  visRgn:                  RgnHandle;  {Visible region.}
  clipRgn:                 RgnHandle;  {Clipping region.}
  bkPat:                   Pattern;  {Background pattern.}
  fillPat:                 Pattern;  {Fill pattern.}
  pnLoc:                   Point;  {Pen location.}
  pnSize:                  Point;  {Pen size.}
  pnMode:                  integer;  {Pen mode.}
  pnPat:                   Pattern;  {Pen pattern.}
  pnVis:                   integer;  {Pen visibility.}
  txFont:                  integer;  {Font number for text.}
  txFace:                  Style;  {Text's font style.}
  txMode:                  integer;  {Transfer mode for text.}
  txSize:                  integer;  {Font size for text.}
  spExtra:                 Fixed;  {Spacing for full justification.}
  fgColor:                 longint;  {Foreground colour.}
  bkColor:                 longint;  {Background colour.}
  colrBit:                 integer;  {Colour bit}
  patStretch:              integer;  {(Used internally.)}
  picSave:                 Handle;   {Picture being saved. (Used internally.)}
  rgnSave:                 Handle;   {Region being saved. (Used internally.)}
  polySave:                Handle;   {Polygon being saved. (Used internally.)}
  grafProcs:               QDProcsPtr; {Low-level drawing routines.}
  end;
GrafPtr = ^GrafPort;
WindowPtr = GrafPtr;
```

BitMap

```
BitMap = record
  baseAddr:  Ptr;        {Pointer to bit image.}
  rowBytes:  integer;    {Row width.}
  bounds:    Rect;       {Boundary rectangle.}
  end;

BitMapPtr = ^BitMap;
BitMapHandle = ^BitMapPtr;
```

Polygon

```
Polygon = record
  polySize:  integer;
  polyBBox:  Rect;
  polyPoints:array [0..0] of Point;
  end;

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

PenState

```
PenState = record
  pnLoc:     Point;
  pnSize:    Point;
  pnMode:    integer;
  pnPat:     Pattern;
  end;
```

Routines

Initialising QuickDraw

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

Opening and Closing Basic Graphics Ports

```
procedure  OpenPort(port: GrafPtr);
procedure  InitPort(port: GrafPtr);
procedure  ClosePort(port: GrafPtr);
```

Saving and Restoring Graphics Ports

```
procedure  SetPort(port: GrafPtr);
procedure  GetPort(var port: GrafPtr);
```

Managing BitMaps, Port Rectangles and Clipping Regions

```
procedure  ScrollRect(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(var r: Rect);
function   BitMapToRegionGlue(region: RgnHandle; var bMap: BitMap): OSErr;
procedure SetPortBits(var bm: BitMap);
```

Manipulating Points in Graphics Ports

```
procedure  LocalToGlobal(var pt: Point);
procedure  GlobalToLocal(var pt: Point); void  AddPt(Point src,Point *dst);
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;
```

Managing the Graphics Pen

```
procedure  HidePen;
procedure  ShowPen;
procedure  GetPen(var pt: Point);
procedure  GetPenState(var pnState: PenState);
procedure  SetPenState(var pnState: PenState);
procedure  PenSize(width: integer; height: integer); void  HidePen(void);
procedure  PenMode(mode: integer);
procedure  PenPat(var pat: Pattern);
procedure  PenNormal;
```

Changing the BackGround Bit Pattern

```
procedure  BackPat(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);
```

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(var src1: Rect; var src2: Rect; var dstRect: Rect): boolean;
procedure  UnionRect(var src1: Rect; var src2: Rect; var dstRect: Rect);
function   PtInRect(pt: Point; var r: Rect): boolean;
procedure  Pt2Rect(pt1: Point; pt2: Point; var dstRect: Rect);
procedure  PtToAngle(var r: Rect; pt: Point; var angle: integer);
function   EqualRect(var rect1: Rect; var rect2: Rect): boolean;
function   EmptyRect(var r: Rect): boolean;
```

Drawing Rectangles

```
procedure  FrameRect(var r: Rect);
procedure  PaintRect(var r: Rect);
procedure  EraseRect(var r: Rect);
procedure  InvertRect(var r: Rect);
procedure  FillRect(var r: Rect; var pat: Pattern);
```

Drawing Rounded Rectangles

```
procedure  FrameRoundRect(var r: Rect; ovalWidth: integer; ovalHeight: integer);
procedure  PaintRoundRect(var r: Rect; ovalWidth: integer; ovalHeight: integer);
procedure  EraseRoundRect(var r: Rect; ovalWidth: integer; ovalHeight: integer);
procedure  InvertRoundRect(var r: Rect; ovalWidth: integer; ovalHeight: integer);
procedure  FillRoundRect(var r: Rect; ovalWidth: integer; ovalHeight: integer;
           var pat: Pattern);
```

Drawing Ovals

```
procedure  PaintOval(var r: Rect);
procedure  EraseOval(var r: Rect);
procedure  InvertOval(var r: Rect);
procedure  FillOval(var r: Rect; var pat: Pattern);
```

Drawing Arcs and Wedges

```
procedure  FrameArc(var r: Rect; startAngle: integer; arcAngle: integer);
procedure  PaintArc(var r: Rect; startAngle: integer; arcAngle: integer);
procedure  EraseArc(var r: Rect; startAngle: integer; arcAngle: integer);
procedure  InvertArc(var r: Rect; startAngle: integer; arcAngle: integer);
procedure  FillArc(var r: Rect; startAngle: integer; arcAngle: integer; var pat: Pattern);
```

Creating and Managing Polygons

```
function   OpenPoly: PolyHandle;
procedure  ClosePoly;
procedure  KillPoly(poly: PolyHandle);
procedure  OffsetPoly(poly: PolyHandle; dh: integer; dv: integer);
```

Drawing and Painting Polygons

```
procedure  FramePoly(poly: PolyHandle);
procedure  PaintPoly(poly: PolyHandle);
procedure  ErasePoly(poly: PolyHandle);
procedure  InvertPoly(poly: PolyHandle);
procedure  FillPoly(poly: PolyHandle; var pat: Pattern); PolyHandle  OpenPoly(void);
```

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; 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); void
           DisposeRgn(RgnHandle rgn);
procedure  XorRgn(srcRgnA: RgnHandle; srcRgnB: RgnHandle; dstRgn: RgnHandle);
function   PtInRgn(pt: Point; rgn: RgnHandle): boolean;
function   RectInRgn(var r: Rect; rgn: RgnHandle): boolean;
function   EqualRgn(rgnA: RgnHandle; rgnB: RgnHandle): boolean;
function   EmptyRgn(rgn: RgnHandle): boolean;
function   BitMapToRegion(region: RgnHandle; var bMap: BitMap): OSErr;
```

Drawing Regions

```
procedure  FrameRgn(rgn: RgnHandle);
procedure  PaintRgn(rgn: RgnHandle);
procedure  EraseRgn(rgn: RgnHandle);
procedure  InvertRgn(rgn: RgnHandle);
procedure  FillRgn(rgn: RgnHandle; var pat: Pattern);
```

Setting Text Characteristics

```
procedure  TextFont(font: integer);
procedure  TextFace(face: Style);
procedure  TextMode(mode: integer);
procedure  TextSize(size: integer);
procedure  SpaceExtra(extra: Fixed);
procedure  GetFontInfo(var info: FontInfo);
```

Drawing Text

```
procedure  DrawChar(ch: char);
procedure  DrawString(s: ConstStr255Param);
procedure  DrawText(textBuf: UNIV Ptr; firstByte: integer; byteCount: integer);
```

Measuring Text

```
function   CharWidth(ch: char): integer;
function   StringWidth(s: ConstStr255Param): integer;
function   TextWidth(textBuf: UNIV Ptr; firstByte: integer; byteCount: integer): integer;
```

Scaling and Mapping Points, Rectangles, Polygons, and Regions

```
procedure  ScalePt(var pt: Point; var srcRect: Rect; var dstRect: Rect);
procedure  MapPt(var pt: Point; var srcRect: Rect; var dstRect: Rect);
procedure  MapRect(var r: Rect; var srcRect: Rect; var dstRect: Rect);
procedure  MapRgn(rgn: RgnHandle; var srcRect: Rect; var dstRect: Rect);
procedure  MapPoly(poly: PolyHandle; var srcRect: Rect; var dstRect: Rect);
```

Copying Images

```
procedure  CopyBits(var srcBits: BitMap; var dstBits: BitMap; var srcRect: Rect;
           var dstRect: Rect; mode: integer; maskRgn: RgnHandle);
procedure  CopyMask(var srcBits: BitMap; var maskBits: BitMap;
           var dstBits: BitMap; var srcRect: Rect; var maskRect: Rect;
           var dstRect: Rect);
procedure  CopyDeepMask(var srcBits: BitMap; var maskBits: BitMap; var dstBits: BitMap;
           var srcRect: Rect; var maskRect: Rect; var dstRect: Rect; mode: integer;
           maskRgn: RgnHandle);
```

Drawing With the Eight-Color System

```
procedure  ForeColor(color: longint);
procedure  BackColor(color: longint);
procedure  ColorBit(whichBit: integer);
```

Determining Whether QuickDraw has Finished Drawing

```
function   QDDone(port: GrafPtr): boolean;
```

Getting Pattern Resources

```
function   GetPattern(patternID: integer): PatHandle;
procedure  GetIndPattern(var thePat: Pattern; patternListID: integer; index: integer);
```

Demonstration Program

```
{ ######################################################################################
// BasicQuickdrawPascal.p
// ######################################################################################
//
// This program:
//
// •  Opens a window in which the results of various basic QuickDraw drawing operations
//    are displayed.
//
//    Individual drawing operations (eg, draw lines, draw rectangles, draw polygons, etc)
//    are selected from a pull-down menu titled 'Demonstration'.
//
// •  Quits when the user selects Quit from the File menu or clicks the window's close
//    box.
//
// The program utilizes 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 transfer modes demonstration.
//
// •  A 'PICT' resource (purgeable) used for the CopyBits demonstration.
//
// •  'STR#' resources (purgeable) containing strings used by the CopyBits and text
//    demonstrations.
//
// ###################################################################################### }

program BasicQuickdrawPascal(input, output);

{ ………………………………………………………………………………………… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, Icons, Segload, Sound;

{ …………………………………………………………………………………………………………………… define the following constants }

const

mApple = 128;
mFile = 129;
  iQuit = 11;
mDemonstration = 131;
  iLine = 1;
  iRectAndOval = 2;
  iArcAndWedge = 3;
  iPolygon = 4;
  iRegion = 5;
  iTransferMode = 6;
  iCopyBits = 7;
  iText = 8;
  iBasicColour = 9;
  iDrawWithMouse = 10;
rMenubar = 128;
rWindow = 128;
rSmallWindow = 129;
rCrossIcon = 128;
rSquareIcon = 129;
rModeStringList = 128;
rTextStringList = 129;
rPicture = 128;
kMaxLong = $7FFFFFFF;

{ ……………………………………………………………………………………………………………………………………………… global variables }

var

gDone : boolean;
gWindowPtr : WindowPtr;
```

```
gDrawWithMouseActivated: boolean;
menubarHdl : Handle;
menuHdl : MenuHandle;
eventRec : EventRecord;
gotEvent : boolean;

{ ####################################################################### DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ####################################################################### DoRandomNumber }

function DoRandomNumber(range : integer) : integer;

  begin
  DoRandomNumber := (Abs(Random) mod (range + 1));
  end;
    {of function DoRandomNumber}

{ ########################################################################### DoRectOval }

procedure DoRectOval;

  var
  theRect : Rect;
  finalTicks : UInt32;
  systemPattern : Pattern;

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

  PenPat(qd.black);
  PenSize(10, 20);
  PenMode(patCopy);

  SetRect(theRect, 10, 20, 245, 130);

  MoveTo(10, 15);
  DrawString('FrameRect');
  FrameRect(theRect);
  Delay(30, finalTicks);

  MoveTo(255, 15);
  DrawString('PaintRect');
  OffsetRect(theRect, 245, 0);
  PenPat(qd.ltGray);
  PaintRect(theRect);
  Delay(30, finalTicks);

  MoveTo(10, 154);
  DrawString('FillRoundRect');
  OffsetRect(theRect, -245, 140);
  GetIndPattern(systemPattern, sysPatListID, 12);
  FillRoundRect(theRect, 120, 60, systemPattern);
  Delay(30, finalTicks);

  MoveTo(255, 154);
  DrawString('FrameOval');
  OffsetRect(theRect, 245, 0);
  PenSize(40, 20);
  PenPat(qd.dkGray);
```

```
  FrameOval(theRect);

  SetWTitle(gWindowPtr, 'Click Mouse For Invert and Erase');
  while not (Button) do ;
  SetWTitle(gWindowPtr, 'Basic QuickDraw');
  SetRect(theRect, 10, 145, 490, 154);
  EraseRect(theRect);
  SetRect(theRect, 255, 160, 490, 270);
  Delay(30, finalTicks);

  MoveTo(10, 154);
  DrawString('InvertRoundRect');
  OffsetRect(theRect, -245, 0);
  InvertRoundRect(theRect, 120, 60);
  Delay(30, finalTicks);

  MoveTo(255, 154);
  DrawString('EraseOval');
  OffsetRect(theRect, 245, 0);
  EraseOval(theRect);
  Delay(30, finalTicks);
  end;

{ ########################################################################### DoArcWedge }

procedure DoArcWedge;

  var
  theRect : Rect;
  a : integer;
  finalTicks : UInt32;
  systemPattern : Pattern;

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

  PenSize(60, 10);
  PenPat(qd.dkGray);
  PenMode(patCopy);

  SetRect(theRect, 10, 20, 245, 278);

  MoveTo(10, 15);
  DrawString('FrameArc');
  for a := 0 to 269 do
    FrameArc(theRect, 135, a);
  Delay(30, finalTicks);

  MoveTo(255, 15);
  DrawString('FillArc');
  OffsetRect(theRect, 245, 0);
  GetIndPattern(systemPattern, sysPatListID, 16);
  FillArc(theRect, 315, 270, systemPattern);
  Delay(30, finalTicks);
  OffsetRect(theRect, -30, 0);
  FillArc(theRect, 225, 90, systemPattern);
  end;
    {of procedure DoArcWedge}

{ ############################################################################ DoPolygon }

procedure DoPolygon;

  var
  polygonHdl : PolyHandle;
  finalTicks : UInt32;
  systemPattern : Pattern;

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

  PenSize(10, 30);
  PenPat(qd.gray);
  PenMode(patCopy);

  polygonHdl := OpenPoly;
  MoveTo(10, 20);
```

```
  LineTo(225, 40);
  LineTo(100, 120);
  LineTo(215, 248);
  LineTo(10, 248);
  LineTo(50, 200);
  ClosePoly;

  MoveTo(10, 15);
  DrawString('FramePoly');
  FramePoly(polygonHdl);
  Delay(30, finalTicks);

  MoveTo(265, 15);
  DrawString('FillPoly');
  OffsetPoly(polygonHdl, 255, 0);
  GetIndPattern(systemPattern, sysPatListID, 9);
  FillPoly(polygonHdl, systemPattern);

  KillPoly(polygonHdl);
  end;
    {of procedure DoPoly}

{ ############################################################################# DoRegion }

procedure  DoRegion;

  var
  regionHdl : RgnHandle;
  theRect : Rect;
  finalTicks : UInt32;

  begin
  FillRect(gWindowPtr^.portRect, qd.white);
  PenPat(qd.gray);
  PenMode(patCopy);

  regionHdl := NewRgn;

  OpenRgn;
  SetRect(theRect, 10, 20, 100, 130);
  FrameRect(theRect);
  SetRect(theRect, 155, 20, 245, 130);
  FrameRect(theRect);
  SetRect(theRect, 55, 30, 200, 120);
  FrameOval(theRect);
  CloseRgn(regionHdl);

  MoveTo(10, 15);
  DrawString('FrameRgn');
  PenPat(qd.black);
  PenSize(10, 20);
  FrameRgn(regionHdl);
  Delay(30, finalTicks);

  MoveTo(255, 15);
  DrawString('1. FillRgn');
  OffsetRgn(regionHdl, 245, 0);
  FillRgn(regionHdl, qd.dkGray);
  Delay(30, finalTicks);

  MoveTo(10, 154);
  DrawString('2. InsetRgn (10 horizontal, 10 vertical)');
  OffsetRgn(regionHdl, -245, 140);
  InsetRgn(regionHdl, 10, 10);
  PenPat(qd.dkGray);
  PaintRgn(regionHdl);
  Delay(30, finalTicks);

  MoveTo(255, 154);
  DrawString('3. InsetRgn (-10 horizontal, -10 vertical)');
  OffsetRgn(regionHdl, 245, 0);
  InsetRgn(regionHdl, -10, -10);
  PenPat(qd.dkGray);
  PaintRgn(regionHdl);

  DisposeRgn(regionHdl);
  end;
```

```
    {of procedure DoRegion

{ ####################################################################### DoTransferMode }

procedure  DoTransferMode;

  var
  crossIconHdl, squareIconHdl : Handle;
  destRect : Rect;
  a, b, i, j : integer;
  squareIconMap : BitMap;
  finalTicks : UInt32;
  sourceMode : integer;
  sourceString : string;

  begin
  sourceMode := 0;
  FillRect(gWindowPtr^.portRect, qd.white);

  PenSize(1, 1);
  PenPat(qd.gray);
  PenMode(patOr);

  crossIconHdl := GetIcon(rCrossIcon);
  if (crossIconHdl = nil) then
    begin
    SysBeep(10);
    Exit(DoTransferMode);
    end;

  squareIconHdl := GetIcon(rSquareIcon);
  if (squareIconHdl = nil) then
    begin
    SysBeep(10);
    Exit(DoTransferMode);
    end;

  SetRect(destRect, 120, 8, 190, 78);
  PlotIcon(destRect, crossIconHdl);
  FrameRect(destRect);
  MoveTo(200, 48);
  DrawString('Destination');

  SetRect(destRect, 270, 8, 340, 78);
  PlotIcon(destRect, squareIconHdl);
  FrameRect(destRect);
  MoveTo(350, 48);
  DrawString('Source');

  for i := 0 to 1 do
    begin
    a := i * 100 + 91;
    for j := 0 to 3 do
      begin
      b := j * 120 + 30;
      SetRect(destRect, b, a, b+70, a+70);
      PlotIcon(destRect, crossIconHdl);
      end;
    end;

  HLock(squareIconHdl);

  squareIconMap.baseAddr := squareIconHdl^;
  squareIconMap.rowBytes := 4;
  SetRect(squareIconMap.bounds, 0, 0, 31, 31);

  for i := 0 to 1 do
    begin
    a := i * 100 + 91;
    for j := 0 to 3 do
      begin
      b := j * 120 + 30;
      Delay(30, finalTicks);
      SetRect(destRect, b, a, b+70, a+70);
      CopyBits(squareIconMap, qd.thePort^.portBits, squareIconMap.bounds,
            destRect, sourceMode, nil);
      sourceMode := sourceMode + 1;
```

```
      GetIndString(sourceString, rModeStringList, sourceMode);
      MoveTo(b, a + 82);
      DrawString(sourceString);
      end;
    end;

  HUnlock(squareIconHdl);
  end;
    {of procedure DoTransferMode}

{ ########################################################################### DoCopyBits }

procedure DoCopyBits;

  var
  myWindowPtr : WindowPtr;
  oldPort : GrafPtr;
  pictureHdl : PicHandle;
  sourceRect, destRect : Rect;
  finalTicks : UInt32;

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

  myWindowPtr := GetNewWindow(rSmallWindow, nil, WindowPtr(-1));
  if (myWindowPtr = nil) then
    ExitToShell;

  GetPort(oldPort);
  SetPort(myWindowPtr);

  pictureHdl := GetPicture(rPicture);
  if (pictureHdl = nil) then
    begin
    DisposeWindow(myWindowPtr);
    SysBeep(10);
    Exit(DoCopyBits);
    end;

  HNoPurge(Handle(pictureHdl));
  SetRect(sourceRect, 65, 40, 165, 182);
  DrawPicture(pictureHdl, sourceRect);
  HPurge(Handle(pictureHdl));

  SetWTitle(myWindowPtr, 'Click Mouse for CopyBits');
  while not (Button) do ;

  SetRect(destRect, 20, 21, 210, 272);

  CopyBits(myWindowPtr^.portBits, oldPort^.portBits, sourceRect, destRect,
           srcCopy, nil);

  SetWTitle(myWindowPtr, 'Click Mouse to Close');
  Delay(60, finalTicks);
  while not (Button) do ;

  DisposeWindow(myWindowPtr);
  SetPort(oldPort);
  end;
    {of procedure DoCopyBits}

{ ############################################################################### DoText }

procedure DoText;

  var
  windowCentre, a, fontNum, widthOfString : integer;
  textString : string;
  theWindow : WindowRef;

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

  theWindow := FrontWindow;
  windowCentre := trunc((theWindow^.portRect.right - theWindow^.portRect.left) / 2);

  for a := 1 to 8 do
```

```
    begin
    if (a = 1) then
      begin
      GetFNum('Geneva', fontNum);
      TextFont(fontNum);
      TextFace([]);
      end

    else if (a = 2) then
        TextFace([bold])

    else if (a = 3) then
        begin
        GetFNum('Times', fontNum);
        TextFont(fontNum);
        TextFace([italic]);
        end

    else if (a = 4) then
      TextFace([underline])

    else if (a = 5) then
      begin
      GetFNum('Helvetica', fontNum);
      TextFont(fontNum);
      TextFace([outline]);
      end

    else if (a = 6) then
      TextFace([shadow])

    else if (a = 7) then
      begin
      GetFNum('Chicago', fontNum);
      TextFont(fontNum);
      TextFace([condense]);
      end

    else if (a = 8) then
      begin
      TextFace([extend]);
      TextMode(grayishTextOr);
      end;

    if (a < 7)
      thenTextSize(a * 2 + 10)
      elseTextSize(12);

    GetIndString(textString, rTextStringList, a);
    widthOfString := StringWidth(textString);
    MoveTo(trunc(windowCentre - (widthOfString / 2)), a * 35 - 10);
    DrawString(textString);
    end;
      {of for loop}

  GetFNum('Geneva', fontNum);
  TextFont(fontNum);
  TextSize(10);
  TextMode(srcOr);
  TextFace(Style(0));
  end;
    {of procedure DoText}

{ ####################################################################### DoBasicColours }

procedure DoBasicColours;

  var
  a : integer;
  theRect : Rect;
  finalTicks : UInt32;

  begin
  FillRect(gWindowPtr^.portRect, qd.dkGray);
  PenPat(qd.black);
  PenMode(patCopy);
```

```
  for a := 1 to 8 do
    begin
    Delay(30, finalTicks);
    if (a = 1) then ForeColor(blackColor);
    if (a = 2) then ForeColor(whiteColor);
    if (a = 3) then ForeColor(redColor);
    if (a = 4) then ForeColor(greenColor);
    if (a = 5) then ForeColor(blueColor);
    if (a = 6) then ForeColor(cyanColor);
    if (a = 7) then ForeColor(magentaColor);
    if (a = 8) then ForeColor(yellowColor);

    SetRect(theRect, 35, a*28, 465, a*28+23);
    PaintRect(theRect);
    end;

  ForeColor(blackColor);
  end;
    {of procedure DoBasicColours}

{ ############################################################################## DoLines }

procedure DoLines;

  var
  top, left, bottom, right, a, b, c : integer;
  oldClipRgn : RgnHandle;
  newClipRect : Rect;
  systemPattern : Pattern;
  finalTicks : UInt32;

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

  PenMode(patCopy);

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

  oldClipRgn := NewRgn;
  GetClip(oldClipRgn);
  SetRect(newClipRect, left, top, right, bottom);
  ClipRect(newClipRect);

  for a := 1 to 38 do
    begin
    b := DoRandomNumber(gWindowPtr^.portRect.right - gWindowPtr^.portRect.left);
    c := DoRandomNumber(gWindowPtr^.portRect.right - gWindowPtr^.portRect.left);

    GetIndPattern(systemPattern, sysPatListID, a);
    PenPat(systemPattern);
    PenSize(a * 2, 1);

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

    Delay(15, finalTicks);
    end;

  SetClip(oldClipRgn);
  DisposeRgn(oldClipRgn);

  SetWTitle(gWindowPtr, 'Click Mouse for More Lines');
  while not (Button) do ;
  SetWTitle(gWindowPtr, 'Basic QuickDraw');

  FillRect(gWindowPtr^.portRect, qd.white);
  PenSize(1, 1);
  PenPat(qd.black);
  PenMode(patXor);

  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}

{ ###################################################################### DoDrawWithMouse }

procedure DoDrawWithMouse;

  var
  mouseDownMouse, previousMouse, currentMouse : Point;
  drawRect : Rect;
  thePattern : Pattern;

  begin
  PenSize(1, 1);
  PenPat(qd.gray);
  PenMode(patXor);

  GetMouse(mouseDownMouse);

  drawRect.left := mouseDownMouse.h;
  drawRect.right := mouseDownMouse.h;
  drawRect.top := mouseDownMouse.v;
  drawRect.bottom := mouseDownMouse.v;

  GetMouse(previousMouse);

  while StillDown do
    begin
    GetMouse(currentMouse);

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

      drawRect.right := currentMouse.h;
      drawRect.bottom := currentMouse.v;

      FrameRect(drawRect);
      end;

    previousMouse.v := currentMouse.v;
    previousMouse.h := currentMouse.h;
    end;

  FrameRect(drawRect);

  PenMode(patCopy);

  PenSize(2, 2);
  PenPat(qd.black);
  ForeColor(redColor);
  FrameRect(drawRect);

  InsetRect(drawRect, 10, 10);
  PenSize(8, 8);
  GetIndPattern(thePattern ,0, 5);
  PenPat(thePattern);
  ForeColor(blueColor);
  FrameRoundRect(drawRect, 40, 40);

  InsetRect(drawRect, 16, 16);
  PenSize(14, 14);
  GetIndPattern(thePattern,0, 6);
  PenPat(thePattern);
  ForeColor(greenColor);
  PaintOval(drawRect);
```

```
  PenMode(patCopy);
  ForeColor(blackColor);

  end;
    {of procedure DoDrawWithMouse}

{ ################################################################ DoDemonstrationMenu }

procedure DoDemonstrationMenu(menuItem : integer);

  begin

  gDrawWithMouseActivated := false;

  case (menuItem) of

    iLine:
      begin
      DoLines;
      end;

    iRectAndOval:
      begin
      DoRectOval;
      end;

    iArcAndWedge:
      begin
      DoArcWedge;
      end;

    iPolygon:
      begin
      DoPolygon;
      end;

    iRegion:
      begin
      DoRegion;
      end;

    iTransferMode:
      begin
      DoTransferMode;
      end;

    iCopyBits:
      begin
      DoCopyBits;
      end;

    iText:
      begin
      DoText;
      end;

    iBasicColour:
      begin
      DoBasicColours;
      end;

    iDrawWithMouse:
      begin
      FillRect(gWindowPtr^.portRect, qd.white);
      MoveTo(10, 25);
      DrawString('Click in the window and drag the mouse to the right and down');
      gDrawWithMouseActivated := true;
      end;

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

{ ####################################################################### DoMenuChoice }

```

```
procedure DoMenuChoice(menuChoice : longint);

  var
  menuID, menuItem : integer;
  itemName : string;
  daDriverRefNum : integer;

  begin
  menuID := HiWord(menuChoice);
  menuItem := LoWord(menuChoice);

  if (menuID = 0) then
    Exit(DoMenuChoice);

  case (menuID) of

    mApple:
      begin
      GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
      daDriverRefNum := OpenDeskAcc(itemName);
      end;

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

    mDemonstration:
      begin
      DoDemonstrationMenu(menuItem);
      end;

    end;
      {of case statement}

  HiliteMenu(0);
  end;
    {of procedure DoMenuChoice}

{ ########################################################################## DoMouseDown }

procedure DoMouseDown(var theEvent : EventRecord);

  var
  myWindowPtr : WindowPtr;
  partCode : integer;

  begin
  partCode := FindWindow(theEvent.where, myWindowPtr);

  case (partCode) of

    inMenuBar:
      begin
      DoMenuChoice(MenuSelect(theEvent.where));
      end;

    inSysWindow:
      begin
      SystemClick(theEvent, myWindowPtr);
      end;

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

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

    inGoAway:
```

```
      begin
      if (TrackGoAway(myWindowPtr, theEvent.where)) then
        gDone := true;
      end;

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

{ ############################################################################## DoEvents }

procedure DoEvents(var theEvent : EventRecord);

  var
  myWindowPtr : WindowPtr;
  charCode : char;

  begin
  myWindowPtr := WindowPtr(theEvent.message);

  case (theEvent.what) of

    mouseDown:
      begin
      DoMouseDown(theEvent);
      end;

    keyDown, autoKey:
      begin
      charCode := chr(BAnd(theEvent.message, charCodeMask));
      if (BAnd(theEvent.modifiers, cmdKey) <> 0) then
        DoMenuChoice(MenuKey(charCode));
      end;

    updateEvt:
      begin
      BeginUpdate(myWindowPtr);
      EndUpdate(myWindowPtr);
      end;
    end;
      {of case statement}
  end;
    {of procedure DoEvents}

{ ################################################################ start of main program }

begin

  { ………………………………………………………………………………………………………………………………… initialize managers }

  DoInitManagers;

  { ……………………………………………………………………………………………………………………… set random number generator }

  GetDateTime(UInt32(qd.randSeed));

  { ………………………………………………………………………………………………………………………… set up menu bar and menus }

  menubarHdl := GetNewMBar(rMenubar);
  if (menubarHdl = nil) then
    ExitToShell;
  SetMenuBar(menubarHdl);
  DrawMenuBar;

  menuHdl := GetMenuHandle(mApple);
  if (menuHdl = nil)
    thenExitToShell
    elseAppendResMenu(menuHdl, 'DRVR');

  { …………………………………………………………………………………………………………………………………………… open window }

  gWindowPtr := GetNewWindow(rWindow, nil, WindowPtr(-1));
  if (gWindowPtr = nil) then
    ExitToShell;

  SetPort(gWindowPtr);
```

```
922    TextSize(10);
923
924    { ........................................................................................ eventLoop }
925
926    gDone := false;
927
928    while not (gDone) do
929      begin
930      gotEvent := WaitNextEvent(everyEvent, eventRec, kMaxLong, nil);
931      if (gotEvent) then
932        DoEvents(eventRec);
933      end;
934
935  end.
936    {of program}
937
938  { ##################################################################################### }
```

Demonstration Program Comments

When this program is run, the user should invoke demonstrations of various basic QuickDraw drawing operations by choosing items from the Demonstration menu.

The constant declaration block

Lines 45-67 establish constants related to menu, window, icon, string list, and picture resources, menu IDs, and menu item numbers. Line 68 defines kMaxLong as the maximum possible long value. This value will be assigned to WaitNextEvent's sleep parameter.

The variable declaration block

gDone will be set to true when the user selects Quit from the File menu or clicks the window's close box, thus causing program termination. gWindowPtr will be assigned the pointer to the main window's 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.

The procedure DoRandomNumber

DoRandomNumber generates and returns a random number between 0 and the value passed to it. Random returns a random number between -32,767 to 32,767, which is then made positive by taking its absolute value. Applying 'mod' to this returns a number between 0 and range, and this is the value returned.

The procedure DoRectOval

DoRectOval draws a framed rectangle, a painted rectangle, a filled round rectangle, and a framed oval. It then inverts the round rectangle and erases the oval.

Lines 120-124 fill the port rectangle with the pattern white, set the pen pattern to black, set the pen size to 10 pixels wide by 20 pixels high, and set the pen mode to patCopy. Line 126 defines the Rect required as a parameter by the drawing routines.

Line 130 draws a framed rectangle.

Lines 135-137 offset the rectangle to the right, set the pen pattern to ltGray and paint a rectangle.

Lines 142-144 offset the rectangle to the left and down, retrieve one of the system patterns, and fill a rounded rectangle with that pattern. The rounded rectangle is drawn with corner curvatures of 120 wide and 60 high.

Lines 149-152 offset the rectangle to the right, set the pen size to 40 pixels wide by 20 pixels high, set the pen pattern to dkGray and frame an oval.

After waiting for the user to click the mouse button, and after some text is erased (Lines 155-159, Lines 164-165 offset the rectangle so that it is back over the rounded rectangle and invert the rounded rectangle with a call to InvertRoundRect.

Lines 170-171 offset the rectangle so that it is back over the oval and erase the oval with a call to EraseOval.

The procedure DoArcWedge

DoArcWedge draws an arc and a two wedges. The drawing of the arc is animated.

Lines 186-190 fill the port rectangle with the pattern white, and set the pen size, pattern, and mode. Line 192 defines the bounding rectangle for the arc and wedges.

Lines 196-197 draw an arc with the routine FrameArc 274 times. The starting angle remains fixed at 135 and the extent of the arc is incremented by one each time around the loop. The effect is to animate the drawing of an arc in the shape of a large C.

Lines 202-207 offset the rectangle to the right, retrieve one of the system patterns, use that pattern in a call to FillArc to draw a 270° wedge from the 10.30 o'clock position, offset the rectangle to the left, and call FillArc to draw a 90° wedge from the 7.30 o'clock position with the same pattern.

The procedure DoPolygon

DoPolygon draws a framed and filled polygon.

Lines 221-225 fill the port rectangle with the pattern white, and set the pen size, pattern, and mode.

Lines 227-234 initiate the recording of the polygon definition (Line 227), define the polygon (Lines 228-233), and stop the recording (Line 234). Note that, in this demonstration, the last vertex is not joined to the first vertex.

Line 238 draws the polygon with the FramePoly routine. (Because the last vertex was not joined to the first during definition, FramePoly does not draw that part of the polygon. Note also that the pen hangs to the right and down of the (infinitely thin) lines which define the polygon.)

Lines 243-245 offset the polygon to the right, retrieve one of the system patterns and draw a filled polygon with that pattern. (Note that FillPoly, in effect, joins the last vertex to the first when it draws the shape.)

Line 247 deallocates the memory used to store the polygon.

The procedure DoRegion

DoRegion draws a framed region and a filled region. DoRegion then demonstrates the effects of the InsetRgn routine to shrink and then expand the region.

Lines 261-263 fill the port rectangle with the pattern white, and set the pen pattern and mode.

Line 265 allocates memory for a new region and a region pointer, initialises the contents of the region and make it an empty rectangle.

Lines 267-274 initiate the recording of a region shape (Line 267), create a region definition comprising two rectangles and an overlapping oval (Lines 268-273) and terminate the recording (Line 274).

Lines 278-280 set the pen pattern and pen size and draw a framed region based on the region definition.

Lines 285-286 offset the region to the right and then draw a filled region with the pattern dkGray.

Lines 291-294 offset the region to the left and down, inset (shrink) the region by 10 pixels horizontally and vertically, and paint the shrunken region. (Note that the inset is applied to each outline in the region).

Lines 299-302 offset the region to the right, inset (expand) the region by 10 pixels horizontally and vertically and paint the expanded region. (Note that the inset is once again applied to each outline in the region. Note also that the demonstration shows that information can be lost when a region is shrunk and then expanded again.).

Line 304 deallocates the memory used to store the region.

The procedure DoTransferMode

DoTransferMode demonstrates the effects of the source modes srcCopy, srcOr, srcXor, and srcBic.

Lines 322-327 fill the port rectangle with the pattern white, and set the pen size, pattern, and mode.

Lines 329-336 retrieve 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.

Lines 344 and 350 use PlotIcon to draw the icons, expanding them into the 71 pixel by 71 pixel rectangle defined at Lines 343 and 349. The expanded icons are then outlined in a one pixel line and identified to the user as the destination image (the square) and the source image (the cross).

Lines 355-363 then draw the cross icon, once again expanded into a 71 pixel by 71 pixel square, eight times in two rows of four images.

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 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 (Line 366). The address of the square icon image data is then assigned to the baseAddr field of a BitMap record (Line 368), the rowBytes field is assigned the value 4 (Line 369), and the bounds field is assigned a rectangle defining the normal icon size (Line 370).

Lines 372-387 copy the bit image into the graphics port eight times, overdrawing the previously drawn cross icons. Line 379 establishes the expanded destination rectangle which governs the size at which the image will be drawn. This is used in the call to CopyBits at Line 380. Note that, in this call, the value of the parameter which specifies the source mode is incremented each time through the loop so that the square image overdraws the cross image once in each of the eight available source modes. Lines 383-385 retrieve the appropriate string containing the relevant source mode from the 'STR#' resource and print this string under each image.

Line 389 unlocks the icon image data.

The procedure DoCopyBits

DoCopyBits copies a bit image from one graphics port to another, resizing and reshaping the image in the process.

Line 405 prepares the way by filling the port rectangle with the pattern white.

Line 407 opens a small window over the right side of the main window. Lines 411-412 save the current graphics port and make the new window's port the current graphics port.

Line 414 loads a picture from a 'PICT' resource. Since the purgeable bit of the resource's attributes is set, the resource is immediately made non-purgeable (Line 422), used immediately (Line 424), and immediately made purgeable again (Line 425). The picture is drawn in the current graphics port (the small window).

When the user clicks the mouse button (Line 428), a large rectangle is defined to represent the size and shape in which the copied image is to be drawn (Line 430). This is used in the call to CopyBits (Line 432), which copies the image from the small window's graphics port to the main window's graphics port.

When the user again clicks the mouse button (Line 437), Lines 439-440 dispose of the small window and reset the current graphics port to that of the main window.

The procedure DoText

DoText draws text in various fonts, sizes and styles. The last line of text is drawn using the grayishTextOr transfer mode.

Line 454 prepares the way by filling the port rectangle with the pattern white.

Line 457 gets a position half way across the window. This will be used to centre the lines of text in the window as they are drawn.

Line 459-512 is a loop within which the text font, size and style are changed, a string is retrieved from a 'STR#' resource (Line 508), the width of the string in pixels is determined (Line 509), and the string is drawn centrally (from left to right) in the window (Lines 510-511).

Note that, the last time around the loop, the transfer mode is set to grayishTextOr (Line 501).

Lines 515-519 reset the font, size, transfer mode, and style back to the settings which existed before doText was called.

The procedure DoBasicColours

DoBasicColours draws eight rectangles in each of the eight colours pre-defined by basic QuickDraw. (On black and white screens, all colours except white will be drawn in black. On greyscale screens the colours will appear as shades of gray.)

The procedure DoLines

DoLines demonstrates line drawing with various pen patterns. doLines also demonstrates clipping. (Note that, as is the case with all drawing demonstration functions in this program, some of the code is related to program execution (for example, delays, setting the window title, waiting for mouse clicks before proceeding, etc) and not to drawing operations per se. Those parts of the code will generally be disregarded in the following comments.)

At Line 569, FillRect is called to fill the entire port rectangle with the pattern white. At Line 571, the pen mode is set to patCopy for the lines demonstration.

Lines 573-581 set the window's clipping region to a rectangle 10 pixels inside the port rectangle. Lines 573-576 assign appropriate values to four variables which will be used to define the Rect representing the new clipping region, Lines 578-579 save the old clipping region, and Line 580 defines the Rect which is used in the call to ClipRect at Line 581 to establish the new clipping region.

Lines 583-596 draw 38 lines using the 38 patterns in the 'PAT#' resource of the System file. Each time around the loop, the variables b and c are assigned separate random numbers between 0 and the width of the port rectangle (Lines 585-586), the next system pattern is retrieved (Line 588), the pen pattern is set to this pattern (Line 589), the width of the pen is increased (Line 590), and a line is drawn from somewhere at the top of the port rectangle to somewhere at the bottom of the port rectangle (Lines 592-593). The line drawing is, of course, clipped to the clipping region established at Line 581, which is 10 pixels inside the port rectangle.

Preparatory to the second part of the line drawing demonstration, the old clipping region is restored and the memory in which it was saved is deallocated (Lines 598-599).

Lines 610-624 illustrate a well-known but nonetheless exotic capability of the humble line when it operates in the pattern mode patXor. Lines 605-608 set all the port's pixels to white, the pen size to 1 pixel by 1 pixel, the pen pattern to black and the pattern mode to patXor. Proceeding clockwise, Lines 611-624 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 if the source pixel is black. Thus, 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 moiré (watered silk) pattern.

The procedure DoDrawWithMouse

doDrawWithMouse is called when the user has chosen the Draw With Mouse item from the Demonstration menu and subsequently clicks in the window. While the mouse button remains down, a "rubber-band" rectangle is continually erased and redrawn as the mouse is moved. When the mouse button is released, a rectangle, a rounded rectangle, and a painted rectangle are drawn at a location and size determined by the "rubber-band" rectangle.

Lines 638-640 set the pen size to 1 pixel wide and high, the pen pattern to gray, and the pen mode to patXOr.

Line 642 gets the mouse location where the mouse-down occurred. Those coordinates are then used to initialise the fields of a Rect structure, the left and top fields of which will remain unchanged from this point.

Line 649 assigns the same mouse location to another Point variable, which will be used for comparison purposes within the while loop entered at Line 651.

The while loop continues to execute while the mouse button remains down. Within the loop, the current mouse location is retrieved (Line 653) and compared with the previous mouse location (Line 655). If the mouse has moved, FrameRect is called, the current mouse coordinates are assigned to the bottom and right fields of the Rect, and FrameRect is again called (657-662). Because the drawing mode is patXor, the first call to FrameRect erases the old rectangle. Note that, because Lines 657-662 only execute if the mouse has moved, the flicker which would otherwise occur when the mouse is stationary is avoided. At Lines 665-666, and preparatory to the next Line 655 comparison, the current mouse position is assigned to the variable which holds the previous mouse position.

When the mouse button is released, Line 669 erases the final "rubber-band" rectangle. Lines 671-693 then draw a rectangle, a rounded rectangle, and a painted rectangle based on the location and size of the "rubber-band" rectangle when the mouse button was released.

The procedures DoDemonstrationMenu, DoMenuChoice

DoMenuChoice and DoDemonstrationMenu handle menu choices from the Apple, File and Demonstration menus. Note that, at Lines 753-759, the global variable gDrawWithMouseActivated is set to true when the Draw With Mouse menu item is chosen.

The procedures DoMouseDown, DoEvents

DoEvents and DoMouseDown perform minimal event handling consistent with the satisfactory operation of the drawing demonstration aspects of the program. Note that, at Lines 835-836, the application-defined function doDrawWithMouse is called if the global variable gDrawWithMouseActivated contains true.

The main program block

The main function initialises the system software managers (Line 896), seeds the random number generator (Line 900), sets up the menus (Lines 904-913), opens the main window and sets its graphics port as the current port for drawing operations (Lines 917-921), sets the text size (Line 922), and enters the main event loop (Lines 926-933).

Random numbers are used in the application-defined function doLines. randSeed (Line 900) 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. Line 900 shows one way to seed the generator. The parameter to the GetDateTime call receives the number of seconds since midnight, January 1, 1904, a value which is bound to be different each time the program is run.

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

Creating 'PICT' Resources Using ResEdit

Open the chap10cw_demo demonstration program folder. Double-click on the BasicQuickDraw.µ.rsrc icon to start ResEdit and open BasicQuickDraw.µ.rsrc. The **BasicQuickDraw.µ.rsrc** window opens.

Double-click the PICT icon. The **PICTs from BasicQuickDraw.µ.rsrc** window opens. A thumbnail image of one 'PICT' resource (ID 128) appears in the window. Double-click the thumbnail image. The **PICT ID = 128 from BasicQuickDraw.µ.rsrc** window opens, displaying the picture.

To procedure for creating the 'PICT' resource is as follows:

- Within a paint or draw application, copy an image to the Clipboard.
- Open BasicQuickDraw.µ.rsrc in ResEdit. Choose **Resource/Create New Resource**. A small dialog opens. Click the **PICT** item in the scolling list, and then click the dialog's **OK** button. The **PICTs from BasicQuickDraw.µ.rsrc** window opens, followed by the **PICT ID = 128 from BasicQuickDraw.µ.rsrc** window. (ResEdit automatically assigns 128 as the resource ID of the first 'PICT' resource you create.)
- Choose **Edit/Paste**. The picture appears in the **PICT ID = 128 from BasicQuickDraw.µ.rsrc** window.

Further 'PICT' resources may be created by copying other images to the Clipboard and, within ResEdit, choosing **Edit/Paste** while the **PICTs from BasicQuickDraw.µ.rsrc** window is open and in front. (ResEdit automatically increments the resource ID at each successive paste.)

Another way to copy an image to the Clipboard for the purpose of creating a `'PICT'` resource is to capture the image directly from the screen using a screen capture utility such as Flash-It™.